See all versions of this document

Vitis Reference Guide

UG1702 (v2025.2) November 20, 2025

AMD

Table of Contents

Chapter 1: Navigating Content by Design Process 5

Chapter 2: Vitis Commands and Utilities 6

Launching the Vitis Unified IDE 7
v++ Command 9
emconfigutil Utility 159
kernelinfo Utility 160
launch_emulator Utility 162
manage_ipcache Utility 176
package_xo Command 177
platforminfo Utility 182
vitis-run Command 190
xrt-smi Utility 193
xbmgmt Utility 194
xclbinutil Utility 195
xrt.ini File 203

Chapter 3: Using the Vitis Unified IDE 213

Migration to Vitis Unified IDE 213
Launching Vitis Unified IDE 215
Vitis Unified IDE View and Features 216
Creating an AI Engine Component 236
Creating an HLS Component 237
Creating an Application Component 237
Creating a Platform Component 238
Embedded Software Development Flow 239
Creating a System Project for Heterogeneous Computing 241
Creating a Bare-metal System 261
Migrating Command-line Projects to the Vitis Unified IDE 265
Debugging the System Project and AI Engine Components 268
Programming Device and Flash Memory 290
Vitis Interactive Python Shell 291

Python API: Managing Vitis IDE Components 296
Vitis IDE Examples 306

Chapter 4: Managing the Vitis HLS Components in the Vitis Unified IDE 307
Using the Flow Navigator 307
Creating Vitis HLS Components 309
Building and Running an HLS Component 315
Creating HLS Components from the Command Line 363
Opening HLS Component Flow Step Reports 366

Chapter 5: Managing the AI Engine Component in the Vitis Unified IDE 368
Using the Vitis Unified IDE to Build and Simulate AI Engine Components 368
Generating AI Engine Prototype Code 391
Single Kernel Development 393
Exporting Summary Tables from the Analysis View 394

Chapter 6: Managing the Integration Project Component in the Vitis Unified IDE 395
Creating an Integration Project Component 395
Possible Input Combinations for the Integration Project 400

Chapter 7: Enabling Third-Party Simulators in the Vitis IDE 401

Chapter 8: Working with the Analysis View (Vitis Analyzer) 404
Configuring the Analysis View 405
Open Summary Reports 407
Open Trace Summary Using Time Window 409
Analysis View Window Manager 411
Viewing AI Engine Compilation Summary Reports 414
Importing JSON Output Files 428

Chapter 9: Additional Information 430
Output Structure of the Vitis Tools 430
Python XSDB Commands 454

Appendix A: Additional Resources and Legal Notices 485
Finding Additional Documentation 485

Support Resources....486
References....486
Revision History....486
Please Read: Important Legal Notices....487

Navigating Content by Design Process

AMD Adaptive Computing documentation is organized around a set of standard design processes to help you find relevant content for your current development task. You can access the AMD Versal™ adaptive SoC design processes on the Design Hubs page. You can also use the Design Flow Assistant to better understand the design flows and find content that is specific to your intended design needs. This document covers the following design processes:

- **System and Solution Planning:** Identifying the components, performance, I/O, and data transfer requirements at a system level. Includes application mapping for the solution to PS, PL, and AI Engine.
- **Embedded Software Development:** Creating the software platform from the hardware platform and developing the application code using the embedded CPU. Also covers XRT and Graph APIs.
- **AI Engine Development:** Creating the AI Engine graph and kernels, library use, simulation debugging and profiling, and algorithm development. Also includes the integration of the PL and AI Engine kernels.
- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the AMD Vivado™ timing, resource use, and power closure. Also involves developing the hardware platform for system integration.
- **Software Development for Acceleration:** Create an algorithm accelerator kernel with HLS and/or AI Engine. Includes platform design, organization, and management.

Vitis Commands and Utilities

The AMD Vitis Design Suite includes a command-line interface and an integrated development environment (Vitis Unified IDE). The tool uses the `v++` command and the `vitis-run` command to compile and run design components. The reference materials contained in this section include the following commands.

- **v++ Command:** The v++ Command section provides a description of the compiler options (`--compile`) and includes the following commands.
  - AI Engine and HLS compiler modes.
  - Linking options (`--link`) for creating the device binary.
  - Packaging options (`--package`) for building boot files and generating an SD card for the system.

  The v++ Command section also includes a discussion of `--config` command and configuration files.
- **HLS Pragmas:** The HLS Pragmas section provides a description of pragmas for use in HLS components.
- **`xrt.ini` file:** The Xilinx Runtime (XRT) file initializes XRT to produce reports, debug, and profiling data between the host and kernels. This file is useful for emulation and hardware builds. You need to create the file manually when running the build process from the command line.

The reference materials also include various AMD utilities for the Vitis tools and Xilinx Runtime (XRT). The utlities provide detailed information about platform resources. The resources include SLR and memory resource availability to help you construct the `v++` command line and manage the build and run process.

- emconfigutil Utility
- kernelinfo Utility
- launch_emulator Utility
- manage_ipcache Utility
- package_xo Command
- platforminfo Utility

- vitis-run Command
- xrt-smi Utility
- xbmgmt Utility
- xclbinutil Utility

TIP: *The Xilinx Runtime (XRT) Architecture reference material is available on the Xilinx Runtime GitHub repository.*

Launching the Vitis Unified IDE

To launch the Vitis Unified IDE, follow these steps:

1. Use the following command to load the Vitis software platform environment.

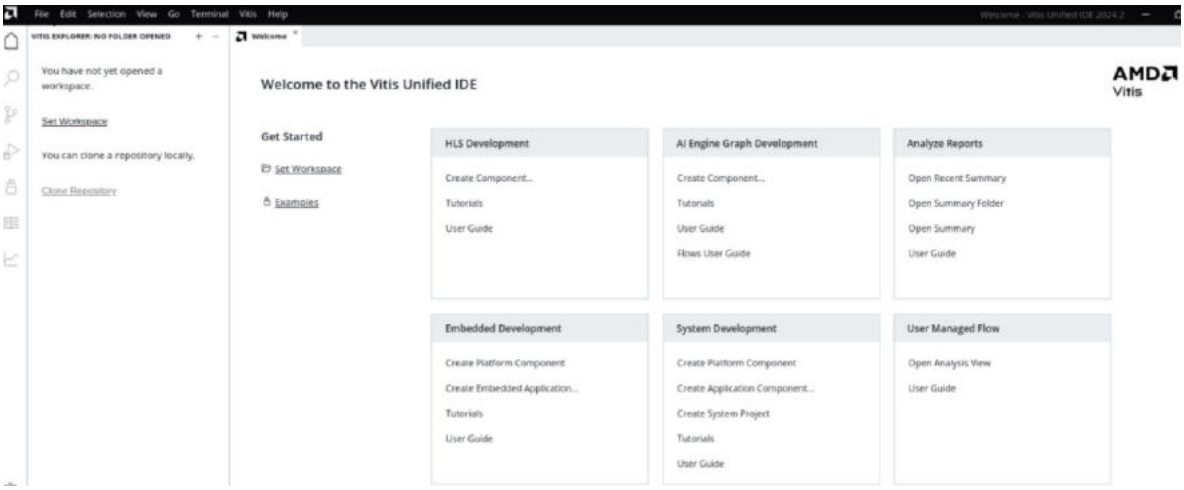
   ```
   source <Vitis_Installation_Directory>/settings64.sh
   ```

2. Use the following command to launch the Vitis Unified IDE.

   ```
   vitis -w <workspace>
   ```

   Here, `<workspace>` indicates a folder to hold all the contents of your design project. The workspace groups together the source and data files that make up a design, or multiple designs, and stores the configuration of the tool for that workspace.

Figure 1: **Welcome Screen**



For other supported launch modes, see Launch Options.

Note: Before launching the Vitis Unified IDE, you can set up other environmental settings to ensure that the tool can pick up these settings. For example, you can set up Xilinx Runtime (XRT) for building and running data center acceleration applications:

```
source <XRT_Install_Path>/setup.sh
```

You can set up the platform repository path with the following example:

```
export PLATFORM_REPO_PATHS=<platform_path>
```

Launch Options

The Vitis Unified IDE supports the following modes:

- **Workspace Mode:** Opens the Vitis Unified IDE with the specified workspace.

  ```
  vitis -w <workspace>
  ```

- **Analysis Mode:** Launches the tool directly into the Vitis Analyzer (see Chapter 8: Working with the Analysis View (Vitis Analyzer)). You can review the summary reports the tool generates during build, run, and debug.

  ```
  vitis -a
  ```

- **Interactive Mode:** Allows you to enter commands through the command-line interface, outside of the GUI.

  ```
  vitis -i
  ```

  Note: Type `help()` from the interactive command prompt to explore the available command modules.

- **Batch Mode:** Executes the specified script and exits.

  ```
  vitis -s <script>.py
  ```

- **Jupyter Notebook Mode:** Launches Jupyter Notebook server with the Vitis environment and the front end UI in your default web browser.

  ```
  vitis -j
  ```

  You can use `vitis_server` command line interface in this environment as described in Vitis Interactive Python Shell.

You can use `-h` to print the options of `vitis`.

```
vitis -h

Syntax: vitis [-a | -w | -i | -s | -h | -v]

Options:
  -a/--analyze [<summary file | folder | waveform file: *.[wdb|wcfg]>]
         Open the summary file in the Analysis view.
```

```
        Opening a folder opens the summary files found in the folder.
        Open the waveform file in a waveform view tab.
        If no file or folder is specified, opens the Analysis view.
-w/--workspace <workspace_location>
        Launches Vitis IDE with the given workspace location.
-i/--interactive
        Launches Vitis python interactive shell.
-s/--source <python_script>
        Runs the given python script.
-j/--jupyter
        Launches Vitis Jupyter Web UI.
-h/--help
        Display help message.
-v/--version
        Display Vitis version.
```

v++ Command

This section describes the AMD Vitis™ compiler command, `v++`, and the various options it supports for building the device binaries. `v++` is a standalone command line utility with three command modes.

- **`--compile` (`-c`):** Compiles C++ to create AI Engine binaries and PL modules. For embedded applications, to develop AI Engine kernels and compile the AI Engine graph applications into a `libadf.a` file, refer to Compiling an AI Engine Graph Application in the *AI Engine Tools and Flows User Guide* (UG1076). For developing and compiling HLS PL kernel into a XO object file, refer to HLS Kernel Development in the *Embedded Design Development Using Vitis* (UG1701).To target a data center card, refer to Compiling C/C++ PL Kernels in the *Data Center Acceleration using Vitis* (UG1700).
- **`--link` (`-l`):** For linking AI Engine kernels, HLS PL kernels, and Vitis subsystems into an extensible XSA platform to create a fixed XSA hardware design or emulation design. Also used to create Vitis subsystems and Vitis metadata archive that can be imported into a Vivado project for design iteration and completion.
- **`--package` (`-p`:** For packaging AI Engine `libadf.a` binaries into `.pdi` and `.xclbin` images as described in Packaging for Vitis Flow in the *Embedded Design Development Using Vitis* (UG1701). Also generates an SD card image for Zynq Ultrascale target devices.

Beyond these three command modes, many other options are available to customize the build process. The following sections describe these options. Some of the options are supported for all three command modes, and some options are specific to compilation, linking, or packaging.

v++ General Options

The `v++` command supports options and directives for compilation, linking, and packaging processes as described below.

TIP: *Specify all* `v++` *command-line options in a configuration file for use with the* `--config` *option (see Vitis Compiler Configuration File). For example, the* `--platform` *option can be specified in a configuration file using the following syntax:*

```
platform=xilinx_u50_gen3x16_xdma_5_202210_1
```

--advanced

- **Applies to:** Compile, Link, Package

Specifies parameters and properties for use by the `v++` command. See --advanced Options for more information.

--board_connection

- **Applies to:** Link

```
--board_connection
```

Specifies a dual in-line memory module (DIMM) board file for each DIMM connector slot. The board is specified using the Vendor:Board:Name:Version (vbnv) attribute of the DIMM card as it appears in the board repository.

For example:

```
<DIMM_connector>:<vbnv_of_DIMM_board>
```

-c | --compile

- **Applies to:** Compile

```
--compile
```

Required for compilation, but mutually exclusive with `--link` and `--package`. Run `v++ -c` to generate XO files from kernel source files.

--clock

- **Applies to:** Link

Provide a method for assigning clocks to kernels during the linking process. See --clock Options for more information.

--config

- **Applies to:** Compile, Link, Package

```
--config <config_file> ...
```

Specifies a configuration file containing `v++` command options. The configuration file can be used to capture compilation, linking, or packaging strategies, that can be easily reused by referring to the config file on the `v++` command line. In addition, the config file allows the `v++` command line to be shortened to include only the options that are not specified in the config file. Refer to the Vitis Compiler Configuration File for more information.

TIP: *Multiple configuration files can be specified on the `v++` command line. Vitis tools require a separate `--config` switch for each file used. For example:*

```
v++ -l --config system.cfg --config vivado.cfg ...
```

--connectivity

- **Applies to:** Link

Specifies directives to the v++ linker to integrate compiled components. See --connectivity Options for more information.

--custom_script

- **Applies to:** Compile, Link

```
--custom_script <kernel_name>:<file_name>
```

This option lets you specify custom Tcl scripts for compilation or linking during the build process. Use with the `--export_script` option to create, edit, and run the scripts to customize the build process.

When used with the `v++ --compile` command, this option lets you specify a custom HLS script to be used when compiling the specified kernel. The script lets you modify or customize the Vitis HLS tool. Use the `--export_script` option to extract a Tcl script. This Tcl script is useful for Vitis HLS to compile the kernel, modify the script and resubmit using the `--custom_script` option to better manage the kernel build process.

The argument lets you specify the kernel name and path to the Tcl script to apply to that kernel. For example:

```
v++ -c -k kernel1 --export_script ...
*** Modify the exported script to customize in some way, then resubmit. ****
v++ -c --custom_script kernel1:./kernel1.tcl ...
```

You can use this option with the `v++ --link` command for the hardware build target (`-t hw`). The option lets you specify the absolute path to an `run_script_map.dat` file. The `run_script_map.dat` file contains a list of steps in the build process and Tcl scripts that the Vitis or Vivado tool runs during the build process. You can edit `run_script_map.dat` to specify custom Tcl scripts to run at various steps in the build process. Use the following steps to customize the Tcl scripts:

1. Run the build process to specify the `--export_script` option.

   ```
   v++ -t hw -l -k kernel1 --export_script ...
   ```

2. Copy the Tcl scripts in the `run_script_map.dat` file for any of the steps you want to customize. For example, copy the Tcl file specified for the synthesis run, or the implementation run. You must copy the file to a separate location, outside of the project build structure.
3. Edit the Tcl script to add or modify any of the existing commands to create a new custom Tcl script.
4. Edit the `run_script_map.dat` file to point a specific implementation step to the new custom script.
5. Relaunch the build process using the `--custom_script` option. Specify the absolute path to the `run_script_map.dat` file as shown below.

   ```
   v++ -t hw -l -k kernel1 --custom_script /path/to/run_script_map.dat
   ```

IMPORTANT! *When editing a custom synthesis run script, you must either comment out the lines related to the `dont_touch.xdc` file, or edit the lines to point to a new user-specified `dont_touch.xdc` file. See the following for the specific lines to comment or edit.*

```
read_xdc dont_touch.xdc
set_property used_in_implementation false [get_files dont_touch.xdc]
```

The synthesis run returns an error related to a missing `dont_touch.xdc` file if this is not done.

--debug

- **Applies to:** Link

Specifies the addition of debug IP core insertion into the hardware design. See --debug Options for more information.

-D | --define

- **Applies to:** Compile, Link

```
--define <arg>
```

Valid macro name and definition pair: `<name>=<definition>`.

Predefines name as a macro with definition. This option passes to the `v++` preprocessor.

***Note*:** -D only applies to classic CLI (for example, "v++ -c -k") for compiling HLS kernels.

--export_archive

- **Applies to:** Link

```
--export_archive
```

This option instructs the v++ linker to generate a Vitis metadata archive (`.vma`) that can be imported into a Vivado hardware project in lieu of a fixed hardware XSA or `.xclbin` file. The flow for this command and the use of the `.vma` file is described in Packaging for Vitis Export to Vivado Flow in the *Embedded Design Development Using Vitis* (UG1701).

--export_script

- **Applies to:** Compile, Link, Package

```
--export_script
```

This option runs the build process up to the point of exporting one or more script files, and then stops execution. You must use the `--custom_script` option to complete the build process. The `--custom_script` option lets you edit the exported script, or list of scripts, and then rerun the build using your custom scripts.

When used with the `v++ --compile` command, this option exports a Tcl script for the specified kernel, `<kernel_name>.tcl`. The script can execute Vitis HLS, but stops the build process before actually launching the HLS tool. You can interrupt the build process to edit the generated Tcl script, and then restart the build process using the `--custom_script` option. The following is an example.

```
v++ -c -k kernel1 -export_script ...
```

When used with the `v++ --link` command for the hardware build target (`-t hw`), this option exports a `run_script_map.dat` file in the current directory. This file contains a list of steps in the build process. The file also contains Tcl scripts that Vitis and Vivado tool run during those steps. You can edit the specified Tcl scripts, customizing the build process in those scripts, and relaunch the build using the `--custom_script` option. Export the `run_script_map.dat` file using the following command:

```
v++ -l -t hw --export_script ...
```

--freqhz

- **Applies to:** Compile and Link

```
--freqhz <value>::<cu>[.<clk_pin>][,<cu_n>[.<clk_pin_n>]]
```

Specifies a frequency for a PL component during compilation, or for the system during linking. The `--freqhz` option overrides any default clock frequency and applies the stated frequency during compilation and linking.

Specify a clock frequency in Hertz and a list of associated compute unit names and optionally their clock pins:

- **<value>:** Specify this in Hertz. For example, 150 MHz is specified as 150000000. Units of Hz and MHz are also accepted, for example 150000000Hz and 150MHz.
- **[cu[.clk_pin]]:** Optionally specifes a PL kernel or instance of a PL kernel (Compute Unit) target for the specified clock frequency. The PL kernel or instance of a PL kernel lets you specify a different frequency for different instances of kernels. You can also specify the clock pin of the CU by adding the `.clk_pin` syntax. You can add multiple CU and clock pins in a single `--freqhz` option with a single frequency value.

For example:

```
v++ -l -t hw --platform <pfm_name> --freqhz=200000000:mm2s \
--freqhz=300000000:s2mm -config system.cfg
```

IMPORTANT! *The `--freqhz` option can consolidate the various methods or commands associated with specifying clocks in the AI Engine, PL kernels, or system design. The `--freqhz` option takes precedence over other clock commands, and `v++` can return an error if multiple commands are used.*

--from_step

- **Applies to:** Compile, Link, Package

```
--from_step <arg>
```

Specifies a step name for the Vitis compiler to start the build process. If intermediate results are available, the link process fast forwards and begins execution at the named step when possible. You can run the build through a `--to_step`, and then resume the build process at the `--from_step`. You can use the `--list_steps` option to determine the list of valid steps.

IMPORTANT! *`--to_step`/`--from_step` are sequential build options that require you to use `--from_step` to resume the build in the same project directory that you used when starting the build with `--to_step`.*

For example:

```
v++ --link --from_step vpl.update_bd
```

-g

- **Applies to:** Compile, Link

```
-g
```

Generates code for debugging the kernel during hardware emulation. Using this option adds features to facilitate debugging the kernel as it is compiled.

For example:

```
v++ -g ...
```

-h | --help

- **Applies to:** Compile, Link, Package

```
-h
```

Prints the help contents for the `v++` command. For example:

```
v++ -h
```

--hls

- **Applies to:** Compile

Specifies commands for the Vitis HLS synthesis process during kernel compilation.

-I | --include

- **Applies to:** Compile, Link

```
--include <arg>
```

Adds the specified directory to the list of directories for searching header files. This option passes to the Vitis compiler pre-processor.

- **Classic Compile Mode (`v++ -c -k`):** Usage: `v++ -c -k --include <path>`
- **HLS Mode (`v++ -c --mode hls`):** Usage: `--include <path>` is not recommended. Specify include paths in an HLS configuration file instead of with compiler flags.
- **AIE Mode (`v++ -c --mode aie`):** Usage: `v++ -c -k --include <path>`

--input_files <input_file>

- **Applies to:** Compile, Link, Package

```
--input_files <input_file1> <input_file2> ...
```

Specifies a C/C++ kernel source file for `v++ -c -k classic` compilation or `v++ -c --mode aie`, or Xilinx object (XO) files for `v++` linking.

For example:

```
v++ -l --input_files kernel1.xo kernelRTL.xo ...
```

TIP: *You can also specify input files positionally without the* `--input_files` *option. For example:*

```
v++ -l kernel1.xo kernelRTL.xo ...
```

The `--input_files` option can be used to specify positional arguments on the command line, but typically, file names are provided directly without the option.

Example for Linking:

```
v++ -l -t hw --input_files mykernel.xo
```

is more commonly written as:

```
v++ -l -t hw mykernel.xo
```

- **Classic Compilation::** `v++ -c -k mykernel --input_files mykernel.cpp` is commonly used as: `v++ -c -k mykernel mykernel.cpp`
- **AIE Mode::** `v++ -c --mode aie --input_files mygraph.cpp` is typically: `v++ -c --mode aie mygraph.cpp`
- **HLS Mode::** Usage of `--input_files` with HLS mode: `v++ -c --mode hls --input_files myhlskernel.cpp` is not supported.

Linking:

The `--input_files` option can specify files like `libadf.a`, `.xo`, and `.vss`. Generally, these are specified directly as positional arguments.

Packaging:

When packaging, `--input_files` can include `libadf.a`, `.vss`, `fixed.xsa`, and `.xclbin` files, especially for Zynq UltraScale devices. The output from linking is typically `fixed.xsa` or `.xclbin` files.

--interactive

- **Applies to:** Link

```
--interactive [ impl ]
```

`v++` configures the required environment and launches the Vivado tool with the implementation project.

Because you are interactively launching the Vivado tool, the linking process is stopped after the `vpl` step, which is the equivalent of using the `--to_step vpl` option in your `v++` command.

When you are done interacting with the Vivado tool, save the design checkpoint (DCP). You can then resume the Vitis compiler linking process using the `v++ --from_step rtdgen`, or use the `--reuse_impl` or `--reuse_bit` options to read in the implemented DCP file or bitstream.

For example:

```
v++ --interactive impl
## Interactively use the Vivado tool
v++ --from_step rtdgen
```

-k | --kernel

- **Applies to:** Compile

```
--kernel <arg>
```

Compiles only the specified kernel from the input file. Only one `-k` option is allowed per `v++` command. Valid values include the name of the kernel to be compiled from the input `.c/.cpp` kernel source code.

Vitis tools requires this option for C/C++ kernels. You must identify the kernel by `-k` or `--kernel`.

- **Classic compilation:** Use `-k` with `v++ -c` to identify the kernel to compile from a source file.
- **AIE mode:** Do not use `-k` with `v++ -c --mode aie`.
- **HLS mode:** Do not use `-k` with `v++ -c --mode hls`.
- **Invalid combination:** `v++ -c -k --mode hls` is not a valid command.

--kernel_frequency

IMPORTANT! *This command specifies kernel frequencies only for legacy Alveo platforms with scalable clocks. Platforms using fixed clocks, including both Alveo and embedded platforms, use the --clock Options for clock management. Refer to Managing Clock Frequencies in the Data Center Acceleration using Vitis (UG1700) for more information.*

- **Applies to:** Link

```
--kernel_frequency <freq> | <clockID>:<freq>[<clockID>:<freq>]
```

Specifies a user-defined clock frequency (in MHz) for the kernel, overriding the default clock frequency defined on the hardware platform. The `<freq>` specifies a single frequency for kernels with only a single clock, or can be used to specify the <clockID> and the <freq> for kernels that support two clocks.

The syntax for overriding the clock on a platform with only one kernel clock, is to simply specify the frequency in MHz:

```
v++ --kernel_frequency 300
```

To override a specific clock on a platform with two clocks, specify the clock ID and frequency:

```
v++ --kernel_frequency 0:300
```

To override both clocks on a multi-clock platform, specify each clock ID and the corresponding frequency. For example:

```
v++ --kernel_frequency 0:300|1:500
```

TIP: *If Vivado place and route tools are unable to meet specification frequencies during design implementation, the tools can scale the clock frequency to an achievable frequency.*

-l | --link

- **Applies to:** Link

```
--link
```

This is a required option for the linking process, which follows compilation, but is mutually exclusive with `--compile` or `--package`. Run `v++` in link mode to link XO input files and generate a fixed hardware platform (XSA). For MPSoC and UltraScale targets, generates an `.xclbin` file instead of fixed `.xsa`.

--linkhook

- **Applies to:** Link

Lets you customize the build process for the device binary by specifying Tcl scripts to be run at specific steps in the implementation flow. See --linkhook Options for more information.

--list_steps

- **Applies to:** Compile, Link, Package

```
--list_steps
```

Lists valid run steps for a given target. This option returns steps that can be used in the `--from_step` or `--to_step` options. You must specify this command with the following options:

- `-t | --target [hw_emu | hw ]`:
- `[ --compile | --link ]`: Specifies the list of steps from either the compile or link process for the specified build target.

For example:

```
v++ -t hw_emu --link --list_steps
```

--log_dir

- **Applies to:** Compile, Link, Package

```
--log_dir <dir_name>
```

Specifies a directory to store log files into. If you do not specify `--log_dir`, the tool saves the log files to `./_x/logs`. Refer to Output Directories of the v++ Command for more information.

For example:

```
v++ --log_dir /tmp/myProj_logs ...
```

--message_rules

- **Applies to:** Compile, Link, Package

```
--message-rules <file_name>
```

Specifies a message rule file with rules for controlling messages. Refer to Using the Message Rule File for more information.

For example:

```
v++ --message_rules ./minimum_out.mrf ...
```

--mode

- **Applies to:** Compile, Link

Specifies a compilation mode for the `v++` command, one of `aie`, `hls` or `vss` to target AI Engine, PL, or to create a Vitis subsystem, respectively.

- **`v++ -c --mode aie`:** Creates a `libadf.a` component. See *AI Engine Tools and Flows User Guide* (UG1076) for more details. The compilation mode can be used with configuration commands described under v++ Mode AI Engine.
- **`v++ -c --mode hls`:** Creates an .xo kernel or IP an HLS component as described in *Vitis High-Level Synthesis User Guide* (UG1399), and can be used with configuration file commands described in v++ Mode HLS.
- **`v++ -l --mode vss`:**

  The `v++ --mode vss` selects a link flow that creates a Vitis Subsystem (VSS) artifact. The accepted value is `vss`. If mode is not specified, the standard link flow is used.

  A VSS is a platform-independent, reusable design component consisting of compiled `.xo` and `libadf.a` sub-components.

The flow generates a subsystem package (with metadata and binaries) that can be instantiated and linked within a system context on the target platform later.

See Linking a VSS Component in the *Embedded Design Development Using Vitis* (UG1701) for details.

--no_ip_cache

- **Applies to:** Link

```
--no_ip_cache
```

Disables the IP cache for out-of-context (OOC) synthesis for Vivado Synthesis. Disabling the IP cache repository requires the tool to regenerate the IP synthesis results for every build, and can increase the build time. However, it also results in a clean build, eliminating earlier results for IP in the design.

For example:

```
v++ --no_ip_cache ...
```

-O | --optimize

- **Applies to:** Link

```
--optimize <arg>
```

This option specifies the optimization level of the Vivado implementation results. Valid optimization values include the following:

- `0`: Default optimization. Reduces compilation time.
- `1`: Optimizes to reduce power consumption by running Vivado implementation strategy `Power_DefaultOpt`. This takes more time to build the design.
- `2`: Optimizes to increase kernel speed. This option increases build time, but also improves the performance of the generated kernel by adding the `PHYS_OPT_DESIGN` step to implementation.
- `3`: This optimization provides the highest level performance in the generated code, but compilation time can increase considerably. This option specifies retiming during synthesis, and enables both `PHYS_OPT_DESIGN` and `POST_ROUTE_PHYS_OPT_DESIGN` during implementation.
- `s`: Optimizes the design for size. This reduces the logic resources of the device used by the kernel by running the `Area_Explore` implementation strategy.
- `quick`: Reduces Vivado implementation time, but can reduce kernel performance, and increases the resources used by the kernel. This enables the `Flow_RuntimeOptimized` strategy for both synthesis and implementation.

For example:

```
v++ --link --optimize 2
```

-o | --output

- **Applies to:** Compile, Link, Package

```
-o <output_name>
```

Specifies the name of the output file generated by the `v++` command. The compilation (`-c`) process output name must end with the XO file suffix, for Xilinx object file. This applies for v++ -c -k classic compilation. The linking (`-l`) process output file must end with an `.xsa` (or an `.xclbin` for Zynq platforms). The `--export_archive` command requires the `.vma` suffix. The `--package` command takes the `.xsa` as input and outputs the `.xclbin`.

For example:

```
v++ -o krnl_vadd.xo
```

If `--o` or `--output` are not specified, the output file names default to the following:

- **Compilation:** `a.o`
- **Linking:** An `.xsa` (or an `.xclbin` for Zynq platforms)
- **Packaging:** `a.xclbin`

-p | --package

- **Applies to:** Package

Specifies options for the Vitis compiler to package your design for either running emulation or running on hardware. See v++ Command for more information.

--part

- **Applies to:** Compile, Link, Package

```
--part <part_name>
```

Specifies an AMD part for use in compiling, linking, and packaging the components of your design. The `--part` command can be used to specify a device rather than a full platform or XSA. Specifying `v++ --link --part <Versal Gen 1 part only>` instructs the tools to generate a simple hardware emulation platform for the specified AMD Versal™ device.

For example:

```
v++ --link --part xcvc1902-vsva2197-2MP-e-S --target hw_emu ...
```

-f | --platform

- **Applies to:** Compile, Link, Package

```
--platform <platform_name>
```

Specifies the name of a supported acceleration platform as determined by the `$PLATFORM_REPO_PATHS` environment variable, or the full path to the extensible hardware `.xsa` or base platform `.xpfm` file. For a list of supported platforms for the release, see the Vitis Software Platform Installation in the *Vitis Software Platform Release Notes* (UG1742).

This is a required option for both compilation and linking. The `--platform` option accepts either a platform name, or the path to a platform file `xpfm`, using the full or relative path. Packaging can accept a `fixed.xsa` as a platform.

IMPORTANT! *The specified platform and build targets for compiling, linking, and packaging must match. The `--platform` and `-t` options specified when the XO file is generated by compilation, must be the `--platform` and `-t` used during linking, and packaging. For more information, see platforminfo Utility.*

For example:

```
v++ --platform vek385_base
```

--profile

- **Applies to:** Compile, Link

Specifies options to configure the Xilinx Runtime environment to capture application performance information. See --profile Options for more information.

--remote_ip_cache

- **Applies to:** Link

```
--remote_ip_cache <dir_name>
```

Specifies the location of the remote IP cache directory for Vivado Synthesis to use during out-of-context (OOC) synthesis of IP. OOC synthesis lets the Vivado synthesis tool reuse synthesis results for IP that have not been changed in iterations of a design. This can reduce the time required to build your `.xclbin` files, due to reusing synthesis results.

When the `--remote_ip_cache` option is not specified the IP cache is written to the current working directory from which `v++` was launched. You can use this option to provide a different cache location, used across multiple projects for instance.

For example:

```
v++ --remote_ip_cache /tmp/IP_cache_dir ...
```

--report_dir

- **Applies to:** Compile, Link, Package

```
--report_dir <dir_name>
```

Specifies a directory to store report files into. If `--report_dir` is not specified, the tool saves the report files to `./_x/reports`. Refer to Output Directories of the v++ Command for more information.

For example:

```
v++ --report_dir /tmp/myProj_reports ...
```

-R | --report_level

- **Applies to:** Compile, Link, Package

```
--report_level <arg>
```

Valid report levels: `0`, `1`, `2`, `estimate`.

These report levels have mappings kept in the `optMap.xml` file. You can override the installed `optMap.xml` to define custom report levels.

- `-R0` specification turns off all intermediate design checkpoint (DCP) generation during Vivado implementation. Turns on post-route timing report generation.
- The `-R1` specification includes everything from `-R0`, plus `report_failfast pre-opt_design`, `report_failfast post-opt_design`, and enables all intermediate DCP generation.
- The `-R2` specification includes everything from `-R1`, plus `report_failfast post-route_design`.
- The `-Restimate` specification forces Vitis HLS to generate a `design.xml` file if it does not exist and then generates a System Estimate report, as described in System Estimate Report in the *Data Center Acceleration using Vitis* (UG1700).

For example:

```
v++ -R2 ...
```

--reuse_bit

```
--reuse_bit <arg>
```

- **Applies to:** Link

Specifies the path and file name of generated bitstream file (`.bit`) to use when generating the device binary (`xclbin`) file. As described in Using --to_step and Launching Vivado Interactively in the *Data Center Acceleration using Vitis* (UG1700), you can specify the `--to_step` option to interrupt the Vitis build process and manually place and route a synthesized design to generate the bitstream.

> **IMPORTANT!** *The `--reuse_bit` option is a sequential build option that requires you to use the same project directory when resuming the Vitis compiler with `--reuse_bit` that you specified when using `--to_step` to start the build.*

For example:

```
v++ --link --reuse_bit ./project.bit
```

--reuse_impl

```
--reuse_impl <arg>
```

- **Applies to:** Link

Specifies the path and file name of an implemented design checkpoint (DCP) file to use when generating the device binary (`xclbin`) file. The link process uses the specified implemented DCP to extract the FPGA bitstream and generates the `xclbin`. You can manually edit theVivado project created by a previously completed Vitis build, or specify the `--to_step` option to interrupt the Vitis build process and manually place and route a synthesized design, for instance. This allows you to work interactively with Vivado Design Suite to change the design and use DCP in the build process.

> **IMPORTANT!** *The `--reuse_impl` option is an incremental build option that requires you to use the same project directory when resuming the Vitis compiler with `--reuse_impl` that you specified when using `--to_step` to start the build.*

For example:

```
v++ --link --reuse_impl ./manual_design.dcp
```

-s | --save-temps

- **Applies to:** Compile, Link, Package

```
--save-temps
```

Directs the `v++` command to save intermediate files/directories created during the compilation and link process. Use the `--temp_dir` option to specify a location to write the intermediate files to.

> **TIP:** *This option is useful for debugging when you encounter issues in the build process.*

***Note*:** This option is mandatory if using Bootgen to package the SD card, flash or equivalent.

For example:

```
v++ --save-temps ...
```

-t | --target

- **Applies to:** Compile, Link, Package

```
-t [ hw_emu | hw ]
```

Specifies the build target, as described in Selecting the Build Target in the *Data Center Acceleration using Vitis* (UG1700). The build target determines the results of the compilation and linking processes. You can choose to build an emulation model for debug and test, or build the actual system to run in hardware. The build target defaults to `hw` if `-t` is not specified.

> **IMPORTANT!** *The specified platform and build targets for compiling and linking must match. The `--platform` and `-t` options specified when the XO file is generated by compilation must be the `--platform` and `-t` used during linking.*

The valid values are as follows:

- `hw_emu`: Hardware emulation
- `hw`: Hardware

For example:

```
v++ --link -t hw_emu
```

--temp_dir

- **Applies to:** Compile, Link, Package

```
--temp_dir <dir_name>
```

This allows you to manage the location where the tool writes temporary files created during the build process. The temporary results are written by the `v++` compiler, and then removed, unless the `--save-temps` option is also specified.

If `--temp_dir` is not specified, the tool saves the temporary files to `./_x/temp`. Refer to Output Directories of the v++ Command for more information.

For example:

```
v++ --temp_dir /tmp/myProj_temp ...
```

--to_step

- **Applies to:** Compile, Link, Package

```
--to_step <arg>
```

Specifies a step name, for either the compile or link process, to run the build process through that step. You can use the `--list_step` option to determine the list of valid compile or link steps.

The build process terminates after completing the named step. At this time, you can interact with the build results. For example, manually accessing the HLS project or the Vivado Design Suite project to perform specific tasks before returning to the build flow, launch the `v++` command with the `--from_step` option.

IMPORTANT! *`--to_step`/`--from_step` are sequential build options that require you to use `--from_step` to resume the build in the same project directory that you used when starting the build with `--to_step`.*

You must also specify `--save-temps` when using `--to_step` to preserve the temporary files required by the Vivado tools. For example:

```
v++ --link --save-temps --to_step vpl.update_bd
```

--user_board_repo_paths

- **Applies to:** Link

```
--user_board_repo_paths
```

Specifies an existing user board repository for DIMM board files. This value is pre-pended to the `board_part_repo_paths` property of the Vivado project.

--user_ip_repo_paths

- **Applies to:** Link

```
--user_ip_repo_paths <repo_dir>
```

Specifies the directory location of one or more user IP repository paths to be searched first for IP used in the kernel design. This value is appended to the start of the `ip_repo_paths` used by the Vivado tool to locate IP cores. IP definitions from these specified paths are used ahead of IP repositories from the hardware platform (`.xsa`) or from the AMD IP catalog.

TIP: *Multiple `--user_ip_repo_paths` can be specified on the `v++` command line.*

The following lists show the priority order in which IP definitions are found during the build process, from high to low.

***Note*:** All of following entries can possibly include multiple directories in them.

- For the system hardware build (`-t hw`):
  1. IP definitions from `--user_ip_repo_paths`.
  2. Kernel IP definitions (`vpl --iprepo` switch value).
  3. IP definitions from the IP repository associated with the platform.
  4. IP cache from the installation area (for example, `<Install_Dir>/Vitis/2025.2/data/cache/`).
  5. AMD IP catalog from the installation area (for example, `<Install_Dir>/Vitis/2025.2/data/ip/`)
- For the hardware emulation build (`-t hw_emu`):
  1. IP definitions and User emulation IP repository from `--user_ip_repo_paths`.
  2. Kernel IP definitions (`vpl --iprepo` switch value).
  3. IP definitions from the IP repository associated with the platform.
  4. IP cache from the installation area (for example, `<Install_Dir>/Vitis/2025.2/data/cache/`).
  5. `$::env(XILINX_VITIS)/data/emulation/hw_em/ip_repo`
  6. `$::env(XILINX_VIVADO)/data/emulation/hw_em/ip_repo`
  7. AMD IP catalog from the installation area (for example, `<Install_Dir>/Vitis/2025.2/data/ip/`)

For example:

```
v++ --user_ip_repo_paths ./myIP_repo ...
```

-v | --version

```
-v
```

Prints the version and build information for the `v++` command. For example:

```
v++ -v
```

--vivado

- **Applies to:** Link

Specifies properties and parameters to configure the Vivado synthesis and implementation environment prior to building the device binary. See --vivado Options for more information.

v++ Compilation Options

The v++ command provides several features for compilation of PL kernels and AI Engine graphs. These options can be broken down into the following categories:

- `--compile`: These are the classic top-down compilation commands for the Vitis acceleration flow. This flow is still supported, and in some cases necessary. For more information refer to the section on v++ Command.
- `--mode aie`: This compilation mode in the `v++` compiler enables the generation of the AI Engine graph application, and also supports the use of the `x86simulator` and `aiesimulator` tools for analysis. These options are described in detail in v++ Mode AI Engine.
- `--mode hls`: This compilation mode in the `v++` compiler enables the bottom-up generation of an HLS component, either for the AMD Vivado™ IP flow or the Vitis Kernel flow as described in *Vitis High-Level Synthesis User Guide* (UG1399). The options to configure and analyze an HLS component are described in v++ Mode HLS.

v++ General Compilation Options

The general `v++` compilation options can appear on the command-line or in a configuration file. It can include the `-c` (or `--compile`) option, a `--mode` when generating an AI Engine component or HLS component, the `--platform` or `--part`, the build `--target`, and a `--config` file as shown below:

```
v++ -c --mode aie --platform vek385_base --target hw \
-config ./aie_config.cfg <input_files...>
```

TIP: *Relative paths used on the command line are relative to the current working directory where the command is launched. Relative paths in the config file are relative to the location of the config file.*

Command options for `v++` can be generally be used in a config file or on the command-line. However, the following options are only available for use on the command-line:

- **-c [--compile]:**

  Run compilation mode.

- **--config <file_name>:** Specifies the config file path and name.
- **--mode [aie | hls]:**

  Specifies the compilation mode as either compile an AI Engine component, or an HLS component.

IMPORTANT! *The absence of `--mode` puts the `v++ -c` command in the traditional top-down compilation mode for the Vitis kernel flow as described in Compiling C/C++ PL Kernels in the Data Center Acceleration using Vitis (UG1700).*

- **-h [--help]:**

  Print usage message for options. Can be combined with the `--mode` option to list the available options for the AI Engine or HLS components.

- **-v [--version]:**

  Prints version information.

- **--input_files <arg>:**

  Specify input file(s). Input file(s) can also be specified positionally without using the `--input_files` option.

- **-o [--output] <arg>:**

  Applies to: Link, Package. Specify the output file name and path. Default: `a.xclbin` for the `v++ --link` command.

- **-f [--platform]:**

  This is a path to a Vitis platform file that defines the hardware and software components available for AI Engine components, HLS components, and Applications. The argument can be a hardware platform (XSA) as recommended for hardware target in Versal devices, or a base platform (XPFM).

- **--log_dir <arg>:**

  Specify a directory to copy internally generated log files.

- **--macro_dir <arg>:**

  Specify a macro to define a base directory: `<macro_name>=<base_directory>` Macros with a $ sign can be used to specify a relative path.

- **--report_dir <arg>:**

  Specify a directory to copy report files.

- **--work_dir <arg>:**

  Optionally specify a working directory for the build and output files. For `--mode aie` the work directory defaults to `./Work`. For `--mode hls` it defaults to the top-level function specified by `hls.syn.top`. This option can only be specified on the command line.

- **--aie_legacy:**

  Use legacy options for `aiecompiler`.

The following options can either be used in a config file or on the command-line:

TIP: *As these are general options they do not need to be placed under a header (such as `[AIE]` or `[HLS]`) in the config file.*

- **-t [--target] <arg>:**

  The `--target` argument has different values when used with or without `--mode`.

  - Used with `--mode` the target defines the compilation target of the AI Engine or HLS component.
    - `x86`: Specifies compilation for x86 simulation of an AI Engine component, or for C-simulation of the HLS component.
    - `hw`: Specifies compilation for use with AI Engine simulator for AI Engine components, for C/RTL Co-simulation for HLS components, or to run on the physical device for either components.
  - Used without `--mode`: the target defines the tradition build targets (`hw_emu`, `hw`) of the Application flow as described in Creating an Integration Project Component.

  Specify a target as `hw_emu`, or `hw` for the classic `v++` compilation mode; or as `x86` simulation for AI Engine compilation mode or `hw` for AI Engine simulation. The default target is `hw`.

- **--part <arg>:**

  Specify part family or part value. Note this option is mutually exclusive with `--platform` or `--hls.board`.

- **-D [--define] <name=definition>:**

  Predefine `<name>` as a macro with `<definition>`.

- **-I [--include] <arg>:**

  Include the specified directory in the list of directories to be searched for header files.

v++ Mode AI Engine

The AI Engine compilation mode provides for the development, optimization, analysis, and export of AI Engine graph applications (`libadf.a`). The AI Engine compilation mode uses the following command:

```
v++ -c --mode aie --target hw --platform vek385_base \
--work_dir ./myWork --config ./config.cfg <Input File>
```

- `v++ -c --mode aie --target hw`: Indicates compilation of an AI Engine component for the HW target (default) indicating the generation of the `libadf.a` file for use in AI Engine simulation and to run on the physical device. You can also specify `x86sim` for the target to build the component for x86 functional simulator. The contents of the output directory depend on the target.

- `--platform vek385_base`: Specifies the platform or physical device to build the component for.
- `--work_dir ./myWork`: Specifies the work directory to use for building the AI Engine component.

  TIP: *By default, the compiler writes all outputs to a directory called `Work` in a sub-directory of the current directory where the tool was launched, and creates a file called `libadf.a` in the current directory.*

- `--config ./config.cfg`: Specifies the AI Engine component config file containing a variety of options. The options are available to the AI Engine compiler, x86 Simulator, and AI Engine Simulator. The options are part of the analysis and profiling tools for the AI Engine component.
- `<Input File>` specifies the data flow graph code that defines the `main()` application for the AI Engine graph. The input flow graph uses a data flow graph language. For a description of the data flow graph, refer to Introduction to Graph Programming in the *AI Engine Kernel and Graph Programming Guide* (UG1079).

Refer to the `v++ -c --mode aie` command options in the following sections. Add the AI Engine options to a configuration file under the `[AIE]` section.

TIP: *Some `[AIE]` options can be specified on the command line instead of in a config file, although using a config file is the recommended approach. To use an `[AIE]` option on the command line add `--aie.` to the start of the command name from the following sections. For example, to specify the constraints from the command line use `--aie.constraints`*

The AI Engine component (with `--target hw`) can run through the `aiesimulator` using the following command:

```
aiesimulator --pkg-dir=./Work --i=../..
```

`--pkg-dir` specifies the Work directory where compilation occurs. `--i` specifies the input directory path for the simulator.

After compilation, the AI Engine component (with `--target x86sim`) can then run through the `x86simulator` using the following command.

```
x86simulator --pkg-dir=./Work --i=../../
```

AI Engine Options

The following options apply to the AI Engine compilation process.

- **constraints:**

  Specifies one or more constraint JSON files for location and bounding box, etc.

  ```
  constraints=constraints.json
  ```

- **enable-partition:**

  The Vitis command line option to create an AI Engine partition. The partition is within the AI Engine array and specifies the starting column, the number of columns and the partition's name.

  ```
  enable-partition=<START_COLUMN>:<NUM_OF_COLUMN>:<PARTITION_NAME>
  ```

  Note: For more details regarding the use of this setting, see Compiling AI Engine Graph for Independent Partitions in the *AI Engine Tools and Flows User Guide* (UG1076).

- **heapsize:**

  Heap size (in bytes) that the AI Engine uses.

  In an AI Engine, the size of the heap determines the maximum amount of dynamically allocated memory at any given time.

  The heap size for each AI Engine is measured in bytes. The default size is 1024 bytes.

  The stack, heap, and sync buffer allocate up to the tile data memory. The sync buffer is a memory of size 32 bytes. It stores graph run iteration information. Before changing the heap size to a different value, ensure that the sum of the stack, heap, and sync buffer sizes does not exceed the tile data memory. For example, this size is 32768 bytes in an AI Engine device.

  This option is also useful for allocating remaining file-scoped data not explicitly connecting to the user graph.

  ```
  heapsize=512
  ```

- **pl-freq:**

  Specifies the interface frequency (in MHz) for all PL kernels and PLIOs. The default frequency is a quarter of the AI Engine clock frequency. The maximum frequency is half of the AI Engine clock frequency. You can provide the PL frequency specific to the interface in the graph. You can override this value at v++ --link time.

  ```
  pl-freq=500
  ```

  Note: AMD recommend setting port frequency explicitly to avoid ambiguity. This is specifically important when using Vitis Subsystem design flow.

- **pl-register-threshold:**

  Specifies the frequency threshold in MHz for registered AI Engine-PL crossings. The default frequency is one-eighth of the AI Engine clock frequency.

  ```
  pl-register-threshold=300
  ```

Note: For more details regarding the use of this setting, see *AI Engine Kernel and Graph Programming Guide* (UG1079).

> **IMPORTANT!** *The system ignores values above a quarter of the AI Engine array and uses a quarter instead.*

- **stacksize:**

  Stack size (in bytes) used by each AI Engine tile. The default stack size is set to 1024 bytes. Used as a standard compiler calling convention including stack-allocated local variables and register spilling.

  The stack is a region of memory used by programs to store temporary data during execution. In the case of an AI Engine, every tile relies on a specific amount of stack memory to run the kernel mapped to it. The amount of memory allocated as stack space is measured by the number of bytes assigned to each tile. This size of the stack is an essential consideration when designing and optimizing the performance of the AI Engine.

  By default, the stack size for each tile is set at 1024 bytes. This value takes into account factors like stack-allocated local variables and register spilling. The factors are due to a register no longer having room for data, so it is "spilled" onto the stack temporarily to free up register space. It is essential to optimize the stack size for each tile to prevent potential issues that could arise. These issues include memory overflows that can lead to errors and other performance problems.

  > **TIP:** *The system allocates up to 32768 bytes of data memory to the stack, heap, and sync buffer (32 bytes). Before changing the stack size, ensure the sum of the stack, heap, and sync buffer sizes does not exceed 32768 bytes.*

  ```
  stacksize=512
  ```

CDO Options

The CDO options apply to generator code for graph configuration and initialization in configuration data object (CDO) format. The system uses the options during SystemC-RTL simulation and during actual hardware execution.

- **broadcast-enable-core:**

  Enables all AI Engine cores associated with a graph using broadcast. This option reserves one broadcast channel in the array for core enabling purpose. This is enabled by default. You can be disable it by setting the argument to false.

  ```
  broadcast-enable-core=false
  ```

Compiler Debugging Options

The Debugging options specify features of the compiler to enable troubleshooting errors during the build process. These features include increased log details and consistency checking in the source code.

- **adf-api-log-level:**

  ADF API log-level. Available values are as follows:

  - 0: errors
  - 1: level-0 + warnings
  - 2: level-1 + info messages
  - 3: level-2 + debug messages

  The default is 2.

  ```
  adf-api-log-level=3
  ```

- **kernel-linting:**

  Perform consistency checking between graphs and kernels. Accepted values are true and false. The default is false.

  ```
  kernel-linting=true
  ```

- **quiet:**

  Suppress output of the compiler. Accepted values are true and false. The default is false.

  ```
  quiet=true
  ```

- **verbose:**

  Verbose output of the compiler. Accepted values are true and false. The default is false.

  ```
  verbose=true
  ```

  TIP: *Can be used with* `log-level` *to increase the verbosity of the logs. For example:*

  ```
  log-level=4
  ```

Design Rule Check Options

The AI Engine mapper and router support configuring a list of Design Rule Checks (DRCs). Enter the DRC commands below into the config file outside of any heading (for example `[AIE]`).

- **drc.disable:**

Disable the DRC rule check for the specified ID. If you disabled a check, it does not execute.

```
drc.disable=AIE-ROUTER-3
```

- **drc.enable:**

  Enable a previously disabled DRC rule check for the specified ID.

  ```
  drc.enable=AIE-ROUTER-3
  ```

- **drc.severity:**

  Change the severity of a DRC rule check. The format of the argument is `<ID>:<severity>[:context]`

  Where:

  - <ID> is the DRC rule ID which can be found on DRC messages reported by the tool.
  - <severity> represents the new severity to assign to the specified ID.
  - [context]

  ```
  drc.severity=AIE-ROUTER-3:warning
  ```

- **drc.waive:**

  Waive the Design Rule Check for the specified ID. The system still performs a check, but marks it as waived.

  ```
  drc.waive=AIE-ROUTER-3
  ```

File Options

This section describes file options for file management activities through the config file. These options include specifying include files and output file names.

- **include:**

  Includes additional directories in the include path for the compiler front-end processing. Specify one or more include directories.

- **output:**

  Specifies an `output.json` file the front-end produces for input data flow graph file. The output file passes to the back-end for mapping and AI Engine device code generation. Other types of inputs ignores this file.

- **output-archive:**

Specifies the output archive name that contains compiled AI Engine artifacts (default: `libadf.a`). The output archive writes to the component directory above the Work directory.

```
output-archive=myGraph.a
```

Module Specific Options

Options that apply to kernel implementation on AI Engine tiles.

IMPORTANT! *Only modified AI Engine kernels are recompiled in subsequent compilations of the AI Engine graph. Any un-modified kernels are not recompiled. It is possible for the application of these options to require recompilation of the touching kernel source.*

- **Xchess arg:**

  Passes kernel specific options to the AI Engine compiler, which in turn uses the CHESS compiler to compile code for each AI Engine. Specify the option with the string `<kernel>:<optionid>=<value>`. The compilation of source files on the kernel mapped AI Engine includes the option string.

  ```
  Xchess=main:darts.xargs=-nb
  ```

- **Xelfgen arg:**

  Passes additional command-line options to the ELF generation phase of the compiler. Currently, the options run as a `make` command to build all AI Engine ELF files. Provide the values for this option in quotes, useful if you want to pass special characters. For example, `Xelfgen="CLANG_OPTS='-g1 -mllvm --issue-limit=6"`.

  For example, to limit the number of parallel compilations to four, use `Xelfgen="-j4"`.

  Note: If errors with `bad_alloc` appear in the log, or if the Vitis IDE crashes, it can be due to insufficient memory on your workstation. A possible workaround is to limit the parallelism used by the compiler during code generation. Pass this option to the Elf Generator Makefile (`-j1` or `-j2`).

  ```
  Xelfgen=-j2
  ```

- **Xmapper arg:**

  Passes additional command-line options to the mapper phase of the compiler. Useful for when the design is either failing to converge in the mapping or routing phase, or for better performance via reduction in memory bank conflict.

  ```
  Xmapper=DisableFloorplanning
  ```

- **Xpreproc arg:**

Passes general options to the PREPROCESSOR phase for all source code compilations AIE/PS/PL/x86sim of the AI Engine component (e.g., `-D`...).

```
Xpreproc=-D<var>=<value>
```

- **Xpslinker arg:**

  Pass general option to the PS LINKER phase of the AI Engine component (e.g., `-L`... `-l`...).

  ```
  Xpslinker=-L<libpath> -l<libname>
  ```

- **Xrouter arg:**

  Pass general option to the ROUTER phase.

  ```
  Xrouter=dmaFIFOsInFreeBankOnly
  ```

- **Xx86sim arg:**

  Pass x86sim-specific option to the compiler.

  ```
  Xx86sim=clangStaticAnalyzer
  ```

- **fast-floats:**

  Enable fast implementation for linear floating point scalar operations like `add`, `sub`, `mul`, and `compare`. Accepted values are true and false. The default is false.

  ```
  fast-floats=true
  ```

- **fast-nonlinearfloats:**

  Enable fast implementation for non-linear floating point scalar operations like `sine/cosine`, `sqrt`, and `inv`. Accepted values are true and false. The default is false.

  ```
  fast-nonlinearfloats=true
  ```

- **fastmath:**

  Enable fast implementations of `float2fix`, `fplt`, and `fpge`. Accepted values are true and false. The default is false.

  ```
  fastmath=true
  ```

- **float-accuracy:**

  Option available only for AI Engine-ML. It selects the required floating-point compute accuracy emulated with multiple `bfloat16` numerical type operations.

  ```
  float-accuracy=<arg>
  ```

Available argument values are as follows:

- `safe`: Accuracy is slightly better than single precision floating-point (FP32).
- `fast`: Improved performance with similar accuracy to FP32 (default).
- `low`: Best performance with better accuracy than FP16 and `bfloat16`.

Event Tracing Options

This section describes the options to define the event-tracing characteristics of the compiled kernels and graph. These options prepare the design for event tracing during runtime. See Event Trace Build Flow in the *AI Engine Tools and Flows User Guide* (UG1076) for more information.

- **event-trace:**

  Event trace configuration value.

  ```
  event-trace=runtime
  ```

  ***Note*:** This option selects events traced in hardware through an xsdb command line or a line in the file `xrt.ini`.

- **event-trace-port:**

  Sets the AI Engine event tracing port. Accepted values are `plio` and `gmio`.

  The `gmio` option uses the AI Engine-to-NoC pathway to capture trace data.

  The alternative is to use `plio`. This option uses the AI Engine-PL pathway to capture trace data. The option uses programming logic resources to capture data from AI Engine to DDR. The default value is `gmio`. This value is the recommended event-trace-port configuration.

  ```
  event-trace-port=plio
  ```

- **graph-iterator-event:**

  Generates `user_event()` whenever the graph iterator increments. Provides the ability to delay the start of the hardware event trace based on the graph iteration.

- **num-trace-streams:**

  Specifies the number of trace streams. Allows up to 16 streams. The default value is 4.

  ```
  num-trace-streams=8
  ```

- **trace-plio-width:**

  PLIO width for trace streams: 32, 64 (default), 128

Optimization Options

- **xlopt:**

  Enable kernel optimizations based on the specified value:

  - 0: No kernel optimizations.
  - 1: Computation of heap size, generation of guidance based on LLVM IR analysis, and insertion of loop pragmas.
  - 2: The same optimizations as 1 with the addition of loop peeling for unrolled loops, and automatic inlining.

  ```
  xlopt=2
  ```

- **Xxloptstr:**

  Option string to enable or disable optimizations in XLOpt level 1,2.

  - -xlinline-threshold=T: set inlining threshold to T (default T = 5000, only for optlevel=2).
  - -annotate-pragma: insertion of loop unrolling, pipelining, and flattening pragmas (default = true)
  - -align-global-array=0,1 if set to 1, global arrays are aligned on 2 times the width of the physical memory bank. This automatic alignment optimizes load/store performance. Load/Store ports bitwidth is twice the width of a physical bank and aligned on even index physical bank. This option automatically aligns the data to the even index bank address.

  ```
  Xxloptstr=-annotate-pragma
  ```

Miscellaneous Options

- **swfifo-threshold:**

  Above this threshold, the system implements a FIFO as a DMA FIFO in an AI Engine application. The system generates an error in an AI Engine-ML application. To avoid an error message in AI Engine-ML, increase the threshold. Increasing the threshold requires additional resources to route the design.

- **evaluate-fifo-depth:**

  This option analyzes re-convergent data paths. Data can be sent on multiple paths. Sometimes data can re-converge and result in a deadlock. Resolve deadlocks by adding FIFOs to the appropriate data paths.

  The steps for evaluating and resolving deadlocks as a result of re-convergent data paths is as follows:

  1. Compile the design with this option.

2. Run `aiesimulator` on the design.
3. Open Vitis Unified IDE with the simulation `run_summary`.

   Take note of the FIFO depth in theEstimated FIFO column in the Nets table. This FIFO depth value is the recommended value (in 32-bit words) that the `fifo_depth(net)` constraint uses on a specific net connection in the graph. The purpose is to resolve the deadlock situation.
4. Apply the recommended FIFO depth value using the `fifo_depth` constraint on specified nets on the graph, and recompile the design. In AI Engine-ML, if the recommendation is above the current fifo depth threshold, increase it using the `swfifo-threshold` option.

- **disable-multirate:**

  Disable multirate in ADF graphs. Accepted values are true and false. The default is false.

  ```
  disable-multirate=true
  ```

- **known-tripcount:**

  This option converts an unknown trip count to a known trip count. The option takes no arguments and is disabled when omitted.

  ```
  known-tripcount
  ```

- **no-init:**

  This option disables initialization of window buffers in AI Engine data memory. This option enables faster loading of the binary images into the SystemC-RTL co-simulation framework. Accepted values are true and false. The default is false.

  ```
  no-init=true
  ```

- **nodot-graph:**

  By default, the AI Engine compiler produces DOT and PNG files to visualize the user-specified graph and its partitioning onto the AI Engines. This option can be used to eliminate the dot graph output. Accepted values are true and false. The default is false.

  ```
  nodot-graph=true
  ```

- **compile-testbench-only:**

In HW compilation, the AI Engine compiler allows compilation of `main` only (`graph.cpp` test bench) using this flag. This helps to compile `graph.cpp` separately. The option is useful in scenarios where only `main()` or the test bench is changed and there are no changes in graph and kernels. This helps save compilation time of the complete AI Engine design and only compile test bench file whenever necessary.

```
v++ --compile --mode aie --target=hw --aie.compile-testbench-only
graph.cpp
```

Note: The AI Engine compiler assumes that the initial compilation of the AI Engine graph was successful when using `compile-testbench-only`. Ensure that only the test bench code was modified and that no changes were made to graph or kernels.

v++ Mode HLS

The HLS compilation mode provides access to numerous features for the development, optimization, analysis, and export of Vitis kernels (.xo) or Vivado IP (.xci) files. Access the HLS mode command with the following command:

```
v++ -c --mode hls -h [options] <input_files...>
```

Specify the HLS compilation options in a configuration file for the `v++` command using the `--config` option. Place the HLS options under a section head of `[HLS]` in the config file. For example, the following config file specifies the part, the source file, the test bench files, and the flow target. `part` is not specified under the `[HLS]` header because this is a general option for the `v++` compiler.

```
part=xcvu11p-flga2577-1-e

[hls]
clock=8
flow_target=vitis
syn.file=../../src/dct.cpp
syn.top=dct
tb.file=../../src/out.golden.dat
tb.file=../../src/in.dat
tb.file=../../src/dct_test.cpp
tb.file=../../src/dct_coeff_table.txt
syn.output.format=xo
clock_uncertainty=15%
```

The following sections describe the HLS mode command options.

HLS General Options

IMPORTANT! *The following options must appear in the HLS configuration file under the `[hls]` header.*

- **clock:**

Specifies the clock period in `ns` or MHz (`ns` is default). If you don't provide a period, the system uses a default period of 10 ns.

```
clock=8ns
```

> **IMPORTANT!** *If your HLS configuration file uses `platform=` instead of `part=` then you must also specify `freqhz=` instead of `clock=` as shown here to change the default clock frequency of the platform.*

- **clock_uncertainty:**

  Specifies the clock period as a margin of uncertainty by HLS. The system subtracts the margin of uncertainty from the clock period to create an effective clock period. The clock uncertainty is in units of `ns`, or as a percentage of the clock period. The clock uncertainty defaults to 27% of the clock period. When specifying a value, the default unit is `ns` but % or MHz can also be used.

  ```
  clock_uncertainty=15%
  ```

- **flow_target:**

  Sets the flow target to synthesize either a Vitis kernel (`vitis`) or a Vivado IP (`vivado`). The Application Acceleration flow uses the AMD Vitis™ kernel, while the embedded design flow uses the Vivado IP.

  > **IMPORTANT!** *Vivado IP and Vitis kernels have differences in the interface definition.*

- **relative_roots:**

  Specify an ordered list of one or more absolute paths. The list is searched to resolve relative file paths of source files and cflags include paths with similar behavior to PATH environment variable. The first entry of the list can also be used as the root of any output files paths that are relative.

  All entries in the relative_roots list must be an absolute path or be a path that starts with one of the following which have special meaning:

  - `file`: directory that contains the config file where the relative_root is specified
  - `cwd`: current working directory of the v++/vitis-run process

  The list of relative_roots can be specified via multiple entries in a config file. Each successive value appends to the list of entries. Within each entry multiple paths can be specified with a semicolon separator. Whitespace before and after the semicolon is ignored/removed.

  ```
  [hls]
  relative_roots=/tmp/aaa ; /tmp/bbb ; file/../ccc ; cwd
  syn.file=top.cpp
  ```

Note: If relative_roots is specified on the v++/vitis-run command it will be ignored and cause a warning message.

C-Synthesis Sources

- **syn.cflags:**

  Defines compilation flags applying to all `syn.file` defined source files for use during synthesis.

  ```
  syn.cflags=-I../../src/
  ```

- **syn.csimflags:**

  Defines compilation flags to be applied to all `syn.file` source files for use during C-simulation or RTL/Co-simulation.

- **syn.file:**

  Specifies the file path and name of a source file to be used during synthesis of the HLS component. Multiple files require multiple `syn.file` statements.

  Specify the file paths as either absolute or relative. Relative paths are relative to the location of the config file, whether inside the HLS component or outside the component.

  ```
  syn.file=../../src/dct.cpp
  ```

- **syn.file_cflags:**

  Applies a compilation flag for synthesis to the specified source file. Specify the file path and name first, followed by a comma, followed by the cflags:

  ```
  syn.file_cflags=../../src/dct.cpp,-I../../src/
  ```

- **syn.file_csimflags:**

  Applies a compilation flag for simulation to the specified source file. Specify the file path and name first, followed by a comma, followed by the csimflags.

  ```
  syn.file_csimflags=../../src/dct.cpp,-Wno-unknown-pragmas
  ```

- **syn.blackbox.file:**

  Specifies the JSON file for an RTL blackbox. The HLS compiler uses this file during synthesis and when running RTL/Co-simulation.

  ```
  syn.blackbox.file=../../RTL/fft.json
  ```

- **syn.top:**

Specifies the name of the function synthesized as the top-level function for the HLS component. Helps identify the top function in source code that defines multiple functions.

```
syn.top=dct
```

IMPORTANT! *Functions that the top-level function calls also become part of the HLS component.*

Test Bench Sources

- **tb.cflags arg:**

  Defines compilation flags for all source files defined in `tb.file` for simulation or co-simulation.

  ```
  tb.cflags=-Wno-unknown-pragmas
  ```

- **tb.file arg:**

  Specifies the file path and name of a test bench source file for HLS component simulation or co-simulation. Multiple files require multiple `tb.file` statements.

  The file paths can be either absolute or relative. Relative paths are relative to the location of the config file, whether inside the HLS component or outside the component.

  ```
  tb.file=../../src/dct_test.cpp
  ```

- **tb.file_cflags arg:**

  Applies a compilation flag for simulation or co-simulation to the specified test bench source file. Specify the file path and name first, followed by a comma, followed by the cflags:

  ```
  syn.file_cflags=../../src/dct.cpp,-Wno-unknown-pragmas
  ```

Array Partition Configuration

The `syn.array_partition` commands specify the default behavior for array partitioning for the entire design. You can override these settings for specific arrays using the `syn.directive.array_partition`.

- **syn.array_partition.complete_threshold <value>:**

  Sets the threshold for completely partitioning arrays. Arrays with fewer elements than the specified value are partitioned into individual elements.

  ```
  syn.array_partition.complete_threshold=12
  ```

- **syn.array_partition.throughput_driven [auto | off]:**

  Enables automatic partial and/or complete array partitioning.

- `auto`: Enables automatic array partitioning with smart trade-offs between area and throughput. This is the default value.
- `off`: Disables automatic array partitioning.

```
syn.array_partition.throughput_driven=off
```

Array Stencil Configuration

The `syn.array_stencil` command specifies the default behavior of array stenciling for the entire design. You can override these settings for specific arrays using the `syn.directive.array_partition`.

- **syn.array_stencil.throughput_driven:**

  Enables automatic array stenciling with trade-offs between area and throughput.

  - `auto`: Enables automatic array stenciling with smart trade-offs between area and throughput.
  - `off`: Disables automatic array stenciling. This is the default value.

```
syn.array_partition.throughput_driven=auto
```

C-Simulation Configuration

The `csim` options apply to the C-simulation process for validating the design's C/C++ language. Refer to Running C Simulation in the *Vitis High-Level Synthesis User Guide* (UG1399) for more information.

- **csim.O:**

  Enables optimizing compilation, which eliminates debug constructs. The default value is false. Compiles in debug mode to enable debugging.

```
csim.O=true
```

- **csim.argv:**

  Specifies an argument list for the behavioral test bench. The `<arg>` passes to the `main()` function in the C test bench.

```
csim.argv=arg1 arg2
```

- **csim.clean:**

  Enables clean build. The default is false. Without this option, the design compiles incrementally.

```
csim.clean=true
```

- **csim.profile_tripcount:** Enables trip count profiling.

```
csim.profile_tripcount=true
```

- **csim.ldflags:**

  Specifies the options passed to the linker for simulation. This option is typically used to pass include path information or library information for the C/C++ test bench.

```
csim.ldflags=ldExample
```

- **csim.mflags:**

  Provides options passed to the compiler for C simulation for speeding up compilation.

```
csim.mflags=mExample
```

- **csim.profile_tripcount:**

  Enables loop trip count profiling, default is `false`.

```
csim.profile_tripcount=true
```

- **csim.setup:**

  Creates a simulation binary in the `csim` directory of the current HLS component. You can launch simulation later from the compiled executable. The default value is false. Simulation runs after completing setup.

```
csim.setup=true
```

Co-Simulation Configuration

The `cosim` options apply to the C/RTL Co-Simulation process for validating the RTL produced by HLS synthesis. The options include using the C/C++ test bench used earlier in C-simulation. The options also include using the RTL design in behavioral simulation as described in C/RTL Co-Simulation in Vitis HLS in the *Vitis High-Level Synthesis User Guide* (UG1399).

- **cosim.O:**

  Enables optimizing compilation which eliminates debug constructs. The default value is false. Performs compilation in debug mode to enable debugging. Enabling optimized compilation of the C/C++ test bench and RTL wrapper increases compilation time, but results in better runtime performance.

```
cosim.O=true
```

- **cosim.argv:**

Specifies an argument list for the behavioral test bench. The specified `<arg>` passes to the `main()` function in the C test bench.

```
cosim.argv=arg1 arg2
```

- **cosim.compiled_library_dir:**

  Specifies the compiled library directory used during simulation with third-party simulators. The `<arg>` is the path name to the compiled library directory. Compile the library ahead of time using the `compile_simlib` command as explained in the *Vivado Design Suite Tcl Command Reference Guide* (UG835).

  ```
  cosim.compiled_library_dir=../../simLib
  ```

- **cosim.coverage:**

  Enables the coverage feature during simulation with the VCS simulator.

  ```
  cosim.coverage=true
  ```

- **cosim.disable_binary_tv:**

  Disables the binary test vector format in co-simulation.

  ```
  cosim.disable_binary_tv=true
  ```

- **cosim.disable_deadlock_detection:**

  Disables deadlock detection, and opening the Cosim Deadlock Viewer in co-simulation.

  ```
  cosim.disable_deadlock_detection=true
  ```

- **cosim.disable_dependency_check:**

  Disables dependency checks when running co-simulation.

  ```
  cosim.disable_dependency_check=true
  ```

- **cosim.enable_dataflow_profiling:**

  Enables the dataflow channel profiling to track channel sizes during co-simulation. You must enable this feature to capture dataflow data as described in the Dataflow viewer section of the *Vitis High-Level Synthesis User Guide* (UG1399).

  ```
  cosim.enable_dataflow_profiling=true
  ```

- **cosim.enable_fifo_sizing:**

Enables automatic FIFO channel size tuning for dataflow profiling during co-simulation.

```
cosim.enable_fifo_sizing=true
```

- **cosim.enable_tasks_with_m_axi:**

  Enables stable `m_axi` interfaces for use with `hls::task`.

  ```
  cosim.enable_tasks_with_m_axi=true
  ```

- **cosim.hwemu_trace_dir:**

  Specifies the location of test vectors generated during hardware emulation for a test bench during co-simulation. The `syn.rtl.cosim_trace_generation` command generates the test vectors as described in RTL Configuration. This argument specifies the kernel and instance name of the Vitis kernel in the hardware emulation simulation results. The goal is to help you locate the test vectors for the HLS component.

  ```
  cosim.hwemu_trace_dir=../../dct/dct_2
  ```

- **cosim.ldflags <arg>:**

  Specifies the options passed to the linker for simulation. Passes include path information or library information for the C/C++ test bench.

  ```
  cosim.ldflags=ldExample
  ```

- **cosim.mflags <arg>:**

  Provides options passed to the compiler for C simulation. Can speed up compilation.

  ```
  cosim.mflags=mExample
  ```

- **cosim.random_stall:**

  Enables random stalling of top level interfaces during co-simulation.

  ```
  cosim.random_stall=true
  ```

- **cosim.rtl:**

  Specifies either Verilog or VHDL as the language to use for C/RTL co-simulation. The default is Verilog.

  ```
  cosim.rtl=vhdl
  ```

- **cosim.setup:**

The system creates the simulation binary in the `cosim` directory of the current HLS component. However, this does not perform simulation. You can launch simulation later from the compiled executable. The default is value is false. Co-simulation runs after setup is complete.

```
cosim.setup=true
```

- **cosim.stable_axilite_update:**

  Enables `s_axilite` to configure registers that are stable compared with the prior transaction.

  ```
  cosim.stable_axilite_update=true
  ```

- **cosim.tool:**

  Specifies the HDL simulator for co-simulating the RTL with the C test bench. The Vivado simulator (xsim) is the default, unless otherwise specified.

  - auto
  - vcs
  - modelsim
  - riviera
  - isim
  - xsim
  - ncsim
  - xceilum

  ```
  cosim.tool=modelsim
  ```

- **cosim.trace_level:**

  Determines the level of waveform trace data to save during C/RTL co-simulation.

  - `none` does not save trace data. This is the default.
  - `all` results in all port and signal waveforms being saved to the trace file.
  - `port` only saves waveform traces for the top-level ports.
  - `port_hier` save the trace information for all ports in the design hierarchy.

  ```
  cosim.trace_level=port
  ```

  The trace file saves in the `sim/Verilog` or `sim/VHDL` folder of the component when the simulation executes. The location depends on the selection in the `cosim.rtl` option.

- **cosim.user_stall:**

  Specifies the JSON stall file used during co-simulation. The stall file can be generated using the `cosim_stall` command.

  ```
  cosim.user_stall=../../stall.json
  ```

- **cosim.wave_debug:**

  Opens the Vivado simulator GUI to view waveforms and simulation results. Enables waveform viewing of all processes in the generated RTL. The proccesses include dataflow and sequential processes. Use this option only when using the Vitis simulator for co-simulation. You must set `cosim.tool=xsim`. See Viewing Simulation Waveforms for more information.

  ```
  cosim.wave_debug=true
  ```

Compile Options

The `syn.compile` commands specify the default behavior for compilation of the HLS component.

- **syn.compile.design_size_max_warning:** Specifies the design size that triggers a warning related to slow compilation or poor QoR.

  ```
  syn.compile.design_size_maximum_warning=300000
  ```

- **syn.compile.enable_auto_rewind:**

  When set to true, enables an alternative HLS implementation for pipelined loops that uses automatic loop rewind. See the Rewinding Pipelined Loops for Performance section of the *Vitis High-Level Synthesis User Guide* (UG1399) for additional information.

  ```
  syn.compile.enable_auto_rewind=1
  ```

- **syn.compile.ignore_long_run_time:**

  Disables the long runtime warning.

  ```
  syn.compile.ignore_long_run_time=1
  ```

- **syn.compile.name_max_length <arg>:**

  Specifies the maximum length of function names. If the length is over the threshold, the system truncates the last part of the name.

  ```
  syn.compile.name_max_length=13
  ```

- **syn.compile.no_signed_zeros:**

The `no_signed_zeros` option ignores the signedness of floating point zero. By ignoring the signedness, the compiler can perform aggressive optimizations on floating point operations.

```
syn.compile.no_signed_zeros=1
```

- **syn.compile.performance_budgeter:**

  By enabling this option, the top-level performance pragma conducts a comprehensive, design-wide performance analysis. Much like selecting the most efficient route from point A to point B among many possibilities, the analysis considers multiple optimization strategies across the entire design. This analysis helps identify the best path to meet performance targets.

  enable: Activates the performance budget for a comprehensive, design-wide performance analysis.

  disable: Deactivates the performance budget, preventing any comprehensive analysis.

  auto: Automatically enables design-wide performance analysis if the system detects a performance pragma.

- **syn.compile.pipeline_flush_in_task arg:**

  Specifies that pipelines in `hls::tasks` is flushing (`flp`) by default to reduce the probability of deadlocks in C/RTL Co-simulation. This option applies to pipelines that achieve an II=1 with the default option of `ii1`. However, you can also specify it as applying `always` to enable flushing pipelines in either `hls::tasks` or dataflow, or can be completely disabled using `never`.

  IMPORTANT! *Flushing pipelines (`flp`) are compatible with the rewind option specified in the PIPELINE pragma or directive when the `syn.compile.enable_auto_rewind` option is also specified.*

  - **`always`:** Always make pipelines flushable in `hls::tasks` or dataflow regardless of II.
  - **`never`:** Never make pipeline flushable unless specifically overridden by other directives or pragmas.
  - **`ii1`:** Make pipelines that achieve II=1 flushable in `hls::tasks`. This is the default setting.

```
syn.compile.pipeline_flush_in_task=always
```

- **syn.compile.pipeline_loops arg:**

  Loops with a tripcount equal to or greater than the value you specify in this setting is pipelined automatically.

```
syn.compile.pipeline_loops=20
```

- **syn.compile.pipeline_style arg:**

Set default pipeline style, this is a preference not a hard constraint. Refer to Flushing Pipelines and Pipeline Types in *Vitis High-Level Synthesis User Guide* (UG1399) for more information. The three styles are: stallable (`stp`), flushable (`flp`), and free-running (`frp`). The default is free-running.

```
syn.compile.pipeline_style=flp
```

- **syn.compile.pragma_strict_mode:**

  Enable errors instead of warnings for unrecognized and improper pragma syntax.

```
syn.compile.pragma_strict_mode=1
```

- **syn.compile.unsafe_math_optimizations:**

  The `unsafe_math_optimizations` option ignores the signedness of floating point zero and enables associative floating point operations. Allows compiler to do aggressive optimizations on floating point operations.

```
syn.compile.unsafe_math_optimizations=1
```

> **IMPORTANT!** *Using this option can change the result of any floating point calculations and result in a mismatch in C/RTL co-simulation. Ensure your test bench is tolerant of differences and checks for a margin of difference, not exact values.*

- **syn.compile_use_csim_directives:**

  Use directives generated by C simulation during C synthesis. Default is `false`.

```
syn.compile_use_csim_directives=true
```

Dataflow Configuration

Synthesis: Dataflow Options

The `syn.dataflow` commands configure the dataflow analysis for the whole design. These settings specify the default memory channel and FIFO depth used by `syn.directive.dataflow`.

- **syn.dataflow.default_channel:**

  By default, when configuring RAM memory in ping-pong fashion and use dataflow pipelining, the Vitis tool uses the data between functions or loops. When using streaming (where the data is always read and written in consecutive order), a FIFO memory is more efficient. You can select the FIFO memory as the default memory type.

  The available channels are `fifo` and `pingpong`. The default is `pingpong`.

```
syn.dataflow.default_channel=fifo
```

> **TIP:** *Arrays must be set to streaming using the* `set_directive_stream` *command to perform FIFO accesses.*

- **syn.dataflow.disable_fifo_sizing_opt:**

  Disable FIFO sizing optimizations that increase resource usage; this can improve performance and reduce deadlocks.

  ```
  syn.dataflow.disable_fifo_sizing_opt=1
  ```

- **syn.dataflow.fifo_depth:**

  An integer value specifying the default depth of FIFOs. This option has no effect when using `pingpong` memories. By default, the depth of FIFOs in the channel sets to the size of the largest producer or consumer (whichever is largest). In some cases, this approach can be too conservative and results in FIFOs larger than needed. This option can specify the depth when you know the FIFOs are larger than required.

  ```
  syn.dataflow.fifo_depth=6
  ```

  > **IMPORTANT!** *Incorrect use of this option can result in a design that fails to operate correctly.*

- **syn.dataflow.override_user_fifo_depth:**

  Specify a depth for every `hls::stream`, overriding any user settings.

  ```
  syn.dataflow.override_user_fifo_depth=12
  ```

  This setting is useful for checking if a deadlock is due to insufficient FIFO depths in the design. By setting the override to a very large value, if there is no deadlock, you can:

  - Use the FIFO depth profiling options of co-simulation.
  - Use the GUI to find the minimum depth that ensures performance and avoids deadlocks.

  A large value for example is the maximum depth printed by co-simulation at the end of simulation.

- **syn.dataflow.scalar_fifo_depth:**

  An integer value specifying the minimum depth of the scalar value propagation FIFOs. See the Specifying Compiler-Created FIFO Depth section of the *Vitis High-Level Synthesis User Guide* (UG1399). These FIFOs are used to forward the value of scalar arguments of the dataflow regions to processes which have predecessors in the region itself. They do not affect functional correctness. An sufficient automatically computed size can result in loss of performance and even deadlock.

  ```
  syn.dataflow.scalar_fifo_depth=4
  ```

When this option is not specified, the minimum depth is the value of the `syn.dataflow.fifo_depth` option, or it is 2. As a rule of thumb, a good value is the average number of times the process forwarding the scalar value can start before the last process that reads it starts.

- **syn.dataflow.start_fifo_depth:**

  An integer value specifying the minimum depth of the start propagation FIFOs. See Specifying Compiler-Created FIFO Depth in the *Vitis High-Level Synthesis User Guide* (UG1399). These FIFOs forward the `ap_start` handshake signal to processes that have predecessors in the region. They do not affect functional correctness, but an insufficient automatically computed size can result in loss of performance.

  ```
  syn.dataflow.start_fifo_depth=5
  ```

  When you do not specify this option, the minimum depth is either the value of the `syn.dataflow.fifo_depth` option, or the value 2. As a rule of thumb, a good value is the expected average number of times a process should be allowed to start in advance compared to its successors.

- **syn.dataflow.strict_mode:**

  Sets the severity for dataflow canonical form messages. The available modes are: `error`, `warning`, `off`. The default is `warning`.

  ```
  syn.dataflow.strict_mode=error
  ```

- **syn.dataflow.strict_stable_sync:**

  Force synchronization of stable ports with `ap_done`. Refer to the Stable Arrays section of the *Vitis High-Level Synthesis User Guide* (UG1399) for more information.

  ```
  syn.dataflow.strict_stable_sync=1
  ```

- **syn.dataflow.task_level_fifo_depth:**

  Default task-level FIFO depth. For FIFOs automatically created to transfer scalars between processes. A FIFO is synchronized by `ap_ctrl_chain`. The write is the `ap_done` of the producer, the read is the `ap_ready` of the consumer. Like a PIPO in terms of synchronization, and like a FIFO in terms of access.

  ```
  syn.dataflow.task_level_fifo_depth=7
  ```

Debug Options

The `syn.debug` commands enable debugging in the HLS component and specify where to write debugging output.

- **syn.debug.enable:**

Enable debug file generation. If you do not enable it, the HLS you can optimize the component during compilation. However, the component cannot support debug.

```
syn.debug.enable=1
```

> **TIP:** *This option relates to the `v++ -c -g` option. You must manually enable this option in the HLS component when debugging at the Application level.*

- **syn.debug.directory:**

  Specifies the location of to write the output of HLS debugging. If you do not specify it, the location sets to `hls/.debug`.

  ```
  syn.debug.directory=../../debug
  ```

Interface Configuration

The `syn.interface` commands configure the default interface settings applied to the HLS component. These settings can be overridden for specific top-level interface ports using `syn.directive.interface`.

- **syn.interface.clock_enable:**

  Add a clock-enable port (`ap_ce`) to the design. The clock enable prevents all clock operations when it is active-Low, and disables all sequential operations. The default is false.

  ```
  syn.interface.clock_enable=1
  ```

- **syn.interface.default_slave_interface:**

  Specify the default slave interface. The two modes are `s_axilite` and `none`. The default is `s_axilite`.

  ```
  syn.interface.default_slave_interface=none
  ```

- **syn.interface.m_axi_addr64:**

  Enable 64-bit addressing for all `m_axi` interfaces. This option is enabled by default. When disabled, the `m_axi` interfaces uses 32-bit addressing.

  ```
  syn.interface.m_axi_addr64=1
  ```

- **syn.interface.m_axi_alignment_byte_size:**

  Specify default alignment byte size for all `m_axi` interfaces. The default when this option is not specified is 64 bytes for Vitis kernel flow. The default is 1 byte for the Vivado IP flow. See the Vivado/Vitis Flows section of the *Vitis High-Level Synthesis User Guide* (UG1399).

  ```
  syn.interface.m_axi_alignment_byte_size=16
  ```

> **TIP:** *A value of 0 is not valid.*

- **syn.interface.m_axi_auto_id_channel:**

  Enable automatic assignment of channel IDs for `m_axi` interfaces. By default, this option is disabled. Refer to the AXI4 Master Interface section of the *Vitis High-Level Synthesis User Guide* (UG1399) for additional information.

  ```
  syn.interface.m_axi_auto_id_channel=1
  ```

- **syn.interface.m_axi_auto_max_ports:**

  Enable automatic creation of separate `m_axi` interface adapters for each argument or port on the interface. This option is disabled by default, reducing the `m_axi` interfaces to minimum requirements. Refer to the section M_AXI Bundles of the *Vitis High-Level Synthesis User Guide* (UG1399) or more information.

  ```
  syn.interface.m_axi_auto_max_ports=1
  ```

- **syn.interface.m_axi_buffer_impl:**

  Specify the implementation resource for all buffers internal to the `m_axi` adapters. The choices are `auto`, `lutram`, `bram`, `uram`. The default is `bram`.

  ```
  syn.interface.m_axi_buffer_impl=lutram
  ```

- **syn.interface.m_axi_cache_impl:**

  Specify the implementation resource for cache added to the `m_axi` adapters. The choices are `auto`, `lutram`, `bram`, `uram`. The default is `auto`.

  ```
  syn.interface.m_axi_cache_impl=lutram
  ```

- **syn.interface.m_axi_conservative_mode:**

  Configure all `m_axi` adapters to work in conservative mode. The Vitis tool waits to issue a write request until the associated write data is entirely available. Typically, the tool waits until the data is buffered into the adapter or already emitted. The tool uses a buffer inside the M_AXI adapter to store all the data for a burst (both in case of reading and writing). This feature is enabled by default, and might slightly increase write latency but can resolve deadlock due to concurrent requests (read or write) on the memory subsystem. Disable conservative mode by setting it to `false`.

  ```
  syn.interface.m_axi_conservative_mode=0
  ```

- **syn.interface.m_axi_flush_mode:**

Configure all `m_axi` adapters to be flushable, writing or reading garbage data. This option only applies when a burst is interrupted due to pipeline blocking. For example, when missing data inputs when not in conservative mode or missing output space. By default, this option is disabled.

```
syn.interface.m_axi_flush_mode=1
```

- **syn.interface.m_axi_latency:**

  Specify the expected latency of the `m_axi` interface globally. Allows the design to initiate a bus request a number of cycles (latency) before the read or write is expected. The default value is 64 for the Vitis Kernel flow. The default value is for the Vivado IP flow. See the section Defaults of Vivado and Vitis Flows in the *Vitis High-Level Synthesis User Guide* (UG1399).

  ```
  syn.interface.m_axi_latency=5
  ```

- **syn.interface.m_axi_max_bitwidth:**

  Specifies the maximum bitwidth for the `m_axi` interfaces data channel. The default is 1024 bits. The specified value must be a power-of-two, between 8 and 1024. This decreases throughput if the actual accesses are bigger than the required interface. The reason is due to the throughput being split into a multi-cycle burst of accesses.

  ```
  syn.interface.m_axi_max_bitwidth=128
  ```

- **syn.interface.m_axi_max_read_burst_length:**

  Specifies a global maximum number of data values read during a burst transfer for all `m_axi` interfaces. The default is 16.

  ```
  syn.interface.m_axi_max_read_burst_length=12
  ```

- **syn.interface.m_axi_max_widen_bitwidth:**

  Enable automatic port width resizing to widen bursts for the `m_axi` interface, up to the chosen bitwidth. The specified value must be a power of 2 between 8 and 1024, and must align with the `-m_axi_alignment_size`. The default value is 512 for the Vitis Kernel flow, and 0 for the Vivado IP flow.

  ```
  syn.interface.m_axi_max_widen_bitwidth=64
  ```

- **syn.interface.m_axi_max_write_burst_length:**

  Specifies a global maximum number of data values written during a burst transfer for all `m_axi` interfaces. The default is 16.

  ```
  syn.interface.m_axi_max_write_burst_length=12
  ```

- **syn.interface.m_axi_min_bitwidth:**

  Specifies the minimum bitwidth for `m_axi` interfaces data channel. The default is 8 bits. The value must be a power of 2, between 8 and 1024. This does not necessarily increase throughput if the actual accesses are smaller than the required interface.

  ```
  syn.interface.m_axi_min_bitwidth=64
  ```

- **syn.interface.m_axi_num_read_outstanding:**

  Specifies how many read requests can be made to the `m_axi` interface without a response, before the design stalls. This implies internal storage in the design, and a FIFO of size:

  ```
  num_read_outstanding*max_read_burst_length*word_size
  ```

  The default value is 16.

  ```
  syn.interface.m_axi_num_read_outstanding=8
  ```

- **syn.interface.m_axi_num_write_outstanding:**

  Specifies how many write requests can be made to the `m_axi` interface without a response, before the design stalls. This implies internal storage in the design, and a FIFO of size:

  ```
  num_write_outstanding*max_write_burst_length*word_size
  ```

  The default value is 16.

  ```
  syn.interface.m_axi_num_write_outstanding=8
  ```

- **syn.interface.m_axi_offset:**

  Specify the default offset mechanism for all `m_axi` interfaces. The options are:

  - `off`: No offset port is generated.
  - `slave`: Generates an offset port and automatically maps it to an `s_axilite` interface. This option is the default value.
  - `direct`: Generates a scalar input offset port for directly passing the address offset into the IP through the offset port.

  ```
  syn.interface.m_axi_offset=slave
  ```

- **syn.interface.register_io:**

  Globally enables registers for all scalar inputs, outputs, or all scalar ports on the top function. Arrays are always registered. The options are `off`, `scalar_in`, `scalar_out`, `scalar_all`. The default is `off`.

  ```
  syn.interface.register_io=scalar_out
  ```

- **syn.interface.s_axilite_auto_restart_counter:**

  Enables the auto-restart behavior for kernels. Use 1 to enable the auto-restart feature, or 0 to disable it which is the default. When enabled, the tool establishes the auto-restart bit in the `ap_ctrl_chain` control protocol for the `s_axilite` interface. For more information refer to Auto-Restarting Mode in the *Vitis High-Level Synthesis User Guide* (UG1399).

  ```
  syn.interface.s_axilite_auto_restart_counter=1
  ```

- **syn.interface.s_axilite_data64:**

  Enable 64 bit data width for `s_axilite` interface. Use 1 to enable 64 bit data width and 0 to disable it. The default is 32 bit data width.

  ```
  syn.interface.s_axilite_data64=1
  ```

- **syn.interface.s_axilite_interrupt_mode:**

  Specify the interrupt mode for `s_axilite` interface to be Clear on Read (`cor`) or Toggle on Write (`tow`). The Vitis tool can complete clear on Read interrupts in a single transaction, while `tow` requires two. `Tow` is the default interrupt mode.

  ```
  syn.interface.s_axilite_interrupt_mode=cor
  ```

- **syn.interface.s_axilite_mailbox:** Enables the creation of a mailbox for non-stream non-stable `s_axilite` arguments. The mailbox feature sets and manages the auto-restart kernels. See the Auto-Restarting Mode section in the *Vitis High-Level Synthesis User Guide* (UG1399).The available options are:

  - `in`: Enable mailbox for only input arguments
  - `out`: Enable mailbox for only output arguments
  - `both`: Enable mailbox for input and output arguments
  - `none`: No mailbox created. This option is the default value.

  ```
  syn.interface.s_axilite_mailbox=in
  ```

- **syn.interface.s_axilite_status_regs:**

  Enables exposure of ECC error bits in the `s_axilite` register file. Achieves exposure through two clear-on-read (COR) counters per block RAM or URAM with ECC enabled. The options are `ecc` to enable or `off` to disable the feature. The default is set to disabled.

  ```
  syn.interface.s_axilite_status_regs=ecc
  ```

- **syn.interface.s_axilite_sw_reset:**

Enables SW reset in `s_axilite` adapter.

```
syn.interface.s_axilite_sw_reset=1
```

Package Options

Output Options

The package.output options specifies the nature of the exported files that the synthesized RTL design generates. These options include the following:

- **package.output.file:** Specifies the output file path and name for the exported file. If you do not specify this option, the Vitis Unified IDE names the output file after the HLS component. See the following for an example command:

  ```
  package.output.file=../kernel.xo
  ```

- **package.output.format:** Specifies the exported format of the output generated after synthesis. The supported values are:
  - `package.output.format=ip_catalog`: A format suitable for adding to the Vivado IP catalog.
  - `package.output.format=xo`: A format accepted by the `v++` compiler for linking in the Vitis application acceleration flow.
  - `package.output.format=syn_dcp`: Synthesized checkpoint file for Vivado Design Suite. Using this option executes RTL synthesis. Vivado implementation can be optionally added using `vivado.flow=impl`
  - `package.output.format=sysgen`: Generates an IP and .zip archive for use in System Generator.
  - `package.output.format=rtl`: Outputs the RTL files from C synthesis. Does not package the IP or XO used for downstream processes.

  > **TIP:** *The RTL format is useful for development, analysis, and debug of the HLS component. However, to export files you must specify one of the other output formats.*

- **package.output.syn:** Enables the running of the Package step as part of the C synthesis step to generate the required export files during synthesis.

IP Options

The `package.ip` commands specify details of the IP that the HLS component generates. These are settings to define the IP Vendor, Library, Name, and Version (VLNV) for example to identify it in the IP catalog.

- **package.ip.description:**

Provides a description for the catalog entry for the generated IP, used when packaging the IP.

```
package.ip.description=details of IP
```

- **package.ip.display_name:**

  Provides a display name for the catalog entry for the generated IP, used when packaging the IP.

  ```
  package.ip.display_name=randy1
  ```

- **package.ip.library:**

  Provides the library component of the `<Vendor>:<Library>:<Name>:<Version>` (VLNV) identifier for generated IP.

  ```
  package.ip.library=testLib
  ```

- **package.ip.name:**

  Provides the name component of the `<Vendor>:<Library>:<Name>:<Version>` (VLNV) identifier for the generated IP.

  ```
  package.ip.name=randy1
  ```

- **package.ip.taxonomy:**

  Specifies the taxonomy for the catalog entry for the generated IP, used when packaging the IP. Taxonomy is a category to help identify the purpose or application of the IP.

  ```
  package.ip.taxonomy=video
  ```

- **package.ip.vendor:**

  Provides the vendor component of the `<Vendor>:<Library>:<Name>:<Version>` (VLNV) identifier for generated IP.

  ```
  package.ip.vendor=randyCom
  ```

- **package.ip.version:**

  Provides the version component of the `<Vendor>:<Library>:<Name>:<Version>` (VLNV) identifier for generated IP.

  ```
  package.ip.version=2.3
  ```

- **package.ip.xdc_file:**

Specifies an XDC file. The implementation in the Vivado tool uses the contents of the packaged IP.

```
package.ip.xdc_file=../../ip.xdc
```

- **package.ip.xdc_ooc_file:**

  Specifies an out-of-context (OOC) XDC file. The contents of this file are included in the packaged IP and used during out-of-context Vivado synthesis for the exported IP.

```
package.ip.xdc_ooc_file=../../ooc,xdc
```

Operator Configuration

The `syn.op` command lets you configure the default implementation style, latency, and precision for different operators used for the HLS component. You can add multiple `syn.op` commands to a config file to specify the details of different operators. If an operator is not specified, the Vitis tool determines the default values for the component.

You can override the default settings specified by the `syn.op` command by using the syn.directive.bind_op command for specific variables.

- **syn.op:**

  The following is the syntax for the `syn.op` command.

```
syn.op=op:mul impl:dsp
syn.op=op:add impl:fabric latency:6
syn.op=op:fmacc precision:high
syn.op=op:hdiv latency:5
```

  - `syn.op=`: Starts the command
  - `op:<operator>`: Specifies the `op` keyword followed by the operator being defined.
    - `mul`: integer multiplication operation
    - `add`: integer add operation
    - `sub`: integer subtraction operation
    - `fadd`: single precision floating-point add operation
    - `fsub`: single precision floating-point subtraction operation
    - `fdiv`: single precision floating-point divide operation
    - `fexp`: single precision floating-point exponential operation
    - `flog`: single precision floating-point logarithmic operation
    - `fmul`: single precision floating-point multiplication operation

    - `frsqrt`: single precision floating-point reciprocal square root operation
    - `frecip`: single precision floating-point reciprocal operation
    - `fsqrt`: single precision floating-point square root operation
    - `dadd`: double precision floating-point add operation
    - `dsub`: double precision floating-point subtraction operation
    - `ddiv`: double precision floating-point divide operation
    - `dexp`: double precision floating-point exponential operation
    - `dlog`: double precision floating-point logarithmic operation
    - `dmul`: double precision floating-point multiplication operation
    - `drsqrt`: double precision floating-point reciprocal square root operation
    - `drecip`: double precision floating-point reciprocal operation
    - `dsqrt`: double precision floating-point square root operation
    - `hadd`: half precision floating-point add operation
    - `hsub`: half precision floating-point subtraction operation
    - `hdiv`: half precision floating-point divide operation
    - `hmul`: half precision floating-point multiplication operation
    - `hsqrt`: half precision floating-point square root operation
    - `facc`: single precision floating-point accumulate operation
    - `fmacc`: single precision floating-point multiply-accumulate operation
    - `fmadd`: single precision floating-point multiply-add operation

    TIP: *Comparison operators, such as `dcmp`, implements in LUTs. The operators cannot implement outside of the fabric, or map to DSPs. You cannot configure operators with the `syn.op` or `syn.directive.bind_op` commands.*

- `impl:<value>`: Specifies the implementation (`impl`) keyword followed by the value for the specified operator. When `impl` is not specified, the default is for the tool to determine the best implementation for a given operator. Supported values include:
    - `all`: All implementations. This is the default setting.
    - `dsp`: Use DSP resources
    - `fabric`: Use non-DSP resources

    - `meddsp`: Floating Point IP Medium Usage of DSP resources
    - `fulldsp`: Floating Point IP Full Usage of DSP resources
    - `maxdsp`: Floating Point IP Max Usage of DSP resources
    - `primitivedsp`: Floating Point IP Primitive Usage of DSP resources
    - `auto`: enable inference of combined `facc` | `fmacc` | `fmadd` operators
    - `none`: disable inference of combined `facc` | `fmacc` | `fmadd` operators
- `latency:<value>`: Specifies the `latency` keyword followed by the value. Defines the default latency for the binding of the operator to the implementation resource. The valid value range varies for each implementation (`impl`) of the operation. The default is -1, which lets the tool apply the standard latency for the implementation resource.

  TIP: *You can specify the latency for a specific operation without specifying the implementation. This leaves the tool free to choose the implementation while managing the latency.*
- `precision:<value>`: Used for floating point operators (`facc`, `fmacc`, and `fmadd`), this specifies the `precision` keyword followed by one of the following:
    - `low`: Use a low precision (60 bit and 100 bit integer) accumulation implementation when available. This option is only available on certain non-AMD Versal™ devices, and can cause RTL/Co-Sim mismatches due to insufficient precision with respect to C++ simulation. However, devices can be pipelined with an II=1 without source code changes. This option uses approximately three times the resources of `standard` precision floating point accumulation.
    - `high`: Use high precision (one extra bit) fused multiply-add implementation when available. This option is useful for high-precision applications. The options is very efficient on Versal devices, although it can cause RTL/Co-Sim mismatches due to the extra precision with respect to C++ simulation. The option uses more resources than `standard` precision floating point accumulation.
    - `standard`: standard precision floating point accumulation and multiply-add is suitable for most uses of floating-point, and is the default setting. It always uses a true floating-point accumulator that can be pipelined with II=1 on Versal devices, and II that is typically between 3 and 5 (depending on clock frequency and target device) on non-Versal devices.

RTL Configuration

The `syn.rtl` commands configure various attributes of the compiled RTL, the type of reset used, and the encoding of the state machines. The commands also allow you to use specific identification in the RTL. By default, these options are apply to the top-level design and all RTL blocks within the design.

- **syn.rtl.cosim_trace_generation:**

Generates test vectors during hardware emulation in the Vitis tool flow. The option applies when the kernel synthesizes as a Vitis kernel. The process is useful C/RTL Co-simulation test bench in future design iterations.

```
syn.rtl.cosim_trace_generation=1
```

- **syn.rtl.deadlock_detection:**

  Enables simulation or synthesis deadlock detection in the top-level RTL of an exported IP/XO file. The options are as follows:

  - none: Disables deadlock detection.
  - sim: Enables deadlock detection only for simulation/emulation. This is the default setting.
  - hw: Enables detection in synthesized RTL IP. Adds `ap_local_deadlock` and `ap_local_block` signals to the IP to enable local and global deadlock detection.

  ```
  syn.rtl.deadlock_detection=hw
  ```

- **syn.rtl.deadlock_diagnosis:**

  Enables deadlock detection diagnosis for Vitis kernels (.xo) during hardware emulation of the Application.

  ```
  syn.rtl.deadlock_diagnosis=1
  ```

- **syn.rtl.fsm_encoding:**

  Specifies the 'fsm_encoding' RTL attribute value to guide RTL synthesis. The options are as follows:

  - auto: Allow the RTL Synthesis tool to determine the best state machine encoding
  - gray: Uses gray state machine encoding.
  - johnson: Uses johnson state machine encoding.
  - one_hot: Uses one_hot state machine encoding.
  - sequential: Uses sequential state machine encoding.
  - none: Disables state machine encoding. The state machine is synthesized exactly using the state code specified in the RTL (which is one_hot).

  ```
  syn.rtl.fsm_encoding=gray
  ```

- **syn.rtl.fsm_safe_state:**

  Specify the 'fsm_safe_state' RTL attribute value to guide RTL synthesis. This attribute can affect the quality of synthesis results, typically resulting in less performance with greater area. The options are as follows:

- auto_safe_state: Implies Hamming-3 encoding.
- default_state: Return the state machine to the default state.
- power_on_state: Return the state machine to the POWER_ON state.
- reset_state: Return the state machine to the RESET state.
- none: Attribute not added to RTL, the state machine will not include safe state logic.

```
syn.rtl.fsm_safe_state=auto_safe_state
```

- **syn.rtl.header:**

  Specify a file whose contents will be inserted at the beginning of all generated RTL files. This allows you to ensure that the generated RTL files contain user specified content.

  ```
  syn.rtl.header=../../myHeader.txt
  ```

- **syn.rtl.kernel_profile:**

  Add top level event and stall ports required by kernel profiling.

  ```
  syn.rtl.kernel_profile=1
  ```

  IMPORTANT! *This option relates to the `v++ -c --profile.stall` command. You must manually add this option to the HLS component to ensure the stall profiling is available for the linked application.*

- **syn.rtl.module_auto_prefix:**

  Specifies the top level function name as the prefix value for generated RTL modules. This option is ignored if `syn.rtl.module_prefix` is also specified. This option is enabled by default.

  ```
  syn.rtl.module_auto_prefix=1
  ```

- **syn.rtl.module_prefix:**

  Specifies a prefix to be used for all generated RTL module names. Use this to override the defaul module prefix of the top-level function.

  ```
  syn.rtl.module_prefix=newTop
  ```

- **syn.rtl.mult_keep_attribute:**

  Enables keep attribute.

  ```
  syn.rtl.mult_keep_attribute=1
  ```

- **syn.rtl.register_all_io:**

Uses a register by default for all I/O signals.

```
syn.rtl.register_all_io=1
```

- **syn.rtl.register_reset_num:**

  Number of registers to add to reset signal.

  ```
  syn.rtl.register_reset_num=2
  ```

- **syn.rtl.reset:** Variables initialized in the C/C++ code are always initialized to the same value in the RTL and therefore in the bitstream. This initialization is performed only at power-on. It is not repeated when a reset is applied to the design.

  Apply this setting with the `-reset` option to determine how registers and memories are reset.

  - `none`: Does not add resets to the design.
  - `control`: Resets control registers, such as those used in state machines and those used to generate I/O protocol signals. This is the default setting.
  - `state`: Resets control registers and registers or memories derived from static or global variables in the C/C++ code. Any static or global variable initialized in the C/C++ code resets to its initialized value.
  - `all`: Resets all registers and memories in the design. Any static or global variable initialized in the C/C++ code is reset to its initialized value.

  ```
  syn.rtl.reset=state
  ```

- **syn.rtl.reset_async:**

  Causes all registers to use an asynchronous reset. If you do not specify this option, a synchronous reset is used.

  ```
  syn.rtl.reset_async=1
  ```

- **syn.rtl.reset_level:**

  Defines the polarity of the reset signal to be either active-Low or active-High. The default setting is active-High.

  ```
  syn.rtl.reset_level=low
  ```

TIP: *The AXI protocol requires an active-Low reset. If your design uses AXI interfaces, the Vitis tool defines this reset level with a warning if the `syn.rtl.reset_level` is active-High.*

Schedule Setting

The `syn.schedule` command configures the default type of scheduling performed.

- **syn.schedule.enable_dsp_full_reg:**

  Specifies that the DSP signals need to be fully registered. This option is enabled by default.

  ```
  syn.schedule.enable_dsp_full_reg=0
  ```

Storage Configuration

Sets the global default options for the HLS micro-architecture binding of FIFO storage elements to memory resources.

You can override the default configuration defined by `syn.storage` for FIFO storage by syn.directive.bind_storage for a specific design element. You can also override the default configuration by specifying the `storage_type` option for syn.directive.interface for objects on the interface.

- **syn.storage:**

  See the following for the syntax of the `syn.storage` command.

  ```
  syn.storage=fifo impl=auto auto_srl_max_bits=512 auto_srl_max_depth=3
  ```

  - `syn.storage=fifo`: Starts the command to configure FIFOs.

    ***Note*:** FIFO is the only type supported at this time.

  - `impl=<value>`: Specifies the implementation (`impl`) keyword followed by the value. When `impl` is not specified, the default is `auto`. The default value allows the Vitis tool to determine the best implementation for a given operator. The option supports `auto`, `bram`, `lutram`, `uram`, `memory`, and `srl` values.
  - `auto_srl_max_bits=<value>`: Only valid when for `impl:auto` (the default). Specifies the maximum allowed SRL total bits (depth * width) for auto implementations. The default is 1024.
  - `auto_srl_max_depth=<value>`: Specifies the maximum allowed SRL depth for auto-srl implementation. The default is 2.

Implementation Configuration

The `syn.vivado` commands configure the Vivado synthesis and implementation runs used to derive resource utilization and timing estimates. These settings generally do not affect the RTL created for the HLS component, but can affect the example hardware design used for estimates.

- **vivado.clock:**

  Override the HLS clock constraint used in the Vivado out-of-context run for determination of timing analysis. This does not affect the output IP or XO files.

  ```
  vivado.clock=37
  ```

- **vivado.flow:**

  Run the Vivado flow for synthesis only (`syn`), or for implementation (`impl`). Running synthesis will be faster than running implementation, but will lack some of details of the implementation run. The default setting is `impl`.

  ```
  vivado.flow=syn
  ```

  > **TIP:** *This is different from the `flow_target` option that defines the target as being either the Vitis kernel flow or the Vivado IP flow.*

- **vivado.impl_strategy:**

  Specify a Vivado synthesis or implementation strategy. Strategy names can be found in Defining Implementation Strategies in the *Vivado Design Suite User Guide: Implementation* (UG904). This is used for resource usage and timing analysis and will not affect the output files.

  ```
  vivado.impl_strategy=Performance_Explore
  ```

- **vivado.max_timing_paths:**

  Specify the max number of timing paths to report when the timing is not met in the Vivado synthesis or implementation.

  ```
  vivado.max_timing_paths=12
  ```

- **vivado.optimization_level:**

  Vivado optimization level, sets other `vivado_*` options. This will not apply to IP/XO output. The choices are 0, 1, 2, 3. This only applies for report generation and will not apply to the exported IP. The default setting is 0.

  ```
  vivado.optimization_level=2
  ```

- **vivado.pblock:**

  Specify a Pblock range to use during implementation for reporting purposes.

  ```
  vivado.pblock={SLICE_X8Y105:SLICE_X23Y149}
  ```

- **vivado.phys_opt:**

  Run Vivado physical optimization at the specified implementation stage. The choices are no optimization (`none`), placement optimization (`place`), routing optimization (`route`), or both placement and routing (`all`). This option only applies to the implementation run used to generate resource and timing estimates.

  ```
  vivado.phys_opt=route
  ```

- **vivado.report_level:**

  Specify the report level for Vivado synthesis and implementation run. The valid values and the associated reports are:

  - 0: Post-synthesis utilization. Post-implementation utilization and timing.
  - 1: Post-synthesis utilization, timing, and analysis. Post-implementation utilization, timing, and analysis.
  - 2: Post-synthesis utilization, timing, analysis, and failfast. Post-implementation utilization, timing, and failfast. This is the default setting.

  ```
  vivado.report_level=1
  ```

- **vivado.rtl:**

  Specifies RTL language (`verilog`/`vhdl`) to select in Vivado flow.

  ```
  vivado.rtl=vhdl
  ```

- **vivado.synth_design_args:**

  Specifies arguments for the Vivado `synth_design` command. Available `synth_design` arguments can be found in the *Vivado Design Suite Tcl Command Reference Guide* (UG835).

  ```
  vivado.synth_design_args=arg1 arg2
  ```

- **vivado.synth_strategy:**

  Specifies a Vivado synthesis strategy to use when generating the example design to use for resource and timing estimates. Specific synthesis strategies can be found in *Vivado Design Suite User Guide: Synthesis* (UG901).

  ```
  vivado.synth_strategy=Synthesis_Defaults
  ```

Unroll Setting

The `syn.unroll` setting provides a global tripcount threshold below which loops are automatically unrolled. Loops with a greater tripcount can be unrolled using `syn.directive.unroll`.

- **syn.unroll.tripcount_threshold:**

  All loops which have fewer iterations than the specified value are automatically unrolled. The default value is 0, which means that loops are not automatically unrolled.

  ```
  syn.unroll.tripcount_threshold=6
  ```

v++ Mode VSS

Link

The v++ mode vss selects a link flow that creates a Vitis Subsystem (VSS) artifact. A VSS is a platform-independent, reusable design component that you can customize with AI Engine and PL content. Using `v++ --mode vss`, you package your design elements (for example, AI Engine graphs and PL kernels, and optionally PS domains) into a subsystem that is intended to be integrated into a larger system using Vitis tools. This flow does not produce a deployable hardware bitstream; instead, it generates a subsystem package (with metadata and binaries) that can be instantiated and linked within a system context on the target platform later.

`--mode` (`-m`):

```
 -m [ --mode ] arg Specify a mode.
                   [aie|hls] Must be used with --compile/-c option.
                   [vss] Must be used with --link/-l option.
```

See Linking a VSS Component in the *Embedded Design Development Using Vitis* (UG1701) for details.

HLS Optimization Directives

IMPORTANT! *The following options must appear in the HLS configuration file under the `[hls]` header.*

Directives, or the `syn.directive.xxx` commands allow you to customize the synthesis results for the same source code across multiple implementations. Change the directives to change the results. The `syn.directive.xxx` commands are for use in config files with the Vitis Unified IDE. See Building and Running an HLS Component in the *Vitis High-Level Synthesis User Guide* (UG1399).

***Note*:** You can also use pragmas in your source code, rather than directives in your config file. Doing so has the advantage of the pragmas being stored directly in your source code. Refer to the HLS Pragmas section of the *Vitis High-Level Synthesis User Guide* (UG1399) for more information.

Directives applied through a configuration file must include a `<location>` as an argument to the directive. The `<location>` defines what element of the source code the directive applies to, such as function, loop, region, or variable.

The example syntax for `syn.directive.xxx` commands include the location and the arguments for the directive. For example:

```
syn.directive.pipeline=dct2d II=4
```

Where `dct2d` is the location (function name) to apply the PIPELINE directive to, and II=4 is one of the possible arguments to the directive.

syn.directive.aggregate

Description

This directive collects the data fields of a struct into a single wide scalar. Vitis HLS performs a similar operation as `syn.directive.array_reshape` for arrays inside the struct. Vitis HLS completely partitions and reshapes the array into a wide scalar and packs it with other elements of the struct.

TIP: *Vitis HLS restructures arrays of structs as arrays of aggregated elements. The optimization does not support structs that contain other structs.*

You can infer the bit alignment of the resulting new wide-word from the declaration order of the struct elements. The first element takes the least significant sector of the word and so forth until all fields are mapped.

Syntax

```
syn.directive.aggregate=[OPTIONS] <location> <variable>
```

- `<location>` is the location (in the format `function[/label]`) that contains the packed variable.
- `<variable>` is the struct variable to be packed.

Options

- **`compact=[bit | byte | none | auto]`:** Specifies the alignment of the aggregated struct. Alignment can be on the bit-level (packed), the byte-level (padded), none, or automatically determined by the tool which is the default behavior.

Examples

The following example aggregates the variable `AB`. The variable is a struct pointer containing three 8-bit fields (`typedef struct {unsigned char R, G, B;}pixel`) in function `func`. The fields are abggregated into a new 24-bit pointer, aligning data at the bit-level.

```
syn.directive.aggregate=func AB compact=bit
```

See Also

- syn.directive.array_reshape
- syn.directive.disaggregate

syn.directive.alias

Description

Specifies that two or more `M_AXI` pointer arguments point to the same underlying buffer in memory (DDR or HBM). Indicates aliasing between the pointers by setting the distance or offset between them.

IMPORTANT! *`syn.directive.alias` applies to top-level function arguments mapped to `M_AXI` interfaces.*

Vitis HLS considers different pointers to be independent channels and generally does not provide any dependency analysis. However, in cases where the host allocates a single buffer for multiple pointers, this relationship can be communicated through `syn.directive.alias`. Communicating this way maintains dependency analysis. Communicating through `syn.directive.alias` also enables data dependence analysis in Vitis HLS by defining the distance between pointers in the buffer.

Requirements for `syn.directive.alias`:

- You must assign all ports to `M_AXI` interfaces and also assign them to different bundles. See example below for details.
- You can only use each port one time in a `syn.directive.alias` command.
- All ports assigned to `syn.directive.alias` must have the same depth.
- Each port can have one offset if you specify the offset.
- You must specify the offset for the `M_AXI` port as `slave` or `direct`. `offset=off` is not supported.

Syntax

```
syn.directive.alias=[OPTIONS] <location> <ports>
```

- `<location>` is the location string in the format `function[/label]` that the ALIAS pragma applies to.
- `<ports>` specifies the ports to alias.

Options

- **`distance=<integer>`:** Specifies the difference between the pointer values passed to the ports in the list.
- **`offset=<string>`:** Specifies the offset of the pointer passed to each port in the `ports` list with respect to the origin of the array.

Example

For the following function `top`:

```
void top(int *arr0, int *arr1, int *arr2, int *arr3, ...) {
  #pragma HLS interface M_AXI port=arr0 bundle=hbm0 depth=0x40000000
  #pragma HLS interface M_AXI port=arr1 bundle=hbm1 depth=0x40000000
  #pragma HLS interface M_AXI port=arr2 bundle=hbm2 depth=0x40000000
  #pragma HLS interface M_AXI port=arr3 bundle=hbm3 depth=0x40000000
```

The following command defines aliasing for the specified array pointers, and defines the `distance` between them:

```
syn.directive.alias=top arr0,arr1,arr2,arr3 distance=10000000
```

Alternatively, the following command specifies the `offset` between pointers, to accomplish the same effect:

```
syn.directive.alias=top arr0,arr1,arr2,arr3
offset=00000000,10000000,20000000,30000000
```

See Also

- syn.directive.interface

syn.directive.allocation

Description

Specifies instance restrictions for resource allocation.

`syn.directive.allocation` can limit the number of RTL instances and hardware resources used to implement specific functions, loops, or operations. If the C/C++ source has four instances of a function `foo_sub`, `syn.directive.allocation` ensures there is only one instance of `foo_sub` in the final RTL. All four instances are implemented using the same RTL block. This reduces resources used by the function, but negatively impacts performance by sharing those resources.

The `syn.directive.allocation` command also limits operations in the C/C++ code (additions, multiplications, array reads, and writes).

Syntax

```
syn.directive.allocation=[OPTIONS] <location> <instances>
```

- `<location>` is the location string in the format `function[/label]`.
- `<instances>` is a function or operator. The function can be any function in the original C/C++ code that has not been either inlined by the `syn.directive.allocation` command or inlined automatically by Vitis HLS.

Options

- `limit=<integer>`:

  Sets a maximum limit on the number of instances (of the type defined by the `type` option) to be used in the RTL design.

- `type=[function|operation]`: The instance type can be `function` (default) or `operation`. For a complete list of supported operations refer to Operator Configuration.

Examples

Given a design `foo_top` with multiple instances of function `foo`, limits the number of instances of `foo` in the RTL to 2.

```
syn.directive.allocation=limit=2 type=function foo_top foo
```

Limits the number of multipliers used in the implementation of `My_func` to 1. This limit does not apply to any multipliers that can reside in sub-functions of `My_func`. To limit the multipliers used in the implementation of any sub-functions, specify an allocation directive on the sub-functions or inline the sub-function into function `My_func`.

```
syn.directive.allocation=limit=1 type=operation My_func mul
```

See Also

- syn.directive.inline

syn.directive.array_partition

Description

> **IMPORTANT!** *`syn.directive.array_partition` and `syn.directive.array_reshape` are not supported for `M_AXI` Interfaces on the top-level function. Instead, you can use the `hls::vector` data types as described in the Vector Data Types section of the Vitis High-Level Synthesis User Guide (UG1399).*

`syn.directive.array_partition` partitions an array into smaller arrays or individual elements.

This partitioning can do the following.

- Results in RTL with multiple small memories or multiple registers instead of one large memory
- Increases the number of read and write ports for the storage
- Capacity to improve the throughput of the design
- Requires more memory instances or registers

Syntax

```
syn.directive.array_partition=[OPTIONS] <location> <array>
```

- `<location>` is the location (in the format `function[/label]`) that contains the array variable.
- `<array>` is the array variable to be partitioned.

Options

- **`dim=<integer>`**: Specifies the dimension of the array for partitioning.
  - The dimension is relevant for multi-dimensional arrays only.
  - All dimensions are partitioned with the specified options if using a value of 0.
  - Any other value partitions only that dimension. For example, for a value of 1, only the first dimension is partitioned.
- **`type=(block|cyclic|complete)`**:
  - `block` partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into *N* equal blocks where *N* is the integer defined by the `-factor` option.
  - `cyclic` partitioning creates smaller arrays by interleaving elements from the original array. For example, if `-factor 3` is used, the following is the result.
    - The directive assigns element 0 to the first new array.
    - Element 1 to the second new array.
    - Element 2 to the third new array.
    - Element 3 to the first new array again.
  - The `complete` partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers. For multi-dimensional arrays, specify the partitioning of each dimension, or use `-dim 0` to partition all dimensions.

  The default is `complete`.
- **`factor=<integer>`**: Used for `block` or `cyclic` partitioning only, this option specifies the number of smaller arrays that are to be created.
- **`off=true`**: Disables the ARRAY_PARTITION feature for the specified variable. Does not work with `dim`, `factor`, or `type`.

Example 1

Partitions array AB[13] in function `func` into four arrays. Because four is not an integer factor of 13:

- Three arrays have three elements.
- One array has four elements (AB[9:12]).

```
syn.directive.array_partition=func AB type=block factor=4
```

Partitions array AB[6][4] in function `func` into two arrays, each of dimension [6][2].

```
syn.directive.array_partition=func AB type=block factor=2 dim=2
```

Partitions all dimensions of AB[4][10][6] in function `func` into individual elements.

```
syn.directive.array_partition=func AB type=complete dim=0
```

Example 2

You can address partitioned arrays in your code by the new structure of the partitioned array. See the following code example when using the following directive.

```
syn.directive.array_partition=top b type=complete dim=1
```

The code can be structured as follows:

```
struct SS
{
  int x[N];
  int y[N];
};

int top(SS *a, int b[4][6], SS &c) {...}

syn.directive.interface mode=ap_memory top b[0]
syn.directive.interface mode=ap_memory top b[1]
syn.directive.interface mode=ap_memory top b[2]
syn.directive.interface mode=ap_memory top b[3]
```

See Also

- syn.directive.array_reshape

syn.directive.array_reshape

Description

> **IMPORTANT!** *`syn.directive.array_partition` and `syn.directive.array_reshape` are not supported for `M_AXI` Interfaces on the top-level function. Instead you can use the `hls::vector` data types as described in Vector Data Types in Vitis High-Level Synthesis User Guide (UG1399).*

`syn.directive.array_reshape` combines array partitioning with vertical array mapping to create a single new array with fewer elements but wider words.

The `syn.directive.array_reshape` command has the following features:

- Splits the array into multiple arrays (like `syn.directive.array_partition`).
- Automatically recombine the arrays vertically to create a new array with wider words.

Syntax

```
syn.directive.array_reshape=[OPTIONS] <location> <array>
```

- `<location>` is the location (in the format `function[/label]`) that contains the array variable.
- `<array>` is the array variable to be reshaped.

Options

- **`dim=<integer>`:** Specifies the dimension of the array for partitioning.
  - The dimension is relevant for multidimensional arrays only.
  - All dimensions are partitioned with the specified options if using a value of 0.
  - Any other value partitions only that dimension. For example, if using a value of 1, only the first dimension is partitioned.
- **`type=(block|cyclic|complete)`:**
  - `block` reshaping creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into *N* equal blocks where *N* is the integer defined by the `-factor` option and then combines the *N* blocks into a single array with `word-width*N`. The default is `complete`.
  - `cyclic` reshaping creates smaller arrays by interleaving elements from the original array. For example, if `-factor 3` is used, element 0 is assigned to the first new array, element 1 to the second new array, element 2 is assigned to the third new array, and then element 3 is assigned to the first new array again. The final array is a vertical concatenation (word concatenation, to create longer words) of the new arrays into a single array.

  - `complete` reshaping decomposes the array into temporary individual elements and then recombines them into an array with a wider word. For a one-dimension array this is equivalent to creating a very-wide register. For example, if the original array was *N* elements of *M* bits, the result is a register with `N*M` bits. This option is the default.

- **`factor=<integer>`:** Used for `block` or `cyclic` partitioning only, this option specifies the number of smaller arrays that are to be created.
- **`object`:**

  ***Note*:** Relevant for container arrays only.

  Applies reshape on the objects within the container. If the option is specified, all dimensions of the objects are reshaped. However, all dimensions of the container are maintained.

- **`off=true`:** Disables the ARRAY_RESHAPE feature for the specified variable.

Example 1

Reshapes 8-bit array AB[17] in function `func` into a new 32-bit array with five elements.

Because four is not an integer factor of 17:

- Index 17 of the array, AB[17], is in the lower eight bits of the reshaped fifth element.
- The upper eight bits of the fifth element are unused.

```
syn.directive.array_reshape=type=block factor=4 func AB
```

Partitions array AB[6][4] in function `func`, into a new array of dimension [6][2], in which dimension 2 is twice the width.

```
syn.directive.array_reshape=type=block factor=2 dim=2 func AB
```

Reshapes 8-bit array AB[4][2][2] in function `func` into a new single element array (a register), 4*2*2*8 (= 128)-bits wide.

```
syn.directive.array_reshape=type=complete dim=0 func AB
```

Example 2

You can address reshaped arrays in your code by the new structure of the array. See the code below when using the following directive.

```
syn.directive.array_reshape=top b type=complete dim=0
```

The code can be structured as follows:

```
struct SS
{
  int x[N];
  int y[N];
};

int top(SS *a, int b[4][6], SS &c) {...}
```

And the array interface defined as:

```
syn.directive.interface=mode=ap_memory top b[0]
```

See Also

- syn.directive.array_partition
- syn.directive.interface

syn.directive.bind_op

Description

> **TIP:** *This pragma or directive does not play a part in DSP multi-operation matching (MULADD, AMA, ADDMUL and corresponding Subtraction operations). In fact the use of BIND_OP for assigning an operator to DSP can prevent the HLS compiler from matching multi-operation expressions.*

Vitis HLS implements the different operations in the code using specific hardware resources (or implementations). The tool automatically determines the resource to use. You can also define `syn.op` to specify the implementation (see Operator Configuration).

The `syn.directive.bind_op` command indicates the variable an operation to map to a specific implementation in the RTL. For example `mul`, `add`, or `sub` can map to `impl` in the RTL. For example, to indicate that a specific multiplier operation (`mul`) is implemented in the device fabric rather than a DSP, you can use the `syn.directive.bind_op` command.

You can also specify the latency with `syn.directive.bind_op` using the `latency` option as described below.

> **IMPORTANT!** *To use the `latency` option, the operation must have an available multi-stage implementation. The HLS tool provides a multi-stage implementation for all basic arithmetic operations (add, subtract, multiply, and divide), and all floating-point operations.*

Syntax

```
syn.directive.bind_op=[OPTIONS] <location> <variable>
```

- `<location>` is the location (in the format `function[/label]`) which contains the variable.
- `<variable>` is the variable to be assigned. The variable in this case is one that is assigned the result of the operation that is the target of this directive.

Options

- **`op=<value>`:** Defines the operation to bind to a specific implementation resource. Supported functional operations include: `mul`, `add`, `sub`

  Supported floating point operations include: `fadd`, `fsub`, `fdiv`, `fexp`, `flog`, `fmul`, `frsqrt`, `frecip`, `fsqrt`, `dadd`, `dsub`, `ddiv`, `dexp`, `dlog`, `dmul`, `drsqrt`, `drecip`, `dsqrt`, `hadd`, `hsub`, `hdiv`, `hmul`, and `hsqrt`

  TIP: *Floating-point operations include single precision (f), double-precision (d), and half-precision (h).*

- **`impl=<value>`:** Defines the implementation to use for the specified operation.Supported implementations for functional operations include `fabric` and `dsp`.Supported implementations for floating point operations include: `fabric`, `meddsp`, `fulldsp`, `maxdsp`, and `primitivedsp`.

  ***Note*:** `primitivedsp` is only available on Versal devices.

- **`latency=<int>`:** Defines the default latency for the implementation of the operation. The valid latency varies according to the specified `op` and `impl`. The default is -1, which lets Vitis HLS choose the latency.The tables below reflect the supported combinations of operation, implementation, and latency.

Table 1: **Supported Combinations of Functional Operations, Implementation, and Latency**

Operation	Implementation	Min Latency	Max Latency
add	fabric	0	4
add	dsp	0	4
mul	fabric	0	4
mul	dsp	0	4
sub	fabric	0	4
sub	dsp	0	0

TIP: *Comparison operators, such as `dcmp`, are implemented in LUTs and cannot be implemented outside of the fabric, or mapped to DSPs, and so are not configurable with the `config_op` or `bind_op` commands.*

Table 2: **Supported Combinations of Floating Point Operations, Implementation, and Latency**

Operation	Implementation	Min Latency	Max Latency
fadd	fabric	0	13
fadd	fulldsp	0	12
fadd	primitivedsp	0	3
fsub	fabric	0	13
fsub	fulldsp	0	12
fsub	primitivedsp	0	3
fdiv	fabric	0	29
fexp	fabric	0	24
fexp	meddsp	0	21
fexp	fulldsp	0	30
flog	fabric	0	24
flog	meddsp	0	23
flog	fulldsp	0	29
fmul	fabric	0	9
fmul	meddsp	0	9
fmul	fulldsp	0	9
fmul	maxdsp	0	7
fmul	primitivedsp	0	4
fsqrt	fabric	0	29
frsqrt	fabric	0	38
frsqrt	fulldsp	0	33
frecip	fabric	0	37
frecip	fulldsp	0	30
dadd	fabric	0	13
dadd	fulldsp	0	15
dsub	fabric	0	13
dsub	fulldsp	0	15
ddiv	fabric	0	58
dexp	fabric	0	40
dexp	meddsp	0	45
dexp	fulldsp	0	57
dlog	fabric	0	38
dlog	meddsp	0	49
dlog	fulldsp	0	65
dmul	fabric	0	10
dmul	meddsp	0	13
dmul	fulldsp	0	13
dmul	maxdsp	0	14

Table 2: **Supported Combinations of Floating Point Operations, Implementation, and Latency** *(cont'd)*

Operation	Implementation	Min Latency	Max Latency
dsqrt	fabric	0	58
drsqrt	fulldsp	0	111
drecip	fulldsp	0	36
hadd	fabric	0	9
hadd	meddsp	0	12
hadd	fulldsp	0	12
hsub	fabric	0	9
hsub	meddsp	0	12
hsub	fulldsp	0	12
hdiv	fabric	0	16
hmul	fabric	0	7
hmul	fulldsp	0	7
hmul	maxdsp	0	9
hsqrt	fabric	0	16

Examples

In the following example, a two-stage pipelined multiplier using fabric logic is specified to implement the multiplication for variable `c` of the function `foo`.

```
int foo (int a, int b) {
int c, d;
c = a*b;
d = a*c;
return d;
}
```

And the `syn.directive.bind_op` command is as follows:

```
syn.directive.bind_op=op=mul impl=fabric latency=2 foo c
```

TIP: *The HLS tool selects the implementation to use for variable `d` because no `syn.directive.bind_op` is specified for that variable.*

See Also

- syn.directive.bind_storage

syn.directive.bind_storage

Description

Vitis HLS assigns arrays in the code using specific memory resources (or types). The tool automatically determines the resource to use. Alternatively, you can define `syn.storage` to generally specify the memory type, as explained in Storage Configuration.

The `syn.directive.bind_storage` command assigns a specific variable in the code (an array, or function argument) to a specific memory type (`type`) in the RTL. For example, you can use the `syn.directive.bind_storage` command to specify which type of memory, and which implementation to use for an array variable. You can control whether the array implements as a single or a dual-port RAM.

IMPORTANT! *This feature is important for arrays on the top-level function interface. The memory type associated with the array determines the number and type of ports needed in the RTL. See the Arrays on the Interface section of the Vitis High-Level Synthesis User Guide (UG1399). For variables assigned to top-level function arguments, you must assign the memory type and implementation using the `storage_type` and `storage_impl` options of `syn.directive.interface`.*

You can also use the `latency` option of `syn.directive.bind_storage` to specify the latency of the implementation. For block RAMs on the interface, the `latency` option allows you to model off-chip, non-standard SRAMs at the interface. An example is modeling support for an SRAM with a latency of 2 or 3. For internal operations, the `latency` option allows the operation implementation using more pipelined stages. These additional pipeline stages can help resolve timing issues during RTL synthesis.

IMPORTANT! *To use the `latency` option, the memory must have an available multi-stage implementation. The HLS tool provides a multi-stage implementation for all block RAMs.*

Syntax

```
syn.directive.bind_storage=[OPTIONS] <location> <variable>
```

- `<location>` is the location (in the format `function[/label]`) which contains the variable.
- `<variable>` is the variable to be assigned.

TIP: *If the variable is an argument of a top-level function, then use the `storage_type` and `storage_impl` options of `syn.directive.interface`.*

Options

- **`type=<value>`:** Defines the type of memory to bind to the specified variable. Supported types include: `fifo`, `ram_1p`, `ram_1wnr`, `ram_2p`, `ram_s2p`, `ram_t2p`, `rom_1p`, `rom_2p`, and `rom_np`.

Table 3: **Storage Types**

Type	Description
FIFO	A FIFO. Vitis HLS determines how to implement this in the RTL, unless the `-impl` option is specified.
RAM_1P	A single-port RAM. Vitis HLS determines how to implement this in the RTL, unless the `-impl` option is specified.
RAM_1WNR	A RAM with 1 write port and N read ports, using N banks internally.
RAM_2P	A dual-port RAM that allows read operations on one port and both read and write operations on the other port.
RAM_S2P	A dual-port RAM that allows read operations on one port and write operations on the other port.
RAM_T2P	A true dual-port RAM with support for both read and write on both ports.
ROM_1P	A single-port ROM. Vitis HLS determines how to implement this in the RTL, unless the `-impl` option is specified.
ROM_2P	A dual-port ROM.
ROM_NP	A multi-port ROM.

TIP: *If you specify a single port RAM for a PIPO, then the binder will typically allocate a merged-PIPO, with only one bank, and reader and writer accessing different halves of the bank. The info message reported by the HLS compiler says "Implementing PIPO using a single memory for all blocks." This is often cheaper for small PIPOs, where all banks can share a single RAM block. However, if you specify a dual-port RAM for a PIPO to get higher bandwidth and the scheduler uses both ports either in the producer or in the consumer, then the binder will typically allocate a split-PIPO, where the producer will use 1 port and the consumer 2 ports of different banks. The info message reported by the HLS compiler in this case says "Implementing PIPO using a separate memory for each block."*

- **`impl=<value>`:** Defines the implementation for the specified memory type. Supported implementations include: `bram`, `bram_ecc`, `lutram`, `uram`, `uram_ecc`, `srl`, `memory`, and `auto` as described below.

