Embedded Design Development Using Vitis User Guide

UG1701 (v2025.2) November 20, 2025

AMD

Table of Contents

Chapter 1

Getting Started with Vitis Unified Software Platform

The AMD Vitis™ unified software platform is a development environment for heterogeneous applications supporting AMD devices. This document is intended for audience interested in developing embedded designs targeting AMD Versal™ adaptive SoC devices, and AMD Zynq™ MPSoC devices. The core concepts of Vitis flows and platforms and the process of using Vitis tools to construct AI Engine and PL kernels are explained. Additionally, this document explores the seamless integration and deployment of these kernels with an AMD Vivado™ hardware design and a software platform targeting an operating system.

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 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 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 Software Platform Installation

Installing the Vitis Software Platform

Ensure your system meets all requirements described in Installation Requirements in the *Vitis Software Platform Release Notes* (UG1742).

TIP: *To reduce installation time, disable anti-virus software and close all open programs that are not needed.*

1. Go to the AMD Adaptive Computing Downloads Website.
2. Download the installer for your operating system.
3. Run the installer, `xsetup` on Linux, or `xsetup.exe` on Windows, which opens the Welcome screen. Extract the installer package.
4. Click **Next**.
5. If installing with the web installer:
   a. At the Select Install Type screen, enter your AMD user account credentials, and select **Download and Install Now**.
   b. Click **Next** to open the Accept License Agreements screen of the installer.
   c. Accept the terms and conditions by clicking each **I Agree** check box.
   d. Click **Next** to open the next screen (only required on the web installer).
6. On the Select Product to Install screen:
   a. To install the full Vitis Software Platform for embedded software and application acceleration development, choose **Vitis** and click **Next**. This full Vitis installation includes Vivado, v++ and AI Engine toolchains.
   b. If you only need to perform software development for embedded processors, and require a small installation footprint, choose **Vitis Embedded Development** and click **Next**. If you downloaded the Vitis Embedded installer, this is the only option available on the installer.
7. Customize your installation by selecting design tools and devices (this step is only for the full Vitis installation).

   The default Design Tools selections are for standard Vitis Unified Software Platform installations, and include Vitis, Vivado, Vitis HLS and Vitis Model Composer. You do not need to separately install Vivado tools.

Although not required, you can enable **Vitis IP Cache** to install cache files (for example designs found in the release). The Vitis tool installs the files at `<install_dir>/<release>/Vitis/data/cache/xilinx`.

You can enable the devices required for your development.

> **IMPORTANT!** *For the Vitis acceleration flow, the following device choice is required for installation:* ***Devices → Install devices for Alveo and Edge acceleration platforms***

8. Click **Next** to open the Accept License Agreements screen of the installer and accept the terms as appropriate.
9. Click **Next** to open the Select Destination Directory screen of the installer.
10. Specify the installation directory and review the location summary. Ensure there is enough disk space and click **Next** to open the Installation Summary screen.
11. Click **Install** to begin the installation of the software.

After a successful installation of the full Vitis unified software, a confirmation message appears, with a prompt to run the `installLibs.sh` script.

1. Locate the script at `<install_dir>/<release>/Vitis/scripts/installLibs.sh`, where `<install_dir>` is the location of your installation, and `<release>` is the installation version.

   ***Note*:** Windows do not require this script.

2. Run the script with `sudo` privileges as follows:

```
sudo installLibs.sh
```

The command installs a number of necessary packages for the Vitis tool based on the OS of your system.

> **IMPORTANT!** *Pay attention to any messages returned by the script. Be sure to install any missing packages manually. For example, if your installation of Linux does not include the zip command-line utility, you need to manually install it. This utility is required by some Vitis tools, and the `installLibs.sh` script will not install it for you.*

Installing Xilinx Runtime and Platforms

Xilinx Runtime (XRT)

Xilinx Runtime (XRT) is implemented as a combination of user-space and kernel driver components. XRT supports Versal adaptive SoC and AMD Zynq™ UltraScale+™ MPSoC-based embedded system platforms. XRT provides a software interface to AMD programmable logic devices.

For XRT product details, refer to https://www.xilinx.com/xrt.

The XRT library is available for x86 and Arm® Linux OS. It is needed on both application development and deployment environments.

Embedded Platforms

For Embedded platforms (e.g., Arm), the most common method is developing applications on a server using the cross compiling technique. Embedded platforms require a Linux kernel, a `rootfs` with integrated Xilinx Runtime, and a `sysroot` to cross-compile the host application. These components can be extracted from a pre-built common Linux software image. See Installing the Vitis Platform in the *Vitis Software Platform Release Notes* (UG1742) for details and a list of compatible boards.

1. To install a platform, download the ZIP file and extract it into `/opt/xilinx/platforms`, or extract it into a separate location and add that location to the `PLATFORM_REPO_PATHS` environment variable.
2. Install the common image using `sdk.sh -d <install_path>`. See Setting Up the Vitis Environment for details.
3. After installation, set up the application project with:

```
SYSROOT: <install_path>/sysroots/cortexa72-cortexa53-amd-linux
Image: <install_path>/Image
rootfs: <install_path>/rootfs.ext4
```

***Note*:** Using a common image does not require installing PetaLinux tools.

Custom Created Platforms

For custom created platforms, the required components are assembled and compiled using tools like PetaLinux or Yocto. Adding XRT to the target Linux image is done via PetaLinux config or Yocto recipes (see Build XRT from Yocto Recipes).

***Note*:** Creating custom Linux platforms require installing PetaLinux. For details on how to use PetaLinux and determine output locations, refer to the PetaLinux Tools Reference Guide.

For examples on how to create a fully customizable design, refer to *Versal Custom Thin Platform Extensible System* in the *Vitis Tutorials: AI Engine Development*.

For examples on how to customize a common image using PetaLinux Tools, refer to *PetaLinux Building and System Customization* in the *Vitis Tutorials: Vitis Platform Creation*.

Installing Embedded Platforms

To install a platform, download the zip file and extract it into `/opt/xilinx/platforms`, or extract it into a separate location and add that location to the `PLATFORM_REPO_PATHS` environment variable.

You can build your own platform software, or use a pre-built software common image. To download a pre-built common image, look for the **Common images for Embedded Vitis platforms** block on the downloads page, and download and extract the common image for your platform architecture.

Running `sdk.sh` extracts and installs the `sysroot`. The option `-d` gives you the option to choose where to install the `sysroot`. This package also provides a pre-compiled kernel image and `rootfs`.

You can add the `sysroot` to a Makefile for your command line project, or the Vitis IDE will prompt you to add it to your application project. For example, in your Makefile point `<SYSROOT>` to `/<install_path>/cortexa72-cortexa53-amd-linux`, which is generated when running `sdk.sh`.

Setting Up the Vitis Environment

The AMD Vitis™ unified software platform includes three elements that must be installed and configured to work together properly: the Vitis core development kit and XRT. The requirements of installation and configuration are described in Vitis Software Platform Installation in the *Vitis Software Platform Release Notes* (UG1742).

If you have the elements of the Vitis software platform installed, you need to setup the environment to run in a specific command shell by running the following scripts (`.csh` scripts are also provided):

```
#setup XILINX_VITIS and XILINX_VIVADO variables
source <Vitis_install_path>/settings64.sh
#setup XILINX_XRT
source /opt/xilinx/xrt/setup.sh
```

TIP: *The `PLATFORM_REPO_PATHS` environment variable points to directories containing platform files (`.xpfm`).*

TIP: *`.csh` scripts are also provided.*

This sets up the tools for the Vitis application development flow, the Vitis embedded software development flow, and the AI Engine tools for development on Versal adaptive SoC AI Engine devices.

To use any platforms you have downloaded as described in Installing Xilinx Runtime and Platforms in the *Embedded Design Development Using Vitis* (UG1701), set the following environment variable to point to the location of the platforms:

```
export PLATFORM_REPO_PATHS=<path to platforms>
```

This identifies the location of platform files for the tools, and makes them accessible to your design projects.

Introduction to Vitis

The AMD Vitis™ tool suite contains design technologies to develop heterogeneous embedded applications targeting AMD devices such as AMD Versal™ adaptive SoC devices, AMD Zynq™ MPSoC, and AMD Alveo™ Data Center Accelerator cards.

Vitis tools include:

- C/C++ compilers and libraries for targeting AI Engines and programmable-logic (PL)
- Conventional toolchains and libraries for Arm and MicroBlaze™ CPUs
- Graphical integrated design environment (IDE)
- System linker to configure complex device subsystems like AI Engine, NoC, and Control & Integrated Processing system (CIPS), and integrate them with PL modules and kernels in high-performance multi-rate systems
- Debuggers and HW/SW instrumentation automation to help locate and address performance bottlenecks and problems in your embedded system
- Seamless compilation, linking, and running of heterogeneous simulation spanning PL (HDL), AI Engines (System-C), and CPUs (QEMU)
- Program analyzers to profile and visualize hardware/software performance in simulation and on target
- Xilinx Runtime (XRT) that provides Linux userspace APIs for runtime image loading, memory management and kernel control
- Development platforms that provide examples for heterogenous designs
- Hardware-optimized libraries for DSP, vision, image processing, linear algebra, and many other application domains

The Vitis tools suite works with the AMD Vivado™ Design Suite, supporting C/C++ programming of hardware and software, a flexible and scalable system connectivity specification to facilitate top-down, middle-out, and bottom-up design iterations, integration of RTL modules, and clean flow automation of and handoffs to the underlying Vivado hardware tools.

Vitis Key Concepts

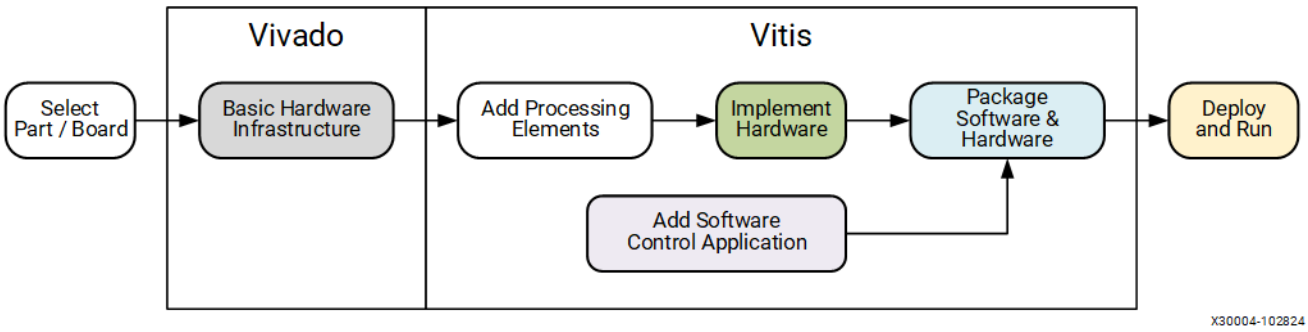
This topic introduces key concepts for understanding and using Vitis tools for embedded system design. The tools support a variety of development flows using either scripted execution or interactive design entry through the Vitis IDE. All of the development flows aim for the same outcome, but the designer's skill level and project requirements can influence the preference for a specific flow type. The design steps are described in the following table.

Table 1: **Conceptual Design Steps**

Step	Comment
Select device	Declared as part or board
Setup basic hardware infrastructure	Referred to as the hardware platform created with AMD Vivado™. A valid hardware platform consists of Vivado IP integrator components such as the CIPS, NoC, and at least one clock with associated reset. Optionally, the AI Engine, NoC DDRMC, AXI SmartConnect and AXI Interrupt controller can be added, in addition to clock domains, resets, and custom RTL IP. Within the block design, there will be PFM attributes on cells and ports that define potential attachment points for kernel control, accessing DDR and other memories, streaming input/output, clocks, resets, and interrupts. This initial design is exported to Vitis as an extensible hardware platform.
Add processing elements	Referred to as Vitis components, which includes PL Kernels and AI Engine graphs. ***Note*:** These components are developed and verified independently before being added. Component development is not in scope of this document.
Implement Hardware	In Vitis terminology, this is the process of compiling and linking the hardware system. The Vitis implementation step leverages Vivado for PL synthesis and place and route.
Add Software Applications	Control and/or processing applications running on processors like APU, RPU, and MicroBlaze.
Package Software and Hardware	The process of combining loadable object and executable applications to a binary deliverable like SD Card, QSPI Flash, etc.
Deploy and Run on target	This step involves loading a binary deliverable and running on hardware or in hardware emulation. The Vitis implementation step leverages Vivado for PL synthesis and place and route.

The following figure demonstrates a conceptual flow for designing and integrating an embedded system with AMD tools.

Figure 1: **Vitis Conceptual Design Flow**



Vitis tools support different design flows as described in Chapter 3: Vitis Flows and Build Environment. The tools and techniques for creating and integrating these different components are the focus of the following sections.

Terminology for Embedded System Design

The following introduces some of the tools and terms you will encounter in this document.

- **Vitis core development kit:** Provides a framework for designing, building, and debugging heterogeneous applications using standard programming languages for both software and hardware components.
- **Vivado Design Suite:** An RTL language design, synthesis, and implementation tool that enables hardware designers to create and export hardware designs (`.xsa`).
- **Hardware Design File (`.xsa`):** Is a hardware container exported from the Vivado Design Suite for multiple uses, including in a fixed or extensible platform.
- **Fixed Platform (`.xpfm`):** Includes a completed hardware design (`.xsa`) and supporting software files defining the operating system, libraries, and boot files. In this context, "fixed" simply means that the hardware design is complete.
- **Extensible Platform (`.xpfm`):** The target platform of the Vitis heterogeneous system design flow. In this context, the "extensible" design can be further customized by adding programmable content such as PL kernels and AI Engine graph applications to the platform to build the embedded system. Extensible Platform can also be used to develop software like the fixed platform.
- **PL kernel (`.xo`):** A hardware function that can be added to the PL region of an extensible platform to define custom hardware. PL kernels can be defined using C++ code in Vitis HLS, or using RTL code and the IP packager feature of the Vivado Design Suite.
- **Vitis HLS:** A high-level synthesis tool that translates C/C++ functions into RTL for implementation in the programmable logic (PL) region of a device. Vitis HLS generates a compiled object (`.xo`) file that can be imported into the Vitis environment.

- **Vitis Compiler:** The `v++` command used to compile PL kernels (`.xo`) from C++ code, and to link multiple PL kernels with hardware platforms and AI Engine graph applications to build the device binary.
- **PS Application:** A user-defined software application to be run on an Arm processor in an AMD MPSoC or adaptive SoC device, that can control and interact with PL kernels and AI Engine graph.
- **Xilinx Runtime library (XRT):** Provides an API and drivers to let your software application control, transfer data to, and read the status of the PL kernels and AI Engine graph application in the hardware design.
- **AI Engine kernel and graph applications:** Compiled by the Vitis `aiecompiler` and linked into the embedded system with `v++`. Kernels are functions that run on Versal AI Engines and form the fundamental building blocks of a data flow graph application. The AI Engine graph application is an adaptive dataflow graph with deterministic behavior.
- **`aiecompiler/aiesimulator`:** Vitis tools for the compilation and simulation of AI Engine graph applications.
- **Vitis Subsystem (VSS):** A platform-independent, reusable design component that targets a specific part and combines AI Engine and/or programmable logic (PL) kernels, or other VSSs. It is compiled and linked with v++ into a .vss library component.
- **Vitis Functional Simulation (VFS):** Toolset for functional simulation and verification of AI Engine graphs and HLS PL kernels using MATLAB or Python environment.
- **Device Binary (`.xclbin`) file:** Contains the programmable device image (PDI) for Versal adaptive SoC or the bitstream for Zynq MPSoC, and metadata needed to control the hardware design. Metadata-only `.xclbin` files can be used for runtime control of a loaded PDI image.

Vitis Tools Overview

Vitis tools are essential for designing, integrating and evaluating the embedded design through the development process. They are accessible either through the Vitis Unified IDE or command line. Both approaches are scriptable: using CMake or Python CLI for the IDE, and Makefiles or Shell Scripts for the command line. The following sections introduce the tools and their functionalities.

Vitis Unified IDE

The Vitis Unified IDE is a design environment for developing applications for AMD Adaptive SoC and FPGA devices. The Vitis Unified IDE lets you develop and integrate the extensible platform with the components of a system: AI Engine graphs, C/C++ sourced HLS components, packaged RTL kernels, and software applications. The components are then integrated into a top-level system project. The single unified development environment provides all the features needed to compile, run, debug, and analyze the different elements of an heterogeneous embedded system design.

The Vitis Unified IDE lets you create AI Engine components using the very-long instruction word (VLIW) processor arrays of AMD Versal™ devices. It also allows for synthesizing C/C++ code into RTL designs using HLS components, run C-simulation and C/RTL Co-simulation, in addition to reviewing and analyzing build and run summaries in the newly integrated Vitis analyzer tool.

The Vitis Unified IDE works with the common command-line features of the `v++` and `vitis-run` commands. Whether working from the command-line or from the Vitis Unified IDE, the environment provides a single tool-set for end-to-end application development without the need to jump between multiple tools for design, debug, integration and analysis.

You can perform the following tasks with the Vitis Unified IDE:

- Develop embedded applications that run on processors for Adaptive SoC, including Versal and Zynq UltraScale+ MPSoC devices
- Develop AI Engine applications and kernels for Versal Adaptive SoC
- Design programmable logic with C/C++ by creating HLS components
- Develop system projects for AMD Adaptive SoC devices

The Vitis Unified IDE provides the following benefits:

- **Architected for ease-of-use:**
  - The Flow Navigator helps you manage the work flow for different designs
  - The design flow supports example template for new users to view all available examples, increasing productivity
  - Non-blocking commands can now run multiple build and analysis jobs at the same time
  - The AI Engine pipeline view is enhanced from single core to multi-core; you can select the pipeline view for any active cores
  - The AI Engine microcode view is enhanced with user selectable filters
- **Easier switching between GUI and command-line (CLI) mode:**
  - Combining strengths of both GUI and CLI
  - Configuration files are rendered in real time

  - CLI can be used to build projects, and the Vitis Unified IDE for debugging and analysis
  - GUI operations are saved in python log for batch rebuilding
- **Modern look and framework:**
  - Light and dark themes
  - Quick actions with fully customizable shortcut keys
  - User-friendly command palette
  - Up-to-date C++ syntax highlighting and IntelliSense

Detailed description and reference guide for Vitis Unified IDE is covered in Using the Vitis Unified IDE in the *Vitis Reference Guide* (UG1702).

Compilers

The Vitis compilers for HLS, AI Engine and PS applications can be used from the command line or from the Vitis Unified IDE.

HLS and AI Engine Components

For HLS and AI Engine components, compilation is performed with the `v++ --compile` command. When setting up Vitis IDE components, the v++ compiler mode is automatically identified. For command line mode, the mode needs to be specified using `--mode aie` or `--mode hls`.

PS Applications

For PS applications, compilation is managed through cross compilers. Cross compilers can be selected from the pre-built Linux sysroot or from PetaLinux/Yocto sysroot outputs.

Further Reading on Using the Vitis Compilers

The following topics can be informative:

- AI Engine Kernel and Graph Development
- HLS Kernel Development

Linker

In hardware, an AI Engine design can include hundreds of processors and interfaces to the programmable logic (PL) fabric and network-on-chip (NoC). The Vitis linker automatically configures the AI Engine IP core interfaces and connects them to the PL, establishing clock associations and wiring, and automatically performing rate-matching through data width and clock conversion.

The Vitis linker integrates compiled Vitis PL and AIE components and the hardware platform into binary containers. Depending on the choice of design flow, the linker creates either an intermediate binary or final binary container.

The linking process is managed through a Vitis IDE system project or with `v++ -link` command. To add and connect Vitis components to the system, specify the connections through configuration files. These configuration files also specify the clock domains and additional implementation controls to be used during integration and implementation. The steps for linking the design is described in Integrating AIE and PL Components.

Packager

The Vitis packager integrates the embedded system by assembling the final hardware and software products, and configures the boot system for the device. The package command controls several aspects of the completed system, such as starting the PL and AIE components at boot time or launched them explicitly by the software application.

The packaging process is controlled by the Vitis System Project in the Vitis IDE or by `v++ --package` from the command line. The steps for packaging the design depend on the choice of development flow and is described in Integrating the System.

Analyzer

The Vitis analyzer is a graphical tool that can be used to inspect, review and debug Vitis tool output results from AI Engine designs, HLS kernels, linker, and run summaries. It provides a visual representation of the kernels, graphs, and dataflow for the compiled or linked design.

Specifically for AI Engine, it shows the physical mapping to the AI Engine array.

To further analyze the results, you can open the AI Engine simulator run summary file to view profiling data and inspect transactions of states and signals across the AI Engine array. The Vitis Analyzer can be used to identify and optimize performance, constrain area and resolve conflicts or design hazards.

For further details, refer to Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702).

Debugger

Vitis provides a graphical interface to GDB (GNU debugger) for debugging multiple domains and processors in parallel. With this feature, you can debug not only individual components but also the interactions between components and domains in a heterogeneous system, as well as the controlling software application.

The debugging can be performed at the hardware emulation stage or directly during hardware execution. For details on various debugging methods refer to Chapter 11: Debugging System Projects.

Tutorials and Examples

To help you quickly get started with the AMD Vitis™ core development kit, you can find tutorials, example applications, and hardware kernels in https://github.com/xilinx/Vitis-Tutorials.

Chapter 3

Vitis Flows and Build Environment

This chapter provides an overview of the AMD Vitis™ development flows and build environments that are supported for different user scenarios. Understanding the development flow and interactions between the heterogeneous components will help you in setting up an efficient design process and staffing the various roles for a design project.

In the final sections of this chapter, guidance is provided on advanced design management, which includes iterative development and considerations for migrating between devices and upgrading tools.

Workflow

Vitis tools provide two distinct approaches to design development: the Vitis Integrated Flow and the Vitis Export to Vivado Flow. Each caters to different needs.

The Vitis Integrated Flow prioritizes ease of use and automation. It streamlines the design process, reducing development time and making it ideal for beginners and rapid prototyping.

The Vitis Export to Vivado Flow, on the other hand, emphasizes synthesis and implementation control. It leverages the advanced features of Vivado, offering precise control over design optimization. This flow is well-suited for experienced users tackling demanding projects.

In essence, the Vitis Integrated Flow is simple and fast, while the Vitis Export to Vivado Flow is powerful and customizable. Choose the flow that aligns best with your experience and project requirements.

Vitis Subsystem enables platform independent development usable in both Vitis Integrated Flow and Vitis Export to Vivado Flow by preparing AI Engine and PL kernels using part number instead of platform as target. Vitis Subsystem Flow shows this feature combined with Vitis Export to Vivado Flow.

Common Steps For The Flows

Vitis development begins with choosing your target device, either by selecting a specific part or a board. This selection is crucial for preparing the hardware platform, which forms the foundation for your bootable design. This platform will later be extended in Vitis to host the signal and data processing algorithms.

For AMD development boards, Vitis offers pre-built and tested base platforms, allowing you to jumpstart the development process. However, if you're working with custom boards or designs, you will need to create the hardware platform in Vivado.

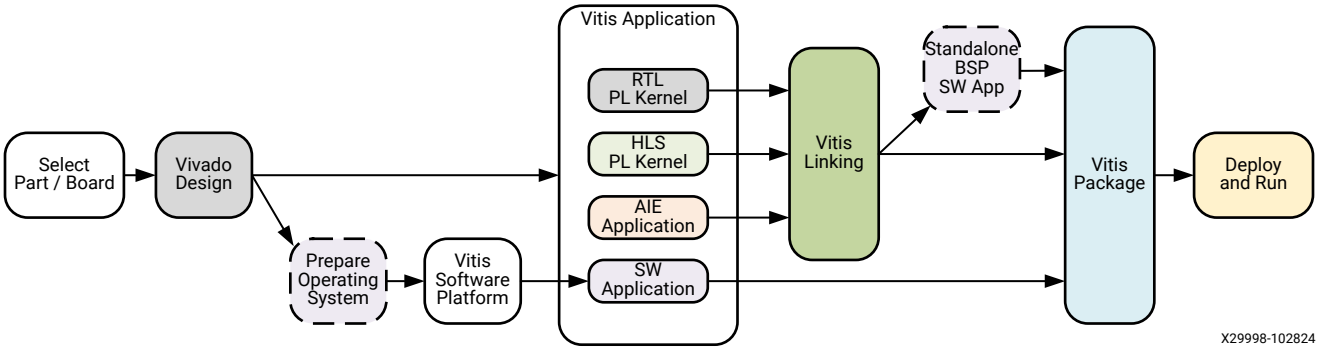
The part option allows platform independent design, compilation, and verification of HLS kernels and AI Engine graph components.

The following sections explores the different Vitis design flows, highlighting their advantages and differences to help you choose the best approach for your project.

Vitis Integrated Flow

The Vitis Integrated Flow prioritizes the ease of use and automation for rapid prototyping and deployment. It leverages a prebuilt Vivado base platform design, or alternatively allows you to reuse a custom Vivado platform. Vitis handles the linking process, automating synthesis and implementation through a temporary Vivado project, minimizing user interaction. A simplified overview of the flow is represented in the following figure:

Figure 2: **Vitis Integrated Flow**



This flow starts with selecting a device part or board. For AMD development boards, AMD provides ready-to-use base platforms, eliminating the need for manual platform creation. The steps on how to create a custom platform are described in Building Custom Platforms.

Next, Vitis applications extend the hardware design with a modular heterogeneous subsystem, represented by PL kernels and AIE graph, and provide an environment for software application development, verification and debugging. Chapter 4: Developing Vitis Kernels describes how to design and compile the subsystem, and the section on Integrating AIE and PL Components describes how to connect it to the hardware design. In addition, the software host application development requires the preparation of dependencies described in Software Platform.

For standalone Bare-metal, RTOS and non-XRT applications, the board support package and device driver software headers is generated from the output of v++ linker. The software platform can be created using the fixed XSA as input, which can be used to generate the required board support packages.

The Vitis Package step combines hardware binaries and software executable into a deliverable package, generating files for SD Card or binary containers for QSPI Flash. The section on Integrating the System describes the process of packaging the design.

Finally, the system is deployed and run on the hardware or in a hardware/software simulator called Vitis hardware emulator. More details can be found in Deploying and Running the System.

Key features of this flow include:

- Rapid prototyping and deployment: Prebuilt or custom Vivado base platform designs expedite development.
- Automated linking: Vitis handles synthesis and implementation, minimizing user interaction.
- Modular heterogeneous subsystems: Extending the hardware design with PL kernels and AIE graphs.
- Integrated software development environment: Developing, verifying, and debuging software applications.
- Deliverable package creation: Combining hardware binaries and software executables for easy deployment.

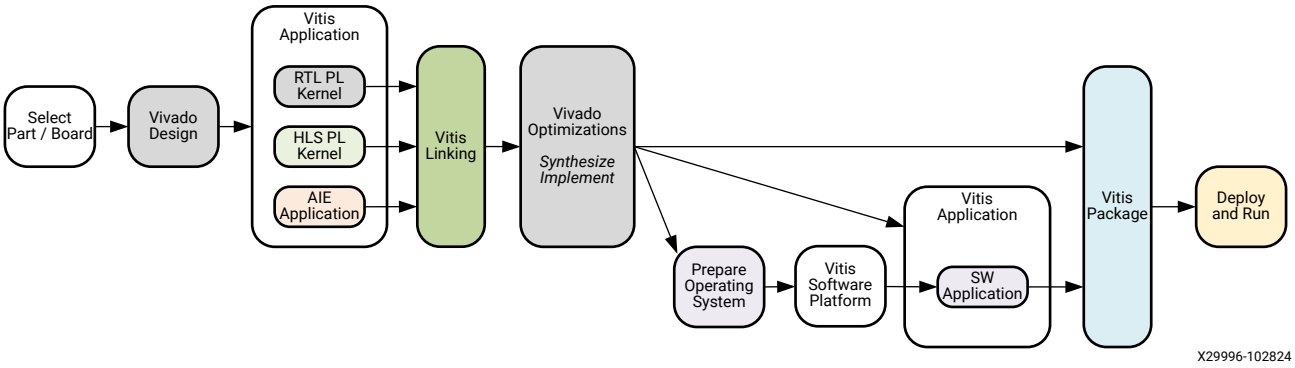
While the Vitis Integrated Flow fully supports AMD Versal and MPSoC embedded software flows, the Xilinx Runtime (XRT) offers complementary Linux userspace APIs for controlling Vitis components packaged into `xclbin` binaries. This eliminates the need for custom kernel drivers or device trees for the Vitis content, and provides a compact and readable host application coding style.

For more fine grained control over synthesis and implementation in Vivado, consider the Vitis Export to Vivado Flow.

Vitis Export to Vivado Flow

In certain situations, exporting Vitis content to a Vivado project offers advantages. The Vitis export to Vivado flow uses a Vitis linker generated metadata archive (VMA) imported by Vivado to update the original block design with Vitis content. This allows fine-grained control and analysis of synthesis and implementation within the Vivado design project. A simplified overview of the flow is represented in the following figure.

Figure 3: **Vitis Export to Vivado Flow**



The Vitis export to Vivado flow mirrors the Vitis Integrated Flow until the Vitis linking step, with the exception that the software application is created after the Vitis package step.

In this flow, the synthesis and implementation of the design occur within Vivado. The Vitis linking step primarily assembles the Vitis components using pre-synthesis and generates the archive (VMA). The VMA is imported into the original Vivado project to complete implementation. These steps are described in Vitis Export to Vivado Flow Detailed Example.

The software application development takes place after the Vivado implementation and Vitis package steps, using the extracted device information. The Deploying and Running the System step remains similar to the Vitis flow.

Key features of the Vitis export to Vivado flow

- Modular heterogeneous subsystems: Integration of PL kernels and AIE graphs into the hardware design.
- Implementation and timing control: Detailed implementation and timing control in Vivado.
- Integrated software development environment: Complete software development, verification, and debugging capabilities.
- Deliverable package customization: Simplified package creation with Vitis packager or advanced configuration with Bootgen.

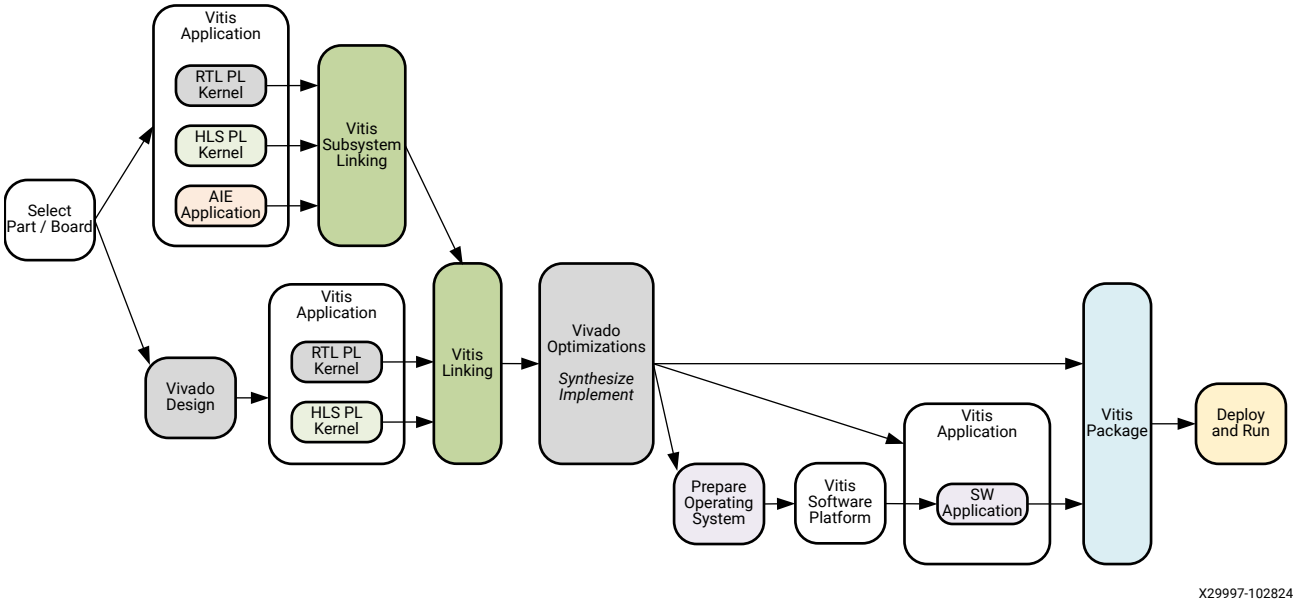
While this flow provides more control over the design process, the Vitis export to Vivado flow requires a deeper understanding of design management, especially with regard to dependencies between the extensible platform, Vitis domain and Vivado implementation.

Vitis Subsystem Flow

The Vitis subsystem flow enables the development, testing, and delivery of AI Engine graphs and PL kernels packaged as a platform-independent subsystem. This flow allows for the simulation of the subsystem without the need for a hardware platform, making it faster and more flexible. The VSS component can be configured and rebuilt for testing on multiple devices, making it ideal for feasibility studies, custom library development, and reusable modules.

The subsystem flow uses `v++` to link AI Engine graphs and PL kernels into a `.vss` deliverable, which can be used as a component in Vitis or incorporated into another VSS. The VSS component can be used in either the Vitis Integrated Flow or the Vitis Export to Vivado Flow. A simplified overview of the flow is represented in the following figure:

Figure 4: **Vitis Subsystem Flow with Vitis Export to Vivado**



The primary objective of the VSS flow is to facilitate the development of a combination of AI Engine and PL, independent of the platform. Multiple VSS can contain AI Engine content targeting disjoint partitions. A VSS component can be simulated independently of a platform or running hardware emulation, allowing for AI Engine and PL simulation.

The specific steps for creating and integrating a VSS component are described in Linking a VSS Component and Linking the VSS Component to the Platform.

Key Features of the VSS Flow

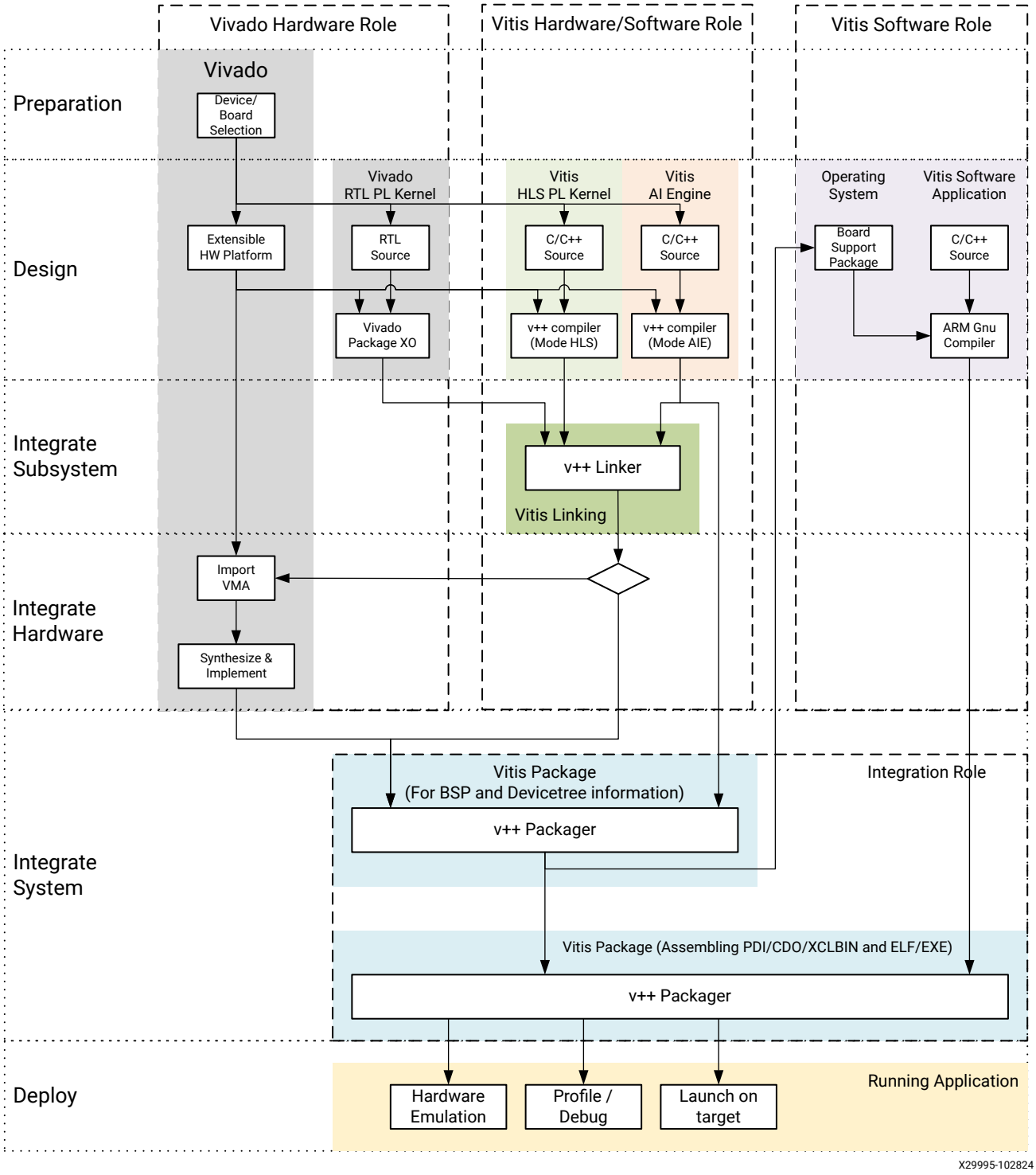
The key features of the VSS flow include:

- Platform independent heterogeneous subsystems: Integration of PL kernels and AIE graphs into a VSS component.
- Simulation without needing a platform or running Hardware Emulation: Provides a significant advantage by enabling faster and more flexible testing without platform dependency or hardware emulation.
- Early exploration of AI Engine and PL kernels: Facilitates the early development and testing of key components. It is also very useful when migrating functionality to the AI Engine from the PL.
- Aligns with both Vitis Integrated Flow and Vitis Export to Vivado Flow.

Understanding the Vitis Build Flow

This section uses the Vitis Export to Vivado Flow to describe the roles and activities associated with the Vitis build environment throughout the development process. Moreover, the inputs and hand-offs between each step in the flow are also described. A simplified representation of the process and roles can be seen in the following figure.

Figure 5: **Understanding the AMD Vitis Build Flow**



The Vivado hardware role is responsible for the design and implementation of a hardware platform, similar to the classic FPGA hardware design role. To facilitate incorporating Vitis signal processing components, an extensible hardware platform is exported from Vivado to Vitis.

The Vitis role develops the subsystem that gets integrated with the extensible platform. The subsystem (usually) consists of AIE and PL. The AIE part is done using C/C++. The PL part is done using C/C++ and/or RTL.

Note: Vitis can integrate RTL PL kernel objects, but rely on the Vivado HW role for designing and packaging these kernel objects.

Next, the Vitis subsystem is linked to a Vitis metadata archive which is imported into the Vivado hardware design project. The Vivado hardware role can make changes to the Vivado project throughout the design process, but caution must be exercised when modifying clocks, resets and interfaces between the platform and the Vitis subsystem. For further guidance, see Advanced Design Management.

Once the design is ready for system integration, the Vivado hardware role runs synthesis and implementation to generate the entire hardware design, also known as the fixed platform. The integration role then extracts metadata to generate a board support package or device tree, which represents low-level APIs for bare metal and Linux applications, respectively, thus preparing all the prerequisites for the Vitis software role.

The Vitis software role is involved in designing, compiling, and debugging the application, and delivering the executables to the Integration Role for packaging the hardware and software into a deployable containers, such as SD cards and QSPI Flash, etc.

The final step involves deploying and running the design. This can be done either in the hardware emulation environment or on the target board for system profiling and debugging before final delivery.

Vitis IDE Workspace

The Vitis Unified IDE workspace serves as an organizer for Vitis projects to keep track of project settings, source code and build recipes including constraints and directives. You can choose to import source code either by copying the files into the workspace folder structure or as reference to a location you specify. When importing without copying the files, the workspace serves as managing the project settings and build recipes, but the files remain at their original location. This is practical when source code is shared among several projects and handled separately by SCM tools.

The benefit of using the Vitis Unified IDE is that the workspace automatically detects changes to source files and prerequisite settings, and calls rebuilding Vitis components dependencies.

Note: Vitis can only rebuild Vitis managed components automatically. Components like RTL PL kernels require you to manually rebuild with Vivado.

Details on using the IDE can be found in Using the Vitis Unified IDE in the *Vitis Reference Guide* (UG1702).

Command Line Recommendations

This section provides good practices for command line users who wish to set up a workflow using scripts and Makefiles.

A command line flow is suitable for users preferring to run design builds in batch mode with minimal user interaction. You should have a good understanding on the prerequisite inputs and generated outputs for each build step. Once these conditions of the flow are established, the build mechanism can be reused and repeated in several projects and/or design iterations.

Dependency chains can be managed through setting up Makefile recipes to trigger building the outputs for each step in the design process. As the command line is agnostic to the tools, these recipes can stretch across all tools used in the flow. This makes the flow specifically usable for complete system development process covering all the roles.

***Note*:** When updating or re-importing Vitis Susbsytem to Vivado, special care is required. This is explained further in Concurrent and Iterative Development.

Files and Directories

When using command line and scripted flows, it's your responsibility to arrange files and directories to make it clear what the input prerequisites and generated output files are for each build step. As the Vitis tools produce various intermediate temporary files, logs, and debug information along with the generated output, it's good practice to separate the source inputs and recipes from the directory where the tools are launched.

To further emphasize the generated output files, which will be used as prerequisite input to next build stage, redirecting the outputs to specific names and directories help keep the handoffs organized.

Makefiles and configuration files are useful for managing build recipes, dependency chains and project settings. If well written, they can serve as templates for several projects, reducing the effort to setup and use the build environment.

It's strongly recommended to put source files, project config files and Makefiles in source code management systems like Git.

The Vitis Tutorials (design tutorials and developer contributed tutorials) contain examples for setting up an end-to-end Makefile flow.

Guidelines for Revision Control

It is recommended to revision control the project sources while developing the design using the Vitis and Vivado tools. In this flow, a number of design iterations are required for the hardware design development. You can adopt revision control mechanism according to your standards. The following files are recommended for revision control:

- C++ source files
- Config files
- Tcl scripts
- Makefiles

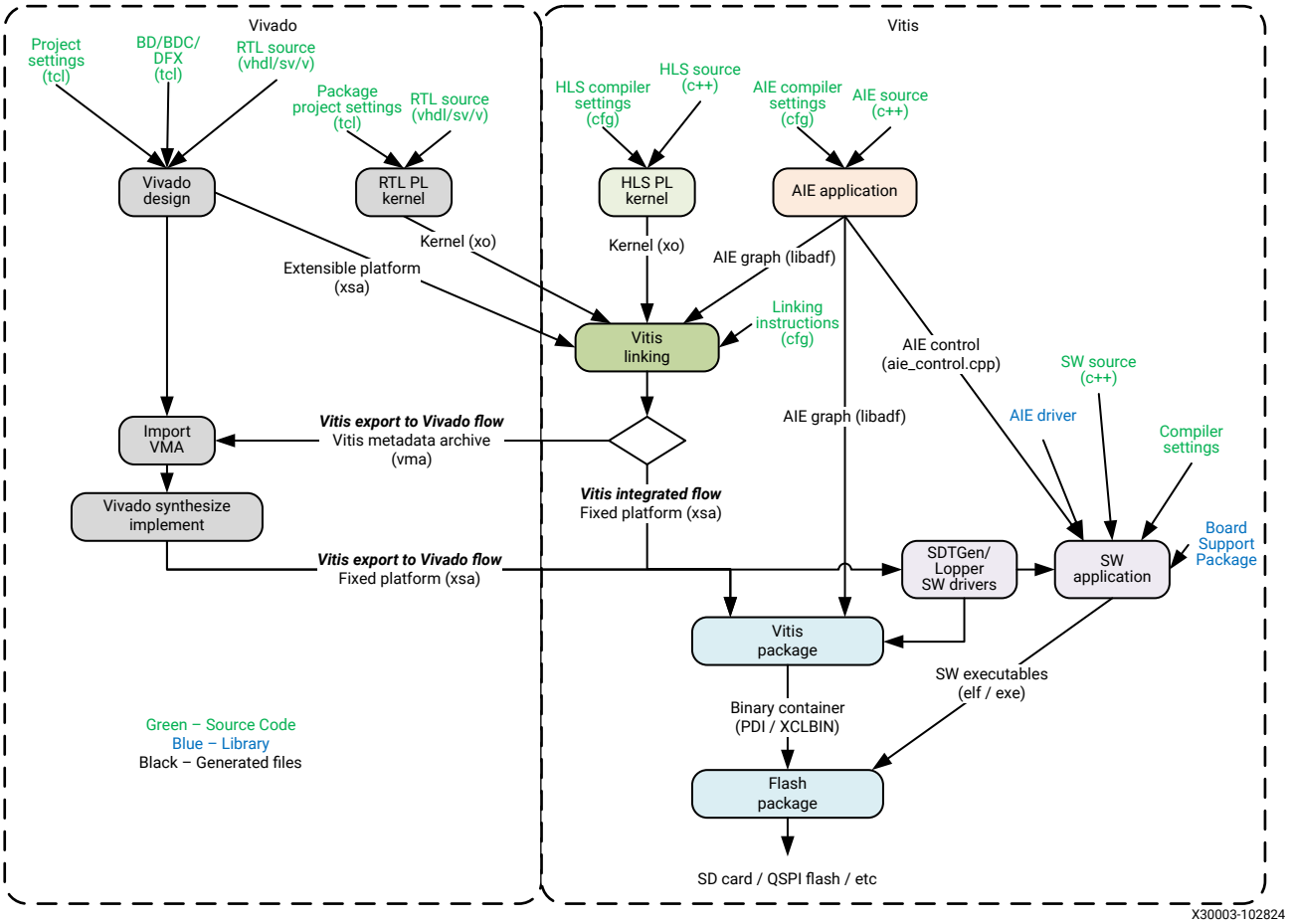
Optionally, the following generated binary files can be added for additional delivery control:

- AI Engine graphs (`libadf.a`)
- `.vma`
- `.xsa`

Vivado and Vitis System Level Dependencies and Build Handoffs

This section provides a basic overview of the file dependencies and build handoffs passed along the design steps in the flows. The Vitis Export to Vivado Flow is used as case example. The following figure describe a simplified overview of the most important files required by each step and what outputs they produce.

Figure 6: **Vivado and Vitis System Level Dependencies and Build Handoffs**



These dependencies determine the build order and where the development can benefit from concurrent design activities. This can then be used to setup automated build flows, either by workspace arrangements in Vitis Unified IDE or by scripts and Makefiles for batch mode operation. From a project management perspective, the handoffs and converging stages help to define the team roles and responsibilities to setup a multi-user design environment.

***Note*:** Specific details and complete list of inputs and outputs are described in respective chapters in this document.

TIP: *Set up a basic build flow on a simple design to run through the complete build, and when you are confident about the mechanism, introduce more advanced development techniques, such as concurrent design methodology.*

Setting up and using advanced design development is further discussed in Concurrent and Iterative Development.

Device Selection and Execution Target

The first step when building the application is to specify a target device and what kind of execution target to use. Device selection and corresponding description is done through the following methods:

Table 2: **Device Selection and Execution Target**

Device selection method	v++ Command line option	AMD Vitis™ IDE option
Part	`--part <part_number>`	Select the **Part** dialog when creating new Vitis Component, or change in Vitis Component settings.
Custom Platform	`--platform <platform_name>`	Select the **Platform** dialog when creating a new Vitis Component, or change in Vitis Component settings.
Prebuilt Base HW Platform	`--platform <platform_name>`	Select the **Platform** when creating a new Vitis Component, or change in Vitis Component settings.

The execution target of the Vitis tool defines the nature and contents of the FPGA binary output created during compilation and linking. There are two different build targets: one emulation target used for validation and debugging purposes: hardware emulation, and the default system hardware target used to generate the FPGA binary loaded into the AMD device.

The emulation run is performed in a simulation environment, which offers enhanced debug visibility and does not require physical hardware board to launch design execution.

Selecting execution targets is applicable to compile, link and package commands with v++ using the option `--target [hw_emu | hw]`.

Table 3: **Comparison of Emulation Flows with Hardware Execution**

Hardware Emulation	Hardware Execution
Host application runs with a simulated RTL model of the kernels. SystemC models and external TGs are also supported.	Host application runs with actual hardware implementation of the kernels.
Test the host / kernel integration, get performance estimates.	Confirm that the system runs correctly and with desired performance.
Best debug capabilities with increased visibility of the kernels.	Final FPGA implementation with accurate (actual) performance results.

Details on device selection and execution target options are available in v++ General Options in the *Vitis Reference Guide* (UG1702).

Part

By using the part for the selection of device, the development of Vitis components can be started without the existence of an extensible hardware platform. This is useful for several cases such as:

- Early prototyping and proof of concepts
- Concurrent development, avoiding waterfall development process
- Subsystem development
- Exploring device migration and design reuse options

The part option can also be used to generate a generic platform design during the linking process. This can be used for testing, evaluation, and quick studies; however, it is recommended to switch to custom or base platform after the design matures.

Custom Platform

When using a custom hardware platform, the device information is automatically extracted from the extensible platform archive. This option gives full control over the design, but also requires knowledge of hardware designer role. Choosing a custom hardware platform is useful in scenarios like:

- Custom board designs
- When the design requirements deviate from prebuilt base platform with respect to clocking strategies, external interfaces on an AMD development board
- Designs that require custom IPs added in Vivado.

***Note*:** You can create a custom hardware platform in Vivado and use configurable example designs to quickly create a starting design.

Once the design flow steps are completed, using a custom hardware platform design is the most flexible design flow option. The following link provides tutorials and examples for using custom hardware platforms: https://github.com/Xilinx/Vitis-Tutorials/tree/master/Developer_Contributed/01-Versal_Custom_Thin_Platform_Extensible_System.

Prebuilt Base HW Platform

Prebuilt hardware platforms are available for AMD development boards. They provide an AMD verified and bootable design targeting boards such as VCK190 and VEK280. The prebuilt platforms allow for fast integration and deployment of a Vitis subsystem so designers can focus on the signal processing algorithms. The prebuilt base HW platforms are useful in these scenarios:

- Early design and evaluation of signal processing subsystems

- Concurrent development with custom hardware platform
- Independent testing and hardware debugging of the subsystem using development boards
- Performance testing of algorithms using predesigned testing examples like AI Engine test harness.

AMD recommends verifying the subsystem before integrating with the custom platform. This ensures the subsystem being verified in a known environment to ensure functional and performance targets are met before integrating in the intended target system.

The following links provide useful tutorials and guides on subsystem testing and verification:

- https://github.com/Xilinx/Vitis-Tutorials/tree/master/Vitis_Platform_Creation/Feature_Tutorials/04_platform_validation
- https://github.com/Xilinx/AI-Engine-Test-Harness

Advanced Design Management

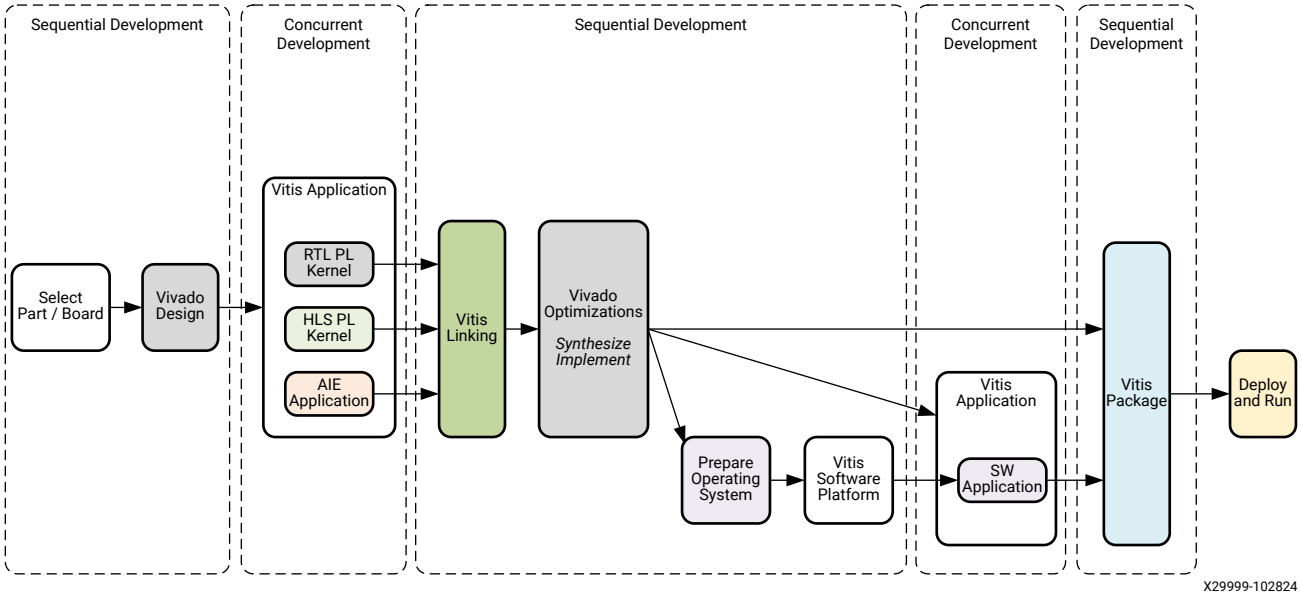
The introduction chapters and getting started tutorials in this document focus primarily on demonstrating the Vitis features in well behaving context few conditional requirements, taking a design from start to end in a forward progress. This section provides techniques for advanced design management, often where several team members collaborate and the design evolves in an iterative process. This section also describes migration between devices and guidance for upgrading AMD tools.

Concurrent and Iterative Development

Although it's possible to manage the complete build flow by a single user, the platform concepts and Vitis subsystem components enables the design project to increase productivity by adopting to multiple user and concurrent development methodology.

First, split the work into roles and activities, while maintaining a forward sequential flow for merging the splits. This requires all prerequisite dependencies to be available before the step can be completed, and avoids difficult synchronization of partial results. A simplified example of this kind of concurrency is shown in following figure.

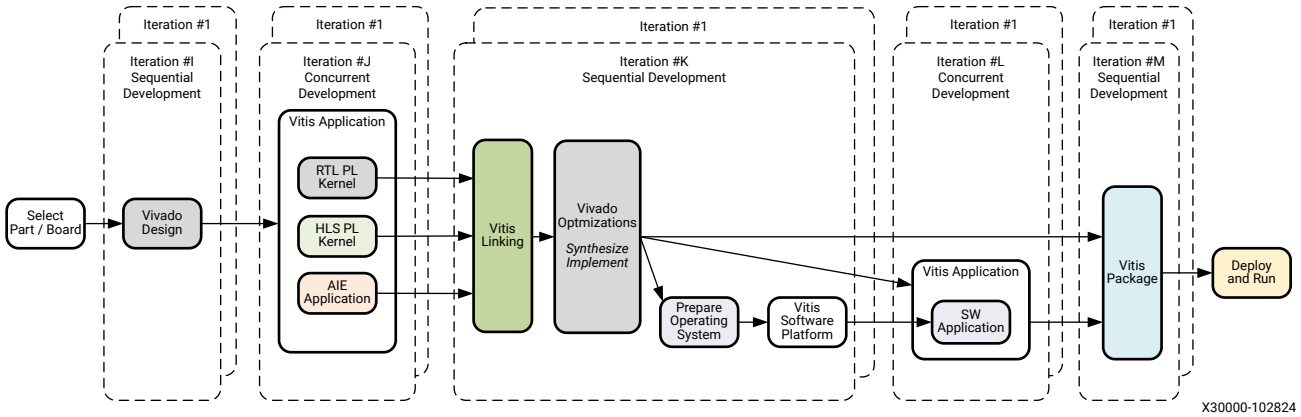
Figure 7: **Concurrent Development**



The integration, importing, and exporting steps involve access to the entire design. These steps are best performed by a single user in a sequential manner, after which the next design step can proceed.

To further jumpstart the Vitis component, SW application development, and board verification, an iterative methodology can be adopted. This is similar to the approach as using prebuilt Vivado base platforms to accelerate the development, but uses a custom hardware platform for incremental design builds. The following figure represents a design flow using both iterative and concurrent methodology.

Figure 8: **Concurrent and Sequential Development**



The principle with this flow is to complete the first iteration with minimalistic content to generate basic dependencies so each contributor can start their development. It's highly recommended to have a lead designer in charge of integrating and synchronizing release iterations from the contributors. Once the design is in sync with the release contributions, concurrent development activities can resume.

***Note*:** As the design matures to it's final version, higher caution and control should be used when iterating changes and updates.

Migrating Between Devices

The ease of migrating a subsystem across multiple devices and in specific cases even device families is one of the strongest advantages of using the platform methodology. By using higher level API and generic RTL for designing the Vitis subsystem components, the portability across devices improves.

To enable easy design migration the following conditions should be met:

Table 4: **Device Migration Checklist**

Item	Notes
Use features and language constructs that are compatible with both devices.	The devices need to be compatible either by hardware feature or API. If by API, expect differences in performance and resource utilization. Similar applies for different speed grade and device families as they have.
Test and validate Vitis subsystem portability with intended parts.	Alternatively test with a pre-built base platform if it exists.
Create separate Vivado hardware platform projects for each target device.	This is required to separate build outputs per target device.
Create separate Vitis workspaces and output folders.	Each target require rebuilding for that device, so the outputs and intermediate build objects need to be kept apart.
Put Vitis Component source files in a common folder and reference them to each workspace.	It is important from source code management perspective to add sources as reference. If copied into the project, they depart from being shared between the devices.
Build and compare design outputs.	For migration between different speed grades, check timing reports in Vivado. For designs migrating between different device sizes, check utilization reports in Vivado.

The Versal Thin Custom Platform tutorial demonstrates how to set up a project suitable for targeting both VCK190 and VEK280 development boards.

***Note*:** The current version of this tutorial is set up to build one target at a time. To build for several targets, either clone/copy for each target or use the design as inspiration to setup a customized flow.

Upgrading Tools

When upgrading a design to run with the this version of the AMD tools, the design should be recompiled to have all metadata in sync with the tool release. To upgrade the design, follow the checklist in the following table.

Table 5: **Tool upgrade checklist**

Item	Action
Upgrade Base Hardware Platform	Download and reference to compatible Base HW Platform.
Upgrade Custom Hardware Platform	Upgrade Vivado version check and IP versions in block design using Vivado IP integrator and regenerate the extensible hw platform. Upgrading Vivado designs is described in Updating Designs for New Release in the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994).
Optional: Migrate from classic Vitis IDE	Projects from the deprecated Vitis Classic IDE need to be migrated to the Vitis 2024.2 release before upgrading to this release.
Copy or create a new workspace	If the workspace is in version control, use the SCM tool to create a new branch for the upgraded design. Otherwise make a copy/new workspace for the upgraded design so you can compare build results between the versions.
Rebuild targets	Rebuild the design and compare results with design version in previous tools.

Designs using command line or Makefile build flow with custom hardware platform can run rebuild after upgrading the custom hardware platform using the guide in preceding table. This should be done in a new folder so the build can be compared with the previous design version.

Once rebuilt, verify and compare the results with the previous design to ensure functionality and performance meets expectations.

AI Engine Partitions and Runtime Control and Reload

The AI Engine supports column-based partitions by creating independent graphs that can be compiled and simulated separately. Each graph is assigned to a specific column or a contiguous set of columns within the AI Engine array.

AI Engine partitions allow independent compilation, linking, and runtime reloading of AI Engine graphs while keeping other partitions active. The Vitis linker and embedded XRT enable flexible partitioning, runtime control, and reloading of AI Engine partitions in Versal devices.

For information on AI Engine partitions and build flow, refer to Compiling AI Engine Graph for Independent Partitions in the *AI Engine Tools and Flows User Guide* (UG1076).

For runtime control and reload of AI Engine partitions, refer to Runtime Control and Reload of AI Engine Partitions in the *AI Engine Tools and Flows User Guide* (UG1076).

Developing Vitis Kernels

This chapter describes the key elements of the AMD Vitis™ development environment, which is used in the development of Vitis kernels and applications. It provides guidance on the following topics:

- AI Engine Kernel Development
- HLS Kernel Development
- RTL Kernel Development
- Host Application Development

Introduction to Kernels in Vitis

This chapter covers the different Vitis kernels, including their design flow and methods to control these kernels in the host application. The Vitis kernels can be classified into two categories:

- AI Engine Kernels: Kernels instances in a graph are compiled as `.a` files, such as `libadf.a`.
- PL kernels: These kernels are compiled as `.xo` files.

Vitis PL kernels can further be classified into two categories based on the programming languages used to develop them:

- HLS PL kernels: These are kernels developed using the High-Level Synthesis (HLS) programming language.
- RTL kernels: These are kernels developed using Register-Transfer Level (RTL) programming language.

Vitis PL kernels can also be categorized based on their host control mechanism:

- Software-Controllable Kernels: These kernels provide a programmable register interface that allows host software to interact with the kernel through APIs or register reads and writes.
  - XRT managed kernels: Kernels that are controlled by XRT APIs.
  - User managed kernels: Kernels that are free-running or state machine-controlled via register writes and reads.

- Data Driven Kernels: These types of kernels do not require host control. These kernels are present in the device but are not managed by the software application. Instead, they are triggered by the arrival of data at the interface.

In the Vitis core development kit, embedded platforms provide a foundation for designs. Targeted devices can include Versal adaptive SoCs, Zynq UltraScale+ MPSoCs, or AMD UltraScale+™ FPGAs. These devices contain a programmable logic (PL) region. A device binary (`.xclbin`) file can be loaded and executed on these devices, it contains and connects PL kernels compiled as object (`.xo`) files and can also contain AI Engine graphs.

These platforms can contain one or more interfaces to global memory (DDR or HBM), and optional streaming interfaces connected to other resources such as AI Engines and external I/O. PL kernels can access data through global memory interfaces (`m_axi`) or streaming interfaces (`axis`). The memory interfaces of PL kernels must be connected to memory interfaces of the platform. The streaming interfaces of PL kernels can be connected to any streaming interfaces of the platform, of other PL kernels, or of the AI Engine array. Both memory-based and streaming connections are defined through Vitis linking options, as described in Linking the System.

Multiple kernels (`.xo`) can be implemented in the PL region of the AMD device binary (`.xclbin`), allowing for modular and portable building blocks for a signal processing subsystem. A single kernel can also be instantiated multiple times. The number of instances of kernels and how they are connected to the subsystem is specified by a linking configuration used when building the device binary.

For Versal devices, the `.xclbin` file can also contain compiled AI Engine graphs. The compiled AI Engine graph (`libadf.a`) and PL kernels (`.xo`) are linked with the target platform (`.xpfm`) to define the hardware design. The AI Engine can be accessed by PL kernels using `axis` interfaces. The AI Engine can also be controlled through the Arm processor (PS) via runtime parameters (RTP) in the graph and GMIO on Versal adaptive SoC devices. Refer to *AI Engine Tools and Flows User Guide* (UG1076) for more information.

AI Engine Kernel and Graph Development

As described in *AI Engine-ML Kernel and Graph Programming Guide* (UG1603) and *AI Engine Kernel and Graph Programming Guide* (UG1079), an AI Engine kernel is a C/C++ program that is written using the AI Engine API and/or specialized intrinsic functions that target the VLIW scalar and vector processors.

The AI Engine compiler produces ELF files that are run on AI Engine processors. Multiple AI Engine kernels are combined in an adaptive data flow (ADF) graph that consists of nodes and edges where nodes represent compute kernel functions, and edges represent data connections. The ADF graph is a static data flow graph with kernels operating in parallel on data streams. The ADF graph interacts with the PL kernels in the device binary, global memory, and the host application. The AI Engine compiler produces a `libadf.a` file containing the graph application, which can be linked with PL kernels and extensible platform by the Vitis compiler to produce the device binary.

As described in *AI Engine Tools and Flows User Guide* (UG1076), the AI Engine compiler (`v++` or `aiecompiler`) is available as a standalone command, or as part of the Vitis unified IDE. The IDE provides a language sensitive editor, simulation, profiling, and debug in a unified environment. The AI Engine code can be developed in the Vitis environment, then linked into the overall platform and project using either the command line or the IDE as described in Chapter 8: Building and Running the System, or in the Vitis Unified IDE as described in Using the Vitis Unified IDE in the *Vitis Reference Guide* (UG1702).

For `v++` command usage on compiling AI Engine graphs, refer to v++ Mode AI Engine in the *Vitis Reference Guide* (UG1702).

HLS Kernel Development

In the Vitis development flow, PL kernels as compiled object files (`.xo`) are the processing elements executing in the programmable logic region of the AMD device. The Vitis core development kit supports PL kernels written in C/C++ compiled by the `v++` HLS compiler, and RTL IP packaged in the Vivado Design Suite.

This section covers PL kernels in C/C++ by HLS compiler, with details on kernel interfaces, clock and reset requirements and host controlling code. Unless otherwise stated, the interface, clock, reset and host code practices in this section can also apply to RTL Kernels, if RTL kernels matches the specific requirements.

For the `v++` command to compile C/C++ code with HLS compiler, refer to v++ Mode HLS in the *Vitis Reference Guide* (UG1702).

HLS Kernel Interface Requirements

The kernel interfaces are used to exchange the data with the host application, other kernels, or device I/Os. HLS kernel interface requirements are listed below:

- **Control interface:**
  - XRT-managed or user-managed kernels: Kernels can only have a single AXI4-Lite interface.

    - Data driven kernels: The AXI4-Lite interface is optional and used to pass scalar values to the kernel.
- **Data interfaces:**
    - XRT-managed or user-managed kernels: Any number and combination of AXI4 and AXI4-Stream interfaces.
    - Data driven kernels: The kernel requires at least one AXI4-Stream interface.
- **Clocks and resets:** As described in Clock and Reset Requirements.

> **TIP:** *XRT-managed kernels have specific requirements for control registers in the AXI4-Lite interface (including start and stop bits) as described in Control Requirements for XRT-Managed Kernels. User-managed kernels can implement the control scheme the user specifies.*

Refer to the following table for the type of interface required based on the characteristics of the data movement in your application.

Table 6: **Kernel Interface Types**

Register (AXI4-Lite)	Memory Mapped (M_AXI)	Streaming (AXI4-Stream)
• Register interfaces must be implemented using a single AXI4-Lite interface. • Designed for transferring scalars between the host application and the kernel. • Register reads and writes are initiated by the host application. • The kernel acts as a slave.	• Memory mapped interfaces must be implemented using one or more AXI4 Masters interfaces. • Designed for bi-directional data transfers with global memory (DDR, PLRAM, HBM). • Introduces additional latency for memory transfers. • The kernel acts as a master accessing data stored into global memory. • The host application allocates the buffer for the size of the dataset. • The base address of the buffer is provided by the host application to the kernel via the AXI4-Lite interface.	• Streaming interfaces must be implemented using one or more AXI4-Stream interfaces. • Designed for uni-directional data transfers between kernels. • The access pattern is sequential. • Does not use global memory. • Data set is unbounded. • A sideband signal can be used to indicate the last value in the stream.

User-managed PL kernels have no predefined execution mode. It is up to the kernel designer to implement the control protocol and the execution mechanism. It is the application developer's responsibility to manage the operation of the kernel by executing appropriate sequences of register reads and writes from the software application, in accordance with the user-defined control protocol of the kernel.

XRT-managed PL kernels, as described in Supported Kernel Execution Models in the XRT documentation, provide defined kernel execution modes supporting overlapping execution of the kernel, or sequential execution.

- A kernel is started by the software application using an XRT API call. When the kernel is ready for new data, it notifies the host application through bits in the control register.
- The default control protocol, `ap_ctrl_chain`, supports pipelined execution enabling multiple executions of the same PL kernel to be overlapped to improve the overall application throughput.
- If required, pipelined execution can be disabled by using the `ap_ctrl_hs` control protocol which forces kernels to run sequentially, waiting until the prior run has completed before starting the next run.
- Finally, a kernel can be auto-restarting, allowing it to run for a specified number of iterations, or until reset by the host application as described in Auto-Restarting Kernels in *Vitis High-Level Synthesis User Guide* (UG1399).

Clock and Reset Requirements

The following clock and reset requirements apply to both software controllable and non-software controllable kernels.

Table 7: **Requirements**

HLS Kernel	RTL Kernel
• HLS kernel does not require any input from user on clock ports and reset ports. The HLS tool always generates RTL with clock port `ap_clk` and reset port `ap_rst_n`. • HLS kernels can only have one clock and reset.	• RTL kernels require at least one clock port, but a kernel can have multiple clocks. The number of clocks that an RTL can have is primarily determined by the number of clocks that the platform supports. Most embedded platforms can have multiple clocks. • An active-Low reset port can optionally be associated with a clock through the `ASSOCIATED_RESET` parameter on the clock.

Data Driven Kernels

Data driven kernels are present in the device and enabled automatically after downloading but are not directly managed by the software application, instead triggered by the arrival of data at the interface. The kernels must have at least one AXI4-Stream interface. The kernel synchronizes with the rest of the system through these streaming interfaces. These kernels can have scalar inputs and outputs connected through the AXI4-Lite interface. You can read or write to the kernels by native XRT register read and write APIs (`xrt::ip::read_register`, `xrt::ip::write_register`).

Data driven kernels do not require a software API, as the host application is not required to interact directly with the kernel. The kernel can be developed as either an RTL IP or synthesized from C/C++ source code by HLS compiler.

When compiled by HLS compiler, the data driven kernels are specified with `ap_ctrl_none` interface protocol. Refer to *Vitis High-Level Synthesis User Guide* (UG1399) for protocol usage.

RTL Kernel Development

In the AMD Vitis™ development flow, RTL IP from the Vivado Design Suite can be packaged as compiled AMD object (`.xo`) files) that can be linked into a device binary (`.xclbin`), as long as they adhere to Vivado IP packaging guidelines, and requirements of the Vitis compiler for linking the system.

RTL kernels can be user-managed kernels that do not meet XRT requirements for execution control, but rather implement any number of possible control schemes specified by existing RTL designs. Alternatively, RTL kernels can adhere to the requirements of the `ap_ctrl_chain` or `ap_ctrl_hs` control protocols needed for XRT-managed kernels.