Table 4: **Supported Implementation**

Name	Description
MEMORY	Generic memory for FIFO, lets the Vivado tool choose the implementation.
URAM	UltraRAM resource
URAM_ECC	UltraRAM with ECC
SRL	Shift Register Logic resource
LUTRAM	Distributed RAM resource
BRAM	Block RAM resource
BRAM_ECC	Block RAM with ECC
AUTO	Vitis HLS automatically determine the implementation of the variable.

Table 5: **Supported Implementations by FIFO/RAM/ROM**

Type	Command/Pragma	Scope	Supported Implementations
FIFO	`bind_storage`[1]	local	**AUTO**, BRAM, LUTRAM, URAM, MEMORY, SRL
FIFO	`config_storage`	global	**AUTO**, BRAM, LUTRAM, URAM, MEMORY, SRL
RAM* | ROM*	`bind_storage`	local	**AUTO** BRAM, BRAM_ECC, LUTRAM, URAM, URAM_ECC
RAM* | ROM*	`config_storage`[2]	global	N/A
RAM_1P	`set_directive_interface s_axilite -storage_impl`	local	**AUTO**, BRAM, URAM
	`config_interface -m_axi_buffer_impl`	global	AUTO, **BRAM**, LUTRAM, URAM

Notes:

1. When there is no implementation, the directive uses AUTOSRL behavior as a default. However, this value cannot be specified.
2. `config_storage` only supports FIFO types.

- `latency=<int>`: Defines the default latency for the binding of the storage type to the implementation. The valid latency varies according to the specified `type` and `impl`. The default is -1, which lets Vitis HLS choose the latency.

Table 6: **Supported Combinations of Memory Type, Implementation, and Latency**

Type	Implementation	Min Latency	Max Latency
FIFO	BRAM	1	4
FIFO	LUTRAM	1	4
FIFO	MEMORY	1	4
FIFO	SRL	1	4
FIFO	URAM	1	4
RAM_1P	AUTO	1	3
RAM_1P	BRAM	1	3
RAM_1P	LUTRAM	1	3
RAM_1P	URAM	1	3
RAM_1WNR	AUTO	1	3
RAM_1WNR	BRAM	1	3
RAM_1WNR	LUTRAM	1	3
RAM_1WNR	URAM	1	3
RAM_2P	AUTO	1	3
RAM_2P	BRAM	1	3
RAM_2P	LUTRAM	1	3
RAM_2P	URAM	1	3
RAM_S2P	BRAM	1	3

Table 6: **Supported Combinations of Memory Type, Implementation, and Latency** *(cont'd)*

Type	Implementation	Min Latency	Max Latency
RAM_S2P	BRAM_ECC	1	3
RAM_S2P	LUTRAM	1	3
RAM_S2P	URAM	1	3
RAM_S2P	URAM_ECC	1	3
RAM_T2P	BRAM	1	3
RAM_T2P	URAM	1	3
ROM_1P	AUTO	1	3
ROM_1P	BRAM	1	3
ROM_1P	LUTRAM	1	3
ROM_2P	AUTO	1	3
ROM_2P	BRAM	1	3
ROM_2P	LUTRAM	1	3
ROM_NP	BRAM	1	3
ROM_NP	LUTRAM	1	3

IMPORTANT! *`syn.directive.bind_storage` only support combinations of memory types and implementations in the prior table.*

Examples

In the following example, the `coeffs[128]` variable is an argument of the function `func1`. The directive specifies that `coeffs` uses a single port RAM implemented on a BRAM core from the library.

```
syn.directive.bind_storage=func1 coeffs type=RAM_1P impl=bram
```

TIP: *The ports created in the RTL to access the values of `coeffs` are defined in the RAM_1P core.*

See Also

- syn.directive.bind_op

syn.directive.dataflow

Description

This directive performs all operations sequentially in a C/C++ description. In the absence of any directives that limit resources (such as `set_directive_allocation`), Vitis HLS seeks to minimize latency and improve concurrency. Data dependencies can limit this. For example, functions or loops that access arrays must finish all read/write accesses to the arrays before they complete. This prevents the next function or loop that consumes the data from starting operation.

However, it is possible for the operations in a function or loop to start operation before the previous function or loop completes all its operations. `syn.directive.dataflow` specifies that dataflow optimization be performed on the functions or loops, improving the concurrency of the RTL implementation. When `syn.directive.dataflow` is specified, the HLS tool analyzes the dataflow between sequential functions or loops and creates channels. The channels are based on ping-pong RAMs or FIFOs that allow consumer functions or loops to start operation before the producer functions or loops complete. Functions or loops can operate in parallel, decreasing latency and improving the throughput of the RTL.

TIP: *The `syn.dataflow.xxx` command specifies the default memory channel and FIFO depth used by `syn.directive.dataflow` as explained in Dataflow Configuration.*

If no initiation interval is specified, Vitis HLS attempts to minimize the initiation interval and start operation as soon as data is available. The initiation interval is number of cycles between the start of one function or loop and the next. For the DATAFLOW optimization to work, the data must flow through the design from one task to the next. The following coding styles prevent the HLS tool from performing the `DATAFLOW` optimization.

- Single-producer-consumer violations
- Feedback between tasks
- Conditional execution of tasks
- Loops with multiple exit conditions

IMPORTANT! *If any of these coding styles are present, the HLS tool issues a message and does not perform DATAFLOW optimization.*

Finally, the DATAFLOW optimization has no hierarchical implementation. If a sub-function or loop contains additional tasks that might benefit from the DATAFLOW optimization, apply optimization to the loop and the sub-function. You can also inline the sub-function.

Syntax

```
syn.directive.dataflow=<location> disable_start_propagation
```

- `<location>` is the location (in the format `function[/label]`) at which dataflow optimization is to be performed.
- `disable_start_propagation` disables the creation of a start FIFO used to propagate a start token to an internal process. Such FIFOs can sometimes be a bottleneck for performance.

Examples

Specifies DATAFLOW optimization within function `foo`.

```
syn.directive.dataflow=foo
```

See Also

- syn.directive.allocation

syn.directive.dependence

Description

Vitis HLS detects dependencies within loops. Dependencies within the same iteration of a loop are loop-independent dependencies. Dependencies between different iterations of a loop are loop-carried dependencies.

There is impact on the dependencies when you schedule operations (especially during function and loop pipelining).

- **Loop-independent dependence:** Accesses the same element in a single loop iteration.

```
for (i=1;i<N;i++) {
 A[i]=x;
 y=A[i];
}
```

- **Loop-carried dependence:** Accesses the same element from a different loop iteration.

```
for (i=1;i<N;i++) {
 A[i]=A[i-1]*2;
}
```

Under certain circumstances, the dependence analysis can be too conservative and fail to filter out false dependencies. An example of this circumstance is during variable dependent array indexing. Another example is when an external requirement needs to be enforced (for example, two inputs do not share the same index). The `syn.directive.dependence` command allows you to explicitly define the dependencies and eliminate a false dependence.

IMPORTANT! *Specifying a false dependency when the dependency is not false can result in incorrect hardware. Ensure dependencies are correct (true or false) before specifying them.*

Syntax

```
syn.directive.dependence=[OPTIONS] <location>
```

- **<location>:** The location in the code, specified as `function[/label]`, where the dependence is defined.

Options

- **class=(array | pointer):** Specifies a class of variables in which the dependence needs clarification. This is mutually exclusive with the `variable` option.
- **variable=<variable>:** Defines a specific variable to apply the dependence directive. Mutually exclusive with the `class` option.

  IMPORTANT! *You cannot specify a `dependence` for function arguments that are bundled with other arguments in an `m_axi` interface. This is the default configuration for `m_axi` interfaces on the function. You also cannot specify a dependence for an element of a struct, unless the struct has been disaggregated.*

- **dependent=(true | false):** This argument should be specified to indicate whether a dependence is `true` and needs to be enforced, or is `false` and should be removed. However, when not specified, the tool will return a warning that the value was not specified and will assume a value of `false`.
- **direction=(RAW | WAR | WAW):**

  ***Note*:** Relevant only for loop-carried dependencies.

  Specifies the direction for a dependence:

  - **RAW (Read-After-Write - true dependence):** The write instruction uses a value used by the read instruction.
  - **WAR (Write-After-Read - anti dependence):** The read instruction gets a value that is overwritten by the write instruction.
  - **WAW (Write-After-Write - output dependence):** Two write instructions write to the same location, in a certain order.

- **distance=<integer>:**

  ***Note*:** Relevant only for loop-carried dependencies where `-dependent` is set to `true`.

  Specifies the inter-iteration distance for array access.

- **type=(intra | inter):** Specifies whether the dependence is:
  - Within the same loop iteration (`intra`), or
  - Between different loop iterations (`inter`) (default).

Examples

Removes the dependence between `Var1` in the same iterations of `loop_1` in function `func`.

```
syn.directive.dependence=variable=Var1 type=intra \
dependent=false func/loop_1
```

The dependence on all arrays in `loop_2` of function `func` informs Vitis HLS that all reads must happen *after* writes in the same loop iteration.

```
syn.directive.dependence=class=array type=intra \
dependent=true direction=RAW func/loop_2
```

See Also

- syn.directive.disaggregate
- syn.directive.pipeline

syn.directive.disaggregate

Description

The `syn.directive.disaggregate` command lets you deconstruct a `struct` variable into its individual elements. The contents of the struct itself determine the number and type of elements.

IMPORTANT! *Structs used as arguments to the top-level function are aggregated by default, but can be disaggregated with this directive or pragma.*

Syntax

```
syn.directive.disaggregate=<location> <variable>
```

- `<location>` is the location (in the format `function[/label]`) where the variable to disaggregate is found.
- `<variable>` specifies the struct variable name.

Options

This command has no options.

Example 1

The following example shows disaggregates the struct variable `a` in function `top`:

```
syn.directive.disaggregate=top a
```

Example 2

You can address disaggregated structs by the using standard C/C++ coding style. The following shows the code when using the directives.

```
syn.directive.disaggregate=top a
syn.directive.disaggregate=top c
```

The code can be structured as follows.

```
struct SS
{
  int x[N];
  int y[N];
};

int top(SS *a, int b[4][6], SS &c) {

syn.directive.interface=mode=s_axilite top a->x
syn.directive.interface=mode=s_axilite top a->y

syn.directive.interface=mode=ap_memory top c.x
syn.directive.interface=mode=ap_memory top c.y
```

IMPORTANT! *Notice the different methods shown above for accessing an element of a pointer (a) versus an element of a reference (c)*

Example 3

The following example shows the `Dot` struct containing the `RGB` struct as an element. If you apply `syn.directive.disaggregate` to variable `Arr`, then only the top-level `Dot` struct is disaggregated.

```
struct Pixel {
char R;
char G;
char B;
};

struct Dot {
Pixel RGB;
unsigned Size;
};

#define N 1086
void DUT(Dot Arr[N]) {
...
}

syn.directive.disaggregate=DUT Arr
```

If you want to disaggregate the whole struct, `Dot` and `RGB`, then you can assign the `set_directive_disaggregate` as shown below.

```
void DUT(Dot Arr[N]) {
#pragma HLS disaggregate variable=Arr->RGB
...
}

syn.directive.disaggregate=DUT Arr->RGB
```

The following code illustrates the results.

```
void DUT(char Arr_RGB_R[N], char Arr_RGB_G[N], char Arr_RGB_B[N], unsigned
Arr_Size[N]) {
...
}
```

See Also

- syn.directive.aggregate

syn.directive.expression_balance

Description

Sometimes C/C++ code has a sequence of operations, resulting in a long chain of operations in RTL. With a small clock period, this can increase the latency in the design. By default, the Vitis HLS tool rearranges the operations using associative and commutative properties. The rearrangement creates a balanced tree that can shorten the chain, potentially reducing latency in the design at the cost of extra hardware.

Expression balancing rearranges operators to construct a balanced tree and reduce latency.

- For integer operations expression balancing is on by default but can be disabled.
- For floating-point operations, expression balancing is off by default but can be enabled.

The `syn.directive.expression_balance` command allows this expression balancing to be turned off, or on, within a specified scope.

Syntax

```
syn.directive.expression_balance=[OPTIONS] <location>
```

- `<location>` is the location (in the format `function[/label]`). It specifies where expression balancing should be disabled or enabled.

Options

- `off`: Turns off expression balancing at the specified location. Specifying the `syn.directive.expression_balance` command enables expression balancing in the specified scope. Adding the `off` option disables it.

Examples

Disables expression balancing within function `My_Func`.

```
syn.directive.expression_balance=off My_Func
```

Explicitly enables expression balancing in function `My_Func2`.

```
set_directive_expression_balance=My_Func2
```

syn.directive.function_instantiate

Description

By default:

- Functions remain as separate hierarchy blocks in the RTL, or are decomposed (inlined) into higher-level functions.
- All instances of a function, at the same level of hierarchy, uses the same RTL implementation (block).

The `syn.directive.function_instantiate` command creates a unique RTL implementation for each instance of a function. Each instance can be optimized around a specific argument or variable.

By default, the following code results in a single RTL implementation of function `func_sub` for all three instances, or if `func_sub` is a small function it is inlined into function `func`.

TIP: *By default, the Vitis HLS tool automatically inlines small functions. This is true even for function instantiations. Using the `set_directive_inline off` option can prevent this automatic inlining.*

```
char func_sub(char inval, char incr)
{
  return inval + incr;
}
void func(char inval1, char inval2, char inval3,
  char *outval1, char *outval2, char * outval3)
{
  *outval1 = func_sub(inval1, 1);
  *outval2 = func_sub(inval2, 2);
  *outval3 = func_sub(inval3, 3);
}
```

Using the directive as shown in the example section below results in three versions of function `func_sub`, each independently optimized for variable `incr`.

Syntax

```
syn.directive.function_instantiate=<location> <variable>
```

- `<location>` is the location (in the format `function[/region label]`) where the instances of a function are to be made unique.
- `<variable>` specifies the function argument as a constant in the various function instantiations.

Options

This command has no options.

Examples

For the example code shown above, the following Tcl (or pragma placed in function `func_sub`) allows each instance of function `func_sub` to be independently optimized with respect to input `incr`.

```
syn.directive.inline off=func_sub
syn.directive.function_instantiate=func_sub incr
```

See Also

- syn.directive.allocation
- syn.directive.inline

syn.directive.inline

Description

`syn.directive.inline` removes a function as a separate entity in the RTL hierarchy. After inlining, the function dissolves into the calling function and no longer appears as a separate level of hierarchy.

IMPORTANT! *Inlining a child function also dissolves any pragmas or directives applied to that function. In Vitis HLS, any directives applied in the child context are ignored.*

In some cases, inlining a function allows operations within the function to be shared and optimized more effectively with the calling function. However, an inlined function cannot be shared or reused, so if the parent function calls the inlined function multiple times, this can increase the area and resource utilization.

By default, inlining is only performed on the next level of function hierarchy.

Syntax

```
syn.directive.inline=[OPTIONS] <location>
```

- `<location>` is the location (in the format `function[/label]`) where inlining is to be performed.

Options

- **`off`:** By default, Vitis HLS performs inlining of smaller functions in the code. Using the `off` option disables inlining for the specified function.
- **`recursive`:** By default, only one level of function inlining is performed. The functions within the specified function are not inlined. The `recursive` option inlines all functions recursively within the specified function hierarchy.

Examples

The following example inlines function `func_sub1`, but no sub-functions called by `func_sub1`.

```
syn.directive.inline=func_sub1
```

This example inlines function `func_sub1`, recursively down the hierarchy, excluding function `func_sub2`:

```
syn.directive.inline=recursive func_sub1
syn.directive.inline=off func_sub2
```

See Also

- syn.directive.allocation

syn.directive.interface

Description

Only the top-level function supports the `syn.directive.interface` directive. You cannot use this directive for sub-functions of the HLS component. The INTERFACE pragma or directive specifies how function arguments creates RTL ports during interface synthesis. See the Defining Interfaces section of the *Vitis High-Level Synthesis User Guide* (UG1399). The Vitis HLS tool automatically determines the I/O protocol used by any sub-functions.

Ports in the RTL implementation come from the following elements.

- The data type and direction of the arguments of the top-level function and function return.
- The `flow_target` for the HLS component.
- The default interface configuration settings as specified by `syn.interface.xxx` commands (see Interface Configuration).

- `syn.directive.interface`.

Each function argument can have its own I/O protocol (such as valid handshake or acknowledge handshake).

> **TIP:** *Global variables required on the interface must be explicitly defined as an argument of the top-level function as described in the Global Variables section of the Vitis High-Level Synthesis User Guide (UG1399). However, if a global variable is accessed, but all read and write operations are local to the design, the resource is created in the design. There is no need for an I/O port in the RTL.*

The interface also defines the execution control protocol of the HLS component. See the Block-Level Control Protocols section of the *Vitis High-Level Synthesis User Guide* (UG1399) for details. The control protocol controls when the HLS component (or block) starts execution, and when the block completes operation, is idle and ready for new inputs.

Syntax

```
syn.directive.interface=[OPTIONS] <location> <port>
```

- `<location>` is the location (in the format `function[/label]`) where the function interface or registered output can be specified.
- `<port>` is the parameter (function argument) to synthesize the interface. You do not need the port name with these block control modes: `ap_ctrl_chain`, `ap_ctrl_hs`, or `ap_ctrl_none`.

Options

> **TIP:** *Many of the options below have default values defined with `syn.interface.xxx` commands. You can define local values for the interface defined here to override the default values.*

- **`mode=<mode>`:**

  The following three categories illustrate the modes and how the tool implements them in RTL.

  1. Port-Level Protocols:
     - **`ap_none`:** No port protocol. The interface is a simple data port.
     - **`ap_stable`:** No protocol. The interface is a simple data port. The Vitis HLS tool assumes the data port is always stable after reset. Allows optimizations to remove unnecessary registers.
     - **`ap_vld`:** Implements the data port with an associated `valid` signal to indicate when the data is valid for reading or writing.
     - **`ap_ack`:** Implements the data port with an associated `acknowledge` signal to acknowledge writing or reading of data.

- **ap_hs:** Implements the data port with both `valid` and `acknowledge` signals to provide a two-way handshake to indicate when the data is valid for reading and writing. Acknowledges that the data was read or written
- **ap_ovld:** Implements the output data port with an associated `valid` signal to indicate when the data is valid for reading or writing.

  > **TIP:** *For `ap_ovld` Vitis HLS implements the input argument or the input half of any read/write arguments with mode `ap_none`.*

- **ap_memory:** Implements array arguments as a standard RAM interface. If you use the RTL design in Vivado IP integrator, separate ports make up the interface.
- **ap_fifo:** Implements the port with a standard FIFO interface using data input and output ports with associated active-Low FIFO `empty` and `full` ports.

  Note: You can only use the `ap_fifo` interface on read arguments or write arguments. `ap_fifo` mode does not support bidirectional read/write arguments.

2. AXI Interface Protocols:
   - **s_axilite:** Implements the port as an AXI4-Lite interface. The tool produces an associated set of C driver files when exporting the generated RT for the HLS component.
   - **m_axi:** Implements the port as an AXI4 interface. You can use the `syn.interface.m_axi_addr64` command to specify either 32-bit (default) or 64-bit address ports and to control any address offset.
   - **axis:** Implements the port as an AXI4-Stream interface.
3. Block-Level Control Protocols:
   - **ap_ctrl_chain:** Implements a set of block-level control ports to perform the following actions.
     - Star the design operation with `start`
     - Indicate when the design is `idle`
     - Indicate when the design is `done`
     - New input data with `ready`
   - **ap_ctrl_hs:** Implements a set of block-level control ports to `start` the design operation. Also indicates when the design is `idle`, `done`, and `ready` for new input data.
   - **ap_ctrl_none:** No block-level I/O protocol.

     Note: Using the `ap_ctrl_none` mode can prevent the design from being verified using C/RTL co-simulation.

- **bundle=<string>:**

By default, the HLS tool groups (or bundles) function arguments that are compatible into a single interface port in the RTL code. All interface ports with compatible options, such as `mode`, `offset`, and `bundle`, are grouped into a single interface port.

> **TIP:** *This default can be changed using the `syn.interface.m_axi_auto_max_ports` command.*

The `bundle=<string>` option lets you define bundles to group ports together, overriding the default behavior. The `<string>` specifies the bundle name. A combination of the `mode` and `bundle` provides the port name in the generated RTL. The `name` can also specify the port name.

> **IMPORTANT!** *You must specify `bundle` names using lower-case characters.*

- **`clock=<string>`:** By default, the AXI4-Lite interface clock is the same clock as the system clock. This option specifies a separate clock for an AXI4-Lite interface. If the `bundle` option groups multiple top-level function arguments into a single AXI4-Lite interface, you only need to specify the clock option on one of bundle members.
- **`channel=<string>`:** To enable multiple channels on an `m_axi` interface specify the channel ID. Multiple `m_axi` interfaces can be combined into a single `m_axi` adapter using separate channel IDs.
- **`depth=<int>`:** Specifies the maximum number of samples for the test bench to process. This setting indicates the maximum size of the FIFO needed in the verification adapter that the HLS tool creates for RTL co-simulation.

  > **TIP:** *While `depth` is usually an option, it is needed to specify the size of pointer arguments for RTL co-simulation.*

- **`interrupt=<int>`:** Only `ap_vld`/`ap_hs` uses this option. This option enables managing of I/O in interrupt. This option creates the corresponding bits in the `ISR` and `IER` in the `s_axilite` register file. The integer value N=16..31 specifies the bit position in both registers. By default, the bit position is assigned contiguously from 16.
- **`latency=<value>`:** This option can be used on `ap_memory` and `m_axi` interfaces.
  - In an `ap_memory` interface, the option specifies the read latency of the RAM resource driving the interface. The default is a read operation of 1 clock cycle. This option allows modeling of external RAM with more than 1 clock cycle of read latency.
  - In an `m_axi` interface the option specifies the expected latency of the AXI4 interface, allowing the design to initiate a bus request the specified number of cycles (latency) before the read or write is expected. If this figure it too low, the design can be ready too soon and can stall waiting for the bus. If this figure is too high, bus access can be idle waiting on the design to start the access.

- **max_read_burst_length=<int>:** For use with the m_axi interface, this option specifies the maximum number of data values read during a burst transfer. Refer to the AXI Burst Transfers section of the *Vitis High-Level Synthesis User Guide* (UG1399) for more information.
- **max_write_burst_length=<int>:** For use with the m_axi interface, this option specifies the maximum number of data values written during a burst transfer.
- **max_widen_bitwidth=<int>:** Specifies the maximum bit width available for the interface when automatically widening the interface. This overrides the default value specified by the syn.interface.m_axi_max_bitwidth command. Refer to Automatic Port Width Resizing in the *Vitis High-Level Synthesis User Guide* (UG1399) for more information.
- **name=<string>:** Specifies a name for the port for the generated RTL to use. A combination of the mode and bundle specifies the port name. The exception is if name specifies the name.
- **num_read_outstanding=<int>:** For use with the m_axi interface, this option specifies how many read requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design, and a FIFO of size:

  ```
  num_read_outstanding*max_read_burst_length*word_size
  ```

- **num_write_outstanding=<int>:** For use with the m_axi interface. This option specifies how many write requests the AXI4 bus can take, without a response, before the design stalls. This implies internal storage in the design, and a FIFO of size:

  ```
  num_write_outstanding*max_write_burst_length*word_size
  ```

- **offset=<string>:** Controls the address offset in AXI4-Lite (s_axilite) and AXI4 memory mapped (m_axi) interfaces for the specified port.
  - In an s_axilite interface, <string> specifies the address in the register map.
  - In an m_axi interface, this option overrides the global option specified by the config_interface -m_axi_offset option and <string>. See the following for details.
    - off: Does not generate an offset port.
    - direct: Generates a scalar input offset port.
    - slave: Generates an offset port and automatically map it to an AXI4-Lite slave interface. This is the default offset.
- **register:** Registers the signal and any associated protocol signals. Instructs the signals to persist until at least the last cycle of the function execution. The syn.interface.register_io command controls default registering of all interfaces on the top function, while this option lets you override the default for the current interface. This option applies to the following interface modes:
  - s_axilite

- `ap_fifo`
- `ap_none`
- `ap_stable`
- `ap_hs`
- `ap_ack`
- `ap_vld`
- `ap_ovld`

> **TIP:** *The `register` option cannot be used on the return port of the function (`port=return`). Use the `syn.directive.latency` instead.*

- **`register_mode=(both|forward|reverse|off)`:** This option applies to AXI4-Stream interfaces, and specifies if registers are placed on the following places.
  - Forward path (`TDATA` and `TVALID`), or
  - The reverse path (`TREADY`) on both paths, or
  - None of the ports signals (`off`).

  The default is `register_mode=both`.

  > **TIP:** *AXI4-Stream side-channel signals are data signals. The signals register whenever the `TDATA` is register.*

- **`storage_impl=<impl>`:** For use with `mode=s_axilite` only, this options defines a storage implementation to assign to the interface. Supported `<impl>` values include `auto`, `bram`, and `uram`. The default is `auto`.

  > **TIP:** *`uram` is a synchronous memory with a single clock for two ports available only on certain devices. Therefore `uram` cannot be specified for an `s_axilite` adapter with two clocks, or when the specified part does not support `uram`.*

- **`storage_type=<type>`:**

  For use with `mode=ap_memory` or `mode=bram` only, this options defines a storage type (for example, RAM_T2P) to assign to the variable.

  Supported types include: `ram_1p`, `ram_1wnr`, `ram_2p`, `ram_s2p`, `ram_t2p`, `rom_1p`, `rom_2p`, and `rom_np`.

  > **TIP:** *For objects on the interface of the top-level function, you can define this option using `syn.directive.bind_storage`.*

Example 1

This example disables function-level handshakes for function `func`.

```
syn.directive.interface=mode=ap_ctrl_none func return
```

Example 2

Argument `InData` in function `func` has a `ap_vld` interface. The input need to be registered.

```
set_directive_interface=func InData mode=ap_vld register
```

See Also

- syn.directive.bind_storage
- syn.directive.latency

syn.directive.latency

Description

Specifies a maximum or minimum latency value, or both, on a function, loop, or region.

Vitis HLS always aims for minimum latency. The following illustrates the behavior of the tool after specifying minimum and maximum latency values.

- Latency is less than the minimum: If Vitis HLS can achieve less than the minimum specified latency, it extends the latency to the specified value. This action can enable increased sharing.
- Latency is greater than the minimum: The constraint is satisfied. The tool performs no further optimizations.
- Latency is less than the maximum: The constraint is satisfied. The tool performs no further optimizations.
- Latency is greater than the maximum: If Vitis HLS cannot schedule within the maximum limit, it increases effort to achieve the specified constraint. If it still fails to meet the maximum latency, it issues a warning. Vitis HLS then produces a design with the smallest achievable latency.

TIP: *You can also use `syn.directive.latency` to limit the efforts of the tool to find an optimum solution. Specify latency constraints for scopes within the code. Loops, functions, or regions, reduces the possible solutions within that scope and can improve tool compilation time. Refer to the Improving Synthesis Runtime and Capacity section of the Vitis High-Level Synthesis User Guide (UG1399) for more information.*

To limit the total latency of all loop iterations, apply the latency directive to a region that encompasses the entire loop. For example: `syn.directive.latency Region_All_Loop_A max=10`

```
Region_All_Loop_A: {
Loop_A: for (i=0; i<N; i++)
  {
  ..Loop Body...
  }
}
```

In this case, even if the loop is unrolled, the latency directive sets a maximum limit on all loop operations.

Syntax

```
syn.directive.latency=[OPTIONS] <location>
```

- `<location>` is the location (function, loop or region) (in the format `function[/label]`) to be constrained.

Options

- **`max=<integer>`:** Limits the maximum latency.
- **`min=<integer>`:** Limits the minimum latency.

Examples

Function `foo` has a minimum latency of 4 and a maximum latency of 8.

```
syn.directive.latency=min=4 max=8 foo
```

In function `foo`, loop `loop_row` is specified to have a maximum latency of 12.

```
syn.directive.latency=max=12 foo/loop_row
```

syn.directive.loop_flatten

Description

Flattens nested loops into a single loop hierarchy.

In the RTL implementation, it costs a clock cycle to move between loops in the loop hierarchy. Flattening nested loops allows optimization of loops to a single loop. This saves clock cycles, potentially allowing for greater optimization of the loop body logic.

RECOMMENDED: *Apply this directive to the inner-most loop in the loop hierarchy. You can only flatten perfect and semi-perfect loops in this manner.*

- **Perfect loop nests:**
  - Only the innermost loop has loop body content.
  - There is no logic specified between the loop statements.
  - All loop bounds are constant.
- **Semi-perfect loop nests:**
  - Only the innermost loop has loop body content.
  - There is no logic specified between the loop statements.
  - The outermost loop bound can be a variable.
- **Imperfect loop nests:**

  When the inner loop has variables bounds, or if the loop body is not exclusively inside the inner loop, do the following. Try to restructure the code, or unroll the loops in the loop body to create a perfect loop nest.

Syntax

```
syn.directive.loop_flatten=[OPTIONS] <location>
```

- `<location>` is the location (inner-most loop), in the format `function[/label]`.

Options

- **`off`:** Prevents loop flattening from taking place. Can prevent some loops from being flattened while all others in the specified location are flattened.

  IMPORTANT! *The presence of the LOOP_FLATTEN pragma or directive enables the optimization. The addition of `off` disables it.*

Examples

Flattens `loop_1` in function `foo` and all (perfect or semi-perfect) loops above it in the loop hierarchy, into a single loop.

```
set_directive_loop_flatten=foo/loop_1
```

Prevents loop flattening in `loop_2` of function `foo`.

```
set_directive_loop_flatten=off foo/loop_2
```

See Also

- syn.directive.loop_flatten

syn.directive.loop_merge

Description

Merges all loops into a single loop. Merging loops:

- Reduces the number of clock cycles required in the RTL to transition between the loop-body implementations.
- Allows implementation of loops in parallel (if possible).

The rules for loop merging are:

- If the loop bounds are variables, they must have the same value (number of iterations).
- If loops bounds are constants, the tool uses the maximum constant value as the bound of the loop being merged.
- Loops with both variable bounds and constant bounds cannot be merged.
- The code between merging loops cannot have side effects. Multiple executions of this code need to generate the same results.
  - `a=b` is allowed
  - `a=a+1` is not allowed.
- Loops cannot be merged when they contain FIFO reads. Merging changes the order of the reads. Reads from a FIFO or FIFO interface must always be in sequence.

Syntax

```
syn.directive.loop_merge=[options] <location>
```

- `<location>` is the location (in the format `function[/label]`) at which the loops reside.

Options

- **`force`:** Forces loops to be merged even when Vitis HLS issues a warning. Ensures that the merged loop functions correctly.

Examples

Merges all consecutive loops in function `foo` into a single loop.

```
syn.directive.loop_merge=foo
```

All loops inside `loop_2` of function `foo` (but not `loop_2` itself) are merged by using the `force` option.

```
syn.directive.loop_merge=force foo/loop_2
```

See Also

- syn.directive.loop_flatten
- syn.directive.unroll

syn.directive.loop_tripcount

Description

The *loop tripcount* is the total number of iterations performed by a loop. Vitis HLS reports the total latency of each loop (the number of cycles to execute all iterations of the loop). This loop latency is therefore a function of the tripcount (number of loop iterations).

IMPORTANT! *`syn.directive.loop_tripcount` is for analysis only, and does not impact the results of synthesis.*

The tripcount can be a constant value. It can depend on the value of variables used in the loop expression (for example, `x<y`) or control statements used inside the loop.

Vitis HLS cannot determine the tripcount in some cases. These cases include, for example, those in which the variables used to determine the tripcount are:

- Input arguments, or
- Variables calculated by dynamic operation

In the following example, the value of input `num_samples` determines the maximum iteration of the for-loop. The value of `num_samples` is not defined in the C function, but comes into the function from the outside.

```
void foo (num_samples, ...) {
   int i;
   ...
   loop_1: for(i=0;i< num_samples;i++) {
     ...
     result = a + b;
   }
}
```

`syn.directive.loop_tripcount` allows you to specify minimum, maximum, and average iterations for a loop if the loop latency is unknown. The tool can analyze how the loop latency contributes to the total design latency in reports, helping you determine appropriate design optimizations.

TIP: *If you use a C assert macro to limit the size of a loop variable, Vitis HLS performs the following. Vitis HLS uses the macro to both define loop limits for reporting and create hardware that is exactly sized to these limits.*

Syntax

```
syn.directive.loop_tripcount=[OPTIONS] <location>
```

- `<location>` is the location of the loop (in the format `function[/label]`) at which the tripcount is specified.

Options

- **`avg=<integer>`**: Specifies the average number of iterations.
- **`max=<integer>`**: Limits the maximum number of iterations.
- **`min=<integer>`**: Limits the minimum number of iterations.

Examples

`loop_1` in function `foo` has a minimum tripcount of 12, and a maximum tripcount of 16:

```
syn.directive.loop_tripcount=min=12 max=16 avg=14 foo/loop_1
```

syn.directive.occurrence

Description

The OCCURRENCE pragma or directive specifies the following when pipeling functions and loops. The code in a pipelined function call within the pipelined function or loop executes at a lower rate than the surrounding function or loop. Executing at a lower rate allows the pipelined call to pipeline at a lower rate and potentially shared within the top-level pipeline. For example:

- A loop iterates *N* times.
- Part of the loop is protected by a conditional statement and only executes *M* times, where *N* is an integer multiple of *M*.
- The code protected by the conditional is said to have an occurrence of *N*/*M*.

Identifying a region with an OCCURRENCE rate allows the functions and loops in this region to be pipelined with an initiation interval that is slower than the enclosing function or loop.

Syntax

```
syn.directive.occurrence=[OPTIONS] <location>
```

- `<location>` specifies the block of code that contains the pipelined function call(s) with a slower rate of execution.

Options

- **`cycle=<int>`**: Specifies the occurrence *N*/*M* where:

- *N* is the number of times the enclosing function or loop is executed.
- *M* is the number of times the conditional region is executed.

> **IMPORTANT!** *N must be an integer multiple of M.*

- `off=true`: Disable occurrence for the specified function.

Examples

Region `Cond_Region` in function `foo` has an occurrence of 4. It executes at a rate four times slower than the code that encompasses it.

```
syn.directive.occurrence=cycle=4 foo/Cond_Region
```

See Also

- syn.directive.pipeline

syn.directive.performance

Description

> **TIP:** *`syn.directive.performance` applies to loops and loop nests, and requires a known loop tripcount to determine the performance. If your loop has a variable tripcount then you must also specify `syn.directive.tripcount`.*

The `syn.directive.performance` lets you specify a high-level constraint, `target_ti` or `target_tl`. The directive allows you to define the number of clock cycles between successive starts of a loop. The directive lets the tool infer lower-level UNROLL, PIPELINE, ARRAY_PARTITION, and INLINE directives for achieving the desired result. The `syn.directive.performance` does not guarantee the specified value will be achieved, and so it is only a target.

The `target_ti` is the interval between successive starts of the loop. It is also the interval between the start of the first iteration of the loop, and the next start of the first iteration of the loop. In the following code example, a `target_ti=T` means the target interval for the start of loop `L2` between two consecutive iterations of `L1` is 100 cycles.

```
const int T = 100;
L1: for (int i=0; i<N; i++)
  L2: for (int j=0; j<M; j++){
   #pragma HLS PERFORMANCE target_ti=T
   ...
   }
```

The `target_tl` is the interval between start of the loop and end of the loop, or between the start of the first iteration of the loop and the completion of the last iteration of the loop. In the preceding code example a `target_tl=T` means the target completion of loop `L2` for a single iteration of `L1` should be 100 cycles.

***Note*:** `syn.directive.inline` applies automatically to functions inside any pipelined loop with II=1 to improve throughput. If you apply the PERFORMANCE pragma, the tool also triggers auto-inline optimization. If you apply a directive that infers a pipeline with II=1, the tool also triggers auto-inline optimization. You can disable this for specific functions by using `syn.directive.inline off`.

The transaction interval is the initiation interval (II) of the loop times the number of iterations, or tripcount: target_ti = II * loop tripcount. Conversely, `target_ti` = FreqHz / Operations per second.

For example, assume an image processing function processes a single frame per invocation with a throughput goal of 60 fps. The target throughput for the function is 60 invocations per second. If the clock frequency is 180 MHz, then `target_ti` is 180M/60, or 3 million clock cycles per function invocation.

Syntax

Specify the directive for a labeled loop.

```
syn.directive.performance=<location> [Options]
```

Where:

`<location>` specifies the loop in the format `function/loop_label`.

Options

- **`target_ti=<value>`:** Specifies a target transaction interval that is the number of clock cycles for the loop to complete an iteration. The transaction interval refers to the number of clock cycles from the first transaction of a loop, or nested loop to the start of the next transaction of the loop. The <value> can be an integer, floating point, or constant expression that is resolved by the tool as an integer.

  ***Note*:** The tool returns a warning if truncation occurs.

- **`target_tl=<value>`:**

  Specifies a target latency that is the number of clock cycles for the loop to complete all iterations. The transaction latency is the interval between the start of the first iteration of the loop, and the completion of the last iteration of the loop. The <value> can be an integer, floating point, or constant expression that is resolved by the tool as an integer.

- **unit=[sec | cycle]:** Specifies the unit associated with the `target_ti` or `target_tl` values. The unit can either be seconds, or clock cycles. When the unit is specified as seconds, you can specify it with a value to indicate nanoseconds (ns), picoseconds (ps) or microseconds (us).

Example 1

The loop labeled `loop_1` has a target transaction interval of 4 clock cycles:

```
syn.directive.performance=loop_1 target_ti=4
```

See Also

- syn.directive.inline

syn.directive.pipeline

Description

Reduces the initiation interval (II) for a function or loop by allowing the concurrent execution of operations. See the Pipelining Paradigm section of the *Vitis High-Level Synthesis User Guide* (UG1399) for details. A pipelined function or loop can process new inputs every N clock cycles, where N is the initiation interval (`II`). An II of 1 processes a new input every clock cycle.

As a default, the tool generates the minimum II for the design according to the clock period constraint. The emphasis is on meeting timing, rather than on achieving II, unless you specified the `-II`.

`syn.compile.pipeline_style` defines the default type of pipeline command as described in Compile Options. You can override it with the PIPELINE pragma or directive.

If Vitis HLS cannot create a design with the specified II, it issues a warning and creates a design with the lowest achievable II. You can analyze the design with the warning messages to determine the steps for creating a design satisfying the required initiation interval.

Syntax

```
syn.directive.pipeline=[OPTIONS] <location>
```

Where:

- `<location>` is the location (in the format `function[/label]`) to be pipelined.

Options

- **`II=<integer>`**: Specifies the desired initiation interval for the pipeline. Vitis HLS tries to meet this request. Based on data dependencies, the actual result can have a larger II.
- **`off`**: Turns off pipeline for a specific loop or function. Use this option when using `config_compile -pipeline_loops` to globally pipeline loops.
- **`rewind`**:

  Note: Applicable only to a loop.

Optional keyword. Enables rewinding as described in Rewiding Pipelined Loops for Performance of*Vitis High-Level Synthesis User Guide* (UG1399). This enables continuous loop pipelining with no pause between one execution of the loop ending and the next execution starting. Rewinding is effective only if there is one single loop (or a perfect loop nest) inside the top-level function. The code segment before the loop:

- Is considered as initialization.
- Is executed only once in the pipeline.
- Cannot contain any conditional operations (`if-else`).

- **`style=<stp | frp | flp>`:** Specifies the type of pipeline to use for the specified function or loop. For more information on pipeline styles refer to the Flushing Pipeline and Pipeline Types section of the *Vitis High-Level Synthesis User Guide* (UG1399). The types of pipelines include:
  - **`stp`:** Stall pipeline. Runs only when input data is available otherwise it stalls. This is the default setting, and is the type of pipeline used by Vitis HLS for both loop and function pipelining. Use this when you don't require a flushable pipeline. For example, when there are no performance or deadlock issues due to stalls.
  - **`flp`:** Defines the pipeline as a flushable pipeline. This type of pipeline typically consumes more resources and can have a larger II because you cannot share resources among pipeline iterations.
  - **`frp`:** Free-running, flushable pipeline. Runs even when input data is not available. Use this when you need better timing due to reduced pipeline control signal fanout, or when you need improved performance to avoid deadlocks. However, this pipeline style can consume more power as the pipeline registers are clocked even if there is no data.

IMPORTANT! *This constraint is not a hard constraint. The tool checks design conditions for enabling pipelining. It is possible for some loops to not conform to a particular style and the tool reverts to the default style (`stp`) when necessary.*

Examples

Function `func` is pipelined with the specified initiation interval.

```
syn.directive.pipeline=func II=1
```

See Also

- syn.directive.dependence

syn.directive.protocol

Description

Specifies a region of code, a protocol region, in which Vitis HLS does not insert clock operations unless the code explicitly specifies. The tool does not insert any clocks between operations in the region, including clocks that read or write to function arguments. The synthesized RTL strictly follows the order of read and writes.

A region of code can be created in the C/C++ code by enclosing the region in braces "{ }" and naming it. The following defines a region named `io_section`:

```
io_section:{
...
lines of code
...
}
```

You can specify a clock operation in C code using an `ap_wait()` statement. You can specify a clock operation in C++ code using the `wait()` statement.

TIP: *The `ap_wait` and `wait` statements have no effect on the simulation of the design.*

Syntax

```
syn.directive.protocol=[OPTIONS] <location>
```

The `<location>` specifies the location (in the format `function[/label]`) where you define the protocol region.

Options

- **`mode=[floating | fixed]`:**
  - `floating`: Lets code statements outside the protocol region overlap and execute in parallel with statements in the protocol region in the final RTL. The protocol region remains cycle accurate, but outside operations can occur at the same time. This is the default mode.
  - `fixed`: The fixed mode ensures that statements outside the protocol region do not execute in parallel with the protocol region.

Examples

The example code defines a protocol region, `io_section` in function `foo`. The following directive defines that region as a fixed mode protocol region:

```
syn.directive.protocol=mode=fixed foo/io_section
```

syn.directive.reset

Description

The RESET pragma or directive adds or disables reset ports for specific state variables (global or static).

Use the reset port to restore the registers and block RAM connecting to the port to an initial value. Do this anytime when you apply a reset signal. Globally, the `syn.rtl.reset` configuration settings configure the presence and behavior of the RTL reset port. The reset configuration settings include the ability to define the polarity of the reset, and specify whether the reset is synchronous or asynchronous. More importantly, the reset configuration settings controls which registers are reset when the reset signal is applied. For more information, see Controlling Initialization and Reset Behaviour in *Vitis High-Level Synthesis User Guide* (UG1399).

The RESET pragma provides more specific control over reset. For global or static variables, the RESET pragma explicitly enables a reset when none is present. You can also remove the variable from the reset by turning `off` the pragma. This can be particularly useful when static or global arrays are present in the design.

***Note*:** For public variables of a class, you must use the RESET pragma as the reset configuration settings only apply to variables declared at the function or file level. In addition, you must apply the RESET pragma to an object of the class in the top-function or sub-function. You cannot apply it to private variables of the class.

Syntax

Place the pragma in the C source within the boundaries of the variable life cycle.

```
syn.directive.reset=<location> [ variable=<a> | off=true ]
```

Where:

- **`location>`:** Specifies the location (in the format `function[/label]`) where you define the variable.
- **`variable=<a>`:** Specifies the variable to which you apply the RESET pragma.
- **`off`** or **`off=true`:** Indicates that reset is not generated for the specified variable.

Examples

Adds reset to variable `a` in function `foo` even when the global reset (`syn.rtl.reset`) setting is `none` or `control`.

```
syn.directive.reset=foo a
```

Removes reset from variable `static int a` in function `foo` even when the global reset setting is `state` or `all`.

```
syn.directive.reset=off foo a
```

The following example shows the use of the RESET pragma or directive with class variables and methods. When you declare a variable in a class, it must be a public static variable in the class. You can create the RESET pragma for the variable in a method of the class. You can also create the RESET pragma after creating an object in the top function or sub-function of the HLS design.

```
class bar {
public:
  static int a; // class level static; must be public
  int a1;       // class-level non-static; must be public
  bar(...) {
  // #pragma reset does not work here for a or a1
  }
  // this is the function that is called in the top
  void run(...) {
  // #pragma reset does not work here for non-static variable a1
  // but does work for static variable a
  #pragma reset variable=a
  }
};
static int c; // file level
int d; // file level
void top(...) {
#pragma HLS top
  static int b; // function level
  bar t;
  static bar s;

  // #pragma reset works here for b, c and d, as well as t and s members
  // except for t.a1 because it is not static
  #pragma reset variable=b
  #pragma reset variable=c
  #pragma reset variable=d
  #pragma reset variable=t.a
  #pragma reset variable=s.a
  #pragma reset variable=s.a1
  t.run(...);
  s.run(...);
}
```

The following shows the `syn.directive.reset` command specified for the function `top`.

```
syn.directive.reset=top variable=b
syn.directive.reset=top variable=c
syn.directive.reset=top variable=d
syn.directive.reset=top variable=t.a
syn.directive.reset=top variable=s.a
syn.directive.reset=top variable=s.a1
```

See Also

- RTL Configuration

syn.directive.stable

Description

Apply the `syn.directive.stable` to arguments of a DATAFLOW or PIPELINE region. The pragma indicates that an input or output of this region can be ignored when generating the synchronizations at entry and exit of the DATAFLOW region. Reading accesses of that argument do not need to be part of the "first stage" of the task-level (resp. fine-grain) pipeline for inputs. Write accesses do not need to be part of the last stage of the task-level (resp. fine-grain) pipeline for outputs.

This pragma can be specified at any point in the hierarchy, on a scalar or an array. The pragma automatically applies to all the DATAFLOW or PIPELINE regions below that point. A DATAFLOW or PIPELINE region can start another iteration even though the value of the previous iteration has not been read yet. For an output, a write of the next iteration can occur even when the previous iteration is not complete.

Syntax

```
syn.directive.stable=<location> <variable>
```

- `<location>` is the function name or loop name for constraining the directive.
- `<variable>` is the name of the array to be constrained.

Examples

In the following example, without the STABLE directive, `proc1` and `proc2` synchronize to acknowledge the reading of their inputs (including `A`). With the directive, `A` is no longer considered as an input requiring synchronization.

```
void dataflow_region(int A[...], int B[…] ...
    proc1(...);
    proc2(A, ...);
```

The directives for this example are described below.

```
syn.directive.stable=dataflow_region variable=A
syn.directive.dataflow dataflow_region
```

See Also

- syn.directive.dataflow
- syn.directive.pipeline

syn.directive.stream

Description

By default, array variables implement as RAM.

- Top-level function array parameters implement as a RAM interface port.
- General arrays implement as RAMs for read-write access.
- Arrays involved in sub-functions, or loop-based DATAFLOW optimizations implement as a RAM ping-pong buffer channel.

If the data stored in the array is consumed or produced in a sequential manner, it is more efficient to use streaming data, with FIFOs instead of RAMs. When specifying an argument of the top-level function as INTERFACE `mode=ap_fifo`, the array implements as streaming. See the Interfaces for Vivado IP Flow section of the *Vitis High-Level Synthesis User Guide* (UG1399) or more information.

IMPORTANT! *To preserve the accesses, it might be necessary to prevent compiler optimizations (in particular dead code elimination) by using the `volatile` qualifier. See the Type Qualifiers section of the Vitis High-Level Synthesis User Guide (UG1399).*

Syntax

```
syn.directive.stream=[OPTIONS] <location> <variable>
```

- `<location>` is the location (in the format `function[/label]`) which contains the array variable.
- `<variable>` is the array variable to be implemented as a FIFO.

Options

- **`depth=<integer>`:**

  ***Note*:** Relevant only for array streaming in dataflow channels.

  By default, the depth of the FIFO in the RTL is the same size as the array in the C code. This options allows you to modify the size of the FIFO.

  When implementing the array in a DATAFLOW region, you can use the `-depth` option to reduce the size of the FIFO. For example, in a `DATAFLOW` region where all loops and functions are processing data at a rate of II = 1, you don't need a large FIFO. Data is produced and consumed in each clock cycle. You can use the `-depth` to reduce the FIFO size to 2 to substantially reduce the area of the RTL design.

  The `config_dataflow` command with the `-depth` option provides the same functionality for all arrays in a DATAFLOW region. The `-depth` option with `set_directive_stream` overrides the default in `config_dataflow`.

- **type=<arg>:** Specifies a mechanism to select between FIFO, PIPO, synchronized shared (`shared`), and un-synchronized shared (`unsync`). The supported types include:
  - `fifo`: A FIFO buffer with the specified `depth`.
  - `pipo`: A regular Ping-Pong buffer, with as many "banks" as the specified depth (default is 2).
  - `shared`: A shared channel, synchronized like a regular Ping-Pong buffer, with depth, but without duplicating the array data. Consistency can be ensured by setting the depth small enough, which acts as the distance of synchronization between the producer and consumer.

    TIP: *The default depth for shared is 1.*
  - `unsync`: Does not have any synchronization except for individual memory reads and writes. Consistency (read-write and write-write order) must be ensured by the design itself.

Examples

Specifies array `A[10]` in function `func` to be streaming and implemented as a FIFO.

```
syn.directive.stream=func A type=fifo
```

Array B in named loop `loop_1` of function `func` is set to streaming with a FIFO depth of 12. In this case, place the pragma inside `loop_1`.

```
syn.directive.stream=depth=12 type=fifo func/loop_1 B
```

Array C has streaming implemented as a PIPO.

```
syn.directive.stream=type=pipo func C
```

See Also

- syn.directive.dataflow
- syn.directive.interface
- Dataflow Configuration

syn.directive.top

Description

Attaches a name to a function. The name can then be used by the `set_top` command to set the named function as the top. This is typically used to synthesize member functions of a class in C++.

Syntax

```
syn.directive.top=[OPTIONS] <location>
```

- `<location>` is the function to be renamed.

Options

- **`name=<string>`:** Specifies the name of the function to be used by the `syn.top` command as described in HLS General Options.

Examples

Function `foo_long_name` renames to `DESIGN_TOP`. The top-level is then specified as `DESIGN_TOP`. If the pragma is placed in the code, the `set_top` command must still be issued in the top-level specified in the GUI project settings.

```
syn.directive.top=name=DESIGN_TOP foo_long_name
```

Followed by the `syn.top=DESIGN_TOP` command.

See Also

- `syn.top` in HLS General Options

syn.directive.unroll

Description

Transforms loops by creating multiples copies of the loop body.

The loop induction variable specifies the number of iterations the loop executes. Logic inside the loop body (for example, break or modifications to any loop exit variable) also impacts the number of iterations. A block of logic representing the loop body implements the loop in RTL. The logic executes for the same number of iterations.

The `syn.directive.unroll` command allows partial or full unrolling of the loop. Fully unrolling the loop creates as many copies of the loop body in the RTL as there are loop iterations. Partially unrolling a loop by a factor *N*, creates *N* copies of the loop body and adjusting the loop iteration accordingly.

If factor *N* for partial unrolling is not an integer multiple of the original loop iteration count, check the original exit condition after each loop body's unrolled fragment.

To unroll a loop completely, the loop bounds must be known at compile time. This is not required for partial unrolling.

Syntax

```
syn.directive.unroll=[OPTIONS] <location>
```

- `<location>` is the location of the loop (in the `format function[/label]`) to be unrolled.

Options

- **`factor=<integer>`:** Specifies a non-zero integer indicating that partial unrolling is requested.

  The loop body repeats the integer number of times. The iteration information adjusts accordingly.

- **`skip_exit_check`:** Effective only if a factor is specified (partial unrolling).
  - Fixed bounds

    Does not perform exit condition check if the iteration count is a multiple of the factor.

    If the iteration count is *not* an integer multiple of the factor, the tool:
    - Prevents unrolling.
    - Issues a warning that the exit check is required to proceed.
  - Variable bounds

    The exit condition check is removed. You must ensure that:
    - The variable bounds is an integer multiple of the factor.
    - Requires no exit check.

- **`off=true`:** Disables unroll for the specified loop.

Examples

Unrolls loop `L1` in function `foo`. Places the pragma in the body of loop `L1`.

```
syn.directive.unroll=foo/L1
```

Specifies an unroll factor of 4 on loop `L2` of function `foo`. Removes the exit check. Place the pragma in the body of loop `L2`.

```
syn.directive.unroll=skip_exit_check factor=4 foo/L2
```

See Also

- syn.directive.loop_flatten
- syn.directive.loop_merge

HLS Pragmas

Optimizations in Vitis HLS

In the AMD Vitis™ software platform, a kernel defined in the C/C++ language must be compiled into the register transfer level (RTL) that can be implemented into the programmable logic of an AMD device. The `v++` compiler calls the Vitis High-Level Synthesis (HLS) tool to synthesize the RTL code from the kernel source code.

The HLS tool is intended to work with the Vitis IDE project without interaction. However, the HLS tool also provides pragmas that can be used to optimize the design, reduce latency, improve throughput performance, and reduce area and device resource usage of the resulting RTL code. These pragmas can be added directly to the source code for the kernel.

The HLS pragmas include the optimization types specified in the following table.

For detailed pragma information, refer to the *Vitis High-Level Synthesis User Guide* (UG1399).

Table 7: **Vitis HLS Pragmas by Type**

Type	Attributes
Kernel Optimization	• pragma HLS aggregate • pragma HLS bind_op • pragma HLS bind_storage • pragma HLS expression_balance • pragma HLS latency • pragma HLS reset • pragma HLS top
Function Inlining	• pragma HLS inline
Interface Synthesis	• pragma HLS interface
Task-level Pipeline	• pragma HLS dataflow • pragma HLS stream
Pipeline	• pragma HLS pipeline • pragma HLS occurence
Loop Unrolling	• pragma HLS unroll • pragma HLS dependence
Loop Optimization	• pragma HLS loop_flatten • pragma HLS loop_merge • pragma HLS loop_tripcount
Array Optimization	• pragma HLS array_partition • pragma HLS array_reshape
Structure Packing	• pragma HLS aggregate • pragma HLS dataflow

v++ Linking and Packaging Options

Vitis Subsystem (VSS) Creation using the V++ Linker

A Vitis Subsystem (VSS) is created by the v++ linker using `--mode vss`. An example command is shown below:

```
v++ --link --mode vss --save-temps --part <part_name> --
config ./src/vss_conn.cfg <list_of_xo> <list_of_libadf_partitions> --
out_dir <build_folder>
```

The following are options used in VSS generation.

- `--part` specifies a target part
- `--config` defines connectivity and other directives
- `--out_dir` specifies a folder containing metadata (`.vss`) and binary files

See Linking a VSS Component in the *Embedded Design Development Using Vitis* (UG1701) for details.

Linking

The individual `v++ --link` commands can be found in the v++ Command chapter. See Linking the System in the *Embedded Design Development Using Vitis* (UG1701).

Packaging

The details of the v++ packaging process are provided in Integrating the System in the *Embedded Design Development Using Vitis* (UG1701).

The individual `v++ --package` commands can be found in v++ Command.

--advanced Options

The `--advanced.param` and `--advanced.prop` options specify parameters and properties for use by the `v++` command. When compiling or linking, these options offer fine-grain control over the hardware generated by the Vitis core development kit, and the hardware emulation process.

The arguments for the `--advanced.xxx` options are specified as `<param_name>=<param_value>`. For example:

```
v++ --link --advanced.param compiler.enableXSAIntegrityCheck=true
--advanced.prop kernel.foo.kernel_flags="-std=c++0x"
```

TIP: *The order of precedence between the command line and the config file, and multiple entries in the config file are described in Vitis Compiler Configuration File. However, the* `--advanced` *commands described here have the opposite precedence: config file commands take precedence over command-line arguments, and the last command encountered takes precedence over any earlier occurrences.*

--advanced.param

```
--advanced.param <param_name>=<param_value>
```

Specifies advanced parameters as described in the following table.

Param Options

Table 8: **Param Options**

Parameter Name	Valid Values	Description
`compiler.acceleratorBinaryContent`	Type: String Default Value: `<empty>`	Design content to insert in the generated `xclbin` file. Valid options include `bitstream`, `pdi`, or `dcp`. `bitstream` and `pdi` are mutually exclusive. `pdi` applies to Versal platforms, `bitstream`applies to non-Versal platforms. **TIP:** You can specify two values to have `v++` generate two `xclbin` files: one containing a DCP file, and the other containing either a bitstream or PDI file. For example: `--advanced.param compiler.acceleratorBinaryContent=dcp,bitstream --advanced.param compiler.acceleratorBinaryContent=dcp,pdi` This parameter is used while building the hardware target, this option applies to: • `v++ --link` • `vpl.impl` • `xclbinutil`
`compiler.axiDeadLockFree`	Type: Boolean Default Value: TRUE	Avoid dead locks. This option is enabled by default for Vitis HLS.

Table 8: **Param Options** *(cont'd)*

Parameter Name	Valid Values	Description
`compiler.deadlockDetection`	Type: Boolean Default Value: FALSE	Enables detection of kernel deadlocks during the simulation run as part of hardware emulation. The tool posts an Error message to the console and the log file when the application is deadlocked: `// ERROR!!! DEADLOCK DETECTED at 42979000 ns! SIMULATION WILL BE STOPPED! //` The message is repeated until the deadlock is terminated. You must manually terminate the application to end the deadlock condition. **TIP:** *When deadlocks are encountered during simulation, you can open the kernel code in Vitis HLS for additional deadlock detection and debug capability.* Applies to: • `v++ --compile` • Vitis HLS • `config_export`
`compiler.emulationMode=<mode>`	Type: String func | rtl	Indicates that the kernel should be compiled as RTL code for use in hardware emulation and hardware design, or as a C functional model with a SystemC wrapper for use in hardware emulation. This option applies to `v++ --compile`. The default is to compile the kernel as RTL code.
`compiler.enableIncrHwEmu`	Type: Boolean Default Value: FALSE	Use to enable incremental compilation of the hardware emulation `xclbin` when there are minor changes made to the platform. This enables a quick rebuild of the device binary for hardware emulation when the platform has been updated. Applies to: • `v++ --link` • `vpl.impl`
`compiler.errorOnHoldViolation`	Type: Boolean Default Value: TRUE	After the last step of Vivado implementation, during timing analysis check, and clock scaling if needed. If hold violations are found, `v++` quits and returns an error by default, and does not generate an `xclbin`. This parameter lets you over ride the default behavior. Applies to: • `v++ --link` • `vpl.impl`
`compiler.interfaceRdBurstLen`	Type: Int Range Default Value: `0`	Specifies the expected length of AXI read bursts on the kernel AXI interface. This is used with option `compiler.interfaceRdOutstanding` to determine the hardware buffer sizes. Values are 1 through 256. Applies to: • `v++ --compile` • Vitis HLS • `config_interface`

Table 8: **Param Options** *(cont'd)*

Parameter Name	Valid Values	Description
`compiler.interfaceWrBurstLen`	Type: Int Range Default Value: `0`	Specifies the expected length of AXI write bursts on the kernel AXI interface. This is used with option `compiler.interfaceWrOutstanding` to determine the hardware buffer sizes. Values are 1 through 256. Applies to: • `v++ --compile` • Vitis HLS • `config_interface`
`compiler.interfaceRdOutstanding`	Type: Int Range Default Value: `0`	Specifies how many outstanding reads to buffer are on the kernel AXI interface. Values are 1 through 256. Applies to: • `v++ --compile` • Vitis HLS • `config_interface`
`compiler.interfaceWrOutstanding`	Type: Int Range Default Value: `0`	Specifies how many outstanding writes to buffer are on the kernel AXI interface. Values are 1 through 256. Applies to: • `v++ --compile` • Vitis HLS • `config_interface`
`compiler.skipTimingCheckAndFrequencyScaling`	Type: Boolean Default Value: FALSE	This parameter causes the Vivado tool to skip the timing check and optional clock frequency scaling that occurs after the last step of implementation process, which is either `route_design` or post-route `phys_opt_design`. Applies to: • `v++ --link` • `vpl.impl`
`compiler.userPreCreateProjectTcl`	Type: String Default Value: `<empty>`	Specifies a Tcl script to run before creating the Vivado project in the Vitis build process. Applies to: • `v++ --link` • `vpl.create_project`
`compiler.userPreSysLinkOverlayTcl`	Type: String Default Value: `<empty>`	Specifies a Tcl script to run after opening the Vivado IP integrator block design, before running the compiler-generated `dr.bd.tcl` script in the Vitis build process. Applies to: • `v++ --link` • `vpl.create_bd`
`compiler.userPostSysLinkOverlayTcl`	Type: String Default Value: `<empty>`	Specifies a Tcl script to run after running the compiler-generated `dr.bd.tcl` script. Applies to: • `v++ --link` • `vpl.update_bd`

Table 8: **Param Options** *(cont'd)*

Parameter Name	Valid Values	Description
`compiler.userPostDebugProfileOverlayTcl`	Type: String Default Value: `<empty>`	Specifies a Tcl script to run after debug profile overlay insertion in Vivado IP integrator block design in the `vpl.update_bd` step. Applies to: • `v++ --link` • `vpl.updated_bd`
`compiler.worstNegativeSlack`	Type: Float Default Value: 0	During timing analysis check, this specifies the worst acceptable negative slack for the design, specified in nanoseconds (ns). When negative slack exceeds the specified value, the tool might try to scale the clock frequency to achieve timing results. This specifies an acceptable negative slack value instead of zero slack. Applies to: • `v++ --link` • `vpl.impl`
`compiler.xclDataflowFifoDepth`	Type: Int Default Value: `-1`	Specifies the depth of FIFOs used in kernel data flow region. Applies to: • `v++ --compile` • Vitis HLS • `config_dateflow`
`hw_emu.aie_shim_sol_path`	Type: String Default Value: `<empty>`	For use by Versal platforms, this option specifies the path to the AI Engine Interface Tile constraints file which is generated by the `aiecompiler`. Used during simulation, compilation, and elaboration, the file provides a logical mapping to the physical interface. This is needed for third-party simulators like Mentor Graphics Questa Advanced Simulator or Cadence Xcelium Logic Simulation.
`hw_emu.compiledLibs`	Type: String Default Value: `<empty>`	Uses mentioned `clibs` for the specified simulator. Applies to Hardware Emulation and Debug.
`hw_emu.debugMode`	wdb Default Value: `wdb`	The default value is WDB and runs simulation in waveform mode. This option only works in combination with the `-g` or `--debug` options. Applies to Hardware Emulation and Debug.
`hw_emu.enableProtocolChecker`	Type: Boolean Default Value: FALSE	Enables the lightweight AXI protocol checker (lapc) during HW emulation. This is used to confirm the accuracy of any AXI interfaces in the design. Applies to Hardware Emulation and Debug.
`hw_emu.json_device_file_path`	Type: String Default Value: `<empty>`	For use by Versal platforms, this option specifies the path to the AI Engine JSON Device file located in the Vitis software installation area. Used during simulation, compilation, and elaboration, the file specifies the size of the AI Engine array. This is needed for third-party simulators like Mentor Graphics Questa Advanced Simulator or Cadence Xcelium Logic Simulation.

Table 8: **Param Options** *(cont'd)*

Parameter Name	Valid Values	Description
`hw_emu.platformPath`	Type: String Default Value: `<empty>`	Specifies the path to the custom platform directory. The `<platformPath>` directory should meet the following requirements to be used in platform creation: • The directory should contain a subdirectory called `ip_repo`. • The directory should contain a subdirectory called `scripts` and this `scripts` directory should contain a `hw_em_util.tcl` file. The `hw_em_util.tcl` file should have the following two procedures defined in it: ◦ `hw_em_util::add_base_platform` ◦ `hw_em_util::generate_simulation_scripts_and_compile` Applies to Hardware Emulation and Debug.
`hw_emu.post_sim_settings`	Type: String	Specifies the path to a Tcl script that is used to configure the settings of the Vivado simulator prior to running hardware emulation. This script is run after the default configuration of the tool, but prior to launching simulation. You can use the Tcl script to override specific settings, or to custom configure the simulator as needed. Applies to Hardware Emulation and Debug.
`hw_emu.reduceHwEmuCompileTime`	Type: Boolean Default Value: FALSE	Move the generation of the top-level block design into the Generate Targets step of `v++ --link`. Applies to Hardware Emulation and Debug.
`hw_emu.scDebugLevel`	none | waveform | log | waveform_and_log Default Value: waveform_and_log	Sets the TLM transaction debug level of the Vivado logic simulator (`xsim`). • NONE to disable TLM debug • LOG to dump TLM transaction log info into report file • WAVEFORM for enabling the TLM transaction waveform view • WAVEFORM_AND_LOG for both the Log Messages and Waveform view Applies to Hardware Emulation and Debug.
`hw_emu.simulator`	XSIM | QUESTA Default Value: XSIM	Uses the specified simulator for the hardware emulation run. Applies to Hardware Emulation and Debug.
`package.enableEdfHwEmu`	Default values: 1 or 0	For designs that use base platforms prepared for Embedded Development Framework (EDF), but does not have EDF as default flow (like VRK160 boards), you need to set `--advanced.param package.enableEdfHwEmu=1` when packaging. This switch uses QEMU file sets and packaging outputs suitable for EDF. For VEK385, EDF flow is default and you do not need to set this additional parameter.

For example:

```
--advanced.param compiler.addOutputTypes="hw_export"
```

TIP: *This option can be specified in a configuration file under the `[advanced]` section head using the following format:*

```
[advanced]
param=compiler.addOutputTypes="hw_export"
```

--advanced.prop

```
--advanced.prop <arg>
```

Specifies advanced kernel or solution properties for kernel compilation where `<arg>` is one of the values described the following table.

Table 9: **Prop Options**

Property Name	Valid Values	Description
`kernel.<kernel_name>.kernel_flags`	Type: String Default Value: `<empty>`	Sets specific compile flags on the kernel `<kernel_name>`.
`solution.kernel_compiler_margin`	Type: Float Default Value: 12.5% of the kernel clock period.	The clock margin (in ns) for the kernel. This value is subtracted from the kernel clock period prior to synthesis to provide some margin for place and route delays.

--advanced.misc

```
--advanced.misc <arg>
```

Specifies advanced tool directives for kernel compilation.

--clock Options

The `--clock` options are used by the Vitis tool to assign the clock frequency during the compilation of the AI Engine graph, HLS kernels and linking of the design.

For clock directives supported by Vitis compilation, the Vitis tool compiles the AI Engine graph and HLS components. There are different modes to compile AI Engine graph and HLS components. The AI Engine runs at a single rate, although the compiler supports directives to specify expected PLIO peer clock frequencies. If clock directives are not used, the Vitis tool decides the clock frequency based on whether `--part` or `--platform` is used in the command.

Table 10: **Clock Directives for AI Engine Graph Compilation**

Mode	Design	Clock Directive	Vitis Tool Consideration
v++ -c --mode aie	--platform	No Directive	Platform default clock
v++ -c --mode aie	--part	No Directive	0.25 x (AI Engine PLL Frequency)
v++ -c --mode aie	--platform OR --part	--pl-freq=200 --freqhz=200000000 --freqhz=200MHz	Compiles at 200 MHz

***Note*:** The value of `--freqhz` supports either no units (in which case, Vitis considers the unit to be Hz) or MHz. The unit MHz is case insensitive. The frequency should not be a hertz fractional value. For example, `--freqhz=312.005MHz` is supported whereas `--freqhz=312.0000005MHz` is not supported. There must be no space between the value and the units (in MHz). `freqhz` is applicable to all Vitis compilation modes and linking.

The following are the commands with different directives in CLI mode:

- `v++ -c --mode aie --platform <pfm.xpfm> --config <aie.cfg> --freqhz=200000000 ./aie_graph.cpp`
- `v++ -c --mode aie --platform <pfm.xpfm> --config <aie.cfg> --freqhz=200MHz ./aie_graph.cpp`
- `v++ -c --mode aie --platform <pfm.xpfm> --config <aie.cfg> --aie.pl-freq=200 ./aie_grapch.cpp`

Different directives can be given through the configuration file as well:

- `freqhz=200000000`
- `freqhz=200MHz`

The directive `--aie.pl-freq` (the same is applicable to `--freqhz`) can be used to address other PLIO design scenarios as shown below:

- To compile all AI Engine PLIO at 200MHz (Default): `--aie.pl-freq=200`
- To compile specific PLIO at 200MHz: `--aie.pl-freq=<PLIO_PORT_NAME>:<FREQ>`
- To compile multiple graphs at 200MHz: `--aie.pl-freq=<AIE_GRAPH_NAME>.<PLIO_PORT_NAME>:<FREQ>`

***Note*:** `--aie.pl-freq` is declared in MHz, while `--freqhz` is declared in Hz.

Table 11: **Clocking Directives for HLS Compilation**

Mode	Design	Clock Directive	Vitis Tool Consideration
v++ -c --mode hls	--platform	No Directive	Platform default clock
v++ -c --mode hls	--part	No Directive	Default clock=100 MHz
v++ -c --mode hls	--platform OR --part	--hls.clock=200 --hls.clock=0.5ns --freqhz=200000000 --freqhz=200MHz	Compiles HLS kernel at 200 MHz

The following are the commands with different directives in CLI mode:

- `v++ -c --mode hls --platform <pfm.xpfm> --config <hls.cfg> --freqhz=200000000 --work_dir <work_dir_path>`

- `v++ -c --mode hls --platform <pfm.xpfm> --config <hls.cfg> --freqhz=200MHz --work_dir <work_dir_path>`
- `v++ -c --mode hls --platform <pfm.xpfm> --config <hls.cfg> --hls.clock=200 --work_dir <work_dir_path>`
- `v++ -c --mode hls --platform <pfm.xpfm> --config <hls.cfg> --hls.clock=5ns --work_dir <work_dir_path>`

Different directives can also be given via the configuration file:

- ```
  freqhz=200000000
  ```
- ```
  freqhz=200MHz
  ```
- ```
  [hls]
  clock=200
  ```
- ```
  [hls]
  clock=5ns
  ```

Note: Clocking directive support for `v++ -c --mode hls` is analogous to `v++ -c -k`.

For clock directives supported for Vitis linking, the Vitis tool links AI Engine graph, HLS components, and hardware configuration (platform/part) using a configuration file. During the linking, clock directives are used to connect HLS components to the requested clock frequency. In v++ linking, how the Vitis tool connects the requested clock frequency to HLS components and how the v++ linker handles the scenario depends on the clock directives used. Refer to Managing Clock Frequencies in the *Data Center Acceleration using Vitis* (UG1700) for more information about the v++ linking behavior for the clocking directives discussed below.

Table 12: **Clocking Directives for v++ Linking:**

Mode	Design	Clock Directive	Vitis Tool Consideration
v++ -l	--platform	No Directive	Platform default clock
v++ -l	--part	No Directive	For Versal, default clock is 300 MHz. Not supported for other devices.
v++ -l	--platform --part	--clock.freqHz=200000000:<cu_0>[.<clk_pin>] --freqhz=200000000:<cu_0>[.<clk_pin>] --freqhz=200MHz:<cu_0>[.<clk_pin>] --clock.id=<id_value>:<cu_0>[.<clk_pin>]	For freqhz/clock.freqHz: Connect HLS kernel to 200 MHz For clock.id: Connect HLS kernel to clock source <id_value of 200 MHz>

The following are the commands with directives in CLI mode:

- `v++ -l --platform <pfm.xpfm> ./libadf.a ./<kernel_name.xo> --config <system.cfg> --freqhz=200000000:mm2s:aclk -o <fixed.xsa>`
- `v++ -l --platform <pfm.xpfm> ./libadf.a ./<kernel_name.xo> --config <system.cfg> --freqhz=200MHz:mm2s.aclk -o <fixed.xsa>`
- `v++ -l --platform <pfm.xpfm> ./libadf.a ./<kernel_name.xo> --config <system.cfg> --clock.freqHz=200000000:mm2s.aclk -o <fixed.xsa>`
- `v++ -l --platform <pfm.xpfm> ./libadf.a ./<kernel_name.xo> --config <system.cfg> --clock.id=<id_value> -o <fixed.xsa>`

Different directives can be given through configuration file:

- `freqhz=200000000:mm2s.aclk`
- `freqhz=200MHz:mm2s.aclk`
- ```
  [clock]
  freqHz=200000000:mm2s.aclk
  ```

There are two ways to connect the kernel to the specified frequency in the v++ link phase. Clocks available in the platform can be considered either as a clock frequency or as a clock source by the Vitis tool, which depends on the directive. The clock directive `--freqhz or --clock.freqHz` can be used to connect the clock as a clock frequency and `--clock.id` can be used to connect the clock as a clock source:

- `--clock.freqHz` or `--freqhz`: These both directives work in a similar way and do not depend on platform clocking. You can use these directives to connect a platform clock frequency to the kernel:
  - If a platform has multiple clocks available at the requested frequency using `--freqhz`, the Vitis tool selects any one of the platform clock by considering the `clock.tolerance` value (default being 5%).
  - If the platform has multiple clock frequencies available at the requested clock frequency, and you want the Vitis tool to pick a particular clock, you must use the directive `--clock.id` in place of `--freqhz`.
  - If a platform does not have a match for the requested frequency, the Vitis tool generates the requested clock frequency using the available platform fixed clock.
- `--clock.id`: This option can be used to connect a particular platform clock as a clock source to the kernel. The v++ linker can add IPs such as FIFO, data width converter, and clock converter, depends on the Vivado design implemented.
  ```

The PFM.CLOCK status attribute specifies how the v++ linker treats the clock. It can either be fixed or `fixed_non_ref` (scalable clocks are supported only on legacy Alveo platforms). The linker does not employ a `fixed_non_ref` clock as input reference for generated clocks. To set the clock attribute, open the Vivado design, go to Platform Setup, under the Clock section set the status property of the clock as shown below:

Figure 2: **Platform Setup for Clock**

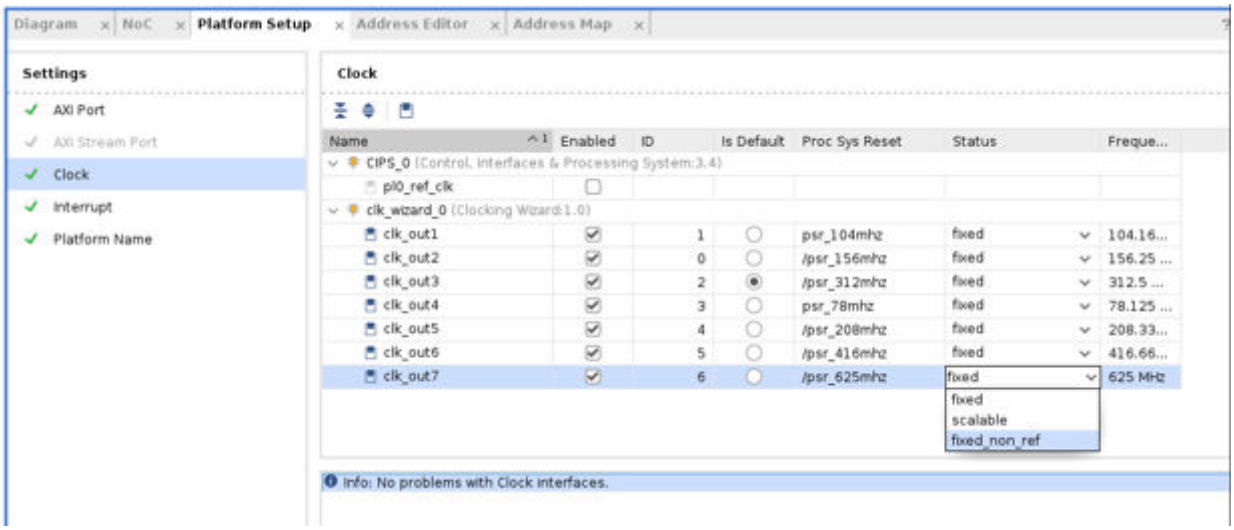


Table 13: **PFM.CLK Status Options**

Status Value	Description
`Scalable`	Scalable clocks used by XRT. Only supported in legacy Alveo designs.
`Fixed`	Used as a reference clock to connect to IP. Used in Embedded design.
`fixed_non_ref`	Cannot be used as a reference clock; can be used to connect IPs in embedded design.

***Note*:** Reference clocks are clocks which the Vitis tool uses (as a clock input) to generate other clocks using MMCM.

For example, suppose a clocking requirement is to generate the output clock with a divider setting of 1, 2, 4, and 8. It is recommended to use the primitive `MBUFGCE` in place of `BUFG` to reduce the clock skew between synchronous clock outputs. The output clocks of `MBUFGCE` cannot be used as a reference clock. Thus, you can set the attribute `status=fixed_non_ref` so that the Vitis tool cannot use these clocks (output of `MBUFGCE`) as reference clocks.

***Note*:** You can determine the available clock IDs for a target platform using the `platforminfo` utility as described in platforminfo Utility. The `platforminfo` report includes the clock status of all the platform clocks.

To determine the fixed clocks available on the platform `xilinx_vck190_base_bdc_202420_1.xpfm`:

Go to `plaftorminfo ../ xilinx_vck190_base_bdc_202420_1.xpfm` and refer to the "Clock Information" section in the report:

```
=================
Clock Information
=================
Default Clock Index: 2
Clock Index: 0
Status: Fixed
Frequency: 104.166666
Clock Index: 1
Status: Fixed
Frequency: 156.250000
Clock Index: 2
Status: Fixed
Frequency: 312.500000
Clock Index: 3
Status: Fixed
Frequency: 78.125000
Clock Index: 4
Status: Fixed
Frequency: 208.333333
Clock Index: 5
Status: Fixed
Frequency: 416.666666
Clock Index: 6
Status: Fixed
Frequency: 625.000000
```

--clock.tolerance

```
--clock.tolerance <arg>
```

Specifies a clock tolerance. When specifying `--clock.freqHz`, you can also specify the tolerance with either a value or percentage of the clock frequency. This updates the timing constraints to reflect the accepted tolerance. `<arg>` is specified as `<tolerance>:<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]`

- **`<tolerance>`:**
  - Can be specified either as a whole number, indicating the `clock.freqHz` ± the specified tolerance value, or as a percentage of the clock frequency specified as a decimal value.
  - `<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]`: Applies the defined clock tolerance to the specified CUs, and optionally to the specified clock pin on the CU.
- **`<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]`:** Applies the defined clock tolerance to the specified CUs, and optionally to the specified clock pin on the CU.

IMPORTANT! *The default clock tolerance is 5% of the specified frequency when this option is not specified.*

For example:

```
v++ --link --clock.tolerance 0.10:vadd_1,vadd_3
```

TIP: *This option can be specified in a configuration file under the* `[clock]` *section head using the following format:*

```
[clock]
tolerance=0.10:vadd_1,vadd_3
```

--connectivity Options

As discussed in Linking the System in the *Embedded Design Development Using Vitis* (UG1701), there are a number of `--connectivity.XXX` options that let you define the interconnection network of `.xo` kernels, AI Engine PLIO and GMIO interfaces, and platform NoC, memories, and stream interfaces. These commands are an integral part of the build process, critical to the definition and construction of the application.

--connectivity.region

```
--connectivity.region <arg>
```

Use this option to specify an affinity for a PL kernel to a PFM.REGION declared in the extensible `.xsa` hardware platform. The typical use of a PFM.REGION declaration is to match a Pblock defined in the hardware platform. An example is to demarcate a Super Logic Region (SLR) within a Stacked Silicon Interconnect (SSI) device such as an `xcvp2802` SoC.

You must use a separate `--connectivity.region` option to map each compute unit to a particular region. The build process automatically places any compute unit that you do not explicitly map to a region through the `--connectivity.region` option in an available region.

RECOMMENDED: *AMD recommends that you specify region labels when working with SSI devices as this provides the greatest placement control and helps ensure that the placer co-locates kernels with their associated resources in the same SLR. However, you can also rely on automatic placement if regional constraints are not critical.*

Valid values include:

```
<region_label>:<cu_name>
```

Where:

- `<region_label>` is the string identifier that you specify in the platform's `PFM.REGION` attribute for the target SLR or logical region.
- `<cu_name>` is the name of the compute unit that you specify in the `--connectivity.nk` option. This defaults to `<kernel_name>_1` unless you specify a different name.

In the linker configuration file, you specify the region assignment as follows:

```
[connectivity]
region=<region_label>:<cu_name_1>,<cu_name_2>
```

--connectivity.nk

```
--connectivity.nk <arg>
```

Where `<arg>` is specified as `<kernel_name>:#:<cu_name1>,<cu_name2>,...<cu_name#>`.

This instantiates the specified number of instances ('compute units') of the specified kernel (`kernel_name`) in the generated FPGA binary (`.xclbin`) file during the linking process. The `cu_name` is optional. If the `cu_name` is not specified, the instances of the kernel are simply numbered: `kernel_name_1`, `kernel_name_2`, and so forth. By default, the Vitis compiler instantiates one compute unit for each kernel.

For example:

```
v++ --link --connectivity.nk vadd:3:vadd_A,vadd_B,vadd_C
```

TIP: *This option can be specified in a configuration file under the `[connectivity]` section head using the following format:*

```
[connectivity]
nk=vadd:3:vadd_A,vadd_B,vadd_C
```

--connectivity.stream_connect

```
--connectivity.sc <arg>
```

Create a streaming connection between two compute units through their AXI4-Stream interfaces. Use a separate `--connectivity.sc` option for each streaming interface connection. The order of connection must be from a streaming output port of the first kernel to a streaming input port of the second kernel. Valid values include:

```
<cu_name>.<streaming_output_port>:<cu_name>.<streaming_input_port>[:<fifo_depth>]
```

Where:

- `<cu_name>` is the compute unit name specified in the `--connectivity.nk` option. Generally this is `<kernel_name>_1` unless a different name was specified.
- `<streaming_output_port>/<streaming_input_port>` is the function argument for the compute unit port that is declared as an AXI4-Stream.
- `[:<fifo_depth>]` inserts a FIFO of the specified depth between the two streaming ports to prevent stalls. The value is specified as an integer.

For example, to connect the AXI4-Stream port `s_out` of the compute unit `mem_read_1` to AXI4-Stream port `s_in` of the compute unit `increment_1`, use the following:

```
--connectivity.sc mem_read_1.s_out:increment_1.s_in
```

TIP: *This option can be specified in a configuration file under the `[connectivity]` section head using the following format:*

```
[connectivity]
sc=mem_read_1.s_out:increment_1.s_in
```

The inclusion of the optional <fifo_depth> value lets the `v++` linker add a FIFO between the two kernels to help prevent stalls. This uses BRAM resources from the device when specified, but eliminates the need to update the HLS kernel to contain FIFOs. The tool also instantiates a Clock Converter (CDC) or Datawidth Converter (DWC) IP if the connections have different clocks, or different bus widths.

--connectivity.system_port

```
--connectivity.sp <arg>
```

Use this option to specify the assignment of kernel arguments to system ports within the platform. A primary use case for this option is to connect kernel arguments to specific memory resources. A separate `--connectivity.sp` option is required to map each argument of a kernel to a particular memory resource. Any argument not explicitly mapped to a memory resource through the `--connectivity.sp` option is automatically connected to an available memory resource during the build process.

Valid values include:

```
<cu_name>.<kernel_argument_name>:<sptag[min:max]>
```

Where:

- `<cu_name>` is the name of the compute unit as specified in the `--connectivity.nk` option. Generally this is `<kernel_name>_1` unless a different name was specified.
- `<kernel_argument_name>` is the name of the function argument for the kernel, or the compute unit interface port.
- `<sptag>` represents a system port tag, such as for memory controller interface names from the target platform. Valid `<sptag>` names include DDR, PLRAM, and HBM.
- `[min:max]` enables the use of a range of memory, such as DDR[0:2]. A single index is also supported: DDR[2].

TIP: *The supported `<sptag>` and range of memory resources for a target platform can be obtained using the `platforminfo` command. Refer to platforminfo Utility for more information.*

The following example maps the input argument of an AI Engine GMIO to interleaved LPDDR C2_C3 on the vek385_base platform:

```
v++ --link -f vek385_base --connectivity.sp ai_engine_0:out_gmio:LPDDR23
```

TIP: *This option can be specified in a configuration file under the* `[connectivity]` *section head using the following format:*

```
[connectivity]
sp=vadd_1.A:DDR[0:3]
sp=vadd_1.B:HBM[0:31]
sp=vadd_1.C:PLRAM[2]
```

--connectivity.noc.connect

```
  --connectivity.noc.connect <arg>
```

Where `<arg>` is in the form of `<compute_unit_name>.<kernel_interface_name>:<noc interface>`, and specifies a connection between the PL kernel interface and the Versal NoC. Valid values are internal memory controllers, or master interfaces on the Versal NoC cell.

The Vitis compiler estimates kernel bandwidth requirements based on NoC connectivity and M_AXI properties (datawidth * clock freq) across the dynamic region, and automatically sets NoC configuration settings for read and write bandwidth, scaling as needed to avoid exceeding the available bandwidth.

For example:

```
[connectivity]
noc.read_bw=mm2s.M_AXI:2000.16
noc.write_bw=mm2s.M_AXI:2010.16
noc.connect=mm2s.M_AXI:M00_INI
```

--connectivity.noc.read_bw

```
--connectivity.noc.read_bw <arg>
```

Where `<arg>` is in the form `<compute_unit_name>.<kernel_interface_name>:<Bandwidth>.<Avg_burst_length>`, and specifies both the bandwidth and burst length of the connection. The bandwidth is specified in MB/s.

This option specifies expected read traffic characteristics on M_AXI interfaces to let you override the automatic Versal NoC configuration.

--connectivity.noc.write_bw

```
--connectivity.noc.write_bw <arg>
```

Where <arg> is in the form `<compute_unit_name>.<kernel_interface_name>:<Bandwidth>.<Avg_burst_length>`, and specifies both the bandwidth and burst length of the connection. The bandwidth is specified in MB/s.

This option specifies expected write traffic characteristics on M_AXI interfaces to let you override the automatic Versal NoC configuration.

--connectivity.connect

```
--connectivity.connect <X:Y>
```

This option can be used to make connections through the Vivado IP integrator, but `v++` does not perform any error checking on the specified connections. Use this to specify general connections between kernels and non-AXI elements of the target platform, such as connections to GT ports.

The X and Y connections must be specified as arguments compatible with either the IP integrator `connect_bd_net` or `connect_bd_intf_net` commands. The specific format of `<X:Y>` is:

```
src/hierarchy_name/cell_name/pin_name:dst/hierarchy_name/cell_name/pin_name
```

These cannot include connections between AXI4-Stream interfaces which require the use of `--conectivity.sc`, or M_AXI interfaces which require the use of `--connectivity.sp` as described above.

TIP: *This option can be specified in a configuration file under the `[connectivity]` section head using the following format:*

```
[connectivity]
connect=<X:Y>
```

--debug Options

This option enables debug IP core insertion in the device binary (`.xclbin`) for hardware debugging. This option lets you specify the type of debug core to add, and which compute unit and interfaces to monitor with ChipScope™ as described in Debugging During Hardware Execution in the *Data Center Acceleration using Vitis* (UG1700). The `--debug.xxx` options lets you attach AXI protocol checkers and System ILA cores at the interfaces to kernels or specific compute units (CUs) for debugging and performance monitoring purposes:

- The System Integrated Logic Analyzer (ILA) provides transaction level visibility into an accelerated kernel or function running on hardware. AXI traffic of interest can also be captured and viewed using the System ILA core.
- The AXI Protocol Checker debug core is designed to monitor AXI interfaces on the accelerated kernel. When attached to an interface of a CU, the AXI Protocol Checker actively checks for protocol violations and provides an indication of which violation occurred.

The `--debug.xxx` commands can be specified in a configuration file under the `[debug]` section head using the following format as an example:

```
[debug]
protocol=all:all            # Protocol analyzers on all CUs
protocol=cu2:port3          # Protocol analyzer on port3 of cu2
chipscope=cu2               # ILA on cu2
```

The various options of `--debug` include the following:

--debug.aie.chipscope

```
--debug.aie.chipscope <interface_name> | <adf_graph_arg_name>
```

Enables hardware debug for the Versal AI Engine through ChipScope. The `<interface_name>` argument applies to non-PL kernel interfaces such as AI Engine PLIO interfaces, or AXIS interfaces. The `<adf_graph_arg_name>` specifies arguments of the graph.

--debug.chipscope

```
--debug.chipscope <cu_name>[:<interface_name>]
```

Adds the System Integrated Logic Analyzer debug core to the specified CUs in the design.

IMPORTANT! *The `--debug.chipscope` option requires the `<cu_name>` to be specified and does not accept the keyword `all`. You can optionally specify an `<interface_name>`.*

For example, the following command adds an ILA core to the `vadd_1` CU:

```
v++ --link --debug.chipscope vadd_1
```

--debug.list_ports

Shows a list of valid compute units and port combinations in the current design. This is informational to help you with crafting a command line or config file for the `--debug` command.

This option needs to be specified during linking, but does not run the linking process. The required elements of the command line are shown in the following example, which returns the available ports when linking the specified kernels with the listed platform:

```
v++ --platform <platform> --link --debug.list_ports <kernel.xo>
```

--debug.protocol

```
--debug.protocol all|<cu_name>[:<interface_name>]
```

Adds the AXI Protocol Checker debug core to the design. This can be specified with the keyword `all`, or the `<cu_name>` and optional `<interface_name>` to add the protocol checker to the specified CU and interface.

For example:

```
v++ --link --debug.protocol all
```

--drc Options

This option lets you manage the Design Rule Check (DRC) messages returned by the Vivado Design Suite during implementation of the design. Messages can be disabled or enabled, reduced in severity, or waived to allow the design to proceed.

--drc.disable

```
--drc.disable <arg>
```

Lets you disable the DRC message specified by the DRC ID. Disabled DRCs are not checked.

For example:

```
v++ --link --drc.disable TIMING-18
```

--drc.enable

```
--drc.enable <arg>
```

Lets you enable the DRC message specified by the DRC ID. DRC checks that have been disabled can be re-enabled as needed.

For example:

```
v++ --link --drc.enable TIMING-18
```

--drc.severity

```
--drc.severity <arg>
```

Change the severity of a DRC check. For example, instead of a {CRITICAL WARNING} a violation is only reported as a WARNING. The DRC must be specified by ID, and the new severity to apply.

For example:

```
v++ --link --drc.severity TIMING-18:Warning
```

--drc.waive

```
--drc.waive <arg>
```

Lets you waive the specified DRC ID. This lets the Vivado tool proceed through implemntation, ignoring DRCs that might otherwise prevent completion of the design. A waived rule is still checked, but it is marked as waived.

For example:

```
v++ --link --drc.waive TIMING-18
```

--linkhook Options

The `--linkhook.XXX` options described below are used to specify Tcl scripts to run at specific steps during the Vitis linking process. Valid steps can be determined using the `--linkhook.list_steps` command as described below.

--linkhook.custom

```
--linkhook.custom <step name, path to script file>
```

Specifies a Tcl script to execute at a predefined point in the build process. The path to specify the script can be an absolute path, or partial path relative to the build directory.

For example, the following command runs the specified Tcl script before the SysLink step in the build:

```
v++ -l --linkhook.custom preSysLink,./runScript.tcl
```

-linkhook.do_first

```
--linkhook.do_first <step name, path to script file>
```

Specifies a Tcl script to execute before the given step name. The path to specify the script can be an absolute path, or partial path relative to the build directory.

For example, the following command runs the specified Tcl script before the `place_design` step in the build:

```
v++ -l --linkhook.do_first vpl.impl.place_design,runScript.tcl
```

-linkhook.do_last

```
--linkhook.do_last <step name, path to script file>
```

Specifies a Tcl script to execute immediately after the given step completes. The path to specify the script can be an absolute path, or partial path relative to the build directory.

For example, the following command runs the specified Tcl script after the `place_design` step in the build:

```
v++ -l --linkhook.do_last vpl.impl.place_design,runScript.tcl
```

-linkhook.list_steps

```
--linkhook.list_steps
```

List default and optional build steps that support script hooks for a specified target. This command requires the `--target` to be specified in addition to the `--link` option.

For example:

```
v++ --target hw -l --linkhook.list_steps
```

The command returns both default steps that are always enabled during the build process, and optional steps that you can enable if needed. For directions on enabling optional steps, refer to the following:

- For data center acceleration: Managing Vivado Synthesis, Implementation, and Timing Closure in the *Data Center Acceleration using Vitis* (UG1700)
- For embedded design development: Managing Vivado Synthesis, Implementation, and Timing Closure in the *Embedded Design Development Using Vitis* (UG1701)

--package Options

Introduction

As described in Packaging for Vitis Flow in the *Embedded Design Development Using Vitis* (UG1701), the `v++ --package`, or `-p` step, generates and packages the final product at the end of the `v++` compile and link build process. This is a required step for all embedded platforms, including Versal devices, AI Engine, and AMD Zynq™ UltraScale+™ MPSoC devices.

The syntax of the `--package` command for Versal platforms is as follows:

```
v++ --package --platform fixed.xsa -o output.xclbin --package.<other
options>
```

TIP: *Package command options can be specified in a configuration file for use with the `--config` option, as discussed in the Vitis Compiler Configuration File.*

For non-Versal platforms the syntax for the `--package` command is:

```
v++ --package -t < hw_emu | hw> --platform <platform> input.xclbin \
[ -o output.xclbin --package.<options> ]
```

The various options to specify as `--package.<options>` as shown in the syntax above include the following:

--package.aie_metadata_only

```
--package.aie_metadata_only {L_ij: i=1,...,P, any j}
```

Where `{L_ij: i=1,...,P, any j}` specifies the `libadf.a` library files for AIE partitions to include metadata only in the `.xclbin` file for independent control against a loaded image. The L_{ij} represents libraries. The index `i` represents the i^{th} partition and `j` can be any value. The default behavior is to include full AIE partition data and metadata.

Additional Outputs:

- `aie.pdi`
- `aie.bif`

Use Case: Creates a metadata-only `.xclbin` file for independent control against a loaded image. This allows runtime control of AIE partitions without including the full partition data in the `.xclbin` file.

Example:

```
v++ -p -f fixed.xsa --package.aie_metadata_only {L_ij: i=1,...,P, any j} -o
aie-meta.xclbin
```

--package.aie_overlay

```
--package.aie_overlay {L_ij}
```

Where this option specifies that the package command should include the `aie.pdi` file for merged positional argument `libadf.a` files and set the LOAD_PDI action mask. The `{L_ij}` must represent adjacent partitions. The default behavior is to not create an AIE overlay.

Additional Outputs:

- `aie.pdi` - Includes the PDI file for merged positional argument `libadf.a` files
- `aie.bif` - Boot Image Format file

Important: The L_{ij} libraries must represent adjacent partitions for this option to work correctly.

Use Case: Enables AIE overlay functionality, allowing users to load a `pl.xclbin` and then load/reload any number of `aie.xclbin` overlays. This is particularly useful in DFX designs where AIE overlays can be dynamically loaded on top of a PL-only image.

Example:

```
v++ -p -f fixed.xsa {L_ij} --package.aie_overlay -o aie.xclbin
```

***Note*:** In DFX designs, AIE overlays are not permitted on top of full AIE+PL images because unloading tears down the `.pdi` image.

--package.aie_resources_bin

```
--package.aie_resources_bin <arg>
```

Where `<arg>` specifies the AIE resources binary file.

IMPORTANT! *Detailed information about this option is not available in the current documentation. Consult the v++ command help or additional Vitis documentation for complete details.*

--package.pl_metadata_only

```
--package.pl_metadata_only
```

Where this option specifies that only PL metadata should be included in the `.xclbin` file, required for PL hardware context for a loaded image. The default behavior is to include full PL configuration data and metadata.

Use Case: Creates a PL metadata-only `.xclbin` file that is required for establishing a PL hardware context for a loaded image. This is useful when the PL configuration has already been loaded and only the metadata is needed for runtime control.

Example:

```
v++ -p -f fixed.xsa --package.pl_metadata_only -o pl-meta.xclbin
```

Combined Use: This option can be combined with `--package.aie_metadata_only` to create a metadata-only `.xclbin` for runtime control of both PL and AIE components of a loaded image:

```
v++ -p --package.pl_metadata_only --package.aie_metadata_only {L_ij:
i=1,...,P, any j} -o pl_aie-metadata.xclbin
```

--package.aie_debug_port

```
--package.aie_debug_port <arg>
```

Where `<arg>` specifies a TCP port in which the emulator listens for incoming connections from the debugger to debug Versal AI Engine cores. The default port value is `10100`.

For example:

```
v++ -l --package.aie_debug_port 1440
```

--package.bl31_elf

```
--package.bl31_elf <arg>
```

Where `<arg>` specifies the absolute or relative path to Arm trusted FW ELF that executes on A72 #0 core. If this option is not specified, then the Vitis compiler searches for the bl31 in the platform.

For example:

```
v++ -l --package.bl31_elf ./arm_trusted.elf
```

--package.boot_mode

```
--package.boot_mode <arg>
```

Where `<arg>` specifies the `<ospi | qspi | sd>` boot mode used for running the application in emulation or on hardware. For embedded platforms, the default boot mode is SD. Custom platforms boot mode can be configured to use QSPI or OSPI as appropriate.

TIP: *The `xilinx_vck190_v202410_1` embedded base platform provided by AMD does not support the QSPI option.*

For example:

```
v++ -l --package.boot_mode sd
```

--package.defer_aie_run

```
--package.defer_aie_run
```

Where this option specifies that the Versal AI Engine cores are enabled by an embedded processor (PS) application. When not specified, the tool generates CDO commands to enable the AI Engine cores during PDI load instead. By default this option is disabled, or FALSE.

For example:

```
v++ -l --package.defer_aie_run
```

--package.domain

```
--package.domain <arg>
```

Where `<arg>` specifies a domain name. If this option is not specified, then the Vitis compiler picks up the default domain from the software platform (SPFM) file. For AI Engine designs, this should always be `aiengine`.

For example:

```
v++ -l --package.domain xrt
```

--package.dtb

```
--package.dtb <arg>
```

Where `<arg>` specifies the absolute or relative path to device tree binary (DTB) used for loading Linux on the APU. If this options is not specified, then Vitis compiler searches for the `dtb` in the platform.

For example:

```
v++ -l --package.dtb ./device_tree.image
```

--package.enable_aie_debug

```
--package.enable_aie_debug
```

When enabled, the tool generates CDO commands to halt the AI Engine cores during the PDI load, forcing them into debug halt mode. Once debugger connected, the user can debug the AI Engine cores step by step. By default this option is disabled, or FALSE.

For example:

```
v++ -l --package.enable_aie_debug
```

--package.image_format

```
--package.image_format <arg>
```

Where `<arg>` specifies `<ext4 | fat32>` output image file format used on the SD card. For embedded platforms with a Linux domain, the default image format is `ext4`. For all others, the image format is `fat32`.

- ext4: Linux file system
- fat32: Windows file system

IMPORTANT! *EXT4 format is not supported on Windows.*

For example:

```
v++ -l --package.image_format fat32
```

--package.kernel_image

```
--package.kernel_image <arg>
```

Where `<arg>` specifies the absolute or relative path to a Linux kernel image file. Overrides the existing image available in the platform. The platform image file is available for download from xilinx.com. Refer to the Vitis Software Platform Installation in the *Vitis Software Platform Release Notes* (UG1742) for more information. If this option is not specified, then the Vitis compiler copies the Linux image from the platform to the SD card folder.

For example:

```
v++ -l --package.kernel_image ./kernel_image
```

--package.no_image

```
--package.no_image
```

Bypass SD card image creation. Valid for `--package.boot_mode sd`. By default this option is disabled, or FALSE.

--package.out_dir

```
--package.out_dir <arg>
```

Where `<arg>` specifies the absolute or relative path to the output directory of the `--package` command. The default output directory is the directory from which the Vitis compiler is launched.

For example:

```
v++ -l --package.out_dir ./out_dir
```

--package.ps_debug_port

```
--package.ps_debug_port <arg>
```

Where `<arg>` specifies the TCP port where emulator listens for incoming connections from the debugger to debug PS cores.

For example:

```
v++ -l --package.debug_port 3200
```

--package.ps_elf

```
--package.ps_elf <arg>
```

Where `<arg>` specifies `<path_to_elf_file,core>`.

- `path_to_elf_file`: Specifies the ELF file for the PS core.
- `core`: Indicates the PS core it should run on.

Used when a baremetal ELF file is running on a device processor core. This option specifies an ELF file and processor core pair to be included in the boot image. The available processors for supported devices are listed below:

- Versal processor core values include: `a72-0`, `a72-1`, `a72-2`, and `a72-3`.
- Zynq UltraScale+ MPSoC processor core values include: `a53-0`, `a53-1`, `a53-2`, `a53-3`, `r5-0`, and `r5-1`.
- Zynq 7000 processor core values include: `a9-0` and `a9-1`.

TIP: *Specify the option separately for each ELF/Core pair.*

For example:

```
v++ -l --package.ps_elf a53_0.elf,a53-0 --package.ps_elf r5_0.elf,r5-0
```

--package.rootfs

```
--package.rootfs <arg>
```

Where `<arg>` specifies the absolute or relative path to a processed Linux root file system file. The platform RootFS file is available for download from Xilinx.com. Refer to the Vitis Software Platform Installation in the *Vitis Software Platform Release Notes* (UG1742) for more information. If this option is not specified, then the Vitis compiler picks up the default `rootfs` path from the software platform (SPFM) file.

For example:

```
v++ -l --package.rootfs ./rootfs.ext4
```

--package.sd_dir

```
--package.sd_dir <arg>
```

Where `<arg>` specifies a folder to package into the `sd_card` directory/image. The contents of the directory are copied to a sub-folder of the `sd_card` folder.

For example:

```
v++ -l --package.sd_dir ./test_data
```

--package.sd_file

```
--package.sd_file <arg>
```

Where `<arg>` specifies an ELF or other data file to package into the `sd_card` directory/image. This option can be used repeatedly to specify multiple files to add to the `sd_card`. The `.xclbin` and `libadf.a` files are automatically copied to the `out-dir` or `sd_card` folder.

For example:

```
v++ -l --package.sd_file ./arm_trusted.elf
```

--package.uboot

```
--package.uboot <arg>
```

Where `<arg>` specifies a path to U-Boot ELF file which overrides a platform U-Boot. If this option is not specified, then the Vitis compiler searches for the `uboot` in the platform.

For example:

```
v++ -l --package.uboot ./uboot.elf
```

--profile Options

As discussed in Enabling Profiling in Your Application in the *Data Center Acceleration using Vitis* (UG1700), there are a number of `--profile` options that let you enable profiling of the application and kernel events during runtime execution. This option enables capturing transaction details for data traffic between the kernel and host, kernel stalls, the execution times of kernels and compute units (CUs), in addition to monitoring activity in Versal AI Engines.

IMPORTANT! *Using the `--profile` option in `v++` also requires the addition of one of the profile or trace options in the `xrt.ini` file. Refer to xrt.ini File for more information.*

The `--profile` commands can be specified in a configuration file under the `[profile]` section head using the following format, for example:

```
[profile]
data=all:all:all            # Monitor data on all kernels and CUs
data=k1:all:all             # Monitor data on all instances of kernel k1
data=k1:cu2:port3           # Specific CU master
data=k1:cu2:port3:counters # Specific CU master (counters only, no trace)
memory=all                  # Monitor transfers for all memories
memory=<sptag>              # Monitor transfers for the specified memory
stall=all:all               # Monitor stalls for all CUs of all kernels
stall=k1:cu2                # Stalls only for cu2
exec=all:all                # Monitor execution times for all CUs
exec=k1:cu2                 # Execution tims only for cu2
aie=all                     # Monitor all AIE streams
aie=DataIn1                 # Monitor the specific input stream in the SDF
graph
aie=M02_AXIS                # Monitor specific stream interface
```

The various options of the command are described below:

--profile.aie:<arg>

Enables profiling of AI Engine streams in adaptive data flow (ADF) applications, where `<arg>` is:

```
<ADF_graph_argument|pin name|all>
```

- `<ADF_graph_argument>`: Specifies an argument name from the ADF graph application.
- `<pin_name>`: Indicates a port on an AI Engine kernel.
- `<all>`: Indicates monitoring all stream connections in the ADF application.

For example, to monitor the `DataIn1` input stream use the following command:

```
v++ --link --profile.aie:DataIn1
```

--profile.aie_trace_offload:<arg>

Enables profiling of AI Engine streams in adaptive data flow (ADF) applications, where `<arg>` is HSDP, or high-speed debug port.

```
v++ --link --profile.aie_trace_offload:HSDP
```

The use of the HSDP is described in *Event Trace Offload using High Speed Debug Port* in *AI Engine Tools and Flows User Guide* (UG1076).

--profile.data:<arg>

Enables monitoring of data ports through monitor IP that are added into the design. This option needs to be specified during linking.

Where `<arg>` is:

```
[<kernel_name>|all]:[<cu_name>|all]:[<interface_name>|all](:[counters|all])
```

- `[<kernel_name>|all]` defines either a specific kernel to apply the command to. However, you can also specify the keyword `all` to apply the monitoring to all existing kernels, compute units, and interfaces with a single option.
- `[<cu_name>|all]` when `<kernel_name>` has been specified, you can also define a specific CU to apply the command to, or indicate that it should be applied to all CUs for the kernel.
- `[<interface_name>|all]` defines the specific interface on the kernel or CU to monitor for data activity, or monitor all interfaces.
- `[<counters|all]` is an optional argument, as it defaults to `all` when not specified. It allows restricting the information gathering to only `counters` for larger designs, while `all` will include the collection of actual trace information.

For example, to assign the data profile to all CUs and interfaces of kernel `k1` use the following command:

```
v++ --link --profile.data k1:all:all
```

--profile.exec:<arg>

This option records the execution times of the kernel and provides minimum port data collection during the system run. This option needs to be specified during linking.

TIP: *The execution time of a kernel is collected by default when `--profile.data` or `--profile.stall` is specified. You can specify `--profile.exec` for any CUs not covered by `data` or `stall`.*

The syntax for `exec` profiling is:

```
[<kernel_name>|all]:[<cu_name>|all](:[counters|all])
```

For example, to profile to execution of `cu2` for kernel `k1` use the following command:

```
v++ --link --profile.exec:k1:cu2
```

--profile.stall:<arg>

IMPORTANT! *This option must be specified during both `v++` compilation and linking.*

Adds stall monitoring logic to the device binary (`.xclbin`) which requires the addition of stall ports on the kernel interface. To facilitate this, the `stall` option must be specified during both compilation and linking.

The syntax for `stall` profiling is:

```
[<kernel_name>|all]:[<cu_name>|all](:[counters|all])
```

For example, to monitor stalls of `cu2` for kernel `k1` use the following command:

```
v++ --compile -k k1 --profile.stall ...
v++ --link --profile.stall:k1:cu2 ...
```

--profile.trace_memory:<arg>

TIP: *This option applies to hardware build targets (`-t=hw`) only, and should not be used for software or hardware emulation flows.*

When building the hardware target (`-t=hw`), use this option to specify the type and amount of memory to use for capturing trace data. You can specify the argument as follows:

```
<FIFO>:<size>|<MEMORY>[<n>][:<SLR>]
```

This argument specifies memory type to use for capturing trace data. Use the `--profile.trace_memory` command to define the type or memory to use, with the `trace_buffer_size` switch in the `xrt.ini` file to define the amount of memory to use as described in xrt.ini File. The default memory type used is the first memory defined in the platform, and the default buffer size is 1 MB.

When `trace_memory` is not specified but `device_trace` is enabled in the xrt.ini File, the profile data is captured to the default platform memory with 1 MB allocated for the trace buffer.

- **FIFO:<size>:** Specified in KB. The maximum is 128K, although 64K is the maximum recommended.
- **Memory:** Specifies the type and number of memory resource on the platform. Memory resources for the target platform can be identified with the `platforminfo` command. Supported memory types include HBM, DDR, PLRAM, HP, ACP, MIG, and MC_NOC. For example, `DDR[1]`.
- **[:<SLR>]:** Optionally indicates that CUs assigned to the specified `<SLR>` should use the DDR or HBM resources specified in the `<MEMORY>` field. Note that this syntax can only be used with DDR or HBM memory banks.

You can specify the `--profile.trace_memory` command with the memory size and unit such as `FIFO:8k`, or specify the memory bank such as `DDR[0]` or `HBM[3]`. In this case, the profile data for all CUs are captured in the specified memory.

Or you can specify the memory to use to capture profile data, and the SLR assignment for that memory. In this case, the SLR assignment indicates that any CUs assigned to the specified SLR should have profile data captured in the specified memory. This is shown in the following config file example:

```
[profile]
trace_memory=DDR[1]:SLR0
trace_memory=DDR[2]:SLR1
```

In the example above, profile data for CUs assigned to SLR0 are captured in DDR bank 1, and CUs assigned to SLR1 are captured in DDR bank 2. CUs are assigned to SLRs using the `--connectivity.slr` command as described in Assigning Compute Units to SLRs on Alveo Accelerator Cards in the *Data Center Acceleration using Vitis* (UG1700).

IMPORTANT! *You cannot mix DDR and HBM memory banks in a single design, and when you specify the <SLR> syntax you must use that syntax for all `trace_memory` commands in the design.*

--vivado Options

The `--vivado.XXX` options are used to configure the Vivado tools for synthesis and implementation of your device binary (`.xclbin`). For instance, you can specify the number of jobs to spawn, LSF commands to use for implementation runs, or the specific implementation strategies to use. You can also configure optimization, placement, timing, or specify which reports to output.

IMPORTANT! *Familiarity with the Vivado Design Suite is required to make the best use of these options. See the Vivado Design Suite User Guide: Implementation (UG904) for more information.*

--vivado.impl.jobs

```
--vivado.impl.jobs <arg>
```

Specifies the number of parallel jobs the Vivado Design Suite uses to implement the device binary. Increasing the number of jobs allows the Vivado implementation step to spawn more parallel processes and complete faster jobs.

For example:

```
v++ --link --vivado.impl.jobs 4
```

--vivado.impl.lsf

```
--vivado.impl.lsf <arg>
```

Specifies the `bsub` command line as a string to pass to an LSF cluster. This option is required to use the IBM Platform Load Sharing Facility (LSF) for Vivado implementation.

For example:

```
v++ --link --vivado.impl.lsf '{bsub -R \"select[type=X86_64]\" -N -q
medium}'
```

--vivado.impl.strategies

```
--vivado.impl.strategies <arg>
```

Specifies a comma-separated list of strategy names for Vivado implementation runs. Use `ALL` to run all available implementation strategies. This lets you run a variety of implementation strategies at the same time during the build process and allows you to more quickly resolve the timing and routing issues of the design.

IMPORTANT! *Running ALL implementation strategies might launch 30 or more runs in the Vivado tool. This can be a tremendous drain on resources, and is not advised. You can prevent this by defining a set of strategies to run, and using a command queue to distribute the process load in some managed way, such as through the `--vivado.impl.jobs` or the `--vivado.impl.lsf` commands.*

--vivado.param

```
--vivado.param <arg>
```

Specifies parameters for the Vivado Design Suite to be used during synthesis and implementation of the FPGA binary (`xclbin`).

TIP: *You can use the `report_param` Tcl command in the Vivado tool to identify the available parameters.*

--vivado.prop

```
--vivado.prop <arg>
```

Specifies properties for the Vivado Design Suite to be used during synthesis and implementation of the FPGA binary (`xclbin`).

Table 14: **Prop Options**

Property Name	Valid Values	Description

`vivado.prop <object_type>.<object_name>.<prop_name>`	Type: Various	This allows you to specify any property used in the Vivado hardware compilation flow.
		`<object_type>` is `run|fileset|file|project`.
		The `<object_name>` and `<prop_name>` values are described in *Vivado Design Suite Properties Reference Guide* (UG912).
		Examples:
		`vivado.prop run.impl_1.{STEPS.PLACE_DESIGN.ARGS.MORE OPTIONS}={-no_bufg_opt}`
		`vivado.prop fileset.current.top=foo`
		If `<object_type>` is set to `file`, `current` is not supported.
		If `<object type>` is set to `run`, the special value of `__KERNEL__` can be used to specify run optimization settings for ALL kernels, instead of the need to specify them one by one.

For example, from the command line:

```
v++ --link --vivado.prop run.impl_1.STEPS.PHYS_OPT_DESIGN.IS_ENABLED=true
--vivado.prop run.impl_1.STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE=Explore
--vivado.prop run.impl_1.STEPS.PLACE_DESIGN.TCL.PRE=/…/xxx.tcl
```

The example above enables the optional PHYS_OPT_DESIGN step as part of the Vivado implementation process, sets the `Explore` directive for that step, and specifies a Tcl script to run before the PLACE_DESIGN step.

TIP: *Each step in the Vivado synthesis and implementation process can have a Tcl prescript to run before the step, and a Tcl postscript to run after the step. This lets you customize the build process by inserting pre-processing or post-processing Tcl commands around the different steps. These scripts can be specified as shown in the example above.*

These options can also be specified in a configuration file under the `[vivado]` section head using the following format:

```
[vivado]
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.IS_ENABLED=true
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE=Explore
prop=run.impl_1.STEPS.PLACE_DESIGN.TCL.PRE=/…/xxx.tcl
```

IMPORTANT! *Some Vivado properties have spaces in their name, such as MORE OPTIONS and Tcl syntax requires these properties to be enclosed in braces, {}. However, the placement of the braces in the `--vivado` options is important. You must surround the complete property name with braces, rather than a portion of it. For instance, the correct placement would be:*

```
--vivado.prop run.impl_1.{STEPS.PLACE_DESIGN.ARGS.MORE OPTIONS}={-
no_bufg_opt}
```

While the following would result in an error during the build process:

```
--vivado.prop "run.impl_1.{STEPS.PLACE_DESIGN.ARGS.MORE OPTIONS}={-
no_bufg_opt}"
```

--vivado.synth.jobs

```
--vivado.synth.jobs <arg>
```

Specifies the number of parallel jobs the Vivado Design Suite uses to synthesize the device binary. Increasing the number of jobs allows the Vivado synthesis to spawn more parallel processes and complete faster jobs.

For example:

```
v++ --link --vivado.synth.jobs 4
```

--vivado.synth.lsf

```
--vivado.synth.lsf <arg>
```

Specifies the `bsub` command line as a string to pass to an LSF cluster. This option is required to use the IBM Platform Load Sharing Facility (LSF) for Vivado synthesis.

For example:

```
v++ --link --vivado.synth.lsf '{bsub -R \"select[type=X86_64]\" -N -q
medium}'
```

Vitis Compiler Configuration File

Configuration files are the recommended way of working with either the Vitis Unified IDE, or the common command-line supported by `v++`. A configuration file provides an organized way of passing options to the tools by grouping similar commands together, creating reusable configuration files to perform specific tasks like connectivity or profiling, and minimizing and simplifying the `v++` command line. Some of the features that can be controlled through config file entries include:

- AI Engine commands to configure component creation using `v++ -c --mode aie`

- HLS commands to configure interface definition, configure synthesis and simulation, and control packaging when using `v++ -c --mode hls`

> **TIP:** *Almost all HLS commands must be specified in a configuration file. The exceptions are `--platform` and `--freqhz`, which are permitted on the command line or in a config file.*

- Connectivity directives for system linking specifying the number of kernels to instantiate, or assigning AI Engine streaming ports to PL kernel ports when using `v++ --link`
- Package directives to configure the creation of boot files, the genration of the .xclbin file from a .xsa, and the creation of an SD card when using `v++ --package`
- Directives for the Vivado Design Suite to manage hardware synthesis and implementation.
- Comments can be added to the configuration file by starting the line with a "#":

```
# This is a comment line
```

The Vitis Unified IDE use configuration files to drive component build and simulation processes. The configuration file can be created by the tool at the time of component creation, can be imported from an existing component, or can be custom created and added to the component separately. For the command-line flow the configuration file is specified through the use of the `v++ --config` option as discussed in the v++ General Options. An example of the `--config` option follows:

```
v++ --link --config ../src/system.cfg
```

Config file commands can be non-composing, which means they are only permitted once such as `--platform`. In this case the commands are read from the config file in the order they are encountered. The first command read is used. Config file commands can also be composing, which means that multiple values can be specified to apply to specific ports or CUs in the design. In this case the commands are cumulative, with any later conflicting commands either ignored or resulting in an error.

Commands are read in the order they are encountered. If the same switch is repeated with conflicting information, the first switch read is used or an error is returned. The order of precedence for switches is as follows, where item one takes highest precedence:

1. Command line switches.
2. Config files as specified on the command line from left-to-right.
3. Within a single config file, precedence is as encountered from top-to-bottom.

In general, any `v++` command option can be specified in a configuration file. However, the configuration file supports defining sections containing groups of related commands under command headers to help manage build options and strategies. The following table lists the defined section headers.

TIP: *Multiple processes can be defined in a single configuration file. However, it is recommended to keep separate configuration files for separate processes, such as HLS component creation, system linking, and system packaging.*

Table 15: **Section Tags of the Configuration File**

Section Name	Description
Unlabeled	Generally, an unlabelled section can be placed at the top of a configuration file to contain commands that are not specific to one of the following section heads. Examples of these command can include `--part` or `--platform`, `--freqhz`, and `--debug`.
[advanced]	--advanced Options
[aie]	Used for AI Engine compilation with the `v++ -c --mode aie` command
[clock]	--clock Options
[connectivity]	--connectivity Options
[debug]	--debug Options
[hls]	Used for HLS component compilation using the `v++ -c --mode hls` command. These commands are described in v++ Mode HLS
[linkhook]	--linkhook Options
[package]	--package Options
[profile]	--profile Options
[vivado]	--vivado Options:

Using the Message Rule File

The `v++` command executes various AMD tools during kernel compilation and linking. These tools generate many messages that provide build status to you. These messages might or might not be relevant to you depending on your focus and design phase. The Message Rule file (`.mrf`) can be used to better manage these messages. It provides commands to promote important messages to the terminal or suppress unimportant ones. This helps you better understand the kernel build result and explore methods to optimize the kernel.

The Message Rule file is a text file consisting of comments and supported commands. Only one command is allowed on each line.

Comment

Any line with "`#`" as the first non-white space character is a comment.

Supported Commands

By default, `v++` recursively scans the entire working directory and promotes all error messages to the `v++` output. The `promote` and `suppress` commands below provide more control on the `v++` output.

- `promote`: This command indicates that matching messages should be promoted to the `v++` output.

- `suppress`: This command indicates that matching messages should be suppressed or filtered from the `v++` output. Errors cannot be suppressed.

Enter only one command per line.

Command Options

The Message Rule file can have multiple `promote` and `suppress` commands. Each command can have one and only one of the options below. The options are case-sensitive.

- `-id [<message_id>]`: All messages matching the specified message ID are promoted or suppressed. The message ID is in format of nnn-mmm. As an example, the following is a warning message from HLS. The message ID in this case is 204-68.

  ```
  WARNING: [V++ 204-68] Unable to enforce a carried dependence constraint
  (II = 1, distance = 1, offset = 1)
  between bus request on port 'gmem'
  (/matrix_multiply_cl_kernel/mmult1.cl:57) and bus request on port 'gmem'-
  severity [severity_level]
  ```

  For example, to suppress messages with message ID 204-68, specify the following: `suppress -id 204-68`.

- `-severity [<severity_level>]`: The following are valid values for the severity level. All messages matching the specified severity level will be promoted or suppressed.
  - `info`
  - `warning`
  - `critical_warning`

    For example, to promote messages with severity of 'critical-warning', specify the following: `promote -severity critical_warning`.

Precedence of Message Rules

The `suppress` rules take precedence over `promote` rules. If the same message ID or severity level is passed to both `promote` and `suppress` commands in the Message Rule file, the matching messages are suppressed and not displayed.

Example of Message Rule File

The following is an example of a valid Message Rule file:

```
# promote all warning, critical warning
promote -severity warning
promote -severity critical_warning
# suppress the critical warning message with id 19-2342
suppress -id 19-2342
```

emconfigutil Utility

When running software or hardware emulation in the command line flow, it is necessary to create an emulation configuration file, `emconfig.json`, used by the runtime library during emulation. The emulation configuration file defines the device type and quantity of devices to emulate for the specified platform. A single `emconfig.json` file can be used for both software and hardware emulation.

***Note*:** When running on real hardware, the runtime and drivers query the installed hardware to determine the device type and quantity installed, along with the device characteristics.

To use the `emconfigutil` utility to automate the creation of the emulation file, specify the target platform and additional options in the `emconfigutil` command line:

```
emconfigutil --platform <platform_name> [options]
```

At a minimum, you must specify the target platform through the `-f` or `--platform` option to generate the required `emconfig.json` file. The specified platform must be the same as specified during the host and hardware builds.

The `emconfigutil` options are provided in the following table.

Table 16: **emconfigutil Options**

Option	Valid Values	Description
`-f` or `--platform`	Target device	Required. Defines the target device from the specified platform.
`--nd`	Any positive integer	Optional. Specifies number of devices. The default is 1.
`--od`	Valid directory	Optional. Specifies the output directory. When running emulation, the `emconfig.json` file must be in the same directory as the host executable. The default is to write the output in the current directory.
`-s` or `--save-temps`	N/A	Optional. Specifies that intermediate files are not deleted and remain after the command is executed. The default is to remove temporary files.
`--xp`	Valid AMD parameters and properties.	Optional. Specifies additional parameters and properties. For example: `--xp prop:solution.platform_repo_paths=<xsa_path>` This example sets the search path for the target platforms.
`-h` or `--help`	N/A	Prints command help.

The `emconfigutil` command generates the `emconfig.json` configuration file in the output directory or the current working directory.

TIP: *When running emulation, the location of the `emconfig.json` file can be specified by the `$EMCONFIG_PATH` variable, or must be found in the same directory as the host executable.*

The following example creates a configuration file targeting two `xilinx_u200_gen3x16_xdma_2_202110_1` devices.

```
$emconfigutil --xilinx_u200_gen3x16_xdma_2_202110_1 --nd 2
```

kernelinfo Utility

The `kernelinfo` utility extracts and displays information from Xilinx object (XO) files which can be used during host code development. This information includes hardware function names, arguments, offsets, and port data.

The following command options are available:

Table 17: **kernelinfo Commands**

Option	Description
`-h [ --help ]`	Print help message.
`-x [ --xo_path ] <arg>`	Absolute path to file including file name and `.xo` extension.
`-l [ --log ] <arg>`	By default, information is displayed on the screen. Otherwise, you can use the `--log` option to output the information as a file.
`-j [ --json ]`	Output the file in JSON format.
`[input_file]`	XO file. Specify the XO file positionally or use the `--xo_path` option.
`[output_file]`	Output from AMD v++ Compiler. Specify the output file positionally, or use the `--log` option.

To run the `kernelinfo` utility, enter the following in a Linux terminal:

```
kernelinfo <filename.o>
```

The output is divided into three sections:

- Kernel Definitions
- Arguments
- Ports

The report generated by the following command is reviewed to help better understand the report content. The report is broken down into specific sections for better understandability.

```
kernelinfo krnl_vadd.xo
```

Where `krnl_vadd.xo` is a compiled kernel.

Kernel Definition

Reports high-level kernel definition information. Importantly, for the host code development, the kernel name is given in the `name` field. In this example, the kernel name is `krnl_vadd`.

```
=== Kernel Definition ===
name: krnl_vadd
language: c
vlnv: xilinx.com:hls:krnl_vadd:1.0
preferredWorkGroupSizeMultiple: 1
workGroupSize: 1
debug: true
containsDebugDir: 1
sourceFile: krnl_vadd/cpu_sources/krnl_vadd.cpp
```

Arguments

Reports kernel function arguments.

In the following example, there are four arguments: `a`, `b`, `c`, and `n_elements`.

```
=== Arg ===
name: a
addressQualifier: 1
id: 0
port: M_AXI_GMEM
size: 0x8
offset: 0x10
hostOffset: 0x0
hostSize: 0x8
type: int*

=== Arg ===
name: b
addressQualifier: 1
id: 1
port: M_AXI_GMEM
size: 0x8
offset: 0x1C
hostOffset: 0x0
hostSize: 0x8
type: int*

=== Arg ===
name: c
addressQualifier: 1
id: 2
port: M_AXI_GMEM1
size: 0x8
offset: 0x28
hostOffset: 0x0
hostSize: 0x8
type: int*

=== Arg ===
name: n_elements
addressQualifier: 0
id: 3
```

```
port: S_AXI_CONTROL
size: 0x4
offset: 0x34
hostOffset: 0x0
hostSize: 0x4
type: int const
```

Ports

Reports the memory and control ports used by the kernel.

```
=== Port ===
name: M_AXI_GMEM
mode: master
range: 0xFFFFFFFF
dataWidth: 32
portType: addressable
base: 0x0

=== Port ===
name: M_AXI_GMEM1
mode: master
range: 0xFFFFFFFF
dataWidth: 32
portType: addressable
base: 0x0

=== Port ===
name: S_AXI_CONTROL
mode: slave
range: 0x1000
dataWidth: 32
portType: addressable
base: 0x0
```

launch_emulator Utility

For embedded platforms that have an Arm® subsystem, the AMD Vitis™ tool uses QEMU to emulate the PS subsystem. The QEMU processes must be run along with the RTL simulator process to emulate the entire system in hardware emulation. The `launch_emulator.py` is a utility which launches QEMU and manages the synchronization of the PL simulator processes. It launches QEMU and the simulation process with provided arguments. The Vitis IDE also calls this command when starting and stopping the emulator.

TIP: *For help inside QEMU, press **Ctrl + a h** while in the emulator shell. To terminate the QEMU command, press **Ctrl + a x** while in the emulator shell.*

For embedded platforms, the v++ Command command generates the `launch_hw_emu.sh` script to call the `launch_emulator.py` command with the required arguments based on the platform and the target application.

You can pass additional arguments to the `launch_emulator` utility from the command line when using the `launch_hw_emu.sh` wrapper script. Simply append the option to the command line when running the script. This allows you to customize the `launch_emulator` utility as needed to support your specific platform or application.

The following table shows the list of available options.

Table 18: **Common Options for launch_emulator**

Option	Accepted Value	Description
`-add-env ADD_ENV_CMD`	N/A	Specify additional Environment Variables for the emulation shell.
`-aie-sim-options`	AIE_SIM_OPTIONS file	Points to an AI Engine sim options file that has various AI Engine debug flags that are required for debugging the AI Engine SystemC module. Refer to the section on Reusing AI Engine Simulator Options in the *AI Engine Tools and Flows User Guide* (UG1076) for more information. The options file should be specified with a relative path with respect to `package.hw_emu/sim/behav_waveform/xsim/`. **TIP:** *This is optional and only applies to AI Engine designs.* You can enable profiling options that were used during `aiesimulator` to be applied to hardware emulation of the system-level design. To do this, add the following command: `-aie-sim-options ../aiesimulator_output/aiesim_options.txt`
`-enable-debug`	N/A	Debug mode opening two different XTERMs for QEMU and PL. **IMPORTANT!** *This is very useful for the batch mode users to understand the flow and handshake between the QEMU and PL process.*
`-graphic-qemu`	N/A	Start the Quick Emulator(QEMU) in GUI mode
`-help`	N/A	Prints help message.
`-run-app`	<application_script_name>	Ensure that the application script is packaged using `--package.sd_file` option during package step. Only when it is packaged in `sd_card`, the application script is available to run after QEMU is up running and mounted. **TIP:** *When using the `-run-app` option, all QEMU messages are initially written to a file called `qemu_output.log` inside the `package.hw_emu`, and then re-written to the console after some delay. You can examine the contents of the `qemu_output.log` if you need to verify whether any problems incurred.*

Table 18: **Common Options for launch_emulator** *(cont'd)*

Option	Accepted Value	Description
`-timeout`	<n>	Terminates emulation after <n> seconds. The default value when -run-app is used is 4000 seconds. This means the application terminates after running for 4000 seconds without user intervention.
`-user-post-sim-script`	Path to Tcl script required to be done post simulation before quit	Creates Tcl for any post operations into a Tcl file and pass the Tcl script to this switch.
`-user-pre-sim-script`	Path to Tcl script	For the first run, launch the script `launch_emulator.py` in GUI mode and add the signals that you want to observe. Copies the commands from the Tcl console and save into a Tcl script. From the next run, pass the Tcl script in batch mode, `launch_emulator.py -user-pre-sim-script <path_to_saved_tcl_script>`. Only supports the Vivado simulator (xsim).
`-verbose`	N/A	Enables additional debug messages
`-wcfg-file-path`	N/A	Specifies the wcfg file created by the XSIM to open during GUI simulation. Requires complete absolute path of the file.
`-wdb-File`	Path to WDB file	Specifies the wdb file to load. Requires complete absolute path of the file.
`-xtlm-aximm-log`	N/A	This switch generates xTLM AXI4 transaction logs for interface connection between two SystemC models (with information like address/data/size, etc). While running the emulation log is available at (directory structure can vary based on `v++` options and simulator used): `package.hw_emu/sim/behav_waveform/xsim/xsc_report.log`
`-xtlm-axis-log`	N/A	This switch generates xTLM AXI4-Stream transaction logs for interface connection between two SystemC models. While running emulation log is available at (directory structure can vary based on `v++` options and simulator used): `package.hw_emu/sim/behav_waveform/xsim/xsc_report.log`

Table 19: **Advanced Options for launch_emulator**

Option	Accepted Value	Description
`-disable-host-completion-check`	N/A	Skips the check for host/test completion. Generally used in applications where python scripts check for the test completion status PASS/FAIL. By default, search for "TEST PASSED" string when `-run-app` switch is used.
`-enable-tcp-sockets`	N/A	Enables TCP Sockets
`-kill`	<pid>	Kills a specified emulator process.
`-kill-pid-file`	N/A	Specifies the file to be used to kill the process. This file stores the group PID of the process. Can be created using -pid-file.

Table 19: **Advanced Options for launch_emulator** *(cont'd)*

Option	Accepted Value	Description
`-no-reboot`	N/A	Exits QEMU instead of rebooting. Used to exit gracefully from QEMU by executing command `reboot -f` at the embedded Linux prompt.
`-no_build`	N/A	Enables a check of the build command without running the build process.
`-no_run`	N/A	Builds but does not run the emulation.
`-ospi-image`	OSPI Image file	Specifies an OSPI image file for booting.
`-pl-sim-args`	Arguments to simulator	These arguments are appended to the simulator command line. They are an alternative to `pm-sim-args-file`.
`-pmc-args`	Arguments to PMC	The PMC/PMU is emulated by `qemu-system-microblazeel`. The most common command line switches of the PMC are captured in `pmc_args.txt`. Instead of writing to a file called `pmc_args.txt`, you can directly provide all the arguments that need to be appended to the PMC command line. This is an alternative to `-pmc-args-file`. PMC/PMU arguments for specific devices can be found in Versal PS and PMC Arguments for QEMU and Zynq UltraScale+ MPSoC PS and PMU Arguments for QEMU. **TIP:** *This option is not supported for AMD Zynq™ 7000 devices.*
`-pmc-args-file`	PMC QEMU arguments file name	Any option to be passed to PMU/PMC can be provided in this file. The specific format is determined by the base file on your chosen platform. This is auto passed in the `v++` package generated script. PMC/PMU arguments for specific devices can be found in Versal PS and PMC Arguments for QEMU and Zynq UltraScale+ MPSoC PS and PMU Arguments for QEMU. **TIP:** *This option is not supported for Zynq 7000 devices.*
`-print-qemu-version`	N/A	Prints the version of QEMU being used.
`-qemu-args`	Arguments to QEMU	The PS is emulated by `qemu-system-aarch64`. The most common command line switches of the PS are captured in `qemu_args.txt`. Instead of writing into a file called `qemu_args.txt`, you can directly provide all the arguments that need to be appended to the QEMU command line. This is an alternative to `qemu-args-file`. PS arguments for specific devices can be found in Versal PS and PMC Arguments for QEMU and following sections for AMD Zynq™ UltraScale+™ MPSoC.

Table 19: **Advanced Options for launch_emulator** *(cont'd)*

Option	Accepted Value	Description
`-qemu-dtb`	<path_to_DTB_file>	`v++ --package` automatically creates a DTB file based on the addressing in the design and passes it to the `launch_emulator` command. This option lets you specify the DTB file to override the defaults. ***Note***: Ensure the DTB is compatible with the `noc_memory_config.txt` file used.
`-qspi-high-image`	Specify QSPI high image file	The image file which is passed as a QEMU argument in the form of boot mode. This is auto passed in the `V++` package generated script. Required only when DUAL QSPI mode is used.
`-qspi-image`	Specify `qspi.bin`	The image file is passed as a QEMU argument in the form of boot mode. This is auto passed in the `V++` package generated script. Required only when you opt for QSPI mode.
`-qspi-low-image`	Specify QSPI low image file	The image file is passed as a QEMU argument in the form of boot mode. This is auto passed in the `V++` package generated script. Required only when DUAL QSPI mode is used.
`-result-string`	N/A	Result string searches for the status of test completion. Default = "TEST PASSED."
`-use-qemu-version-v4`	N/A	Use QEMU version 4.2

Table 20: **Auto-Populated by V++ in Emulation Script**

Option	Accepted Value	Description
`-aie-sim-config`	N/A	Points to the AI Engine sim config file that provides various AI Engine files required for the SystemC Model of AI Engine. This is auto passed by the `v++` package. Required for AI Engine designs.
`-boot-bh`	Path to BH file	Specify the boot BH file path
`-device-family`	7Series | UltraScale | Versal	Required to specify the device family for the platform. This is auto passed by the `v++` package generated script `launch_hw_emu.sh`. This needs to be passed manually for direct usage of the `launch_emulator` command.
`-enable-prep-target`	N/A	Enable pre-process target.
`-forward-port`	<target> <host>	Forwards TCP port from target to host.
`-gdb-port`	Port number	QEMU waits for GDB connection on <port>.
`-noc-memory-config <path/to/noc_memory_config.txt>`	N/A	By default, `v++ --package` creates the NoC memory configuration based on the design configuration, and you can see this file parallel to simulation binaries. You can override this file by replacing the file specified in the simulation binary folder. Use the `-user-pre-sim-script` option to copy your `noc_memory_config.txt` file to the simulation binary area and to get the configuration applied.

Table 20: **Auto-Populated by V++ in Emulation Script** *(cont'd)*

Option	Accepted Value	Description
`-pid-file`	File name	Write process ID to <file> for later use with `-kill`. Used by the Vitis software platform to kill when emulation is successful.
`-pl-sim-dir`	Simulation directory	Start the Programmable Logic Simulator by launching the scripts from this directory. This is auto passed in the `v++` package generated script. The tool expects a file called `simulate.sh` in the specified directory and will execute it to launch the PL simulator (for example, XSIM).
`-pl-sim-script`	Simulation script location	Advanced users can have one direct script to launch simulation (for example, Vivado users). When this option is given, run the script, other options are of no value.
`-platform-name`	NAME	The platform name
`-pmc-args-file`	PMC QEMU arguments file name	Any options to be passed to PMU/PMC can be given in this file. The specific format is determined by the base file on your chosen platform. This is auto passed in the `v++` package generated script. PMC/PMU arguments for specific devices can be found in Versal PS and PMC Arguments for QEMU and Zynq UltraScale+ MPSoC PS and PMU Arguments for QEMU. **TIP:** *This option is not supported for Zynq 7000 devices.*
`-pmc-dtb`	<path_to_DTB_file>	`v++ --package` automatically creates a device-tree binary (DTB) file based on the addressing in the design and passes it to the `launch_emulator` command. This option lets you specify the DTB file to override the defaults. ***Note*:** Ensure the DTB is compatible with the `noc_memory_config.txt` file used. PMC/PMU arguments for specific devices can be found in Versal PS and PMC Arguments for QEMU and Zynq UltraScale+ MPSoC PS and PMU Arguments for QEMU. **TIP:** *This option is not supported for Zynq 7000 devices.*
`-protoinst-File`	Path to ProtoInst file	Specify the protoinst file to load. Provide the complete absolute path.
`-qemu-args-file`	PS QEMU Arguments file name	Any options to be passed to QEMU can be given in this file. This is specific format where you fetch the base file from the platform chosen. This is auto passed in the `v++` package generated script.

Table 20: **Auto-Populated by V++ in Emulation Script** *(cont'd)*

Option	Accepted Value	Description
`-qemu-dtb`	<path_to_DTB_file>	`v++ --package` automatically creates a DTB file based on the addressing in the design and passes it to the `launch_emulator` command. This option lets you specify the DTB file to override the defaults. ***Note***: Ensure the DTB is compatible with the `noc_memory_config.txt` file used.
`-sd-card-image`	Specify `sd_card.img`	The image file is passed as a QEMU argument in the form of boot mode. This is auto passed in the `V++` package generated script. Required only when SD mode is used.
`-t` | `-target`	`hw_emu`	`v++` package generates the hardware emulation script `launch_hw_emu.sh`.
`-xtlm-log-state`	WAVEFORM | LOG | BOTH	Option to specify what the XTLM Log should contain. It can contain the waveform, the text log, or both. This option is used only for hardware emulation.

Versal PS and PMC Arguments for QEMU

In the AMD Versal™ device, the PS(a72) is emulated by `qemu-system-aarch64` and PMC is emulated by `qemu-system-microblazeel`. Most common command line switches of PS are captured in `qemu_args.txt` and PMC command line switches are captured in `pmc_args.txt`.

TIP: *You can add comments to the `pmc_args.txt`, `qemu_args.txt`, and `pmu_args.txt` files using the '#' symbol at the start of the line.*

Table 21: **Versal Options for qemu_args.txt**

Arg Name	Value	Description	Source	How to Extract the Info
-boot	mode=<boot-number> Ex. for sd1 `-boot mode=5`	Specify boot mode on your platform: • qspi24 = 1 • qspi32 = 2 • sd0 = 3 • sd1 = 5 • emmc0 = 6 • ospi = 8	v++ -p	DRC needed between CIPS enabled boot modes and v++ -p selection
-display	none	By default QEMU creates display for user I/O. This option disables the display	Static	Specify none to disable the display

Table 21: **Versal Options for qemu_args.txt** *(cont'd)*

Arg Name	Value	Description	Source	How to Extract the Info
-hw-dtb	<ps-dtb-file>	Device tree file which describes the PS (a72). The Vitis compiler `--package` command generates the `dtb` file and appends it to `qemu-args.txt`.	v++ -p	
-M	arm-generic-fdt	This specifies the QEMU machine to create. `arm-generic-fdt` machine option tells QEMU to parse `dtb` for machine generation, passes by `-hw-dtb user.dtb`.	Device specific	Hard coded for Versal

-serial	-serial null -serial null -serial mon:stdio	By default, QEMU connects invoking terminal to UART0 for user I/O operations. You can override this behavior by specifying this option. Versal platforms have four UARTs specified using positional arguments: the first two are for debug and the last two are UART0 and UART1.
		To connect UART0 to the terminal, specify `-serial null -serial null -serial mon:stdio` which ignores debug related UARTS and connects to UART0 to terminal.
		Similarly to connect only UART1 to terminal specify `-serial null -serial null -serial null -serial mon:stdio`
			Based on UARTs enabled on CIPS configuration.	The Versal device has four UARTs. The first two are debug related UARTs.
				When enabling UART0:
				`CONFIG.PS_UART0_PERIPHERAL_ENABLE = 1`
				`CONFIG.PS_UART1_PERIPHERAL_ENABLE =0 or 1`
				Then specify `-serial null -serial null -serial mon:stdio`
				When enabling only UART1:
				`CONFIG.PS_UART0_PERIPHERAL_ENABLE = 0`
				`CONFIG.PS_UART1_PERIPHERAL_ENABLE = 1`
				Then specify: `-serial null -serial null -serial null -serial mon:stdio`

-sync-quantum	Time in milliseconds	Specifies how frequently QEMU will sync with the RTL simulator. Modifying this can have an impact on the speed of simulation.	Static	Hard coded for Versal devices (user need to change)

Versal CIPS has two Ethernet interfaces. Most of AMD Versal CIPS board has eth0 enabled. If no `-net` or `-netdev` is specified then QEMU by default enables eth0 and maps to user mode backend.

Table 22: **Versal Options for pmc_args.txt**

Switch Name	Value	Description	Source of the Config	How to Extract the Info
-M	microblaze-fdt	Specifies the QEMU machine to create. QEMU creates MicroBlaze™ with nodes from `dtb`.	Static	Hard coded for Versal Devices
-display	none	By default, QEMU creates display for user I/O. This option instructs QEMU that there is no need for display.	Static	Hard coded for Versal Devices
-device	loader,file=<BOOT_bh.bin>, addr=0xf201e000,force-raw	Specifies Boot header file with load address as `0xF201E000` (`BOOT_bh.bin` is loaded at address `0xF201E000`). This is fixed argument in pmc_args.txt which is processed by v++ -p for final argument which has absolute path of `BOOT_bh.bin` file. `BOOT_bh.bin` is generated by v++ -p from final PDI. `BOOT_bh.bin` is directly loaded onto QEMU because there is no BOOT ROM access for QEMU to load boot header from PDI.	v++ -pack	v++ pack extracts `BOOT_bh.bin` and generates this switch
-device	loader,file=<pmc_cdo.bin>,addr=0xF2000000,force-raw	Specifies `pmc_cdo.bin` with load address as `0xF2000000`. This is fixed argument in `pmc_args.txt`. This is processed by v++ -p for final argument which has absolute path of `pmc_cdo.bin` file.	v++ -pack	v++ pack extracts `pmc_cdo.bin` and generates this switch
-device	loader,file=<plm.bin>,addr=0xF0200000,force-raw	Specifies plm binary firmware with load address as `0xF0200000`. This is a fixed argument in `pmc_args.txt` which is processed by v++ -p for final argument. This has an absolute path of `plm.bin` file. This plm is executed by PPU1 when it is out of reset.	v++ -pack	v++ pack extracts `plm.bin` and generates this switch
-device	loader,addr=0xf0000000,data=0xba020004,data-len=4 -device loader,addr=0xf0000004,data=0xb800fffc,data-len=4	Specifies PPU0 process to be in boot loop. As there is no BOOTROM access for QEMU, PPU0 is put in bootloop which generally loads BOOT header from PDI to memory.	Static	Hard coded for Versal Devices
-device	loader,addr=0xF1110624,data=0x0,data-len=4 -device loader,addr=0xF1110620,data=0x1,data-len=4	Makes PPU1 out of reset and puts in executing mode.	Static	Hard coded for Versal Devices

Table 22: **Versal Options for pmc_args.txt** *(cont'd)*

Switch Name	Value	Description	Source of the Config	How to Extract the Info
-hw-dtb	<ps-dtb-file>	`dtb` file which describes aout PS(a72). v++ pack generates this `dtb` file and appends to `qemu-args.txt`.	v++ pack	v++ pack generates `dtb` files based on DDR config on device

Zynq UltraScale+ MPSoC PS and PMU Arguments for QEMU

The Zynq UltraScale+ MPSoC PS (a53) is emulated by `qemu-system-aarch64` and PMU is emulated by `qemu-system-microblazeel`. Most common command line switches of PS are captured in `qemu_args.txt` and PMC command line switches are captured in `pmu_args.txt`.

TIP: *You can add comments to the `pmc_args.txt`, `qemu_args.txt`, and `pmu_args.txt` files using the '#' symbol at the start of the line.*

Table 23: **Zynq UltraScale+ MPSoC Options for qemu_args.txt**

Switch Name	Value	Description	Source of the Config	How to Extract the Info
-M	arm-generic-fdt	This specifies the QEMU machine to create. `arm-generic-fdt` machine option tells QEMU to parse `dtb` for machine generation, passes by `-hw-dtb user.dtb`.	Static	Hard-coded for Zynq UltraScale+ MPSoC devices

Table 23: **Zynq UltraScale+ MPSoC Options for qemu_args.txt** *(cont'd)*

Switch Name	Value	Description	Source of the Config	How to Extract the Info

-serial	mon:stdio	-serial is a positional argument. Redirect the serial port to specified char dev (i.e., stdio, tcp port, file, etc.).	Based on UART configuration on Zynq UltraScale+ MPSoC	Zynq UltraScale+ MPSoC has two UARTs.
				When enabling UART0:
				`CONFIG.PSU__UART0__PERIPHERAL__ENABLE = 1`
				`CONFIG.PSU__UART1__PERIPHERAL__ENABLE = 0 or 1`
				Then specify: `-serial mon:stdio`
				When enabling only UART1:
				`CONFIG.PSU__UART0__PERIPHERAL__ENABLE = 0`
				`CONFIG.PSU__UART1__PERIPHERAL__ENABLE = 1`
				Then specify: `-serial null -serial mon:stdio`

-global	xlnx,zynqmp-boot.cpu-num=0	Make the specified CPU come out of reset.	Static	Hard coded for Zynq UltraScale+ MPSoC devices

Table 23: **Zynq UltraScale+ MPSoC Options for qemu_args.txt** *(cont'd)*

Switch Name	Value	Description	Source of the Config	How to Extract the Info

-net	-net nic -net nic -net nic -net nic -net user	`-net` is positional argument. Initialize network interfaces gem3. Connect the specified network adapter to user mode network.
		TIP: *`-net none` will disable all Ethernet interfaces.*
			Static	Based on Ethernet configurations:
				If gem0(eth0) is enabled:
				`CONFIG.PSU__ENET0__PERIPHERAL__ENABLE =1`
				Then specify `-net nic -net user`
				If gem1 is enabled:
				`CONFIG.PSU__ENET1__PERIPHERAL__ENABLE = 1`
				Then specify `-net nic -net nic -net user`
				If gem2 is enabled:
				`CONFIG.PSU__ENET2__PERIPHERAL__ENABLE = 1`
				Then specify `-net nic -net nic -net nic -net user`
				If gem 3 is enabled:
				`CONFIG.PSU__ENET3__PERIPHERAL__ENABLE = 1`
				Then specify `-net nic -net nic -net nic -net nic -net user`
				TIP: *If no `-net` (and/or `-netdev`) is mentioned then by default QEMU will enable the first Ethernet (gem0) and map it to user mode backend.*

-m	4G	Enabling 4 GB DDR on Zynq UltraScale+ MPSoC.	Static	Emulating full DDR on Zynq UltraScale+ MPSoC
-device	loader,file=<bl31.elf>,cpu-num=0	Load `bl31.elf` file on A53 core 0.	Static	`v++ --package` should replace `bl31.elf` with absolute path of `bl31.elf`

Table 23: **Zynq UltraScale+ MPSoC Options for qemu_args.txt** *(cont'd)*

Switch Name	Value	Description	Source of the Config	How to Extract the Info
-device	loader,file=<u-boot.elf>	Loading `u-boot.elf`.	Static	`v++ --package` should replace `bl31.elf` with absolute path of `u-boot.elf`
-hw-dtb	<ps-dtb-file>	`dtb` file which describes PS which is emulated by QEMU can be specified using `-hw-dtb`.	Static	Hard coded for Zynq UltraScale+ MPSoC devices: `<ps-dtb-file>=/proj/xbuilds/HEAD_daily_latest/installs/lin64/Vitis/HEAD//data/emulation/dtbs/zynqmp/zynqmp-arm-cosim.dtb`

Table 24: **Zynq UltraScale+ MPSoC Options for pmu_args.txt**

Switch Name	Value	Description	Source of the Config	How to Extract the Info
-M	microblaze-fdt	This specifies the QEMU machine to create. microblaze-fdt tells QEMU to parse `dtb` for machine generation, passes by `-hw-dtb user.dtb`.	Static	Hard coded for Zynq UltraScale+ MPSoC devices
-device	loader,file=<pmufw.elf>	Load `pmufw.elf` file on PMU RAM.	Static	Hard coded for Zynq UltraScale+ MPSoC devices
-machine-path	<path-to-xsim-dir>	Point `-machine-path` to folder to create shared RAM and remote-port sockets.	Static	`launch_emulator` command will set this machine path
-display	none	By default, QEMU creates display for user I/O. This option disables the display.	Static	Hard coded for Zynq UltraScale+ MPSoC devices
-hw-dtb	<pmu-dtb-file>	`dtb` file which describes PMU which is emulated by QEMU can be specified using `-hw-dtb`.	Static	<pmu-dtb-file>=/proj/xbuilds/HEAD_daily_latest/installs/lin64/Vitis/HEAD//data/emulation/dtbs/zynqmp/zynqmp-pmu.dtb

TIP: *Although the file is called `pmu_args.txt` here, the file is specified for `launch_emulator` using the `-pmc-args-file` command.*

Zynq 7000 PS Arguments for QEMU

Zynq 7000 PS(a9) is emulated by `qemu-system-aarch64` QEMU binary. Most common command line switches of PS are captured in `qemu_args.txt`.

TIP: *You can add comments to the `pmc_args.txt`, `qemu_args.txt`, and `pmu_args.txt` files using the '#' symbol at the start of the line.*

Table 25: **Zynq 7000 Options for qemu_args.txt**

Switch Name	Value	Description	Source of the Config	How to Extract the Info
-M	arm-generic-fdt-7series	Indicates the QEMU machine to create. arm-generic-fdt-7series tells QEMU to parse `dtb` for machine generation, passes by `-hw-dtb user.dtb`.	Static	Hard coded for Zynq 7000 devices

-serial	-serial /dev/null -serial mon:stdio	Redirect the serial port to specified char dev (i.e., stdio, tcp port, file, etc.)	Based on the UART configuration of the Zynq IP.	Zynq 7000 has two UARTs.
				When enabling UART0:
				`CONFIG.PCW_UART0_PERIPHERAL_ENABLE = 1`
				`CONFIG.PCW_UART1_PERIPHERAL_ENABLE = 0 or 1`
				Then specify: `-serial mon:stdio`
				When enabling only UART1:
				`CONFIG.PCW_UART1_PERIPHERAL_ENABLE = 1`
				Then specify: `-serial null -serial mon:stdio`

-device	loader,addr=0xf8000008,data=0xDF0D,data-len=4 -device loader,addr=0xf8000140,data=0x00500801,data-len=4 -device loader,addr=0xf800012c,data=0x1ed044d,data-len=4 -device loader,addr=0xf8000108,data=0x0001e008,data-len=4 -device loader,addr=0xF800025C,data=0x00000005,data-len=4 -device loader,addr=0xF8000240,data=0x00000000,data-len=4	Register writes to SLCR block, to set PLL and CLK_CTRL regs (required for Linux).	Static	Hard coded for Zynq 7000 devices
-boot	mode=5	Boot mode 5 is for SD boot.	`v++ -p`	

Table 25: **Zynq 7000 Options for qemu_args.txt** *(cont'd)*

Switch Name	Value	Description	Source of the Config	How to Extract the Info
-kernel	<u-boot.elf>	Guest software to load during boot up.	Static	<u-boot.elf> is replaced with the absolute path of `u-boot.elf` from the target platform
-machine	linux=on	Make QEMU itself a loader of the Linux image.	Static	Hard coded for Zynq 7000 devices

manage_ipcache Utility

To provide better performance during synthesis of kernels in your application designs, the AMD Vitis™ compiler uses an IP cache to store and reuse synthesis results. This lets the build process for the `.xclbin` file avoid having to repeat synthesis for kernels and CUs that have not changed. The IP cache stores the synthesis results and applies them for unchanged kernels in the design.

By default, the IP cache is stored inside the Vitis IDE workspace for a project, or at the level of your builds when running `v++` from the command line. You can customize the location for the IP cache using `--remote_ip_cache` to specify a new location, or disable the use of the IP cache using `--no_ip_cache`. See v++ General Options for information on these options.

The `manage_ipcache` utility is a standalone utility to help you manage the contents of your IP cache repository. It lets you report statistics on the IP cache repository and remove entries based on a variety of criteria.

Table 26: **manage_ipcache Options**

Option	Description
`-c | --cache`	Required. Specifies the IP Cache directory to work on.
`-d | --disk_space <size>`	Delete all but the most recently used entries that fit in the disk space specified in MB.
`-h | --help`	Prints help for the `manage_ipcache` command.
`-k | --keep_top <N>`	Delete all but the top N most recently used entries (N is an integer).
`-o | --outfile <file>`	Report stats for the IP cache to the specified file.
`-p | --purge`	Delete ALL cache entries.
`-r | --report`	Report stats for the IP cache to stdout.
`-u | --unused`	Delete cache entries that have never been used (no cache hits).

The following example reports on the entries of the specified IP cache:

```
manage_ipcache --cache ./ip_cache --report
```

The `manage_ipcache` command returns 0 if successful, or returns -1 if an error occurs.

package_xo Command

Syntax

```
package_xo  -kernel_name <arg> [-force] [-kernel_xml <arg>]
            [-output_kernel_xml <arg>] [-design_xml <arg>]
            [-ip_directory <arg>] [-parent_ip_directory <arg>]
            [-kernel_files <args>] [-kernel_xml_args <args>]
            [-kernel_xml_pipes <args>] [-kernel_xml_connections <args>]
            [-ctrl_protocol <arg>] -xo_path <arg> [-quiet] [-verbose]
```

Description

The `package_xo` command is a Tcl command within the AMD Vivado™ Design Suite. Kernels written in RTL are compiled in the Vivado tool using the `package_xo` command line utility which generates a Xilinx object (XO) file which can subsequently used by the `v++` command, during the linking stage.

Table 27: **Arguments**

Argument	Description
`-kernel_name <arg>`	(Required) Specify the name of the RTL kernel.
`-force`	(Optional) Overwrite an existing XO file if one exists.
`-kernel_xml <arg>`	(Optional) Specify the path to an existing kernel XML file. The Vivado tool will create a `kernel.xml` file for the XO file if one is not specified.
`-output_kernel_xml`	(Optional) Specify the path to write the kernel XML file. The Vivado tool will create a `kernel.xml` file to include in the XO file, and also write it to the specified output file. **TIP:** *You can use this option to generate a `kernel.xml` file which you can edit and use as an input in the `package_xo` command.*
`-design_xml <arg>`	(Optional) Specify the path to an existing design XML file
`-ip_directory <arg>`	(Optional) Specify the path to the packaged IP directory.
`-parent_ip_directory`	(Optional) If the kernel IP directory specified contains multiple IPs, specify a directory path to the parent IP where its `component.xml` is located directly below.
`-kernel_xml_args <args>`	(Optional) Generate the `kernel.xml` with the specified function arguments. Each argument value should use the following format: `{name:addressQualifier:id:port:size:offset:type:memSize}` ***Note***: `memSize` is optional.
`-kernel_xml_pipes <args>`	(Optional) Generate the `kernel.xml` with the specified pipe(s). Each pipe value use the following format: `{name:width:depth}`

Table 27: **Arguments** *(cont'd)*

Argument	Description
`-kernel_xml_connections <args>`	(Optional) Generate the `kernel.xml` file with the specified connections. Each connection value should use the following format: `{srcInst:srcPort:dstInst:dstPort}`
`-ctrl_protocol`	Kernel control protocol as described in PL Kernel Properties in the *Data Center Acceleration using Vitis* (UG1700). Valid values: `ap_ctrl_hs`, `ap_ctrl_chain`, `ap_ctrl_none`, `user_managed`. **TIP:** *The default `ap_ctrl_hs` is written to the `kernel.xml` file when `-ctrl_protocol` is not specified.*
`-xo_path <arg>`	(Required) Specify the path and file name of the compiled object (XO) file.
`-quiet`	(Optional) Execute the command quietly, returning no messages from the command. The command also returns TCL_OK regardless of any errors encountered during execution. ***Note***: Any errors encountered on the command-line, while launching the command, will be returned. Only errors occurring inside the command will be trapped.
`-verbose`	(Optional) Temporarily override any message limits and return all messages from this command. ***Note***: Message limits can be defined with the `set_msg_config` command.

Examples

The following example creates the specified XO file containing an RTL kernel of the specified name using the `ap_ctrl_chain` control protocol, and creates the `kernel.xml` file because one has not been specified:

```
package_xo -xo_path Vadd_A_B.xo -kernel_name Vadd_A_B -ctrl_protocol
ap_ctrl_chain -ip_directory ./ip
```

The following example creates the XO file using the specified `kernel.xml` file:

```
package_xo -xo_path Vadd_A_B.xo -kernel_name Vadd_A_B -kernel_xml
kernel.xml -ip_directory ./ip
```

TIP: *The control protocol will be defined in the specified `kernel.xml` file.*

RTL Kernel XML File

TIP: *The `package_xo` command will create a `kernel.xml` file from the `component.xml` of a packaged IP, so you do not need to manually provide one, or generate one using the RTL Kernel wizard.*

An XML kernel description file, called `kernel.xml`, must be created for each RTL kernel, so that it can be used in the AMD Vitis™ application acceleration development flow. The `kernel.xml` file specifies kernel attributes like the register map and ports needed by the runtime and Vitis tool flows. The following code shows is an example of a `kernel.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<root versionMajor="1" versionMinor="6">
  <kernel name="vitis_kernel_wizard_0" language="ip_c"
      vlnv="mycompany.com:kernel:vitis_kernel_wizard_0:1.0"
      attributes="" preferredWorkGroupSizeMultiple="0" workGroupSize="1"
interrupt="true">
    <ports>
      <port name="s_axi_control" mode="slave" range="0x1000" dataWidth="32"
portType="addressable" base="0x0"/>
      <port name="m00_axi" mode="master" range="0xFFFFFFFFFFFFFFFF"
dataWidth="512" portType="addressable"
          base="0x0"/>
    </ports>
    <args>
      <arg name="axi00_ptr0" addressQualifier="1" id="0" port="m00_axi"
size="0x8" offset="0x010" type="int*"
          hostOffset="0x0" hostSize="0x8"/>
    </args>
  </kernel>
</root>
```

***Note*:** The `kernel.xml` file can be created automatically using the RTL Kernel Wizard to specify the interface specification of your RTL kernel.

The following table describes the format of the `kernel.xml` in detail:

Table 28: **Kernel XML File Content**

Tag	Attribute	Description
`<root>`	`versionMajor`	For the current release of Vitis software platform, set to 1.
	`versionMinor`	For the current release of Vitis software platform, set to 6.

Table 28: **Kernel XML File Content** *(cont'd)*

Tag	Attribute	Description
<kernel>	name	Kernel name
	language	Always set to ip_c for RTL kernels.
	vlnv	Must match the vendor, library, name, and version attributes in the component.xml of an IP. For example, if component.xml has the following tags: <spirit:vendor>xilinx.com</spirit:vendor> <spirit:library>hls</spirit:library> <spirit:name>test_sincos</spirit:name> <spirit:version>1.0</spirit:version> The vlnv attribute in kernel XML must be set to:xilinx.com:hls:test_sincos:1.0
	attributes	Reserved. Set it to empty string: ""
	preferredWorkGroupSizeMultiple	Reserved. Set it to 0.
	workGroupSize	Reserved. Set it to 1.
	interrupt	Set to "true" (interrupt="true") if the RTL kernel has an interrupt, otherwise omit.
	hwControlProtocol	Specifies the control protocol for the RTL kernel. • ap_ctrl_hs: Default control protocol for RTL kernels. • ap_ctrl_chain: Control protocol for chained kernels that support dataflow. Adds ap_continue to the control registers to enable ap_done/ap_continue completion acknowledgment. • ap_ctrl_none: Control protocol (none) applied for data driven kernels. • user_managed: Specifies a kernel that meets the interface requirements for Vitis compiler to link the kernel with other kernels and a target platform, but does not adhere to the requirements for execution management by XRT. Refer to the following for more information: ◦ For data center acceleration: Creating User-Managed RTL Kernels in the *Data Center Acceleration using Vitis* (UG1700) ◦ For embedded design development: Creating User-Managed Kernels in the *Embedded Design Development Using Vitis* (UG1701)

Table 28: **Kernel XML File Content** *(cont'd)*

Tag	Attribute	Description
<port>	name	Specifies the port name. **IMPORTANT!** *The AXI4-Lite interface must be named* `S_AXI_CONTROL`.
	mode	At least one AXI4 master port and one AXI4-Lite slave control port are required. AXI4-Stream ports can be specified to stream data between kernels. • For AXI4 master port, set to "master." • For AXI4 slave port, set to "slave." • For AXI4-Stream master port, set to "write_only." • For AXI4-Stream slave port, set it "read_only."
	range	The range of the address space for the port.
	dataWidth	The width of the data that goes through the port, default is 32-bits.
	portType	Indicate whether or not the port is addressable or streaming. • For AXI4 master and slave ports, set it to "addressable." • For AXI4-Stream ports, set it to "stream."
	base	For AXI4 master and slave ports, set to `0x0`. This tag is not applicable to AXI4-Stream ports.
<arg>	name	Specifies the kernel software argument name.
	addressQualifier	Valid values: 0: Scalar kernel input argument 1: global memory 2: local memory 3: constant memory 4: pipe
	id	Only applicable for AXI4 master and slave ports. The ID needs to be sequential. It is used to determine the order of kernel arguments. Not applicable for AXI4-Stream ports.
	port	Specifies the <port> name to which the `arg` is connected.
	size	Size of the argument in bytes. The default is 4 bytes.
	offset	Indicates the register memory address.
	type	The C data type of the argument. For example, `uint*`, `int*`, or `float*`.
	hostOffset	Reserved. Set to `0x0`.
	hostSize	Size of the argument. The default is 4 bytes.
	memSize	For AXI4-Stream ports, `memSize` sets the depth of the created FIFO. **TIP:** *Not applicable to AXI4 ports.*

Table 28: **Kernel XML File Content** *(cont'd)*

Tag	Attribute	Description
The following tags specify additional tags for AXI4-Stream ports. They do not apply to AXI4 ports.		
<connection>	The connection tag describes the actual connection in hardware, either from the kernel to the FIFO inserted for the PIPE, or from the FIFO to the kernel.	
	srcInst	Specifies the source instance of the connection.
	srcPort	Specifies the port on the source instance for the connection.
	dstInst	Specifies the destination instance of the connection.
	dstPort	Specifies the port on the destination instance of the connection.

platforminfo Utility

The `platforminfo` command line utility reports platform meta-data, including information on interface, clock, valid SLRs, allocated resources and memory. The information is in a structured format. You can reference the information when, for example, allocating kernels to SLRs or memory resources.

The following command options are available to use with `platforminfo`:

Table 29: **platforminfo Commands**

Option	Description
`-f [ --force ]`	Overwrite an existing output file.
`-h [ --help ]`	Print help message and exit.
`-k [ --keys ]`	Get keys for a given platform. Returns a list of JSON paths.
`-l [ --list ]`	List platforms. Searches the user repo paths $PLATFORM_REPO_PATHS and then the install locations to find `.xpfm` files.
`-e [ --extended ]`	List platforms with extended information. Use with '--list'.
`-d [ --hw ] <arg>`	Specify the platform definition (`*.xsa`) on which to operate. The value must be a full path, including file name and `.xsa` extension.
`-s [ --sw ] <arg>`	Specify the software platform definition (`*.spfm`) on which to operate. The value must be a full path, including file name and `.spfm` extension.

`-p [ --platform ] <arg>`	AMD platform definition (`*.xpfm`) on which to operate. The value for `--platform` can be a full path including file name and `.xpfm` extension, as shown in example 1 below. If supplying a file name and `.xpfm` extension without a path, this utility will search only the current working directory. You can also specify just the base name for the platform. When the value is a base name, this utility will search the $PLATFORM_REPO_PATHS, and the install locations, to find a corresponding `.xpfm` file, as shown in example 2 below.
	`Example 1: --platform /opt/xilinx/platforms/xilinx_vck190_base_202520_1/xilinx_vck190_base_202520_1.xpfm`
	`Example 2: --platform xilinx_vck190_base_202520_1`

`-o [ --output ] <arg>`	Specify an output file. Default: STDOUT.

Table 29: **platforminfo Commands** *(cont'd)*

Option	Description

`-j [ --json ] <arg>`	Use JSON format for output. When used with no value, prints the entire platform as JSON. Accepts an optional adjacent value that specifies a JSON path. The JSON path, when valid, fetches a JSON subtree, list, or value.
	
	`Example 1:`
	`platforminfo --json="hardwarePlatform" --platform <platform base name>`
	
	`Example 2: Specify the index when referring to an item in a list.`
	`platforminfo --json="hardwarePlatform.devices[0].name" --platform <platform base name>`
	
	`Example 3: When using the short option form (-j), the value must follow immediately.`
	`platforminfo -j"hardwarePlatform.systemClocks[]" -p <platform base name>`

`-v [ --verbose ]`	The platforminfo utility can generate a default, human-readable report for a given platform. The default report contains the most important characteristics of the platform. Use the verbose switch to display the full platform meta-data in human-readable form.

***Note*:** Except when using the `--help` or `--list` options, a platform must be specified. You can specify the platform using the `--platform` option, or using either `--hw`, `--sw`. You can also simply insert the platform name or full path into the command line positionally.

To understand the generated report, condensed output logs, based on the following command are reviewed. The report is broken down into specific sections for better understandability.

```
platforminfo -p $PLATFORM_REPO_PATHS/
xilinx_u200_gen3x16_xdma_2_202110_1.xpfm
```

TIP: *See Platform info for vek385_base for an example of embedded processor platforms.*

Basic Platform Information

Platform information and high-level description are reported.

```
Platform:           gen3x16_xdma_2
File:               $PLATFORM_REPO_PATHS/
xilinx_u200_gen3x16_xdma_2_202110_1.xpfm
Description:
This platform targets the Alveo U200 Data Center Accelerator Card. This
high-performance acceleration platform features up to four channels of
DDR4-2400 SDRAM which are instantiated as required by the user kernels for
high fabric resource availability, and Xilinx DMA Subsystem for PCI Express
with PCIe Gen3 x16 connectivity.

Hardware:                           1
Has Software Platform(s):           1
Has Hardware Emulation:             1
```

Hardware Platform Information

General information on the hardware platform is reported. For the Hardware Emulation field, a "1" indicates this platform is suitable for these configurations.

```
Vendor:                              xilinx
Board:                               U200 (gen3x16_xdma_2)
Name:                                gen3x16_xdma_2
Version:                             202110.1
Generated Version:                   2021.1
Is Extensible:                         1
Supports Hardware Target:              1
Supports Hardware Emulation Target: 1
Is a Hardware Emulation Platform:    1
FPGA Family:                         virtexuplus
FPGA Device:                         xcu200
Board Vendor:                        xilinx.com
Board Name:                          xilinx.com:au200:1.3
Board Part:                          xcu200-fsgd2104-2-e
```

Interface Information

The following shows the reported PCIe interface information.

```
  Interface Name: PCIe
  Interface Type: gen3x16
  PCIe Vendor Id:
  PCIe Device Id:
  PCIe Subsystem Id:
```

Clock Information

Reports the maximum kernel clock frequencies available. The Clock Index is the reference used in the `--kernel_frequency v++` directive when overriding the default value.

```
Default Clock Index: 0
Clock Index: 0
Frequency: 300.000000
Status: scalable
Clock Index: 1
Frequency: 500.000000
Status: scalable
Clock Index: 2
Frequency: 50.000000
Status: fixed
Clock Index: 3
Frequency: 250.000000
Status: fixed
Clock Index: 4
Frequency: 100.000000
Status: fixed
```

Valid SLRs

Reports the valid SLRs in the platform.

```
SLR0, SLR1, SLR2
```

Resource Availability

The total available resources and resources available per SLR are reported. This information can be used to assess applicability of the platform for the design and help guide allocation of compute unit to available SLRs.

```
=====
Total
=====
  LUTs:  979040
  FFs:   1958080
  BRAMs: 1860
  DSPs:  5880
  URAMs:  800

=======
Per SLR
=======
  SLR0:
    LUTs:  388160
    FFs:   776320
    BRAMs: 720
    URAMs: 320
    DSPs:  2280
  SLR1:
    LUTs:  205440
    FFs:   410880
    BRAMs: 420
    URAMs: 160
    DSPs:  1320
  SLR2:
    LUTs:  385440
    FFs:   770880
    BRAMs: 720
    URAMs: 320
    DSPs:  2280
```

Memory Information

Reports the available DDR and PLRAM memory connections per SLR as shown in the example output below.

```
 Type: ddr4
Bus SP Tag: DDR
  Segment Index: 0
    Consumption: automatic
    SP Tag:      bank0
    SLR:         SLR0
    Max Masters: 15
```

```
    Segment Index: 1
      Consumption: default
      SP Tag:      bank1
      SLR:         SLR1
      Max Masters: 15
    Segment Index: 2
      Consumption: automatic
      SP Tag:      bank2
      SLR:         SLR1
      Max Masters: 15
    Segment Index: 3
      Consumption: automatic
      SP Tag:      bank3
      SLR:         SLR2
      Max Masters: 15
  Bus SP Tag: PLRAM
    Segment Index: 0
      Consumption: explicit
      SLR:         SLR0
      Max Masters: 15
    Segment Index: 1
      Consumption: explicit
      SLR:         SLR1
      Max Masters: 15
    Segment Index: 2
      Consumption: explicit
      SLR:         SLR2
      Max Masters: 15
  Bus SP Tag: HOST
    Segment Index: 0
      Consumption: explicit
      SLR:         SLR2
      Max Masters: 15
```

The `Bus SP Tag` heading can be DDR or PLRAM and gives associated information below.

The `Segment Index` field is used in association with the `SP Tag` to generate the associated memory resource index as shown below.

```
Bus SP Tag[Segment Index]
```

For example, if `Segment Index` is 0, then the associated DDR resource index would be DDR[0].

This memory index is used when specifying memory resources in the `v++` command as shown below:

```
v++ … --connectivity.sp vadd.m_axi_gmem:DDR[3]
```

There can be more than one `Segment Index` associated with an SLR. For instance, in the output above, SLR1 has both `Segment Index` 1 and 2.

The `Consumption` field indicates how a memory resource is used when building the design:

- **default:** If the `--connectivity.sp` directive is not specified, it uses this memory resource by default during `v++` build. For example in the report below, DDR with `Segment Index` 1 is used by default.
- **automatic:** When the maximum number of memory interfaces are used and under `Consumption: default` is fully applied, then the interfaces under `automatic` is used. The maximum number of interfaces per memory resource are given in the **Max Masters** field.
- **explicit:** For PLRAM, consumption is set to `explicit` which indicates this memory resource is only used when explicitly indicated through the `v++` command line.

Software Platform Information

Although software platform information is reported, it is only useful for users that have an OS running on the device, and not applicable to users that use a host machine.

```
Number of Runtimes:            2
Default System Configuration:  config0_0
System Configurations:
  System Config Name:                       config0_0
  System Config Description:                config0_0 Linux OS on x86_0
  System Config Default Processor Group:    x86_0
  System Config Default Boot Image:
  System Config Is QEMU Supported:          0
  System Config Processor Groups:
    Processor Group Name:       x86_0
    Processor Group CPU Type:   x86
    Processor Group m_os Name:    Linux OS
  System Config Boot Images:
Supported Runtimes:
  Runtime: C/C++
  Runtime: XRT
```

Platform info for vek385_base

Use the following command to return the platforminfo for the vek385_base:

```
platforminfo -p $PLATFORM_REPO_PATHS/vek385_base
```

The results returned are as follows:

```
==========================
Basic Platform Information
==========================
Platform:           vek385_base
File:               <path_to_platform>/vek385_base.xpfm
Description:         The VEK385 Vitis base platform is designed for the
VEK385 evaluation kit, enabling developers to create high-performance
applications using AI and DSP engines that deliver over 10× the scalar
compute performance of current server-class CPUs. This platform features
a powerful heterogeneous architecture with 8 Arm® Cortex®-A78 application
processors, 10 Arm® Cortex®-R5 real-time processors, and 20GB of high-
speed LPDDR5X memory. It offers a robust foundation for building advanced
solutions in AI, signal processing, and compute-intensive domains using AMD
```

```
Versal™ adaptive SoCs.
Hardware:                                1
Has Software Platform(s):                1
Supports Hardware Emulation Target:      1
Has Hardware Emulation:                  1

====================================
Hardware Platform (Shell) Information
====================================
Vendor:                              xilinx.com
Board:                               vek385_1_base
Name:                                vek385_1_base
Version:                             1.0
Generated Version:                   2025.2
Is Extensible:                         1
Supports Hardware Target:              1
Is a Hardware Emulation Platform:      0
FPGA Family:                         versal
FPGA Device:                         xc2ve3858
Board Vendor:                        xilinx.com
Board Name:                          xilinx.com:vek385_1:1.0
Board Part:                          xc2ve3858-ssva2112-2MP-e-S

==============
AIE Partitions
==============
       Start Col: 0
       # Columns: 36

======================
Available Resources
======================
NoC PL NSU:          24
NoC PL NMU:          30

=================
Clock Information
=================
  Default Clock Index: 1
  Clock Index:         0
    Frequency:         624.993750
    Status:            fixed_non_ref
  Clock Index:         1
    Frequency:         312.496875
    Status:            fixed_non_ref
  Clock Index:         2
    Frequency:         156.248437
    Status:            fixed_non_ref
  Clock Index:         3
    Frequency:         78.124218
    Status:            fixed_non_ref
  Clock Index:         4
    Frequency:         99.999000
    Status:            fixed

========================
AIE Hardware Information
========================
Arch: AIE2PS
NPI Base Address: 0xf6d50000
AXI Base Address: 0x20000000000
Shim Row Start: 0 # Rows: 1
Core Row Start: 3 # Rows: 4
```

```
Mem Row Start: 1 # Rows: 2

==========
Valid SLRs
==========
SLR0

=====================
Resource Availability
=====================
  =====
  Total
  =====
    LUTs:  542394
    FFs:   1085371
    BRAMs: 1342
    DSPs:  2064

==================
Memory Information
==================
  Bus SP Tag: AIE
  Bus SP Tag: LPDDR
  Bus SP Tag: LPDDR01
  Bus SP Tag: LPDDR23
  Bus SP Tag: LPDDR4

============================
Software Platform Information
============================
Number of Runtimes:            2
Default System Configuration:  vek385_base
System Configurations:
  System Config Name:                       vek385_base
  System Config Description:                 The VEK385 Vitis base platform
is designed for the VEK385 evaluation kit, enabling developers to create
high-performance applications using AI and DSP engines that deliver over
10× the scalar compute performance of current server-class CPUs. This
platform features a powerful heterogeneous architecture with 8 Arm® Cortex®-
A78 application processors, 10 Arm® Cortex®-R5 real-time processors, and
20GB of high-speed LPDDR5X memory. It offers a robust foundation for
building advanced solutions in AI, signal processing, and compute-intensive
domains using AMD Versal™ adaptive SoCs.
  System Config Default Processor Group:   xrt
  System Config Default Boot Image:        standard
  System Config Is QEMU Supported:         1
  System Config Processor Groups:
    Processor Group Name:      aiengine
    Processor Group CPU Type:  ai_engine
    Processor Group m_os Name:   aiengine
    Processor Group Name:      xrt
    Processor Group CPU Type:  cortex-a78
    Processor Group m_os Name:   xrt
  System Config Boot Images:
    Boot Image Name:           standard
    Boot Image Type:
    Boot Image BIF:            boot/
    Boot Image Data:           xrt/image
    Boot Image Boot Mode:
    Boot Image RootFileSystem:
    Boot Image Mount Path:
    Boot Image Read Me:
    Boot Image QEMU Args:      qemu/pmc_args.txt:qemu/qemu_args.txt
```

```
    Boot Image QEMU Boot:
    Boot Image QEMU Dev Tree:
Supported Runtimes:
  Runtime: C/C++
  Runtime: XRT
```

vitis-run Command

The `vitis-run` command is provided to enable C-simulation, C/RTL Co-simulation, and Vivado implementation of an HLS component created with the `v++ -c --mode hls` command.

The options of the `vitis-run` command include:

- **--mode hls:** Specifies the use of `vitis-run` for HLS components. This is the only mode currently supported.
- **--csim:** Run C-simulation on an HLS component.
- **--cosim:** Run C/RTL Co-simulation on an HLS component.
- **--package:** Run the package process to generate the IP or XO files from the RTL design of the HLS component. This generates the required export files to use the RTL design in the Vivado Design Suite, or Vitis development flow.
- **--impl:** Run Vivado implementation out-of-context (OOC) run on the HLS component. This is used to provide resource usage and timing estimates from the synthesized or implemented design.
- **--tcl:** Run Vitis HLS using the Tcl scripting language. Tcl commands are described in the *Vitis High-Level Synthesis User Guide* (UG1399) under the heading of Vitis HLS Command Reference.
- **--work_dir:** Specify the working directory. For `-cosim` and `-impl`, the specified working directory must contain a previously compiled HLS component.
- **--config arg:** Specify a config file for use with the tool. Refer to v++ Mode HLS for specific commands to use in the config file.
- **-h [--help]:** Display command help for the tool.

TIP: *The list of options above is not complete list. You can use the --help command to display the complete list of `vitis-run` commands.*

You can use the `vitis-run` command to run C-simulation without an existing design, or run C/RTL Co-simulation or Vivado implementation on an existing HLS component. Some examples follow.

Run C-Simulation

You can run C-simulation on an existing work directory containing a previously compiled HLS component, or specify a new work directory to run C-simulation on the source files directly. The config file for an existing HLS component only needs to specify commands for C-Simulation as described in C-Simulation Configuration. The previously built HLS component provides the foundation for simulation, defining the source files, test bench files, and part or platform for the design. Running C-simulation on a new work directory requires a complete config file, specifying the required source files and part, in addition to any C-simulation options.

An example command line:

```
vitis-run --mode hls --csim --config ./hls_csim.cfg --work_dir newTest
```

The example config file:

```
part=xcvu11p-flga2577-1-e

[hls]
clock=8
flow_target=vitis
syn.file=../../src/dct.cpp
syn.top=dct
tb.file=../../src/out.golden.dat
tb.file=../../src/in.dat
tb.file=../../src/dct_test.cpp
tb.file=../../src/dct_coeff_table.txt
syn.output.format=xo
clock_uncertainty=15%
csim.O=true
csim.code_analyzer=0
csim.clean=true
csim.profile=true
```

Run C/RTL Co-Simulation

You can run Co-simulation on an existing work directory containing a previously compiled HLS component, or specify a new work directory to run C-simulation on the source files directly. The config file for an existing HLS component only needs to specify commands for C-Simulation as described in Co-Simulation Configuration. The previously built HLS component provides the foundation for simulation, defining the source files, test bench files, and part or platform for the design. Running C-simulation on a new work directory requires a complete config file, specifying the required source files and part, in addition to any C-simulation options.

An example command line:

```
vitis-run --mode hls --cosim --config ./cosim.cfg --work_dir myHLS
```

The example config file:

```
part=xcvu11p-flga2577-1-e

[HLS]
clock=8
flow_target=vitis
syn.file=/group/xcoswmktg/randyh/rigel-tests/03-Vitis_HLS/reference-
files/src/dct.cpp
syn.top=dct
tb.file=/group/xcoswmktg/randyh/rigel-tests/03-Vitis_HLS/reference-
files/src/out.golden.dat
tb.file=/group/xcoswmktg/randyh/rigel-tests/03-Vitis_HLS/reference-
files/src/in.dat
tb.file=/group/xcoswmktg/randyh/rigel-tests/03-Vitis_HLS/reference-
files/src/dct_test.cpp
tb.file=/group/xcoswmktg/randyh/rigel-tests/03-Vitis_HLS/reference-
files/src/dct_coeff_table.txt
syn.output.format=xo
clock_uncertainty=15%
cosim.trace_level=port
#cosim.wave_debug=true
cosim.random_stall=true
cosim.enable_dataflow_profiling=true
```

Run Vivado Implementation

You can run the Vivado Design Suite to synthesize and run place and route on the RTL generated by the HLS synthesis process. The synthesis and implementation processes are managed by commands specified in the configuration file as described in Implementation Configuration.

An example command line:

```
vitis-run --mode hls --impl --config ./impl.cfg --work_dir myHLS
```

The example config file:

```
part=xcvu11p-flga2577-1-e

[HLS]
clock=8
flow_target=vitis
syn.file=/group/xcoswmktg/randyh/rigel-tests/03-Vitis_HLS/reference-
files/src/dct.cpp
syn.top=dct
tb.file=/group/xcoswmktg/randyh/rigel-tests/03-Vitis_HLS/reference-
files/src/out.golden.dat
tb.file=/group/xcoswmktg/randyh/rigel-tests/03-Vitis_HLS/reference-
files/src/in.dat
tb.file=/group/xcoswmktg/randyh/rigel-tests/03-Vitis_HLS/reference-
files/src/dct_test.cpp
tb.file=/group/xcoswmktg/randyh/rigel-tests/03-Vitis_HLS/reference-
files/src/dct_coeff_table.txt
syn.output.format=xo
```

```
clock_uncertainty=15%
cosim.trace_level=port
#cosim.wave_debug=true
cosim.random_stall=true
cosim.enable_dataflow_profiling=true
```

Run Tcl Script

You can also use `vitis-run` to run an existing Tcl script, such as the `script.tcl` from an existing project, to build the project and then write a config file from the Tcl script. The example below shows the `vitis-run` command to perform these actions.

An example command line:

```
vitis-run --mode hls --tcl dct-build.tcl
```

An example Tcl script:

```
open_project dctProj
set_top dct
add_files ../03-Vitis_HLS/reference-files/src/dct.cpp
add_files -tb ../03-Vitis_HLS/reference-files/src/dct_coeff_table.txt
add_files -tb ../03-Vitis_HLS/reference-files/src/dct_test.cpp
add_files -tb ../03-Vitis_HLS/reference-files/src/in.dat
add_files -tb ../03-Vitis_HLS/reference-files/src/out.golden.dat
open_solution "solution1" -flow_target vivado
set_part {xcvu11p-flga2577-1-e}
create_clock -period 10 -name default
csynth_design
write_ini ./dct-build.cfg
exit
```

TIP: *The addition of the `write_ini` command at the end of the script creates a config file from the Tcl script, thus providing you with a config file to use with `v++ -c --mode hls`.*

xrt-smi Utility

The xrt-smi utility is a standalone command line utility that is included with the Xilinx Runtime (XRT) installation package. Details of the `xrt-smi` command can be found at https://xilinx.github.io/XRT/master/html/xrt-smi.html.

`xrt-smi` includes multiple commands to validate and identify the installed accelerator card(s) along with additional card details including on card memory, host interface, target platform name, and system information. This information can be used for both card administration and application debugging.

Accelerator cards are partitioned into a user function and a management function to provide different levels of card access. The user function lets you load and run applications, while the management function is for system administrators to manage the card. The `xrt-smi` utility interacts with the user function. The `xbmgmt` utility, which requires root privilege, is for interacting with the management function. The reason for splitting the function access between the two utilities is to provide some security for the management features of the tool.

You can use the `help` command to list the available `xrt-smi` commands and options:

```
xrt-smi --help
```

Set up the `xrt-smi` command as part of the XRT installation using the following scripts:

- For csh shell:

  ```
  $ source /opt/xilinx/xrt/setup.csh
  ```

- For bash shell:

  ```
  $ source /opt/xilinx/xrt/setup.sh
  ```

xbmgmt Utility

AMD Board Management (`xbmgmt`) utility is a standalone command line tool that is included with the Xilinx Runtime (XRT) installation package. Details of the `xbmgmt` command can be found at https://xilinx.github.io/XRT/master/html/xbmgmt.html.

Accelerator cards are partitioned into a user function and a management function to provide different levels of card access. The user function lets you load and run applications, while the management function is for system administrators to manage the card. The `xrt-smi` utility interacts with the user function.

The `xbmgmt` utility is used for card installation and administration, and requires `sudo` privileges when running it. The `xbmgmt` supported tasks include flashing the card firmware, and scanning the current device configuration.

You can use the `help` command to list the available `xbmgmt` commands and options, and access help for individual commands by using the following:

```
xbmgmt --help <command>
```

For detailed help of each sub-command, use the following:

```
xbmgmt help <subcommand>
```

Set up the `xbmgmt` command as part of the XRT installation using the following scripts:

- For csh shell:

```
$ source /opt/xilinx/xrt/setup.csh
```

- For bash shell:

```
$ source /opt/xilinx/xrt/setup.sh
```

xclbinutil Utility

The `xclbinutil` utility can create, modify, and report `xclbin` content information.

The available command options are shown in the following table.

Table 30: **xclbinutil Commands**

Option	Description
`-h [ --help ]`	Print help messages.
`-i [ --input ]<arg>`	Input file name. Reads `xclbin` into memory.
`-o [ --output ]<arg>`	Output file name. Writes in memory `xclbin` image to a file.
`--target <arg>`	Target flow for this image. Valid values: `hw`, and `hw_emu`.
`----private-key <arg>`	Private key used in signing the `xclbin` image.
`--ceritifcate <arg>`	Certificate used in signing and validating the `xclbin` image.
`--digest-algorithm <arg>`	Digest algorithm. Default: `sha512`
`--validate-signature`	Validates the signature for the given `xclbin` archive.
`-v [ --verbose ]`	Display verbose/debug information
`-q [ --quiet ]`	Minimize reporting information.
`--migrate-forward`	Migrate the `xclbin` archive forward to the new binary format.
`--add-section <arg>`	Section name to add to the `xclbin` image. Format: `<section>:<format>:<file>`
`--add-replace-section <arg>`	Replace an existing section or add the section of the `xclbin` image if it does not exist. Format: `<section>:<format>:<file>`
`--add-merge-section <arg>`	Add the section if it does not exist, or merge content with an existing section. Format: `<section>:<format>:<file>`
`--remove-section<arg>`	Section name to remove from the `xclbin` image.
`--dump-section<arg>`	Section to dump. Format: `<section>:<format>:<file>`
`--replace-section<arg>`	Section to replace.
`--key-value<arg>`	Key value pairs. Format: `[USER|SYS]:<key>:<value>`
`--remove-key<arg>`	Removes the given user key from the `xclbin` archive.
`--add-signature<arg>`	Adds a user defined signature to the given `xclbin` image.
`--remove-signature`	Removes the signature from the `xclbin` image.
`--get-signature`	Returns the user defined signature (if set) of the `xclbin` image.

Table 30: **xclbinutil Commands** *(cont'd)*

Option	Description
`--info`	Report accelerator binary content. Including: generation and packaging data, kernel signatures, connectivity, clocks, sections, etc
`--list-sections`	List all possible section names (standalone option).
`--version`	Version of this executable.
`--force`	Forces a file overwrite.

The following are various use examples of the tool.

- **Reporting xclbin information:**

```
xclbinutil --info --input binary_container_1.xclbin
```

- **Extracting the bitstream image:**

```
xclbinutil --dump-section BITSTREAM:RAW:bitstream.bit --input
binary_container_1.xclbin
```

- **Extracting the build metadata:**

```
xclbinutil --dump-section BUILD_METADATA:HTML:buildMetadata.json --input
binary_container_1.xclbin
```

- **Removing a section:**

```
xclbinutil --remove-section BITSTREAM --input binary_container_1.xclbin --
output binary_container_modified.xclbin
```

For most users, details about the contents and how the `xclbin` was created is desired. This information can be obtained through the `--info` option and reports information on the `xclbin`, hardware platform, clocks, memory configuration, kernel, and how the `xclbin` was generated.

The output of the `xclbinutil` command using the `--info` option is shown below divided into sections.

```
xclbinutil -i binary_container_1.xclbin --info
```

xclbin Information

```
  Generated by:           v++ (2020.1) on Mon Apr 13 20:19:40 MDT 2020
  Version:                2.6.436
  Kernels:                CopyKernel
  Signature:
  Content:                Bitstream
  UUID (xclbin):          d081de98-3fd3-4e9b-bab3-108b42c73101
  UUID (IINTF):           862c7020a250293e32036f19956669e5
```

```
   Sections:                DEBUG_IP_LAYOUT, BITSTREAM, MEM_TOPOLOGY,
IP_LAYOUT,
                            CONNECTIVITY, CLOCK_FREQ_TOPOLOGY,
BUILD_METADATA,
                            EMBEDDED_METADATA, SYSTEM_METADATA,
PARTITION_METADATA
```

Hardware Platform Information

```
Vendor:                  xilinx
Board:                   u200
Name:                    xdma
Version:                 201830.1
Generated Version:       Vivado 2018.3 (SW Build: 2388429)
Created:                 Wed Nov 14 20:06:10 2018
FPGA Device:             xcu200
Board Vendor:            xilinx.com
Board Name:              xilinx.com:au200:1.0
Board Part:              xilinx.com:au200:part0:1.0
Platform VBNV:           xilinx_u200_xdma_201830_1
Static UUID:             00194bb3-707b-49c4-911e-a66899000b6b
Feature ROM TimeStamp:   1542252769
```

Clocks

Reports the maximum kernel clock frequencies available. Both the clock names and clock indexes are provided. The clock indexes are identical to those reported in platforminfo Utility.

```
Name:       DATA_CLK
Index:      0
Type:       DATA
Frequency:  300 MHz

Name:       KERNEL_CLK
Index:      1
Type:       KERNEL
Frequency:  500 MHz
```

Memory Configuration

```
Name:          bank0
Index:         0
Type:          MEM_DDR4
Base Address:  0x0
Address Size:  0x400000000
Bank Used:     No

Name:          bank1
Index:         1
Type:          MEM_DDR4
Base Address:  0x400000000
Address Size:  0x400000000
Bank Used:     Yes

Name:          bank2
Index:         2
```

```
Type:          MEM_DDR4
Base Address:  0x800000000
Address Size:  0x400000000
Bank Used:     No

Name:          bank3
Index:         3
Type:          MEM_DDR4
Base Address:  0xc00000000
Address Size:  0x400000000
Bank Used:     No

Name:          PLRAM[0]
Index:         4
Type:          MEM_DDR4
Base Address:  0x1000000000
Address Size:  0x20000
Bank Used:     No

Name:          PLRAM[1]
Index:         5
Type:          MEM_DRAM
Base Address:  0x1000020000
Address Size:  0x20000
Bank Used:     No

Name:          PLRAM[2]
Index:         6
Type:          MEM_DRAM
Base Address:  0x1000040000
Address Size:  0x20000
Bank Used:     No
```

Kernel Information

For each kernel in the `xclbin`, the function definition, ports, and instance information is reported.

The following is an example of the reported function definition.

```
Definition
----------
   Signature: krnl_vadd (int* a, int* b, int* c,
              int const  n_elements)
```

The following is an example of the reported ports.

```
Ports
-----
   Port:           M_AXI_GMEM
   Mode:           master
   Range (bytes):  0xFFFFFFFF
   Data Width:     32 bits
   Port Type:      addressable

   Port:           M_AXI_GMEM1
   Mode:           master
   Range (bytes):  0xFFFFFFFF
   Data Width:     32 bits
```

```
   Port Type:     addressable

   Port:          S_AXI_CONTROL
   Mode:          slave
   Range (bytes): 0x1000
   Data Width:    32 bits
   Port Type:     addressable
```

The following is an example of the reported instance(s) of the kernel.

```
Instance:        krnl_vadd_1
   Base Address: 0x0

   Argument:          a
   Register Offset:   0x10
   Port:              M_AXI_GMEM
   Memory:            bank1 (MEM_DDR4)

   Argument:          b
   Register Offset:   0x1C
   Port:              M_AXI_GMEM
   Memory:            bank1 (MEM_DDR4)

   Argument:          c
   Register Offset:   0x28
   Port:              M_AXI_GMEM1
   Memory:            bank1 (MEM_DDR4)

   Argument:          n_elements
   Register Offset:   0x34
   Port:              S_AXI_CONTROL
   Memory:            <not applicable>
```

Tool Generation Information

The utility also reports the `v++` command line used to generate the `xclbin`. The Command Line section gives the actual `v++` command line used, while the Options section displays each option used in the command line, but in a more readable format with one option per line.

```
$ xclbinutil --input build_dir.hw_emu.xilinx_u250_gen3x16_xdma_4_1_202210_1/
vadd.xclbin --info
XRT Build Version: 2.19.185 (2025.2)
       Build Date: 2025-04-27 06:39:04
          Hash ID: 6d79c5f0e8c3f0ca8c08d35f2cfe6908ca0752a3
------------------------------------------------------------------------------
--
Warning: The option '--output' has not been specified. All operations
will
         be done in memory with the exception of the '--dump-section'
command.
------------------------------------------------------------------------------
--
Reading xclbin file into memory.  File:
build_dir.hw_emu.xilinx_u250_gen3x16_xdma_4_1_202210_1/vadd.xclbin

==============================================================================
==
XRT Build Version: 2.19.185 (2025.2)
```

```
       Build Date: 2025-04-27 06:39:04
          Hash ID: 6d79c5f0e8c3f0ca8c08d35f2cfe6908ca0752a3
================================================================================
==
xclbin Information
------------------
   Generated by:           v++ (2025.2) on 2025-04-27-00:01:42
   Version:                2.19.185
   Kernels:                vadd
   Signature:
   Content:                HW Emulation Binary
   UUID (xclbin):          65022735-bba0-8345-3e99-a17768a9913e
   Sections:               BITSTREAM, MEM_TOPOLOGY, IP_LAYOUT,
CONNECTIVITY,
                           CLOCK_FREQ_TOPOLOGY, BUILD_METADATA,
                           EMBEDDED_METADATA, SYSTEM_METADATA,
                           GROUP_CONNECTIVITY, GROUP_TOPOLOGY
================================================================================
==
Hardware Platform (Shell) Information
-------------------------------------
   Vendor:                 xilinx
   Board:                  u250
   Name:                   gen3x16_xdma_4_1
   Version:                202210.1
   Generated Version:      Vivado 2022.1 (SW Build: 3510589)
   Created:
               Thu Mar 31 07:42:58 2022   FPGA Device:            xcu250
   Board Vendor:           xilinx.com
   Board Name:             xilinx.com:au250:1.4
   Board Part:             xilinx.com:au250:part0:1.4
   Platform VBNV:          xilinx_u250_gen3x16_xdma_4_1_202210_1
   Static UUID:            00000000-0000-0000-0000-000000000000
   Feature ROM TimeStamp:  0

Scalable Clocks
---------------
   Name:      DATA_CLK
   Index:     0
   Type:      DATA
   Frequency: 300 MHz

   Name:      KERNEL_CLK
   Index:     1
   Type:      KERNEL
   Frequency: 300 MHz

System Clocks
------
   Name:           ii_level1_wire_ulp_m_aclk_ctrl_00
   Type:           FIXED
   Default Freq:   50 MHz

   Name:           ii_level1_wire_ulp_m_aclk_pcie_user_00
   Type:           FIXED
   Default Freq:   250 MHz

   Name:           ii_level1_wire_ulp_m_aclk_freerun_ref_00
   Type:           FIXED
   Default Freq:   100 MHz

   Name:           ss_ucs_aclk_kernel_00
   Type:           SCALABLE
```

```
    Default Freq:   300 MHz
    Requested Freq: 0 MHz
    Achieved Freq:  0 MHz

    Name:           ss_ucs_aclk_kernel_01
    Type:           SCALABLE
    Default Freq:   500 MHz
    Requested Freq: 0 MHz
    Achieved Freq:  0 MHz

Memory Configuration
--------------------
    Name:         bank0
    Index:        0
    Type:         MEM_DDR4
    Base Address: 0x4000000000
    Address Size: 0x400000000
    Bank Used:    No

    Name:         bank1
    Index:        1
    Type:         MEM_DDR4
    Base Address: 0x5000000000
    Address Size: 0x400000000
    Bank Used:    Yes

    Name:         bank2
    Index:        2
    Type:         MEM_DRAM
    Base Address: 0x6000000000
    Address Size: 0x400000000
    Bank Used:    No

    Name:         bank3
    Index:        3
    Type:         MEM_DRAM
    Base Address: 0x7000000000
    Address Size: 0x400000000
    Bank Used:    No

    Name:         PLRAM[0]
    Index:        4
    Type:         MEM_DRAM
    Base Address: 0x3000000000
    Address Size: 0x20000
    Bank Used:    No

    Name:         PLRAM[1]
    Index:        5
    Type:         MEM_DRAM
    Base Address: 0x3000200000
    Address Size: 0x20000
    Bank Used:    No

    Name:         PLRAM[2]
    Index:        6
    Type:         MEM_DRAM
    Base Address: 0x3000400000
    Address Size: 0x20000
    Bank Used:    No

    Name:         PLRAM[3]
    Index:        7
```

```
   Type:          MEM_DRAM
   Base Address: 0x3000600000
   Address Size: 0x20000
   Bank Used:    No

   Name:          HOST[0]
   Index:         8
   Type:          MEM_DRAM
   Base Address: 0x2000000000
   Address Size: 0x400000000
   Bank Used:    No
================================================================================
==
Kernel: vadd

Definition
----------
   Signature: vadd (void* in1, void* in2, void* out, unsigned int size)

Ports
-----
   Port:          M_AXI_GMEM0
   Mode:          master
   Range (bytes): 0xFFFFFFFF
   Data Width:    32 bits
   Port Type:     addressable

   Port:          M_AXI_GMEM1
   Mode:          master
   Range (bytes): 0xFFFFFFFF
   Data Width:    32 bits
   Port Type:     addressable

   Port:          S_AXI_CONTROL
   Mode:          slave
   Range (bytes): 0x3C
   Data Width:    32 bits
   Port Type:     addressable

---------------------------
Instance:        vadd_1
   Base Address: 0x1d010000

   Argument:          in1
   Register Offset:   0x10
   Port:              M_AXI_GMEM0
   Memory:            bank1 (MEM_DDR4)

   Argument:          in2
   Register Offset:   0x1C
   Port:              M_AXI_GMEM1
   Memory:            bank1 (MEM_DDR4)

   Argument:          out
   Register Offset:   0x28
   Port:              M_AXI_GMEM0
   Memory:            bank1 (MEM_DDR4)

   Argument:          size
   Register Offset:   0x34
   Port:              S_AXI_CONTROL
   Memory:            <not applicable>
================================================================================
```

```
==
Generated By
------------
   Command:       v++
   Version:       2025.2 - 2025-04-27-00:01:42 (SW BUILD: 6117406)
   Command Line:  v++ --debug --input_files
_x.hw_emu.xilinx_u250_gen3x16_xdma_4_1_202210_1/vadd.xo --link --optimize 0
--output ./build_dir.hw_emu.xilinx_u250_gen3x16_xdma_4_1_202210_1/
vadd.link.xclbin --platform xilinx_u250_gen3x16_xdma_4_1_202210_1 --
report_level 0 --save-temps --target hw_emu --temp_dir ./
_x.hw_emu.xilinx_u250_gen3x16_xdma_4_1_202210_1
   Options:       --debug
                  --input_files
_x.hw_emu.xilinx_u250_gen3x16_xdma_4_1_202210_1/vadd.xo
                  --link
                  --optimize 0
                  --output ./
build_dir.hw_emu.xilinx_u250_gen3x16_xdma_4_1_202210_1/vadd.link.xclbin
                  --platform xilinx_u250_gen3x16_xdma_4_1_202210_1
                  --report_level 0
                  --save-temps
                  --target hw_emu
                  --temp_dir ./
_x.hw_emu.xilinx_u250_gen3x16_xdma_4_1_202210_1
================================================================================
==
User Added Key Value Pairs
--------------------------
   <empty>
================================================================================
==
Leaving xclbinutil.
```

xrt.ini File

The Xilinx Runtime (XRT) library uses various control parameters to specify debugging, profiling, and message logging when running the host application and kernel execution. These control parameters are specified in a runtime initialization file, `xrt.ini` and used to configure features of XRT at start-up.

If you are a command line user, the `xrt.ini` file needs to be created manually and saved to the same directory as the host executable.

The runtime library checks if `xrt.ini` exists in the same directory as the host executable and automatically reads the file to configure the runtime. You can also specify the location of an `xrt.ini` file at runtime by setting the `XRT_INI_PATH` environment variable to point to the file, for example:

```
export XRT_INI_PATH=/path/to/xrt.ini
```

TIP: *The AMD Vitis™ IDE creates an `xrt.ini` file automatically based on your run configuration and saves it in the run configuration folder.*

Runtime Initialization File Format

The `xrt.ini` file is a simple text file with groups of keys and their values. Any line beginning with a semicolon (;) or a hash (#) is a comment. The group names, keys, and key values are all case sensitive.

The following is an example `xrt.ini` file that enables the timeline trace feature, and directs the runtime log messages to the Console view.

```
#Start of Debug group
[Debug]
native_xrt_trace = true
device_trace = fine

#Start of Runtime group
[Runtime]
runtime_log = console
```

There are three groups of initialization keys:

- Runtime
- Debug
  - AIE_profile_settings
  - AIE_trace_settings
- Emulation

The following tables list all supported keys for each group, the supported values for each key, and a short description of the purpose of the key.

Runtime Group

The Runtime group of switches lets you configure elements of the runtime operation as described below.

Table 31: **Runtime Group Keys and Values**

Key	Valid Values	Description
`api_checks`	`[true|false]`	Enables or disables OpenCL API checks. • `true`: Enable. This is the default value. • `false`: Disable.
`cpu_affinity`	`{N,N,...}`	Pins all runtime threads to specified CPUs. Example: `cpu_affinity = {4,5,6}`

Table 31: **Runtime Group Keys and Values** *(cont'd)*

Key	Valid Values	Description
`exclusive_cu_context`	`[true|false]`	This allows the host application to direct OpenCL to acquire exclusive CU access, so that low-level AXI read/write (xclRegRead and xclRegWrite) can be used for regular kernels.
`force_program_xclbin`	`[true|false]`	Forces a reconfigurable module (RM) `.xclbin` file to be reloaded into the device, even if it is already loaded. This is used to force the reloading of a DFX region in the platform with the `.xclbin` when requested.
`runtime_log`	`[null | console | syslog | <filename>]`	Specifies where the runtime logs are printed • `null`: Do not print any logs. This is the default value. • `console`: Print logs to `stdout` • `syslog`: Print logs to Linux syslog. • `<filename>`: Print logs to the specified file. For example, `runtime_log=my_run.log`.
`verbosity`	`[0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 ]`	Verbosity of the log messages. The default value is 4.

Debug Group

The Debug group of switches define key options for the enabling profiling of the application during runtime, or tracing data transfers and execution. These switches apply to both AI Engine and PL kernels in the Vitis acceleration flow, and let you configure aspects of the runtime to control the frequency of data capture, the events to capture, and the amount of memory to reserve or use for recording trace and profile data.

Table 32: **Debug Options**

Key	Valid Values	Description
`aie_profile`	`[true|false]`	Enables the runtime configuration and polling of AI Engine hardware performance counters. Available on hardware and hardware emulation runs. • `true`: Enable. • `false`: Disable. This is the default value.
`aie_trace`	`[true|false]`	Enables the runtime configuration and collection of AI Engine event trace. Available on hardware runs only. • `true`: Enable. • `false`: Disable. This is the default value.

Table 32: **Debug Options** *(cont'd)*

Key	Valid Values	Description
`aie_status`	`[true|false]`	Enables the polling of AI Engine status information. Available on hardware and hardware emulation runs.
`aie_status_interval_us`	integer (default=1000us)	Controls the interval at which AI Engine status information is captured. Specified in microseconds.
`continuous_trace`	`[true|false]`	Enables the continuous dumping of files for trace and the continuous reading of device data into the host. • `true`: Enable. • `false`: Disable. This is the default value. ***Note***: This switch only has an effect if `device_trace` is enabled.
`device_counters`	`[true|false]`	Enables device counter offload only, without enabling trace functionality.
`device_trace`	`[off|fine|coarse|accel]`	Enables the collection of data from monitors inserted on the PL to add to summary and trace. • `accel`: Traces compute unit starts/stops. • `coarse`: Lumps all reads/writes together under each execution of a compute unit. • `fine`: Tracks everything as it happens. • `off`: Turns off reading and reporting of device-level trace during runtime. This is the default value.
`host_trace`	`[true|false]`	Enables trace of host code based on the first protocol encountered. **TIP:** *If your host application uses XRT native API you should manually specify `native_xrt_trace` to capture all events.*
`native_xrt_trace`	`[true|false]`	Enables generation of the Native C/C++ API trace. This also generates the tables for "Host Data Transfer from/to Global memory" in the Profile Summary. • `true`: Enable. • `false`: Disable. This is the default value.
`pl_deadlock_detection`	`[true|false]`	Enables deadlock detection for PL kernels.

Table 32: **Debug Options** *(cont'd)*

Key	Valid Values	Description
`power_profile`	`[true|false]`	Enables the polling of power data during the execution of the application. • `true`: Enable. • `false`: Disable. This is the default value. ***Note***: This feature is not supported on embedded platforms or AWS.
`power_profile_interval_ms`	`<int>(default=20)`	Controls the interval of reading the power counters in milliseconds. The default interval is 20 ms. ***Note***: This switch only has an effect if `power_profile = true`.
`profile_api`	`[true|false]`	Enables access to HAL API directly from the host application to read counters on device profiling monitors during execution. • `true`: Enable. • `false`: Disable. This is the default value.
`stall_trace`	`[off|all|dataflow|memory|pipe]`	Specifies the type of device-side stalls to capture and report in the timeline trace. The default is off. • `off`: Turn off stall trace information. `all`: Record all stall trace information. `dataflow`: Intra-kernel streams (for example, writing to full FIFO between dataflow blocks). `memory`: External memory stalls (for example, AXI4 read from the DDR memory). ***Note***: This switch only has an effect if `device_trace` is enabled.
`trace_buffer_offload_interval_ms`	`<int>`	Controls the reading of device data from the device to the host in milliseconds (ms). The default is 10 ms. ***Note***: This switch only has an effect if `device_trace` is enabled.
`trace_buffer_size`	`<string>`	If the `.xclbin` was created with memory offload of trace specified, as described in --profile Options,this switch determines the size of the buffer to allocate in memory to capture trace data. The default is 1M. ***Note***: This switch only has an effect if `device_trace` is enabled.

Table 32: **Debug Options** *(cont'd)*

Key	Valid Values	Description
`trace_file_dump_interval_s`	`<int>`	Controls the time between dumping of trace files in seconds (s). The default is 5s. ***Note***: This switch only has an effect if `device_trace` is enabled.
`vitis_ai_profile`	`[true|false]`	Profile summary and other files come from Vitis AI application layer. • `true`: Enable. • `false`: Disable. This is the default value.
`xocl_debug`	`[true|false]`	Generates the `xocl.log` file when enabled. When any trace options are also enabled, the debug log is added to the `xrt.run_summary` to view in Vitis Analyzer.
`xrt_trace`	`[true|false]`	Enables generation of low-level HW shim function trace during HW runs. This will be disabled when used with `native_xrt_trace`. • `true`: Enable. • `false`: Disable. This is the default value.

AIE_profile_settings Group

The options specified in this group are applied only if `aie_profile=true` under the `[Debug]` group.

Table 33: **AI Engine Profile Options**

Key	Valid Values	Description
`graph_based_aie_metrics`	<graph name|all>:<kernel name|all>:<off|heat_map|stalls|execution|floating_point|write_bandwidths|read_bandwidths|aie_trace>	Specify the metric sets reported by the AI Engine module of AI Engine tiles on a graph-by-graph basis. **IMPORTANT!** *Currently, only `all` is supported for kernel specification.* Controls the configuration of the statistics read from the AIE core performance counters for the entire AI Engine graph application. heat_map: profile active/stall cycles and vector instruction usage stalls: profile the different types of stalls (i.e., memory, stream, lock, and cascade) execution: profile the AI Engine instructions floating_point: profile floating point exceptions write_bandwidths: profile the write bandwidth of streams and cascades read_bandwidths: profile the read bandwidths of streams and cascades aie_trace: profile amount and stalls of event trace from core and memory modules
`graph_based_aie_memory_metrics`	<graph name|all>:<kernel name|all>:<off|conflicts|dma_locks|dma_stalls_s2mm|dma_stalls_mm2s|write_bandwidths|read_bandwidths>	Specify the metric sets reported by the memory module of AI Engine tiles on a graph-by-graph basis. **IMPORTANT!** *Currently, only `all` is supported for kernel specification.* Controls the configuration of statistics read from the AI Engine memory performance counters for the entire AI Engine graph application. conflicts: profile the DMA memory conflicts dma_locks: profile DMA locks and stalls on lock acquire dma_stalls_s2mm: profile stalls on DMA S2MM channels dma_stalls_mm2s: profile stalls on DMA MM2S channels write_bandwidths: profile bandwidths of DMA S2MM channels read_bandwidths: profile bandwidths of DMA MM2S channels
`tile_based_aie_metrics`	<{<column>,<row>}|all>:<off|heat_map|stalls|execution|floating_point|write_bandwidths|read_bandwidths|aie_trace> ; {<mincolumn,<minrow>}:{<maxcolumn>,<maxrow>}:<off|heat_map|stalls|execution|floating_point|write_bandwidths|read_bandwidths|aie_trace>	Specify the metric sets reported by the AI Engine module of AI Engine tiles on a tile-by-tile basis. This can be used in conjunction with graph-by-graph selection and will take priority on the specified tiles. Refer to descriptions from `graph_based_aie_metrics`

Table 33: **AI Engine Profile Options** *(cont'd)*

Key	Valid Values	Description
`tile_based_aie_memory_metrics`	<{<column>,<row>}|all>:<off|conflicts|dma_locks|dma_stalls_s2mm|dma_stalls_mm2s|write_bandwidths|read_bandwidths> ; {<mincolumn,<minrow>}:{<maxcolumn>,<maxrow>}:<off|conflicts|dma_locks|dma_stalls_s2mm|dma_stalls_mm2s|write_bandwidths|read_bandwidths>	Specify the metric sets reported by the memory module of AI Engine tiles on a tile-by-tile basis. This can be used in conjunction with graph-by-graph selection and will take priority on the specified tiles. Refer to descriptions from `graph_based_aie_memory_metrics`
`tile_based_interface_tile_metrics`	<column|all>:<off|input_bandwidths|output_bandwidths|packets>[:<channel>] ; <mincolumn>:<maxcolumn>:<off|input_bandwidths|output_bandwidths|packets>[:<channel>]	Specify the metric sets reported by the AI Engine interface tiles on a tile-by-tile basis. ***Note***: Interface tiles are separate from the AI Engine tiles and have different metric sets.
`interval_us`	`<int>`	Controls the interval of reading the AI Engine counter values in microseconds (µs). The default interval is 1000 µs. ***Note***: This switch only has an effect if `aie_profile = true`.

AIE_trace_settings Group

The options specified in this group are applied only if `aie_trace=true` under the `[Debug]` group.

Table 34: **AI Engine Trace Options**

Key	Valid Values	Description
`buffer_size`	`<string> (default=8M)`	Controls the total size of the buffers allocated for AI Engine event trace. This size is partitioned evenly into the number of different trace streams coming out of the AI Engine. The default is 8M. ***Note***: This switch only has an effect if `aie_trace = true`.
`buffer_offload_interval_us`	integer (default=10ms)	Interval, in milliseconds, between reading of PLIO mode AI Engine trace from device to Host memory.
`periodic_offload`	true/false (default=true)	Enables continuous offload of PLIO mode AI Engine trace. Generated AI Engine trace output files (one per stream) gets appended with new trace data.
`file_dump_interval_s`	integer (default=5s)	Interval, in seconds, between writing (appending) of raw AI Engine trace data to output files.

Table 34: **AI Engine Trace Options** *(cont'd)*

Key	Valid Values	Description
`graph_based_aie_tile_metrics`	string("") <graph name|all>:<kernel name|all>:<off|functions|functions_partial_stalls|functions_all_stalls>	Specify the metric sets reported by the AI Engine module of AI Engine tiles on a graph-by-graph basis. **IMPORTANT!** *Currently, only `all` is supported for kernel specification.*
`tile_based_aie_tile_metrics`	string("") <{<column>,<row>}|all>:<off|functions|functions_partial_stalls|functions_all_stalls>[:<memory_stalls|stream_stalls|cascasde_stalls|lock_stalls>] {<mincolumn,<minrow>}:{<maxcolumn>,<maxrow>}:<off|functions|functions_partial_stalls|functions_all_stalls>	Specify the metric sets reported by the AI Engine module of AI Engine tiles on a tile-by-tile basis. **IMPORTANT!** *Currently, only `all` is supported for kernel specification.*
`reuse_buffer`	true/false (false)	

Emulation Group

The Emulation group of switches apply to the emulation environments and the AMD Vivado™ simulator.

Table 35: **Emulation Group Keys and Values**

Key	Valid Values	Description
`aliveness_message_interval`	Any integer	Specifies the interval in seconds that aliveness messages need to be printed. The default is 300.
`debug_mode`	`[off|batch|gui]`	Specifies how the waveform is saved and displayed during emulation. • `off`: Do not launch simulator waveform GUI, and do not save `wdb` file. This is the default value. • `batch`: Do not launch simulator waveform GUI, but save `wdb` file • `gui`: Launch simulator waveform GUI, and save `wdb` file ***Note*:** The kernel needs to be compiled with debug enabled (`v++ -g`) for the waveform to be saved and displayed in the simulator GUI.
`print_infos_in_console`	`[true|false]`	Controls the printing of emulation info messages to user's console. Emulation info messages are always logged into a file called `emulation_debug.log` • `true`: Print in user's console. This is the default value. • `false`: Do not print in user console.

Table 35: **Emulation Group Keys and Values** *(cont'd)*

Key	Valid Values	Description
`print_warnings_in_console`	`[true|false]`	Controls the printing emulation warning messages to user's console. Emulation warning messages are always logged into a file called `emulation_debug.log`. • `true`: Print in user's console. This is the default value. • `false`: Do not print in user console.
`print_errors_in_console`	`[true|false]`	Controls printing emulation error messages in user's console. Emulation error messages are always logged into the `emulation_debug.log` file. • `true`: Print in user's console. This is the default value. • `false`: Do not print in user's console.
`user_pre_sim_script`	Path to Tcl file	For the first run, run simulation in GUI mode. Add signals that you want to add. Copy the commands from the Tcl console and save into a Tcl script. For the next run, pass the Tcl script in batch mode.
`user_post_sim_script`	Path to Tcl file	Any post operations can be specified in the Tcl and pass to the switch. All the command provided in the Tcl gets executed after simulation is completed.
`xtlm_aximm_log`	`[true|false]`	Enables the XTLM AXI4 Memory Map transaction logging at runtime and you could see all the transactions in the `xsc_report.log` file.
`xtlm_axis_log`	`[true|false]`	Enables the XTLM AXI4-Stream transaction logging at runtime and you could see all the transactions in the `xsc_report.log` file.
`timeout_scale`	`na/ms/sec/min`	Timeout support for `clPollStream` API in emulation. Provides a scale for the timeout specified in `clPollStream` API. The timeout specified in the code is specified in ms, and might not work for emulation. Therefore use the `timeout_scale` to map ms to another scale if needed for emulation. **IMPORTANT!** *Timeout is not enabled in emulation by default. Use this option to enable `clPollStream` timeout.*

Using the Vitis Unified IDE

The Vitis command-line tool flow described in Building and Running the System in the *Data Center Acceleration using Vitis* (UG1700) is also available in the Vitis unified IDE. The process of building and combining the different components of the system project, the Application component, the AI Engine component, the HLS component, and the platform are described in the following sections.

- Launching Vitis Unified IDE
- Creating an AI Engine Component
- Creating an HLS Component
- Creating an Application Component
- Creating a System Project for Heterogeneous Computing
- Debugging the System Project and AI Engine Components
- Vitis Interactive Python Shell

TIP: *Before using the Vitis unified IDE, you must first set up the development environment as described in Setting Up the Vitis Environment in the Data Center Acceleration using Vitis (UG1700).*

Migration to Vitis Unified IDE

Migrating Vitis Classic IDE Graph Applications to Vitis Unified IDE

Classic Vitis IDE workspaces cannot be opened directly in the Vitis Unified IDE. The classic IDE provides an utility to enable moving a project or workspace into the Vitis Unified IDE. To use this utility, first launch the Vitis classic IDE in the workspace with the following command:

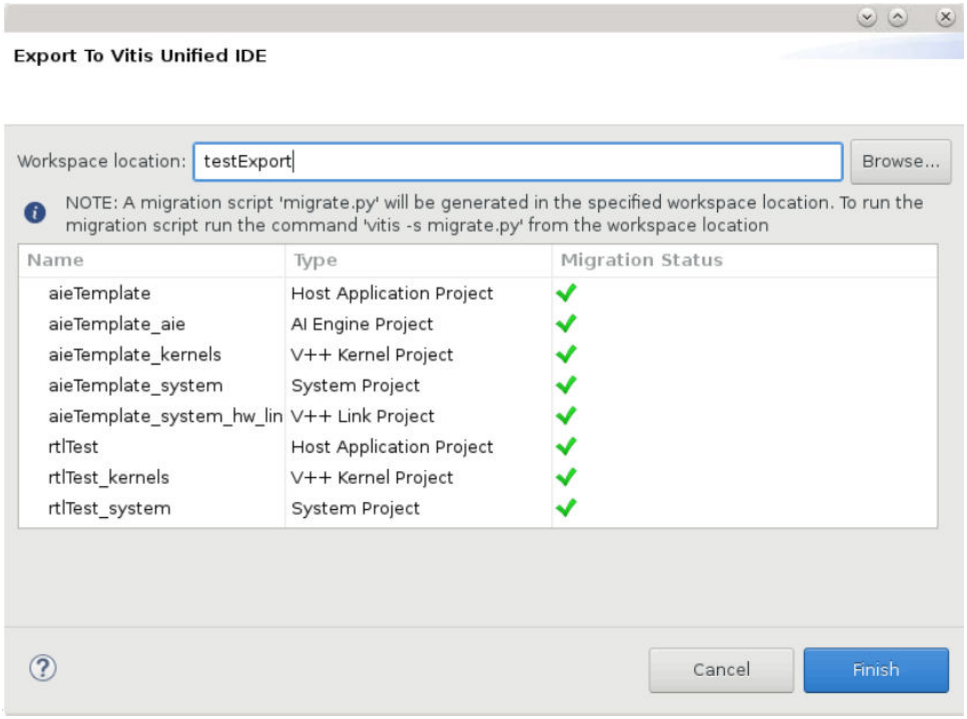
```
vitis --classic -workspace <workspace>
```

After Vitis IDE opens the workspace, use the **Vitis → Export Workspace to Unified IDE** command from the main menu. The tool opens a dialog box listing the projects contained in the workspace, and prompts you for a new workspace to export the projects to. Specify the new workspace location and click **Finish**. The tool will generate a Python migration script (`migrate.py`) and write it to the specified workspace folder.

TIP: *The Workspace location specified for the export script must be a new empty workspace. If you want to make changes to the classic IDE project and re-export it, you must start with a clean export workspace, without any hidden files.*

***Note*:** The migration utility is created to help you speed up the procedure, but you can also recreate the workspace in the Vitis Unified IDE from scratch for migration. The migration utility does not consider all the corner cases and can encounter issues for complex designs.

Figure 3: **Export to Vitis Unified IDE**



After generating the migration script, a pop-up window will display the location of the script. You will need to run the script in the Vitis Unified IDE by using the `vitis -s <script>` form of the command as described in Launch Options in the *Vitis Reference Guide* (UG1702).

```
vitis -s migrate.py
```

Table 36: **Supported Project Types**

Project Type	Limitation	Workaround
Embedded Platforms	Local changes to BSP sources will not be migrated to the new workspace. A new BSP will be created, and the settings are applied on the new BSP.	Copy the sources to the new BSP manually.
	Any embedded software repositories added to the Vitis IDE will not be migrated. A warning is displayed in the wizard if the project has any local SW repos.	All software repositories need to be migrated to lopper first. Refer to *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400) for more information. The path to the migrated repository can be manually added to the migration script prior to running the script.
	IP drivers included in XSAs created with 2023.1 or older releases will not work.	Need to regenerate the XSA.
Embedded Software Applications	Applications referring to platforms that are outside the current workspace cannot be migrated.	Migrate the platform first and update the application to use the new platform before migrating the application.
HW Link	Hardware linker options defined using the **Extra V++ command line options** will not be migrated through the script.	You will need to manually define these options in the `hw_link.cfg` for the System project
Accelerated Host applications	Only compile definitions (-D), include path (-I), library paths (-L) and libraries (-l) are migrated for accelerated host applications.	Any other compiler or linker settings from the C/C++ build settings window will need to be set manually in the Application component.

Launching Vitis Unified IDE

***Note*:** The AMD Vitis™ Unified Integrated Design Environment (IDE) supports data center acceleration and embedded system design, AI Engine and High-Level Synthesis (HLS) component creation, platform creation, and embedded software design. You can launch this tool by using the following command:

```
vitis  -w <workspace>
```

In the AMD Vitis™ unified IDE, you can create a new embedded application project or platform development project. The `vitis` command launches the Vitis unified IDE with your defined options. It provides options for specifying the workspace and options of the project. The following sections describe the `vitis` command options.

Command Options

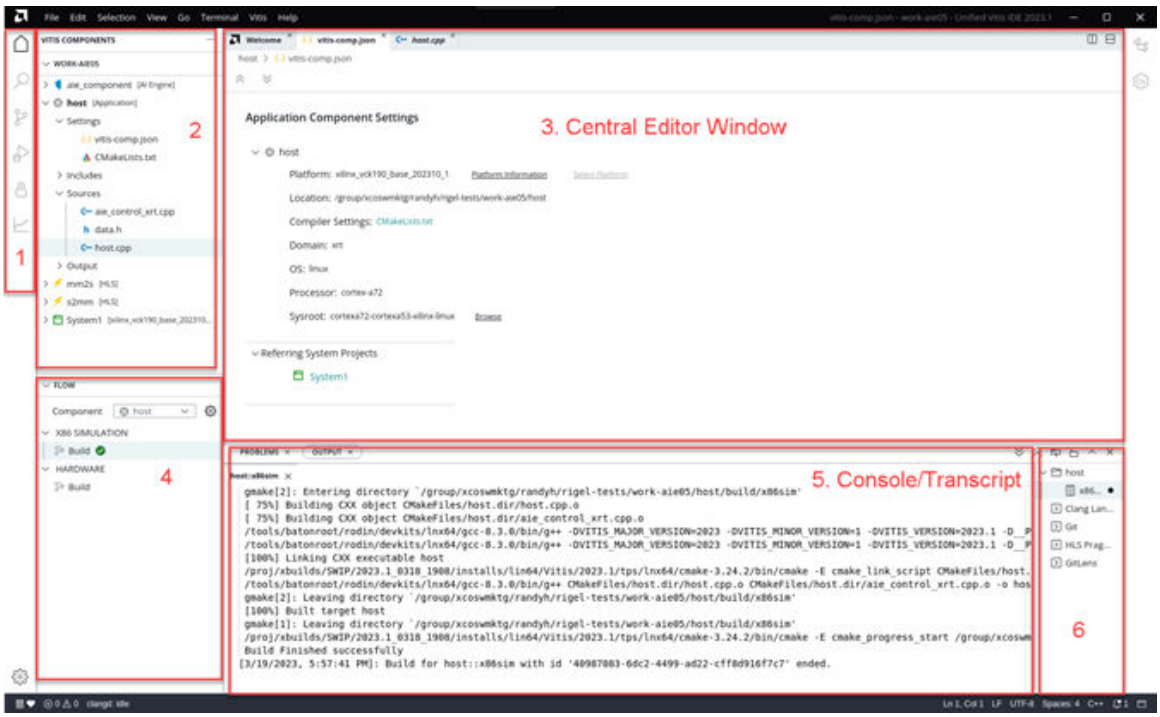
The following command options specify how the `vitis` command is configured for the current workspace and project.

```
vitis [-a | -w | -i | -s | -h | -v]
```

- `-a/--analyze [<summary file | folder | waveform file: *.[wdb|wcfg]>]`: Open the summary file in the Analysis view. Opening a folder opens the summary files found in the folder. Open the waveform file in a waveform view tab. If no file or folder is specified, opens the Analysis view.
- `-workspace <workspace location>`: Specify the workspace directory for Vitis Unified IDE projects.
- `-i/--interactive`: Launches Vitis Python interactive shell.
- `-s/--source <python_script>`: Runs the specified Python script.
- `-j/--jupyter`: Launches Vitis Jupyter web UI.
- `-help`: Displays help information for the Vitis command options.
- `-version`: Displays the Vitis tool release version.

Vitis Unified IDE View and Features

Figure 4: **Vitis Unified IDE**

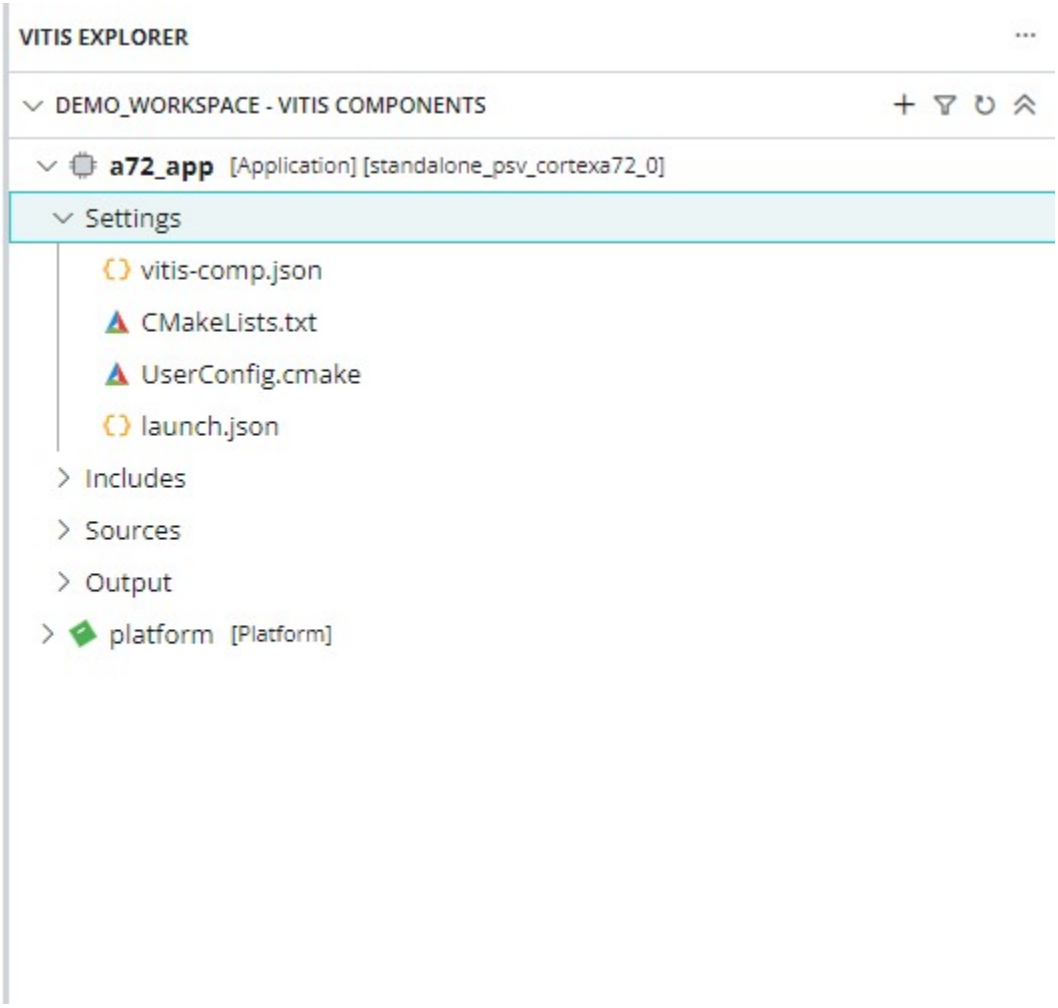


1. The Toolbar menu provides easy access to major features of the tool.
2. You can customize the Toolbar based on your preference using the View dropdown.

3. You can use the Vitis Component Explorer to view a virtual hierarchy of the workspace. The workspace can help you understand the different elements of the component or project.
4. You can use the Central Editor window for editing components, configurations, and source files.
5. The Flow Navigator displays the design flow for the active component:
   - Components have different work flows.
   - The workflow of the active component is in the Flow Navigator.
   - You can specify the active component by selecting it in the Flow Navigator.
6. The Console/Terminal area displays the following:
   - The output transcripts of the tool.
   - Windows such as the Terminal window and the Pipeline view.
   - You can use the folders of the workspace to run scripts on the content.
7. The Index displays a list of transcripts of the build steps during the session. This display allows you to reopen a process transcript you closed.

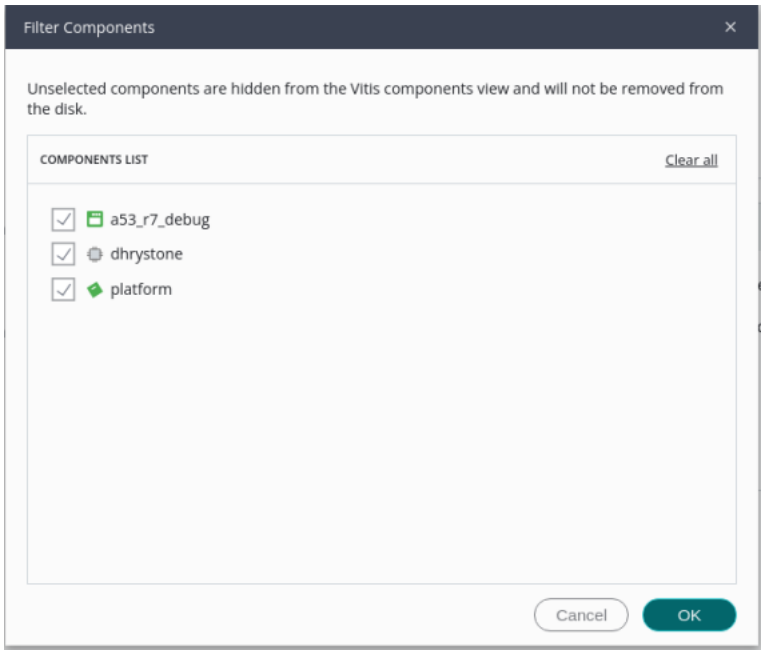
Vitis Explorer View

Figure 5: **Vitis Explorer View**



You can use the Component Explorer to perform the following actions:

- **Using the toolbar menu, you can do the following::**
  - Create a New Component.
  - Select Filter Components to hide or show components.



  - Refresh the Component Explorer.
  - Collapse All to minimize the displayed contents.
- **Using the right-click menu to select a component or project, you can do the following::**
  - Select **Open in Terminal** to open a terminal window and change directory to inside the component.
  - Select **Show in Flow Navigator** and click the component to show it in the Flow Navigator.
  - Select **Delete** to remove the component or project from the workspace.
  - Select **Clone Component** to create a new component from an existing component. This is useful for design exploration where you need to preserve the original component as your baseline. You can then use the cloned component for design exploration. This command does not support System projects.
  - Select **Reset Linker Script** to generate a linker script for a standalone application component.
- **By right-clicking the `Sources` folder of a component or project, you can do the following::**
  - Select **New File** to create a new file in the component or project.
  - Select **New Folder** to create a new folder in the component or project.
  - Select **Open in Terminal** to open a terminal window and change directory to inside the component.
  - Select **Paste** to take a copied item and paste it into the component or project.
  - Select **Import → Files** to import files into the component or project.

  - Select **Import → Folders** to import a folder into the component or project.
  - Select **Add Source File** or create a New Source File.

- **By right-clicking an object in the component or project hierarchy, you can do the following::**
  - Select **Copy** followed by **Paste** to copy an object from one component or project to another.
  - Select **Copy Path** to copy the absolute path of the object. You can paste the path into the Terminal view of a configuration file.
  - Select **Copy Relative Path** to copy the path of the object relative to the current workspace. You can paste the path into the Terminal view of a configuration file.
  - Select **Delete** to remove the object from the workspace.

The Flow window is at the bottom part of the Vitis Components view. When you select a component, the Flow window actively displays the range of actions you can perform. Actions include build and debug.

Deep JSON Based Component Display

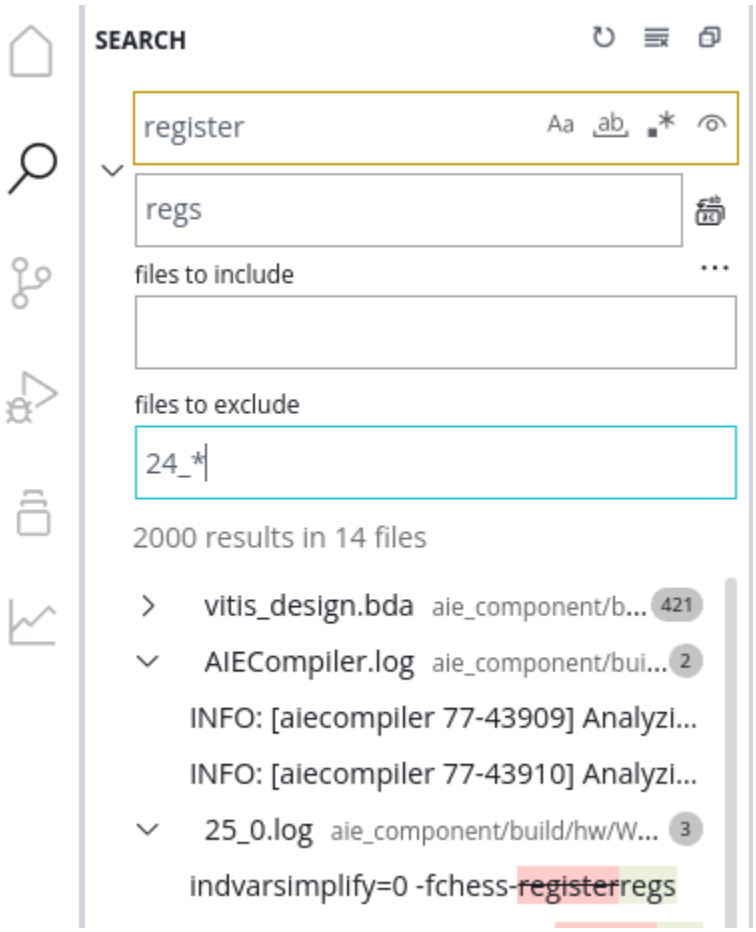
Vitis unified software platform supports displaying all components from the current workspace, no matter which sub directory they are in, in the Component View with unique names. You can directly access, select, edit, compile and use these components.

***Note*:** The system displays notifications about unsupported components.

Search View

You can use the search view to find and replace text globally within the current workspace. The search result includes text files in the workspace. These files include source files, configuration files, and log files. The system performs the search inside files. File names are not part of the search.

Figure 6: **Search View**



As shown in the image above, some of the features of the Search view include the following:

- **Match Case:** Matches the case of the provided search string.
- **Match Whole Word:** Matches only whole words.
- **Use Regular Expression:** Defines the search using regular expressions.
- **Include Ignored Files:** Restores ignored files to the search list.
- **Replace All:** Replaces all search results.
- **Toggle Search Details:** Expands the Search view to add Files to Include and Files to Exclude fields.
- **Files to include:** Specifies a list of files to restrict your search. The listed files can include wildcards, and each entry must be separated by a comma. For example, the following includes (or excludes) the `AIECompiler.log` file and all files with `summary` in the name:

```
AIECompiler.log, *summary*
```

- **Files to exclude:** Specifies a list of files to restrict your search. See above for example.
- **Clear Search Results:** Clears the search string and search results.

> **TIP:** *Be careful when using the `Replace` function, because it can introduce errors into your designs.*

Source Control

Software developers widely use source and version control. This chapter describes how to use Git integration in the Vitis Unified IDE.

Enabling the Source Control View

To enable the Source Control view:

- Initialize your empty workspace as a Git repository.
- Create a workspace.
- Launch the Vitis Unified IDE to open the workspace.
- Add the workspace to your Git repository and perform the following steps.

1. From the Terminal menu, select New Terminal. The terminal defaults to the folder that is your workspace.
2. In the Terminal window, type in the command `git init` and press Enter.

A message similar to `Initialized empty Git repository in /tests/temp/workVADD/.git/` indicates the end of the process.

> **IMPORTANT!** *Using the Source Control view with Git requires you to have a User ID and Password established, and provided to the system.*

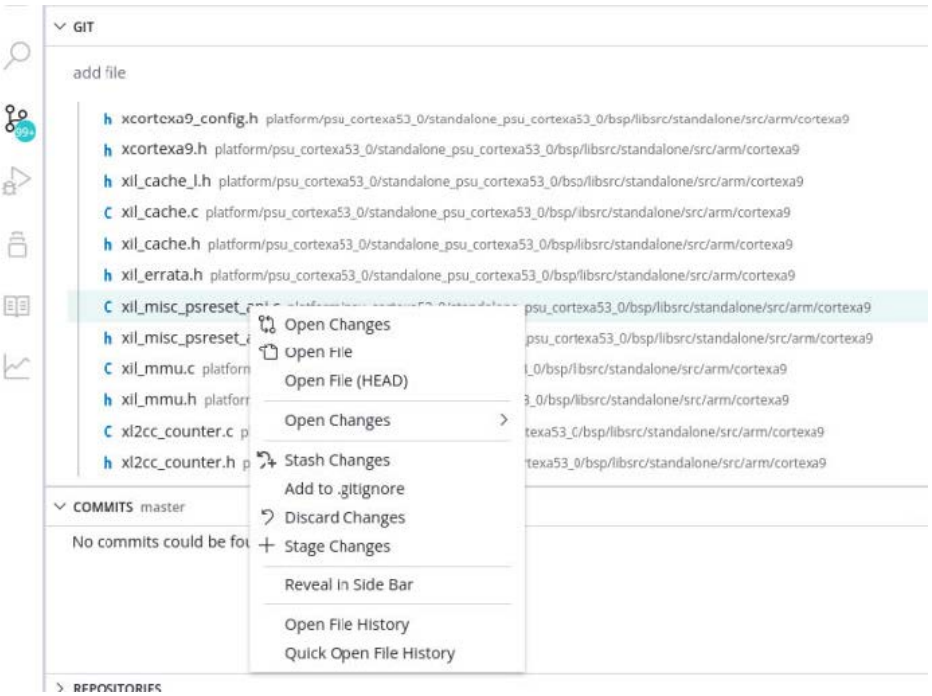
After you create a new component or project in the workspace, Vitis Unified IDE generates a `.gitignore` file. The `.gitignore` file can help you filter out the generated files so that it is easier to pick the files for source control. You can open the `.gitignore` file and edit this file if you have additional requirements.

The Source Control view is a GUI helper for Git. You can use Git commands and the source control view simultaneously for your project. The source control view displays updates in the command line..

Add New File for Source Control

To add a file for source control, you can do the following: Right click the file you want to add and select **+ Stage Changes**.

Figure 7: **Add New File for Source Control**

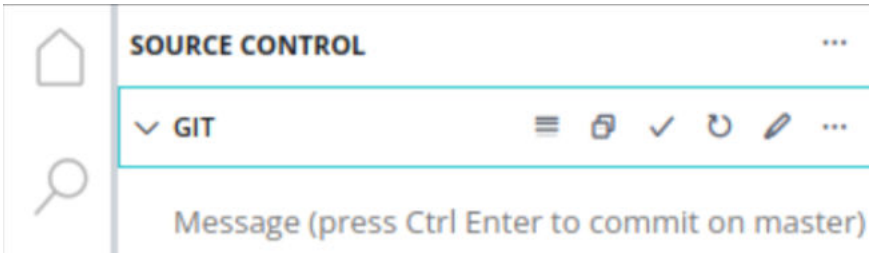


This GUI is equivalent to the following git command.

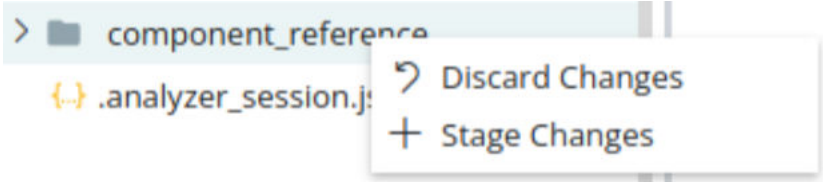
```
git add <file name>
```

Add Components for Source Control

1. Switch to Source Control view.
2. Switch to **View As Tree**.



3. Right click the component you want to add for source control and select **+ Stage Changes**. This action adds all the files for this component to perform source control.



4. Right click the component and select **- Unstage changes**.

This GUI functionality is equivalent to the following git command:

```
git add platform/
```

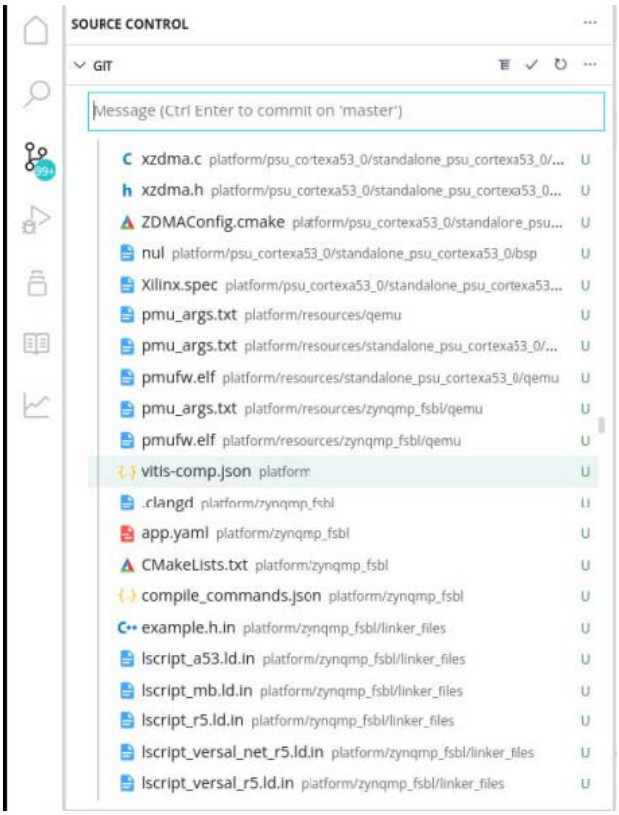
Note: You need to select the components bound with the system project if you want to add system project for source control.

Commit the Changes

To commit the change, you can do either of the following:

Input the commit message and press **Ctrl + Enter** after you select the Git view.

Figure 8: **Git View**



This GUI is equivalent to the following command:

```
git commit -m <commit message>
```

Push the Project to the Remote Repository

To push to your remote repository, you can do either of the following:

```
git push --set-upstream origin master
```

- **origin:** This is the remote repo address.
- **master:** This is the branch of your local workspace coed version.

Relative Path Support

If the imported files share the same initial node in the path as the workspace, the Vitis tool automatically selects the relative path. The following is an example:

- File Path: `/local/drive/source/app.cpp`
- Workspace Path: `/local/drive/workspace`

The system selects this relative path:

```
$COMPONENT_LOCATION/../../sources/app.cpp
```

***Note*:** When moving the workspace or committing it to source control, remember to keep the relative path relationship.

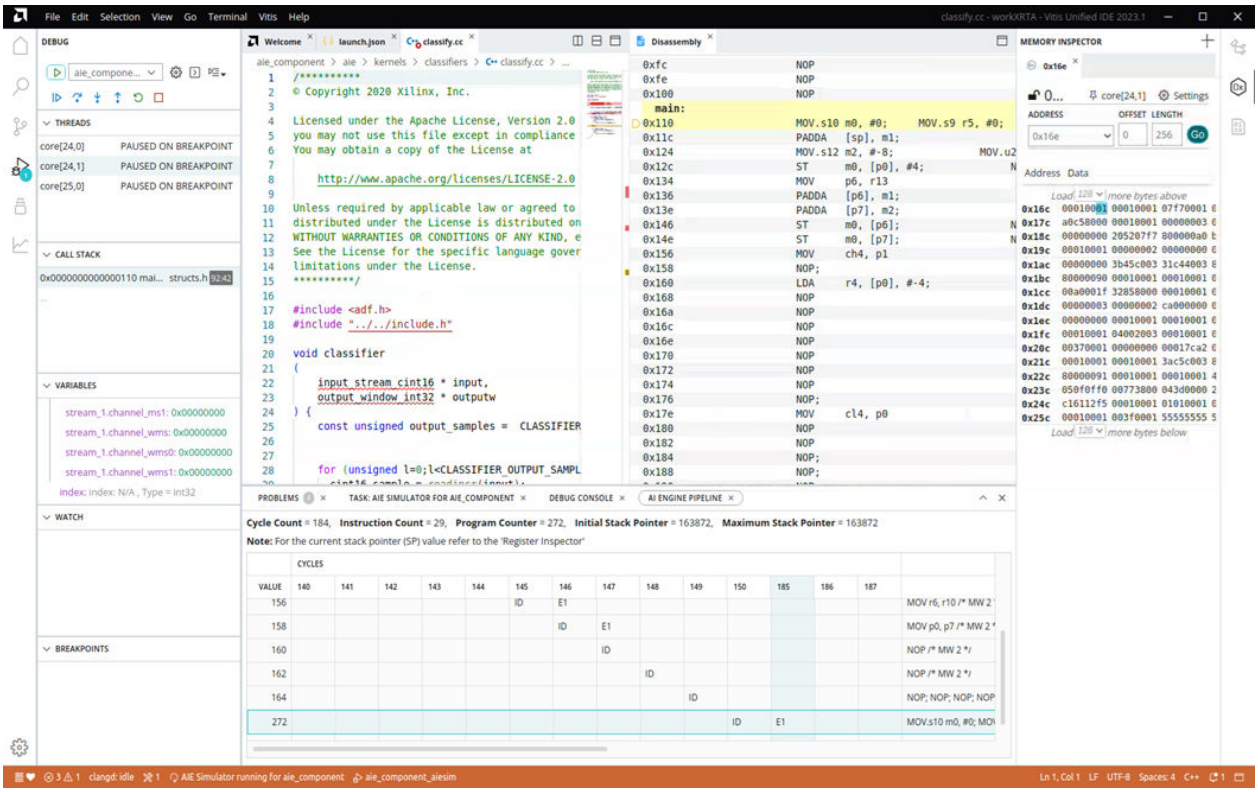
Debug View

The Vitis IDE debug environment has many features found in traditional GUI-based debug environments, such as GDB. In the debug environment, you can do the following:

- Add break points to the code.
- Step over or step into specific lines of code, loops, or functions.
- Examine the state of variables and force them to desired values.

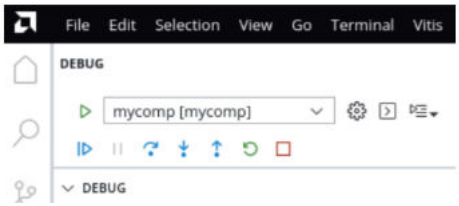
The AI Engine Debug view contains several windows or views as shown in the following figure.

Figure 9: **AIE Component Debug View**



- **Control Panel:** The Debug view's Control Panel is at the upper-left corner of the screen. During debugging, you can use the control buttons such as Continue, Step Over, Step Into, Step Out, Restart, and Stop to control the debugging process.

Figure 10: **Debug Control Panel**



- **Threads:** Threads shows the related debugging threads. The debugging process creates and destroys threads. You can switch between multiple threads.
- **Call Stack:** Call Stack shows the function call stack updating when the application is running.
- **Variables:** Variables shows the current value of global and local variables. The variable information updates when switching threads.
- **Watch:** Watch shows variables and expressions you have specified to watch. To add watch points select **Add Expression** (+).

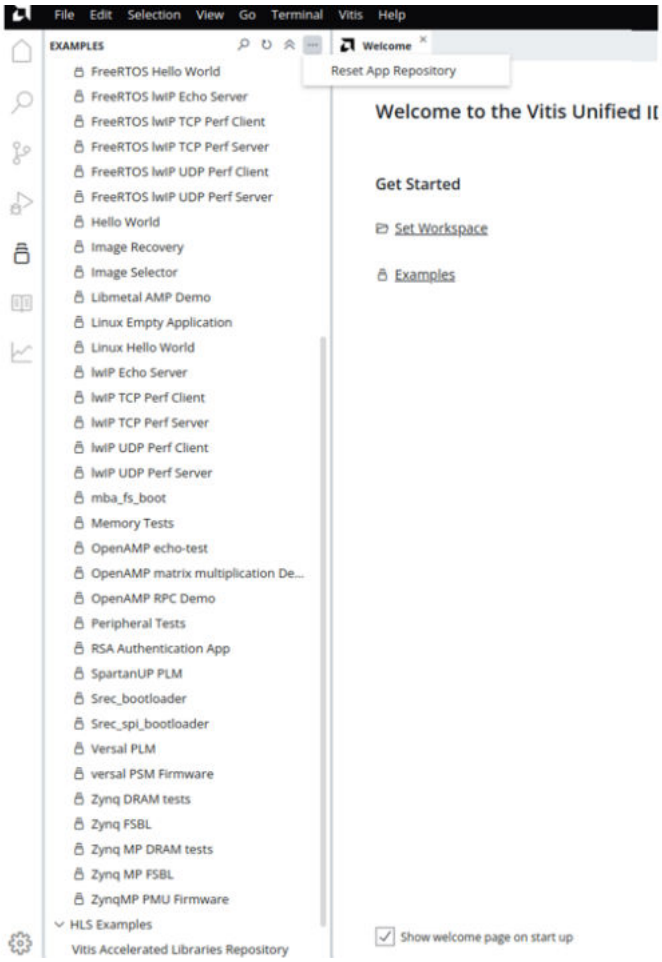
- **Breakpoints:**
  - The system sets break points at the host component's **main** function if it is debuggable.
  - The system sets break points at the top function of the PL kernels if they are debuggable.
  - To add breakpoints, open the source file and click the left side of the line number when a red dot appears.
  - To remove breakpoints, click on a previously added breakpoint.
  - To add conditional breakpoints, right-click when the red dot appears and select **Add Conditional Breakpoint**. To log a message, select **Add Logpoint** when reaching the breakpoint.
- **Source Code Editor:** The Source Code view opens after launching Debug from Flow Navigator or Launch Configuration.
- **Memory Inspector:** You can manually open the Memory Inspector. The Memory Inspector displays the content of specific memory addresses.
- **Register Inspector:** You can manually open the Register Inspector. This view shows the following:
  - The registers of the Cortex-A72 at a triggered breakpoint in the Application Component source code.
  - The registers of the AI Engine at a triggered breakpoint in the AI Engine kernel.
- **Disassembly View:** You can open the Disassembly view from the Source Code window right-click menu.
- **Debug Console:** This view displays the transcript of the debug process and messages received from the application.

***Note*:** You can see all the prints from MDM (MicroBlaze/ V Debug Module) UART in the Debug Console.

Example View

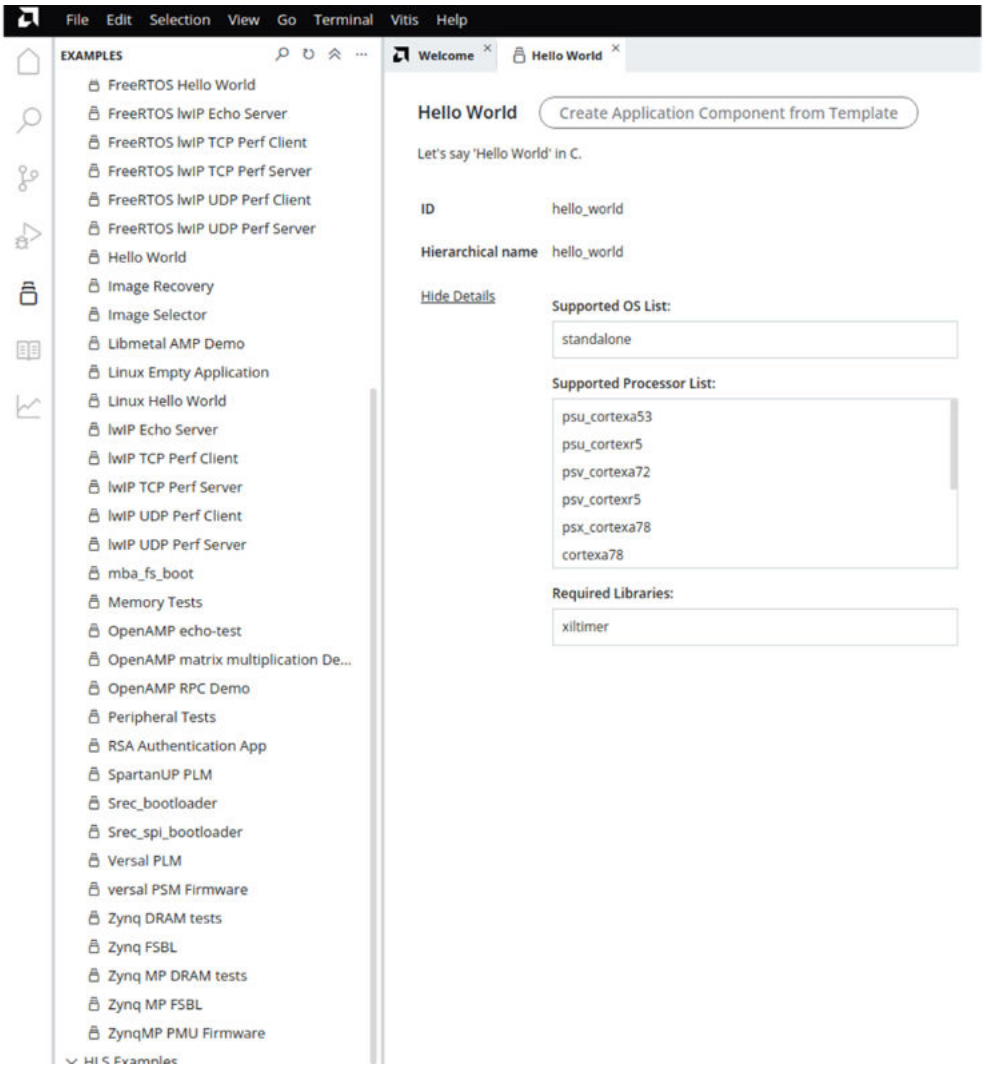
You can access the Examples View from the left side panel or from **View → Examples**, or by using **Ctrl + Shift + R** shortcut keys. In this view, you can create example System projects to explore the tool, and manage example repositories.

Figure 11: **Examples View**



To use an example, select it from the Examples view and a template opens to let you create a new application component. Click **Show Details** to display the supported OS and processor for this application.

Figure 12: **Example View Supported Devices**



Code View and Smart Editor

The Source Code editor in the Vitis Unified IDE supports the following features:

- Syntax highlight for C, C++, Python, Makefile, CMakeList.txt file
- Hint for variable names, function names
- Jumping to the definition of variables or functions
- Peeking the definition of variables or functions so you do not to leave the editing file
- Reporting the references of variables and function

You can open the outline window by selecting the button on the right, or selecting the Outline View from the View menu. The Outline window lists the function names in your source code.

The smart editor for `launch.json` files and `build.json` files can switch views between GUI rendering, Text Rendering and Text Editor. Use the table button and code button to switch between views. The GUI rendering only displays the options that it can recognize for the context. For advanced use cases, you need to edit the configuration file manually. The modifications in one view updates the other view instantly.

Figure 13: **GUI Format**

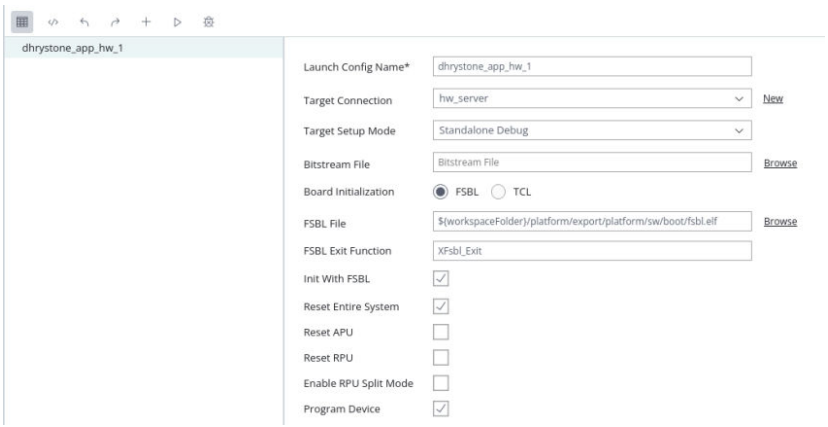
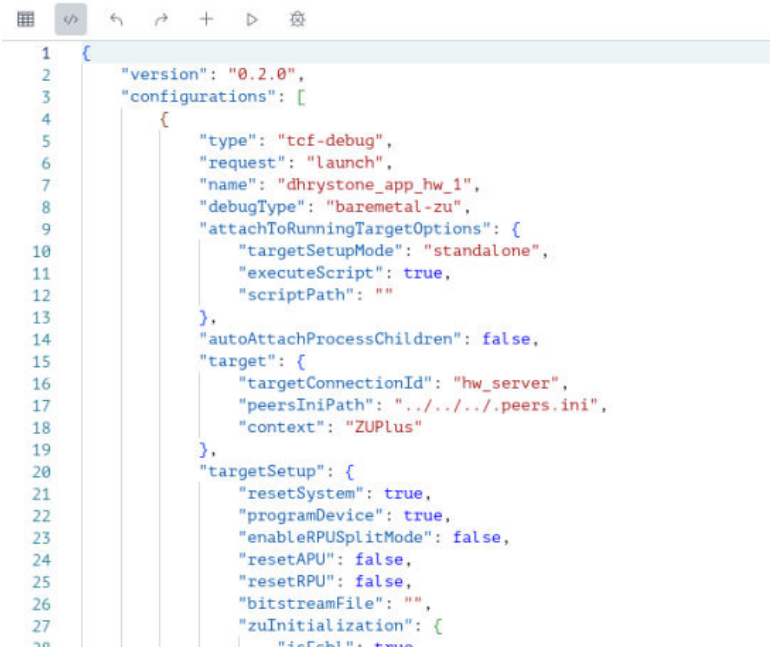


Figure 14: **Text Format**



The system saves the changes automatically when you enable Auto Save. Optionally, you can manually save changes in a file with the keyboard shortcut `Ctrl + S`.

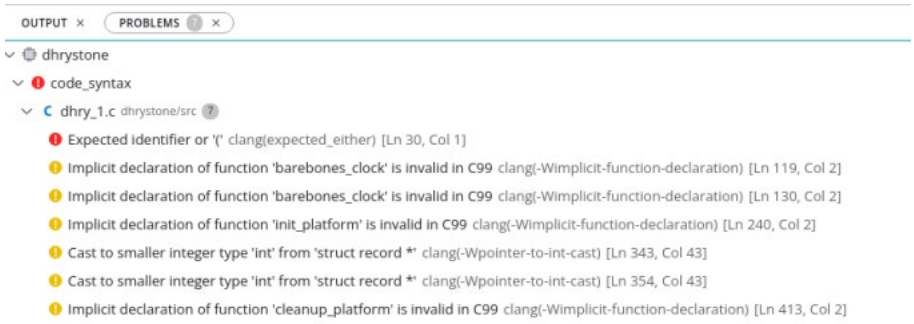
Issue and Source Code Cross-probing

You can quickly jump to source code from errors in the output console by pressing **Ctrl** and left-clicking an issue simultaneously.

Problem View

The output channel displays all warning and error messages. You can quickly navigate to the source of the error or warning in problem pane by clicking the error or warning message.

Figure 15: **Problem View**



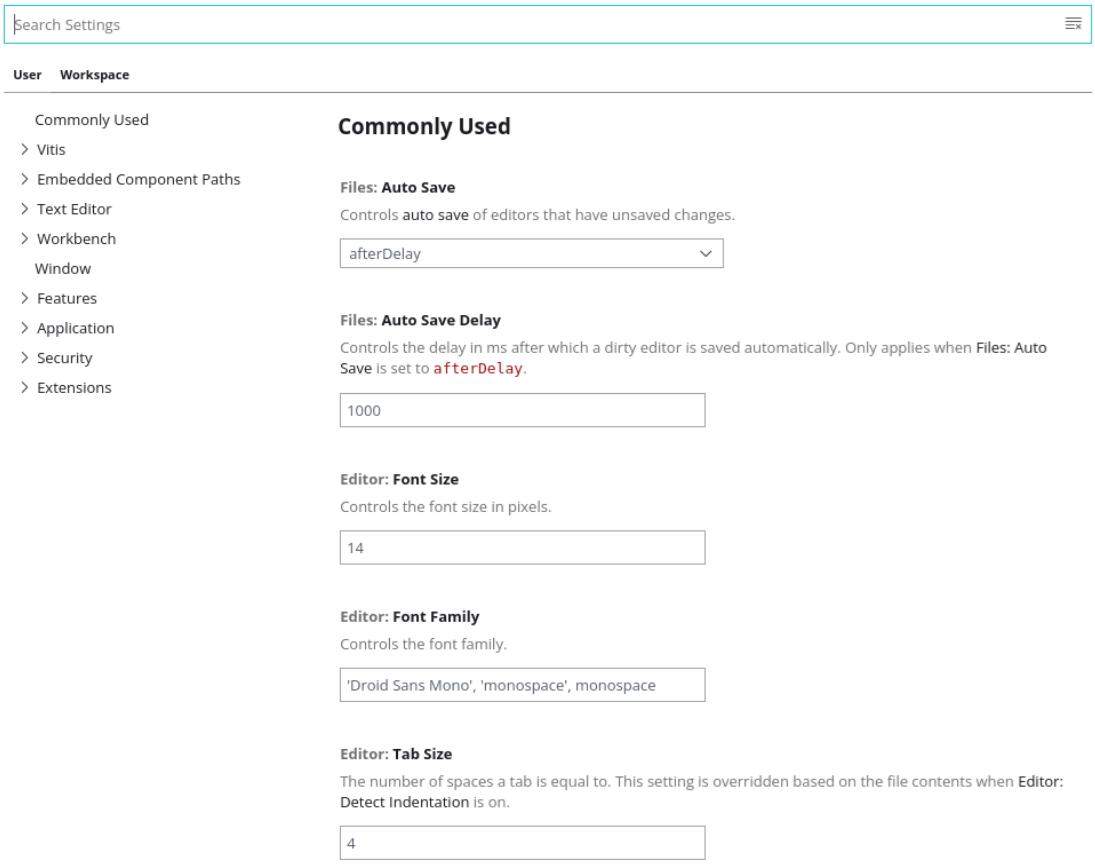
Preferences

The **File → Preferences** menu leads to user preferences that can be set and maintained to configure the Vitis Unified IDE.

Settings

- **Auto Save:** Automatically saves changes to configuration files, source code, `CMakeLists.txt` files, and more. This setting is enabled by default. You can also define Auto Save triggers and Auto Save Delay values.
- **Color Themes:** Specify a Light Theme or a Dark Theme for the Vitis Unified IDE according to your preferences, or available lighting.
- **Open Settings (UI):** Opens the Preferences view, displaying an array of options to configure your design environment.

Figure 16: **Common Preferences**



- **Open Keyboard Shortcuts:** Define keyboard shortcuts for various commands, as described in Keyboard Shortcuts, Command Palette, and Quick Find.
- **File Icon Theme:** Specify a file icon theme for your environment.