The following sections describe the kernel interface requirements for the Vitis compiler to link kernels into a system. These requirements are common to software controllable and non-software controlled kernels. The control requirements for XRT-managed kernels are also described, in addition to any additional requirements. Finally, the development flow is described to help you package the RTL IP in theVivado Design Suite as RTL kernels for use in the Vitis environment.

To be integrated into the Vitis tool flow, an RTL module must minimally meet the requirements in RTL Kernel Interface Requirements. The need to meet the kernel interface requirements applies to both XRT-managed and user-managed kernels.

In addition, XRT-managed kernels must satisfy the requirements described in Control Requirements for XRT-Managed Kernels to be executed and profiled by XRT. RTL kernels support the hardware emulation and the hardware flows.

User-managed kernels must have the signal interfaces needed by the Vitis compiler to allow it to link the kernels to other kernels and to the target platform, but do not need to adhere to the strict execution protocol of XRT. In this way, existing RTL IP can be more rapidly and simply integrated into the Vitis environment.

Revise your RTL module to meet the kernel requirements outlined in the following sections.

RTL Kernel Interface Requirements

To enable the Vitis compiler to connect kernels into the target platform, an RTL kernel must also adhere to the requirements described in the following table.

Table 8: **RTL Kernel Interface and Port Requirements**

Port or Interface	Description	Comment
Clock	One or more clock inputs. ***Note*:** If multiple clocks are used by the RTL Kernel, you are responsible for designing proper clock domain crossing within the RTL Kernel.	• At least one clock is required for the kernel.[1] • Can be named anything, but must be packaged with a bus interface. **IMPORTANT!** *All ports in the RTL IP must be associated with an interface when packaging the RTL for use in the Vitis environment. If this is not the case, an error similar to the following occurs:* `ERROR: UNDEF When packaging for Vitis, pins that are not part of an interface are not supported`
Reset	Primary active-Low reset input port	• Optional port. • Can be named anything, but must be associated with a Clock signal through the ASSOCIATED_RESET property on the Clock. • This signal should be internally pipelined to improve timing. • The signal is driven by a synchronous reset in the associated Clock domain.
interrupt	Active-High interrupt.	• Optional port. • When used, the name must be exactly as shown.
s_axi_control	One (and only one) AXI4-Lite slave control interface	• Required port. The `s_axilite` interface is generally required with exception for some cases using AXI4-Stream interfaces. It is not required for non-software controlled kernels. • When used, the name must be exactly as shown, and is case-sensitive. **TIP:** *The address range of the `s_axilite` interface can be edited in the `kernel.xml` file and repackaged using the `package_xo` command if needed. However, XRT imposes a 64K (16 bit) address range limitation. The tool will return an error if the `s_axilite` interface is greater than 16 bits wide.*
AXI4_Memory Mapped Interface (m_axi)	AXI4 memory mapped interfaces for global memory access	• Optional port. • All AXI4 memory mapped interfaces must have 64-bit addresses. • The RTL kernel developer is responsible for partitioning global memory spaces. Each partition in the global memory becomes a kernel argument. The memory offset for each partition must be provided by the SW applications to the kernel through a register in the AXI4-Lite interface. • AXI4 memory mapped must not use Wrap or Fixed burst types and must not use narrow (sub-size) bursts. This means that AxSIZE should match the width of the AXI data bus. • Any user logic or RTL code that does not conform to the requirements above, must be wrapped or bridged to satisfy these requirements.

Table 8: **RTL Kernel Interface and Port Requirements** *(cont'd)*

Port or Interface	Description	Comment
AXI4_STREAM (axis)	AXI4-Stream interfaces for one-way data transfers between kernels or between the host application and kernels.	• Optional port. • Cannot be used with bi-directional ports. • Use the STREAM interface template in the Vivado Design Suite. • Refer to AXI4-Stream Interfaces for additional information on interface requirements in the *Vitis High-Level Synthesis User Guide* (UG1399).

Notes:

1. The RTL kernel clock frequency specification in the system can refer to Managing Clock Frequencies.

IMPORTANT! *The port names `interrupt` and `s_axi_control` must be defined exactly as shown.*

Control Requirements for XRT-Managed Kernels

IMPORTANT! *User-managed kernels do not require control registers and signals described below, but they can implement a control structure using registers in an `s_axilite` interface as discussed in Creating User-Managed RTL Kernels. If your RTL module implements a different control structure, you can define it as a `user_managed` kernel or it must be adapted to conform to the XRT-managed requirements described here.*

The following table outlines the required register map for an XRT-managed kernel to be used within the Vitis tools and XRT. The control register is required by kernels that specify `ap_ctrl_hs` and `ap_ctrl_chain` control protocols as described in HLS Kernel Interface Requirements. Kernels that implement `ap_ctrl_none` and `user_managed` control protocols do not require the control registers described below.

TIP: *The interrupt related registers are only required for designs that implement interrupts.*

All user-defined registers must begin at location `0x10`; locations below this are reserved. These include registers for kernel arguments such as scalar values and address offsets passed to memory mapped interfaces.

Table 9: **Register Address Map**

Offset	Name	Description
0x0	Control	Controls and provides kernel status.
0x4	Global Interrupt Enable	Used to enable interrupt to the host.
0x8	IP Interrupt Enable	Used to control which IP generated signals are used to generate an interrupt.
0xC	IP Interrupt Status	Provides interrupt status.
0x10	Kernel arguments	This would include scalars and global memory arguments for example.

The following table shows the control signals that are accessed through the control register (`offset 0x0`). The control register and its signals are determined by the kernel execution mode, `ap_ctrl_hs` and `ap_ctrl_chain`.

The available signals are used by the different control protocols as explained in Supported Kernel Execution Models in the XRT documentation. For example, for the sequential execution mode `ap_ctrl_hs` the host typically writes `0x00000001` to the offset 0 control register which sets Bit 0, clears Bits 1 and 2, and polls on reading `ap_done` signal until it is a 1.

Table 10: **Control Register Signals**

Bit	Name	Description
0	ap_start	Asserted when the kernel can start processing data. Cleared on handshake with ap_done being asserted.
1	ap_done	Asserted when the kernel has completed operation. Cleared on read.
2	ap_idle	Asserted when the kernel is idle.
3	ap_ready	Asserted by the kernel when it is ready to accept the new data
4	ap_continue	Asserted by the XRT to allow kernel keep running
7	auto_restart	Used to enable automatic kernel restart as described in the chapter Auto-Restarting Kernels in *Vitis High-Level Synthesis User Guide* (UG1399).
31:5	Reserved	Reserved

The following interrupt related registers are only required if the kernel has an interrupt.

Table 11: **Global Interrupt Enable (0x4)**

Bit	Name	Description
0	Global Interrupt Enable	When asserted, along with the IP Interrupt Enable bit, the interrupt is enabled
31:1	Reserved	Reserved

Table 12: **IP Interrupt Enable (0x8)**

Bit	Name	Description
0	Interrupt Enable	When asserted, along with the Global Interrupt Enable bit, the interrupt is enabled
31:1	Reserved	Reserved

Table 13: **IP Interrupt Status (0xC)**

Bit	Name	Description
0	Interrupt Status	Toggle on write
31:1	Reserved	Reserved

Interrupt

XRT-managed RTL kernels can optionally have an `interrupt` port containing a single interrupt. The port name must be called `interrupt` and be active-High. It is enabled when both the global interrupt enable (`GIE`) and interrupt enable register (`IER`) bits are asserted in the Control Register block.

The Vitis compiler (`v++`) will link the interrupt signal of a PL kernel into the available signals on the platform, provided the platform has interrupts available for connection as described in Adding Hardware Interfaces. If no interrupt is enabled on the platform, then you must manually connect the interrupt of the kernel.

By default, the IER uses the internal `ap_done` signal to trigger an interrupt. Further, the interrupt is cleared only when writing a 1 to bit-0 of the IP Interrupt Status Register.

This logic should be reflected in the Verilog code for the RTL kernel, and also in the associated `component.xml` and `kernel.xml` files. The `kernel.xml` file is stored inside the `kernel.xo` file and is generated automatically when using the `package_xo` command or RTL Kernel Wizard.

IMPORTANT! *The XRT native API does not support triggering or catching interrupts in the host application for user-managed RTL kernels.*

Creating User-Managed RTL Kernels

If your RTL IP does not satisfy the AXI interface requirements for the Vitis compiler as outlined in HLS Kernel Interface Requirements, you must modify the IP to implement the required interfaces. However, if your RTL IP does not satisfy the XRT control protocols of `ap_ctrl_hs` or `ap_ctrl_chain`, you can define it as a user-managed kernel rather than having to rewrite your IP.

A user-managed kernel does not need to satisfy the control requirements of XRT, and can implement any of a variety of execution mechanisms. User-managed kernels are meant to let you take advantage of the system building capabilities of the Vitis compiler, while letting your kernel implement your own control scheme. There is no prescribed method of starting or stopping, or otherwise controlling your kernel. This is largely up to you, and the specific requirements of your application or system. Some of the available control schemes include:

- Accessing registers through an `s_axilite` control interface, similar to the method used by XRT though open to your own implementation
- Accessing the hardware through software drivers, such as UIO drivers, implemented in your host application
- Triggering the start or stop response of your kernel from a signal provided by a separate component, or from another kernel

- Providing a data-driven approach, as described in the topic Auto-Restarting Mode in the *Vitis High-Level Synthesis User Guide* (UG1399).

> **IMPORTANT!** *One limitation of implementing a control register in an `s_axilite` interface for a user-managed kernel is that the control register cannot be named CTRL. That name is specifically reserved for XRT-managed kernels, and returns a Critical Warning when found on a user-managed kernel, or `ap_ctrl_none` kernel.*

Packaging the RTL Code as a Vitis XO

> **IMPORTANT!** *The RTL IP need to be first thoroughly verified with traditional RTL verification methods before being packaged as a kernel.*
>
> ***Note:*** XRT controlled RTL IP need to be packaged as Vitis XO for xclbin to contain RTL kernel information.

As discussed in RTL Kernel Interface Requirements, the RTL kernel must be packaged with the following required interfaces:

- The AXI4-Lite interface name must be packaged as `S_AXI_CONTROL`, but the underlying AXI ports can be named differently.
- Any memory-mapped AXI4 interfaces must be packaged as AXI4 master endpoints with 64-bit address support.

> **RECOMMENDED:** *AMD strongly recommends that AXI4 interfaces be packaged with AXI meta data `HAS_BURST=0` and `SUPPORTS_NARROW_BURST=0`. These properties can be set in an IP-level `bd.tcl` file. This indicates wrap and fixed burst type is not used, and narrow (sub-size burst) is not used.*

- You can also implement the AXI4-Stream interface.
- At least one clock is required for the kernel, though it can support multiple clocks.
  - Each clock must have an associated Bus Interface identifying it as a clock.
  - Each clock can have an optional active-Low reset, specified by the ASSOCIATED_RESET property on the clock.
  - A clock must be associated with each AXI4-Lite, AXI4, and AXI4-Stream interface on the kernel.

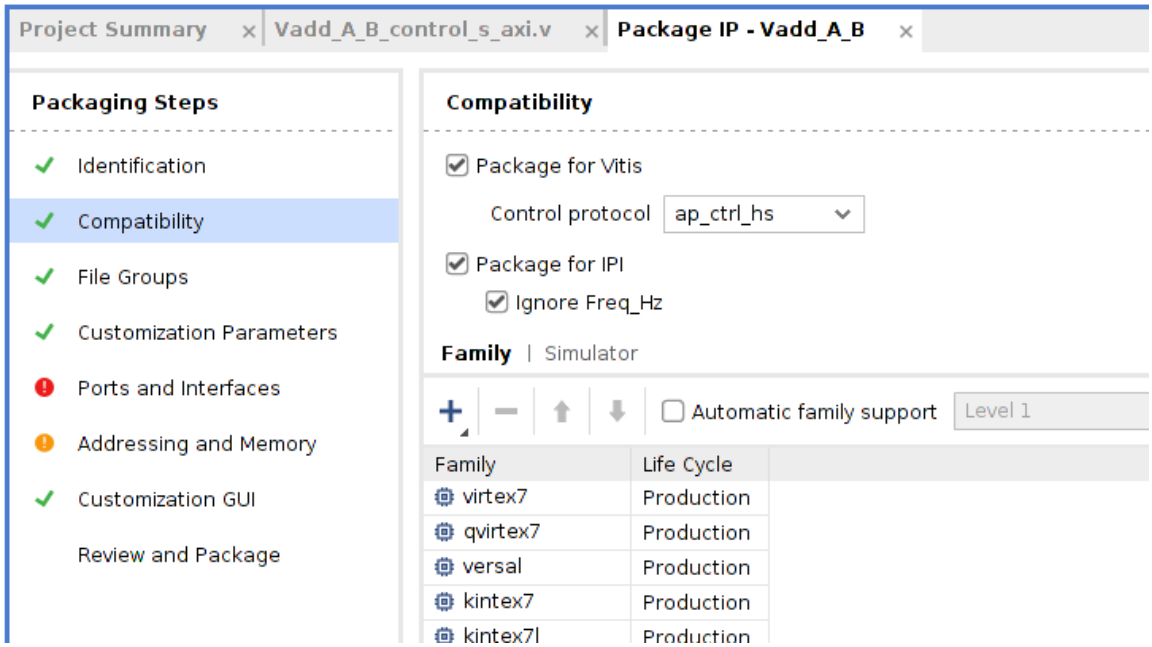
To package the IP, use the following steps:

1. Create and package a new IP.
   a. From a Vivado project, with your RTL source files added, select **Tools → Create and Package New IP**.
   b. Select **Package your current project**, and click **Next**.
   c. Specify the location for your packaged IP. You can select the default location, or choose a different location.

d. Review the Summary page and click **Finish** to open the Package IP window.

The Package IP window opens to display the Identification page. For details on working with the IP packager in the Vivado tool, refer to the *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118).

2. Select **Compatibility** under the Packaging Steps. This displays the Compatibility view as shown in the following figure.



a. Select the **Package for Vitis** check box to enable the process of packaging the RTL IP as an XO for use in the Vitis environment.

b. Select the **Control Protocol** for the RTL Kernel. This determines the control mechanism used to operate the kernel. The choices are:

- `user_managed`: Defines a SW-controllable kernel, that is user-managed rather than XRT-managed. This is the preferred option. Refer to Creating User-Managed RTL Kernels for additional information.
- `ap_ctrl_hs`: This is the default, and specifies the simple sequential execution model for XRT -managed kernels as described in Control Requirements for XRT-Managed Kernels.
- `ap_ctrl_chain`: Specifies a pipelined execution model for XRT-managed kernels.
- `ap_ctrl_none`: Indicates no control protocol as described in Data Driven Kernels.

c. Check to ensure that both **Package for IPI** and **Ignore Freq_Hz** are enabled as well.

Enabling these check boxes enables design rule checks (DRC) that the `ipx::check_integrity` command runs prior to packaging the IP and generating the XO. The DRCs include checks for required signals as described in Requirements of an RTL Kernel in the *Data Center Acceleration using Vitis* (UG1700), and checks for control protocols and registers for XRT-managed kernels. As shown in the preceding figure, any issues are reported to the Package IP tool as they are encountered.

3. Associate the clock to the AXI interfaces.

   Select the **Ports and Interfaces** step of the Package IP window, you can associate the primary kernel clock with the AXI4 interfaces, and reset signal if needed.

   a. Right-click an AXI4 interface, and select **Associate Clocks**.

      This opens the Associate Clocks dialog box which lists any identified clock signals.

   b. Select the appropriate clock and click **OK** to associate it with the interface.

   c. Ensure to repeat this step to a clock signal with each of the AXI interfaces.

4. Click the **Addressing and Memory** step to add control registers and offsets.

   XRT-managed kernels using the `ap_ctrl_hs` or `ap_ctrl_chain` control protocol require control registers as discussed in Control Requirements for XRT-Managed Kernels. The following table shows a list of the required registers.

> **TIP:** *While `ap_ctrl_none` and `user_managed` control protocols do not require control registers, they can still use them if an `s_axilite` interface is included as part of the RTL design. In this case, the specific registers can differ from the following table, but the process of assigning `names`, `offsets`, and `widths` is the same.*

Table 14: **Address Map**

Register Name	Description	Address Offset	Size
CTRL	Control Signals as described in Control Requirements for XRT-Managed Kernels.	0x000	32
GIER	Global Interrupt Enable Register. Used to enable interrupt to the host.	0x004	32
IP_IER	IP Interrupt Enable Register. Used to control which IP generated signal are used to generate an interrupt.	0x008	32
IP_ISR	IP Interrupt Status Register. Provides interrupt status.	0x00C	32
<kernel_args>	This includes a separate entry for each kernel argument as needed on the software function interface. All user-defined registers must begin at location `0x10`; locations below this are reserved.	0x010	32/64 Scalar arguments are 32-bits wide.`m_axi` and `axis` interfaces are 64 bits wide.

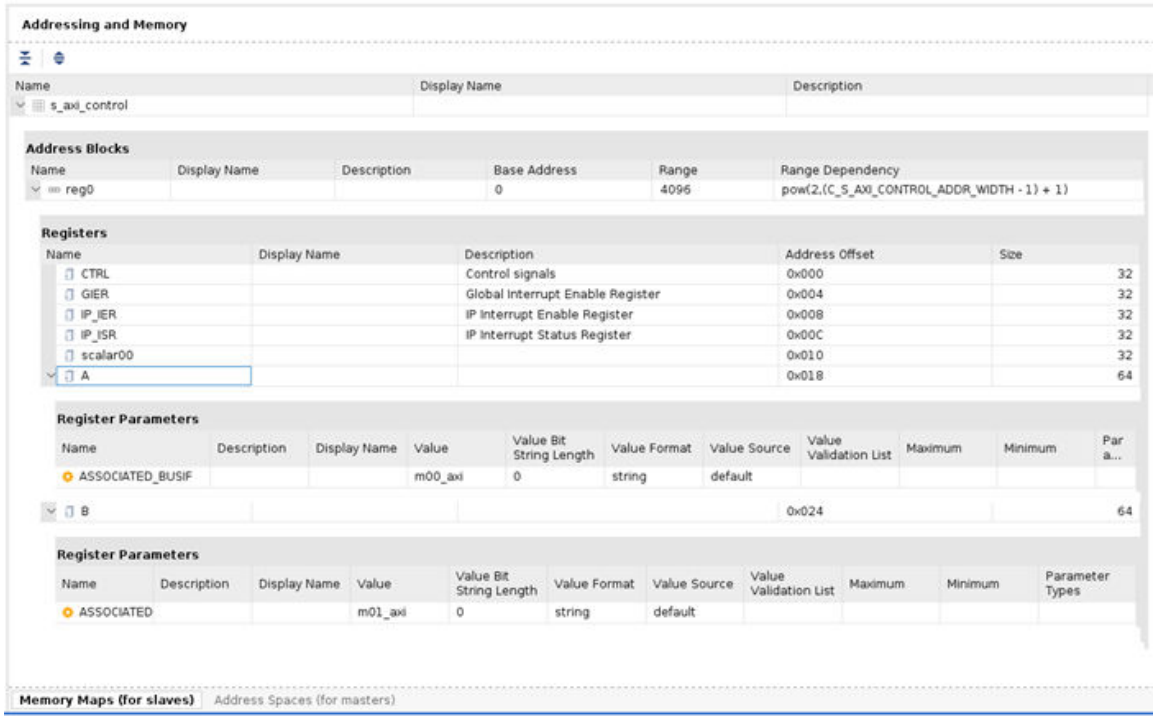
   a. To create the address map described in the table, right-click the Address Blocks and select the **Add Register** command.

This opens the Add Register view in which you can enter one of the register names from the preceding table.

> **IMPORTANT!** *The Range value under Address Blocks specifies the address range for the `s_axilite` interface. You can modify this value to change the range for the kernel.*

b. Repeat as needed to add all required registers.

This creates a Registers table in the Addressing and Memory section. You can edit the table to add the Description, Address Offset, and Size to each register. The Registers table should look similar to the following example.



> **TIP:** *The Tcl commands for each step of this process are written to the Tcl Console. You can use this fact to execute the process, and then use the Tcl transcript to create scripts to automate the process for future iterations.*

c. Finally, select the register for each of the pointer arguments from your table, right-click and select the **Add Register Parameter** command. Enter the name `ASSOCIATED_BUSIF` into the dialog box that opens, and click **OK**.

This lets you define an association between the register and the AXI4 Interface. In the value field of the added parameter, enter the name of the `m_axi` interface assigned to the specific argument you are defining. In the example above, the argument `A` uses the `m00_axi` interface, and the argument `B` uses the `m01_axi` interface.

5. At this point you should be ready to package your IP.

a. Select the **Review and Package** section of the Package IP view, review the Summary and After Packaging sections, and make whatever changes are needed.

> **IMPORTANT!** *You must enable the generation of an IP archive file. If the After Packaging section indicates An archive will not be generated, you must select the* ***Edit packaging settings*** *link and enable the* ***Create archive of IP*** *setting.*

b. When you are ready, click **Package IP**.

   The Vivado tool packages your kernel IP, automatically runs the `package_xo` command as needed to produce the XO file, and opens a dialog box to inform you of success.

   The generated XO file for the RTL kernel can be used by the Vitis compiler during the linking process to connect to other HLS or RTL kernels, and for linking with the target platform to complete the system. Refer to Chapter 8: Building and Running the System for more information.

c. If your RTL kernel has some custom features that are not standard for the `package_xo` command that is run automatically, you can run the command manually to regenerate the XO file and kernel with custom settings. Refer to package_xo Command in the *Vitis Reference Guide* (UG1702) for details of the command. Some specific reasons why you might need to manually run the `package_xo` command include:

   - Specify a different IP directory or XO path
   - Output a copy of the `kernel.xml` file to modify it and repackage

6. Optional: Test the Packaged IP.

   To test if the RTL kernel is packaged correctly for the IP integrator, try to instantiate the packaged kernel IP into a block design in the IP integrator. For information on the tool, refer to *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994).

   The kernel IP should show the various interfaces described above. Examine the IP in the canvas view. The properties of the AXI interface can be viewed by selecting the interface on the canvas. Then in the Block Interface Properties view, select the **Properties** tab and expand the CONFIG table entry. If an interface is to be read-only or write-only, the unused AXI channels can be removed and the `READ_WRITE_MODE` is set to read-only or write-only.

7. Optional: Configure Design Constraints.

   If the RTL kernel has design constraints (`.xdc`) which refer to elements of the static region of the platform, such as clocks, then the constraint file needs to be marked as late processing order to ensure RTL kernel constraints are correctly applied.

   There are two methods to mark constraints for late processing:

   a. If the constraints are given in a `.ttcl` file, add `<: setFileProcessingOrder "late" :>` to the `.ttcl` preamble section of the file as follows:

```
<: set ComponentName [getComponentNameString] :>
<: setOutputDirectory "./" :>
<: setFileName $ComponentName :>
<: setFileExtension ".xdc" :>
<: setFileProcessingOrder "late" :>
```

b. If constraints are defined in an `.xdc` file, then add the following four lines starting at `<spirit:define>` in the `component.xml`. The four lines in the `component.xml` need to be next to the area where the `.xdc` file is called. In the following example, `my_ip_constraint.xdc` file is being called with the subsequent late processing order defined.

```
<spirit:file>
        <spirit:name>ttcl/my_ip_constraint.xdc</spirit:name>
        <spirit:userFileType>ttcl</spirit:userFileType>
        <spirit:userFileType>USED_IN_implementation</
spirit:userFileType>
        <spirit:userFileType>USED_IN_synthesis</spirit:userFileType>
        <spirit:define>
             <spirit:name>processing_order</spirit:name>
             <spirit:value>late</spirit:value>
        </spirit:define>
</spirit:file>
```

Design Recommendations for RTL Kernels

RTL kernels should be designed with recommendations from the *UltraFast Design Methodology Guide for FPGAs and SoCs* (UG949). In addition to adhering to the interface and packaging requirements, the kernels should be designed with the following performance goals in mind.

Memory Performance Optimizations for AXI4 Interface

The AXI4 interfaces typically connects to DDR memory controllers in the platform.

RECOMMENDED: *For optimal frequency and resource usage, it is recommended that one interface is used per memory controller.*

For best performance from the memory controller, the following is the recommended AXI interface behavior:

- Use an AXI data width that matches the native memory controller AXI data width, typically 512-bits.
- Do not use `WRAP`, `FIXED`, or sub-sized bursts.
- Use burst transfer as large as possible (up to 4k byte AXI4 protocol limit).
- Avoid use of deasserted write strobes. Deasserted write strobes can cause error-correction code (ECC) logic in the DDR memory controller to perform read-modify-write operations.
- Use pipelined AXI transactions.
- Avoid using threads if an AXI interface is only connected to one DDR controller.
- Avoid generating write address commands if the kernel does not have the ability to deliver the full write transaction (non-blocking write requests).
- Avoid generating read address commands if the kernel does not have the capacity to accept all the read data without back pressure (non-blocking read requests).

- If a read-only or write-only interfaces are desired, the ports of the unused channels can be commented out in the top level RTL file before the project is packaged into a kernel.
- Using multiple threads can cause larger resource requirements in the infrastructure IP between the kernel and the memory controllers.

Quality of Results Considerations

The following recommendations help improve results for timing and area:

- Pipeline all reset inputs and internally distribute resets avoiding high fanout nets.
- Reset only essential control logic flip-flops.
- Consider registering input and output signals to the extent possible.
- Understand the size of the kernel relative to the capacity of the target platforms to ensure fit, especially if multiple kernels will be instantiated.
- Recognize platforms that use stacked silicon interconnect (SSI) technology. These devices have multiple die and any logic that must cross between them should be flip-flop to flip-flop timing paths.

Debug and Verification Considerations

- RTL kernels should be verified in their own test bench using advanced verification techniques including verification components, randomization, and protocol checkers. The AXI Verification IP (VIP) is available in the Vivado IP catalog and can help with the verification of AXI interfaces. The RTL kernel example designs contain an AXI VIP-based test bench with sample stimulus files.
- You can add ILA inside of RTL kernels as described in Debugging with ChipScope.
- Hardware emulation should be used to test the host code software integration or to view the interaction between multiple kernels.

Chapter 5

Vitis Subsystem

The AMD Vitis™ subsystem allows combining AI Engine and PL kernels mapping to a part to create a platform-independent subsystem. This chapter describes how to setup and use a Vitis subsystem.

Preparing a Vitis Subsystem

The Vitis subsystem enables a bottoms-up methodology by providing one or several intermediate integration stages for combining AI Engine graphs, programmable logic kernels, and/or another VSS into a subsystem.

The PL kernels is created using either HLS or RTL, but delivered to Vitis subsystem as packaged kernels (Vitis XO). For a description how to prepare PL kernels, see HLS Kernel Development and RTL Kernel Development

For preparing AI Enginegraphs, follow the links presented in AI Engine Kernel and Graph Development. VSS support AI Enginepartitions, but require special conditions according to the following list.

- Each partition compiles into a `libadf`, which need inclusion when linking a VSS component.
- Partitions do not overlap in column assignment.
- There is only one AI Engineinstance which is reflected in the VSS linker connectivity.
- Simulating a VSS with AI Engine is only allowed with a single partition. A multiple partition design need to be merged into a temporary single partition AI Engine graph to use with the simulation.

You can create hierarchies by linking a VSS into another VSS. The VSS additively links to a extensible platform. You can combine VSS with additional Vitis kernels when linking the design to a platform.

IMPORTANT! *You can only link a VSS to another VSS or extensible platform that use the same part number.*

Linking a VSS Component

A Vitis Subsystem (VSS) is a platform-independent, reusable design component that can be customized with AI Engine and PL content. The VSS is integrated into a larger system using Vitis tools.

A configuration file is used to declare the specific PL kernels and AI Engine graphs to be used, as well as the interface connections between the kernels and graphs. The syntax is similar to how the Vitis linker connects components to an extensible platform, except with the additional declaration of kernels, graph, or other VSS components to be added to a VSS component. The VSS component can be defined as follows:

```
vss=amd.com:<vss_library_name>:<vss_component_name>:<version_number>:<list_o
f_instances of PL kernels and/or AI Engine graphs>

[connectivity]
nk=producer:2:p0,p1
nk=consumer:2:c0,c1

vss=amd.com:myLib:MyVSSComponent:1.0:p0,p1,c0,c1,ai_engine_0

stream_connect=p0.outs:c0:ins
stream_connect=p1:outs:ai_engine_0.si_0
stream_connect=ai_engine_0.so_0:c1:ins
```

Table 15: **Description of the VSS Component Syntax**

Item	Example	Description
vss_library_name	my_vss_lib	Identifier name to associate the VSS to a library.
vss_component_name	vss_top	Will be used as VSS archive name and concatenated with instance names to provide unique name when integrating VSS component to an extensible platform.
version_number	1.0	To keep track of library/component version.
list_of_instance	counter_0,ai_engine_0	Comma separated list of instance names. The names need to match kernel names declared with "nk" in the config file. ***Note***: The ai_engine_0 instance name need to match the default AI Engine IP instance name in the Vivado HW platform.

Hierarchical VSS

The Vitis subsystem provides hierarchical constructs for organizing complex designs, allowing for multiple VSS components within a single VSS. PL kernels can be arranged within these VSS components. Each of these VSS components must be created independently before adding them in another VSS component.

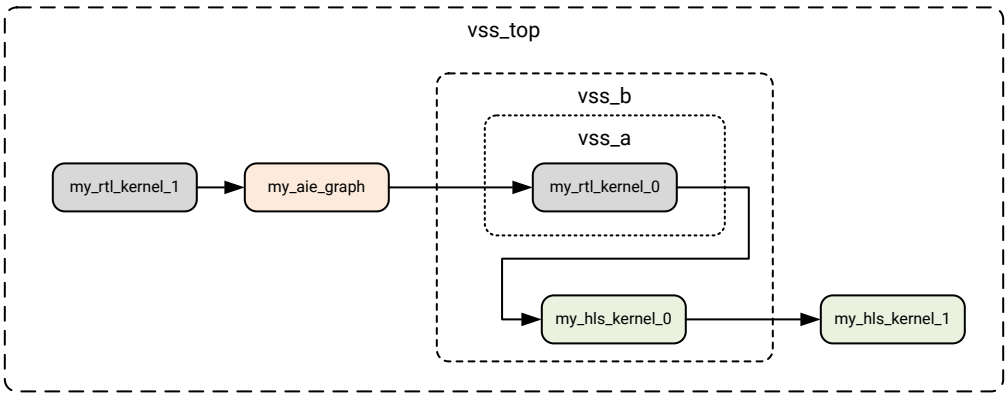
To handle multiple independent AI Engine graphs, consider using the AI Engine partition feature. Every AI Engine partition must be prepended by the AI Engine instance IP name. There can only be a single instance of the AI Engine IP in a Versal AI Engine design.

During integration of a VSS component into another VSS or platform, the compiler flattens the design and resolves instance names by prepending the corresponding VSS component names based on the hierarchical structure.

IMPORTANT! *Proper naming convention for the kernels and VSS components is important when resolving the flattened instance name. Choose the names with care to avoid ambiguity.*

The following example illustrates how VSS and kernel instance names are resolved in a hierarchical use case.

Figure 9: **Hierarchical VSS Example**



The innermost VSS component `vss_a` contains a RTL kernel with instance name `my_rtl_kernel_0`. The VSS component `vss_b` instantiates `vss_a` and a HLS kernel `my_hls_kernel_0`. In this example, the AI Engine contains a single graph with one input port named `in0` and one output port named `out0`. It is placed in the VSS component `vss_top` together with `vss_b`, `my_rtl_kernel_1` and `my_hls_kernel_1`.

```
# Declare VSS component A
vss=amd.com:my_vss_lib:vss_a:1.0:my_rtl_kernel_0
```

```
# Declare hierarchical VSS component B
vss=amd.com:my_vss_lib:vss_b:1.0:my_hls_kernel_0,vss_a
# Establish connection from kernel in vss_a  to kernel in vss_b
sc=vss_a_my_rtl_kernel_0.port_out0:my_hls_kernel_0.port_in0
```

```
# Declare  hierarchical VSS component top
vss=amd.com:my_vss_lib:vss_top:1.0:ai_engine_0,my_rtl_kernel_1,my_hls_kernel
_1,vss_b

# Flattened names are used when connecting ports to instances:
# Connections in vss_top
sc=my_rtl_kernel_1.port_out0:ai_engine_0.in0
# Establish connection from vss_top  through vss_b into vss_a to connect to
the RTL kernel
sc=ai_engine_0.out0:vss_b_vss_a_my_rtl_kernel_0.port_in0
# Establish connection from vss_b kernel  to kernel in vss_top
sc=vss_b_my_hls_kernel_0.port_out0:my_hls_kernel_1.port_in0
```

Generating the VSS Component

The VSS archive is generated using the linker with `--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>
```

Table 16: **Description of Files and Folders Created When Generating a VSS Component**

Folder	Files	Description
<build_folder>	`*.log`, `*.link_summary`	Output folder containing log files and summary from v++ linker.
<build_folder>/<vss_component_name>	`<vss_component_name>.vss`	VSS Component archive.
	`*.a`	AIE graph libadf files.
	`*.xo`	RTL and HLS packaged kernels.
<build_folder>/<vss_component_name>/ip_repo/*		Folders with files associated with packaged HLS and RTL kernels.
<build_folder>/<vss_component_name>/VitisRegion		Blockdesign metadata for VitisRegion.
<build_folder>/_x		Temporary work folder for v++ linker.

Simulation and Verification in Vitis

This chapter describes various approaches to simulate and verify AMD Vitis™ kernels and subsystems. They are categorized in three levels, Functional, Cycle approximate, and Performance, each serving different purpose in verifying functional and performance of design components prior to integrating them to a larger system. The following table compares the differences and applications of the methods.

Table 17: **Comparison of Verification Methods**

Feature	Applicable to	Verification Type	Comments
Vitis Functional Simulation	• AI Engine graphs • HLS kernels • Systems of AI Engine graphs and HLS kernels	Functional	Focuses on functional correctness using x86 simulation and c models for fast simulation in MATLAB or Python environments. Systems are created by instantiating multiple simulation objects and pass data via the simulation test bench.
Vitis Model Composer	• AI Engine graphs and kernels • HLS kernels • RTL using Vitis Model Composer library elements • Systems of AI Engine and PL graphs and kernels	Functional or Cycle approximate	Vitis Model Composer enable rapid design exploration and can generate images for accelerated verification on hardware.
Vitis Subsystem Simulation	VSS component comprising of AI Engine graphs, RTL	Cycle approximate	Uses a RTL test bench to drive and monitor AXI4-Stream traffic to and from a VSS component. Provides a lightweight system simulation to verify cycle approximate PLIO handshaking between PL kernels and AIE graphs.
AI Engine testharness	AI Engine graphs	Performance	Uses a pre-built hardware design infrastructure to quickly test AI Engine graphs with PLIO traffic.

For details, see the corresponding chapters Vitis Functional Simulation Overview, *Vitis Model Composer User Guide* (UG1483), Simulation with the Vitis Subsystem, and AI Engine Test Harness in the *AI Engine Tools and Flows User Guide* (UG1076).

Vitis Functional Simulation Overview

In AMD Vitis™ tools, there have traditionally been two options for functional simulation of subsystems. The first is Vitis Model Composer, which enables simulation of subsystems comprising of AIE engines and programmable logic (PL) within the Simulink graphical environment. The second is a command-line interface (CLI) approach, which involves manually providing data files for each simulation run to model the subsystem.

Vitis Functional Simulation (VFS) introduces an alternative by supporting simulation management of the design under test (DUT) in both MATLAB and Python—two widely used, high-level, text-based environments.

Supported Versions and Setup

VFS is supported with the following tools:

- Python versions 3.9, 3.10, 3.11, 3.12, and 3.13
- MATLAB R2024a, R2024b, R2025a, and R2025b.

VFS is automatically set up when running the script as described in Setting Up the Vitis Environment. To use in Python, add VFS and VARRAY as a libraries with:

```
import vfs
import varray
```

VFS is designed to manage AI Engine graphs and HLS kernels as simulation objects. These objects are instantiated in a test bench and simulated by running the instance with the provided input data. The simulations of the AI Engine and HLS kernels are bit-accurate but not cycle-accurate, as they aim to demonstrate functionality. The methods for instantiation are in the VFS Objects section.

***Note*:** Currently, there are limitations on HLS kernels that can get simulated in VFS. These limitations apply to kernels using tasks, return non void value, or use non-blocking read or writes.

A test bench can include multiple simulation objects, allowing both block-level and subsystem-level functional verification.

To simplify data management for the simulation objects, VFS uses the VARRAY library. The library provides data constructs for handling and converting stimuli data created in the test bench to and from the simulation object. The VARRAY Supported Data Types section describes the types and syntax used for AI Engine and HLS simulation object input and output data structures.

See the following Vitis tutorials using this feature.

- Farrow Filters Tutorial
- Black Projection SAR Tutorial
- Vitis Functional Simulation Tutorial

VFS Objects

You can use VFS object classes to instantiate and control the simulation of AI Engine graphs and HLS kernels. You can instantiate a VFS simulation object using either `aie_graph_handle = vfs.aieGraph(<arg>)` or `hls_kernel_handle = vfs.hlsKernel(<arg>)`. The VFS object is compiled and simulated with a `<vfs_handle>.run` method. This method checks the compilation status of the object and acts as an API to pass input and output to the object. The `run` method is described in Methods to Pass Inputs and Get Outputs with the Run API.

Provide the following instantiation arguments.

1. Key-Value pairs
2. Configuration file
3. Reference to existing build directory of an object

The first two initialization options share the following properties:

- They create a `vfsBuildDir` under the component's work directory.
- Both absolute and relative paths are accepted. The relative paths are relative to the location of the MATLAB or Python script.
- All files and directories are checked for correctness.
- An error is thrown if validation fails.

During test bench execution, VFS creates a `vfs_work` folder structure and compiles the design if required.

The third option is useful when the object has already been compiled and is subject to validation or added as a sub component to verify with a larger system.

Note: Relative paths are determined from the location of the VFS test-bench file.

Passing Key Value Pairs

The key-value pairs method provides a flexible handling of the simulation objects, as the values can be defined or generated by the test bench. VFS uses these parameters to generate a configuration file and checks for a corresponding existing build. If there is no matching build, a new build is initiated.

The following is an example of an AIE Graph using key-value pairs:

```
# MATLAB example for AIE Graph vfs object using key-value pairs:
aie_graph = vfs.aieGraph(
  input_file = '<path_to_src>/front_ifft_with_twid.cpp',...
  platform = '<path_to_platforms>/xilinx_vck190_base_202520_1.xpfm',...
  include_paths = {...
    '<path_to_src>',...
    '<path_to_inc>',...
    '<path_to_libraries>'...
    });
```

The following is an example of an HLS Kernel using key-value pairs:

```
# Python example for HLS Kernel vfs object using key-value pairs:
hls_kernel = vfs.hlsKernel(
        part = 'xcvs1902-vsva2197-2MP-e-S',
        hls_function = '<top_function_name>',
        input_files = [
            '../prj/hls_src/ifft_transpose.cpp',
            '../prj/hls_src/ifft_transpose.h'
        ]
)
```

Details for Key-value pairs:

- The `part` or `platform` is used to select the device. If not specified, the default is `xcvs1902-vsva2197-2MP-e-S`.
- For AI Engine graphs:
  - `input_file` is a path to a `.cpp` top test bench that instantiates the graph and contains the main function.
  - A directory named `input_file_hash` is created.
- For HLS kernels:
  - `hls_function` declares the kernel that is the top function.
  - A directory named `top_function_hash` is created

Table 18: **Key-Value Pair Requirements**

Key	Applies to VFS object type	Type	Required	Description
`input_file`	`aieGraph`	string	Yes	Point to the top test bench graph file.

Table 18: **Key-Value Pair Requirements** *(cont'd)*

Key	Applies to VFS object type	Type	Required	Description
`input_files`	`hlsKernel`	array (Python) or cell array (MATLAB) of string	Yes	Point to source files used by the HLS kernel.
`hls_function`	`hlsKernel`	string	Yes	Top function name of the HLS kernel.
`include_paths`	`aieGraph` / `hlsKernel`	array (Python) or cell array (MATLAB) of string	No	Path to include files.
part	`aieGraph` / `hlsKernel`	string	No	Select target device using part number. Default is `xcvs1902-vsva2197-2MP-e-S`.
platform	`aieGraph` / `hlsKernel`	string	No	Path to a platform `.xpfm` file. ***Note***: If this option is used, omit the `part` key-value pair.

Passing a Configuration File

The following example demonstrates how to use the `configuration file` when instantiating the simulation object. Compilation occurs the first time the object is created or if a dependent file is modified. When the test bench is re-run, it checks for any changes in source files or configuration file parameters and reconstructs the object with a new hash if necessary.

```
# AIE Graph example using config file (MATLAB/Python/C++):
myGraph = vfs.aieGraph(config_file = "<path_to_design>/my_aie_config.cfg")
# HLS Kernel example using config file:
myKernel = vfs.hlsKernel(config_file = "<path_to_design>/my_hls_config.cfg")
```

When using the configuration file, the following conditions apply:

- The configuration file must be the only parameter specified
- For AI Engine, a directory named `input_file_hash` is created
- For HLS, a directory named `top_function_hash` is created

Example of a configuration file for a VFS graph object:

```
include = <path_to>/src
include = <path_to>/inc
input_files = <aie_graph_top>.cpp
part = xcvc1902-vsva2197-1LP-e-S
target = x86sim
```

Example of a configuration file for a VFS kernel object:

```
part = xcvc1902-vsva2197-1LP-e-S
[hls]
flow_target = vivado
package.output.format = ip_catalog
package.output.syn = false
syn.cflags =
syn.file = <hls_kernel>.cpp
syn.top = <hls_kernel>
```

The following is an example of the `vfs_work` folder structure using the configuration file option:

```
vfs_work
  |-→ <name_from_cfg>_hash
        |-→ work (aie or hls work directory)
        |-→ <copy_of_config_file>
        |-→ vfsBuildDir
              |-→ libvfssim.so
              |-→ vfs_interface_spec.json
```

Passing a VFS build directory

To use an existing build directory to instantiate a VFS object, the keyword `vfs_build_dir` is used.

```
# Python example for HLS Kernel
myKernel = vfs.hlsKernel(vfs_build_dir = "<path_to_build_dir>")
```

```
# MATLAB example for AIE Graph
myGraph = vfs.aieGraph(vfs_build_dir = "<path_to_build_dir">);
```

This use case skips compiling the object, but the specified folder must contain a `libvfssim.so` and a `vfs_interface_spec.json` files.

Methods to Pass Inputs and Get Outputs with the Run API

The simulation object produce outputs only when sufficient input data is provided. To help determine relevant input and output sizes, you can query an AI Engine graph object or HLS kernel for the expected input and output data size and types using the `<vfs_object>.getInputSpec()` and `<vfs_object>.getOutputSpec()` methods. The following example shows a query of the input specification of a graph object.

```
    Index         Name          DataType         Type          Size
    -----     -------------     --------     ------------     -----

      1       "fir_sig_in0"     "cint16"     "iobuffer"       "256"
      2       "fir_sig_in1"     "cint16"     "iobuffer"       "256"
      3       "coeff[0]"        "int16"      "rtp(async)"     "16"
      4       "coeff[1]"        "int16"      "rtp(async)"     "16"
```

There are three methods available to pass the data.

The following table illustrates an HLS kernel with two input ports and two outputs as an example. Each input port requires 4 samples of `int16` data, and the kernel produces two outputs of the same size.

Table 19: **Comparing Input Methods**

Method to pass data	Example	Comments
Each input separately	`# Python example` `import vfs` `import varray as va` `import numpy as np` `in1 =` `va.array([0,1,2,3],va.int16)` `in2` `= va.array(np.array(range(4))` `+10,va.int16)` `out = myKernel.run(in1, in2);` `% MATLAB example` `in1 = varray.int16([0:3]);` `in2 = varray.int16([0:3]+10);` `[out1, out2] =` `myKernel.run(in1, in2);`	In this case, one input is used per port. ***Note*:** If the input vector size is smaller than specified, the output is not produced until the next run provides sufficient input data to meet the graph or kernel specification. In this case, the output is an empty array.
Cell array in MATLAB or a list in Python	`# Python example` `in_list = [in1 in2]` `out_list =` `myKernel.run(in_list)` `% MATLAB example` `in_cellarray = {in1, in2};` `out_cellarray =` `myKernel.run(in_cellarray);`	This case is suitable for kernels or graphs with numerous inputs or outputs, providing a compact style for managing the data.
Matrix	`# Python example` `in_matrix =` `va.array(np.matrix('0, 10; 1,` `11; 2, 12; 3, 13',va.int16));` `out_list =` `myKernel.run(in_matrix)` `% MATLAB example` `in_matrix = varray.int16([0,` `10; 1, 11; 2, 12; 3, 13]);` `[out1, out2] =` `myKernel.run(in_matrix);`	For matrix inputs, the shape of the matrix must match the input specification. The columns of the matrix represent the ports. **IMPORTANT!** *Only one input matrix is supported.* ***Note*:** The output format with this method is a list in Python or cell array in MATLAB. It is up to the user to reformat the output into a matrix if needed.

VARRAY Supported Data Types

Varray library uses the `varray` class to handle data to and from simulation objects. The `varray` class represents data as an array and support data types that are compatible with AI Engine and HLS kernels.

To form a subsystem, two or more simulation objects can be chained together using intermediate `varray` to pass data from one to another.

Table 20: **Supported AI Engine Data Types**

AIE data type	Supported in AIE	Supported in AIE-ML	Supported in AIE_MLv2	MATLAB	Python Numpy	varray type
int8	Yes	Yes	Yes	int8	numpy.int8	varray.int8
uint8	Yes	Yes	Yes	uint8	numpy.uint8	varray.uint8
int16	Yes	Yes	Yes	int16	numpy.int16	varray.int16
uint16	Yes	Yes	Yes	uint16	numpy.uint16	varray.uint16
cint16	Yes	Yes	Yes	complex int16	N/A	varray.cint16
int32	Yes	Yes	Yes	int32	numpy.int32	varray.int32
uint32	Yes	Yes	Yes	uint32	numpy.uint32	varray.uint32
cint32	Yes	Yes	Yes	complex int32	N/A	varray.cint32
float8	No	No	Yes	N/A	N/A	varray.float8
float16	No	No	Yes	N/A	numpy.float16	varray.float16
float	Yes	Yes	Yes	single	numpy.float32	varray.float
cfloat	Yes	Yes	No	complex single	numpy.float64	varray.cfloat
bfloat8	No	No	Yes	N/A	N/A	varray.bfloat8
bfloat16	No	Yes	Yes	N/A	N/A	varray.bfloat16
cbfloat16	No	Yes	No	N/A	N/A	varray.cbfloat16
mx9	No	No	Yes	N/A	N/A	varray.mx9
mx6	No	No	Yes	N/A	N/A	varray.mx6
mx4	No	No	Yes	N/A	N/A	varray.mx4

***Note*:** The data types support differ per device family. For details on the bfloat and mx format support refer to Data Types in the *AI Engine-ML Kernel and Graph Programming Guide* (UG1603).

Table 21: **Supported HLS Kernel Data Types**

HLS data type	MATLAB	Python Numpy	varray type
int4	N/A	N/A	varray.int4
uint4	N/A	N/A	varray.uint4
char	int8	numpy.int8	varray.int8
unsigned char	uint8	numpy.uint8	varray.uint8
std::complex<char>	complex int8	N/A	varray.cint8
short	int16	numpy.int16	varray.int16
unsigned short	uint16	numpy.uint16	varray.uint16
std::complex<short>	complex int16	N/A	varray.cint16
int	int32	numpy.int32	varray.int32
unsigned int	uint32	numpy.uint32	varray.uint32
std::complex<int>	complex int32	N/A	varray.cint32
long long	int64	numpy.int64	varray.int64
unsigned long long	uint64	numpy.uint64	varray.uint64

Table 21: **Supported HLS Kernel Data Types** *(cont'd)*

HLS data type	MATLAB	Python Numpy	varray type
float16	N/A	numpy.float16	varray.float16
float	single	numpy.float32	varray.float
std::complex<float>	complex single	numpy.complex64	varray.cfloat
double	double	numpy.double	varray.double
std::complex<double>	complex double	numpy.complex128	varray.cdouble
ap_fixed<W,I,Q,O,N>	N/A	N/A	varray.fi(1, W, W-I)
ap_ufixed<W,I,Q,O,N>	N/A	N/A	varray.fi(0, W, W-I)
ap_int<W>	N/A	N/A	varray.fi(1, W, 0)
ap_uint<W>	N/A	N/A	varray.fi(0, W, 0)

***Note*:** The ap_fixed arguments are Word length, number of Integer bits, Quantization mode, Overflow mode, and Number of saturation bits, see Arbitrary Precision Fixed-Point Data Types in the *Vitis High-Level Synthesis User Guide* (UG1399) for details.

***Note*:** The `varray.fi` types use rounding mode `Nearest` and overflow mode `Saturate`.

Operators, functions and casting on VARRAY objects

The varray objects support comparative operators and operations listed in the tables below. Any operation that perform value modification is required to cast the array to native data formats prior to performing math operations.

To casting from varray to language native data type use `numpy.asarray(<vfs_array>, <dtype>)` with Python or `cast(<vfs_array>)` with MATLAB. If omitted, the data type is inferred for Python. For MATLAB, the data type default to double precision floating point. For details on Python NumPy and MATLAB operations, refer to respective language API reference manuals.

Table 22: **Supported varray operations**

Python function	MATLAB methods	Notes
numpy.imag	imag()	Returns a varray with the imaginary part of the data type.
numpy.real	real()	Returns a varray with the real part of the data type.
numpy.concatenate	vertcat() / horzcat()	Concatenate arrays ***Note*:** All inputs must be varrays of the same data type.
numpy.transpose	transpose / .	Transpose the varray object. ***Note*:** For MATLAB, all types except varray.fi types are supported.
numpy.reshape	reshape()	Change the shape of the varray object.

Table 22: **Supported varray operations** *(cont'd)*

Python function	MATLAB methods	Notes
numpy.array_equal	isequal()	Returns logical value, 1 (True)/0 (False) testing if dimensions and elements match.
numpy.isreal	isreal()	Test if varray object has real data type.
numpy.max()	max()	Return the max value as varray object.
numpy.min()	min()	Return the min value as varray object
numpy.all	all()	Test if all elements of varray object evaluates to True.
numpy.any	any()	Test if any element of varray object evaluates to True.
numpy.savetxt	N/A	Save the varray object as a text file.

Debugging with VFS

You can debug a VFS simulation object in a standard `gdb` environment by stepping through the AIE or HLS kernel code. To do this, you need the process ID of the VFS object. You can obtain the process ID as follows:

- In MATLAB: `vfs.getpid()`

  ***Note*:** All VFS objects in a MATLAB script share the same process ID, which is separate from the MATLAB's own process ID.

- In Python:

  ```
  import os
  os.getpid()
  ```

  ***Note*:** All VFS objects run in the same process as the Python script.

Additionally, you can use `printf` for simpler debugging. For MATLAB, any printed output appears in the terminal where the application was launched.

Debugging with x86 simulation dump files

VFS simulation of AI Engine graph objects use the x86 simulator which has a feature to dump data snapshots. The snapshots allow users to inspect data traffic at kernel ports without using the debugger. This feature is enabled in VFS with `<vfs_graph_object>.setPortDump("<dir_name>")`. The argument `<dir_name>` is optional, with default value `"aieGraph"`.

***Note*:** The `<dir_name>` argument is used to separate the dump output for graph object and can only be set once.

The x86 simulator writes the dump files after simulation is finished. As VFS keeps the simulation thread running for interaction, the simulation object need to be destructed to terminate the thread. Running `<vfs_graph_object>.writePortDump()` automatically handles the destruction. The command also decodes the generated dump file to readable format by the VFS workspace.

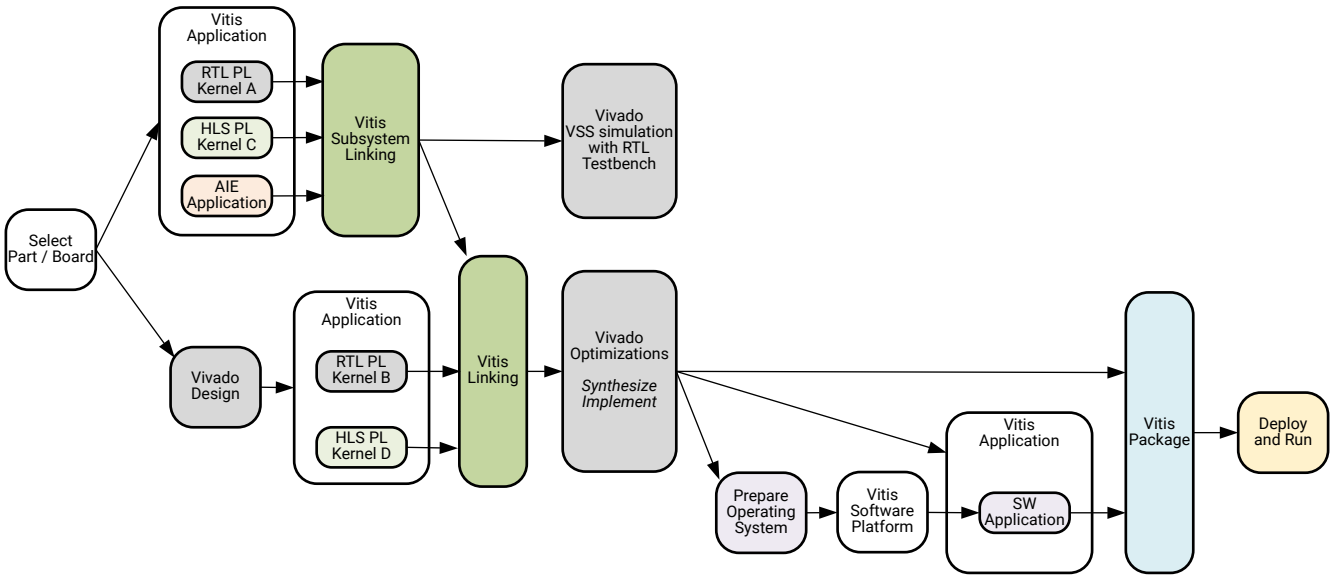
Simulation with the Vitis Subsystem

This chapter describes how to get started with simulating PL and AI Engine with VSS. By simulating the AI Engine design together with PL kernels, valuable knowledge of the interaction between the domains can be modeled and visualized.

***Note*:** Simulating with VSS is an early access feature and can be subject to change.

Simulating the VSS is an independent parallel activity to linking the VSS to an extensible platform allowing for various development strategies. This figure show the context of VSS simulation in the Vitis export to Vivado flow.

Figure 10: **VSS Simulation Context Overview**

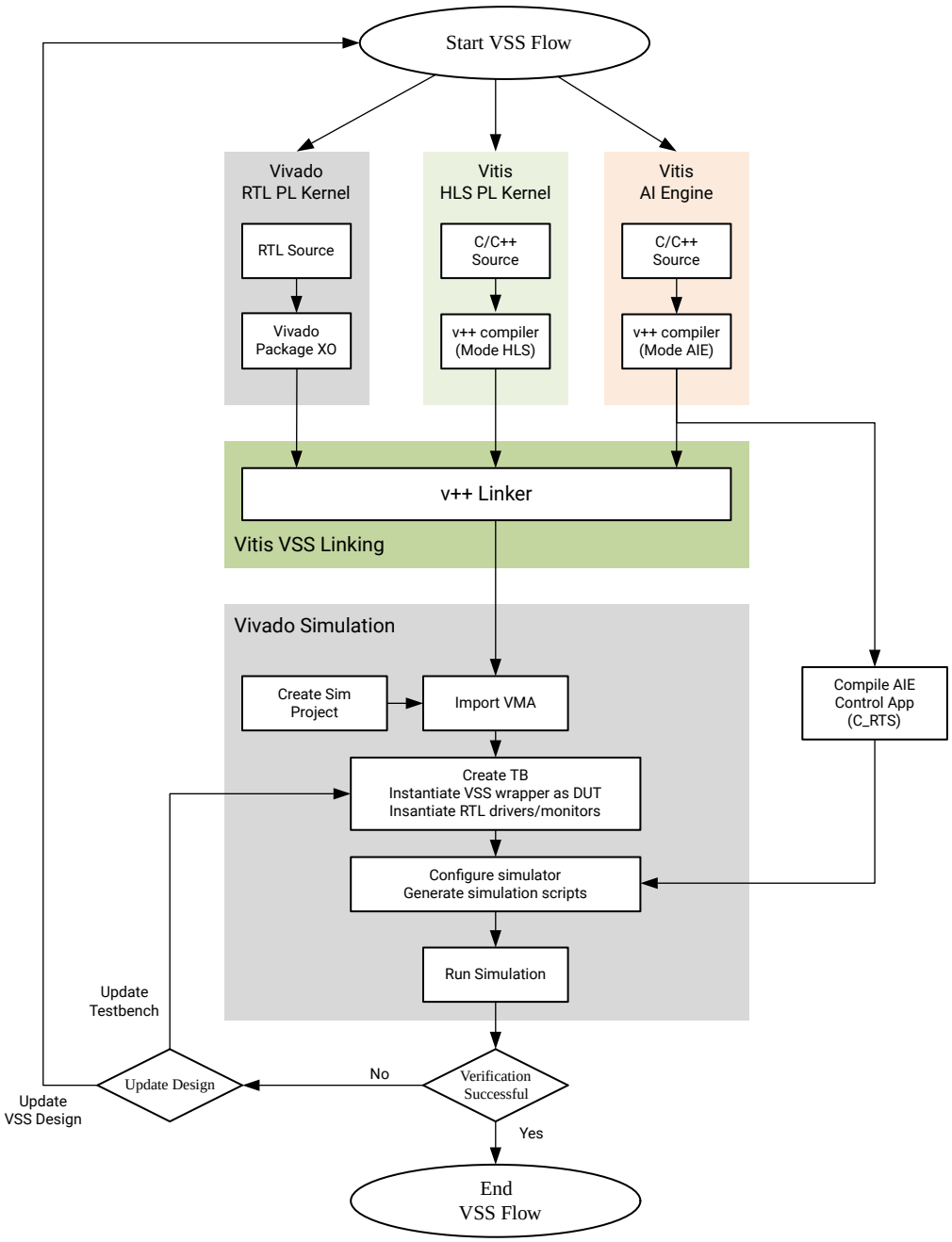


VSS simulation focuses on the simulation and verification work flow for the VSS. VSS simulation shares the steps for creating and compiling the PL kernels and AI Engine graphs, as well as linking the VSS together.

Within the AI Engine work folder, Makefiles corresponding to RTL simulators are generated. Depending on the choice of RTL simulator, one of the Makefiles are compiled to serve as control application during the simulation. This creates a simulation shared object that is attached to the RTL simulator.

The following figure represents the VSS simulation work flow using Vivado simulation project with XSim.

Figure 11: **VSS Simulation work flow**



1. Create a Vivado project.
2. Import VSS using the following code.

   ```
   vitis::import_vss <path_to_vss>/<vss_name>.vss
   ```

3. Create a wrapper for the VSS block design.

4. Add a VSS wrapper and relevant RTL test bench files (for example, RTL test bench, drivers, and monitors) to the project.
5. Set the simulation top to the test bench.
6. Setup simulation option.
7. Launch Vivado simulation.
8. Optional: open wave configurations and add signal logging.
9. Run simulation.
10. Check the results.

Example for setting up a VSS simulation is available on Vitis Tutorials on Github. See https://github.com/Xilinx/Vitis-Tutorials/tree/HEAD/Vitis_System_Design/Feature_Tutorials/02-Vitis_Subsystem_Simulation for details.

To verify incremental changes to the VSS components, the following table illustrates which steps require re-running.

Table 23: **Steps Requiring Re-Running**

Update type	Steps to update and verify	Comments
AI Engine graph without PLIO change	1. Regenerate `libadf.a` 2. Recreate VSS 3. Recompile AI Engine control app 4. Verify with simulation	
AI Engine graph with VSS interface change	1. Regenerate `libadf.a` 2. Recreate VSS 3. Rebuild simulation project a. Create new wrapper b. Update RTL test bench, drivers, and monitors. 4. Verify with simulation	Changes to interfaces require re-running wrapper to expose port updates so it can be connected in test bench.
RTL or HLS Kernel, changes within VSS	1. Regenerate `.xo` file 2. Recreate VSS 3. Verify with simulation	
RTL or HLS Kernel, changes VSS interface	1. Regenerate `.xo` file 2. Recreate VSS 3. Rebuild simulation project a. Create new wrapper b. Update RTL test bench, drivers, and monitors. 4. Verify with simulation	Changes to interfaces require re-running wrapper to expose port updates so it can be connected in test bench.
Test bench updates	1. Regenerate simulation scripts 2. Verify with simulation	Changes to files (RTL test bench, drivers and monitors) are typically detected when running interactively in Vivado.

Comparing AI Engine Graph Control with Simulation

The following table compares AI Engine graph control and graph inputs and outputs with different methods of simulating an AI Engine design.

Table 24: **Comparing AI Engine Graph Control with Simulation**

Features	AIESIM	VSS Simulation	HW_EMU
Graph control: `graph.init()` `graph.run()` `graph.wait()` `graph.end()`	`graph.cpp` - C_RTS	`graph.cpp` - C_RTS	`host.cpp` - PS (QEMU)
RTP	`graph.cpp` - C_RTS	`graph.cpp` - C_RTS	`host.cpp` - PS (QEMU)
PLIO	File I/O	RTL test bench	`host.cpp` - PS (QEMU)
GMIO	`graph.cpp` - C_RTS	Not supported	`host.cpp` - PS (QEMU)

Chapter 7

Creating and Using Vitis Platforms

A Vitis platform is comprised of a hardware and a software design that targets a specific device. A Vitis platform design comes with specialized properties indicating that it is a platform capable of connecting to Vitis components. Using Vitis tools and flows, various components can be connected to the platform, including AI Engine graphs, HLS kernels, and PS applications. This chapter discusses the various types of platforms and guides you through the process of creating and validating them. Subsequent chapters will focus on the creation and building of different types of Vitis components.

Fixed Platforms versus Extensible Platforms

There are two distinct platform types as described below.

Fixed Platform Design

A fixed platform design is a completed hardware design developed in the Vivado Design Suite.

Once your design is complete, use the `write_hw_platform -fixed` command to generate a Hardware Design (XSA) file. Within the Vitis IDE, VitisPython API or XSDB, you can then define a platform project, import the fixed `.xsa`, configure processor domains, and set up your board support package (BSP). You can use PetaLinux or Yocto to create a Linux BSP, and the Vitis IDE to create bare metal BSPs. See XSCT to Python API migration in the *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400) for details.

Combining the fixed .xsa with the software BSPs generates a Vitis platform component (.xpfm) in the Vitis IDE.

This Vitis platform component can be used in a traditional embedded software design flow to create embedded applications for Versal adaptive SoC devices, Zynq MPSoC, Zynq 7000, or MicroBlaze devices.

To develop software applications for embedded processors, you need to use the hardware drivers provided as part of the exported hardware container (.xsa) and manage them to access your hardware design from your software application.

This traditional embedded software design flow is fully documented in the *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400).

Fixed platforms cannot be used to integrate AIE graphs and PL kernels using the v++ link command. However, they do support both bare-metal and Linux-based software development. Additionally, you can modify the AI Engine software while maintaining the AI Engine-PL interface. A fixed XSA can be generated from an extensible platform-based project during the Vitis linking process.

Extensible Platform Design

An extensible platform is built with the Vivado Design Suite. These platforms begin as initial hardware designs in the Vivado Design Suite (.xsa), but unlike fixed platforms, they are designed to evolve and grow. You can integrate programmable components like AI Engine and PL kernels to the platform using either the Vitis Integrated Flow or the Vitis Export to Vivado Flow. You can optionally use VitisPython API or XSDB to create a Vitis platform component .xpfm platform that includes a software BSP for the extensible hardware platform, or defer until you have implemented fixed platform hardware.

Extensible platforms enable a parallel development process, allowing different teams to work concurrently. The application team can initiate their work on an AMD base platform like `VCK190` or `ZCU104` to develop the system's programmable components. This parallel approach enables independent development and testing on a verified ready-to-use system context, while the platform team focuses on creating and validating a custom hardware platform. AMD base platforms can be found in the Vitis installation, and their source code can be found in the Vitis Embedded Platform Source repository on GitHub.

Extensible platform designs are required to contain a processing system, memory controllers, and clock sources, and typically also include an interrupt controller. More information on building custom platforms and their interface requirements can be found at Building Custom Platforms and Adding Hardware Interfaces.

In essence, extensible platforms offer a flexible approach to hardware design, allowing you to start simple and gradually evolve your design using Vitis workflows. At any stage of development, you can implement your design using the Vitis Integrated Flow or the Vitis Export to Vivado Flow to obtain a Fixed XSA against which you can create a software platform and develop software.

Pre-built Base Platforms

AMD provides pre-built platforms for select embedded evaluation boards and recommends using the Vitis Integrated Flow, which enables loosely coupled parallel development of subsystems using the evaluation board while the platform development team brings up a custom platform. Rapid progress can be made by working in this manner. Using a pre-built platform means that the subsystem can be developed, integrated, and tested independently using a pre-verified, known-good foundation. After the subsystem is in a sufficiently advanced and stable state, the subsystem can be integrated with appropriate versions of the custom platform. Overall, this approach greatly streamlines the system integration process.

Pre-built embedded base platforms are installed with the Vitis installer. See Vitis Embedded Platforms in the *Vitis Software Platform Release Notes* (UG1742) for a list of supported platforms.

All pre-built AMD platforms include a software platform with Linux domain supporting the Xilinx Runtime (XRT). Common software images can be downloaded from the Embedded Platforms download page.

The common image packages contain the following components:

- Pre-built Linux kernel
- Pre-built root file system
- boot files (system.dtb, U-Boot.elf, boot.scr, bl31.elf, etc)
- `sdk.sh` script to generate the Sysroot

Pre-built Platform Naming Convention

Pre-built Vitis embedded platforms use the following naming convention.