***Note*:** You can press **Ctrl + +** to zoom into the display, or **Ctrl - -** to zoom out.

Keyboard Shortcuts, Command Palette, and Quick Find

Keyboard Shortcuts

You can review and edit keyboard shortcuts from **File → Preferences → Open Keyboard Shortcuts**. Another method is by using the shortcut key Alt + Ctrl + Comma. For example:

1. Use the **Alt + Ctrl + Comma** shortcut to open the Keyboard Shortcuts window.
2. Type `build` in the search bar.
3. The Build: Hardware key-binding is Alt+Shift+BH.
4. To edit the keyboard shortcut for build hardware, hover over the line of Build: Hardware. Click the **Edit Keybinding** icon on the left and input your preferred key-binding.

Command Palette

The command palette lets you quickly find and execute commands without remembering keyboard shortcuts or menu location. For example, to run build hardware with the command palette, do the following:

1. Select your component or project in the workspace.
2. Type Ctrl + Shift + P to open the Command Palette menu.
3. Type the keyword `build`, `run` or `debug`. For example, the Command Palette filters the command names for commands matching your entry.
4. Use the mouse or the up or down arrow keys to scroll through the selection of commands. Select the command you want to execute.
5. Press Enter to execute the selected command.

For example, select an HLS component in the Component Explorer and type **Ctrl + Shift + P**, then in the Command Palette type C Syn. The HLS: C Synthesis is one of the options. Select the option and press Enter. The tool runs C synthesis on the selected HLS component.

Quick Find

Click **Ctrl + P** to open the Quick Find box. Type in a file name and the tool begins to find files that match your search string. Select a file and hit enter to open it.

With a file open, type `@` in the Quick Find box to go to a function within the opened code. You can also type `:40`, for example, to go to line 40 of the file. You can use this method to quickly find and jump to a function or line number.

With a file open, you can type `#` in the Quick Find box to go to a function within the workspace. You can use this method to quickly find and jump to a function.

Click **Ctrl + T** to open the Quick Find box. Type the symbol name and the tool begins to find the symbol that matches your search string. Additionally, it searches the files.

Parallel Compiling

The **Vitis Unified IDE** provides fast responses to actions. It employs non-blocking build commands that allow you continue working and run multiple builds at the same time.

***Note*:** Expect a slower response if you launch multiple build or emulation jobs at the same time on a slow server.

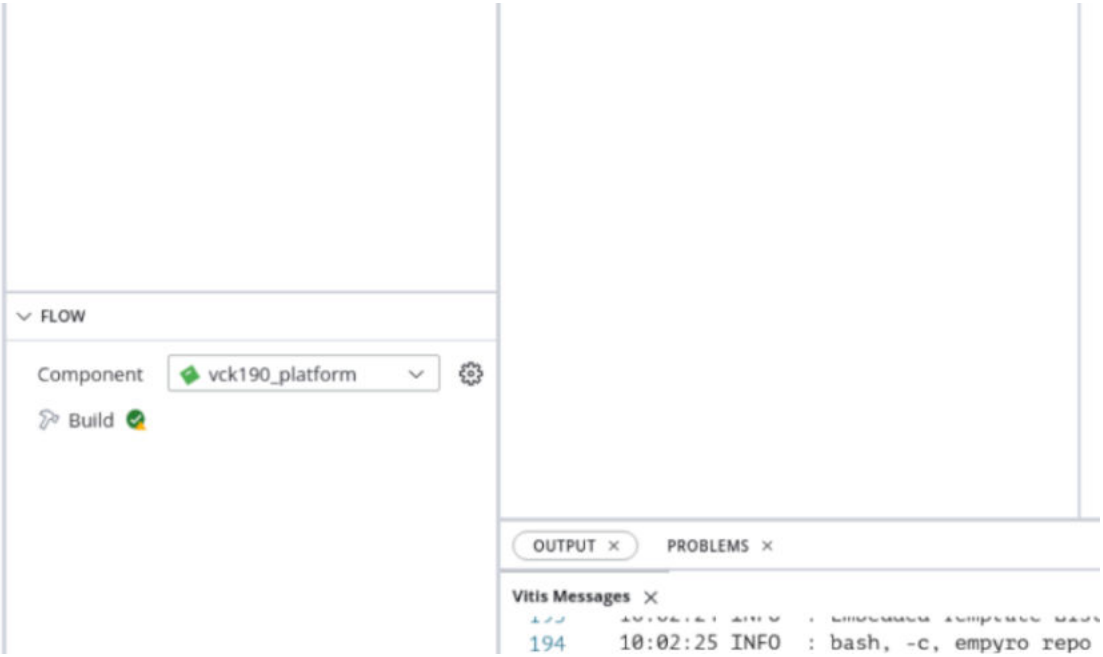
The application building job can launch multiple components in parallel. You can build and launch jobs for the referenced components in parallel.

The Vitis tool disables the Parallel Build feature by default. You can enable it by the **File → Preferences → Open Settings (UI)** command and selecting the **Vitis → Build → Parallel Build → Disable** setting.

Notification for File Change

When you modify files, the yellow indicator beside the build status icon prompts you to review and rebuild the component if necessary.

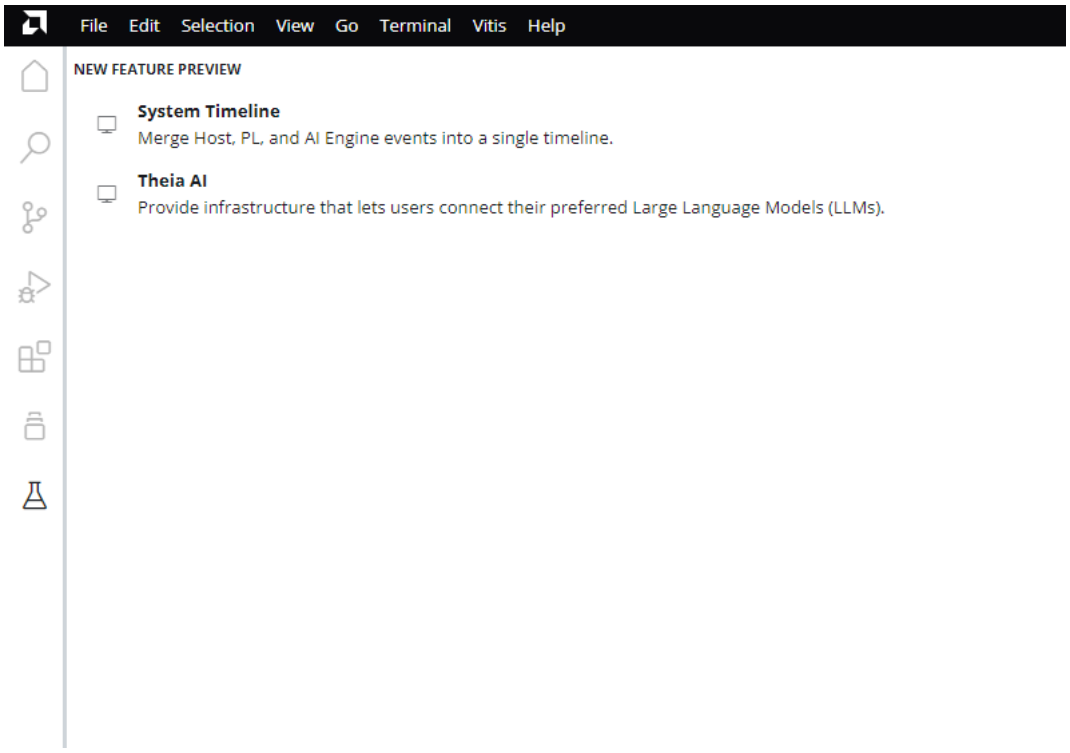
Figure 17: **Out-of-date Notification for Components**



New Feature Preview

New features are for you to explore in advance. It is possible these features do not address all use cases and require further development. You are encouraged to share your feedback. You can view the new features by clicking **Vitis → New Feature Preview**.

Figure 18: **New Feature Preview**



Before using a new feature, you need to enable it.

Disclaimer: preview features are currently in an early access phase. They are currently being developed. There is a chance these features are not be fully functional or stable. By using these features, you acknowledge and accept the following:

- Limited functionality: Early access features can have limited functionality compared to fully released features. Early access features can lack functions, have bugs, or experience performance issues.
- Potential instability: Early access features are still being tested and refined. As a result, they are prone to crashes, errors, or unexpected behavior. Use these features with caution.
- Feedback and improvements: Your feedback is crucial in improving early access features. AMD encourages you to report any issues, bugs, or suggestions you encounter while using these features. Your input helps AMD to enhance their performance and stability.
- No guarantees: Early access features are provided on an "as-is" basis, without any warranties or guarantees of any kind, whether expressed or implied. AMD does not guarantee that these features are released in their current form or at all. AMD reserves the right to modify, suspend, or discontinue these features without prior notice.
- Use at your own risk: By using early access features, you understand and accept the risks involved. AMD is not to be held liable for any damages, losses, or inconveniences arising from the use of these features.

IMPORTANT! *Carefully consider these factors before accessing and using early access features. Your participation in testing and providing feedback is greatly appreciated.*

Workspace Journal

The Workspace Journal is a log file that records back end commands corresponding to the actions triggered by the user after opening the workspace. It is a good approach to check the corresponding commands used by GUI and recreate the workspace as well. To open the Journal file, go to menu **Vitis**-> **Workspace Journal**.

You can run following command to recreate the projects with the journal file in the current workspace:

```
vitis -s workspace_journal.py
```

***Note*:** In `workspace_journal.py`, relative paths are used whenever possible to improve portability. However, there are certain instances where absolute paths are necessary. Prior to executing the script, ensure the source and destination paths in `workspace_journal.py` are properly configured and accessible.

External Lopper Support

Lopper is a backend Python utility used by AMD Vitis™ to extract hardware metadata from the system device tree. Although included in the Vitis installation, Lopper is also available on GitHub. Vitis provides an environment variable to enable the use of an external Lopper without modifying the Vitis installation. To use an external version of Lopper, run the following command:

```
export VITIS_LOPPER_INSTALL_LOC="path/to/Lopper"
```

Creating an AI Engine Component

The following section is for Versal projects only and is not applicable to datacenter projects. As described in Chapter 5: Managing the AI Engine Component in the Vitis Unified IDE, Adaptive data flow (ADF) graph applications targeting AI Engines can be developed as AI Engine components. These components can be built, simulated, analyzed, and debugged as a standalone component, and then added to a system project as part of an embedded system design, or for application acceleration in a data center. The creation of an AI Engine component is described in Chapter 5: Managing the AI Engine Component in the Vitis Unified IDE.

The use of an AI Engine component in a larger system design can be accomplished from the command line as described in Building and Running the System in the *Data Center Acceleration using Vitis* (UG1700), or in the Vitis unified IDE as described under Creating a System Project for Heterogeneous Computing.

Creating an HLS Component

In an HLS component, the tool synthesizes a C or C++ function into RTL code for implementation in the programmable logic (PL) region of an AMD Versal™ adaptive SoC, AMD Zynq™ MPSoC, or AMD FPGA device. HLS components can be built, simulated, analyzed, and debugged as a standalone component in a bottom-up design flow. The creation of an HLS component is described in Chapter 4: Managing the Vitis HLS Components in the Vitis Unified IDE.

The HLS component can be added to a System project as part of an embedded system design, or for application acceleration in a data center. The use of an HLS component in a larger system design is described in this document under Building and Running the System in the *Data Center Acceleration using Vitis* (UG1700), or in the Vitis unified IDE as described under Creating a System Project for Heterogeneous Computing.

Creating an Application Component

Embedded Application Component

An Application component can be created for the development of embedded software applications targeted towards embedded processors such as AMD Versal™ adaptive SoC, and AMD Zynq™ MPSoC, as described in the Applications in the *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400), or as described in Host Application Development in the *Embedded Design Development Using Vitis* (UG1701).

Data Center Application Component

Application components can also be created to run on x86 processors for use with accelerated systems running on Alveo Data Center accelerator cards as described in Introduction to Data Center Acceleration for Software Programmers in the *Data Center Acceleration using Vitis* (UG1700). The following steps describe creating an Application component.

1. With the AMD Vitis™ IDE opened, from the main menu select **File → New Component → Application**.

   TIP: *You can also select the **Create Application Component** command from the Welcome page.*

This opens the Name and Location page of the Create Host Component wizard.

2. Specify the Component Name and Component Location and select **Next**. This opens the Platform page.
3. Select a Platform from the list and click **Next**.

   The choice of a platform in the Application component is an important one. First, you can choose an extensible platform to support the Vitis development flow for application acceleration and heterogeneous system design as described in Introduction to Data Center Acceleration for Software Programmers in the *Data Center Acceleration using Vitis* (UG1700). Within the Vitis development flow, you can choose a Data Center acceleration card such as an AMD Alveo™ card.

   There are a number of base platforms installed with the tools, and platforms that you can install and configure.
4. Depending on the Platform selected, the Summary page is displayed for Data Center platforms, and for Embedded Platforms the Domain page is opened to let you select the processor domain for the Application component and then the Summary page is displayed. Select the domain as needed, review the summary page, and click Finish to create the Application component.

After the Application component is created the `vitis-comp.json` file is opened in the central editor window. You will need to import source files and edit the `UserConfig.make` to specify needed include files or libraries.

1. Right-click the Sources folder in the expanded view of the Application component in the Vitis Component Explorer.
2. Select **Import → Files** or **Import → Folders** and import one or more files into the component as needed.
3. In the Compiler Settings of the `vitis-comp.json` select the `UserConfig.make` to open in an editor. Scroll down to Directories, and select **Add Item** under Include Paths.
4. Click **Add files** or **Add folders** to import source files or folder for the application.

   Note: User could check the option: Copy Sources to Component to copy source files to workspace.
5. Edit these settings as needed for your source code.

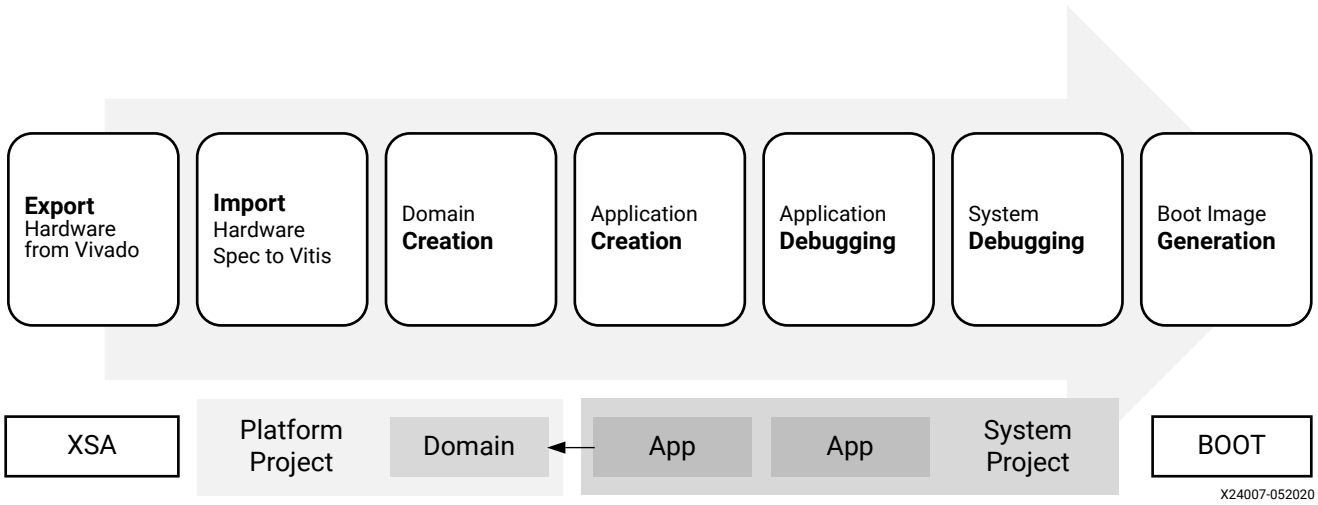
Creating a Platform Component

To create an embedded platform component, refer to Target Platform in the *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400) and successive topics for details.

Embedded Software Development Flow

The embedded software development flow is designed to be used for the development of embedded software applications targeted to run on embedded processors. The Vitis Unified IDEworks with hardware designs created with AMD Vivado™ Design Suite. The following figure shows the embedded software application development workflow for the Vitis tools.

Figure 19: **Embedded Software Application Development Workflow**



- Hardware engineers design the logic and export information required by software development from the AMD Vivado™ Design Suite to an XSA archive file.
- Software developers import the XSA into the Vitis tools by creating a platform. To unify the Vitis software platform architecture for all kinds of applications, software development projects now migrate to platform and application architecture. A platform includes hardware specification and software environment settings.
- The software environment settings are called domains, which are also a part of a platform.
- Software developers create applications based on the platform and domains.
- Applications can be debugged in the Vitis IDE.
- In a complex system, several applications can run at the same time and communicate with each other. Thus, system-level verification should be done as well.
- After everything is ready, the Vitis IDE can help to create boot images which initialize the system and launch applications.

Creating an Embedded Application Component

The embedded software flow is different from the Vitis application acceleration flow in that it requires a fixed platform or XSA to integrate your Application executable with. The embedded software flow described here requires a fixed hardware design and drivers for that hardware.

To create an Application component you can select **File → New Component → Application**.

TIP: *Alternatively you can select **Create Embedded Application** from the Embedded Development group of the Welcome page.*

1. Input the Application component name and specify the location. Click **Next**.
2. Select a fixed platform and select **Next**.

   Note: If you do not have an existing fixed platform, refer to Chapter 4: Managing the Vitis HLS Components in the Vitis Unified IDE to create a platform. If your target platform is not in the list, you can add it by clicking the '+' button.
3. Select the processor domain to run your application in, and select **Next**.
4. Review the summary page of the Application component and select **Finish** to create the component.

With the Application component created for the specific processor domain of a fixed platform, you can now add the source files and include files to the component as described in Creating an Application Component.

Command-line to Create an Embedded Application Component

- **Lanuch Vitis mode:**

```
vitis -i
```

- **Import Vitis library:**

```
import vitis
```

- **Create a client:**

```
client = vitis.create_client()
```

- **Create an Application component:**

```
comp = client.create_app_component(name="< >", platform = "< >", domain
= "< >", template = "< >")# <> is the name of host component, platform
path, domain and the template mode.
```

- **Build the Application component:**

```
comp.build()
```

Command-line example to create a standalone Application component:

```
import vitis
client = vitis.create_client()
client.set_workspace(path="/ref_files/workspace")
comp = client.create_app_component(name="host_component", platform
= "/ref_files/embedded_application/platform/export/platform/platform.xpfm",
domain = "standalone_psu_cortexa53_0", template = "host_component")
comp.build()
```

Save the example code in a single file, such as build.py, then execute the command below to create the Application component:

```
vitis --new -s build.py
```

Creating a System Project for Heterogeneous Computing

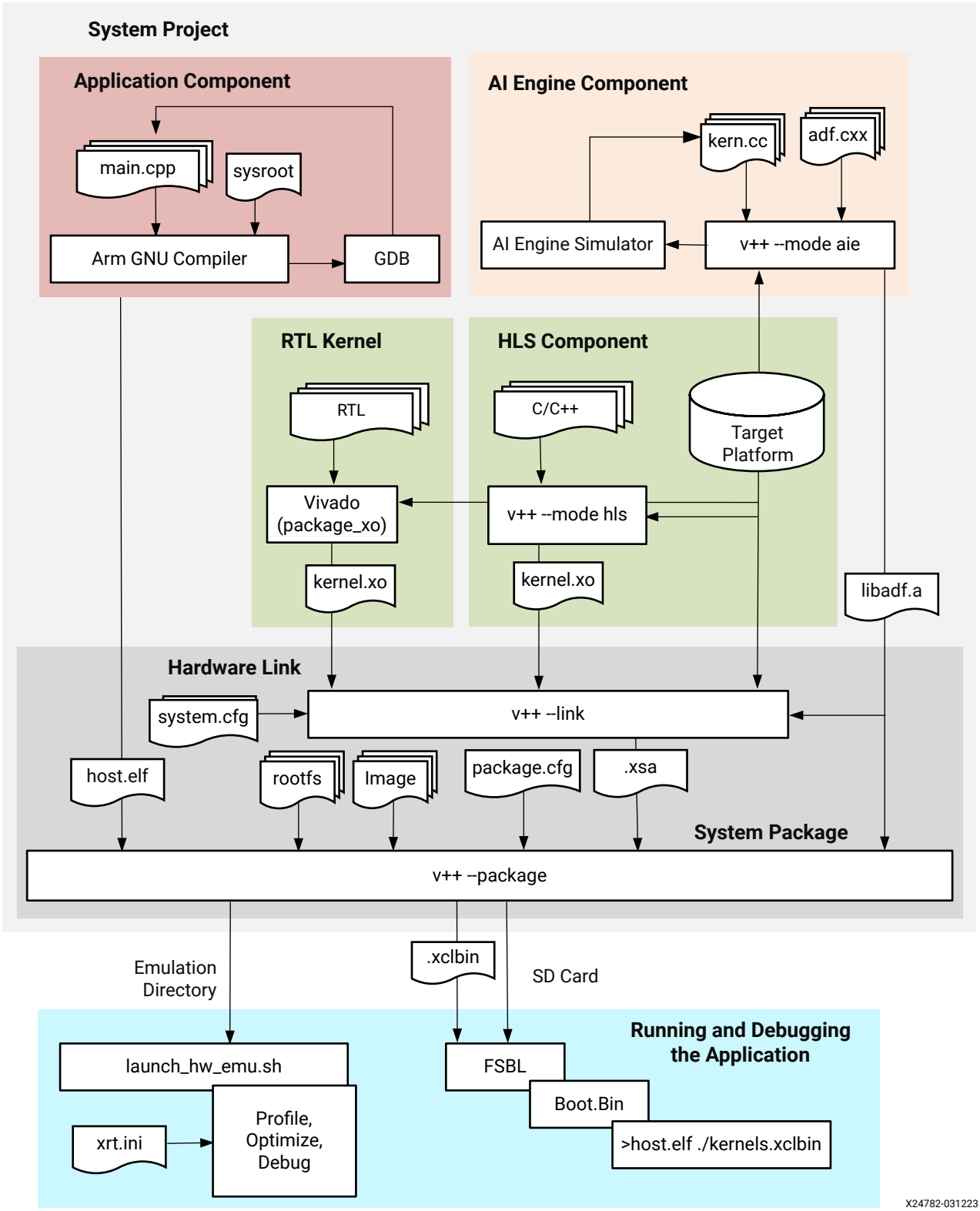
The Vitis Unified IDE manages designs at the component level and the System project level. The top-level System project can contain different components all of which can be separately developed, built, and analyzed in the new Vitis Unified IDE.

Embedded System Project

For a system project targeting embedded platform, the following components are integrated into a single heterogeneous system as shown in the next figure.

- Application Component for PS
- AI Engine Component (graph)
- HLS Component
- RTL Kernel
- Platform Component

Figure 20: **Vitis Accelerated Embedded Flow**

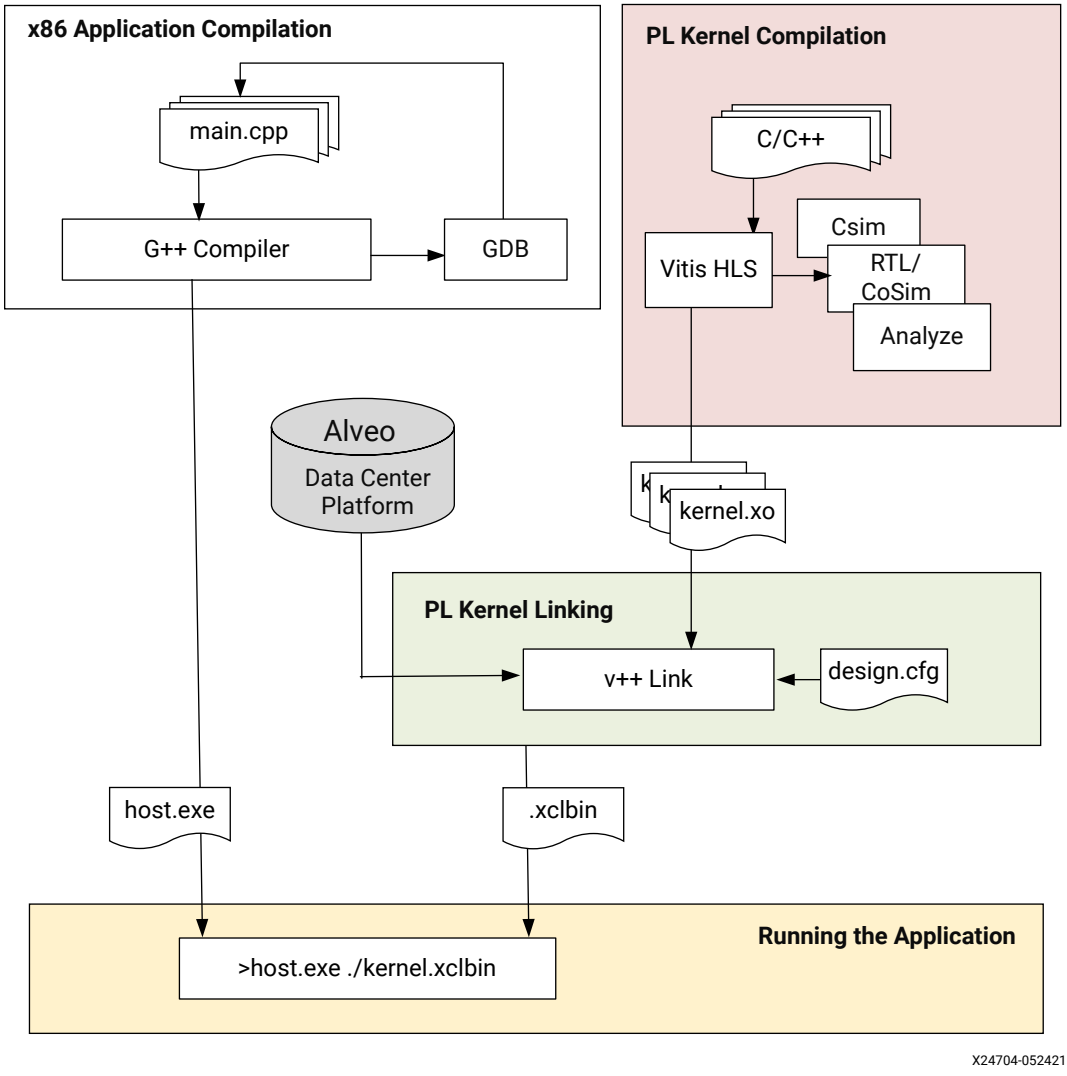


Data Center System Project

For a system project targeting data center acceleration using AMD FPGA-based Alveo Accelerator cards, the following components are integrated as shown in the figure below:

- Application Component running on x86
- C/C++ HLS Component (PL Kernel)

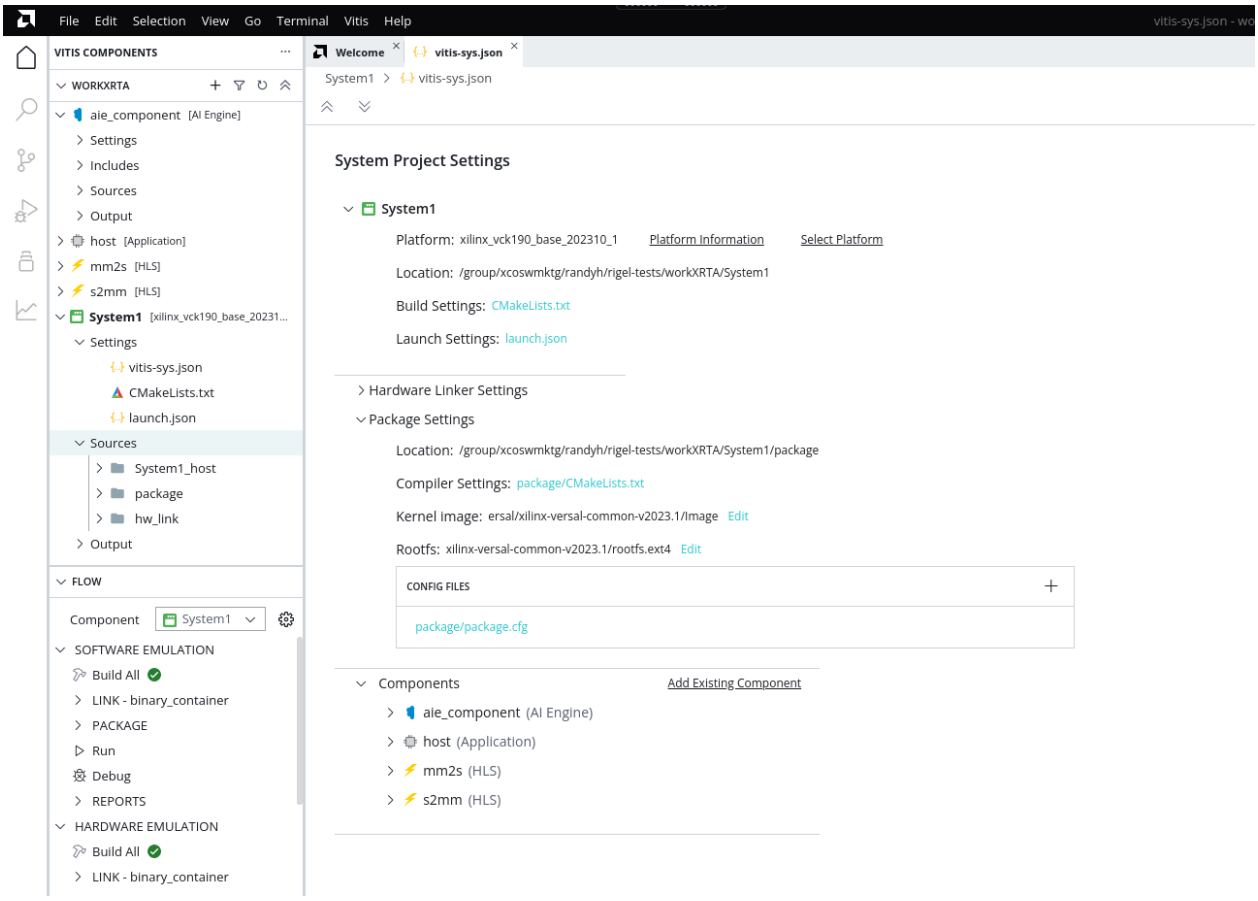
Figure 21: **Vitis Accelerated Data Center Flow**



System Project Structure

The Vitis IDE System project contains components: platforms derived from Platform components, HLS components, AI Engine components, Application components. The System project refers to the components, defines and stores the linking configuration required to build the system, and defines and stores the packaging configuration required to package the system. In the figure below you can see the elements of the System project under the System Project Settings.

Figure 22: **Project Structure**



- **Application Component:** Software applications that run on x86 CPU for data center acceleration, or on Arm® processors for embedded systems. The host program interacts with the AI Engine kernels and kernels in the PL region.
- **HLS Component:** Programmable logic (PL) kernels that run on FPGA fabric written with C/C++ code and synthesized into RTL code. HLS components are compiled for implementation in the PL region of the target platform using the `v++ --compile --mode hls` command.

- **AI Engine Component:** Asynchronous dataflow graphs and kernels that run on AI Engines of specific Versal devices. AI Engine components are compiled into libadf.a files using the `v++ --compile --mode aie` command.

  ***Note*:** This setting is for the embedded flow only.

- **Platform Component:** Defines a custom platform for use in embedded system designs. The acceleration flow also supports the inclusion of predefined platforms installed with the system, or downloaded separately. Custom platforms can be created as described in Integrating the System in the *Embedded Design Development Using Vitis* (UG1701).

- **HW Linker Settings:** The hardware link process connects the PL kernels and/or AI Engine graph with the platform and builds the hardware system device binary. The `hw_link/binary_container.cfg` file defines the configuration settings for the linking process.

- **Package Settings:** The Package settings define the files required to package the finished system for use. The system package process uses the `v++ --package` command to gather the required files to configure and boot the system, to load and run the application, including the AI Engine graph and PL kernels. This builds the necessary package to run emulation and debug, or run your application on hardware.

  ***Note*:** This setting is for the embedded flow only.

Files for project settings are stored in the top directory.

- **`vitis-sys.json` and `vitis-comp.json`:** This is the top level settings file for applications and components. It defines the components of the application, their source code and the compiler settings file.

- **CMakeList.txt:** The Vitis IDE uses CMake to manage the project. The top level `CMakeList.txt` defines the global build parameters and adds the component level of the build settings `CmakeList.txt` as sub-modules. You can modify the settings in `CMakeList.txt` by editing content defined between `START OF USER SETTINGS START` and `END OF USER SETTINGS`

- **Component/compile_commands.json:** The auto-generated file to enable IntelliSense support using Clang.

- **Component/config.cfg:** The configuration file for PL kernel compilation. It is passed to `v++` compiler.

Build output for each target is stored in its own folder under the `./build` directory. For example, `aie_component/build/x86sim`

The runtime configuration fle (`xrt.ini`) is stored in `<project>/<application_component>/runtime/<target>_xrt.ini`. Where `<target>` is hardware emulation, or hardware.

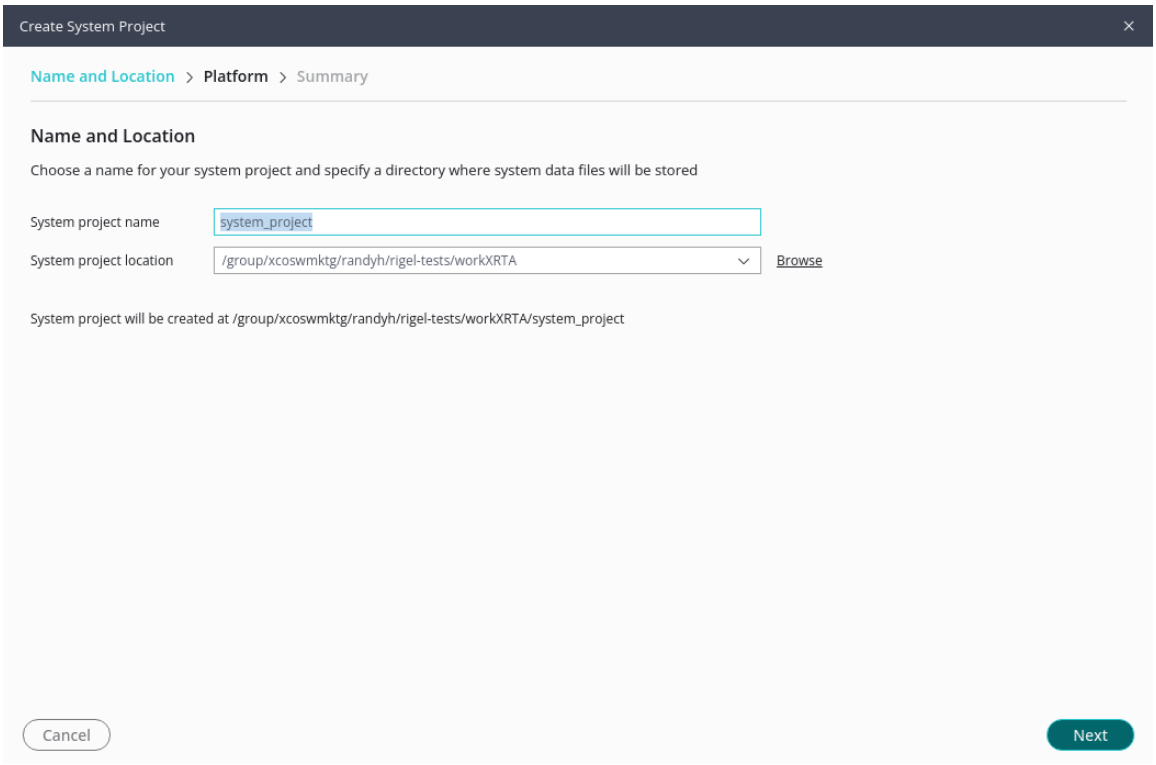
Creating a System Project

There are a number of ways to create a System project in the new Vitis IDE. The following steps describe a recommended approach.

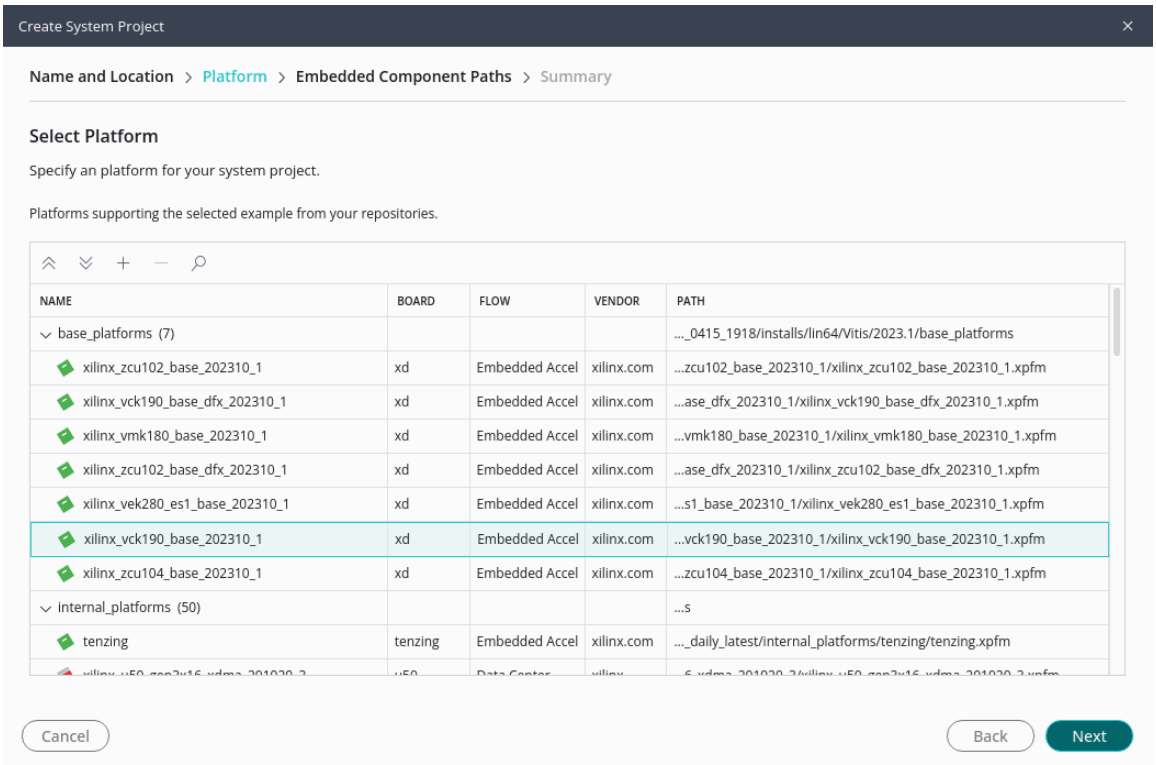
1. With the Vitis IDE opened, from the main menu select **File → New Component → System Project**.

   TIP: *You can also select the **Create System Project** command from the Welcome page.*

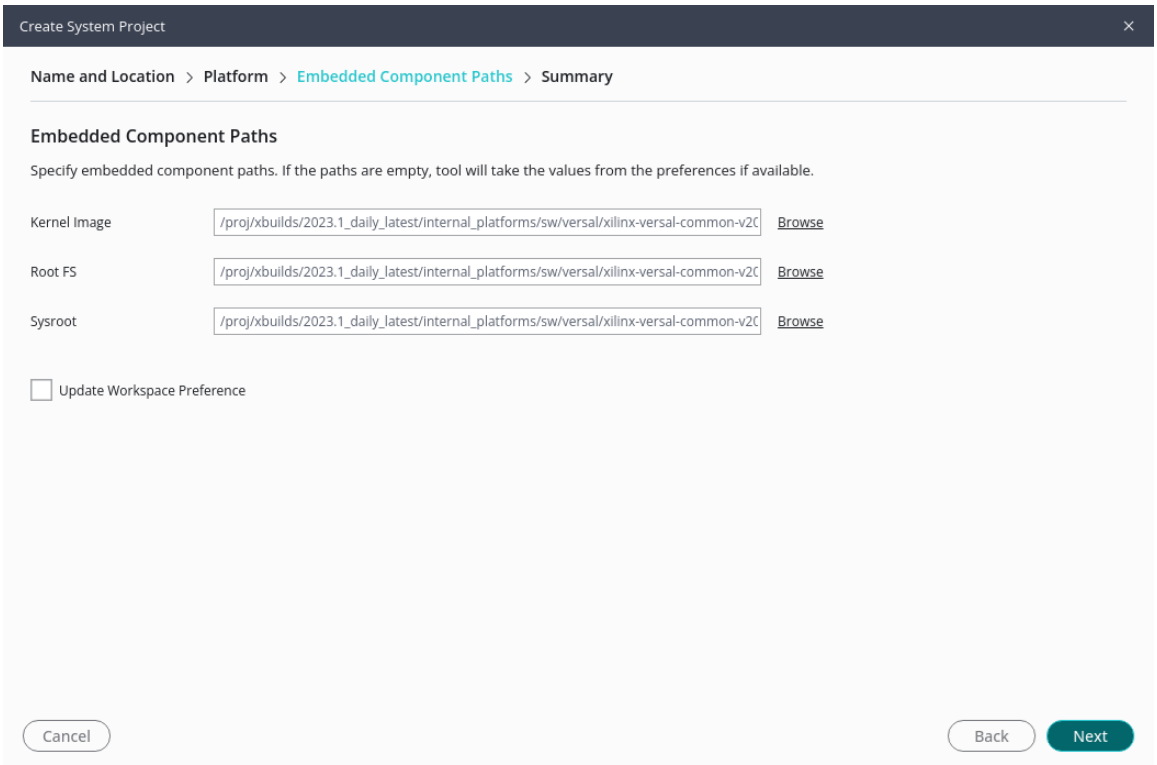
   This opens the Name and Location page of the Create System Project wizard as shown below.



2. Enter a Component name and Component location and select **Next**. This opens the Platform page of the wizard as shown below.



3. Specify a platform to use for the System project and select **Next**. If you select an embedded platform, this opens the Embedded Components Path page of the wizard as shown below. If you select an AMD Alveo™ Data Center accelerator card, then the Summary view is displayed.



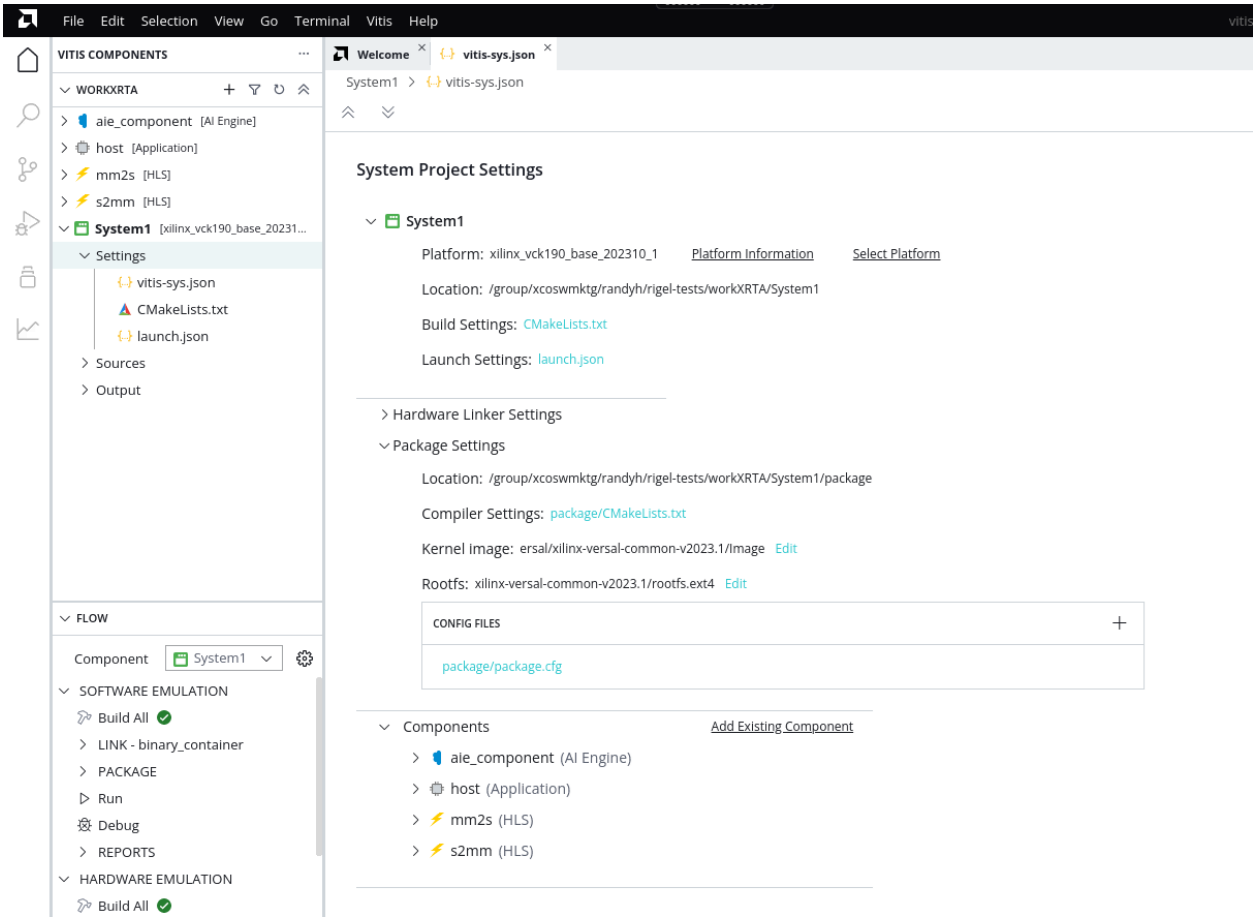
The Embedded Component Paths specify the Kernel Image file, the Root FS, and the Sysroot required by the embedded system for compiling embedded applications, and for booting and configuring the embedded systems. These paths can be populated by the Preferences view, or can be manually entered when the System project is created.

> **TIP:** *The Update Workspace Preference check box lets you specify that any manually entered paths should also override or update the Preferences view.*

4. Click **Next** to display the Summary page.

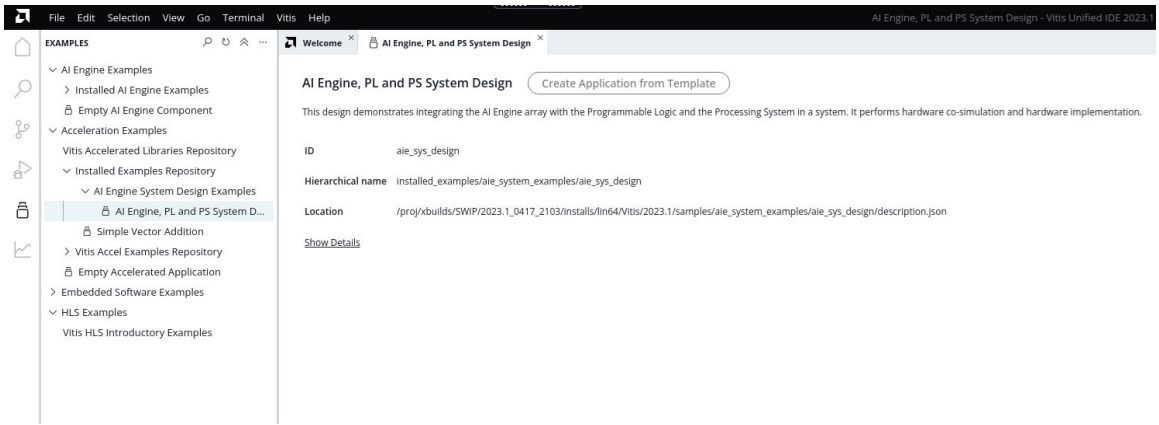
   The Summary page displays the choices you have made on the prior pages. Review the summary and select **Finish** to create the System project, or select **Back** to return to earlier pages and change your selections.

When the System project is created the `vitis-sys.json` file for the System is opened in the central editor window, and the System project becomes active in the Flow Navigator. You can see from the red text and the red dot next to the System in the Component Explorer that it is missing key components. Provide the missing components in the next steps.



Creating a System Project from Examples

You can create a System project from the Examples installed with the Vitis IDE.



1. In the Welcome page, click **Examples**. The Examples panel appears as shown above.

2. Select the System project or Component you want to create. There are AI Engine Components, and AI Engine System Projects combining AI Engine components with HLS components, and an Application component to run on the processor. There is a Simple Vector Addition example, and you can download additional example repositories from GitHub, such as the Vitis Accel Examples.
3. When you select an Example it opens the Application Template in the central window. Select **Create Application from Template**.
4. This opens the Create System Project wizard for you to specify the System name and location, and platform. The platforms are restricted by the example you have selected. Click **Finish** from the Summary page to create the example System project.

After the System is created you can configure the project as described in the following sections.

Managing Components in System Projects

After creating the System project you can add components to the design. As previously discussed, a System project can contain multiple components.

You can add components to the System Project Settings in the open `vitis-sys.json` by selecting the **Add Existing Component** command under the Component heading; or you can add them directly to the Binary Container under the Hardware Linker Settings heading.

1. Add the components by selecting the **Add Existing Component** command:
   - Select the type of component to add in the pop-up window: HLS, AI Engine, Host.
   - After you select the type of component to add the tool will present a list of all the components of the selected type in the current workspace. Choose the component to add.
   - If there is only a single binary container the tool will automatically assign it. For DFX platforms that support multiple device binaries you can select the target binary container. If the Application does not have any binary container select the **Container Name** to create a new binary container.
2. Add components directly to the Binary Container:

   ***Note*:** The Host component is the only type that cannot be added directly to the binary container. You must add it under the Component heading.

   - You can select the **Add Pre-Built Binary** command in the Binary Containers configuration box. This command lets you add the `kernel.xo` file or `libadf.a` file from a previously compiled HLS component, RTL kernel, or AI Engine component. The pre-built binary can be added from folders outside the current workspace. Selecting this command opens a file browser letting you navigate to and select the file of interest.

     TIP: *This method can be used to add a `libadf.a` file that was generated by Model Composer into the System project.*

- You can also select to add the component type in the Binary Containers configuration box. You can add Kernels or AI Engine Graphs. This opens an Add Components dialog box that lists all the components of the selected type in the current workspace.

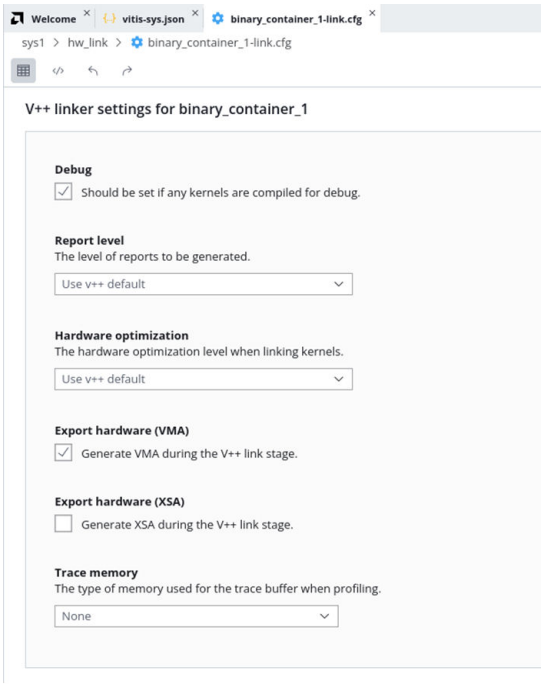
TIP: *To remove a component from the System project, you can either select to delete it from the Binary Containers or from the Components. Either will result in the removal of the component from the project.*

Defining the HW_Link System Configuration

V++ Linker Settings

The Hardware Link (hw_link) configuration file contains information used by the Vitis compiler (v++) for linking the system. To open the config file from the `vitis-sys.json` file for the System project click the **hw_link/binary_container-link.cfg** hyperlink. This will open the Config File Editor in the central editor window as displayed below.

Figure 23: **Configuring the Hardware Linker Settings**



***Note*:** All of the options found in the `binary_container-link.cfg` are `v++` command line options (see v++ Command).

The V++ linker settings provides the following options:

- **Debug:** Enable debug file generation. When not enabled the HLS component and the system design can be optimized but will not support debug.

Note: This specifies the `-g` option for the `v++ --link` command as described in v++ General Options. To enable debug for the HLS component you must specify `syn.debug.enable` in the HLS component config file (`hls_config.cfg`).

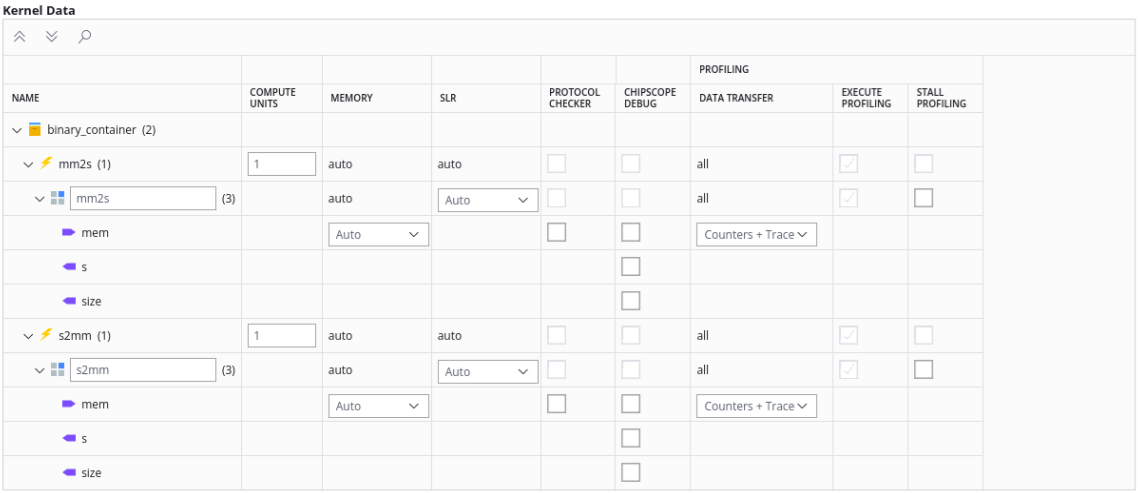
- **Report Level:** Specify the level of detail; included in reports generated by the `v++ --link` command.
- **Hardware Optimization:** This option specifies the optimization level of the AMD Vivado™ implementation results.
- **Export Archive:** Indicates the export of the `.vma` file for use in the Vivado Design Suite as described in Packaging for Vitis Export to Vivado Flow in the *Embedded Design Development Using Vitis* (UG1701).

  Note: This feature requires the use of a Versal platform or XSA designed with a block design container. If your system project does not meet this requirement, the extensible XSA will return an error. BDC (Block Design Containers) are not required.
- **Export Hardware:** Specifies that a Fixed XSA should be generated from the linked design produced by the v++ command. This is the default output for Versal devices.
- **Trace Memory:** When building the hardware target for the System project, this option lets you specify the type and amount of memory to use for capturing trace data. This option relates to the `--profile.trace_memory` command as described at --profile Options.

Kernel Data

For each PL kernel in the System project there are additional settings available under the Kernel Data heading.

Figure 24: **Configuring Kernel Data**



The Kernel Data section refers to config commands that specifically apply to the PL kernels generated from the HLS component, the ports and interfaces, and the debug and profile options available.

The specific options that can be set include:

- CU Name: Lets you define the naming sequence when multiple compute units are generated from a PL kernel as described under the --connectivity Options.
- Compute Units: Specifies the number of compute units to generate from a PL kernel. Also related to `--connectivity.nk`
- Memory: Use this option to specify the connection of kernel ports to system ports within the platform. A primary use case for this option is to connect kernel arguments to specific memory resources as described in Mapping Kernel Ports to Memory in the *Data Center Acceleration using Vitis* (UG1700).
- SLR Assignment: is described in Assigning Compute Units to SLRs on Alveo Accelerator Cards in the *Data Center Acceleration using Vitis* (UG1700)
- Protocol Checker: Described under --debug Options.
- Chipscope Debug: Also described under --debug Options.
- Data Transfer: Enables monitoring of data ports through special monitor IP that are added into the design for profiling. This option relates to the --profile.data command
- Execute Profiling: This option records the execution times of the kernel and provides minimum port data collection during the system run. This option relates to the `--profile.exec` command.
- Stall Profiling: This option relates to the `--profile.stall` command. It adds stall monitoring logic to the device binary (`.xclbin`) which requires the addition of stall ports on the kernel interface. Therefore the `stall` option must be specified during both compilation and linking.

> **IMPORTANT!** *The `--profile.stall` command must be specified during compilation of the HLS component using `syn.rtl.kernel_profile=1` in the HLS component config file (`hls_config.cfg`) to be implemented during `v++` linking.*

Adding Commands Using the Text Editor

After selecting config items using the Config File editor GUI features, you should take a look at the Text Editor view. You can toggle between the GUI table view and the Text Editor view by using the commands in the toolbar.

Examine the configuration file commands that have bean added by the GUI, and be aware of any configuration commands to add to your design that are not presented by the GUI. The Text Editor offers the ability to add configuration commands, whether they are in the GUI or not.

An example of this applies to the use of the AI Engine component in a System project. The AI Engine component must be connected into the platform or to PL kernels using the `--connectivity.sc` command. You can also map the system port to a memory using the `--connectivity.sp` command to specify a connection of a GMIO to a specific memory.

TIP: *Keep in mind that the GUI does not provide access to all the command options for linking the system. Think about which options your design will need to use. The Text Editor provides a good way to enter them.*

Adding AI Engine Connectivity

As explained in Linking the System in the *Data Center Acceleration using Vitis* (UG1700), for AI Engine graph applications the Vitis compiler needs some instruction on how to make connections between the graph and PL kernels. The number of kernels to be instantiated is already configured in the IDE. However the definition of the connections between the PL kernels and the graph must be specified in the config file as shown below:

```
[connectivity]
stream_connect=mm2s_1.s:ai_engine_0.DataIn1
stream_connect=ai_engine_0.clip_in:polar_clip_1.in_sample
stream_connect=polar_clip_1.out_sample:ai_engine_0.clip_out
stream_connect=ai_engine_0.DataOut1:s2mm_1.s
```

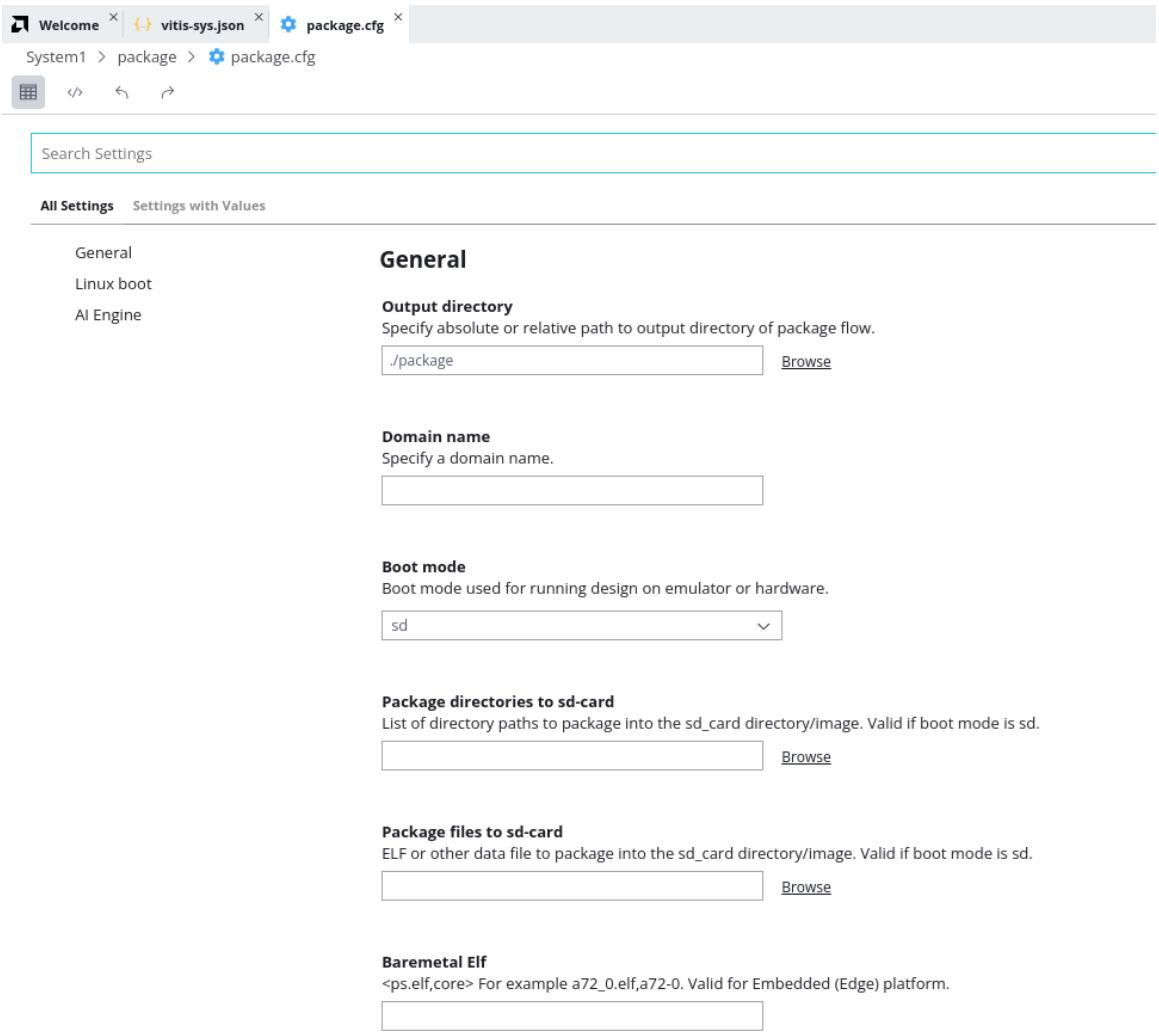
The connection scheme must be defined in the Text Editor, or manually added to the config file for the system project. Connections can be defined as the streaming output of one kernel connecting to the streaming input of a second kernel, or to a streaming input port on an IP implemented in the target platform.

The `system.cfg` file has some differences between the Vitis IDE and the command-line flow. The primary difference is that the IDE provides the connectivity `nk` options of the config file, instantiating a specified number of compute units (CUs) per kernels, as specified in the build settings. In addition, the IDE uses a naming convention for CUs in the form of `<kernel>_#`, where `#` indicates the CU instance. In the case where there is only one CU instance, it has the `_1` extension. This means that for the Vitis IDE, the `system.cfg` should not specify the `nk` option, and the `sc` option should use the CU instance name from the IDE.

Defining the Package Configuration

The **Package Settings** configuration file contains information used by the Vitis compiler (v++) for packaging the system. To open the config file from the `vitis-sys.json` file for the System project, click the **package/package.cfg** hyperlink. This will open the Config File Editor in the central editor window as displayed below.

Figure 25: **Configuring Package Settings**



The packaging process is described in Integrating the System in the *Embedded Design Development Using Vitis* (UG1701). The package commands mentioned below are documented in --package Options.

The Package config file includes the following sections:

- **General:** Applies to the construction of the SD card or boot files.
  - Output Directory: Specifies the absolute or relative path to the output directory of the `--package` command. The default output directory for the Vitis IDE is `./build/<target>/package`.
  - Domain Name: Specifies a domain name from the current platform. If this option is not specified, then the tool uses the default domain for the platform.

- Boot Mode: Specifies the boot mode used for booting the device and running the application in emulation or on hardware. For embedded platforms, the default boot mode is SD card.
- Package directories to sd-card: Specifies a directory to package as a sub-folder of the SD card directory/image.
- Package files to sd-card: Specifies an `elf`, `xclbin`, or other files to package into the SD card directory/image.

  ***Note*:** You can include more than one file in the text mode (click the **<>** icon to enter text mode):

  ```
  [package]
  out_dir=./package
  sd_dir=../../../data
  enable_aie_debug=true
  kernel_image=/versal/xilinx-versal-common-v2025.2/Image
  rootfs=/versal/xilinx-versal-common-v2025.2/rootfs.ext4
  sd_file=../../../b.txt
  sd_file=../../../a.txt
  ```

- Baremetal Elf: This option relates to `--package.ps_elf` and is used when a baremetal ELF file is running on a device processor core.
- PS Debug Port: Specifies the TCP port where emulator listens for incoming connections from the debugger to debug PS cores.

- **Linux Boot:** Specifies options to configure the device boot process for Linux.
  - Arm Trusted firmware elf: This option is related to `--package.bl31_elf`, and identifies a trusted firmware ELF that executes on the A72#0 core.
  - DTB file: Specifies the absolute or relative path to device tree binary (DTB) used for loading Linux on the APU.
  - U-Boot ELF: Specifies a path to U-Boot ELF file which overrides the platform U-Boot.
  - Image Format: Specifies `<ext4 | fat32>` output image file format used on the SD card. For embedded platforms with a Linux domain, the default image format is `ext4`. For all others, the image format is `fat32`.
  - Kernel Image: Specifies the absolute or relative path to a Linux kernel image file.
  - Root File System: Specifies the absolute or relative path to a processed Linux root file system file.
  - Do Not Create Image: Bypass SD card image creation when `--package.boot_mode sd` is used.

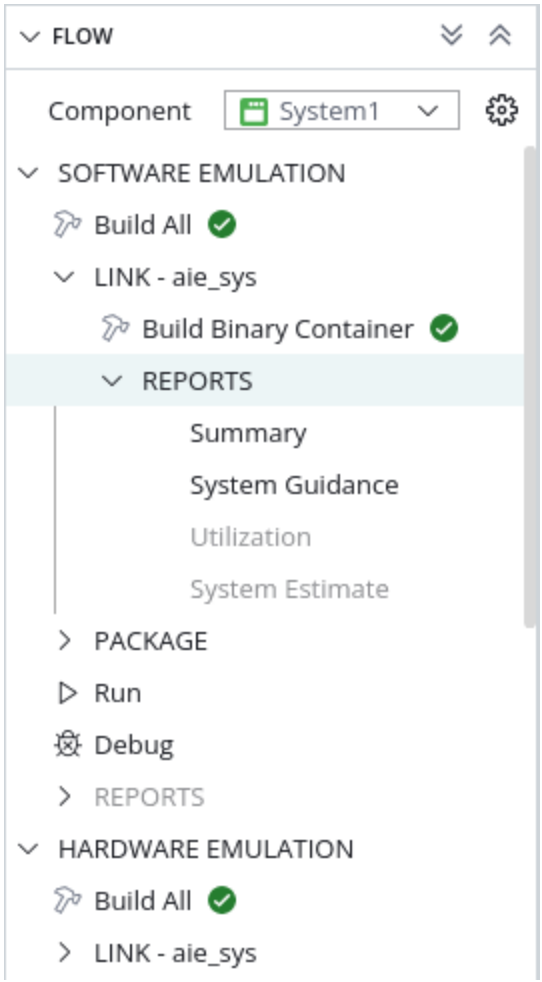
- **AI Engine:** Defines features related to the AI Engine array.
  - Debug port: Specifies a TCP port where emulator listens for incoming connections from the debugger to debug Versal AI Engine cores. The default port value is `10100`.
  - Do not enable cores: Specifies that the Versal AI Engine cores are enabled by an embedded processor (PS) application rather than by default at boot time.

- Enable debug: When enabled, the tool generates CDO commands to stop the AI Engine cores during PDI load.

Building the System Project

After creating the System project you can select it in the Component Explorer, or choose it from Component drop-down in the Flow Navigator to make it the active component in the tool. Then select **Build All** under the Hardware Emulation, or Hardware headings as shown in the following image.

Figure 26: **Building the System Project**



The Build All option combines the Build Binary Container command to run the `v++ --link` command as described in Linking the System in the *Data Center Acceleration using Vitis* (UG1700), and the Build Package command to run the `v++ --package` command as described in Packaging for Vitis Flow in the *Embedded Design Development Using Vitis* (UG1701), into a single process step. Alternatively, you can run these as separate process steps. You can expand the Link section and select **Build Binary Container** to build only the `.xclbin` or the `.xsa`. Then also expand the Package section and select **Build Package** command to complete the build process.

TIP: *Both the Link and Package commands have their own configuration files that can be managed from the System project `vitis-sys.json` file as previously described.*

The System project supports two build targets.

1. Hardware Emulation is a build for simulation of the hardware design in the Vivado logic simulator
2. Hardware is a build that is intended to be run on the physical device.

The builds take progressively longer to complete as you move from 2 to 3. You can use the hardware emulation to get a cycle accurate simulation of the hardware, finally build and run your hardware for the real performance test.

A transcript of the build process is displayed in the Output console, titled with the System project name and the build target, for example `System1::hw_emu`. When the build is complete you can review the transcript, or the `hw_linker.log` file written to the `<system>/build/<target>/log` folder. You should see `Build Finished successfully` in the transcript, or you could see errors reported instead. The Flow Navigator displays a green circle with a check mark in it, or a red circle with an x depending on the results of the build. If the build completes with errors, review the transcript or the log file to determine the cause and rerun the build as needed.

You can select and expand the folders of the build directories. You can see the output files of the Vitis compiler package process (`v++ --package`) in the output hierarchy. The build process generates the emulation data and boot files needed for the system, and writes them to the `sd_card` folder.

***Note*:** Two folders are created under the Hardware folder, `package` and `package_no_aie_debug`. The `sd_card.img` file within the `package` folder is for hardware debug purposes whereas the `sd_card.img` file in the `package_no_aie_debug` folder is for regular application execution.

After a successful build the Flow Navigator will display a series of reports generated during the compilation process. You can access the reports by expanding Reports under the Link and Package sections. The available reports will vary depending on the build target. You can select any of the available reports to view, or switch to the Analysis view to complete a review of the reports. Refer to Chapter 8: Working with the Analysis View (Vitis Analyzer) for details.

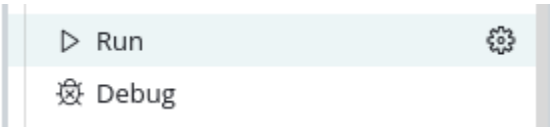
After the build completes successfully you can choose to **Run** or **Debug** the System project as described in the next sections.

Launching Run or Debug of the System Project

Once the System project has been built, you can launch run or debug with the following steps.

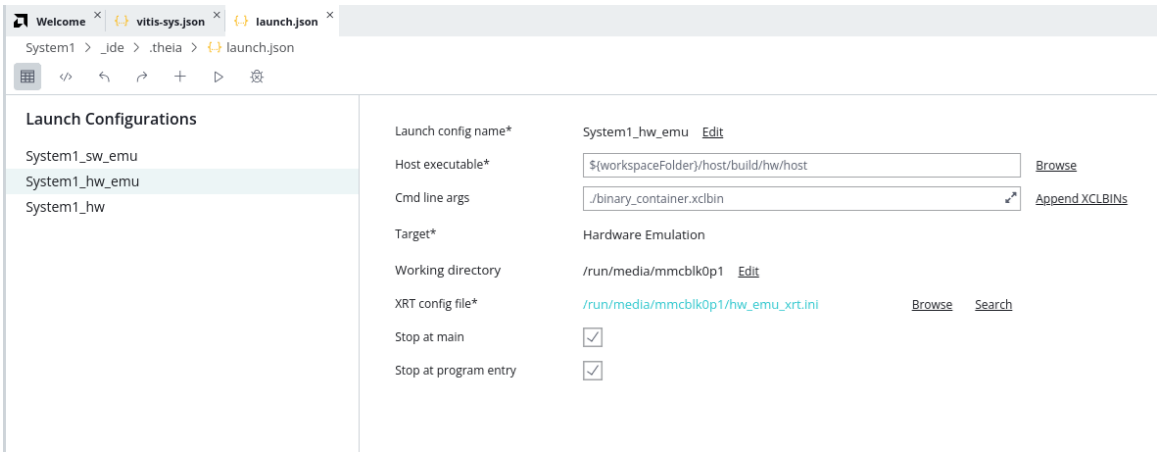
1. Open the Launch Configuration editor (`launch.json`) for the System project by either selecting it from the Settings folder of the project in the Component Explorer, or select **Open Settings** next to Run or Debug in the Flow Navigator.

   TIP: *The Open Settings command is a hidden icon that appears when you place your cursor in the Flow Navigator next to either Run or Debug.*

2. Edit an existing launch configuration, or create a new one to modify to establish the specific requirements for the run or debug session you are about to start. You can define separate launch configurations for the three types of targets supported for the System project. The build targets include Hardware Emulation (`hw_emu`), and Hardware (`hw`) as described in Selecting the Build Target in the *Data Center Acceleration using Vitis* (UG1700).

   Figure 27: **Launch Configurations**

   The preceding figure shows the System project Launch Configuration editor with the options available to configure the System project for Run or Debug.

   The preceding figure shows the System project hardware emulation launch configuration which includes the following settings:

   - Name: The name of the launch configuration.
   - Host Executable: the name of the host application contained in the Application component.
   - Cmd Line Args: arguments to be passed to the host application, including the binary container (`.xclbin`) if the code requires it.

- Target: The build target that launch configuration should be used with.
- Working Directory: This is the working directory that the host application is run from.
- XRT Config File: This is the location of the `xrt.ini` file to be used when running the application. Refer to xrt.ini File for details on the available options.
- Stop at main: Enable checkbox to add a breakpoint at the entry point to the main application for debugging.
- Stop at program entry: Enable checkbox to add a breakpoint at the entry point to AI Engine program.

After configuring the launch configuration, you can select Run or Debug commands in the Launch Configuration editor, or by selecting Run or Debug from the Flow Navigator. If multiple launch configurations are available for the build target you are running, the Vitis IDE will prompt you to select a launch configuration to use.

IMPORTANT! *For embedded platforms running the QEMU environment you must launch the emulator prior to running or debugging the system.*

For hardware emulation of embedded devices or embedded platforms you must start the QEMU environment by selecting the **Emulator** command from the Flow Navigator prior to selecting Run or Debug. The QEMU environment takes a few minutes to launch. You should wait to launch Run or Debug until the QEMU is up and running.

When launching hardware emulation, you can specify options for the AI Engine simulator that runs the graph application, as described in *Reusing AI Engine Simulator Options* in *AI Engine Tools and Flows User Guide* (UG1076). The options can be specified in the Emulator Arguments field by specifying the following command.

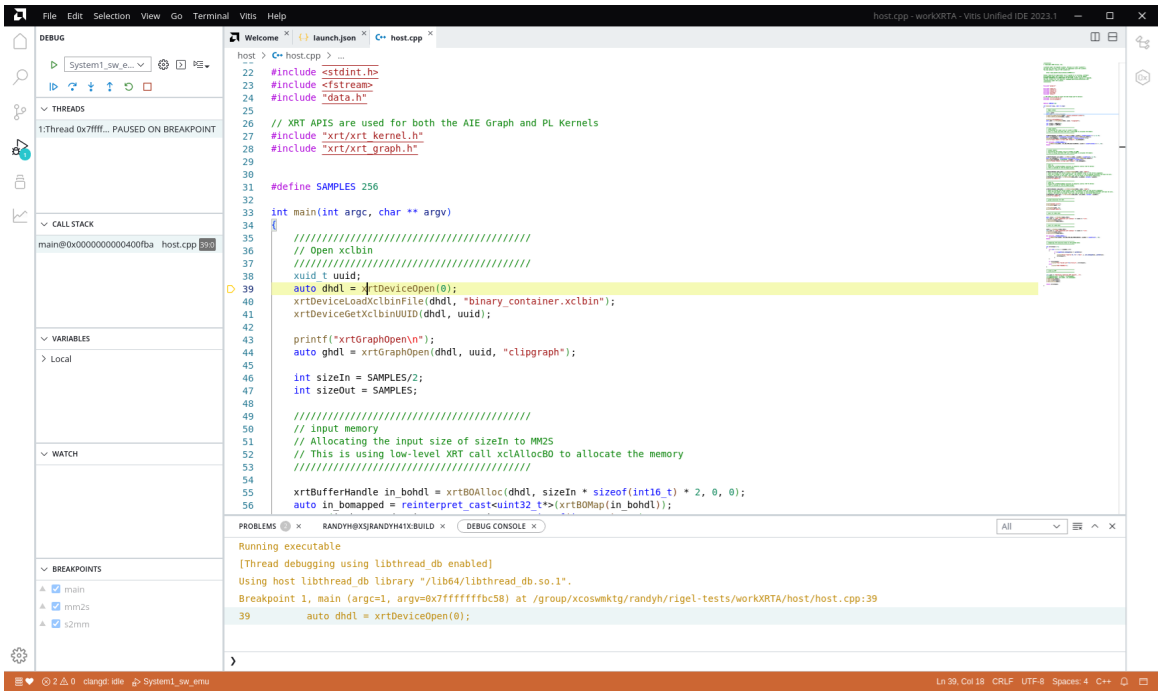
```
-aie-sim-options ${FULL_PATH}/aiesim_options.txt
```

Debugging the System Project

After you have launched the Debug view for an System project, you will see several windows or views displayed as shown in the following figure. The Debug view shows the state of cores that are being debugged. It shows the source file and line number where the debugger stops and what action it is taking (breakpoint/step over/...).

If your System project includes AI Engine components, refer to the Debugging the AI Engine Application chapter in the *AI Engine Tools and Flows User Guide* (UG1076). If your System project includes PL kernels and Application components only, then refer to for additional information.

Figure 28: **Debug View - System Project**

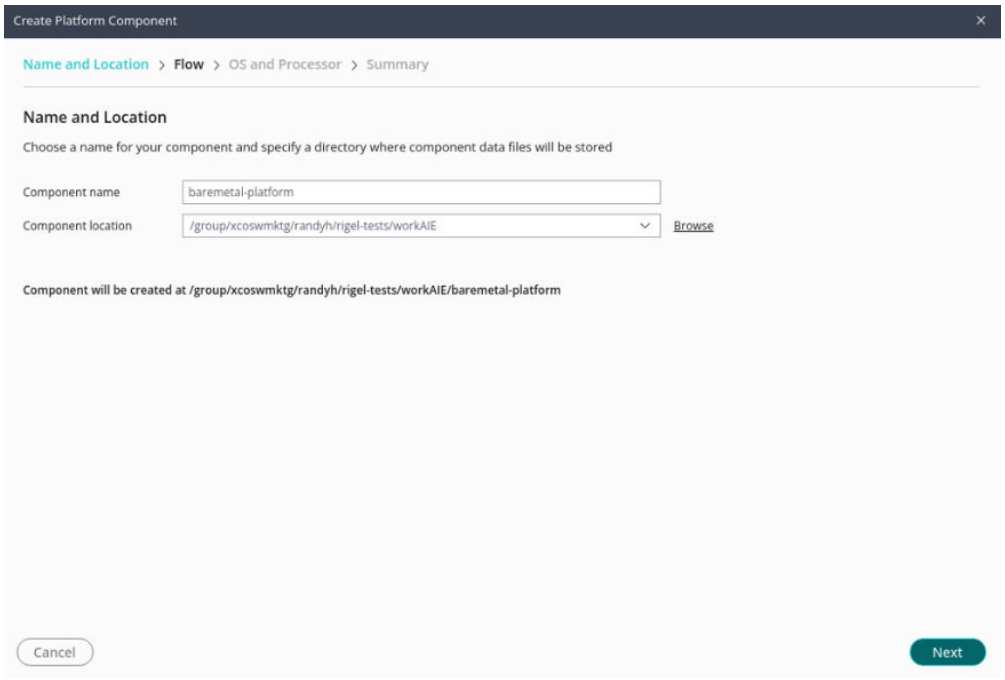


Creating a Bare-metal System

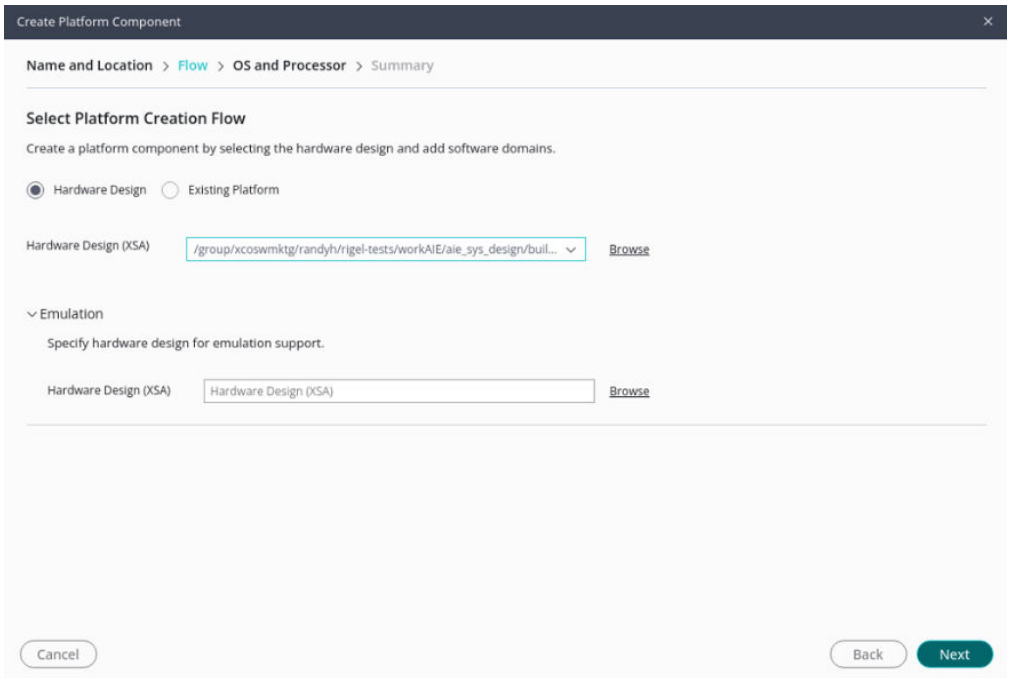
Building a bare-metal system requires a deviation from the standard System project flow previously described. The branch in the process occurs after the Linking the System in the *Data Center Acceleration using Vitis* (UG1700) step, with the bare-metal system requiring a new process branching after the generating the `.xsa` file from the `v++ --link` command. The specific steps required are detailed below.

1. Create a bare-metal software platform from the `.xsa` produced by the linking process for the hardware build target.

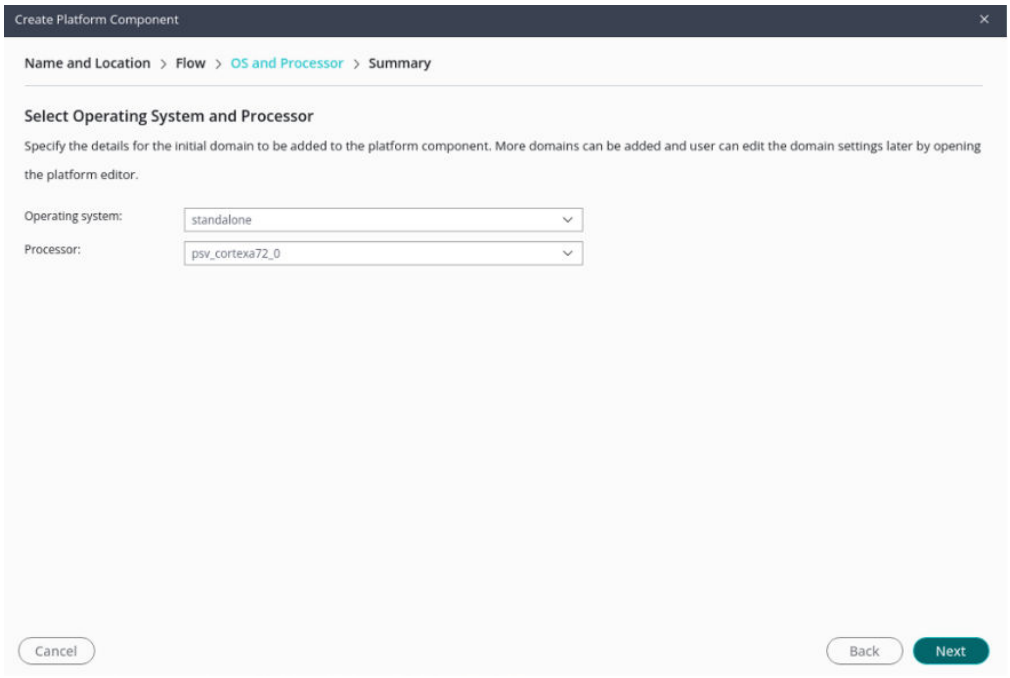
   a. Select the **File → New Component → Platform** command in the Vitis unified IDE. This opens the Create Platform Component wizard as shown.



b. Specify a Component name and click **Next** to proceed. This displays the Flow page of the wizard where you select **Hardware Design** and specify the `binary_container_1.xsa` from the System project to create the new platform.



c. After you select the XSA, the Vitis IDE reads the file, determines the Operating system and Processor for the domain defined by the XSA, and populates it in the dialog box.

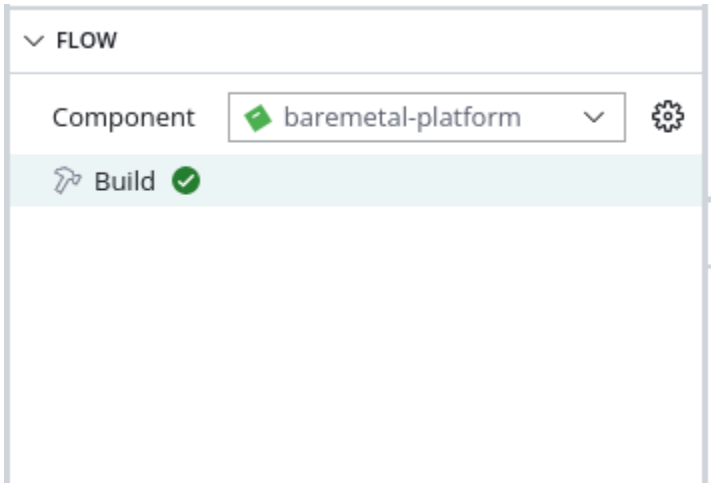


d. Click **Next** to go to the Summary page and click **Finish** to create the platform project.

> **TIP:** *The bare-metal platform is valid for either hardware emulation or hardware builds depending on the fixed-XSA selected for the platform.*

e. Click the **Build** command to build the newly created platform. A copy of the completed platform is written to the export folder of the project, and shows in the Explorer view for the project.

   As shown in the following figure, the exported platform has the `platform.xpfm` meta-data file, as well as the `./hw` and `./sw` folders for the different elements of the platform.



The Vitis unified IDE automatically adds the new platform to your platform repository making it available to use in projects. You can also add the file location to your `$PLATFORM_REPO_PATHS` environment variable to make the platform available to the command-line tools as well as the Vitis unified IDE.

> **IMPORTANT!** *The bare-metal platform will be only used for building the bare-metal PS application and is not used any other places in the flow.*

2. Create a new embedded software Application component.
   a. Select the **File → New Component → Application** command in the Vitis IDE. This opens the New Application Project wizard.
   b. Specify a **Component name** and click **Next**.
   c. Select the **baremetal-platform** you created in the last step, and click **Next** to proceed.
   d. Review the Domain page and click **Next** to proceed.
   e. Review the Summary page and click **Finish** to create the Application component.
   f. Add the source files and build the application as described in Creating an Application Component. The source code for the PS application must be written specifically for the bare-metal project using the provided drivers for accessing the hadware system. For AI Engine designs you must also add the bare-metal AI Engine control file (`aie_control.cpp`), which is created by the `aiecompiler` command, and can be found in the `./Work/ps/c_rts` folder.
   g. Edit the `UserConfig.cmake` file of the Application component located in the Settings folder in the Vitis Components Explorer. This step is only if you have AIE. Edit the Include paths under Directories to include the following:

```
$XILINX_VITIS/aietools/include
<aie_component>/src
```

   h. After updating the `UserConfig.cmake` file you can build the Application component.
3. Package the System project with the new PS application added to the SD card files.
   a. With the ELF file produced for the bare-metal PS application, you are ready to package the System project and Application component for the bare-metal platform. You must run the package process to generate the final bootable image (PDI), and write the SD card content for booting the device and running the application, as described in Packaging for Vitis Flow in the *Embedded Design Development Using Vitis* (UG1701).
   b. Add the following to the Packaging config as described in Defining the Package Configuration:

```
[package]
ps_elf=../../baremetal_app/Debug/<baremetal_app>.elf,a72-0
sd_file=<baremetal_app>.elf
```

***Note*:** To enable debugging for both the AI Engine component and the bare-metal PS application, do not add the `--package.ps_elf` option. To debug only the AI Engine component, then add the option as described.

c. After updating the `package.cfg` file, run the **Build Package** command on the System project.

   This adds the ELF file you created in Step 2 above for the bare-metal Application component, assigns it to a processor core (`--package.ps_elf`), and packages the System project files and boot system. In the original Linux-based System project, you added a PS application into the system project to build and debug as part of the system. Here you are building the PS application as part of a separate bare-metal project and adding it as a boot file to the package process in the original System project. This results in a System project build for the platform hardware, and a PS application that runs in the baremetal domain.

Now that you have built the bare-metal system, you can continue to run or debug the application.

IMPORTANT! *You cannot debug the hardware emulation build for bare-metal projects in the Vitis unified IDE.*

Migrating Command-line Projects to the Vitis Unified IDE

In the AI Engine application development flow, it is common to build and simulate the AI Engine graph application using the command line (with a Makefile or config file-based approach). However, to debug and analyze the results, it is recommended that you use the AMD Vitis™ Unified IDE. The Vitis IDE provides a user-managed flow that uses the pre-compiled results from the command-line in the IDE for debug and analysis. This section provides a methodology to migrate your AI Engine graph application from the command line interface to the Vitis Unified IDE.

To migrate your command-line project to the Vitis Unified IDE:

1. Launch the Vitis tool and select **Set Workspace** from the Welcome page.
2. Navigate to and select the directory where the command-line based AI Engine graph application is built.

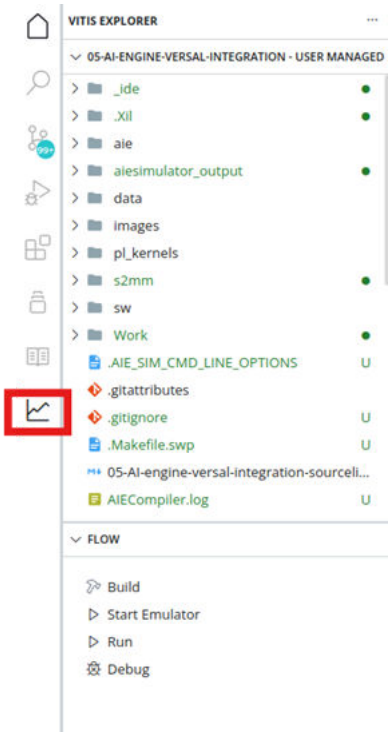
   The IDE recognizes that there are no managed components in the directory and switches to the user-managed flow.

Analyzing Results

In a user-managed flow in the Vitis IDE, you can analyze project results as follows:

1. In the Vitis Unified IDE, note the following two views:
   - **Vitis Component Explorer:** In this view, the directory opens, displaying several output directories that are generated after compiling and/or simulating the AI Engine graph.

     The AI Engine compile output from the command-line flow is located in the `Work/` directory available in the Vitis IDE. A LST file and a mapping file available in the Work/ directory was generated by the AI Engine compiler for each active AI Engine core.
   - **Flow Navigator:** In this view, you can access options for building, running, and debugging the design.
2. From the toolbar to the left of the screen, click the Graph button to open the Analysis View to examine the results.



   In the Analysis view, you can view and analyze all summary files in the current workspace.

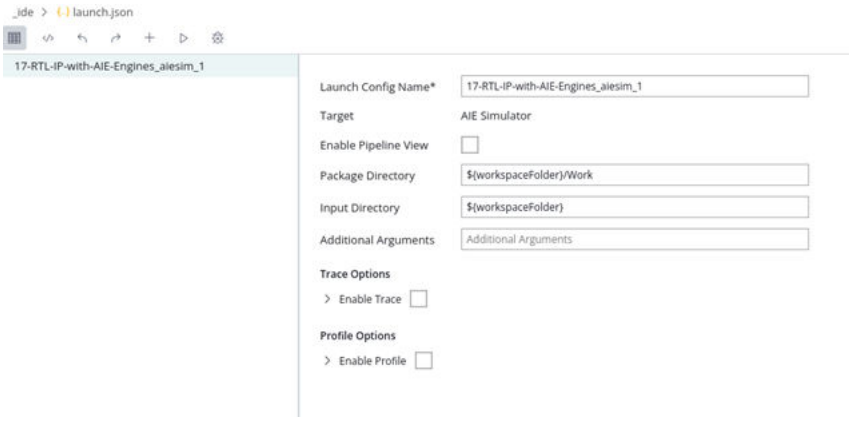
Debugging the AI Engine Code

You can debug the AI Engine code in the user-managed flow in the Vitis IDE as follows:

1. First, create and add a launch configuration. To do so:
   a. In the Component Explorer or Flow Navigator view, right-click and select **Edit launch configuration**.
   b. Select **New launch configuration**.

c. In the Create Launch Configuration dialog box, specify the build directory with the AI Engine compile summary files, and click **Submit**.

d. Enter a name for the launch configuration, and enable trace and profile settings as needed. For more information about trace and profile options, see Profiling and Debugging the Application in the *Data Center Acceleration using Vitis* (UG1700).



2. In the Flow Navigator, select **Debug**, and set breakpoints in the source code. Using the breakpoints, step through the code and observe the contents of registers and memory in the Register Inspectorand Memory Inspector views, respectively.

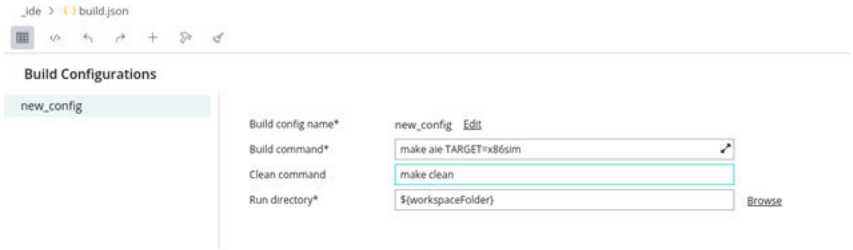
3. You can recompile the AI Engine graph application in the user-managed flow with the steps below:

a. In the Explorer view, right-click and select **Edit Build Configurations**.

b. Select **New Build Configuration**.

c. Optionally, edit the default configuration name, and add a build command.

The build command you enter in the **Build** command field is the build command from your Makefile, for example `make aie TARGET=x86sim`, as shown below.



Debugging the System Project and AI Engine Components

You can debug AI Engine and HLS components in a standalone capacity, or you can debug the System project, including the top-level PS application, and any additional components. Within this framework, you can also debug applications built from the command line, or system projects built in the Vitis Unified IDE. You can only debug applications running on the Linux OS. Baremetal applications debug support is not available through the IDE. You can debug hardware emulation builds that let you simulate the application, or debug the actual application running on hardware. All of these configurations are addressed in the following topics.

The following process is recommended for debugging AI Engine applications and system designs:

1. Use x86 simulation for both single and multi-kernel debug. It supports breakpoints and single stepping using the GNU debugger.
2. Use `aiesimulator` to verify timing and to check that it will fit within the program memory as well as stack/heaps size within the available hardware memory space. The `aiesimulator` and `x86simulator` and their use are described in *Simulating an AI Engine Graph Application* in *AI Engine Tools and Flows User Guide* (UG1076).
3. Hardware emulation for complete integration testing including PL, PS, and AI Engine domains as applicable.
4. Hardware-based testing of the final booted system.

If code changes are necessary at any step, repeat all steps from the beginning.

***Note*:** It is recommended to use the compiler optimization option `--xlopt = 0` because higher compiler optimizations will reduce debug visibility.

Launching Debug in the Vitis Unified IDE

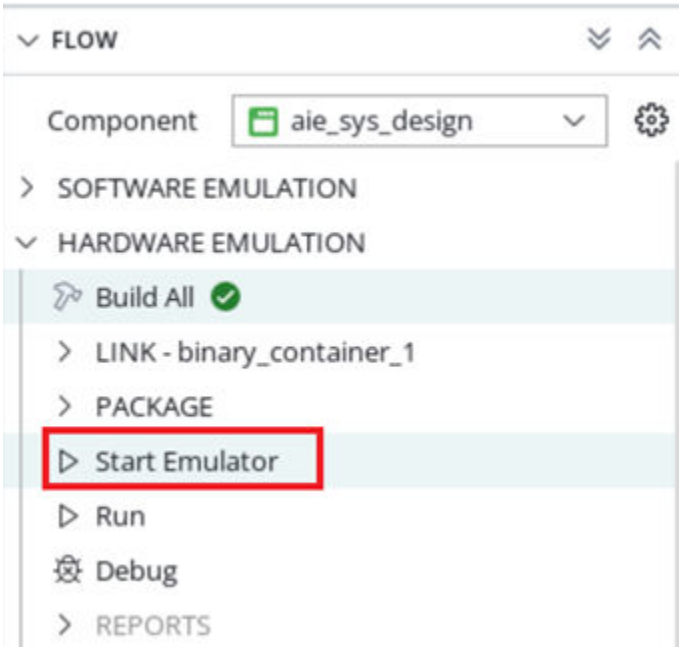
After building the System project, incorporating the kernels running in the AI Engine domain, the PL domain, and an application running in the PS domain, as described in Creating a System Project for Heterogeneous Computing, you can launch debug. These system-level projects can be debugged together from within the Vitis unified IDE using the many features as described in the Debug View, or you can debug the individual components like the AI Engine graph application, focusing on a particular design problem.

This section discusses running the Debug Environment from the Vitis unified IDE for hardware emulation and hardware builds.

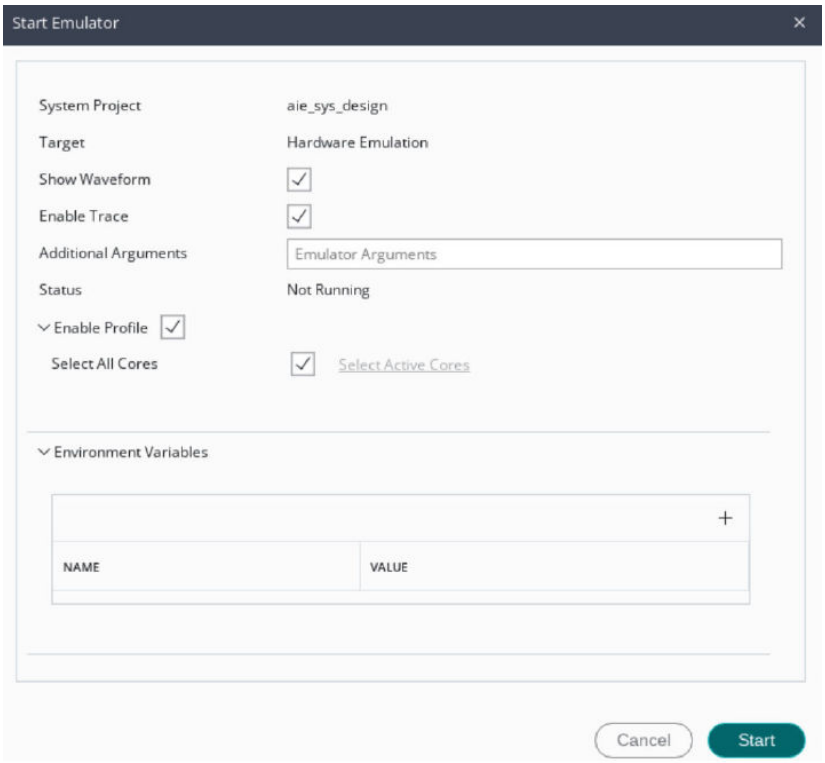
Component Based Flow Design Debug

The following steps illustrate component based flow design debug for hardware emulation.

1. Start the QEMU emulation environment by selecting the Start Emulator command from the Flow Navigator.



2. This opens the following dialog box.



Where:

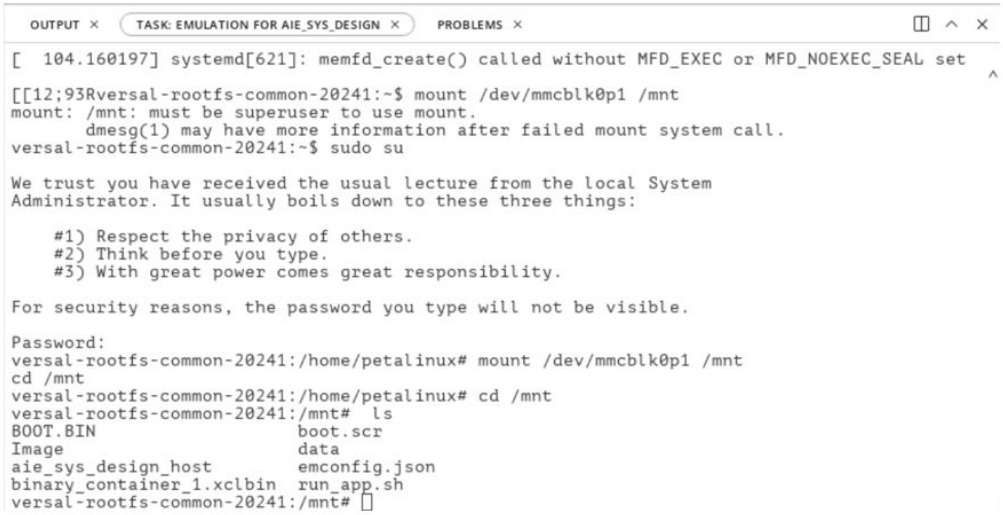
- **Show Waveform**: Enables the display of the Live simulation Waveform view from within the Vivado Logic Simulator during the Emulation run. This lets you examine waveforms in real-time. However, it takes more time and resources to run.
- **Enable Trace**: Enables capturing trace data from hardware emulation.
- **Additional Arguments**: Used to pass additional arguments to the launch_emulator utility from the command line when starting QEMU. For example, this can be used for specifying options to pass to the AI Engine simulator that runs the graph application, as described in the Reusing AI Engine Simulator Options in *AI Engine Tools and Flows User Guide* (UG1076).

The options can be specified as follows:

```
-aie-sim-options ../aiesim_options.txt
```

- **Enable Profile**: Enables profiling of the design as explained in Profiling the Application in the *Data Center Acceleration using Vitis* (UG1700).
- **Key/Value Pairs**: Specifies parameters to use when running the application in the form of the parameter name and value.

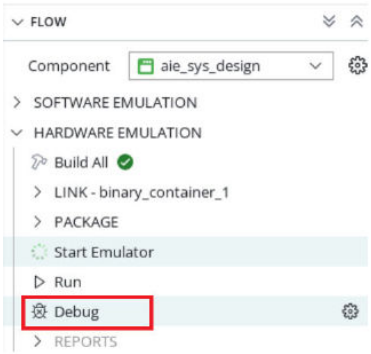
3. Click **Start** to launch the emulator. Then you must wait for the QEMU environment to boot the emulation system before starting your application.



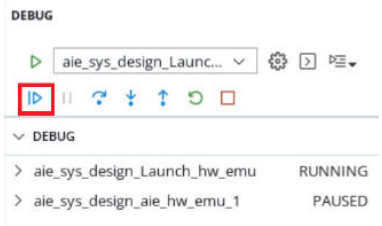
The TASK: EMULATION output terminal shows a transcript of the QEMU launch and boot process. You will know the process has completed when the `/mnt#` command prompt is displayed on the terminal window. From the command prompt, you can type commands in the QEMU environment, and review the transcript of the boot process for details.

> **IMPORTANT!** *Do not run the emulation from this window. The Vitis Unified IDE will manage the application when launched from the Run or Debug command in the Flow Navigator after the emulator has been started.*

4. After the QEMU has started, click **Debug** from the Flow Navigator. This opens the Debug view in the Vitis Unified IDE and connects to the PS application and AI Engine graph running on their respective cores in the QEMU. The application automatically breaks at the `main()` function for all the ELF files.



5. Click Continue ▶ from the Control Panel to start the process.



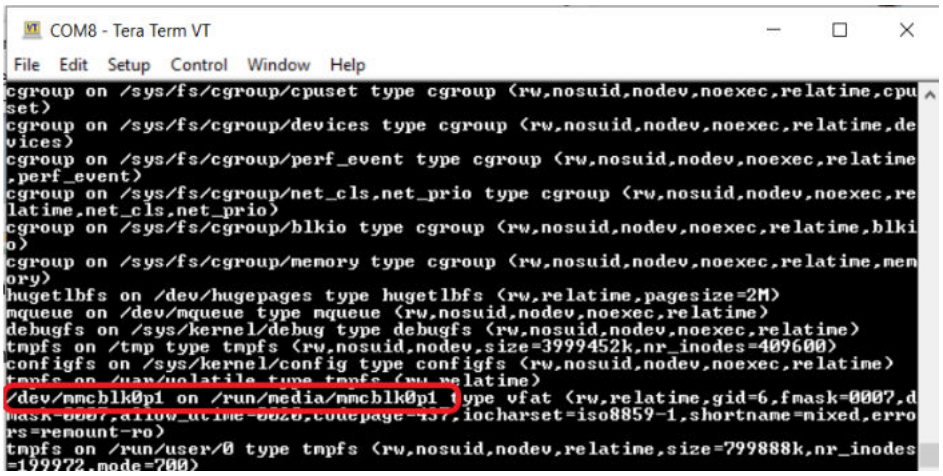
From this point, you can perform debug activities such as step in, step over, viewing variables and plant break points in the emulation environment. Refer to Using the Debug Environment for more information.

Hardware Debug from the Vitis IDE

With the top-level system project built, you can use the following steps to debug the system running in the Hardware platform.

1. Burn `<project>/Hardware/package/sd_card.img` to a physical SD card. This creates a boot-able medium for your target platform.
2. Insert the SD card into the card reader of the VCK190 evaluation kit.
3. Change the boot-mode settings of the card to SD boot mode, and power up the board.
4. After the VCK190 is booted, enter the `mount` command at the command prompt to get a list of mount points. As shown in the following figure, the `mount` command displays mounting information for the system.

   TIP: *Be sure to capture the proper path for the `cd` command in the next step, and subsequent commands, based on the results of the `mount` command.*

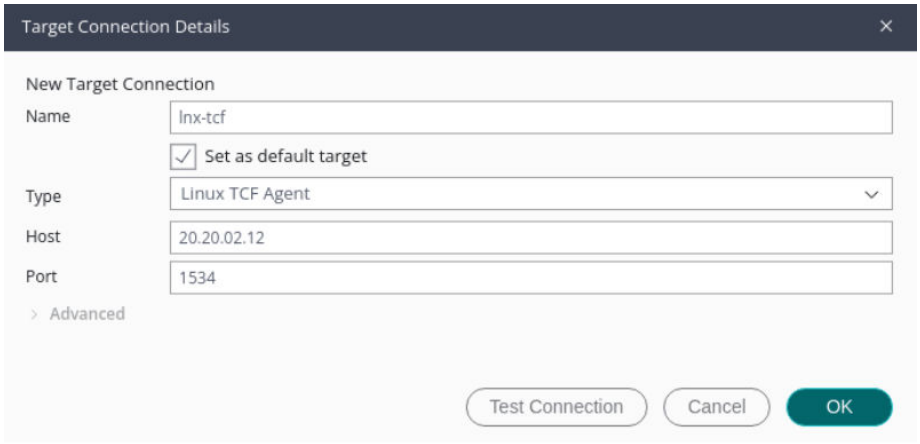


5. Execute the following commands:

```
cd /run/media/mmcblk0p1
```

6. Run `ifconfig` to get the IP address of the target card. The IP address is used to set up a TCF agent connection in Vitis unified IDE. The target needs to connect to the network assigned IP address.
7. From within the IDE select or create a target connection to the remote accelerator card as follows:
   a. Select the **Vitis → Target Connections** command from the main menu to open the Target Connections dialog box.

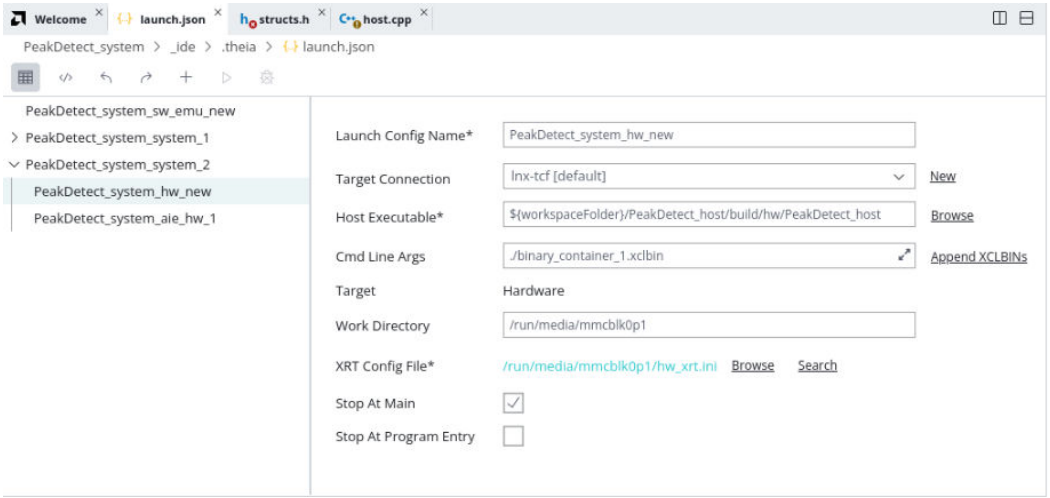
b. Select an existing Linux TCF Agent connection, or right-click **Linux TCF Agent** and select the **New Target** command to open the New Target Connection dialog box as shown in the following figure.



c. Specify the Target Name, enable the **Set as default target** check box, and specify the Host IP address of the accelerator card that you obtained from the `ifconfig` command.

d. Click **OK** to close the dialog box and continue.

8. Select the **Open Settings** command in the Flow Navigator next to the Debug command, or open the `launch.json` from the System project in the Vitis Components Explorer. Validate the launch configurations for the Hardware build, or select the **Add Configuration** (+) command to create a new configuration as shown in the following figure:



Be sure to set the following fields on the dialog box as shown in the preceding figure.

- Target Connection: Select the new Linux TCF agent you built with the specified IP address for the accelerator card
- Host Executable: Specify the PS application to run from the SD card
- Cmd Line Args: Specify any command line arguments required for the software application, such as the xclbin file to load

- Work Directory: Specify the remote mount location for the SD card
- Stop At Main and Stop At Program Entry: To stop the application and AI Engine from running before starting the Debug process

9. Click **Debug** from the Flow Navigator under the HARDWARE heading.

   TIP: *The tool will prompt you to create two launch configurations if you haven't already done so. One for the top-level system project, and a second for the PS application.*

   This opens the Debug view and connects to the PS application and AI Engine graph running on their respective cores. The application automatically breaks at the `main()` function for all the ELF files.

From this point you can do all the debug activities such as, step in, step over, viewing variables, apply break points in the emulation environment. See Using the Debug Environment for more information.

Bare-Metal Debug from the Vitis Unified IDE

With the bare-metal system project built as described in Creating a Bare-metal System, you must use the following steps to debug both the AI Engine graph, together with the bare-metal PS application on the Hardware build target.

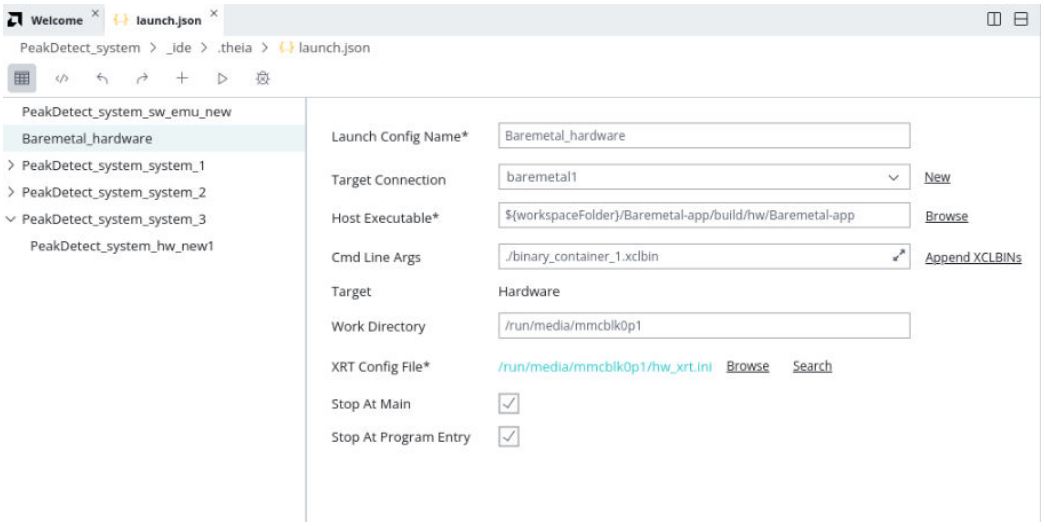
This process is a bit more complex than debugging a Linux-based System project where the PS application is built as a part of the system project. Here the PS application is built as a separate application component, and must be manually included in the launch configuration for debug. This is detailed in the following steps.

1. Right-click on the top-level system project and select the **Debug As → Debug Configurations** command.

   This opens the Debug Configurations dialog box to let you set up the tool.

   IMPORTANT! *For the Hardware build, you will need to create two Debug configurations: one for the top-level system project, and a second for the bare-metal PS application.*

2. In the Debug Configurations dialog box select the **New Launch Configuration** ( ) command to open the Debug Configurations dialog box as shown.



   Notice the following fields on the Debug Configurations dialog box:

   - Project: Reflects the name of the top-level system project which includes the AI Engine graph application, the PL kernels, and the HW-Link projects.
   - Hardware Server: Specifies a local connection to the board. You can configure this differently for a remotely connected board.
   - Linux TCF Agent: Is disabled for bare-metal systems.
   - Stop At Main and Stop At Program Entry: Enable these options to prevent the application from running before you have a chance to start debug.

3. Select **Apply** to save and apply your changes, and select **Close** to close the dialog box.
4. Click **Debug** from the Flow Navigator under the HARDWARE heading.

   TIP: *The tool will prompt you to create two launch configurations if you haven't already done so. One for the top-level system project, and a second for the PS application.*

   This opens the Debug view and connects to the PS application and AI Engine graph running on their respective cores. The application automatically breaks at the `main()` function for all the ELF files.

From this point you can do all the debug activities: step in, step over, viewing variables, apply break points in the emulation environment. Refer to Using the Debug Environment for more information.

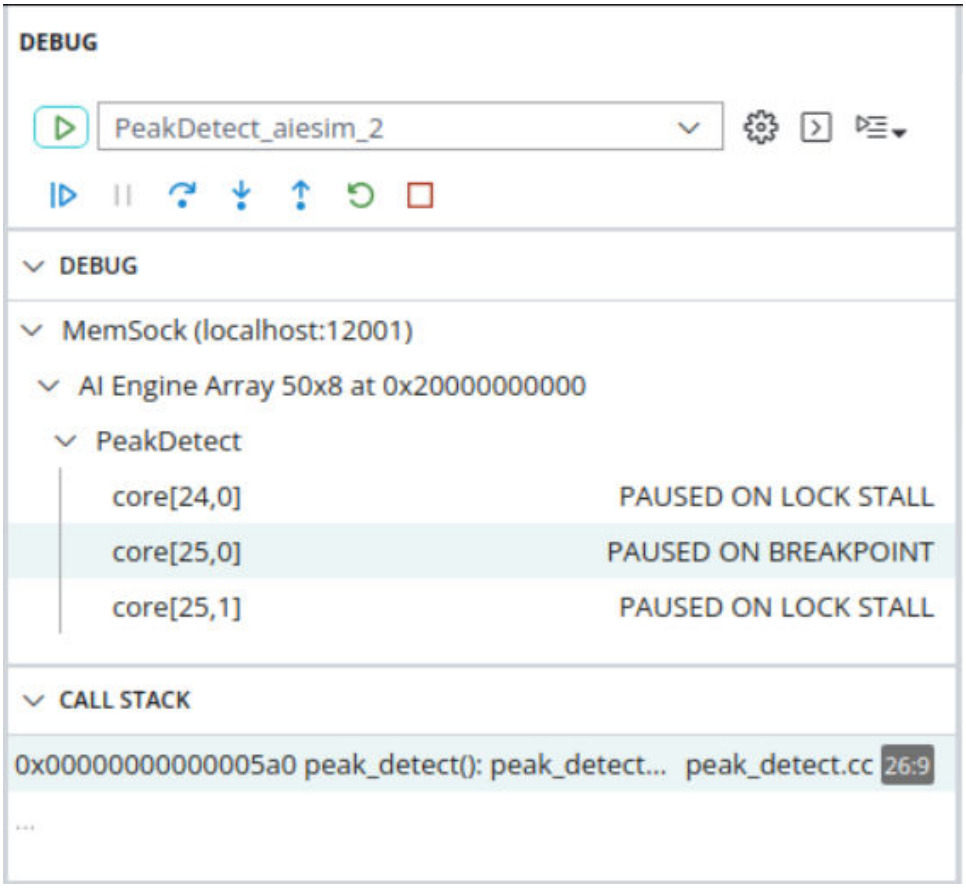
Using the Debug Environment

The Vitis IDE debug environment has many features found in traditional GUI-based debug environments, such as GDB. You can add break points to the code, step over or step into specific lines of codes, loops, or functions, and examine the state of variables and force them to specific values. These are a few of the features in the Vitis IDE debug environment.

After you have launched the Debug perspective, you will see several windows or views displayed, such as the Debug view in the upper left of the display, as shown in the following figure. During the debug process, several windows show the debug state that include call stack, code at breakpoint, step over states, breakpoints view, variables view, registers view, disassemble view, and pipeline view.

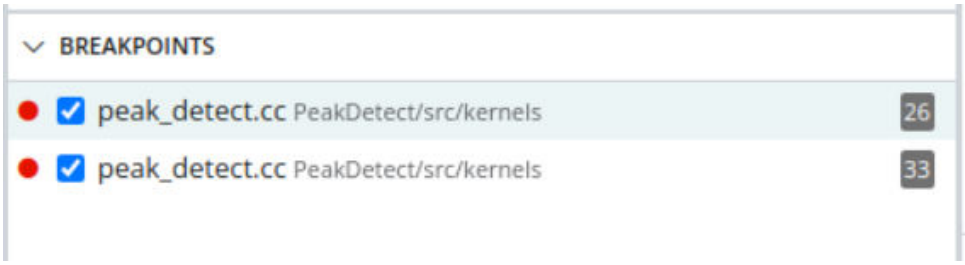
The Debug view shows states of cores that are under debug. It shows where (which file and which line of source code of the file) debugger stops at and with what action it is taking (breakpoint/step over/...) as shown in the following figure.

Figure 29: **Debug View**



The following figure shows the Breakpoints information with current setup breakpoints. The square with a check mark indicates the breakpoint is enabled. Click on the check mark to clear the check mark and disable breakpoint during debugging. This lets you manage breakpoints without having to remove them or add them back into the code.

Figure 30: **Breakpoints View**



IMPORTANT! *Each AI Engine tile supports four breakpoints when debugging on the AI Engine simulator, or co-simulation. The TCF framework stops at the AI Engine kernel* `main()` *by default. Breakpoints attached to a* `while` *statement consume two breakpoint resources. A workaround is to attach the breakpoint inside the* `while` *loop. This only consumes one breakpoint.*

In a source-level debugger, you can trace the source variables allocated to memory or registers. Their location and contents can be visualized. However, when all optimizations are enabled, tracing source variables is not straightforward:

- Local scalar variables typically reside in registers, often different ones at different points in the execution.
- Variables allocated to memory are not always directly updated. For example, a global variable can be loaded in registers in front of a loop. Then, it can be operated on in registers inside the loop, and only stored back in memory after the loop. This is referred to as optimization redundant store removal.
- Similarly, stepping through the source code becomes difficult when the code of a single source line is scattered in the eventual object code.

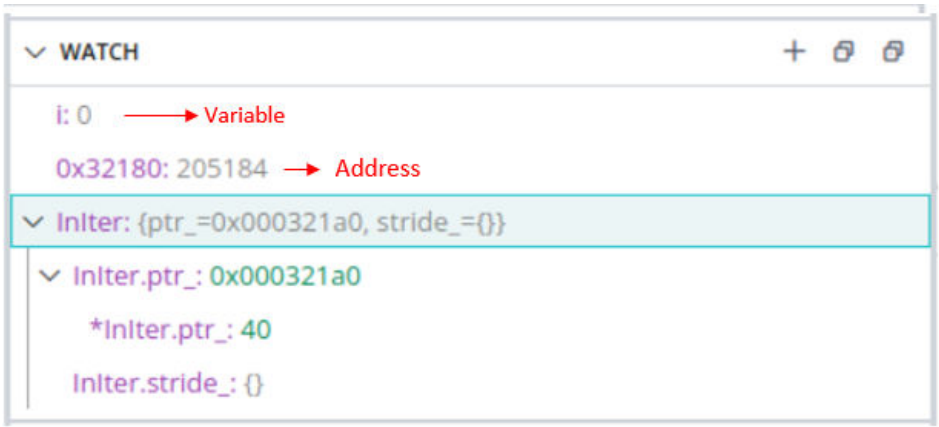
Watchpoints

A watchpoint is a type of breakpoint. Watchpoints can be used to stop the execution of a core when the value at an address changes. These are useful in determining the place where the value changes. For example, a variable value can be overwritten unexpectedly, which can break the program flow. Watchpoints help in detecting such cases.

AI Engine architecture supports read/write watchpoints. This means, a watchpoint is triggered on a read or write access, or both (based on your configuration). Each AI Engine tile supports two watchpoints per memory bank. Because every AI Engine core (excluding tiles on boundaries) has access to memory banks in adjacent tiles, a maximum of eight watchpoints can be used per core. However, because the memory banks are shared, the number of available watchpoints per core

depends on how many watchpoints are already used from the memory bank. For example, if a core uses all eight watchpoints from its four adjacent memory banks, the other cores which share those four memory banks cannot use watchpoints from the shared memory banks. Debugger keeps track of watchpoints that are allocated from each bank, and throws an error if there are no unused watchpoints in that tile.

1. To add watch point in the Vitis IDE, locate the WATCH window in the bottom left side. Hover your mouse on that window and click on the **Add Expression** (+) command as highlighted below.
2. You can enter the memory address or the variable name of interest. You can observe that the values in the watchpoints can appear as "not available", if you select a core other than the core for which the watch point is created. For example, if you created watch points by selecting the core in the debug view, the updated values are seen only for that particular core. For rest all other cores, the watch point values can appear as "not available."



3. You can edit, copy, and remove one or all watch points by right-clicking on any watch point.

Triggering of Watchpoints

A watchpoint is triggered on a read access or write access, or both, based on how the watchpoint is configured. A triggered watchpoint causes the core to stop at the instruction that accessed the address. Debugger detects the core has stopped because of the watchpoint and reports that the core has stopped because of a watchpoint.

IMPORTANT!

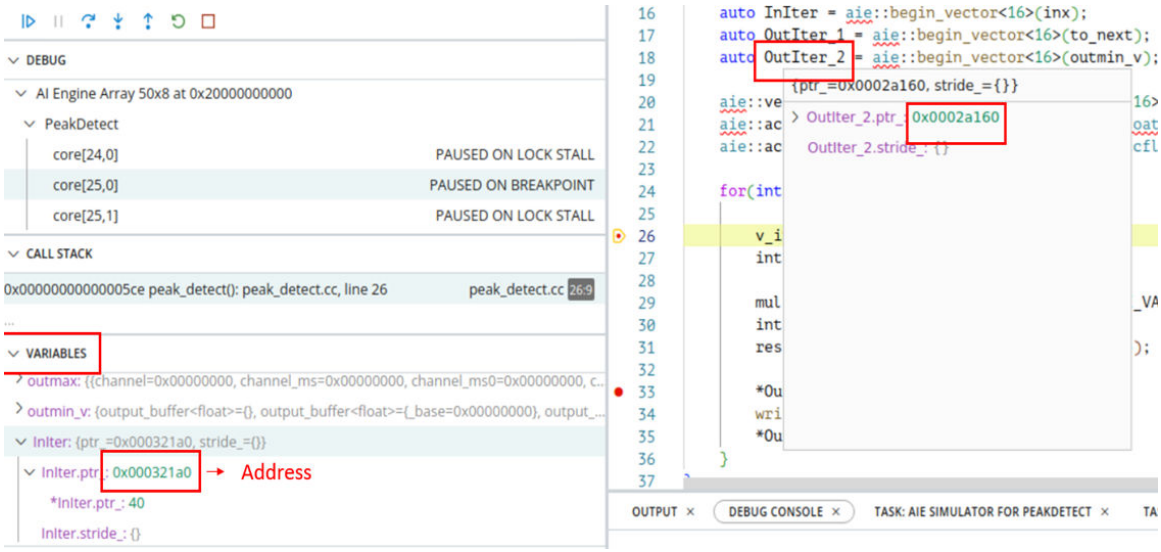
- Watchpoints work on hardware only. AI Engine system C model is not supported.
- Memory banks are shared. It might not be possible to add watchpoints for a core in a specific bank, if another core uses both the watchpoints from that bank.
- Watchpoints must not be set for memory addresses which fall in unused tiles/memory banks. Setting watchpoints to unused tiles can cause AXI errors. Used AI Engine and memory tiles can be found at `Work/aie/active_cores.json`.
- Watchpoints are triggered for memory access in full 16-byte aligned address range.

- Debugger uses two broadcast channels to handle watchpoint events. When enabling watchpoints during debug, ensure that there are no conflicts in using these two broadcast channels.

Variable View and Memory Inspector

The Variables view displays the values of kernel variables. Clicking on a variable shows its type, value, and the address of the variable. For array/structure variables, clicking on the arrow of the variable expands array/structure content of the array.

Figure 31: **Variables View**

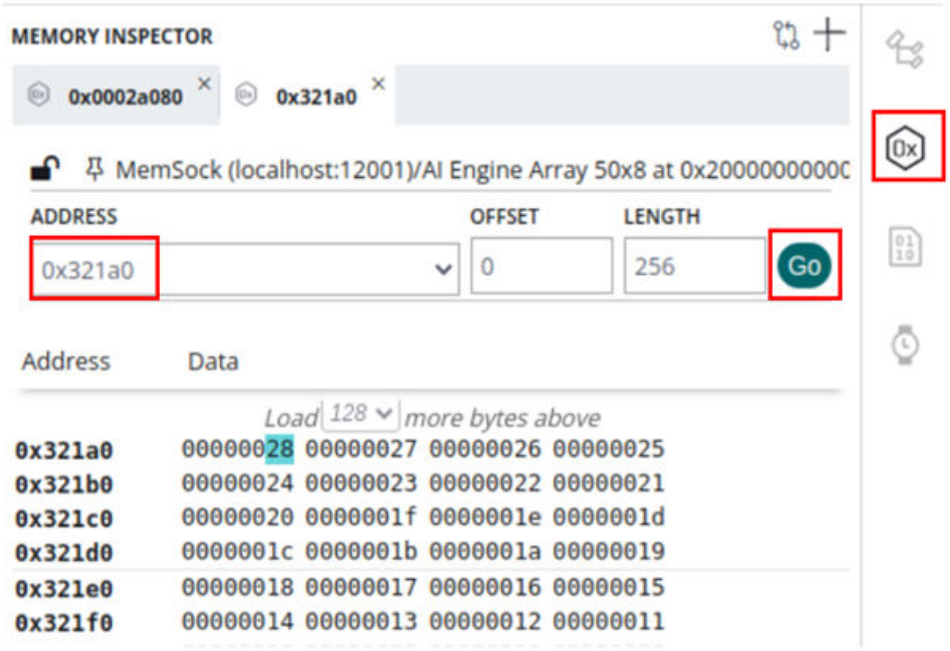


Click on the variable in the VARIABLES view or hover your mouse over the variables in the code view on the right-side as shown above to get the address and add that address in the memory inspector as shown below. As you step-in, step-over the code, the values in the variable and memory view changes.

TIP: *You can export the contents of the Memory window by selecting the* ***Copy to Clipboard*** *command from the right-click menu to copy and paste the contents into a text editor and save to a file.*

From the **Debug** view, you can open the Memory Inspector by selecting **View → Memory Inspector** from the main menu, or by clicking the **Memory Inspector** icon on the right hand side of the display, as shown in the image below.

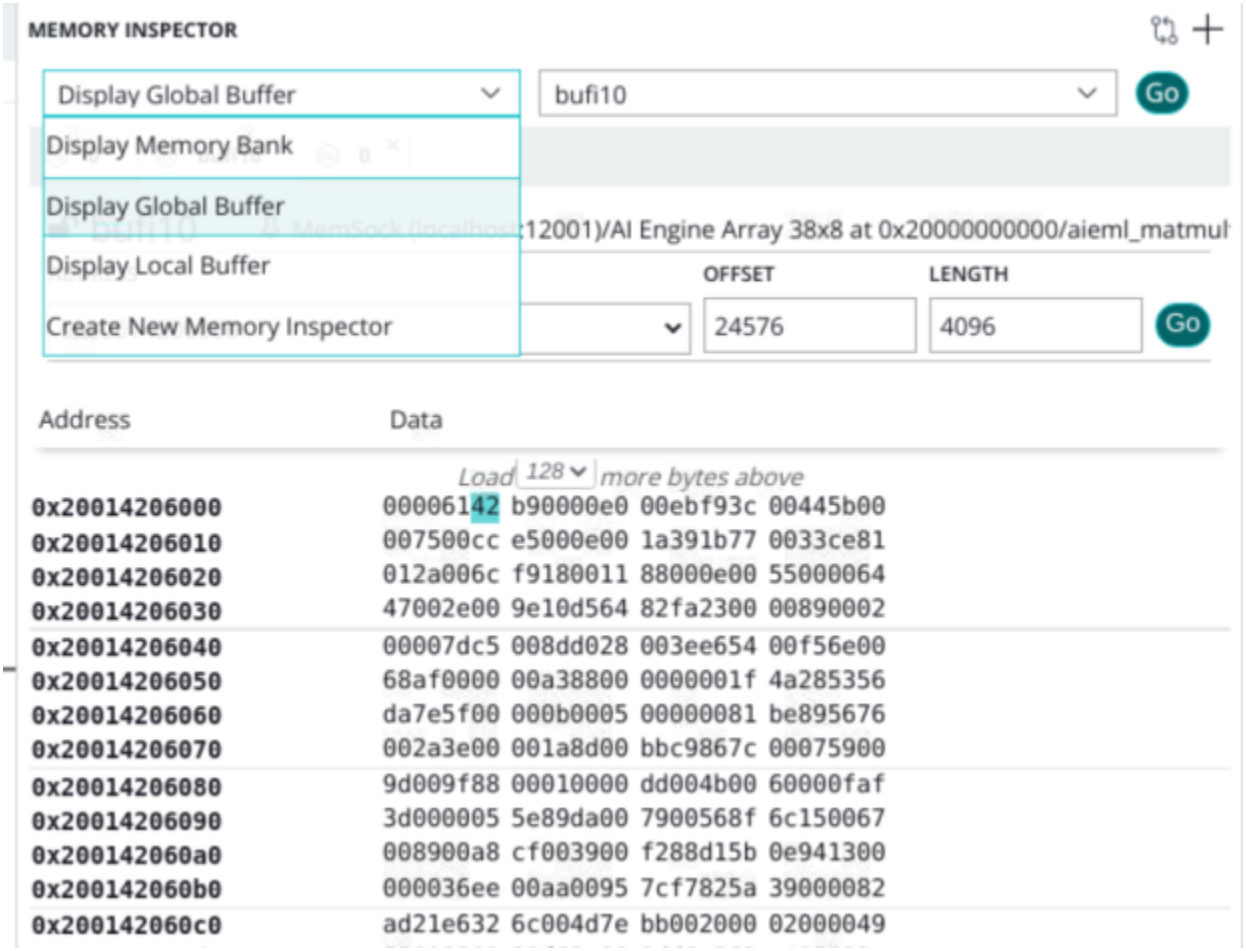
Figure 32: **Memory Inspector**



The Memory Inspector view is displayed as shown above. In the Threads view, select the desired AI Engine core to examine the core's memory. Enter an address, offset and length from which you would like to retrieve the memory range. Click the **Go** command to view the memory contents.