```
<Vendor>_<Board>_<Feature>_<Vitis Tool Version>_<Release_Version>
```

***Note*:** Starting with VEK385 and newer boards, the <Vendor> label will no longer be used in the naming convention. Previously released boards will keep the old naming convention.

Where:

- **<Vendor>:** The board vendor. For all AMD-created pre-built platforms, use `xilinx`.
- **<Feature>:** The special function of this platform. For example:
  - `base` indicates that it connects all possible resources for you to use in an application.
  - `DFX` indicates that it supports AMD Dynamic Function eXchange (DFX).

- **<Vitis Tool Version>:** The specific version of the Vitis development platform that the platform is designed for. This also indicates the version of the AMD Vivado™ Design Suite tools that the pre-built platform is created by.
- **<Release_Version>:** The release version of the platform. The first version is `1`.

For example, the following platform names follow the naming convention:

- xilinx_vck190_base_202520_1
- xilinx_vck190_base_dfx_202520_1

***Note*:** Platform source code uses a git branch for versions. The directory name is `<Board>_<Feature>` (for example, `vck190_base`). The platform generated from the source in https://github.com/Xilinx/Vitis_Embedded_Platform_Source has the name `xilinx_vck190_base_202520_1`.

Building Custom Platforms

Platform Creation Basics

In the Vitis Integrated Flow, a hardware design is developed using Vivado Project Flows and Vitis tools. The base hardware design or extensible platform is developed in Vivado, typically consisting of a block design containing processing system (PS), network-on-chip (NoC), memory controllers, and clock sources, and other IPs and RTL modules suitable for your application. The platform also contains a minimally configured AI Engine IP instance with NoC paths needed to configure the array at boot and runtime. Potential platform attachment points for Vitis are annotated in the block design through typed PFM properties for control bus, memory, streaming input/output, and clocks.

The Vitis tools are used to compile ADF graphs and kernels into the AI Engine, compile PL kernels through HLS, and link them into the base platform by configuring the AI Engine IP, NoC, and other IPs with the platform block design. Often, the AI Engine and PL kernel network can be developed against an AMD-provided development platform, and then integrated into a custom base platform simply through like PFM attachment attributes or by straightforward changes to a connectivity specification.

AMD provided development platforms are intended to facilitate rapid development of AI Engine and PL kernel networks or subsystems using Vitis tools. As such, they provide integrated hardware and embedded Linux software platforms to enable you to develop, debug, and analyze subsystems on a board, and to do so in ways that are easy to migrate into a custom platform.

You can employ similar platform-based design methodology with custom platforms, but adopting a similar integrated hardware/software custom platform is not a requirement for subsystem development, and AMD recommends using standard platforms with custom platforms to loosen the coupling between base platform and subsystem design iterations.

Platform Components and Architecture

A platform is the starting point for Vitis tools, whether using an AMD-provided embedded platform or designing a custom platform. This section describes integrated components provided with AMD platforms.

***Note*:** Hardware platform development can be decoupled from software platform and application development when developing custom applications, as is often the case when targeting AMD SoCs.

Understanding how AMD platforms are structured helps you understand how best to align your hardware and software development practices to make best use of Vitis and Vivado hardware tools with Vitis, PetaLinux, and Yocto software development tools.

An embedded platform consists of two key components: the hardware platform and the software platform.

Hardware Platform

The hardware platform provides the basis for the design, consisting of an extensible Hardware Design (XSA) file that encapsulates a Vivado project for use in Vitis.

Vitis extends and enhances the hardware design by adding kernels and essential infrastructure modules that facilitate data flow within the system. These PL kernels and AI Engine graphs seamlessly share data with platform IPs.

Software Platform

The software platform provides the environment to run and control kernels. It serves as the control mechanism for both fixed and extensible platforms. It includes the domain setup and boot components, setup to reset and configure the hardware platform. Key elements of the software platform include:

- **Root File System (RFS):** This includes the binaries, core libraries, and configurations necessary for a functional Linux file system. The AMD-provided common rootfs comes pre-installed with XRT (Xilinx Runtime), enabling the execution of applications within the Linux environment.
- **Kernel Image:** This component contains the compiled Linux kernel. The AMD-provided common kernel includes most of AMD's peripheral device drivers, streamlining setup and configuration.
- **Sysroot:** An essential tool for cross-compilation, the Sysroot provides the required libraries for compiling applications for the target system.

Extensible Hardware Platforms

Hardware platforms are encapsulated in an extensible XSA that encompasses a Vivado project containing a block design (BD) in which Vitis tools operate. Vivado supports two types of extensible XSAs.

- Flat: Simplest design style, consisting of a single block design representing the top-level module.
- DFX (Dynamic Function eXchange): A block design container (BDC) based platform where the Vitis region BDC has been tagged with the HD.RECONFIGURABLE property (see DFX Based Hardware Platform for details).

IMPORTANT! *A BDC based extensible platform must follow the guidelines set by Vivado, see Exporting Platforms to Vitis in the Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator (UG994).*

Follow the guidelines below when building hardware platforms in Vivado:

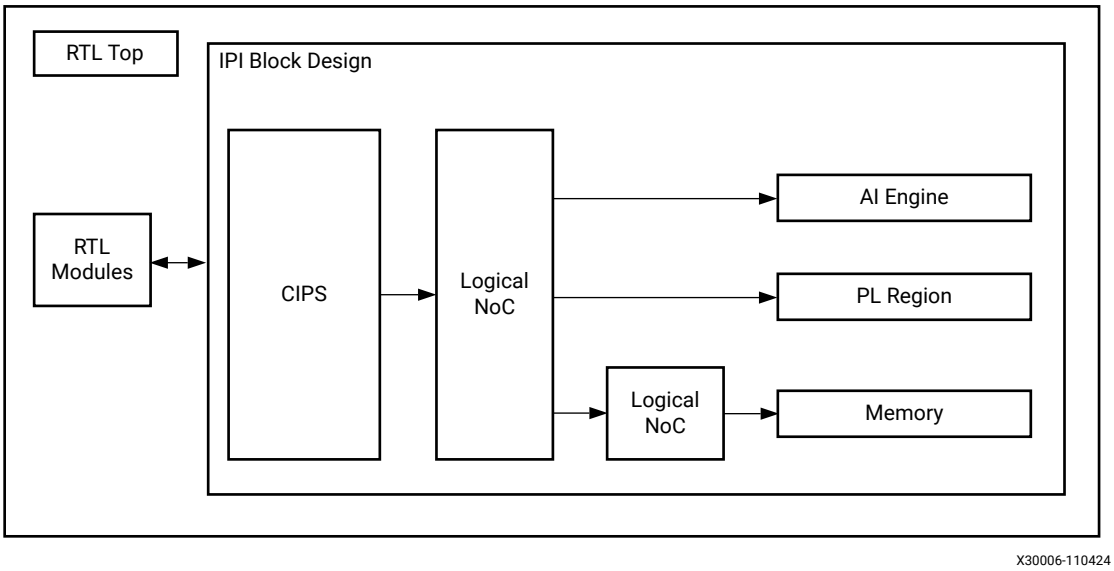
- IP Restrictions:
  - All IPs in the platform design must be local to the Vivado Design Suite project. External IP repository references are not supported for extensible XSA creation.
  - Processing subsystem requirement is the following. Include a CIPS (Control Interfaces and Processing System) IP for Versal adaptive SoC, or Processing System for AMD Zynq™ UltraScale+™ MPSoC, or Zynq 7000 SoC. Because processing system IPs have complex configurations, start from an IP instance provided as part of an AMD-verified Configurable Example Design (CED).
  - MicroBlaze restrictions are the following. Avoid using MicroBlaze processors for controlling PL kernels, except for UltraScale FPGAs.
- Platform interface requirements are the following. For more details, see Adding Hardware Interfaces.
  - Platform interface declarations have implied system semantics.
    - AXI4 slave interfaces represent memory apertures
    - AXI4/AXI4-Lite master interfaces represent control bus
    - AXI4-Stream masters and slaves represent data sources and sinks
  - Interrupt
  - Clock
  - Reset
- Clock Pin Requirements for a Platform IP:

    - Any platform IP with an AXI interface used by the Vitis linker to link to PL kernels and AI Engine graphs must also have associated clock pins. This enables v++ to accurately infer and insert clock domain crossing logic when necessary.
- Undeclared bus interfaces and signals:
    - The v++ linker supports blind connectivity for non-AXI, non-AXI4-Stream bus interfaces and signals, but such use is atypical. Refer to the Linking the System for further details.
- Project source file requirement:
    - All source files for all elements of the Vivado project must be local to the project before exporting it as an XSA. Failing to do so can result in errors when using the platform in the Vitis development flow.
- Building an `extensible` platform:
    - Ensure the project type is set as `Extensible Vitis Platform` within the Project Manager Settings or via `set_property platform.extensible true [current_project]` (Tcl command).

Flat Hardware Design Platform

The AMD Vivado Design Suite provides the capability to develop a flat hardware design in the IP integrator. This design incorporates various IP blocks, such as NoC, DDR memory controller, and AI Engine, which are specific to the chosen device and consolidated into a singular top-level block design (BD). The following figure showcases a Versal-based design that highlights the connectivity between DDR-NOC and CIPS and AI Engine-PL subsystems within the IP integrator Block design. Notably, this BD can be integrated within a broader RTL top-level design. Additionally, the design allows for the connection of custom RTL modules to specific IPs within the IP integrator BD, based on the specific requirements of the project.

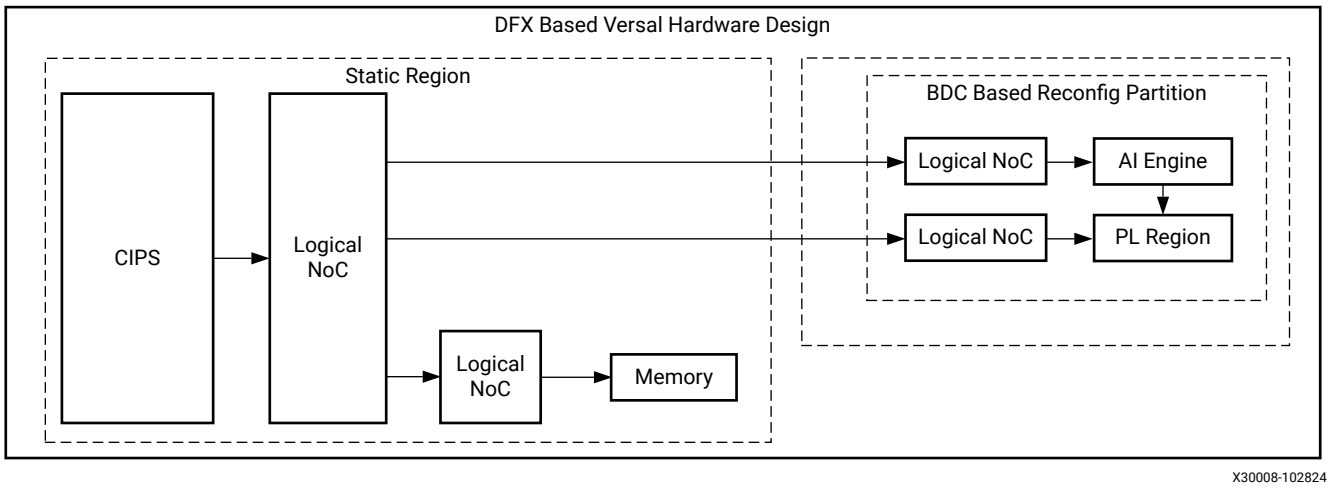
Figure 12: **Flat Hardware Design Platform**



DFX Based Hardware Platform

The hardware design that incorporates Dynamic Function eXchange (DFX) technology consists of a static region and a reconfigurable partition(RP) spanning the AI Engine and PL. The static region contains IPs such as CIPS, NoC and memory, while the reconfigurable partition region is where the dynamic PL logic or the AI Engine and its associated PL logic can be placed. For Versal devices that include the AI Engine, the AI Engine must be located in a reconfigurable partition region. Vitis and XRT only support a single reconfigurable partition and expect AIE, if used, to be part of that reconfigurable partition.

Figure 13: **DFX Based Hardware Platform**



Each Vitis linker run creates a reconfigurable module (RM) that can be loaded at runtime in the context defined by the static region. Refer to *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947) to understand the working of DFX design in more detail.

Settings for DFX-Based Platform in Vivado

This sections details the settings specific to DFX based platform in Vivado.

- **Address Aperture Setting:** To ensure that the CIPS and SmartConnect IP can access kernels in the dynamic region you need to set the address range for the dynamic region. This involves explicitly setting apertures to DDR memory interfaces in the static region block design, so that the kernel's accessible DDR range can be exported. To lock the bus definition settings, enter the following command in the Tcl console:

```
set_property HDL_ATTRIBUTE.LOCKED TRUE [get_bd_intf_pins /<RP BDC name>/
PL_CTRL_S_AXI]
set_property HDL_ATTRIBUTE.LOCKED TRUE [get_bd_intf_pins /<RP BDC name>/
DDR_0]
set_property HDL_ATTRIBUTE.LOCKED TRUE [get_bd_intf_pins /<RP BDC name>/
DDR_1]
set_property HDL_ATTRIBUTE.LOCKED TRUE [get_bd_intf_pins /<RP BDC name>/
DDR_2]
set_property HDL_ATTRIBUTE.LOCKED TRUE [get_bd_intf_pins /<RP BDC name>/
DDR_3]
```

- **Set up Block Design Container (BDC) for DFX:** In Vivado, the hardware design needs to define a Block Design Container (BDC) for the RP. The BDC establishes the dynamic region or RP. The RP consists of a BDC hierarchy managed by Vitis, and to configure it, you must update its properties, set the container boundaries, use the DFX wizard, and include a configuration.
- **Set Up Block Design Container (BDC) for DFX:**

```
# Specify that this platform supports DFX
set_property platform.uses_pr true [current_project]

# Specify the dynamic region instance path for hardware run
set_property platform.dr_inst_path {design_1_i/<RP BDC name>}
[current_project]

# Specify the dynamic region instance path for emulation
set_property platform.emu.dr_bd_inst_path {design_1_wrapper_sim_wrapper/
design_1_wrapper_i/design_1_i/<RP BDC name>} [current_project]
```

Static region and dynamic regions share the DDR memory. An AXI NoC IP is required in the Vitis Region, in the RP, to export the memory interface. The V++ linker connects the PL kernel memory interfaces to the AXI NoC IP to access memory.

- **DFX Decoupler IP:** For security reasons, DFX platforms require the DFX Decoupler IP to control kernel operations during reconfiguration. This IP plays a crucial role in ensuring system stability and preventing unexpected behavior.
- **DFX Decoupler IP Functions:**

1. Disabling communication channels during reconfiguration to avoid unintended requests from the static region that could lead to invalid AXI interface states or cause random toggles due to partial reconfiguration.
2. Preventing unexpected side effects by isolating the static and reconfigurable regions, thereby avoiding potential disruptions in the static region caused by reconfiguration.
3. Allowing Xilinx Runtime (XRT) control, which automatically activates and deactivates the DFX Decoupler IP before and after reconfiguration, respectively.

In summary, the DFX Decoupler IP serves as a safety measure during reconfiguration, ensuring security and preventing unwanted interactions between the static and reconfigurable regions of the DFX platform.

Vitis Python API DFX Platform Creation Flow

The process for creating a DFX (Dynamic Function eXchange) platform using the Vitis Python API involves three main steps, assuming the Vitis Python API client object is already initialized.

Vitis Python API Flow for DFX Platform Creation

This flow requires referencing both the static hardware design and the reconfigurable partition (RP) XSA files.

Step 1: Add Reconfigurable Partition (RP) Info Arguments

You must first define the arguments specific to the reconfigurable partitions that will be part of the DFX platform. The number of elements here must correspond exactly to the number of reconfigurable partitions defined in your static XSA.

```
rp_info_args = client.add_rp_info_args(
    rp_xsa_path="rp.xsa"
)
```

***Note*:** Replace `rp.xsa` with the actual path to your Reconfigurable Partition XSA file.

Step 2: Create the Platform Component with DFX Support

Next, use the `create_platform_component` command, passing the `rp_info_args` created in the previous step to enable DFX support.

```
platform = client.create_platform_component(
    name="platform_dfx_static",
    hw_design="$COMPONENT_LOCATION/../../../xsa/static.xsa",
    os="standalone",
    cpu="psv_cortexa72_0",
    domain_name="standalone_psv_cortexa72_0",
    generate_dtb=True,
    rp_info_args=rp_info_args,
    hw_boot_bin="BOOT.BIN"
)
```

The arguments required for create_platform_component include the following.

- **name:** The desired name for the platform component (for example, `platform_dfx_static`).
- **hw_design:** The path to the static XSA file, which contains the hardware architecture defining the static region.
- **os:** Specifies the operating system (for example, `standalone`) and the target processor (e.g., `psv_cortexa72_0`) for the platform.
- **cpu:** Specifies the target processor (for example, `psv_cortexa72_0`) for the platform.
- **rp_info_args:** The list of reconfigurable partition arguments generated in Step 1.
- **hw_boot_bin:** The path to the BOOT.BIN. See *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400) and *Bootgen User Guide* (UG1283) on how to generate the boot image.

Step 3: Retrieve and Build the Platform Component

Retrieve the newly created platform component object and trigger the build process.

```
platform = client.get_component(name="platform_dfx_static")
status = platform.build()
```

***Note*:**

You must update the paths specified for `rp_xsa_path` and `hw_design` to reflect the actual locations of your XSA files.

This script requires that the client object for the Vitis Python API is initialized prior to execution.

The overall definition of the DFX system involves specifying a fixed or static region and detailing the reconfigurable partition (RP).

The end to end flow to create the DFX based design is detailed in the Vitis tutorial https://github.com/Xilinx/Vitis-Tutorials/tree/master/Vitis_Platform_Creation/Design_Tutorials/04_Edge_VCK190_DFX.

For more details on DFX design, refer to *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909).

SSI Technology Devices and Hardware Platforms

Stacked Silicon Interconnect (SSI) technology enhances Versal Adaptive SoCs by allowing the expansion of PL resources with multiple Super Logic Regions (SLR) connected through an interposer layer. Each SLR includes a dedicated PCM (Power Control Module) block that configures its corresponding PL region. The bottom-most SLR is designated as primary, and handles device boot and initialization of other SLRs through their respective PMCs through the NoC. AI Engine capable SSI devices access and program the AI Engine through the PMC in the top SLR. Details on boot flow for SSI devices is available in Versal Devices Using SSI Technology in the *Versal Adaptive SoC System Software Developers Guide* (UG1304)

Platform Creation with SSI Devices and AI Engine

The physical placement of the AI Engine array above the top SLR necessitates enabling the corresponding NoC connection to AI Engine for configuration and access. Furthermore, it is essential for the `HSM0` clock in the top SLR to use the clock frequency equivalent to `AIE_REF_CLK_FREQMHZ` of the AI Engine IP.

```
# Example of SLR1 PMC setting for VC2502 device
  set CIPS_0 [ create_bd_cell -type ip -vlnv xilinx.com:ip:versal_cips:
CIPS_0 ]
  set_property -dict [list \
    CONFIG.PS_PMC_CONFIG { \
      PMC_CIPS_MODE {ADVANCE} \
      PMC_USE_PMC_NOC_AXI0 {1} \
      PMC_USE_PMC_NOC_AXI1 {1} \
      PMC_CRP_HSM0_REF_CTRL_ACT_FREQMHZ {33.333333} \
      PMC_CRP_HSM0_REF_CTRL_DIVISOR0 {36} \
      PMC_CRP_HSM0_REF_CTRL_FREQMHZ {33.333} \
      PMC_CRP_HSM0_REF_CTRL_SRCSEL {PPLL} \
      PMC_HSM0_CLK_ENABLE {0} \
      PMC_HSM0_CLK_OUT_ENABLE {0} \
      SLR1_PMC_CRP_HSM0_REF_CTRL_ACT_FREQMHZ {33.333333} \
      SLR1_PMC_CRP_HSM0_REF_CTRL_DIVISOR0 {36} \
      SLR1_PMC_CRP_HSM0_REF_CTRL_FREQMHZ {33.333} \
      SLR1_PMC_CRP_HSM0_REF_CTRL_SRCSEL {PPLL} \
      SLR1_PMC_HSM0_CLK_ENABLE {1} \
      SLR1_PMC_HSM0_CLK_OUT_ENABLE {0} \
    } \
    CONFIG.PS_PMC_CONFIG_APPLIED {1} \
  ] $CIPS_0
```

```
# Example of SLR3 PMC setting for VC2802 device
  set CIPS_0 [ create_bd_cell -type ip -vlnv xilinx.com:ip:versal_cips:
CIPS_0 ]
  set_property -dict [list \
    CONFIG.PS_PMC_CONFIG { \
      PMC_CIPS_MODE {ADVANCE} \
      PMC_USE_PMC_NOC_AXI0 {1} \
      PMC_USE_PMC_NOC_AXI1 {1} \
      PMC_USE_PMC_NOC_AXI2 {1} \
      PMC_USE_PMC_NOC_AXI3 {1} \
      PMC_CRP_HSM0_REF_CTRL_ACT_FREQMHZ {33.333333} \
      PMC_CRP_HSM0_REF_CTRL_DIVISOR0 {36} \
      PMC_CRP_HSM0_REF_CTRL_FREQMHZ {33.333} \
      PMC_CRP_HSM0_REF_CTRL_SRCSEL {PPLL} \
```

```
      PMC_HSM0_CLK_ENABLE {0} \
      PMC_HSM0_CLK_OUT_ENABLE {0} \
      SLR3_PMC_CRP_HSM0_REF_CTRL_ACT_FREQMHZ {33.333333} \
      SLR3_PMC_CRP_HSM0_REF_CTRL_DIVISOR0 {36} \
      SLR3_PMC_CRP_HSM0_REF_CTRL_FREQMHZ {33.333} \
      SLR3_PMC_CRP_HSM0_REF_CTRL_SRCSEL {PPLL} \
      SLR3_PMC_HSM0_CLK_ENABLE {1} \
      SLR3_PMC_HSM0_CLK_OUT_ENABLE {0} \
    } \
    CONFIG.PS_PMC_CONFIG_APPLIED {1} \
  ] $CIPS_0
```

Note: These examples for VC2502 and VC2802 CIPS configuration showcase the process of enabling correct clock reference and programming of the AI Enginethrough the NoC. Additional CIPS setup might be required based on the specific requirements of your board.

For details on CIPS and PMC Power Domain Clocks, see PMC Power Domain Clock in the *Control, Interface and Processing System LogiCORE IP Product Guide* (PG352).

Associating the Platform with Vitis Kernels

A new PFM attribute that can be applied to `BD` cells in IP integrator during platform creation. This attribute, named `PFM.REGION` functions as a string identifier that associates different `BD` components intended for joint usage.

```
set_property PFM.REGION {"MY_SLR_TAG"} [get_bd_cells /my_bd_instance_0]
```

In Vivado sources, the platform design can include a `PBLOCK` constraint to impose physical constraint on cells. One intended use case of the `REGION` attribute is to label a control `NoC` or other IP instance to reside in a specific SLR (`PBLOCK`), so that a v++ user can specify linker affinity to the instance. Connections from PL kernels to constrained cells provide impetus to the Vivado placer to place kernels in the same `PBLOCK`. To add Vitis linking specifications to resources, see Specifying SLR Region for SSI Devices.

Note: While `PFM.REGION` can help to keep resources in the same SLR, additional constraints can be necessary to fix them to a specific region. For non SSI devices it can be used to establish associations between control, memory resources and reset resources to be used together.

Platform Support for Segmented Configuration Flow

A Segmented Configuration build flow in Vivado generates the configuration bitstream as two images:

- A `boot.pdi` image that boots the processing system, DDR/LPDDR memory controllers, and NoC paths between them.
- A `pl.pdi` image that can be loaded after the `boot.pdi`. The `pl.pdi` represents the design content not contained within the `boot.pdi` image, which can be loaded after the `boot.pdi`.

The `pl.pdi` image can be reloaded multiple times while the processing system and DDR remain active. The Vitis Integrated flow, VMA Export, and VSS generation flows all support Segmented Configuration.

Details on the Segmented Configuration feature and steps to enable it in Vivado designs can be found in the Segmented Configuration Design Tutorial.

Deferred Load Flow

The deferred load flow defers PL/AIE configuration (`pl.pdi/pl_aie.pdi`) until after a fast OS boot using `boot.pdi`. This prioritizes essential components (CIPS/PSXC, NoC, DDRMCs) for quicker startup. This separation facilitates a fast OS boot, while deferring the full PL (or PL and AIE) configuration until it is needed.

Dynamic Load / Reload Flow

The dynamic load / reload flow enables dynamic, on-the-fly PL reconfiguration (using `pl.pdi`) by separating the Processing System (PS) and Operating System (OS) boot process (using `boot.pdi`) from the rest of the device configuration. This offers similar capabilities to DFX but with simplified implementation. The `pl.pdi` acts as a reconfigurable module, encompassing all design elements except CIPS, NoC, and DDRMCs.

A key aspect of this approach is managing activity within the processing domain during a reload. While automatic isolation is enabled and disabled at the PS-PL boundary, you are responsible for managing activity within the processing domain during the transition, including pausing activity and unloading and reloading drivers as needed. Furthermore, the segmented configuration supports multiple `pl.pdi`, enabling the creation of several PL configurations that are compatible at both boot and runtime with a single `boot.pdi`. This allows for flexible and dynamic reconfiguration of the PL without requiring a full system reboot or changes to the boot image.

Benefits and Implementation

The primary benefits of Segmented Configuration include a smaller boot image required to bring up the processor and OS. Another benefit is the ability to enable runtime PL reconfigurability without the complexity of a full DFX build flow.

The PL PDI can be loaded on primary or secondary boot paths without PL dependency, using PCIe endpoints via CPM or from remote storage or boot devices.

Platform and Deferred Load of PL (and AIE) Image

The process of generating and using an extensible XSA tailored for deferred flow in the Vivado and Vitis environments involves the following steps.

Vivado:

In Vivado, the flow begins by configuring the project for segmented configuration using the command:

```
set_property segmented_configuration [current_project]
```

IMPORTANT! *This property is always enabled for all Versal AI Edge Series Gen2 device based designs. This property is optional and turned off by default for first-generation Versal adaptive SoCs.*

This property is followed by setting the PFM properties as per design requirements and executing `generate_target all` to prepare the design. At this stage, the design remains in its pre-synthesis phase, meaning it has not yet been implemented within Vivado. The `write_hw_platform` command then generates the `extensible.xsa` file using `write_hw_platform -hw extensible.xsa`, which serves as the hardware platform definition.

The Tcl script snippet of the steps is shown below:

```
generate_target all [get_files vck190sc.bd]

set_property platform.extensible true [current_project]
set_property platform.platform_state "pre_synth" [current_project]
set_property platform.design_intent.embedded "true" [current_project]
set_property platform.design_intent.server_managed "false" [current_project]
set_property platform.design_intent.external_host "false" [current_project]
set_property platform.design_intent.datacenter "false" [current_project]
set_property platform.run.steps.place_design.tcl.pre [get_files
prohibit_select_bli_bels_for_hold.tcl]
update_compile_order -fileset sources_1
save_bd_design

# deferred load
write_hw_platform -force ./vck190sc-deferred.xsa
```

Vitis:

This `extensible.xsa` can be used in both the Vitis Integrated Flow and Vitis Export to Vivado Flow. In the Vitis integrated flow, v++ --link creates a fixed XSA containing the `boot.pdi` and `pl.pdi` files which can be packaged with the AIE content using the v++ package flow.

In the Vitis Export to Vivado flow, the VMA is exported to Vivado, and the Vivado project can then be implemented according to specific requirements of the project to generate the final `boot.pdi` and `pl.pdi` files.

This deferred approach allows for a separation of concerns, enabling hardware platform definition before full implementation.

Platform and Dynamic Load/Reload of PL (and AIE) Image

Vivado

In Vivado, the flow begins by configuring the project for segmented configuration using the command:

```
set_property segmented_configuration [current_project]
```

This is followed by setting the PFM properties as per the design requirements and implementing the design. Use the `write_hw_platform` command to generate the `extensible.xsa`. When you run this command, it automatically checks if segmented configuration property is enabled. If enabled, the generated extensible XSA file will contain the routed design checkpoint (DCP) and the NoC configuration, including the necessary boot paths. This ensures everything required for dynamic reload of the PL(and AIE) is included in the XSA.

The Tcl script snippet of the steps is shown below.

```
generate_target all [get_files vck190sc.bd]

set_property platform.extensible true [current_project]
set_property platform.platform_state "pre_synth" [current_project]
set_property platform.design_intent.embedded "true" [current_project]
set_property platform.design_intent.server_managed "false" [current_project]
set_property platform.design_intent.external_host "false" [current_project]
set_property platform.design_intent.datacenter "false" [current_project]
set_property platform.run.steps.place_design.tcl.pre [get_files
prohibit_select_bli_bels_for_hold.tcl]
update_compile_order -fileset sources_1
save_bd_design

# deferred load
write_hw_platform -force ./vck190sc-deferred.xsa

if { [lindex $argv 0] == "dynamic_reload" } {
  update_compile_order -fileset sources_1
  launch_runs impl_1 -to_step write_device_image -jobs 5
  wait_on_run impl_1
  if {{get_property PROGRESS [get_runs impl_1]} != "100%"} {
    error "ERROR: Implementation failed"
  }
  open_run impl_1
}

#dynamic reload
write_hw_platform -hw -force vck190sc.xsa
```

Vitis

This `extensible.xsa` can be used in both the Vitis integrated flow and Vitis export to Vivado flow. In the Vitis integrated flow, v++ --link creates a fixed XSA containing the `boot.pdi` and `pl.pdi` files which can be packaged with the AIE content using the v++ package flow.

In the Vitis export to Vivado flow, the VMA is exported to Vivado, and the Vivado project must then be implemented according to specific requirements of the project to generate the final `boot.pdi` and `pl.pdi` files. Adherence to Vivado's segmented configuration requirements for dynamic reload is crucial to ensure proper functionality. This approach allows for dynamic updates to the PL without requiring a full system reboot.

Packaging the AIE

A fixed XSA generated by the v++ linker or by Vivado in the VMA Export flow contains a `pl.pdi` but not an AI Engine.pdi containing core ELF and configuration data objects (.cdo). The v++ package tool extracts the AI Engine image data from compiled `libadf.a`, archives and merges it with the PL content in a fixed XSA to create image files to load onto the target device. The package command generates several output files, including `pl_aie.pdi`, `pl_aie.dtbo`, and `pl_aie.xclbin`. The command used to package these files is:

```
v++ -p -s -f <fixed>.xsa libadf.a --package.out_dir <package
directory> -o <xclbin_name>.xclbin
```

The package command generates a merged `pl_aie.pdi` and `pl_aie.xclbin` file that can be loaded by the `xrt::loadXclbin()` API. The Linux OS requires a `pl_aie.dtbo` device tree overlay to configure PL drivers, which can be loaded with the `pl_aie.pdi` using the `fpgautil` utility. Because the `pl_aie.pdi` image is loaded by `fpgautil`, the `xrt::loadXclbin()` invocation by the software application loads only design metadata used by XRT.

Adding Hardware Interfaces

Specific hardware interfaces must be declared as `bd_cell` properties in the Vivado project before exporting the platform. These interfaces include AXI ports, AXI4-Stream ports, clocking, and interrupts. They are used by the Vitis linker to connect Vitis components (such as AI Engine graphs and PL kernels) to various blocks and resources in the extensible platform. The following sections detail the various types of hardware interfaces and provide guidelines on how to add platform properties to the hardware design in Vivado that correspond to these interfaces.

The following table shows the possible interfaces that the Vitis linker can use and the minimal interface requirements for an embedded platform.

Table 25: **Interfaces**

Inputs	Types Vitis Can Use	Minimum Requirements for AXI4 Kernels
Control Interfaces	AXI Master Interfaces from PS or from AXI Interconnect IP or SmartConnect IP	One AXI4-Lite Master for kernel control
Memory Interfaces	AXI Slave Interfaces	One memory interface for data exchange
Streaming Interfaces	AXI4-Stream Interfaces	Not required
Clock	Multiple clock signals	One clock

Table 25: **Interfaces** *(cont'd)*

Inputs	Types Vitis Can Use	Minimum Requirements for AXI4 Kernels
Interrupt	Multiple interrupt signals	One Interrupt

General Requirements

IMPORTANT! *The source files for all elements of the Vivado project should be local to the project prior to exporting it as an XSA, or an error can be returned when using the platform in the Vitis tool.*

- Every IP used in the platform design that is not part of the standard Vivado IP catalog should be local to the Vivado Design Suite project. References to IP repository paths external to the project are not supported when creating extensible XSA.
- Any platform interface, used for linking to kernels by the Vitis compiler, should be an AXI4, AXI4-Lite, AXI4-Stream, interrupt, clock, or reset type of interface.

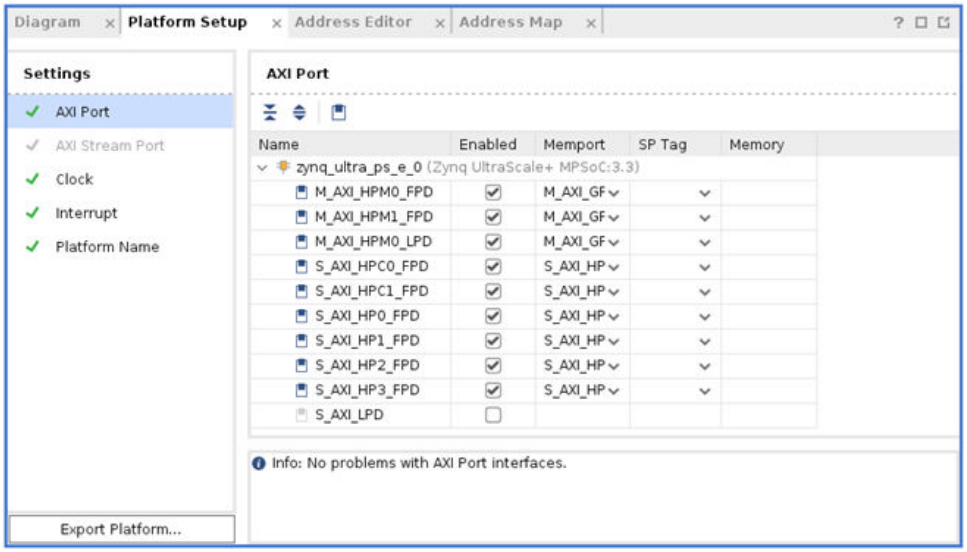
  Note: Resets are optional, but must be associated with a clock when used.
- Any platform IP that has an AXI interface for linking to kernels by the Vitis compiler should also have associated clock pins to enable `v++` to correctly infer and insert clock domain crossing logic when needed.

Adding Platform Interfaces

If a component in block design has a PFM property, this component can be recognized by the `v++` linker. The component can be linked to Vitis components.

In the Vivado IDE, the platform interface (PFM) properties can be set in the Platform Setup tab if the project is created as an extensible platform project. Click **Window menu → Platform Setup** to open the settings.

Figure 14: **Platform Setup**



TIP: *Platform interfaces can be defined manually in the Tcl Console, or by a Tcl script as well.*

The four Platform Interface Tcl APIs include:

- AXI memory-mapped interfaces:

```
set_property PFM.AXI_PORT { <port_name> {parameters} <port2>
{parameters} ...} [get_bd_cells <cell_name>]
```

- AXI4-Stream interfaces:

```
set_property PFM.AXIS_PORT { <port_name> {parameters} <port2>
{parameters} ...} [get_bd_cells <cell_name>]
```

- Clocks and resets:

```
set_property PFM.CLOCK { <port_name> {parameters} <port2>
{parameters} ...} [get_bd_cells <cell_name>]
```

- Interrupts:

```
set_property PFM.IRQ {pin_name {id id_number range irq_count}}
[get_bd_cells <cell_name>]
```

The requirements for the PFM properties are:

- The value of the PFM interface properties must be specified as a Tcl dictionary, a list of name/"value" pairs.

  IMPORTANT! *The "value" must be quoted, and both the name and value are case-sensitive.*

- A `bd_cell` can have multiple PFM interface definitions. However, for each type of PFM interface, all ports are required to be set in a single `set_property` Tcl command.

- For each PFM interface property, the name specified for the port object must match the name of an external port or interface on a `bd_cell`. Each external port or interface object can only have one PFM interface definition.
- Each different type of PFM interface can have different parameters.
- Setting the PFM property with a NULL ("") string deletes previously defined PFM interfaces.

Adding AXI Interfaces

To support AXI interfaces the platform needs to declare at least one AXI control interface with AXI memory-mapped master port (M_AXI) and one memory interface with AXI Slave port (S_AXI). They can be exported from the PS block directly or have an interconnect IP connected. If the platform does not support AXI4 kernels, these interfaces are not required.

The following is the Tcl command syntax:

```
set_property PFM.AXI_PORT { <port_name> {parameters} <port2>
{parameters} ...} [get_bd_cells <cell_name>]
```

The AXI control interfaces and AXI memory interfaces share the same `PFM.AXI` property. They have different `memport` types.

- **Memport:** AXI control interface can be defined as `M_AXI_GP`. Memory interfaces use other types:`S_AXI_HP`, `S_AXI_ACP`,`S_AXI_HPC`, `S_AXI_NOC` or `MIG`. The tag indicate how the port is intended to be used.
- **SP Tag ID:** (Optional) A user-defined ID that should start with an alphabetic character. The ID is case-sensitive. The system port tag (`sptag`) is a symbolic identifier that represents a connection point to a platform resource or a class of platform resources, such as `S_AXI_HP` and `S_AXI_ACP`. Multiple block design platform ports can share the same `sptag`. The Vitis linker assigns the port to requested resource according to the linking configuration. For more information on how `sptags` are used, see Mapping Kernel Ports to Memory.

  IMPORTANT! *Ports that share same SP Tag IP must be equivalent resources otherwise the design will fail.*

- **Memory:** (Optional) Specify the associated MIG IP instance and `address_segment`. The memory tag is a unique identifier that combines the Cell Name and Base Name columns in the IP integrator Address Editor. This tag is associated with connections to the Memory Subsystem HIP, where multiple block design platform ports can share the same memory tag.

This section outlines the requirements for exporting AXI interconnect master and slave ports on Versal, Zynq, and Zynq UltraScale+ MPSoC devices.

- **General Requirements:**
  - All ports used within the platform must have a lower index than any declared platform interfaces.
  - There should be no gaps in the port index numbering.

  - The maximum number of master ports declared on an interconnect connected to an `M_AXI_GP` port is 64. Therefore, available master ports to declare must be one of `{M00_AXI, M01_AXI, ..., M63_AXI}`. Avoid declaring ports used within the platform itself.

- **AXI Ports:**
  - The maximum number of master IDs for the `S_AXI_ACP` port is 8. Therefore, available master ports to declare must be one of `{S00_AXI, S01_AXI, ..., S07_AXI}`.
  - The maximum number of master IDs for an `S_AXI_HP`, `S_AXI_HPC`, or `MIG` port is 16. Therefore, available master ports to declare must be one of `{S00_AXI, S01_AXI, ..., S15_AXI}`.
  - ***Note:*** `S_AXI_ACP`, `S_AXI_HPC`, `S_AXI_HP`, and `MIG` ports are used for Zynq and ZynqMP devices
  - The maximum number of master IDs for an `S_AXI_NOC` port is 28. Therefore, available master ports to declare must be one of `{S00_AXI, S01_AXI, ..., S27_AXI}`.
  - `S_AXI_NOC` ports are only used for Versal devices. The maximum number is device dependent. Setting a lower number is used to deliberately constrain how many resources the design is allowed to use. Setting a higher number cause Vitis to share NMUs if resources are exhausted.

Following these requirements ensures that your exported AXI interconnect master and slave ports are compatible with the Vitis platform and avoids the use of cascaded interconnects.

The following Tcl script defines AXI master ports on AXI Interconnect IP:

```
set parVal []
for {set i 2} {$i < 64} {incr i} {
lappend parVal M[format %02d $i]_AXI \
{memport "M_AXI_GP"}
}
set_property PFM.AXI_PORT $parVal [get_bd_cells /axi_interconnect_0]
```

The following Tcl script defines AXI memory ports with MIG on SmartConnect IP:

```
set parVal []
for {set i 1} {$i < 16} {incr i} {
lappend parVal S[format %02d $i]_AXI
{memport "MIG" sptag "Bank0"}
}
set_property PFM.AXI_PORT $parVal [get_bd_cells /smartconnect_0]
```

The following is an example of the `PFM.AXI_PORT` setting for control interface and memory interface for ZynqMP device.

```
set_property PFM.AXI_PORT {
M_AXI_HPM1_FPD {memport "M_AXI_GP"}
S_AXI_HPC0_FPD {memport "S_AXI_HPC" sptag "HPC0" memory "zynq_ultra_ps_e_0
HPC0_DDR_LOW"}
S_AXI_HPC1_FPD {memport "S_AXI_HPC" sptag "HPC1" memory "zynq_ultra_ps_e_0
HPC1_DDR_LOW"}
S_AXI_HP0_FPD {memport "S_AXI_HP" sptag "HP0" memory "zynq_ultra_ps_e_0
```

```
HP0_DDR_LOW"}
S_AXI_HP1_FPD {memport "S_AXI_HP" sptag "HP1" memory "zynq_ultra_ps_e_0
HP1_DDR_LOW"}
S_AXI_HP2_FPD {memport "S_AXI_HP" sptag "HP2" memory "zynq_ultra_ps_e_0
HP2_DDR_LOW"}
} [get_bd_cells /ps_e]
```

The following is an example of the `PFM.AXI_PORT` setting for control interface and memory interface for Versal device.

```
set_property PFM.AXI_PORT {S00_AXI {memport "S_AXI_NOC" sptag "LPDDR"}
S01_AXI {memport "S_AXI_NOC" sptag "LPDDR"} } [get_bd_cells /aggr_noc]

set_property PFM.AXI_PORT {M02_AXI {memport "M_AXI_GP" sptag "" memory ""}
M03_AXI {memport "M_AXI_GP" sptag "" memory ""} M04_AXI {memport "M_AXI_GP"
sptag "" memory ""}} [get_bd_cells /icn_ctrl_1]
```

TIP: *In the examples above, `zynq_ultra_ps_e_0` is the instance name of the Zynq UltraScale+ MPSoC module, and `HPC0_DDR_LOW` is the address range name.*

Adding AXI Stream Interfaces

A platform can support AXI4-Stream connections to the Vitis subsystem. A `PFM.AXIS_PORT` attribute declares a platform stream attachment point for the v++ linker to connect PL kernels and AI Engine PLIOs, through the `--connectivity.stream_connect` (or `--connectivity.sc`) directive. Because AXI4-Streams are point-to-point, each such `sptag` must be unique.

The following is the Tcl command syntax:

```
set_property PFM.AXIS_PORT { <port_name> {type <type_value> sptag
<sp_tag_value> is_range <true/false>} [get_bd_cells <cell_name>]
```

Argument Description

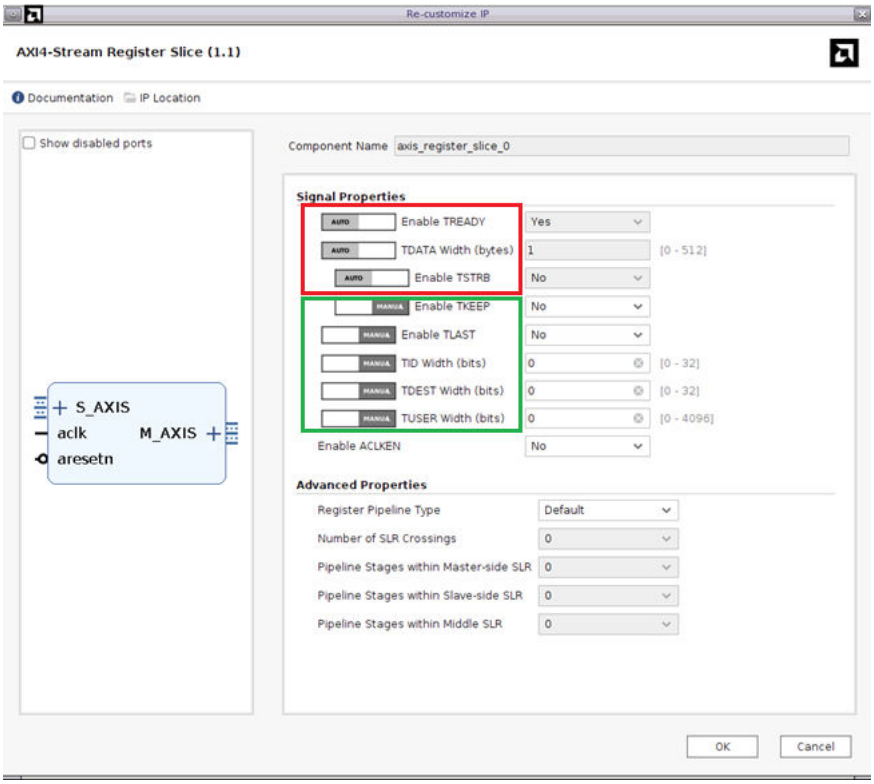
- **Port_name:** AXI4-Stream port name.
- **Type:** Type value: Streaming interface port type. Valid values for type include:
  - M_AXIS: A AXI master port
  - S_AXIS: A AXI slave port
- **SP Tag ID:** A user-defined unique alphanumeric string that starts with an alphabetic character. The ID is case-sensitive. The system port tag (`sptag`) is a symbolic identifier that represents a connection point to a platform resource.
- **is_range:** Metadata used by Vivado Integrated Design Environment (IDE).

Example

```
set_property PFM.AXIS_PORT {AXIS_P0 {type "S_AXIS" sp_tag "MY_AXIS_PORT0"
is_range "false"}} [get_bd_cells /zynq_ultra_ps_e_0]
```

IMPORTANT! *A `PFM.AXIS_PORT` must have a fixed data width. As an example, the following figure shows how to set port `CONFIG` options for an AXI4-Stream Register Slice IP.*

Figure 15: **AXI4-Stream Register Slice**



***Note*:** For more information on linking AXI4-Stream interfaces between kernels and platforms, see Specifying Streaming Connections.

Adding Clocks and Resets

Platforms have a variety of clock associations, for example, processor, PL, and AI Engine associated clocks. The following table explains the specific IPs in which these clocks can be configured.

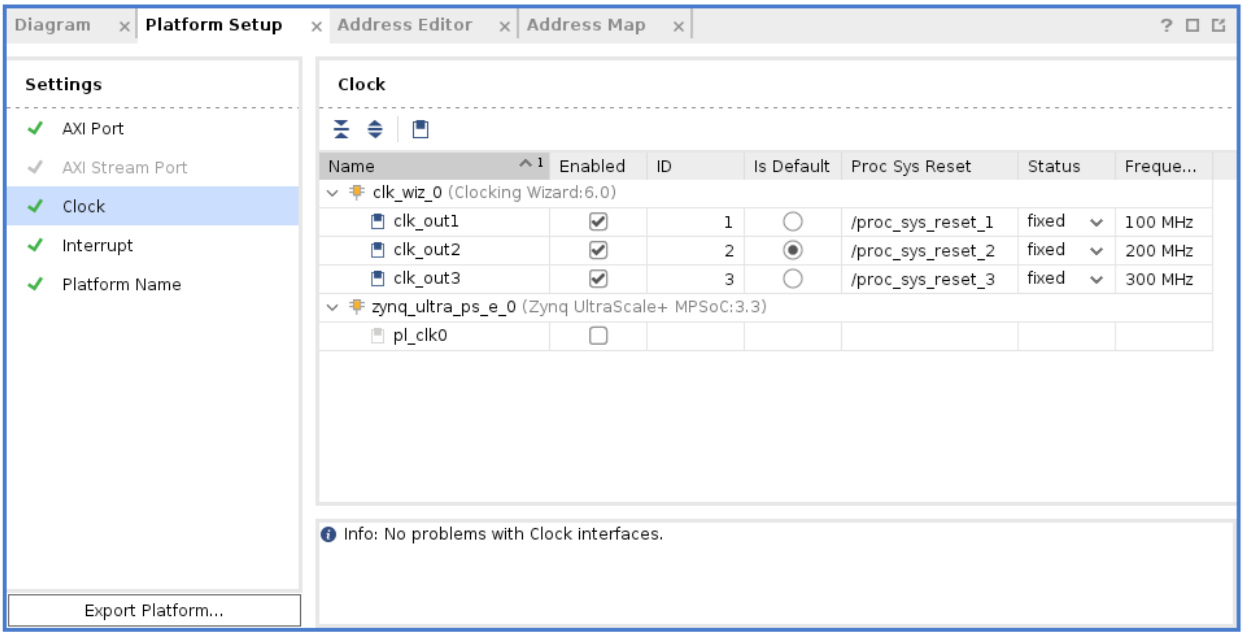
Table 26: **Hardware Design Clocks**

Clock	Description
AI Engine	Can be configured in the platform in the AI Engine IP.
Processor	Can be configured in the platform in the CIPS IP.
Programmable Logic (PL)	Can have multiple clocks and can be configured in the platform.
NoC	Device dependent and can be configured in the platform in the CIPS and NoC IP.

Note: These clocks are derived from the platform and are affected by the device, speed grade and operating voltage.

For information on AMD Versal™ device clocks, see *Versal AI Core Series Data Sheet: DC and AC Switching Characteristics* (DS957), *Versal AI Edge Series Data Sheet: DC and AC Switching Characteristics* (DS958), *Versal Prime Series Data Sheet: DC and AC Switching Characteristics* (DS956) and *Versal Premium Series Data Sheet: DC and AC Switching Characteristics* (DS959).

Figure 16: **Platform Setup – Clock**



You can export any clock source with the platform, but for each clock you must also export synchronized reset signals using a Processor System Reset IP block in the platform. For details of defining clocks and resets, see the Vitis-Tutorials/Vitis_Platform_Creation. The PFM.CLOCK property can be set on a BD cell, external port, or external interface.

In the preceding figure you can see the details of the platform clocks. There must be at least one clock enabled for the platform and one clock chosen as the default.

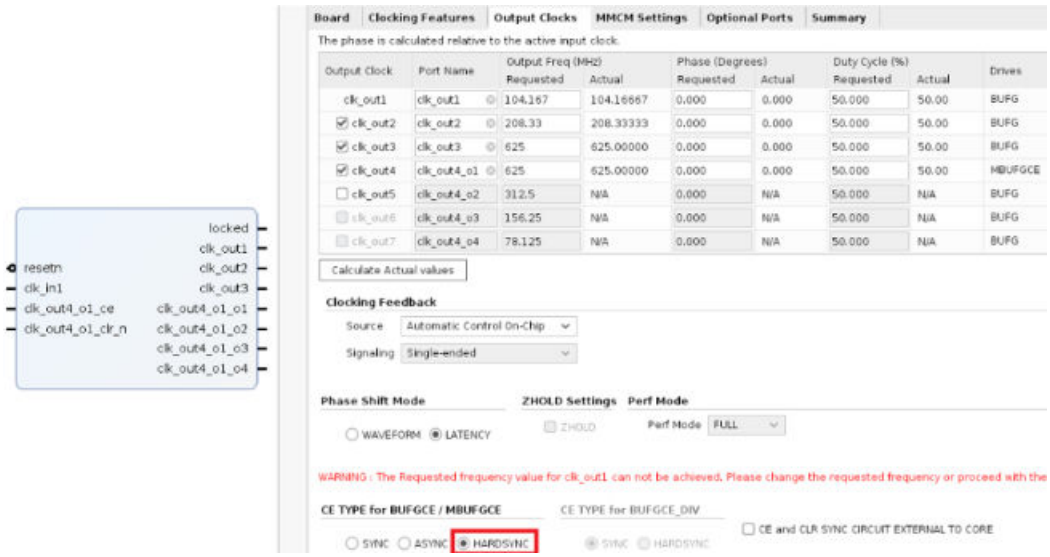
The following is the Tcl command for setting the PFM.CLOCK property:

```
set_property PFM.CLOCK { <port_name> {parameters} \
<port2> {parameters} ...} [get_bd_cells <cell_name>]
```

Clock Wizard Configuration in Vivado

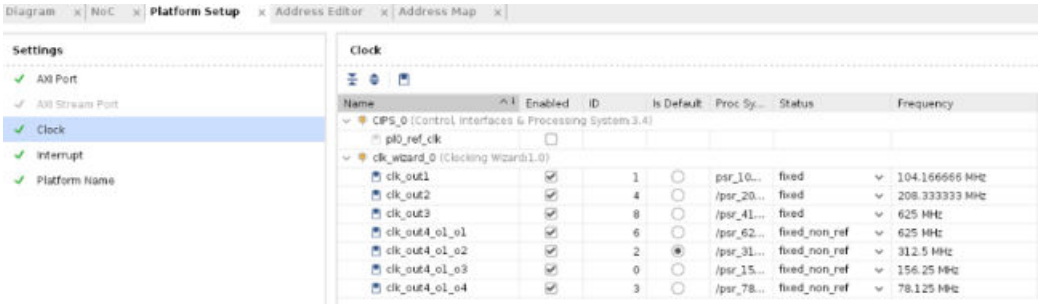
When setting the clock configuration in the Vivado design using the clock wizard IP, the following rules are a recommended practice.

Figure 17: **Clock Wizard - Output Clocks Configuration**



- Using MBUFGCE:
  - Use `MBUFGCE` along with `HARDSYNC` option to lower clock skew.
  - When generating specific clock frequencies using MBUFGCE (for example, 1/1, 1/2, 1/4, and 1/8), it is recommended to use them with the HARDSYNC option. However, MBUFGCE clocks cannot be used as reference clocks for further clock generation.
  - In the Versal VCK190 base platform, the clocks with frequencies 625, 312.5, 156.25, and 78.125 MHz generated by MBUFGCE should not be used as reference clocks.
  - For example, if you need a 625 MHz reference clock, use the `clk_out3` clock generated by BUFG instead of the `clk_out4_o1_o1` clock generated by MBUFGCE. The former originates directly from the MMCM, while the latter uses MBUFGCE for enhanced skew control.

Figure 18: **Platform Setup:Clock Status Attribute**



- PFM.CLK Status:
  - Clocks designated as `fixed_non_ref`clocks can be used directly by IPs, however, cannot be used as a reference clock.

- Clocks designated as `fixed` clocks can be used as reference clocks or directly by IPs.
- The Tcl script to set the PFM clock attributes is as follows:
  ```
  set_property PFM.CLOCK ..... clk_out7 {id "6" is_default"false"
  proc_sys_reset"/psr_625mhz" status "fixed_non_ref"freq_hz
  "625000000"}} [get_bd_cells/clk_wizard_0]
  ```

Adding Interrupts

The Vitis linker automatically connects the kernel `interrupt` signal to an IRQ in the platform during the `v++` linking stage. However, this requires the interrupts on the platform to be defined using the `PFM.IRQ` property as shown below:

```
set_property PFM.IRQ {pin_name {id id_number}} bd_cell
set_property PFM.IRQ {port_name {id id_number range irq_count}}
[get_bd_cell <cell_name>]
```

Argument Description

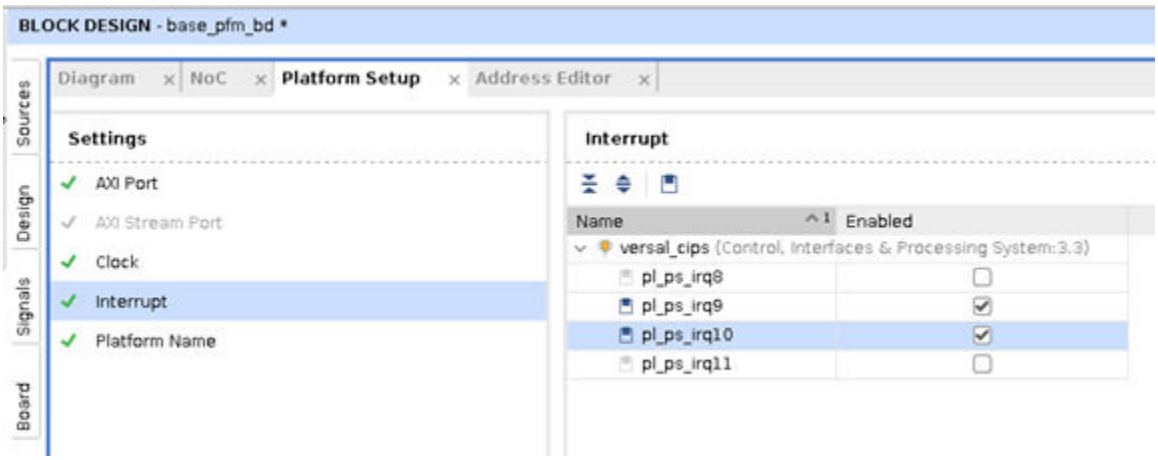
- **Port_name:** IRQ port name of `bd_cell`.
- **id_number:** Integer from 0 to 127 to specify the IRQ number or the starting number if range is specified.
- **irq_count:** Used for labeling interfaces that are otherwise subject to parameter propagation for specifying sizing of a bus (for example, interrupt controller intr interface).

The example shows how to enable 32 IRQ inputs to `axi_intc_0` intr port.

```
set_property PFM.IRQ {intr {id 0 range 32}} [get_bd_cells /axi_intc_0]
```

The interrupts for AMD Versal™ CIPS IP core can be found on the IP customization dialog box by selecting the PS PMC configuration command and selecting **Interrupts**. Here you can enable the interrupts on the CIPS IP for use on the platform. Then you must enable the interrupts on the platform for connection by `v++` during linking. On the Platform Setup tab of the Block Design, select **Interrupt** and then enable the visible interrupts on the CIPS IP as shown below. Refer to Creating a Hardware Platform for the Platform-Based Design Flow of the *Versal Adaptive SoC Hardware, IP, and Platform Development Methodology Guide* (UG1387) for more information.

Figure 19: **Platform Setup – Interrupt**



This results in the following Tcl command for enabling the interrupts:

```
set_property PFM.IRQ {pl_ps_irq9 {is_range "false"} pl_ps_irq10 {is_range
"false"}} [get_bd_cells /versal_cips]
```

TIP: *Multiple interrupts can also be connected to the CIPS IP through the use of the Concat block.*

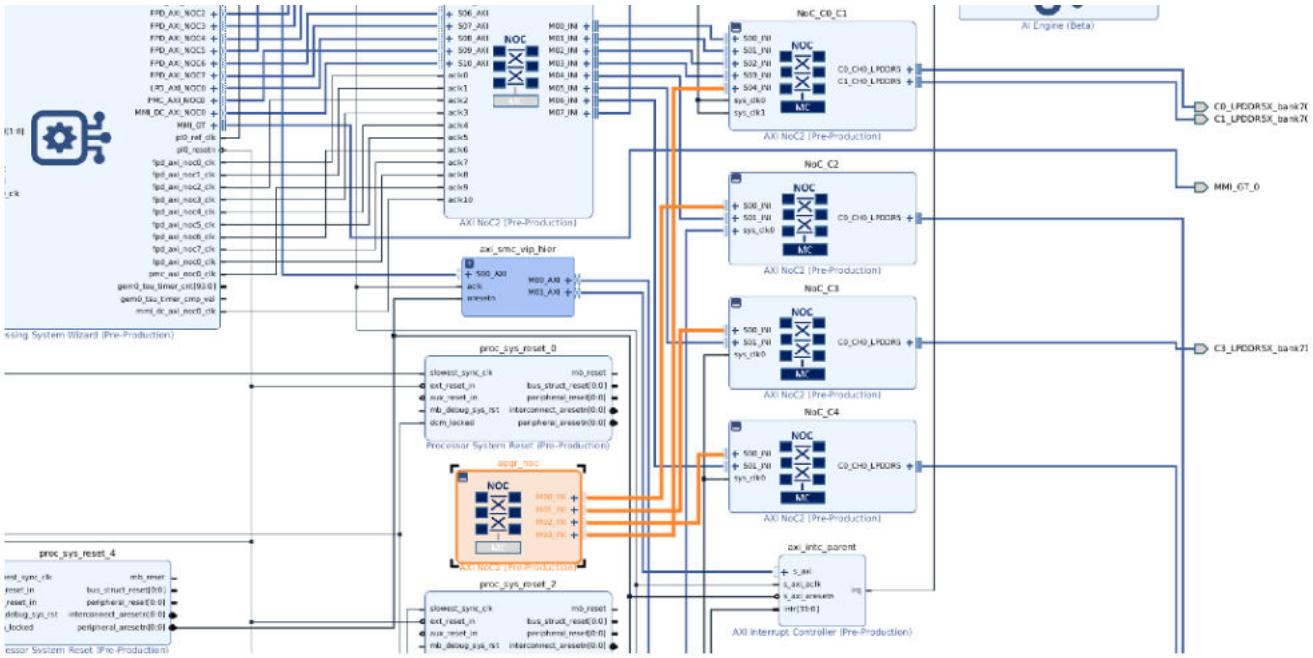
NoC Memory Configuration

The extensible platform built in Vivado as described in Platform Creation Basics provides a comprehensive system for processing system (PS), network-on-chip (NoC), memory controllers, clock sources, other IPs and RTL modules tailored to specific system requirements. The NoC memory configuration, a crucial aspect of this platform, allows access to DDR /LPDDR memories available on the device. Understanding this configuration is paramount for optimizing memory usage and maximizing performance. For more details on the NoC memory configuration, see Configuring the Memory Controller in the *Versal Adaptive SoC Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313).

NoC Memory Configuration in Vivado

The hierarchical approach to NoC memory configuration enables the accessibility of multiple DDR/LPPDDR memories within a platform. An example showcasing this approach is an IP integrator design in Vivado that features four NoC IPs with integrated memory controllers (`NoC_C0_C1`, `NoC_C2`, `NoC_C3` and `NoC_C4`). These memory controller NoC IPs are further connected to an aggregate NoC (`aggr_noc`) IP. This configuration effectively distributes memory access across multiple NoC IPs, enhancing system efficiency.

Figure 20: **Memory Hierarchy in IPI Design**

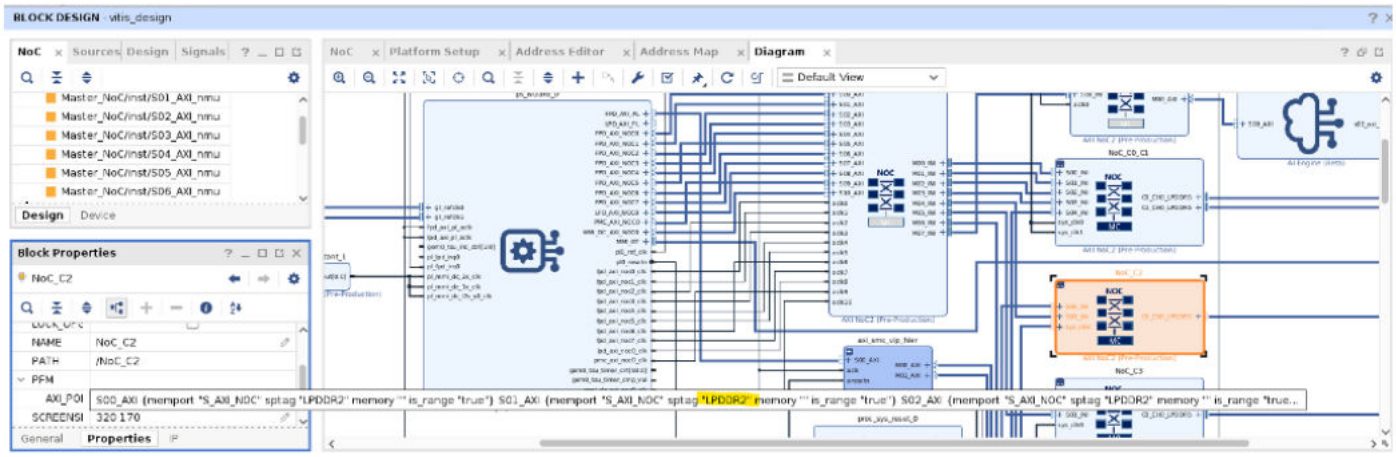


Each of the four memory controller NoC IPs have platform attributes with unique tags set on them, enabling individual access to assigned LPPDR memories For example, `NoC_C2` has `LPDDR2 sptag allowing exclusive access to its associated memory.`

Similarly, the aggregate NoC IP `aggr_noc`has its own unique tag that grants flexibility to access all four LPPDR memories, offering extensive memory utilization as per the application requirements.

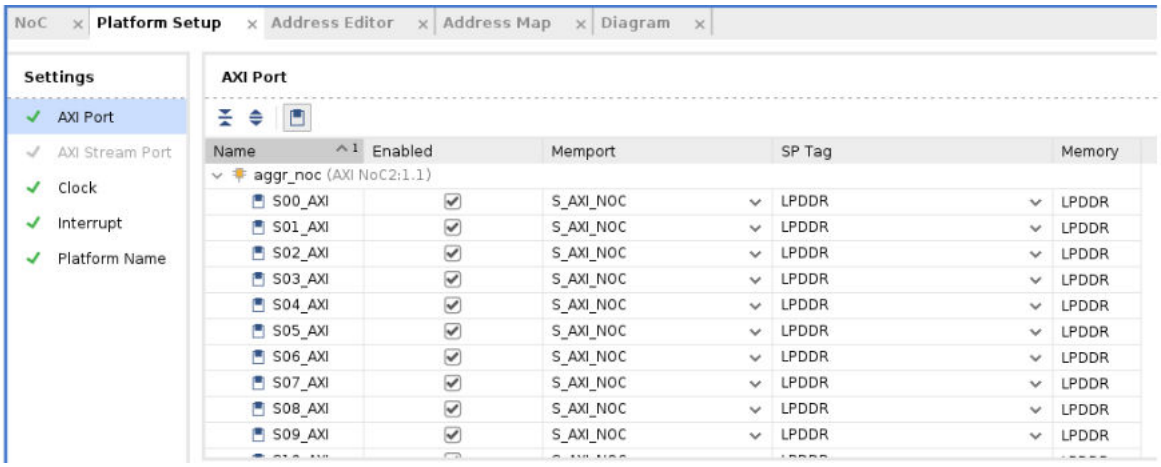
This hierarchical memory structure facilitates access to the entire memory space within the platform through the aggregate NoC or individual memory access based on specific needs. This approach enhances memory management and streamlines application performance.

Figure 21: **Platform Block Properties**



The Platform Setup tab in Vivado displays the SP tag information associated with each NoC IPs.

Figure 22: **SP Tags in Platform Setup**



Accessing Memories in Vitis

To facilitate memory access in Vitis, the linker needs connectivity information to establish connection with the LPPDR memories. As mentioned previously, memory selection is accomplished using unique sptags. For designs involving LPPDR memory access, the input/output AXI ports must be connected to a specific tag using a Vitis connectivity file (system.cfg) as shown below. This file is passed to v++ --link command.

```
[connectivity]
sp=ai_engine_0.M00_AXI:LPDDR
sp=ai_engine_0.M01_AXI:LPDDR
```

```
v++ --link --platform <platform_path> <libadf.a> <kernel.xo> --config
<system.cfg>
```

Leveraging the hierarchical memory setup in the platform design in Vivado, Vitis provides access to multiple memory configurations, catering to diverse application requirements:

1. To access all available memory space: use the sptag for the `aggr_noc` IP. This grants access to all memories connected to the `aggr_noc`, encompassing `NoC_C0_C1`, `NoC_C2`, `NoC_C3` and `NoC_C4`.

   ```
   [connectivity]
   sp=ai_engine_0.M00_AXI:LPDDR
   sp=ai_engine_0.M01_AXI:LPDDR
   ```

2. Accessing Individual Memories:

   a. Accessing `NoC_C0_C1` only

      Utilize the `sptag` LPDDR01 within the Vitis connectivity file to access only `NoC_C0_C1`

      ```
      [connectivity]
      sp=ai_engine_0.M00_AXI:LPDDR01
      sp=ai_engine_0.M01_AXI:LPDDR01
      ```

   b. Accessing `NoC_C2` only

      ```
      [connectivity]
      sp=ai_engine_0.M00_AXI:LPDDR2
      sp=ai_engine_0.M01_AXI:LPDDR2
      ```

   c. Accessing `NoC_C3` only

      ```
      [connectivity]
      sp=ai_engine_0.M00_AXI:LPDDR3
      sp=ai_engine_0.M01_AXI:LPDDR3
      ```

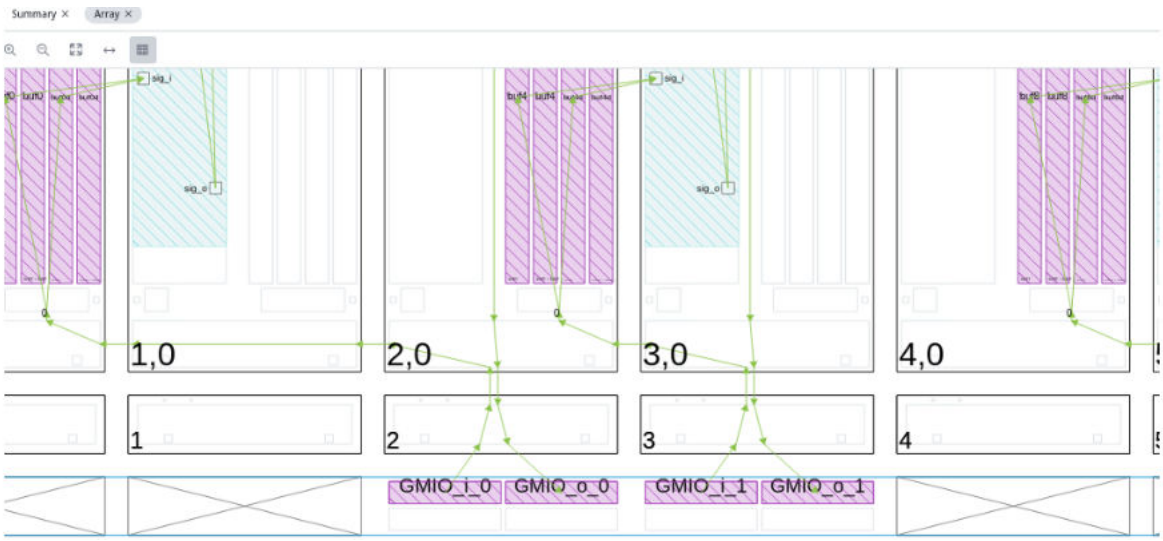
   d. Accessing `NoC_C4` only

      ```
      [connectivity]
      sp=ai_engine_0.M00_AXI:LPDDR4
      sp=ai_engine_0.M01_AXI:LPDDR4
      ```

For AI Engine designs, AIE GMIOs connect to access LPDDR using similar connectivity file as mentioned above.

For instance, for GMIOs (`gmio_i_0, gmio_o_0` and `gmio_i_1, gmio_o_1`) placed on the respective interface tile, the following connectivity options can be used.

Figure 23: **Array View**



Accessing LPDDR using aggreg_noc::

```
[connectivity]
sp=ai_engine_0.gmio_i_0:LPDDR
sp=ai_engine_0.gmio_o_1:LPDDR
```

```
Accessing NoC_C2 individually:
[connectivity]
sp=ai_engine_0.gmio_i_0:LPDDR2
sp=ai_engine_0.gmio_o_1:LPDDR2
```

It is crucial to modify the system device tree (`user-system.dtsi`) to reflect any changes in the NoC memory configuration. This ensures the reserved memory space is accurately reflected in the device tree. For details, refer to the custom platform creation tutorial.

Enabling Hardware Emulation

The following steps enable hardware emulation in extensible hardware platform.

1. Start with your extensible hardware Vivado project with the BD, RTL, test bench, and other sources. This project can be a flat, or DFX based hardware design.
   a. Configure the AI Engine block with only one slave AXI4 Memory-Mapped port connected to NoC. Vitis linker automatically connects the AI Engine Graph interface to the platform.

2. Update the design to include the right simulation models.
   a. For Versal adaptive SoCs only, prepare the platform design to enable SystemC models. Update the CIPS and NoC IP `SELECTED_SIM_MODEL` property to `TLM`. This ensures that for CIPS IP, the design uses QEMU model on which host application runs. Similarly, for Zynq 7000 and Zynq UltraScale+ MPSoCs, set the `SELECTED_SIM_MODEL` on the processing system IP instance. Following Tcl command can be used in the design. Also, set the parameter to enable SystemC simulation in Vivado:

```
foreach tlmCell [get_bd_cells * -hierarchical -filter {VLNV =~
"*:*:axi_noc:*" || VLNV =~ "*:*:versal_cips:*"}] {set_property
SELECTED_SIM_MODEL tlm $tlmCell }
set_param bd.generateHybridSystemC true
```

3. Ensure the test bench is setup correctly.
   a. Create a test bench in the sim_1 fileset that instantiates the `<top>` module of your design.
   b. Your test bench should include a BD wrapper instead of the BD directly. This is because the Vitis hardware emulator uses BD wrappers to insert NoC elements during simulation.
   c. For Versal adaptive SoCs, use the `launch_simulation -scripts_only` command to generate a file called `<top>_sim_wrapper.v`. This file is a wrapper module that instantiates additional simulation models related to the aggregated NoC.
   d. The following Tcl commands generates the necessary NoC simulation files and can be used in your simulation sources.
   e. Ensure the top synthesis module is also set as the top for simulation in the sim_1 fileset.

```
# Ensure that your top of synthesis module is also set as top for
simulation

set_property top <rtl_top> [get_filesets sim_1]

# Generate simulation top for your entire design which would include
# aggregated NOC in the form of xlnoc.bd

launch_simulation -scripts_only
update_compile_order -fileset sim_1

# Set the auto-generated <rtl_top>_sim_wrapper as the sim top

set_property top <rtl_top>_sim_wrapper [get_filesets sim_1]
update_compile_order -fileset sim_1

#Generate the final simulation script which will compile
# the <syn_top>_sim_wrapper and xlnoc.bd modules also
launch_simulation -scripts_only
launch_simulation -step compile
launch_simulation -step elaborate
```

f. Compile the design and start simulation. The design is configured to use QEMU, so the CIPS IP does not generate transactions because there is no software present while simulating in the Vivado simulator. The following ERROR message is observed during Vivado simulation, but this indicates that the basic design has loaded correctly.

```
############################################################
#
 #  Simulation does not work as Versal CIPS
Emulation (SELECTED_SIM_MODLE=tlm) only works with Vitis
tool(launch_emulator.py tool in Vitis)
#
############################################################

ERROR: [Simtcl 6-50] Simulation engine failed to start: The
Simulation shut down unexpectedly during initialization.
```

***Note*:** To confirm that the design generates the right transactions, you can simulate the design using the CIPS VIP. Ensure the `SELECTED_SIM_MODEL` property is set to `RTL` for the NoC and CIPS IP. Also, create a test bench that drives the CIPS VIP and meets the requirement of the NoC Verilog model. Refer to the CIPS VIP and NoC IP documentation for additional details on how to set up test bench for Verilog-based simulation.

4. Package the HW Emulation only XSA.
   a. Package the HW Emulation only XSA. Ensure that all source files are local to the project before exporting it. Use the Vivado **File → Export → Export Platform** option or the following Tcl command to do this:

```
set_property platform.platform_state "pre_synth" [current_project]
write_hw_platform -hw_emu -file  platform_hw_emu.xsa
```

   b. This XSA can be used with pre-built Linux images or with PetaLinux to create a custom Linux image.

Versal Hardware Platform using a CED Design

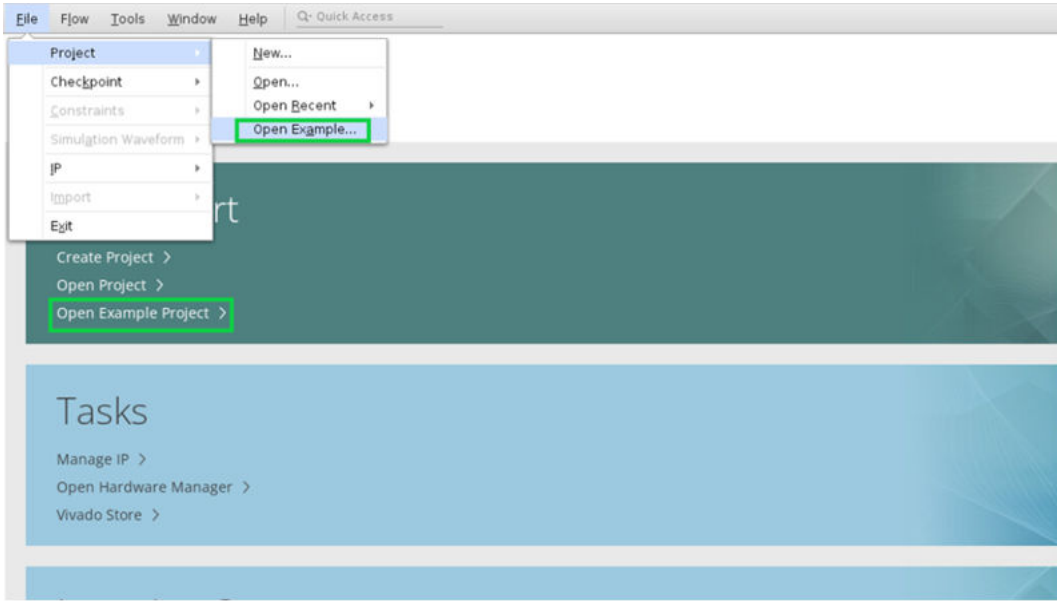
The Configurable Example Design (CED) serves as a valuable resource for creating a flat extensible hardware designs in Vivado. This design acts as a reference for demonstrating the configuration of processing system IPs and the connections between them.

The CED offers a practical example for understanding the intricacies of IP configurations and their interconnectedness. It allows you to follow along with the settings of complex processing system IPs. You can gain insights on how the settings should be configured and connected within a flat hardware design.

IMPORTANT! *A CED can be used as a starting template for a design. However, for production environments, it is your responsibility to verify and qualify that the design setup meets the production requirements.*

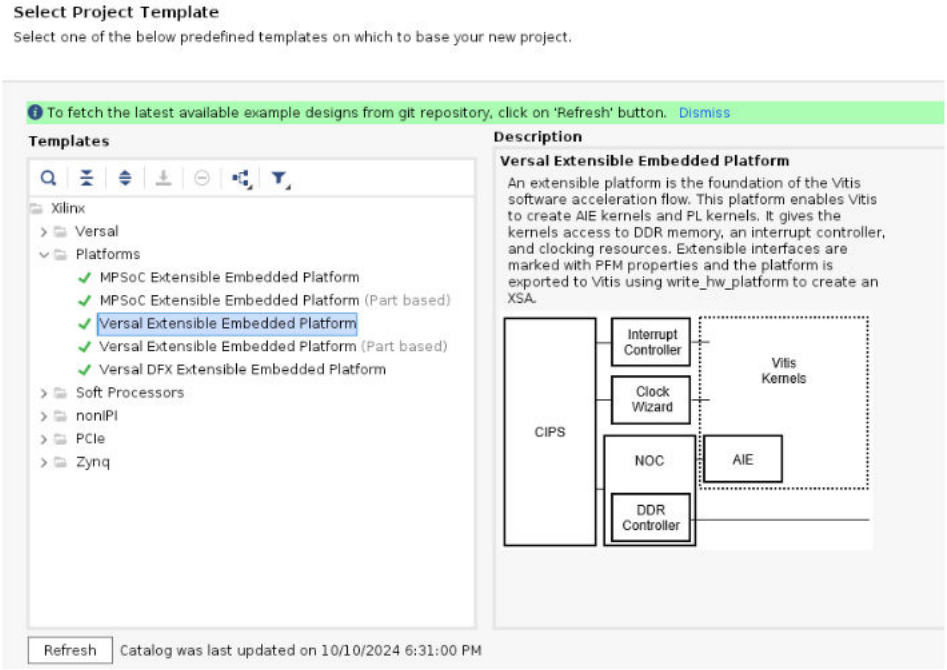
The steps to create a flat hardware design in the Vivado tool by using a CED are as follows.

1. a. Click **File → Project → Open Example**, or click **Open Example Project** from the Quick Start screen.



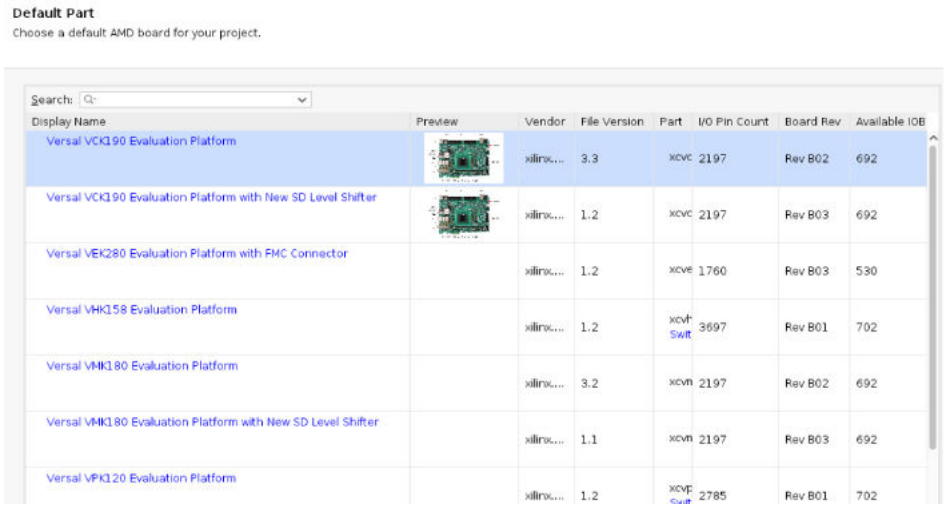
b. Click **Next** on the Create an Example Project screen.

c. Select **Versal Extensible Embedded Platform** in the Select Project Templates window.



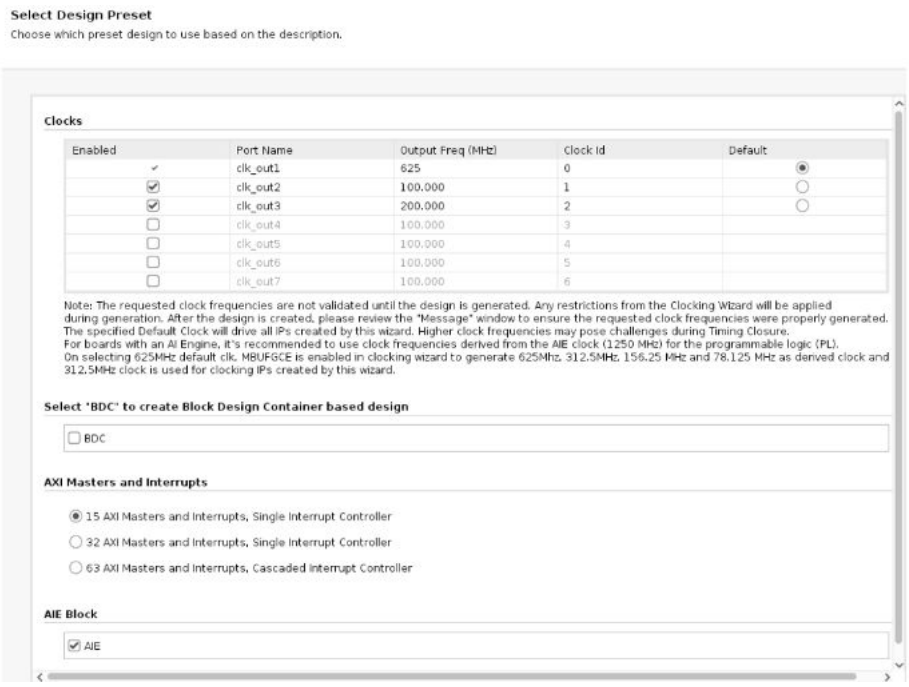
d. Input the project name and project location. Keep Create project sub-directory checked. Click **Next**.

e. Select the target evaluation board in the Default AMD board window. In this example design, select Versal VCK190 Evaluation Platform. Click **Next**.



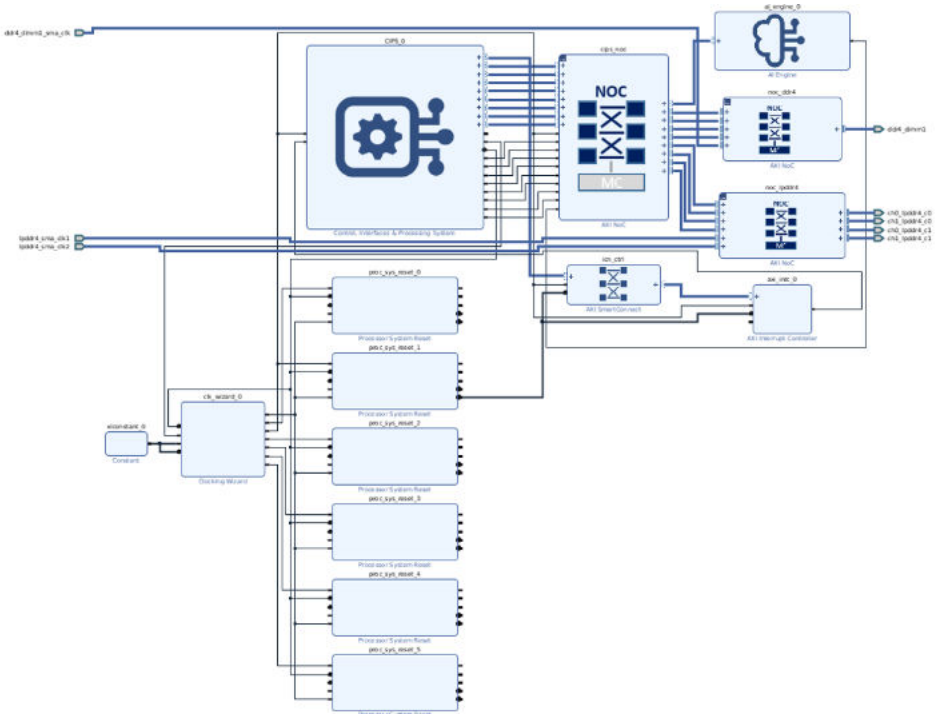
f. In the Select Design Preset section, set the following design options:

- **Clocks:** You can enable clocks, update the output frequency and define default clock in this view. Clock `clk_out1`is set to 625 MHz by default. For AI Engine-based designs, it is recommended to set the clock to 625 MHz. This enables addition of the MBUFGCE clock buffer in the clock wizard to generate 625 MHz, 312.5 MHz, 156.25 MHz and 78.125 MHz. In this example, `clk_out2` and `clk_out3` are enabled with Output Frequency 100 MHz and 200 MHz respectively.
- **Design type:** If the BDC (Block Design Container) option is left unselected a flat design is generated. If you select the BDC option, a BDC based design is generated which is explained in the next section.
- **Configure Interrupt Settings:** Choose interrupts as per the design requirement. The example design supports following:
  - 15 and 32 interrupts, which uses single interrupt controller only.
  - 63 interrupts, which uses two interrupt controller in cascade.
- **AI Engine Block:** This block is selected by default.



g. Click **Next**.

h. Review the project summary and click **Finish**.



i. Observe the following sections in the generated design in the IP integrator:

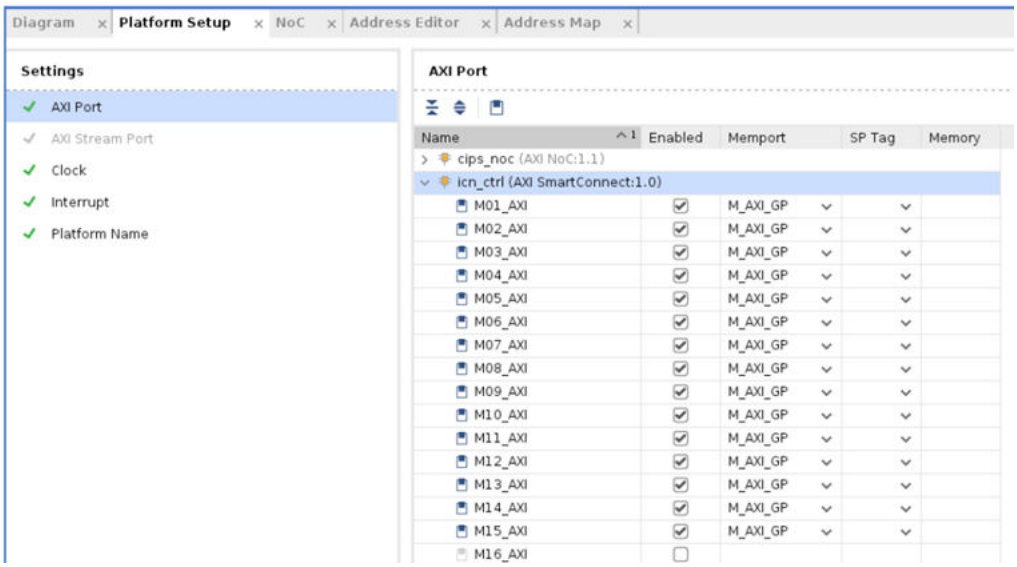
- **AI Engine IP Block:** The AIE IP block is instantiated in the design, which is connected to CIPS through cips_noc. A configuration port used by CIPS using the cips_noc IP to configure the AI Engine IP.

- **Memory Section:** Two types of DDR memory controllers are instantiated in the design: `DDR4` and `LPDDR4` through noc_ddr4 and noc_lpddr4 NoC IPs respectively. These memory controllers are connected to the CIPS IP through cips_noc. You can adjust the memory controller configuration through the noc_ddr4 and noc_lpddr4 settings. In this example, the default NoC settings are used.
- **Clock Section:** Three clocks have been generated using the clock wizard.
- **Interrupt Section:** Review the section and confirm the settings.

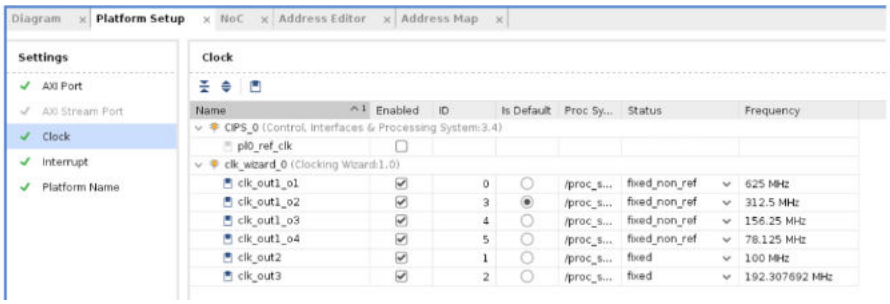
Review Platform Setup

The steps below review the hardware design and confirm that all the appropriate platform properties and tags have been set.

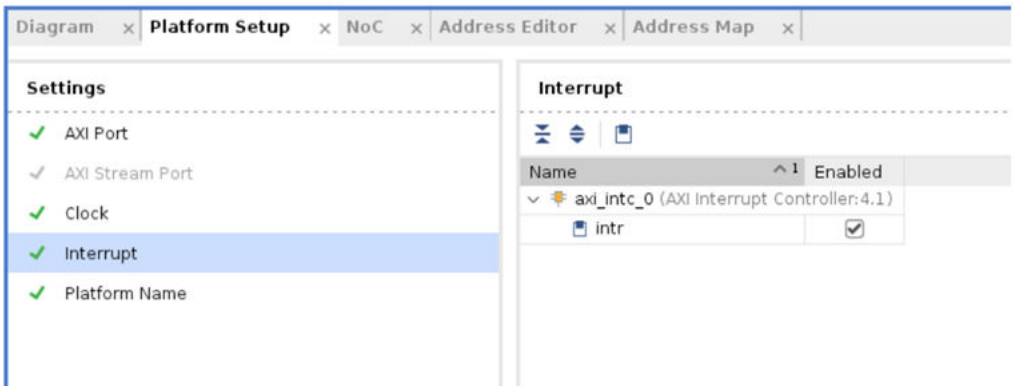
1. AXI port settings:
   - In `noc_ddr4` IP, ports `S00_AXI` to `S13_AXI` are enabled and SPTAG value is set to `DDR`.
   - In `icn_ctrl` IP, ports`M01_AXI` to `M15_AXI` are enabled and set to `M_AXI_GP`. The SP Tag should be left empty. These ports are the AXI master interfaces used to control PL kernels.



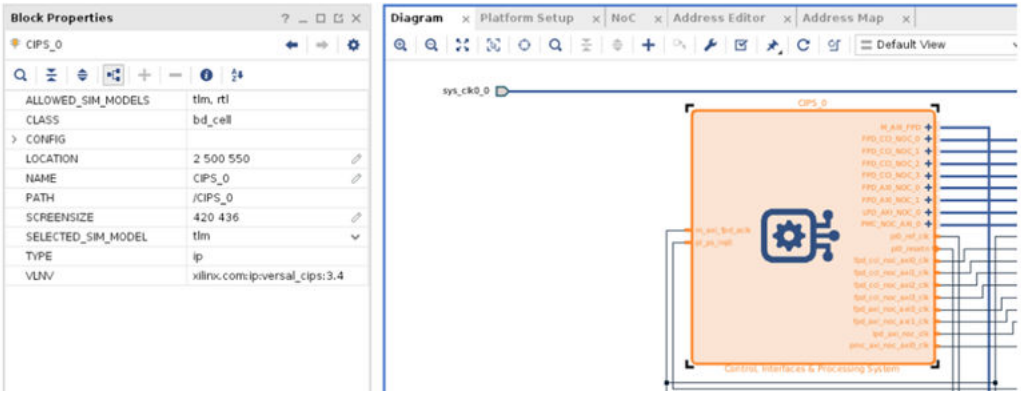
2. In the Clock tab, `clk_out1_o1`, `clk_out1_o2`, `clk_out1_o3`, `clk_out1_o4`, `clk_out2`, `clk_out3` from `clk_wizard_0` are enabled with id {0,3,4,5,1,2}, and frequency {625 MHz, 312.5 MHz, 156.25 MHz, 78.125 MHz, 100 MHz, 200 MHz}. `clk_out1_o2` is the default clock. The v++ linker uses this clock to connect the kernel if there are no clocks specified in the link configuration. The Proc Sys Reset property should be set to the synchronous reset signal associated with each clock. The clock status `fixed_non_ref` indicates that v++ linker cannot use these clocks as a reference to generate a requested clock during the linking phase, it can be used only to connect to IPs. The clock status `fixed` indicates that v++ linker can use these clocks as a reference clock as well as to connect IPs as defined in `system.cfg`. Refer Managing Clock Frequencies for more details.



3. Interrupt settings:
   - In the Interrupt tab, confirm that ports `In0` to `In31` are enabled.



4. Simulation model:
   - Select the **CIPS** instance and verify that the ALLOWED_SIM_MODELS in the Properties tab, in the Block Properties window is `tlm`, and `rtl`, and the SELECTED_SIM_MODEL is `tlm`. It is mandatory to select the same simulation model for NoC and AI Engine.



Generate XSA

Once your hardware platform is ready, use the `write_hw_platform` command to create a Hardware Design (XSA) file of the hardware design.

Extensible XSA

The extensible hardware design (`.xsa`) is created in Vivado and contains platform interfaces that allow the addition of programmable components such as AI Engine, PL kernels and subsystem features. The extensible `.xsa` is a container that is used by the Vitis linker command in both the Vitis Integrated Flow, and Vitis Export to Vivado Flow. This XSA file must be generated in Vivado prior to use by the Vitis linker.

The Export Hardware Platform wizard in Vivado supports three types of extensible hardware designs:

1. Export XSA for hardware only: used to export extensible XSA to support hardware flow only.
2. Export XSA for hardware emulation only: used to export extensible XSA to support hardware emulation flow only.
3. Export XSA for hardware and hardware emulation: used to export extensible XSAs for both hardware and hardware emulation flows.

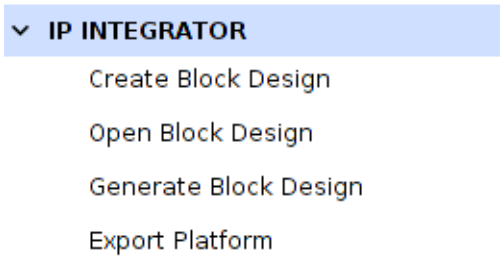
***Note*:** For hardware emulation, if you have an IP which does not have a simulation model (does not support hardware emulation), you can remove that IP to create a hardware emulation XSA for the complete project except for that IP. Create another hardware XSA for the Vivado project with that IP, and create an `xpfm` to merge both XSAs into `.xpfm`.

When the Vivado project type is set to the extensible Vitis platform, the Export Platform wizard is available from the **File → Export → Export Platform** menu command.

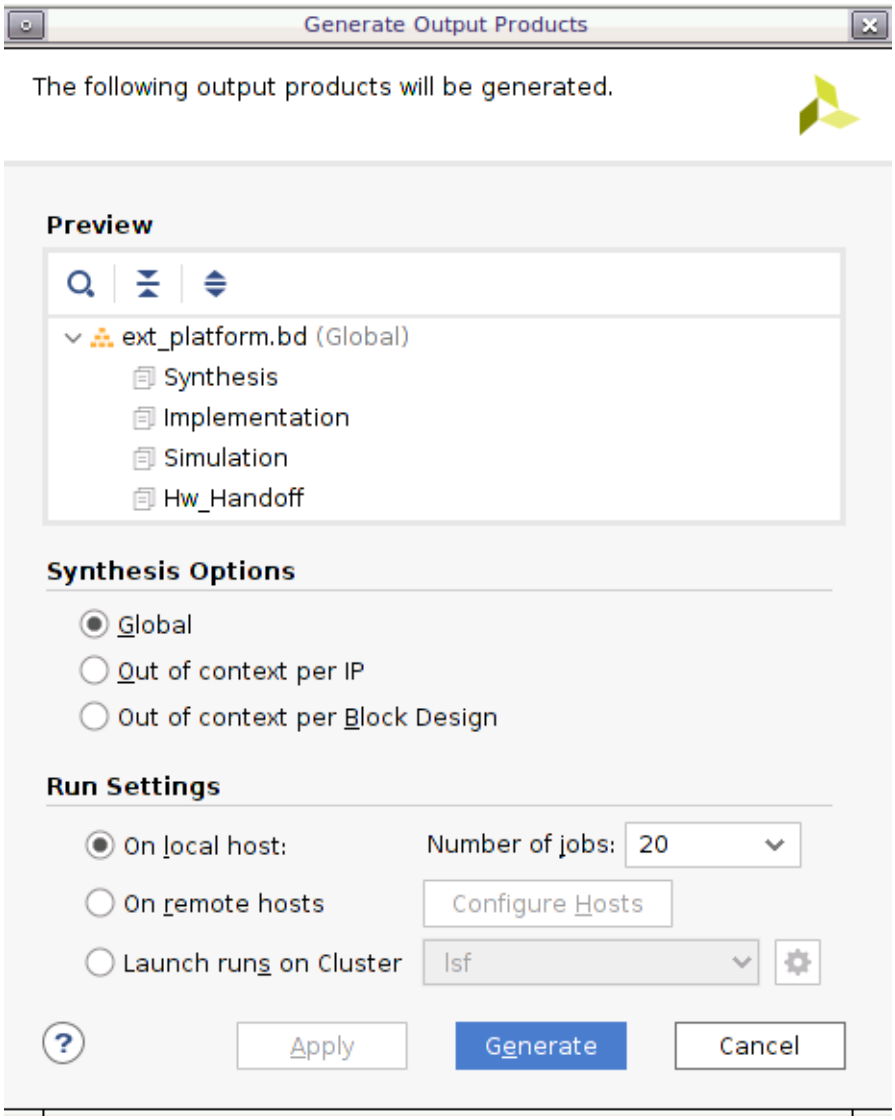
IMPORTANT! *The block design must have an HDL wrapper and have output targets generated to export the platform XSA. Use the Create HDL Wrapper from the right-click menu in the Sources window to create the wrapper, and Generate Block Design from the Flow Navigator in the Vivado Design Suite.*

Export Extensible XSA

1. Generate the Block Diagram.
   a. Click **Generate Block Diagram** from the Flow Navigator window.



   b. Select **Synthesis Options** to **Global** to save generation time. Click the **Generate** button.



2. Export the hardware platform with the following scripts.
   a. Click **File → Export → Export Platform**. Alternatively, you can select from the **Flow Navigator** window: **IP Integrator → Export Platform**, or the **Export Platform** button at the bottom of the Platform Setup tab.
   b. Click **Next** from the Export Hardware Platform screen.
   c. Select **Hardware**. If there are IPs that don't support simulation, the Hardware XSA and Hardware Emulation XSA need to be generated separately. Click **Next**.
   d. Select **Pre-synthesis**, as a DFX platform is not being created. Click **Next**.
   e. Input the name and click **Next**.
   f. Update the filename and click **Next**.
   g. Review the summary and click **Finish**.

You can also perform this in the command line using the following command:

```
set_property pfm_name {vendor:board:name:version} [get_files <bd_file>]
write_hw_platform -hw -force <XSA file>
```

Note: The emulation design should match the hardware in most cases. For large and complex designs, it might require decomposition for faster emulation. To ensure emulation equivalence, ensure the two design are logically identical.

To create and combine a hardware XSA and a hardware emulation XSA, use the following commands:

```
write_hw_platform -hw <hw_platform>
write_hw_platform -hw_emu <hw_emu_platform>
combine_hw_platform -hw <hw_platform> -hw_emu <hw_emu_platform> -o
<combined_platform>
```

Commands can be used to export the XSA file for only the DFX platform.

```
#emulation XSA
set_property platform.platform_state "pre_synth" [current_project]
write_hw_platform -hw_emu -force -file vck190_custom_dfx_hw_emu.xsa
#hardware and RP XSA
set_property platform.platform_state "impl" [current_project]
write_hw_platform -force -fixed -static -file vck190_custom_dfx_static.xsa
write_hw_platform -force -rp design_1_i/VitisRegion vck190_custom_dfx_rp.xsa
```

Note: To export the XSA from DFX and BDC hardware design, use the following code.

```
For DFX design, RP BD should have the PLATFORM.NAME. Static BD should not
have this property set.
For BDC design, child BD should have the PLATFORM.NAME. Parent BD should
not have this propert set.
```

Fixed XSA

A fixed XSA is the hardware handoff that contains hardware design with implementation and design closure completed in the Vivado Design Suite. The fixed XSA is generated by the Vitis link step as described below.

In the context of the Vitis Integrated Flow, a fixed XSA is generated when an extensible platform is linked to Vitis components using the Vitis link process by running `Linking the System` command.

In the context of the Vitis Export to Vivado Flow, start with the VMA (Vivado Management Archive) generated by the Vitis linker, import it in Vivado, make necessary modifications, perform synthesis and implementation, achieve timing closure, and finally generate the fixed XSA in Vivado as described below.

Use the Tcl command to create the fixed XSA in Vivado:

```
write_hw_platform -fixed <fixed_xsa>.xsa
```

For information on creating an embedded design in Vivado Design Suite and generating an XSA file, see the following embedded design tutorials:

- *Embedded Design Tutorials: Versal Adaptive Compute Acceleration Platform* (UG1305)
- *Zynq UltraScale+ MPSoC: Embedded Design Tutorial* (UG1209)
- *Zynq 7000 SoC: Embedded Design Tutorial* (UG1165)

This fixed XSA can be packaged with the software platform and taken to hardware. It can also be used with the Vitis tools to develop or modify application software or the AI Engine software while maintaining the AI Engine-PL interface.

Software Platform

The software platform serves as the foundation for running applications on AMD-powered Versal and ZynqMP devices. The software platform provides the environment to run and control kernels. It serves as the control mechanism for both fixed and extensible platforms. It includes the domain setup and boot components for setting up, resetting and configuring the hardware platform. The software platform encompasses several key elements:

- **System Device Tree:** This file defines the components and configurations of your hardware, enabling proper interaction with the software.
- **Board Support Package (BSP):** This package houses libraries and drivers specific to your platform, providing access to functions like peripherals, memory management, and processor communication.
- **Operating System and Boot Components:** These components manage loading and running the software on your platform. The Linux option offers extensive features, while bare-metal and RTOS provide more focused control and real-time performance.

***Note*:** It is crucial to remember that hardware platform modifications necessitate updating the software platform to ensure compatibility.

Support for Operating System and Processors

The software platform supports various operating systems across different processors:

- **Bare-Metal:** Ideal for resource-constrained applications, it runs on APU, RPU, and PMC processors and provides a basic single-threaded environment.
- **Linux:** Designed for multi-user and multi-tasking environments, it operates on APU processors and leverages open-source drivers for all processing system peripherals.

- **RTOS:** This option caters to applications requiring deterministic scheduling and low latency, typically found in APU, RPU, and PMC environments.

Bare-metal

AMD provides a bare-metal software stack included within the AMD Vitis tools package. This software includes a simple, single-threaded environment tailored for resource-constrained hardware. It provides support for standard input/output and access to essential processor hardware features. The bare-metal software includes board support packages (BSP) containing standalone drivers and libraries. These components are configurable to provide the necessary functionality with minimal overhead, ensuring efficient resource utilization. This software stack is specifically designed for RPU and APU processors commonly found in embedded systems. The XSCT command line tool can be used to configure and build the BSPs.

Refer to Bare-Metal Software Stack in the *Versal Adaptive SoC System Software Developers Guide* (UG1304)

Refer to *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400) for more details on XSCT command line tool and steps to configure and package the bare-metal BSP.

The bare-metal drivers can be found in the following location of the Vitis install:

```
<Vitis Installation
Directory>\Vitis\<version>\data\embeddedsw\XilinxProcessorIPLib\drivers
```

The bare-metal libraries can be found in the following path:

```
<Vitis Installation
Directory>\Vitis\<version>\data\embeddedsw\lib\sw_services
```

Linux Based

The Linux-based software platform is supported on the APU processors of both Versal and Zynq UltraScale+ MPSoC devices. The software platform components can also be generated using PetaLinux, Yocto, or other third-party frameworks. To control the AI Engine graphs and/or PL kernels from the software application running on Linux you must configure and enable the XRT drivers in the software platform.

This platform consists of:

- **Root File System (RFS):** This includes essential binaries, libraries, and configurations for a functional Linux file system.
- **Kernel Image:** This refers to the compiled Linux kernel with supported peripheral drivers.
- **Sysroot:** This tool allows for cross-compilation of the host application, providing the necessary libraries for compiling applications for the target system.

More details can be found here *PetaLinux Tools Documentation: Reference Guide* (UG1144).

All pre-built AMD platforms have the software platform provided. By default, the platforms have a Linux domain with the Xilinx Runtime (XRT) enabled so that applications can run on the platform. Because the device tree is unique to each platform, it is provided as a component with the Linux XRT domain inside the platform. Common software images can be downloaded from the Embedded Platforms download page.

The source files for embedded platforms are available on GitHub at Vitis Embedded Platform Source. You can use these files as a reference for your custom software platform development.

Note: For more details on the Linux software stack, refer to Linux Software Stack in the *Versal Adaptive SoC System Software Developers Guide* (UG1304).

RTOS

The FreeRTOS BSP provides a lightweight, multi-threaded environment tailored for embedded systems. It offers basic functionalities like standard input, output, and access to processor hardware features. The BSP and its associated libraries are highly configurable, allowing developers to achieve necessary functionalities with minimal resource overhead.

The FreeRTOS software stack closely resembles the bare-metal stack, but it incorporates the FreeRTOS library, enabling real-time multitasking capabilities. Importantly, AMD device drivers included with the standalone libraries can be seamlessly integrated within FreeRTOS, provided that only a single thread requires access to the device.

For more insight into the FreeRTOS software stack, refer to the section: FreeRTOS Software Stack in the *Versal Adaptive SoC System Software Developers Guide* (UG1304).

Boot Components

The software platform components are compiled as binaries, and are packaged along with the hardware platform into device images. The AMD boot image format (BIF) file is used to stitch binary files together and generate device boot images. Bootgen defines multiple properties, attributes, and parameters that used in creating boot images for use in an AMD device.

The following is an example of a BIF file targeting a ZYNQMP device:

```
the_MCS_image:
{
[bootloader, destination_cpu=a53-0] <plnx-proj-root>/images/linux/
zynqmp_fsbl.elf
[pmufw_image] <plnx-proj-root>/images/linux/pmufw.elf
[destination_device=pl] <plnx-proj-root>/project-spec/hw-description/
project_1.bit
[destination_cpu=a53-0, exception_level=el-3, trustzone] <plnx-proj-root>/
images/linux/bl31.elf
[destination_cpu=a53-0, load=0x00100000] <plnx-proj-root>/images/linux/
system.dtb
[destination_cpu=a53-0, exception_level=el-2] <plnx-proj-root>/images/
linux/u-boot.elf
}
```

The following is an example of a BIF file targeting a Versal device:

```
the_ROM_image:
{
        { type=bootimage, file=<system_project_path>/system.pdi }
    }
    image
    {
        name=aie_image, id=0x1c000000
        { type=cdo
          /* The following commented lines show the CDOs used to create a
merged CDO 'aie.cdo.merged.bin'. For debugging purpose,
             uncomment these CDOs and comment the line that adds
'aie.cdo.merged.bin' */
          /*
          file = <aie_project_path>/package/libadf/sw/aie.cdo.reset.bin
          file = <aie_project_path>//package/libadf/sw/
aie.cdo.clock.gating.bin
          file = <aie_project_path>//package/libadf/sw/
aie.cdo.error.handling.bin
          file = <aie_project_path>//package/libadf/sw/aie.cdo.elfs.bin
          file = <aie_project_path>//package/libadf/sw/aie.cdo.init.bin
          */
          <aie_project_path>/package.hw/aie.merged.cdo.bin
        }
    }
    image
    {
        name=default_subsys, id=0x1c000000
        { load=0x1000, file=<project_path>/system.dtb }
        { core=a72-0, exception_level=el-3, trustzone, file=<project_path>/
boot/bl31.elf }
        { load=0x8000000, core=a72-0, exception_level=el-2,
file=<project_path>/boot/u-boot.elf }
    }
```

The steps to integrate a hardware platform with the software platform and generate boot images that can be used on the board, based on the Vitis flow used, is described here: Integrating the System.

Creating an Embedded Platform (XPFM)

You can package the extensible hardware design (XSA) with the software platform, and generate a Vitis platform (xpfm). You can do this using either the Vitis Unified IDE or the Python CLI (command line) tool.

- In the Vitis IDE, select **File → New Component → Platform** to create a Vitis platform.
- Using Python CLI, you can use the platform related command to create a platform and the associated domains.
- For more information about these flows, see *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400).

The platform (xpfm) is an encapsulation of multiple hardware and software components. This encapsulation makes it easier to hand off deliveries from hardware-oriented engineers to application developers. The v++ tool supports consuming XSA files directly, bypassing the traditional platform (XPFM) stage. This allows hardware developers to focus on designing and validating the hardware, leaving the xpfm platform creation for later.

The following files and information are packaged into the xpfm platform.

- **Hardware Specification:** This is an extensible XSA file.
- **Software Components:** These are added to the platform as a Linux or Bare-metal domain. Software components include the following:
  - Boot components with their corresponding BIF file for Bootgen to generate the boot.bin file.
  - Boot components directory containing all files described in the BIF file.
  - Image directory (optional): Contents copied into the final SD card image's FAT32 partition.
  - Required Linux domain: Includes kernel, RootFS, and sysroot information added during platform or application creation.
  - Emulation support files (optional).

Validating an Embedded Platform

The following sections describe different actions you can take to examine and exercise the embedded platform for software, graph, and kernel development to ensure the platform meets your needs and expectations.

Validating the Hardware Platform

The hardware platform design you created in a Vitis platform is static after the platform creation process is complete.

Vitis does modify parameters based on certain IPs (for example, SmartConnect, NoCs) by adding additional master/slave interfaces. In some situations, PS/CIPS interfaces can also be modified and AMD Versal™ adaptive SoC and AI Engine IP is instantiated in the platform.

The following table shows the workflows to validate the hardware platform design on your board.

Table 27: **Platform Workflows**

Workflow	Development	Validation
Basic board bring-up	Processor basic parameter setup.	Standalone Hello world and Memory Test application run properly.
Advanced hardware setup	Enable advanced I/O in Processing System (such as USB, Ethernet, Flash, PCIe®, or RC). Add I/O related IP in PL (such as MIPI, EMAC, or HDMI_TX). Add non-Vitis IP (such as AXI block RAM Controller, or Video Processing Subsystem (VPSS) IP).	If these IP have standalone drivers, test them.
Software setup	Create PetaLinux project based on hardware platform. Enable kernel drivers. Configure boot mode. Configure rootfs.	Linux boots up successfully. Peripherals work properly in Linux.

Check Platform Metadata

During the platform creation process you will define the resources available in the platform. You can perform a quick examination of the platform definition, and the available resources using the `platforminfo` command. This utility can print platform metadata to let you confirm that the defined resources of the packaged platform meet your expectation. For more information about this command, refer to platforminfo Utility in the *Vitis Reference Guide* (UG1702).

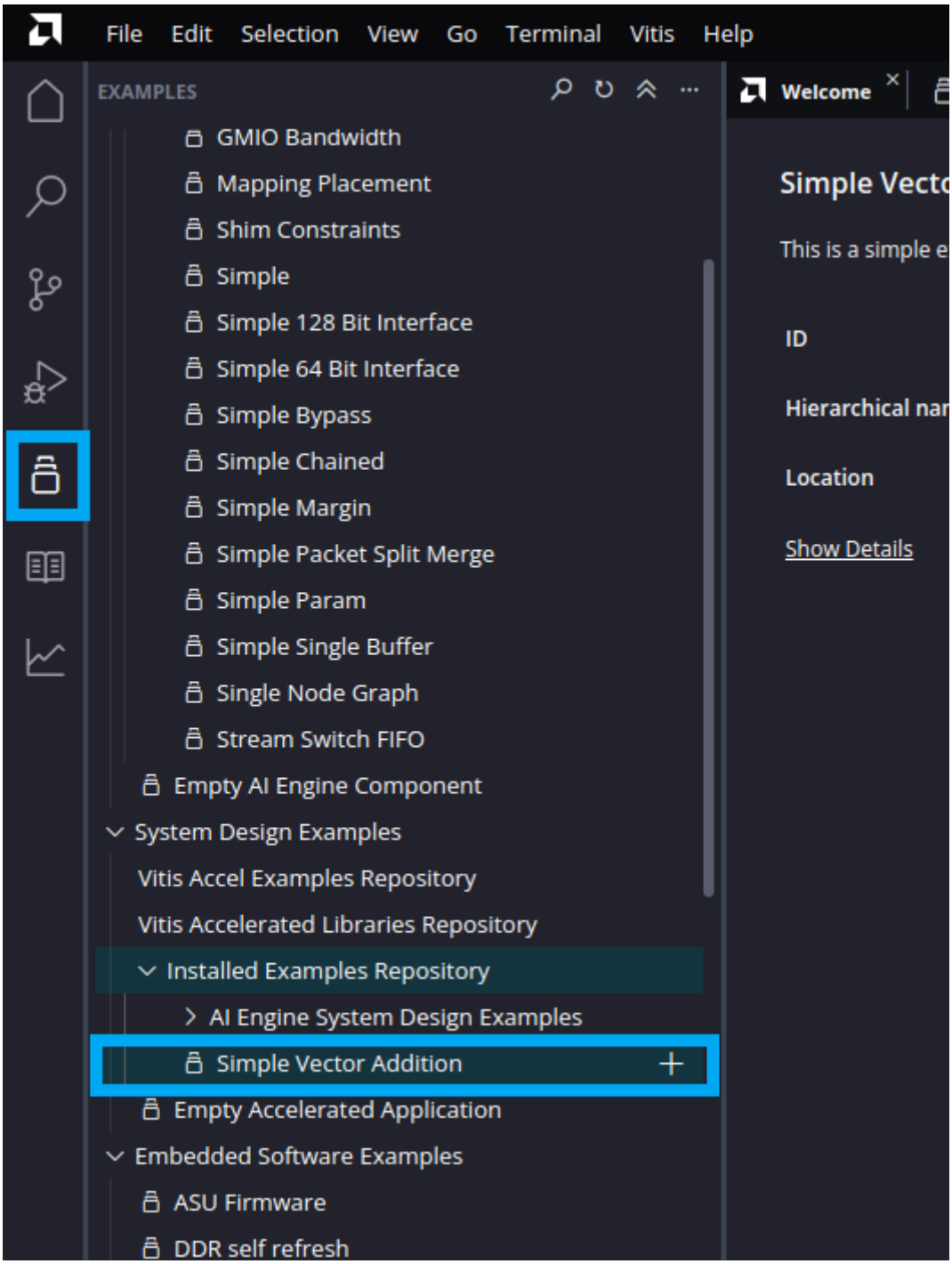
Use the Platform with Simple Test Case

By using the platform with a simple test case, you can ensure that the systems work as expected, without needing to debug the host application, graph, or kernel interactions. A simple test can exercise a specific feature set of the platform, without the complexities of design. AMD provides a number of examples and tutorials for use as test cases, some of which can be found at the following location:

https://github.com/Xilinx/Vitis-Tutorials

A simple test (located in Examples of Vitis GUI) as shown below can help to check whether the clock and reset, AXI4 interface settings, and interrupts are correct and functioning as expected.

Figure 24: **Simple Test**



Design Validation Using XSA instead of XPFM

The v++ tool now supports using XSA files directly as input instead of relying on the traditional platform (XPFM), which includes both hardware and software components that need to be prepared meticulously. Traditionally, you had to link the platform with the kernel using the v++ tool to obtain the final hardware design. However, with the new capability of consuming XSA files directly via the `v++ --platform` option, you can bypass the packaging platform (XPFM) stage. This enhancement allows hardware developers to focus on designing and validating hardware quickly, delaying the platform creation step until later in the process.

The following is a `v++ usage` example:

```
v++ -l <kernel> --platform <extensible xsa> -s -g  -t <target> -o fixed.xsa
```

Refer to Vitis Tutorials for Versal Extensible Hardware Design examples.

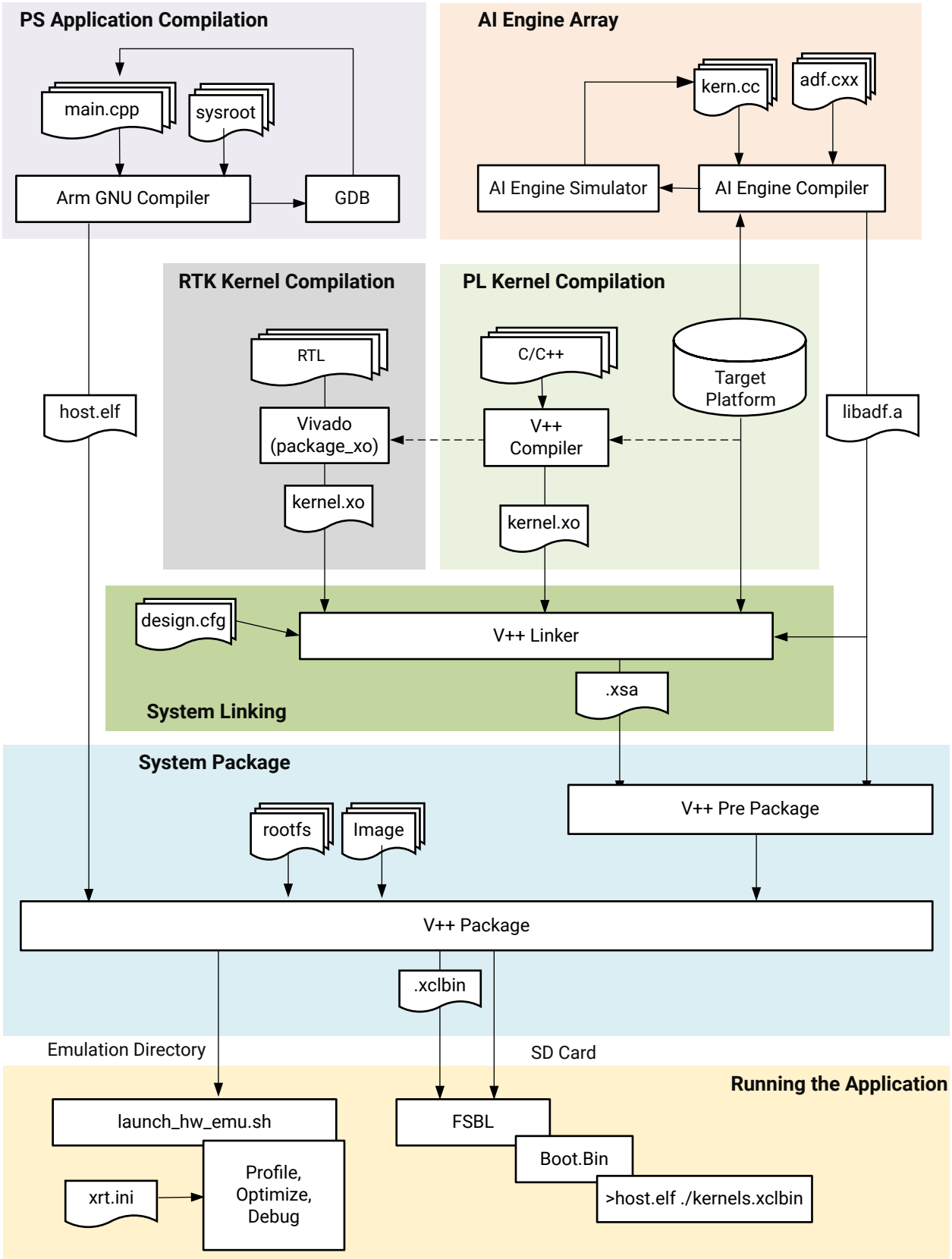
Building and Running the System

The AMD Vitis™ tool simplifies hardware design and integration with a software-like compilation and linking flow, integrating the four domains of the AMD Versal™ device: the AI Engine array, the programmable logic (PL) region, the network-on-chip (NoC) and the processing system (PS). The Vitis linker flow lets you integrate your compiled AI Engine design graphs (`libadf.a`) with additional kernels implemented in the PL region of the device, including HLS and RTL kernels, and link them for use on a target platform. The Vitis linker provides abstract directives for accessing system memory, CPU control, and streaming I/O, so it is often possible to develop AI Engine graphs and kernels on a standard development platform and quickly re-target the AI Engine code to a custom platform developed for your specific application. You can control AI Engine graphs and PL kernels from code running on an embedded Arm® processor on the Versal device or from an external CPU.

The following figure shows the high-level steps required to use the Vitis tools flow to integrate your application. The command-line process to run this flow is described here.

Note: You can also use this flow from within the Vitis IDE as explained in Launching Vitis Unified IDE in the *Vitis Reference Guide* (UG1702).

Figure 25: **Vitis Integrated Flow**



***Note*:** Running the application is either on hardware or in emulator.

IMPORTANT! *Using Vitis tools and AI Engine tools require the setup described in Setting Up the Vitis Environment.*

The following steps can be adapted to any Vitis project targeting a Versal device.

1. AMD provides Pre-built Base Platforms for select devices and recommends using the Vitis Integrated Flow. Pre-built embedded base platforms installed with the Vitis installer can be targeted for the design flow or a Custom Platform created can be targeted for the design flow.
2. As described in Compiling an AI Engine Graph Application in the *AI Engine Tools and Flows User Guide* (UG1076), the first step is to create and compile the AI Engine graph into a `libadf.a` file using the AI Engine compiler. You can iterate between the AI Engine compiler and the AI Engine simulator to develop the graph until you are ready to proceed.
3. PL Kernel Compilation: PL kernels are compiled for implementation in the PL region of the target platform using the `v++ --mode hls` command, see HLS Kernel Development. In addition to Vitis compilation, you can use Vivado to package RTL modules as kernels in the compiled `.xo` format as described in Packaging the RTL Code as a Vitis XO. See v++ Mode HLS in the *Vitis Reference Guide* (UG1702).
4. Linking the System: Link the compiled AI Engine graph with the HLS kernels and RTL kernels onto a target platform. The process creates an XSA file to encapsulate the implemented hardware system to create boot and loadable images.

   ***Note*:** During linking, a NoC design rule check is performed. See Validate NoC DRCs in the *Versal Adaptive SoC Hardware, IP, and Platform Development Methodology Guide* (UG1387) for more details.
5. PS application compilation: Optionally compile a host application to run on the Cortex®-A72 core processor using the GNU Arm cross-compiler to create an ELF file. The host program interacts with the AI Engine kernels and kernels in the PL region. This compilation step is optional because there are several ways to deploy and interact with the AI Engine kernels, and the host program running on the PS is one way.
6. Packaging for Vitis Flow: Use the `v++ --package` process 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. The packager can also be invoked to build the necessary package to run emulation and debug, or run your application on hardware.

Integrating AIE and PL Components

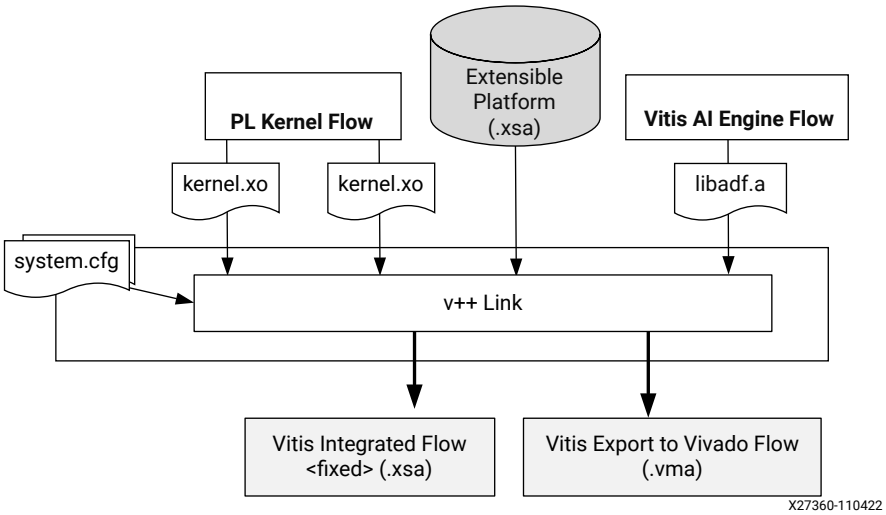
Vitis components (AIE graphs and PL kernels) are assembled together and integrated with an extensible platform using the Vitis v++ linker.

The following sections provide details of the linking process.

Linking the System

The following figure shows two paths through the linking process, starting with the `v++ --link` command. The first path is the Vitis Integrated Flow, where the `v++` command links the elements of the system, automatically launches the Vivado tools for implementation of the design, and outputs a `.xsa` file. The second path is the Vitis Export to Vivado flow, where the `v++` command links the elements of the system and outputs a `.vma` file for you to use in the Vivado tools for synthesis, implementation, and timing closure. These two paths are explained below.

Figure 26: **Linking the System Design**



Note: The Export to Vivado flow is currently only supported for custom Versal platforms.

At the design level, the `v++` link command operates within a Vitis managed region as a hierarchy in the extensible hardware platform block design. Based on source input files, the `v++` linker instantiates user-defined PL kernels, configures platform IP such as AI Engine, NoC, and soft interconnects, adds required design IP for AXI buses, clock domain crossing, data width conversion, and FIFO buffering, adds networks for hardware debugging, trace, and clocking, and creates connections between IP within the Vitis managed region.

After Linking

After the linking step is complete, any reports generated during this process are collected into the `<kernel_name>.link_summary`. This collection of reports can be viewed by opening the link_summary in the Analysis view of Vitis analyzer, and includes a Summary report, System Estimate providing timing and resources estimates andSystem Guidance offering suggestions for improving linking and the performance of the system. Refer to Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702) for additional information.

TIP: *Refer to Output Directories of the v++ Command in the Vitis Reference Guide (UG1702) for an understanding of the location of various output files.*

The linking process defines important architectural details of the system design. In particular, this is where the design is enabled for profiling or debug, where you specify the number of kernel instances to instantiate into hardware, where kernel instances are assigned to SLRs, and where you define connections from PL kernel ports to global memory or to AI Engine applications. The following sections discuss some of these build options: Chapter 10: Profiling and Tracing the Application and Chapter 11: Debugging System Projects.

Linking with the Vitis Integrated Flow

In the Vitis Integrated Flow, the linker automatically runs Vivado synthesis and implementation on the project (hence the name "Integrated"), and creates a fixed hardware platform (.xsa) for use by the Vitis packaging process as described in Packaging for Vitis Flow.

The following is an example command line to link a PL kernel (`vadd.co`) with an AI Engine graph (`libadf.a`) and a Versal adaptive SoC platform, using specific linking options (`system.cfg`):

```
v++ -t hw_emu --platform xilinx_vck190_base_202520_1 --link vadd.xo
libadf.a --config ./system.cfg -o binary_container.xsa
```

This command contains the following arguments:

- **`-t <arg>`:** Specifies the build target. When linking, you must use the same `-t` and `--platform` arguments specified when compiling the PL kernels and AI Engine graph application.
- **`--platform <arg>`:** Specifies the platform to link with the system design. In the example command above the `custom_vck190` platform is a custom platform designed to work with the `--export_archive` command.
- **`--link`:** Link the kernels, graph, and platform into a system design.
- **`<input>.xo`:** Specifies the input PL kernel object files (`.xo`) to link with the AI Engine graph and the target platform. This is a positional parameter.
- **`libadf.a`:** Specifies the input compiled AI Engine graph application to link with the PL kernels and the target platform. This is a positional parameter. If your design uses multiple partitions, you need to specify each corresponding `libadf` file when linking the design. For detailed instructions on handling multiple partitions, refer to Compiling AI Engine Graph for Independent Partitions in the *AI Engine Tools and Flows User Guide* (UG1076).
- **`--config ./system.cfg`:** Specify a configuration file that is used to provide `v++` command options for a variety of uses, including connectivity and debug options. Refer to Vitis Compiler Configuration File in the *Vitis Reference Guide* (UG1702) for more information on the `--config` option.
- **`-o binary_container.xsa`:** Specifies the output file name. The output file in the link stage will be an `.xsa` file due to the use of a Versal platform. The default output name is `a.xsa`.

TIP: *For AMD Zynq™ MPSoC based platforms, the output of the link command will be an* `.xclbin` *file rather than the* `.xsa`.

Linking with the Vitis Export to Vivado Flow

In the Vitis Export to Vivado flow, the linker stops before running RTL synthesis and outputs a project archive (VMA file) for export to the Vivado Design Suite. This flow lets you open the Vitis archive in the Vivado tools for directed synthesis, place and route, and timing closure. The Vitis Export to Vivado flow requires custom Versal platforms designed specifically to support this feature.

An example command follows:

```
v++ --target hw --platform custom_vck190 --link vadd.xo libadf.a --config ./
system.cfg \
--export_archive -o hw-vadd.vma
```

IMPORTANT! *The* `--export_archive` *command cannot be used with the* `--target hw_emu` *(or* `-t`*) command. An error will be returned.*

This command is similar to the prior command, with the following differences:

- **`--export_archive`:** Specifies the creation of the `.vma` file to export to the Vivado Design Suite. This option stops `v++` from automatically running Vivado synthesis and place and route, and instead lets you manually launch and direct the implementation and timing closure of the design as described in Vitis Export to Vivado Flow.
- **`--platform custom_vck190`:** The `--export_archive` command can only be used with a custom platform compatible with the Vitis export to Vivado flow.
- **`-o hw-vadd.vma`:** Specifies the output file name for the `.vma` file produced by the `--export_archive` command.

The `.vma` file is imported into the original extensible platform project in the Vivado Design Suite using the `vitis::import_archive` Tcl procedure. Development can then continue in the Vivado project, including additional design modifications, simulation, synthesis, and implementation.

After implementation and timing closure, the `write_hw_platform -fixed` command has been enhanced to encapsulate the XRT metadata from the `.vma` file into the output `.xsa`. You can also export the XSA for the hardware emulation target from the Vivado tool and run emulation in the Vitis tool.

If the `.vma` is updated or iterative design methodology is used, the previously imported VMA needs to be removed using `vitis::remove_archive_hierarchy` before the updated VMA is imported.

Vitis Export to Vivado Flow Detailed Example further explains the flow.

Linking the VSS Component to the Platform

The linking and generation of the VMA with a VSS component follows the same setup as Vitis export to Vivado, with the addition of the `.vss` archive.

```
v++ --link --target hw --export_archive -save-temps --platform
<platform_name> --config ./src/system.cfg <list_of_xo> <VSS archive> --
output <VMA file>
```

Enabling Profile and Debug when Linking

To capture and visualize profiling and trace information, or to enable your design for debugging, you will need to add specific commands during the `v++` linking phase, and sometimes during `v++` compilation. The tool must instrument the profile IP using --profile Options in the `v++` linking phase and enable the profiling during the runtime. To enable debugging the application you can specify one of the --debug Options.

During `v++` linking, the application developer needs to add profile IP to the design to capture the profiling data and preferably choose the memory resources for storing and offloading data during the runtime.

- You can add PL monitors to capture tracing information on their design by using `--profile` command. This adds the logic to capture profile data for data traffic between the kernel and host, kernel stalls, the execution times of kernels, and compute units (CUs). The instrumentation can be added using options, `--profile.data`, `--profile.stall`, and `--profile.exec`, as described in --profile Options.
- You also can specify the choice of memory resources or FIFO in the PL to store captured data. On large designs that span multiple SLRs, the tracing infrastructure can cause timing issues as there is one offload point and all trace data must cross SLRs to reach it. For these use cases, multiple memory resources can be used for offloading the trace data.

To enable the capture of profile data or trace information during the application runtime, you can choose from a variety of options to add to the xrt.ini File, which configures the runtime. See Profiling the Application in the *Data Center Acceleration using Vitis* (UG1700) for more information.

Creating Multiple Instances of a Kernel

By default, the linker builds a single hardware instance from a kernel. However, you can instantiate multiple hardware compute units (CUs) from a single kernel. This can improve performance as the host program can now make multiple calls to a given kernel, and see them executed in parallel on the different compute units.

Multiple CUs of a kernel can be created by using the `connectivity.nk` option in the `v++` config file during linking. Edit a config file to include the needed options, and specify it in the `v++` command line with the `--config` option, as described in v++ Command in the *Vitis Reference Guide* (UG1702).

For example, for the `vadd` kernel, two hardware instances can be implemented in the config file as follows:

```
[connectivity]
#nk=<kernel name>:<number>:<cu_name>,<cu_name>...
nk=vadd:2
```

Where:

- **`<kernel_name>`:** Specifies the name of the kernel to instantiate multiple times.
- **`<number>`:** The number of kernel instances, or CUs, to implement in hardware.
- **`<cu_name>,<cu_name>...`:** Specifies the instance names for the specified number of instances. This is optional, and the CU name will default to kernel_1 when it is not specified. The delimiter between kernel instances is a comma. In the example above, the `kernel_name` and the `number` of CUs are specified, but not the `cu_name`. In this case vadd_1 and vadd_2 will be added to the design.

Then the config file is specified on the `v++` command line:

```
v++ --config vadd_config.cfg ...
```

TIP: *You can check the results by using the `xclbinutil` command to examine the contents of the `xclbin` file. Refer to xclbinutil Utility in the Vitis Reference Guide (UG1702).*

The following example results in three CUs of the `vadd` kernel, named `vadd_X`, `vadd_Y`, and `vadd_Z` in the `xclbin` binary file:

```
[connectivity]
nk=vadd:3:vadd_X,vadd_Y,vadd_Z
```

Mapping Kernel Ports to Memory

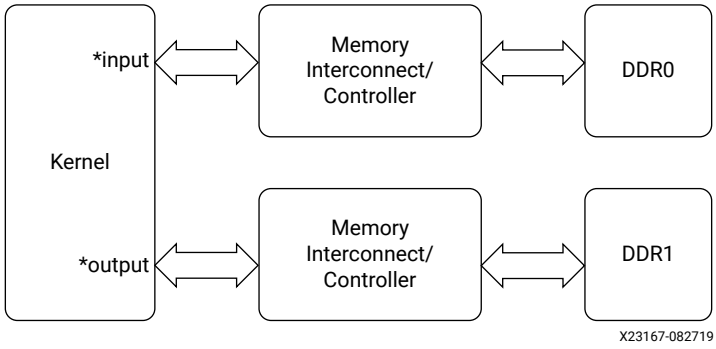
The link phase is when the memory ports of the kernels are connected to memory resources such as DDR, HBM, and PLRAM.

By default, all kernel memory interfaces are connected to the same global memory bank (or `gmem`). As a result, only one kernel interface can transfer data to or from the memory bank at one time, limiting the performance of the application due to memory access.

Because of this, it is important to explicitly specify which global memory bank each kernel argument (or interface) is connected to. Proper configuration of kernel to memory connectivity is important to maximize bandwidth, optimize data transfers, and improve overall performance. Even if there is only one compute unit in the device, mapping its input and output arguments to different global memory banks can improve performance by enabling simultaneous accesses to input and output data.

The following block diagram illustrates the Using Multiple DDR Banks example in Vitis-Tutorials on GitHub. This example connects the input pointer interface of the kernel to DDR bank 0, and the output pointer interface to DDR bank 1.

Figure 27: **Global Memory Two Banks Example**



IMPORTANT! *Up to 15 separate kernel interfaces can be connected to a given global memory bank. Therefore, if there are more than 15 memory interfaces in the design you must explicitly perform the memory mapping as described here, using the* `--conectivity.sp` *option to distribute connections across different memory banks.*

Start by assigning the kernel arguments to separate bundles to increase the available interface ports, then assign the arguments to separate memory banks. The following example uses the interfaces described in HW Interfaces in the *Data Center Acceleration using Vitis* (UG1700).

1. In the C/C++ kernel code assign arguments to separate bundles using the INTERFACE pragma prior to compiling them:

```
void cnn( int *pixel, // Input pixel
  int *weights, // Input Weight Matrix
  int *out, // Output pixel
  ... // Other input or Output ports

#pragma HLS INTERFACE m_axi port=pixel offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
```

In this example, the `cnn` kernel has 3 arguments: `pixel`, `weights` and `out`. Using the bundle attribute of the INTERFACE pragma, each argument is mapped to a specific interface. The `pixel` and `out` arguments are both mapped to the same interface called `gmem`. The weights argument is mapped to a different interface called `gmem1`. The resulting kernel has therefore 2 distinct interfaces (`gmem` and `gmem1`) which can be connected to different memory banks.

IMPORTANT! *You must specify `bundle=` names using all lowercase characters to be able to assign it to a specific memory bank using the `--connectivity.sp` option.*

2. Use the `--connectivity.sp` option, or include it in a config file, as described in --connectivity Options.

   For example, for the `cnn` kernel shown above, the `connectivity.sp` option in the config file would be as follows:

```
[connectivity]
#sp=<compute_unit_name>.<argument>:<bank name>
sp=cnn_1.pixel:DDR[0]
sp=cnn_1.weights:DDR[1]
sp=cnn_1.out:DDR[0]
#sp=<aie_instance>.<gmio_port>:<memory_sp_tag_name>[bank_number]
sp=ai_engine_0.my_gmio:LPDDR[0]
```

   Where:

   - `<compute_unit_name>` is an instance name of the CU as determined by the `connectivity.nk` option, described in Creating Multiple Instances of a Kernel, or is simply `<kernel_name>_1` if multiple CUs are not specified.
   - `<argument>` is the name of the kernel argument. Alternatively, you can specify the name of the kernel interface as defined by the HLS INTERFACE pragma for C/C++ kernels, including `m_axi_` and the `bundle` name. In the `cnn` kernel above, the ports would be `m_axi_gmem` and `m_axi_gmem1`.

     TIP: *For RTL kernels, the interface is specified by the interface name defined in the `kernel.xml` file.*

   - `<bank_name>` is denoted as `DDR[0]`, `DDR[1]`, `DDR[2]`, and `DDR[3]` for a platform with four DDR banks. You can also specify the memory as a contiguous range of banks, such as DDR[0:2], in which case XRT will assign the memory bank at runtime.

     Some platforms also provide support for LPDDR, PLRAM, HBM, HP or MIG memory, in which case you would use LPDDR[0], PLRAM[0], HBM[0], HP[0] or MIG[0]. You can use the `platforminfo` utility to get information on the global memory banks available in a specified platform. Refer to platforminfo Utility in the *Vitis Reference Guide* (UG1702) for more information.

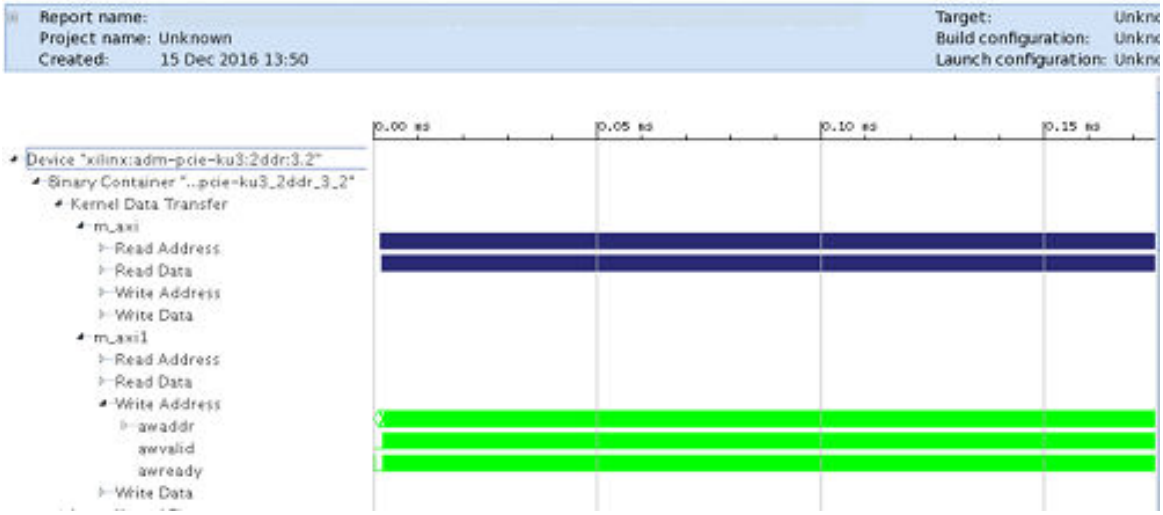
     In platforms that include both DDR and HBM memory banks, kernels must use separate AXI interfaces to access the different memories. DDR and PLRAM access can be shared from a single port.

Note: The SP tag name is declared in Vivado block design as part of the platform properties. The SP tag names can be customized by user to help distinguish specific memory controllers.

TIP: *Assigning kernel interfaces to specific memory banks might also require you to specify the SLR to place the kernel into. For more information, refer to Assigning Compute Units to SLRs on Alveo Accelerator Cards in the Data Center Acceleration using Vitis (UG1700).*

You can use the Device Hardware Transaction view in Vitis Analyzer to observe the actual DDR Bank communication, and to analyze DDR usage.

Figure 28: **Device Hardware Transaction View Transactions on DDR Bank**



Specifying Streaming Connections

Support for hardware accelerator pipelines that communicate through streams is one of the major advantages of FPGAs, FPGA-based SoCs, and Versal adaptive SoC devices; it can be used in DSP and image processing applications, in addition to communication systems. Kernel ports involved in streaming are defined within the kernel, and are not addressed by the host program. There is no need to send data back to global memory before it is forwarded to another kernel for processing. The connections between the kernels are directly defined during the `v++` linking process as described below.

A streaming data output port of one kernel can be connected to the streaming data input port of another kernel, or between a PL kernel and the PLIO of an ADF graph application, during linking using the `--connectivity.sc` option. This option can be specified at the command line, or from a `config` file that is specified using the `--config` option, as described in v++ Command in the *Vitis Reference Guide* (UG1702).

IMPORTANT! *An error occurs if the `--connectivity.sc` kernel drives itself.*

To connect the streaming output port of a producer kernel to the streaming input port of a consumer kernel, set up the connection in the `v++` config file using the `connectivity.stream_connect` option as follows:

```
[connectivity]
#stream_connect=<cu_name>.<output_port>:<cu_name>.<input_port>:
[<fifo_depth>]
stream_connect=vadd_1.stream_out:vadd_2.stream_in
stream_connect=vadd_2.stream_in:ai_engine_0.DataIn0
```

Where:

- `<cu_name>` is an instance name of the CU as determined by the `connectivity.nk` option, described in Creating Multiple Instances of a Kernel. The `cu_name` can be specified in the config file as described in Creating Multiple Instances of a Kernel, or is defined automatically by the tool when not otherwise specified.
- `<output_port>` or `<input_port>` is the streaming port defined in the producer or consumer kernel.

  IMPORTANT! *If the port-width of the output and input ports do not match, the Vitis compiler automatically inserts a data-width converter between the two ports as part of the build process. The inclusion of the data-width converter is either truncate a larger bit-width output to a smaller bit-width input, or expand a smaller bit-width to a larger bit-width.*

- `[:<fifo_depth>]` inserts a FIFO of the specified depth between the two streaming ports to prevent stalls. The value is specified as an integer.

Specifying SLR Region for SSI Devices

In Vitis linker, a CU or kernel is associated to instances marked with `PFM.REGION` by using the same string label with the additional connectivity directive `connectivity.region`. This indicates that the kernel is to be placed close with other kernels or IPs in the same SLR.

For the `v++` linker, the region is set with the command line option `--connectivity.region arg <cu_name>:<label>`. In the linker configuration file, it is specified as follows:

```
[connectivity]
region=<region_label>:<cu_name_1>,<cu_name_2>
```

Managing Clock Frequencies

The selection of clocks and associated frequencies is an important part of defining the performance of algorithms and signal processing blocks of a system. Depending on the source of the input data and destination of the results, different solutions need to be engineered to meet the design requirements. This section describes the clocking of processing elements such HLS, RTL PL kernels or AI Engine kernels added with Vitis to an extensible platform. The interface to input data and output results can be categorized as streaming or memory access.

***Note*:** The number of clocking resources available is device dependent and it is recommended to carefully plan the clock usage.

For streaming access, the bit-width of the data and clock frequency determines the throughput. The HLS, RTL or AI Engine kernels processing the data need to sustain the throughput to avoid loss of data. The throughput used here is defined as:

throughput = bit-width * clock frequency in Hz / initiation interval (bits /second)

For AI Engine, the interface clock frequency is specified on PLIO to determine the DMA scheduling or stream access rate, but the kernel itself always run at AIE Clock. For details see the topic AI Engine-to-PL Rate Matching in the *AI Engine Kernel and Graph Programming Guide* (UG1079).

Table 28: **Clocking Examples for Kernels with Stream Access**

Data access type	Design impact for Vitis	Comments
Synchronous single-rate	Connect to platform clocks and data paths	Kernel clocks match the source and destination in the platform.
Synchronous multi-rate	• Connect to platform clocks and data paths • Optional: Add new clocks • Optional: Add DWC • Optional: Add FIFO	To reduce the clock rate while maintaining throughput requirements, the bit-width can be increased. Vitis will add a Data Width Converter (DWC) block to manage the relationship between the bit-width and clocks whose frequency is in a powers of 2 relation. If necessary, Vitis will also infer missing clocks. For non powers of 2 relations, you need advanced clocking and handshaking techniques. Refer to *Versal Adaptive SoC Clocking Resources Architecture Manual* (AM003) and *Clocking Wizard for Versal Adaptive SoC LogiCORE IP Product Guide* (PG321). ***Note*:** A multi-rate design is synchronous if the clocks have rational relation and a common reference (originates from the same PLL/MMCM). If only one of the multi-rate clocks exist in the platform, Vitis will infer a clock wizard to satisfy this condition.
Time division multiplexing	• Connect to platform clocks and data paths • Optional: Add new clocks • Optional: Add DWC • Add FIFO or buffers	The kernel exploits running at higher throughput than incoming data by buffering the incoming data and processing each buffer in sequence. This is closely related to multi-rate signal processing, except that having buffers is mandatory. ***Note*:** Vitis can infer DWC and additional clocks to support powers of 2 rate changes. The multiplexing mechanism and buffers need to be designed by the kernel developer.

Table 28: **Clocking Examples for Kernels with Stream Access** *(cont'd)*

Data access type	Design impact for Vitis	Comments
Packet-switching	Need buffers and logic for handling the control and payload.	Similar to multi-rate, but can require clock rate overhead to manage the control headers. ***Note***: Vitis does not support automatically inferring packet switching. The packet handling mechanism need to be designed by the kernel developer.
Asynchronous	• Connect to platform clocks and data paths • Optional: Add clocks • Add CDC • Add FIFO	CDC (Clock domain crossing) logic is required to transfer data across unrelated clock domains. The processing kernel throughput needs to be equal or higher than the input data. FIFO buffers need to be inserted to handle differences in throughput and stall handshaking. Refer to Specifying Streaming Connections for adding FIFO. ***Note***: AI Engine is clocked by a separate PLL. Even if the PLIO has same frequency, the phase relation is unknown so CDC are always inserted by Vitis.

Figure 29: **Synchronous Single-Rate with Clock from Platform**

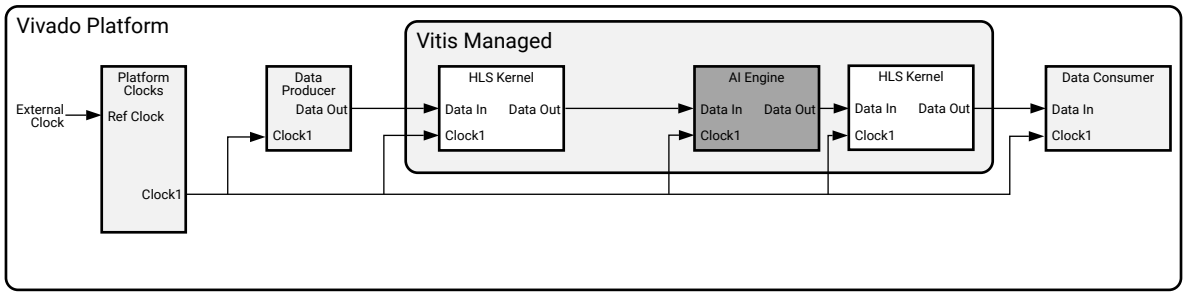


Figure 30: **Synchronous Multi-rate with Clock from Platform and Inferred DWC**

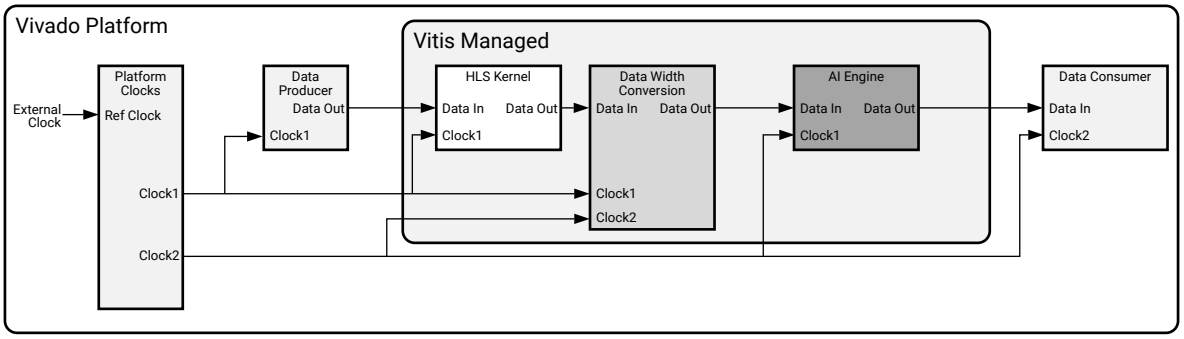


Figure 31: **Synchronous Multi-rate with Inferred Clocks and CDC**

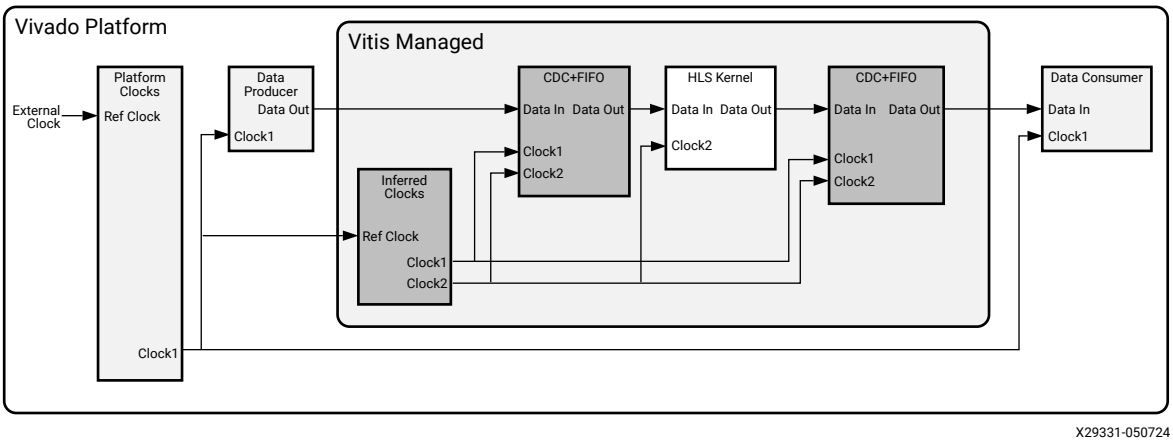
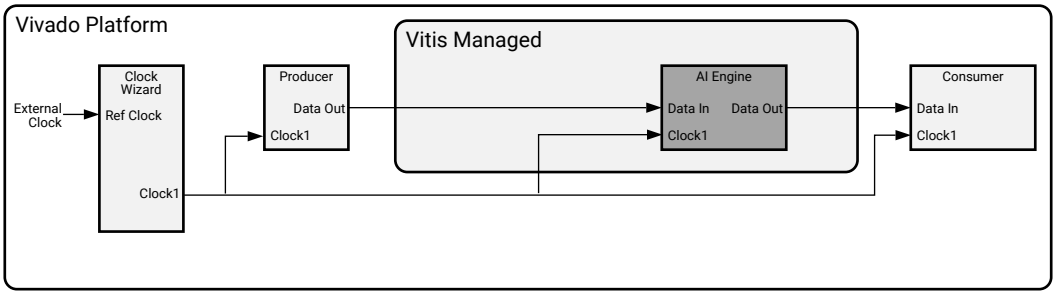


Figure 32: **Asynchronous Single-rate with CDC Related to AI Engine PLIO**



***Note*:** The figures above are simplified for illustrative purposes. The data produced and consumed can be the same block or multiple blocks. The HLS and AI Engine can interact with as many other kernels as resource and interface permits.

In Vitis, all memory types are modeled as AXI slave interfaces with metadata, and any clock conversion is handled implicitly through the AXI network connecting kernel AXI master to the memory AXI slave. In cases with HLS kernels converting from memory to stream or stream to memory, or if the memory access has contention from several kernels, the HLS kernel coding could need performance optimizations to meet the required access type. This is also true if the access is to a shared resource. For details on connecting memories, refer to Mapping Kernel Ports to Memory. For details on HLS coding, refer to Memory Mapped Interfaces in the *Vitis High-Level Synthesis User Guide* (UG1399).

The extensible platform XSA contains information of available clock domains. Running the `platforminfo -d <platformname>.xsa` utility will list the platform clock domains under Clocking Information. For further details and examples, refer to Identifying Platform Clocks.

Table 29: **Using Clock Options in Vitis**

Expected outcome	Link options	Comment
Use default platform clock source pin	Not needed	Vitis automatically connects unspecified kernel to default clock source pin.
Use non-default platform clock	Use `clock.id`	All kernel clock pins will be driven by the platform clock source pin with ID. If the kernel contains multiple clock pins, you can specify `<kernel>.<clk pin>` to differentiate between the clocks.
	• Use `freqhz` • Optional: Use `clock.default_tolerance`	The requested frequency must match any existing platform clocks within the tolerance range. Unless explicitly set, the default tolerance will be 5%.
Add a new clock	Use `freqhz`	Vitis will add a clock wizard to generate the requested clock frequency. In some cases when it's not possible to generate the exact frequency, the tool will generate the closest acceptable frequency within the tolerance range.

For details on how to use link options, refer to --clock Options.

Identifying Platform Clocks

The kernels can have any number of independent and edge-aligned clocks, and platforms can have multiple kernels running at different clock frequencies under user control. Platforms have a variety of clocking: processor, PL, and AI Engine clocking. The following table explains the clocking for each type.

Table 30: **Platform Clocks**

Clock	Description
AI Engine	Can be configured in the platform in the AI Engine IP. The AI Engine PLL frequency must match the CIPS HSM0 frequency.
Processor	Can be configured in the platform in the CIPS IP. The HSM0 frequency must match the AI Engine IP PLL frequency.

Table 30: **Platform Clocks** *(cont'd)*

Clock	Description
Programmable Logic (PL)	Can have multiple clocks and can be configured in the platform.
NoC	Device dependent and can be configured in the platform in the CIPS and NoC IP.

Notes:

1. These clocks are derived from the platform and are affected by the device, speed grade, and operating voltage.

Vitis clocking automation differentiates between three types of platform clock: scalable, fixed, and fixed_non_ref, which are specified in Vivado during platform capture, and define how automation can employ clocks to meet v++ clocking directives.

TIP: *You cannot mix fixed and scalable clocks on a single kernel, but they can be mixed across different kernels within a single* `.xclbin` *file.*

You can determine the clocks available in the target platform by using the `platforminfo` command.

```
==================
Clock Information
==================
  Default Clock Index: 2
  Default Clock Frequency: 312.499712
  Default Clock Pretty Name: PL 2
  Clock Index:          0
    Frequency:          156.249856
    Status:             fixed
    Name:               clk_wizard_0_clk_out2
    Pretty Name:        PL 0
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             6.400006
    Normalized Period:  .006400
  Clock Index:          1
    Frequency:          104.166570
    Status:             fixed
    Name:               clk_wizard_0_clk_out1
    Pretty Name:        PL 1
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             9.600009
    Normalized Period:  .009600
  Clock Index:          2
    Frequency:          312.499712
    Status:             fixed
    Name:               clk_wizard_0_clk_out3
    Pretty Name:        PL 2
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             3.200003
    Normalized Period:  .003200
  Clock Index:          3
    Frequency:          78.124928
    Status:             fixed
    Name:               clk_wizard_0_clk_out4
    Pretty Name:        PL 3
```

```
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             12.800012
    Normalized Period:  .012800
  Clock Index:          4
    Frequency:          208.333141
    Status:             fixed
    Name:               clk_wizard_0_clk_out5
    Pretty Name:        PL 4
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             4.800004
    Normalized Period:  .004800
  Clock Index:          5
    Frequency:          416.666283
    Status:             fixed
    Name:               clk_wizard_0_clk_out6
    Pretty Name:        PL 5
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             2.400002
    Normalized Period:  .002400
  Clock Index:          6
    Frequency:          624.999425
    Status:             fixed
    Name:               clk_wizard_0_clk_out7
    Pretty Name:        PL 6
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             1.600001
    Normalized Period:  .001600
```

For Versal devices, the `--part` option can be used instead of `--platform` with the `v++ --link` and `v++ --package` commands. With `--part` the tool generates a base design for use on the device, and can generally be used while waiting for the development of a full platform specification. However, the v++ linker generated base platform design employs PLRAM only, so is unsuitable for running Linux applications on the target.

- **Fixed Clocks:**

  Platform clocks declared as 'fixed' can be used to drive kernels at their specified frequency, or as a reference clock for v++ clock automation to derive clocks as needed.

  Fixed clocks use MMCMs to generate frequencies other than the fixed frequencies defined on the platform. For example, if you specify clock frequencies: 60, 200, and 450, the Vitis compiler adds all the necessary logic to generate the required clocks from the available platform fixed clocks.

  Use the `--freqhz` option to specify the clock frequency for a kernel. The --clock Options can also be used to specify PL kernel connections to specific platform clocks, or to specify clock frequencies that are generated from fixed clocks on the platform.

  - **fixed_non_ref:**

Platform clocks declared as fixed_non_ref can be used to drive kernels at their specified frequency, but cannot be used as a reference clock by v++ clock automation.

The canonical justification for declaring a platform clock `fixed_non_ref` is because it is driven by an MBUFG logical clock resource.

Refer to --clock Options for more details.

TIP: *You cannot mix fixed and scalable clocks on a single kernel, but they can be mixed across different kernels within a single .xclbin file.*

You can determine the clocks available in the target platform by using the `platforminfo` command.

```
=================
Clock Information
=================
  Default Clock Index: 2
  Default Clock Frequency: 312.499712
  Default Clock Pretty Name: PL 2
  Clock Index:          0
    Frequency:          156.249856
    Status:             fixed
    Name:               clk_wizard_0_clk_out2
    Pretty Name:        PL 0
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             6.400006
    Normalized Period:  .006400
  Clock Index:          1
    Frequency:          104.166570
    Status:             fixed
    Name:               clk_wizard_0_clk_out1
    Pretty Name:        PL 1
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             9.600009
    Normalized Period:  .009600
  Clock Index:          2
    Frequency:          312.499712
    Status:             fixed
    Name:               clk_wizard_0_clk_out3
    Pretty Name:        PL 2
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             3.200003
    Normalized Period:  .003200
  Clock Index:          3
    Frequency:          78.124928
    Status:             fixed
    Name:               clk_wizard_0_clk_out4
    Pretty Name:        PL 3
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             12.800012
    Normalized Period:  .012800
  Clock Index:          4
    Frequency:          208.333141
    Status:             fixed
```

```
    Name:               clk_wizard_0_clk_out5
    Pretty Name:        PL 4
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             4.800004
    Normalized Period:  .004800
  Clock Index:          5
    Frequency:          416.666283
    Status:             fixed
    Name:               clk_wizard_0_clk_out6
    Pretty Name:        PL 5
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             2.400002
    Normalized Period:  .002400
  Clock Index:          6
    Frequency:          624.999425
    Status:             fixed
    Name:               clk_wizard_0_clk_out7
    Pretty Name:        PL 6
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             1.600001
    Normalized Period:  .001600
```

For Versal devices, the `--part` option can be used instead of `--platform` with the `v++ --link` and `v++ --package` commands. With `--part` the tool generates a base design for use on the device, and can generally be used while waiting for the development of a full platform specification. However, the v++ linker generated base platform design employs PLRAM only, so is unsuitable for running Linux applications on the target.

Syncing PL Clocks with the AI Engine

In the ADF programming model, a PLIO represents an AXI4-Stream attachment point from the AI Engine to a PL component (either a PL kernel or a platform IP). PLIO clock frequencies can be specified explicitly to match the PL interfaces in simulation. In addition, when you link the ADF graph into the platform using the `v++ -link` command, you can direct the tools to generate precisely the clock frequencies required by your application. PL kernels can be independently clocked, and the `v++` linker will automatically insert clock domain crossing circuitry into the design as needed.

The recommended best practice to sync AI Engine clocks with PL is as follows:

1. If you know the PLIO clock frequencies, they should be specified in the ADF graph PLIO constructors. If every PLIO will be clocked at the same frequency, instead of specifying in constructors, use `v++ -c --mode aie --freqhz <frequency>`.

   If PLIO clock frequencies are not known at ADF graph compilation time, defer the actual binding until v++ link time and the compiler will assume all are clocked at 100 MHz for simulation.

2. For the Vitis compilation of a PL kernel peer of a PLIO, specify the same clock frequency.

   ```
   v++ -c --mode hls -freqhz <frequency>
   ```

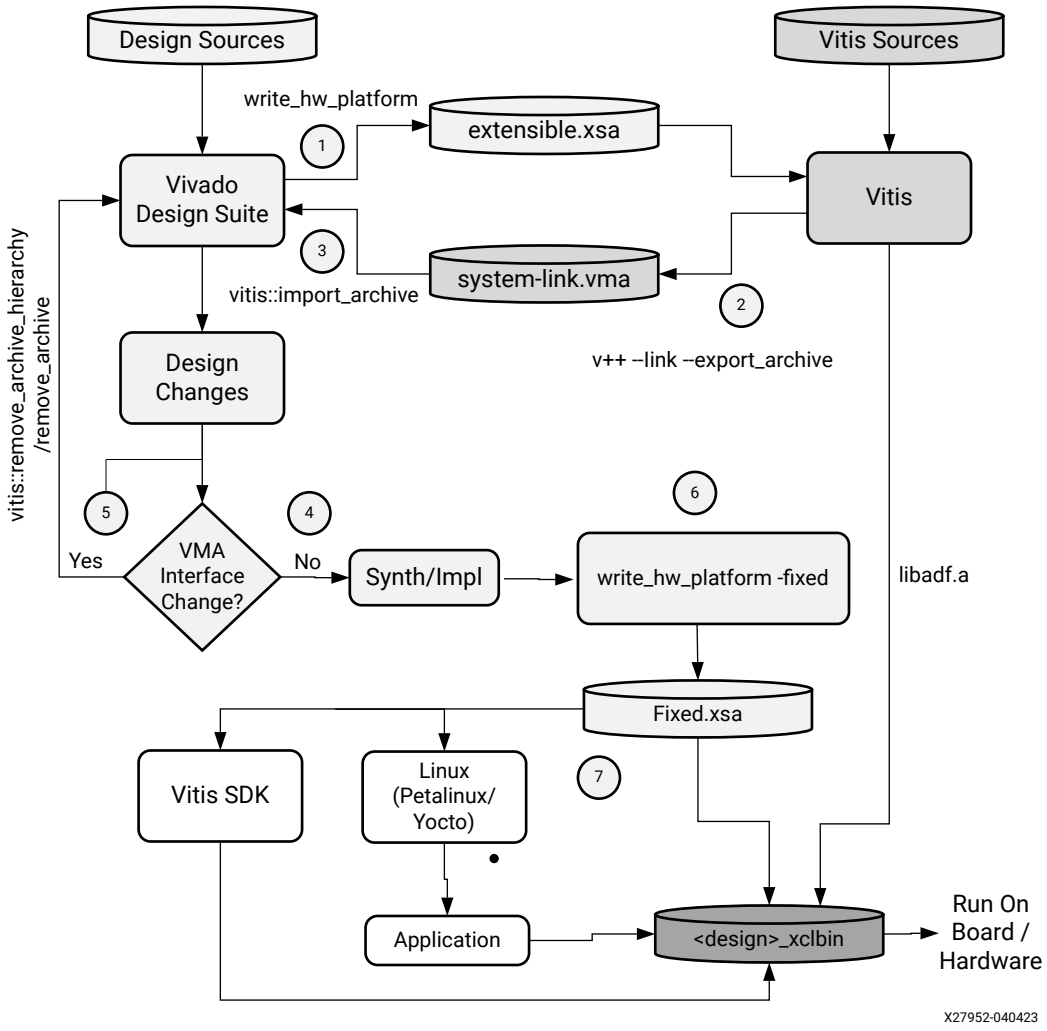
3. When linking the compiled ADF graph (`libadf.a`) with the compiled .xo PL kernels, specify the same PL frequencies. If needed, the link-time clock frequencies can override compile time frequencies. In respect to PLIOs, AI Engine simulation behavior can differ from hardware.

```
v++ -l --platform <pfm_name> --freqhz <frequency>
```

Vitis Export to Vivado Flow Detailed Example

This section explains the Vitis Export flow as shown in the following diagram. The complete flow (from hardware design creation to exporting `.xclbin`) has seven steps.

Figure 33: **Vitis Export to Vivado Flow**



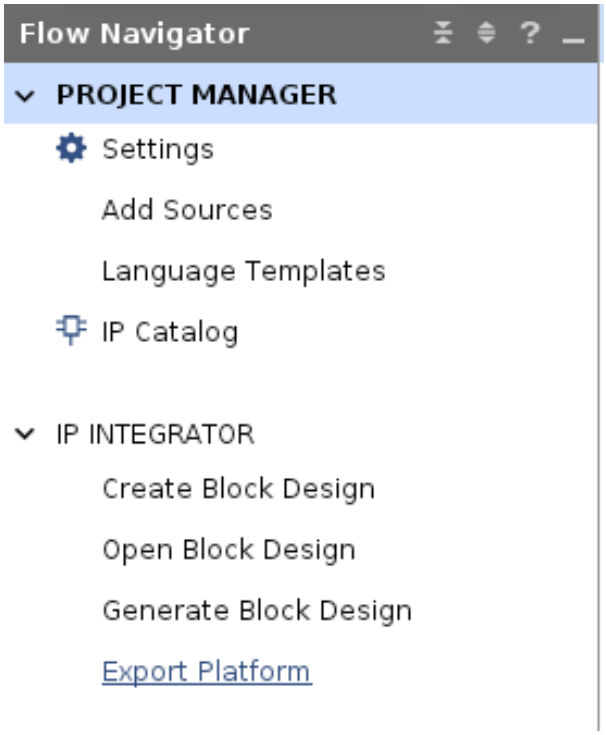
You can execute the flow in the following steps.

1. Start by creating the Versal custom platform design. The Vitis export flow can use either a Flat Hardware Design Platform or a BDC Based Hardware Platform. You can use HLS-based components, or RTL-based packaged IP, or standard IP from the IP catalog in your Vivado design.

   Note: For details on exporting platform from Vivado to Vitis, see Exporting Platforms to Vitis in the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994).

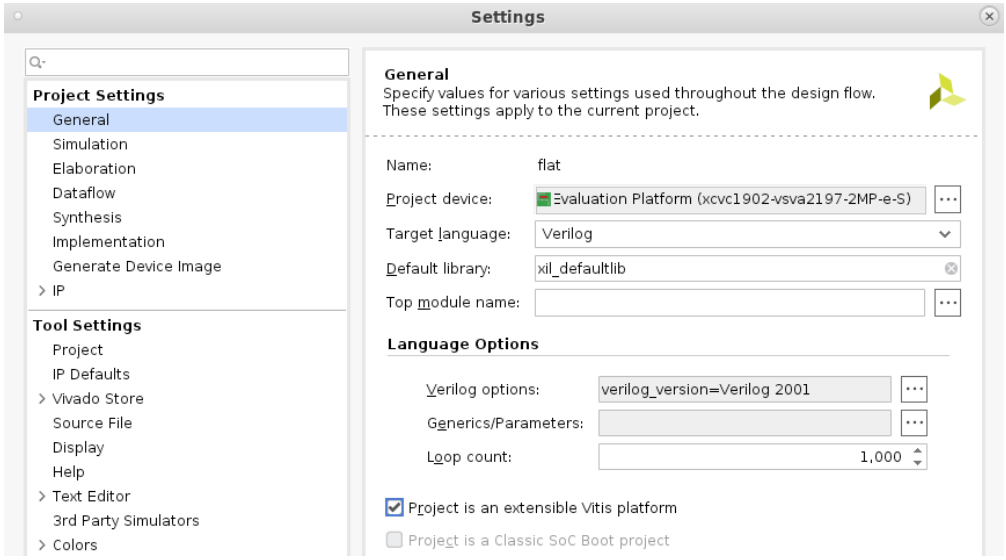
   After creating the platform, run synthesis to clean any issues related to hardware realization. Only use the flow for the Versal device platforms. Export the platform into `extensible.xsa` using following steps:

   a. Click on **Export Platform** under IP Integrator in the Flow Navigator.



   In Export Platform, select the **Presynthesis** option to generate the `extensible.xsa`.

   TIP: *If the **Export Platform** option is disabled, go to Project Settings and select **Project is an extensible Vitis Platform**.*



b. From the Tcl console, enter the command `write_hw_platform -f <filename>.xsa`to generate the extensible XSA.

2. In the Vitis tool, the exported extensible XSA from Vivado is used to compile the AI Engine graph, HLS kernels and run v++ link to export VMA. The steps are described below:

   - Compiles the AI Engine graph (`libadf.a`)
   - Compiles PL kernels (`.xo`)
   - Updates `system.cfg` file for connectivity
   - Runs `v++` linker with the `--export_archive` option

   In the Vitis Export to Vivado flow, the system linking process occurs as usual, but the automatic launch of Vivado synthesis and place and route is skipped. Instead, the `v++ --link --export_archive` command is used to generate the Vitis metadata archive (`.vma`) to export to the Vivado Design Suite.

```
v++ --link --export_archive --platform ../<>.xsa --config ../system.cfg \
<>.xo ./libadf.a -o <vma_file>.vma
```

> **IMPORTANT!** *The `--export_archive` command supports either `target=hw (--target hw)` only, or without specifying either `--target` or `-t`). An error is reported if `target = hw_emu` is used.*

3. Open the Vivado project after generating the `.vma` file from the Vitis tools. Import the `.vma` file in the Vivado project with the following Tcl procedure:

```
vitis::import_archive ./vma_path/<vma_file>.vma
```

   a. A new variant for the dynamic region block design container will be created by cloning the VMA's block design and made active.

b. Within the dynamic region block design, Vitis content is encapsulated within a level of hierarchy that the user should consider essentially read-only whenever they intend to use XRT after implementing the design in Vivado. See the Vitis Export Flow Guidelines and Limitations section for more details on how to preserve XRT metadata consistency while updating the design in Vivado.

c. The `.vma` region can be opened in Vivado. Alternatively, to rerun Vitis flows, you can re-export an extensible XSA after removing Vitis content via `vitis::remove_archive_hierarchy`.

4. After importing the `.vma` to the Vivado tools, design modifications can only be completed in Vivado. See the Vitis Export Flow Guidelines and Limitations section for more information.

   a. If there are changes to the `.vma` file, such as changes to the AI Engine design, PL kernels, or the PLIO boundary, go to Step 5 to regenerate the `.xsa` file and reiterate through the Vitis Export to Vivado flow.

   b. If the design changes are only related to Vivado, simulate the design, synthesize and implement the design to meet timing, and go to Step 6 for generating the `fixed.xsa`.

5. If the design requires changes related to the AI Engine, PL kernels, or updates in the PLIO boundary, these changes require updates to the linked system design, and regenerating the `.vma` in the Vitis tool. Thus, you must first remove the previously imported `.vma` from the Vivado project using one of two Tcl procedures:

   a. Use the `vitis::remove_archive_hierarchy` procedure to remove the imported `.vma` file while preserving any work done to the Vivado project after importing the `.vma`.

   b. Use the `vitis::remove_archive`procedure to restore the Vivado project to its state prior to importing the `.vma` file, removing both the `.vma` and any changes to the project.

   After removing `.vma` from the Vivado design, you can make any changes to the project. Vitis depends only upon the dynamic region block design and to potential connectivity points declared through PFM APIs. Update `system.cfg` to update the boundary connections. If there is a need to export the `extensible.xsa` for the second iteration or later from Vivado, use the `vitis::remove_archive` command and repeat Step 1 to export the `extensible.vma`; repeat Step 2 and 3 to export the VMA from Vitis and import VMA to Vivado respectively.

6. Once you have implemented your design, you can generate a `fixed.xsa` from the Vivado project by using the following command:

   ```
   write_hw_platform -fixed ./<fixed_xsa>.xsa
   ```

   You can use this XSA to perform application development for PetaLinux / Yocto or XRT-based apps development. You can use it to perform PS-based apps development through Vitis embedded software flow, or baremetal flow as it has been done traditionally.

You can use the fixed XSA from the preceding command to test the design on the hardware target only. If you want to run hardware emulation, remove the VMA by the `vitis::remove_archive` command and generate the extensible XSA, then take the extensible XSA through the Vitis flow.

7. After modifying the design, you can test the design on hardware or run hardware emulation. Follow the steps below to generate the fixed XSA for testing the design on hardware and hardware emulation.

   To test the design on hardware, first run synthesis and implementation on the design. Use the command `write_hw_platform -fixed ./<path to fixed.xsa>` to generate fixed XSA.

   To run hardware emulation, follow the steps below to generate the fixed XSA:

   a. Re-generate the output products
   b. Execute the command `launch_simulation -scripts only`
   c. Run `compile.sh`
   d. Run `elaborate.sh`
   e. Execute the command `write_hw_platform -fixed -include_sim_content <path to fixed xsa>`

   When you are ready to run the design on the hardware target (`-t=hw`) or hardware emulation target (`-t=hw_emu`), run `v++ --package` to generate the `.xclbin`. To generate the xclbin for hardware and hardware emulation, use the respective `fixed.xsa`. Set the `t= hw` or `hw_emu`, then provide the required software binaries and files to generate the `.xclbin`.

Vitis Export Flow Guidelines and Limitations

The `v++` compiler operates on a Vivado project that has been encapsulated in an extensible XSA built in Vivado. Conversely, the block design of the VMA is imported into a project as a design source that the user can continue to modify in Vivado.

In general, any modification to the Vivado project after `vitis::import_archive` that does not invalidate the contract between the imported design and the `.xclbin` metadata contained within the VMA is supported. The following table enumerates supported and prohibited operations.

Supported Vivado modifications after importing a VMA:

- Adding, removing, and reconfiguring IPs and RTL modules outside of and unconnected to the Vitis region hierarchy within the dynamic region block design.
- Add, removing, or changing connections unconnected to the Vitis region hierarchy within the dynamic region block design.
- Changing clock frequencies on clock wizard instances outside of the Vitis region hierarchy within the dynamic region block design.

- Changing QoS settings on `axi_noc` instances in the dynamic region block design.
- Adding `.xdc` constraints associated with any part of the design, including within the Vitis region hierarchy within the dynamic region block design.

Vivado modifications that require removing VMA hierarchy, updating Vitis kernels, connectivity and relink VMA export, and then reimport the VMA:

- Adding or deleting any IP instances or connections within the Vitis region hierarchy within the dynamic region block design.
- Adding or deleting connections between the dynamic region and the Vitis region hierarchy.
- Changes to the address map that modifies any address APERTURES or IP addressing in the Vitis region hierarchy within the dynamic region block design.

Current limitations of the Vitis Export flow include the following:

- Supported for Versal platforms only
- Project changes that modify the netlist path to the Vitis region hierarchy within the dynamic region block design.

Managing Vivado Synthesis, Implementation, and Timing Closure

TIP: *This topic requires an understanding of the Vivado Design Suite tools and design methodology as described in UltraFast Design Methodology Guide for FPGAs and SoCs (UG949), or the Versal Adaptive SoC Design Guide (UG1273).*

All the flows introduced in Chapter 3: Vitis Flows and Build Environment use the Vivado Design Suite for synthesis and implementation of the linked system design. The difference is how the user interacts with Vivado tools. In the Vitis Integrated Flow, this is controlled by command line or configuration file argument sections applying strategies and settings, while the Vitis Export to Vivado Flow uses traditional Vivado methods. This document covers how this is managed for the Vitis Integrated Flow.

Working with Vivado in the Vitis Integrated Flow

The Vitis Integrated Flow automatically launches the Vivado Design Suite to synthesize the linked system design, place and route the elements of the design, resolve timing, and generate the bitstream for the design. In most cases, the Vitis Integrated Flow completely abstract away the underlying process of synthesis and implementation of the hardware design. This removes the application developer from the typical hardware development process and the need to manage constraints such as logic placement and routing delays. The Vitis Integrated Flow automates much of the FPGA implementation process.

While automated, this flow does offer some opportunity for manual intervention. The process is broken down into a series of major steps that can be interrupted to enable customization when necessary. In some cases, you might want to exercise some control over the synthesis and implementation processes deployed by the Vitis linker, especially when large designs are being implemented. The Vitis Integrated Flow offers some control through specific options that can be specified in a `v++` configuration file, or from the command line. The following sections describe some of the methods you can use to control the Vivado synthesis and implementation results.

- Using the `--vivado` options to manage the Vivado tool.
- Using multiple implementation strategies to achieve timing closure on challenging designs.
- Using the `-to_step` and `-from_step` options to run the compilation or linking process to a specific step, perform some manual intervention on the design, and resume from that step.
- Interactively editing the Vivado project, and using the results for generating the FPGA binary.

Using the --vivado and --advanced Options

Using the `--vivado` option, as described in --vivado Options, and the `--advanced` option as described in --advanced Options, you can perform a number of interventions on the standard Vivado synthesis or implementation.

1. You can specify the strategy to be used by the Vivado tool during synthesis, implementation, or when generating reports during the build process. The strategy specified can be one of the standard tool strategies, or can be a custom-defined strategy that you have previously created in the Vivado tool. Use the `--vivado.prop` command as shown below.

   The original Tcl command to set the property on a run object looks like the following:

   ```
   set_property strategy Flow_AreaOptimized_medium [get_runs synth_1]
   ```

   The `v++` command is rewritten as shown below:

   - Synthesis Strategy:

     ```
     --vivado.prop run.synth_1.strategy=Flow_AreaOptimized_medium
     ```

- Implementation Strategy:

```
--vivado.prop run.impl_1.strategy=Performance_ExtraTimingOpt
```

- Report Strategy: Can be specified for synthesis or implementation runs.

```
--vivado.prop run.synth_1.report_strategy=MyCustom_Reports
```

```
--vivado.prop run.impl_1.report_strategy={Timing Closure Reports}
```

The command line is broken down as follows:

- `--vivado.prop` is the `v++` command-line option to assign properties to objects as described in --vivado Options.
- `run.<run_name>.strategy=<strategy_name>` to assign a `strategy` property (or `report_strategy`) to the specified synthesis or implementation run. Default run names are `synth_1` for synthesis or `impl_1` for implementation.
- Strategy names with spaces in them, such as `{Timing Closure Reports}` require braces or double-quotes to group the words as shown above.

> **TIP:** *You can also specify multiple implementation strategies to run as described in Running Multiple Implementation Strategies for Timing Closure.*

2. Pass Tcl scripts with custom design constraints or scripted operations.

   You can create Tcl scripts to assign XDC design constraints to objects in the design, and pass these Tcl scripts to the Vivado tools using the PRE and POST Tcl script properties of the synthesis and implementation steps. For more information on Tcl scripting, refer to the *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894). While there is only one synthesis step, there are a number of implementation steps as described in the *Vivado Design Suite User Guide: Implementation* (UG904). You can assign Tcl scripts for the Vivado tool to run before the step (PRE), or after the step (POST). The specific steps you can assign Tcl scripts to include the following: `SYNTH_DESIGN`, `INIT_DESIGN`, `OPT_DESIGN`, `PLACE_DESIGN`, `ROUTE_DESIGN`, `WRITE_BITSTREAM`.

> **TIP:** *There are also some optional steps that can be enabled using the `--vivado.prop run.impl_1.steps.phys_opt_design.is_enabled=1` option. When enabled, these steps can also have Tcl PRE and POST scripts.*

   An example of the Tcl PRE and POST script assignments follow:

```
--vivado.prop run.impl_1.STEPS.PLACE_DESIGN.TCL.PRE=/…/xxx.tcl
```

   In the preceding example a script has been assigned to run before the PLACE_DESIGN step. The command line is broken down as follows:

   - `--vivado` is the `v++` command-line option to specify directives for the Vivado tools.
   - `prop` keyword to indicate you are passing a property setting.
   - `run.` keyword to indicate that you are passing a run property.

- `impl_1.` indicates the name of the run.
- `STEPS.PLACE_DESIGN.TCL.PRE` indicates the run property you are specifying.
- `/.../xx.tcl` indicates the property value.

> **TIP:** *Both the `--advanced` and `--vivado` options can be specified on the `v++` command line, or in a configuration file specified by the `--config` option. The example above shows the command line use, and the following example shows the config file usage. Refer to Vitis Compiler Configuration File in the Vitis Reference Guide (UG1702) for more information.*

3. Setting properties on run, file, and fileset design objects.

   This is very similar to passing Tcl scripts as described above, but in this case you are passing values to different properties on multiple design objects. For example, to use a specific implementation strategy such as `Performance_Explore` and disable global buffer insertion during placement, you can define the properties as shown below:

   ```
   [vivado]
   prop=run.impl_1.STEPS.OPT_DESIGN.ARGS.DIRECTIVE=Explore
   prop=run.impl_1.STEPS.PLACE_DESIGN.ARGS.DIRECTIVE=Explore
   prop=run.impl_1.{STEPS.PLACE_DESIGN.ARGS.MORE OPTIONS}={-no_bufg_opt}
   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.ROUTE_DESIGN.ARGS.DIRECTIVE=Explore
   ```

   In the example above, the `Explore` value is assigned to the `STEPS.XXX.DIRECTIVE` property of various steps of the implementation run. Note the syntax for defining these properties is:

   ```
   <object>.<instance>.property=<value>
   ```

   Where:

   - `<object>` can be a design run, a file, or a fileset object.
   - `<instance>` indicates a specific instance of the object.
   - `<property>` specifies the property to assign.
   - `<value>` defines the value of the property.

   In this example the object is a run, the instance is the default implementation run, `impl_1`, and the property is an argument of the different step names, In this case the DIRECTIVE, IS_ENABLED, and {MORE OPTIONS}. Refer to --vivado Options for more information on the command syntax.

4. Enabling optional steps in the Vivado implementation process.

   The build process runs Vivado synthesis and implementation to generate the device binary. Some of the implementation steps are enable and run as part of the default build process, and some of the implementation steps can be optionally enabled at your discretion.

Optional steps can be listed using the `--list_steps` command, and include: `vpl.impl.power_opt_design`, `vpl.impl.post_place_power_opt_design`, `vpl.impl.phys_opt_design`, and `vpl.impl.post_route_phys_opt_design`.

An optional step can be enabled using the `--vivado.prop` option. For example, to enable PHYS_OPT_DESIGN step, use the following config file content:

```
[vivado]
prop=run.impl_1.steps.phys_opt_design.is_enabled=1
```

When an optional step is enabled as shown above, the step can be specified as part of the `-from_step`/`-to_step` command as described below in *Running --to_step or --from_step*, or enable a Tcl script to run before or after the step as described in --linkhook Options.

5. Passing parameters to the tool to control processing.

   The `--vivado` option also allows you to pass parameters to the Vivado tools. The parameters are used to configure the tool features or behavior prior to launching the tool. The syntax for specifying a parameter uses the following form:

   ```
   --vivado.param <object><parameter>=<value>
   ```

   The keyword `param` indicates that you are passing a parameter for the Vivado tools, rather than a property for a design object. You must also define the `<object>` it applies to, the `<parameter>` that you are specifying, and the `<value>` to assign it.

   The following example project indicates the current Vivado project, `writeIntermedateCheckpoints`, is the parameter being passed and the value is 1, which enables this boolean parameter.

   ```
   --vivado.param project.writeIntermediateCheckpoints=1
   ```

6. Managing the reports generated during synthesis and implementation.

   IMPORTANT! *You must also specify `--save-temps` on the `v++` command line when customizing the reports generated by the Vivado tool to preserve the temporary files created during synthesis and implementation, including any generated reports.*

   You might also want to generate or save more than the standard reports provided by the Vivado tools when run as part of the Vitis tools build process. You can customize the reports generated using the `--advanced.misc` option as follows:

   ```
   [advanced]
   misc=report=type report_utilization name
   synth_report_utilization_summary steps {synth_design} runs {__KERNEL__}
   options {}
   misc=report=type report_timing_summary name
   impl_report_timing_summary_init_design_summary steps {init_design} runs
   {impl_1} options {-max_paths 10}
   misc=report=type report_utilization name
   impl_report_utilization_init_design_summary steps {init_design} runs
   {impl_1} options {}
   misc=report=type report_control_sets name
   impl_report_control_sets_place_design_summary steps {place_design} runs
   ```

```
{impl_1} options {-verbose}
misc=report=type report_utilization name
impl_report_utilization_place_design_summary steps {place_design} runs
{impl_1} options {}
misc=report=type report_io name impl_report_io_place_design_summary
steps {place_design} runs {impl_1} options {}
misc=report=type report_bus_skew name
impl_report_bus_skew_route_design_summary steps {route_design} runs
{impl_1} options {-warn_on_violation}
misc=report=type report_clock_utilization name
impl_report_clock_utilization_route_design_summary steps {route_design}
runs {impl_1} options {}
```

The syntax of the command line is explained using the following example:

```
misc=report=type report_bus_skew name
impl_report_bus_skew_route_design_summary steps {route_design} runs
{impl_1} options {-warn_on_violation}
```

- **`misc=report=`:** Specifies the `--advanced.misc` option as described in --advanced Options, and defines the report configuration for the Vivado tool. The rest of the command line is specified in name/value pairs, reflecting the options of the `create_report_config` Tcl command as described in *Vivado Design Suite Tcl Command Reference Guide* (UG835).
- **`type report_bus_skew`:** Relates to the `-report_type` argument, and specifies the type of the report as the `report_bus_skew`. Most of the `report_*` Tcl commands can be specified as the report type.
- **`name impl_report_bus_skew_route_design_summary`:** Relates to the `-report_name` argument, and specifies the name of the report. Note this is not the file name of the report, and generally this option can be skipped as the report names will be auto-generated by the tool.
- **`steps {route_design}`:** Relates to the `-steps` option, and specifies the synthesis and implementation steps that the report applies to. The report can be specified for use with multiple steps to have the report regenerated at each step, in which case the name of the report will be automatically defined.
- **`runs {impl_1}`:** Relates to the `-runs` option, and specifies the name of the design runs to apply the report to.
- **`options {-warn_on_violation}`:** Specifies various options of the `report_*` Tcl command to be used when generating the report. In this example, the `-warn_on_violation` option is a feature of the `report_bus_skew` command.

> **IMPORTANT!** *There is no error checking to ensure the specified options are correct and applicable to the report type specified. If you indicate options that are incorrect the report will return an error when it is run.*

Associating an ELF File with MicroBlaze Processor

You can use the following steps to associate an ELF file with a MicroBlaze™ processor in your design. Associating the ELF file configures a memory target, such as a set of Block RAMs. The information that is needed for ELF file association includes:

- The location of the ELF file to be loaded
- The address space accessible via a master interface to a memory location, which the ELF file will be stored
- The mapped peripheral within that address space representing the memory where ELF file will be stored and from where it will be accessed at run time

IMPORTANT! *This flow requires you to have access to the level of design hierarchy containing the MicroBlaze processor, and an existing ELF file.*

This process uses SCOPED_TO_REF and SCOPED_TO_CELLS properties on the MicroBlaze processor itself, and not to the Block RAMs that are the actual target of the ELF file data.

You can associate the ELF file to the MicroBlaze processor during the `v++ --link` process using the `--advanced.param <param_name>=<param_value>` command as described in --advanced Options. An example for a config file is shown below.

```
[advanced]
param=hw_emu.post_sim_settings=<file_path>/link.tcl
```

The `link.tcl` should add the ELF file to the Vivado Design Suite project, exclude it for use in simulation, and associate it with the MicroBlaze processor, as shown in the example below.

```
add_files <file_path>/executable.elf
set_property used_in_simulation 0 [get_files <file_path>/executable.elf]
set_property SCOPED_TO_REF base_microblaze_design [get_files -all \
-of_objects [get_fileset sources_1] {<file_path>/executable.elf}]
set_property SCOPED_TO_CELLS { microblaze_0 } \
[get_files -all -of_objects [get_fileset sources_1] {<file_path>/
executable.elf}]
```

This information will be used to generate a BMM file which will be used by programs such as `data2mem` to generate a `.mem` file that will populate the Block RAMs that are generated from the `block_memory_generator`.

Running Multiple Implementation Strategies for Timing Closure

For challenging designs, it can take multiple iterations of Vivado implementation using multiple different strategies to achieve timing closure. This topic shows you how to launch multiple implementation strategies at the same time in the hardware build (`-t hw`), and how to identify and use successful runs to generate the device binary and complete the build.

As explained in --vivado Options, the `--vivado.impl.strategies` command enables you to specify multiple strategies to run in a single build pass. The command line would look as follows:

```
v++ --link -s -g -t hw --platform xilinx_zcu102_base_202010_1 -I . \
--vivado.impl.strategies "Performance_Explore,Area_Explore" -o
kernel.xclbin hello.xo
```

In the example above, the `Performance_Explore` and `Area_Explore` strategies are run simultaneously in the Vivado build to see which returns the best results. You can specify the `ALL` to have all available strategies run within the tool.

> **IMPORTANT!** *Running ALL implementation strategies might launch 30 or more runs in the Vivado tool, including any user-defined strategies stored in your home directory (`~/.Xilinx/Vivado/202X.X/strategies`). This can be a tremendous drain on resources, and is not advised. You can prevent this by defining specific 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.*

You can also determine this option in a configuration file in the following form:

```
#Vivado Implementation Strategies
[vivado]
impl.strategies=Performance_Explore,Area_Explore
```

The Vitis compiler automatically picks the first completed run results that meets timing to proceed with the build process and generate the device binary. However, you can also direct the tool to wait for all runs to see the result of all strategies. This would require the use of the `{}ALL_IMPL{}` macro to apply custom settings to all runs and the `multiStrategiesWaitOnAllRuns` directive to see the result of all strategies:

```
[advanced]
#param=compiler.multiStrategiesWaitOnAllRuns=1

[vivado]

impl.strategies=ALL
prop=run.{}ALL_IMPL{}.STEPS.PLACE_DESIGN.TCL.PRE=../../vpp_cfg/
place_design_pre.tcl
prop=run.{}ALL_IMPL{}.STEPS.ROUTE_DESIGN.TCL.PRE=../../vpp_cfg/
route_design_pre.tcl
```

`compiler.multiStrategiesWaitOnAllRuns=0` represents the default behavior. If you want `v++` to wait for all runs to complete, which will get their report files, change that parameter value to 1. This includes an overview of the implementation results, in addition to a Timing Summary report. Seeing all results will give you an indication on how hard it is to close timing for the tool for the all strategies that are allowed to run to completion. You can use this feature to review the different strategies and results.

You can also manually review the results of all implementation strategies after they have completed. Then, use the results of any of the implementation runs by using the `--reuse_impl` option as described in Using --to_step and Launching Vivado Interactively.

Using --to_step and Launching Vivado Interactively

The Vitis compiler lets you stop the build process after completing a specified step (`--to_step`), manually intervene in the design or files in some way, and then continue the build by specifying a step the build should resume from (`--from_step`). The `--from_step` directs the Vitis compiler to resume compilation from the step where `--to_step` left off, or some earlier step in the process. The `--to_step` and `--from_step` are described in v++ Command in the *Vitis Reference Guide* (UG1702).

IMPORTANT! *The `--to_step` and `--from_step` options are sequential build options that require you to use the same project directory when launching `v++ --link --from_step` as you specified when using `v++ --link --to_step`.*

The Vitis compiler also provides a `--list_steps` option to list the available steps for the compilation or linking processes of a specific build target. For example, the list of steps for the link process of the hardware build can be found by:

```
v++ --list_steps --target hw --link
```

This command returns a number of steps, both default steps and optional steps that the Vitis compiler goes through during the linking process of the hardware build. Some of the default steps include: `system_link`, `vpl`, `vpl.create_project`, `vpl.create_bd`, `vpl.generate_target`, `vpl.synth`, `vpl.impl.opt_design`, `vpl.impl.place_design`, `vpl.impl.route_design`, and `vpl.impl.write_bitstream`.

Optional steps include: `vpl.impl.power_opt_design`, `vpl.impl.post_place_power_opt_design`, `vpl.impl.phys_opt_design`, and `vpl.impl.post_route_phys_opt_design`.

TIP: *An optional step must be enabled before specifying it with `--from_step` or `--to_step` as previously described in Using the --vivado and --advanced Options.*

Launching the Vivado IDE for Interactive Design

***Note*:** The Packaging for Vitis Export to Vivado Flow in the *Embedded Design Development Using Vitis* (UG1701) is the preferred method.

With the `--to_step` command, you can launch the build process to Vivado synthesis and then start the Vivado IDE on the project to manually place and route the design. To perform this you would use the following command syntax:

```
v++ --target hw --link --to_step vpl.synth --save-temps --platform
<PLATFORM_NAME> <XO_FILES>
```

TIP: *As shown in the example above, you must also specify `--save-temps` when using `--to_step` to preserve any temporary files created by the build process.*

This command specifies the link process of the hardware build, runs the build through the synthesis step, and saves the temporary files produced by the build process.

You can launch the Vivado tool directly on the project built by the Vitis compiler using the `--interactive` command. This opens the Vivado project found at `<temp_dir>/link/vivado/vpl/prj` in your build directory, letting you interactively edit the design:

```
v++ --target hw --link --interactive impl --save-temps --platform
<PLATFORM_NAME> <XO_FILES>
```

When invoking the Vivado IDE in this mode, you can open the synthesis or implementation runs to manage and modify the project. You can change the run details as needed to close timing and try different approaches to implementation. You can save the results to a design checkpoint (DCP), or generate the project bitstream (`.bit`) to use in the Vitis environment to generate the device binary.

After saving the DCP from within the Vivado IDE, close the tool and return to the Vitis environment. Use the `--reuse_impl` option to apply a previously implemented DCP file in the `v++` command line to generate the `xclbin`.

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

The following command completes the linking process by using the specified DCP file from the Vivado tool to create the `project.xclbin` from the input files.

```
v++ --link --platform <PLATFORM_NAME> -o'project.xclbin' project.xo --
reuse_impl ./_x/link/vivado/routed.dcp
```

You can also use a bitstream file generated by the Vivado tool to create the `project.xclbin`:

```
v++ --link --platform <PLATFORM_NAME> -o'project.xclbin' project.xo --
reuse_bit ./_x/link/vivado/project.bit
```

***Note*:** The `project.bit` used for `--reuse_bit` is a partial bit and not a full bit.

Additional Vivado Options

Some additional switches that can be used in the `v++` command line or config file include the following:

- `--export_script`/`--custom_script` edit and use Tcl scripts to modify the compilation or linking process.
- `--remote_ip_cache` specify a remote IP cache directory for Vivado synthesis.
- `--no_ip_cache` turn off the IP cache for Vivado synthesis. This causes all IP to be re-synthesized as part of the build process, scrubbing out cached data.

Integrating the System

This section describes how the hardware design and software applications can be combined to form a complete integrated system. A prerequisite for software application development is the information about the hardware design and their corresponding address map, which is the hardware specification. The software applications are compiled to binaries, then packaged together with hardware configuration data into device images, which also contain boot instructions.

The packaging process consists of two steps:

1. The first step involves generating loadable images from the Vivado hardware design and AI Engine handoffs into a binary container using Vitis packager. Also, in this step, a draft BIF file is created.
2. In the second step, the binary containers and software executables are collected and assembled into a delivery package which contains instructions for the boot process loading. This delivery package can be an SD card, QSPI flash image, or other similar type of package. This second step can be performed either using Vitis packager or Bootgen.

During the system integration packaging process, several different types of files are used as listed in the following table:

Table 31: **System Integration Packaging Process Files**

Input File Type	Name	Description	Applicable Devices
XSA	Support Archive	Primary design handoff archive file from Vivado. The XSA file can be fixed or extensible.	Versal adaptive SoC / Zynq / Zynq UltraScale+
BIF	Boot Image Format	File that is used by Bootgen to determine how to generate boot images, configure them in a PDI file.	Versal adaptive SoC / Zynq / Zynq UltraScale+
PDI	Programmable Device Image	Output of Bootgen. It is the image file containing bootloader,, descriptions and partitions related to processing input data files (ELF, PL configuration and other binary files).	Versal adaptive SoC
XCLBIN	XCLBIN	Enhanced PDI container with metadata on PL kernels and AI Engine. XCLBIN is exclusive to XRT API usage on the Linux OS.	Versal adaptive SoC / Zynq / Zynq UltraScale+
CDO	Configuration data objects	List of commands executed in sequence to configure various components in the system.	Versal adaptive SoC

Table 31: **System Integration Packaging Process Files** *(cont'd)*

Input File Type	Name	Description	Applicable Devices
libadf.a	AI Engine graph library	• Output archive: `libadf.a` is the primary output of compiling an AI Engine graph. It contains the compiled program for the AI Engine. It includes both CDO and Elf files. • CDOs (Compiled Device Objects): These objects define the setup and configuration of the AI Engine, including its resources and topology. • ELFs (Executable and Linkable Format): These files contain the program instructions that the AI Engine's tile processors execute. • Partitions: When enabled, the AI Engine allows for dividing the workload into partitions. Each partition targets a subset of the columns of the AI Engine array and generates its own `libadf` file. • libadf files: These files contain the compiled program and configuration for each partition. All of these files must be specified when packaging a design containing partitions. See Compiling AI Engine Graph for Independent Partitions in the *AI Engine Tools and Flows User Guide* (UG1076)	Versal adaptive SoC
FSBL	First stage bootloader	Image for PL bitstream, code and data to start the initial design. This can bring up the entire design or pass on to a second bootloader to finalize the boot.	Zynq / Zynq UltraScale+
DTSI	Devicetree system includes	Adding user settings to override devicetree defaults like MAC address, UART baud rates, etc on hardware devices.	Versal adaptive SoC / Zynq / Zynq UltraScale+

Table 31: **System Integration Packaging Process Files** *(cont'd)*

Input File Type	Name	Description	Applicable Devices
qemu_args.txt / pmc_args.txt	QEMU command arguments file	Command line arguments used when launching QEMU as the DTB for emulation differs from Linux device tree.	Versal adaptive SoC

The package process varies depending on the domain selected, as this has an impact on the boot order and how the hardware specification is extracted for the host application. The following sections discuss these variants.

The first package step require using Vitis package, while the second step can use either Vitis package or Bootgen. It's recommended to be familiar with the boot components and how the BIF setup the boot order described in Software Platform. Advanced users should consider using *Bootgen User Guide* (UG1283) for details and custom packaging control options. The sections below describe using the Vitis package.

***Note*:** Packaging for HW emulation require the design to be linked using `--target hw_emu`.

Packaging Process for Bare-metal and RTOS Applications

Bare-metal applications interact with the hardware through registers defined in the hardware specification and low level drivers. Additionally drivers for frequently used services like ethernet, file handling, FPGA management, etc., can added via BSP (board support package).

The hardware specification is extracted from the fixed XSA, and when creating the Vitis platform component for bare metal or RTOS domains, the `xparameters.h` file is generated. See Board Support Package Settings Page in the *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400) on how to add and configure Bare-metal and RTOS domains. If changes to hardware affect the hardware specification, the BSP and `xparameters.h` needs to be regenerated.

Alternatively, PetaLinux tools for multiconfig can be used to regenerate BSP: Building multiconfig Applications in the *PetaLinux Tools Documentation: Reference Guide* (UG1144).

First, generate a loadable PDI and extract the AI Engine CDO below.

```
v++ -p -s -f <fixed.xsa> <libadf.a> --temp-dir <temp_dir> --save-temps
```

IMPORTANT! *If Bootgen is used for step 2, it is required to use `--save-temps` option as shown above to preserve the files needed by Bootgen.*

Next, integrate the hardware and software platform using Vitis packager:

```
v++ -p -s -f <fixed.xsa> <libadf.a> --package.generate_sd_card --
package.sd_file <pdi, elf, xclbin, etc.> --package.sd_dir <outdir>
```

Full details on command line options for `v++` are described in v++ Command in the *Vitis Reference Guide* (UG1702).

Packaging Process for Linux Applications

There are two ways that Linux applications can use drivers: the standard driver approach with the system device tree, or the XRT API.

With the XRT API, drivers executed in user space query address information from the XCLBIN file. When design changes affect hardware specification registers on Vitis managed components, the XCLBIN file is automatically updated during linking and packaging.

Refer to Software Platform for system device tree drivers that need to be regenerated using the fixed XSA.

User specific settings to the Linux drivers are adjusted through DTSI, see Device Tree Configuration in the *PetaLinux Tools Documentation: Reference Guide* (UG1144).

Once the device tree and configurations are set up, packaging is done similar to Bare-metal and RTOS packaging.

Packaging and Boot Configuration using Bootgen

After the first packaging step, the AMD Vitis™ packager collects and assembles the binaries and executables to boot and run a design on AMD SoC devices using the BIF file. Details on device specific boot and configuration are available in the following user guides:

- Versal adaptive SoC: Boot and Configuration in the *Versal Adaptive SoC System Software Developers Guide* (UG1304)
- Zynq UltraScale+: Validate NoC DRCs in the *Versal Adaptive SoC Hardware, IP, and Platform Development Methodology Guide* (UG1387)
- Zynq: Boot and Configuration in the *Versal Adaptive SoC System Software Developers Guide* (UG1304)

Packaging Specifies for Versal Designs

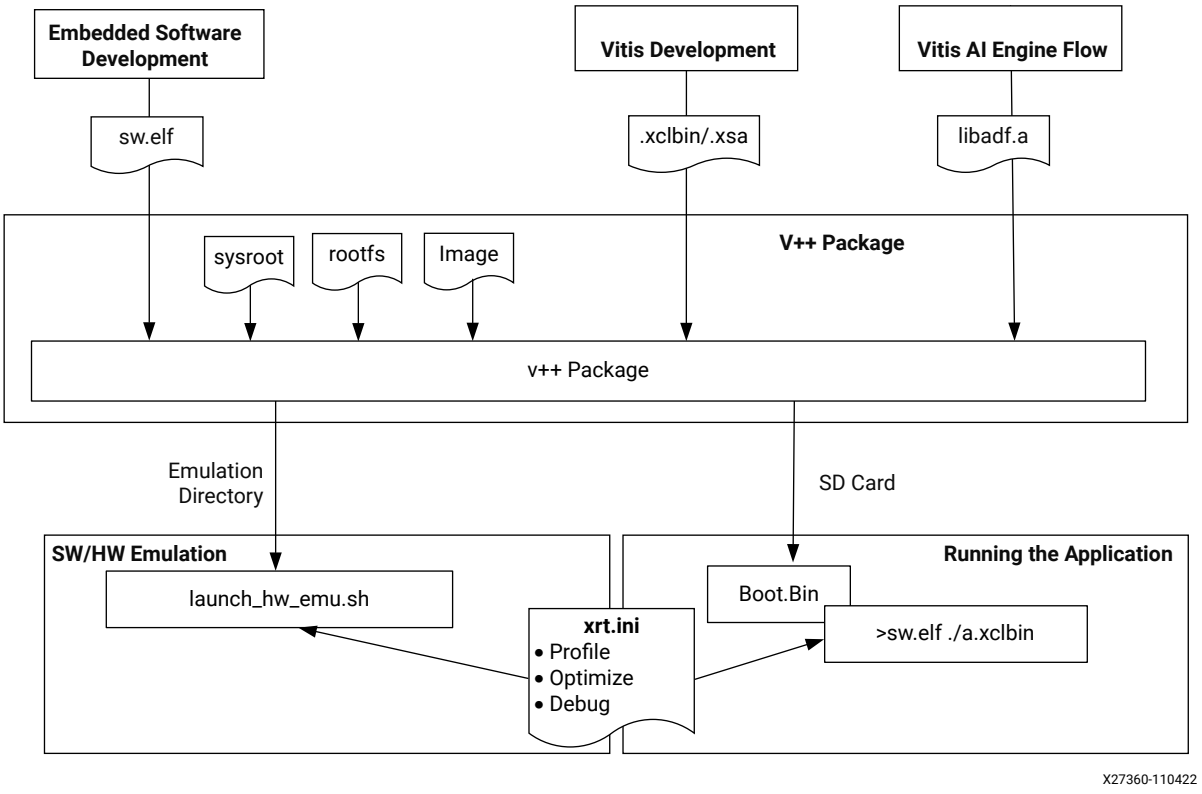
The Versal AI Engine compiler generates output in the form of a library file, `libadf.a`, which contains ELF and CDO files, as well as tool-specific data and metadata, for hardware and hardware emulation flows. To create a loadable image binary, this data must be combined with PL-based configuration data, boot loaders, and other binaries. The Vitis packager performs this function, combining information from `libadf.a` and the Vitis linker generated XSA file.

For Versal adaptive SoCs, the programmable device image (PDI) file is used to boot and program the hardware device. For hardware emulation the `--package` command adds the PDI, EMULATION_DATA sections and the XSA file, and outputs an XCLBIN file. For hardware builds, the package process creates an XCLBIN file containing ELF files and graph configuration data objects (CDOs) for the AI Engine application. The XCLBIN file includes the following information:

- **PDI:** Programming information for the AI Engine array
- **Debug data:** Debug information when included in the build
- **Memory topology:** Defines the memory resources and structure for the target platform
- **IP layout:** Defines layout information for the implemented hardware design
- **Metadata:** Various elements of platform meta data to let the tool load and run the XCLBIN file on the target platform

Packaging for Vitis Flow

Figure 34: **Linking the System Design**



The Vitis `v++ --package` command generates SD card and other Flash images required for booting the system, in addition to the `.xclbin` device binary from the `.xsa` generated for Versal devices. The `v++ --package` step, or `-p` option, packages the final system at the end of the `v++` compile, link, and package process. As described in Packaging for Vitis Export to Vivado Flow, this is a required step for all Versal platforms, including AI Engine platforms, and embedded processor platforms.

A fixed `.xsa` can be used to create custom boot, and software platform images as described in the *Versal Adaptive SoC System Software Developers Guide* (UG1304). However, the --package Options in the *Vitis Reference Guide* (UG1702) let you package your design and define various files required for booting and configuring the AMD device for use during emulation or in production systems. It collects the various elements to create an SD card, or other means to program the device, to define the operating system, and to load the application and kernel code.

In the Vitis unified IDE, the package process is automated and the tool creates the required files based on the build target, platform, and OS. However, in the command line flow, you must specify the Vitis packaging command (`v++ --package`) with the correct options for the job.

After packaging the design the AMD Vitis™ compiler generates a `v++.package_summary` that includes the packaging command and log file. The summary file can be viewed in Analysis view of the Vitis analyzer alongside the compile, link, and run summaries as explained in Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702).

Packaging for Vitis Export to Vivado Flow

For AMD Zynq™ UltraScale+™ MPSoC and AMD Zynq™ 7000 embedded platforms, the `--package` command line is shown below:

```
v++ --package -t [hw_emu | hw] --platform <platform> input.xclbin [ -o
output.xclbin ]
```

Note: If the output option (`-o`) is not specified, the tool creates an output file with the default name of `a.xclbin`.

For Versal devices the `v++ --link` command creates an `.xsa` file instead of the `.xclbin` file. In this case the `.xsa` must be provided to the package process to generate the `.xclbin` file. The `--package` command line for Versal devices is as follows:

```
v++ --package -t [hw_emu | hw] --platform <platform> input.xsa [ -o
output.xclbin ]
```

In the case of Versal platforms, the package process takes the `.xsa` file generated during the `v++ --link` command, and also takes the `libadf.a` file produced by the `aiecompiler` command and integrates it into the output device binary.

The `--package` command has a range of options for use with the different platforms and build targets supported by the Vitis tools. In the Vitis IDE, the package process is automated and the tool creates the required files as needed. However, in the command line flow, you must specify the `v++ --package` command or add the `[package]` tag in the `config` file with the right options for the job. The following is an example command for hardware emulation:

```
v++ --package --config package.cfg ./aie_graph/libadf.a \
./project.xsa -o aie_graph.xclbin
```

Where, the `--config package.cfg` option specifies a configuration file for the Vitis compiler with the various options specified for the package process. The following is an example configuration file:

```
platform=xilinx_vck190_base_202520_1
target=hw_emu
save-temps=1

[package]
boot_mode=sd
out_dir=./emulation
rootfs=<downloaded_common_image>/rootfs.ext4
image_format=ext4
kernel_image=<downloaded_common_image>/Image
sd_file=host.exe
```

For hardware emulation, the command takes the `.xclbin` or `.xsa` file as input, produces a script to launch emulation (`launch_hw_emu.sh`). To specify the output folder, use the `--package.out_dir` option.

For Linux, and with fixed `.xsa`, it is required to add `--package.dtb=<path to system.dtb>`, `--package.bl31_elf=<path to bl31.elf>`, and `--package.uboot=<path to linux u-boot.elf>`.

Additional files required for running the application, such as data files needed as input or to validate the application, or the `xrt.ini` file for profiling and debug, must be included in the output files, and can be transferred individually using the `--package.sd_file` option, or transferred as a directory using the `sd_dir` option as explained in --package Options in the *Vitis Reference Guide* (UG1702).

IMPORTANT! *For Linux a boot recipe is mandatory to automate the boot process. This is added by using `--package.sd_file <path to boot.scr>`.*

For hardware builds, the `--package` command creates an `sd_card` folder, or the `QSPI.img` depending on the boot mode specified with the `--package.boot_mode` option.

TIP: *For bare metal ELF files running on PS cores, you should also add the following option to the command line:*

```
--package.ps_elf <elf>,<core>
```

The package command creates an output folder called `sd_card`, that contains all of the files needed to run hardware emulation for the application, modeling the boot process of an `sd_card`. For hardware builds, it contains the files required for creating an SD card to boot the device. The following is an example of the packaging output for hardware emulation:

```
|-- BOOT_bh.bin    //Boot header
|-- BOOT.BIN             //Boot File
|-- boot_image.bif
|-- launch_hw_emu.sh       //Hardware emulation launch script
|-- libadf                  //AIE emulation data folder
|   `-- cfg
|       |-- aie.control.config.json
|       |-- aie.partial.aiecompile_summary
|       |-- aie.shim.solution.aiesol
|       |-- aie.sim.config.txt
|       `-- aie.xpe
|-- plm.bin                 //PLM boot file
|-- pmc_args.txt            //PMC command argument specification file
|-- pmc_cdo.bin             //PMC boot file
|-- qemu_args.txt           //QEMU command argument specification file
|-- sd_card
|   |-- BOOT.BIN
|   |-- boot.scr
|   |-- aie_graph.xclbin
|   |-- host.exe
|   |-- Image
|-- sd_card.img
`-- sim                      //Vivado simulation folder
```

After creating the `sd_card` folder for the hardware build, copy the contents to an SD card to create the boot image for your physical device.

Note: On Windows, you must use a third-party tool, such as Etcher, to write on the SD card for use in booting the AMD device.

Packaging Segmented Configuration

The packaging process for segmented configuration has two steps. The first step is for extracting the loading and configuration meta-data for PL and AI Engine. The second step is for combining that with the PS and NoC DDRMC boot and configuration. For Versal AI Edge Series Gen 2, Versal Prime Series Gen 2, and Versal Premium Series Gen 2, segmented configuration is mandatory and `v++ --package` automatically identifies it require to create both artifacts. For first generation Versal devices, you need to run `v++ --package` a second time to create a SD card.

Note: If EDF is used, the PDI and `dtbo` needs to be added to the EDF `wic` image, see AMD Embedded Development Framework.

Note: Segmented configuration is primarily for separating the boot process and loading the PL/AIE running Linux on hardware. For hardware emulation, the boot and load is performed in one step, see Packaging the System in HW Emulation.

The PL/AIE package require these inputs:

```
v++ -p -s \
  -f <fixed_design>.xsa \
  <graph_name>.libadf \
  --package.out_dir <path_to_sdcard> \
  -o <design_name>.xclbin
```

To create a SD card combining the boot artifacts, Linux, host application and the prepared PL/AIE, run `v++` package a second time using the following inputs.