For AI Engine-ML, the options in the Memory menu enable you to choose the memory to be inspected.

Figure 33: **Memory Menu on AI Engine-ML**



- **Display Memory Bank:** Enables you to access to the four neighbor memory modules.
- **Display Global Buffer:** Enables you to access to all the shared buffers declared in the design.

In the previous image, the content of the shared buffer `bufi10` is displayed.

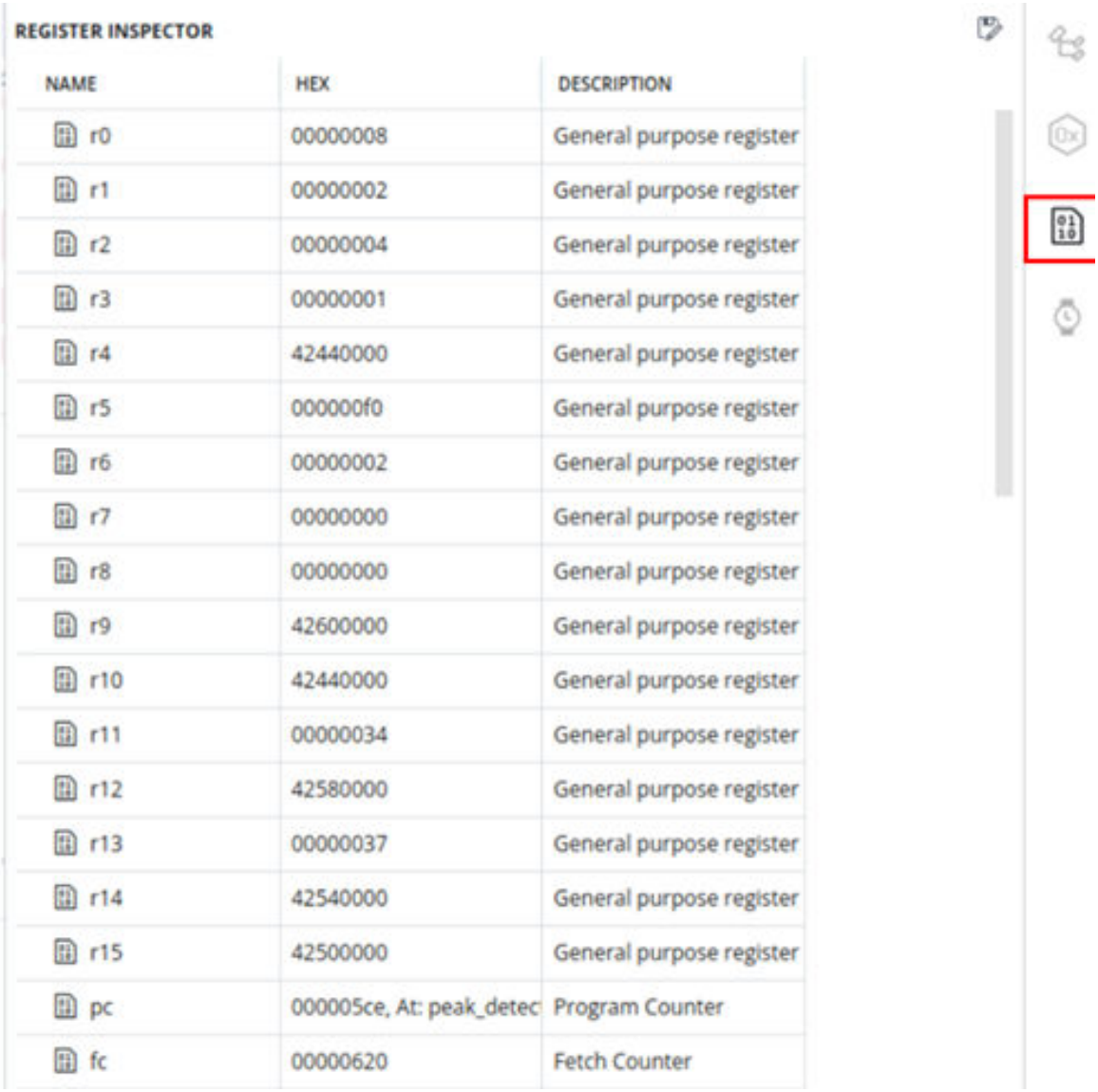
Select the **Settings** command at the top of the Memory Inspector to define attributes of the memory, such as Byte Size, Bytes Per Group, or Endian. You can edit the contents of the displayed memory by selecting an address, clicking in it and entering a new value. If a data needs to be updated from the address range, it will highlight a blue under liner. Click on the highlighted address and update with the required value and select **Apply Changes**.

Register Inspector

From the Debug view, you can open the Register Inspector by selecting the **View → Register Inspector** command from the main menu, or by clicking the **Register Inspector** icon on the right hand side of the display, as shown in the image below.

In the Registers Inspector, the values are updated during the debug process as you step-in or through the code, and are highlighted as shown in the following figure.

Figure 34: **Register Inspector**



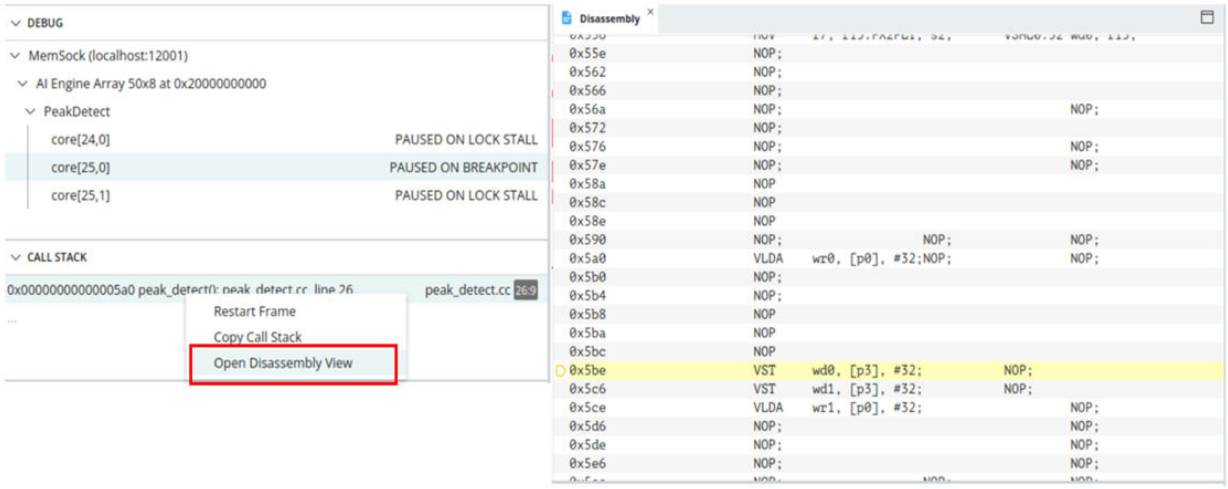
The Register Inspector view displays as shown above. All general-purpose registers, Program Counter, Stack Pointer and related items are available in the Register Inspector. In the Threads view select the desired AI Engine core to display the registers.

You can specify the HEX value of a specific register by selecting it, and entering the desired value, and hit enter to update it.

Disassembly View

The Disassembly view displays machine code and assembly code. C/C++ source code is also embedded between the lines for source code referencing. In the Explorer view, select the AI Engine core in Debug view and right-click on 'CALL STACK' and select Open Disassembly View. It will bring up the following disassembled code.

Figure 35: **Disassembly View**



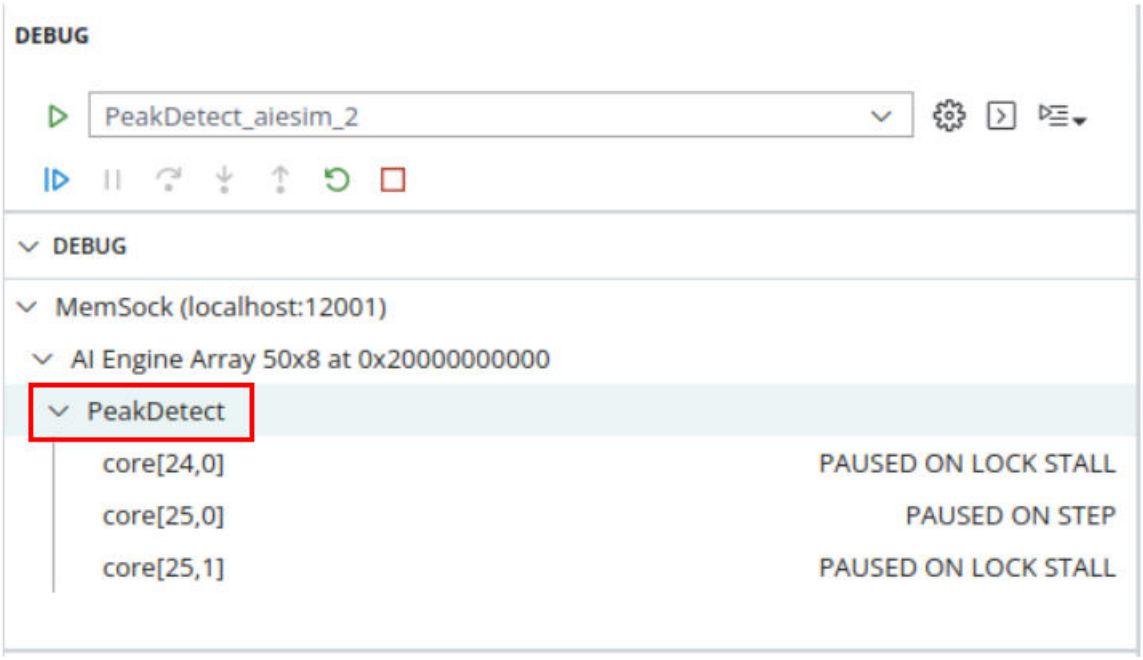
You can step over the disassembly code using the debug controls as shown below. These control commands let you step into/over/return from the source code. Other commands let you resume, stop, terminate, or disconnect the debugger from the Vitis IDE.

Multi-Kernel Debug

The Vitis IDE supports multi-kernel debug and multi-domain (PS and AI Engine) debug. Depending on the application, there can be hundreds of kernels being debugged. Having granular control of each core, in addition to all cores within one domain is important.

Select a single core from the Vitis IDE and click the **Continue** command to resume debug execution for the selected core only. Select one level above all cores within the AI Engine domain, and click the **Continue** command to resume all AI Engine core executions. Resuming execution for all the kernels in the graph is dependent on a variety of conditions such as the availability of data to each of the kernels to avoid stalls, or the setup of individual kernel breakpoints. You must also ensure that the kernels that are not being debugged are free to run.

Figure 36: **Disassembly View**



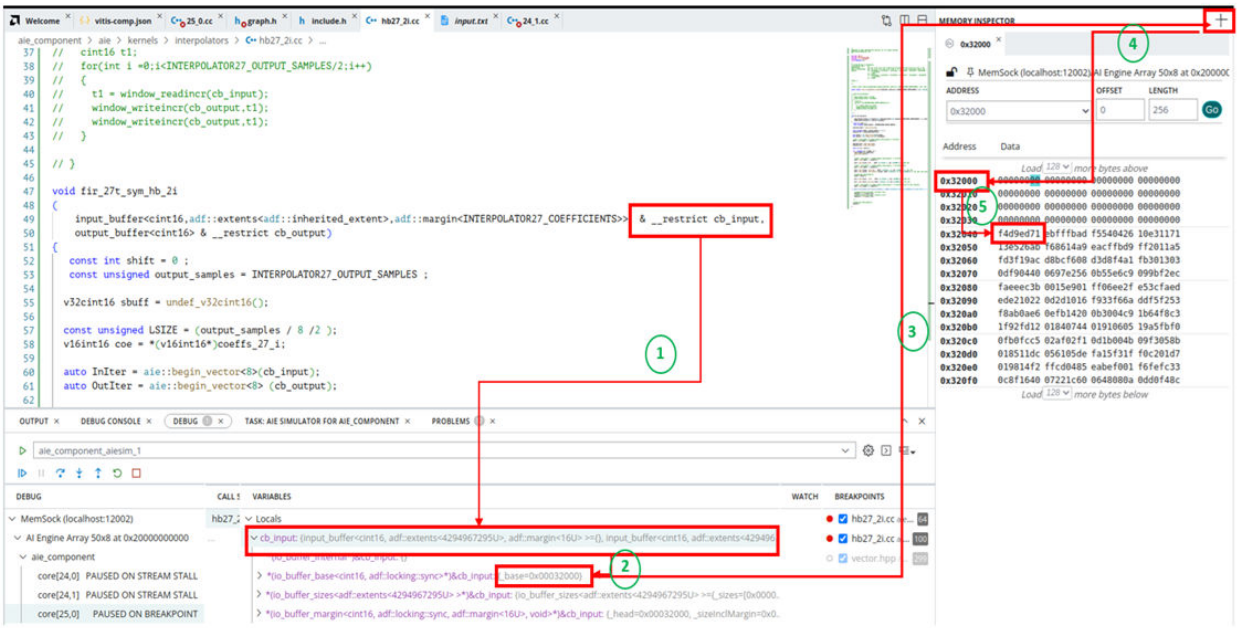
***Note*:** The `main()` function is created automatically by the AI Engine compiler for each core that wraps around the kernel implementation. The Debug view displays the complete source code including the inserted portion.

***Note*:** The Vitis IDE facilitates hierarchical debugging for all AI Engine cores. Users can effortlessly select the top-level AI Engine array and execute operations such as stepping in, stepping out, continuing, and pausing with ease.

Viewing Data from Buffer Port Interfaces

During debug, you want to see the value of data passing through the kernel. In the following figure you can trace through objects and observe data stored in the designated memory location from the Vitis IDE.

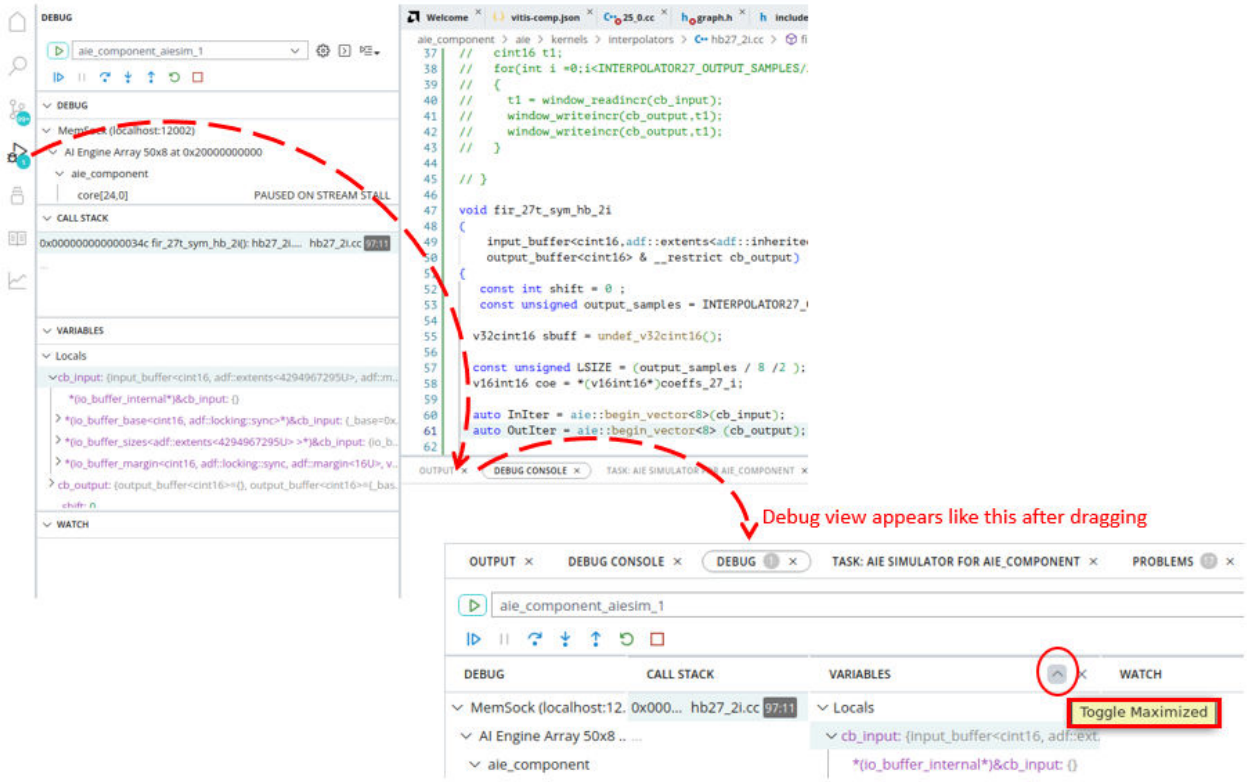
Figure 37: **Tracing Data in Vitis unified IDE**



1. In preceding code example, Step 1 shows the variable `cb_input` is the reference to `input_buffer` of type `cint16`.
2. In Step 2 move to the `cb_input` variable in the Variables view. This is the pointer representation of the data access buffer port that holds the input data for the kernel. However, the kernel functions merely operate on pointers to the buffer data structures passed to them as arguments. The input buffer port holds the data. Examine the address of the variable `cb_input`. It is at address `0x32000`.
3. In Step 3, enable the Memory Inspector window on the top right and click on the **+** sign to input the address `0x32000`.
4. In Step 4, the Memory window displays the content at address of `0x32000`. This is the data contained in the data access buffer port defined by the `cb_input0` variable.
5. This example has 16 elements as the margin size and each element is `cint16` type, so the actual data starts from `0x32000 + 0x40 = 0x32040`. You can examine the data contents, export to a file and hover your mouse on the hexadecimal values to display it in a specific data format.

***Note*:** You can click the **Debug** command and drag it from the left side Tool Bar menu to the bottom middle and drop it inside the space where you have the Debug console. This helps in viewing the variables in expanded view by clicking on Arrow button as shown in the following figure. You can drag it back to the left side at any time.

Figure 38: **Rearranging Windows**



Viewing Data from Stream Interfaces

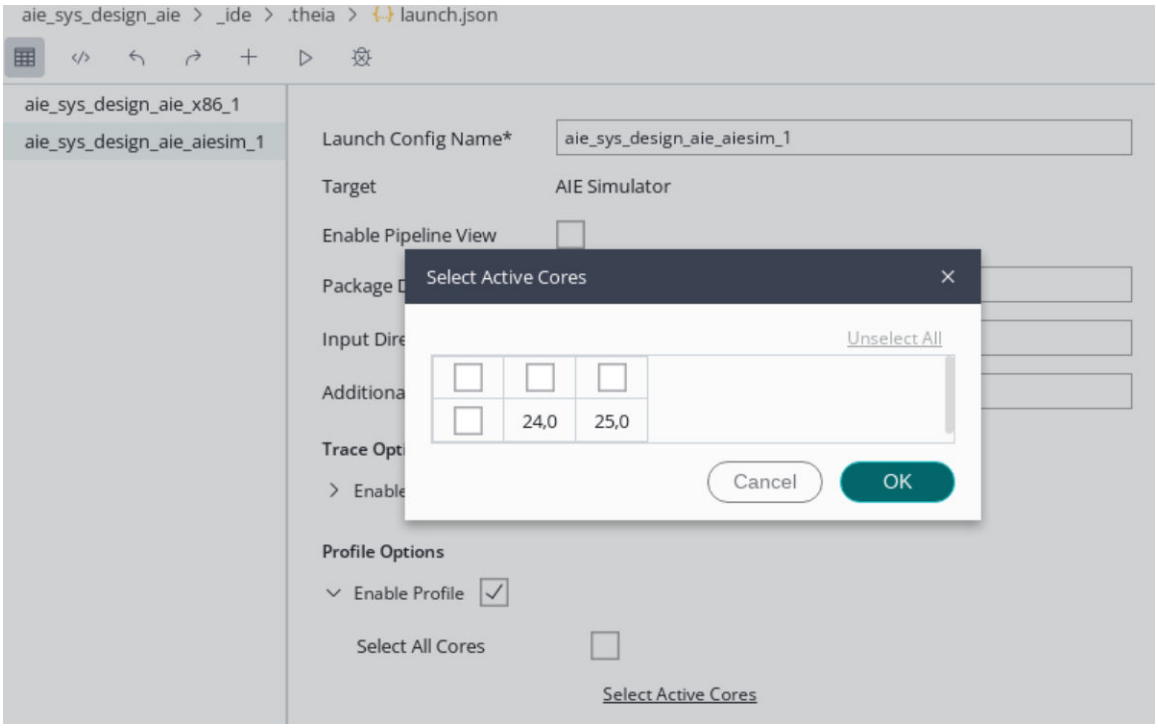
Data flows into and out of an AI Engine kernel through access windows, or a stream interface. During debug you might want to see the value of these data access windows as data is passing through the kernels. In the case of data windows, the Vitis IDE debug environment provides methods to view and access the data as described in Viewing Data from Buffer Port Interfaces. In the case of stream interface connections, it is recommended to add `printf()` statements to your code to let you examine the data passing through the kernel.

IMPORTANT! *Adding `printf()` statements to the code suppresses compiler optimizations, and results in a larger kernel executable program that might not fit into the available memory of the AI Engine processor.*

When adding the `printf()` statement in your kernel code you must select **Enable Profile** in the launch configuration for Debug as shown in the following figure. You can also select the cores for which profiling should be enabled. By default, all cores are selected.

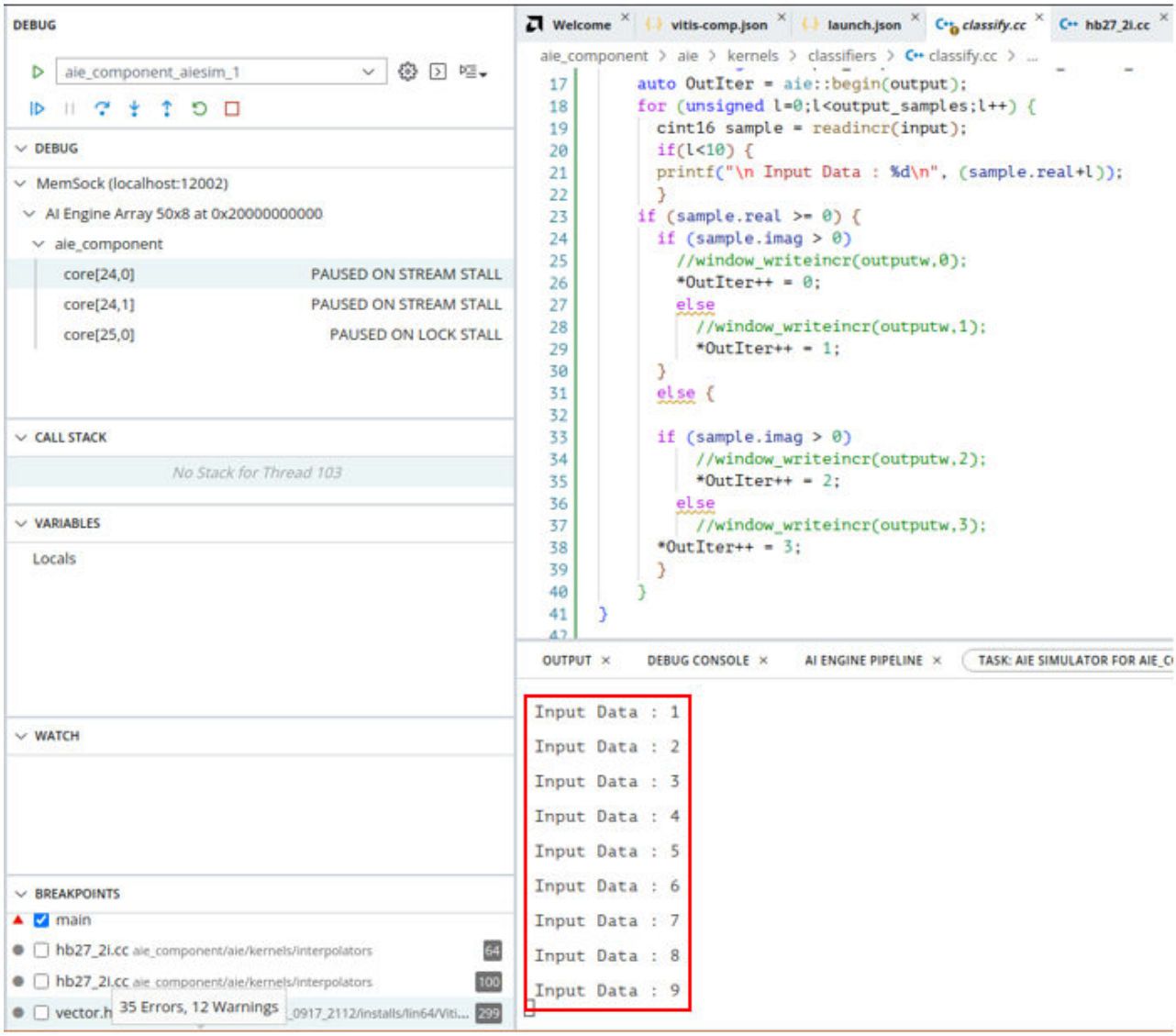
When adding the `printf()` statement in your kernel code you must also add the `--profile` option in the Run Configurations or Debug Configurations dialog box in the Vitis IDE. Add `--profile` to the Arguments tab of the Debug Configuration, along with whatever other options are already specified, as shown in the following figure.

Figure 39: **Enable Profiling**



Adding the `printf()` statements to your source code, results in the output of streaming data as it is processed by the kernel. The following figure shows an example of such output in the console window. This provides visibility to capture and debug the dataflow through streaming interfaces.

Figure 40: **Printing Data from Stream Interfaces**

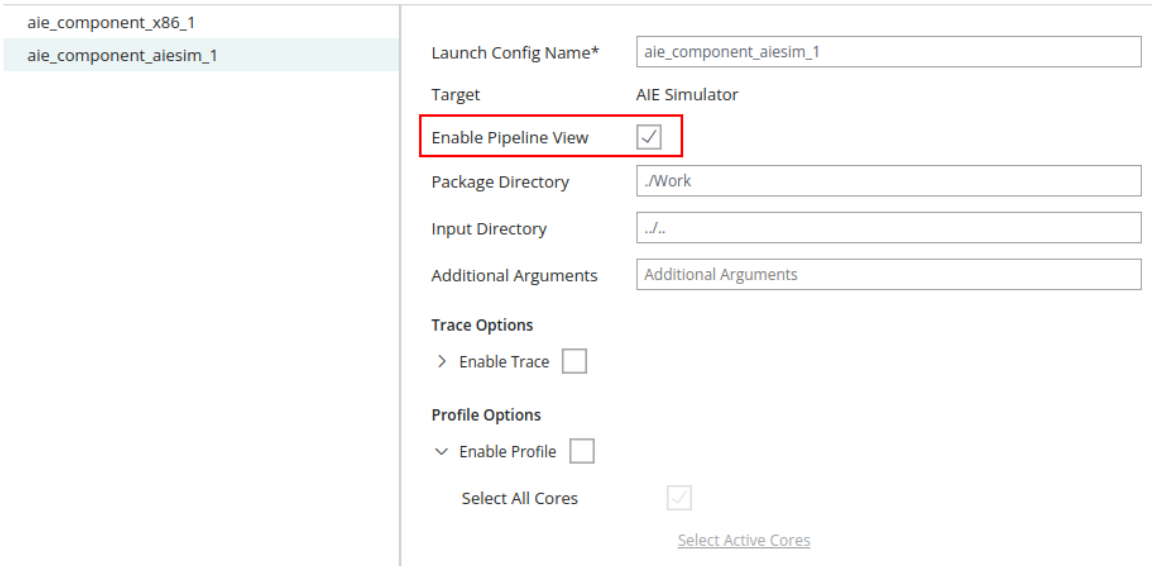


AI Engine Pipeline View

The AI Engine Pipeline view in the Vitis Unified IDE allows you to correlate instructions executed in a specific clock cycle with the labels in the Disassembly view. The underlying AI Engine pipeline is exposed in debug mode using the pipeline view. The Vitis Unified IDE supports viewing pipeline view for all the kernels in your graph.

To enable the Pipeline view select **Enable Pipeline View** from the Debug launch configuration after the project has been built successfully. The Pipeline view is available with the AI Engine simulator only.

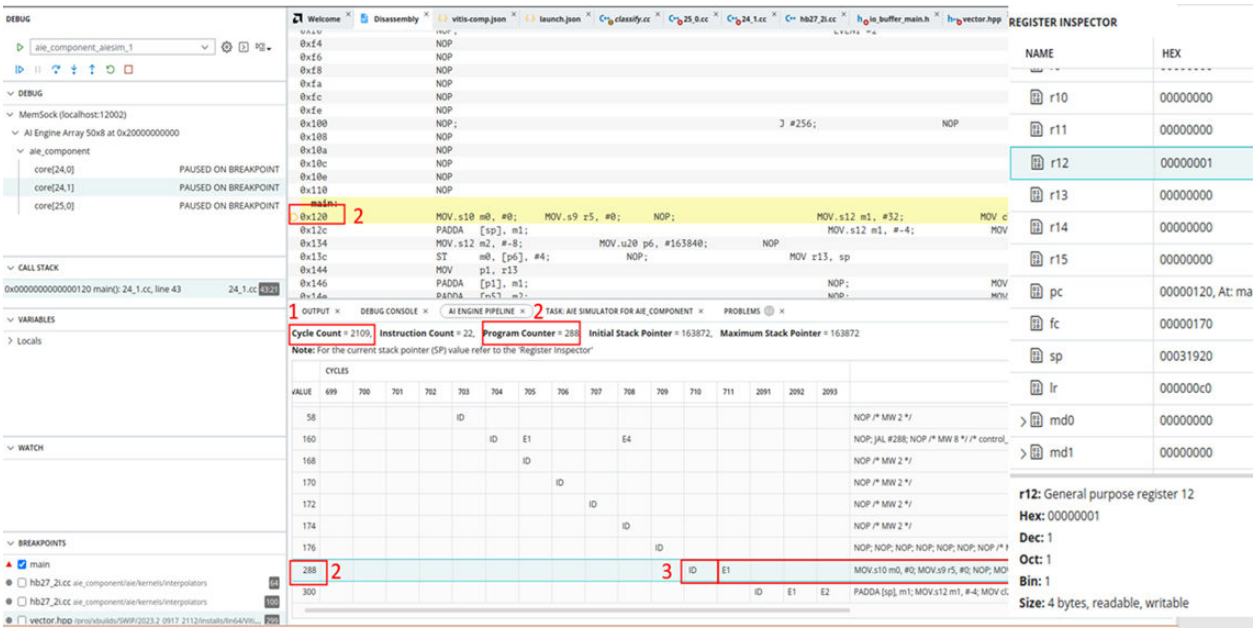
Figure 41: **Enable Pipelining**



TIP: *When Enable Profile is selected the Pipeline view is enabled as well.*

Click on **Debug** to start debugging the application. Note the Pipeline view shows up automatically in the Debugger Console window.

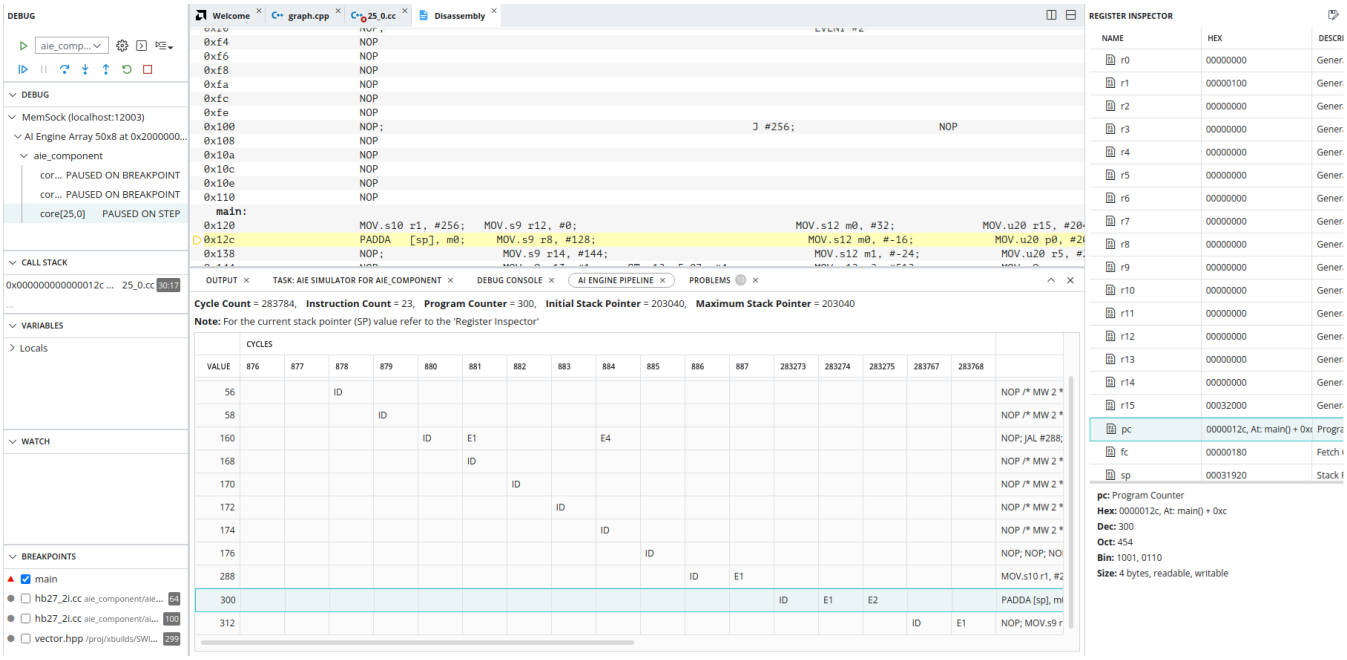
Figure 42: **Pipeline View**



Runtime statistics of the kernel in the pipeline view are highlighted in the previous figure.

1. AI Engine kernel cycle count
2. Program counter
3. ID = Instruction decode
4. E1-E7 are the AI Engine execution stages. Almost all operations in the scalar unit are scheduled in E1 stage of the pipeline besides non-linear operations. The vector unit scheduling spans from the ID stage to the E6 stage. Address Generation Units (AGUs) span over two pipeline stages. The address is ready in the E2 stage of the pipeline. For load units, the data will be available in the AI Engine from the memory module in the E7 stage. For the store unit, the data will be sent out from the AI Engine to the memory module in the E5 or E6 stage of the pipeline, depending on the type of instruction.
5. **Step Over** in the debug view is shown in the following figure. As shown the previous instruction, this is run in E1 stage. The result is updated in Register Inspector.

Figure 43: **Stepping Over with Pipeline View**



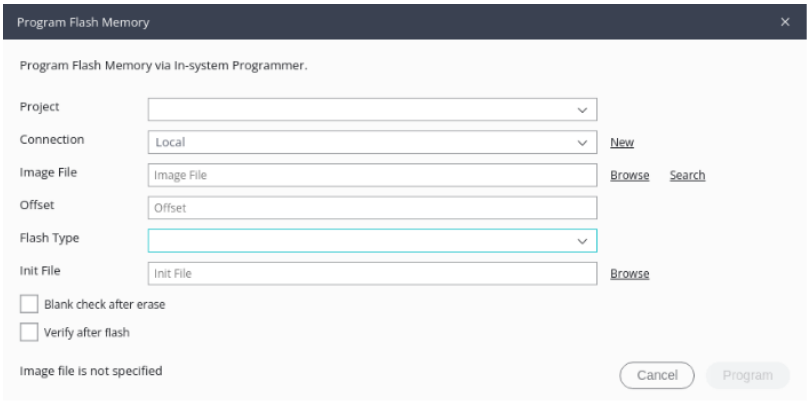
Programming Device and Flash Memory

The Vitis IDE has a feature for programming the board/part using JTAG, allowing you to bypass the need to copy the contents of `sd_card` to an actual SD card to boot on the board.

Programming Flash

After successfully building a System project for hardware, select **Program Flash** from the Vitis menu in the main menu. You should see a dialog box similar to the following:

Figure 44: **Program Flash Memory Dialog Box**



In this screen specify the **Image File**, which is generally the `BOOT.BIN`, any offset (if necessary for the flash memory), in addition to the **Init File**. The Init File is the partial PDI from the platform you are targeting. To make sure the flash is programmed properly, check the **Verify after flash** check box to confirm it was programmed properly. If the flash memory already has existing data, you can do a more secure erase by checking the **Blank check after erase** check box to make sure it is fully erased before programming.

Make sure that the **Packaging options** in the **System Project Settings** window has `--package.boot_mode=qspi` set.

Programming the Device

If you do not want to program the flash memory and only want to program the device through JTAG, there is an option under the Vitis menu to select **Program Device**.

Vitis Interactive Python Shell

As described in Launching the Vitis Unified IDE, the Vitis Unified IDE supports an interactive mode that lets you enter commands through the command-line interface (CLI) of a Python command shell. You can launch the tool in interactive mode as follows:

```
vitis -i
```

The Vitis command-line prompt is displayed:

```
Vitis [1]:
```

***Note*:** Type `help()` or `help(vitis)` from the interactive command prompt to explore the available command modules.

CLI examples can be found at `<Vitis_Installation_Dir>/cli/examples` directory.

The following topics describe some of the features of the Vitis Python command shell.

Auto-Completion of Python Statements

You can provide the starting few letters of a method and press tab to get the list of matching methods.

Basic Commands supported in Interactive Mode

The following basic commands are used in the Vitis interactive mode:

Table 37: **Commands Supported in Vitis Interactive Mode**

Commands	Description	Example
`cd`, `pwd`, `ls`	Basic unix commands	`Vitis [1]: <unix command>`
`edit`	Makes edits in the Python file	`Vitis [1]: edit <filename>.py`
`run`	Runs Python scripts	`Vitis [1]: run <filename>.pyVitis [1]: run -t <filename>.py` Displays timing information at the end of run
`source`	Runs Python script in batch mode	`$ vitis -source <filename>.py`

`env`	Sets and lists environment variables	Lists all env variables:
		`Vitis [1]: env`
		Gets the value of variable:
		`Vitis [1] env <var_name>`
		Sets the value of variable:
		`Vitis [1]: env XILINX_VITIS= <path>`

Table 37: **Commands Supported in Vitis Interactive Mode** *(cont'd)*

Commands	Description	Example

`history`	Prints input command history from the current session	Returns the command history with no line numbers.
		`Vitis [1]: history`
		Prints history with line numbers like 1, 2:
		`Vitis[1]: history -n 1-2`

`time`	Displays execution time, CPU time and wall clock times.	`Vitis [1]: time client=vitisng.create_client()`
`clear`	Clears the session	`Vitis [1]: clear`
`help()`	Gets help for Vitis commands	`Vitis [1]: help(vitis)Vitis [1]: help(vitis.create_client())`

Commands to Edit a File

Following are the commands to edit and execute code in file:

- **edit:** Used to bring up the default editor to edit the file.

```
Vitis [1]: edit <filename>.py
```

- **Launch a specific editor as external command:** Add an exclamation mark prior to the editor executable to launch the editor as external commands.

```
Vitis [1]: !vim <filename>.py
Vitis [1]: !nano <filename>.py
```

- **Multi-line editing in the CLI:** In the new Vitis CLI, you can edit and update the multi-line code using arrow keys.

Commands to Run Python Scripts

Following are the commands to edit and execute code in a file:

- **Run Python code in the next generation Vitis IDE CLI:** Run the named file

```
Vitis [1]: run test.py
```

Option `-t` displaces the timing information at the end of run

```
Vitis [1]: run -t test.py
```

- **Run Python code with the next generation Vitis IDE executable in batch mode:** `vitis` executable can run Python script in batch mode with `-source` option.

```
$ vitis -source $XILINX_VITIS/cli/examples/dc_use_case_1.py
```

Setting and Listing Environment Variables

Following are the commands to set and list environment variables:

- **env:** Used to list all environment variables/values
- **env var:** Used to get value for var
- **env var val:** Used to set value for var
- **env var=val:** Used to set value for var
- **env var=$val:** Used to set value for var, using python expansion if possible

```
Vitis [1]: env XILINX_VITIS
Out[1]: '/tools/Xilinx/Vitis/2025.2'
```

Command to Display History

Use `history` command to print recent input history.

```
Vitis [1]: ls
vadd/ logs/

Vitis [2]: history
ls
history
```

By default, input history is printed without the line numbers so that it can be pasted directly into an editor. Use `-n` to show line numbers.

```
Vitis [3]: history -n 1
1: ls
```

By default, the input history from the current session is displayed. Ranges of the history can be indicated using the following syntax:

```
Vitis [4]: history -n 1-2
1: ls
2: history
```

Command to Search the History

In the next generation Vitis IDE, you can search the history using the following commands:

- **Ctrl-p (or the up arrow key):** Used to access previous command in the history.
- **Ctrl-n (or the down arrow key):** Used to access the next command in the history.
- **Ctrl-r:** Used to reverse search through the command history.

Command to Display Execution Time

The following command is used to display the next generation Vitis IDE Python methods or expression execution time:

```
time
```

The CPU and wall clock times are printed, and the value of the expression (if any) is returned.

Note: Win 32, system time is always reported as 0, because it cannot be measured.

```
Vitis [1]: time client=vitisng.create_client()
Vitis Server started on port '59936'.
CPU times: user 71 ms, sys: 12.8 ms, total: 83.8 ms
Wall time: 17.2 s
```

Command to Clear the Session

The following command is used to clear the session:

```
clear
```

Library Function References

The new Vitis IDE can create, configure, and build a System project from the command-line using the `vitis` python library. Launch the Vitis tool in interactive mode using the following command:

```
vitis -i
```

Use the following commands while in the interactive mode to get help for the `vitis` library:

```
Vitis [1]: help(vitis)
Vitis [2]: help(vitis.create_client())
```

Python API: Managing Vitis IDE Components

The Vitis Unified IDE supports Python APIs to automate the management of Vitis IDE components. The Vitis IDE contains a Python Interpreter and script building logger. The Python interpreter supports Python APIs and executes the APIs in the Vitis tool. The script building logger converts the Vitis tool GUI user actions (create and build Vitis IDE components such as AI Engine component) into equivalent Python APIs. The Python file generated by the script building logger can be used to create and build components using the Python interpreter. The Vitis Unified IDE works on the server-client architecture. To execute Python APIs, you need to first establish the connection between the server and the client. Python APIs can be executed in command line mode or as a Python script. The AMD Vitis™ Unified IDE supports two methods to run Python APIs:

1. Run Python API in CLI (Command-Line Interface) mode: Executing the Python API in CLI mode is supported in interactive mode only. More details on interactive mode can be found at Vitis Interactive Python Shell.
2. Run Python API in a Python Script: The Vitis Unified IDE supports the execution of Python script in batch mode as well as in an interactive mode.

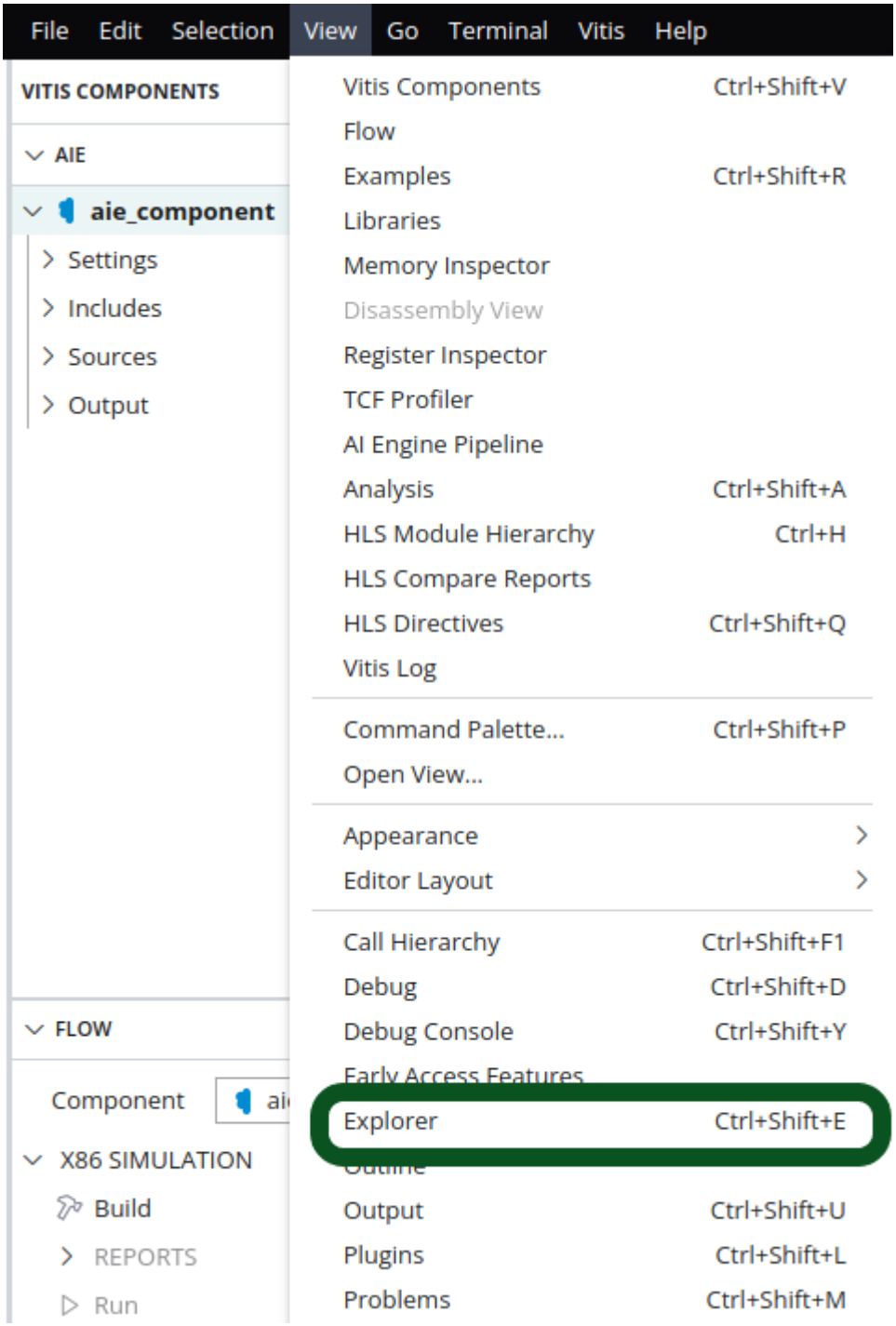
```
In batch mode:
$ vitis -s <.py>
In Interactive mode:
vitis [1]: run <.py>
```

Before running the script, you must set up the environment variables for the Vitis Unified IDE. Refer to Setting Up the Vitis Tool Environment in the *AI Engine Tools and Flows User Guide* (UG1076) to set up the Vitis Unified IDE environment. The Vitis IDE supports Python APIs for project flow management, creation and building of the following Vitis components:

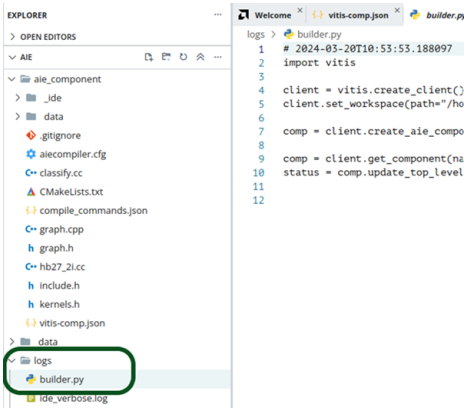
- AI Engine Component
- HLS Component
- Platform Component
- Application Component
- System Project

The script building logger logs actions from the GUI and saves the file as `builder.py`. To access `builder.py`, follow the steps below:

1. Navigate to **View → Explorer** in the Menu bar or press `Ctrl+Shit+E`.



2. In the **Explorer** window, go to `logs: builder.py`. The Python interpreter creates `builder.py`, which logs the equivalent Python APIs for the commands used in the Vitis IDE. You can run `builder.py` later to automate the process.

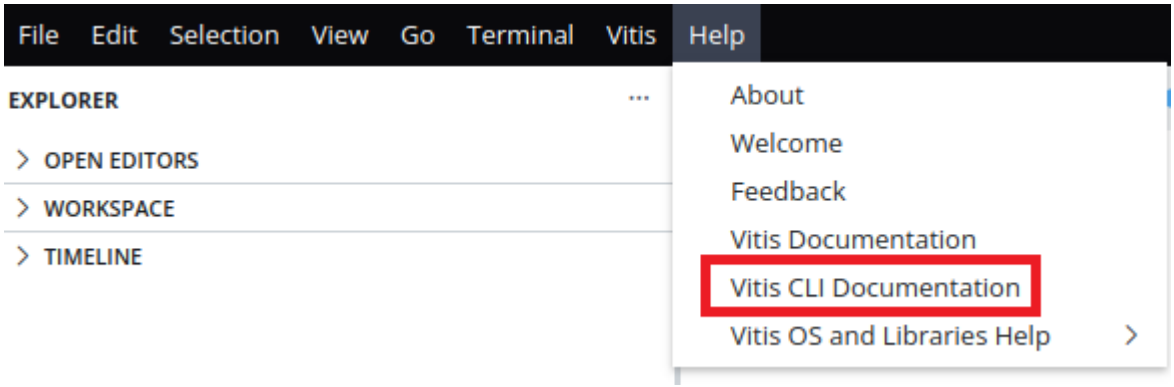


Some Python APIs are helpful in managing client life cycles, create Vitis workspaces and manage Vitis project flows. These APIs are described in the following table. Refer to the Vitis API CLI documentation for more details.

Table 38: **Python APIs: Manage Client**

Python API	Description	Example
create_client	Creates a client instance.	client=vitis.create.client()
dispose	Closes all client connections and terminates the connection to the server.	client.dispose()
exit	Closes the session.	exit()

For more details on all the Python APIs supported by Vitis, refer to `<vitis_installation_path>/cli/api_docs/build/html/vitis.html` or in the Vitis Unified IDE, go to **Help → Vitis CLI Documentation**.



Python examples are provided with the installation for reference:
`<vitis_installation_path>/cli/examples`

After creating and building the components and system project, the workspace can be directly opened in the Vitis IDE using the command below:

```
vitis -w <workspace_path>
```

Components and system projects are opened in the Vitis IDE with build and created status=done, if they are created and build through Python APIs.

The Python APIs used for component creation and build are explained in the following sections.

Create and Build AI Engine Component

The Vitis Unified IDE supports Python APIs to create and build an AI Engine component. You can either create and build the component through the GUI (see Creating an AI Engine Component in the *AI Engine Tools and Flows User Guide* (UG1076)) or using Python APIs. Some of the Python APIs that are required to create and build AI Engine components are shared in the following table. For details of all Vitis supported Python APIs, refer to the link: `<vitis_install_path>cli/api_docs/build/html/vitis.html`. Multiple config files are not supported in the Vitis IDE to create the component. If you would like to use a custom config file to create the component, remove the config file generated by the Vitis tool (default) as shown below:

1. Remove the tool generated configuration file using the Python command:

   ```
   remove_cfg_file('<config_file.cfg>')
   ```

2. Add the custom configuration file using the Python command:

   ```
   add_cfg_file('<custom_config_file.cfg>')
   ```

Table 39: **Python APIs: AI Engine Component**

Python API	Description	Python API Example
`create_aie_component`	Create an AI Engine Component	`aie_comp=client.create_aie_component(name=my_component, platfrom=<.xpfm or .xsa>, template="empty")`

`import_files`	Import files to the AI Engine component	To import files:
		`aie_comp.import_files(from_loc="<input_folder_path>", files=['file1.cpp','file2.txt'])`
		To import folders:
		`aie_comp.import_files(from_loc="<input_folder_path>)`

`add_cfg_file`	Add configuration file to the AI Engine component	`aie_comp.add_cfg_file('<.cfg file path'>)`
`remove_cfg_file`	Remove configuration file from the component.	`aie_comp.remove_cfg_file('<.cfg file>')`

Table 39: **Python APIs: AI Engine Component** *(cont'd)*

Python API	Description	Python API Example
`update_top_level_file`	Update the top level file of AI Engine component	`aie_comp.update_top_level_file('<main_graph>.cpp')`
`build`	Initiate the build of AI Engine Component for x86 and AIESIM/HW	`aie_comp.build(target=hw)` For x86 simulator (default): `target=x86sim` For hardware: `target=hw`
`report`	Print the component information	`aie_comp.report()`
`clean`	Clean the AI Engine component for the specified build target	`aie_comp.clean(target=<hw OR x86sim>)`

The Python script to create an AI Engine component and build for `target=Hardware` is described below:

```
# Add package: Vitis Python CLI
import vitis

# Create a Vitis client object -
client = vitis.create_client()

# Set Vitis Workspace
client.set_workspace(path=<workspace_location>)

# Create aie component.
aie_test_comp = client.create_aie_component(name = comp_name, platform =
<absolute path of platform>, template = "empty_aie_component")

# Import source files to the component
aie_test_comp.import_files(from_loc = <absolute path of source folder>,
files = ['file1.cpp', 'file2.h', 'file3.h', 'kernel1.cc', 'kernel2.cc'])

# Set top file from the imported source
aie_test_comp.update_top_level_file(top_level_file = 'main_graph.cpp')

# Build component on target Hardware
aie_test_comp.build(target="hw")

# Print component information
aie_test_comp.report()
```

After the build, the AI Engine component outputs (`libadf.a`, Work folder etc.) are created at: `<workspace>/<comp_name>/build/hw/`

Create and Build HLS Component

The Vitis Unified IDE supports creation and building of HLS components using Python APIs. The following Python APIs can be used to create and build HLS components:

Table 40: **Python APIs: HLS Component**

Python API	Description	Python API Example
`create_hls_component`	Create a HLS component	`hls_comp = client.create_hls_component(name=<hls_comp_name>,cfg_file = ["hls_config.cfg"],template = "empty_hls_component")`
`add_cfg_file`	Add configuration file to the component	`hls_comp_cfg=client.add_cfg_file(cfg_file=<config_file_path.cfg>)`

`set_value`	Set the value of the key (directive) in a specific section of a configuration file. Any earlier values for the key will be overwritten.	`hls_comp_cfg.set_values(section='hls', key='flow_target', values='vitis')`
		To set value in config file:
		`[hls]`
		`flow_target=vitis`

`run`	Run a specified operation on the HLS component.	`hls_comp=run(operation='SYNTHESIS')Operation = SYNTHESIS`

The following is the Python script to create and build a HLS component:

```
# Add package: Vitis Python CLI
import vitis

# Create a Vitis client object -
client = vitis.create_client()

# Set Vitis Workspace
client.set_workspace(path=<workspace_location>)

# Create HLS component
hls_comp = client.create_hls_component(name='HLS_comp',cfg_file =
["hls_config.cfg"],template = "empty_hls_component")

# Add Configuration File
hls_comp_cfg = client.add_config_file(cfg_file='/hls_config.cfg')

#Add values to the configuration file
hls_comp_cfg.set_value(key='part', value=part)
hls_comp_cfg.set_value(section='hls', key='flow_target', value='vitis')
hls_comp_cfg.set_value(section='hls', key='package.output.syn', value='1')
hls_comp_cfg.set_value(section='hls', key='package.output.format',
value='xo')
......

# Build HLS component in Vitis Mode
hls_comp.run(operation="SYNTHESIS")
hls_comp.run(operation="PACKAGE")
```

After building the HLS component, the output files can be found at:

```
workspace/HLS_comp/<kernel_name>/
```

Create and Build System Project

The Vitis Unified IDE supports Python APIs to create and build a system project. After building the AI Engine component and HLS component, the system project is created and built. The system project has a binary container where components such as AI Engine and HLS kernels must be added. After adding the components, update the configuration file to ensure that the appropriate v++ linking directives and connections are declared under the Connectivity section. If you want to use a custom configuration file, you must remove the configuration file created by the Vitis tool to avoid build issues connected to the config file. By default, the system project build considers the tool generated config file. To use the custom config file to build the system project, follow the steps below:

1. Remove configuration file:
   `remove_cfg_files([<config_file.cfg>],'<binary_container_name>')`
2. Add custom configuration file:
   `add_cfg_files(['custom_config_file.cfg'],name='<container_name>')`

Building the system project (v++ linking) through the Python API builds all the components added in the binary container, and also builds the system.

Table 41: **Python APIs: System Project**

Python API	Description	Python API Example
create_sys_project	Creates a system project for the given template.[1]	`proj = client.create_sys_project(name ='system_project', platform=<platform_path>)`
add_container	Adds a binary container with given kernels, and config files.	`proj.add_container(name='syste m_prj_lab1')`
add_component	Adds the specified component to the given system project	`proj.add_component(name='aie_c omponent', container_name=['system_prj_la b1'])` `proj.add_component(name='hls_c omponent', container_name=['system_prj_la b1'])`
set_value	Sets the value of the key in a specific section of a config file. Any earlier values for the key will be removed. The same is used as the one in HLS.	`cfg.set_value(key='save-temps', value='1')` `cfg.set_value(key='export_arch ive', value='1')`
add_values	Adds more repeated values for the key in a specific section of a config file. This will add one key=value assignment for each value. Any earlier values for the key will not be removed.	`cfg.add_values(section='connec tivity', key='sc', values=['mm2s_1.s:ai_engine_0. DataIn1'])` `cfg.add_values(section='connec tivity', key='sc', values=['ai_engine_0.DataOut1: s2mm_1.s'])`
remove_cfg_files	Remove configuration file from the component	`proj.remove_cfg_files(['<confi g_file.cfg>'],'<container_name >')`

Table 41: **Python APIs: System Project** *(cont'd)*

Python API	Description	Python API Example
add_cfg_file	Add configuration file to the component.	`proj.add_cfg_files(['<config_file.cfg>'], name='<container_name>')`
build	Initiates the build of a system project for the given build target.	`proj.build(target='hw')`

Notes:

1. Accelerated flows are not supported for the Embedded installer.

The Python script to create a system project and build for `target=Hardware` to export Vitis metadata archive as illustrated below:

```
# Add package: Vitis Python CLI
import vitis

# Create a Vitis client object -
client = vitis.create_client()

# Set Vitis Workspace
client.set_workspace(path=<workspace_location>)

# Create system project
proj = client.create_sys_project(name='system_project',
platform=platform_path)
# Add a binary container, and add components to it
proj.add_container(name='system_prj_lab1')
proj.add_component(name='aie_component',
container_name=['system_prj_lab1'])
proj.add_component(name='mm2s', container_name=['system_prj_lab1'])
proj.add_component(name='s2mm', container_name=['system_prj_lab1'])
# Populate the link config file
cfg = client.get_config_file(proj.project_location+'/hw_link/
system_prj.cfg')
cfg.set_value(key='debug', value='1')
cfg.set_value(key='save-temps', value='1')
cfg.set_value(section='advanced', key='param',
value='compiler.addOutputTypes=hw_export')
cfg.add_values(section='connectivity', key='sc',
values=['mm2s_1.s:ai_engine_0.DataIn1'])
cfg.add_values(section='connectivity', key='sc',
values=['ai_engine_0.DataOut1:s2mm_1.s'])

# Build System Project
proj.build(target='hw')
```

After building the system project, the output is generated at:

```
workspace>system_project>build>hw>hw_link
```

Create and Build Platform Component

The Vitis unified IDE supports python APIs to create and build the platform component. The platform component (`.xpfm`) is created from extensible XSA and system software files of the operating system (for example, PetaLinux). The platform component with no system software files for any operating system is to run the baremetal applications.

Table 42: **Python APIs: Platform Component**

Python API	Description	Python API Example

`create_platform_component`	Creates a new platform	For Baremetal:
		`platform_comp=client.create_platform_component(name='platform', hw_design=<hw_xsa_path>)`
		For Linux:
		`client.create_platform_component(name = platform_name, hw_design = <hw_xsa_path>, cpu = "psu_cortexa53", os = "linux", domain_name = "linux_a53")`

`add_domain`	Adds the domain to the platform	`microblaze = platform.add_domain(name = "microblaze", cpu = "microblaze_0", os = "standalone")`
`build`	Initiates a build of platform component	`platorm_comp.build()`

The Python script to create a platform component for baremetal application is described below:

```
# Add package: Vitis Python CLI
import vitis

# Create a Vitis client object -
client = vitis.create_client()

# Set Vitis Workspace
client.set_workspace(path=<workspace_location>)

# Create Platform Component:
platform_comp=client.create_platform_component(name='platform',
hw_design=<hw_extensible_xsa_path>)

# Build the platform component
platform_comp.build()
```

After building the platform component, the output products are generated at: `workspace > platform > export > platform > platform.xpfm`

Create and Build Application Component

The Vitis Unified IDE supports Python APIs to create the application component. An application component requires an XPFM platform. Starting from a fixed hardware XSA, first create an XPFM with `create_platform`.

Table 43: **Python APIs: Application Component**

Python API	Description	Python API Example
`create_app_component`	Create an application component.	`app_comp = client.create_app_component(name ='app_comp', platform = <platform_path>, template = "empty")`
`set_app_config`	Set the value of the provided build setting. Values can be single value or a list of values.	`app_comp.set_app_config(key = 'USER_LINK_LIBRARIES', values = 'xrt_coreutil')`
`set_sysroot`	Set `sysroot` for the app component.	`app_comp.set_sysroot(<sysroot_dir_path>)`
`build`	Initiates the build of an application component.	`app_comp.build(target="hw")`

The Python script to create an application component and build for `target=Hardware` can be seen as:

```
# Add package: Vitis Python CLI
import vitis

# Create a Vitis client object -
client = vitis.create_client()

# Set Vitis Workspace
client.set_workspace(path=<workspace_location>)

# Create an application component:
app_comp = client.create_app_component(name ='ps_comp', platform =
<platform_path>, template = "empty")

# Import sources to the component
app_comp.import_files(from_loc = <app_src_path>, files = ['host.cpp',
'host.h','data.h'], dest_dir_in_cmp = 'src')

#Add the directives in the USERconfig.cmake file
app_comp.set_app_config(key = 'USER_LINK_LIBRARIES', values =
'xrt_coreutil')
......

# Setting sysroot
app_comp.set_sysroot(<sysroot_dir_path>)

# Build Component
app_comp.build(target="hw")
```

After building an application component, the output products are generated at:

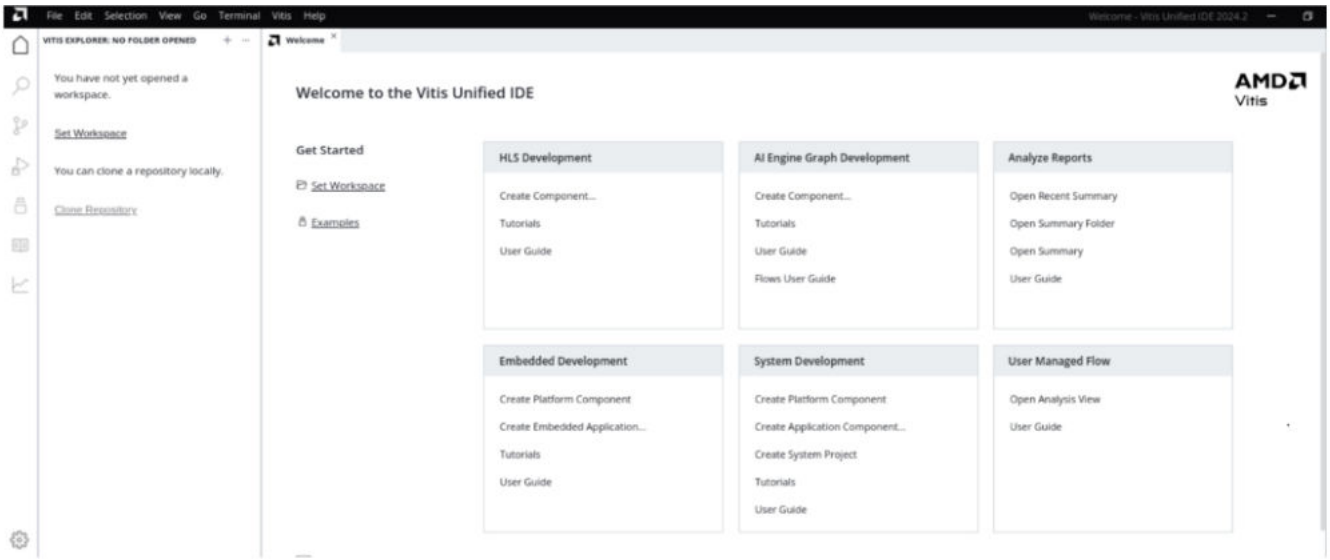
```
workspace > app_component > build > hw
```

Vitis IDE Examples

To accelerate familiarization of Vitis Components and workflows, the Vitis Unified IDE provides various examples and coding style templates to quickly get started.

The examples are accessible through the view menu, examples icon or the welcome screen to ensure quick access to user. Additionally, the welcome screen provides links to tutorials and user guides.

Figure 45: **Vitis Unified IDE Examples**



***Note*:** If the Welcome screen has been closed, it can be reopened again via the **Help** menu.

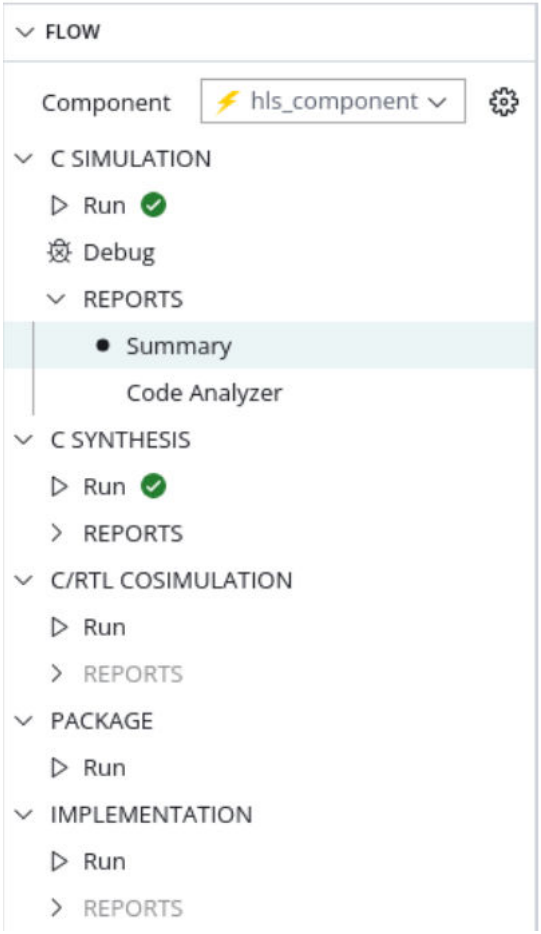
Managing the Vitis HLS Components in the Vitis Unified IDE

See the following sections for details on managing the Vitis HLS Components in the AMD Vitis™ Unified IDE.

Using the Flow Navigator

For the HLS component in the Vitis Unified IDE, the Flow Navigator is a process flow representation with each step in the process represented by actions that you can launch to work through the flow. All reports and report viewers are also available through the Flow Navigator as each step is completed.

Figure 46: **Flow Navigator**



The process steps for the HLS component include:

- **C SIMULATION:** Runs C Simulation and the Code Analyzer using the specified source code and test bench and any input data provided as described in Run C-Simulation.
- **C SYNTHESIS:** Opens the C Synthesis dialog box, and lists the available reports after synthesis has been run, as discussed in Running C Synthesis.
- **C/RTL CO-SIMULATION:** Opens the C/RTL Cosimulation dialog box, and lists the available reports after simulation has been run, as described in Running C/RTL Co-Simulation.
- **PACKAGE:** Generates the Vivado IP (`.zip`) or Vitis Kernel (`.xo`) and exports it from the synthesized RTL design as described in Packaging the RTL Design.
- **IMPLEMENTATION:** Run Vivado synthesis and implementation to generate more detailed utilization and timing reports as described in Running Implementation.

The Flow Navigator also has a few features to help you view the associated files and design state. As seen in the preceding figure, the Open Settings command is available next to the Component menu and each process step in the design flow. Select **Open Settings** next to Component to open the `vitis-comp.json` file for the component. Select it next to the process step to open the Config File Editor.

You can cancel a running process from within the Flow Navigator by using the **Cancel Run** command. This command is hidden by default and is revealed by hovering over the running step to reveal the command.

The Flow Navigator displays a warning sign (⚠) when certain steps are not complete, or need to be rerun due to changes in the design source or associated files. Hovering over the warning displays a message indicating the possible problem.

Creating Vitis HLS Components

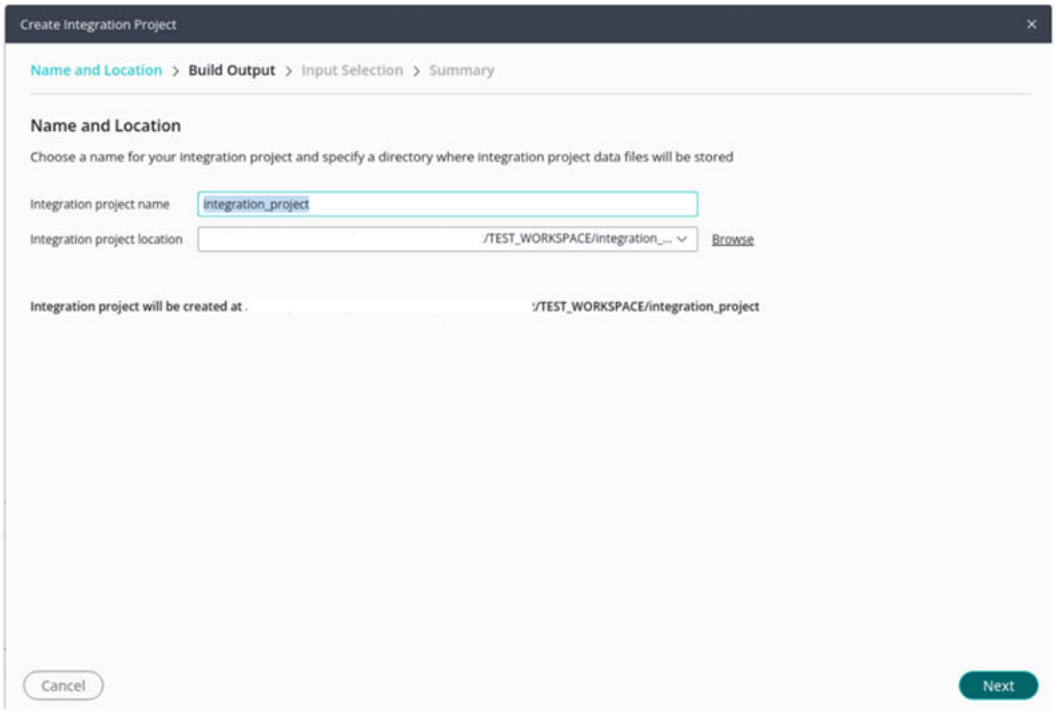
There are a number of ways to create an HLS component in the Vitis Unified IDE. The following steps describe a recommended approach.

TIP: *This section presents the HLS component in the Vitis IDE. For a command-line approach refer to Creating HLS Components from the Command Line.*

1. With the Vitis IDE opened, from the main menu select **File → New Component → HLS**.

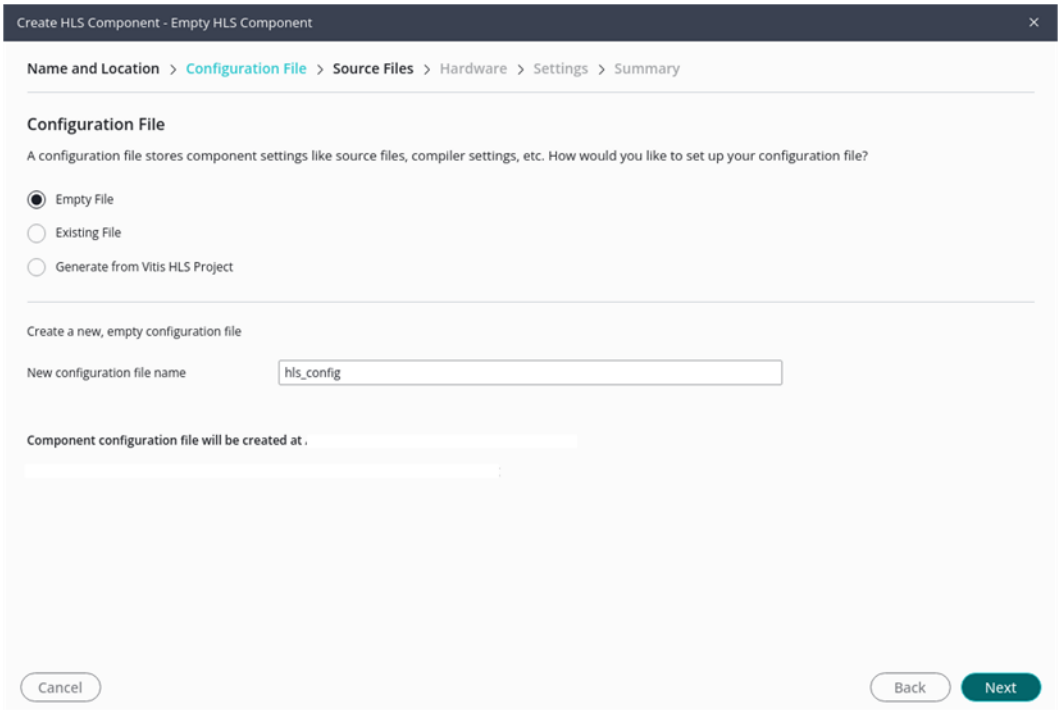
   TIP: *You can also select **New Component** on the Welcome page, or from the right-click menu in the Explorer view.*

2. This opens the Choose Name and Location page of the Create HLS Component wizard as shown below.



Enter a Component name and Component location and select **Next**.

3. This opens the Configuration File page of the wizard as shown below. The HLS component configuration file contains commands and directives used for the synthesis, simulation, and export of the component.

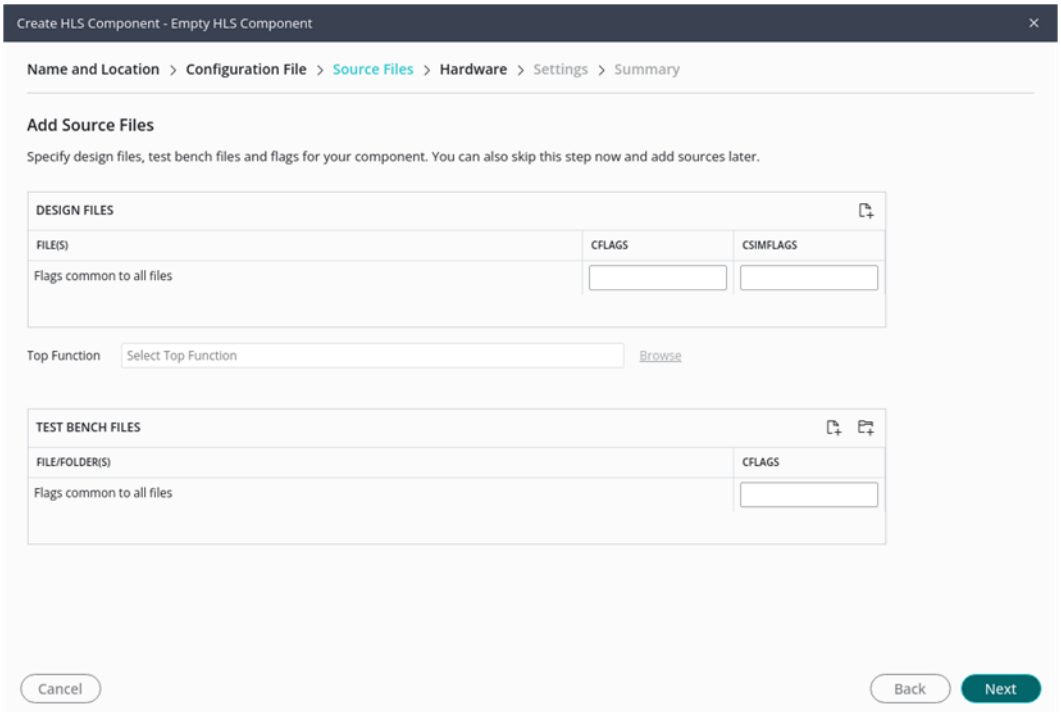


The Configuration File page of the Create HLS Component wizard provides three ways to add a configuration file to your component:

- Empty File: Specifies the creation of a new empty configuration file to be populated after component creation. Selecting this option lets you specify a name in the New configuration file name field.
- Existing File: Specifies the use of an existing configuration file. Selecting this option lets you enter the path and name in the File field, or select **Browse** to locate the file.
- Generate from Vitis HLS Project: This option generates a new config file from an existing Vitis HLS project `hls.app` file. The tool lets you identify the `hls.app` file inside the project, and enter a New configuration file name for the config file generated from the `hls.app` file.

Choose the method of adding the config file to your HLS component, and specify a name or files as needed. Select **Next** to proceed to the Source Files.

4. The Source Files page of the wizard opens as shown below.

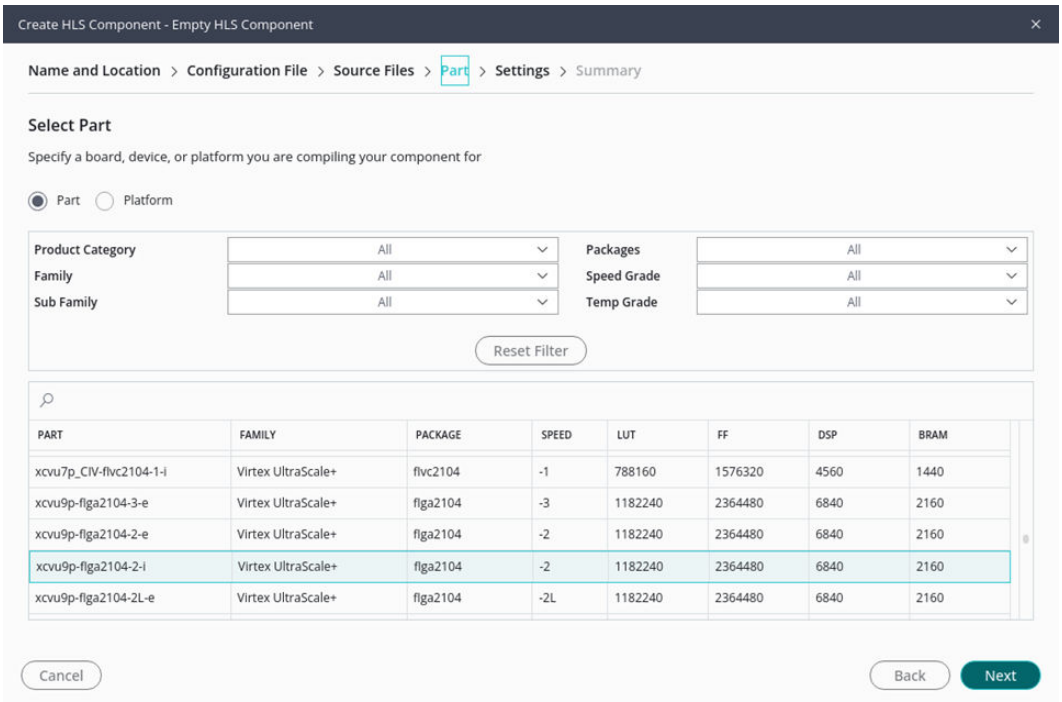


Under Design Files you can you can select **Add files** to select the primary source file for the design, or add multiple files as needed. After adding one or more source files, you can identify the Top Function which is the function to be synthesized as the top-level function for the HLS component. Any functions called by the top-function will also be synthesized into RTL.

Under Test Bench Files you can select **Add files** or select **Add folders** to add files containing test bench and associated files for the HLS component.

For both design files and test bench files you can add CFLAGS and CSIMFLAGS to specify directives to be used during compilation and simulation. You can specify directives to be applied to all files or to specific files from each category. After adding files and folders and directives, select **Next** to proceed.

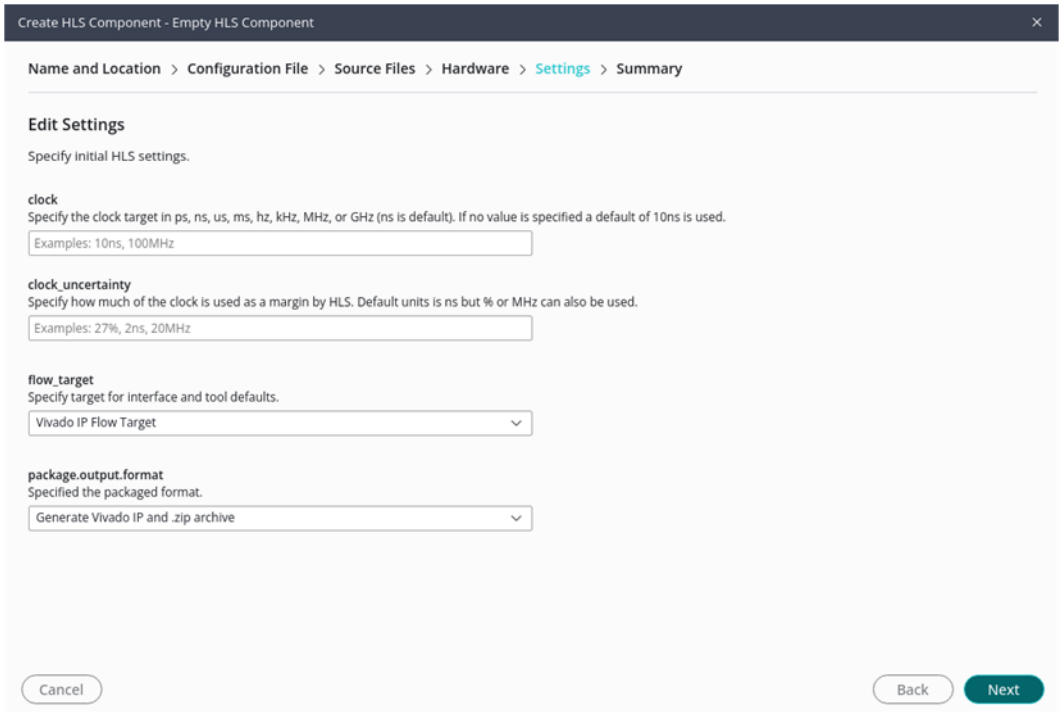
5. The Select Part page of the wizard is opened as shown below.



Select the **Part** or select the **Platform** for the HLS component and select **Next** to proceed.

> **TIP:** *If you select **Platform** when creating the HLS component, the Vitis Unified IDE extracts the part from the specified platform and adds* `part=` *to your configuration file.*

6. This displays the Settings page of the wizard as shown below.



The Settings page lets you specify a clock period or frequency for the design, and a clock uncertainty. The default clock for the tool is a clock period of 10 ns (or 100 MHz), and a clock uncertainty of 27%.

The Settings page also lets you specify the flow_target for the HLS component build process as being either to generate a Vivado IP or a Vitis kernel. The interface requirements of the Vivado IP or a Vitis kernel are different. The default flow_target is the Vivado IP flow.

Below the flow_target you can also specify the package.output.format for the tool to generate when packaging the design. The default output format is the Vivado IP, which generates a `.zip` file to be added to your IP catalog, letting you use the HLS generated RTL design in other designs.

Note: Selecting the Vitis flow_target does not automatically change the output format to Vitis kernel. The default is still Vivado IP, but you can manually change the output format if needed.

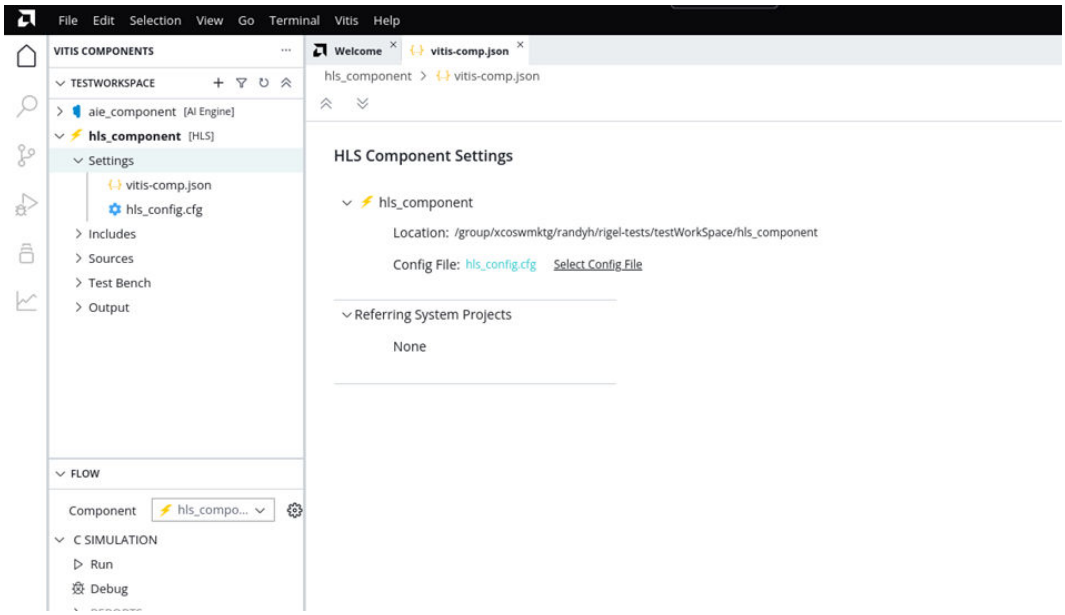
Also keep in mind that the HLS component uses a bottom-up design flow, generating only RTL output until you package the output. The bottom-up flow lets you develop multiple design iterations as you sort through the design challenges, and only package the output when the design is ready. This approach saves significant time each time you run Synthesis or Simulation.

Select **Next** to proceed to the Summary page.

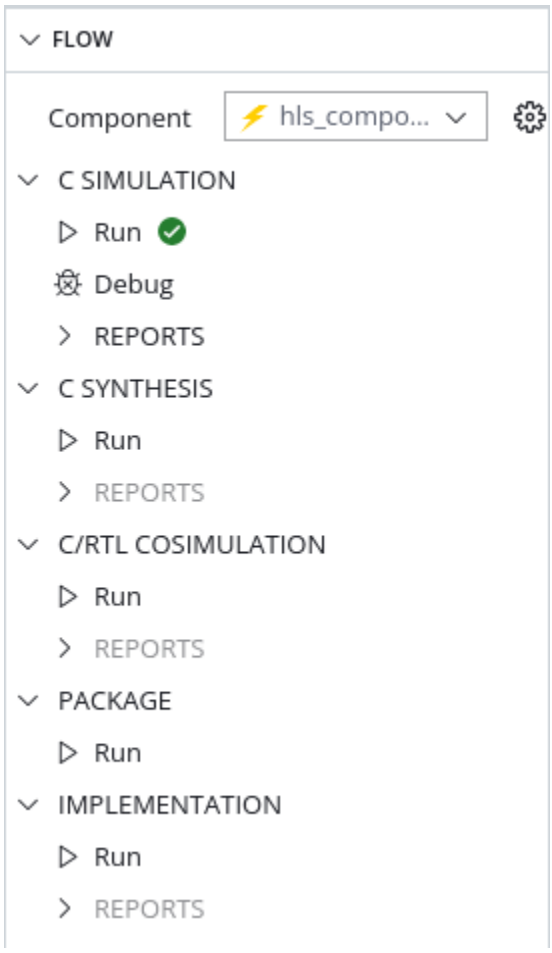
7. The Summary page reflects the choices you have made on the prior pages. Review the summary and select **Finish** to create the HLS component, or select **Back** to return to earlier pages and change your selections.

When the HLS component is created the `vitis-comp.json` file for the component is opened in the central editor window, and the HLS component becomes active in the Flow Navigator as shown below.

TIP: *When multiple HLS components are opened, the component name appears in the tab alongside the `vitis-comp.json` file name to help you identify the different components.*



After creation, the HLS component becomes the active component in the Flow Navigator, or you can select the **Component** to make it the active component in the tool. When the HLS component is the active component, the Flow Navigator enables running C Simulation, C Synthesis, C/RTL Co-simulation, Package, and Implementation as shown in the following image.



To manage the actions of these steps however, you will need to edit the HLS config file to achieve your design objectives as described below.

Building and Running an HLS Component

The AMD Vitis™ Unified IDE and `v++` common command line provide the ability to create an HLS component using bottom-up design techniques to ensure the component meets design requirements before integrating it with other elements into a system design. In an HLS component, the tool synthesizes a C or C++ function into RTL code for implementation in the programmable logic (PL) region of an AMD Versal™ adaptive SoC, AMD Zynq™ MPSoC, or AMD FPGA device.

An HLS component can be used to develop and export:

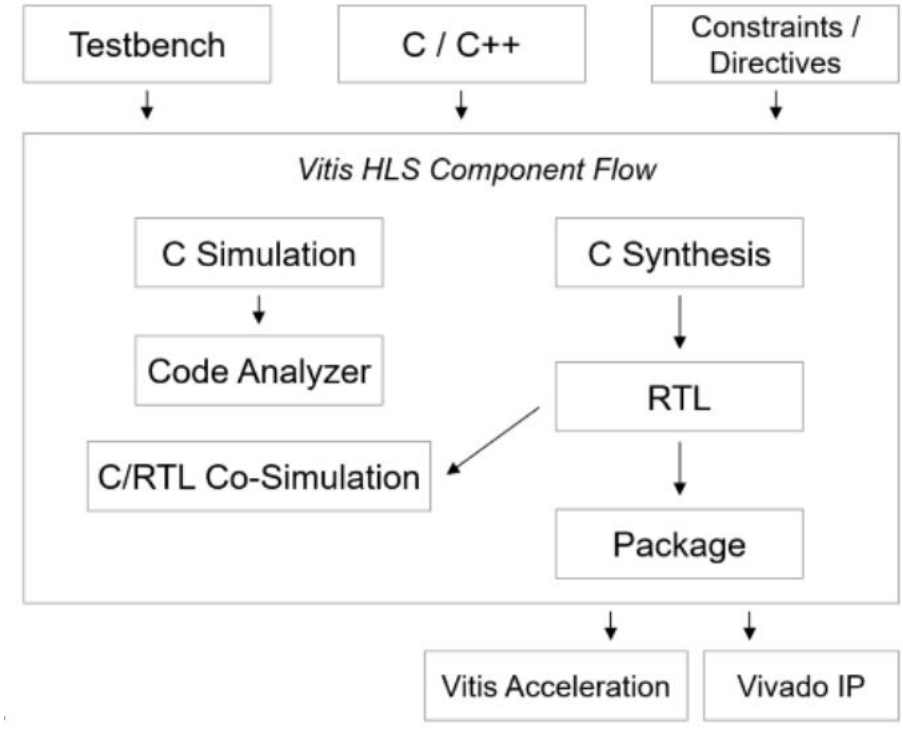
- Vivado IP to be integrated into hardware designs using the Vivado Design Suite and the Embedded Software Development flow as described in *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400).

- Vitis kernels for use in hetergeneous compute systems and Data Center application acceleration as described in *Embedded Design Development Using Vitis* (UG1701).

The following are the steps for the development of the C++ function.

1. Architect the C/C++ algorithm based on Design Principles in the *Vitis HLS User Guide* (UG1399)
2. Verify the logic of the source code with a C test bench (C-Simulation)
3. Analyze the parallelism and performance of the design (Code Analyzer)
4. Generate the RTL from the source code (C-Synthesis)
5. Verify the generated RTL module with the C test bench and RTL logic simulation (C/RTL Co-Simulation)
6. Review and analyze the HLS synthesis reports and co-simulation reports (Analyze)
7. Repeat previous steps until performance goals are met

Figure 47: **Vitis HLS Development Flow**

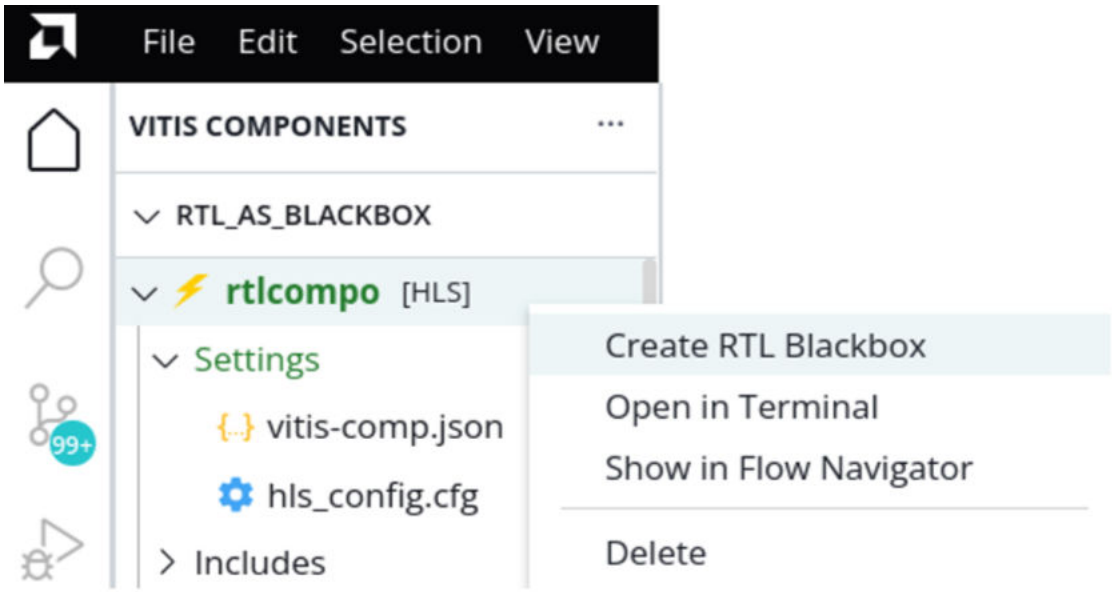


The HLS component implements the design based on the target flow, default tool configuration, configuration commands, and any optimization pragmas or directives you specify for the design. The following sections describe creating and building the HLS component.

Using the RTL Blackbox Wizard

Navigate to the project, right-click to open the **RTL Blackbox Wizard** as shown in the following figure:

Figure 48: **Opening RTL Blackbox Wizard**



The Wizard is organized into pages that break down the process for creating a JSON file. To navigate between pages, click **Next** and select **Back**. Once the options are finalized, you can generate a JSON by clicking **OK**. Each of the following section describes each page and its input options.

C++ Model and Header Files

In the Blackbox C/C++ files page, you provide the C++ files which form the functional model of the RTL IP. This C++ model is only used during C++ simulation and C++/RTL co-simulation. The RTL IP is combined with Vitis HLS results to form the output of synthesis.

Figure 49: **Blackbox C/C++ Files Page**

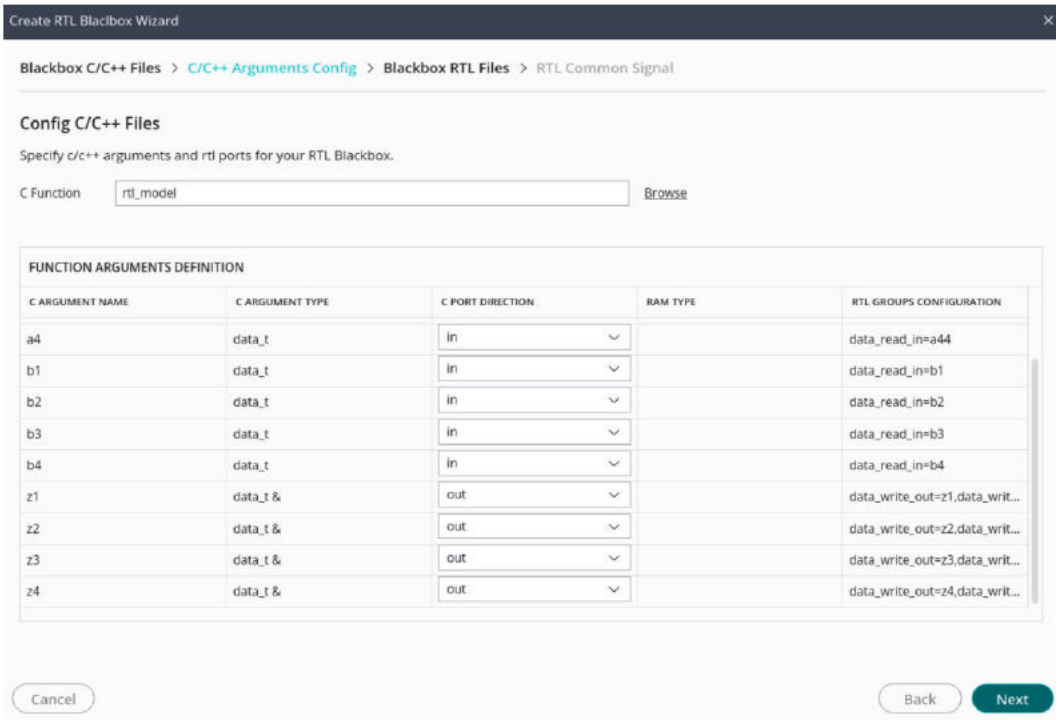
In this page, you can perform the following:

- Click **Add Files** to add files.
- Click **Edit CFLAGS** to provide a linker flag to the functional C model.
- Click **Next** to proceed.

The C File Wizard page lets you specify the values used for the C functional model of the RTL IP. The fields include:

- **C Function:** Specify the C function name of the RTL IP.
- **C Argument Name:** Specify the name(s) of the function arguments. These should relate to the ports on the IP.
- **C Argument Type:** Specify the data type used for each argument.
- **C Port Direction:** Specify the port direction of the argument, corresponding to the port in the IP.
- **RAM Type:** Specify the RAM type used at the interface.
- **RTL Group Configuration:** Specifies the corresponding RTL signal name.

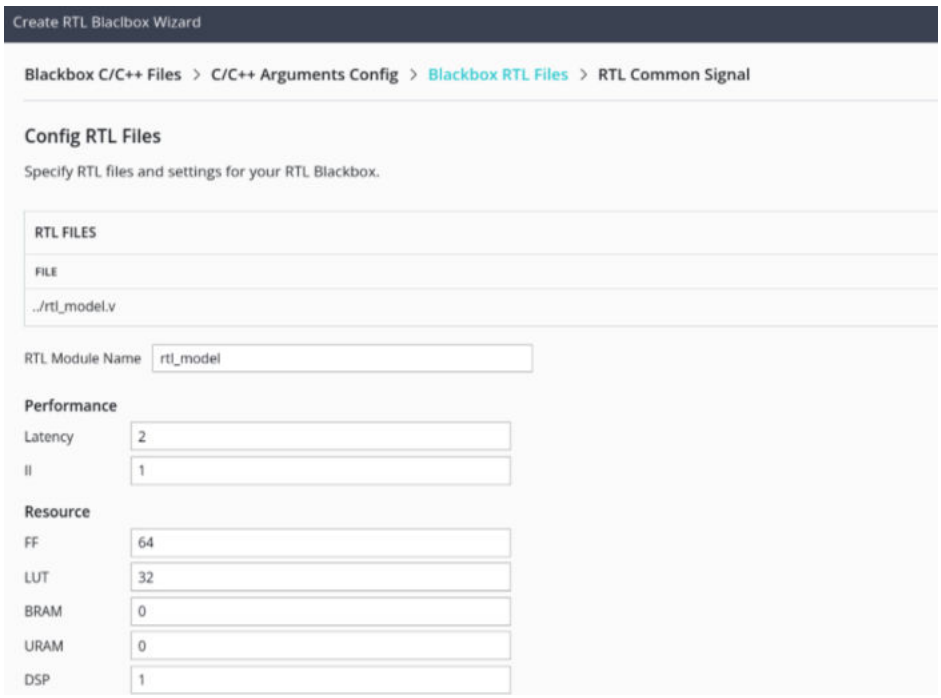
Figure 50: **C File Wizard Page**



Click **Next** to proceed.

RTL IP Definition

Figure 51: **RTL Blackbox Wizard**



The RTL Wizard page lets you define the RTL source for the IP. The fields to define include:

- **RTL Files:** This option is used to add or remove the pre existing RTL IP files.
- **RTL Module Name:** Specify the top level RTL IP module name in this field.
- **Performance:** Specify performance targets for the IP.
  - **Latency:** Latency is the time required for the design to complete. Specify the Latency information in this field.
  - **II:** Define the target II (Initiation Interval). This is the number of clocks cycles before new input can be applied.
- **Resource:** Specify the device resource utilization for the RTL IP. The resource information provided here will be combined with utilization from synthesis to report the overall design resource utilization. You should be able to extract this information from the Vivado Design Suite

Click **Next** to proceed to the RTL Common Signal page, as shown below.

Figure 52: **RTL Common Signals**

- **module_clock:** Specify the name the of the clock used in the RTL IP.
- **module_reset:** Specify the name of the reset signal used in the IP.
- **module_clock_enable:** Specify the name of the clock enable signal in the IP.
- **ap_ctrl_chain_protocol_start:** Specify the name of the block control start signal used in the IP.
- **ap_ctrl_chain_protocol_ready:** Specify the name of the block control ready signal used in the IP.
- **ap_ctrl_chain_protocol_done:** Specify the name of the block control done signal used in the IP.
- **ap_ctrl_chain_protocol_continue:** Specify the name of the block control continue signal used in the RTL IP.

Click **Finish** to automatically generate a JSON file for the specified IP. This can be confirmed through the log message as shown below.

Log Message:

```
"[2019-08-29 16:51:10] RTL Blackbox Wizard Information: the "foo.json" file
has been created in the rtl_blackbox/Source folder."
```

The JSON file can be accessed through the Source file folder, and will be generated as described in the next section.

Running C Synthesis

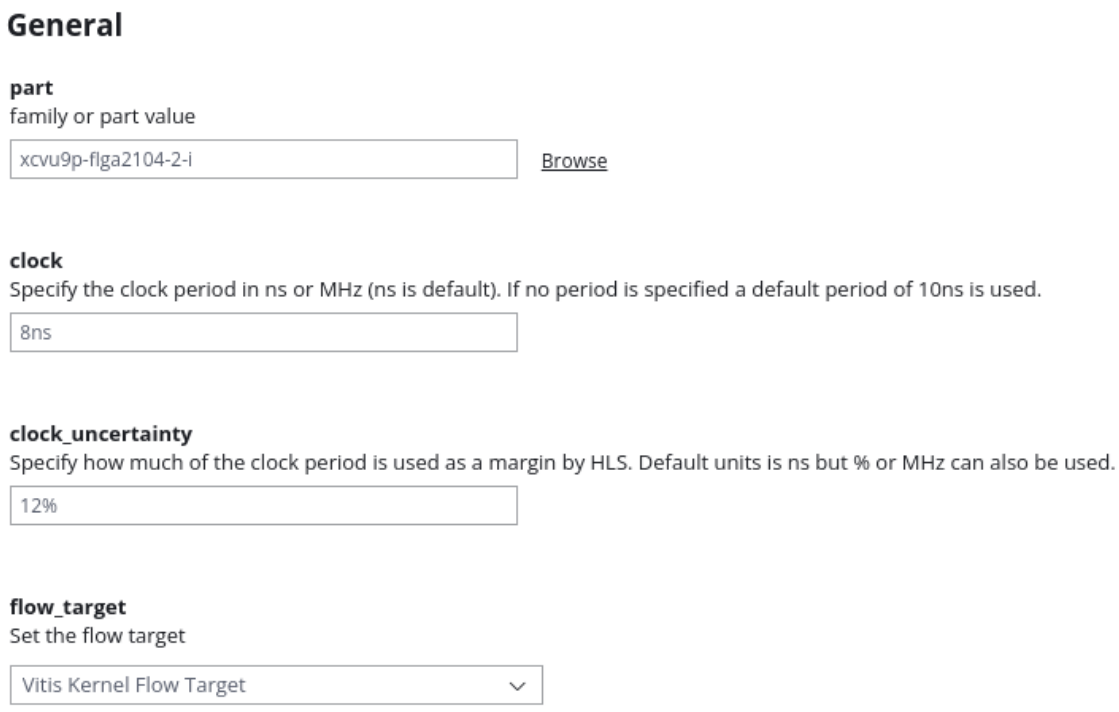
Make sure the HLS component is active in the Flow Navigator, or select it from the Component menu to make it the active component in the tool. When the HLS component is the active component, the Flow Navigator enables running C Simulation, C Synthesis, C/RTL Co-simulation, Packaging, and Implementation to build and analyze the HLS component. To synthesize the HLS component select **Run** beneath the C SYNTHESIS heading in the Flow Navigator. Running C synthesis on the HLS component generates the RTL from the C/C++ source code.

TIP: *Running synthesis in the Vitis Unified IDE uses the* `v++ -c --mode hls` *command.*

Specifying the Flow Target and General Settings

The HLS component can be used to create synthesis results for the Vivado IP flow, or the Vitis kernel flow. The flow target determines the default interface ports applied to the IP or kernel and other details of the synthesized design.

Figure 53: **HLS Component General Settings**



In the preceding figure the flow_target is currently set for the Vitis Kernel flow. The default flow target is the Vivado IP flow. This target is generally set when the HLS component is created, but can be modified as needed.

The part or platform can also be specified under the General settings. These options determine the physical device that the C/C++ code will be synthesized for. The selected device can feature hardware resources that expand or restrict the implementation choices of the RTL synthesis.

Finally, the clock period and clock_uncertainty can be specified to define the timing details of the RTL design. These can affect the timing and scheduling of operations in the design.

These selections result in the following config file entries:

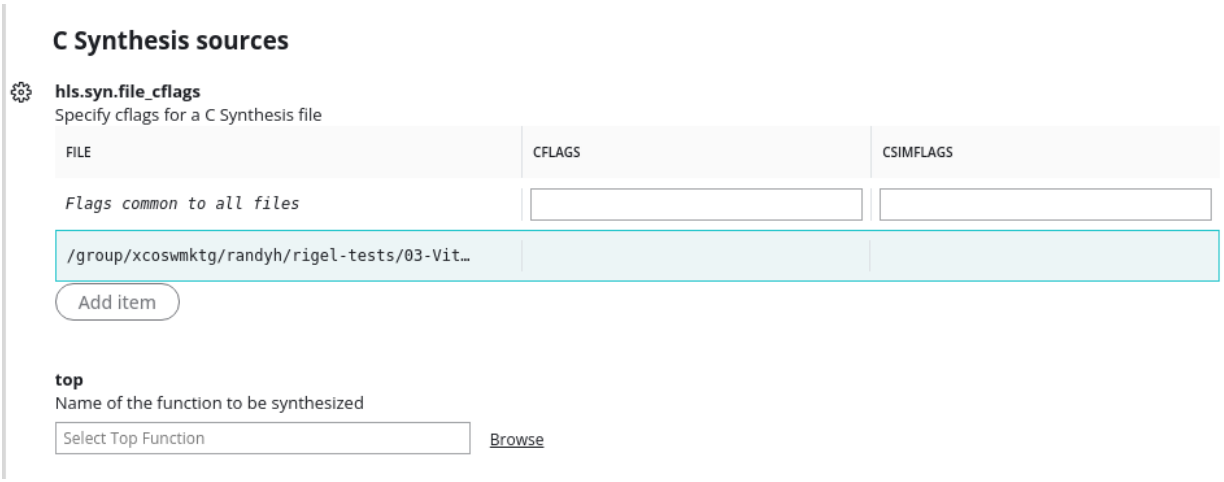
```
part=xcvc1902-vsva2197-2MP-e-S

[hls]
flow_target=vitis
clock=8ns
clock_uncertainty=15%
```

Loading Source Files and Identifying the Top Function

Generally the source code for the HLS component is defined or loaded at the time of creation. However, you can bypass that step when creating the component, and then you will need to add the source files prior to synthesizing the design. To add source files to an existing HLS component open the component config file and navigate to the C Synthesis sources section as shown below. This section of the Config Editor lets you add one or more source files, and specify the top function.

Figure 54: **HLS Component Synthesis Sources**



The C Synthesis sources displays currently added source files, and lets you edit, or delete current source files, add new source files, and add or modify CFLAGS and CSIMFLAGS.

- Add item: Select this command to add new source files to the HLS component. This opens a File Browser, and lets you navigate to and select source files to add.

- Add CFLAGS or CSIMFLAGS: You can add compilation flags for synthesis (CFLAGS) and for simulation (CSIMFLAGS) to be applied to all source files, or to be added to specific source files. As shown in the preceding figure, simply add the flags to the appropriate text entry box for Flags common to all files. To add flags for specific source files, select the source file and select **Edit item** to add the flags, or modify the file name and path.
- top: Lets you specify the function to use as the top-level RTL module for synthesis. The top-level module determines the RTL ports that will be added, and which sub-functions to include in the HLS component. This opens the Select Top Function dialog box that displays a list of functions defined in the source files for you to choose the top function.

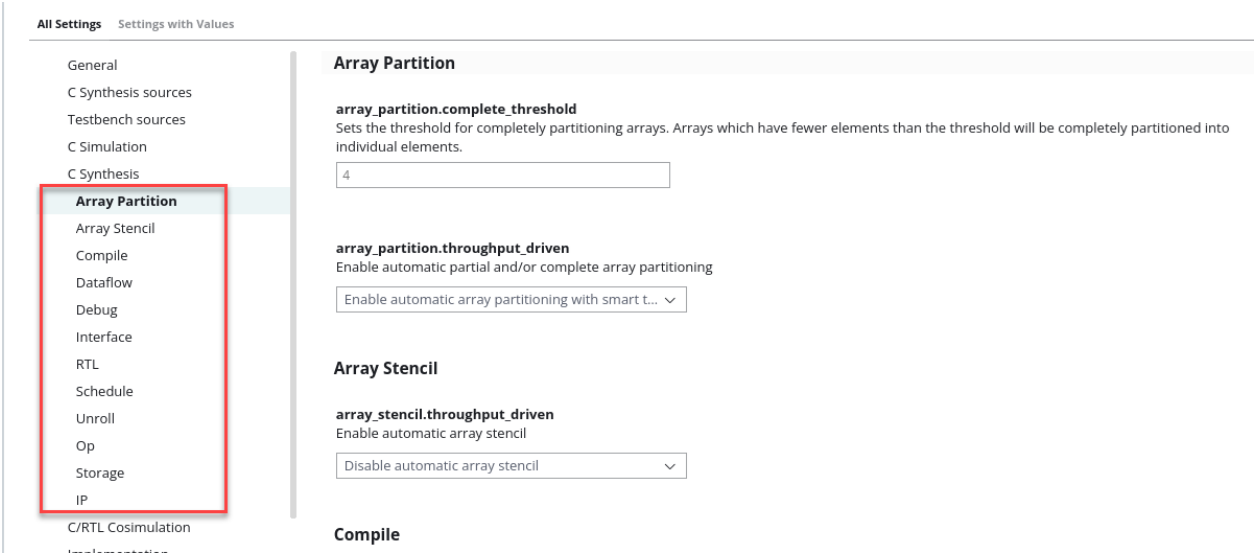
These selections result in the following config file entries:

```
[HLS]
syn.file=<path/to/file.cpp>
syn.file_cflags=<path/to/file.cpp>,<cflag>
syn.file_csimflags=<path/to/file.cpp>,<csimflag>
syn.cflags=<cflag for all files>
syn.csimflags=<csimflag for all files>
syn.top=<top function name>
```

Configuring Default Settings

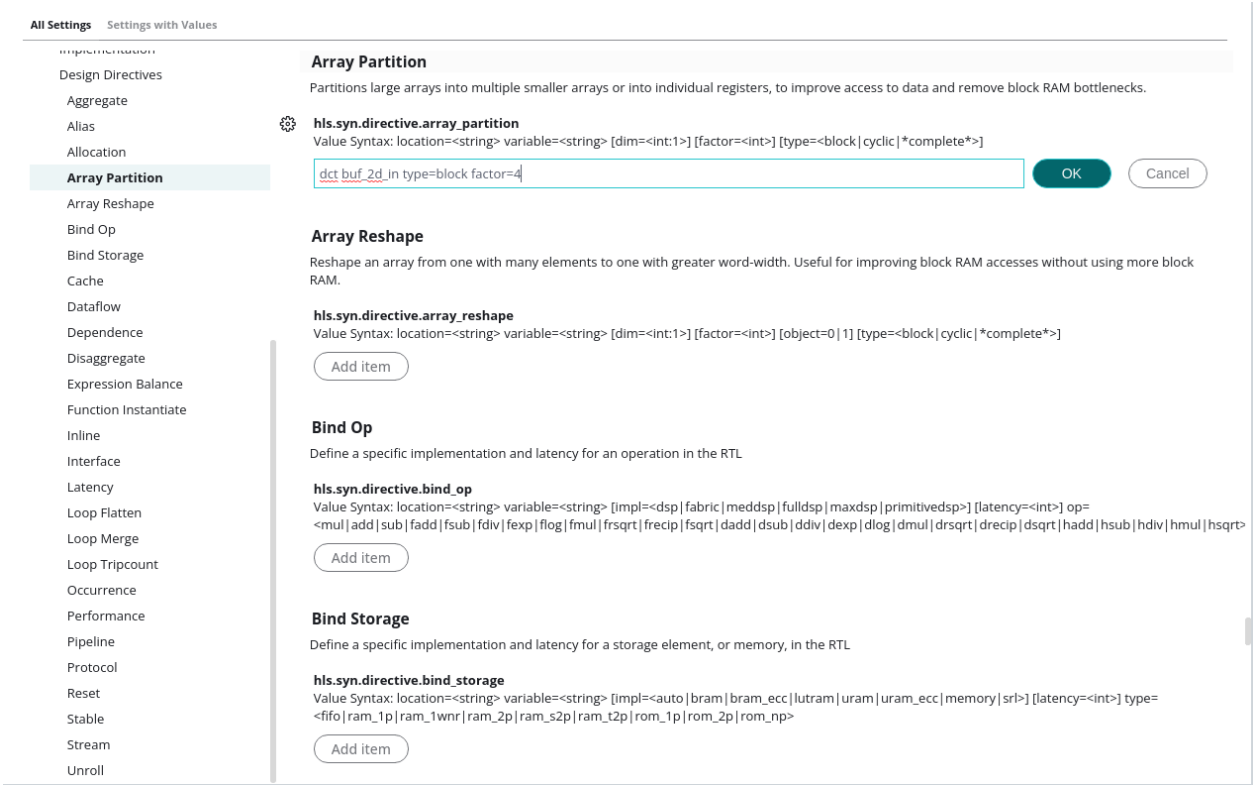
For the HLS component, the Vitis Unified IDE offers a variety of commands that can be used to configure the default settings of the tool for synthesis and simulation. These configuration commands are presented in Config Editor under the heading of C Synthesis as shown in the figure below. Generally these options are documented as presented in the figure, such as Compile Options, Interface Configuration, and RTL Configuration.

Figure 55: **HLS Component Default Settings**



Assigning Design Directives

Figure 56: **HLS Component Directives**



Design directives let you to customize the synthesis results for the source code. Change the directives across multiple synthesis runs to change the results or optimize your design. They can be added as HLS pragmas directly to the source code, or as HLS Optimization Directives to the configuration file for use in an HLS component. In the Config File editor you will see the various directives listed, and you can select **Add Item** to open the Directive editor.

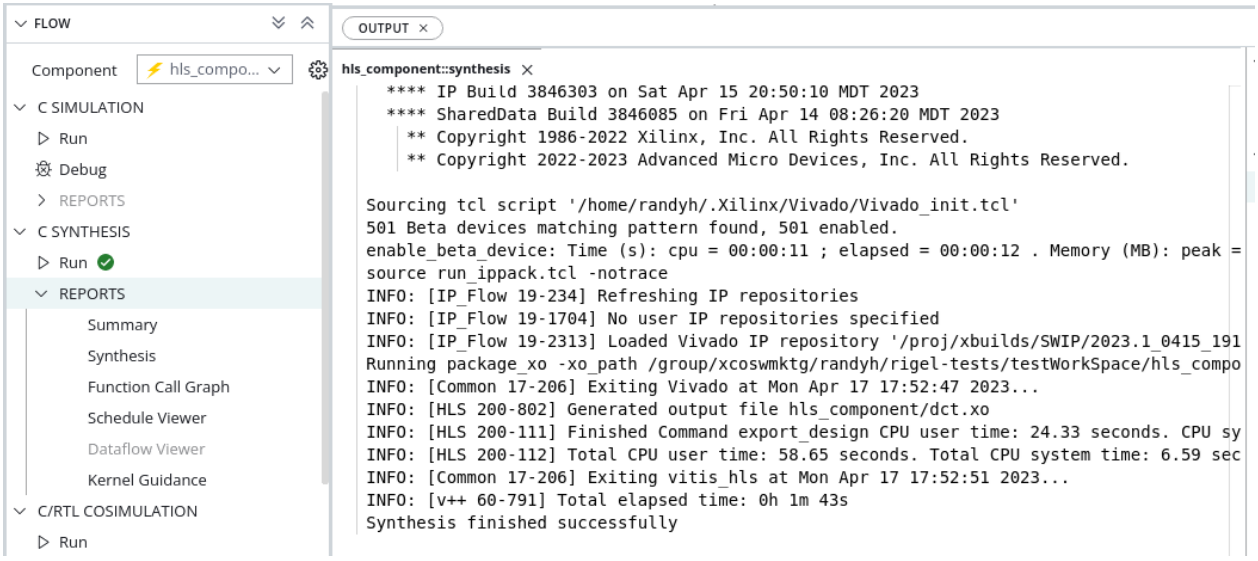
Some example design directive entries in the config file:

```
[HLS]
syn.directive.dataflow=dct
syn.directive.array_partition=dct buf_2d_in type=block factor=4
syn.directive.pipeline=dct2d II=4
```

Run Synthesis

With the key elements of the HLS component defined in the config file you are ready to run Synthesis. Select **Run** from the Flow Navigator to begin Synthesis. You can track the progress of the synthesis run in the Output window. The transcript for the synthesis run will have the top function name as `<component-name>::synthesis`. During the synthesis process messages are transcripted to the console window, and to the `vitis_hls.log` file.

Figure 57: **HLS Component Running Synthesis**



After synthesis is complete, you should see the Synthesis finished successfully message at the end of the transcript. You will also see the Reports folder under the Run command populated as shown in the preceding figure. The reports available after synthesis are as follows:

- Summary: reports the command line used and the time stamp on the results.
- Synthesis: reports the synthesis results with information on quality of results, HW interfaces, burst transactions and more. Refer to Synthesis Summary for more information.
- Function Call Graph: Displays the HLS component after C Synthesis or C/RTL Co-simulation to show the throughput of the design in terms of latency and II as described in Function Call Graph.
- Schedule Viewer: Shows each operation and control step of the function, and the clock cycle that it executes in, as described in Schedule Viewer.
- Dataflow Viewer: Only available when the DATAFLOW pragma or directive is used in the design. This report shows the dataflow structure inferred by the tool as described in Dataflow Viewer.
- Kernel Guidance: reports Guidance messages to provide design advice for the source code and synthesis results.

Output of C Synthesis

In the Vitis Unified IDE the hierarchy of the HLS component is structured as follows: `<workspace>/<component>/<component>/hls/`. You can select the HLS component in the Vitis Component Explorer and select **Open In Terminal** from the right-click menu to explore the contents of the component folders.

When synthesis completes, the `syn` folder is created inside the `<component>/hls` folder. This folder contains the following elements:

- The `verilog` and `vhdl` folders contain the output RTL files.
  - The top-level file has the same name as the top-level function for synthesis.
  - There is one RTL file created for each sub-function that has not been inlined into a higher level function.
  - There could be additional RTL files to implement sub-blocks of the RTL hierarchy, such as block RAM, and pipelined multipliers.
- The `report` folder contains a report file for the top-level function and one for every sub-function that has not been in-lined into a higher level function by the HLS compiler. The report for the top-level function provides details on the entire design.

IMPORTANT! *Do not directly use the RTL files generated in the `syn/verilog` or `syn/vhdl` folder for synthesis in the Vivado tool. Instead you must use the exported output files generated by the package process as described in Packaging the RTL Design. The HLS compiler sometimes uses IP from the Vivado IP catalog in the synthesized RTL code, such as with floating point designs, and the `verilog` and `vhdl` folders will not contain the complete design.*

Synthesis Summary

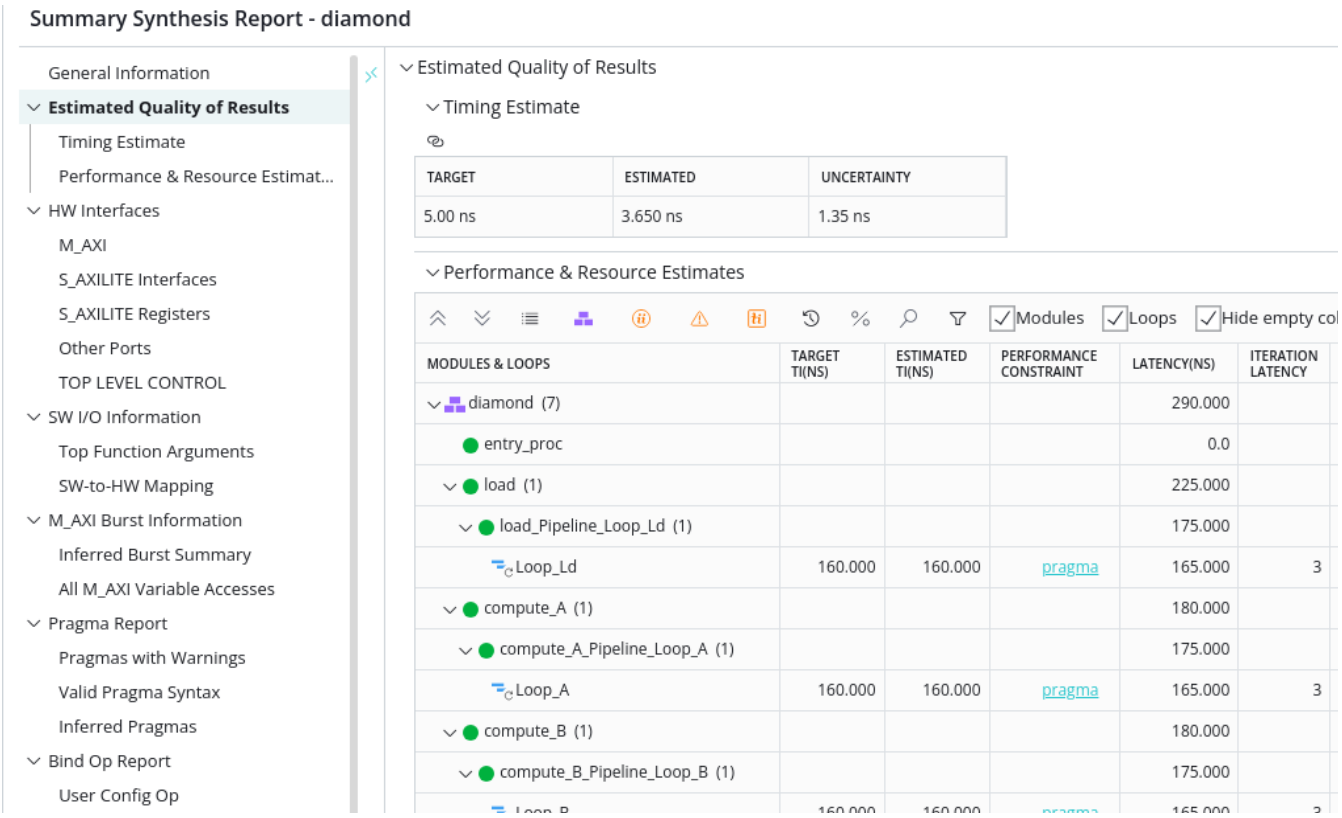
When synthesis completes, Vitis HLS generates a Synthesis Summary report for the top-level function that opens automatically in the information pane.

The specific sections of the Synthesis Summary are detailed below.

General Information

Provides information on when the report was generated, the version of the software used, the project name, the solution name and target flow, and the technology details.

Figure 58: **Synthesis Summary Report**



Timing Estimate

Displays a quick estimate of the timing specified by the solution, as explained in Specifying the Clock Frequency. This includes the Target clock period specified, and the period of Uncertainty. The clock period minus the uncertainty results in the Estimated clock period.

TIP: *These values are only estimates of the HLS tool. More accurate timing estimates can be provided by Vivado synthesis and implementation as explained in Running Implementation.*

Performance & Resource Estimates

The Performance Estimate columns report the timing slack, latency, and initiation interval for the top-level function and any sub-blocks instantiated in the top-level. Each sub-function called at this level in the C/C++ source is an instance in the generated RTL block, unless the sub-function was in-lined into the top-level function using the INLINE pragma or directive, or automatically in-lined.

The Slack column displays any timing issues in the implementation.

The two Latency columns display the number of cycles it takes to produce the output, and display latency in time (ns).

The Iteration Latency is the latency of a single iteration for a loop.

The Initiation Interval is the number of clock cycles before new inputs can be applied. In the absence of any PIPELINE directives, the latency is one cycle less than the initiation interval (the next input is read after the final output is written).

TIP: *When latency is displayed as a "?" it means that the tool cannot determine the number of loop iterations. If the latency or throughput of the design is dependent on a loop with a variable index, the HLS compiler reports the latency of the loop as being unknown. In this case, use the LOOP_TRIPCOUNT pragma or directive to manually specify the number of loop iterations. The LOOP_TRIPCOUNT value is only used to ensure the generated reports show meaningful ranges for latency and interval and does not impact the results of synthesis.*

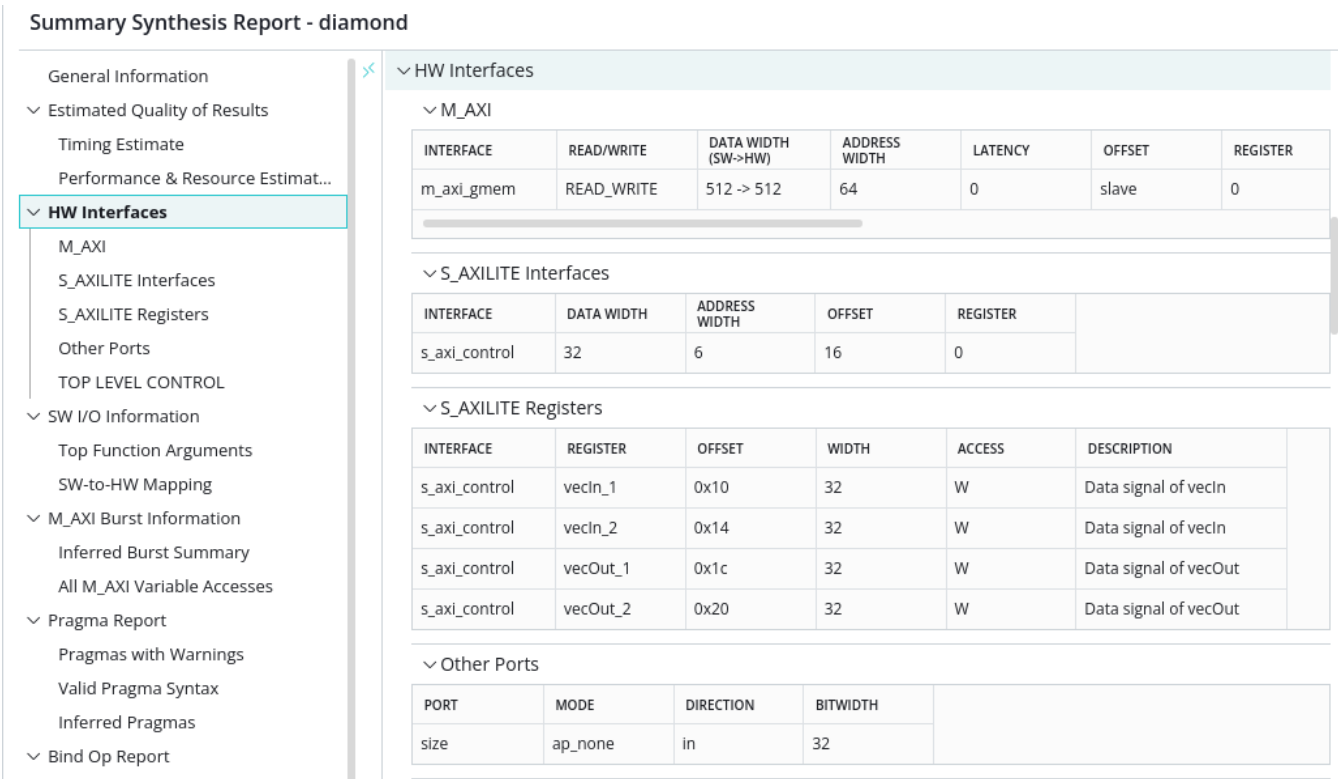
The Trip Count column displays the number of iterations a specific loop makes in the implemented hardware. This reflects any unrolling of the loop in hardware.

The resource estimate columns of the report indicates the estimated resources needed to implement the software function in the RTL code. Estimates of the block RAM, DSP, FFs, and LUTs are provided.

HW Interfaces

The HW Interfaces section of the synthesis report provides tables for the different hardware interfaces generated during synthesis. The type of hardware interfaces generated by the tool depends on the flow target specified by the solution, as well as any INTERFACE pragmas or directives applied to the code. In the following image, the solution targets the Vitis Kernel flow, and therefore generates AXI interfaces as required.