```
v++ -p -t hw \
  -f <fixed_design>.xsa \
  <graph_name>.libadf \
  --package.generate_sdcard \
  --package.defer_aie_run \
  --package.dtb <vitis_workspace>/<pfm_name>/export/<pfm_name>/sw/boot/
system.dtb \
  --package.image_format <ext4|fat32> \
  --package.bl31_elf <path_to_linux>/bl31.elf \
  --package.uboot <path_to_linux>/uboot.elf \
  --package.sd_file <path_to_linux>/Image \
  --package.sd_file <path_to_linux>/boot.scr \
  --package.sd_file <host_app>.exe \
  --package.sd_file <design_name>.xclbin \
  --package.sd_file <path_to_sdcard>/<PL_PDI_NAME>.pdi \
  --package.sd_file <path_to_sdcard>/<PL_PDI_NAME>.dtbo \
  --package.out_dir <path_to_sdcard> \
  -o <design_name>.xclbin
```

On hardware, load the second stage after the PS/NoC boot procedure completes using the `fpgautil` tool.

```
fpgautil -b <PL_PDI_NAME>.pdi -o <PL_PDI_NAME>.dtbo
```

The `PL.pdi` represents the PL and AI Engine design. `PL.dtbo` represents the device tree blob overlay containing the meta-data information on how to configure and setup the PL. XRT queries the XCLBIN file for controlling the PL and AIE graph when the host application executes.

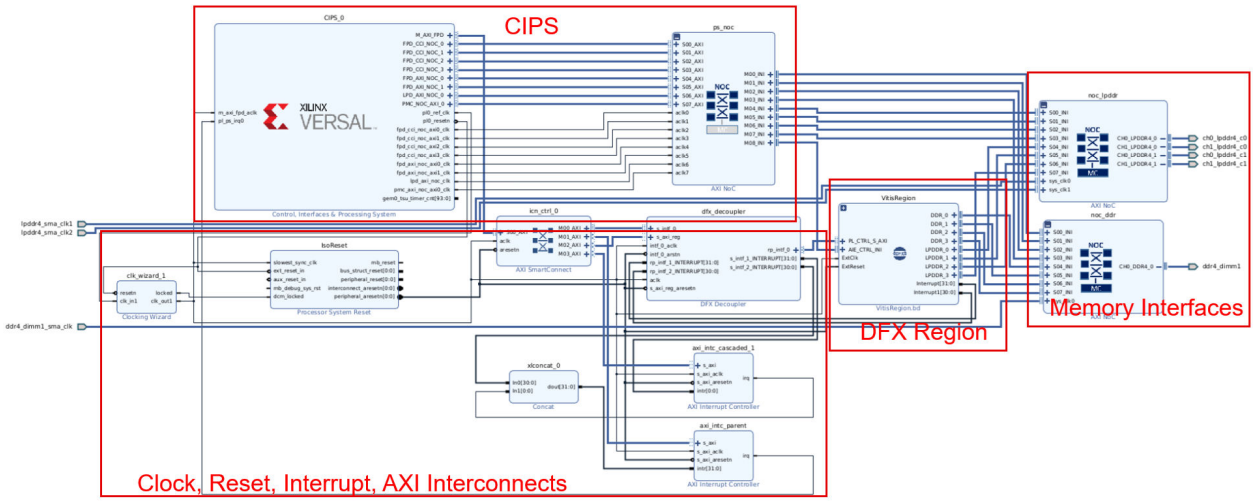
Packaging for DFX Platforms

Vitis Dynamic Function eXchange (DFX) platforms contain a static region and DFX regions.

- **Static Region:** Hardened block that is loaded in system booting and is not reconfigurable at runtime.
- **DFX region:** A reconfigurable partition (RP) that is implemented by the Vivado IP integrator block design container (BDC). This partition can be reprogrammed repeatedly during runtime to dynamically trade out one function, or reconfigurable module (RM), for another.

AMD provides base DFX platforms, such as `xilinx_zcu102_base_dfx_202520_1` and `xilinx_vck190_base_dfx_202520_1`. These are single RP platforms, meaning that it contains a static region and only one DFX region in which RMs can be dynamically exchanged at runtime. In the case of `xilinx_vck190_base_dfx_202520_1`, the RM contains an AI Engine array and PL kernels. When loading the RM the entire AI Engine array and all PL kernels are loaded at the same time. The following picture shows the block design (BD) of the platform.

Figure 35: **Block Design of the Base DFX Platform**

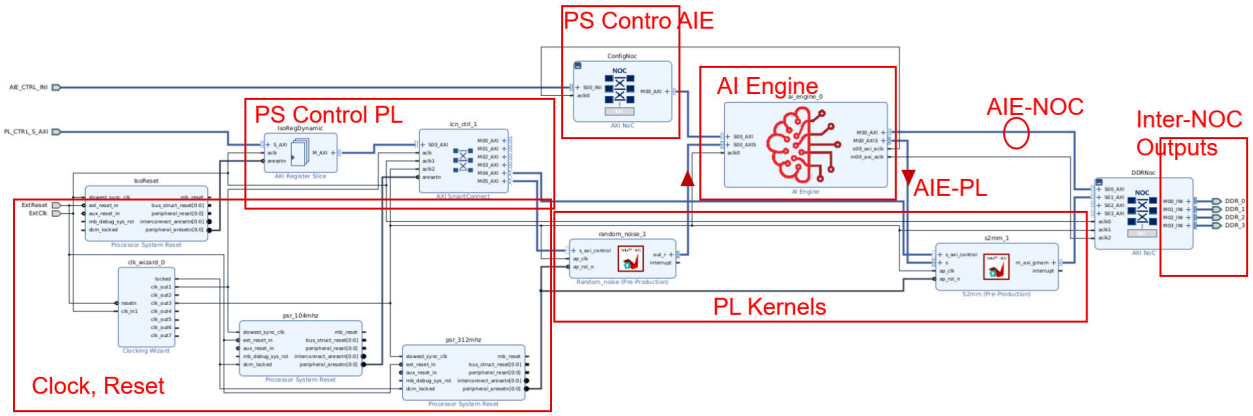


The static partition contains the following:

- CIPS and PS NoC
- Clock, Reset, Interrupt, and AXI interconnections
- NoC Interface and DDR memory controller

The reconfigurable module (RM) to load into the DX region of the `xilinx_vck190_base_dfx_202520_1` platform contains an AI Engine array and PL kernels that are linked together by the Vitis linker (`v++ -l`). Refer to *Versal Adaptive SoC Hardware, IP, and Platform Development Methodology Guide* (UG1387). The following picture shows an example of the RM.

Figure 36: **Block Design of the RM**



The RM, with interfaces to the static region, includes:

- Clock and Reset
- Inter-NoC Interface (INI) outputs to general memory
- INI input from PS to control AI Engine
- AXI Interface from PS to control PL kernels

The contents of the RM can include:

- AI Engine
- PL kernels
- AXI-NoC IP
- Clock and reset modules
- AXI Interconnect module
- ILA
- FIFO, Data Width Converter (DWC), and CDC modules

The `v++ --compile` and `v++ --link` commands are the same for both a DFX platform and a non-DFX platform. However, the `v++ --package` command is different for the DFX platform than the non-DFX platform, and is different between the hardware and hardware emulation targets as described below.

Hardware Packaging and Deployment

For a `hw` target the `v++ --package` command for the DFX platform requires two steps:

1. The first step generates an `xclbin` file that contains the RM PDI that will be loaded at runtime.

```
v++ -p -t hw -f xilinx_vck190_base_dfx_202520_1
  --package.defer_aie_run \
  -o rm.xclbin \
  ${XSA} \
  libadf.a
```

2. The second step packages the RM `xclbin` file (`rm.xclbin`) into the SD card that can be read by the host program.

```
v++ -p -t hw -f xilinx_vck190_base_dfx_202520_1
  --package.rootfs ${ROOTFS}
  --package.kernel_image ${IMAGE} \
  --package.boot_mode=sd \
  --package.image_format=ext4 \
  --package.sd_dir data \
  --package.sd_file ${HOST_EXE} \
  --package.sd_file rm.xclbin
```

When running the hardware, after Linux has started type the following command to load the RM:

```
cd /run/media/mmcblk0p1
./host.exe rm.xclbin
```

***Note*:** If the XRT detects that the same XCLBIN has been downloaded already, it will not download the XCLBIN again by default. Instead, it loads metadata from the `xclbin` file only.

To make sure XCLBIN is downloaded and the device is clean every time, set `xrt.ini` to enforce XCLBIN download to the device as follows:

1. Add following configuration to the `xrt.ini` file:

```
[Runtime]
force_program_xclbin=true
```

2. Add the `xrt.ini` file to the SD card during the package process: `--package.sd_file xrt.ini`

This ensures the XRT downloads the XCLBIN to device every time.

Hardware Emulation Packaging and Deployment

For `hw_emu` target the `v++ --package` command can be:

```
v++ -p -t hw_emu -f xilinx_vck190_base_dfx_202520_1 \
  --package.defer_aie_run \
  --package.rootfs ${ROOTFS} \
  --package.kernel_image ${IMAGE} \
  --package.boot_mode=sd \
  --package.image_format=ext4 \
  --package.sd_dir data \
```

```
  --package.sd_file ${HOST_EXE} \
  --package.sd_file emconfig.json \
  -o rm.xclbin \
  ${XSA} \
  libadf.a
```

Launch the emulation with the following command:

```
./launch_hw_emu.sh
```

IMPORTANT! *In hardware emulation you can load the XCLBIN only once. The RM cannot be reloaded.*

In QEMU, following commands can be run:

```
cd /run/media/mmcblk0p1
export XCL_EMULATION_MODE=hw_emu
./host.exe rm.xclbin
```

To run multiple RMs on a DFX platform during hardware emulation you must start and stop the emulator, and load each RM `.xclbin` for a separate test run.

Deploying and Running the System

This section describes deploying and running the system in the Hardware Emulator or on the target device on a physical board.

Running an Application on the Board

The Linux application can be launched automatically at boot time. You can also execute the control application on the command line. For bare metal, the application is launched automatically during boot.

When launching, the packaging uses the defer running AIE at boot option. See --package Options in the *Vitis Reference Guide* (UG1702).

If you are using the common Linux components that are provided by AMD, perform the following steps to run an application on the platform:

1. Write the `sd_card.img` generated by the Vitis compiler command `v++ --package` to the SD card.
2. Boot the board.
3. Run the command `cd /run/media/sd-mmcblk0p1/`.
4. Run the application. For example, for vector addition, run `./vadd ./binary_container_1.xclbin`.

The application uses Xilinx Runtime (XRT) to communicate with kernels.

Note: If the `sd_card.img` file has already been written to the SD card and you are only updating the application, you can save time in the debugging phase by copying all files from `<Vitis System Project>/Hardware/package/sd_card` to a FAT32 partition on a SD card to replace existing files. The Ext4 partition does not change in `sd_card.img`.

Handling Files and Images for SD Card

This section provides guidance for managing SD Card files and images.

Writing Images to the SD Card

You can use the Vitis Unified Software Platform accelerated flow to target an embedded platform. `initramfs` uses Double Data Rate SDRAM (DDR SDRAM) for file system storage; targeting an embedded platform facilitates packaging and creates an SD image with RootFS as an EXT4 partition. The process limits the real usable DDR memory for Linux kernel and applications when the file system size increases; it cannot retain RootFS changes after reboot.

To write EXT4 RootFS to an SD Card:

1. Prepare an SD card binary image file with FAT32 partition for boot and EXT4 partition for RootFS.
2. Write SD card images to the SD card via a tool such as Etcher on Windows or the `dd` command on Linux.

   Note: Refer to AMD Answer Record 73711 for detailed information about these tools.

There are various ways to prepare an SD card image. You can use the v++ package tool to generate it, or use an open source tool. The v++ package tool generated `sd_card.img` has two partitions:

- **FAT32 Partition:** 1 GB size, initialized with the kernel image provided by common Linux components.
- **EXT4 Partition:** 2 GB size, initialized with RootFS provided by common Linux components.

To make the pre-built SD card image boot, you must copy the following boot components to the FAT32 partition:

- `pre-built/BOOT.BIN`
- `boot.scr` from the `<VITIS_INSTALL_DIR>/base_platform/<PLATFORM_NAME>/sw/xrt/image` directory
- `bl31.elf`, `u-boot.elf`, and `system.dtb` from the `<VITIS_INSTALL_DIR>/base_platform/<PLATFORM_NAME>/sw/boot` directory
- A `platform_desc.txt` file containing the name of the platform must also be added to the FAT32 partition.

The pre-built SD card image can be used for evaluation usage and by Windows users. It does not require Vitis or PetaLinux to be installed.

Note: The `v++ --package` with Ext4 partition is not supported on Windows.

For hands on examples, review the packaging step in the tutorial available on GitHub: https://github.com/Xilinx/Vitis-Tutorials/tree/HEAD/Vitis_System_Design/Design_Tutorials/02-Versal_Vitis_Subsystem_Flow.

Using FAT32 Formatted SD Card

For Windows OS, there is no native support for EXT4 formats, so Windows can only access SD Card with FAT32 partitions. In this case, you can format a single primary partition to FAT32 and copy the files from the generated output to the `sd_card` folder.

Note: The BOOT partition using FAT32 is currently limited to a maximum size of 16 GB.

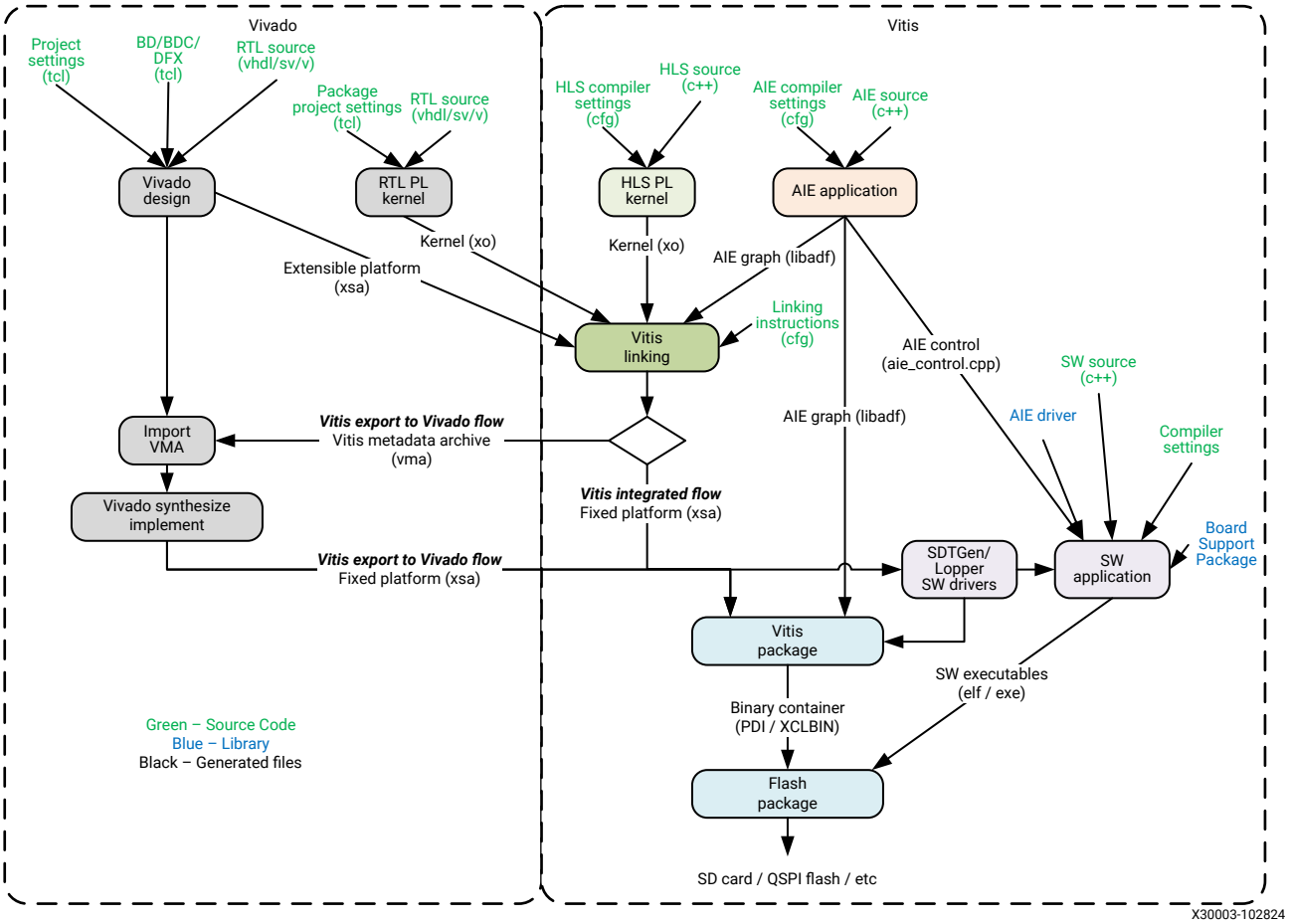
Incremental Design Management

Constructing the project in a single, monolithic build process as described in this chapter so far does not account for the iterative nature of practical design scenarios. As a project grows in complexity, managing changes and updates to individual components becomes essential. By adopting an incremental build strategy, you can significantly reduce the project build time by focusing only on the components that are directly affected by a change. This section explains the methods of managing design changes, allowing you to identify the specific steps that require rebuilding, streamlining the development process and optimizing project workflows.

Understanding the Build Order

To effectively manage the build process, it is crucial to understand the order in which various tools and steps are executed. The Vitis Integrated Flow and Vitis Export to Vivado Flow, provide a high-level overview of the build sequence. Next, it is important to consider the file-level handoffs between Vivado and Vitis flows. The following diagram serves as a starting point for the discussion, illustrating the key components involved in this process.

Figure 37: **System Level Dependencies and Build Handoffs**



The diagram uses boxes to represent activities and arrows to depict the flow of inputs and outputs between them. The activities are governed by a range of settings which can be specified through the IDE GUI, command line options, or a configuration file. Typically these are defined using tcl, python scripts, Makefiles, and configuration text files. In addition to the settings, the build process also relies on source files representing the design objects.

While it can seem intuitive that any step following an activity would necessitate a rebuild, there are nuanced distinctions to consider. The following table highlights some scenarios to shed light on what triggers a rebuild.

Rebuild Strategies

This section lists common scenarios encountered during incremental design evolution. The scenarios are categorized from both hardware and software perspectives. For hardware changes, the software might need to be rebuilt, because software is often used to control and monitor hardware devices. Conversely, pure software changes allow for significant shortcuts as it only involves simple recompile and repackaging, leading to shorter iteration cycles.

Table 32: **Use Case Scenarios for Hardware Rebuild Strategy**

Scenario	Action	Comments
Initial build	Build/Compile all steps according to Vitis Integrated Flow or Vitis Export to Vivado Flow.	When using the "part" selection method for the device, all activities that generate inputs for the Vitis linker can be compiled in any order. This flexibility allows for parallel processing and potentially faster build times. ***Note***: For Vitis Integrated Flow projects with XRT host applications, the extensible XSA can be used to setup Linux OS. This allows concurrent development of software applications alongside the AIE and PL kernels. **IMPORTANT!** *While incremental builds offer significant benefits, it's crucial to perform regular full builds as part of regression testing. This ensures the design's integrity and readiness for reuse or branching into other projects.*
Modify Vivado RTL or IP integrator blocks not related to the Vitis region.	1. Open the Vivado project containing the imported VMA. 2. Edit the block design, validate and save 3. Reset synthesis and implementation runs, and then rerun them as performed in initial design iteration. 4. Rebuild/recompile all steps following the Vivado synthesis and implementation step.	This scenario involves modifying Vivado RTL or IP integrator blocks that are not directly related to the Vitis region, assuming these changes do not impact the AI Engine or Vitis region itself. Typical use cases include modifying synthesis and implementation strategies, addressing engineering change orders (ECOs), or adding additional debug ILAs.
Modify Vivado RTL or IP integrator blocks connected to Vitis region	1. Open the Vivado project containing the imported VMA. 2. Disconnect and remove the Vitis region with `vitis::remove_archive_hierarchy` 3. Update the block design, validate and write a new extensible xsa. 4. Update and recompile affected PL kernels or AIE graph. 5. Update the link instructions and run Vitis linker. 6. Rebuild/recompile all steps subsequent to Vitis linker.	This scenario involves modifications to Vivado RTL or IP integrator blocks that have connections to the Vitis region. Typical use cases include implementing incremental design strategies or merging ECOs into the base design.

Table 32: **Use Case Scenarios for Hardware Rebuild Strategy** *(cont'd)*

Scenario	Action	Comments
Modify RTL or HLS PL kernels	1. Open the Vivado project containing the imported VMA. 2. Disconnect and remove the Vitis region with `vitis::remove_archive_hierarchy` 3. Update and recompile affected PL kernels. 4. Update the link instructions and run Vitis linker. 5. Rebuild/recompile all steps subsequent to Vitis linker.	This scenario addresses changes to RTL or HLS IP kernels within the Vitis region, assuming these modifications do not impact the physical interfaces to/from the AI Engine. ***Note*:** If you adjust the number of instances of an existing kernel XO, step 3 can be skipped.
Modify AI Engine graph	1. Open the Vivado project containing the imported VMA. 2. Disconnect and remove the Vitis region with `vitis::remove_archive_hierarchy` 3. Update and recompile AI Engine graph as required. 4. Update and recompile any affected PL kernels. 5. Update the linking instructions and run Vitis linking. 6. Rebuild/recompile all steps subsequent to Vitis linker.	**TIP:** *If the interface protocols between AI Engine and PL or GMIO remain unchanged, recompiling the graph can be treated as a software update. This allows for skipping all steps except recompiling SW host application and rerunning Vitis package and Flash package steps.*
Modify VSS components	1. Follow the methods presented in previous scenarios to update and recompile PL kernels and AI Engine graph. 2. Update and recompile the affected VSS components. 3. Follow the methods presented in previous scenarios to rebuild/recompile with Vitis linker.	Changes to VSS follow the same principle as listed in the previous scenarios with the addition of recompiling the VSS components and repeating the steps mentioned in the respective scenarios.

Table 33: **Use Case Scenarios for Software Rebuild Strategy**

Scenario	Action	Comments
Modify host software applications	1. Modify the software source code and recompile to executable. 2. Rerun Vitis flash package.	This assumes no new hardware changes are made to the design. ***Note*:** The new executable can alternatively be downloaded directly to the boards Linux file system using FTP or secure copy. To ensure the new file is properly saved, use the `poweroff` or `reboot` command to synchronize the file system.

Table 33: **Use Case Scenarios for Software Rebuild Strategy** *(cont'd)*

Scenario	Action	Comments
Modify device drivers	1. Modify the device tree system include. 2. Rebuild Vitis platform component to update system device tree and board support packages. 3. Update and recompile affected SW host applications 4. Rerun Vitis package steps.	

Key takeaways:

- By analyzing these scenarios and the dependencies between the flow steps, you can leverage concurrent development strategies to streamline the design process.
- When multiple designers collaborate in developing a design, it is crucial to ensure the ability to rebuild the design from scratch to a known state.
- A clear understanding of the merge points and opportunities for splitting activities across a team is essential for efficient collaboration.

Application Verification Using Vitis Emulation Flow

Development of an application and hardware kernels targeting an FPGA requires a phased development approach. Because FPGA, AMD Versal™ adaptive SoC, and AMD Zynq™ UltraScale+™ MPSoC are programmable devices, building the device binary for hardware takes time. To enable faster iterations without going through the full hardware compilation flow, the AMD Vitis™ tool provides hardware emulation target to perform C-RTL co-simulation of the software application and PL kernels. Compiling for hardware emulation target is significantly faster than compiling for the actual hardware. Additionally, hardware emulation target provides full visibility into the application, making it easier to perform debugging. Once your design passes in hardware emulation, you can compile and run the application on the hardware platform in the late stages of development.

The Vitis tool provides the following emulation target:

- **Hardware emulation (hw_emu):** The host program runs in the QEMU, but the kernel code is compiled into an RTL behavioral model which is run in the AMD Vivado™ simulator or other supported third-party simulators. This build and run loop takes longer but provides a cycle-accurate view of kernel logic.

Compiling and linking for emulation targets is seamlessly integrated into the Vitis command line and IDE flows. You can compile your host and kernel source code for hardware emulation target, without making any change to the source code. For your host code, you do not need to compile differently for emulation as the same host executable or PS application ELF binary can be used in emulation. Hardware emulation target support most of the features including XRT APIs, buffer transfer, platform memory SP tags and kernel-to-kernel connections. The following sections describes the features and requirements of the hardware emulation flow.

In a typical development flow, verification using HW Emulation is done prior to hardware run. The following are some prerequisites that can be followed during the application verification:

1. Functional emulation of the system for verifying the functional correctness of the complete system. This includes running C simulation on HLS components, or running the x86 simulator for verifying AI Engine components.
2. AI Engine simulator for verifying that the AI Engine kernel and graph meets performance needs of the application.

3. Hardware emulation of the system for verifying timing, and accuracy of the full design prior to Hardware run.

   Note: This can also include running C/RTL co-simulation on HLS components, or running the AIE Simulator for verifying AI Engine components.

4. Testing and debugging on hardware.

Using Embedded Platforms

Emulation is supported for embedded platforms. The device is modeled as separate x86 process emulating the hardware. The user host code and the device model process communicate using RPC calls. For embedded platforms, where the CPU code is running on the embedded Arm processor, emulation flows use QEMU (Quick Emulator) to mimic the Arm-based PS-subsystem. In QEMU, you can boot embedded Linux and run Arm binaries on the emulation targets.

For running emulation of an embedded application, you will compile the application for an Arm processor using Arm-GCC and launch the QEMU emulation environment on an x86 processor to model the execution environment of the Arm processor. This requires the use of the `launch_emulator.py` command, or `launch_emulator.sh` shell scripts generated during the build process. The details of this flow are explained in Running the System on Embedded Processor Platform.

QEMU

QEMU stands for Quick Emulator. It is a generic and open source machine emulator. AMD provides a customized QEMU model that mimics the Arm based processing system present on AMD Versal™ adaptive SoC, AMD Zynq™ UltraScale+™ MPSoC, and Zynq 7000 SoC. The QEMU model provides the ability to execute CPU instructions at almost real time without the need for real hardware. For more information, refer to the QEMU User Documentation.

For hardware emulation, the AMD Vitis™ emulation targets use QEMU. It is co-simulated with an RTL and SystemC-based model for the rest of the design to provide a complete execution model of the entire platform. You can boot an embedded Linux kernel on the model and run XRT-based applications. Because the QEMU can execute Arm instructions, you can take the Arm binaries and run them in emulation flows as-is without the need to recompile. QEMU also allows you to debug your application using GDB and TCF-based target connections from Xilinx System Debugger (XSDB).

The Vitis emulation flow also uses QEMU to emulate the MicroBlaze™ processor to model the platform management modules (PLM and PMU) of the devices. On Versal devices, the PLM firmware is used to load the PDI to program sections of the PS and AI Engine model.

To ensure that the QEMU configuration matches the platform, there are additional files that must be provided as part of `sw` directory of Vitis platforms. Two common files, `qemu_args.txt` and `pmc_args.txt`, contain the command line arguments to be used when launching QEMU. When you create a custom platform, these two files are automatically added to your platform with default contents. You can review the files and edit as needed to model your custom platform. Refer to an AMD embedded platform for an example.

Because QEMU is a generic model, it uses a Linux device tree style DTB formatted file to enable and configure various hardware modules. A default QEMU hardware DTB file is shipped with the Vitis tools in the `<vitis_installation>/data/emulation/dtbs` folder. However, if your platform requires a different QEMU DTB, you can package it as part of your platform.

TIP: *The QEMU DTB represents the hardware configuration for QEMU, and is different from the DTB used by the Linux kernel.*

Building the System in HW Emulation

To simulate the entire system, including AI Engine graph and PL logic along with XRT-based host application to control the AI Engine and PL, for a specific board and platform, you must use the Vitis hardware emulation flow. This flow includes the SystemC model of the AI Engine, transaction-level SystemC models for the NoC, DDR memory, PL Kernels (RTL), and the PS (running on QEMU).

Building the system involves building the device binary for HW Emulation target including AI Engine graph and the PL kernel and building the XRT based PS application. For details on building the system in HW Emulation, see Chapter 8: Building and Running the System.

The Vitis tool provides two distinct end-to-end flows for embedded systems: hardware emulation (hw_emu) and hardware (hw). These flows are intentionally separate because they target different execution models, generate different artifacts, and serve different validation goals.

What the Linker and Packager Builds

See the following for the tasks hw_emu performs.

- `v++ --link -t hw_emu` builds a co-simulation model that integrates the following.
  - Processing System (PS) running functionally in QEMU.
  - AI Engine (AIE) modeled in SystemC (cycle-approximate).
  - Programmable Logic (PL) kernels compiled to RTL and simulated in an RTL simulator (for example, XSIM).
  - Transaction-level models for NoC, memory, and other data paths where applicable.

- The linker still produces an `.xclbin` that XRT consumes. Packaging (`v++ -p -t hw_emu`) generates emulation artifacts and scripts (for example, `launch_hw_emu.sh`). Packaging also generates emulation-oriented boot components (rootfs, device tree) required to drive QEMU and the simulators.

hw:

- `v++ --link -t hw` generates a deployable `.xclbin` that corresponds to a fully implemented PL bitstream, AIE binaries, and associated metadata to run on the device.
- Packaging produces SD/flash images for the board. The images contain real bitstream/PDI, AIE binaries, and application/software stack for execution on silicon.

Execution Model and Fidelity

hw_emu:

- Runs in a mixed simulation environment. The PS is functionally accurate (QEMU). AIE and interconnect and memory are cycle-approximate SystemC models. The user PL kernels execute as RTL in the simulator (RTL-accurate within the simulated region).
- This option is not intended to be cycle-accurate at the full-system level. Latency, bandwidth, and contention behavior can differ from hardware.

hw:

- Runs on real hardware with actual clocks, I/O, DDR/LPDDR behavior, NoC routing, board peripherals, and software stack.
- Required to validate timing closure, I/O margins, power/thermal behavior, and full-chip interactions.

Debug, visibility, and analysis

hw_emu:

- Emphasis on visibility and bring-up. These include RTL waveforms, SystemC/TLM traces, AIE graph inspection, and rich XRT profiling/tracing.
- Supports system-level checks such as deadlock detection workflows and controlled traffic generation.
- Memory monitoring: supports AXI Interface Monitors (AIMs) on memory ports in counters-only configurations.

hw:

- Debug uses on-chip capabilities (for example, ILA/ChipScope, XVC) and software debuggers. Visibility is more limited than in emulation.
- Used to confirm real-world performance, bandwidth under contention, and platform behavior that only appears on silicon.

Turnaround and optimization

hw_emu:

- Faster build–run cycles because it avoids full synthesis/place/route and board programming.
- Ideal for functional integration, driver/XRT flow debug, and early performance trend analysis.
- hw:
  - Longer compile and deployment cycles due to full Vivado implementation and device programming.
  - Essential for QoR assessment (timing/resource), final performance/power, and board-level validation.

Segmented configuration and dynamic reload

hw_emu:

- Segmented configuration (multi-PDI overlays and dynamic PL reload) is not supported. Attempts to use segmented configuration in hw_emu will result in an error.

hw:

- Segmented configuration is supported (platform- and version-dependent), enabling dynamic PL reload and multi-PDI use cases. It provides an entry point to DFX-like scenarios without a full Vivado DFX flow and supports isolation/subsystem use cases where PS-to-PL paths remain fixed.

When to use which flow

Choose hw_emu to:

- Functionally validate PS–AIE–PL integration with high debug visibility.
- Develop and debug host/driver/AIE/PL interactions and XRT flows.
- Investigate deadlocks and tune buffering/latencies using approximate system models.
- Gather early profiling data and performance trends (not final performance).

Choose hw to:

- Validate timing, resource usage, bandwidth, and performance on the actual device.
- Exercise board-level I/O and full software stacks under realistic conditions.
- Use segmented configuration/dynamic PL reload where applicable.
- Prepare the design for deployment and sign-off.

Packaging the System in HW Emulation

Packaging the entire system after compilation and linking is required for embedded processor platforms. Packaging the system generates the SD card and necessary flash images that helps boot the system prior to running HW Emulation. The packaging for HW Emulation requires target to be set `v++ -p -t hw_emu`. 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 more details on package options, see Packaging for Vitis Flow.

Running the System on Embedded Processor Platform

Running Emulation target is achieved in the QEMU environment on embedded processors. Hardware emulation target has specific drivers which are loaded at runtime by XRT. Thus, the same CPU binary can be run as-is without recompiling, by changing the target mode during runtime. Based on the value of the `XCL_EMULATION_MODE` environment variable, XRT loads the target specific driver and makes the application interface with an emulation model of the hardware. The allowed value of `XCL_EMULATION_MODE`is `hw_emu`. If `XCL_EMULATION_MODE` is not set, then XRT will load the hardware driver.

IMPORTANT! *It is required to set `XCL_EMULATION_MODE` when running emulation.*

Use the `xrt.ini` file to configure various options applicable to emulation. There is an `[Emulation]` specific section in `xrt.ini`, as described in xrt.ini File.

Emulation for a system with the AI Engine can be useful in:

- Checking initial system behavior with a limited known data set
- Functional integration and debugging of PS, PL, and ADF graph using GDB
- Testing the system with external traffic generator using Python, or C++
- Running system with C-based models for RTL kernels
- Applying AI Engine simulation options through the `aiesim_options.txt` found in the `aiesimulator_output` directory

When launching hardware emulation, specify options for the AI Engine component according to the simulator being used. These options can be specified from the `launch_hw_emu.sh` script using the `-aie-sim-options`.

Reusing AI Engine Simulator Options

The AI Engine simulator generates an options file that lists the options used for simulating the AI Engine graph application. The options file is automatically generated when the AI Engine simulator is run. This helps reuse the AI Engine simulator options from the initial graph-level simulation later in the system-level hardware emulation. You can also manually edit the options file to specify other options as required. The following table lists the options that can be specified in the `aiesim_options.txt` file. This file is located in the `aiesimulator_output` directory and is created if the `--dump-vcd` option is used with the `aiesimulator` command. An example command line is as follows.

```
./launch_hw_emu.sh \
-aie-sim-options ${FULL_PATH}/aiesimulator_output/aiesim_options.txt
```

where `${FULL_PATH}` must be the full path to the file or directory.

Table 34: **AI Engine Options for Hardware Emulation**

Command	Arguments	Description
`AIE_DUMP_VCD`	<filename>	When `AIE_DUMP_VCD` is specified, the simulation generates VCD data and writes it to the specified `<filename>.vcd`. For more information about the options used to generate select signals associated with specific modules, see Generating VCD with Select Signals in the *AI Engine Tools and Flows User Guide* (UG1076)
`AIE_DEBUG_AXIMM`	True | False	When `AIE_DEBUG_AXIMM` is enabled (`True`), simulation generates memory-mapped AXI4 transaction data and writes it to a `aiesim_debug_axi_mm_dump.txt` file.
`AIE_PKG_DIR`	`/path_to_work_dir/Work`	This is a mandatory option that sets the path to the `Work` directory generated by the AI Engine compiler. If you do not specify this option, the generated `sim/behav_waveform/xsim/default.aierun_summary` file will not have the correct `Work` directory setting, which will impact the display of the summary file in the Vitis analyzer.

***Note*:** Any command that has a path to a file must use an absolute path.

When creating a simulation option file manually it needs to follow the format of `COMMAND=ARGUMENT`, with each command being on a separate line. The following example shows best practice.

```
AIE_DUMP_VCD=foo
```

To view the AMD Vivado™ simulator waveform GUI, use the `launch_hw_emu` script with `-g` option, and also use the `-g` option during the `v++ --link` stage as well.

```
./launch_hw_emu.sh -g
```

When the emulation environment is fully booted and the Linux prompt is up, make sure to set the following environment variables in the QEMU environment to ensure that the host application works. These must also be set when running on hardware.

```
export XILINX_XRT=/usr
export LD_LIBRARY_PATH=/mnt/sd*1:
export XCL_EMULATION_MODE=hw_emu
```

Below are end-to-end steps to launch HW Emulation:

RECOMMENDED: *The file size limit on your machine should either be set to unlimited or a higher value (over 16 GB) because embedded HW Emulation can create files with larger file size for memory.*

TIP: *Set up the command shell or window as described in Setting Up the Vitis Environment prior to running the builds.*

1. Set the desired runtime settings in the `xrt.ini` file.

   As described in xrt.ini File, the file specifies various parameters to control debugging, profiling, and message logging in XRT when running the host application and kernel execution. As described in Enabling Profiling in Your Application this enables the runtime to capture debugging and profile data as your application is running.

   The `xrt.ini` file, as well as any additional files required for running the application, must be included in the output files as explained in Packaging for Vitis Flow

   TIP: *Be sure to use the `v++ -g` option when compiling your kernel code for emulation mode.*

2. Launch the QEMU emulation environment by running the `launch_hw_emu.sh` script.

   The script is created in the emulation directory during the packaging process, and uses the `launch_emulator.py` command to setup and launch QEMU. When launching the emulation script you can also specify options for the `launch_emulator.py` command. Such as the `-forward-port` option to forward the QEMU port to an open port on the local system. This is needed when trying to copy files from QEMU as discussed in Step 5 below. Refer to launch_emulator Utility in the *Vitis Reference Guide* (UG1702) for details of the command.

   For AI Engine, specify the AI Engine Options for HW Emulation as described in above paragraph.

   Another example would be to specify `launch_hw_emu.sh -enable-debug` to configure additional XTERMs to be opened for QEMU and PL processes to observe live transcripts of command execution to aid in debugging the application. This is not enabled by default, but can be useful when needed for debug.

Additionally, you can add more advanced options to log waveform data without having to launch emulation with the Vivado logic simulator GUI. An example command line is as follows.

```
./launch_hw_emu.sh \
-user-pre-sim-script pre-sim.tcl
```

The `pre-sim.tcl` contains Tcl commands to add waveforms or log design waveforms. For an example, for Tcl commands see *Vivado Design Suite User Guide: Logic Simulation* (UG900).

TIP: *Details on debugging techniques in Hardware Emulation can be found in Profile and Debug in Hardware Emulation.*

3. Once the emulation environment is fully booted and the Linux prompt is up, mount and configure the QEMU shell with the required settings.

   The AMD embedded base platforms have `rootfs` on a separate EXT4 partition on the SD card. After booting Linux, this partition needs to be mounted. If you are running emulation manually, you need to run the following commands from the QEMU shell:

   ```
   mount /dev/mmcblk0p1 /mnt
   cd /mnt
   export LD_LIBRARY_PATH=/mnt:/tmp:$LD_LIBRARY_PATH
   export XCL_EMULATION_MODE=hw_emu
   export XILINX_XRT=/usr
   export XILINX_VITIS=/mnt
   ```

   TIP: *You can set the `XCL_EMULATION_MODE` environment variable to `hw_emu` for hardware emulation. This configures the host application to run in emulation mode.*

4. Run the application from within the QEMU shell.

   With the runtime initialization (`xrt.ini`), the `XCL_EMULATION_MODE` environment set, run the host executable with the command line as required by the host application. For example:

   ```
   ./host.elf kernel.xclbin
   ```

   TIP: *This command line assumes that the host program is written to take the name of the `xclbin` file as an argument, as most AMD Vitis™ examples and tutorials do. However, your application can have the name of the `xclbin` file hard-coded into the host program, or can require a different approach to running the application.*

5. After the application run has completed is generated, the `xrt.run_summary` can be found in the `/mnt` folder inside the QEMU environment. However, to view the file you must copy them from the QEMU Linux system back to your local system.

   ***Note*:** The files can be copied either from the Guest or Host machine using the `scp` command. Key terminology to notice here are *Host* and *Guest*. The *Host* machine is the machine hosting QEMU, and the *Guest* machine is the one with PetaLinux running on QEMU. `Host-ip-address` is the IP address of the machine from which QEMU is launched.

a. Copying files from the Host machine:

   i. First, copy the required files from the root area to the `petalinux` (default) user home area using the following command from the Guest machine: `cp -rf /mnt/<files> /home/petalinux/`

   ii. Switch from the default login user `petalinux` using the `exit` command

   iii. Change the permission of the copied files using `sudo chmod 755 <files>`

   The files can be copied using the scp command from the Host machine as follows: `scp -P <port-num> <guest-machine-user-name>@<host-ip-address>:<source-file> <dest-path>`. For example: `scp -P 1440 root@192.168.1.xxx:/mnt/xrt.run_summary`.

   `<port-num>`: To copy Guest machine files to the Host, the Port number through which Host and Guest are connected is required. 1440 is the QEMU port to connect to the Guest port. The `-forward-port 1440 22` is passed to the `launch_emulator` to map the Guest port to the Host port. Here, 22 is the Guest TCP port and 1440 is the Host port. Both ports are mapped. To access the Guest TCP services (on port 22), use the Host machine's mapped port (for example 1440). Accessing the Host's port 1440 means accessing Guest port 22.

   `<guest-machine-user-name>`The guest machine user name can be found by using the `whoami` command from the PetaLinux terminal.

   `<host-ip-address>`: The host IP address can be found with a Linux command such as `nslookup <machine name>` or `hostname-i`

   `<source-file>`: The path and name of the file you want to copy from the QEMU environment.

   `<dest-path>`: The destination path to copy the file to on the local system.

b. Copying files from the Guest machine:

   You can copy from the guest machine using the `scp` command: `scp <guest-file> <userid-of-hostmachine>@<host-ip-address>:<host's-dir-path>` For example: `scp /mnt/xrt.run_summary userabcd@192.168.1.xxx:~`

6. When your application has completed emulation and you have copied the required files, press the **Ctrl + a + x** keys to terminate the QEMU shell and return to the Linux shell.

   ***Note*:** If you have trouble terminating the QEMU environment, you can kill the processes it launches to run the environment. The tool reports the process IDs (pids) at the start of the transcript, or you can specify the `-pid-file` option to capture the pids when launching emulation.

Viewing the Run Summary

After hardware emulation is launched and run with the above simulation options, the run summary is available as follows:

For AI Engine run summary generated inside `<package_dir>/sim/behav_waveform/<simulator>/default.aierun_summary`, the VCD file is generated at the same location, for example: `<package_dir>/sim/behav_waveform/<simulator>/<vcd filename>`.

***Note*:** Viewing summary from the third-party simulator is also available.

AI Engine Throughput Estimates

Hardware Emulation reports the average throughput of each PLIO and GMIO port at the end of a simulation.

As described in the previous section, emulation is launched using the following command:

```
./launch_hw_emu.sh -aie-sim-options ${FULL_PATH_TO}/
<user_aiesim_options.txt>
```

Here, `<user_aiesim_options.txt>` must include the `AIE_PKG_DIR` option that sets the path to the `Work` directory generated by the AI Engine compiler.

The throughput report is generated inside `simulate.log` found at `<design_directory>/sim/behav_waveform/xsim/simulate.log`.

```
Info: /OSCI/SystemC: Simulation stopped by user.

Info: SystemC end_of_simulation Dump:
103765200
ps .vitis_design_wrapper_sim_wrapper.vitis_design_wrapper_i.vitis_design_i.a
i_engine_0.inst.aie_logical.aie_xtlm.math_engine: end of simulation invoked!
-------------------------------------------------------------
| Intf Type    | Port Name       | Type   | Throughput(MBps)  |
-------------------------------------------------------------
| gmio         | gmioIn0         | IN     | 3404.26           |
|              | gmioOut0        | OUT    | 3453.82           |
|              | gmioIn1         | IN     | 3890.58           |
|              | gmioOut1        | OUT    | 3957.06           |
|              | gmioIn2         | IN     | 3968.99           |
|              | gmioOut2        | OUT    | 3903.18           |
|              | gmioIn3         | IN     | 3244.61           |
|              | gmioOut3        | OUT    | 3278.27           |
-------------------------------------------------------------
```

Simulator Support in Hardware Emulation

The AMD Vitis™ tool uses the AMD Vivado™ logic simulator (`xsim`) as the default simulator for all platforms, AMD Versal™ and AMD Zynq™ UltraScale+™ MPSoC embedded platforms. However, for Versal embedded platforms, like `xilinx_vck190_base` or custom platforms similar to it, the Vitis tool also supports the use of third-party simulators for hardware emulation: Mentor Graphics Questa Advanced Simulator, Xcelium, and VCS. The specific versions of the supported simulators are the same as the versions supported by Vivado Design Suite.

Enabling Third-Party Simulators

Third-party simulators are supported in the AMD Vitis™ Hardware Emulation flow. There are specific settings around third-party simulators that need to be provided in the AMD Vitis™ configuration file. The AMD Vitis™ Unified IDE also supports hardware emulation flow using third-party simulators. Third-party simulators such as Questa Advanced Simulator (Mentor Graphics), Xcelium (Cadence), VCS (Synopsys), and Riviera Simulator (Aldec) are supported when executing hardware emulation of your design. For more details, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900) for third-party simulator setup.

You can enable these simulators by updating the AMD Vitis™ configuration file (`config.ini` or `system.cfg`). When the settings are added to the AMD Vitis™ configuration file, build the design using the v++ link and package flow as described in the script `launch_hw_emu.sh`. This will launch hardware emulation using the third party simulator specified.

The following are the configuration options to enable third-party simulator setup during `v++ -link` command line flow. To see enabling third party simulators in the AMD Vitis™ Unified IDE, refer to Enabling Third Party Simulators in the *Vitis Reference Guide* (UG1702).

Table 35: **Vitis Link Settings**

Simulator	Vitis configuration file settings (`config.ini` or `system.cfg`)
Questa Advanced Simulator	`[advanced]` `param=hw_emu.simulator=QUESTA` `[vivado]` `prop=project.__CURRENT__.simulator.questa_install_dir=<SIMULATOR DIRECTORY>/questa/2025.2/bin` `prop=project.__CURRENT__.compxlib.questa_compiled_library_dir=<SIMULATOR LIBRARY DIRECTORY>/questa/2025.2/lin64/lib/` `prop=fileset.sim_1.questa.compile.sccom.cores={16}` `prop=fileset.sim_1.questa.elaborate.vopt.more_options={-stats=all}` `prop=fileset.sim_1.questa.simulate.vsim.more_options={-stats=all}`

Table 35: **Vitis Link Settings** *(cont'd)*

Simulator	Vitis configuration file settings (`config.ini` or `system.cfg`)
Xcelium	`[advanced]` `param=hw_emu.simulator=XCELIUM` `[vivado]` `prop=project.__CURRENT__.simulator.xcelium_install_dir=<SIMULATOR DIRECTORY>/bin/` `prop=project.__CURRENT__.compxlib.xcelium_compiled_library_dir=<SIMULATOR LIBRARY DIRECTORY>/xcelium/25.03.002/lin64/lib/` `prop=fileset.sim_1.xcelium.elaborate.xmelab.more_options={-timescale 1ns/1ps -STATUS]`
VCS	`[advanced]` `param=hw_emu.simulator=VCS` `[vivado]` `prop=project.__CURRENT__.simulator.vcs_install_dir=<SIMULATOR DIRECTORY>/vcs/X-2025.06/bin/` `prop=project.__CURRENT__.compxlib.vcs_compiled_library_dir=<SIMULATOR LIBRARY DIRECTORY>/vcs/X-2025.06/lin64/lib/` `prop=project.__CURRENT__.simulator.vcs_gcc_install_dir=<SIMULATOR DIRECTORY>/synopsys/vg_gnu/X-2025.06/linux64/gcc-9.2.0_64/bin` `param=project.alignLibraryPathEnvForVCS=true` `prop=fileset.sim_1.vcs.compile.vlogan.more_options={-v2005}`
Riviera	`[advanced]` `param=hw_emu.simulator=RIVIERA` `[vivado]` `prop=project.__CURRENT__.simulator.riviera_install_dir=<SIMULATOR DIRECTORY>/riviera/2024.10-lin64/bin/` `prop=project.__CURRENT__.compxlib.riviera_compiled_library_dir=<SIMULATOR LIBRARY DIRECTORY>/riviera/2024.10/lin64/lib/` `prop=project.__CURRENT__.simulator.riviera_gcc_install_dir=<SIMULATOR DIRECTORY>/riviera/2024.10-lin64/gcc_Linux64/bin/` `prop=fileset.sim_1.riviera.simulate.asim.more_options={+access +r}`

After generating the configuration file you can use it in the `v++` command line, for example:

```
v++ -link --config system.cfg
```

You can use the `-user-pre-sim-script` and `-user-post-sim-script` options from the `launch_emulator.py` command to specify Tcl scripts to run before the start of simulation, or after simulation completes. As an example, in these scripts, you can use the `$cwd` command to get the run directory of the simulator and copy any files needed prior to simulation, or copy any output files generated at the end of simulation.

To enable hardware emulation, you must set up the environment for simulation in the Vivado Design Suite. A key step for setup is pre-compiling the RTL and SystemC models for use with the simulator. To do this, you must run the `compile_sim_lib` command in the Vivado tool. For more information on pre-compilation of simulation models, refer to the *Vivado Design Suite User Guide: Logic Simulation* (UG900).

When creating your Versal platform ready for simulation, the Vivado tool generates a simulation wrapper which must be instantiated in your simulation test bench. So, if the top most design module is `<top>`, then when calling `launch_simulation` in the Vivado tool, it will generate a `<top>_sim_wrapper` module, and also generates `xlnoc.bd`. These files are generated as simulation-only sources and will be overwritten whenever `launch_simulation` is called in the Vivado tool. Platform developers need to instantiate this module in the test bench and not their own `<top>` module.

Using the Simulator Waveform Viewer

Hardware emulation uses RTL and SystemC models for execution. A regular application and HLS-based kernel developer does not need to be aware of the hardware level details. The Vitis analyzer provides sufficient details of the hardware execution model. For advanced users who are familiar with HW signal and protocols, you can launch hardware emulation with the simulator waveform running, as described in Waveform View and Live Waveform Viewer.

By default, when running `v++ --link -t hw_emu`, the tool compiles the simulation models in optimized mode. However, when you also specify the `-g` switch, you enable hardware emulation models to be compiled in debug mode. During the application runtime, use the `-g` switch with the `launch_hw_emu.sh` command to run the simulator interactively in GUI mode with waveforms displayed. By default, the hardware emulation flow adds common signals of interest to the waveform window. However, you can pause the simulator to add signals of interest and resume simulation.

Waveform View and Live Waveform Viewer

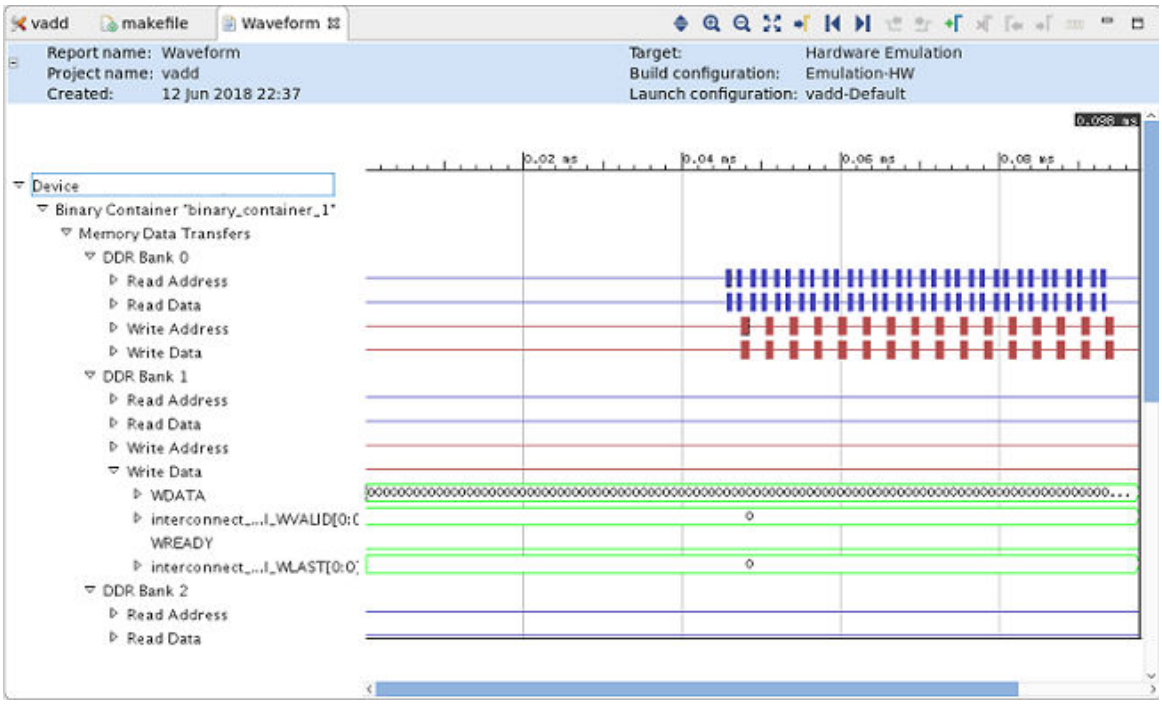
The Vitis core development kit can generate a Waveform view when running hardware emulation. It displays in-depth details at the system-level, CU level, and at the function level. The details include data transfers between the kernel and global memory and data flow through inter-kernel pipes. These details provide many insights into performance bottlenecks from the system-level down to individual function calls to help optimize your application.

The Live Waveform Viewer is similar to the Waveform view, however, it provides even lower-level details with some degree of interactivity. The Live Waveform Viewer can also be opened using the Vivado logic simulator, `xsim`.

***Note*:** The Waveform view allows you to examine the device transactions from within the Vitis analyzer (see Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702)). In contrast, the Live Waveform Viewer opens the Vivado simulation waveform viewer to examine the hardware transactions in addition to any user selected signals.

Waveform data is not collected by default because it requires the runtime to generate simulation waveforms during hardware emulation, which consumes more time and disk space. Refer to Generating and Opening the Waveform Reports for instructions on enabling these features.

Figure 38: **Waveform View**



You can also open the waveform database (`.wdb`) file with the Vivado logic simulator through the Linux command line:

```
xsim -gui <filename.wdb> &
```

TIP: *The `.wdb` file is written to the directory where the compiled host code is executed.*

Generating and Opening the Waveform Reports

Follow these instructions to enable waveform data collection from the command line during hardware emulation and open the viewer.

1. Enable debug code generation during compilation and linking using the `-g` option.

   ```
   v++ -c -g -t hw_emu ...
   ```

2. Create an `xrt.ini` file in the same directory as the host executable with the following contents (see xrt.ini File for more information).

   ```
   [Emulation]
   debug_mode=batch
   ```

The `debug_mode=batch` enables the capture of waveform data (`.wdb`) by running simulation in batch mode. You can also enable the Live Waveform Viewer to launch simulation in interactive mode using the following setting in the `xrt.ini`.

```
[Emulation]
debug_mode=gui
```

TIP: *If Live Waveform Viewer is enabled, the simulation waveform opens during the hardware emulation run.*

3. Run the hardware emulation build of the application as described in Chapter 9: Application Verification Using Vitis Emulation Flow. The hardware transaction data is collected in the waveform database file, `<hardware_platform>-<device_id>-<xclbin_name>.wdb`. Refer to Output Directories of the v++ Command in the *Vitis Reference Guide* (UG1702) for more information on locating these reports.
4. Open the Waveform view in the Vitis analyzer by opening the Run Summary, and opening the Waveform report.

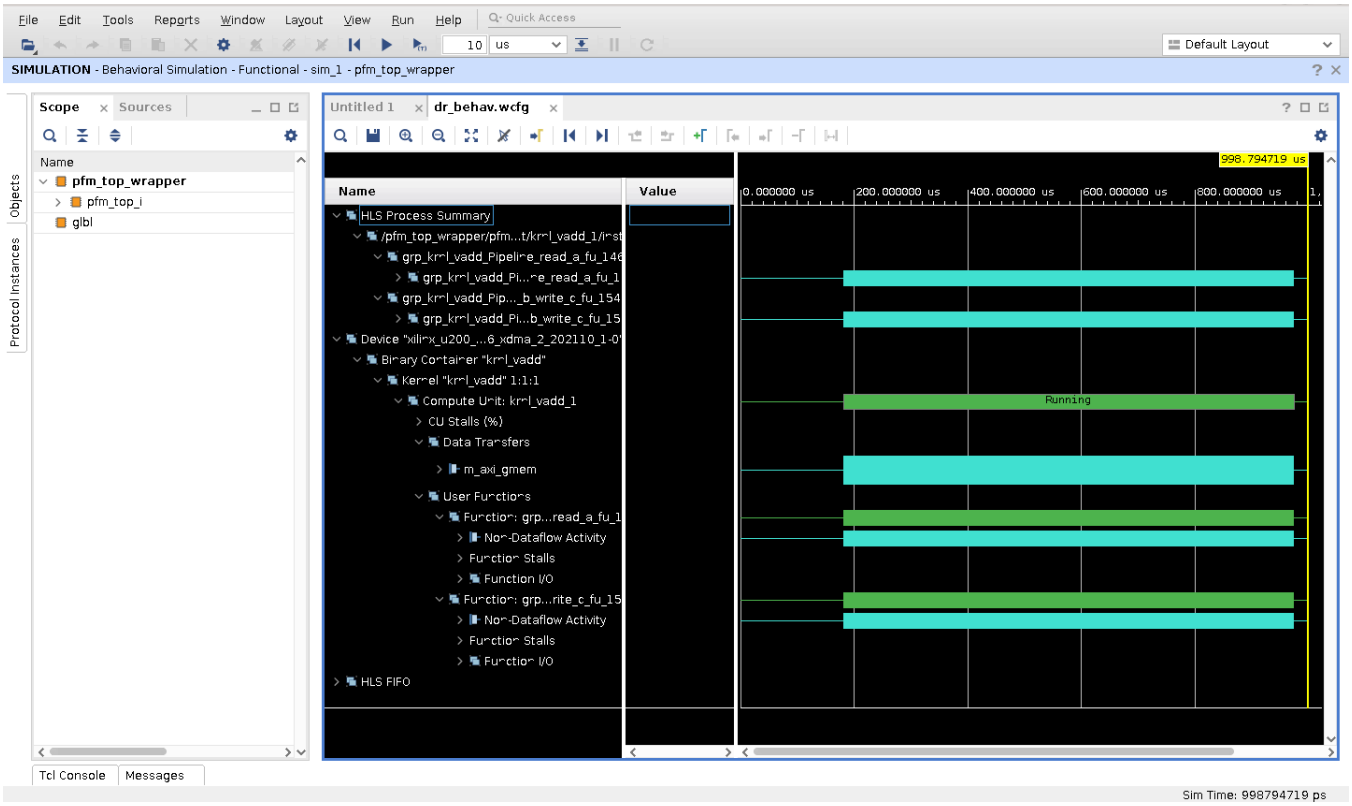
   ```
   vitis_analyzer xrt.run_summary
   ```

5. Waveforms for TLM transactions can also be dumped for third-party simulators (support limited to Mentor Graphics Questa Advanced Simulator and Cadence Xcelium). Wave data dump is enabled when `v++` link is done with `-g` option (as mentioned in step 1). The format of the wave database dumped is simulator specific (for example, `.wlf` for Questa Advanced Simulator and `.shm` for Xcelium).

Interpreting Data in the Waveform Views

The following image shows the Waveform view:

Figure 39: **Waveform View**



The Waveform and Live Waveform views are organized hierarchically for easy navigation.

- The Waveform view is based on the actual waveforms generated during hardware emulation (Kernel Trace). This allows the viewer to descend all the way down to the individual signals responsible for the abstracted data. However, because the Waveform view is generated from the post-processed data, no additional signals can be added to the report, and some of the runtime analysis cannot be visualized, such as DATAFLOW transactions.
- The Live Waveform viewer is displaying the Vivado logic simulator (`xsim`) run, so you can add extra signals and internals of the register transfer (RTL) design to the live view. Refer to the *Vivado Design Suite User Guide: Logic Simulation* (UG900) for information on working with the Waveform viewer.

The hierarchy of the Waveform and Live Waveform views include the following:

- **HLS Process Summary:**
  - Hierarchical view of processes and their sub-processes corresponding to the user functions in CU
  - Entry for each kernel instance which has processes modeling user function in it (entries can be unfolded to show the processes in it)
  - Handshake transactions on all the processes corresponding to user-functions

- Both dataflow and non-dataflow/non-piplelined processes
- The transactions on the processes including stalls, are shown using the corresponding protocol analyzer instance
- Allows you to get an overview about the usage of the individual processes over the execution time (similar to C/C++ profile capabilities)

- **Device "name":** Target device name.
  - **Binary Container "name":** Binary container name.
    - **Memory Data Transfers:** For each DDR Bank, this shows the trace of all the read and write request transactions arriving at the bank from the host.
    - **Kernel "name" 1:1:1:** For each kernel and for each compute unit of that kernel, this section breaks down the activities originating from the compute unit.
      - **Compute Unit: "name":** Compute unit name.
      - **CU Stalls (%):** Stall signals are provided by Vitis HLS to inform you when a portion of the circuit is stalling because of external memory accesses, or internal streams (that is, dataflow). The stall bus shown in detailed kernel trace compiles all of the lowest level stall signals and reports the percentage that are stalling at any point in time. This provides a factor of how much of the kernel is stalling at any point in the simulation.

        For example, if there are 100 lowest level stall signals and 10 are active on a given clock cycle, then the CU Stall percentage is 10%. If one goes inactive, then it is 9%.
      - **Data Transfers:** This shows the read/write data transfer accesses originating from each Master AXI port of the compute unit to the DDR.
      - **User Functions:** This information is available for the HLS kernels and shows the user functions.
        - **Function: "name":** Function name.
          - **Dataflow/Pipeline Activity:** This shows the number of parallel executions of the function if the function is implemented as a dataflow process.
            - **Active Iterations:** This shows the currently active iterations of the dataflow. The number of rows is dynamically incremented to accommodate the visualization of any concurrent execution.
            - **StallNoContinue:** This is a stall signal that tells if there were any output stalls experienced by the dataflow processes (function is done, but it has not received a continue from the adjacent dataflow process).
            - **RTL Signals:** These are the underlying RTL control signals that were used to interpret the above transaction view of the dataflow process.
          - **Function Stalls:** Shows the different types of stalls experienced by the process.

    - **External Memory:** Stalls experienced while accessing the DDR memory.
    - **Internal-Kernel Pipe:** If the compute units communicated between each other through pipes, then this shows the related stalls.
  - **Intra-Kernel Dataflow:** FIFO activity internal to the kernel.
  - **Function I/O:** Actual interface signals.
- **HLS FIFO:**
  - This shows waveform for size of HLS FIFOs created inside non-RTL kernels
  - The waveform is in Analog style
  - It shows one entry for each kernel instance which has FIFO in it
  - The analog waveform is produced by tracing on an internal HDL signal of the kernel which gives the current number of elements in the FIFO during simulation.
  - **CU name:** Name of the CU containing FIFO
  - **FIFO instance name:** Name of the FIFO instanc
  - **mOutPtr["size:"0]:** HDL signal which gives the number of elements currently in the FIFO during simulation

Interpreting TLM Waveform Data for Third-Party Simulators

1. Under the respective design hierarchy in the waveform windows, for each TLM—TLM socket connection, the following information is visible as waveforms.
   a. For memory mapped AXI4 interfaces, the bus transaction is visible as six channels:
      - `<socket_name>`: This channel contains statistics such as whether transaction is read/write and a unique mark to differentiate information in sub channels. This also indicates the number of transactions completed.
      - `<socket_name>_AW`: This channel contains the transaction information of Write Address.
      - `<socket_name>_W`: This channel contains the transaction information of Write Data.
      - `<socket_name>_B`: This channel contains the transaction information of the corresponding Write Response.
      - `<socket_name>_AR`: This channel contains the transaction information of Read Address.
      - `<socket_name>_R`: This channel contains the transaction information of Read Data.

      Detailed attributes for each channel like burst size, burst type, response, etc. are visible as attributes in each channel.

b. For AXI4-Stream, the bus transaction is visible in one channel only named after the `socket_name`. This contains the information like TID, TDEST, TDATA, etc. as attributes.

***Note*:** TLM waveform viewing is only supported for Questa Advanced Simulator and Xcelium. The information on the usage of waveform, adding socket to waveform view, and detailed view of attributes can be referred to its respective third-party simulator user guide.

AXI Transactions Display in XSIM Waveform

Many models in hardware emulation use SystemC transaction-level modeling (TLM). In these cases, interactions between the models cannot be viewed as RTL waveforms. However, Vivado simulator (xsim) provides a transaction level viewer. For standard platforms, these interface objects can be added to the waveform view, similar to how RTL signals are added. As an example, to add an AXI interface to the waveform, use the following Tcl command in `xsim`:

```
add_wave <HDL_objects>
```

Using the `add_wave` command, you can specify full or relative paths to HDL objects. For additional details on how to interpret the TLM waveform see Interpreting TLM Waveform Data for Third-Party Simulators, or refer to *Vivado Design Suite User Guide: Logic Simulation* (UG900).

Generating Test Vectors for Vitis HLS during Hardware Emulation

It is possible to instruct the Vitis tool to generate test vectors for simulation during hardware emulation, without re-running `v++` compilation and linking. The test vectors will enable Vitis HLS to run C/RTL Co-simulation without a dedicated C++ test bench for:

- Deadlock analysis
- FIFO depth optimization
- Other performance optimizations

Use the following steps:

1. Create an `hlsPre.tcl` file and insert this command:

   ```
   config_export -cosim_trace_generation
   ```

2. Run `v++ --compile` and keep the HLS project directories, under the `<compile_dir>`
3. Run `v++ --link --target hw_emu`
4. Run the application for hardware emulation
5. Locate the `hls_cosim` inside the HW_EMU run directory

a. This directory contains one directory for each kernel, with one directory for each kernel instance below it:

```
<build_dir>/.run/<run_number>/hw_em/device0/binary_0/behav_waveform/
xsim/hls_cosim/<kernel_name>
```

6. Copy the appropriate kernel directory to the HLS project directory, i.e.:

```
cp -r <build_dir>/.run/<run_number>/.../xsim/hls_cosim/<kernel_name>
<compile_dir>/<kernel_name>/<kernel_name>
```

7. Open the Vitis HLS tool and run C/RTL Co-simulation in batch mode or GUI mode:

```
cosim_design -hwemu_trace_dir <kernel_name>/<instance_name> ...
```

The traces generated from HW_EMU are valid only as long as:

- The functionality of the kernel does not change
- The top interface of the kernel does not change
- The number of top interface reads and writes (`s_axilite` registers, `m_axi` interfaces, `axis` interfaces) does not change.

Profile and Debug in Hardware Emulation

While running emulation, you can specify a number of profile options as described in Enabling Profiling in Your Application to capture design data during runtime. Any reports generated during the run are collected into the `xrt.run_summary` file. This collection of reports can be viewed by opening the run_summary in Vitis analyzer, and includes a Summary report, System and Platform Diagrams to illustrate the hardware design, Run Guidance offering any suggestions for improving the performance of the system, and a Profile Summary and Timeline Trace when enabled in the `xrt.ini` file during runtime. Refer to Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702) for additional information.

The following table shows some specific techniques for debugging different scenarios in hardware emulation.

Table 36: **Techniques for Debugging in Hardware Emulation**

Debug Focus	Description	Steps
x86 host (XRT)	Enable detailed XRT logs by updating `xrt.ini`.	Add the following to `xrt.ini` file. Run the test case with the updated `xrt.ini`. Review the generated `xrt_hal.log`. `[runtime]hal_log=xrt_hal.log`

Table 36: **Techniques for Debugging in Hardware Emulation** *(cont'd)*

Debug Focus	Description	Steps
AXI traffic on SystemC models like AI Engine, NOC, CIPS	The host transactions are routed through `sim_qdma SystemC` module. These transactions can be dumped into log file. If PL is also SystemC, then even PL transactions can be viewed there. If PL is RTL, then boundary of PL can be viewed in waveform.	In `xrt.ini` add, `[Emulation]xtlm_aximm_log=true xtlm_axis_log=true` View file `xsc_report.log`.
Viewing DDR memory content	The DDR model saves its contents in a binary file. In the folder where simulation is run (`package.hw_emu/sim/behav_waveform/xsim dir`), look for files named like below. Each such binary file corresponds to a region of DDR memory at a particular offset. `qemu-memory-_ddr@0x00000000` or `qemu-memory-_mem_0xc080000000@0xc080000000ULL` These files represent DDR/LPDDR memory contents in binary form. There is a backdoor connection between QEMU to NOC_DDR model. Thus, users will not see any transactions in the waveform nor will they see any logs. The shared memory is directly updated. To view memory contents, users can directly dump the memory contents.	The contents of the memory can be seen by using `hexdump` command. an example command is shown below. `hexdump qemu-memory-_mem_0x6000000000@0x60000000000ULL -s 0x80000000 -n 4096 -v -e '1/4 "%02X" "\n"' > dump_600.log`
PS (QEMU)	During firmware and software execution, the Arm APU can run into issues. You see details of various transactions and state on PS using these mechanisms.	1. Setenv variable `ENABLE_RP_LOGS=true` in the shell where QEMU is being run. Look for a log file named `sim/behav_waveform/xsim/qemu_rp.log`. This contains transactions from PS to rest of the peripherals (other than DDR). 2. Review `package.hw_emu/qemu_output.log` to see the output. This contains PS output to STDOUT (UART). If there is any error during PLM or other SW execution, those will be captured here. Look at the details of the error (CDO address, U-Boot stage etc) to narrow down which CDO is causing the error. 3. Run `launch_hw_emu.sh -enable_debug` mode which will direct QEMU logs in its own xterm window.

Table 36: **Techniques for Debugging in Hardware Emulation** *(cont'd)*

Debug Focus	Description	Steps
AI Engine	PDI is downloaded from PS to AI Engine through fast mode, thus transactions cannot be seen in the waveform. Users can enable logging all transactions at AI Engine AXI interfaces. There is a separate file per AXI interface	1. Enable `setenv ENABLE_AIE_DBG_TRACE`. View transaction log in the simulation folder at `aie_log/S00_AXI.txt` file. 2. Enable AI Engine VCD Dump and view contents in Vitis Analyzer by adding switch `-aie-sim-options` to `launch_hw_emu.sh` cmd line. See UG1076 for details of AI Engine-sim-options file. These are common between `aiesim` and `hw_emu`. 3. Create a file (`aie_sim_config.txt`) to enable AI Engine simulation options. a. In this file, add `"AIE_DEBUG_AXIMM=True"` b. Pass this file on launch emulator cmd line: `./package.hw_emu/ launch_hw_emu.sh -aie-sim-options <Absolute path to the options .txt file>`
Dumping Waveform in DC flow (Batch Mode)	Make sure the design is linked with the `-g` option and you have run the design at least once in Xsim GUI mode to save the required `.wcfg` file and save the list of relevant signals. Dump the waveform and open it in xsim along with saved `.wcfg` file	Update following options in `xrt.ini` option `[Emulation] user_pre_sim_script=<use pre sim script absolute path>` Pre-simulation script is needed to enable signal dumping: `log_wave -r *`

Using I/O Traffic Generators

Some user applications such as video streaming and Ethernet-based applications make use of I/O ports on the platform to stream data into and out of the platform. For these applications, performing software and hardware emulation of the design, or running AI Engine simulation, requires a mechanism to mimic the hardware behavior of the I/O port, and to simulate data traffic running through the ports. I/O traffic generators let you model traffic through the I/O ports during software and hardware emulation in the AMD Vitis™ application development flow, during the AI Engine simulation flows (x86sim, AI Enginesim), or during logic simulation in the AMD Vivado™ Design Suite.

IMPORTANT! *Hardware emulation supports both AXI4-Stream and AXI4 memory map interface I/O emulation.*

Traffic generators can be written in Python, MATLAB, C/C++, or RTL (Verilog/SV) modules. They are launched in an external process that communicates with Vitis Emulation process or AI Engine simulation process using Inter Process Communication (IPC). The IPC connections are established using IPC AXI4-Stream master/slave modules as described in Running Traffic Generators in Python/C++.

Traffic generators are designed to pass data into your system, or receive data from your system. The APIs are provided to handle data transfer for standard datatypes, or for more complex datatypes. The API you use to create the traffic generator is determined by the datatype your system requires. For instance, standard datatypes are covered by simple API such as send_data, or receive_data as described in Python API for AI Engine Graphs or Python API for PL Kernels. More complex datatypes require API like those described in General Purpose Python API.

The following are additional details on ways to integrate traffic generators in Python/C/C++/Verilog with subsequent PL kernels or the AI Engine kernels based on your application.

Adding Traffic Generators to Your Design

AXI Traffic Generator kernels provide a method to inject traffic onto the I/O of your system design, AI Engine graph, or PL kernels during simulation. AMD provides a library that enables interfacing AXI4-Stream to mimic a streaming data flow for software and hardware emulation, and AXI4 memory mapped interface to mimic memory mapped data transfers for hardware emulation.

The AXI Traffic generators are provided as XO files which can be linked into your System project using the Vitis compiler (`v++`). These XO files are called `sim_ipc_axis_master_XY.xo` and `sim_ipc_axis_slave_ZW.xo` where XY and ZW correspond to the number of bits in the PLIO interface. For example, `sim_ipc_axis_master_128.xo` provides an AXI4-Stream master data bus that is 128 bits wide. A wider interface allows the PL to achieve the same

throughput at a lower clock frequency and allows the AI Engine array to maximize its memory bandwidth. However, the PLIO interface tiles are each 64 bits wide and they are a limited resource. Using one 64-bit PLIO interface at twice the clock speed provides an equivalent bandwidth to a 128-bit PLIO while using only one PLIO tile. This requires the PL to run at twice the clock speed and the optimal choice will vary from application to application.

Two steps are required to use the traffic generators in your system design:

1. Specify the connections between the traffic generator (`sim_ipc`) modules and their corresponding AXI4-Stream ports on the AI Engine array. This is typically done in the `system.cfg` file using the `--connectivity.nk` and `--connectivity.sc` commands, as described in Linking the System. The following is an example:

   ```
   [connectivity]
   nk=sim_ipc_axis_master:1:inst_sim_ipc_axis_master
   nk=sim_ipc_axis_slave:1:inst_sim_ipc_axis_slave
   stream_connect=sim_ipc_axis_master.M00_AXIS:ai_engine_0.DataIn
   stream_connect=ai_engine_0.DataOut:sim_ipc_axis_slave.S00_AXIS
   ```

   The syntax for connecting the `sim_ipc_axis` XO files is as follows.

   ```
   nk=sim_ipc_axis_master:<Number Of
   Masters>:<inst_name_1>.<inst_name_2>.<...>
   nk=sim_ipc_axis_slave:<Number Of
   Slaves>:<inst_name_1>.<inst_name_2>.<...>
   ```

   Where:

   - The `sim_ipc_axis_master/slave` specifies the XO kernel in your design
   - The `<Number Of Masters>` or `<Number Of Slaves>` field lets you specify up to 8 different traffic generator kernels in your design
   - The `<inst_name>` should be meaningful in your application

2. Next, add the XO files to the Vitis link command as shown below.

   IMPORTANT! *The traffic generator XO can only be used in hardware emulation with `hw_emu` target.*

   ```
   v++ -l --platform <platform.xpfm> sim_ipc_axis_master_128.xo
   sim_ipc_axis_slave_128.xo libadf.a -target hw_emu --config system.cfg
   ```

For additional information on how to use XO files with the Vitis compiler, see Building the Device Binary in the *Data Center Acceleration using Vitis* (UG1700).

Using Traffic Generators for AI Engine Designs

Overview

This section describes how to provide input and capture the output from the AI Engine array in all simulation and emulation modes using AXI traffic generators. In the AI Engine simulator, the input data stimulus is provided using the PLIO object which specifies a text file containing the data:

```
input_plio plin = input_plio::create("DataIn", adf::plio_32_bits, "data/
input.txt");
```

Although this is a fast process to get your first simulation in place, the main limitation of this approach is that if you want to change the input file name for another simulation, you need to recompile the entire application. To avoid file name specification and rely on the independent External Traffic Generator to generate data traffic on the PLIO, see below:

```
input_plio plin = input_plio::create("DataIn", adf::plio_32_bits);
```

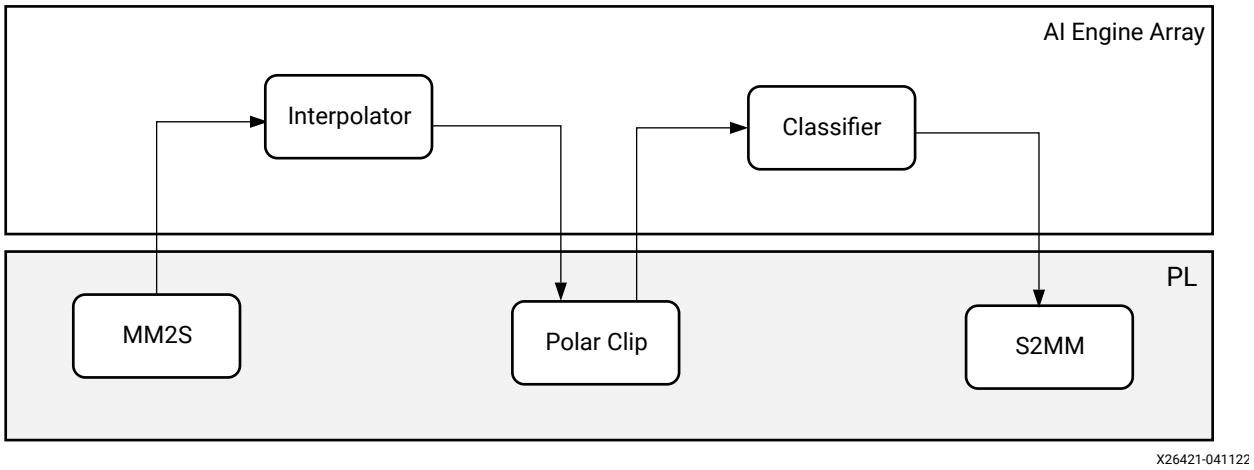
For hardware emulation, an equivalent feature exists that emulates the behavior of this PLIO and AXI4-Stream interface. Both Python and C++ APIs are provided to create these External Traffic Generators that will be connected seamlessly on any of these simulation or emulation modes.

The primary external data interfaces for the AI Engine array are AXI4-Stream interfaces. These are known as PLIOs and allow the AI Engine to receive data, operate on the data, and send data back on a separate AXI4-Stream interface. The input interface to the AI Engine is an AXI4-Stream consumer, and the output is an AXI4-Stream producer. To interact with these top level interfaces during hardware emulation complementary AXI4-Stream modules are provided. These complementary modules are referred to as the AXI traffic generators.

TIP: *The width of a PLIO interface is an important system level design decision. The wider the interface the more data can be sent per PL clock cycle.*

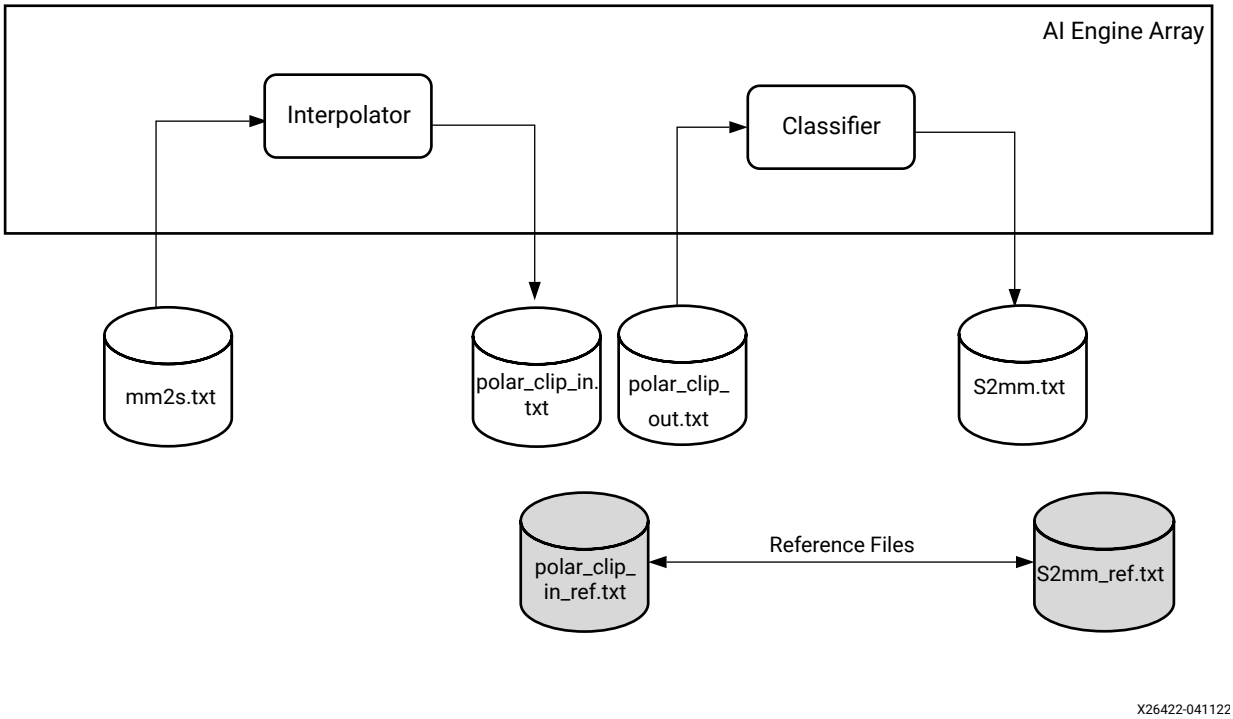
When developing an AI Engine application, you can test it standalone either in simulation (`x86simulator`, `aiesimulator`), or as part of a system project in emulation (`hw_emu`). In either case, you need to send the input data from a predefined reference file and capture the output data in a separate file. Furthermore, if your AI Engine graph is intertwined with kernels that are located in the Programmable Logic (HLS C++ or RTL) then you also have to deal with these data flow interruptions. For example, a full system design might look like the following figure:

Figure 40: **AI Engine + Programmable Logic Application**



In a first step you replace all the connections which are not in the AI Engine array with text files to provide input data or capture output data:

Figure 41: **Initial Simulation Framework**

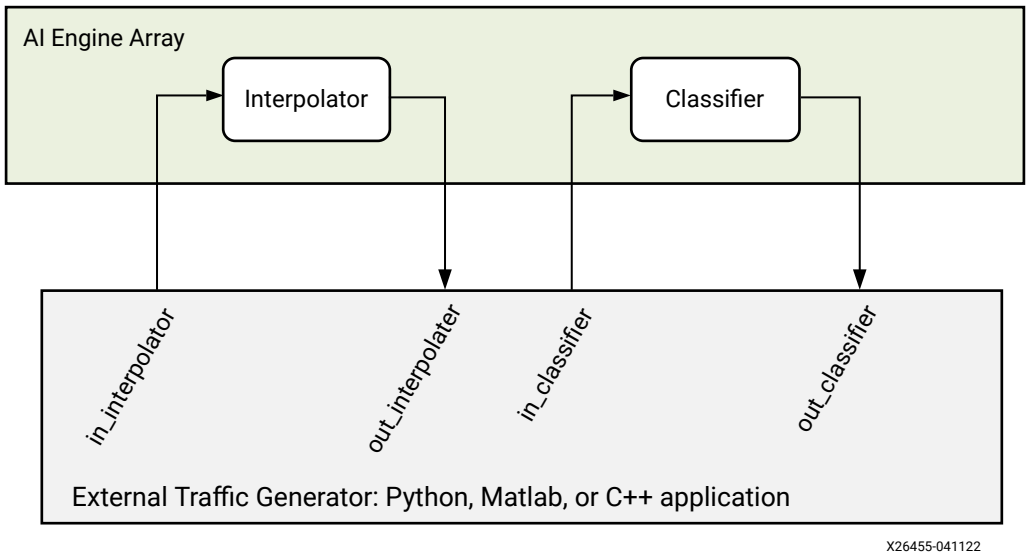


For more flexibility in data generation and verification you can exchange the text files with external traffic generators which enable dynamic simulated communication between the PL and the AI Engine array through AXI4-Stream TLM connected to Unix sockets. The power of these external traffic generator is that they can be used in all simulation/emulation framework without modification:

- x86 Simulation
- AI Engine simulation
- HW Emulation

The overall simulation framework is illustrated in the following figure:

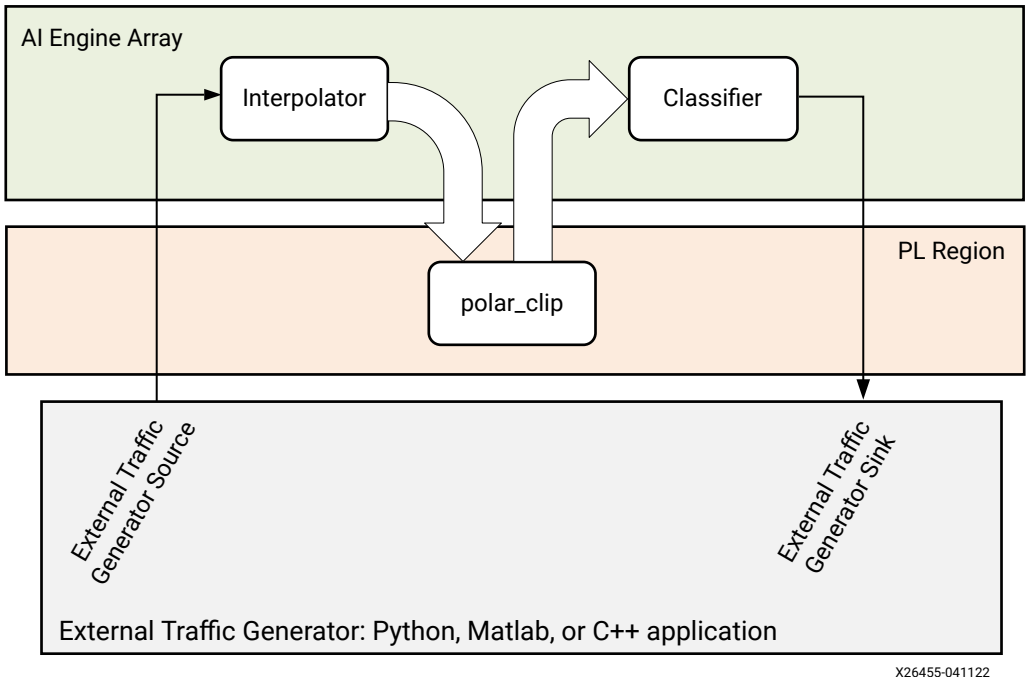
Figure 42: **External Traffic Generator-Based AI Engine Simulation Flow**



Each AI Engine block can be validated using an external test bench written in Python, MATLAB®, or C++.

For system projects, incorporating AI Engine graph applications and PL kernels, the data movers are replaced with external Traffic Generators source and sink, and the PL processing kernel is a streaming kernel connected to the AI Engine kernels. See the following figure for details.

Figure 43: **Full System Emulation**



The traffic generators are used to feed and flush the data into the full system with PL logic as well. You do not need to model the PS code writing data to DDR memory and model the data moving to the AI Engine kernels. This apporach replaces the data movers with external traffic generators dynamically producing data. The following sections describe the step-by-step changes that are needed to interface external traffic generators with an AI Engine system design for the emulation flow.

AI Engine Graph Modifications

Nothing has to be changed within the graph concerning the kernel connections. The definition of the traffic generators as the source of data from the PLIO port is the only change required as shown below. The example below is based on the code in *Design Flow Using RTL Programmable Logic* in *AI Engine Kernel and Graph Programming Guide* (UG1079).

```
plin = input_plio::create("DataIn1",adf::plio_32_bits);
clip_in = output_plio::create("clip_in",adf::plio_32_bits);
clip_out = input_plio::create("clip_out",adf::plio_32_bits);
plout = output_plio::create("DataOut1",adf::plio_32_bits);
```

The first parameter of the input/output plio declaration is important as this is the name that will be used on the traffic generator side to connect to the right socket.

x86 simulation and AI Engine simulation can be launched working with the traffic generators. Launching simulation requires running the `aiesimulator` or the `x86simulator` in parallel with the external traffic generator.

PL Kernels Change

When developing AI Engine applications for hardware emulation, you must model data transfers between AI Engine and programmable logic. However during initial development phase, the PL kernels are often unfinished and not ready to be used in Vitis link. The solution is to insert hooks in the programmable logic interface to connect to external traffic generators. AMD provides a complete set of pre-compiled `.xo` files that can be used for this purpose:

- `$(XILINX_VITIS)/data/emulation/XO/sim_ipc_axis_slave_32.xo, $(XILINX_VITIS)/data/emulation/XO/sim_ipc_axis_master_32.xo`
- `$(XILINX_VITIS)/data/emulation/XO/sim_ipc_axis_slave_64.xo, $(XILINX_VITIS)/data/emulation/XO/sim_ipc_axis_master_64.xo`
- `$(XILINX_VITIS)/data/emulation/XO/sim_ipc_axis_slave_128.xo, $(XILINX_VITIS)/data/emulation/XO/sim_ipc_axis_master_128.xo`

The `.xo` files must be copied to the right location in your project and specified in the configuration file during the Vitis link stage.

Preparing Connectivity to Link the Traffic Generators

During the Vitis link stage (`v++ -l`), the previously defined `.xo` files will be used to connect the related kernel instances to the AI Engine graph. The `hw_link.cfg` configuration file is created in such a way that the kernel instance names matches the names you defined in the graph for the `input_plio` and the `output_plio`. For example, the code below matches the PLIO assignments in the example above:

```
[connectivity]

nk=sim_ipc_axis_master_32:1:in_interpolator
nk=sim_ipc_axis_slave_32:1:out_classifier
nk=polar_clip:1:polar_clip

sc=in_interpolator.M00_AXIS:ai_engine_0.in_interpolator
sc=ai_engine_0.out_interpolator:polar_clip.in_sample
sc=polar_clip.out_sample:ai_engine_0.in_classifier
sc=ai_engine_0.out_classifier:out_classifier.S00_AXIS
```

The format of the `--connectivity.nk` command is the kernel name such as `sim_ipc_axis_master_32`, the number of kernel instances to create, and the names of each kernel instance (`in_interpolator`). Refer to --connectivity Options for more information on the command.

The `--connectivity.sc` command defines the streaming connections between PL kernels, or between PL kernels and the AI Engine graph. In the example above the output port of the traffic generator `in_interpolator.M00_AXIS` is connected to the input port `ai_engine_0.in_interpolator`.

With this naming approach, the same external traffic generator can be used for multiple simulation or emulation runs. In the case of hardware emulation (`hw_emu`), you can write the external traffic generator in C++, Python, MATLAB, or HDL if familiar with RTL coding.

Host Code

The host code creation is relatively simple. As there are no programmable logic kernels, you can avoid all the stages where you look for and run the PL kernels as well as the parts where you allocate memory for all the buffer objects. The stages are:

- Open the device
- Load the `xclbin` file
- Register XRT to connect to the design
- Run the AI Engine graph

After compiling the host code, you can package the entire project. Running the emulation consists of running the external traffic generator in parallel with the standard emulation launch.

Using Traffic Generators in AI Engine Graphs

Overview

This section describes the steps to interface external traffic generators with AI Engine components. Simulation using external traffic generators can be run by launching the simulator/emulator and the traffic generator in parallel. The traffic generators can be written either in Python, MATLAB, or in C++, using multi-threading capabilities of these languages. Then the AI Engine must be written to work with the traffic generator during simulation or emulation. The steps below describe interfacing the traffic generator into your AI Engine graph for standalone AI Engine simulation or the full system emulation flow (hw_emu).

In order to interface external traffic generators with AI Engine components, you need to use certain APIs in your external script or source code written in Python, MATLAB, or C++.

***Note*:** External Traffic Generators provides only the PL stream traffic injection and recording capabilities for AI Engine Simulation without any expectation of Cycle Accuracy or Cycle Locked simulation.

Using Python API

You must write a Python script using dedicated APIs to interface the external traffic generator process with the AI Engine simulation process running the AI Engine graph.

```
# Mandatory
import os, sys

import multiprocessing as mp
import threading
import struct
```

```
from aie_input_plio import aie_input_plio
from aie_output_plio import aie_output_plio

# Optional for ease of use
import numpy as np
import logging
```

The Python traffic generator uses API described in Writing Python Traffic Generators.

Use the following steps to integrate the external traffic generator data into the graph:

1. Modify the graph code to declare the PLIO, but do not specify the PLIO based data text file in the PLIO constructors. This is needed for the external traffic generator to work properly. Take a note of the names (first argument) of the PLIO constructors. These will be used to hook up the external traffic generators and the same name should be used for creating the ports in the external traffic generators:

   ```
   plin = input_plio::create("DataIn1",adf::plio_32_bits);
   plout = output_plio::create("DataOut1",adf::plio_32_bits);
   ```

2. Instantiate the corresponding AXI master and slave connections in the Python script. This will establish connections to the PLIO ports of the graph:

   ```
   pl_in = aie_input_plio("DataIn1", 'int16')
   pl_out = aie_output_plio("DataOut1", 'int16')
   ```

   TIP: *You need to specify the PLIO datatype during object creation.*

3. Send data to the AI Engine.

   The data to be sent to the PLIO port in the AI Engine graph must be set up in the Python script. Using the `send_data()` API, you can send the data in the form of a list.

   ```
   plin.send_data send_data(<data_list>, tlast)
   ```

4. Receive data from the AI Engine.

   Use the `receive_data_with_size()` API to receive data from the AI Engine. This API returns the received values in `recv_data` from the following example. This is a blocking API and will wait until expected bytes of data are received. For more information, refer to Writing Python Traffic Generators.

   ```
   recv_data = plout.receive_data_with_size(<expected_bytes>)
   ```

Using C++ Traffic Generator APIs

When using the C++ language to implement an external traffic generator, various headers are necessary to use some libraries in the external traffic generator source code. The headers useful for handling these libraries are:

```
# For the traffic generator
#include "xtlm_ipc.h"
#include <thread>
```

Also, the C++ traffic generator uses APIs available as part of `xtlm_ipc` sources. For a list of the APIs and their corresponding usage, see Writing Traffic Generators in C++.

The Makefile dependencies are:

```
# Libraries directories
PROTO_PATH=$(XILINX_VIVADO)/data/simmodels/xsim/<vivado_version>/
lnx64/6.2.0/ext/protobuf/
IPC_XTLM= $(XILINX_VIVADO)/data/emulation/ip_utils/xtlm_ipc/
xtlm_ipc_v1_0/cpp/src/
IPC_XTLM_INC= $(XILINX_VIVADO)/data/emulation/ip_utils/xtlm_ipc/
xtlm_ipc_v1_0/cpp/inc/
LOCAL_IPC= $(IPC_XTLM)../

LD_LIBRARY_PATH:=$(XILINX_VIVADO)/data/simmodels/xsim/<vivado_version>/
lnx64/6.2.0/ext/protobuf/:$(XILINX_VIVADO)/lib/lnx64.o/Default:$
(XILINX_VIVADO)/lib/lnx64.o/:$(LD_LIBRARY_PATH)

# Kernel directories
PLKERNELS_DIR := ../../pl_kernels
PLKERNELS := $(PLKERNELS_DIR)/polar_clip.cpp
PLHEADERS := $(PLKERNELS_DIR)/polar_clip.hpp $(PLKERNELS_DIR)/s2mm.hpp
$(PLKERNELS_DIR)/mm2s.hpp

# XTLM source files
IPC_SRC := $(LOCAL_IPC)/src/axis/*.cpp $(LOCAL_IPC)/src/common/
*.cpp $(LOCAL_IPC)/src/common/*.cc

# Compiler/linker flags
INC_FLAGS := -I$(LOCAL_IPC)/inc -I$(LOCAL_IPC)/inc/axis/ -I$(LOCAL_IPC)/inc/
common/ -I$(PROTO_PATH)/include/ -I$(PLKERNELS_DIR) -I$(XILINX_HLS)/include
LIB_FLAGS := -L$(PROTO_PATH)/ -lprotobuf -L$(XILINX_VIVADO)/lib/
lnx64.o/ -lrdizlib -L$(GCC)/../../lib64/ -lstdc++ -lpthread

# Compilation
compile: main.cpp $(PLHEADERS) $(PLKERNELS)
  $(GCC) -g main.cpp $(PLKERNELS) $(IPC_SRC)
$(INC_FLAGS) $(LIB_FLAGS) -o chain
```

Below are the steps to integrate external traffic generator using C++ APIs in the AI Engine component:

1. Declare the external PLIOs in the graph code as below:

   ```
   plin = input_plio::create("DataIn1",adf::plio_32_bits);
   plout = output_plio::create("DataOut1",adf::plio_32_bits);
   ```

2. Instantiate AXI master and sender.

```
xtlm_ipc::axis_master plin("DataIn1");
xtlm_ipc::axis_slave plout("DataOut1");
```

3. Prepare the data. This is the user logic.
4. Send the data.

   A simple API is available if you prefer not to have fine granular control and send the data.

```
std::vector<char> data; // The sender API expects data to be in the form
of vector of char
```

   The C++ XTLM library provides the utility conversion APIs to easily convert user readible data type to byte array:

```
std::vector<char> type
std::vector<char> byte_array = plin.uInt8ToByteArray(user_list)
```

   Here `uint8` is the user data type and contains user list/array for `uint8` type, such as `std::vector<uint8> user_list;`. For more details on conversion API, see Writing Traffic Generators in C++.

```
// Write a user logic to fill in the data
// Send the data using send_data() API call
plin.send_data(byte_array, tlast)
```

5. Receive the data.

   A simple API is available if you do not need fine grain control. It returns the data in the form of byte array (`std::vector<char>`) and waits until expected `data_size` is received.

```
std::vector<char> data; // create empty vector of char
plout.receive_data_with_size(data, data_size);
```

   The C++ library provides another set of utility conversion API to convert the received byte array into user readable data format. For example:

```
std::vector<int8> recv_data;
recv_data = plout .byteArrayTouInt8(data)
```

   For more details on the API and their usage in the external traffic generator CPP code, see Writing Traffic Generators in C++.

The interest of the C++ traffic generator is that you can use and test your HLS kernels as soon as they are created, without having to synthesize them in a `.xo` file. This allows you to add more and more realism and flexibility to your simulations without having to recreate a `.xclbin` file.

Using SystemVerilog Traffic Generator APIs

You can also drive traffic generators from external System Verilog/Verilog traffic generators and test benches to the `aiesimulator` or `x86simulator`.

Prior to integrating a traffic generator module, declare the external PLIOs in the graph.

```
pl_in0 = adf::input_plio::create("in_classifier",adf::plio_32_bits);
out0 = adf::output_plio::create("out_interpolator",adf::plio_32_bits);
```

To establish the connection between the SystemVerilog/Verilog traffic generator and the AI Engine graph's external PLIOs, the `xtlm_ipc` SystemC modules are required. The external AI Engine wrapper is generated based on the external PLIO declarations in the ADF graph. Follow the steps below to generate this wrapper module for the AI Engine.

1. Use the following command to perform the ADF graph compilation to generate a `scsim_config.json` file that resides in the `work/config/scsim_config.json` directory. This config file contains information on the PLIOs declared in the graph.

   ```
   v++ -c --mode aie --target hw \
     --platform vek385_base \
     --work_dir ./newAIE/hw/Work \
     --config --config ./config.cfg aie/graph.cpp
   ```

2. This config file is passed as an argument to the python script available inside `${XILINX_VITIS}/data/emulation/scripts/gen_aie_wrapper.py` to generate the Verilog-based AI Engine wrapper module. For more details on how to generate the AI Engine wrapper module, see External RTL Traffic Generator and AI Engine Simulation.

3. After generating the AI Engine wrapper module, you need to instantiate it in an external test bench to make the connection between the System Verilog/Verilog traffic generator and the AI Engine graph PLIOs. For details on integrating the wrapper module into the external RTL test bench, see Instantiating AI Engine Wrapper in the Test Bench.

After the connection is established, you can launch the HDL simulation in parallel with AI Enginesimulation or x86 simulator. For details on how to launch the HDL simulation, see Running the RTL Traffic Generator with AI Engine Simulation.

AXI4-Stream I/O Model for Streaming Traffic

The following section is specific to AXI4-Stream. The streaming I/O model can be used to emulate streaming traffic on the platform, and also support delay modeling. You can add streaming I/O to your application when targeted for hardware emulation, or add them to your custom platform design in the context of hardware emulation as described below:

- Streaming I/O kernels can be added to the device binary (`xclbin`) file like any other compiled kernel object (XO) file, using the `v++ --link` command. Using these pre-compiled XO files reduces `v++` compile time. The Vitis installation provides kernels for AXI4-Stream interfaces of various data widths. The standard bit widths supported are 8, 16, 32, 64, 128, 256, 512. These can be found in the software installation at `$XILINX_VITIS/data/emulation/XO`.

Add these to your designs using the following example command:

```
v++ -t hw_emu --link $XILINX_VITIS/data/
emulation/XO/sim_ipc_axis_master_32.xo $XILINX_VITIS/data/emulation/XO/
sim_ipc_axis_slave_32.xo ...
```

In the example above, the `sim_ipc_axis_master_32.xo` and `sim_ipc_axis_slave_32.xo` provide 32-bit master and slave kernels that can be linked with the target platform and other kernels in your design to create the `.xclbin` file for the emulation build.

- In case of hardware emulation, IPC modules can also be added to a platform block design using the Vivado IP integrator for AMD Versal™ and AMD Zynq™ UltraScale+™ MPSoC custom platforms. The tool provides `sim_ipc_axis_master_v1_0` and `sim_ipc_axis_slave_v1_0` IP to add to your platform design. These can be found in the software installation at `$XILINX_VIVADO/data/emulation/hw_em/ip_repo`.

  The following is an example Tcl script used to add IPC IP to your platform design, which will enable you to inject data traffic into your simulation from an external process written in Python or C++:

```
#Update IP Repository path if required
set_property  ip_repo_paths $XILINX_VIVADO/data/emulation/hw_em/ip_repo
[current_project]
## Add AXIS Master
create_bd_cell -type ip -vlnv xilinx.com:ip:sim_ipc_axis_master:1.0
sim_ipc_axis_master_0
#Change Model Property if required
set_property -dict [list CONFIG.C_M00_AXIS_TDATA_WIDTH {64}]
[get_bd_cells sim_ipc_axis_master_0]

##Add AXIS Slave
create_bd_cell -type ip -vlnv xilinx.com:ip:sim_ipc_axis_slave:1.0
sim_ipc_axis_slave_0
#Change Model Property if required
set_property -dict [list CONFIG.C_S00_AXIS_TDATA_WIDTH {64}]
[get_bd_cells sim_ipc_axis_slave_0]
```

Writing Python Traffic Generators

Introduction

You can include an external traffic generator process while simulating your application to dynamically generate data traffic on the I/O traffic generators, or to capture output data from the emulation process. The AMD provided Python library can be used to create the traffic generator code as described in the following sections.

The Python library having the API is divided into API for sending and receiving user data on instantiated traffic generator objects. There are also advanced APIs for packet level granularity if you need more control over transactions. For seamless interaction of traffic generators with simulation process, the API gives you control over sending or receiving the data without worrying too much about managing packet level details like `TLAST`, `TKEEP`, `TSRB`, etc. This is auto managed by the `sim_IPC` infrastructure when using the API. Also, an application can communicate to multiple I/O interface. It is not necessary to have each instance of I/O utilities to be in a separate process/thread. If your application requires it, you might consider the non-blocking API to send the data.

The Python API are categorized as follows:

- Python API for AI Engine Graphs
- Python API for PL Kernels
- General Purpose Python API

Python API for AI Engine Graphs

Here is the list of AI Engine specific APIs and their usage in integrating the traffic generators. With the Python API you can create the traffic generator code to generate data to pass into or collect data from your AI Engine graph. Use the following API to instantiate objects to send and receive data. You can provide any datatype vector/list to `send_data` or `receive_data`.

- **Instantiating Classes to Send or Receive Data:**

```
aie_input_plio( name, datatype)
aie_output_plio(name, datatype)
```

Parameters:

- name: A string value that should match a PLIO name from the graph.
- datatype: A string with the following supported values `[ "int8", "uint8","int16", "uint16","int32", "uint32","int64", "uint64","float", "bfloat16"]`

Table 37: **Supported AI Engine Data Types**

#num	AIE Kernel Datatype	API Enum String	Usage	User Value List Representation (In Python)
1	int8	int8	input_port = aie_input_plio('input_port_name',"int8")	data = [12 23 2 -2 23]
2	int16	int16	input_port = aie_input_plio('input_port_name',"int16")	data = [1212 121 232 -2323]
3	int32	int32	input_port = aie_input_plio('input_port_name',"int32")	data = [121 23234 2323232 23 -878787]

Table 37: **Supported AI Engine Data Types** *(cont'd)*

#num	AIE Kernel Datatype	API Enum String	Usage	User Value List Representation (In Python)
4	int64	int64	input_port = aie_input_plio('input_port_name',"int64")	data = [232 2342232 -23482947 238273]
5	uint8	uint8	input_port = aie_input_plio('input_port_name',"uint8")	data = [23 23 12]
6	uint16	uint16	input_port = aie_input_plio('input_port_name',"uint16")	data = [236 23728 2378 8237]
7	uint32	uint32	input_port = aie_input_plio('input_port_name',"uint32")	data = [267 2376 2362736 232767362]
8	uint64	uint64	input_port = aie_input_plio('input_port_name',"uint64")	data = [347 2348 327 98932 872389]
9	float	float	input_port = aie_input_plio('input_port_name',"float")	data = [3.14 2.3234 3472.23]
10	bfloat16	bfloat16	input_port = aie_input_plio('input_port_name',"bfloat16")	data = [3.14 2.3234 3472.23] ***Note*:** Float and Bfloat16 representations in floating point are similar. But when transferred to AIE, float will be transmitted in 32 bits. bfloat16 will be transmitted in 16 bits.
11	cfloat	float	input_port = aie_input_plio('input_port_name', "float")	data = [3.14 2.2323 23.3 23.33] ***Note*:** From left, even positions will be real and odd positions will be imaginary.
12	cint16	int16	input_port = aie_input_plio('input_port_name',"int16")	data = [32 232 232 234] ***Note*:** From left, even positions will be real and odd positions will be imaginary.
13	cint32	int32	input_port = aie_input_plio('input_port_name',"int32")	data = [23 -23 23 -232] ***Note*:** From left, even positions will be real and odd positions will be imaginary.

Note: For some types like `cint32` you can use `int32` as it is expected that you will provide pairs of real and imaginary values in the same list. This means the size of the user list for complex types is always even.

For example, before creating external ports, you need to do AI Engine graph modifications as described at Using Traffic Generators in AI Engine Graphs. Once graph modifications are done, you can create the sender and receiver objects for the AI Engine in your python script.

```
input = aie_input_plio("in_data", 'int16')
output = aie_output_plio("out_data", 'int16')
```

Here, `in_data` and `out_data` define the PLIO matching in the graph code; `input` and `output` are the objects created.

The second parameter `int16` is the datatype of the interfaced AI Engine kernel with traffic generator.

- **send_data():**

```
send_data(data, tlast)
creates a non-blocking call to send data
```

Parameters:

- data: The list of specified datatype
- tlast: Boolean value, can be true (1) or false (0)

Note: The datatype must be specified during object instantiation.

The following is an example of creating the traffic generator object and sending data through it:

```
input = aie_input_plio("in_data", "uint32")
input.send_data(<data_list>, "false")
```

This API call sends the data to the AI Engine kernel via "in_data" PLIO connected to the traffic generator.

- **receive_data():**

```
receive_data()
creates a blocking call to receive data
```

RETURNS a list of specified datatype

Note: The datatype must be specified during object instantiation

For example:

```
recv_data = output.receive_data()
```

`recv_data` is a list containing the returned value of `receive_data()`

- **receive_data_with_size():**

```
receive_data_with_size(data_size)
creates a blocking call to receive a specified amount of data
```

Parameters:

- data_size: integer value indicating the amount of data in bytes to receive

RETURNS a list of specified datatype

***Note*:** Data size is specified in bytes only.

For example:

```
recv_data = output.receive_data_with_size(1024)
```

This is a blocking API which blocks until the specified bytes (1024 bytes) of data is received.

- **receive_data_on_tlast():**

```
receive_data_on_tlast()
creates a blocking call returning data after receiving tlast packet
```

RETURNS a list of specified datatype

***Note*:** The datatype must be specified during object instantiation.

For example:

```
recv_data = output.receive_data_on_tlast()
```

`recv_data` contains the returned data from `receive_data_on_tlast()`. This is a blocking API which will wait until it gets the TLAST signal.

Python API for PL Kernels

Here is the list of PL kernel specific APIs and their usage in integrating the traffic generators. With the Python API you can create the traffic generator code to generate data to pass into or collect data from PL kernels. Use the following API to instantiate objects to send and receive data. You can provide any datatype vector/list to `send_data` or `receive_data`.

- **Instantiating Classes to Send or Receive Data:**

The APIs are found in the following library path:

```
${XILINX_VIVADO}/data/emulation/python/xtlm_ipc_v2/
```

You need to set `PYTHONPATH` to point to this library.

```
export PYTHONPATH=${XILINX_VIVADO}/data/emulation/python/xtlm_ipc_v2/
```

```
hls_input_stream(name, datatype)
hls_output_stream(name, datatype)
```

Parameters:

- name: string name to match the HLS kernel
- datatype: A string value based on HLS kernel datatype. The supported values are `[ "int8", "uint8","int16", "uint16","int32", "uint32","int64", "uint64","float", "bfloat16"]`

Table 38: **Supported PL Kernel Data Types**

	PL Kernel Data Type	EOU API Enum String	Example
1	int8	int8	input_port = hls_input_plio('input_port_name',"int8")
2	int16	int16	input_port = hls_input_plio('input_port_name',"int16")
3	int32	int32	input_port = hls_input_plio('input_port_name',"int32")
4	int64	int64	input_port = hls_input_plio('input_port_name',"int64")
5	uint8	uint8	input_port = hls_input_plio('input_port_name',"uint8")
6	uint16	uint16	input_port = hls_input_plio('input_port_name',"uint16")
7	uint32	uint32	input_port = hls_input_plio('input_port_name',"uint32")
8	uint64	uint64	input_port = hls_input_plio('input_port_name',"uint64")
9	float	float	input_port = hls_input_plio('input_port_name',"float")
10	double	double	input_port = hls_input_plio('input_port_name',"double")

- **send_data():**

```
send_data(data, tlast)
creates a non-blocking call to send data
```

Parameters:

- data: list of specified datatype for the object
- tlast: boolean value, can be true or false

***Note*:** The datatype must be specified during object instantiation

- **receive_data():**

```
receive_data()
creates a blocking call to receive data
```

RETURNS a list of specified datatype

***Note*:** The datatype must be specified during object instantiation

- **receive_data_with_size():**

```
receive_data_with_size(data_size)
creates a blocking call to receive a specified amount of data
```

RETURNS a list of specified datatype

Parameters:

- data_size: integer value indicating the amount of data in bytes to receive

***Note*:** Data size must be specified in bytes.

- **receive_data_on_tlast():**

```
receive_data_on_tlast()
creates a blocking call returning data after receiving tlast packet
```

RETURNS list of specified data type

***Note*:** The data type must be specified during object instantiation.

General Purpose Python API

Here is the list of general purpose APIs that can be used with custom user-defined datatypes. With these API you can send and receive data in the form of `byte_arrays` only. You must convert your custom datatype to `byte_arrays` prior to transport, and convert the received value back to your custom datatype for use by your application. Conversion API are provided as described below.

- **Instantiating Classes to Send or Receive Data:**

  The APIs are found in the following library path:

```
${XILINX_VIVADO}/data/emulation/python/xtlm_ipc_v2/
```

You need to set PYTHONPATH to point to this library:

```
export PYTHONPATH=${XILINX_VIVADO}/data/emulation/python/xtlm_ipc_v2/
```

```
input_port = axis_master(name)
output_port = axis_slave(name)
```

Parameters:

- `name`: string matching the AXI4-Stream interface

- **send_data():**

> **TIP:** *The general purpose API expects data in the form of `byte_array` only. To convert it from a user data type, use the conversion API such as `uInt16ToByteArray` described below.*

```
send_data(data, tlast)
creates a non-blocking call to send data
```

RETURNS nothing

Parameters:

- `data`: list created using create_byte_array() or conversion API described below
- `tlast`: boolean value, can be true or false

For example:

```
input.send_data(data_byte_array, tlast)
```

- **receive_data():**

> **TIP:** *The general purpose API expects data in the form of `byte_array` only. To convert it into a user readable format with data types after receiving it, use the conversion API such as `byteArrayTouInt16` described below.*

```
receive_data(data)
creates a blocking call to receive data
```

RETURNS nothing

Parameters:

- `data`: a byte array that can be converted with conversion API described below

- **receive_data_with_size():**

```
receive_data_with_size(data, data_size)
creates a blocking call to receive a specified amount of data
```

RETURNS a list of specified datatype

Parameters:

- `data`: a byte array that can be converted with conversion API described below
- `data_size`: integer value indicating the amount of data in bytes to receive

Note: Data size is specified in bytes

For example:

```
output.receive_data_with_size(<recv_vector>, 512)
```

Where `recv_vector` is an empty byte array that gets filled with the data received in the form of a byte array

- **receive_data_on_tlast():**

```
receive_data_on_tlast(data)
creates a blocking call returning data after receiving tlast packet
```

RETURNS a list of specified datatype

Parameters:

- data: a byte array that can be converted with the conversion API described in the following table.

Table 39: **Byte_Array Conversion API**

API	Description
byte_array = input_port.uInt8ToByteArray(user_list) byte_array = input_port.uInt16ToByteArray(user_list) byte_array = input_port.uInt32ToByteArray(user_list) byte_array = input_port.uInt64ToByteArray(user_list) byte_array = input_port.int8ToByteArray(user_list) byte_array = input_port.int16ToByteArray(user_list) byte_array = input_port.int32ToByteArray(user_list) byte_array = input_port.int64ToByteArray(user_list) byte_array = input_port.floatToByteArray(user_list) byte_array = input_port.doubleToByteArray(user_list) byte_array = input_port.bfloat16ToByteArray(user_list)	Convert the specified data type to byte_array value to send the data `byte_array = input_port.uInt64ToByteArray(user_list)` `Parameters:` `  Returns list of specified data type converted from byte_array` `  user_list → std::vector<T>` `  // T is the data type present in function signature.` `  // For example in byteArrayToFloat user_list is a vector of type float` `  byte_array → list created using create_byte_array or conversion APIs`

Table 39: **Byte_Array Conversion API** *(cont'd)*

API	Description
user_list = output_port.byteArrayTouInt8(byte_array) user_list = output_port.byteArrayTouInt16(byte_array) user_list = output_port.byteArrayTouInt32(byte_array) user_list = output_port.byteArrayTouInt64(byte_array) user_list = output_port.byteArrayToInt8(byte_array) user_list = output_port.byteArrayToInt16(byte_array) user_list = output_port.byteArrayToInt32(byte_array) user_list = output_port.byteArrayToInt64(byte_array) user_list = output_port.byteArrayToFloat(byte_array) user_list = output_port.byteArrayToDouble(byte_array) user_list = output_port.byteArrayToBfloat16(byte_array)	Convert the `byte_array` value to the specified data type after receiving `user_list = output_port.byteArrayTouInt16(byte_array)` `Parameters:` `  Returns list of specified data type converted from byte_array` `  byte_array → list created using create_byte_array or conversion APIs` `  user_list → std::vector<T>` `  // T is the data type present in function signature.` `  // For example in byteArrayToFloat user_list is a vector of type float`

Writing Traffic Generators in C++

Introduction

You can include an external traffic generator process while simulating your application to dynamically generate data traffic on the I/O, or to capture output data from the emulation process. The AMD provided C++ library can be used to create the traffic generator code as described in the following sections.

The C++ library having the API is divided into API for sending/receiving data in the form of byte array. The user data before sending needs to be converted into byte array and the data is received in the form of byte array which can be further converted to user data type using conversion APIs. There are also advanced API for packet level granularity if you need more control over transactions. For seamless interaction of traffic generators with simulation process, the API gives you control over sending or receiving the data without much worrying about managing packet level details like `TLAST`, this is auto managed by the `sim_IPC` infrastructure.

For C++, the APIs are available at:

```
$XILINX_VIVADO/data/emulation/cpp/inc/xtlm_ipc/axis/
```

You can build the executable with the following include path:

```
-I $XILINX_VIVADO/data/emulation/cpp/inc/xtlm_ipc/axis/
```

and linked against library as:

```
-L $XILINX_VIVADO/data/emulation/cpp/lib/
```

And use `-lxtlm_ipc` with a `gcc` compiler.

A full system-level example is available in the External_IO_CPP tutorial on GitHub.

General Purpose C++ API

Here is the list of general purpose APIs that can be used with custom user-defined datatypes. With these API you can send and receive data in the form of `byte_arrays` only. You must convert your custom datatype to `byte_arrays` prior to transport, and convert the received value back to your custom datatype for use by your application. Conversion API are provided as described below.

- **Instantiating Classes to Send or Receive Data:**

```
xtlm_ipc::axis_master   input_port(std::string name)
xtlm_ipc::axis_slave      output_port(std::string name)

Parameters:
  name → string matching the AXI4-Stream interface
```

- **send_data():**

> **TIP:** *The general purpose API expects data in the form of `byte_array` only. To convert it from a user data type, use the conversion API such as `uInt16ToByteArray` described below.*

```
send_data(data, tlast)
creates a non-blocking call to send data

Parameters:
  RETURNS nothing
  data → std::vector<char>
  tlast → boolean value, can be true or false
```

- **receive_data():**

> **TIP:** *The general purpose API expects data in the form of `byte_array` only. To convert it into a user readable format with data types after receiving it, use the conversion API such as `byteArrayTouInt16` described below.*

```
receive_data(data)
creates a blocking call to receive data

Parameters:
  RETURNS the received data as a byte_array
  data → std::vector<char>
```

The following example creates an empty vector for data and receives data:

```
std::vector<char> recv_data;
output.receive_data(<recv_data>)
```

- **receive_data_with_size():**

```
receive_data_with_size(data, data_size)
creates a blocking call to receive a specified amount of data

Parameters:
  RETURNS the received data as a byte_array
```

```
  data → std::vector<char>
  data_size -> integer value indicating the amount of data in bytes to
receive

Note: data_size is specified in bytes
```

The following example creates an empty vector for data and receives data:

```
std::vector<char> recv_data;
output.receive_data_with_size(<recv_data>, 1024)
```

- **receive_data_on_tlast():**

```
receive_data_on_tlast(data)
creates a blocking call returning data after receiving tlast packet

Parameters:
  RETURNS the received data as a byte_array
  data → std::vector<char>

Note: The datatype must be specified during object instantiation
```

The following example creates an empty vector for data and receives data on the TLAST signal:

```
std::vector<char> recv_data;
output.receive_data_on_tlast(<recv_data>)
```

Table 40: **Byte_Array Conversion API**

API	Description
byte_array = input_port.uInt8ToByteArray(user_list) byte_array = input_port.uInt16ToByteArray(user_list) byte_array = input_port.uInt32ToByteArray(user_list) byte_array = input_port.uInt64ToByteArray(user_list) byte_array = input_port.int8ToByteArray(user_list) byte_array = input_port.int16ToByteArray(user_list) byte_array = input_port.int32ToByteArray(user_list) byte_array = input_port.int64ToByteArray(user_list) byte_array = input_port.floatToByteArray(user_list) byte_array = input_port.doubleToByteArray(user_list) byte_array = input_port.bfloat16ToByteArray(user_list)	Convert the specified data type to byte_array value to send the data `byte_array = input_port.uInt64ToByteArray(user_list)` `Parameters:` `  Returns list of specified data type converted from byte_array` `  user_list → std::vector<T>` `  // T is the data type present in function signature.` `  // For example in byteArrayToFloat user_list is a vector of type float` `  byte_array → list created using create_byte_array or conversion APIs`

Table 40: **Byte_Array Conversion API** *(cont'd)*

API	Description
user_list = output_port.byteArrayTouInt8(byte_array) user_list = output_port.byteArrayTouInt16(byte_array) user_list = output_port.byteArrayTouInt32(byte_array) user_list = output_port.byteArrayTouInt64(byte_array) user_list = output_port.byteArrayToInt8(byte_array) user_list = output_port.byteArrayToInt16(byte_array) user_list = output_port.byteArrayToInt32(byte_array) user_list = output_port.byteArrayToInt64(byte_array) user_list = output_port.byteArrayToFloat(byte_array) user_list = output_port.byteArrayToDouble(byte_array) user_list = output_port.byteArrayToBfloat16(byte_array)	Convert the `byte_array` value to the specified data type after receiving `user_list = output_port.byteArrayTouInt16(byte_array)` `Parameters:` `  Returns list of specified data type converted from byte_array` `  byte_array → list created using create_byte_array or conversion APIs` `  user_list → std::vector<T>` `  // T is the data type present in function signature.` `  // For example in byteArrayToFloat user_list is a vector of type float`

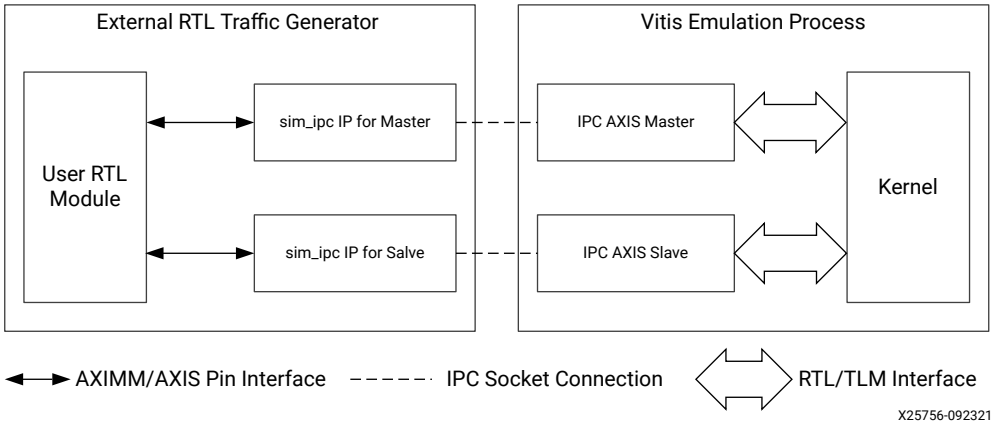
External RTL Traffic Generators using SV/Verilog

Generate traffic using the existing test bench written in the System Verilog/Verilog with slight modification to your test bench hierarchy, as explained below.

External RTL Traffic Generator and Emulation Process

External RTL traffic generators are used to drive traffic to Vitis emulation process or AI Engine simulation process using SystemVerilog or Verilog modules.

Figure 44: **Test Bench Hierarchy**



As shown in the preceding figure, the external test bench (on the left) and the Vitis emulation (on the right), both run as separate simulation processes. To establish communication between two processes using IPC, you must instantiate SIM_IPC Master/Slave modules.

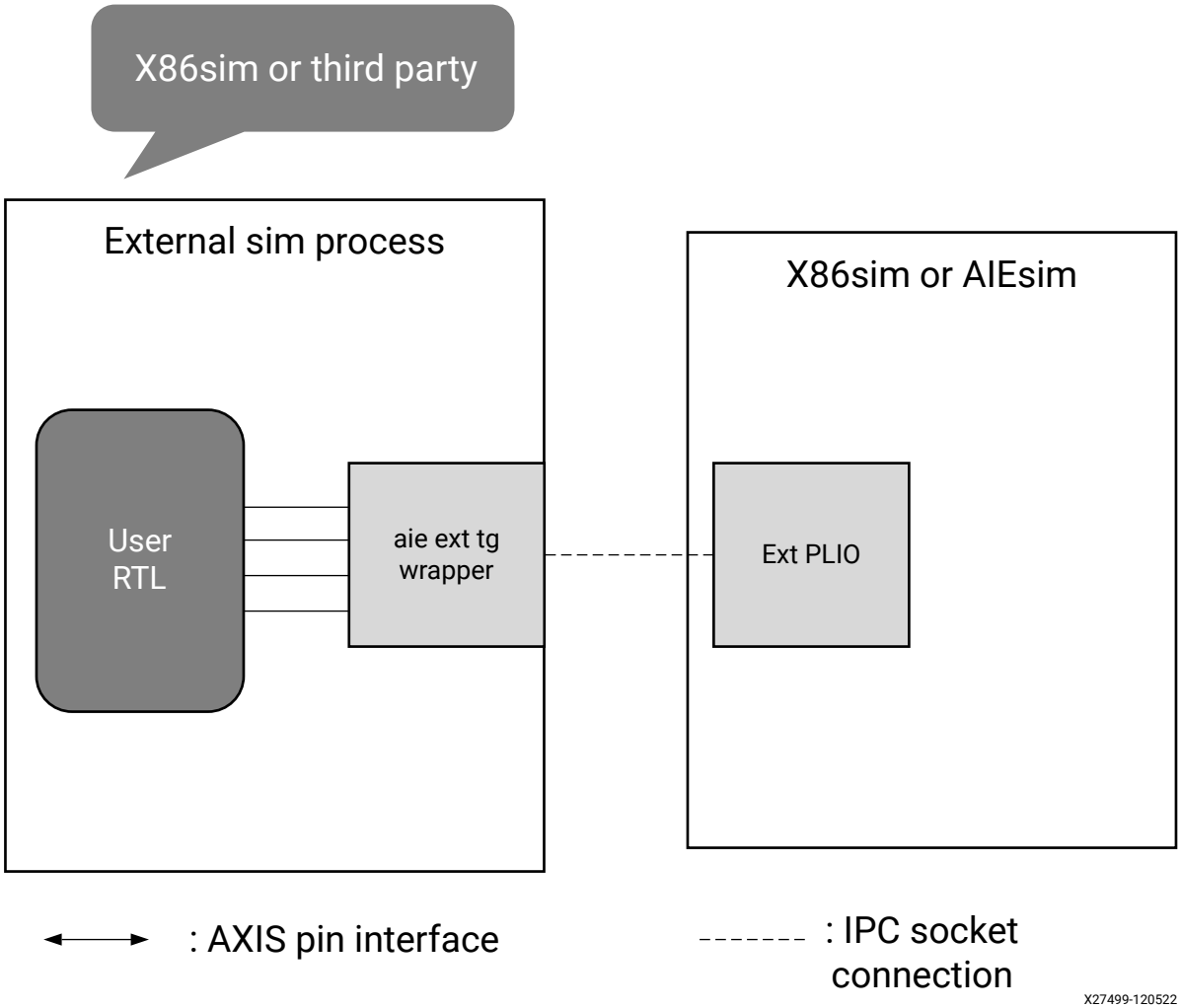
Perform the following modifications:

1. You need to create a project in Vivado simulator. For details on how to create a project, refer *Vivado Design Suite User Guide: Design Flows Overview* (UG892).
2. Once the project is created, you need to instantiate `sim_ipc` IP in the external SV/Verilog test bench.
3. Then run the `export_simulation` command in Vivado to generate the scripts for the simulation.
4. Run the simulation in Vivado simulator. For details running simulation refer to *Vivado Design Suite User Guide: Logic Simulation* (UG900).

External RTL Traffic Generator and AI Engine Simulation

The same technique can be deployed to drive traffic from the external System Verilog/Verilog traffic generators/test benches to the AI Engine simulator or x86-simulator.

Figure 45: **XTLM Test Bench Hierarchy**



To generate the AI Engine wrapper stub module (`aie_wrapper_ext_tb.v`) use the following steps:

1. The wrapper stubs will be generated based on the external PLIO declarations in the ADF graph. You need to perform the ADF graph compilation to generate `scsim_config.json` file that resides in `./Work/config/scsim_config.json` directory. This config file contains information on the PLIOs declared in the graph. For more details on how to perform ADF graph compilation and external PLIOs declaration, refer to *AI Engine Tools and Flows User Guide* (UG1076).
2. You can use this config file as argument to the `gen_aie_wrapper.py` script to auto-generate Verilog stub modules based on ext PLIO declared in ADF Graph:

```
python3 ${XILINX_VITIS}/data/emulation/scripts/gen_aie_wrapper.py \
-json Work/config/scsim_config.json --mode <wrapper/vivado>
```

> **TIP:** *The python script is available in the Vitis installation area as shown in the example above. There are two modes for the script: wrapper and Vivado mode. By default, the script runs in Vivado mode.*

The name of the instance stubs must be identical to the name of the corresponding external PLIOs in the graph and will be reflected in the generated `aie_wrapper_ext_tb.v` file.

After running the `gen_aie_wrapper.py` script, the`aie_wrapper_ext_tb.v` is generated with instances of `sim_ipc_axis` modules that can be directly instantiated in your external test bench.

***Note*:** The module used to send data to/from external traffic generator to AI Engine simulator/x86sim are the XTLM IPC SystemC modules which are present inside the wrapper stub module which includes all the XTLM IPC modules. This wrapper needs to be instantiated in the external test bench to establish the connection as shown in the preceding figure.

Instantiating AI Engine Wrapper in the Test Bench

The aie wrapper module (`aie_wrapper_ext_tb.v`) needs to be instantiated in the external test bench. The `aiesim` expects data to be in beats instead of transaction, so you need to keep `tlast` at high (`1'b1`) all the time.

***Note*:** You can add timescale directive as per your requirement in `aie_wrapper_ext_tb.v`.

***Note*:** Make sure to have the `READY` signal check in your test bench. Due to the fact that backpressure in the slave is modeled in AIE simulation, the AIE will not except the data from the external test bench if it is not ready. You need to check if the `m_axis_tready` (from the slave) is high or not to drive `m_axis_tdata` correctly.

Generating sim_ipc_axis IP for Vivado Project

By default, the python script generates `aie_wrapper_ext_tb_ip.tcl` and `aie_wrapper_ext_tb_proj.tcl` along with the wrapper Verilog file as mentioned in the previous section.

There are two ways to proceed based on the existence of a Vivado project:

1. If you have already created Vivado project, use the IP flow described here. From the **Tcl console** source the `aie_wrapper_ext_tb_ip.tcl` script:

   ```
   source <absolute_path>/aie_wrapper_ext_tb_ip.tcl
   ```

   This Tcl script can be used for generating required `sim_ipc_axis` IP. After sourcing the Tcl file you will see hierarchy created under simulation_sources. You can add the required files and directories for your project.

2. If a Vivado project is not already created, use the project script `aie_wrapper_ext_tb_proj.tcl` to create one. From a terminal use the following command:

   ```
   vivado -mode batch -source aie_wrapper_ext_tb_proj.tcl
   ```

   ***Note*:** To use third party simulators, you need to update the required paths for SIMULATOR_GCC_PATH, SIMULATOR_CLIBS_PATH and INSTALL_BIN_PATH. For more details on how to set the third party simulators, refer to the Logic Simulation chapter of *Vivado Design Suite User Guide: Logic Simulation* (UG900).

   After sourcing `aie_wrapper_ext_tb_proj.tcl`, the tool will generate the `export_sim` directory with sub-directories and scripts required for use with other simulators. This Tcl script sources the `aie_wrapper_ext_tb_ip.tcl` script.

TIP: *The scripts mentioned above only contain the `sim_ipc_axis` modules, so you must add any additional required RTL modules and options to the script. You can modify and directly include required RTL the needed script.*

Running the RTL Traffic Generator with AI Engine Simulation

You can launch the external process using the following steps:

1. If already inside the Vivado project, after the project hierarchy is updated after adding the required sources, you can run the simulation from Vivado. Refer to *Vivado Design Suite User Guide: Logic Simulation* (UG900) for more information.
2. If outside the Vivado project, after the `export_sim` directory is generated with required simulation scripts, you can traverse inside the appropriate simulator directory and run the `<top_module_name>.sh` script to launch the RTL simulation.
3. Also, simultaneously launch the AI Engine simulator process as already mentioned. For details on how to run the `aiesimulator` command, refer to *AI Engine Tools and Flows User Guide* (UG1076).

After simulation is launched you can see the traffic propagating to and from the user RTL.

TIP: *For more details on how to integrate and launch the external RTL traffic generator with AI simulation process refer to the tutorial.*

Running Traffic Generators in Python/C++

After generating an external process binary as shown above using the headers and sources available at `$XILINX_VIVADO/data/emulation/ip_utils/xtlm_ipc/xtlm_ipc_v1_0/<supported_language>`, you can run the emulation using the following steps:

1. For C++, set the `LD_LIBRARY_PATH` as `export LD_LIBRARY_PATH=$XILINX_VIVADO/data/emulation/cpp/lib:$LD_LIBRARY_PATH`
2. For Python, set the `PYTHONPATH` as: `export PYTHONPATH=$XILINX_VIVADO/data/emulation/hw_em/lib/python:$XILINX_VIVADO/data/emulation/python/xtlm_ipc/`
3. From another terminal, launch the external process such as Python/C++/C. If you are running multiple I/O or traffic generator-based solutions on the same machine, set `XTLM_IPC_SOCK_DIR` to be unique to each test case on both the emulation terminal in addition to the external process terminal. For example, `setenv XTLM_IPC_SOCK_DIR <test_case_dir>` (same environment on both emulation process and external process).

   ***Note*:** The traffic generator executable and hw_emu or x86sim/aiesim should be run on the same server/machine.

> **WARNING!** *AMD provides an `end_of_simulation()` API to terminate emulation from master utilities of memory mapped AXI4 and AXI4-Stream interfaces. However, you should not use this method unless there is no way to terminate emulation from the host. In a normal course of emulation, the external process is not expected to terminate emulation. Only use the `end_of_simulation()` in exceptional scenarios.*

Speed and Accuracy of Hardware Emulation

Hardware emulation uses a mix of SystemC and RTL co-simulation to provide a balance between accuracy and speed of simulation. The SystemC models are comprised of purely functional models and performance approximate models. Hardware emulation does not mimic hardware accuracy 100%, therefore you should expect some differences in behavior between running emulation and executing your application on hardware. This can lead to significant differences in application performance, and sometimes differences in functionality can also be observed.

Functional differences with hardware typically point to a race condition or some unpredictable behavior in your design. So, an issue seen in hardware might not always be reproducible in hardware emulation, though most behavior related to interactions between the host and the kernel, or the kernel and the memory are reproducible in hardware emulation. This makes hardware emulation an excellent tool to debug issues with your kernel prior to running on hardware.

The following table lists models that are used to mimic the hardware platform and their accuracy levels.

Table 41: **Hardware Platform**

Hardware Functionality	Description
AMD UltraScale™ DDR Memory, SmartConnect	The SystemC models for the DDR memory controller, AXI SmartConnect, and other data path IPs are usually throughput approximate. They typically do not model the exact latency of the hardware IP. The model can be used to gauge a relative performance trend as you modify your application or the kernel.
AI Engine	The AI Engine SystemC model is cycle approximate, though it is not intended to be 100% cycle accurate. A common model is used between AI Engine Simulator and hardware emulation, thus enabling a reasonable comparison between the two stages.
AMD Versal™ NoC and DDR Models	The Versal NoC and DDR SystemC models are cycle approximate.
Arm Processing Subsystem (PS, CIPS)	The Arm PS is modeled using QEMU, which is a purely functional execution model. For more information, see QEMU.
User Kernel	Hardware emulation uses RTL for the user kernel. As follows, the kernel behavior by itself is 100% accurate. However, the kernel is surrounded by other approximate models.
Other I/O Models	For hardware emulation, there is generic Python or C-based traffic generator which can be interfaced with the emulation process. You can generate abstract traffic at AXI protocol level which mimics the I/O in your design. Because these models are abstract, any issues observed on the specific hardware board will not be shown in hardware emulation.

Because hardware emulation uses RTL co-simulation as its execution model, the speed of execution is orders of magnitude slower as compared to real hardware. AMD recommends using small data buffers. For example, if you have a configurable vector addition and in hardware you are performing a 1024 element `vadd`, in emulation you might restrict it to 16 elements. This will enable you to test your application with the kernel, while still completing execution in reasonable time.

Chapter 10

Profiling and Tracing the Application

Running the system, either in emulation or on the system hardware, presents a series of potential challenges and opportunities. When running the system for the first time, you can profile the application to identify bottlenecks, or performance issues that offer opportunities to optimize the design, as discussed in the sections below.

A system-level view of program execution can be helpful in identifying problems during program execution, including correctness and performance issues. Tracing the application is useful for detecting events not synchronized in between PS, PL and AI Engine. On the AI Engine side, issues such as missing or mismatching locks, buffer overruns, and incorrect programming of DMA buffers are examples that are difficult to debug by using explicit print statements or by using traditional interactive debuggers. A systematic way of collecting system level traces for the program execution is needed. The AI Engine architecture has direct support for generation, collection, and streaming of events as trace data during simulation, hardware emulation, or hardware execution.

Profiling the Application

The AMD Vitis™ core development kit generates various system and kernel resource performance reports during compilation. These reports help you establish a baseline of performance for your application, identify bottlenecks, and help to identify target functions that can be accelerated in hardware kernels as discussed in Methodology for Architecting a Device Accelerated Application in the *Data Center Acceleration using Vitis* (UG1700). The Xilinx Runtime (XRT) collects profiling data during application execution in both emulation and hardware builds. Examples of profiling and event data that can be reported includes:

- Host and device timeline events
- XRT native API call sequences
- Kernel execution sequence
- Kernel start and stop signals
- FPGA trace data including AXI transactions

- Power profile data
- AI Engine profiling and event trace
- User event and range profiling

Profiling reports and data can be used to isolate performance bottlenecks in the application, identify problems in the system, and optimize the design to improve performance. Optimizing an application requires optimizing both the application host code and any hardware or AI Engine kernels. The host code must be optimized to facilitate data transfers and kernel execution, while the PL kernels should be optimized for performance and resource usage and AI Engine kernels can be optimized for throughput and latency.

There are four distinct areas to be considered when performing algorithm optimization in the Vitis environment: System resource usage and performance, kernel optimization (resource, performance, throughput, latency), host optimization, and data transfer optimization. The following Vitis reports and graphical tools support your efforts to profile and optimize these areas:

- Guidance
- System Estimate Report in the *Data Center Acceleration using Vitis* (UG1700)
- HLS Synthesis Report
- Profile Summary Report in the *Data Center Acceleration using Vitis* (UG1700)
- Timeline Trace in the *Data Center Acceleration using Vitis* (UG1700)
- Waveform View and Live Waveform Viewer in the *Data Center Acceleration using Vitis* (UG1700)

When enabled as described in Enabling Profiling in Your Application, these reports are automatically generated while running the active build, either from the command line as described in Chapter 8: Building and Running the System, or when Using the Vitis Unified IDE in the *Vitis Reference Guide* (UG1702). Separate reports are generated for the different build targets and can be found in the respective report directories. Reports can be viewed in the Analysis view in the IDE as described in Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702).

Enabling Profiling in Your Application

To enable event trace data during the execution of your application, you must instrument your application for this task. You must enable additional logic, consume additional device resources to track the host and kernel execution steps, and capture event data. This process requires optionally modifying your host application to capture custom data, modifying your kernel XO during compilation and the `xclbin` during linking to capture different types of profile data from the device side activity. You also need to configure the Xilinx Runtime (XRT) as described in the xrt.ini File to capture data during the application runtime.

> **TIP:** *Even though capturing profile data is a critical part of the profiling and optimization process for building your application, it consumes additional resources and impacts performance. Make sure to clean these elements out of your final production build.*

There are many different types of profiling for your applications, depending on which elements your system includes and what type of data you want to capture. The following table shows some of the levels of profiling that can be enabled, and discusses which are complimentary and which are not.

Table 42: **Profiling Host and Kernels**

Profile/Trace	Description	Comments
Host Application XRT Native API	Specified by the use of the `native_xrt_trace` option in the `xrt.ini` file.	Generates profile summary and trace events for the XRT API as described in Writing the Software Application in the *Data Center Acceleration using Vitis* (UG1700).
Host Application User-Event Profiling	Requires additional code in the host application as described in Custom Profiling of the Host Application.	Generates user range data and user events for the host application. **TIP:** *Can be used to capture event data for user-managed kernels as described in Working with User-Managed Kernels in the Data Center Acceleration using Vitis (UG1700).*
Device Side Profiling	Enabled by the use of `--profile` options during `v++` compilation and linking, as described in --profile Options, and the use of `device_trace` in the `xrt.ini` file.	Enables capturing data traffic between the host and kernel, kernel stalls, the execution times of kernels and compute units (CUs), in addition to monitoring activity in AMD Versal™ AI Engines.
AI Engine Graph and Kernels	Specified by the use of the `aie_profile` option in the `xrt.ini` file. These options can be specified together or separately.	Generates the `default.aierun_summary` report containing the Profile. The `aierun_summary` can be found in the `aiesimulator_output` folder of the AI Engine graph build directory. Refer to the AI Engine Simulation-Based Profiling chapter in the *AI Engine Tools and Flows User Guide* (UG1076) for more information.
Power Profile	Specified by the use of the `power_profile` option in the `xrt.ini` file.	Generates the `power_profile_<device>.csv` report. ***Note***: This feature is not supported on embedded platforms or AWS.
Vitis AI Profiling	Specified by the use of the `vitis_ai_profile` option in the `xrt.ini` file.	Enables counter profiling of DPUs to generate the `xrt.run_summary` for viewing in Vitis analyzer.

The device binary (`xclbin`) file is configured for capturing limited device-side profiling data by default. However, using the `--profile` option during the Vitis compiler linking process instruments the device binary by adding AXI Performance Monitors, and Memory Monitors to the system. This option has multiple instrumentation options: `--profile.data`, `--profile.stall`, and `--profile.exec`, as described in the --profile Options.

As an example, add `--profile.data` to the `v++` linking command line:

```
v++ -g -l --profile.data all:all:all ...
```

TIP: *Be sure to also use the `v++ -g` option when compiling your kernel code for debugging with software or hardware emulation.*

After your application is enabled for profiling during the `v++` compile and link process, data gathering during application runtime must also be enabled in XRT by editing the `xrt.ini` file as discussed above. For example, the following `xrt.ini` file enables power profiling, event and stall trace capture when the application is run:

```
[Debug]
power_profile=true
device_trace=fine
stall_trace=all
```

To enable the profiling of Kernel Internals data, you must also add the `debug_mode` tag in the `[Emulation]` section of the `xrt.ini`:

```
[Emulation]
debug_mode=batch
```

If you are collecting a large amount of trace data, you can increase the amount of available memory for capturing data by specifying the `--profile.trace_memory` option during `v++` linking, and add the `trace_buffer_size` keyword in the `xrt.ini`.

- **`--profile.trace_memory`:** Indicates what type of memory to use for capturing trace data.
- **`trace_buffer_size`:** Specifies the amount of memory to use for capturing the trace data during the application runtime.

TIP: *When `--profile.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 size.*

Custom Profiling of the Host Application

All XRT related actions from the host application are automatically tracked for profiling, through the XRT API calls. However, you can also profile the host application beyond the XRT related events, capturing event data based on user-specified actions or events.

This feature provides two types of custom profiling:

- **User range:** Profiles the specified start/end times across a range of code. This captures the span of time within which an action occurs in the host application.
- **User events:** Marks an event in the timeline. The user event is added to the timeline waveform at whatever point in time it occurs.

The `user_range` and `user_event` data can be captured to the Profile Summary and Timeline Trace reports for display in Vitis analyzer. As seen in the following figure, the Profile Summary shows the number of occurrences of a given event and the range. The User Ranges table also reports the Min/Max/Avg/Total duration of the user-defined ranges in the host code. In the Timeline Trace report `user_range` elements in the host code are displayed in a separate row, and `user_event` markers are added at specific points on the timeline.

Figure 46: **Profile Summary – User Range**

Using custom profiling requires a few changes in your host application source code and build process. You must make use of C or C++ API in your code, as described below, and you must include the `xrt_coreutil` library when linking your host application.

- The C/C++ API are described below, but can also be found at the following URL: https://github.com/Xilinx/XRT/blob/master/src/runtime_src/core/include/experimental/xrt_profile.h.
- For both C and C++ you must add the following:

```
#include experimental/xrt_profile.h
```

- When linking host code, add `-lxrt_coreutil` to the compiler command line.

Profiling of C++ Code

For C++ code the provided objects are:

- **`user_range`:** This object captures the start time and end time of a measured range of activity with the specified ID. The object constructor is:

```
user_range(const char* label, const char* tooltip);
```

- **`user_event`:** This object marks an event occurring at single point in time, adding the specified label onto the timeline trace. The object constructor is:

```
user_event()
```

Use the `user_range` to construct an object and start keeping track of time immediately upon construction. Usage details of the `user_range` objects:

- If a `user_range` is instantiated using the default constructor, no time is marked until the user calls `user_range.start()` with the label and tooltip.
- You can instantiate a `user_range` object passing the label and tooltip strings. This starts monitoring the range immediately.
- You must call `user_range.start()` and `user_range.end()` to capture ranges of time you are interested in.
- If `user_range.end()` is not called, then any range being tracked lasts until the `user_range` object is destructed.
- The `user_range` object can be reused any number of times, by calling `user_range.start()`/`user_range.end()` pairs in the host code.
- Sequential calls to `user_range.start()` ignore all but the first call until `user_range.end()` terminates the range.
- Sequential calls to `user_range.end()` ignore all but the first call until `user_range.start()` starts a new range.

Usage of the `user_event` objects:

- A `user_event` object must be instantiated using the default constructor.
- Calls to `user_event.mark()` creates a user marker on the timeline trace at that particular time.
- `user_event.mark()` takes an optional `const char*` argument which appears as a label on the timeline trace.

With your host application properly instrumented, XRT can capture profile data from these user-defined ranges and events, in addition to the standard XRT API-based events. You must enable profiling in the`xrt.ini` file as explained previously.

Profiling of C Code

For C code the provided functions are:

- **`xrtURStart()`:** This function establishes the start time of a measured range of activity with the specified ID. The function signature is:

```
void xrtURStart(unsigned int id, const char* label, const char* tooltip)
```

- **`xrtUREnd()`:** This function marks the end time of a measured range with the specified ID. The function signature is:

```
void xrtUREnd(unsigned int id)
```

- **`xrtUEMark()`:** This function marks an event occurring at single point in time, adding the specified label onto the timeline trace. The function signature is:

```
void xrtUEMark(const char* label)
```

Use the `xrtURStart()` and `xrtUREnd()` functions to start keeping track of time immediately, and specify an ID to pair the start/end calls and define the user range. Usage details of the `user_range` functions:

- Start/End ranges of one ID can be nested inside other Start/End ranges of a different ID.
- It is your responsibility to make sure the IDs match for the Start/End range you are profiling.

IMPORTANT! *Multiple calls to `xrtURStart` and `xrtUREnd` with the same ID can cause unexpected behavior.*

- The user range can have a label that is added to the timeline, and a tooltip that is displayed when you place the cursor over the user range.

A call to `xrtUEMark()` will create a user marker on the timeline trace at the point of the event.

- `xrtUEMark()` lets you specify a label for the event. The label will appear on the timeline with the mark.
- You can use `NULL` for the label to add an unlabeled mark.

The following is example code:

```
int main(int argc, char* argv[]) {
 58
 59   xrtURStart(0, "Software execution", "Whole program execution") ;
 60 ...
 61   //TARGET_DEVICE macro needs to be passed from gcc command line
 62   if(argc != 2) {
 63       std::cout << "Usage: " << argv[0] <<" <xclbin>" << std::endl;
 64       return EXIT_FAILURE;
 65    }
....
153     q.enqueueTask(krnl_vector_add);
154
155     // The result of the previous kernel execution will need to be
retrieved in
156     // order to view the results. This call will transfer the data from
FPGA to
157     // source_results vector
```

```
158
q.enqueueMigrateMemObjects({buffer_result},CL_MIGRATE_MEM_OBJECT_HOST);
159 ....
160     q.finish();
161
162     xrtUEMark("Starting verification") ;
163
```

Enabling NoC-DDRMC Profiling

The Versal device programmable Network on Chip (NoC) is an AXI-interconnecting network used for sharing data between IP endpoints in the programmable logic (PL), the processing system (PS), and other integrated blocks. This device-wide infrastructure is a high-speed, integrated datapath with dedicated switching. The NoC system is a large-scale interconnection of instances of NoC master units (NMUs), NoC slave units (NSUs), and NoC packet switches (NPSs), each controlled and programmed from a NoC programming interface (NPI). There are 16 NMUs/NSUs on the VC1902, each one is capable of 16 Gb/s of throughput in each direction.

Network performance of the NoC interconnecting network can be monitored by the `Vperf` utility. `Vperf` is a Vitis tool that uses the ChipScoPy functionality to profile the NoC and DDRMC in applications built using a `v++` flow. ChipScoPy is an open-source Python project that enables communication with and control of Versal device debug solutions. The ChipScoPy Python package allows users to program designs and begin debugging in a few simple steps. Refer to *Profiling the NoC* in *AI Engine Tools and Flows User Guide* (UG1076) for more information on this process.

AI Engine Profiling

AI Engine profiling is a form of dynamic program analysis that measures the input/output throughput, the space (memory), time complexity of a program, the usage of specific instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization, and more specifically, performance tuning.

You can obtain profiling data when you run your design in simulation or in hardware at runtime. Analyzing this data helps you gauge the efficiency of the kernels, the stall and active times associated with each AI Engine, and pinpoint the AI Engine kernel whose performance might not be optimal. This also allows you to collect data on design latency, throughput, and bandwidth by using runtime event APIs in your PS host code or using specific parameter lines in the `xrt.ini` file.

There are two ways to gather this information:

1. Use performance counters built into the hardware to monitor AI Engine and memory module events. This feature can be used in hardware and hardware emulation.
2. Use event APIs to profile the AI Engine graph inputs and outputs from the graph host application. This feature can be used in simulation and hardware flows.

Enabling AI Engine Profiling

In the AI Engine, application profiling can be done at different development stage.

- Simulation using AI Engine simulator (`aiesimulator`)
- Hardware Emulation and Hardware run

AI Engine Simulation-Based Profiling

Profiling Data Generation

In the simulation framework, the AI Engine simulator can generate a profiling report for the complete application. This report is generated using the flag `--profile`.