Figure 59: **HW Interfaces**



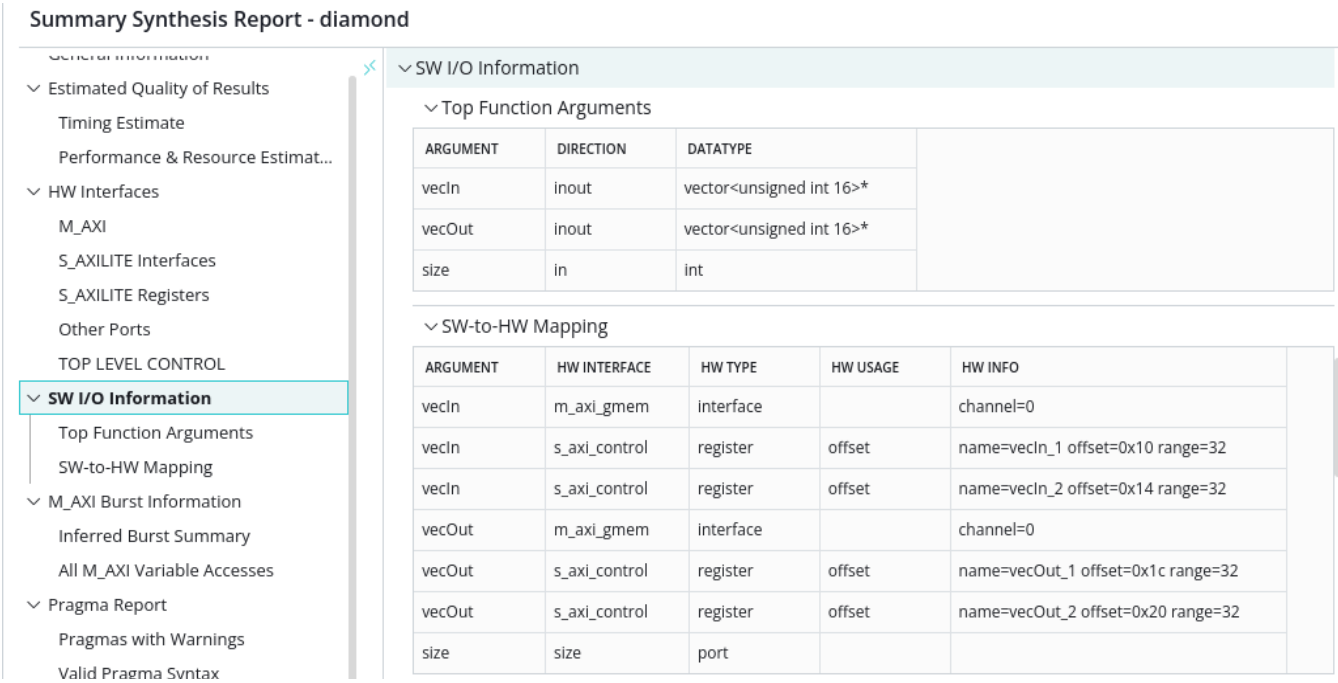
The following should be observed when reviewing these tables:

- Separate tables are provided for the different interfaces.
- Columns are provided to display different properties of the interface. For the M_AXI interface, these include the Data Width and Max Widen Bitwidth columns which indicate whether Automatic Port Width Resizing has occurred, and to what extent. In the example above, you can see that the port was widened to 512 bits from the 16 bits specified in the software. For more information, see Automatic Port Width Resizing.
- The Latency column displays the latency of the interface:
  - In an `ap_memory` interface, the column displays the read latency of the RAM resource driving the interface.
  - For an `m_axi` interface, the column displays the expected latency of the AXI4 interface, allowing the design to initiate a bus request a number of cycles (latency) before the read or write is expected.
- The Bundle column displays any specified bundle names from the INTERFACE pragma or directive.
- Additional columns display burst and read and write properties of the M_AXI interface as described in the INTERFACE pragma or directive.
- The Bit Fields column displays the bits used by an the registers in an `s_axilite` interface.

SW I/O Information

Highlights how the function arguments from the C/C++ source is associated with the port names in the generated RTL code. Additional details of the software and hardware ports are provided as shown below. The SW argument is expanded into multiple HW interfaces. For example, the `input` argument is related to three HW interfaces, the `m_axi` for data, and the `s_axi_lite` for required control signals.

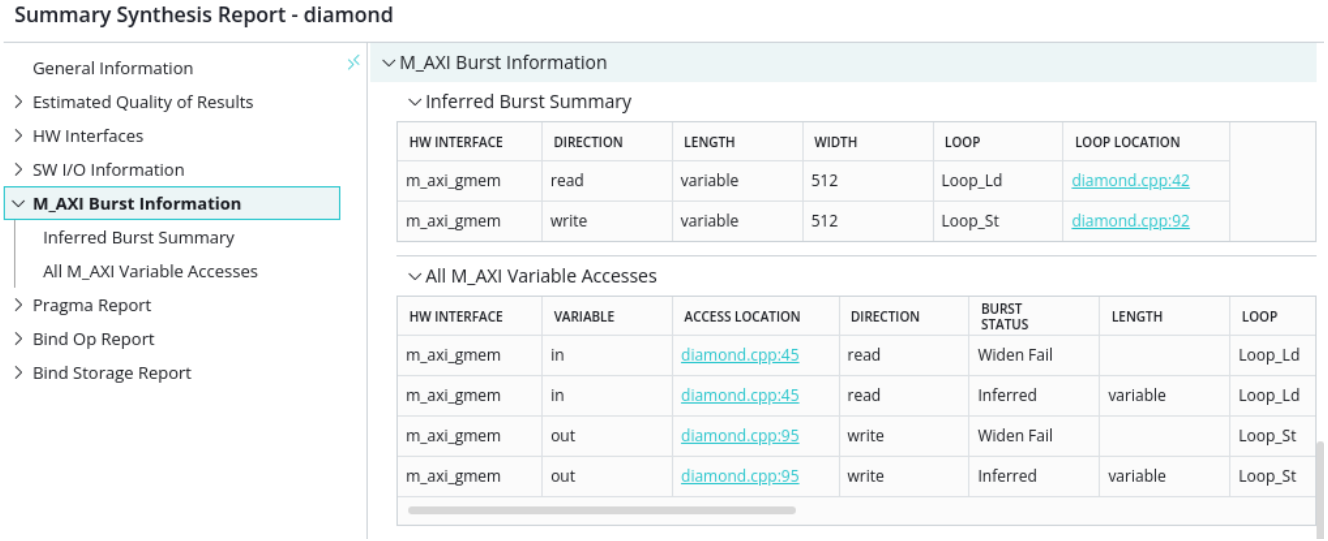
Figure 60: **SW I/O Information**



M_AXI Burst Information

In the M_AXI Burst Information section the Burst Summary table reports the successful burst transfers, with a link to the associated source code. The reported burst length refers to either `max_read_burst_length` or `max_write_burst_length` and represents the number of data values read/written during a burst transfer. For example, in a case where the input type is integer (32 bits), and HLS auto-widens the interface to 512 bits, each burst transfers 1024 integers. Because the widened interface can carry 16 integers at a time, the result is 64 beat bursts. The Burst Missed table reports why a particular burst transfer was missed with a link to Guidance messages related to the burst failures to help with resolution.

Figure 61: **M_AXI Burst Information**

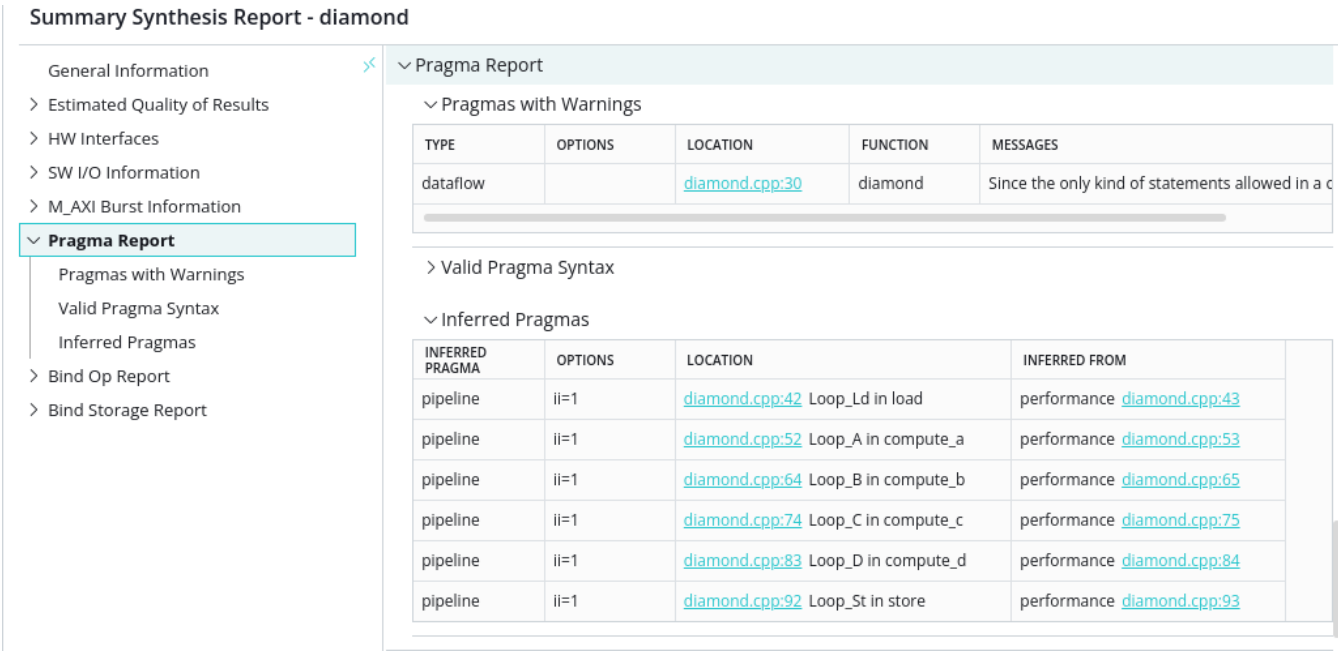


Pragma Report

Displays the pragmas in the design: valid, ignored, inferred. This report is intended to summarize issues that can otherwise be found in the log files. It lets you quickly identify issues with the pragmas used in your design, to see which ones might not have been used as expected. Valid pragmas are separately reported so you can see all pragmas in use in the design.

TIP: *A link is provided to the location of the pragma in the source code. Click the link to view the source code.*

Figure 62: **Pragma Report**



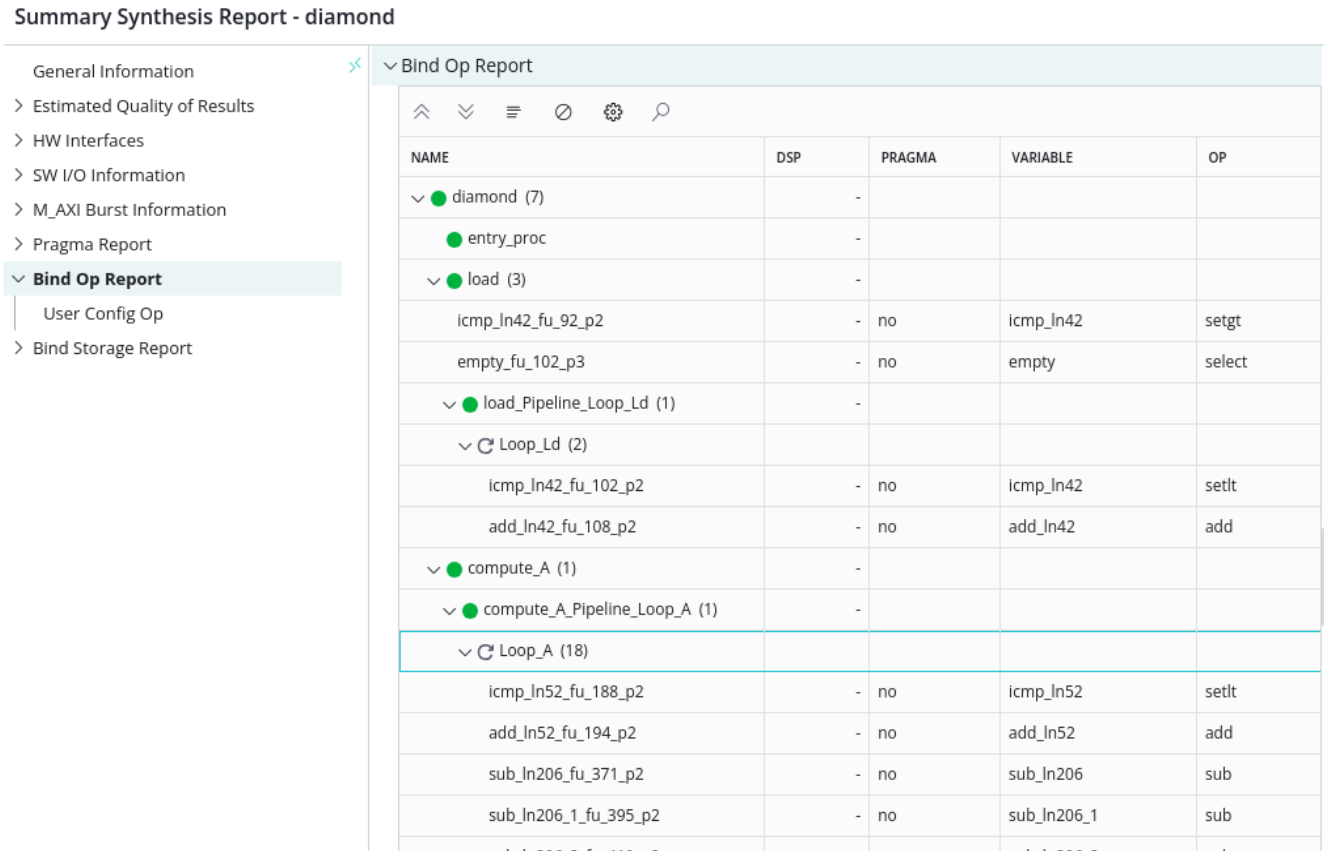
Bind Op and Bind Storage Reports

The Bind Op and Bind Storage reports are added to the Synthesis Summary report. Both reports can help you understand choices made by the HLS compiler when it maps operations to resources. The tool will map operations to the right resources with the right latency. You can influence this process by using the BIND_OP pragma or directive, and requesting a particular resource mapping and latency. The Bind Op report will show which of the mappings were automatically done versus those enforced by the use of a pragma. Similarly, the Bind Storage report shows the mappings of arrays to memory resources on the platform like block RAM/ LUTRAM/URAM.

The Bind Op report displays the implementation details of the kernel or IP. The hierarchy of the top-level function is displayed and variables are listed with any HLS pragmas or directives applied, the operation defined, the implementation used by the HLS tool, and any applied latency.

This report is useful for examining the programmable logic implementation details specified by the RTL design.

Figure 63: **Synthesis Summary**



As shown above, the Bind Op report highlights certain important characteristics in your design. Currently, it calls out the number of DSPs used in the design and shows in a hierarchy where these DSPs are used in the design. The table also highlights whether the particular resource allocation was done because of a user-specified pragma and if so, a "yes" entry will be present in the Pragma column. If no entry exists in the Pragma column, it means that the resource was auto inferred by the tool. The table also shows the RTL names of the resources allocated for each module in the user's design and you can hierarchy descend down the hierarchy to see the various resources.

It does not show all the inferred resources but instead shows resources of interest such as arithmetic, floating-point, and DSPs. The particular implementation choice of fabric (implemented using LUTs) or DSP is also shown. Finally, the latency of the resource is also shown. This is helpful in understand and increasing the latency of resources if needed to add pipeline stages to the design. This is extremely useful when attempting to break a long combinational path when trying to solve timing issues during implementation.

Each resource allocation is correlated to the source code line where the corresponding op was inferred from and the user can right-click on the resource and select the "Goto Source" option to see this correlation. Finally, the second table below the Bind Op report illustrates any global config settings that can also alter the resource allocation algorithm used by the tool. In the above example, the implementation choice for a `dadd` (double precision floating point addition) operation has been fixed to a `fulldsp` implementation. Similarly, the latency of a `ddiv` operation has been fixed to 2.

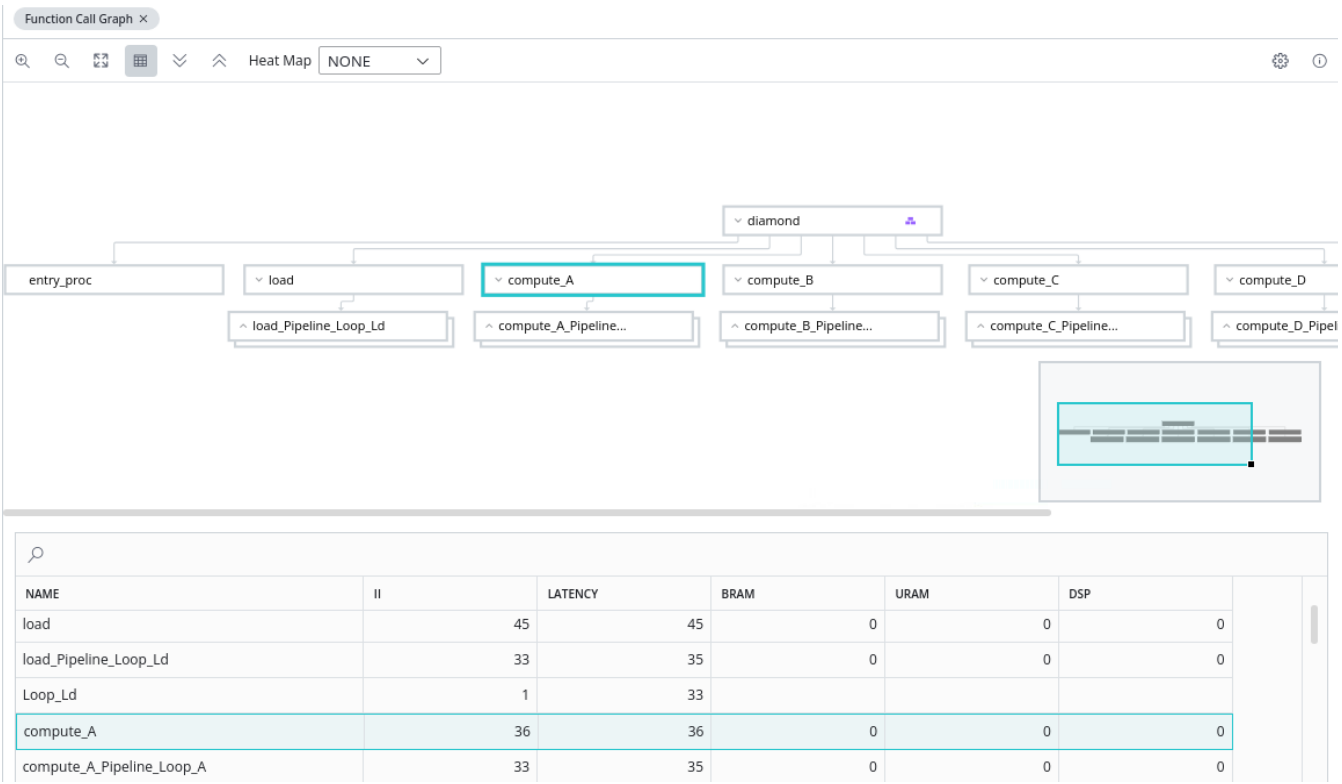
Similar to the BIND_OP pragma, the BIND_STORAGE pragma can be used to select a particular memory type (such as single port or dual port) and/or a particular memory implementation (such as BRAM/LUTRAM/URAM/SRL, etc.) and a latency value. The Bind Storage report highlights the storage mappings used in the design. Currently, it calls out the number of Block RAMs and URAMs used in the design. The table also highlights whether the particular storage resource allocation was done because of a user-specified pragma and if so, a "yes" entry will be present in the Pragma column. If no entry exists in the Pragma column, then this means that the storage resource was auto inferred by the tool. The particular storage type, as well as the implementation choice, are also shown along with the variable name and latency.

Using this information, you can review the storage resource allocation in the design and make design choices by altering the eventual storage implementation depending upon availability. Finally, a second table in the Bind Storage report will be shown if there are any global settings as described in Storage Configuration that can also alter the storage resource allocation algorithm used by the tool.

Function Call Graph

The Function Call Graph illustrates your full design after C Synthesis or C/RTL Co-simulation. The report shows the throughput of the design in terms of latency and II. It helps identify the critical path in your design and helps you identify bottlenecks in the design to improve throughput. It can also show the paths through the design where throughput might be imbalanced leading to FIFO stalls and/or deadlock.

Figure 64: **Performance Metrics Synthesis**



In some cases, the displayed hierarchy of the design might not be the same as your source code as a result of HLS optimizations that convert loops into function pipelines, etc. Functions that are in-lined will no longer be visible in the call graph, as they are no longer separate functions in the synthesized code. If multiple instances of a function are created, each unique instance of the function is shown in the call graph. This lets you see what functions contribute to a calling function's latency and II.

The graph as shown above displays functions as rectangular boxes, and loops as oval boxes. The graph displays hierarchy and nested functions or loops. A table under the graph displays each function or loop with II, latency, and resource or timing data depending on the specific view.

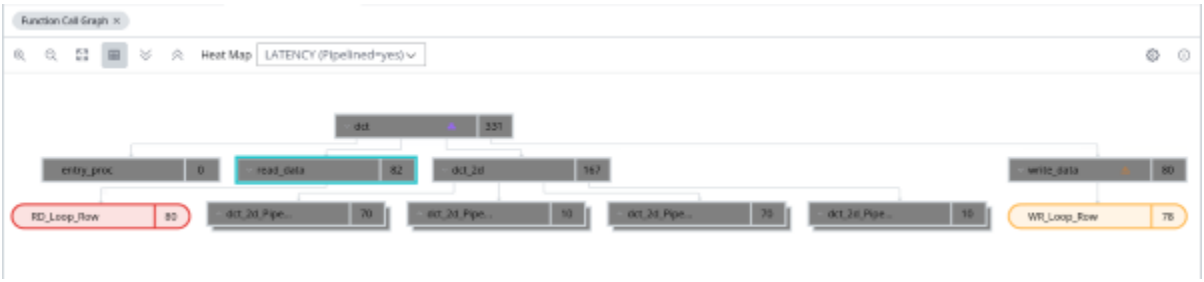
The Function Call Graph report is available under both C SYNTHESIS and C/RTL COSIMULATION headings in the Flow Navigator. The performance and resource metrics that are shown in the graph from the C SYNTHESIS phase are estimates from the HLS tool. The Function Call Graph report available after C/RTL COSIMULATION is more detailed, providing actual II and latency numbers reported along with stalling percentages.

TIP: *For more accurate resource and timing estimates, logic synthesis or implementation can be performed as part of Running Implementation.*

You can also use the **Heat Map** feature to highlight several metrics of interest:

- II (min, max, avg)
- Latency (min, max, avg)
- Resource Utilization
- Stalling Time Percentage

Figure 65: **Performance Metrics**



The heat map uses color coding to highlight problematic modules. Using a color scale of red to green where red indicates the high value of the metric (highest II or highest latency) while green indicates a low value of the metric in question. The colors that are neither red nor green represent the range of values that are in between the highest and lowest values. As shown above, this helps in quickly identifying the modules that need attention. In the example shown above, a heat map for LATENCY is shown and the red module indicates where high latency is observed.

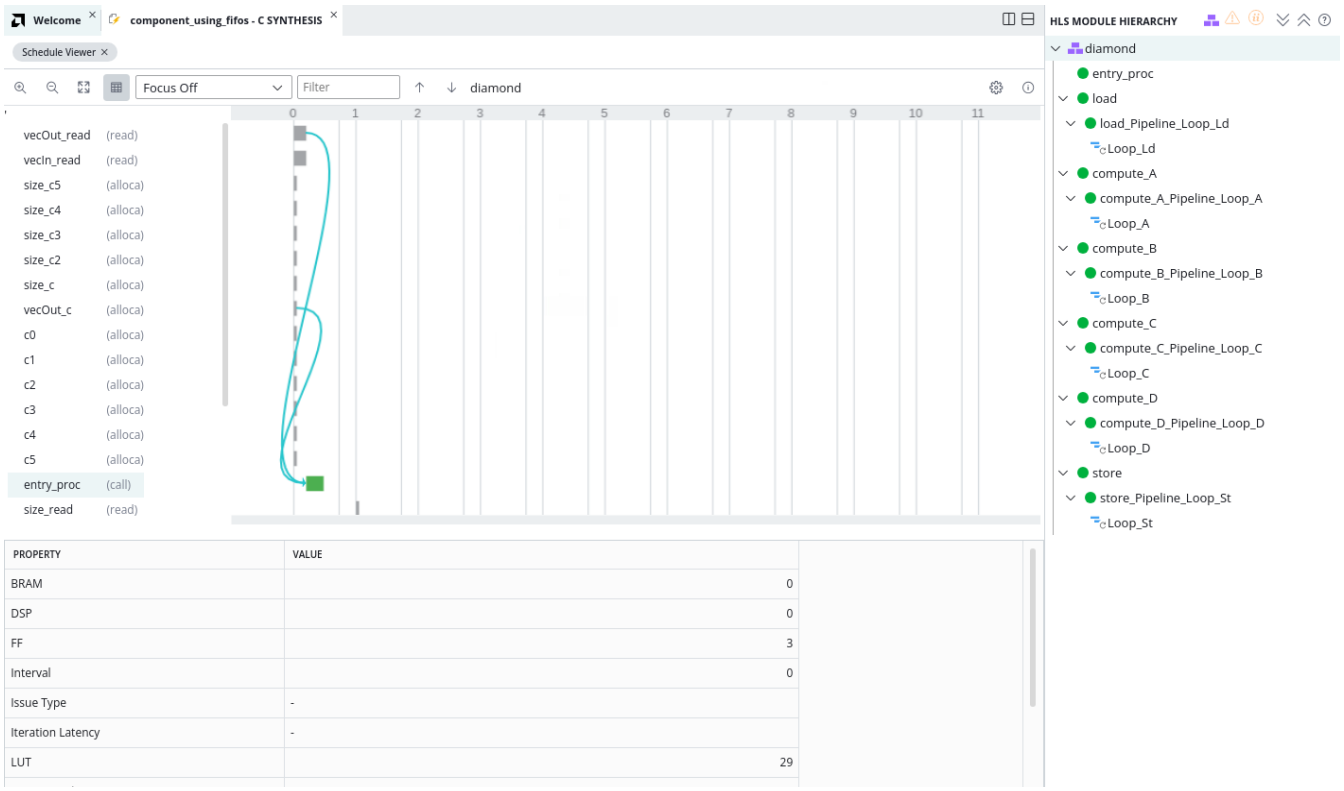
As mentioned before, the Function Call Graph illustrates at a high level, the throughput numbers of your design. The user can view the Function Call Graph as a cockpit from which further investigations can be carried out. Right-click on any of the displayed modules to display a menu of options that you can use to display additional information. This lets you see the overall design and then jump into specific parts of the design which need extra attention. Additional reports include the Schedule Viewer, Synthesis Summary report, Dataflow Viewer, and source files. The Function Call Graph lets you see the full picture of your design and have the latency and II information of each module available for analysis.

TIP: *On the right hand side of the report there is a Settings command and a Legend command to let you display additional details of the report.*

Schedule Viewer

The Schedule Viewer provides a detailed view of the synthesized RTL, showing each operation and control step of the function, and the clock cycle that it executes in. It helps you to identify any loop dependencies that are preventing parallelism, timing violations, and data dependencies.

Figure 66: **Schedule Viewer**



- The left vertical axis shows the names of operations and loops that will get implemented as logic in the RTL hierarchy. Operations are in topological order indicating that an operation on one line can only be driven by operations from a previous line, and will only drive an operation in a later line. Depending upon the type of violations found the Schedule Viewer shows additional information for each operation

  TIP: *Each operation is associated with lines of source code. Right-click the operation to use the* ***Goto Source*** *command to open the input source code associated with the operation.*

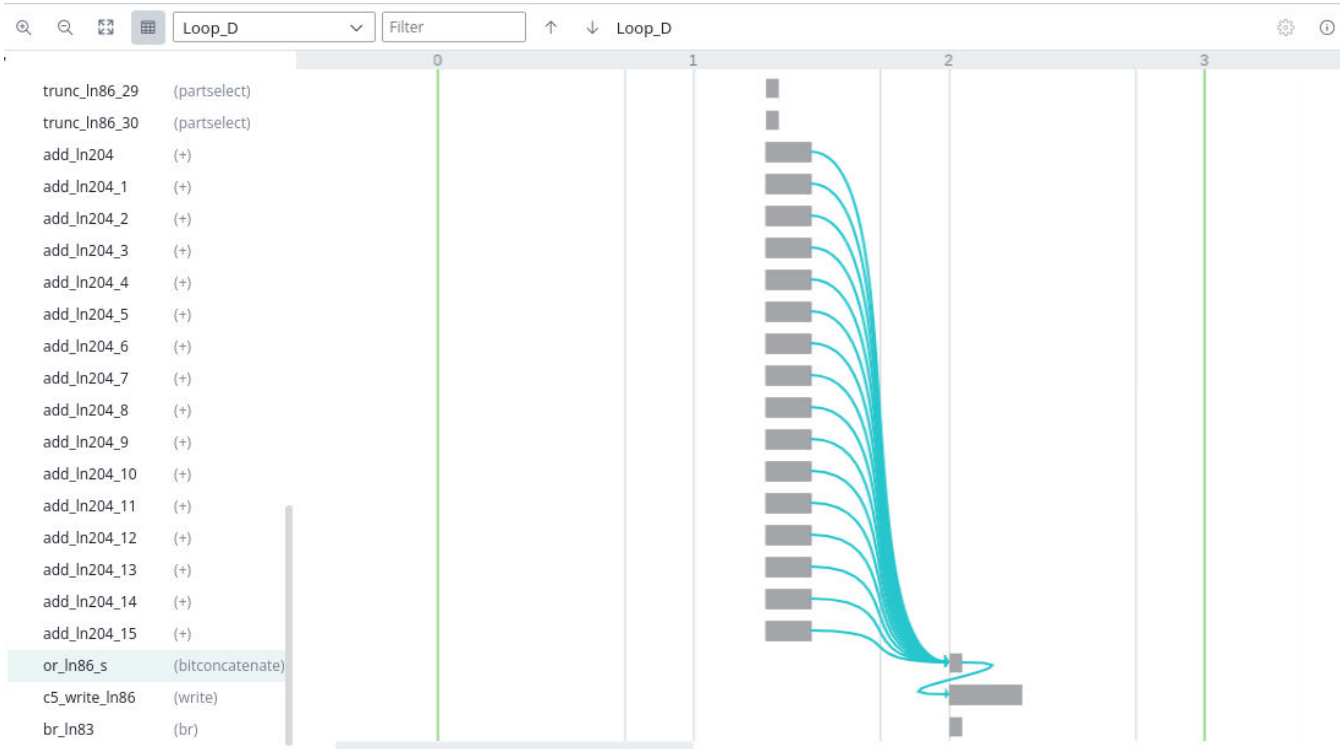
- The top horizontal axis shows the clock cycles in consecutive order.
- The vertical dashed line in each clock cycle shows the reserved portion of the clock period due to clock uncertainty. This time is left by the tool for the Vivado back-end processes, like place and route.
- Each operation is shown as a gray box in the table. The box is horizontally sized according to the delay of the operation as percentage of the total clock cycle. In case of function calls, the provided cycle information is equivalent to the operation latency. Multi-cycle operations are shown as gray boxes with a horizontal line through the center of the box.
- The Schedule Viewer also displays general operator data dependencies as solid blue lines. As shown in the preceding figure, when selecting an operation you can see solid blue arrows highlighting the specific operator dependencies. This gives you the ability to perform detailed analysis of data dependencies. A green dotted line indicates an inter-iteration data dependency. Memory dependencies are displayed using golden lines.

> **TIP:** *By default all dependencies (blue lines) are shown between each operation in the critical timing path.*

- On the far right the HLS Module Hierachy is displayed alongside the Schedule Viewer to let you quickly navigate through the hierarchy of the design.

In the following figure the loop called `COL_DCT_LOOP_DCT_OUTER_LOOP` is selected. This is a pipelined loop and the initiation interval (II) is explicitly stated in the loop bar. Any pipelined loop is visualized unfolded, meaning one full iteration is shown in the schedule viewer. Overlap, as defined by II, is marked by a thick clock boundary on the loop marker. The total latency of a single iteration is equivalent to the number of cycles covered by the loop marker. In this case, it is six clock cycles.

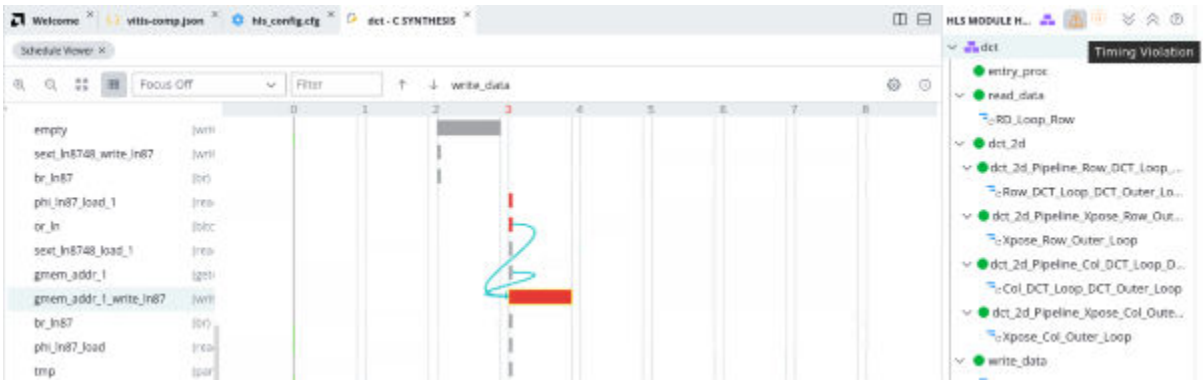
Figure 67: **Pipelined Loop**



The Schedule Viewer displays a menu bar at the top of the report that includes the following features:

- Zoom In/Zoom Out/Zoom Fit commands
- A text field to enter search terms for specific operations or steps, and arrow commands (Previous Match or Next Match) to scroll up or down through the list of objects that match your search text
- Legend command for the display.

Figure 68: **Timing Violations**



You can quickly locate timing violations and II violations using the toolbar menu in the in the Module Hierarchy view as shown in the preceding figure. To locate the operations causing the violation in the source code, right-click the operation and use the **Goto Source** command. A timing violation is a path of operations requiring more time than the available clock cycle. To visualize this, the problematic operation is represented in the Schedule Viewer in a red box.

Properties View

At the bottom of the Schedule Viewer, as shown in the top figure, is the Properties view that displays the properties of a currently selected object in the Schedule Viewer. This lets you see details of the specific function, loop, or operation that is selected in the Schedule Viewer. The types of elements that can be selected, and the properties displayed include:

- Functions or Loops
  - **Initiation Interval (II):** The number of clock cycles before the function or loop can accept new input data.
  - **Loop Iteration Latency:** The number of clock cycles it takes to complete one iteration of the loop.
  - **Latency:** The number of clock cycles (and time) required for the function to compute all output values, or for the loop to complete all iterations.
  - **Name:** Name of function or loop.
  - **Pipelined:** Indicates that the function or loop are pipelined in the RTL design.
  - **Slack:** The timing slack for the function or loop.
  - **Tripcount:** The number of iterations a loop completes.
  - **Resource Utilization:** Displays the number of BRAM, DSP, LUT, or FF used to implement the function or loop.
- Operation and Storage Mapping
  - **Bitwidth:** Bitwidth of the Operation.

- **Impl:** Defines the implementation used for the specified operation or storage.
- **Name:** Name of operation.
- **Op Latency:** Displays the default or specified latency for the binding of the operation or storage.
- **Opcode:** Operation which has been scheduled, for example, `add`, `sub`, and `mult`. For more information, refer to the BIND_OP or BIND_STORAGE pragmas or directives.
- **Schedule Delay:** Specifies the delay associated with the operation.

Dataflow Viewer

The `DATAFLOW` optimization is a dynamic optimization which can only be fully understood after RTL Co-simulation is complete. Due to this fact, after C synthesis the Dataflow viewer lets you see the dataflow structure inferred by the tool, inspect the channels (FIFO/PIPO), and examine the effect of channel depth on performance. Performance data is back-annotated to the Dataflow viewer after co-simulation.

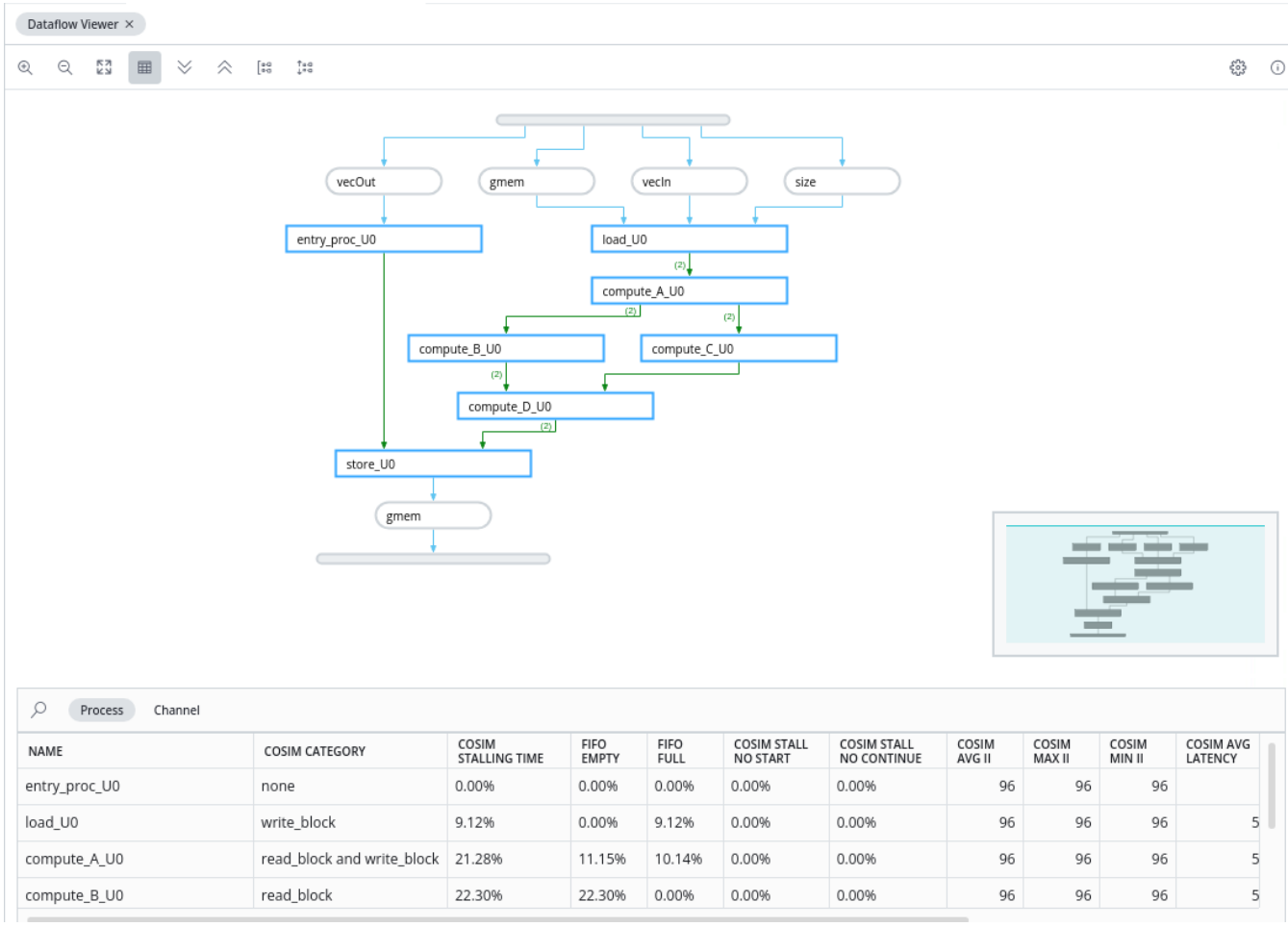
IMPORTANT! *You can open the Dataflow viewer without running RTL co-simulation, but your view will not contain important performance information such as read/write block times, co-sim depth, and stall times.*

You must apply the `DATAFLOW` pragma or directive to your design for the Dataflow Viewer report to be generated. You can apply dataflow to the top-level function, or specify regions of a function, or loops. The Dataflow viewer displays a representation of the dataflow graph structure, showing the different processes and the underlying producer-consumer connections.

The Synthesis Summary, the Module Hierarchy, and the Function Call Graph will display the icon beside the top-level function to indicate the presence of the DATAFLOW pragma.

TIP: *The diagram below is generated from the default `m_axi` interface of the Vitis Kernel flow. The use of the Vivado IP flow can result in a different dataflow diagram.*

Figure 69: **Dataflow Viewer**



Features of the Dataflow viewer include the following:

- Source Code browser
- Automatic cross-probing from process/channel to source code.
- Filtering of ports and channel types.
- Process and Channel table details the characteristics of the design:
  - Channel Profiling (FIFO sizes etc), enabled from the C/RTL Cosimulation settings in the Config File Editor.

    IMPORTANT! *You must use* `cosim.enable_dataflow_profiling=true` *in the HLS config file to capture data for the Dataflow viewer, and your test bench must run at least two iterations of the top-level function.*
  - Process Read Blocking/Write Blocking/Stalling Time reported after RTL co-simulation.
  - Process Latency and II displayed.

- Channel type and widths are displayed in the Channel table.
- Automatic cross-probing from Process and Channel table to the Graph and Source browser.
- Hover over channel or process to display tooltips with design information.

The Dataflow viewer can help with performance debugging your designs. When your design deadlocks during RTL co-simulation, the GUI will open the Dataflow viewer and highlight the channels and processes involved in the deadlock so you can determine if the cause is insufficient FIFO depth, for instance.

When your design does not perform as expected, the Process and Channels table can help you understand why. A process can stall waiting to read input, or can stall because it cannot write output. The channel table provides you with stalling percentages, as well as identifying if the process is "read blocked" or "write blocked."

The Dataflow Viewer displays a menu bar at the top of the report that includes the following features:

- Zoom In/Zoom Out/Zoom Fit commands
- Toggle Table command to show or hide the table beneath the graph
- Group All/Ungroup All groups associated channels together to simpligy the diagram
- Legend command for the display.

Running C/RTL Co-Simulation

Make sure the HLS component is active in the Flow Navigator, or select it from the Component menu to make it the active component in the tool. When the HLS component is the active component, the Flow Navigator enables running C Simulation, C Synthesis, C/RTL Co-simulation, Packaging, and Implementation to build and analyze the HLS component. For C/RTL Co-simulation of the HLS component select **Run** beneath the C/RTL COSIMULATION heading in the Flow Navigator.

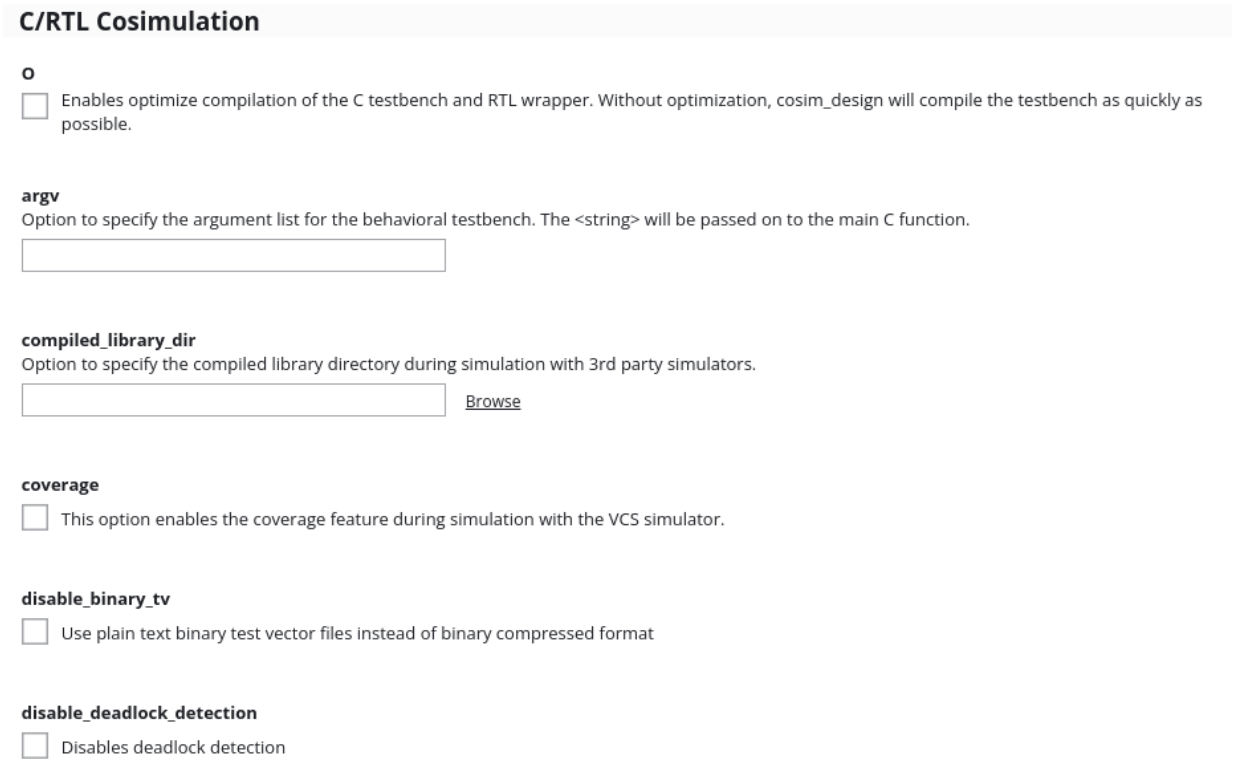
Running Co-simulation on the HLS component requires a test bench, or test bench files, which must be loaded. Prior to running co-simulation, you must also configure the HLS component to support C/RTL co-simulation. Configure the design using the following steps.

IMPORTANT! *Random input test vectors in the C/C++ test bench (for example: `std::random_device`) are not supported for co-simulation. It is recommended that random values be generated once and saved to a file which the C/C++ test bench can then use during co-simulation.*

Configure the Simulator

You can configure the simulator prior to running simulation using the C/RTL Co-Simulation section of the Config Editor, as shown below.

Figure 70: **HLS Component RTL Co-Simulation Settings**



These configuration commands let you specify how the simulation should run. These configuration commands are all documented in Co-Simulation Configuration. Refer to that content for more detailed information. Some of the settings for C/RTL Co-Simulation include the following:

- **trace_level:** Specifies the level of trace file output written to the `sim/Verilog` or `sim/VHDL` directory of the current solution when the simulation executes. Options include:
  - **all:** Output all port and signal waveform data being saved to the trace file.
  - **port:** Output waveform trace data for the top-level ports only.
  - **port_hier:** Output waveform trace data for the complete port hierarchy.
  - **none:** Do not output trace data.
- **random_stall:** Applies a randomized stall for each data transmission.
- **wave_debug:** Enables waveform visualization of all processes in the RTL simulation. This option is only supported when using Vivado logic simulator. Enabling this will launch the Simulator GUI to let you examine dataflow activity in the waveforms generated by simulation. Refer to the *Vivado Design Suite User Guide: Logic Simulation* (UG900) for more information on that tool.

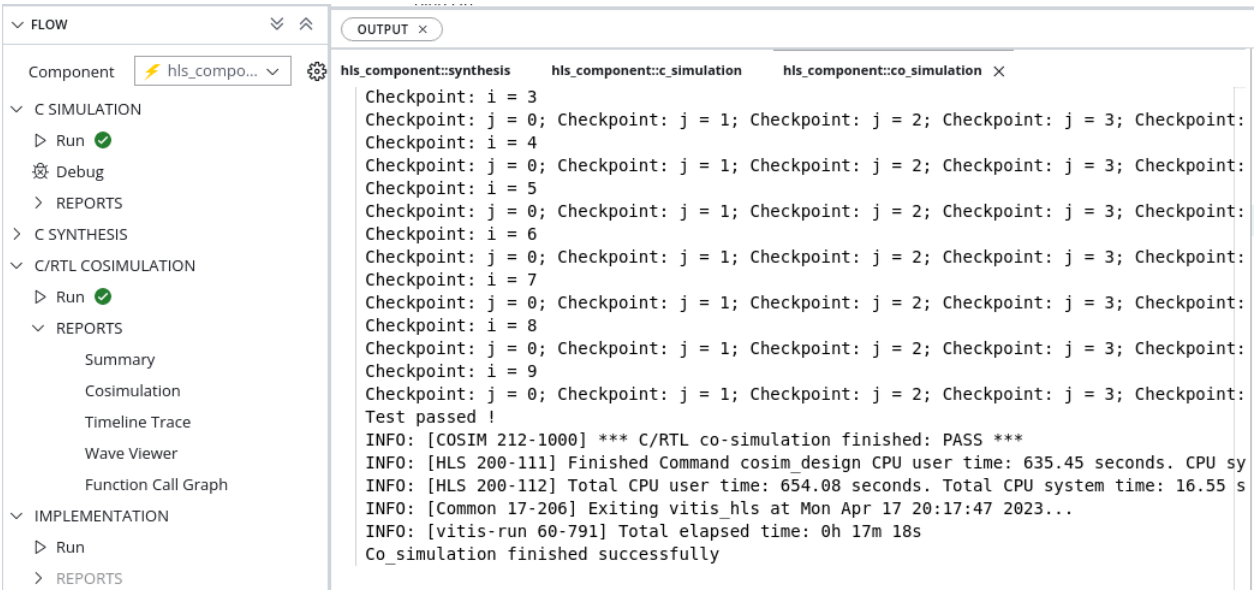
- **disable_deadlock_detection:** Disables deadlock detection, and opening the Cosim Deadlock Detection in co-simulation.
- **enable_dataflow_profiling:** Enables capturing profile data for display in the Dataflow Viewer.
- **Dynamic Deadlock Prevention:** Prevent deadlocks by enabling automatic FIFO channel size tuning for dataflow profiling during co-simulation.

Run C/RTL Co-Simulation

With the C/RTL Co-simulation setup defined in the config file you are ready to select **Run** from the Flow Navigator to begin simulation. You can track the progress of simulation in the Output window. The transcript for the synthesis run will have the top function name as `<component-name>::co_simulation` as shown below.

TIP: *The simulation run in the Vitis unified IDE uses the `vitis-run --mode hls --cosim` command.*

Figure 71: **HLS Component Running C/RTL Co-Simulation**



After the simulation is complete you should see the Co_simulation finished successfully message at the end of the transcript. You will also see the Reports folder under the Run command populated as shown in the preceding figure. The reports available after co-simulation include:

- Summary: reports the command line used and the time stamp on the results.
- Cosimulation: displays general information about the design, displays specific options used during Co-simulation, and displays performance and resource estimates for the design hierarchy. If `enable_dataflow_profiling` is enabled, performance data will be back annotated to the Dataflow Viewer report as described in Dataflow Viewer.

IMPORTANT! *If you want to calculate II, you must ensure there are at least two transactions in the RTL simulation.*

- Timeline Trace: As described in Timeline Trace Viewer.
- Wave Viewer: As described in Viewing Simulation Waveforms.
- Function Call Graph: Displays the post Co-simulation call graph as described in Function Call Graph.

Output of C/RTL Co-Simulation

In the Vitis Unified IDE the hierarchy of the HLS component is structured as follows: `<workspace>/<component>/<component>/hls/`. You can select the HLS component in the Vitis Component Explorer and select **Open In Terminal** from the right-click menu to explore the contents of the component folders.

When C/RTL Cosimulation completes, the `sim` folder is created inside the solution folder. This folder contains the following elements:

- A verification folder named `sim/verilog` or `vhdl` is created for each RTL language that is verified.
  - The RTL files used for simulation are stored in the `verilog` or `vhdl` folder.
  - The RTL simulation is executed in the verification folder.
  - Any outputs, such as trace files and waveform files, are written to the `verilog` or `vhdl` folder.
- The `sim/report` folder contains the report and log file for each type of RTL simulated.
- Additional folders `sim/autowrap`, and `tv`, `wrap` and `wrap_pc` are work folders used by the HLS compiler. There are no user files in these folders.

TIP: *If the `cosim.setup` option was selected in the config file, an executable is created in the `sim/<hdl>` folder but the simulation is not run. The simulation can be manually run by executing the simulation `.sh` or `.exe` in a Terminal.*

Viewing Simulation Waveforms

To view waveform data during RTL co-simulation, you must enable the following in the Config File Editor, or the config file:

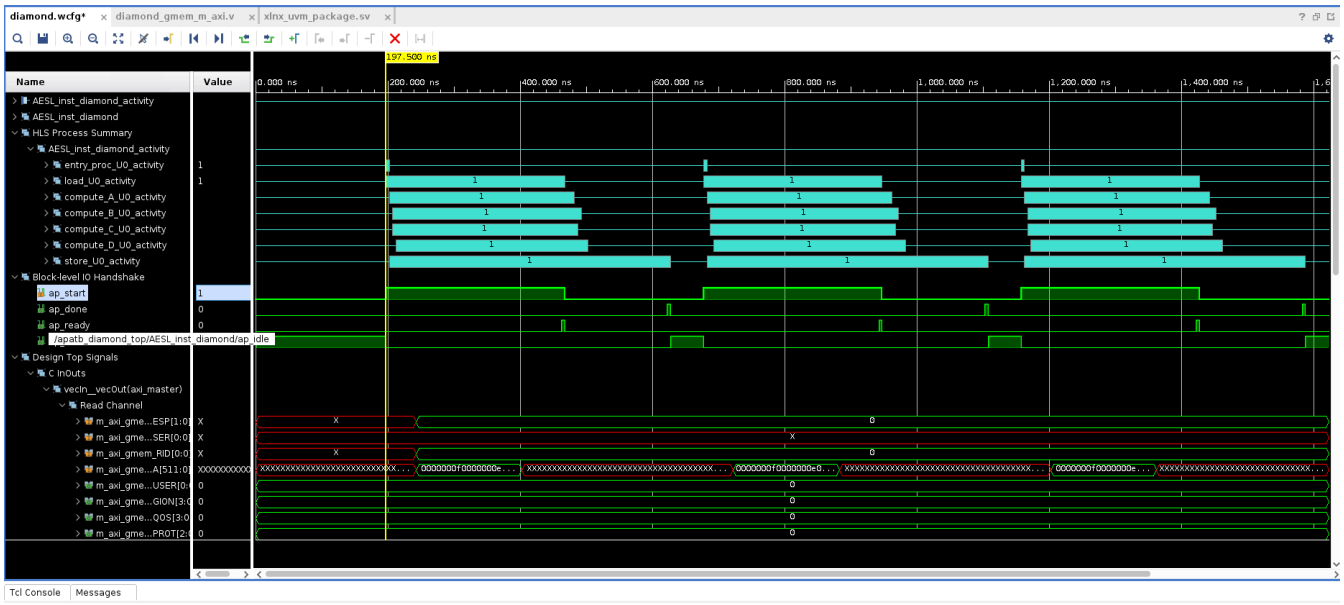
- Under tool select **xsim** as the RTL simulator, or set `cosim.tool=xsim` in the config file
- Under trace_level select **port** or **all**, or set `cosim.trace_level=port` in the config file

Vivado simulator GUI opens and displays all the processes in the RTL design. Visualizing the active processes within the HLS design allows detailed profiling of process activity and duration within each activation of the top module. The visualization helps you to analyze individual process performance, as well as the overall concurrent execution of independent processes. Processes dominating the overall execution have the highest potential to improve performance, provided process execution time can be reduced.

This visualization is divided into two sections:

- HLS process summary contains a hierarchical representation of the activity report for all processes.
  - **DUT name:** <name>
  - **Function:** <function name>
- Dataflow analysis provides detailed activity information about the tasks inside the dataflow region.
  - **DUT name:** <name>
  - **Function:** <function name>
  - **Dataflow/Pipeline Activity:** Shows the number of parallel executions of the function when implemented as a dataflow process.
  - **Active Iterations:** Shows the currently active iterations of the dataflow. The number of rows is dynamically incremented to accommodate for the visualization of any concurrent execution.
  - **StallNoContinue:** A stall signal that tells if there were any output stalls experienced by the dataflow processes (the function is done, but it has not received a continue from the adjacent dataflow process).
  - **RTL Signals:** The underlying RTL control signals that interpret the transaction view of the dataflow process.

Figure 72: **Waveform Viewer**



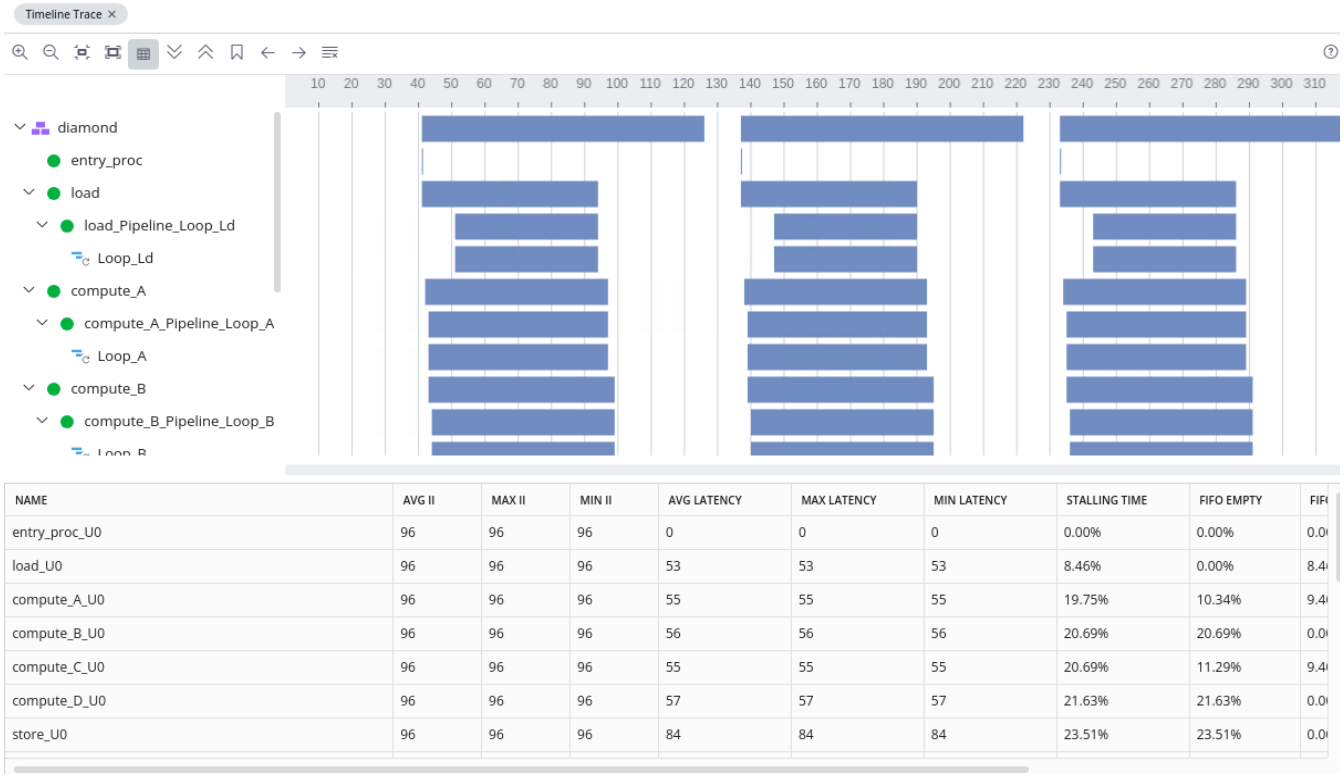
After C/RTL co-simulation completes, you can reopen the RTL waveforms in the Vivado IDE by selecting **Wave Viewer** under the REPORTS header for C/RTL COSIMULATION.

IMPORTANT! *When you open the Vivado IDE using this method, because the waveform was previously generated you can only use the waveform analysis features, such as zoom, pan, and waveform radix.*

Timeline Trace Viewer

The Timeline Trace report is available after C/RTL Co-Simulation. The Timeline Trace viewer displays the runtime profile of the functions of your design. It is especially useful to see the behavior of dataflow regions after Co-simulation, as there is no need to launch the Vivado logic simulator to view the timeline.

Figure 73: **Timeline Trace Viewer**



NAME	AVG II	MAX II	MIN II	AVG LATENCY	MAX LATENCY	MIN LATENCY	STALLING TIME	FIFO EMPTY	FIF
entry_proc_U0	96	96	96	0	0	0	0.00%	0.00%	0.0
load_U0	96	96	96	53	53	53	8.46%	0.00%	8.4
compute_A_U0	96	96	96	55	55	55	19.75%	10.34%	9.4
compute_B_U0	96	96	96	56	56	56	20.69%	20.69%	0.0
compute_C_U0	96	96	96	55	55	55	20.69%	11.29%	9.4
compute_D_U0	96	96	96	57	57	57	21.63%	21.63%	0.0
store_U0	96	96	96	84	84	84	23.51%	23.51%	0.0

Timeline Trace viewer displays multiple iterations through the various sub-functions of a dataflow region as shown in the preceding figure. It shows where the functions are starting and ending, and displays the Co-simulation data in tables below the timeline. To generate the Timeline Trace during C/RTL Co-simulatio you should enable `cosim.trace_level=all` and `cosim.enable_dataflow_profiling=true` options in the config file, or from the Config File Editor.

The Timeline Trace view also shows FIFO and PIPO channel stall/starve states with Full and Empty markers as shown above. In the preceding figure, you can see the `read_data` PIPO is empty, resulting in stalls 68% of the time as reported in the table below the graph.

The Dataflow Viewer displays a menu bar at the top of the report that includes the following features:

- Zoom In/Zoom Out/Zoom Fit/Zoom Full commands
- Toggle Table command to show or hide the table beneath the graph
- Expand All/Collapse All to expand or collapse the design hierarchy
- Previous Marker/Next Marker lets you move from one marker to the next in the timeline
- Delete All Markers remove markers from the timeline
- Legend command for the display

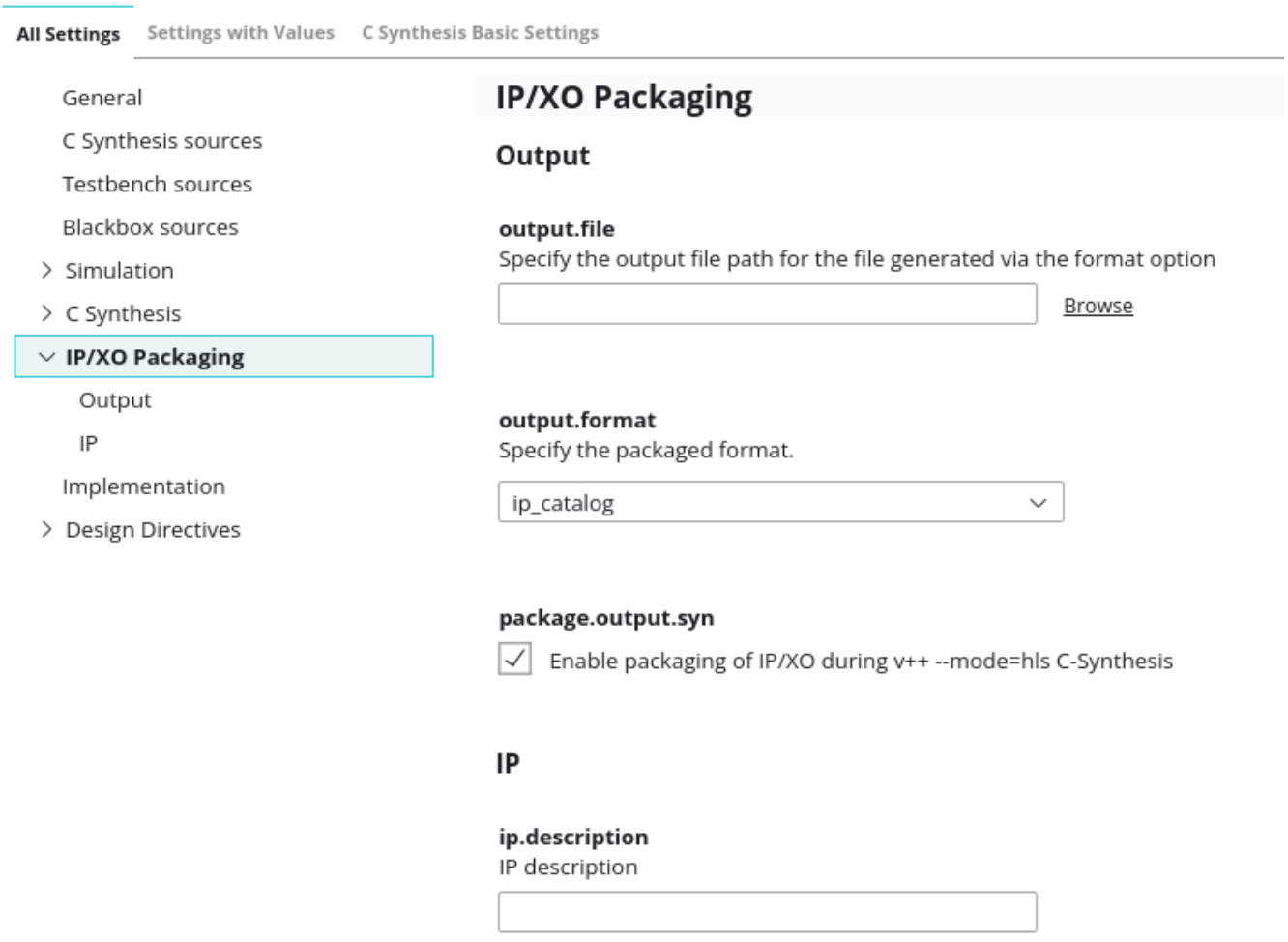
Packaging the RTL Design

The final step in the HLS component flow is to package the RTL design into a form that can be used by other tools in the design flow, such as in the Vivado Design Suite as part of a larger system design. Click the **Package** command in the Flow Navigator to export the RTL as a Vivado IP, Vitis kernel, or a synthesized checkpoint.

Configure Packaging

You can configure the packaging process to export the IP or kernel according to your settings prior to running package. The Config File Editor separates Packaging into Output and IP sections as shown below.

Figure 74: **Config File Package Settings**



Specifying the Output Format and Output File

The output.format for the RTL synthesis results must be specified. The default format is RTL, which lets the tool synthesize the Verilog and VHDL code from the C/C++ source files, but does not generate the Vivado IP or Vitis kernel as an output product. This approach lets you run synthesis quickly without having to generate hardware files at every iteration. However, to use the RTL design in downstream processes like embedded software design, or Application projects, you must generate the hardware files. The flow you are supporting in your design determines the best choice for output format.

The Package options, and their default values are listed below:

- **package.output.file:**

  The output.file name and location are determined by the tool, and default to the name of the top function specified for the HLS component. However, you can specify the name and location to override the default values.

- **package.output.format:**

  The `output.format` command supports several different formats as shown in the following table.

  Table 44: **Package Output Formats**

Package Format	Default Location	Comments
`package.out.format=ip_catalog`	`<comp-name>.zip`	The IP is exported as a ZIP file that can be added to the Vivado IP catalog. The `impl/ip` folder also contains the contents of the unzipped IP.
`package.out.format=xo`	`<comp-name>.xo`	The XO file output can be used for linking by the Vitis compiler in the application acceleration development flow. You can link the Vitis kernel with other kernels, and the target accelerator card, to build the `xclbin` file for your accelerated application.
`package.out.format=sysgen`	`<comp-name>.zip`	The IP is exported as a ZIP file that for use with the Vivado edition of System Generator for DSP. The `impl/ip` folder also contains the contents of the unzipped IP.
`package.out.format=rtl`	Creates Verilog and VHDL folders in the HLS component working directory.	This options lets you skip the generation of the packaged IP or XO modules to save time while iterating on the initial design.

- **package.out.syn:** Enable or disable the creation of the IP or XO during synthesis. Specify `false` to disable generation of the packaged IP or XO, or specify `true` to enable it. This options lets you skip the generation of the packaged IP or XO modules to save time while iterating on the initial design.

These selections result in the following config file entries:

```
[HLS]
package.output.format=xo
package.output.file=../../<filename>
package.output.syn=false
```

IP Configuration

When you specify `package.output.format=ip_catalog` in the HLS configuration file, you can also specify additional fields that will be applied to the generated IP, such as the Vendor, Library, Name, and Version (VLNV) of the IP.

The Configuration information is used to differentiate between multiple instances of the same IP when it is loaded into the Vivado IP catalog. For example, if an implementation is packaged for the IP catalog, and then a new solution is created and packaged as IP, the new solution by default has the same name and configuration information. If the new solution is also added to the IP catalog, the IP catalog will identify it as an updated version of the same IP and the last version added to the IP catalog will be used.

The IP options, and their default values are listed below:

- **package.ip.vendor:** xilinx.com
- **package.ip.library:** hls
- **package.ip.name:** Default to HLS component name
- **package.ip.version:** 1.0
- **package.ip.description:** An IP generated by HLS component
- **package.ip.display_name:** This field left blank
- **package.ip.taxonomy:** This field left blank
- **package.ip.xdc_file:** This field left blank
- **package.ip.xdc_ooc_file:** This field left blank

TIP: *The packaging process in the Vitis Unified IDE uses the `vitis-run --mode hls --package` command as described in vitis, v++, and vitis-run Commands.*

After the packaging process is complete, the ZIP file archive or XO kernel is written to the specified `package.output.file` location, or written in the HLS component folder. The exported IP file can be imported into the Vivado IP catalog and used in any design. The exported Vitis kernel can be used with the `v++ --link` command as paret of a larger system design.

Software Driver Files

For designs that include AXI4-Lite slave interfaces, a set of software driver files is created during the export process. These C driver files can be included in a Vitis embedded software development project, and used to access the AXI4-Lite slave port.

The software driver files are written to directory `solution/impl/ip/drivers` and are included in the packaged IP `export.zip`. Refer to AXI4-Lite Interface for details on the C driver files.

Running Implementation

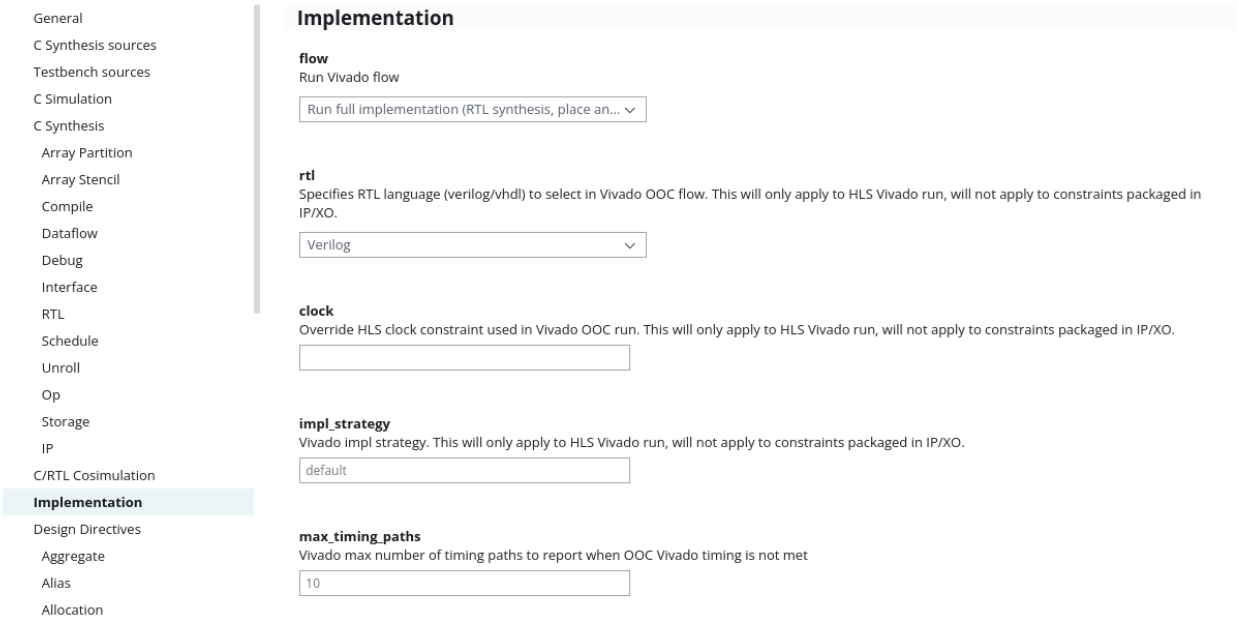
Make sure the HLS component is active in the Flow Navigator, or select it from the Component menu to make it the active component in the tool. When the HLS component is the active component, the Flow Navigator enables running C Simulation, C Synthesis, C/RTL Co-simulation, Packaging, and Implementation to build and analyze the HLS component. Select **Run** beneath the IMPLEMENTATION heading in the Flow Navigator.

When the HLS compiler reports the results of high-level synthesis, it provides an estimate of the results with projected clock frequencies and resource utilization (LUTs, DSPs, BRAMs, etc). These results are only estimates because the tool cannot know what optimizations or routing delays will be in the final synthesized or implemented design. To get a better analysis of the RTL design you can actually run Vivado synthesis and place and route on the generated RTL design, and review the results of timing and resource utilization. However, even these results are only improved estimates, because the IP or kernel being used in a larger design with other IP and kernels will yield different results.

Configure Implementation

You can configure Vivado synthesis and implementation prior to running it by using the Implementation section of the Config File Editor, as shown below, or by editing the HLS config file directly.

Figure 75: **HLS Component Implementation Settings**



The configuration commands for Implementation include:

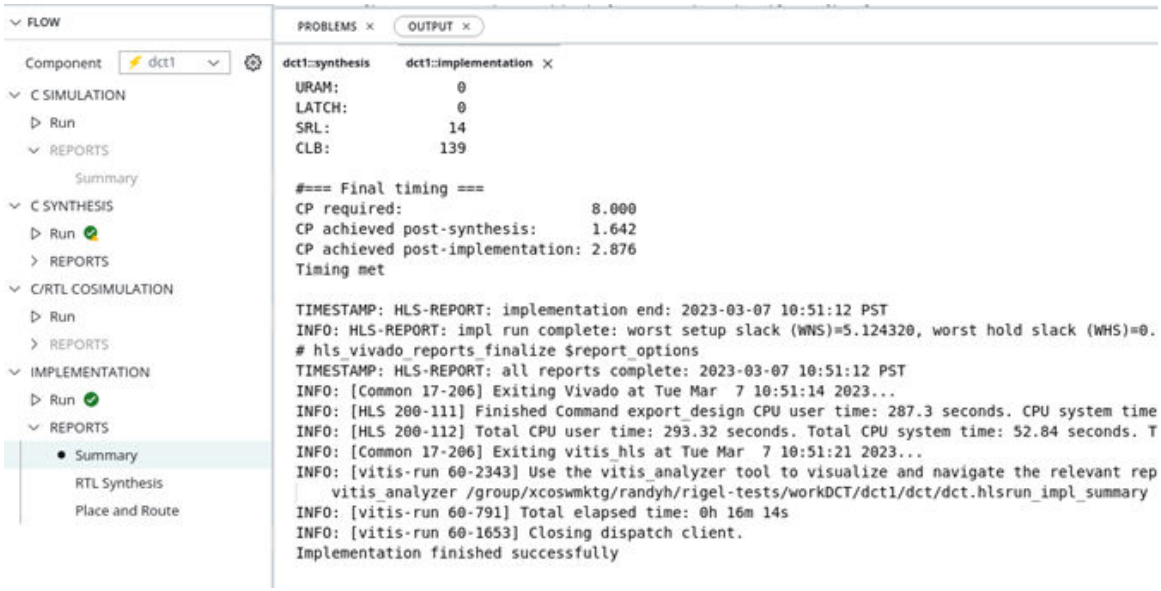
- **vivado.flow:** Specify to run only synthesis or both synthesis and implementation. Synthesis alone will run faster than both synthesis and implementation, but will lack some of details of the implementation run. The default is `syn`.
- **vivado.rtl:** Specifies the language to use when running Vivado out-of-context flow. The default is Verilog.
- **vivado.clock:** Specify the clock period to use during synthesis or implementation. When not specified, the default clock specified when the HLS component is created is used.
- **vivado.impl_strategy:** Specify the strategy to employ in the implementation run. This is only for use during the implementation run for resource utilization and timing estimates, and does not affect the generated Vivado IP or Vitis kernels.
- **vivado.max_timing_paths:** Specify the number of timing paths to extract from the Timing Summary report. The specified number of worst case paths are returned.
- **vivado.optimization_level:** This is a general feature to manage the optimizations performed by the Vivado tool. The higher the setting, the more optimizations are employed, and the longer the runtime as a result.
- **vivado.pblock:** Specifies a Pblock range or value to use during placement and routing to limit the area available for the design.
- **vivado.phys_opt:** Specify the physical optimization to run. Choices include: `none`, `place`, `route`, and `all`.

- **vivado.report_level:** Defines the report-level generated during synthesis or implementation. The report can include the utilization and timing summary, timing path details, or a failfast report, which is the default.
- **vivado.synth_design_args:** Specify options for the `synth_design` command.
- **vivado.synth_strategy:** Specify the strategy to employ in the Vivado synthesis run.

Run Implementation

With the Implementation setup defined in the config file you are ready to select **Run** from the Flow Navigator. You can track the progress of the implementation run in the Output window. The transcript for the run will have the top function name as `<component-name>::implementation` as shown below.

Figure 76: **HLS Component Running Implementation**



TIP: *The implementation run in the Vitis unified IDE uses the `vitis-run --mode hls --impl` command.*

After the implementation is complete you should see the Implementation finished successfully message at the end of the transcript. You will also see the Reports folder under the IMPLEMENTATION heading populated with the following reports:

- Summary: Reports the command line used and the time stamp on the results
- RTL Synthesis: Reports the results of synthesis including resource use and timing
- Place and Route: As described in Implementation Report

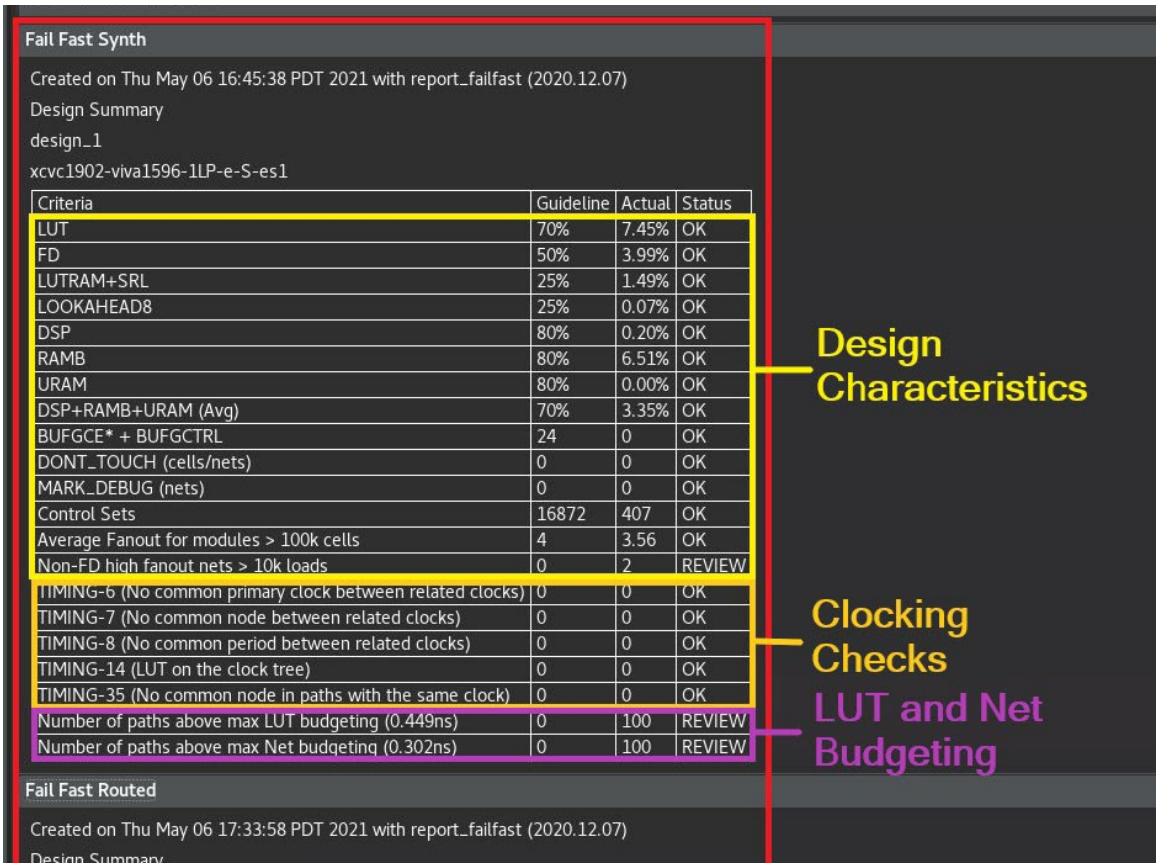
TIP: *You can cancel the Implementation run using the **Cancel Run** command from the Flow Navigator.*

Implementation Report

The Implementation Report contains the results of Synthesis and Place and Route if it was run. The sections of the report include the following:

- **General Information:** Provides general information related to the design and implementation.
- **Run Constraints and Options:** Reports the constraints and options that were set for the RTL Synthesis run and/or the Place & Route run. This shows you what constraints were set and/or modified for the run.
- **Resource Usage/Final Timing:** The Resource Usage and the Final Timing sections show a quick summary of the resources and timing achieved by either the RTL Synthesis run or the Place & Route run. These sections give a very high-level overview of the resource utilization and status on whether timing goals were met or not. The information in the succeeding sections provide details useful in debugging timing issues.
- **Resources:** A detailed per-module split up of resources is shown in this table. In addition, the tables can also show the original variable and source location information from the source code. If a particular resource was the result of a user-specified pragma, then this can also be shown in the table. This allows you to relate your C code with the synthesized RTL implementation. Inspecting this report is very beneficial because this is after Vivado has synthesized the design and therefore, functional blocks like DSPs and other logic units have all now been instantiated in the circuit.
- **Fail Fast:** The fail fast reports that Vivado provides can guide your investigation into specific issues encountered by the tool. In the fail fast report, you should look into anything with the Status of REVIEW to improve the implementation and timing closure. Different sections of the fail fast report include:
  - Design Characteristics: The default utilization guidelines are based on SSI technology devices and can be relaxed for non-SSI technology devices. Designs with one or more REVIEW checks are feasible but are difficult to implement.
  - Clocking Checks: These checks are critical and must be addressed.
  - LUT and Net Budgeting: Use a conservative method to better predict which logic paths are unlikely to meet timing after placement with high device utilization.

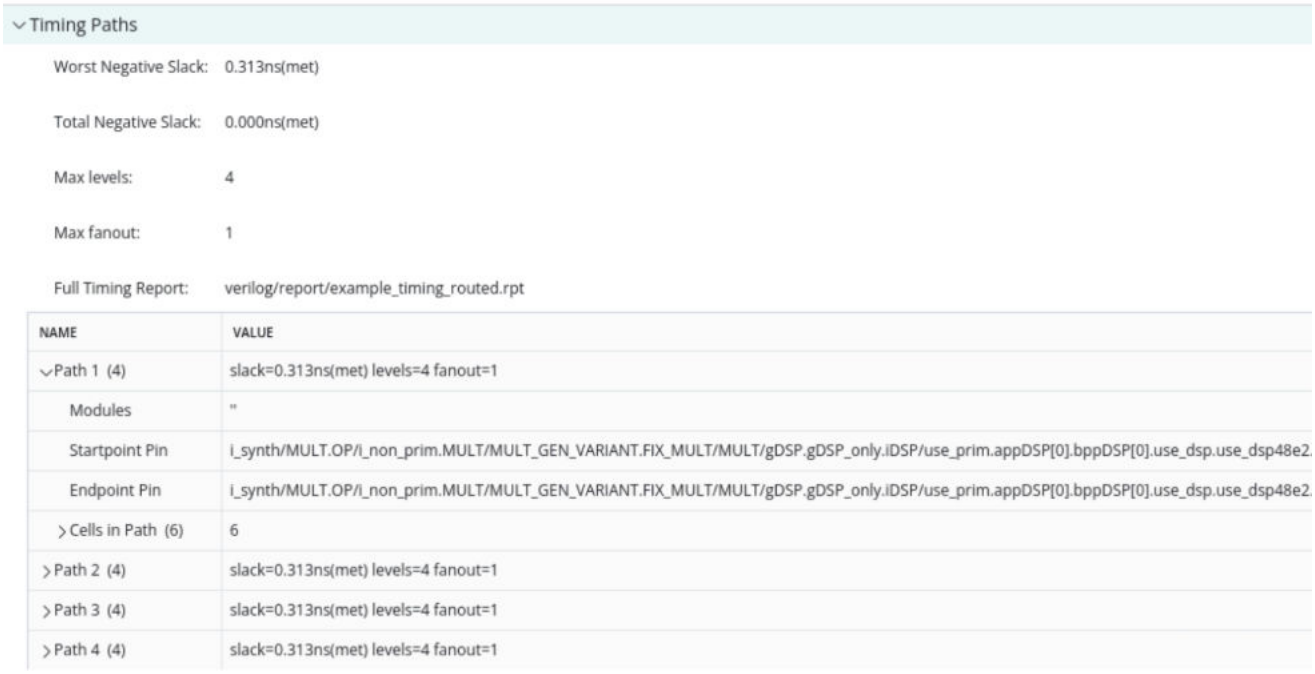
Figure 77: **Design Characteristics**



- **Timing Paths:** The Timing Paths reports show the timing critical paths that result in the worst slack for the design. By default, the tool will show the top 10 worst negative slack paths. Each path in the table has detailed information that shows the combination path between one flip-flop to another. Breaking these long combinational paths will be required to address the timing issues. So you need to analyze these paths and reason where they are coming from and map these paths back to your C code. Using both these paths and the resources table presented earlier can help in determining and correlating the path back to your source code.

  In the following figure, you can see that the top 10 negative slack paths in the Place & Route report actually have higher logic levels (9) as compared to after RTL Synthesis (5), and the max fanout also got worse (64 → 9366). This clearly shows how congestion in the design is causing high logic levels and higher fanouts which in turn causes issues for meeting timing. Using such clues, you can modify your design to remove some of this congestion either by rewriting the C code or making some different design decisions with respect to BRAM/LUTRAM/URAM resource choices.

Figure 78: **Timing Paths Report**

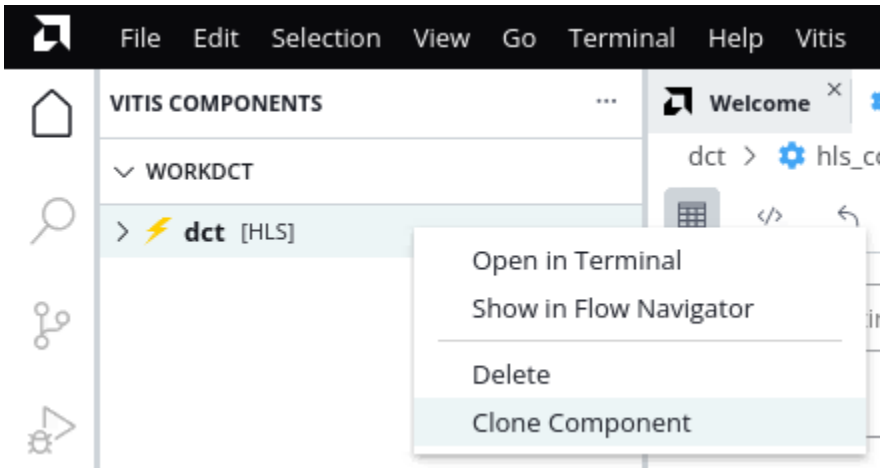


Cloning HLS Components

The most typical use of the HLS compiler is to create an initial design, analyze the results, and then perform optimizations to meet the desired area and performance goals. This is often an iterative process, requiring multiple steps and multiple optimizations to achieve the desired results. When you begin to explore possible solutions to design challenges, you can clone an HLS component to let you preserve one while using the second for exploration.

To clone an HLS component right-click the component in the Vitis Component Explorer and select the **Clone Component** command, as shown in the following figure. This opens the Clone Component dialog box, prompting you for a new name for the component.

Figure 79: **Clone Component**



After creating the cloned component, the design will include the same source files, test bench files, top-level function, and HLS configuration file settings. You can edit the configuration file settings or other settings of the new HLS component and begin working through the build process. Keep the original HLS component in its current state, or develop two different branching approaches to the design.

Component Comparison Feature

The Component Comparison tool allows users to compare a set of HLS components in terms of performance and utilization. Reports can be generated in tabular or graphical format. In addition, the report data can be exported in CSV format.

Compare Reports can be generated for HLS Components only and the components must exist in an opened Vitis unified software platform workspace. To use this feature, the HLS component(s) selected for comparison should have at least one of the following stages completed successfully:

- C Synthesis
- C/RTL Co-simulation
- Implementation

IMPORTANT! *Only completed stage data will be available in the generated HLS Compare Report.*

To generate a report, go to the VITIS COMPONENTS panel in the Vitis Unified IDE and with the mouse hover over any of the HLS components, select its "HLS Compare Reports" icon. The icon will become visible when you hover over it. Alternatively, go to View and then select HLS Compare Reports.

The HLS Compare Report panel will appear. Check the boxes of the desired components from the list of available HLS components. Finally, select 'Compare'.

An HLS Compare Report file in tabular format is generated and visible in the editor panel. Component information and metrics values are populated in tabular form for the completed stages of the selected HLS components.

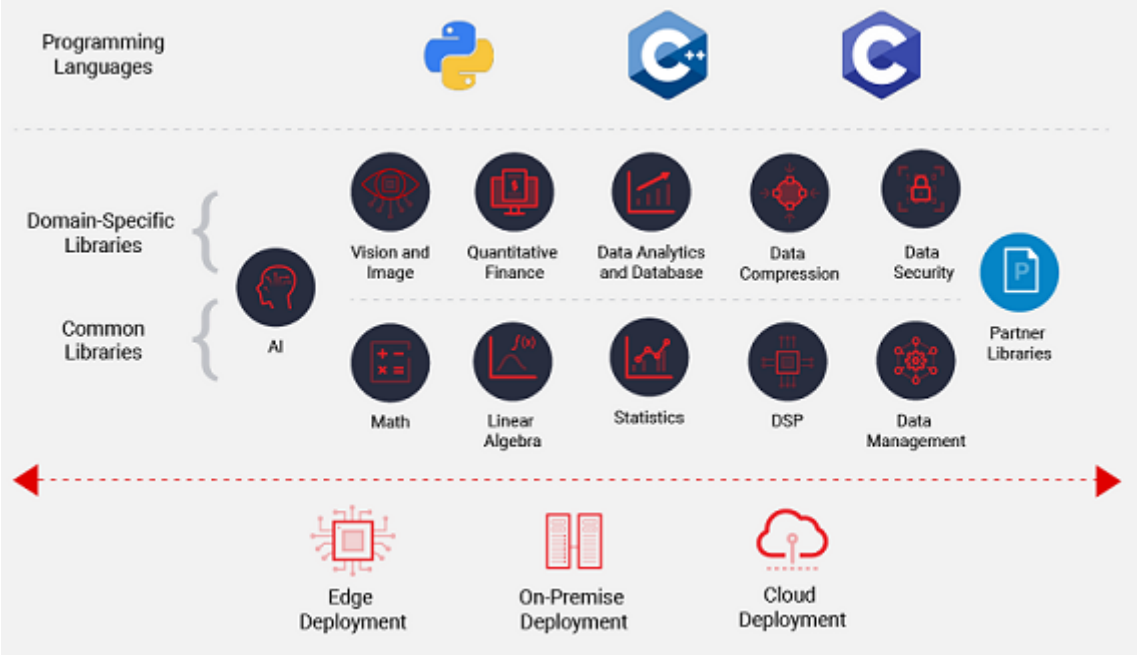
- To export the tabular data as a CSV formatted file, select the export (a cloud with a down arrow) icon in the upper right corner of the report. A Compare Reports Export CSV File panel will appear. Enter the file name and select Save.
- To generate an HLS Compare Report in graphical format, from the tabular HLS Compare Report select the Graph Compare (a bar graph) icon in the upper right corner of the report.
- To switch back to the tabular report, select the Table Compare (a spreadsheet) icon in the upper right corner of the report.

L1 Library Wizard Flow

The L1 Library Wizard Flow feature provides a tool-assisted approach to access a subset of the Vitis Accelerated Libraries HLS functions. You can select library functions and generate examples as well as parametrized template functions that can be incorporated into your design. In this way, you can quickly add IP-based functionality to their design from pre-built HLS-based IP.

The Vitis Libraries contain an extensive set of open-source, performance-optimized libraries that offer out-of-the-box acceleration with minimal to zero-code changes to your existing applications. The common Vitis accelerated libraries for Math, Statistics, Linear Algebra, and digital signal processing (DSP) offer a set of core functionality for a wide range of diverse applications. The domain-specific Vitis accelerated libraries offer out-of-the-box acceleration for workloads like Vision and Image Processing, Quantitative Finance, Database and Data Analytics, and Data Compression.

Figure 80: **Vitis Libraries**



Three types of implementations are provided in the Vitis libraries, namely L1 primitives, L2 kernels and L3 software APIs. These implementations are organized in their corresponding directories L1, L2 and L3. The L1 primitive implementations can be leveraged by FPGA hardware developers. The L2 kernel implementations provide usage examples for Vitis host code developers. The L3 software API implementations provide C, C++, and Python function interfaces to allow pure software developers to offload operations to pre-built FPGA images, also called overlays. The Vitis vision library provides a software interface for computer vision functions accelerated on FPGA and AI Engine devices.

The Vitis L1 Library Flow feature offers a tool-assisted approach to quickly incorporate L1 Library HLS functions into your design for the Vitis Solver and Vision Library subfolders only. The Solver Library provides a collection of matrix decomposition operations, linear solvers, and eigenvalue solvers. The Vitis vision library has been designed to work in the Vitis development environment and provides a software interface for computer vision functions accelerated on an FPGA using Vitis HLS. Vitis vision library functions are mostly similar in functionality to their OpenCV equivalent.

***Note*:** To use the Vitis Vision library, the OpenCV library v4.4.0 must be installed and accessible on your system. For instructions on how to install the OpenCV library on Windows, see Install/Setup OpenCV on Windows 10 and for Linux, see Compiling and Installing OpenCV libraries for use with Vision library.

There are three tool-assisted flows to access the L1 HLS functions:

1. Examples view
2. Create component from Library from the Welcome Page

3. Auto-completion from xf:: namespace in Vitis Unified IDE editor

> **TIP:** *To ensure access to the L1 Library HLS functions, ensure the Solver and Vision subfolders of the Vitis Libraries have been downloaded.*

Flow 1 - Examples - Create L1 HLS function examples

The Examples VIEW flow allows you to select and generate an L1 Library example. The result is a specified L1 Library HLS component project with both test bench and source code files.

To generate an L1 Library Example:

1. Select the Examples icon in the Vitis Unified IDE.
2. In the Examples panel, go to the **HLS Examples → VItis Accelerated Libraries Repository → Vitis Vision Library → L1 → Examples**.
3. Select the desired example, for example, AMD 3D LUT L1 Test.
4. Click on the **Create HLS Component Template** button.
5. Set the component name and location. Select **Next**.
6. Select **Finish** to generate the example HLS component.

Flow 2 - Create component from Library

The "Create component from Library" flow allows you to select an HLS template function and parametrize it. An HLS component project is created with the instantiated template function in a C++ wrapper for the synthesizable file. No test bench file is provided. To perform this flow:

1. Select **Create Component from Library** from the Welcome page.
2. Enter the component name and location in the "Create Component from Library" pop-up panel and select **Next**.
3. Select the desired L1 HLS function and select **Next**.
4. In the parametrization panel, configure the parameters.
    a. Grey text fields are required. In some cases, legal integer ranges are specified. For example, "int(16~2160)" means the field value can be an integer in the range 16 to 2160 inclusive.
5. After defining the parameters, select "Next."
6. Select the Part then select "Next." You can use the default part by selecting "Next" immediately.
7. Edit the Settings then select "Next." You can use default settings by selecting "Next" immediately.
8. Review the new HLS component settings and select "Finish."

Flow 3 - Auto-completion from the xf:: namespace in IDE Editor

The "auto-completion from the xf:: namespace" flow provides a way to insert a parametrized HLS template function into source code that is being actively edited.

1. In the Vitis Unified IDE editor, open a C++ source code file.
2. Type "xf::" in an empty line in the file editor panel.
3. A list of available HLS functions will be displayed for auto-completion. Select a function, for example, xf::cv::absdiff.
4. Once the function is selected, the HLS function's parametrization window opens, fill in the Parameters and Ports sections of the panel.
   a. In the Parameters section, grey text fields are required. In some cases, legal integer ranges are specified. For example, "int(16~2160)" means the field value can be an integer in the range 16 to 2160 inclusive.
   b. In the Ports section, you can rename the arguments to match the source code context.

Creating HLS Components from the Command Line

While the prior content described building and running an HLS component using the Vitis Unified IDE, this section focues on doing that using the command line tools that are available.

The `v++ -c --mode hls` compiler command is used to build an HLS component. The command uses a configuration file command language. Running C simulation, Code Analyzer, C/RTL Co-simulation, implementation, and export all rely on the `vitis-run` command. The different steps for the command-line flows are described below.

TIP: *The following examples are based on the DCT design in the Getting Started With Vitis HLS. The example includes the HLS component, with source files and test bench.*

Running C-Synthesis

To build the `dct` HLS component the `v++` command-line will look as follows:

```
v++ -c --mode hls --config ./dct/hls_config.cfg --work_dir dct
```

Where:

- `--config` specifies a config file with the compiler directives for the build, and to configure the simulator for the run
- `--work_dir` provides a work directory to build the component

> **TIP:** *When creating an HLS component from the command line, the `--work_dir` specifies the HLS component folder, and the parent folder of the `--work_dir` becomes the workspace for launching the Vitis IDE.*

The contents of a configuration file can vary, but for synthesis the `dct` HLS component requires the following commands in the `hls_config.cfg` file:

```
part=xczu9eg-ffvb1156-2-e

[hls]
syn.file=./src/dct.cpp
syn.top=dct
flow_target=vitis
clock=8ns
clock_uncertainty=12%
syn.output.format=rtl
syn.directive.pipeline=dct_2d II=4
```

> **IMPORTANT!** *If your HLS configuration file uses `platform=` instead of `part=` then you must also specify `freqhz=` instead of `clock=` as shown here to change the default clock frequency of the platform.*

The success of the synthesis command largely depends on the contents of the configuration file. There are a few key required elements, and then there are a number of options that you can specify. From the config file provided above, the required elements for synthesis are the `part`, `syn.file`, and `syn.top`. The `flow_target`, `clock`, and `clock_uncertainty` are not required except to override the default values. The `syn.directive.xxx` commands are used to provide specific optimization to the synthesis of the function.

Running C Simulation or Code Analyzer

To run the HLS component in C simulation or in Code Analyzer, use the `vitis-run` command. C simulation does not require C Synthesis to have been run as it does not require the RTL code generated by synthesis.

```
vitis-run --mode hls --csim --config ./dct/hls_config.cfg --work_dir dct
```

Where:

- `--csim` specifies the target for the run.
- `--config` specifies a config file as indicated for synthesis, but includes C simulation specific requirements as shown below.
- `--work_dir` provides a work directory to build the component as indicated for synthesis.

The contents of a configuration file can vary, but for C simulation the config file requires the source files and top specified for synthesis, but also require test bench and input files, as well as csim configuration settings as explained in C-Simulation Configuration:

```
part=xczu9eg-ffvb1156-2-e

[hls]
clock=8ns
clock_uncertainty=12%
flow_target=vitis
syn.output.format=rtl
syn.file=./src/dct.cpp
syn.file=./src/dct.h
syn.top=dct
tb.file=./src/dct_coeff_table.txt
tb.file=./src/dct_test.cpp
tb.file=./src/in.dat
tb.file=./src/out.golden.dat
csim.clean=true
csim.code_analyzer=false
syn.directive.pipeline=dct_2d II=4
```

TIP: *Change `csim.code_analyzer` to `true` to enable the Code Analyzer as well as simulation.*

Running C/RTL Co-Simulation

To run C/RTL Co-simulation on the HLS component use the `vitis-run` command as shown below:

```
vitis-run --mode hls --cosim --config ./dct/hls_config.cfg --work_dir dct
```

The contents of the configuration file required for C/RTL Co-Simulation include the following:

```
part=xczu9eg-ffvb1156-2-e

[hls]
clock=8ns
clock_uncertainty=12%
flow_target=vitis
syn.output.format=rtl
syn.file=./src/dct.cpp
syn.file=./src/dct.h
syn.top=dct
tb.file=./src/dct_coeff_table.txt
tb.file=./src/dct_test.cpp
tb.file=./src/in.dat
tb.file=./src/out.golden.dat
syn.directive.pipeline=dct_2d II=4
cosim.enable_dataflow_profiling=true
cosim.enable_fifo_sizing=true
cosim.trace_level=port
cosim.wave_debug=true
```

***Note*:** The `hls_config.cfg` file above shows some of the settings for the C/RTL co-simulation tool. Refer to Co-Simulation Configuration for more information.

Running Implementation

To run Vivado synthesis or implementation on the HLS component use the `vitis-run` command as shown below:

```
vitis-run --mode hls --impl --config ./dct/hls_config.cfg --work_dir dct
```

Exporting the IP/XO

To export a Vivado IP or Vitis kernel from the synthesized HLS component, you can use the Vitis-run --package command as shown below:

```
vitis-run --mode hls --package --config ./dct/hls_config.cfg --work_dir dct
```

The contents of the configuration file required to export the package IP or XO include the following:

```
part=xcvu9p-flga2104-2-i

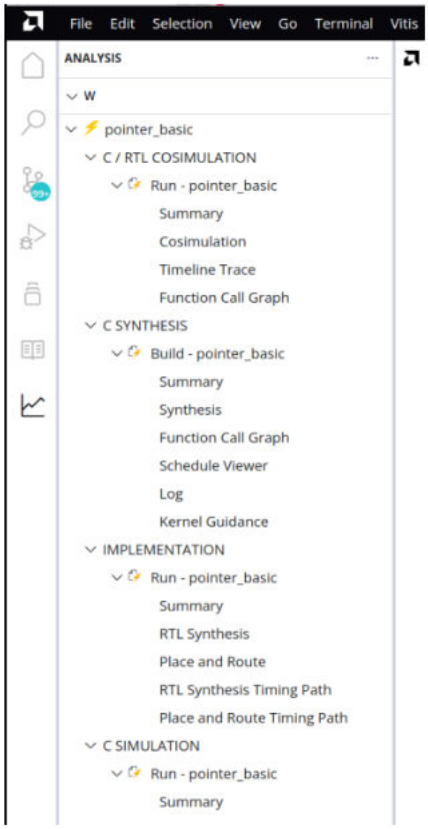
[hls]
syn.file=/group/xcoswmktg/randyh/rigel-tests/03-Vitis_HLS/reference-
files/src/dct.cpp
syn.top=dct
syn.output.format=xo
```

***Note*:** The `--package` command exports an IP or an XO from the previously synthesized HLS component. You can export a Vitis kernel as an IP. However, you cannot export a Vivado flow IP as an XO unless it meets the specific requirements of the Vitis kernel.

Opening HLS Component Flow Step Reports

Once an HLS Component flow steps have been run successfully, the reports for each flow step can be accessed all in one place in the Analysis view as show in the screenshot below.

Figure 81: **Opening HLS Component Flow Step Reports**



Chapter 5

Managing the AI Engine Component in the Vitis Unified IDE

Using the Vitis Unified IDE to Build and Simulate AI Engine Components

To launch the Vitis Unified IDE you must first setup the environment as previously described in Setting Up the Vitis Environment in the *Data Center Acceleration using Vitis* (UG1700). Then launch the tool using the `vitis` command. When the IDE opens, to get started:

1. Click **Open Workspace**.
2. Browse to a particular location to create an AI Engine component.

Alternatively, you can use Python API commands to automate the AI Engine component management (creation, build, edit configuration file, etc.) process in the Vitis Unified IDE. Python APIs in Command Line Interface (CLI) is supported in the Vitis IDE in interactive mode only. The Python API commands used to manage an AI Engine component are found in Managing AI Engine Components with the Python Command Line Interface.

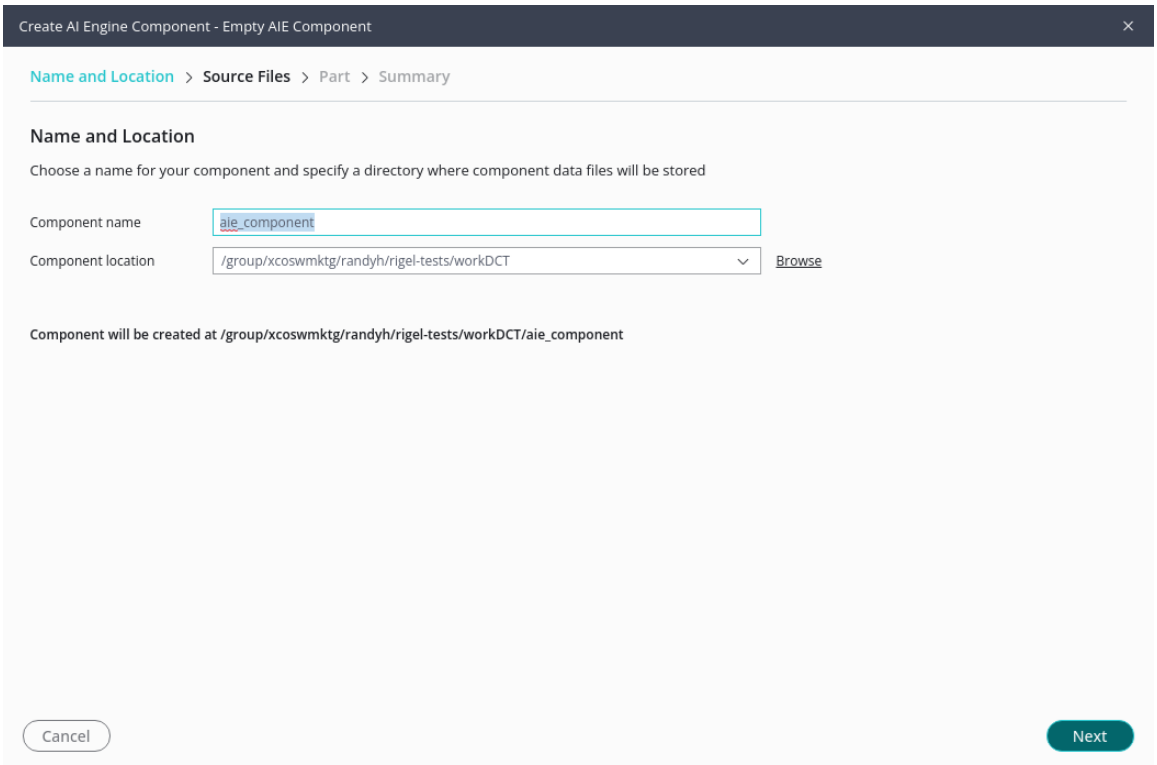
Creating an AI Engine Component

There are a number of ways to create an AI Engine component in the Vitis Unified IDE. The following steps describe the recommended approach.

1. With the Vitis Unified IDE opened, from the main menu select **File → New Component → AI Engine**.

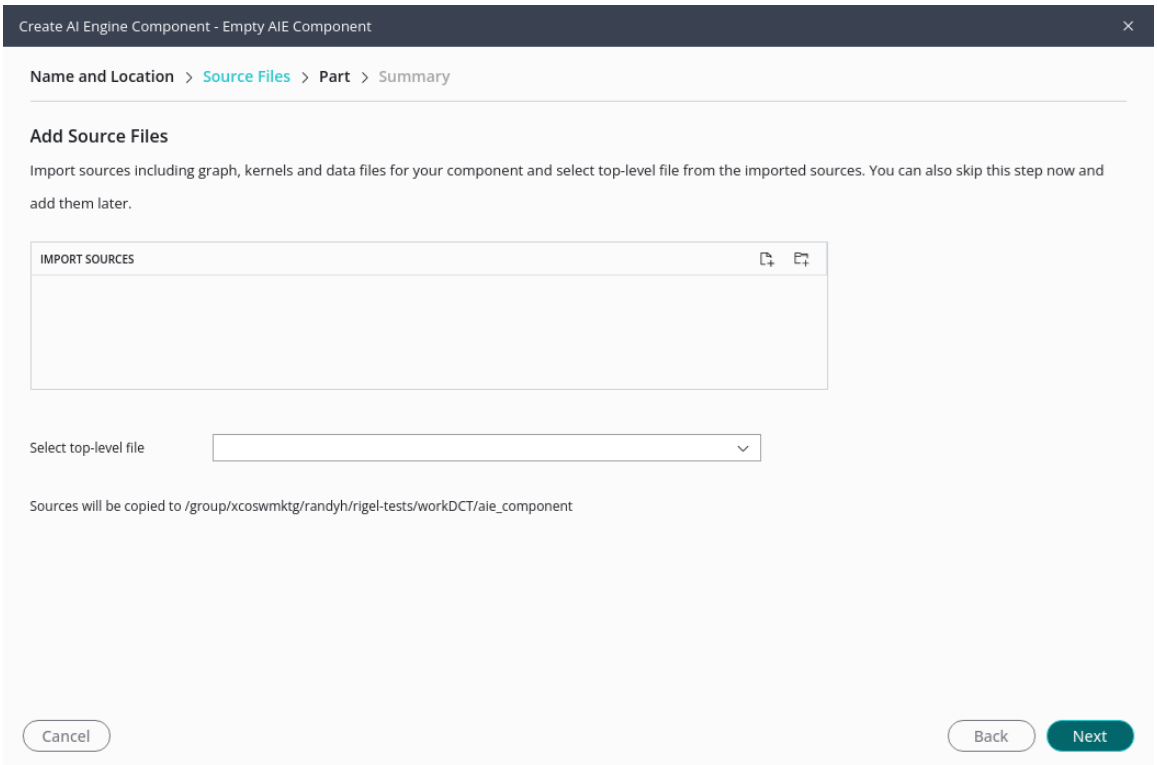
   TIP: *You can also select the* ***New Component*** *command from the Welcome page, or from the right-click menu in the Explorer view.*

   This opens the Choose Name and Location page of the Create AI Engine Component wizard as shown below.



TIP: *The images shown here are taken from the light theme of the Vitis Unified IDE. The default theme is the dark theme so your screens might appear different. You can change the color theme using the* ***File → Preferences → Color Theme*** *menu command.*

2. Enter a Component name and **Component location** and click **Next**. This opens the Add Sources page of the wizard as shown below.



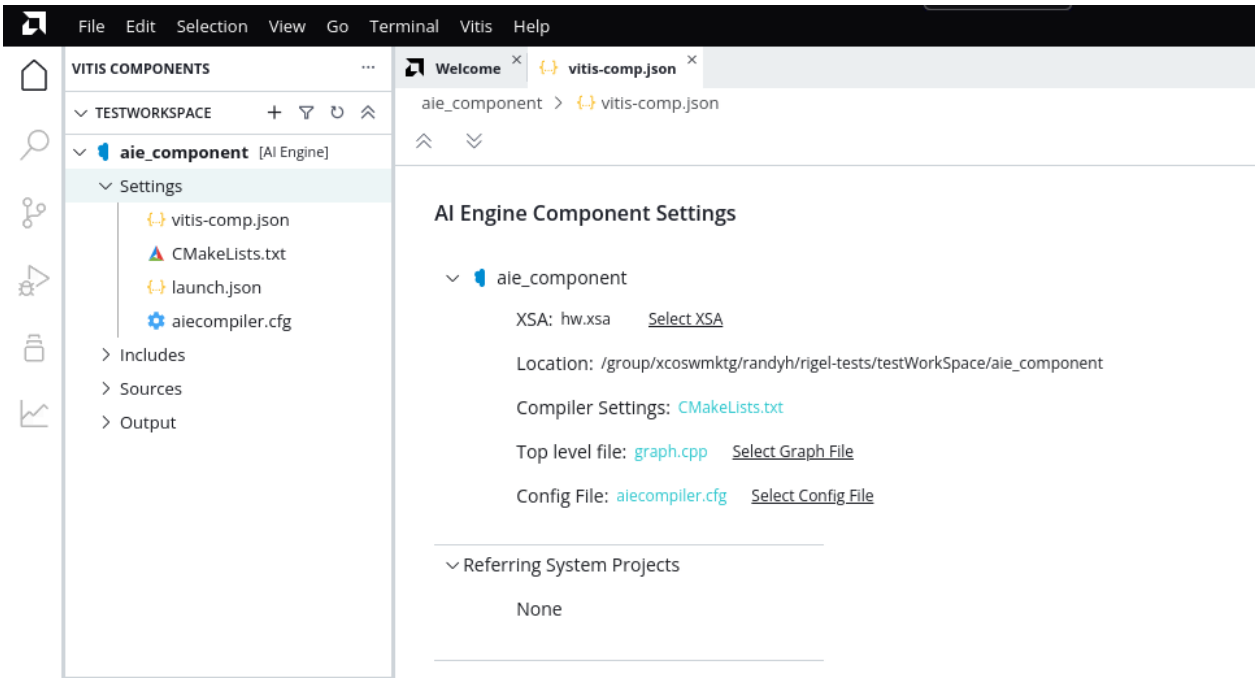
3. Select **Import Sources** to import by file or by folder. Adding a folder will include all of the current files in the folder, as well as sub-directories of the folder.

> **TIP:** *You can also skip this step for now, and add the source files after the AI Engine component is created using the* ***Import*** *command on the component* `Sources` *folder in the* ***Component Explorer*** *view as described in Importing Files.*

4. **Select the top-level file** from the files you have added to the component. The top-level file defines the graph application for the AI Engine component as described in Introduction to Graph Programming in the *AI Engine Kernel and Graph Programming Guide* (UG1079). The Vitis Unified IDE will automatically identify the top-level file containing the `main()` function of the graph application, although you should validate its selection before moving on.
5. Click **Next** and the **Select Part** page of the wizard opens.
6. Select **Part**, or **Platform**, or **XSA**.
7. Click **Next** to display the Summary page of the wizard. The Summary page reflects the choices you have made on the prior pages.
8. Review the summary and click **Finish** to create the component, or click **Back** to return to earlier pages and change your selections.

The AI Engine component is created after clicking **Finish**. The component is opened in the Vitis Unified IDE as shown in the image below. The new component is listed in the Component Explorer view on the left of the screen, and the `vitis-comp.json` file is opened in the central editor window to the right.

Figure 82: **AI Engine Component - vitis-comp.json**



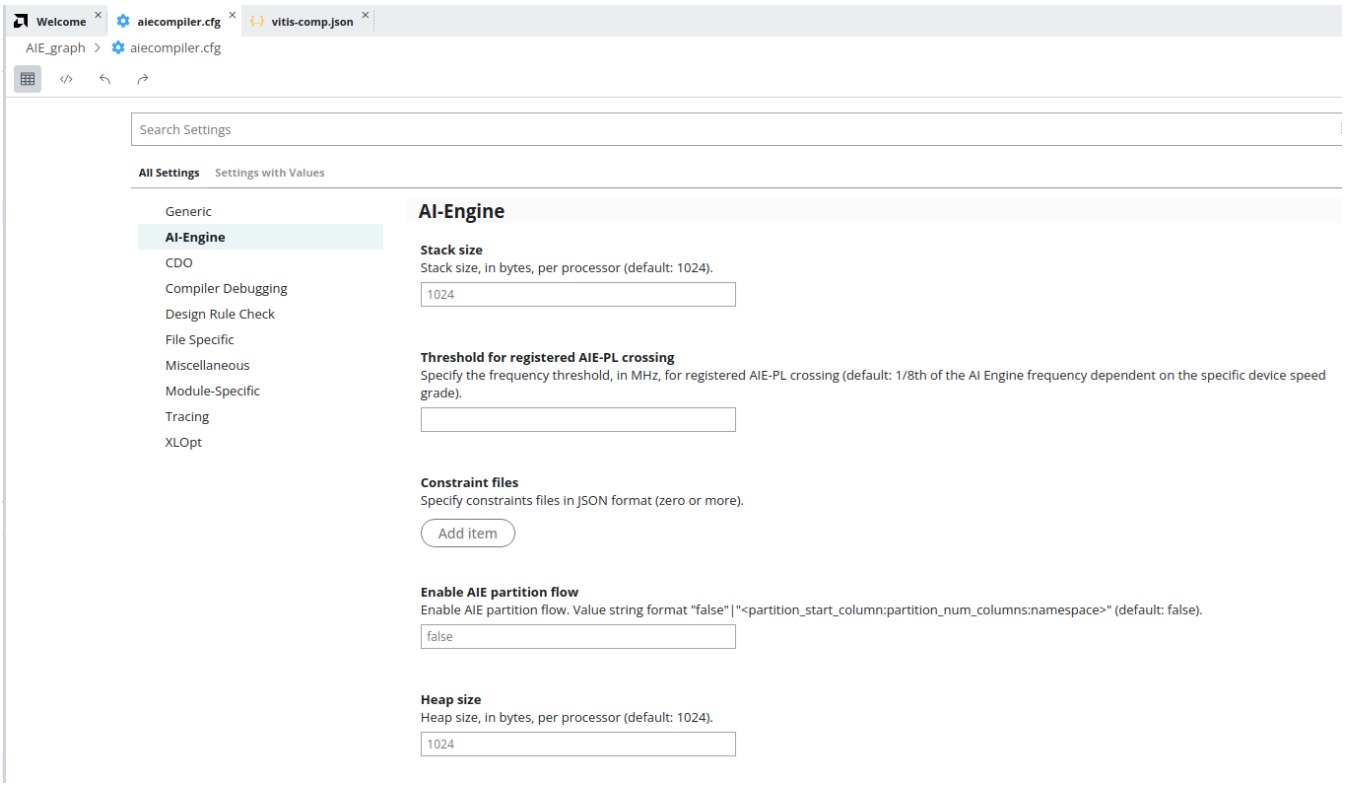
The `vitis-comp.json` is the file that stores the component configuration. It contains the component name, the platform or part the components is built against, the location, compiler settings, top-level file, and config file that define the AI Engine component.

- Compiler Settings provides access to the `CMakeLists.txt` file that lets you configure the compilation process.
- Top-level file provides access to the source code which controls the graph.
- Config File provides access to the configuration file that defines options for the compilation process used by the `v++ --mode aie` command to build the AI Engine component. When you create the component the configuration file is populated with default values for many of the options. The next section describes the config file in greater detail.

Defining the AI Engine Configuration File

The AI Engine component configuration file contains information used for the compilation and simulation of the component. To access the file, click **Configuration File** (`aiecompiler.cfg`) in the settings directory- of the AI Engine component. This will open the Configuration File Editor in the central editor window.

Figure 83: **AI Engine Configuration File**



For the AI Engine component, the configuration file includes the following sections:

- **Generic:** Includes AI Engine options like input files, output file name, kernel frequency, and part name.
- **AI Engine:** Compiler options applied to the graph.
- **CDO:** Options for graph configuration and initialization in configuration data object (CDO) format.
- **Compiler Debugging:** Log-levels and compilation options to help analyze and debug the compilation process.
- **Design Rule Check:** Options to enable, disable or waive Design Rule Check for the specified ID.
- **File Specific:** Source files and include files required for building and running the component.
- **Miscellaneous:** Collection of options for added functionality.
- **Module-Specific:** Compilation options specific to each kernel in the design.
- **Tracing:** Options to configure event trace when running the design in hardware. Some event tracing options require specific resources in the design and can affect the performance of the design.

- **XLOpt:** Option to set the level of kernel optimizations. Each of these levels enables specific combinations of optimizations.

> **TIP:** *Each of the config file options has a brief description in the Vitis IDE for ease-of-use, and are also* `v++ --mode aie` *command-line options. Refer to v++ Mode AI Engine for more detailed information on the options.*

There is a Search bar at the top of the Config File Editor which lets you quickly search through the options using a keyword or phrase.

The Config File Editor offers both a form-based view of the configuration file options, and a text-based editor that can be enabled to edit the configuration file directly. You can switch between the two by using the Configuration File buttons ( ) at the top of the editor window.

Selecting the text-based editor view from time to time is a good way to review the current contents of the config file. You can also quickly edit or add configuration commands as needed.

> **TIP:** *Remember after making changes to the config file, use the* ***File→ Save*** *command, or* `Ctrl-S` *to save your changes. However, you can also enable the* ***File→ Auto Save*** *command to have your work saved automatically.*

Importing Files

If you haven't added the source files when you created the component you can do it after by using the following steps. You can also use this technique to add missing source files or include files required to resolve any build errors and successfully build the AI Engine component.

1. Right click the `sources` folder for importing in the Explorer window, select **Import→ Files/ Folders**
2. Select the a file or folder, or hold **Shift** and select multiple files or folders
3. Click **Open**

> **TIP:** *Imported files and folders are imported directly into the AI Engine component folder in the open workspace in the Vitis IDE.*

If the sources specified in the graph header contain the relative path to the kernel sources, there is no need to add each sub-directory to the config file. If there is no relative path you will need to add the include file manually. In the following code all kernel paths are point relative to the top directory, so adding the kernel folder is sufficient:

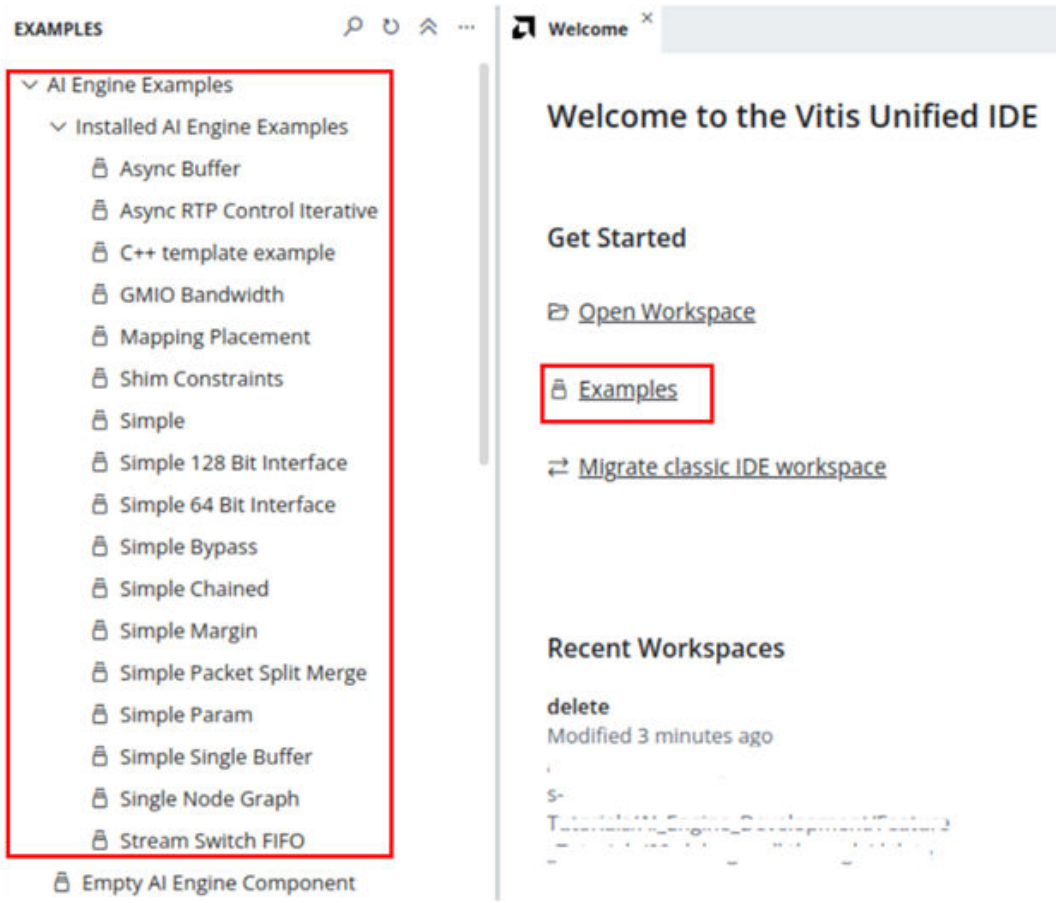
```
adf::source(interpolator) = "kernels/interpolators/hb27_2i.cc";
adf::source(clip) = "kernels/polar_clip.cpp";
adf::source(classify) = "kernels/classifiers/classify.cc";
```

The general recommendation is to add all the sub-directories manually to the config file to ensure completeness. Right-click on any directory under sources in Component Explorer, select **copy path**, and paste it into the config file using the same syntax as existing included files.

Vitis Unified IDE Examples

The Vitis Unified IDE provides examples to get started. Click **Examples** in the Welcome page which allows you to choose from a list of examples available under AI Engine Examples section as shown below.

Figure 84: **AI Engine Examples**



The AI Engine Examples illustrate the basic features of AI Engine programming. You can study these examples, use them as a starting point for your own projects, or mix and match the features to create your own complex computation graphs. The following table describes some of the templates.

Table 45: **Application Template Examples**

Template Name	Description	Further Information
AI Engine, PL and PS System Design	This design demonstrates integrating the AI Engine array with the Programmable Logic and the Processing System in a system. It performs hardware co-simulation and hardware implementation.	Building and Running the System in the *Data Center Acceleration using Vitis* (UG1700)
Async Buffer	A graph to demonstrate asynchronous window APIs.	Asynchronous Buffer Port Access in *AI Engine Kernel and Graph Programming Guide* (UG1079).
Async RTP Control Iterative	A graph to demonstrate simple use of asynchronous RTP update and run with specified test iterations.	Graph Execution Control in *AI Engine Kernel and Graph Programming Guide* (UG1079).
C++ template example	An example demonstrating C++ templated data types and state encapsulation.	C++ Template Support in *AI Engine Kernel and Graph Programming Guide* (UG1079).
GMIO Bandwidth	A graph to demonstrate GMIO performance profiling.	Configuring input_gmio/output_gmio in *AI Engine Kernel and Graph Programming Guide* (UG1079).
Mapping Placement	A templated graph with relocatable mapping and location constraints for kernels.	Location Constraints in *AI Engine Kernel and Graph Programming Guide* (UG1079).
AI Engine Interface Tile (Shim) Constraints	A graph to demonstrate physical channel allocation constraints on the AI Engine to PL interface boundary.	AI Engine/Programmable Logic Integration in *AI Engine Kernel and Graph Programming Guide* (UG1079).
Simple	A simple 2-kernel graph with window based data communication.	Buffer Port-Based Access in *AI Engine Kernel and Graph Programming Guide* (UG1079).
Simple 128 Bit Interface	A graph to demonstrate 128-bit interface between the AI Engine and PL.	Configuring input_plio/output_plio in *AI Engine Kernel and Graph Programming Guide* (UG1079).
Simple 64 Bit Interface	A graph to demonstrate 64-bit interface between the AI Engine and PL.	Configuring input_plio/output_plio in *AI Engine Kernel and Graph Programming Guide* (UG1079).
Simple Bypass	A graph demonstrating the use of bypass for kernels.	Kernel Bypass in *AI Engine Kernel and Graph Programming Guide* (UG1079).
Simple Margin	A graph demonstrating the use of margin in windows (overlapping windows).	Buffer Port-Based Access in *AI Engine Kernel and Graph Programming Guide* (UG1079).
Simple Packet Split Merge	A graph to demonstrate simple split and merge of packet stream data.	Explicit Packet Switching in *AI Engine Kernel and Graph Programming Guide* (UG1079).
Simple Param	A simple 1-kernel graph with scalar parameter update using external trigger.	Specifying Run-Time Data Parameters in *AI Engine Kernel and Graph Programming Guide* (UG1079).
Simple Single Buffer	A graph demonstrating single buffer constraint on connections.	Buffer Allocation Control in *AI Engine Kernel and Graph Programming Guide* (UG1079).
Single Node Graph	A simple single node graph with demonstration window (single buffer and double buffer), stream and RTP array connections.	Single Kernel Development

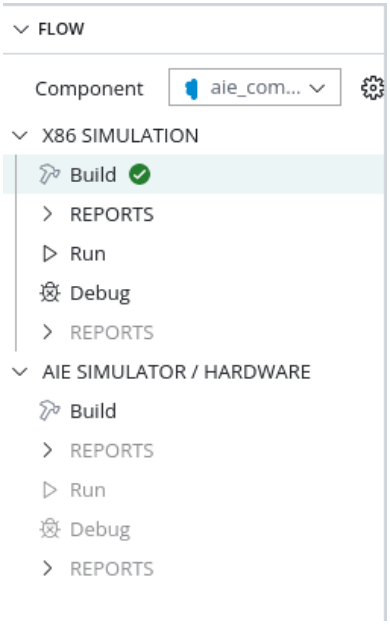
Table 45: **Application Template Examples** *(cont'd)*

Template Name	Description	Further Information
Stream Switch FIFO	A graph to demonstrate use of stream switch FIFO to avoid deadlocks with reconvergent streams.	FIFO Depth in *AI Engine Kernel and Graph Programming Guide* (UG1079).

Building the AI Engine Component

After creating the AI Engine component, you can select the **Component** in the Flow Navigator to make it the active component in the tool. Then select the **Build** command under either the X86 Simulation flow or the AI Engine Simulator/Hardware flow as shown in the following image.

Figure 85: **Build AI Engine Components**



An AI Engine component in the IDE uses the new `v++ --mode aie` feature of the common command-line for compilation as described in v++ Mode AI Engine. The component is run in either `x86simulator` or `aiesimulator` depending on the target of the compilation process, or can be run on the actual hardware device.

The AI Engine compiler supports two build targets:

- The `x86sim` target compiles the code for use in the x86 functional simulator.
- The `hw` target compiles the code and produces a `libadf.a` for use in `aiesimulator`, hardware emulation, or hardware.

The Vitis IDE specifies a work directory (`-work_dir`) as a sub-directory of the AI Engine component in the Workspace used when launching the IDE called `build/<target>/Work`. For example: `--work_dir ./work_space/aie-component/build/hw/Work`. The `build/<target>` and `build/<target>/Work` folders are where the output of the build process can be found. The type of output and contents of the output directory depend on the target specified.

TIP: *The IDE specifies the work directory and ignores any `work_dir` specified in a config file.*

A transcript of the build process is displayed in the Output console, titled with the component name and the build target: for example `aie1::x86sim`. When the build is complete you can review the transcript, or the `AIECompiler.log` file written to the `<component>/build/<target>` folder. You should see `Build Finished` successfully in the transcript, or you could see errors reported instead. The Flow Navigator displays a green circle with a check mark in it, or a red circle with an x depending on the results of the build. If the build completes with errors, review the transcript or the log file to determine what might be the cause and rerun the build as needed.

After the build is complete, you can view the microcode produced by the AI Engine compiler as explained in Viewing AI Engine Microcode. This opens the LST (`.lst`) file which is microcode that corresponds to the kernel code scheduled to be executed on a specific AI Engine tile.

With a successful build the Flow Navigator will display a series of reports generated during the compilation process. You can access the reports by expanding **Reports**. The available reports will vary depending on the build target of `x86sim` or `hw`. You can select any of the available reports to view, or switch to the Analysis view to complete a review of the reports. For details about the available reports, see Chapter 8: Working with the Analysis View (Vitis Analyzer).

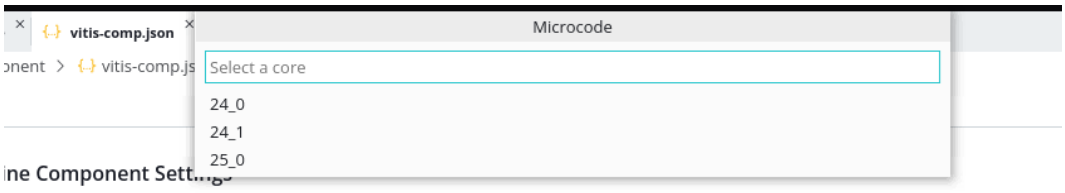
After the build completes successfully you can choose to **Run** or **Debug** the component.

Viewing AI Engine Microcode

After compiling AI Engine source code with a `hw` target, you can view the generated microcode in the Vitis IDE. The microcode can be found in the `<component>/build/<target>/Work/aie/<tile>/Release` directory. Because checking the microcode is a frequently required action, and the generated files are deep in the build hierarchy, the tool makes it easy for you to quickly access the microcode.

Open the AI Engine microcode by right-clicking the component in the Component Explorer view and selecting the **Microcode** command. This displays a Microcode core selector, letting you choose the microcode scheduled to be executed on a specific AI Engine tile.

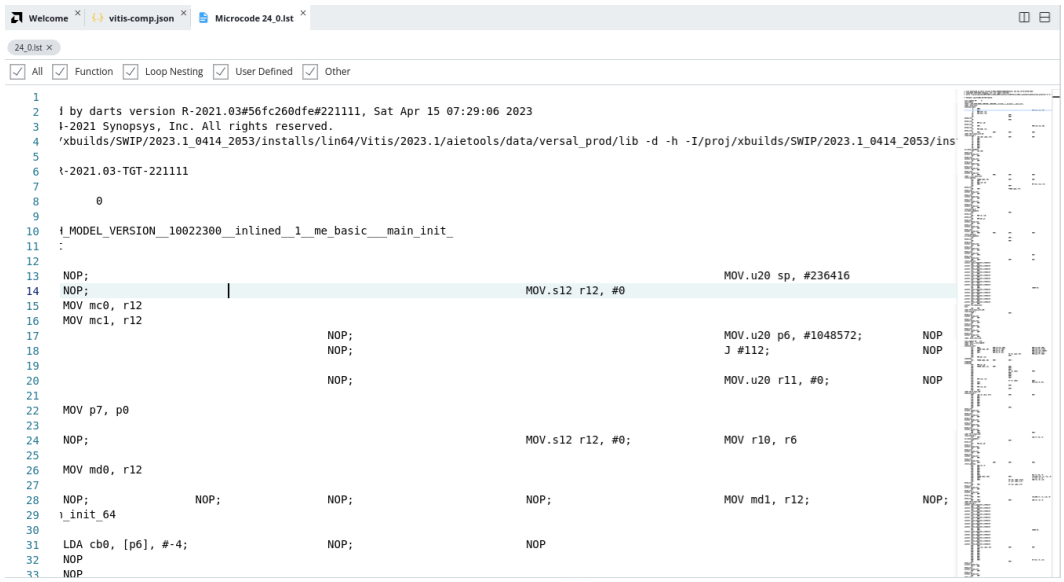
Figure 86: **Select Core to View Microcode**



The Microcode view has the following features:

- Filter instruction lines by enabling/disabling Functions, Loop Nesting, User Defined or Other on the top filter bar
- Cross-probe between the instruction and kernel source code by clicking the lines with blue dots next to the line numbers
- Examine the number of cycles taken by specific lines of kernel code and see where optimizations can be made
- Quickly navigate through the microcode using the global view displayed on the right
- Right-click menu provides access to additional commands

Figure 87: **Microcode View**



Managing AI Engine Components with the Python Command Line Interface

The supports Python APIs to automate the management of AI Engine components. There are two ways to use the Python APIs:

1. Use the Python APIs as commands directly in the interactive mode.

   To launch the Vitis Unified IDE in interactive mode, use the following command:

   ```
   vitis -i
   ```

   The command is executed as follows:

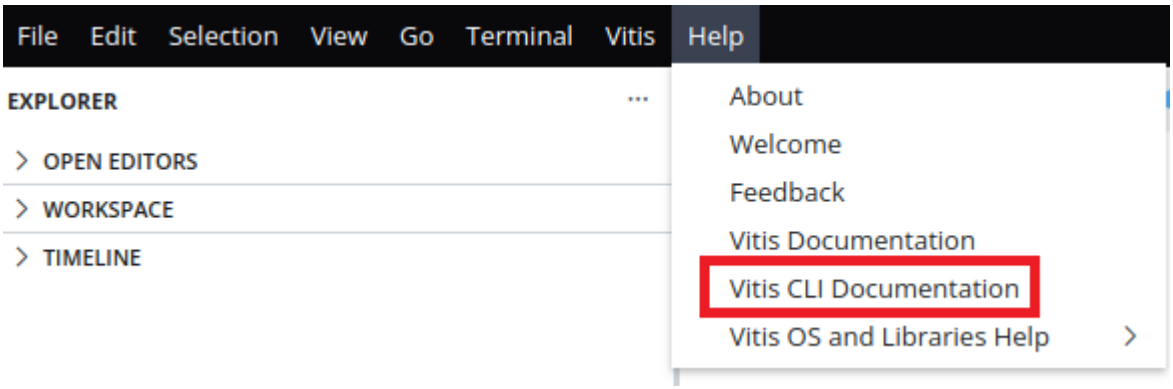
   ```
   bash-4.2$ vitis -i

   ****** Vitis Development Environment
   ****** Vitis v2025.2 (64-bit)

   Welcome to Vitis Python Shell
   Python CLI API documentation is available
   at
   <vitis_install_area>/cli/api_docs/build/html/
   index.html
   Journal file generates at <current_working_directory/
   vitis_journal.py>
   Do not use the file until the session is closed.
   Vitis [1]:
   ```

   Python API commands can now be used to manage the AI Engine components.

   The Python CLI API documentation (`index.html`) is available in the initial log of the Vitis interactive command. The documentation can be accessed through the Vitis Unified IDE also. In the Vitis Unified IDE, select **Help → Vitis CLI Documentation**.

   For more information about the interactive mode, see Vitis Interactive Python Shell.

2. Run a Python script (`.py`), which contains the Python APIs, in one of two ways:

   - With `vitis -i` which launches the Vitis IDE interactive mode. For example, `vitis [1]: run <.py>`.
   - With `vitis -s` option which executes the provided Python script. For example, `vitis -s <.py>`.

Using Python APIs to Manage AI Engine Components

The Vitis IDE works on the server-client architecture. To create any component you must first create a client object. The component can be created using the `create_aie_component` API of the client. For more information on using the Python command line interface, see Vitis Interactive Python Shell.

The general steps to manage AI Engine component are as follows:

1. Create AI Engine Components.

   Use the `create_aie_component` Python API to create AI Engine components. For example:

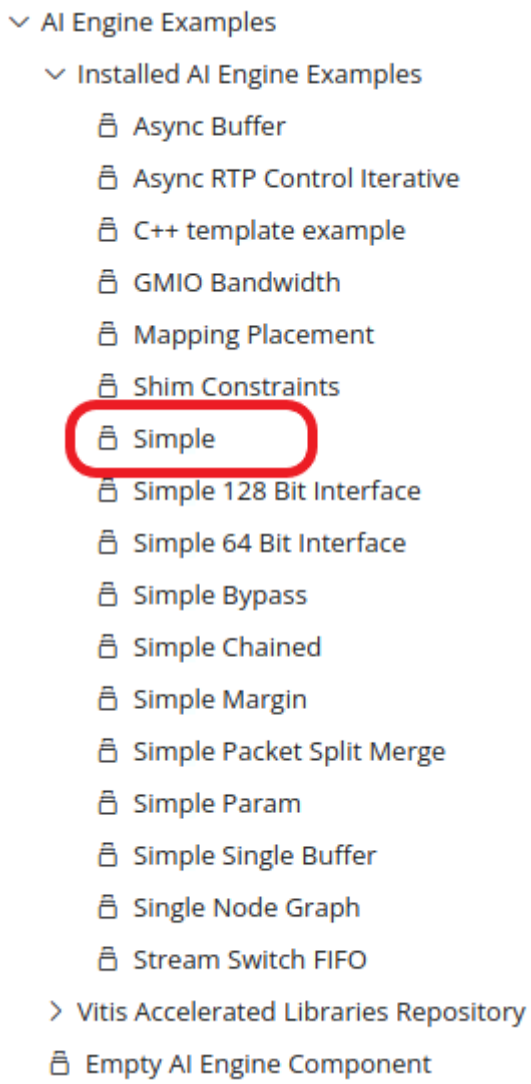
   ```
   aie_test_comp = client.create_aie_component(name = aie_component,
   platform=<abs path of xilinx_vck190_base_202420_1.xpfm>,
   template="empty_aie_component")
   ```

   Arguments for the API are as follows:

   - **`name=<aie_component_name>`:** Name of the AI Engine component.
   - **`platform=<.xpfm> or <.xsa>`:** The hardware design used to compile the AI Engine graph, alternatively the device part is also supported.
   - **`template= "empty_aie_component"`:** You can use a component example from the Installed AI Engine Examples as the template. The `template` value can be defined as `"empty"` (default), `"empty_aie_component"`, or the location and name of the Installed AI Engine Example you wish to use, such as `"installed_aie_examples/simple"`. For values of `"empty"` and `"empty_aie_component"`, the AI Engine component will be created from an empty template.

     ***Note*:** Double quotes (`" "`) are required when defining a template value.

     For example, if the AI Engine Example **Simple** is the template you wish to start with, as shown below:



The Python API command is as follows:

```
aie_test_comp = client.create_aie_component(name = aie_component,
platform=<abs path of xilinx_vck190_base_202420_1.xpfm>,
template="installed_aie_examples/simple")
```

2. Import source files into an AI Engine component.

   Use the `import_files` Python API to import the source files. For example:

   ```
   aie_test_comp.import_files(from_loc = sources_dir, files = ['graph.cpp',
   'graph.h', 'kernels.h', 'include.h', 'classify.cc', 'hb27_2i.cc'])
   ```

   Arguments for the API are as follows:

   - **`from_loc=<location of source or configuration file>`:** The path to the location of the source files can be an absolute path or a relative path to the current working directory. The location can be a directory or a file.

- **`files=['file1','file2',...]`**: List of files to be imported from the `from_loc` path.

3. Build AI Engine components.

   Use the `build` Python API to build AI Engine components. The API supports two targets: `hw`, and `x86sim`.

   - To build the component for AI Engine simulator for hardware, use:

     ```
     aie_test_comp.build(target="hw")
     ```

   - To build the component for x86 simulator, use:

     ```
     aie_test_comp.build(target="x86sim")
     ```

   The default target is `x86sim`.

Using APIs in a Python Script

The Python package `vitis` which includes all Python APIs must be added in the Python script. The example below uses the Python APIs in a Python script to manage an AI Engine component:

```
# Add package: Vitis Python CLI
import vitis

# Create a Vitis client object -
client = vitis.create_client()

# Set Vitis Workspace
client.set_workspace(path=<workspace_location>)

# Create aie component.
aie_test_comp = client.create_aie_component(name = comp_name, platform =
<absolute path of platform>, template = "empty_aie_component")

# Import source files to the component
aie_test_comp.import_files(from_loc = <absolute path of source folder>,
files = ['graph.cpp', 'graph.h', 'kernels.h', 'include.h', 'classify.cc',
'hb27_2i.cc'])

# Set top file from the imported source
aie_test_comp.update_top_level_file(top_level_file = 'graph.cpp')

# Build component on target Hardware
aie_test_comp.build(target="hw")

# Print component information
aie_test_comp.report()
```

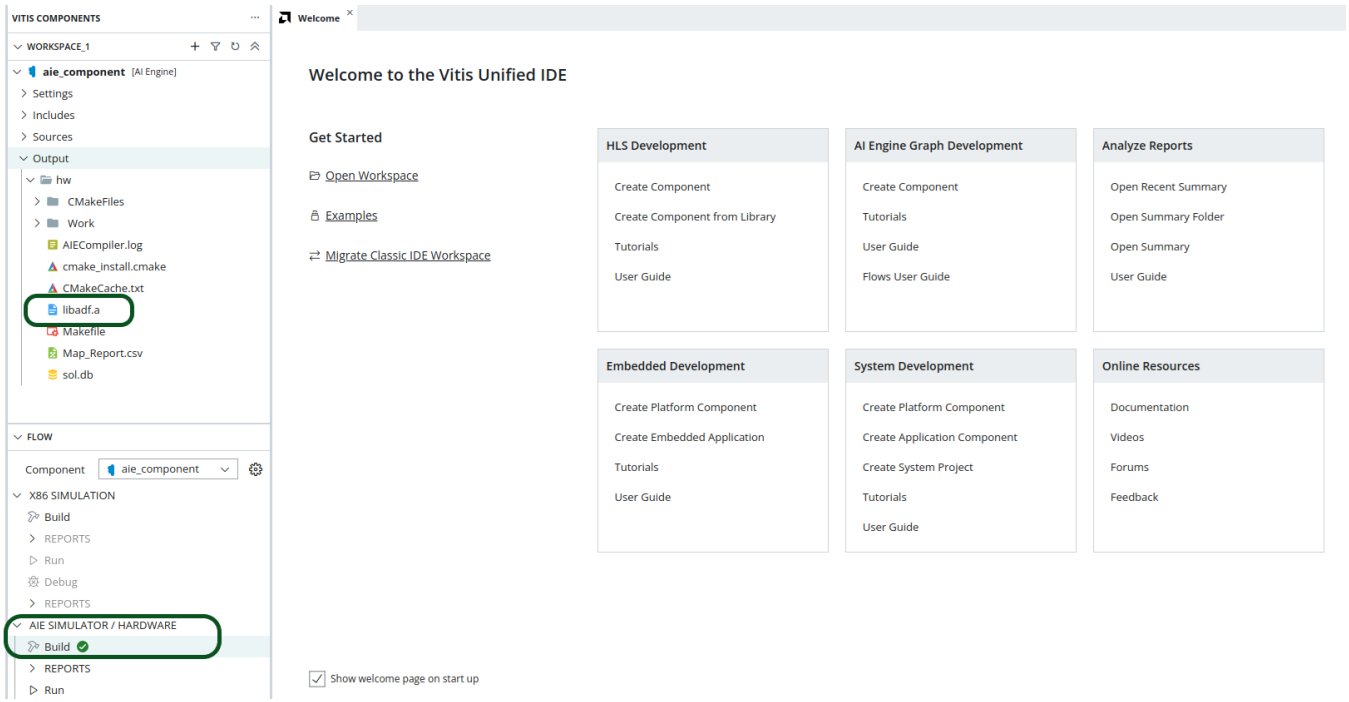
The following command launches the Vitis IDE, and executes the python script:

```
vitis -s aie_comp.py
```

The `vitis -s` command manages the Vitis component (such as, creation, build, editing configuration files, etc.). Following that, you can launch the Vitis IDE and display the AI Engine components in the workspace using the `vitis -w` command.

```
vitis -w <workspace_location>
```

The AI Engine component is created, the `libadf.a` is generated, and the component build is completed successfully.

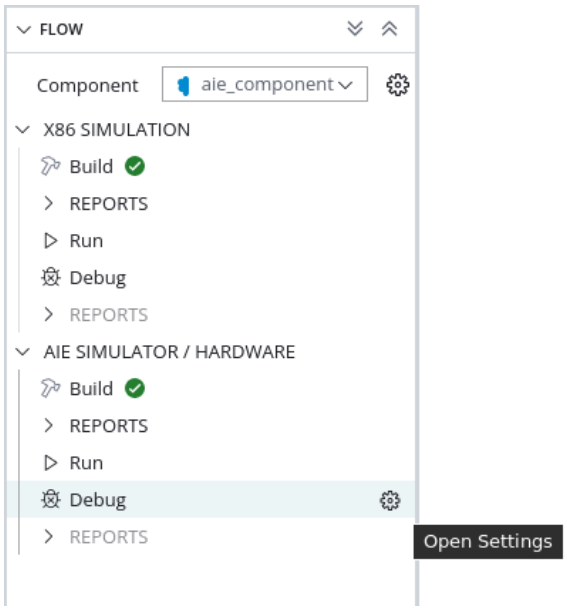


To simulate and debug the AI Engine component, refer to the next section.

Simulating and Debugging the AI Engine Component

In the Flow Navigator, after the AI Engine component has been successfully built, you can run or debug the component under the X86 simulation or AI Engine Simulator/Hardware build targets as shown below.

Figure 88: **Debug AI Engine**



TIP: *If multiple components are listed in the Flow Navigator drop-down, you must select the AI Engine component to make it active in the tool.*

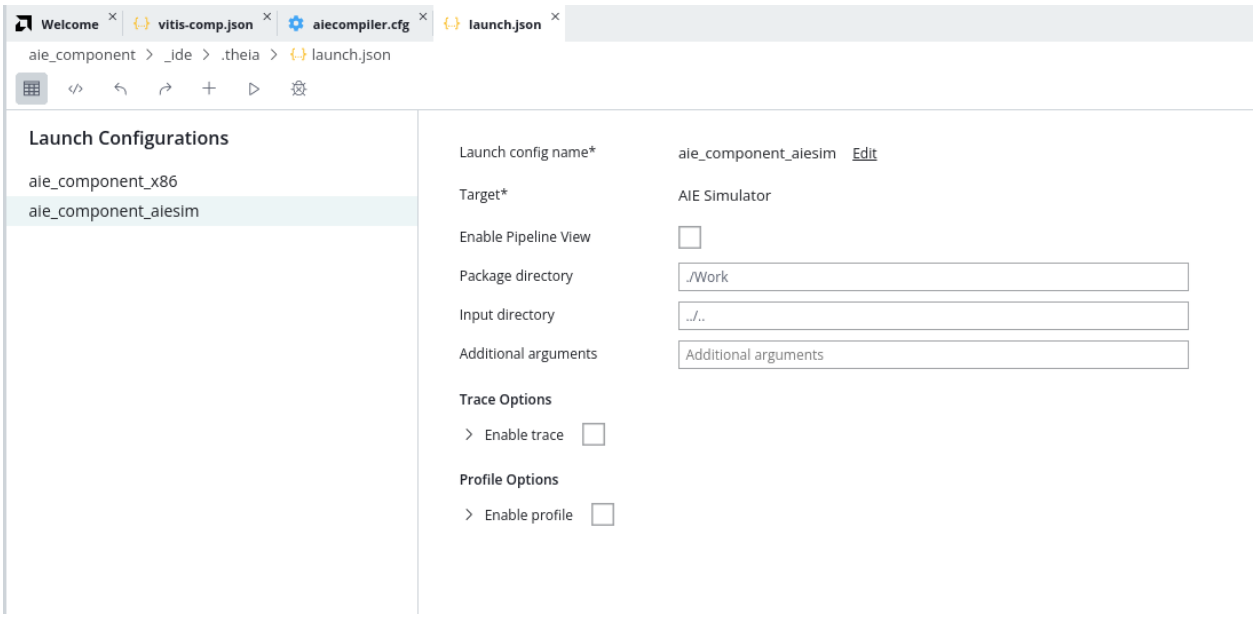
In the preceding figure, the Open Settings button can be used to open a launch configuration for running or debugging the AI Engine component. This button only appears when hovering above the Run or Debug commands in Flow Navigator. Clicking **Open Settings** opens the `launch.json` containing the different launch configurations for the component as described below. You can also choose to run or debug a build target without opening the `launch.json` file by simply selecting **Run** or **Debug** and using the default launch configuration.

Launch configurations are shown in the `launch.json` image below:

- `aie_component_x86`: Supports running or debugging the **x86 Simulation** build
- `aie_component_aiesim`: Supports running or debugging the **AI Engine Simulator/ Hardware** build. This is the launch configuration currently selected and displayed

TIP: *You can create new launch configurations by selecting the **Duplicate** command next to an existing launch configuration, and delete existing configurations by selecting the **Delete** command.*

Figure 89: **AI Engine Launch Configuration**



The image above displays the details of the currently selected launch configuration with the following options:

- Launch config name: Specifies the name. Select the **Edit** command to change the name.
- Target: Specifies the name of the build target used by the launch configuration. The target cannot be changed.
- Enable Pipeline View: Generates the Pipeline view of the AI Engine array during run or debug.
- Package directory: Specifies the directory to use when packaging the AI Engine component. The default directory is the `./Work` directory, but might need to be changed if a different work directory (`-workdir`) was specified during the build.
- Input directory: Specifies the location of any input data used by the component during run or debug.
- Additional arguments: Specifies any additional command line arguments needed by the AI Engine component during run or debug.
- Trace Options: Enables trace capture during run or debug, and specifies options for trace. Refer to Enabling Trace in AI Engine Based Simulation for additional information.
- Profile Options: Enables profiling during run or debug, and specifies options for profile. Refer to Enabling AI Engine Profile for details on the options.

After setting up the launch configuration, you can launch run or debug by selecting the **Run** or **Debug** commands in the `launch.json`, or by selecting **Run** or **Debug** from the Flow Navigator and specifying a launch configuration if more than one is available for the build target.

> **TIP:** *If there are unsaved changes to the launch configuration the tool will prompt you to save prior to running.*

During the run process the output of `x86simulator` or `aiesimulator` is reported to the console. You can review the transcript in real time, or refer to the log file generated after the run is complete. You can open the log file for the simulator by navigating to the component `Output/hw` or `Output/x86sim` folders in the Component Explorer view.

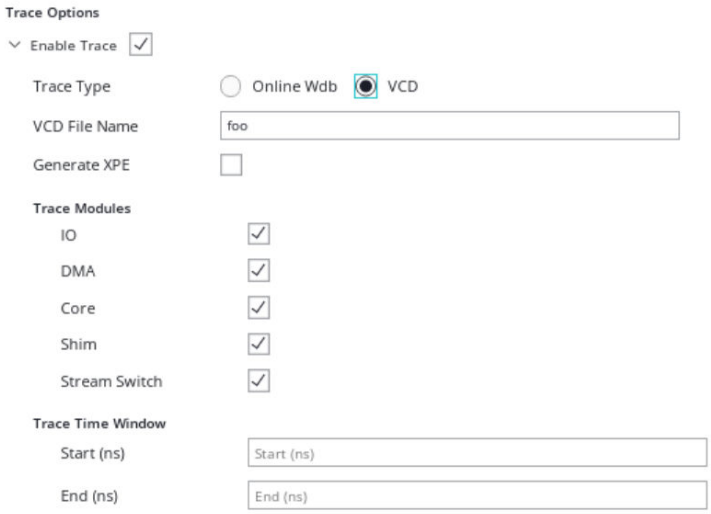
During the debug process, the tool opens the Debug view as explained in Debugging an AI Engine Component.

After successfully running or debugging the AI Engine component, the Flow Navigator will display a series of reports generated during the process. The available reports will vary depending on the build target of `x86` or `hw`. You can select any of the available reports to view, or switch to the Analysis view to complete a review of the reports. For details about the available reports, see Chapter 8: Working with the Analysis View (Vitis Analyzer).

Enabling Trace in AI Engine Based Simulation

The AI Engine architecture has direct support for generation, collection, and streaming of events as trace data during simulation, hardware emulation, or hardware execution. For the AI Engine component `aiesim` launch configuration, you can enable trace and select trace options as shown below.

Figure 90: **AI Engine Launch Configuration - Enable Trace**



After selecting the Enable Trace check box, you can then specify trace options:

- Trace type: Specify the format of the trace data as either online waveform database (WDB) or value change dump (VCD) format. The online WDB file is generated on-the fly to display in the Analysis view. The VCD can contains a detailed dump of the changing hardware signals in the form of value change dump (VCD) files which must be post-processed. After simulation, or emulation, the output file can be processed into events and viewed on a timeline in the Analysis view. The events contain information such as time stamps, different event types, and data associated with each event. This information can also be correlated to the compiler generated debug information.
- Trace Modules: You can enable the capture of trace data from different elements of the AI Engine component as shown above. Enable the specific modules of interest, or all modules.

Using command-line interface you can also generate a trace file using options of the `aiesimulator`:

```
aiesimulator --pkg-dir=./Work --dump-vcd=filename
```

The `--dump-vcd=filename` allows you to have access to a subset of the events by default. I you want another subset, you need to add an option file: `--options-file=filename`.

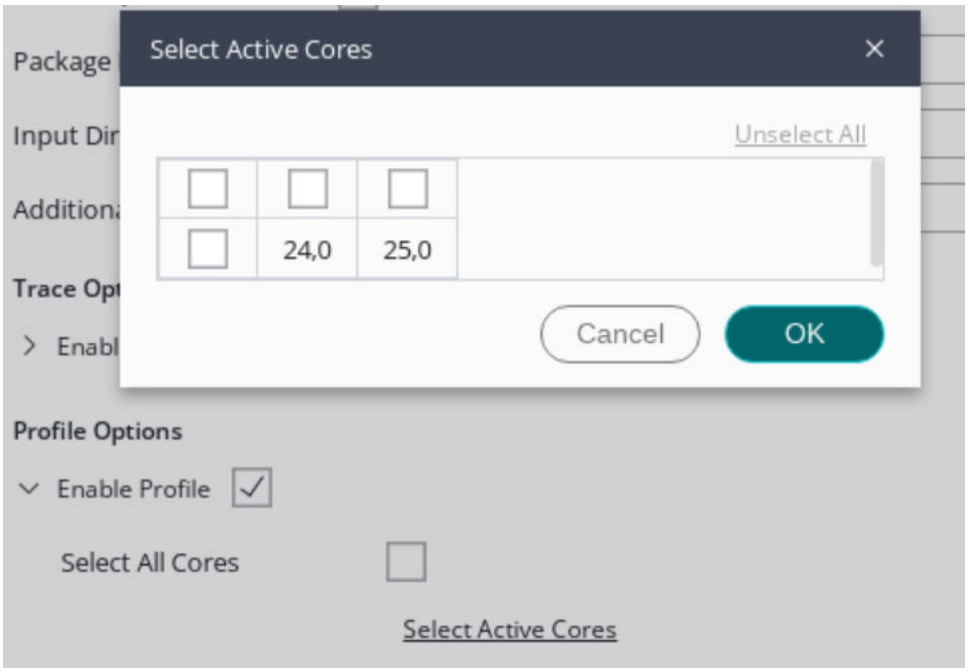
This option file is a text file that contains a description of the signals that you want to capture:

```
AIE_DUMP_VCD_IO=true/false
AIE_DUMP_VCD_CORE=true/false
AIE_DUMP_VCD_SHIM=true/false
AIE_DUMP_VCD_MEM=true/false
AIE_DUMP_VCD_STREAM_SWITCH=true/false
AIE_DUMP_VCD_CLK=true/false
```

Enabling AI Engine Profile

For the AI Engine component `aiesim` launch configurations you can enable profiling and select profile options as shown below. Profiling allows you to collect data on design latency, throughput, and bandwidth. Analyzing profiling data helps you gauge the efficiency of the kernels, the stall and active times associated with each AI Engine tile, and pinpoint kernels with performance issues.

Figure 91: **AI Engine Component Launch Configuration - Enable Profile**



Select the **Enable profile** checkbox and select profiling for all cores.

- All: Profile all the tiles used in the `libadf.a`.
- Custom: Displays the Select Active Cores dialog as displayed above, letting you select individual cores, or select by rows or columns of tiles to profile.

Captured profiling data can be viewed in the Analysis view.

Extracting Throughput and Latency Estimates using Python CLI

Average throughput for all the PLIOs are calculated and displayed at the end of AI Engine simulation. If you generate a VCD file during the simulation, it can also provide estimates for continuous throughput and latency.

Vitis IDE offers Python based APIs to extract these estimates. The following python code extracts throughput and latency including continuous throughput and latency into a csv file.

See Python API: Managing Vitis IDE Components for more details on using the Python CLI to create and manage components in the IDE.

See Latency and Throughput Estimates in the *AI Engine Tools and Flows User Guide* (UG1076) for more details on throughput and latency estimates after AI Engine simulation.

```
#vitis -s <this python file>
import vitis

#Create Vitis server for python front end
client = vitis.create_client()

#initialize vitis_analyzer summary obj
summary = client.get_vitis_analyzer("filter_latency/aiesimulator_output/
default.aierun_summary")

#export Latency table as csv file
summary.export_aiesim_latency("latency.csv", overwrite=True)

#export Throughput shown in I/O and ports table
summary.export_aiesim_throughput("throughput.csv")

#export continuous throughput for each graph iteration for I/O port
summary.export_aiesim_continuous_throughput("PLIO_i_0", 4, "plio1.csv",
overwrite=True,     is_cycle_interval=False,
is_equal_time_interval=False,is_graph_iteration=True)

#export continuous throughput for a Kernel port
summary.export_aiesim_continuous_throughput("aie_dut.filter.filter_kernel_in
s/sig_i", 4, "sig_i_throughput.csv", overwrite=True,
is_cycle_interval=False, is_equal_time_interval=False,
is_graph_iteration=True)

#export continuous latency for a Kernel port pair
summary.export_aiesim_continuous_latency("aie_dut.filter.filter_kernel_ins/
sig_o", "aie_dut.filter.filter_kernel_ins/sig_i", 4, "sig_i_latency.csv",
overwrite=True, is_cycle_interval=False)

plotdata = []
summary.get_aiesim_continuous_throughput("PLIO_i_0", 10, plotdata, True)
summary.get_aiesim_continuous_latency("PLIO_o_0", "PLIO_i_0",  10,
plotdata, False)

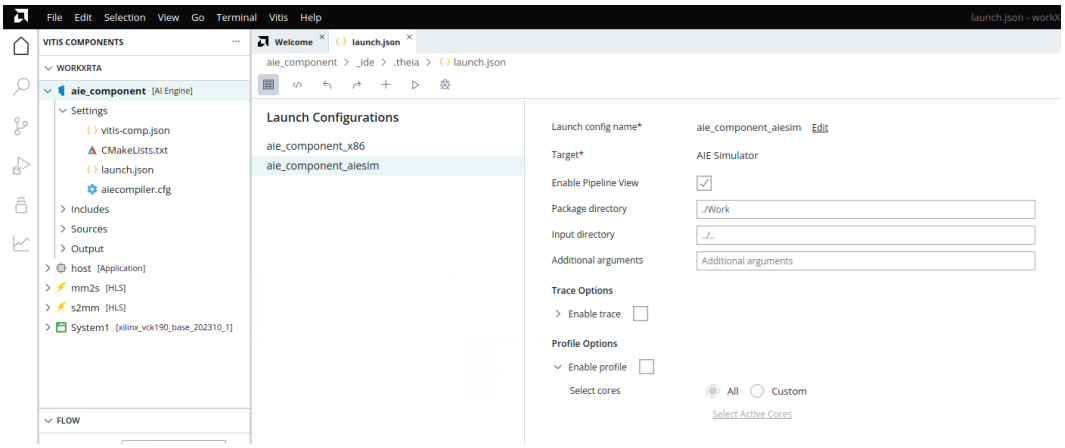
summary.get_aiesim_continuous_latency("aie_dut.filter.filter_kernel_ins/
sig_o","aie_dut.filter.filter_kernel_ins/sig_i",  10,  plotdata, False)

# Close the client connection and terminate the vitis server
vitis.dispose()
```

Enabling AI Engine Pipeline

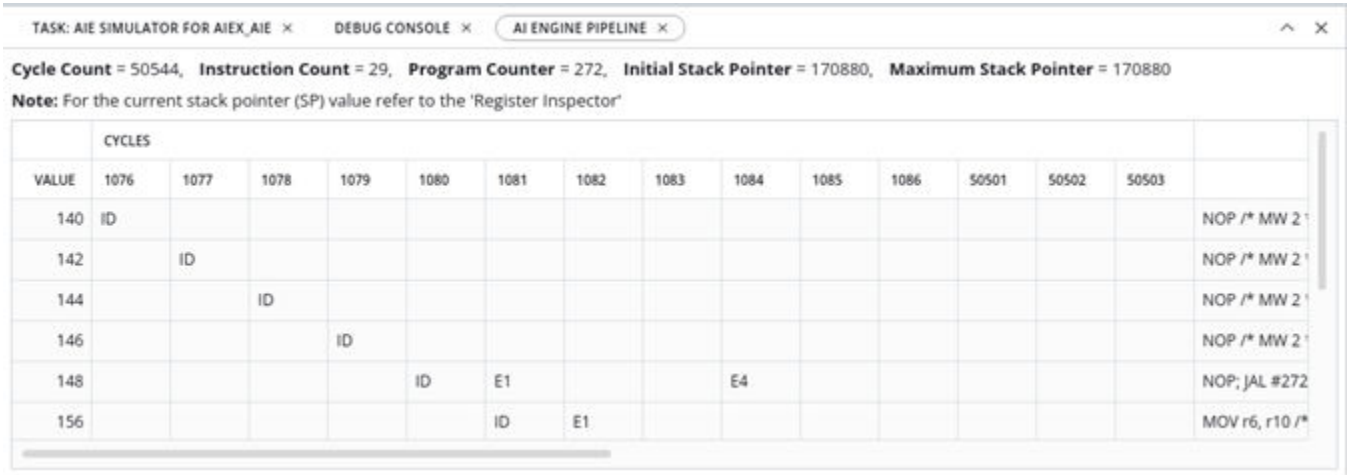
The AI Engine Pipeline view can provide instruction pipeline details. To view AI Engine Pipeline view contents for a core, follow the instructions below.

1. In the Component Explorer, right click on the AI Engine component to expand the Settings folder.
2. Select the `launch.json` to open the Launch Configurations view.
3. In the Launch Configurations view, open or create an AI Engine `AIESIM` launch configuration. To create one, click the '+' command in the toolbar menu.



4. Select the **Enable Pipeline View** checkbox to enable the feature.

When the AI Engine simulator (`AIESIM`) is run, the Pipeline view is opened, as shown below.

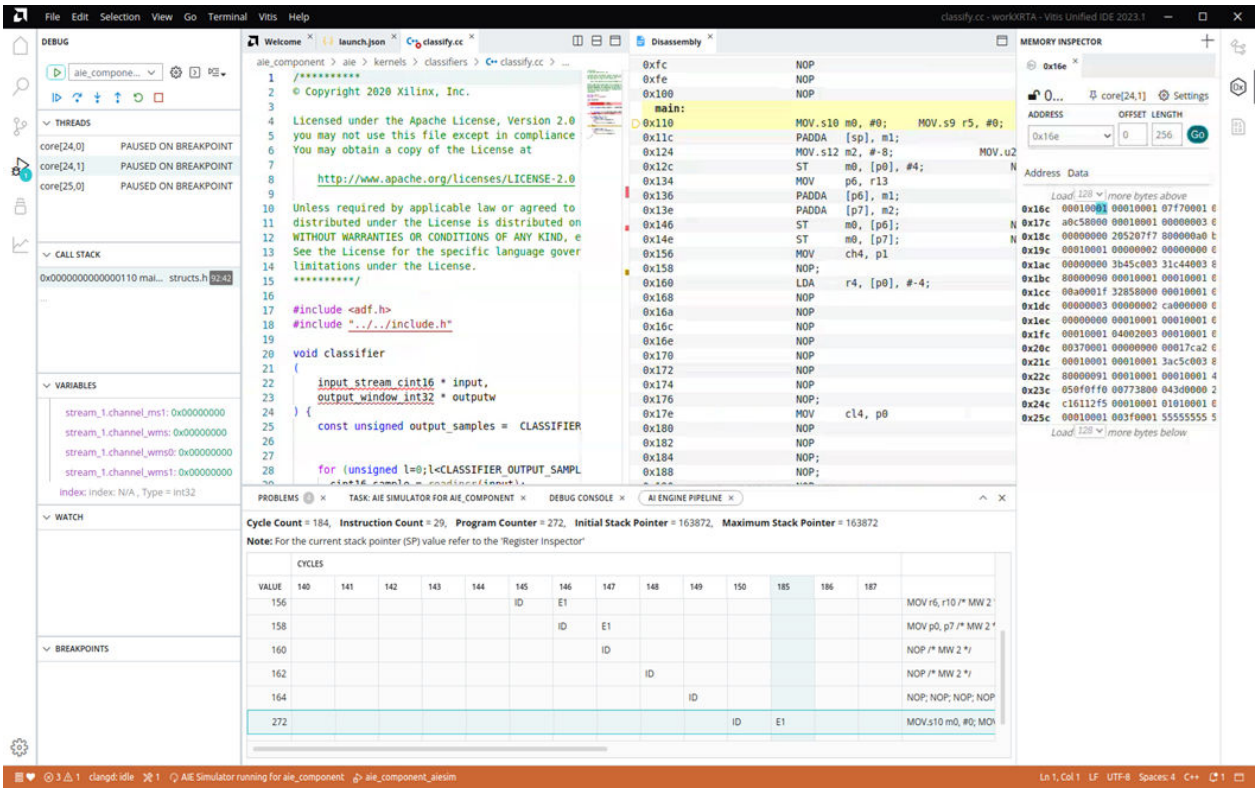


The Pipeline view displays the program counter vertically and the kernel cycle count horizontally. ID represents instruction decode, and E1-E7 are execution stages. Almost all operations in the scalar unit are scheduled in E1 stage of the pipeline besides non-linear operations. The vector unit scheduling spans from the ID stage to the E6 stage. Address Generation Units (AGUs) span over two pipeline stages. The address is ready in the E2 stage of the pipeline. For load units, the data will be available in the AI Engine from the memory module in the E7 stage. For the store unit, the data will be sent out from the AI Engine to the memory module in the E5 or E6 stage of the pipeline depending on the type of instruction.

Debugging an AI Engine Component

After you have launched the Debug view for an AI Engine Component, you will see several windows or views displayed as shown in the following figure. The Debug view shows the state of cores that are being debugged. It shows the source file and line number where the debugger stops and what action it is taking (such as breakpoint or step over).

Figure 92: **Debug View - AI Engine Component**

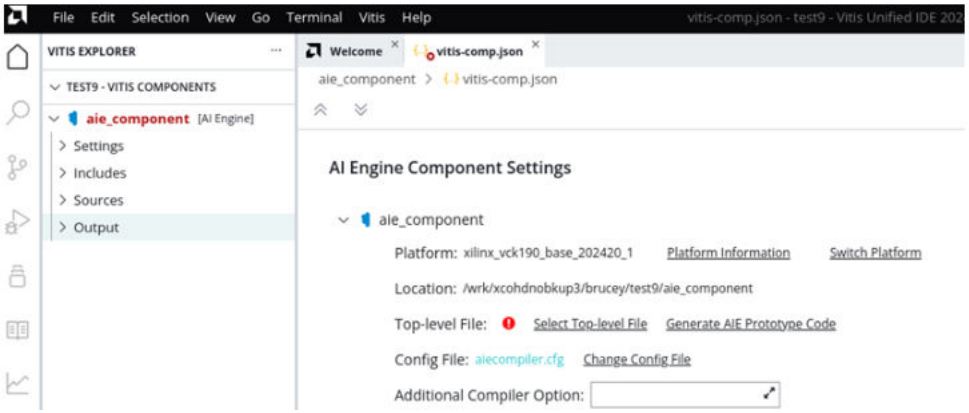


For more information about the Debug View, see Using the Debug Environment.

Generating AI Engine Prototype Code

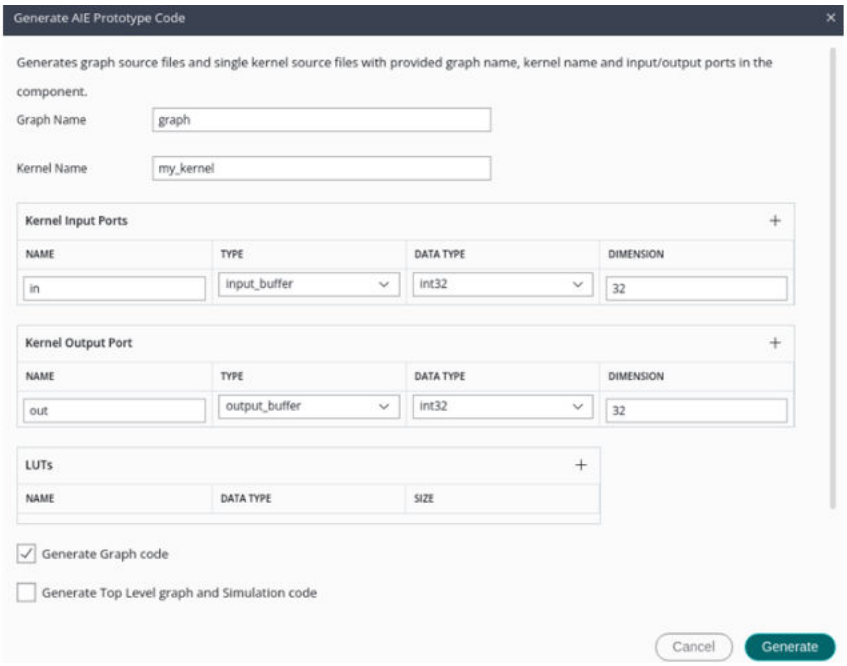
Vitis Unified IDE provides a wizard to generate a single kernel graph prototype code. After an AI Engine component is created, open the **AI Engine Component Settings** on the AI Engine component and click on **Generate AIE Prototype Code**:

Figure 93: **Generate AIE Prototype Code**



In the Generate AIE Prototype Code wizard, you can specify the graph and kernel names, choose the input and output interface and data types, in addition to selecting other additional options.

Figure 94: **Generate AIE Prototype Code Options**



Here are the list of the options:

- Graph Name: name for the generated graph
- Kernel Name: name for the generated kernel
- Kernel Input Ports:
  - NAME: the input interface port name
  - TYPE: the input interface port type

    - DATA TYPE: the input interface port data type
    - DIMENSION: the input interface port dimension which is the size of the port
- Kernel Output Port:
    - NAME: the output interface port name
    - TYPE: the output interface port type
    - DATA TYPE: the output interface port data type
    - DIMENSION: the output interface port dimension which is the size of the port
- LUTs: Option to add a look up table, its data type and size. Refer to Lookup Tables in the *AI Engine Kernel and Graph Programming Guide* (UG1079) for the concepts and usage of LUT.
- An option to Generate Graph code: If checked, the graph code is generated. This graph can be instantiated by other top-level graphs. If not checked, no graph code is generated.
- An option to Generate Top Level graph and Simulation code: If checked, the top-level graph is generated. The top-level graph contain instances of `input_plio`, and `output_plio`. `graph.cpp` code is also generated. The generated top-level graph can be compiled directly. If not checked, you need to add top-level graph which includes the kernel.

Single Kernel Development

To achieve the highest performance on the AI Engine, the primary goal of single kernel programming is to ensure that the use of the vector processor approaches its theoretical maximum. Vectorization of the algorithm is important, but managing the vector registers, memory access, and software pipelining are also required. Because the vector processor is capable of an operation every clock cycle, the programmer must strive to make the data for the next operation load during the current operation. When implementing an algorithm for the AI Engine, it is important to start vectorization based on the data types and the vector intrinsic functions that operate on those data types. Depending on the data type, the various intrinsic functions operate on two or more elements at the same time. When the inner loop has sequential or loop carried dependencies it might be possible to unroll an outer loop and compute multiple values in parallel. There are many creative ways to use the vector intrinsic functions to solve problems. When implementing an algorithm for an AMD Versal™ Adaptive SoC, it is important to understand what the AI Engine does well and what can be better implemented in the other domains, for example, the Processing System, Programmable Logic and DSP engines.

To support AI Engine single kernel development, the Vitis IDE supports AI Engine kernel development in addition to traditional processor support. The Vitis IDE provides a single node graph example that can be used as a starting point for single kernel development. The Vitis IDE has a debug view which displays registers, variables, available breakpoints, variables to register/memory mapping, internal/external memory contents, and an instruction pipeline (pipeline view) for each individual kernel.

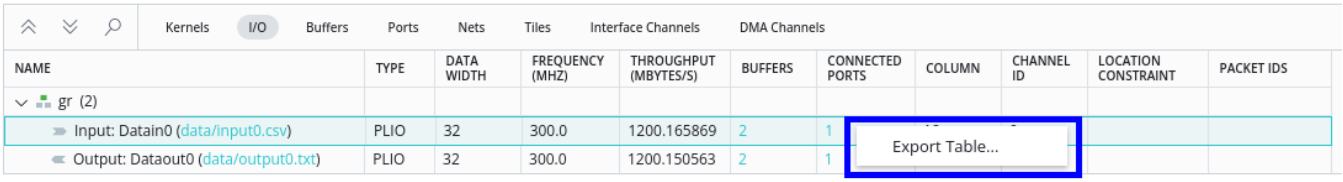
TIP: *Clock cycle count reports are generated with the* `--profile` *option enabled after the emulation run.*

Exporting Summary Tables from the Analysis View

The compile summary and the run summary reports are generated during AI Engine compilation and simulation respectively. The compilation summary table and run summary table are generated in the Analysis view when these reports are opened in the Vitis IDE. The Analysis tool provides the ability to export summary tables to a `.csv` file. Only `.csv` files are supported. The table from any tab (such as, Buffers tab, Ports tab, Nets tab, or Tiles tab) in the Analysis view can be exported.

To export a table from the Analysis view, right-click on the table and select **Export Table**.

Figure 95: **Export Summary Table from the Analysis View**



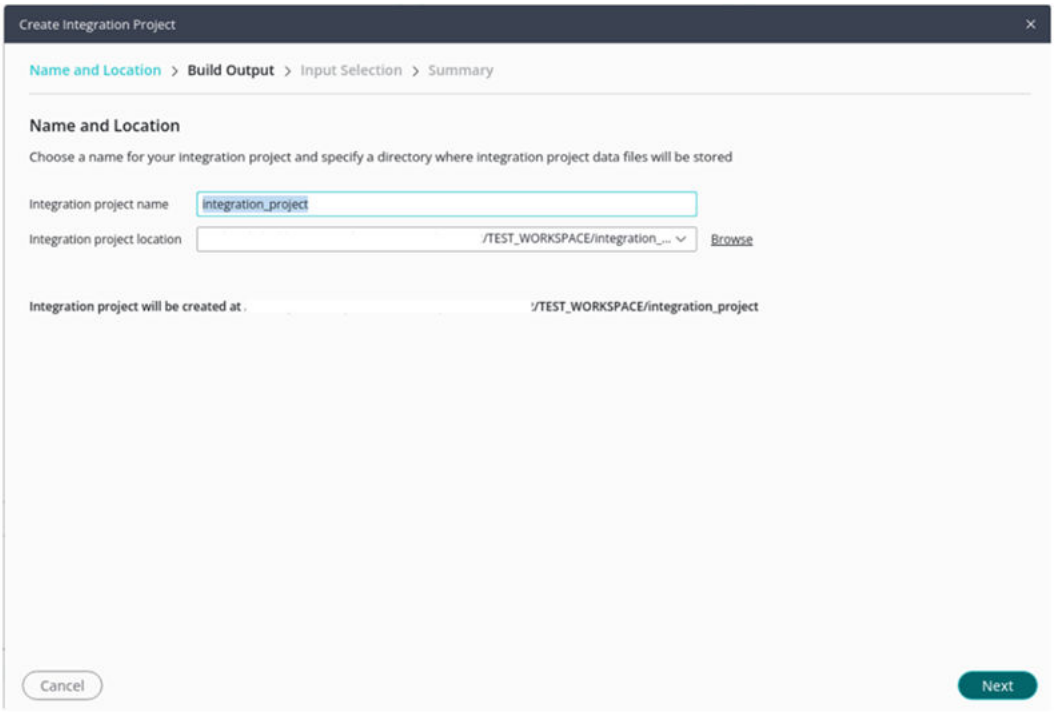
Managing the Integration Project Component in the Vitis Unified IDE

The Vitis Unified IDE supports a component type called Integration Project, designed to decouple the linking and packaging steps in system-level hardware development. This enhancement aligns the IDE workflow with modular development practices and the command-line V++ flow, providing greater flexibility and control for users.

Creating an Integration Project Component

1. With the Vitis Unified IDE, from the main menu, select **File→ New Component→ Integration Project**. This opens the Integration project name and Integration project location page of the Create Integration Project Component wizard as shown below.

Figure 96: **Create Integration Project**



2. Select either Fixed XSA, VMA or VSS build output. The flow diagram shows all possible inputs with the dashed line indicating optional inputs. Next to the flow diagram, the input combinations link shows possible input combinations for the selected build output. There is also link to respective sections of the *Embedded Design Development Using Vitis* (UG1701).

Figure 97: **Fixed XSA Build Output**

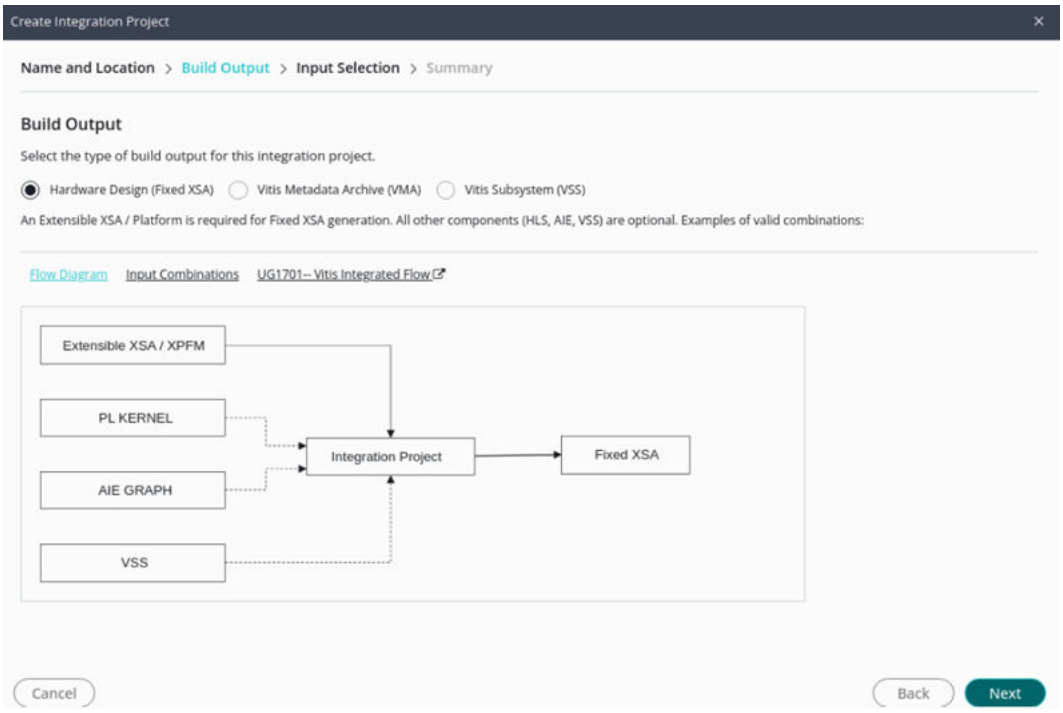


Figure 98: **Metadata Archive (VMA) Build Output**

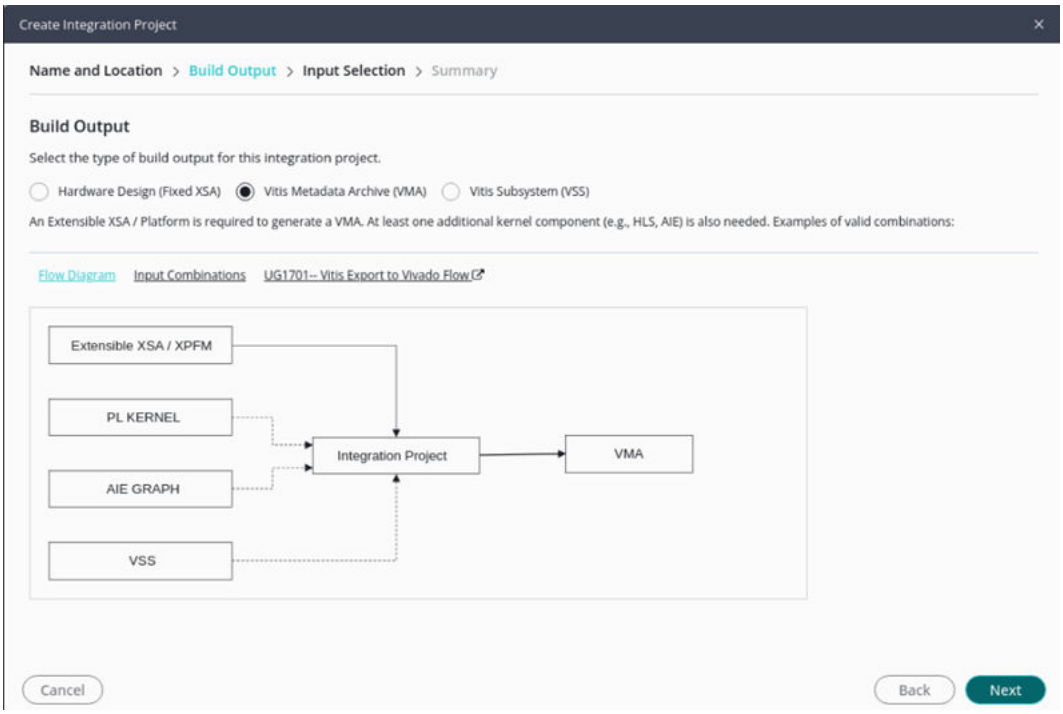
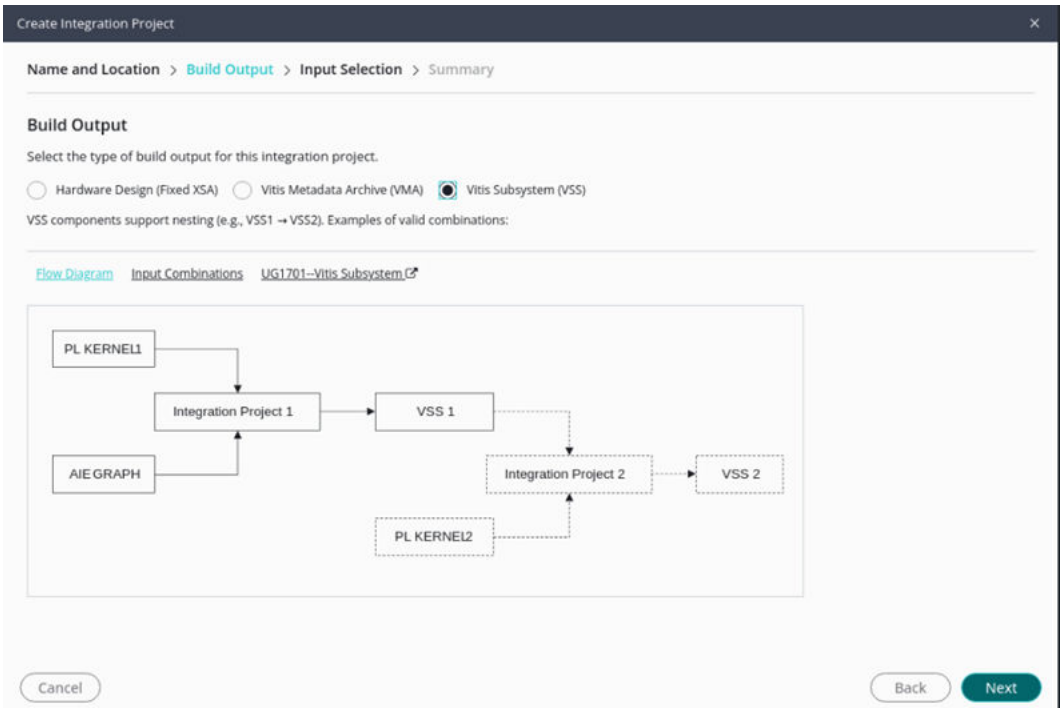


Figure 99: **Vitis Subsystem (VSS) Build Output**



3. In the next step, add the required inputs and one or more optional inputs.

Figure 100: **Input Selection for Fixed XSA Built Output**

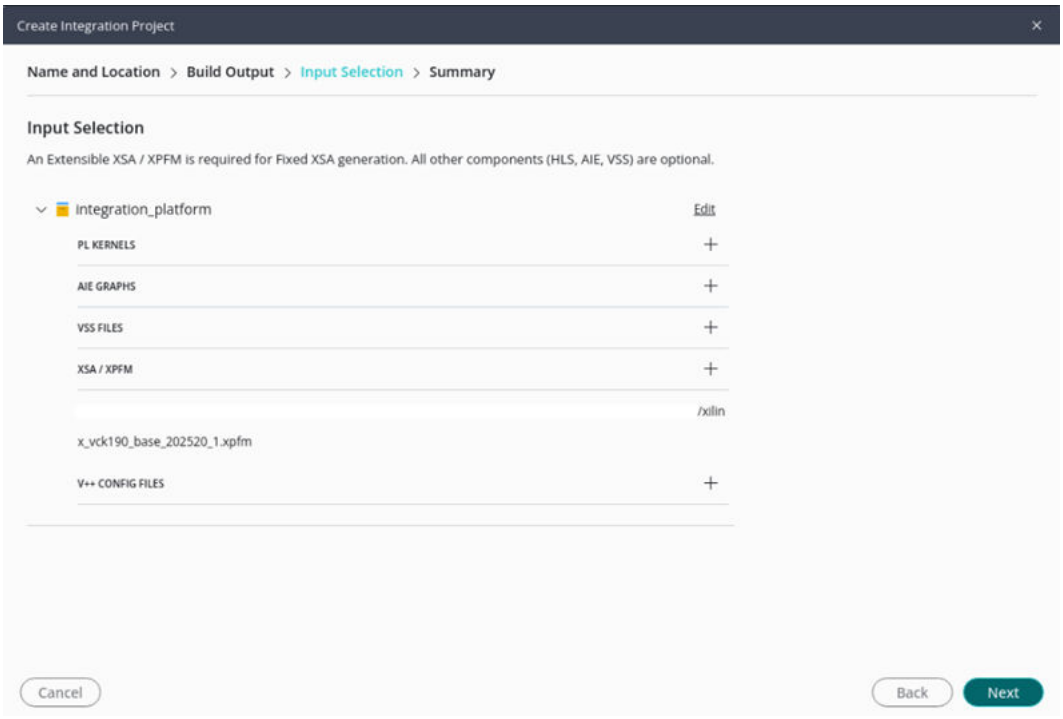


Figure 101: **Input Selection for VMA Build Output**

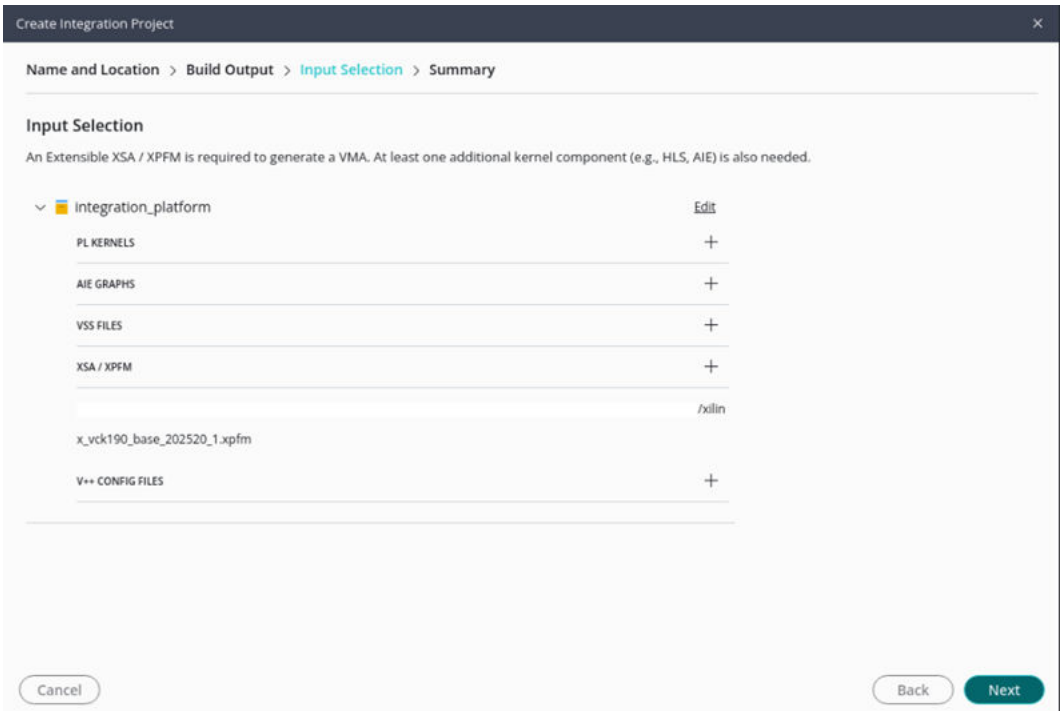


Figure 102: **Input Selection for VSS Build Output**

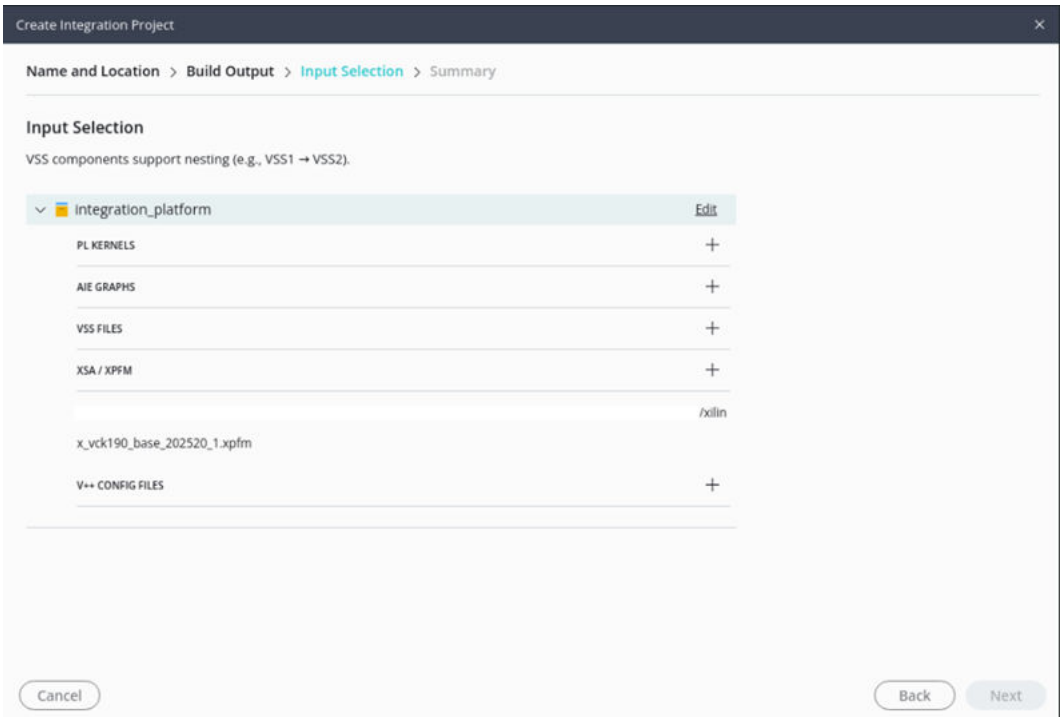
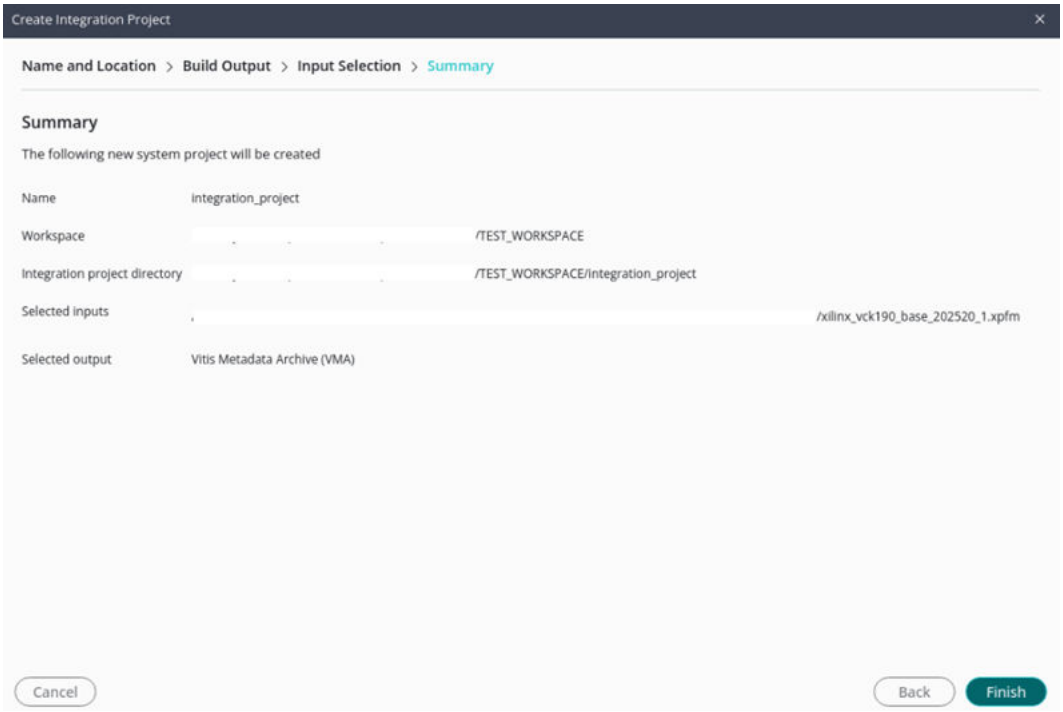


Figure 103: **Integration Project Component Summary**



4. In the final step, the wizard shows the summary before generating the component.

Possible Input Combinations for the Integration Project

The following table lists all possible input combinations for the selected build output.

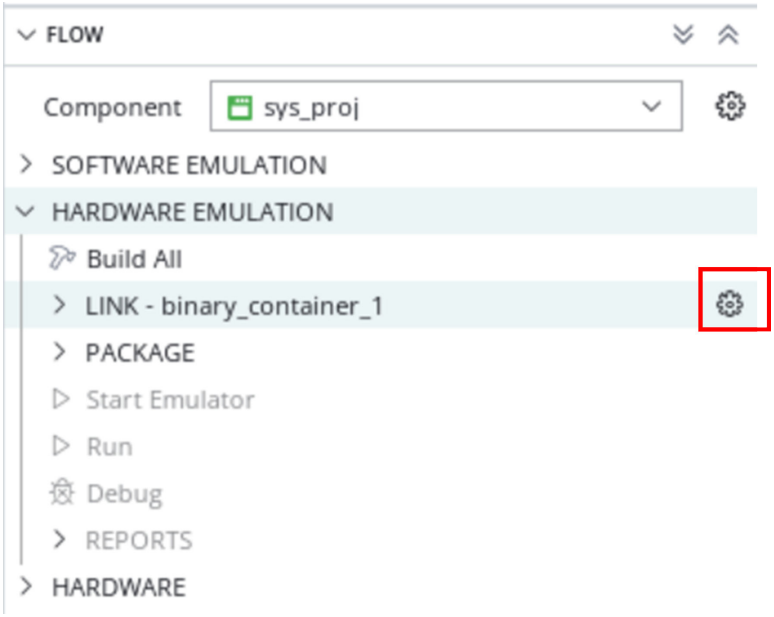
Table 46: **Possible Input Combinations for Selected Build Output**

Hardware Design (Fixed XSA)	Vitis Metadata Archive (VMA)	Vitis Subsystem (VSS)
Extensible XSA / XPFM Extensible XSA / XPFM + AIE GRAPH Extensible XSA / XPFM + PL KERNEL Extensible XSA / XPFM + AIE GRAPH + PL KERNEL Extensible XSA / XPFM + AIE GRAPH**N* + *PL KERNEL**N Extensible XSA / XPFM + VSS Extensible XSA / XPFM + VSS (No AIE GRAPH) + AIE GRAPH	Extensible XSA / PL KERNEL Extensible XSA / XPFM + AIE GRAPH Extensible XSA / XPFM + AIE GRAPH + PL KERNEL Extensible XSA / XPFM + AIE GRAPH**N* + *PL KERNEL**N Extensible XSA / XPFM + VSS (Include AIE GRAPH) Extensible XSA / XPFM + VSS (No AIE GRAPH) + AIE GRAPH	PL KERNEL 1 + PL KERNEL 2 PL KERNEL + AIE GRAPH PL KERNEL + VSS VSS + AIE GRAPH

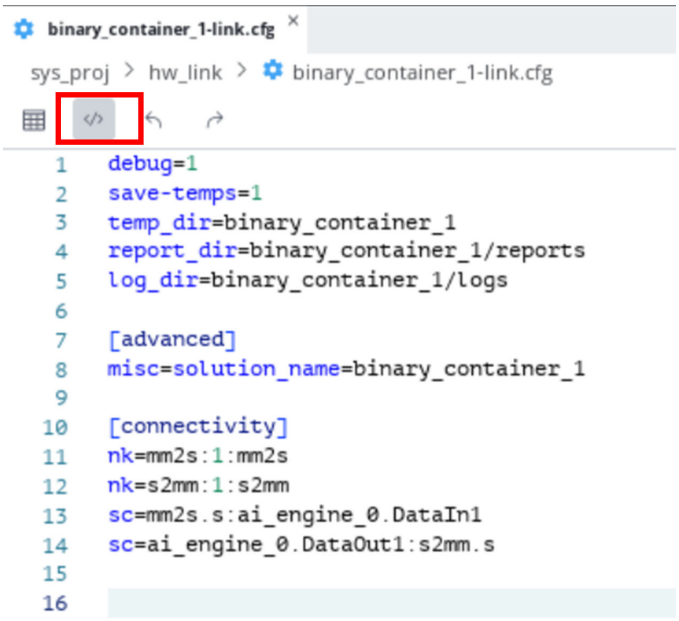
Enabling Third-Party Simulators in the Vitis IDE

Third-party simulator settings can be configured through Vitis IDE also. Below are the step-by-step instructions to do it.

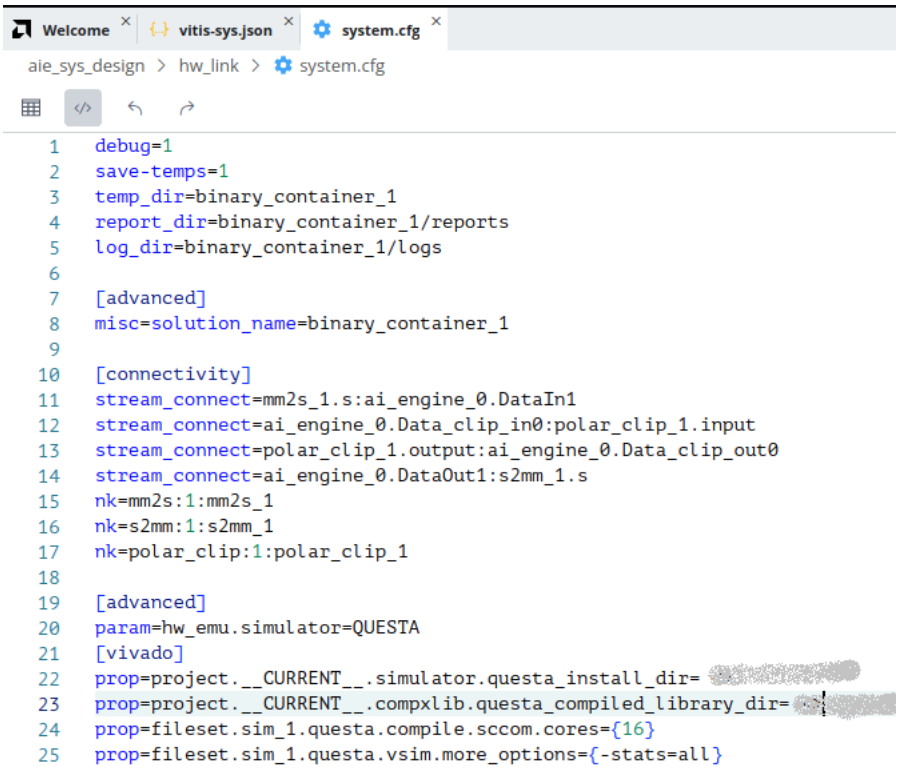
1. In the Flow View, click **Hardware Emulation** to highlight it, and click the Settings button. The Build Configuration Settings window opens.



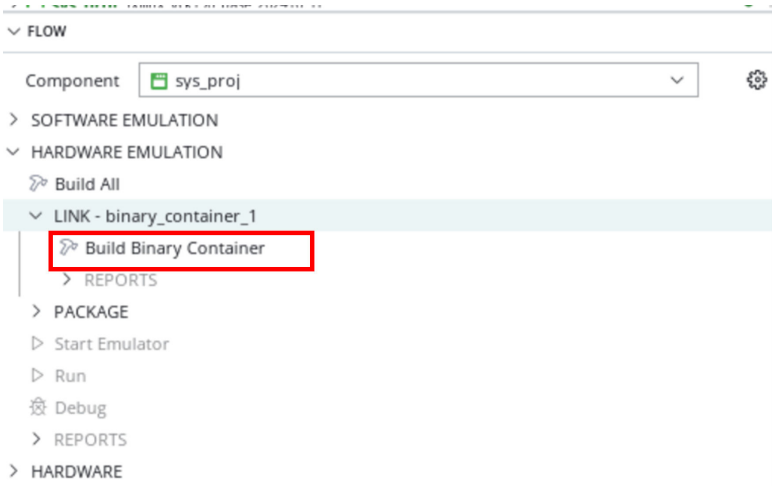
2. In Build Configuration Settings window, go to the Source Editor View, and specify the options and parameters.



3. Specify the simulator to use and settings in the configuration file in the `[advanced]` and `[Vivado]` categories.



4. When the modifications are made, build the design.


FLOW
Component
sys_proj
SOFTWARE EMULATION
HARDWARE EMULATION
Build All
LINK - binary_container_1
Build Binary Container
REPORTS
PACKAGE
Start Emulator
Run
Debug
REPORTS
HARDWARE

Chapter 8

Working with the Analysis View (Vitis Analyzer)

The Analysis view replaces the AMD Vitis™ Analyzer tool to provide an integrated view of the build and run summaries generated by the Vitis tools. The integrated analysis view provides ready access to view the results of a specific compilation step, or run or debug results, letting you quickly move between development and analysis.

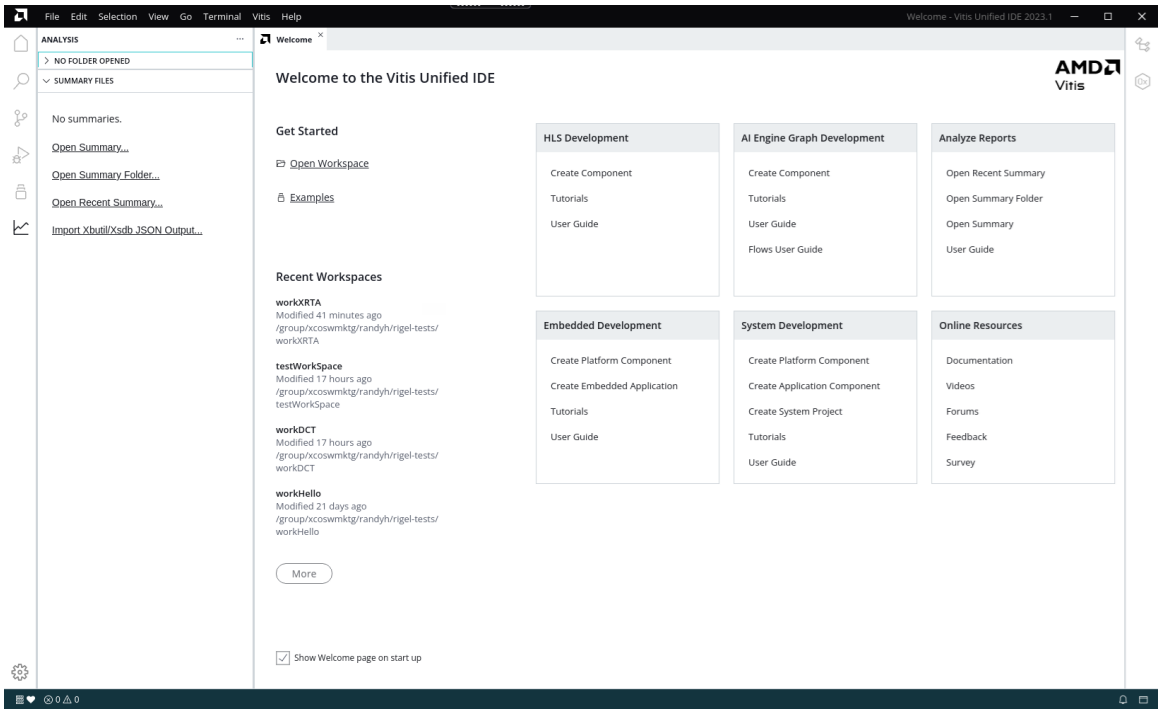
You can open and use Vitis Analyzer in a number of different ways.

1. Use Vitis Analyzer to review specific summary files from a variety of projects by opening the tool and summary directly:
   - Open the tool using the `vitis_analyzer` command (or `vitis -a`) and then select **Open Summary** to view the reports.
   - Open the tool with a specified summary file using `vitis_analyzer <summary_file>` (or `vitis -a` <summary>).

   TIP: *You can also specify a folder, such as a workspace directory, and the tool will scan the sub-folders for summary reports to open.*
2. Use Vitis Analyzer to review the reports generated for the current workspace in the IDE by selecting the Analysis view to see the hierarchy of reports, or by selecting the reports directly from the Flow Navigator for the active component.
3. Combine these two methods, by using the Analysis view to see the hierarchy of reports generated for the open workspace, and open Summary files independently.

When the Vitis Analyzer is opened from the command line, the tool displays a Welcome screen as shown in the following figure. The Analyze Reports box shows commands specific to the Analysis view of the tool.

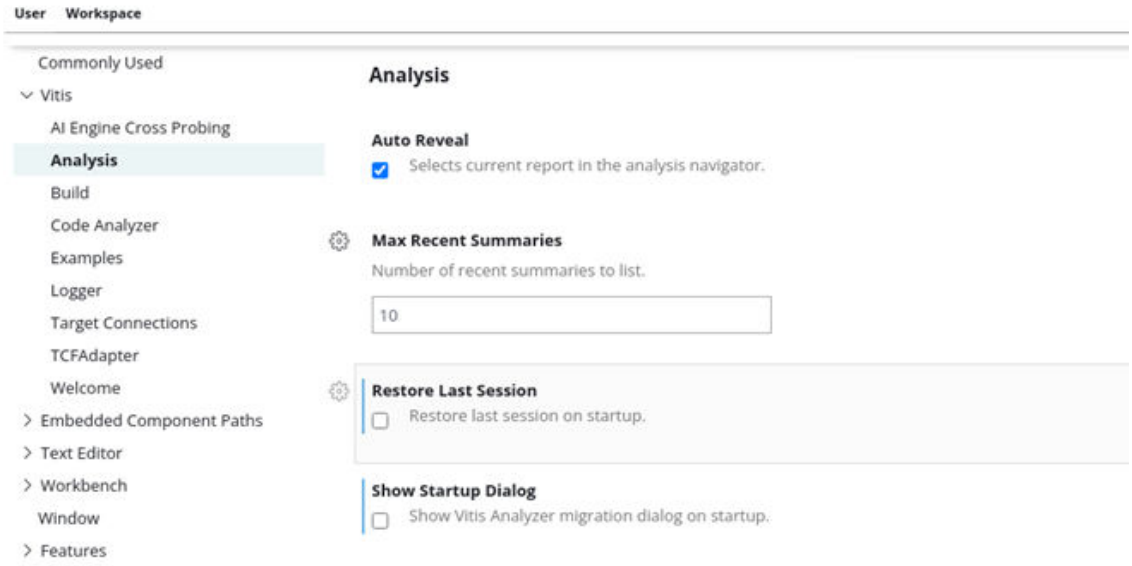
Figure 104: **Analysis View**



Configuring the Analysis View

Select **File → Preferences → Open Settings (UI)** to display the preference settings for the Vitis IDE, including the Analysis view as shown below.

Figure 105: **Analysis Preferences**



- **Auto Reveal:** Highlights the current report in the Analysis navigator. This lets you quickly locate the report you are viewing in the hierarchy of reports. The default is enabled.
- **Max Recent Summaries:** Specifies the number of Summaries to list when presenting Recently Opened Summaries. The default is 10.
- **Restore Last Session:** Opens the Vitis IDE with the summaries opened in the prior session of the tool.
- **Show Startup Dialog:** Displays a dialog box with a message regarding the new features or state of the tool. This dialog box can be dismissed after viewing, and can be re-enabled using this option.

Some additional Preferences that can has an affect on the Analysis view is the Commonly Used Preferences include:

- Auto Save Delay: Specifies the delay for the auto-save feature. This lets the tool save any modified files as the work is proceeding.

  TIP: *Auto Save is enabled on the File menu.*

- Font Size: Specifies the font size for the code editor and log file viewers. The default is 14.
- Font Family: Specifies the font to use in reports.

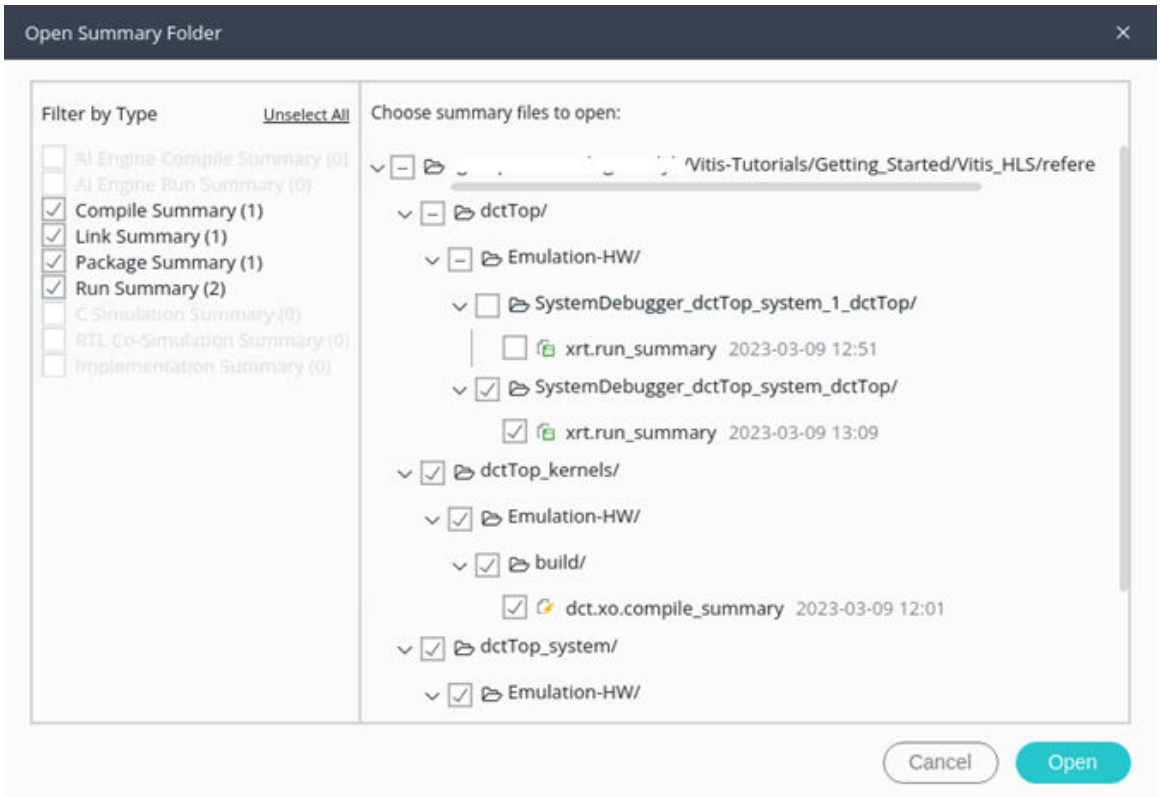
Open Summary Reports

Generally, the Summary reports provide a great overview of the specific steps in building and profiling the application to get a good view of where the application is with regard to performance and optimization.

- For individual components review the Compile Summary.
- For Application and device binary (`xclbin`) refer to the Link Summary, which also loads the Compile Summaries for linked components.
- For embedded processor and AMD Versal™ AI Engine systems, review the Package Summary.
- For performance profiling and event trace data related to the application execution, start with the Run Summary.

The Vitis analyzer offers the ability to open a folder, such as a workspace, using Open Summary Folder from the Welcome view, or from the command line when launching the tool. The tool examines the folder you specify and identifies the various build and run summaries included through the file hierarchy. The Open Summary Folder dialog box opens as shown below.

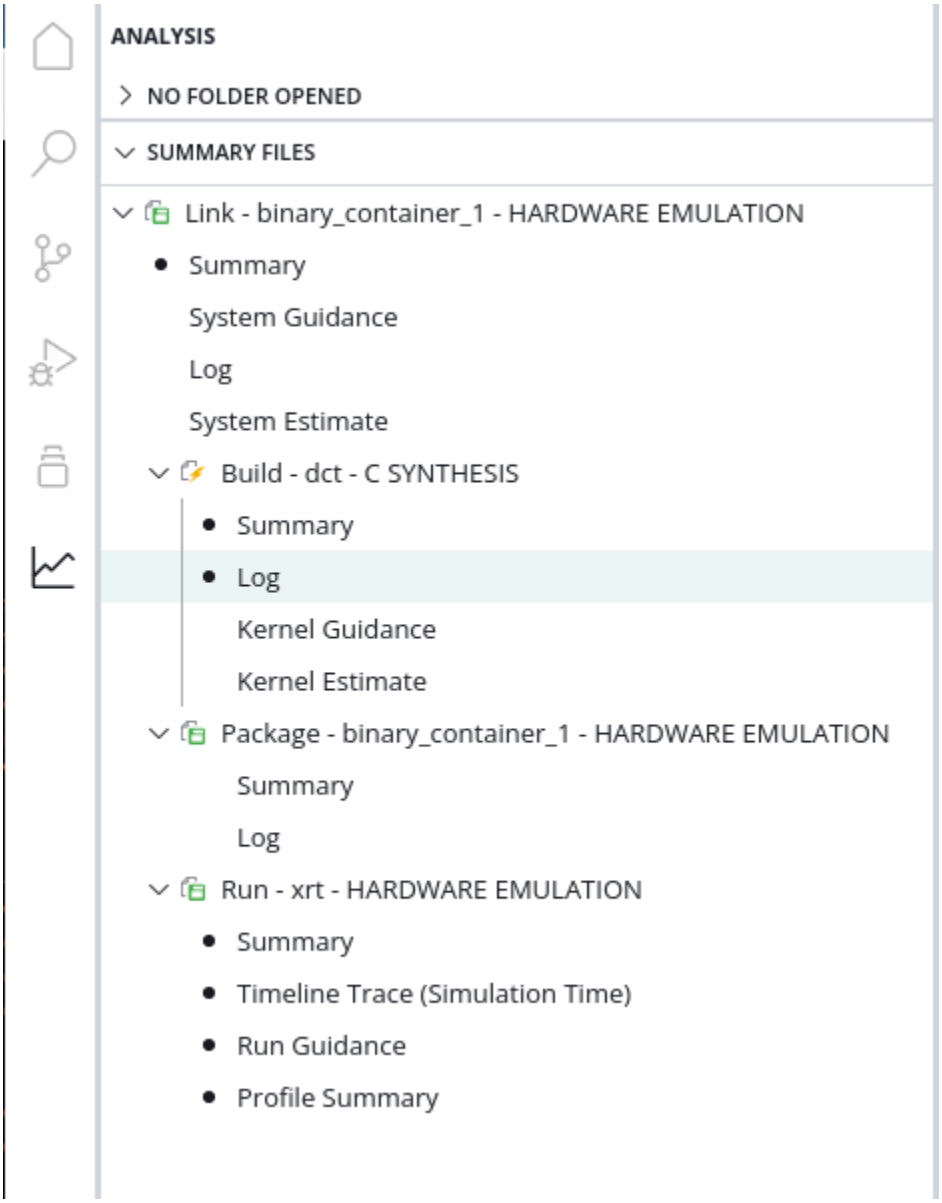
Figure 106: **Open Summary Folder**



The figure above shows the results of the Open Summary Folder when used on a Vitis IDE workspace. Filter by Type highlights the various types of reports found, and lets you enable or disable the reports of interest. Choose summary files to open lets you select specific files to open. In this way, you are viewing the type and specific reports of interest.

Vitis analyzer displays the various Summary reports in the Analysis view. Related Summaries are grouped into a hierarchy, so that the Link Summary for an Application includes the Compile Summaries and Run Summaries for the various components, in addition to any Run Summary for the Application. Separate Applications, or unrelated Components create separate hierarchies as shown below.

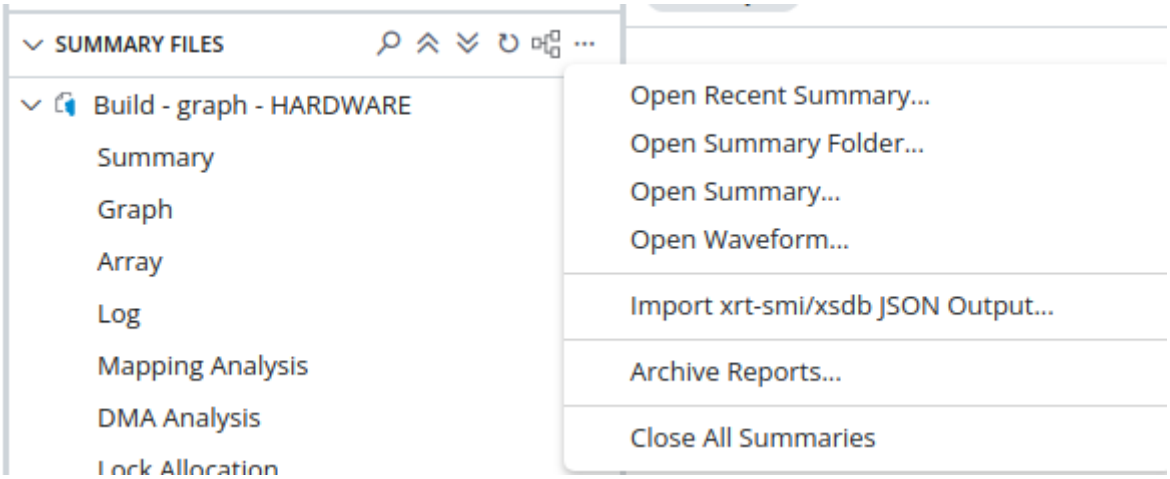
Figure 107: **Analysis View Hierarchy**



The Summary report provides access to the individual reports that are collected into the summary file, such as the Kernel Estimate, Operation Trace, AI Engine Trace, Timeline Trace, System Estimate, Log, and Timing Summary reports. For more information on the individual reports generated by the build and run processes, refer to Profiling the Application in the *Data Center Acceleration using Vitis* (UG1700).

The Summary Files section menu also includes a drop down menu that includes the command Import xrt-smi/xsdb JSON Output as shown in the following figure. This command lets you import a Json file generated by the `xrt-smi` command or `xsdb` commands to display in the Analysis view.

Figure 108: **Analysis View Menu**

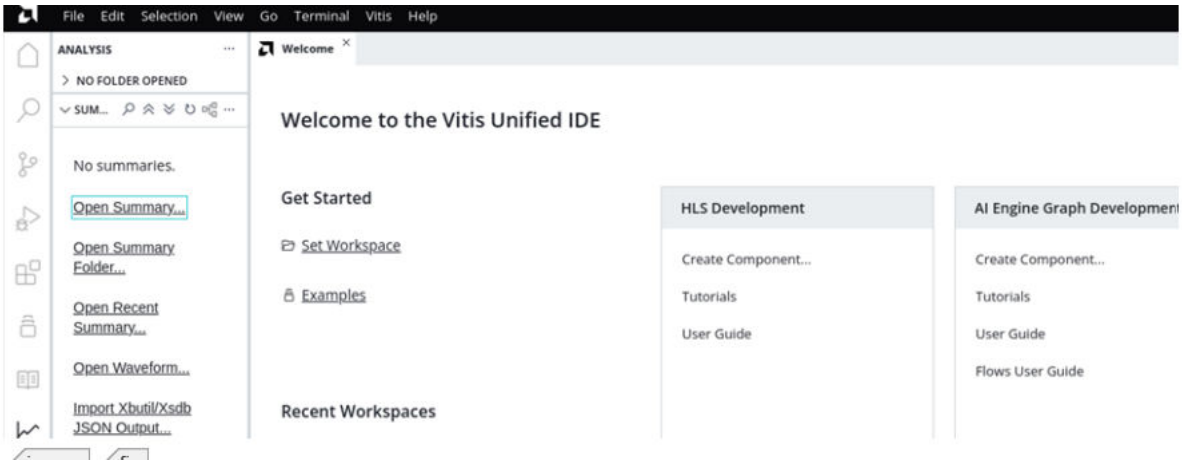


Open Trace Summary Using Time Window

To open the VCD Trace Summary view, use the time window selection utility provided in the Vitis Analysis view. If you need to generate the full trace VCD, but would like to debug and analyze waveform for a dedicated time window, the `vcdanalyze` utility can process the specific part of the VCD based on the user input. Upon running AI Engine simulator with VCD enabled, open the trace view as follows:

Open the run summary using `vitis_analyzer` command or upload the AI Engine run summary: `vitis_analyzer <default.aierun_summary>`

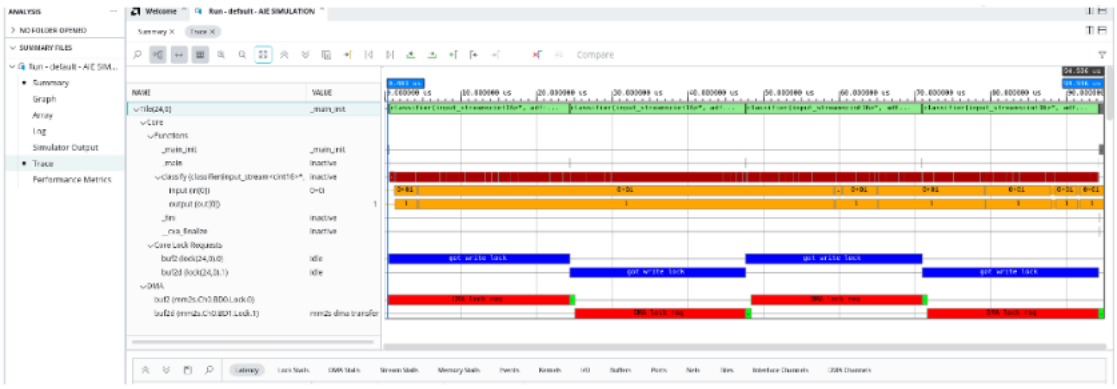
Figure 109: **Open Trace Summary Using Time Window**



Upon opening the summary and clicking on the Trace view, the time window appears to make a selection and specify start time and the end time. By default, full VCD trace time line can be seen in the Analysis view.

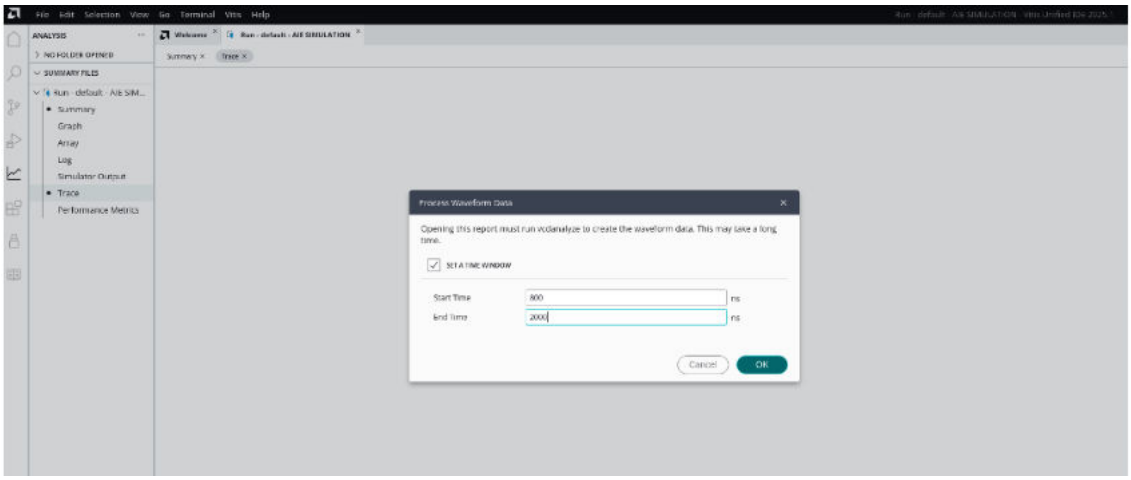
- To go with the default, click OK and proceed. This runs `vcdanalyze` that processes full VCD and displays the trace waveform.

Figure 110: **VCD Analyze**



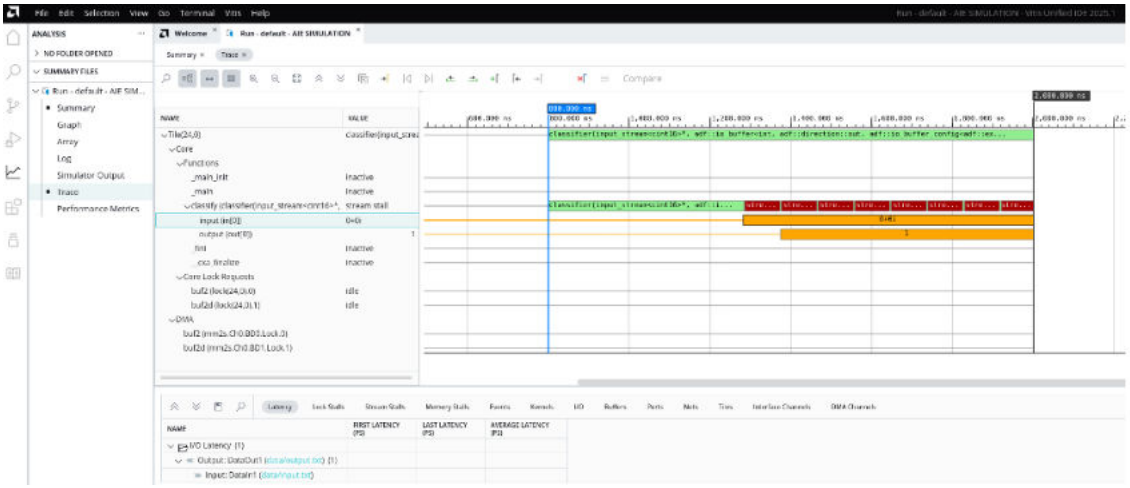
- To get the desired trace for a particular time duration, select "SET A TIME WINDOW", specify "Start Time " and "End Time" to proceed. This runs `vcdanalyze` to process a part of the VCD based on the time window specified and then display the trace waveform.

Figure 111: **Process Waveform Data**



This results in the trace waveform display from 800 ns to 2000 ns.

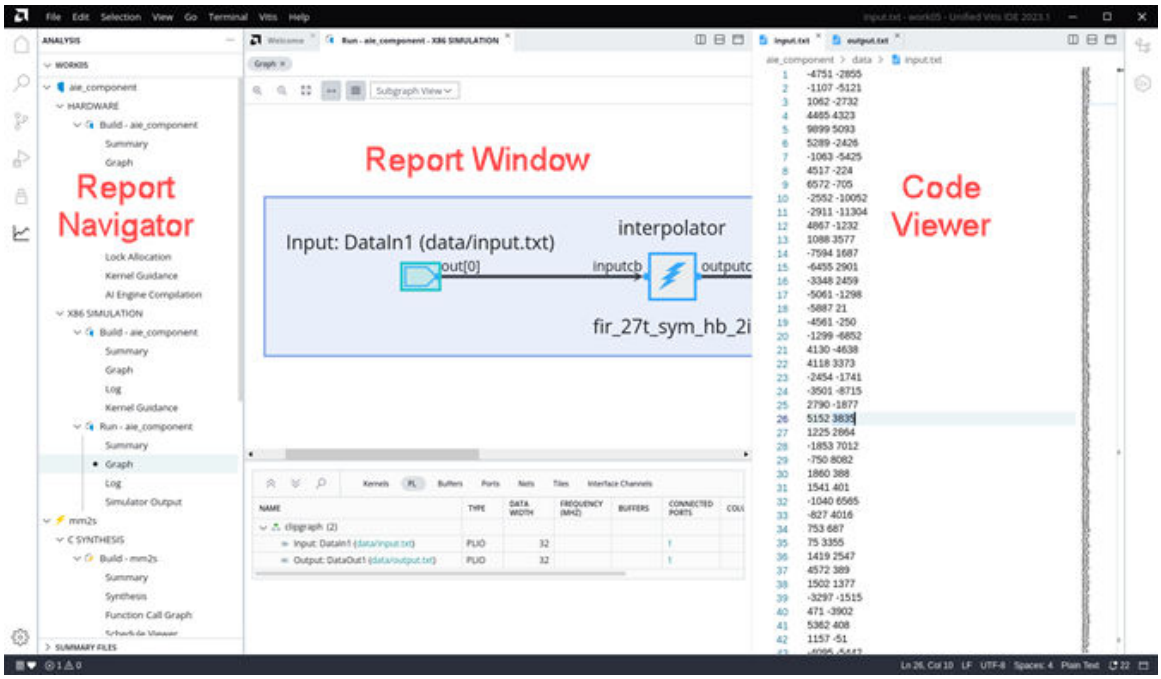
Figure 112: **Wavefrom Display from 800 ns to 2000 ns**



Analysis View Window Manager

As shown in the following figure, the Analysis view can be arranged into three views as shown below. The different views can be opened, closed, and rearranged as needed.

Figure 113: **Analysis View**



- **Report Navigator:** On the left side, this view lists all open summary files and associated reports. The Analysis view displays the hierarchy of reports generated from build and run of components in the open workspace of the Vitis IDE, or displays a selection of Summary files separately opened. You can use this view to quickly find and open a report.

  When you click any file in Report Navigator, it opens as a new tab in the Report Window. Opening a file adds a black dot next to the report name in Report Navigator to indicate the report is already open. Selecting an open report wil simply bring it to the front in the Report Window.

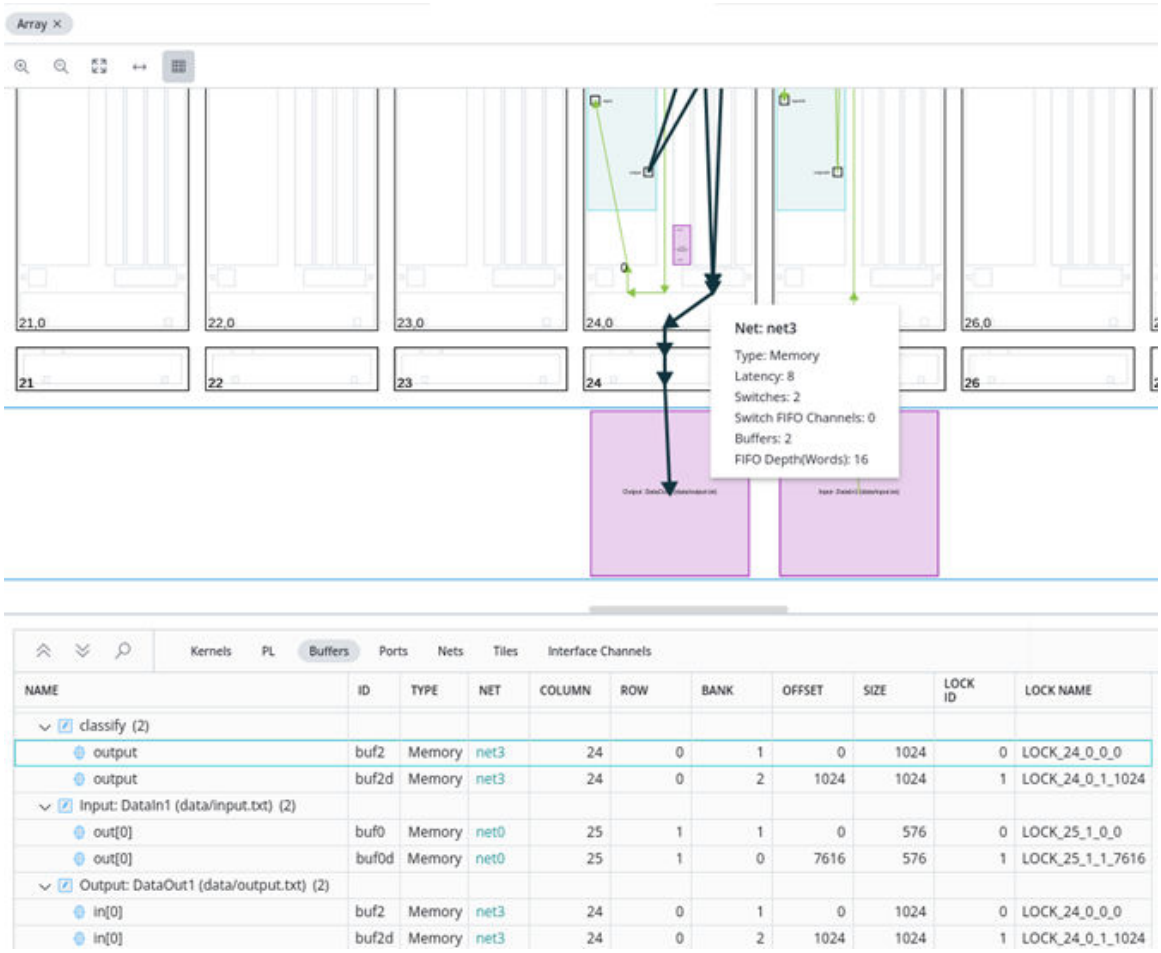
- **Report Window:** The center area displays the contents of the summary files and open reports. You can have multiple reports open in the Reports view, and quickly change from one report to another by selecting the window tab at the top of the view.

  All the reports related to the Compile Summary, Link Summary, or Run Summary are grouped together within a single container.

- **Code Viewer:** The optional Code Viewer is opened on the right side of the workspace. This lets you view and edit source code, based on feedback from the various reports. You can open the Source Code window by selecting a link in another report such as the Graph report for instance.

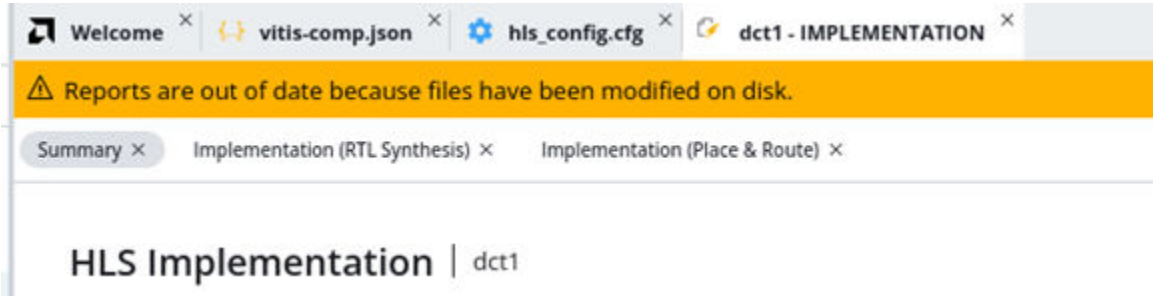
The Analysis view supports cross-probing between reports, such as within the Graph or Array diagrams, and from the Guidance report to other reports. Select the link in a report as shown below, and it will highlight the content in another open report, or open a file in the Code Viewer.

Figure 114: **Cross-Probe Between Reports**



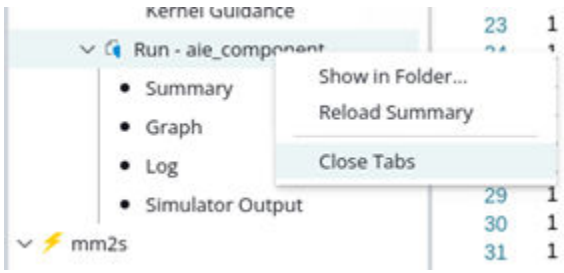
Reports that are currently opened in Vitis analyzer display an "out-of-date" banner if the summary files or reports are updated on the disk, due to recompiling or rerunning the application. You can keep working in the open report, or reload the updated files.

Figure 115: **Out-of-Date Reports**



To close open reports you can right-click the report in the Report Navigator and select **Close Tab**. You can also right-click a Summary file, as shown below and select **Close Tabs** to close all open reports associated with that Summary file.

Figure 116: **Close Tabs**



Viewing AI Engine Compilation Summary Reports

After the compilation of the AI Engine graph, the AI Engine compiler writes a summary of compilation results called `<graph-file-name>.aiecompile_summary` to peruse in the Analysis view of the Vitis Unified IDE. The summary contains a collection of reports, and diagrams reflecting the state of the AI Engine application implemented in the compiled build. The summary is written to the working directory of the AI Engine compiler as specified by the `--workdir` option, which defaults to `./Work`.

To open the AI Engine compiler summary, use the following command:

```
vitis ./Work/graph.aiecompile_summary
```

The Vitis Unified IDE opens the Analysis view displaying the Summary page of the report. The Report Navigator view lists the different reports that are available in the Summary. For more information, see Chapter 8: Working with the Analysis View (Vitis Analyzer).

The listed reports include:

- **Summary:** This is the top-level of the report, and reports the details of the build, such as date, tool version, a link to the graph, and the command used to create the build.
- **Kernel Guidance:** Shows a variety of messages to provide guidance on kernel optimization.
- **Graph:** Provides a flow diagram of the AI Engine graph that shows the data flow through the various kernels. You can zoom into and pan the graph display as needed. At the bottom of the Reports view, a table summarizes the graph with information related to kernels, buffers, ports, and nets. Clicking on objects in the graph diagram highlights the selected object in the tables. (See Graph and Array Details).

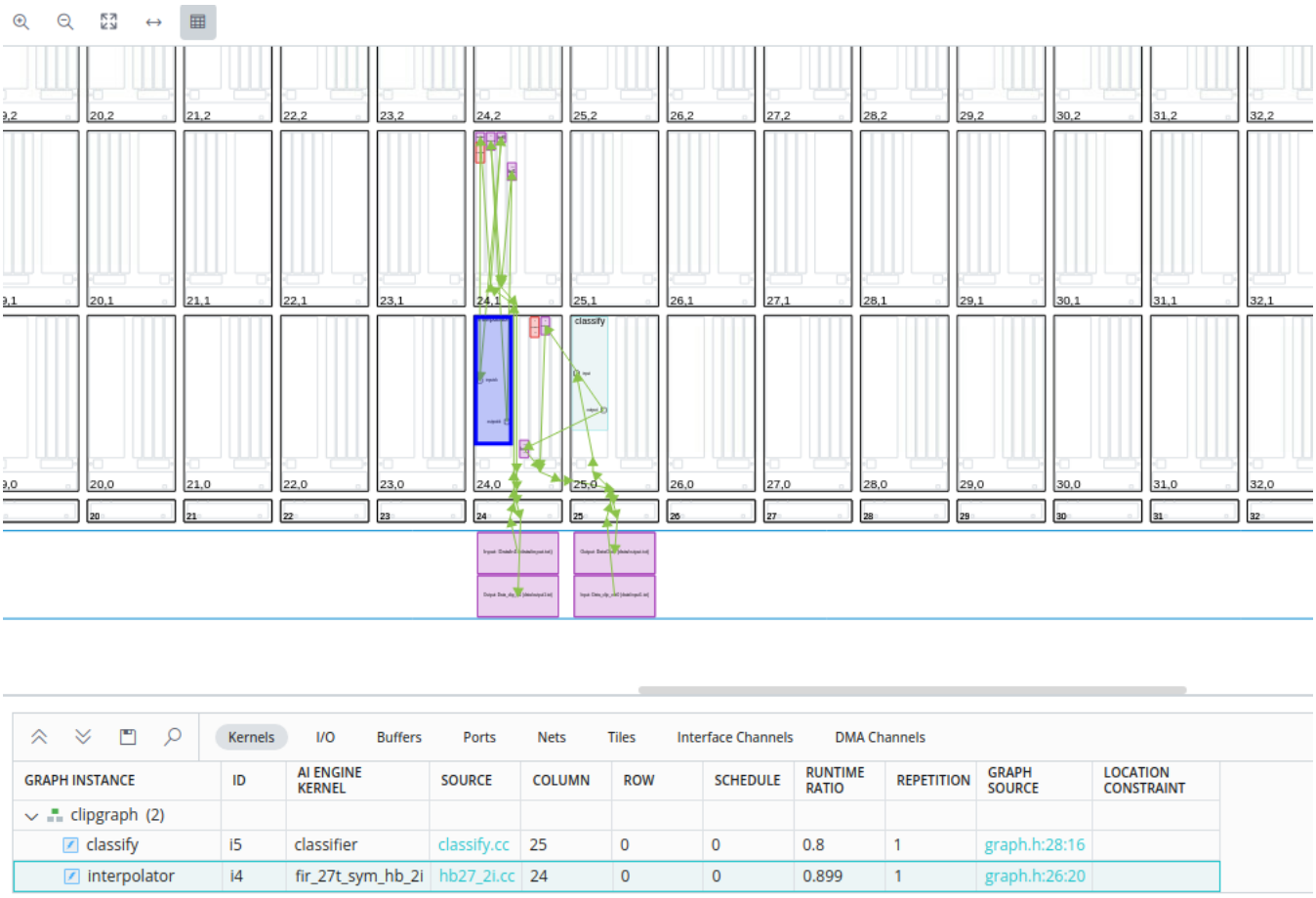
- **Array:** Provides a graphical representation of the AI Engine processor array on the Versal device. The graph kernels and connections are placed within the context of the array. You can zoom into and select elements in the array diagram. Choosing objects in the array also highlights the object chosen in the tables at the bottom of the **Reports** view.

  ***Note*:** The Graph and Array reports both share the same tables. When selecting an item in either of the views, it also selects it in both. For example, selecting a net in the Graph view, also selects it in the Array view.

- **Constraints:** Shows all constraints used within the graph and from .json constraint files.
- **Mapping Analysis:** Displays the text report `graph_mapping_analysis_report.txt`. Reports the block mapping, port mapping, and memory bank mapping of the graph to the device resources.
- **DMA Analysis:** Displays the text report `DMA_report.txt`, providing a summary of DMA accesses from the graph.
- **Lock Allocation:** Displays the text report `Lock_report.txt`, listing DMA locks on port instances.
- **AI Engine Compilation:** Shows the single kernel compilation log file.

The following figure shows the `graph.aiecompile_summary` report open in the Analysis view in the Vitis Unified IDE. In the displayed Array view, an AI Engine kernel is selected, and its associated tables are displayed at the bottom. The source code associated with the kernel is also displayed to the right.

Figure 117: **Array View**



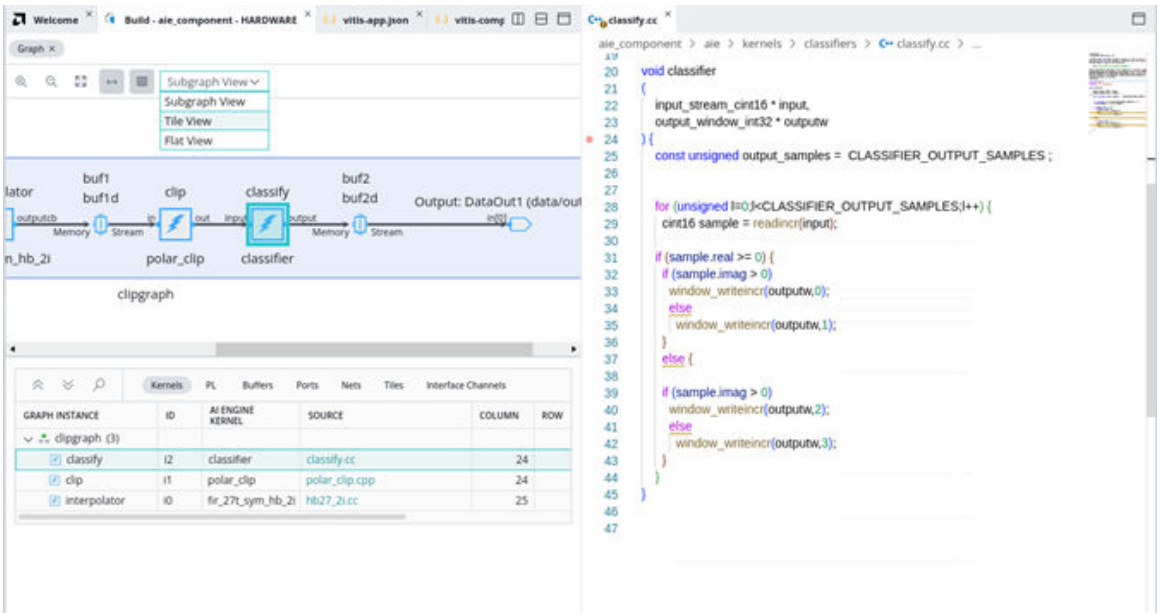
Kernels | I/O | Buffers | Ports | Nets | Tiles | Interface Channels | DMA Channels

GRAPH INSTANCE	ID	AI ENGINE KERNEL	SOURCE	COLUMN	ROW	SCHEDULE	RUNTIME RATIO	REPETITION	GRAPH SOURCE	LOCATION CONSTRAINT
clipgraph (2)										
classify	i5	classifier	classify.cc	25	0	0	0.8	1	graph.h:28:16	
interpolator	i4	fir_27t_sym_hb_2i	hb27_2i.cc	24	0	0	0.899	1	graph.h:26:20	

Viewing AI Engine Graphs and Arrays

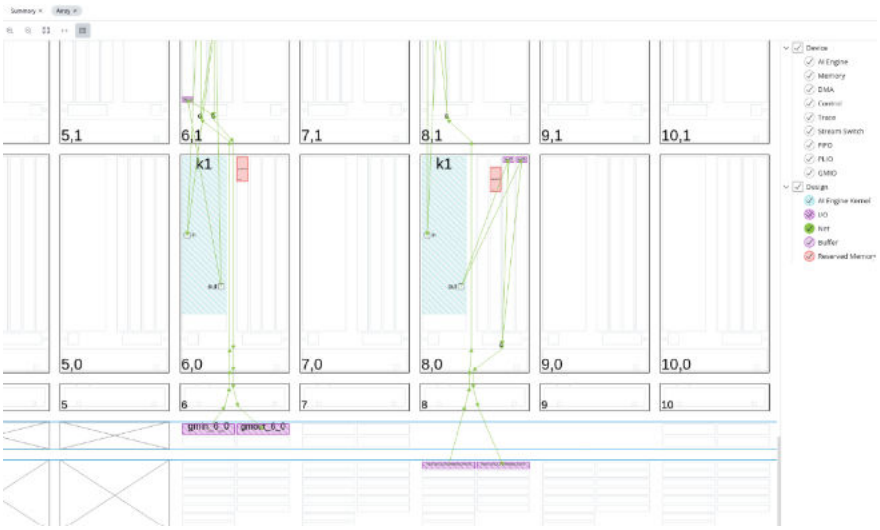
The AI Engine Graph and Array diagrams provide a quick overview of the structure of the ADF graph application implemented in the AI Engine tiles.

Figure 118: **AI Engine Graph Application**



The AI Engine Graph view shows the connectivity of the ADF graph as seen by the AI Engine compiler. The canvas shows nodes representing kernels, kernel arguments, memory buffers, and primary inputs and outputs, with edges drawn to represent the connectivity between these elements. Below the graphical view is a set of tables containing all of the objects drawn in the graphical view: kernels, buffers, ports, and nets, etc. Selecting an element in one of these tables will cross-probe to the graphical view and vice versa.

Figure 119: **AI Engine Array Diagram**



The Array view shows how the ADF graph is spatially placed on the AI Engine tile array. The array canvas contains all the core and memory components of the array. Kernels are drawn in the core where they are programmed, and connectivity between cores and/or memory are shown with connectivity lines. The available PLIO and GMIO interfaces are illustrated by a light grey rectangle. Upon utilization, the rectangle transitions to a pink, hashed appearance. In tiles where neither PLIO nor GMIO interfaces are present, the two regions are depicted as a white rectangle marked by a black cross. When either GMIO or PLIO interfaces are accessible, the remaining area is distinctly left empty.

To the right of the canvas, a Settings panel that can be used to customize the view of the array, by letting you show or hide elements of the resources and/or tile array.

Graph and Array Details

In the graph and array views in the Vitis Unified IDE Analysis view are several tables highlighting the details of the graph and kernels. The following sections provide a detailed description of each of the tables and the information available in the columns in each of the different tables.

Kernels

The Kernels table shows detailed information about the kernels used by the ADF graph. For example, the following figure shows three kernels, interpolator, clip and classify. The following example code shows the `fir_27t_sym_hb_2i`, `polar_clip`, and `classifier` kernel functions being instantiated as kernels in the graph.

```
interpolator = kernel::create(fir_27t_sym_hb_2i);
clip = kernel::create(polar_clip);
classify = kernel::create(classifier);
```

Figure 120: **Kernels Table**

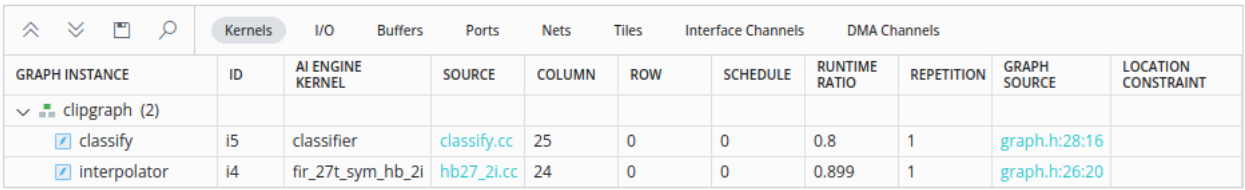


Table 47: **Column Description**

Column	Description
Graph Instance	Shows a hierarchical view of the design graph along with the sub-graphs and kernels.
ID	Unique ID given to the kernel from the AI Engine compiler.
AI Engine Kernel	The kernel function name. This does not need to match the kernel instantiated name in the graph class. For example, the `fir_27t_sym_hb_2i` is the function name and instantiated as interpolator as seen in the preceding code.

Table 47: **Column Description** *(cont'd)*

Column	Description
Source	The kernel source file. Clicking this file name opens up the source file of the kernel.
Column	The column in the AI Engine array where the kernel is mapped.
Row	The row in the AI Engine array where the kernel is mapped.
Schedule	The order in which kernels, if mapped to the same tile (i.e., same Column, Row) executes. A 0 means no scheduling is set.
Runtime Ratio	The runtime ratio set in the graph by using `runtime<ratio>(<kernel>) = n` constraint.
Graph Source	The source file (`graph.h`) with line number where the kernel is instantiated. Clicking on the link opens up the source file at the line number.

I/O

The I/O table, as shown in the following figure, provides detailed information about the PLIO connections to the ADF graph. For example, in this figure, there are two PLIO objects associated with the graph. The name of the PLIO connection, the width of the PLIO data connection, and the simulation test bench file associated with each PLIO connection is provided in the example.

```
input_plio  in  = input_plio::create("DataIn1", plio_32_bits,"data/input.txt");
output_plio out = output_plio::create("DataOut1", plio_32_bits,"data/output.txt");
```

The simulation test bench file also supports CSV file format that can be specified as shown in the below example.

```
input_plio  in  = input_plio::create("DataIn1", plio_32_bits,"data/input.csv");
output_plio out = output_plio::create("DataOut1", plio_32_bits,"data/output.csv");
```

Note: For details on the CSV format, refer to CSV File Format in the *AI Engine Tools and Flows User Guide* (UG1076).

Figure 121: **I/O Table**

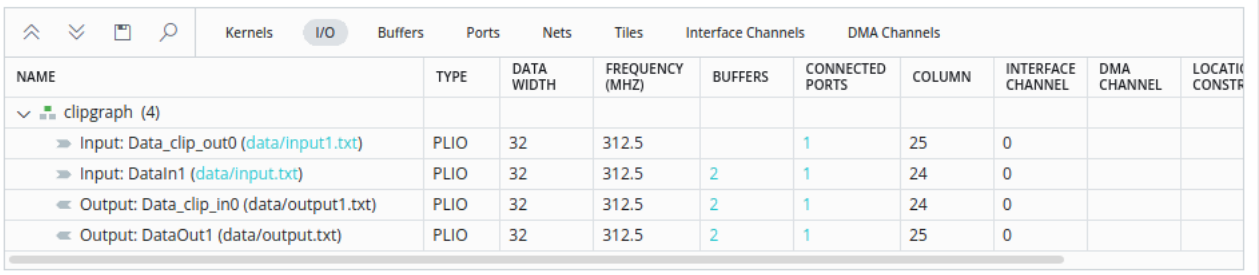


Table 48: **Column Description**

Column Name	Description
Name	The port name of a PLIO connection and whether it is an input or output.
Type	Connection to the PL (PLIO) or connection to the NoC (GMIO).

Table 48: **Column Description** *(cont'd)*

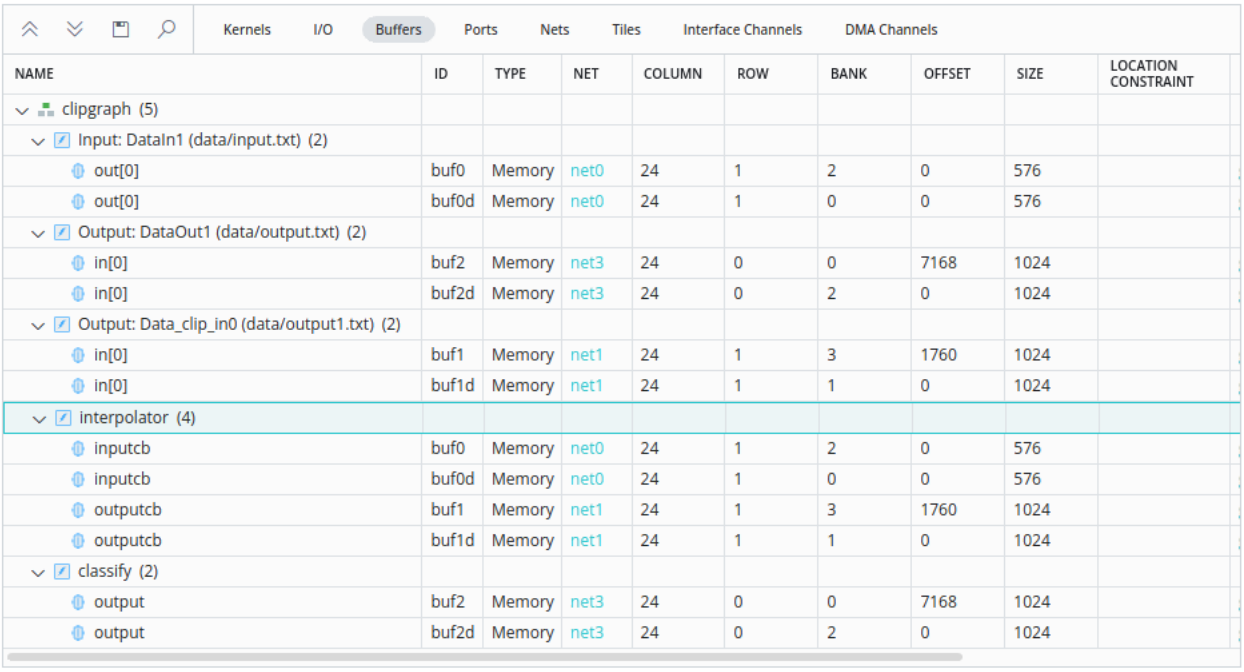
Column Name	Description
Data Width	The data width of the PLIO connection defined in the constructor. The width can be either 32 bits, or 64 bits, or 128 bits. Using 64 bits avoids inference of data width converters in programmable logic.
Frequency (MHz)	The frequency (in MHz) defined (optionally) in the PLIO constructor for the PLIO connection. The default is a quarter of the device speed grade AI Engine frequency.
Buffers	The number of buffers used in a PLIO connection. If a PLIO port is connected to an I/O buffer port of an AI Engine kernel two buffers are used, signifying a ping-pong buffer. A connection from a PLIO port to a stream port of the AI Engine kernel does not consume any buffers.
Connected Ports	The number of ports the PLIO is connected to. This PLIO data can be multicast to multiple destinations in the AI Engine. For more information, see Multicast Support in *AI Engine Kernel and Graph Programming Guide* (UG1079).
Column	The interface column used by the PLIO, which is assigned by the AI Engine compiler. The values could be in the 0-49 range.
Channel ID	The channel within the interface column used by the PLIO.
Packet IDs	The packet switching feature allows you to send packets of data to/from multiple destinations. These packets of data can be sent from/to the PL to/from the AI Engine. This column displays the ID of the packets used when packet switching is used. For more information, see Explicit Packet Switching in *AI Engine Kernel and Graph Programming Guide* (UG1079).

Buffers

The Buffers table contains information related to the buffers that are mapped to the ADF graph. Typically, buffers are used in window connections.

***Note*:** The use of `buf#` and `buf#d` means it is a ping-pong buffer.

Figure 122: **Buffers Table**



NAME	ID	TYPE	NET	COLUMN	ROW	BANK	OFFSET	SIZE	LOCATION CONSTRAINT
clipgraph (5)									
Input: DataIn1 (data/input.txt) (2)									
out[0]	buf0	Memory	net0	24	1	2	0	576	
out[0]	buf0d	Memory	net0	24	1	0	0	576	
Output: DataOut1 (data/output.txt) (2)									
in[0]	buf2	Memory	net3	24	0	0	7168	1024	
in[0]	buf2d	Memory	net3	24	0	2	0	1024	
Output: Data_clip_in0 (data/output1.txt) (2)									
in[0]	buf1	Memory	net1	24	1	3	1760	1024	
in[0]	buf1d	Memory	net1	24	1	1	0	1024	
interpolator (4)									
inputcb	buf0	Memory	net0	24	1	2	0	576	
inputcb	buf0d	Memory	net0	24	1	0	0	576	
outputcb	buf1	Memory	net1	24	1	3	1760	1024	
outputcb	buf1d	Memory	net1	24	1	1	0	1024	
classify (2)									
output	buf2	Memory	net3	24	0	0	7168	1024	
output	buf2d	Memory	net3	24	0	2	0	1024	

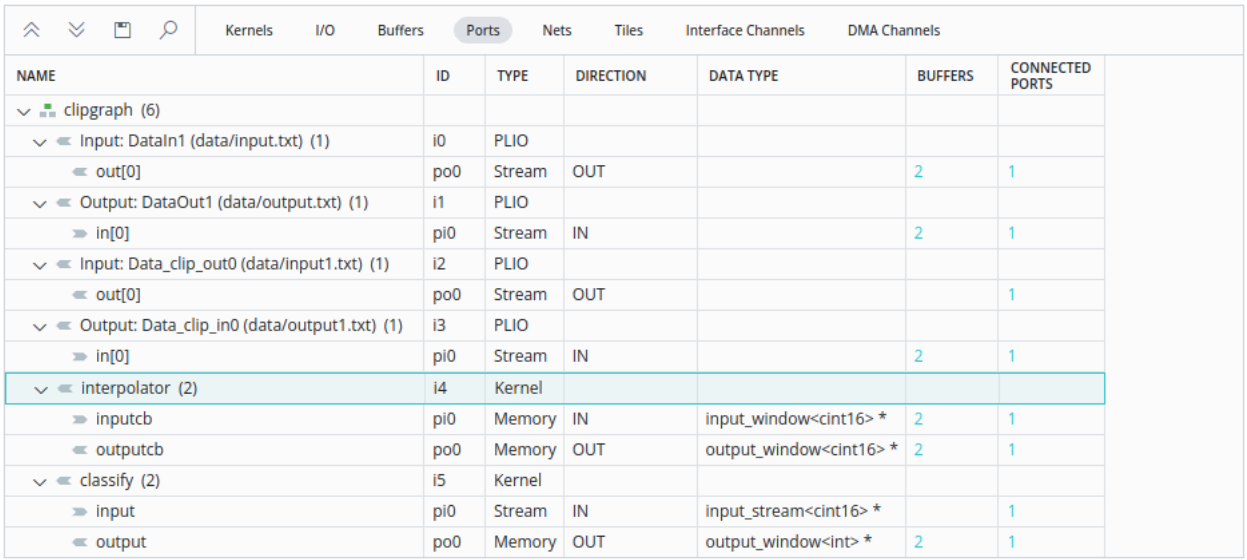
Table 49: **Column Description**

Column Name	Description
Name	The name of the connection where the buffer is allocated.
ID	The unique ID given to the buffer by the AI Engine compiler.
Type	The type of buffer being used. This can either be Memory or Stream. The connection to a window uses a ping-pong buffer, and connection to a stream might use a DMA buffer.
Net	The net with which the buffer is associated.
Column	The column location of the tile where the buffer is mapped by the compiler.
Row	The row location of the tile where the buffer is mapped by the compiler.
Bank	The bank of the tile where the buffer is mapped. The banks are: 0, 1, 2, or 3. ***Note***: The hardware view is eight banks of 128-bit width. The software view is four banks of 256-bit width.
Offset	The address offset of the buffer with the bank.
Size	The size of the buffer in bytes.
Lock ID	Unique ID per buffer if placed in the bank.
Lock Name	The unique name of the lock associated with the buffer. This can be used to debug a lock stall on a buffer.

Ports

The Ports table contains all the ports of the design which can be GMIO ports, PLIO ports and input, inout, and output ports on a kernel.

Figure 123: **Ports Table**



NAME	ID	TYPE	DIRECTION	DATA TYPE	BUFFERS	CONNECTED PORTS
clipgraph (6)						
Input: DataIn1 (data/input.txt) (1)	i0	PLIO				
out[0]	po0	Stream	OUT		2	1
Output: DataOut1 (data/output.txt) (1)	i1	PLIO				
in[0]	pi0	Stream	IN		2	1
Input: Data_clip_out0 (data/input1.txt) (1)	i2	PLIO				
out[0]	po0	Stream	OUT			1
Output: Data_clip_in0 (data/output1.txt) (1)	i3	PLIO				
in[0]	pi0	Stream	IN		2	1
interpolator (2)	i4	Kernel				
inputcb	pi0	Memory	IN	input_window<cint16> *	2	1
outputcb	po0	Memory	OUT	output_window<cint16> *	2	1
classify (2)	i5	Kernel				
input	pi0	Stream	IN	input_stream<cint16> *		1
output	po0	Memory	OUT	output_window<int> *	2	1

Table 50: **Column Description**

Column Name	Details
Name	The port name of the input, inout, output ports on a kernel, GMIO, or PLIO ports.
ID	The unique ID the AI Engine compiler designates the port.
Type	Port type. PLIO ports can contain Stream, Packet Switching, GMIO ports contain Global Memory, Function can contain Memory, or Stream.
Direction	Port direction. Can be: IN, OUT, INOUT.
Data Type	The type definition of the port for kernels. For example, `input_buffer<int16> &`, `input_stream<int16>*`.
Buffers	The number of buffers instantiated for the connection. For streaming connection, no buffers are used. For an I/O buffer connection, it is a ping-pong buffer.
Connected Ports	The number of ports the specific port is connected to. Ports can multicast to more than one port. For more information, see Multicast Support in the *AI Engine Kernel and Graph Programming Guide* (UG1079).

Nets

The Nets table shows details of the net connections made between the AI Engine kernels, or the AI Engine kernel and the PLIO/GMIO ports. For example, the code snippet below from the `graph.h` file provides examples of the connect constraint used to connect to the stream and window connections between the AI Engine kernels in the graph or to the PLIO/GMIO ports.

```
connect(in.out[0], interpolator.in[0]);
connect(interpolator.out[0], clip_in.[0]);
connect(clip_out.out[0], classify.in[0]);
connect(classify.out[0], out.in[0]);
```

A net name can be specified as in the following example:

```
connect net_name(src_port, dst_port)
```

Figure 124: **Nets Table**

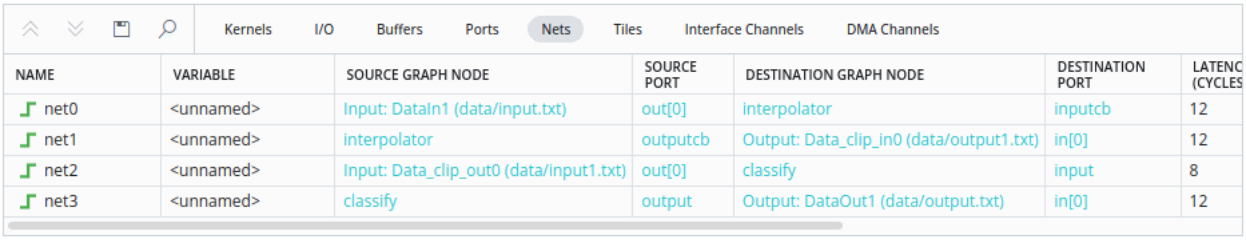


Table 51: **Column Description**

Column Name	Description
Name	The name of the net that is internally generated.
Variable	The name of the net connection (which can be optionally specified in the connect constraint). <unnamed>.net# signifies that the `connect` has no unique naming as part of the connect constraint in the graph.
Source Graph Node	The source node of the graph connection which could be a AI Engine kernel, PLIO or GMIO node.
Source Port	The source port of the graph connection which could be a AI Engine kernel, PLIO or GMIO port.
Source ID	The unique ID the AI Engine compiler designates the source port.
Destination Graph Node	The destination node of the graph connection which could be a AI Engine kernel, PLIO or GMIO node.
Destination Port	The destination port of the graph connection which could be a AI Engine kernel, PLIO or GMIO port.
Destination ID	The unique ID that the AI Engine compiler designates the destination port.
Latency (Cycles)	The minimum cycle count needed to transfer data from the source node to destination node.
FIFO Depth (Words)	The FIFO memory allocated through routing resources in the net. This includes buffers configured as DMA FIFOs, stream switch ports, and stream switch FIFOs. The unit for FIFO depth is 32-bit words.
FIFO Depth Constraint	This reflects the FIFO depth constraint provided in the design.
Buffers	The number of buffers used by the net connection.
Switch Count	The number of switches traversed by the net connection.
Switch FIFOs	The number of stream switch FIFOs used by the net connection.

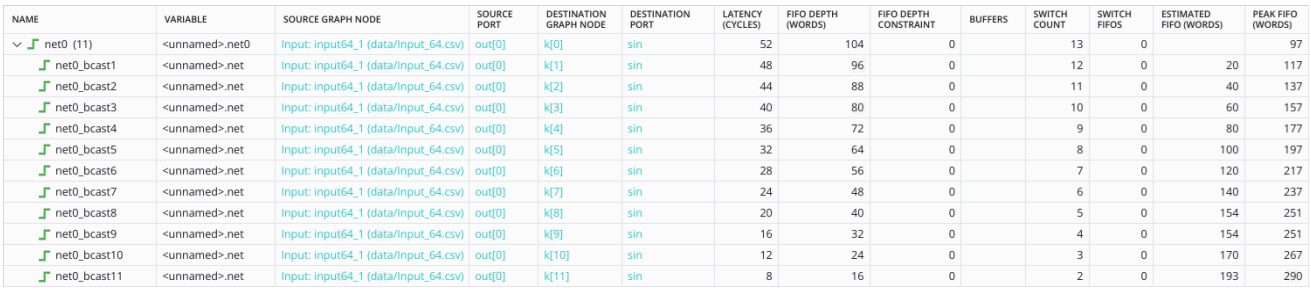
FIFO Depth Evaluation

When the system is deadlocked, the AI Engine compiler provides a way to evaluate the FIFO depth that you should add to your design to avoid these locks.

The tool computes these FIFO lengths from an AI Engine simulation for you. You can compile your graph with the AI Engine compiler using the `--evaluate-fifo-depth` flag.

After the code is compiled and the simulation is run, the simulation run summary can be viewed in the Vitis IDE. Additional FIFO depth columns are displayed in the Nets table.

Figure 125: **Nets Table With the FIFO Depth**



NAME	VARIABLE	SOURCE GRAPH NODE	SOURCE PORT	DESTINATION GRAPH NODE	DESTINATION PORT	LATENCY (CYCLES)	FIFO DEPTH (WORDS)	FIFO DEPTH CONSTRAINT	BUFFERS	SWITCH COUNT	SWITCH FIFOS	ESTIMATED FIFO (WORDS)	PEAK FIFO (WORDS)
net0 (11)	<unnamed>.net0	Input: input64_1 (data/Input_64.csv)	out[0]	k[0]	sin	52	104	0		13	0		97
net0_bcast1	<unnamed>.net	Input: input64_1 (data/Input_64.csv)	out[0]	k[1]	sin	48	96	0		12	0	20	117
net0_bcast2	<unnamed>.net	Input: input64_1 (data/Input_64.csv)	out[0]	k[2]	sin	44	88	0		11	0	40	137
net0_bcast3	<unnamed>.net	Input: input64_1 (data/Input_64.csv)	out[0]	k[3]	sin	40	80	0		10	0	60	157
net0_bcast4	<unnamed>.net	Input: input64_1 (data/Input_64.csv)	out[0]	k[4]	sin	36	72	0		9	0	80	177
net0_bcast5	<unnamed>.net	Input: input64_1 (data/Input_64.csv)	out[0]	k[5]	sin	32	64	0		8	0	100	197
net0_bcast6	<unnamed>.net	Input: input64_1 (data/Input_64.csv)	out[0]	k[6]	sin	28	56	0		7	0	120	217
net0_bcast7	<unnamed>.net	Input: input64_1 (data/Input_64.csv)	out[0]	k[7]	sin	24	48	0		6	0	140	237
net0_bcast8	<unnamed>.net	Input: input64_1 (data/Input_64.csv)	out[0]	k[8]	sin	20	40	0		5	0	154	251
net0_bcast9	<unnamed>.net	Input: input64_1 (data/Input_64.csv)	out[0]	k[9]	sin	16	32	0		4	0	154	251
net0_bcast10	<unnamed>.net	Input: input64_1 (data/Input_64.csv)	out[0]	k[10]	sin	12	24	0		3	0	170	267
net0_bcast11	<unnamed>.net	Input: input64_1 (data/Input_64.csv)	out[0]	k[11]	sin	8	16	0		2	0	193	290

Table 52: **FIFO Depth Columns**

Additional Column Name	Description
Estimated FIFO (Words)	FIFO depth that is recommended to be used in the graph to resolve the deadlock situation.
Peak FIFO (Words)	Maximum FIFO depth that is used to resolve the deadline situation. This is then used to compute the estimated FIFO.

The estimated FIFO depth given in the Estimated FIFO (Words) column can be used in the graph to insert FIFOs with the specified length using the following syntax:

```
fifo_depth(net) = value;
```

Where `net` is a connection on which you want to insert the FIFO.

***Note*:** In AI Engine-ML, if the recommended FIFO depth is above the default threshold (40 words), you will have to use the option `--swfifo-threshold` to increase this limit.

Tiles

The Tiles table shows all the tiles that have mapped kernels and buffers in the ADF graph. For example, in this design there are four tiles used, where three of them contain kernels (Tile [24,0], Tile [24,1] and Tile [25,0]), and three of them have buffers mapped (Tile[24,0], Tile[24,1] and Tile[25,1]).

Figure 126: **Tiles Table**

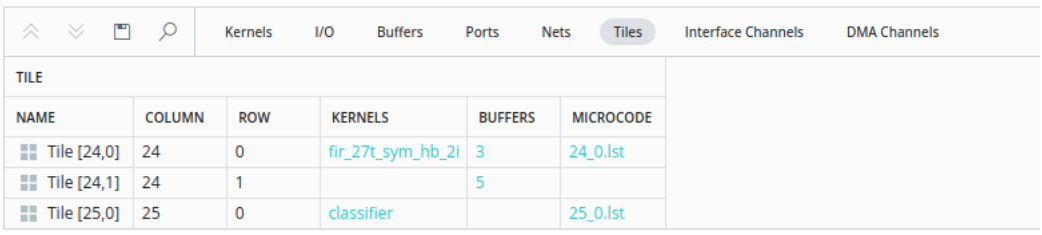


Table 53: **Column Description**

Column Name	Description
Tile	The tile ID.
Column	The column location of the tile.
Row	The row location of the tile.
Kernels	The number of kernels that are mapped to the tile.
Buffers	The number of buffers mapped to the tile. This includes buffers on nets and buffers inside the kernel.

Interface Channels

The interface channel table contains a list of channels that are used in the design and these channels are interfaces between AI Engine and PL, or between AI Engine and the global memory. Typically they are defined as `input_plio`, `output_plio`, `input_gmio` or `output_plio` objects in graph class.

Figure 127: **Interface Channels**

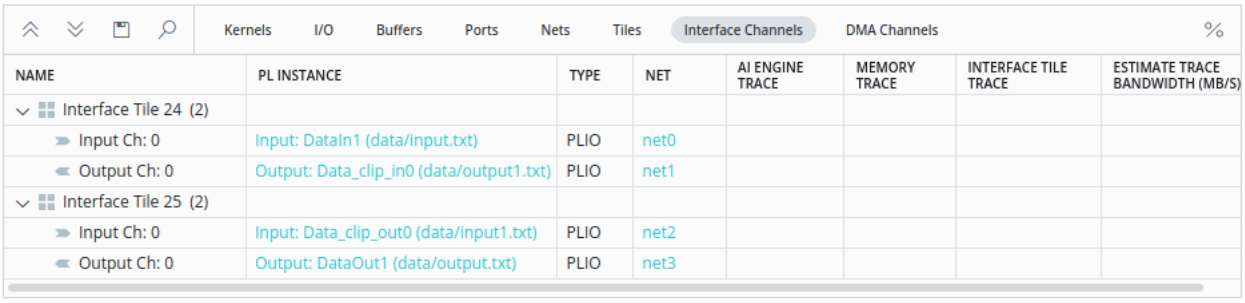


Table 54: **Table Column Description**

Column Name	Description
Name	The interface channel name that contains the tile number, Input or Output, and the channel number.
PL Instance	The design PLIO object information that includes Input or Ouput, and the associated input or output files.
Net	The net with which the interface channel is associated. If the value is a number, the stream is broadcast to 'N' number of sub-nets. If the value is 'netXXX', this means the net is a single path.
AI Engine Trace	The number of AI Engine trace packet streams that are merged onto a particular channel in an interface tile.
Memory Trace	The number of memory trace packet streams that are merged onto a particular channel in an interface tile.
Interface Tile Trace	The number of interface tile trace packet streams that are merged onto a particular channel in an interface tile.

Compilation Summary

After the compilation of the AI Engine graph, the compile summary can be open in the Vitis IDE. The compile summary provides information on the tool version, the build start and end time, the duration of the build, the AI Engine frequency, and the resources used by the AI Engine design. In addition, the report includes AI Engine kernel details.

Figure 128: **Status**

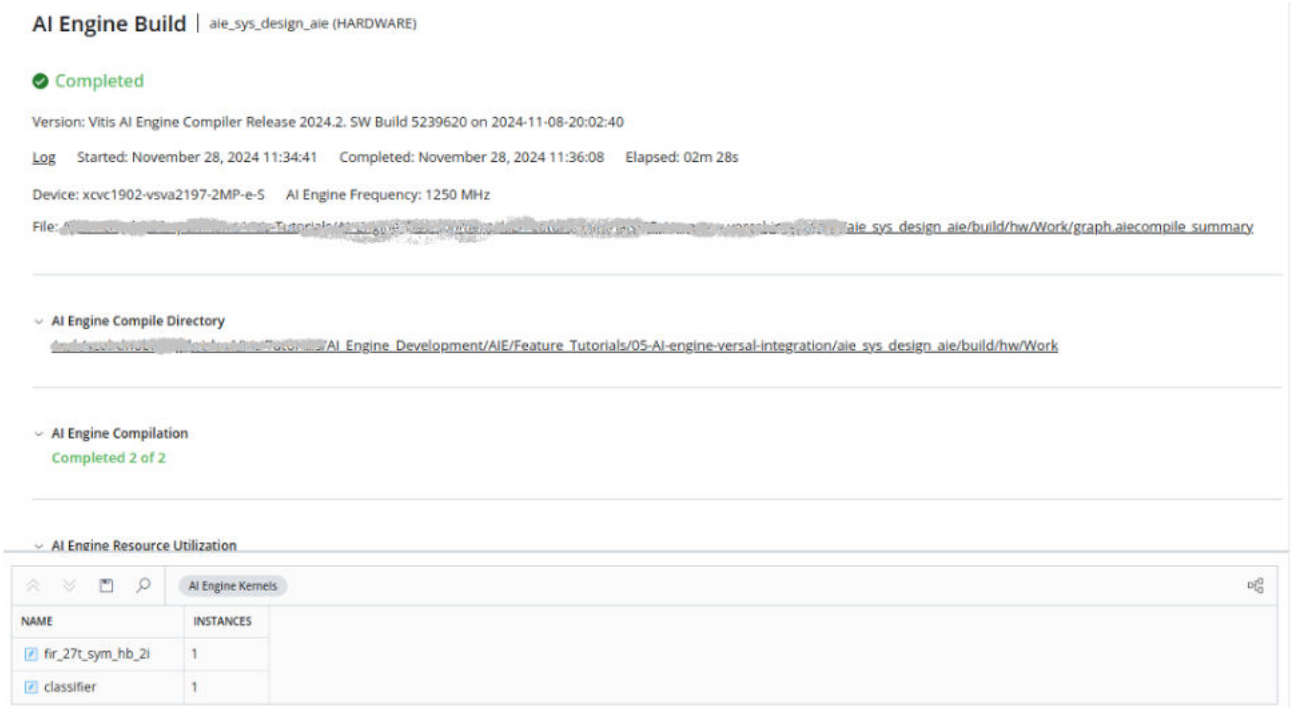
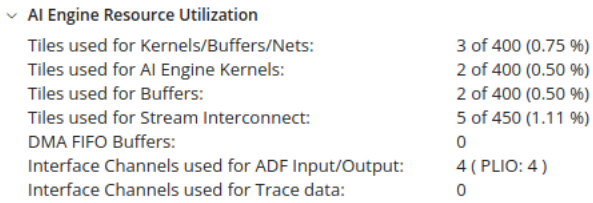
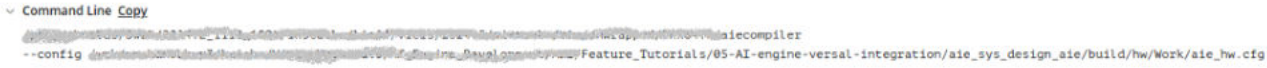


Figure 129: **AI Engine Resource Utilization**



The `COMMAND` section provides details about different options used during compilation.

Figure 130: **Compilation Command**



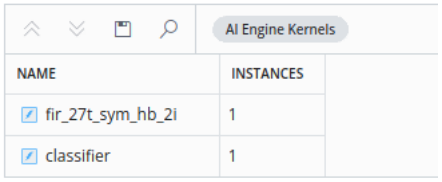
The `AI ENGINE RESOURCE UTILIZATION` section provides an overview of the resource utilization of the design. Details are found in the following table.

Table 55: **AI Engine Resource Utilization Descriptions**

Utilization Parameter	Description
Tiles used for AI Engine Kernels	The number of tiles used by the AI Engine kernels in the design. Format is `X of Y (Z%)`.
Tiles used for Buffers	The number of buffers used by the design. Format is `X of Y (Z%)`.
Tiles used for Stream Interconnect	The number of tiles used to communicate through streams to other AI Engines, PL or DMA.
DMA FIFO Buffers	The number of DMA FIFO buffers used by the design.
Interface channels used for PL and Trace data	The number of interface channels used for PL and Trace stream data.
Interface Channels used for Trace data	The number of interface channels used for Trace stream data.

The `AI ENGINE KERNELS` section highlights AI Engine kernels that instantiate identical kernel functions, but that differ with respect to the parameters used, and the corresponding number of instances of those kernels.

Figure 131: **AI Engine Kernels**



AI Engine Kernel Stack and Heap Reports

After the AI Engine compiler completes the compilation of an AI Engine design, the kernels' stack and heap usage information can be shown in Vitis Unified IDE. The reports can be found from **AI Engine Compilation → Tile [COL_ROW] → Call Tree** or **AI Engine Compilation → Tile [COL_ROW] → Memory**

Following are examples of the snapshots of the reports:

Figure 132: **AI Engine Kernel Call Tree Report**

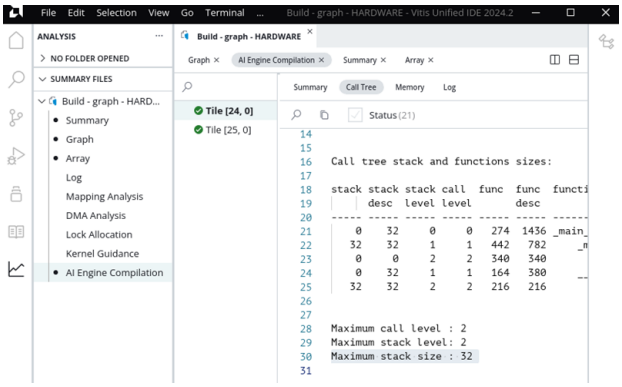
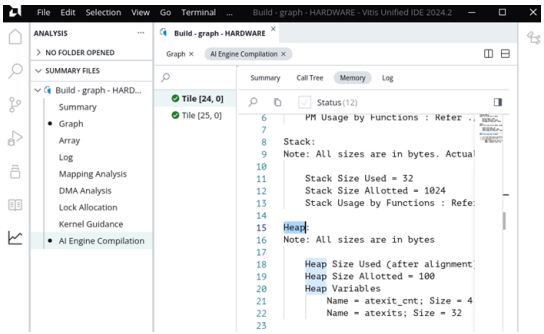


Figure 133: **AI Engine Kernel Memory Report**



Importing JSON Output Files

You can output a single snapshot of the AI Engine status to a JSON file at any time after the device has been loaded. This is a static snapshot of the AI Engine running status, along with the events that happened before. Refer to the chapter on Manual AI Engine Status Output in the *AI Engine Tools and Flows User Guide* (UG1076) for more information. An example command is:

```
xrt-smi examine -r aie -d 0 -f json -o aie_status_xrt-smi.json
```

The `aie_status_xrt-smi.json` file can be imported into the Analysis view using the following specific steps.

1. In the Summary Files section menu of the Analysis view select **Import Xrt-smi/Xsdb JSON Output**.
2. In the displayed dialog box, set the following options.
   - xrt-smi/Xsdb JSON Output File: Select the JSON file that was manually generated with the `xrt-smi` command. For example, select the file `aie_status_xrt-smi.json`.

- AI Engine Compile Summary: Select the AI Engine compile summary file, for example `./Work/graph.aiecompile_summary`.
- Save Run Summary: The run summary file to be written. The run summary can be used to reload the analysis next time by **Analysis → Open Summary** or **File → Open Recent → Summary** .

The Graph and Array views are shown in the Analysis view.

TIP: *If you load the JSON file before loading the compile `workdir`, the information will not be displayed because there is no graph report. An imported JSON file overrides any other JSON file previously set.*

For more information, see Performance Analysis of AI Engine Graph Application during Simulation in the *AI Engine Tools and Flows User Guide* (UG1076) and Performance Analysis of AI Engine Graph Application on Hardware in the *AI Engine Tools and Flows User Guide* (UG1076).

Chapter 9

Additional Information

Output Structure of the Vitis Tools

Output Directories of the v++ Command

The directory structure generated by the command-line flow has been organized to let you easily find and access files from the project. By navigating the various `compile`, `link`, `logs`, and `reports` directories, you can easily find generated files. Similarly, each kernel also has a directory structure created.

You can optionally change the directory structure using the following `v++` options:

```
--temp_dir <dir_name>
--log_dir <dir_name>
--report_dir <dir_name>
```

When using `v++` on the command line, by default it creates a directory structure during compile and link. The `.xo` and `xclbin` files are always generated in the current working directory. All the intermediate files are created under the directory specified by the `--temp_dir` option, which defaults to `_x` when `--temp_dir` is not specified. The `link`, `logs`, and `reports` directories default to inside of the `temp_dir`, and contain the respective information on the builds.

The example directory provided below results from the following command lines:

```
## Kernel Compilation command:
v++ -c -t hw_emu --platform xilinx_u200_gen3x16_xdma_2_202110_1 --
config ./src/u200.cfg \
-k vadd -I./src ./src/vadd.cpp -o hw_emu/vadd.xo

## Device Binary Linking Command:
v++ -l -t hw_emu --platform xilinx_u200_gen3x16_xdma_2_202110_1 --
config ./src/u200.cfg \
hw_emu/vadd.xo -o hw_emu/vadd.xclbin
```

The `u200.cfg` file defines the following options:

```
debug=1
save-temps=1

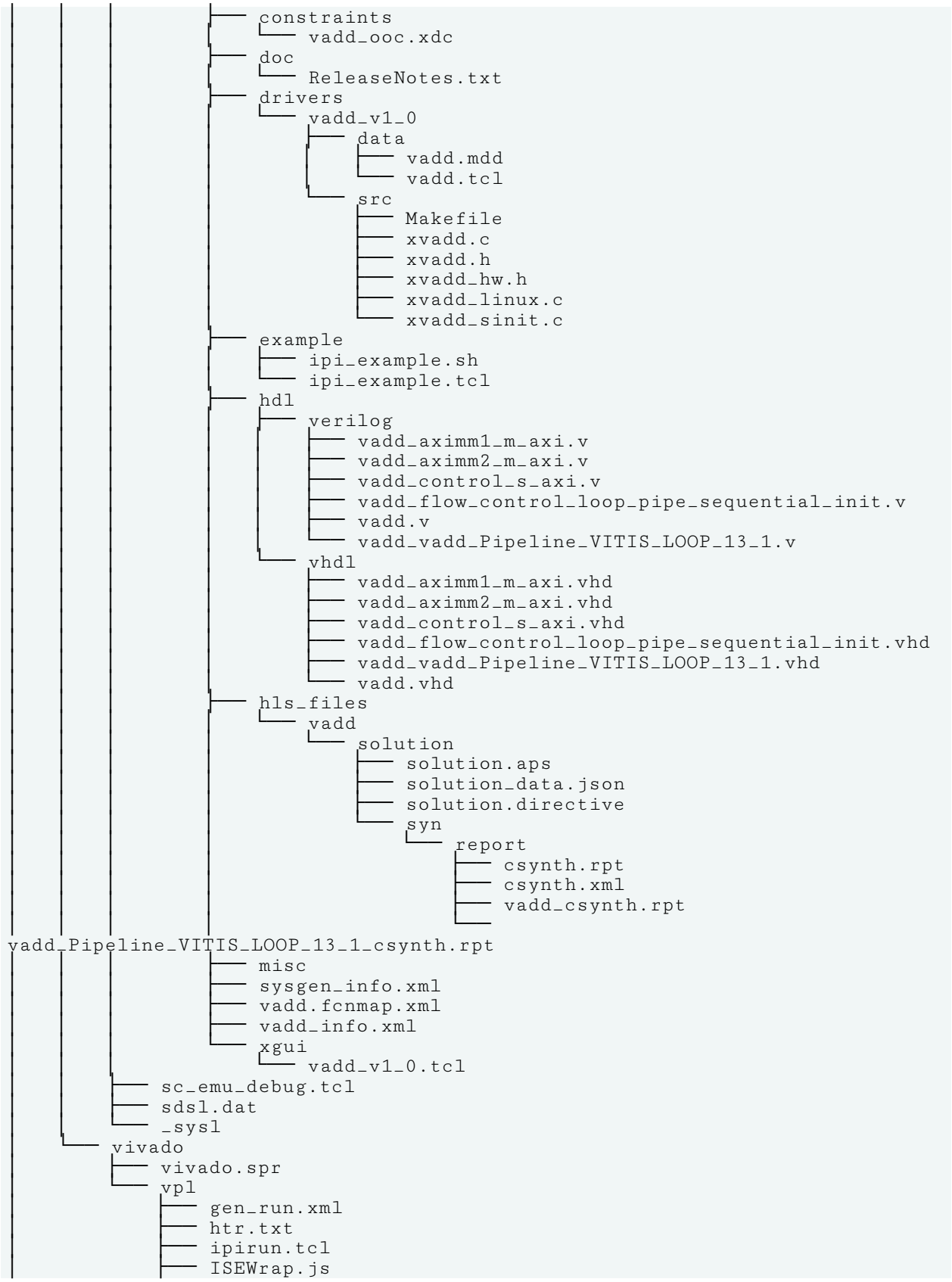
[connectivity]
nk=vadd:1:vadd_1
sp=vadd_1.in1:DDR[1]
sp=vadd_1.in2:DDR[2]
sp=vadd_1.out:DDR[1]

[profile]
data=all:all:all
```

An example of the `temp_dir` output of the `v++` link command follows:

```
├── link
│   ├── activetask.json
│   ├── int
│   │   ├── address_map.xml
│   │   ├── appendSection.rtd
│   │   ├── automation_summary.txt
│   │   ├── automation_summary_update_bd.txt
│   │   ├── behav_waveform
│   │   │   └── xsim
│   │   │       ├── compile.log
│   │   │       ├── compile.sh
│   │   │       ├── dr_behav.protoinst
│   │   │       ├── dr_behav.wcfg
│   │   │       ├── elaborate.log
│   │   │       ├── elaborate.sh
│   │   │       ├── glbl.v
│   │   │       ├── launch_hw_emu.sh
│   │   │       ├── libdpi.so
│   │   │       ├── pfm_top_wrapper.tcl
│   │   │       ├── pfm_top_wrapper_vhdl.prj
│   │   │       ├── pfm_top_wrapper_vlog.prj
│   │   │       ├── pfm_top_wrapper_xsc.prj
│   │   │       ├── post_sim_tool_scripts.tcl
│   │   │       ├── preprocess_profile.tcl
│   │   │       ├── pre_sim_tool_scripts.tcl
│   │   │       ├── prj.smi
│   │   │       ├── profile.tcl
│   │   │       ├── protoinst_files
│   │   │       ├── run.sh
│   │   │       ├── sc_xtlm_bd_d216_interconnect_DDR4_MEM01_0.mem
│   │   │       ├── sc_xtlm_bd_d216_interconnect_M00_AXI_MEM00_0.mem
│   │   │       ├── sc_xtlm_pfm_dynamic_smartconn_data_0_0.mem
│   │   │       ├── sc_xtlm_pfm_top_smartconnect_0_0.mem
│   │   │       ├── simulate.log
│   │   │       ├── simulate.sh
│   │   │       ├── vitis_params.tcl
│   │   │       ├── waveform_debug_enable.txt
│   │   │       ├── xelab.pb
│   │   │       ├── xsc.log
│   │   │       ├── xsc.pb
│   │   │       ├── xsim.dir
│   │   │       ├── xsim.ini
│   │   │       ├── xsim.ini.bak
│   │   │       ├── xvhdl.log
│   │   │       ├── xvhdl.pb
│   │   │       ├── xvlog.log
```

```
│   │   │   │   └── xvlog.pb
│   │   │   ├── behav.xse
│   │   │   ├── cf2sw_full.rtd
│   │   │   ├── cf2sw.rtd
│   │   │   ├── debug_ip_layout.rtd
│   │   │   ├── dr.bd.tcl
│   │   │   ├── kernel_info.dat
│   │   │   ├── _kernel_inst_paths.dat
│   │   │   ├── kernel_service.json
│   │   │   ├── _new_clk_freq
│   │   │   ├── sdsl.dat
│   │   │   ├── syslinkConfig.ini
│   │   │   ├── systemDiagramModel.json
│   │   │   ├── systemDiagramModelSlrBaseAddress.json
│   │   │   ├── system.hdf
│   │   │   ├── vadd_build.rtd
│   │   │   ├── vadd.rtd
│   │   │   ├── vadd.xml
│   │   │   ├── vadd_xml.rtd
│   │   │   ├── vplConfig.ini
│   │   │   ├── vplsettings.json
│   │   │   ├── xclbin_orig.1.xml
│   │   │   ├── xclbin_orig.xml
│   │   │   ├── xclbin_orig.xml.tmp
│   │   │   └── xo
│   │   │       ├── ip_repo
│   │   │       └── vadd
│   │   │           └── vadd
│   │   │               ├── cpu_sources
│   │   │               │   └── vadd.cpp
│   │   │               ├── debug
│   │   │               │   ├── vadd_Pipeline_VITIS_LOOP_13_1.xrf
│   │   │               │   ├── vadd.protoinst
│   │   │               │   └── vadd.xrf
│   │   │               ├── kernel.xml
│   │   │               └── vadd.design.xml
│   │   ├── link.spr
│   │   ├── link.steps.log
│   │   ├── run_link
│   │   │   ├── gen_run.xml
│   │   │   ├── htr.txt
│   │   │   └── vpl.pb
│   │   ├── sys_link
│   │   │   ├── cfgraph
│   │   │   │   └── cfgen_cfgraph.xml
│   │   │   ├── dr.xml
│   │   │   ├── hw_emu.hpfm
│   │   │   ├── iprepo
│   │   │   │   ├── temp
│   │   │   │   │   └── xo0
│   │   │   │   │       ├── ip_repo
│   │   │   │   │       ├── vadd
│   │   │   │   │       │   ├── cpu_sources
│   │   │   │   │       │   │   └── vadd.cpp
│   │   │   │   │       │   ├── debug
│   │   │   │   │       │   │   ├── vadd_Pipeline_VITIS_LOOP_13_1.xrf
│   │   │   │   │       │   │   ├── vadd.protoinst
│   │   │   │   │       │   │   └── vadd.xrf
│   │   │   │   │       │   └── kernel.xml
│   │   │   │   │       └── xo.xml
│   │   │   │   └── xilinx_com_hls_vadd_1_0
│   │   │   │       ├── auxiliary.xml
│   │   │   │       ├── component.xml
```


constraints
vadd_ooc.xdc
doc
ReleaseNotes.txt
drivers
vadd_v1_0
data
vadd.mdd
vadd.tcl
src
Makefile
xvadd.c
xvadd.h
xvadd_hw.h
xvadd_linux.c
xvadd_sinit.c
example
ipi_example.sh
ipi_example.tcl
hdl
verilog
vadd_aximm1_m_axi.v
vadd_aximm2_m_axi.v
vadd_control_s_axi.v
vadd_flow_control_loop_pipe_sequential_init.v
vadd.v
vadd_vadd_Pipeline_VITIS_LOOP_13_1.v
vhdl
vadd_aximm1_m_axi.vhd
vadd_aximm2_m_axi.vhd
vadd_control_s_axi.vhd
vadd_flow_control_loop_pipe_sequential_init.vhd
vadd_vadd_Pipeline_VITIS_LOOP_13_1.vhd
vadd.vhd
hls_files
vadd
solution
solution.aps
solution_data.json
solution.directive
syn
report
csynth.rpt
csynth.xml
vadd_csynth.rpt
vadd_Pipeline_VITIS_LOOP_13_1_csynth.rpt
misc
sysgen_info.xml
vadd.fcnmap.xml
vadd_info.xml
xgui
vadd_v1_0.tcl
sc_emu_debug.tcl
sdsl.dat
_sysl
vivado
vivado.spr
vpl
gen_run.xml
htr.txt
ipirun.tcl
ISEWrap.js

```
│               ├── ISEWrap.sh
│               ├── link_ip_guidance.txt
│               ├── openprj.tcl
│               ├── output
│               │   ├── insert_debug_profiling.tcl
│               │   ├── pfm_dynamic_ooc_copy.xdc
│               │   ├── _post_sys_link_gen_constrs.xdc
│               │   └── vivado_warning.txt
│               ├── prj
│               │   ├── prj.cache
│               │   ├── prj.gen
│               │   ├── prj.hw
│               │   ├── prj.ip_user_files
│               │   ├── prj.sim
│               │   │   └── sim_1
│               │   │       └── behav_waveform
│               │   │           └── xsim
│               │   ├── prj.srcs
│               │   │   └── sources_1
│               │   │       ├── bd
│               │   │       └── imports
│               │   │           └── hdl
│               │   │               └── pfm_top_wrapper.v
│               │   └── prj.xpr
│               ├── rundef.js
│               ├── runme.bat
│               ├── runme.log
│               ├── runme.sh
│               ├── scripts
│               │   └── _vivado_params.tcl
│               ├── vivado_config_hw_emu.tcl
│               ├── vivado.jou
│               ├── vivado.log
│               ├── vivado.pb
│               └── vpl.tcl
├── logs
│   ├── link
│   │   ├── link.steps.log
│   │   └── v++.log
│   ├── optraceViewer.html
│   └── vadd
│       ├── vadd.steps.log
│       ├── vadd_vitis_hls.log
│       └── v++.log
├── reports
│   ├── link
│   │   ├── automation_summary.txt
│   │   ├── automation_summary_update_bd.txt
│   │   ├── system_estimate_vadd.xtxt
│   │   └── v++_link_vadd_guidance.html
│   └── vadd
│       ├── hls_reports
│       │   └── vadd_csynth.rpt
│       ├── system_estimate_vadd.xtxt
│       └── v++_compile_vadd_guidance.html
├── vadd
│   ├── vadd
│   │   ├── htr.txt
│   │   ├── ISEWrap.js
│   │   ├── ISEWrap.sh
│   │   ├── rundef.js
│   │   ├── runme.bat
│   │   ├── runme.log
```

```
│   │   ├── runme.sh
│   │   ├── vadd
│   │   │   ├── hls.app
│   │   │   ├── ip
│   │   │   │   ├── auxiliary.xml
│   │   │   │   ├── component.xml
│   │   │   │   ├── constraints
│   │   │   │   │   └── vadd_ooc.xdc
│   │   │   │   ├── doc
│   │   │   │   │   └── ReleaseNotes.txt
│   │   │   │   ├── drivers
│   │   │   │   │   └── vadd_v1_0
│   │   │   │   │       ├── data
│   │   │   │   │       │   ├── vadd.mdd
│   │   │   │   │       │   └── vadd.tcl
│   │   │   │   │       └── src
│   │   │   │   │           ├── Makefile
│   │   │   │   │           ├── xvadd.c
│   │   │   │   │           ├── xvadd.h
│   │   │   │   │           ├── xvadd_hw.h
│   │   │   │   │           ├── xvadd_linux.c
│   │   │   │   │           └── xvadd_sinit.c
│   │   │   │   ├── example
│   │   │   │   │   ├── ipi_example.sh
│   │   │   │   │   └── ipi_example.tcl
│   │   │   │   ├── hdl
│   │   │   │   │   ├── verilog
│   │   │   │   │   │   ├── vadd_aximm1_m_axi.v
│   │   │   │   │   │   ├── vadd_aximm2_m_axi.v
│   │   │   │   │   │   ├── vadd_control_s_axi.v
│   │   │   │   │   │   ├── vadd_flow_control_loop_pipe_sequential_init.v
│   │   │   │   │   │   ├── vadd.v
│   │   │   │   │   │   └── vadd_vadd_Pipeline_VITIS_LOOP_13_1.v
│   │   │   │   │   └── vhdl
│   │   │   │   │       ├── vadd_aximm1_m_axi.vhd
│   │   │   │   │       ├── vadd_aximm2_m_axi.vhd
│   │   │   │   │       ├── vadd_control_s_axi.vhd
│   │   │   │   │       ├── vadd_flow_control_loop_pipe_sequential_init.vhd
│   │   │   │   │       ├── vadd_vadd_Pipeline_VITIS_LOOP_13_1.vhd
│   │   │   │   │       └── vadd.vhd
│   │   │   │   ├── hls_files
│   │   │   │   │   └── vadd
│   │   │   │   │       └── solution
│   │   │   │   │           ├── solution.aps
│   │   │   │   │           ├── solution_data.json
│   │   │   │   │           ├── solution.directive
│   │   │   │   │           └── syn
│   │   │   │   │               └── report
│   │   │   │   │                   ├── csynth.rpt
│   │   │   │   │                   ├── csynth.xml
│   │   │   │   │                   ├── vadd_csynth.rpt
│   │   │   │   │                   └──
vadd_Pipeline_VITIS_LOOP_13_1_csynth.rpt
│   │   │   │   ├── misc
│   │   │   │   ├── sysgen_info.xml
│   │   │   │   ├── vadd_info.xml
│   │   │   │   └── xgui
│   │   │   │       └── vadd_v1_0.tcl
│   │   │   ├── kernel.xml
│   │   │   ├── kernel.xml.orig
│   │   │   ├── solution
│   │   │   │   ├── impl
│   │   │   │   │   ├── export.xo
```

```
│   │   │   │   │   │   ├── ip
│   │   │   │   │   │   ├── verilog
│   │   │   │   │   │   └── vhdl
│   │   │   │   │   ├── solution.aps
│   │   │   │   │   ├── solution_data.json
│   │   │   │   │   ├── solution.directive
│   │   │   │   │   ├── solution.log
│   │   │   │   │   └── syn
│   │   │   │   └── vadd.design.xml
│   │   │   ├── vadd.tcl
│   │   │   ├── vitis_hls.log
│   │   │   └── vitis_hls.pb
│   │   ├── vadd.spr
│   │   └── vadd.steps.log
│   ├── v++_compile_vadd_guidance.pb3
│   └── v++_link_vadd_guidance.pb3
```

Output Directories of v++ -c --mode aie

The directory structure generated by the `v++ -c --mode aie` command has been organized to let you easily find and access files from the project.

The example directory provided below results from the following command lines:

```
v++ -c --mode aie --target hw --config ./workAIE/aie_sys_design_aie/
aiecompiler.cfg \
--platform $XILINX_VITIS/base_platforms/xilinx_vck190_base_202420_1/
xilinx_vck190_base_202420_1.xpfm \
--work_dir ./newAIE/hw/Work ./workAIE/aie_sys_design_aie/src/graph.cpp
```

Where `aie_config.cfg.cfg` file defines the following options:

```
include=./workAIE/aie_sys_design_aie
include=./workAIE/aie_sys_design_aie/../aie_sys_design_common/src
include=./workAIE/aie_sys_design_aie/src

[aie]
Xchess=main:darts.xargs=-nb
log-level=1
stacksize=1024
heapsize=1024
```

An example of the AI Engine component directory output of the `v++` command follows:

```
newAIE
└── hw
    └── Work
        ├── aie
        │   ├── 24_0
        │   │   ├── 24_0.log
        │   │   ├── Release
        │   │   │   ├── 24_0
        │   │   │   ├── 24_0.#
        │   │   │   ├── 24_0.##
        │   │   │   ├── 24_0.calltree
        │   │   │   ├── 24_0.cmic2
        │   │   │   ├── 24_0.cmico
        │   │   │   ├── 24_0.lst
```

```
│   │   │   │   ├── 24_0.map
│   │   │   │   ├── 24_0.o
│   │   │   │   ├── 24_0.o.lst
│   │   │   │   ├── 24_0.sdr
│   │   │   │   ├── 24_0.srv
│   │   │   │   ├── 24_0.txt
│   │   │   │   └── chesswork
│   │   │   │       ├── 24_0.aliases
│   │   │   │       ├── 24_0.ctt
│   │   │   │       ├── 24_0.dti
│   │   │   │       ├── 24_0.fnm
│   │   │   │       ├── 24_0-
F_Z17fir_27t_sym_hb_2iP12input_windowI6cint16EP13output_windowIS0_E_.#
│   │   │   │       ├── 24_0-
F_Z17fir_27t_sym_hb_2iP12input_windowI6cint16EP13output_windowIS0_E_.o
│   │   │   │       ├── 24_0-
F_Z17fir_27t_sym_hb_2iP12input_windowI6cint16EP13output_windowIS0_E__ra.lib
│   │   │   │       ├── 24_0-
F_Z17fir_27t_sym_hb_2iP12input_windowI6cint16EP13output_windowIS0_E_.sfg
│   │   │   │       ├── 24_0.gvt
│   │   │   │       ├── 24_0.gvt.#
│   │   │   │       ├── 24_0.gvt.o
│   │   │   │       ├── 24_0.ini
│   │   │   │       ├── 24_0.lib
│   │   │   │       ├── 24_0-main_.#
│   │   │   │       ├── 24_0-main_.o
│   │   │   │       ├── 24_0-main__ra.lib
│   │   │   │       ├── 24_0-main_.sfg
│   │   │   │       ├── 24_0.objlist
│   │   │   │       └── 24_0.sfg
│   │   │   ├── scripts
│   │   │   │   ├── 24_0.bcf
│   │   │   │   └── 24_0.prx
│   │   │   ├── src
│   │   │   │   └── 24_0.cc
│   │   │   ├── timestamped_log
│   │   │   │   └── 24_0-2023-09-01-12-08-47.log
│   │   │   └── xlopt.log
│   │   ├── 25_0
│   │   │   ├── 25_0.log
│   │   │   ├── Release
│   │   │   │   ├── 25_0
│   │   │   │   ├── 25_0.#
│   │   │   │   ├── 25_0.##
│   │   │   │   ├── 25_0.calltree
│   │   │   │   ├── 25_0.cmic2
│   │   │   │   ├── 25_0.cmico
│   │   │   │   ├── 25_0.lst
│   │   │   │   ├── 25_0.map
│   │   │   │   ├── 25_0.o
│   │   │   │   ├── 25_0.o.lst
│   │   │   │   ├── 25_0.sdr
│   │   │   │   ├── 25_0.srv
│   │   │   │   ├── 25_0.txt
│   │   │   │   └── chesswork
│   │   │   │       ├── 25_0.aliases
│   │   │   │       ├── 25_0-classifier_.#
│   │   │   │       ├── 25_0-classifier_.o
│   │   │   │       ├── 25_0-classifier__ra.lib
│   │   │   │       ├── 25_0-classifier_.sfg
│   │   │   │       ├── 25_0.ctt
│   │   │   │       ├── 25_0.dti
│   │   │   │       ├── 25_0.fnm
```

```
│   │   │   │   │   ├── 25_0.gvt
│   │   │   │   │   ├── 25_0.gvt.#
│   │   │   │   │   ├── 25_0.gvt.o
│   │   │   │   │   ├── 25_0.ini
│   │   │   │   │   ├── 25_0.lib
│   │   │   │   │   ├── 25_0-main_.#
│   │   │   │   │   ├── 25_0-main_.o
│   │   │   │   │   ├── 25_0-main__ra.lib
│   │   │   │   │   ├── 25_0-main_.sfg
│   │   │   │   │   ├── 25_0.objlist
│   │   │   │   │   └── 25_0.sfg
│   │   │   │   ├── scripts
│   │   │   │   │   ├── 25_0.bcf
│   │   │   │   │   └── 25_0.prx
│   │   │   │   ├── src
│   │   │   │   │   └── 25_0.cc
│   │   │   │   ├── timestamped_log
│   │   │   │   │   └── 25_0-2023-09-01-12-08-47.log
│   │   │   │   └── xlopt.log
│   │   │   ├── active_cores.json
│   │   │   ├── AddressSpace.txt
│   │   │   ├── AliasAnalysisReport.txt
│   │   │   ├── ir
│   │   │   │   ├── 24_0_dependence.json
│   │   │   │   ├── 24_0_guidance.json
│   │   │   │   ├── 24_0.ll
│   │   │   │   ├── 24_0_main.ll
│   │   │   │   ├── 24_0_orig.ll
│   │   │   │   ├── 25_0_dependence.json
│   │   │   │   ├── 25_0_guidance.json
│   │   │   │   ├── 25_0.ll
│   │   │   │   ├── 25_0_main.ll
│   │   │   │   ├── 25_0_orig.ll
│   │   │   │   ├── empty.cc
│   │   │   │   ├── _header.ll
│   │   │   │   ├── i4_hb27_2i_analysis.json
│   │   │   │   ├── i4_hb27_2i_dependence_guidance.json
│   │   │   │   ├── i4_hb27_2i.ll
│   │   │   │   ├── i4_hb27_2i_results.json
│   │   │   │   ├── i4_hb27_2i_spec.json
│   │   │   │   ├── i5_classify_analysis.json
│   │   │   │   ├── i5_classify_dependence_guidance.json
│   │   │   │   ├── i5_classify.ll
│   │   │   │   ├── i5_classify_results.json
│   │   │   │   ├── i5_classify_spec.json
│   │   │   │   └── Makefile
│   │   │   └── Makefile
│   │   ├── aie_hw.cfg
│   │   ├── arch
│   │   │   ├── aie_interface.aieintfcst
│   │   │   ├── aie_partition.json
│   │   │   ├── aie_pl_intf.json
│   │   │   ├── aieshim_solution.aiesol
│   │   │   ├── cfgraph.xml
│   │   │   └── logical_arch_aie.larch
│   │   ├── config
│   │   │   ├── aie_partition_id.json
│   │   │   ├── aie_resources.bin
│   │   │   ├── aiesim_config.txt
│   │   │   └── scsim_config.json
│   │   ├── graph.aiecompile_summary
│   │   ├── logs
│   │   │   ├── aie_hw.log
```

```
│   ├── Work.steps.log
│   └── xcd.log
├── ps
│   ├── cdo
│   │   ├── aie_cdo_clock_gating.bin
│   │   ├── aie_cdo_debug.bin
│   │   ├── aie_cdo_elfs.bin
│   │   ├── aie_cdo_enable.bin
│   │   ├── aie_cdo_error_handling.bin
│   │   ├── aie_cdo_init.bin
│   │   ├── aie_cdo_mem_clear.bin
│   │   ├── aie_cdo_reset.bin
│   │   ├── generateAIEConfig
│   │   ├── generated-objects
│   │   │   ├── cdo_main.o
│   │   │   ├── cdo_main.out
│   │   │   └── gen_cdo.o
│   │   ├── generated-source
│   │   │   ├── cdo_main.cpp
│   │   │   ├── gen_cdo.cpp
│   │   │   └── gen_cdo.h
│   │   └── Makefile
│   └── c_rts
│       ├── aie_control_config.json
│       ├── aie_control.cpp
│       ├── aie_control_xrt.cpp
│       └── systemC
│           ├── generated-objects
│           │   ├── genwrapper_for_ps_i6.o
│           │   └── ps_i6.so
│           ├── generated-source
│           │   └── genwrapper_for_ps_i6.cpp
│           ├── Makefile
│           ├── Makefile.questa
│           ├── Makefile.riviera
│           ├── Makefile.vcs
│           ├── Makefile.xcelium
│           └── Makefile.xsim
├── reports
│   ├── compiler_report.json
│   ├── complexity.csv
│   ├── dma_lock_report.json
│   ├── DMA_report.txt
│   ├── FIFO_guidance_report.txt
│   ├── graph.dot
│   ├── graph_mapped_post.png
│   ├── graph_mapping_analysis_report.txt
│   ├── graph.pdf
│   ├── graph.png
│   ├── graph.xpe
│   ├── guidance.html
│   ├── guidance.pb3
│   ├── Interposer_analysis_report.txt
│   ├── lock_allocation_report.json
│   ├── Lock_report.txt
│   ├── Stream_report.txt
│   ├── sync_buffer_address.json
│   └── v++_compile_Work_guidance.html
├── temp
│   ├── cfg.o
│   ├── debug_config.json
│   ├── DMA_FIFO_info_report.json
│   ├── dummy.cpp
```

```
│   ├── graph_aie_constraints.aiecst
│   ├── graph_aie_constraints_for_placer.aiecst
│   ├── graph_aie_full_netlist.aiexn
│   ├── graph_aieir_dump.txt
│   ├── graph_aie_mapped.aiecst
│   ├── graph_aie_mapped.aiesol
│   ├── graph_aie_netlist.aiexn
│   ├── graph_aie_partitioned.aiesol
│   ├── graph_aie_premapped.aiesol
│   ├── graph_aie_prerouted.aiesol
│   ├── graph_aie_routed.aiecst
│   ├── graph_aie_routed.aiesol
│   ├── graph.ii
│   ├── graph.json
│   ├── graph_link_design.tcl
│   ├── graph_mapped.json
│   ├── graph_mapped_post.dot
│   ├── graph.out
│   ├── graph_partition.json
│   ├── graph.processed.ii
│   ├── hw.o
│   ├── log4cfg
│   ├── router.input
│   ├── router_soln.dot
│   ├── router_soln.json
│   ├── router_soln.pdf
│   ├── router_soln.png
│   └── sw.o
└── Work.spr
```

Output Directories of v++ -c --mode hls

The directory structure generated by the `v++ -c --mode hls` command has been organized to let you easily find and access files from the project.

The example directory provided below results from the following command lines:

```
v++ -c --mode hls --config ./workDCT/dct/hls_config.cfg --work_dir newDCT
```

Where `hls_config.cfg.cfg` file defines the following options:

```
part=xczu7ev-ffvc1156-2-e

[hls]
syn.file=/reference-files/src/dct.cpp
tb.file=/reference-files/src/out.golden.dat
tb.file=/reference-files/src/in.dat
tb.file=/reference-files/src/dct_test.cpp
tb.file=/reference-files/src/dct_coeff_table.txt
syn.top=dct
clock=8ns
clock_uncertainty=12%
csim.code_analyzer=1
csim.clean=true
```

```
flow_target=vitis
package.output.format=xo
syn.directive.pipeline=dct_2d II=4
package.output.syn=1
syn.compile.pipeline_loops=6
```

An example of the `temp_dir` output of the `v++` link command follows:

```
newDCT
├── hls
│   ├── config.cmdline
│   ├── hls_data.json
│   ├── impl
│   │   ├── misc
│   │   │   ├── drivers
│   │   │   │   └── dct_v1_0
│   │   │   │       ├── data
│   │   │   │       │   ├── dct.mdd
│   │   │   │       │   ├── dct.tcl
│   │   │   │       │   └── dct.yaml
│   │   │   │       └── src
│   │   │   │           ├── CMakeLists.txt
│   │   │   │           ├── Makefile
│   │   │   │           ├── xdct.c
│   │   │   │           ├── xdct.h
│   │   │   │           ├── xdct_hw.h
│   │   │   │           ├── xdct_linux.c
│   │   │   │           └── xdct_sinit.c
│   │   │   └── logo.png
│   │   ├── verilog
│   │   │   ├── dct_ama_addmuladd_18s_16s_13ns_29ns_29_4_1.v
│   │   │   ├── dct_ama_submuladd_16s_16s_12ns_29s_29_4_1.v
│   │   │   ├── dct_ama_submuladd_16s_16s_13ns_29s_29_4_1.v
│   │   │   ├── dct_ama_submuladd_18s_16s_14ns_29ns_29_4_1.v
│   │   │   ├── dct_buf_2d_in_RAM_AUTO_1R1W_memcore.v
│   │   │   ├── dct_buf_2d_in_RAM_AUTO_1R1W.v
│   │   │   ├── dct_buf_2d_out_RAM_AUTO_1R1W_memcore.v
│   │   │   ├── dct_buf_2d_out_RAM_AUTO_1R1W.v
│   │   │   ├── dct_control_s_axi.v
│   │   │   ├── dct_dct_2d.v
│   │   │   ├── dct_entry_proc.v
│   │   │   ├── dct_fifo_w64_d6_S.v
│   │   │   ├── dct_flow_control_loop_pipe_sequential_init.v
│   │   │   ├── dct_gmem_m_axi.v
│   │   │   ├── dct_mac_muladd_16s_14ns_29ns_29_4_1.v
│   │   │   ├── dct_mac_muladd_16s_14ns_29s_29_4_1.v
│   │   │   ├── dct_mac_muladd_16s_15s_13ns_29_4_1.v
│   │   │   ├── dct_mac_muladd_16s_15s_29ns_29_4_1.v
│   │   │   ├── dct_mac_muladd_16s_15s_29s_29_4_1.v
│   │   │   ├── dct_mac_muladd_17s_12ns_13ns_29_4_1.v
│   │   │   ├── dct_mac_muladd_17s_12ns_29s_29_4_1.v
│   │   │   ├── dct_mac_muladd_17s_13ns_13ns_29_4_1.v
│   │   │   ├── dct_mac_muladd_17s_13ns_29s_29_4_1.v
│   │   │   ├── dct_mac_muladd_18s_13ns_13ns_29_4_1.v
│   │   │   ├── dct_mac_muladd_18s_14ns_13ns_29_4_1.v
│   │   │   ├── dct_mul_16s_15ns_29_1_1.v
│   │   │   ├── dct_mul_16s_15s_29_1_1.v
│   │   │   ├── dct_mul_17s_13ns_29_1_1.v
│   │   │   ├── dct_mul_17s_14ns_29_1_1.v
│   │   │   ├── dct_read_data_Pipeline_RD_Loop_Col.v
│   │   │   ├── dct_read_data.v
│   │   │   ├── dct_sparsemux_17_3_16_1_1.v
```

```
│   │   │   ├── dct.v
│   │   │   ├── dct_write_data_Pipeline_WR_Loop_Col.v
│   │   │   └── dct_write_data.v
│   │   └── vhdl
│   │       ├── dct_ama_addmuladd_18s_16s_13ns_29ns_29_4_1.vhd
│   │       ├── dct_ama_submuladd_16s_16s_12ns_29s_29_4_1.vhd
│   │       ├── dct_ama_submuladd_16s_16s_13ns_29s_29_4_1.vhd
│   │       ├── dct_ama_submuladd_18s_16s_14ns_29ns_29_4_1.vhd
│   │       ├── dct_buf_2d_in_RAM_AUTO_1R1W_memcore.vhd
│   │       ├── dct_buf_2d_in_RAM_AUTO_1R1W.vhd
│   │       ├── dct_buf_2d_out_RAM_AUTO_1R1W_memcore.vhd
│   │       ├── dct_buf_2d_out_RAM_AUTO_1R1W.vhd
│   │       ├── dct_control_s_axi.vhd
│   │       ├── dct_dct_2d.vhd
│   │       ├── dct_entry_proc.vhd
│   │       ├── dct_fifo_w64_d6_S.vhd
│   │       ├── dct_flow_control_loop_pipe_sequential_init.vhd
│   │       ├── dct_gmem_m_axi.vhd
│   │       ├── dct_mac_muladd_16s_14ns_29ns_29_4_1.vhd
│   │       ├── dct_mac_muladd_16s_14ns_29s_29_4_1.vhd
│   │       ├── dct_mac_muladd_16s_15s_13ns_29_4_1.vhd
│   │       ├── dct_mac_muladd_16s_15s_29ns_29_4_1.vhd
│   │       ├── dct_mac_muladd_16s_15s_29s_29_4_1.vhd
│   │       ├── dct_mac_muladd_17s_12ns_13ns_29_4_1.vhd
│   │       ├── dct_mac_muladd_17s_12ns_29s_29_4_1.vhd
│   │       ├── dct_mac_muladd_17s_13ns_13ns_29_4_1.vhd
│   │       ├── dct_mac_muladd_17s_13ns_29s_29_4_1.vhd
│   │       ├── dct_mac_muladd_18s_13ns_13ns_29_4_1.vhd
│   │       ├── dct_mac_muladd_18s_14ns_13ns_29_4_1.vhd
│   │       ├── dct_mul_16s_15ns_29_1_1.vhd
│   │       ├── dct_mul_16s_15s_29_1_1.vhd
│   │       ├── dct_mul_17s_13ns_29_1_1.vhd
│   │       ├── dct_mul_17s_14ns_29_1_1.vhd
│   │       ├── dct_read_data_Pipeline_RD_Loop_Col.vhd
│   │       ├── dct_read_data.vhd
│   │       ├── dct_sparsemux_17_3_16_1_1.vhd
│   │       ├── dct.vhd
│   │       ├── dct_write_data_Pipeline_WR_Loop_Col.vhd
│   │       └── dct_write_data.vhd
│   ├── kernel.xml
│   └── syn
│       ├── report
│       │   ├── csynth_design_size.rpt
│       │   ├── csynth_design_size.xml
│       │   ├── csynth.rpt
│       │   ├── csynth.xml
│       │   ├── dct_2d_csynth.rpt
│       │   ├── dct_2d_csynth.xml
│       │   ├── dct_csynth.rpt
│       │   ├── dct_csynth.xml
│       │   ├── entry_proc_csynth.rpt
│       │   ├── entry_proc_csynth.xml
│       │   ├── read_data_csynth.rpt
│       │   ├── read_data_csynth.xml
│       │   ├── read_data_Pipeline_RD_Loop_Col_csynth.rpt
│       │   ├── read_data_Pipeline_RD_Loop_Col_csynth.xml
│       │   ├── write_data_csynth.rpt
│       │   ├── write_data_csynth.xml
│       │   ├── write_data_Pipeline_WR_Loop_Col_csynth.rpt
│       │   └── write_data_Pipeline_WR_Loop_Col_csynth.xml
│       ├── verilog
│       │   ├── dct_ama_addmuladd_18s_16s_13ns_29ns_29_4_1.v
│       │   ├── dct_ama_submuladd_16s_16s_12ns_29s_29_4_1.v
```

```
│   │   ├── dct_ama_submuladd_16s_16s_13ns_29s_29_4_1.v
│   │   ├── dct_ama_submuladd_18s_16s_14ns_29ns_29_4_1.v
│   │   ├── dct_buf_2d_in_RAM_AUTO_1R1W_memcore.v
│   │   ├── dct_buf_2d_in_RAM_AUTO_1R1W.v
│   │   ├── dct_buf_2d_out_RAM_AUTO_1R1W_memcore.v
│   │   ├── dct_buf_2d_out_RAM_AUTO_1R1W.v
│   │   ├── dct_control_s_axi.v
│   │   ├── dct_dct_2d.v
│   │   ├── dct_entry_proc.v
│   │   ├── dct_fifo_w64_d6_S.v
│   │   ├── dct_flow_control_loop_pipe_sequential_init.v
│   │   ├── dct_gmem_m_axi.v
│   │   ├── dct_mac_muladd_16s_14ns_29ns_29_4_1.v
│   │   ├── dct_mac_muladd_16s_14ns_29s_29_4_1.v
│   │   ├── dct_mac_muladd_16s_15s_13ns_29_4_1.v
│   │   ├── dct_mac_muladd_16s_15s_29ns_29_4_1.v
│   │   ├── dct_mac_muladd_16s_15s_29s_29_4_1.v
│   │   ├── dct_mac_muladd_17s_12ns_13ns_29_4_1.v
│   │   ├── dct_mac_muladd_17s_12ns_29s_29_4_1.v
│   │   ├── dct_mac_muladd_17s_13ns_13ns_29_4_1.v
│   │   ├── dct_mac_muladd_17s_13ns_29s_29_4_1.v
│   │   ├── dct_mac_muladd_18s_13ns_13ns_29_4_1.v
│   │   ├── dct_mac_muladd_18s_14ns_13ns_29_4_1.v
│   │   ├── dct_mul_16s_15ns_29_1_1.v
│   │   ├── dct_mul_16s_15s_29_1_1.v
│   │   ├── dct_mul_17s_13ns_29_1_1.v
│   │   ├── dct_mul_17s_14ns_29_1_1.v
│   │   ├── dct_read_data_Pipeline_RD_Loop_Col.v
│   │   ├── dct_read_data.v
│   │   ├── dct_sparsemux_17_3_16_1_1.v
│   │   ├── dct.v
│   │   ├── dct_write_data_Pipeline_WR_Loop_Col.v
│   │   └── dct_write_data.v
│   └── vhdl
│       ├── dct_ama_addmuladd_18s_16s_13ns_29ns_29_4_1.vhd
│       ├── dct_ama_submuladd_16s_16s_12ns_29s_29_4_1.vhd
│       ├── dct_ama_submuladd_16s_16s_13ns_29s_29_4_1.vhd
│       ├── dct_ama_submuladd_18s_16s_14ns_29ns_29_4_1.vhd
│       ├── dct_buf_2d_in_RAM_AUTO_1R1W_memcore.vhd
│       ├── dct_buf_2d_in_RAM_AUTO_1R1W.vhd
│       ├── dct_buf_2d_out_RAM_AUTO_1R1W_memcore.vhd
│       ├── dct_buf_2d_out_RAM_AUTO_1R1W.vhd
│       ├── dct_control_s_axi.vhd
│       ├── dct_dct_2d.vhd
│       ├── dct_entry_proc.vhd
│       ├── dct_fifo_w64_d6_S.vhd
│       ├── dct_flow_control_loop_pipe_sequential_init.vhd
│       ├── dct_gmem_m_axi.vhd
│       ├── dct_mac_muladd_16s_14ns_29ns_29_4_1.vhd
│       ├── dct_mac_muladd_16s_14ns_29s_29_4_1.vhd
│       ├── dct_mac_muladd_16s_15s_13ns_29_4_1.vhd
│       ├── dct_mac_muladd_16s_15s_29ns_29_4_1.vhd
│       ├── dct_mac_muladd_16s_15s_29s_29_4_1.vhd
│       ├── dct_mac_muladd_17s_12ns_13ns_29_4_1.vhd
│       ├── dct_mac_muladd_17s_12ns_29s_29_4_1.vhd
│       ├── dct_mac_muladd_17s_13ns_13ns_29_4_1.vhd
│       ├── dct_mac_muladd_17s_13ns_29s_29_4_1.vhd
│       ├── dct_mac_muladd_18s_13ns_13ns_29_4_1.vhd
│       ├── dct_mac_muladd_18s_14ns_13ns_29_4_1.vhd
│       ├── dct_mul_16s_15ns_29_1_1.vhd
│       ├── dct_mul_16s_15s_29_1_1.vhd
│       ├── dct_mul_17s_13ns_29_1_1.vhd
│       ├── dct_mul_17s_14ns_29_1_1.vhd
```

```
│   │   ├── dct_read_data_Pipeline_RD_Loop_Col.vhd
│   │   ├── dct_read_data.vhd
│   │   ├── dct_sparsemux_17_3_16_1_1.vhd
│   │   ├── dct.vhd
│   │   ├── dct_write_data_Pipeline_WR_Loop_Col.vhd
│   │   └── dct_write_data.vhd
├── logs
│   ├── hls_compile.log
│   ├── newDCT.steps.log
│   └── xcd.log
├── newDCT.hlscompile_summary
├── reports
│   ├── hls_compile.rpt
│   └── v++_compile_newDCT_guidance.html
└── vitis-comp.json
```

Output Directories of the Vitis Unified IDE

Unlike the command-line flow, which is defined largely by the user through command or Makefile, the AMD Vitis™ IDE defines the structure of the projects and output directories in a system design project. In the Vitis IDE, an Application project for acceleration or heterogeneous design can have four different projects associated with it. The projects include:

- Top-level System project: This is the project used to build the integrated system design, and is where you will find the consolidated contents of the packaged system design which is the result of the `v++ --package` process
- Software Application project: This contains the source and compiled results of the software application which runs on an x86 or Arm processor
- PL Kernels project: This project contains the source files for one or more PL kernels used by the system, and the results of the `v++ --compile` command
- Hardware Linking project: This project contains the linked system design which is the results of the `v++ --link` command

TIP: *Each of the projects below starts with the `Emulation-HW` folder which is where the Vitis IDE places the build content for the hardware emulation build. The files presented here are for that build target. Some of the files at the deeper levels are not displayed, but can be found by navigating into the hierarchy of completed builds.*

Top-Level System Project

The top-level system project contains all of the other projects as sub-projects. This is where the definition and construction of the system design are provided. This includes the output of the package process, which generates the final fixed platform and device binary, and packages it into an SD card or QSPI image.

```
├── Emulation-HW
│   ├── binary_container_1.xclbin
│   ├── binary_container_1.xclbin.package_summary
│   ├── emulation.pid
│   ├── makefile
```

```
│   ├── package
│   │   ├── emu_qemu_scripts
│   │   │   ├── start_pmc.sh
│   │   │   └── start_qemu.sh
│   │   ├── launch_hw_emu.sh
│   │   ├── pmu_args.txt
│   │   ├── qemu_args.txt
│   │   ├── qemu_output.log
│   │   ├── qemu_resize_img.sh
│   │   ├── sd_card
│   │   │   ├── boot.scr
│   │   │   ├── emconfig.json
│   │   │   ├── Image
│   │   │   ├── vaddTest
│   │   │   └── vadd.xclbin
│   │   ├── sd_card.img
│   │   └── sim
│   │       └── behav_waveform
│   │           └── xsim
│   │               ├── xsim.ini
│   │               ├── xsim.ini.bak
│   │               ├── xsim.jou
│   │               ├── xvhdl.log
│   │               ├── xvhdl.pb
│   │               ├── xvlog.log
│   │               └── xvlog.pb
│   ├── package.build
│   │   ├── logs
│   │   │   └── package
│   │   │       ├── package.steps.log
│   │   │       └── v++.log
│   │   ├── package
│   │   │   ├── behav.xse
│   │   │   ├── extractedSystemDiagram.json
│   │   │   ├── packagedSystemDiagram.json
│   │   │   ├── package.spr
│   │   │   ├── package.steps.log
│   │   │   └── sim
│   │   │       └── behav_waveform
│   │   │           └── xsim
│   │   │               ├── xsim.ini
│   │   │               ├── xsim.ini.bak
│   │   │               ├── xvhdl.log
│   │   │               ├── xvhdl.pb
│   │   │               ├── xvlog.log
│   │   │               └── xvlog.pb
│   │   ├── reports
│   │   │   └── package
│   │   │       └── v++_package_vadd_guidance.html
│   │   ├── v++_package_binary_container_1_guidance.json
│   │   ├── v++_package_binary_container_1_guidance.pb
│   │   ├── v++_package_vadd_guidance.json
│   │   └── v++_package_vadd_guidance.pb
│   ├── package.cfg
│   ├── vaddTest_system_Emulation-HW.build.ui.log
│   ├── vadd.xclbin
│   ├── vadd.xclbin.package_summary
│   ├── v++_binary_container_1.log
│   ├── v++_vadd.log
│   ├── xcd.log
│   └── xrc.log
├── _ide
│   └── launch
```

```
        ├── SystemDebugger_vaddTest_system_1.launch
        └── SystemDebugger_vaddTest_system.launch
├── vadd-system.txt
└── vaddTest_system.sprj
```

Software Application Project

The software application project contains the source code for the software application, and the resulting `.o` object files, and `.exe` or `.elf` executable files.

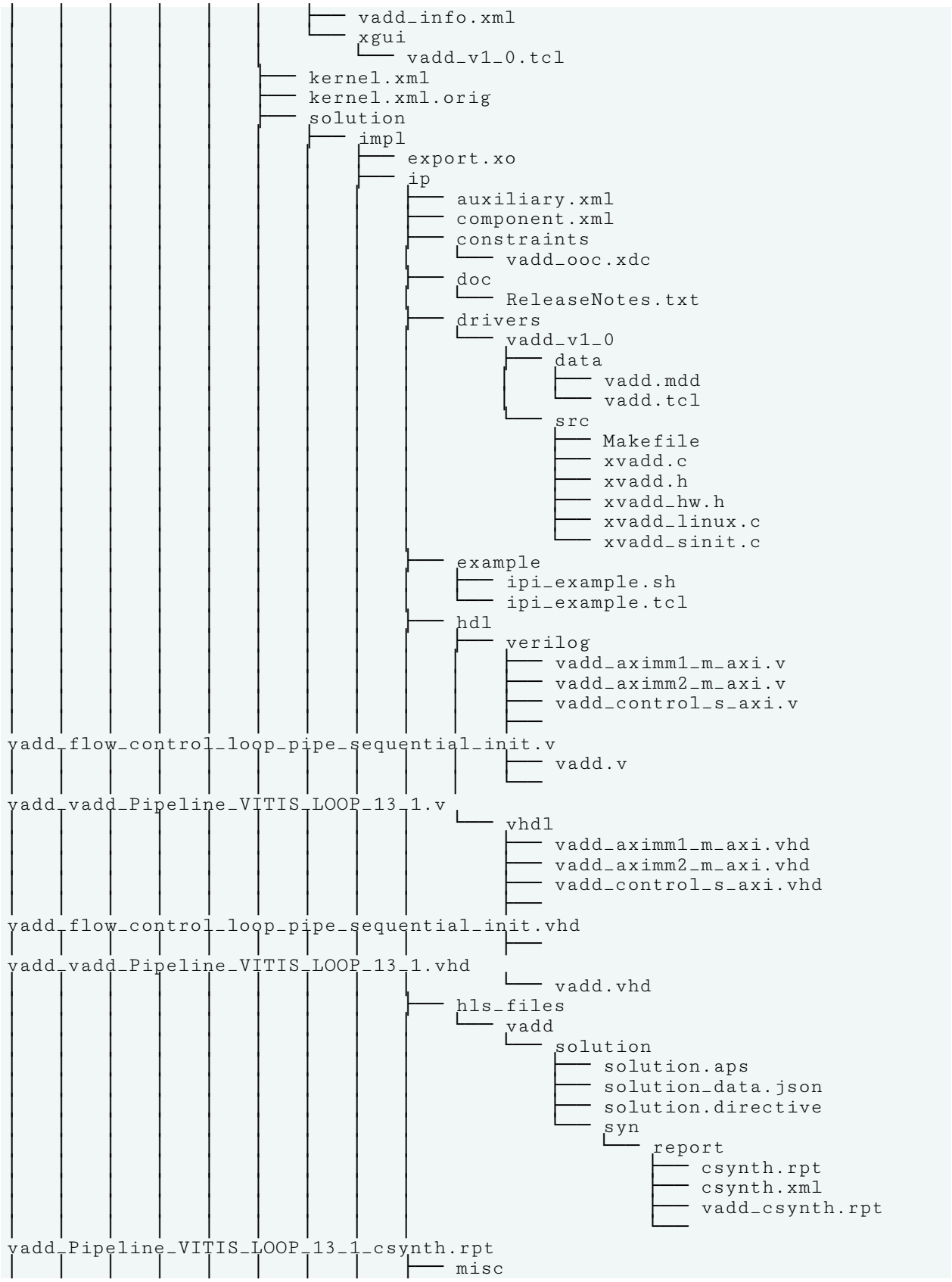
```
├── Emulation-HW
│   ├── emconfig.json
│   ├── guidance.html
│   ├── guidance.json
│   ├── guidance.pb
│   ├── makeemconfig.mk
│   ├── makefile
│   ├── src
│   │   └── host.o
│   ├── SystemDebugger_vaddTest_system_vaddTest
│   │   ├── device_trace_0.csv
│   │   ├── native_trace.csv
│   │   ├── summary.csv
│   │   ├── xrt.ini
│   │   └── xrt.run_summary
│   ├── SystemDebugger_vaddTest_system_vaddTest.launch.log
│   ├── vaddTest
│   ├── vaddTest_Emulation-HW.build.ui.log
│   └── xsa.xml
├── src
│   └── host.cpp
├── vadd-processor.txt
└── vaddTest.prj
```

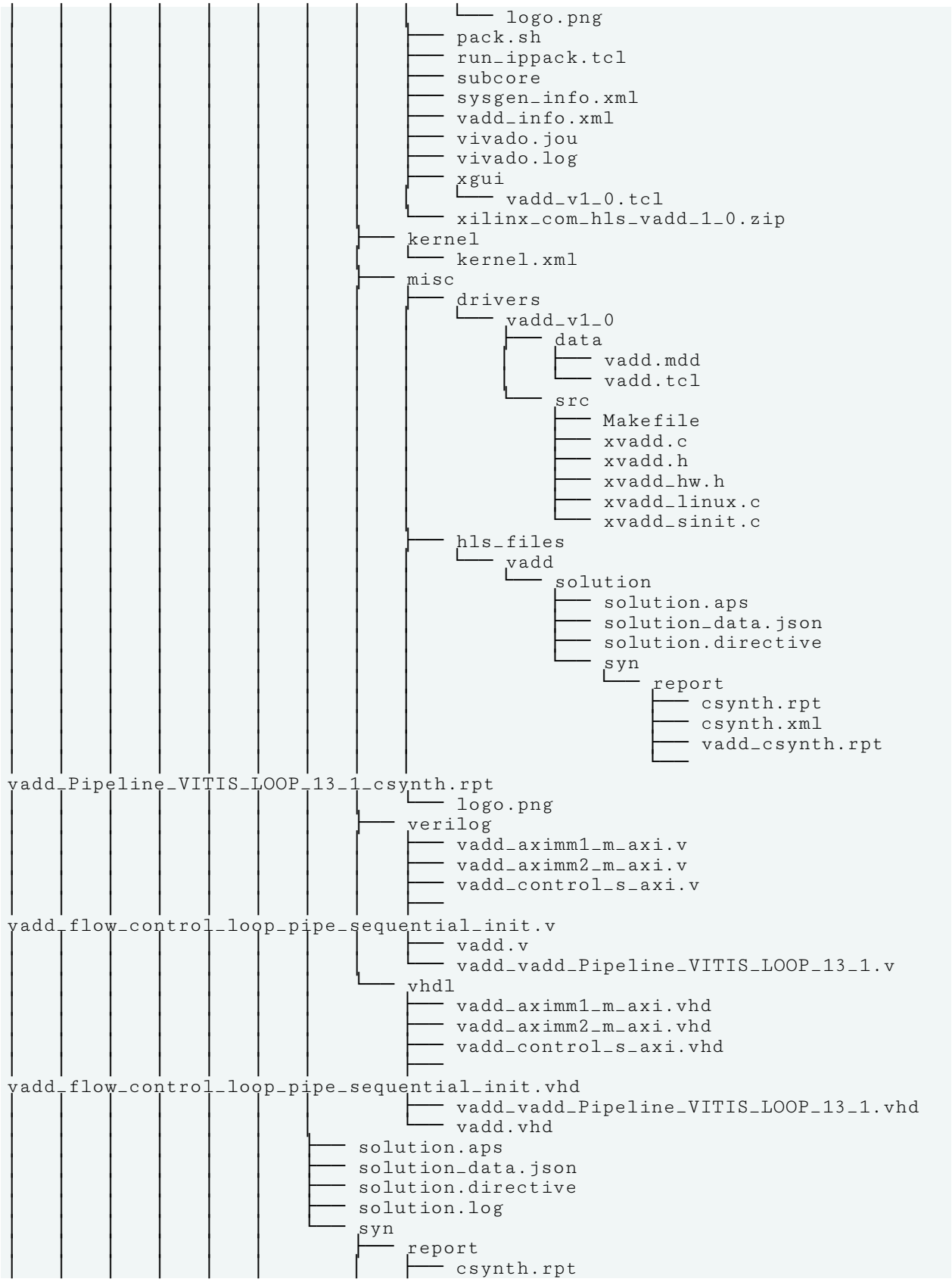
PL Kernels Project

The PL kernels project contains the source code for C++ kernels and the compiled Xilinx object files (`.xo`), as well as RTL kernels (`.xo`), and `libadf.a` files containing AI Engine graph applications. The directory also holds the resulting logs and projects required by Vitis HLS to build the PL kernel objects, such as the `.compile_summary` produced during compilation.

```
├── Emulation-HW
│   ├── build
│   │   ├── logs
│   │   │   ├── vadd
│   │   │   │   ├── vadd.steps.log
│   │   │   │   ├── vadd_vitis_hls.log
│   │   │   │   └── v++.log
│   │   │   └── v++_vadd.log
│   │   ├── reports
│   │   │   └── vadd
│   │   │       ├── hls_reports
│   │   │       │   └── vadd_csynth.rpt
│   │   │       └── system_estimate_vadd.xtxt
│   │   ├── vadd
│   │   │   ├── vadd
│   │   │   │   ├── htr.txt
```

ISEWrap.js
ISEWrap.sh
rundef.js
runme.bat
runme.log
runme.sh
vadd
hls.app
ip
auxiliary.xml
component.xml
constraints
vadd_ooc.xdc
doc
ReleaseNotes.txt
drivers
vadd_v1_0
data
vadd.mdd
vadd.tcl
src
Makefile
xvadd.c
xvadd.h
xvadd_hw.h
xvadd_linux.c
xvadd_sinit.c
example
ipi_example.sh
ipi_example.tcl
hdl
verilog
vadd_aximm1_m_axi.v
vadd_aximm2_m_axi.v
vadd_control_s_axi.v
vadd_flow_control_loop_pipe_sequential_init.v
vadd.v
vadd_vadd_Pipeline_VITIS_LOOP_13_1.v
vhdl
vadd_aximm1_m_axi.vhd
vadd_aximm2_m_axi.vhd
vadd_control_s_axi.vhd
vadd_flow_control_loop_pipe_sequential_init.vhd
vadd_vadd_Pipeline_VITIS_LOOP_13_1.vhd
vadd.vhd
hls_files
vadd
solution
solution.aps
solution_data.json
solution.directive
syn
report
csynth.rpt
csynth.xml
vadd_csynth.rpt
vadd_Pipeline_VITIS_LOOP_13_1_csynth.rpt
misc
logo.png
sysgen_info.xml


vadd_info.xml
xgui
vadd_v1_0.tcl
kernel.xml
kernel.xml.orig
solution
impl
export.xo
ip
auxiliary.xml
component.xml
constraints
vadd_ooc.xdc
doc
ReleaseNotes.txt
drivers
vadd_v1_0
data
vadd.mdd
vadd.tcl
src
Makefile
xvadd.c
xvadd.h
xvadd_hw.h
xvadd_linux.c
xvadd_sinit.c
example
ipi_example.sh
ipi_example.tcl
hdl
verilog
vadd_aximm1_m_axi.v
vadd_aximm2_m_axi.v
vadd_control_s_axi.v
vadd_flow_control_loop_pipe_sequential_init.v
vadd.v
vadd_vadd_Pipeline_VITIS_LOOP_13_1.v
vhdl
vadd_aximm1_m_axi.vhd
vadd_aximm2_m_axi.vhd
vadd_control_s_axi.vhd
vadd_flow_control_loop_pipe_sequential_init.vhd
vadd_vadd_Pipeline_VITIS_LOOP_13_1.vhd
vadd.vhd
hls_files
vadd
solution
solution.aps
solution_data.json
solution.directive
syn
report
csynth.rpt
csynth.xml
vadd_csynth.rpt
vadd_Pipeline_VITIS_LOOP_13_1_csynth.rpt
misc


logo.png
pack.sh
run_ippack.tcl
subcore
sysgen_info.xml
vadd_info.xml
vivado.jou
vivado.log
xgui
vadd_v1_0.tcl
xilinx_com_hls_vadd_1_0.zip
kernel
kernel.xml
misc
drivers
vadd_v1_0
data
vadd.mdd
vadd.tcl
src
Makefile
xvadd.c
xvadd.h
xvadd_hw.h
xvadd_linux.c
xvadd_sinit.c
hls_files
vadd
solution
solution.aps
solution_data.json
solution.directive
syn
report
csynth.rpt
csynth.xml
vadd_csynth.rpt
vadd_Pipeline_VITIS_LOOP_13_1_csynth.rpt
logo.png
verilog
vadd_aximm1_m_axi.v
vadd_aximm2_m_axi.v
vadd_control_s_axi.v
vadd_flow_control_loop_pipe_sequential_init.v
vadd.v
vadd_vadd_Pipeline_VITIS_LOOP_13_1.v
vhdl
vadd_aximm1_m_axi.vhd
vadd_aximm2_m_axi.vhd
vadd_control_s_axi.vhd
vadd_flow_control_loop_pipe_sequential_init.vhd
vadd_vadd_Pipeline_VITIS_LOOP_13_1.vhd
vadd.vhd
solution.aps
solution_data.json
solution.directive
solution.log
syn
report
csynth.rpt

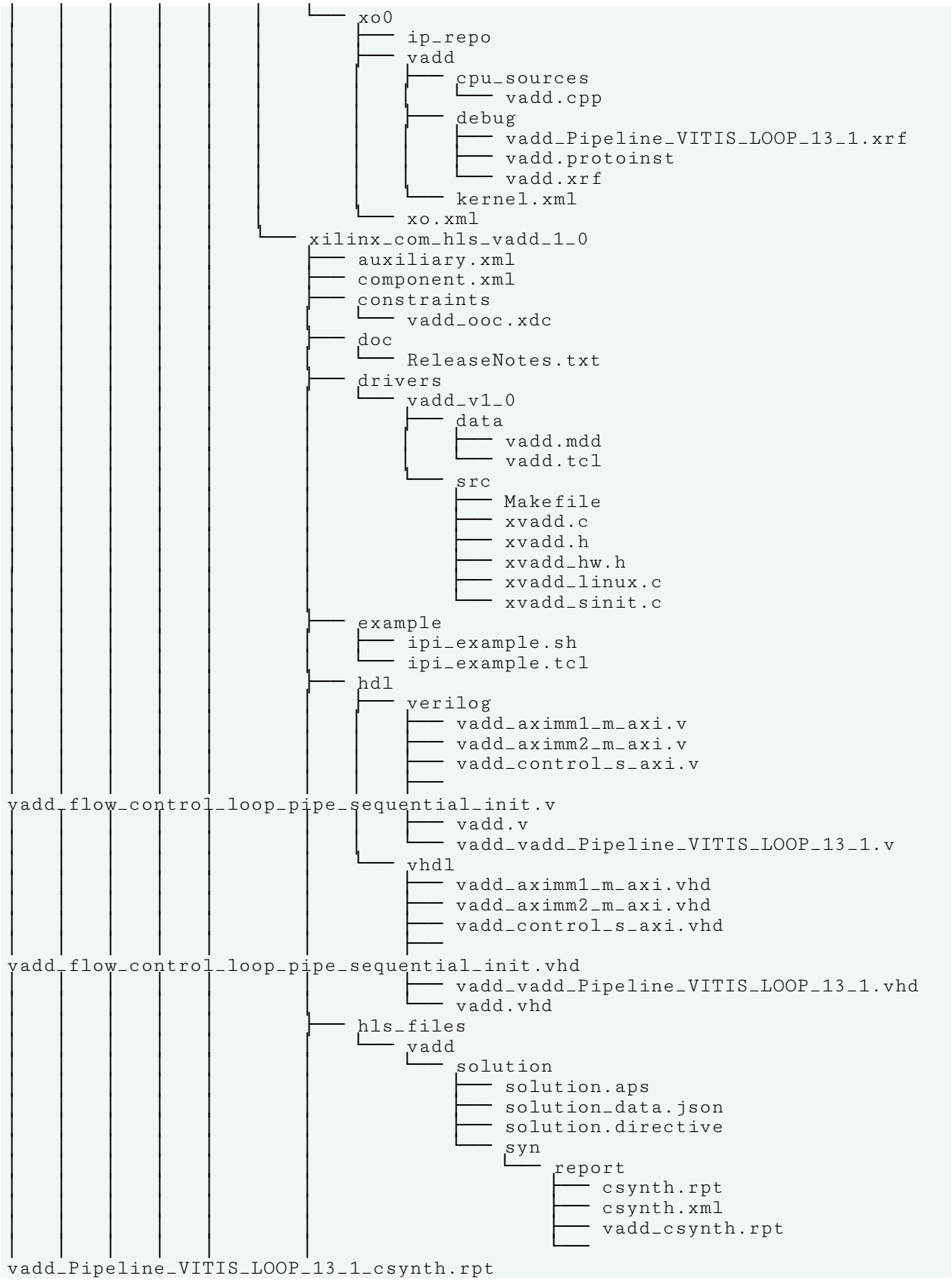
```
│   │   │   │   │   │   │   │   ├── csynth.xml
│   │   │   │   │   │   │   │   ├── vadd_csynth.rpt
│   │   │   │   │   │   │   │   ├── vadd_csynth.xml
│   │   │   │   │   │   │   │   ├── vadd_Pipeline_VITIS_LOOP_13_1_csynth.rpt
│   │   │   │   │   │   │   │   └── vadd_Pipeline_VITIS_LOOP_13_1_csynth.xml
│   │   │   │   │   │   │   ├── verilog
│   │   │   │   │   │   │   │   ├── vadd_aximm1_m_axi.v
│   │   │   │   │   │   │   │   ├── vadd_aximm2_m_axi.v
│   │   │   │   │   │   │   │   ├── vadd_control_s_axi.v
│   │   │   │   │   │   │   │   ├──
vadd_flow_control_loop_pipe_sequential_init.v
│   │   │   │   │   │   │   │   ├── vadd.v
│   │   │   │   │   │   │   │   └── vadd_vadd_Pipeline_VITIS_LOOP_13_1.v
│   │   │   │   │   │   │   └── vhdl
│   │   │   │   │   │   │   │   ├── vadd_aximm1_m_axi.vhd
│   │   │   │   │   │   │   │   ├── vadd_aximm2_m_axi.vhd
│   │   │   │   │   │   │   │   ├── vadd_control_s_axi.vhd
│   │   │   │   │   │   │   │   ├──
vadd_flow_control_loop_pipe_sequential_init.vhd
│   │   │   │   │   │   │   │   ├── vadd_vadd_Pipeline_VITIS_LOOP_13_1.vhd
│   │   │   │   │   │   │   │   └── vadd.vhd
│   │   │   │   │   │   │   └── vadd.design.xml
│   │   │   │   │   │   ├── vadd.tcl
│   │   │   │   │   │   ├── vitis_hls.log
│   │   │   │   │   │   └── vitis_hls.pb
│   │   │   │   │   ├── vadd.spr
│   │   │   │   │   └── vadd.steps.log
│   │   │   │   ├── vadd.mdb
│   │   │   │   ├── vadd.xo
│   │   │   │   └── vadd.xo.compile_summary
│   │   │   ├── guidance.html
│   │   │   ├── guidance.html.bak
│   │   │   ├── guidance.json
│   │   │   ├── guidance.pb
│   │   │   ├── guidance.pb.bak
│   │   │   ├── makefile
│   │   │   ├── vadd-compile.cfg
│   │   │   ├── vaddTest_kernels_Emulation-HW.build.ui.log
│   │   │   └── xcd.log
│   ├── src
│   │   └── vadd.cpp
│   ├── tree-kernel.txt
│   └── vaddTest_kernels.prj
```

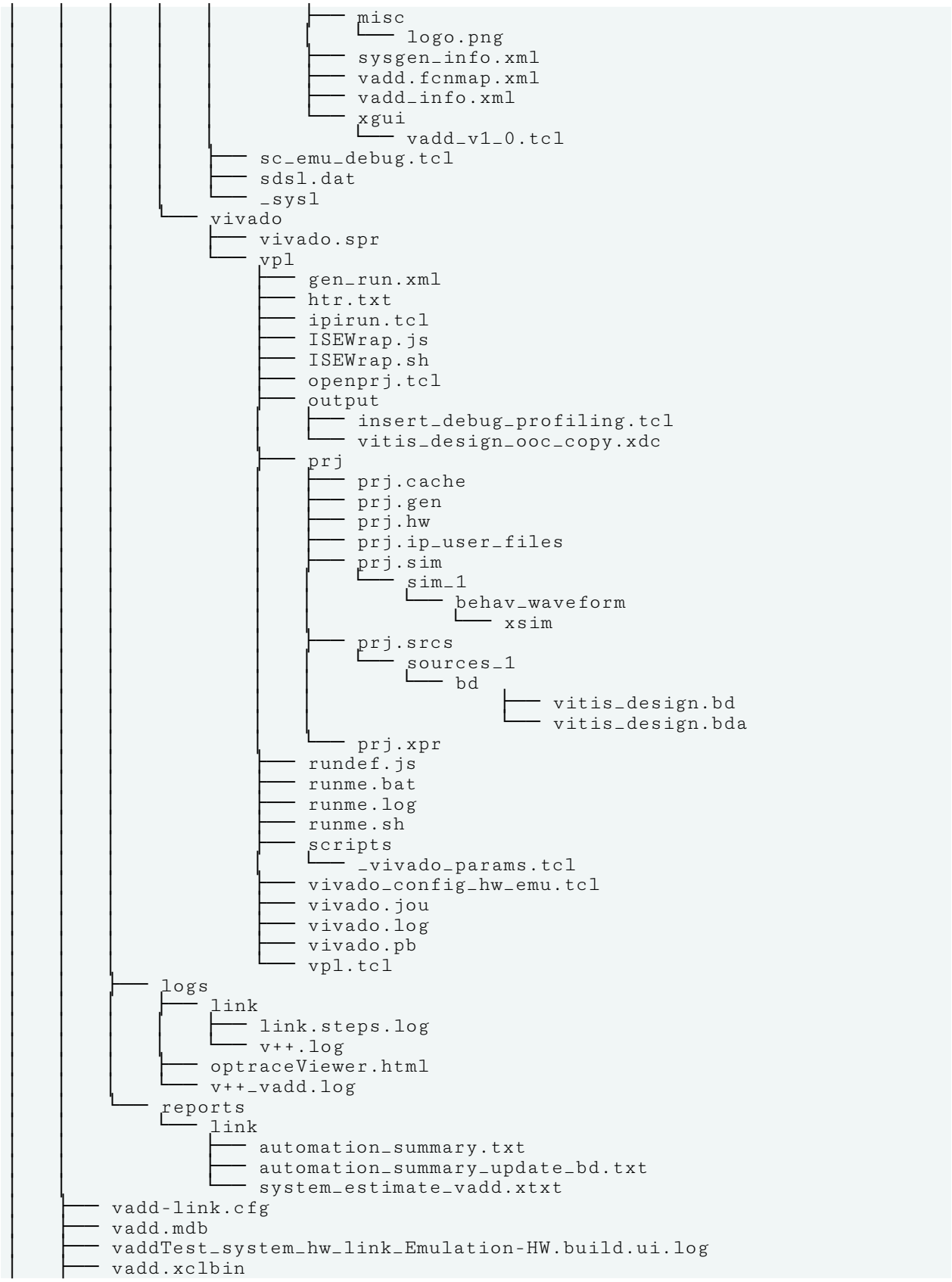
Hardware Link Project

The hardware link project contains the linked fixed hardware platform (`.xsa`) and the device binary (`.xclbin`) used to program the AMD device. The directory also holds the resulting logs and projects produced during the linking process, such as the `.link_summary` that can be viewed in the Vitis analyzer.

```
├── Emulation-HW
│   ├── binary_container_1-link.cfg
│   ├── guidance.html
│   ├── guidance.html.bak
│   ├── guidance.json
│   ├── guidance.pb
│   ├── guidance.pb.bak
│   ├── makefile
```

```
├── vadd.build
│   ├── link
│   │   ├── activetask.json
│   │   ├── int
│   │   │   ├── address_map.xml
│   │   │   ├── automation_summary.txt
│   │   │   ├── automation_summary_update_bd.txt
│   │   │   ├── behav_waveform
│   │   │   │   └── xsim
│   │   │   │       ├── xsim.ini
│   │   │   │       ├── xsim.ini.bak
│   │   │   │       ├── xvhdl.log
│   │   │   │       ├── xvhdl.pb
│   │   │   │       ├── xvlog.log
│   │   │   │       └── xvlog.pb
│   │   │   ├── behav.xse
│   │   │   ├── cf2sw_full.rtd
│   │   │   ├── cf2sw.rtd
│   │   │   ├── debug_ip_layout.rtd
│   │   │   ├── dr.bd.tcl
│   │   │   ├── kernel_info.dat
│   │   │   ├── _kernel_inst_paths.dat
│   │   │   ├── kernel_service.json
│   │   │   ├── _new_clk_freq
│   │   │   ├── sdsl.dat
│   │   │   ├── syslinkConfig.ini
│   │   │   ├── systemDiagramModel.json
│   │   │   ├── systemDiagramModelSlrBaseAddress.json
│   │   │   ├── system.hdf
│   │   │   ├── vadd_build.rtd
│   │   │   ├── vadd.rtd
│   │   │   ├── vadd.xml
│   │   │   ├── vadd_xml.rtd
│   │   │   ├── vplConfig.ini
│   │   │   ├── vplsettings.json
│   │   │   ├── xclbin_orig.1.xml
│   │   │   ├── xclbin_orig.xml
│   │   │   ├── xclbin_orig.xml.tmp
│   │   │   └── xo
│   │   │       ├── ip_repo
│   │   │       └── vadd
│   │   │           └── vadd
│   │   │               ├── cpu_sources
│   │   │               │   └── vadd.cpp
│   │   │               ├── debug
│   │   │               │   ├── vadd_Pipeline_VITIS_LOOP_13_1.xrf
│   │   │               │   ├── vadd.protoinst
│   │   │               │   └── vadd.xrf
│   │   │               ├── kernel.xml
│   │   │               └── vadd.design.xml
│   │   ├── link.spr
│   │   ├── link.steps.log
│   │   ├── run_link
│   │   │   ├── gen_run.xml
│   │   │   ├── htr.txt
│   │   │   └── vpl.pb
│   │   ├── sys_link
│   │   │   ├── cfgraph
│   │   │   │   └── cfgen_cfgraph.xml
│   │   │   ├── dr.xml
│   │   │   ├── hw_emu.hpfm
│   │   │   ├── iprepo
│   │   │   │   ├── temp
```


xo0
ip_repo
vadd
cpu_sources
vadd.cpp
debug
vadd_Pipeline_VITIS_LOOP_13_1.xrf
vadd.protoinst
vadd.xrf
kernel.xml
xo.xml
xilinx_com_hls_vadd_1_0
auxiliary.xml
component.xml
constraints
vadd_ooc.xdc
doc
ReleaseNotes.txt
drivers
vadd_v1_0
data
vadd.mdd
vadd.tcl
src
Makefile
xvadd.c
xvadd.h
xvadd_hw.h
xvadd_linux.c
xvadd_sinit.c
example
ipi_example.sh
ipi_example.tcl
hdl
verilog
vadd_aximm1_m_axi.v
vadd_aximm2_m_axi.v
vadd_control_s_axi.v
vadd_flow_control_loop_pipe_sequential_init.v
vadd.v
vadd_vadd_Pipeline_VITIS_LOOP_13_1.v
vhdl
vadd_aximm1_m_axi.vhd
vadd_aximm2_m_axi.vhd
vadd_control_s_axi.vhd
vadd_flow_control_loop_pipe_sequential_init.vhd
vadd_vadd_Pipeline_VITIS_LOOP_13_1.vhd
vadd.vhd
hls_files
vadd
solution
solution.aps
solution_data.json
solution.directive
syn
report
csynth.rpt
csynth.xml
vadd_csynth.rpt
vadd_Pipeline_VITIS_LOOP_13_1_csynth.rpt


misc
logo.png
sysgen_info.xml
vadd.fcnmap.xml
vadd_info.xml
xgui
vadd_v1_0.tcl
sc_emu_debug.tcl
sdsl.dat
_sysl
vivado
vivado.spr
vpl
gen_run.xml
htr.txt
ipirun.tcl
ISEWrap.js
ISEWrap.sh
openprj.tcl
output
insert_debug_profiling.tcl
vitis_design_ooc_copy.xdc
prj
prj.cache
prj.gen
prj.hw
prj.ip_user_files
prj.sim
sim_1
behav_waveform
xsim
prj.srcs
sources_1
bd
vitis_design.bd
vitis_design.bda
prj.xpr
rundef.js
runme.bat
runme.log
runme.sh
scripts
_vivado_params.tcl
vivado_config_hw_emu.tcl
vivado.jou
vivado.log
vivado.pb
vpl.tcl
logs
link
link.steps.log
v++.log
optraceViewer.html
v++_vadd.log
reports
link
automation_summary.txt
automation_summary_update_bd.txt
system_estimate_vadd.xtxt
vadd-link.cfg
vadd.mdb
vaddTest_system_hw_link_Emulation-HW.build.ui.log
vadd.xclbin

```
│   ├── vadd.xclbin.info
│   ├── vadd.xclbin.link_summary
│   ├── vadd.xclbin.sh
│   └── xcd.log
├── vadd-hw-link.txt
└── vaddTest_system_hw_link.prj
```

Python XSDB Commands

This section describes the Python XSDB commands. You can use these commands to implement the functions supported by XSDB. After the list of commands is a section with several usage examples.

Breakpoints

The following are the commands followed by their options (if any) and the Python method for the option (if any).

bpadd

- **-addr <breakpoint-address>:**

```
t = self.session.targets(id=2)
b1 = t.bpadd(addr='main')
```

- **-file <file-name>:**

```
t.bpadd('helloworld.c:90', type='hw', mode=3)
t.bpadd(file='helloworld.c', line=89)
```

- **-line <line-number>:**

```
t.bpadd(file='helloworld.c', line=89)
```

- **-type <breakpoint-type>:**

```
t.bpadd(addr='main', type='hw', mode=2)
```

- **-mode <breakpoint-mode>:**

```
t.bpadd(addr='main', type='hw', mode=2)
```

- **-enable <mode>:**

```
t = self.session.targets(id=2)
b1 = t.bpadd(addr='main', enable=1)
```

- **-ct-input <list> -ct-output <list>:**

```
t.bpadd(ct_input=0, ct_output=8, skip_on_step=1)
```

- **-skip-on-step <value>:**

```
t.bpadd(ct_input=0, ct_output=8, skip_on_step=1)
```

- **-target-id <id>:**

```
b1 = t.bpadd(addr='main', target_id='all')
```

bpdisable

The following are options for the command followed by their Python method.

- **<id-list>:**

```
t.bpdisable(bp_ids=1)
t.bpdisable(bp_ids=[1, 2])
```

- **-all:**

```
t.bpdisable('--all')
```

bpenable

- **<id-list>:**

```
t.bpenable(bp_ids=1)
t.bpenable(bp_ids=[1, 2])
```

- **-all:**

```
t.bpenable('--all')
```

bplist

The following is the command's Python method

```
t.bplist()
```

bpremove

- **<id-list>:**

```
t.bpremove(bp_ids=2)
t.bpremove(bp_ids=[1,2])
```

- **-all:**

```
t.bpremove('--all')
```

bpstatus

- **<id>:**

```
t.bpstatus(bp_id=3)
```

Connections

The following are the commands followed by their options (if any) and the Python method for the option (if any).

connect

- **-host <host name/ip>:**

```
s.connect(host="xhdbfarmrk9", port=3121)
```

- **-port <port num>:**

```
s.connect(host="xhdbfarmrk9", port=3121)
```

- **-url <url>:**

```
s.connect("--new", url="TCP:xhdbfarmrk9:3121")
```

- **-list:**

```
s.connect("--list")
```

- **-set <channel-id>:**

```
s.connect(set="tcfchan#0")
```

- **-new:**

```
s.connect("--new", url="TCP:xhdbfarmrk9:3121")
```

disconnect

```
s.disconnect()
```

- **<channel-id>:**

```
s.disconnect(chan="tcfchan#1")
```

gdbremote connect

- **-architecture:**

```
session.gdb_connect('localhost:3121')
```

gdbremote disconnect

- **[target-id]:**

```
session.gdb_disconnect(1)
```

targets

```
s.targets()
```

- **[target-id]:**

```
s.targets(1)
s.targets(id=1)
```

- **-set:**

```
s.targets("--set", filter="name =~ #1")
```

- **-nocase:**

```
s.targets("--no_case", filter="name =~ ARM*")
```

- **-filter:**

```
s.targets(filter="name =~ ARM*")
```

- **-target-properties:**

```
self.session.targets("--target_properties")
```

- **-timeout:**

Device

The following are the commands followed by their options (if any) and the Python method for the option (if any).

device authjtag

```
self.session.device_authjtag("tests/elfs/versal/vck190.pdi")
```

device program

```
self.session.device_program("tests/elfs/versal/vck190.pdi")
```

device status

```
self.session.device_status()
```

- **-jreg-name <jtag-register-name>:**

  ```
  s.device_status('error_status')
  ```

fpga

- **-file <bitstream-file>:**

  ```
  tgt.fpga(file='tests/elfs/zynq/zc702.bit')
  ```

- **--partial:**

  ```
  tgt.fpga('--partial', file='tests/elfs/zynq/zc702.bit')
  ```

- **--no_revision_check:**

  ```
  tgt.fpga('--no_revision_check', file='tests/elfs/zynq/zc702.bit')
  ```

- **--skip_compatibility_check:**

  ```
  tgt.fpga('--skip_compatibility_check', file='tests/elfs/zynq/zc702.bit')
  ```

- **--state:**

  ```
  tgt.fpga('--state')
  ```

- **--config_status:**

  ```
  tgt.fpga('--config_status')
  ```

- **--ir_status:**

  ```
  tgt.fpga('--ir_status')
  ```

- **--boot_status:**

  ```
  tgt.fpga('--boot_status')
  ```

- **--timer_status:**

  ```
  tgt.fpga('--timer_status')
  ```

- **--cor0_status:**

```
tgt.fpga('--cor0_status')
```

- **--cor1_status:**

```
tgt.fpga('--cor1_status')
```

- **--wbstart_status:**

```
tgt.fpga('--cor1_status')
```

Download

The following are the commands followed by their options (if any) and the Python method for the option (if any).

dow

```
t.dow('tests/elfs/zynq/core0zynq.elf')
```

- **--data:**

```
t.dow('tests/elfs/zynq/data', '-d', addr=0x10000000)
```

- **--clear:**

```
t.dow('tests/elfs/zynq/core0zynq.elf', '-c', relocate_sections=0x30000000)
```

- **--skip_tcm_clear:**

```
t.dow('--skip_tcm_clear', 'tests/elfs/zynq/core0zynq.elf')
```

```
--keepsym
```

- **--force:**

```
t.dow('tests/elfs/zynq/core0zynq.elf', '-f')
```

- **--bypass_cache_synq:**

```
t.dow('tests/elfs/zynq/core0zynq.elf', '-b')
```

- **relocate_sections:**

```
t.dow('tests/elfs/zynq/core0zynq.elf', '-c', relocate_sections=0x30000000)
```

- **-vaddr:**

```
t.dow('tests/elfs/zynq/core0zynq.elf', '-v')
```

verify

```
t.verify('tests/elfs/zynq/core0zynq.elf')
```

- **-data <file> <addr>:**

```
t.verify('tests/elfs/zynq/core0zynq.elf', '-d', addr=0x10000000)
```

- **-force:**

```
t.verify('tests/elfs/zynq/core0zynq.elf','-f')
```

- **-vaddr:**

```
t.verify('tests/elfs/zynq/core0zynq.elf','-v')
```

IPI

The following are the commands followed by their options (if any) and the Python method for the option (if any).

plm

- **copy-debug-log:**

```
s.plm_copy_debug_log(0x0)
```

- **set-debug-log:**

```
s.plm_set_debug_log(0x0, 0x4000)
```

- **set-log-level:**

```
s.plm_set_log_level(4)
```

- **log:**

```
s.plm_log(0x0, 0x4000)
f = open('tests/data/plm.log', 'w')
s.plm_log(0x0, 0x4000, handle=f)
```

pmc

```
s.pmc('generic', [0x10241, 0x1c000000], response_size=1)
s.pmc(cmd='generic', data=[0x1030115, 0xffff0000, 0x100], response_size=2)
```

- **-ipi:**

```
s.pmc(cmd='generic', data=[0x1030115, 0xffff0000, 0x100], ipi=5,
response_size=2)
```

JTAG

The following are the commands followed by their options (if any) and the Python method for the option (if any).

jtag claim

The following is the Python method.

```
jt.claim()
```

jtag device_properties

- **<idcode>:**

  ```
  jt.device_properties(0x6ba00477)
  ```

- **<key> <value> pairs:**

  ```
  props = {'idcode': 0x6ba00477, 'irlen':
  4}jt.device_properties(props=props)
  DONE
  ```

jtag disclaim

```
jt.disclaim()
```

jtag frequency

```
jt.frequency()
```

- **-list:**

  ```
  jt.frequency('-l')
  ```

- **<frequency>:**

  ```
  jt.frequency(freq)
  ```

jtag lock

```
jt.lock(100)
```

jtag sequence

- **state:**

```
s = self.session
jt3 = s.jtag_targets(3)
jseq = jt3.sequence()
jseq.state("RESET")
```

- **irshift:**

```
jseq.irshift(register='bypass', state="IRUPDATE")
```

- **drshift:**

```
jseq.drshift(capture=True, state="IDLE", tdi=0, bit_len=2)
```

- **delay:**

```
jseq.delay(100)
```

- **get_pin:**

```
jseq.get_pin('TDI')
```

- **set_pin:**

```
jseq.set_pin('TDI', 0)
```

- **atomic:**

```
jseq.atomic()
```

- **run:**

```
jseq.run()
```

- **clear:**

```
jseq.clear()
```

- **delete:**

```
del jseq
```

jtag servers

```
jt.servers()
```

- **-list:**

```
jt.servers('-l')
```

- **-format:**

```
jt.servers('-f')
```

- **-open <server>:**

```
jt.servers(open='xilinx-xvc:localhost:10200')
```

- **-close <server>:**

```
jt.servers(close='xilinx-xvc:localhost:10200')
```

jtag skew

```
jt.skew()
```

- **<clock-skew>:**

```
jt.skew()
```

jtag targets

```
self.session.jtag_targets()
```

- **[target-id]:**

```
self.session.jtag_targets(id=2)
```

- **-set:**

```
self.session.jtag_targets('-s', filter="name == xcvc1902")
```

- **-regexp:**
- **-nocase:**

```
self.session.jtag_targets('-n', filter="name !~ XCVC*")
```

- **-filter <filter-expression>:**

```
self.session.jtag_targets(filter="name == xcvc1902")
```

- **-target-properties:**

```
self.session.jtag_targets('-t')
```

- **-open:**

```
self.session.jtag_targets('-o')
```

- **-close:**

```
self.session.jtag_targets('-c')
```

- **-timeout <sec>:** No method available

jtag unlock

```
jt.unlock()
```

Memory

The following are the commands followed by their options (if any) and the Python method for the option (if any).

init_ps

```
init_data = ["mask_write 0 0x00001FFF 0x00000001",
    "mask_delay 1",
    "mask_poll 4 1 1"]
t.init_ps(init_data)
```

mask_poll

```
t.mask_poll(4,1,1)
```

mask_write

```
t.mask_write(0, 0xFF, 0xFFFF)
```

memap

- **addr <memory-address>:**

```
t.memmap(addr=0xFC000000, size=0x1000, flags=3)
```

- **alignment <bytes>:**

```
t.memmap(addr=0xFC000000, size=0x1000, flags=3, alignment=4)
```

- **size <memory-size>:**

```
t.memmap(addr=0xFC000000, size=0x1000, flags=3)
```

- **flags <protection-flags>:**

```
t.memmap(addr=0xFC000000, size=0x1000, flags=3)
```

- **--list:**

```
t.memmap('-l')
```

- **--clear:**

```
t.memmap('-c', addr=0xFC000000)
```

- **relocate_sections <addr>:**

```
t.memmap(file='tests/elfs/zynq/core0zynq.elf', ,
relocate_sections=0x20000000)
```

- **-osa:**

```
t.memmap(addr=0xFC000000, size=0x1000, flags=3, OSA=1)
```

mrd

```
t.mrd(0x00100000, word_size=4, size=10)
```

- **--force:**

```
t.mrd(0x00100002, '-f', word_size=4, size=10)
```

- **size <access-size>:**

```
t.mrd(0x00100000, word_size=4, size=10)
```

- **--value:**

```
x = t.mrd(0x100000, '-v', word_size=4, size=20)
```

- **--bin:**

```
t.mrd(0x00100000, '-b', word_size=8, size=20, file="tests/data/mrd.bin")
```

- **file <file-name>:**

```
t.mrd(0x00100000, '-b', word_size=8, size=20, file="tests/data/mrd.bin")
```

- **-address-space <name>:**

```
t.mrd(0x100, address_space="APR")
t.mrd(0x80090088, address_space="AP1")
```

- **-unaligned-access:**

```
t.mrd(0x00100000, '-u',word_size=4, size=10)
```

mwr

```
t.mwr(0x00100000, word_size=4, size=10, words=[0x01, 0x02, 0x03])
```

- **--force:**

  ```
  t.mwr(0x00100000, '-f',word_size=4, size=10, words=[0x01, 0x02, 0x03])
  ```

- **--bypass_cache_synq:**

  ```
  t.mwr(0x00100000, '-v',word_size=4, size=10, words=[0x01, 0x02, 0x03])
  ```

- **size <access-size>:**

  ```
  t.mwr(0x00100000, word_size=4, size=10, words=[0x01, 0x02, 0x03])
  ```

- **--bin:**

  ```
  t.mwr(0x00100000, '-b', word_size=4, size = 5, file="tests/data/mrd.bin")
  ```

- **file<file-name>:**

  ```
  t.mwr(0x00100000, '-b', word_size=4, size = 5, file="tests/data/mrd.bin")
  ```

- **-address-space <name>:**

  ```
  t.mwr(0x80090088, address_space="AP1", words=[0x03186004])
  ```

- **-unaligned-access:**

  ```
  t.mwr(0x00100000, '-u',word_size=4, size=10, words=[0x01, 0x02, 0x03])
  ```

osa

```
t.osa('--disable', file='tests/elfs/zynq/core0zynq.elf')
```

- **--disable:**

  ```
  t.osa('--disable', file='tests/elfs/zynq/core0zynq.elf')
  ```

- **--fast_exec:**

  ```
  t.osa('--fast_exec', file='tests/elfs/zynq/core0zynq.elf')
  ```

- **--fast_step:**

  ```
  t.osa('--fast_step', file='tests/elfs/zynq/core0zynq.elf')
  ```

Miscellaneous

The following are the commands followed by their options (if any) and the Python method for the option (if any).

configparams

- **<none>:**

```
s.configparams()
```

- **<parameter>:**

```
s.configparams("silent-mode")
```

- **<parameter> <value>:**

```
s.configparams("silent-mode", 1)
```

- **-all:**

```
s.configparams("--all")
```

loadhw

```
-hw
```

```
-list
```

```
-mem-ranges
[list {start1 end1} {start2 end2}]
```

loadipxact

```
<xml-file>
```

```
-clear
```

```
-list
```

mb_drrd

- **<cmd> <bitlen>:**

```
s.mb_drrd(3, 28)
```

- **-target-id<id>:** No method available

- **-user <bscan number>:**

```
s.mb_drrd(0x07, 288, user=bscan, which=which, target_id=target_id)
```

- **-which<instance>:** No method available

mb_drwr

- **<cmd> <data> <bitlen>:**

```
s.mb_drwr(1, 0x282, 10)
```

- **-target-id <id>:** No method available
- **-user <bscan number>:**

```
s.mb_drwr(0x71, value, 8, user=bscan, which=which)
```

- **-which <instance>:** No method available

mdm_drrd

- **<cmd> <bitlen>:**

```
s.mdm_drrd(0, 32)
```

- **-target-id <id>:** No method available
- **-user <bscan number>:** No method available

mdm_drwr

- **<cmd> <data> <bitlen>:**

```
s.mdm_drwr(8, 0x40, 8)
```

- **-target-id <id>:** No method available
- **-user <bscan number>:** No method available

version

```
s.version('-s')
```

xsdbserver disconnect

```
s.xsdbserver_disconnect()
```

xsdbserver start

```
s.xsdbserver_start()
```

- **-host <addr>:** No method available
- **-port <port>:** No method available

xsdbserver stop

```
s.xsdbserver_stop()
```

xsdbserver version

```
s.xsdbserver_version()
```

Registers

rrd

The following are the commands followed by their options (if any) and the Python method for the option (if any).

- **<none>:**

```
s.targets(2)
s.rrd()
```

- **<register group/name>:**

```
ta = s.targets(2)
# read using session object
s.rrd("usr")
s.rrd("usr r8")# read using target object
ta.rrd("mpcore icdipr ipr95")
```

- **-defs:**

```
s.targets(2)
s.rrd("--defs")
s.rrd("usr", "--defs")
```

- **-no-bits:**

```
s.targets(2)
s.rrd("cpsr", "--no_bits")
```

rwr

- **<register><value>:**

```
s.targets(2)
s.rwr('r0', 0x5555aaaa)
s.rwr('cpsr m', 0x13)
```

Reset

rst

The following are the commands followed by their options (if any) and the Python method for the option (if any).

```
s.rst('--stop', endianness='le', type='cores')
```

- **-processor:** No method available
- **-cores:** No method available
- **-dap:** No method available
- **system:** No method available
- **-srst:** No method available
- **-por:** No method available
- **-ps:** No method available
- **-stop:** No method available
- **- -start:**

```
s.rst('--start', type='cores')
```

- **-endianness <value>:**

```
s.rst('--stop', endianness='le', type='cores')
```

- **-code-endianness <value>:**

```
s.rst('--stop', code-endianness='le', type='cores')
```

- **-isa <isa-name>:**

```
s.rst('--stop', isa='ARM', type='cores')
```

- **-clear-registers:** No method available

- **-type <reset type>:**

```
s.rst(type='system')
```

Running

The following are the commands followed by their options (if any) and the Python method for the option (if any).

backtrace, bt

```
# Using session object
s.backtrace()
# Using target object
t.backtrace()
```

- **-maxframes <num>:**

```
# Using session object
s.backtrace(5)
# Using target object
t.backtrace(5)
```

con

```
# Using session
objects.con()
# Using target
objectt.con()
```

- **-addr <address>:** No method available
- **-block:** No method available
- **-timeout <sec>:** No method available

dis

```
t.dis(0x00100000)
```

- **<address>:**

```
t.dis(0x00100000)
```

- **<address> <num>:**

```
t.dis(0x00100000, 5)
```

locals

```
s.locals()
```

- **[variable-name]:**

```
s.locals(name="test")
# or
s.locals("test")
```

- **[variable-name [variable-value]]:**

```
s.locals(name="test", value=5)
# or
s.locals("test", 7)
```

- **-defs:**

```
s.locals("--defs")
s.locals("--defs", name="val")
```

- **-dict:**

```
s.locals("--dict")
s.locals("--dict", name="val")
```

mbprofile

- **-low <addr>:**

```
s.mbprofile(low='low', high='high')
```

- **-high <addr>:** No method available
- **-freq <value>:**

```
s.mbprofile('--count_instr', low='low', high='high')
```

- **-count-instr:** No method available
- **-cumulate:** No method available
- **-start:**

```
s.mbprofile('--start')
```

- **-stop:**

```
s.mbprofile('--stop', out="tests/elfs/mbprofile/gmon_inst_1.out")
```

- **-out <filename>:**

```
s.mbprofile('--stop', out="tests/elfs/mbprofile/gmon_inst_1.out")
```

mbtrace

- **-start:**

```
s.mbtrace('--start')
```

- **-stop:**

```
s.mbtrace('-f', '--stop', out="tests/elfs/mbprofile/gmon_trace.txt")
```

- **-con:**

```
s.mbtrace('-f', '--con', out="tests/elfs/mbprofile/gmon_trace.txt")
```

- **-stp:** No method available
- **-nxt:** No method available
- **-out <filename>:**

```
s.mbtrace('-f', '--con', out="tests/elfs/mbprofile/gmon_trace.txt")
```

- **-level <level>:** No method available
- **-halt:** No method available
- **-save:** No method available
- **-low <addr>:** No method available
- **-high <addr>:** No method available
- **-format <format>:** No method available

nxt

```
# Using session objects.nxt()
# Using target objectt.nxt()
```

- **[count]:**

```
# Using session objects.nxt(2)
# Using target objectt.nxt(2)
```

nxti

```
# Using session object
s.nxti()
# Using target object
t.nxti()
```

- **[count]:**

```
# Using session object
s.nxti(2)
# Using target object
t.nxti(2)
```

print

- **[expression]:**

```
s.print()
# or
s.print(expr="temp")
# or
s.print("temp")
# or
s.print("x + y - z")
```

- **[expression] <valvue>:**

```
s.print(expr="temp", value=5555)
# or
s.print("temp", 4444)
```

- **-add <expression>:**

```
s.print("--add", expr="temp")
# or
s.print("--add", expr="x + y")
```

- **-defs [expression]:**

```
s.print("--defs")
# or
s.print("--defs", expr="temp")
```

- **-dict [expression]:**

```
s.print("--dict")
# or
s.print("--dict", expr="temp")
```

- **-remove [expression]:**

```
s.print()
# or
s.print("--remove", expr="temp")
```

- **-set <expression>:** No method available

profile

- **-freq <sampling-freq>:**

```
s.profile(freq=10000, addr=0x30000000)
```

- **-scratchaddr <addr>:**

```
s.profile(freq=10000, addr=0x30000000)
```

- **-out <file-name>:**

```
s.profile(out="tests/data/gmon_inst.out")
```

state

```
state = tgt.state()
```

stop

```
# Using session object
s.stop()
# Using target object
t.stop()
```

stp

```
# Using session object
s.stp()
# Using target object
t.stp()
```

- **[count]:**

```
# Using session object
s.stp(2)
# Using target object
t.stp(2)
```

stpi

```
# Using session object
s.stpi()
# Using target object
t.stpi()
```

- **[count]:**

```
# Using session object
s.stpi(2)
# Using target object
t.stpi(2)
```

stpout

```
# Using session object
s.stpout()
# Using target object
t.stpout()
```

- **[count]:**

```
# Using session object
s.stpout(2)
# Using target object
t.stpout(2)
```

STAPL

The following are the commands followed by their options (if any) and the Python method for the option (if any).

stapl config

- **-out <filepath>:**

```
st = self.session.stapl()
st.config(out="tests/data/pystapl.stapl", scan_chain=[{'name':
'xcvc1902'}, {'name': 'xcvm1802'}])
```

- **-handle <filehandle>:**

```
st = self.session.stapl()
handle = open("tests/data/pystapl.stapl", "w+b")
st.config(handle=handle, part=['xcvc1902', 'xcvm1802'])
```

- **-scan-chain <list-of-dicts>:**

```
st = self.session.stapl()
st.config(out="tests/data/pystapl.stapl", scan_chain=[{'name':
'xcvc1902'}, {'name': 'xcvm1802'}])
DONE
```

- **-part <device-name list>:**

```
st = self.session.stapl()
handle = open("tests/data/pystapl.stapl", "w+b")
st.config(handle=handle, part=['xcvc1902', 'xcvm1802'])
```

stapl start

```
st.start()
```

stapl stop

```
st.stop()
```

Streams

The following are the commands followed by their options (if any) and the Python method for the option (if any).

jtagterminal

- **-start:**

  ```
  t.jtagterminal()
  ```

- **-stop:**

  ```
  t.jtagterminal('--stop')
  ```

- **-socket:**

  ```
  t.jtagterminal('--socket')
  ```

readjtaguart

```
t.readjtaguart()
```

- **-start:**

  ```
  t.readjtaguart()
  ```

- **-stop:**

  ```
  t.readjtaguart('--stop')
  ```

- **-handle:**

  ```
  t.readjtaguart(file='tests/data/streams.log', mode='w')
  ```

SVF

The following are the commands followed by their options (if any) and the Python method for the option (if any).

svf con

```
svf = s.svf()
svf.con()
```

svf config

- **-scan-chain:**

```
svf = s.svf()
svf.config(scan_chain=[0x14738093, 12, 0x5ba00477, 4], device_index=1,
cpu_index=0,
out='/home/rpalla/Desktop/svf1.svf')
```

- **device-index <index>:** No method available
- **cpu-index <processor core>:** No method available
- **out <filename>:** No method available
- **delay <tcks>:** No method available
- **--linkdap:** No method available
- **bscan <user port>:** No method available
- **mk_chunksize <size in bytes>:** No method available
- **exec_mode:** No method available

svf delay

```
svf.delay(tcks=1000)
```

svf dow

- **<file>:**

```
svf.dow(file='/proj/rdi/staff/rpalla/nobkup/wkspace_zynq/hello/Debug/
hello.elf')
```

- **-data <file> <addr>:**

```
svf.dow("--data" , file = "data.bin", addr =0x1000)
```

svf generate

```
svf.generate()
```

svf mwr

```
svf.mwr(0xffff0000,0x14000000)
```

svf rst

```
svf.rst("--processor")
```

svf stop

```
svf.stop()
```

TFile

The following are the commands followed by their options (if any) and the Python method for the option (if any).

tfile close

- **handle = <file_handle>:**

```
tfile = s.tfile()
tfile.ls('/tmp')
handle = tfile.open('/tmp/tfile_test.txt', flags=0x0F)
tfile.write(handle, 'hi, this is a test string')
print(tfile.read(handle, size=5))
fstat = tfile.fstat(handle)
print(fstat)
tfile.close(handle)
```

tfile copy

- **src = <src_file>, dest = <dest_file>:**

```
--owner (-o)
Copy owner information.
--permissions (-p)
Copy permissions.
```

```
tfile.copy('/tmp/rigel.txt', '/tmp/rigel2.txt')
tfile.copy('/tmp/rigel.txt', '/tmp/rigel2.txt','-o','-p')
```

tfile fsetstat

- **handle = <file_handle>, file_attr:**

```
handle = tfile.open('/tmp/tfile_test.txt', flags=0x0F)
tfile.write(handle, 'hi, this is a test string')
print(tfile.read(handle, size=5))
fstat = tfile.fstat(handle)
fstat.permissions = 65535
tfile.fsetstat(handle, fstat)
```

tfile fstat

- **handle = <file_handle>:**

```
handle = tfile.open('/tmp/tfile_test.txt', flags=0x0F)
tfile.write(handle, 'hi, this is a test string')
print(tfile.read(handle, size=5))
fstat = tfile.fstat(handle)
```

tfile ls

- **path = <dir_path>:**

```
tfile.ls('/tmp')
```

file lstat

- **path = <link>:**

```
lstat = tfile.lstat('/tmp/rigel.txt')
```

tfile mkdir

- **path = <dir_path>:**

```
tfile.mkdir('/tmp/new')
```

tfile open

- **flags = <flags>**`read mode = 0x00000001 write mode = 0x00000002 append mode = 0x00000004 create mode= 0x00000008 trunc mode = 0x00000010 excl mode= 0x00000020`**:**

```
tfile.open('/tmp/tfile_test.txt', flags=0x0F)
```

tfile opendir

- **path = <dir_path>:**

```
f = tfile.opendir('/tmp')
```

tfile read

- **handle, size:**

```
handle = tfile.open('/tmp/tfile_test.txt', flags=0x0F)
tfile.write(handle, 'hi, this is a test string')
print(tfile.read(handle, size=5))
```

tfile readdir

- **handle = <dir_handle>:**

```
f = tfile.opendir('/tmp')
print(tfile.readdir(f))
```

tfile readlink

- **path = <file_path>:**

```
print(tfile.readlink('/tmp/rigel1.txt'))
```

tfile realpath

- **path = <file_path>:**

```
print(tfile.realpath('/tmp/../tmp/rigel.txt'))
```

tfile remove

- **path = <file_path>:**

```
tfile.remove('/tmp/rigel.txt')
```

tfile rename

- **old_path, new_path:**

```
tfile.rename('/tmp/tfile_test.txt', '/tmp/rigel.txt')
```

tfile rmdir

- **path = <dir_path>:**

```
tfile.rmdir('/tmp/new')
```

tfile roots

```
print(tfile.roots())
```

tfile setstat

- **path = <file_path>, attr = <file_attr>:**

```
stat = tfile.stat('/tmp/rigel.txt')
print(stat)
stat.permissions = 65535
tfile.setstat('/tmp/rigel.txt', stat)
```

tfile stat

- **path = <file_path>:**

```
lstat = tfile.stat('/tmp/rigel.txt')
```

file symlink

- **link = <link_path>, target = <target_path>:**

```
tfile.symlink('/tmp/rigel1.txt', '/tmp/rigel.txt')
```

tfile user

```
print(tfile.user())
```

tfile write

- **handle <file_handle>, data:**

```
Optional:
offset=<offset>
The offset (in bytes) relative to the beginning of the
file from where to start writing. Default is 0.
pos=<pos>
Offset in *data* to write. Default is 0.
size=<size>
Number of bytes to write.Default is length of *data*
```

```
handle = tfile.open('/tmp/tfile_test.txt', flags=0x0F)
tfile.write(handle, 'hi, this is a test string')
```

Usage Examples

Debug Operations on Session Object

In this mode, you can create a session object and run debug operations on different debug targets using the same session object.

```
session=start_debug_session()
session.connect(url="TCP:xhdbfarmrkd11:3121")
session.targets(3)
# All subsequent commands are run on target 3, until the target is changed
with targets() function
session.dow("test.elf")
session.bpadd(addr='main')
session.con()
session.targets(4)
# All subsequent commands are run on target 4
session.dow("foo.elf")
session.bpadd(addr='foo')
session.con()
```

Debug Operations on Target Object

You can also use the Target object returned by targets() functions and run debug operations can be performed on these objects. The following is an example:

```
session = start_debug_session()
session.connect(url="TCP:xhdbfarmrkd11:3121")
ta3 = session.targets(3)
ta4 = session.targets(4)
# Run debug commands using target objects
ta3.dow("test.elf")
ta4.dow("foo.elf")
bp1 = ta3.bpadd(addr='main')
bp2 = ta4.bpadd(addr='foo')
ta3.con()
ta4.con()
...
bp1.status()
bp2.status()
```

For interactive usage, it is recommended to use commands and options instead of functions and arguments, as functions require a lot of extra typing. You have defined an interactive() function in xsdb module, which supports the commands. The following is an example:

```
Vitis-ng [1]: import xsdb
Vitis-ng [2]: xsdb.interactive()
% conn -host xhdbfarmrkb9
tcfchan#0
% ta
1 APU
    2 ARM Cortex-A9 MPCore #0 (Running)
    3 ARM Cortex-A9 MPCore #1 (Running)
4 xc7z020
% ta 2
```

```
<xsdpy._target.Target object at 0x7fb2652d3520>
% stop
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0xffffff28 (Suspended)
% q
Vitis-ng [3]:
```

AMD

Appendix A

Additional Resources and Legal Notices

Finding Additional Documentation

Technical Information Portal

The AMD Technical Information Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Technical Information Portal, go to https://docs.amd.com.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the AMD Vivado™ IDE, select **Help → Documentation and Tutorials**.
- On Windows, click the **Start** button and select **Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

***Note*:** For more information on DocNav, refer to the *Documentation Navigator User Guide* (UG968).

Design Hubs

AMD Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- Go to the Design Hubs web page.

Support Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Support.

References

These documents provide supplemental material useful with this guide.

1. *Data Center Acceleration using Vitis* (UG1700)
2. *Embedded Design Development Using Vitis* (UG1701)
3. *Vitis Software Platform Release Notes* (UG1742)
4. *Introduction to FPGA Design with Vivado High-Level Synthesis* (UG998)
5. *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400)
6. *Vitis High-Level Synthesis User Guide* (UG1399)
7. *AI Engine Tools and Flows User Guide* (UG1076)
8. *AI Engine Kernel and Graph Programming Guide* (UG1079)
9. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
10. *Vivado Design Suite User Guide: Synthesis* (UG901)
11. *Vivado Design Suite Properties Reference Guide* (UG912)
12. *Vivado Design Suite User Guide: Implementation* (UG904)
13. *Vivado Design Suite Tcl Command Reference Guide* (UG835)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
11/20/2025 Version 2025.2	
Chapter 6: Managing the Integration Project Component in the Vitis Unified IDE	New topic describing how the Vitis IDE Integration Project decouples link/package and, via a wizard, builds Fixed XSA, VMA, or VSS from flexible mixes of XSA/XPFM, AIE, PL kernels, and VSS.
General updates	Renamed `xbutil` with `xrt-smi` throughout the document.
v++ General Options	Added content for --mode option.

Section	Revision Summary
v++ Mode VSS	Added new topic.
v++ Linking and Packaging Options	Added introduction for Vitis Subsystem (VSS) Creation using the V++ Linker
07/08/2025 Version 2025.1	
General updates	Editorial and linking revisions.
05/29/2025 Version 2025.1	
--package Options	Removed support for sw_emu
Event Tracing Options	Updated num-trace-streams and trace-plio-width
launch_emulator Utility	Removed support for sw_emu
Miscellaneous Options	Updated section
Open Trace Summary Using Time Window	Updated section and screenshots
Output Directories of the v++ Command	Removed support for sw_emu
Software Platform Information	Removed support for OpenCL
System Project Structure	Removed support for sw_emu
v++ General Compilation Options	Removed support for sw_emu
xclbinutil Utility	Removed support for sw_emu
xrt.ini File	Removed support for OpenCL

Please Read: Important Legal Notices

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes. THIS INFORMATION IS PROVIDED "AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright

© Copyright 2024-2025 Advanced Micro Devices, Inc. AMD, the AMD Arrow logo, Alveo, Kria, UltraScale, UltraScale+, Versal, Virtex, Vitis, Vivado, Zynq, and combinations thereof are trademarks of Advanced Micro Devices, Inc. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the US and/or elsewhere. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.