```
aiesimulator -pkg-dir=Work --profile
```

Text files and XML files are generated in the directory `aiesimulator_output`. Two types of files are generated for the tile located in column C and row R. The `*_funct` reports the number of calls and number of cycles for each function. The `*_instr` is a report that goes down to the assembly code. To visualize the report, use the Analysis View of the Vitis Unified IDE.

```
vitis -a aiesimulator_output/default.aierun_summary
```

The Profile tab opens the Profile report, which shows a menu of sections that show information.

- **Summary:** Reports the total cycle count, total instruction count, and program size in memory.
- **Function Reports:** Shows several key indicators of the functions in the graphs.
  - **Number of calls:** Reports the number of times the function is executed
  - **Total function time (cycles and %):** Reports the function execution time (in cycles and as a percent). This is the time required to execute the code within a function, exclusive of any calls to its descendants.
  - **Total function + descendant time (cycles and %):** Reports the function execution time, as well as the execution time of the descendant functions (descendant functions are functions called by the function whose profile information is being reported). The "Total Function+descendant time" represents the total time required to execute the code within a function and in any function it calls, including the time spent in its descendant functions.

    ***Note*:** The time includes the time spent in the function itself as well as the time spent in all the functions it calls, directly or indirectly.
  - **Min/Avg/Max function time (cycles):** Reports the minimum/average/maximum function execution time (in cycles and as a percent).

  - **Min/Avg/Max function + descendant time (cycles):** Reports the minimum/average/maximum function execution time, as well as the execution time of the descendant functions (descendant functions are functions called by the function whose profile information is being reported).
  - **Program counter Low/High:** Reports the lowest and highest program counter value for a specific function.
- **Profile Details:** Shows the assembly code, function by function, with useful precisions.

For more details on Profile Details and debugging performance issues, refer to the chapter for AI Engine Simulation Based Profiling in *AI Engine Tools and Flows User Guide* (UG1076).

Using Printf for Basic Debug

The simplest form of tracing is to use a formatted `printf()` statement in the code for printing debug messages. Visual inspection of intermediate values, addresses, etc. can help you understand the progress of program execution. No additional include files are necessary for using `printf()` other than standard C/C++ includes (`stdio.h`). You can add `printf()` statements to your code to be processed during simulation, or hardware emulation, and remove them or comment them out for hardware builds.

Adding `printf` statements to your AI Engine kernel code will increase the compiled size of the AI Engine program. Be careful that the compiled size of your kernel code does not exceed the per-AI Engine processor memory limit of 16 KB.

IMPORTANT! *You must use the `aiesimulator --profile` command to enable the `printf()` execution during a simulator run. If `--profile` is not specified, the `printf()` function is ignored.*

A separate driver and binary is used for this functionality to allow the main simulator to remain as fast as possible. Using the debug simulator driver produces a per-tile profile report under the output directory which gives detailed cycle-level statistics of kernel execution. In addition, using the `--profile` option generates a run_summary file that is written to the `./aiesimulator_output` folder that can be viewed as described in the *AI Engine Tools and Flows User Guide* (UG1076).

Hardware Emulation and Hardware Run Profiling

AI Engine profiling uses performance counters at all level of the device:

- Runtime event performance counters for the AI Engine modules
- Runtime memory counters for memory modules and memory tiles
- Runtime interface counters for AI Engine-PL interface tiles.

These performance counters can be configured to track a variety of events in the AI Engine, the memory module and the interface tile. Various features such as error-correction code (ECC) scrubbing, event trace and profiling can use these performance counters. Performance counters count occurrences of a given event in a profile configuration. The profile feature offers several different configurations of these performance counters that can be dynamically applied at runtime to collect various profiling statistics.

No changes are required in PS host code when using performance counters. These counters can be configured, read and collected at runtime while the design is executing in hardware. The following table lists the number of performance counters that are available at different configurations.

Various metrics exist for all different part of the array:

Table 43: **AI Engine Metrics**

Metric Name	Description
heat_map	Reports time where the AI Engine was active, stalled, executing vector instruction.
stalls	Reports time the AI Engine is not active due to memory access, stream access, cascade access or lock acquisition.
execution	Reports the time spent by the AI Engine on vector instructions, load/store Instructions and cumulative instruction time
floating_point	Reports time spent on floating-point exceptions
aie_trace	Reports the amount of data for trace, back-pressure, memory module and memory module back-pressure produced by the AI Engine.
write_throughputs	Reports the time spent by the AI Engine on executing write operations on streams, cascade interface. There is also the write throughput on these interface
read_throughput	Reports the time spent by the AI Engine on executing read operations on streams, cascade interface. There is also the write throughput on these interface
stream_put_get	Reports time spent on executing cascade and stream operations

Table 44: **Memory Module Metrics**

Metric Name	Description
conflicts	Reports time spent on memory conflicts and ECC errors
dma_locks	Reports time spent on stalled locks on both channels
dma_stalls_s2mm	Reports the time spent by each S2MM channel on stalls due to lock acquisition
dma_stalls_mm2s	Reports the time spent by each MM2S channel on stalls due to lock acquisition
s2mm_throughputs	Reports the number of BD packets and the throughput of each S2MM channel. In AI Engine-ML the back-pressure time is also available.

Table 44: **Memory Module Metrics** *(cont'd)*

Metric Name	Description
mm2s_throughputs	Reports the number of BD packets and the throughput of each MM2S channel. In AI Engine-ML the back-pressure time is also available.

Table 45: **Memory Tile Metrics**

Metric Name	Description
s2mm_channels	Reports Transfert/Stalled time, Number of AXI4-Stream packets and BD packets transferred over memory tile input channel
s2mm_channels_details	Reports Transfer, Backpressure, lock stall and stream starvation time on input streams
mm2s_channels	Reports Transfert/Stalled time, Number of AXI4-Stream packets and BD packets transferred over memory tile output channel.
mm2s_channels_details	Reports Transfer, Backpressure, lock stall and stream starvation time on output streams
memory_stats	Reports Group Errors on Memory
s2mm_throughputs	Reports Transfer, Starvation, Backpressure, lock stall time along with S2MM Channel Throughput.
mm2s_throughputs	Reports Transfer, Starvation, Backpressure, lock stall time along with MM2S Channel Throughput.
conflict_stats*N*	Reports the number of 4 consecutive memory bank conflicts, starting at bank 4*N*. N=0,1,2,3

Table 46: **Interface Tile Metrics**

Metric Name	Description
input_throughputs	Reports Transfer, Stalled, Idle time as well as throughput
output_throughputs	Reports Transfer, Stalled, Idle time as well as throughput
input_stalls	Reports Stall and Idle time for channel 0. For AI Engine-ML it will be Backpressure and Starvation time for channels 0 and 1
output_stalls	Reports Stall and Idle time for channel 0. For AI Engine-ML it will be Backpressure and Lock Stall time for channels 0 and 1
packets	Reports number of packets (input/output)
start_to_bytes_transferred	Total clock cycles to transfer byte count for specified graph/port
interface_tile_latency	Total latency in clock cycles between graph1:port1 and graph2:port2

For more details on these metrics, see the chapters on Profiling the AI Engine, Memory Module and Interface Tile in *AI Engine Tools and Flows User Guide* (UG1076).

Launch AI Engine Profiling

There are two ways to launch AI Engine profiling in Hardware:

- XRT flow
- XSDB flow

XRT Flow

In order to use the XRT flow, create the `xrt.ini` file at the same location where the PS host application is located. Specify a line making AI Engine profiling possible, followed by multiple lines specifying the exact settings of the metrics to be used.

An example of `xrt.ini` file is as follows:

```
[Debug]
#
# Profile Counters
#
aie_profile = true

[AIE_profile_settings]
# Sample interval (in usec)
interval_us = 100
#   All tiles
tile_based_aie_metrics = all:heat_map
tile_based_aie_memory_metrics = all:conflicts
tile_based_interface_tile_metrics = all:s2mm_throughputs:0
```

where:

- **`[Debug]`:** Specifies debug section for XRT, this is case sensitive.
- **`aie_profile`:** Enables profile configuration.
- **`[aie_profile_settings]`:** Specifies profile settings for XRT.
- **`aie_profile_interval_us`:** Profiles data collection interval in micro seconds.
- **`tile_based_aie_metrics`:** Configures metric to be applied to the AI Engine on a tile basis.
- **`tile_based_aie_memory_metrics`:** Configures memory metric to be applied on a tile basis.
- **`tile_based_interface_tile_metrics`:** Configures interface metric to be applied on a tile basis.

There exist many ways to define the tiles you want to select for profiling based on tiles or on graph.

For more details, see the chapters on Profiling the AI Engine in Hardware, Profiling Flow and XRT Flow in the *AI Engine Tools and Flows User Guide* (UG1076).

XSDB Flow

When running the application, the profile data is captured in counters that can be retrieved by the debugging and profiling IP. To capture and evaluate this data, you must connect to the hardware device using `xsdb`. This command is typically used to program the device and debug applications. Connect your system to the hardware platform or device over JTAG, launch the `xsdb` command in a command shell, and run the following sequence of commands:

```
xsdb% connect
xsdb% ta 1
xsdb% source $::env(XILINX_VITIS)/scripts/vitis/util/aie_profile.tcl
xsdb% aieprofile start -graphs myGraph -work-dir ./Work \
      -graph-based-aie-metrics "dut:kernel1:heat_map" \
      -tile-based-aie-metrics "all:stalls" \
      -graph-based-aie-memory-metrics "dut:all:write_throughputs" \
      -tile-based-aie-memory-metrics "{4,1}:{6,2}:conflicts;
{8,3}:dma_locks" \
      -tile-based-interface-tile-metrics "2:10:input_throughputs:3" \
      -interval 20  -samples 100
```

where:

- **`connect`:** Launches the hw_server and connects `xsdb` to the device.
- **`source $::env(XILINX_VITIS)/scripts/vitis/util/aie_profile.tcl`:** Sources the Tcl trace command to set up the `xsdb` environment.

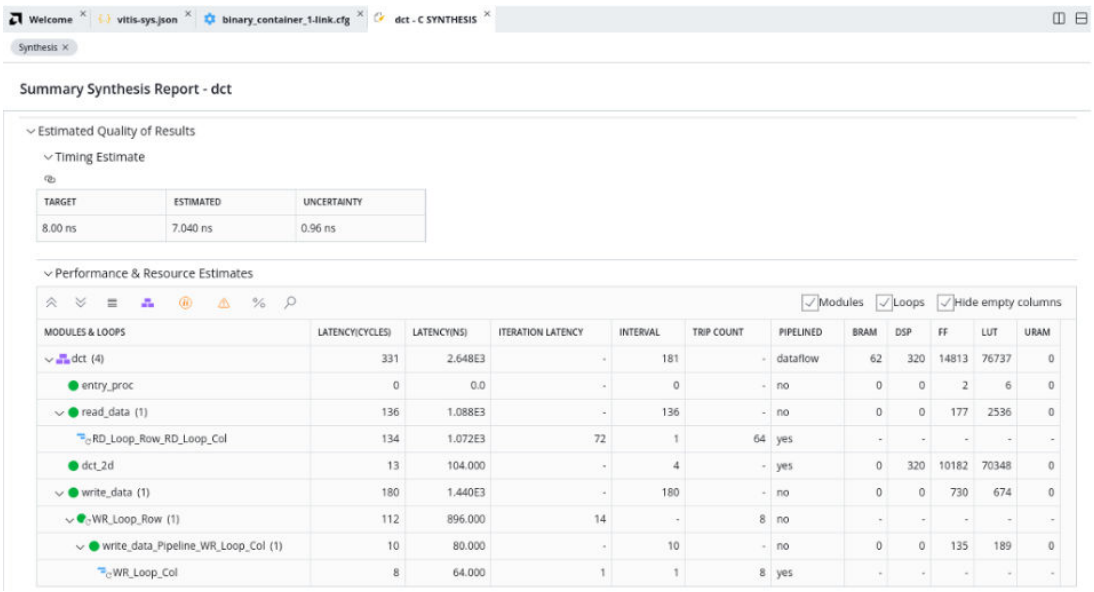
For more details, see the chapters on Profiling the AI Engine in Hardware, Profiling Flow and XRT Flow in the *AI Engine Tools and Flows User Guide* (UG1076).

HLS Synthesis Report

The HLS compiler generates a number of reports for simulation, synthesis, co-simulation. These reports provide details about the high-level synthesis (HLS) compilation of a PL kernel. The main report is the Synthesis Summary report that provides estimated FPGA resource usage, operating frequency, latency, and interface signals of the custom-generated hardware logic. These details provide many insights to guide kernel optimization.

When running from the Vitis Unified IDE, this report can be found in the HLS component directory named `<hls_component>.hlscompile_summary`. The Summary report can be opened from the Flow Navigator in the HLS component under the C Synthesis/Reports heading, or by opening the Compile Summary, or the Link Summary as described in Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702).

Figure 47: **Synthesis Summary Report**



Generating and Opening the HLS Report

IMPORTANT! *You must specify the* `--save-temps` *option during the build process to preserve the intermediate files produced by Vitis HLS, including the reports. The HLS report and HLS guidance are only generated for hardware emulation and system builds for C kernels.*

The HLS report can be viewed through the Vitis analyzer by opening the `<output_filename>.compile_summary` or the `<output_filename>.link_summary` for the application project. The `<output_filename>` is the output of the `v++` command.

You can launch the Vitis analyzer and open the report using the following command:

```
vitis_analyzer <output_filename>.compile_summary
```

When the Vitis analyzer opens, it displays the Compile Summary and a collection of reports generated during the compile process. Refer to Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702) for more information.

Interpreting the HLS Report

The HLS Synthesis report lists the module hierarchy in the left column. This section describes one section of the HLS Synthesis report: Performance and Resource Estimates. Each module and loop generated by the HLS run is represented in this hierarchy. The HLS Synthesis report contains the following columns:

- Issue Type
- Slack

- Latency in clock cycles
- Latency in absolute time (ns)
- Iteration Latency
- Interval
- Trip Count
- Pipelined
- Utilization Estimates of BRAM, DSP, FF, and LUT

If the information is part of a hierarchical block, the information of the blocks contained in the hierarchy are summed up. Therefore, the hierarchy can also be navigated from within the report when it is clear which instance contributes to the overall design.

CAUTION! *The absolute counts of cycles and latency numbers are based on estimates identified during HLS synthesis, especially with advanced transformations, such as pipelining and dataflow. Therefore, these numbers might not accurately reflect the final results. If you encounter question marks in the report, this might be due to variable bound loops, and you are encouraged to set trip counts for such loops to have some relative estimates presented in this report.*

Guidance

The Vitis core development kit has a comprehensive design guidance tool that provides immediate, actionable guidance to the software developer for issues detected in their designs. These issues might be related to the source code, or due to missed tool optimizations. Also, the rules are generic rules based on an extensive set of reference designs. Therefore, these rules might not be applicable for your specific design. It is up to you to understand the specific guidance rules and take appropriate action based on your specific algorithm and requirements.

Guidance is generated from the Vitis HLS, Vitis profiler, and AMD Vivado™ Design Suite when invoked by the `v++` compiler. The generated design guidance can have several severity levels: errors, warning messages, and informational messages are provided during hardware emulation, and system builds. The profile design guidance helps you interpret the profiling results which allows you to focus on improving performance.

Guidance includes message text for reported violations, a brief suggested resolution, and a detailed resolution provided as a web link. You can determine your next course of action based on the suggested resolution. This helps improves productivity by quickly highlighting issues and directing you to additional information in using the Vitis technology.

Design guidance is automatically generated after building or running an application from the command line or Vitis Unified IDE.

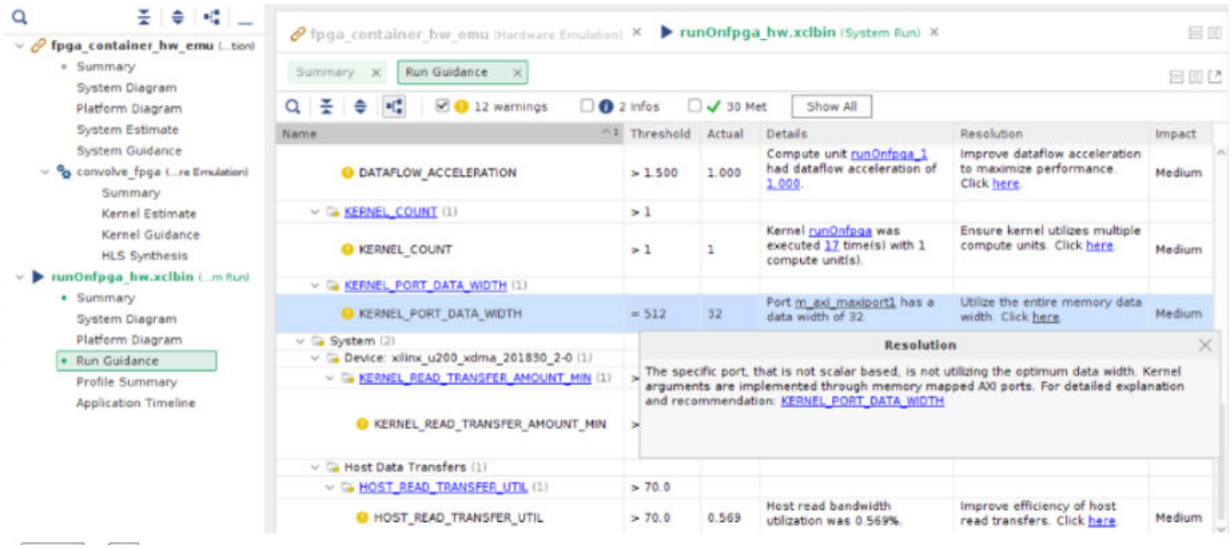
You can open the Guidance report as discussed in Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702). To access the Guidance report, open the Compile Summary, the Link Summary, or the Run Summary, and open the Guidance report.

- Kernel Guidance is generated by the Vitis HLS tool after kernel is built using `v++` compile command. This can be viewed in the Vitis analyzer by opening the Compile Summary report. Kernel guidance and Compile Summary files are generated for each kernel compiled. Kernel guidance includes recommendations on using Dataflow, as well as possible reasons why the expected throughout could not be achieved.
- System Guidance is generated after the `.xclbin` or `.xsa` is built using the `v++` link command. This can be viewed in the Vitis analyzer by opening the Link Summary report. System guidance includes all Kernel Guidance checks, and provides comprehensive review before running your application.
- Run Guidance is generated when your generated `.xclbin` is run, and is a feature of the XRT. This can be viewed by opening the Run Summary in the Vitis analyzer. Run Guidance includes checks like if Kernel Stall is above 50%, recommendations if PLRAM can be used instead of DDR memory, etc.

With the Guidance report open, the Guidance view displays the messages along with resolution columns. The resolutions also have extended weblink help available.

The following image shows an example of the Guidance report displayed in the Vitis analyzer. For example, clicking a link in the Name column opens a description of the rule check. Links in the Details column can open source code, select a design object such as a kernel, or navigate to another report.

Figure 48: **Design Guidance Example**



There is one HTML guidance report for each run of the `v++` command, including compile and link. The report files are located in the `--report_dir` under the specific output name. For example:

- `v++_compile_<output>_guidance.html` for `v++` compilation
- `v++_link_<output>_guidance.html` for `v++` linking

You can click the web link in the Resolution column to get additional details about the resolution. The Vitis HLS Guidance Messaging web page lists all of the current messages for your review.

Kernel objects, in addition to profile reported data values, can also be cross-probed to other views like the Profile Report.

Opening the Guidance Report

When kernels are compiled and when the FPGA binary is linked, guidance reports are generated automatically by the `v++` command. You can view these reports in the Vitis analyzer by opening the `<output_filename>.compile_summary` or the `<output_filename>.link_summary` for the application project. The `<output_filename>` is the output of the `v++` command.

As an example, launch the Vitis analyzer and open the report using this command:

```
vitis_analyzer <output_filename>.link_summary
```

When the Vitis analyzer opens, it displays the link summary report, as well as the compile summaries, and a collection of reports generated during the compile and link processes. Both the compile and link steps generate Guidance reports to view by clicking the **Build** heading on the left-hand side. Refer to Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702) for more information.

Interpreting Guidance Data

The Guidance view places each entry in a separate row. Each row might contain the name of the guidance rule, threshold value, actual value, and a brief but specific description of the rule. The last field provides a link to reference material intended to assist in understanding and resolving any of the rule violations.

In the GUI Guidance view, guidance rules are grouped by categories and unique IDs in the Name column and annotated with symbols representing the severity. These are listed individually in the HTML report. In addition, as the HTML report does not show tooltips, a full Name column is included in the HTML report as well.

The following list describes all fields and their purpose as included in the HTML guidance reports.

- **Id:** Each guidance rule is assigned a unique ID. Use this id to uniquely identify a specific message from the guidance report.
- **Full Name:** The Full Name provides a less cryptic name compared to the mnemonic name in the Name column.
- **Categories:** Most messages are grouped within different categories. This allows the GUI to display groups of messages within logical categories under common tree nodes in the Guidance view.

- **Threshold:** The Threshold column displays an expected threshold value, which determines whether or not a rule is met. The threshold values are determined from many applications that follow good design and coding practices.
- **Actual:** The Actual column displays the values actually encountered on the specific design. This value is compared against the expected value to see if the rule is met.
- **Details:** The Details column describes the specifics of the current rule.
- **Resolution:** The Resolution column provides a pointer to common ways the model source code or tool transformations can be modified to meet the current rule. Clicking the link brings up a popup window or the documentation with tips and code snippets that you can apply to the specific issue.

Tracing The Application

Tracing the application involves a different process whether you have a Linux OS running on the board or if this is a bare-metal application. Linux OS opens the possibility of using XRT while a baremetal application forces you to use `xsdb` to issue trace commands.

During tracing the system with Linux OS, special events such as XRT API commands in the host code (for example start a kernel, data transfer, kernel start, stall or lock stall in the AI Engine array) are recorded along with their timestamps to display the sequence of events. Through this, you can understand the origin of a system stall or data coherency.

Tracing the system can be done at multiple levels:

- Through the application that uses XRT API to start and monitor PL kernels and AI Engine graphs, or
- Through AI Engine array kernels, memories and interfaces.

Host application event tracing is enabled in the file `xrt.ini`, and can be limited within the host code using specific API.

AI Engine event trace tools provide an in-depth investigation into the operation and performance of a design. In support of this, a number of settings are required to capture trace data at runtime. In hardware, you must prepare the design when compiling the AI Engine graph application to ensure the `libadf.a` supports capturing trace data at runtime. In order to trace events in hardware, use the `--event trace` option when compiling the AI Engine graph. This option sets up the hardware device to capture AI Engine runtime trace data.

The AI Engine event trace flow consists of three parts:

1. Event trace build flow.
2. Running the design in hardware and capturing trace data at runtime.

3. Viewing and analyzing the trace data using the Vitis IDE.

***Note*:** Event trace is also supported on BDC platforms vck_190_bdc, as described in Vitis Base Platform for the VCK190 Board, and vek_280_bdc, as described in Vitis Base Platform for the VEK280 Pre-production Board

Enabling Trace in Your Application

To enable event trace data during the execution of your application, you must instrument your application for this task. You must enable additional logic, and consume additional device resources to track the host and kernel execution steps, and capture event data. This process requires optionally modifying your host application to capture custom data, modifying your kernel XO during compilation and the `xclbin` during linking to capture different types of profile data from the device side activity, and configuring the Xilinx Runtime (XRT) as described in the xrt.ini File or using xsdb command line to capture data during the application runtime.

In these traditional hardware event trace, the trace information is stored in DDR memory available in the Versal device initially, and offloaded to SD card after the application run completes. This imposes limitations on the amount of trace information that can be stored and analyzed.

The high-speed debug port (HSDP) debug port provides debugging and trace capability for programmable logic (PL), processing system (PS), and AI Engines through a dedicated Aurora interface and a high-speed debug cable like SmartLynq+. The HSDP leverages the high-speed gigabit transceivers to make debug less intrusive to the system configuration. AI Engine trace offload via HSDP has more DDR memory in the SmartLynq+ module and supports analyzing large quantities of trace information for complex designs. In addition, the SmartLynq+ module offers high bandwidth connectivity to offload trace information via HSDP which is faster that standard JTAG connection. HSDP bandwidth is lower than direct DDR storage but allows much larger trace data set to be stored and analyzed. More details are available on Event Trace Offload using High Speed Debug Port in the *AI Engine Tools and Flows User Guide* (UG1076).

TIP: *While capturing profile data is a critical part of the profiling and optimization process for building your application, it does consume additional resources and impacts performance. You should be sure to clean these elements out of your final production build.*

There are many different types of profiling for your applications, depending on which elements your system includes, and what type of data you want to capture. The following table shows some of the levels of profiling that can be enabled, and discusses which are complimentary and which are not.

Table 47: **Event Trace For Host Application, PL Kernels, and AI Engine Graphs**

Trace	Description	Comments
Host Application XRT Native API	Specified by the use of the `native_xrt_trace` option in the `xrt.ini` file.	Generates profile summary and trace events for the XRT API as described in Writing the Software Application in the *Data Center Acceleration using Vitis* (UG1700).
Host Application User-Event Profiling	Requires additional code in the host application as described in Custom Profiling of the Host Application.	Generates user range data and user events for the host application. **TIP:** *Can be used to capture event data for user-managed kernels as described in Working with User-Managed Kernels in the Data Center Acceleration using Vitis (UG1700).*
AI Engine Graph and Kernels	Specified by the use of the `aie_trace` options in the `xrt.ini` file.	Generates the `default.aierun_summary` report containing the Trace reports. The `aierun_summary` can be found in the `aiesimulator_output` folder of the AI Engine graph build directory. Refer to the AI Engine Simulation-Based Profiling chapter in the *AI Engine Tools and Flows User Guide* (UG1076) for more information.

The device binary (`xclbin`) file is configured for capturing limited device-side profiling data by default. However, using the `--profile` option during the Vitis compiler linking process instruments the device binary by adding AXI Performance Monitors, and Memory Monitors to the system. This option has multiple instrumentation options: `--profile.data`, `--profile.stall`, and `--profile.exec`, as described in the --profile Options.

As an example, add `--profile.data` to the `v++` linking command line:

```
v++ -g -l --profile.data all:all:all ...
```

TIP: *Be sure to also use the `v++ -g` option when compiling your kernel code for debugging with software or hardware emulation.*

After your application is enabled for profiling during the `v++` compile and link process, data gathering during application runtime must also be enabled in XRT by editing the `xrt.ini` file as discussed above.

To enable the profiling of Kernel Internals data, you must also add the `debug_mode` tag in the `[Emulation]` section of the `xrt.ini`:

```
[Emulation]
debug_mode=batch
```

If you are collecting a large amount of trace data, you can increase the amount of available memory for capturing data by specifying the `--profile.trace_memory` option during `v++` linking, and add the `trace_buffer_size` keyword in the `xrt.ini`.

- **`--profile.trace_memory`:** Indicates the type of memory to use for capturing trace data.
- **`trace_buffer_size`:** Specifies the amount of memory to use for capturing the trace data during the application runtime.

> **TIP:** *When `--profile.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 size.*

Finally, as discussed in Continuous Trace Capture you can enable continuous trace capture to continuously offload device trace data while the application is running, so in the event of an application or system crash, some trace data is available to help debug the application.

Continuous Trace Capture

The Vitis tool supports recording continuous trace data while the application is running. The application can run for a very long time thus leading to the capture of significant trace data, which can result in issues like incomplete trace data especially when the memory resource used for trace data is not large enough. Using continuous trace, analysis of the trace can be carried out while the application is still running or if the application has crashed before completion.

With the ability to continuously capture trace data, the Timeline Trace reports can be dynamically updated in the Vitis analyzer tool while your application is running. Once these reports are loaded in Vitis Analyzer, there is a hyperlink available indicating that the current report is being modified on the disk. If new data needs to be loaded, **Reload** or **Auto-Reload** options are available on the banner to let you view the updated report as your application runs and trace data is generated.

Continuous trace is not enabled by default. Additionally, the memory resources of an FPGA are not unlimited. So if the application generates large trace data, a circular buffer for storing the data can be used. The circular buffer can be written, offloaded to the host, and reused again. By enabling a circular buffer with continuous trace, the memory resources needed are even smaller thus saving available resources on the device. However, an application run with continuous trace/circular buffer can result in multiple device trace files.

> **TIP:** *For Hardware emulation, only host side continuous trace is available, for hardware runs both host side and device side continuous trace are available.*

Here are some scenarios where it is recommended to use the memory resource as a circular buffer.

The circular buffer implementation is automatically turned on when continuous trace is enabled in the `xrt.ini`. The flow requires the following settings for enabling continuous trace.

- In the `xrt.ini` file, `continuous_trace` is set to TRUE
- v++ linking option `--profile.trace_memory` is set to DDR or HBM

You can optionally set:

- The size of the trace buffer using `trace_buffer_size` in the `xrt.ini` file. This defaults to 1 MB.
- The interval at which the trace buffer is offloaded from the device using `trace_buffer_offload_interval_ms` in the `xrt.ini` file. The default is 10 ms.
- The interval at which files are dumped by setting `trace_file_dump_interval_s`. The default is 3 seconds.

IMPORTANT! *Circular Buffer can be force enabled by setting `trace_buffer_offload_interval_ms` to 0 ms.*

As an example, if you enable `continuous_trace` with `trace_buffer_size` as 8k and default `trace_buffer_offload_interval_ms` of 10 ms, the trace data rate is 819200 bytes/s which is less than the default of 100 MB/s. In this scenario, the circular buffer is *NOT enabled* by default and an XRT warning is reported:

```
[XRT] WARNING: Unable to use circular buffer for continuous trace offload.
Please increase trace buffer size and/or reduce continuous
trace interval. Minimum required offload rate (bytes per second) :
104857600 Requested offload rate : 819200
```

Here is an example of `xrt.ini` settings:

```
[Debug]
device_trace=fine
stall_trace=all
continuous_trace=true
// The following are optional and needed only in rare circumstances

trace_buffer_size=20M
trace_buffer_offload_interval_ms=10
trace_file_dump_interval_s=2
```

The following are the results of these settings:

- `device_trace`: Enables the collection of kernel activity to be added to profile summary and trace, `device_trace_0.csv` files are created with 0 being the device number.
- `stall_trace`: Enables the hardware generation of stalls into kernel instances.
- `continuous_trace`: Enables the continuous dumping of files for trace and the continuous reading of device data into the host.
- `trace_buffer_size`: Specifies the amount of memory to consume for trace data capture.
- `trace_buffer_offload_interval_ms`: Controls the reading of device data from the device to the host in milliseconds.

- `trace_file_dump_interval_s`: Controls the time between dumping of trace files in seconds.

As a result, there are several CSV files generated in addition to the `xrt.run_summary` as part of the application run using the above `xrt.ini` file. Vitis Analyzer only needs the generated `run_summary` file and will use the relevant CSV files to display the profile summary and timeline trace.

The following are recommendations on setting up an application for trace data dumping:

1. By default the memory used for trace capture is the first memory resource on the platform, which can be determined using the platforminfo Utility in the *Vitis Reference Guide* (UG1702). In most platforms this is either DDR or HBM. The amount of memory reserved for trace data is determined by the `trace_buffer_size` switch in the `xrt.ini` file, which defaults to 1 MB.

   Note: You can also specify the use of FIFO and the size to allocate using the `--profile.trace_memory` option.

2. If still unable to dump maximum trace, disable stall trace by setting `stall_trace=off` or `stall_trace=on` with `device_trace=coarse`.
3. If the application requires larger size of trace buffer, enable circular buffer by setting `continuous_trace=true` with default settings of `trace_buffer_offload_interval_ms=10` and `trace_file_dump_interval_s=5`. Ideally, a continuous trace feature should be used for the following cases:
   - Long-running design with minimal trace generated
   - Debugging application crashes where some `.csv` files might still be available for debugging
4. If the application run is still unable to dump the maximum trace, the `trace_buffer_size` can further be increased.
5. If the application still creates huge trace data that the host cannot keep up, use the smaller size of `trace_file_dump_interval`, which creates multiple files equivalent to the interval provided.
6. Lastly, continuous trace can generate several trace files as part of the application run in addition to `xrt.run_summary` file. The Vitis Analyzer only needs the generated `run_summary` file and can pick the relevant CSV files generated to display profile summary and timeline trace to provide a better experience.

Enabling AI Engine Status Generation

The AI Engine provides the ability to output a summary of AI Engine status, which include error events that have occurred. Warnings are also issued when a deadlock is detected in the hardware. The status and warnings can be further analyzed in the AMD Vitis™ IDE for debug purposes.

There are two ways to output AI Engine status when running designs in hardware.

- **Automated and periodic AI Engine status output:** After initial setup in `xrt.ini`, this method requires minimal user intervention, because the tool will output periodic status at specified time intervals.
- **Manual output the AI Engine status:** You must run a command each time you want a status output report.

Next, you can open the status output in the IDE for further analysis.

The following section covers the steps needed to obtain status output, and analysis of that output. For more details on understanding stalls, see the chapter on AI Engine Stall Analysis in *AI Engine Tools and Flows User Guide* (UG1076).

Periodic AI Engine Status Output Using XRT

You can enable the runtime deadlock detection and status output using the `xrt.ini` file. This is a one-time set up, which results in periodic output of status data, including deadlock detection. To turn on this feature, add the following to the `xrt.ini` file:

```
[Debug]
aie_status=true
```

To specify the interval of probing and analysis the AI Engine status, use the code below.

```
[Debug]
aie_status=true
aie_status_interval_us=1000
```

The AI Engine status is copied to the following files when the host program is running.

- `xrt.run_summary`: Run summary information that can be used by the IDE.
- `aie_status_<device>_<current time>.json`: Status of AI Engine, AI Engine memory and AI Engine Interface tiles.
- `summary.csv`: Always created for other profiling capabilities, like guidance.

For more details, see the chapter on Generating AI Engine Status in *AI Engine Tools and Flows User Guide* (UG1076).

Generating AI Engine Status Using XSDB

It is possible to examine the status of the AI Engine using XSDB both on Linux and Bare metal operating systems. This feature allows you to debug applications and detect the status of the AI Engine in situations where the board is in a deadlock or hung state. Unlike the `xrt-smi` command which requires XRT, the XSDB command is run independent of XRT.

XSDB reports the AI Engine status in `.json` file format using the `aiestatus examine` command. You can use this command before, during and after running the application.

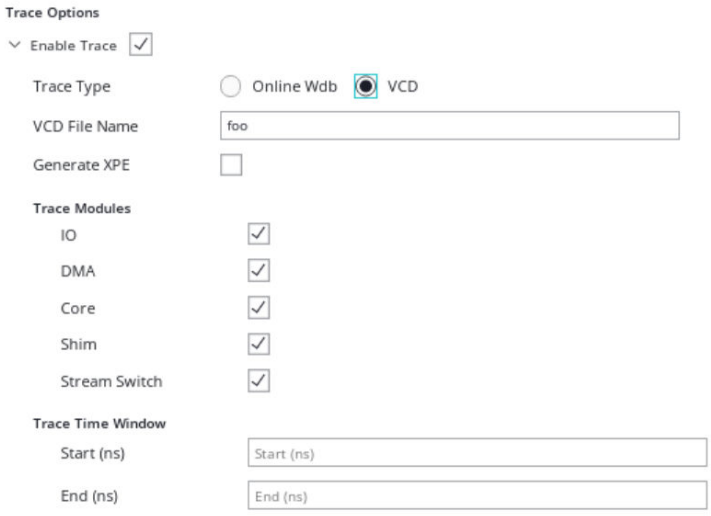
XSDB command is very simple:`aiestatus examine`

For all available options, see the chapter on Generating AI Engine Status using XSDB in *AI Engine Tools and Flows User Guide* (UG1076).

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 49: **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. If 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
```

For more details on the `aiesimulator` options see Simulating an AI Engine Graph Application in the *AI Engine Tools and Flows User Guide* (UG1076) and the AI Engine Simulator chapter.

AI Engine Event Trace in Hardware

To obtain trace data during hardware run, there must be routes dedicated to driving trace data from the AI Engine array to the PL or to the DDR. For this reason, during the graph compilation phase, you need to specify the trace data during hardware run and the interface to be used. See the following code for details.

```
v++ --c --mode aie --verbose --pl-freq=100 --workdir=./myWork \
--event-trace-port=gmio --event-trace=runtime \
--num-trace-streams=8 --xlopt=0 --include="./" \
--include="./src" --include="./src/kernels" --include="./data" \
./src/graph.cpp
```

- For `--event-trace=runtime`: the only possibility here is runtime, indicating that signal selection will be decided at runtime.
- For `--event-trace-port=plio/gmio`: selects GMIO and the NoC pathway instead of PLIO/PL pathway. PLIO uses PL logic, which can induce timing closure difficulties.
- For `--num-trace-streams=8`: up to 16 streams can be used within the AIE Engine array to drive the trace events to the GMIO/PLIO.

For the profiling flow, you can perform event trace using either XSDB or XRT flow.

The metrics for the array are described below.

Table 48: **AI Engine Metrics**

Metric Name	Description
functions	Basic time line of function activity: events generated when kernel functions are being invoked and returned

Table 48: **AI Engine Metrics** *(cont'd)*

Metric Name	Description
partial_stalls	Three types of core stalls are being registered: stream stalls (no data at input or back-pressure at output), cascade stalls and lock stalls.
all_stalls	Same as partial_stalls with memory_stalls (memory conflict) added.
all_dma	Data transfers of all 4 Memory DMA channels (2xS2MM, 2xMM2S)
all_stalls_dma	Core stalls and data transfers of all 4 DMA channels. All core stalls are grouped, no differentiation on the type of stall.
all_stalls_s2mm	Core stalls and data transfer of two S2MM channels[1]
all_stalls_mm2s	Core stalls and data transfer of two MM2S channels[1]
s2mm_channels	Data transfers and stalls of two S2MM channels
mm2s_channels	Data transfers and stalls of two MM2S channels
s2mm_channels_stall	Details of one S2MM channel.[2] In AI Engine-ML v2 based devices only
mm2s_channels_stall	Details of one MM2S channel[2]. In AI Engine-ML v2 based devices only

Notes:

1. In AI Engine based devices, the stall events are concatenated into a group stall event.
2. Includes Buffer Descriptors, tasks, starvation, back-pressure and lock stalls.

Table 49: **Interface Tiles**

Metric Name	Description
input_ports	Data transfers of 4 stream input from the AI Engine Array
input_port_stalls	Data transfers and stalls of 2 inputs from the AI Engine Array
input_port_details	Details on one MM2S channel [1]. For GMIOs only
output_port	Data transfers of 4 stream output to the AI Engine Array
output_port_stalls	Data transfers and stalls of 2 outputs to the AI Engine Array
output_port_details	Details on one S2MM channel. Includes Buffer Descriptors, tasks, starvation, back-pressure and lock stalls. For GMIOs only
input_output_ports	Data transfers of 4 inputs or outputs of AI Engine Array
input_output_ports_stalls	Data transfers and stalls of 2 inputs or output of the AI Engine Array

Table 50: **Memory Tiles (AI Engine-ML and AI Engine-ML v2)**

Metric Name	Description
s2mm_channels	Buffer Descriptor and Task events for two S2MM channels
s2mm_channels_stalls	Details on one S2MM channels, adding lock stalls, back-pressure and stream starvation.
mm2s_channels	Buffer Descriptor and Task events for 2 MM2S channels
mm2s_channels_stalls	Details on one MM2S channel, adding lock stalls, back-pressure and stream starvation.
memory_conflicts1	Memory conflict for data memory banks 0-7
memory_conflicts2	Memory conflicts for data memory bank 8-15

XSDB Flow

When running the application, the trace data is stored in DDR memory by the debugging and profiling IP. To capture and evaluate this data, you must connect to the hardware device using `xsdb`. This command is typically used to program the device and debug bare-metal applications. Connect your system to the hardware platform or device over JTAG, launch the `xsdb` command in a command shell, and run the following sequence of commands. In this sequence, the `source $::env(XILINX_VITIS)/scripts/vitis/util/aie_trace.tcl` command sources the Tcl `trace` command to set up the `xsdb` environment.

```
xsdb% connect
xsdb% ta
xsdb% ta 1
xsdb% source $::env(XILINX_VITIS)/scripts/vitis/util/aie_trace.tcl
xsdb% aietrace start -graphs mygraph -work-dir ./Work -link-summary
$PROJECT/xsa.link_summary -base-address 0x900000000 -depth 0x800000 -tile-
based-aie-tile-metrics "all:functions; {4,1}:{6,2}:all_stalls"

# Execute the PS host application (.elf) on Linux
## After the application completes processing.
xsdb% aietrace stop
```

After the hardware events are captured on the SD card, offload them on your computer and launch the Vitis Unified IDE to import and analyze data:

```
vitis -a aie_trace_profile.run_summary
```

For more details on this flow, see the chapters on Event Tracing in Hardware and XSDB flow in the *AI Engine Tools and Flows User Guide* (UG1076).

XRT Flow

Within the XRT flow, the selection of trace events is performed in the `xrt.ini` file in the SDCard. An example of such an `xrt.ini` file is shown hereafter:

```
# Main switch to turn on aie trace
[Debug]
aie_trace = true
# Continuous trace knobs
[AIE_trace_settings]
reuse_buffer = true
periodic_offload = true
# Time to wait between trace reads
buffer_offload_interval_us = 100
# Total amount of device memory shared between trace streams
buffer_size = 16M
# granularity
graph_based_aie_tile_metrics = all:all:functions
```

For more details, see the chapters on Event Tracing in Hardware and XRT Flow in the *AI Engine Tools and Flows User Guide* (UG1076).

Chapter 11

Debugging System Projects

The AMD Vitis™ unified software platform provides application-level debug features and techniques that allow the Application component, AI Engine component, PL kernels, and the interactions between them to be debugged. However, debugging projects built from the command line is a challenge because the various elements of the system, the compiled AI Engine graph application (`libadf.a`), the device binary (`.xclbin`), and the top-level application (`host.elf`), must be gathered together and presented as a system.

The Vitis unified IDE provides an excellent framework for debugging these heterogeneous systems. There are many advantages to working in the Debug view in the IDE. In fact, you are strongly recommended to debug your command-line driven projects in the IDE. The process for doing this is broken down into two steps:

1. Import your command-line project into the IDE as described in Getting Started with Vitis in the *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400).
2. Debug the system in the IDE as described in Debugging the System Project and AI Engine Components in the *Vitis Reference Guide* (UG1702).

The Vitis tools provide application-level debug features which let the host code, the system project, and the interactions between them be efficiently debugged in the Vitis unified IDE. These features and techniques are split between software debugging and hardware debugging flows. The recommended debugging flow consists of three levels of debugging:

- The AI Engine graph can be simulated and debugged on its own using GDB or the trace and profile tools available within the aiesimulator (see Simulating an AI Engine Graph Application in the *AI Engine Tools and Flows User Guide* (UG1076)). The input and output of the graph can be managed by external traffic generators using C++, Python or Verilog (see External Traffic Generator in the *AI Engine Tools and Flows User Guide* (UG1076)). This is a first step where the host application is not debugged, but the graph can be debugged using realistic data flows.
- Debugging in Hardware Emulation in the *Data Center Acceleration using Vitis* (UG1700) to compile the PL kernels into RTL, confirm the behavior of the generated logic, and evaluate the simulated performance of the hardware.
- Debugging During Hardware Execution to implement the device binary and debug the application running on hardware using Xilinx virtual cable (XVC) running over the PCIe® bus, or debugged using USB-JTAG cables for embedded processor platforms.

This three-tiered approach enables debugging the Application component and System project at different levels of abstraction. Each provides specific insights into the design providing a comprehensive view of the system from software to hardware. All flows are supported through the Vitis unified IDE using basic compile time and runtime setup options.

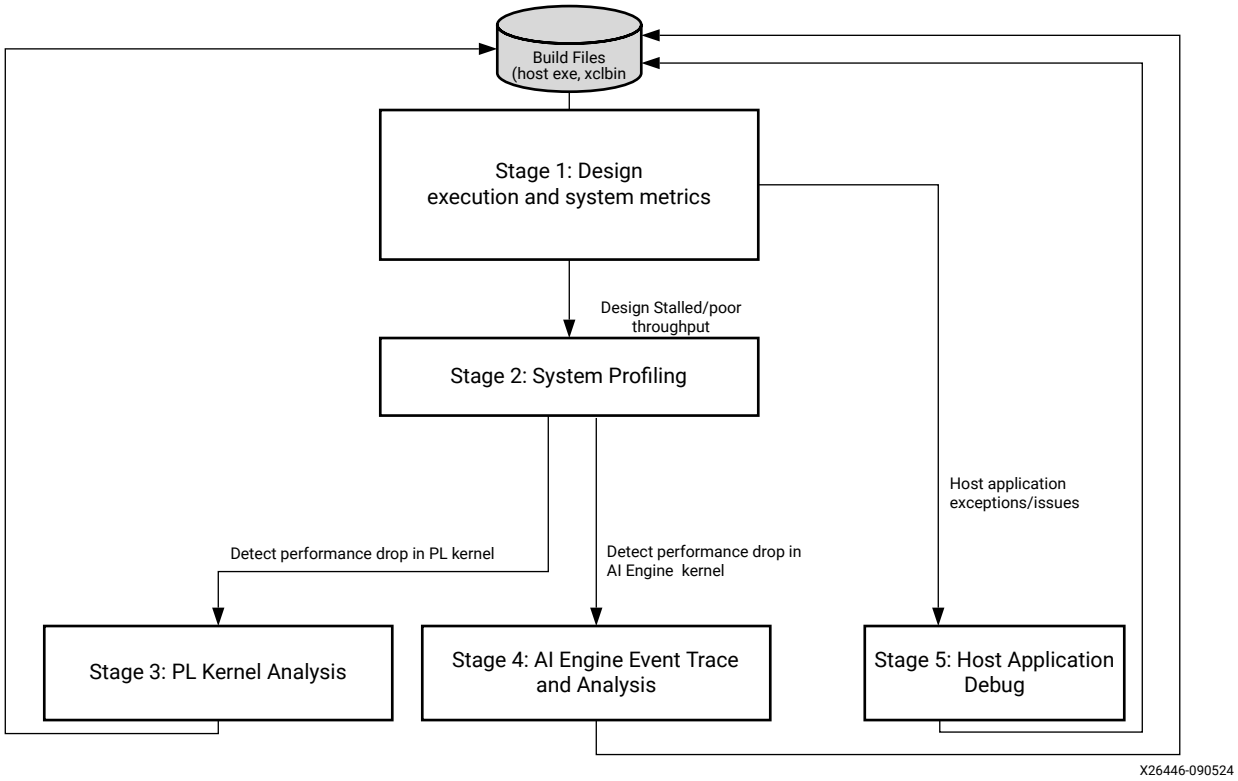
Hardware Profile and Debug Methodology

Designs running on AMD Versal™ AI Engine devices can target the AI Engine, PL, and Arm® host. To ensure a design targeting such multi-domain devices is functionally correct and meets the design performance specification, AMD recommends a five-stage profile and debug methodology in hardware.

The stages are as follows:

1. Design Execution and System Metrics
2. System Profiling
3. PL Kernel Analysis
4. AI Engine Event Trace and Analysis
5. Host Application Debug

Figure 50: **Five Stages of Profile and Debug Methodology**



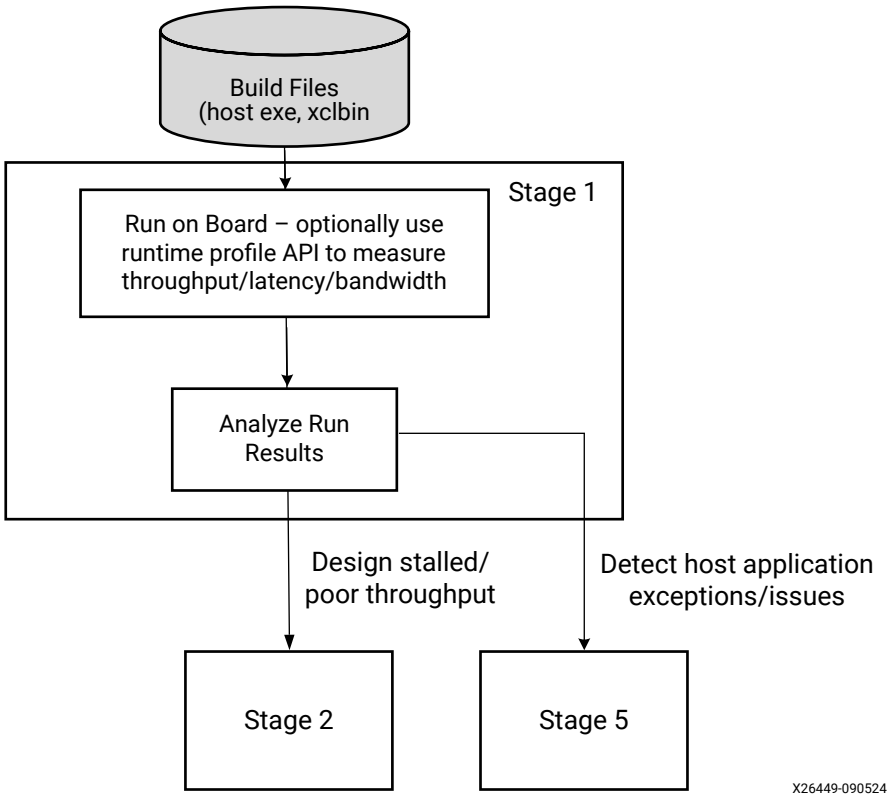
The goal of each stage along with available tools and techniques are described below.

Stage 1: Design Execution and System Metrics

The goal of the first stage is to determine if the design is functionally correct and can run successfully in hardware. You can also analyze results, and proceed to the next stage for further debug and analysis.

The following figure shows the tasks and techniques available in this stage.

Figure 51: **Design Execution and System Metrics**



This stage can help you determine if the design and host application can run successfully in hardware. In this stage you can use APIs in your host application to profile the design as it is running on hardware, and determine if the design meets throughput, latency and bandwidth goals. In addition, you can troubleshoot AI Engine stalls and deadlocks using reports generated when running the design in hardware.

The sections below list the techniques available in this stage.

Error Handling and Reporting in Host Application in Hardware

On Linux, XRT provides error reporting APIs. Use these APIs in your host application to catch and report these errors to get to the root cause of the issue. For details on the XRT error reporting APIs, see AI Engine Error Reporting in the *AI Engine Tools and Flows User Guide* (UG1076). To examine error messages reported by the AI Engine array, enable and examine the `dmesg` logs. Details on AI Engine array-specific error handling can be found in AI Engine Error Events in the *AI Engine Tools and Flows User Guide* (UG1076).

Next stage: Proceed to stage 5 if you determine that the host application needs to be debugged further.

***Note*:** For errors flagged on the AI Engine, the error handling table provides guidance on next steps.

Analyzing Design Stalls in Hardware

If you encounter design stalls in hardware on Linux, track the status of the AI Engine and PL kernels in the design using the XRT xrt-smi utility on Linux. For more information on how to use the utility to generate a report on the current design status and to visualize the results in the Vitis IDE, see Analyzing AI Engine Status in Hardware in the *AI Engine Tools and Flows User Guide* (UG1076).

You can also manually generate the report in XSDB and visualize the results in the Vitis IDE. More more details, see Generating AI Engine Status in the *AI Engine Tools and Flows User Guide* (UG1076). Some examples of deadlock visualizations in the Vitis IDE are found in Analyzing the Automated Status Output in the *AI Engine Tools and Flows User Guide* (UG1076).

Next stage: Proceed to stage 2 if you determine that the design is stalled and further details are needed to get to the root cause of the stall.

Reporting Design Throughput, Latency, Bandwidth in Hardware

You can also determine the AI Engine graph throughput, latency and bandwidth by profiling the graph inputs and outputs via APIs in the host application. Careful consideration is needed on when profiling the API is started and stopped in the host. For details on how to use the APIs in the host application, see Event Profile APIs for Graph Inputs and Outputs in the *AI Engine User Guide* (UG1076).

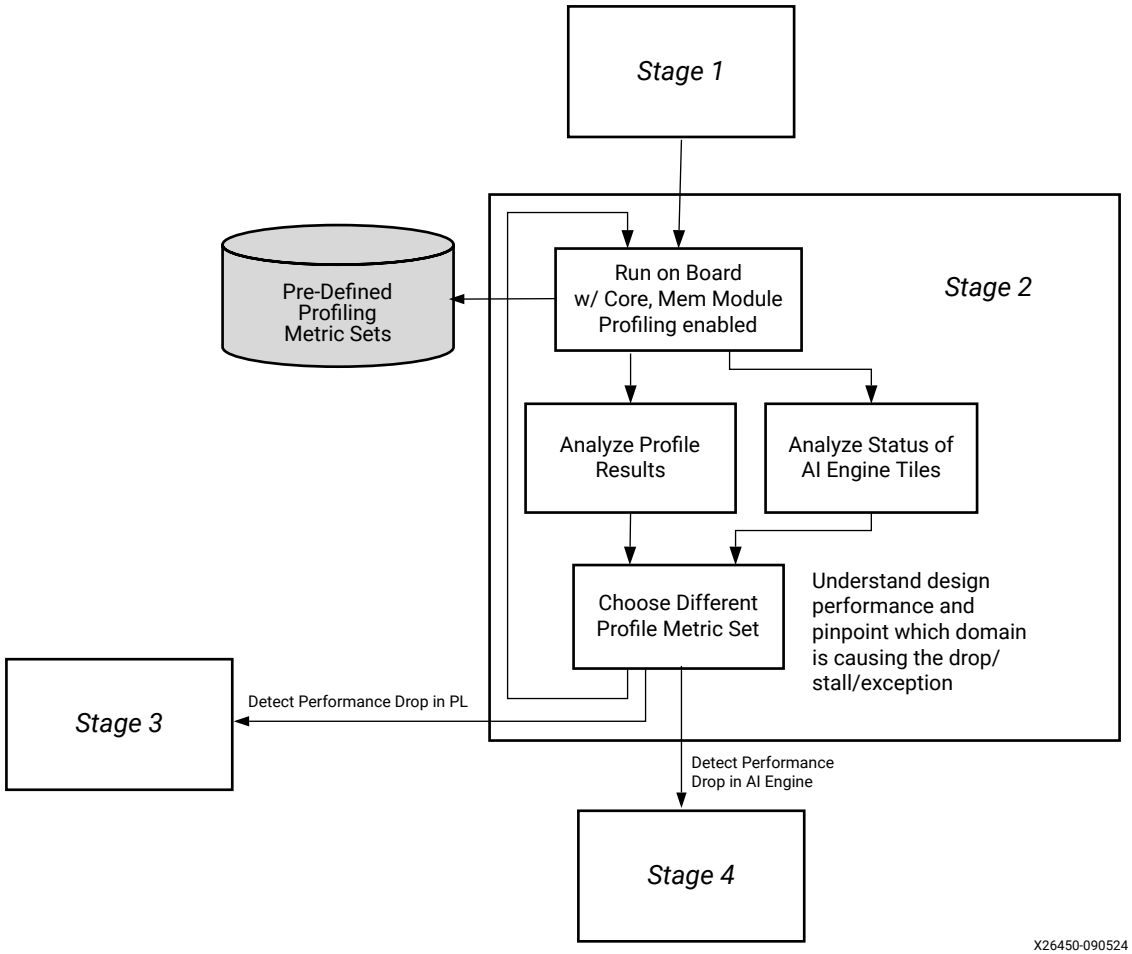
Next stage: Proceed to stage 2 if you determine that the design has sub-par throughput or latency, or the design does not meet the performance goal. In stage 2, you can pinpoint the kernel or I/O that might be contributing to the performance drop.

Stage 2: System Profiling

The goal of this stage is to profile the design and determine which domain (AI Engine, PL, NoC) is causing a throughput drop, which causes the design to stall.

The following figure shows the tasks and techniques available in this stage.

Figure 52: **System Profiling**



The section below lists the technique available in this stage.

Profiling AI Engine Core, Interface and Memory Module

You can profile the AI Engine Core, Interface, and Memory modules in XRT or XSDB flows. This is a non-intrusive feature which can be enabled at runtime using the `xrt.ini` file or running scripts in XSDB. The feature uses performance counters available in the AI Engine array to gather profile data. The amount and type of data gathered is limited by the number of performance counters available.

Table 51: **AI Engine Metrics**

AI Engine metrics	
heat_map	Profiles active, stall, vector instructions and cumulative instructions time. These metrics reveals the efficiency of the AI Engine not only in terms of code efficiency (vector instructions) but also in terms of interaction with memory and streams.
stalls	Profiles memory, stream, lock and cascade stalls. These metrics allows a deeper analysis of the reasons of the stalls detected with `heat_map` metric set.

Table 51: **AI Engine Metrics** *(cont'd)*

AI Engine metrics	
execution	Profiles vector, load and store instructions time. With these metrics you can determine the efficiency of your kernel code.
floating-point	Profiles all floating-point exceptions. If you are using floating-point arithmetic, these metrics highlight the exceptions that occur in the code.
aie_trace	Profiles AI Engine and memory module trace word count and stall count. This is useful to determine if you have congestion in the trace stream when you use the event trace feature.
write_bandwidths	Profiles stream write, cascade write and stalls time. This is an indicator of the efficiency of the stream and cascade output. If there are many stalls, this indicates that the next kernel in the graph cannot consume data quickly enough and this could impact the design throughput.
read_bandwidth	Profiles stream read, Cascade read and stalls time. This is an indicator of the efficiency of the stream and cascade input. If there are many stalls, this indicates that the previous kernel in the graph cannot provide data quickly enough and this could impact the design throughput.

Table 52: **Memory Module Metrics**

Memory Module Metrics	
conflicts	Profiles memory conflicts and memory errors. Memory conflicts happen when two memory chunks reside in the same memory bank and are accessed either by the same AI Engine (using the two read ports) or by two different AI Engines. A potential solution is to constrain the locations of these memories to different banks. In order to get more details about which bank is causing these conflicts, you should analyze the events from an emulation-AI Engine simulation or perform event trace in hardware.
dma_locks	Profiles lock activities on both DMAs. The four DMA channels (2xS2MM and 2xMM2S) are driven by Buffer Descriptors (BDs). The Cumulative DMA Activity is a count of the time taken due to stalled lock acquire events on all channels. All these DMA events will help you understand why some connections through the device are slower than expected.
dma_stalls_s2mm	Profiles DMA stalls on the s2mm channels due to a lock acquisition conflict. A stalling s2mm DMA indicates that there is a conflict when accessing the target memory. This can be due to another s2mm or mm2s DMA accessing the same bank or a kernel performing a memory access leading to a lock acquisition conflict.
dma_stalls_mm2s	Profiles DMA stalls on the mm2s channels due to a lock acquisition conflict. A stalling mm2s DMA indicates that there is a conflict when accessing the source memory. This can be due to another s2mm or mm2s DMA accessing the same bank or a kernel performing a memory access leading to a lock acquisition conflict.
write_bandwidths	Profiles bandwidth used by the s2mm DMA. Allows you to evaluate if you achieve your bandwidth goals.
read_bandwidths	Profiles bandwidth used by the mm2s DMA. Allows you to evaluate if you achieve your bandwidth goals.

Table 53: **Interface Tile Metrics**

Interface Tile Metrics	
input_bandwidths	Profiles input PLIO channel bandwidth in addition to stalls and idle time. If input bandwidth is too low, this can be due to a high stall rate, which means that the AI Engine array does not consume the samples at the right rate. Proceed to AI Engine event trace (stage 4). It can also be due to a high idle rate which means that the PL side of the design does not produce samples at the right rate. Proceed to PL Kernel analysis (stage 3).

Table 53: **Interface Tile Metrics** *(cont'd)*

Interface Tile Metrics	
output_bandwidths	Profiles output PLIO channel bandwidth in addition to stalls and idle time. If output bandwidth is too low, this can be due to a high idle rate, which means that the AI Engine array does not produce the samples at the right rate. Proceed to AI Engine event trace (stage 4). It can also be due to a high stall rate which means that the PL side of the design does not consume samples at the right rate. Proceed to PL Kernel analysis (stage 3).
packets	Profiles number of input and output packets

You can run the design multiple times, rebooting the board in between each run, with different parameters in the file `xrt.ini`. The Vitis IDE allows you to consolidate the different `xrt.run.summary` files reports so that you have a global view on the various bandwidths, stalls and idles at the interface level.

For details on how to enable profiling in hardware and interpreting the results, see Profiling the AI Engine in the *AI Engine Tools and Flows User Guide* (UG1076).

The profile results allow you to quickly identify the exact AI Engine, input stream or output stream involved in the design performance drop.

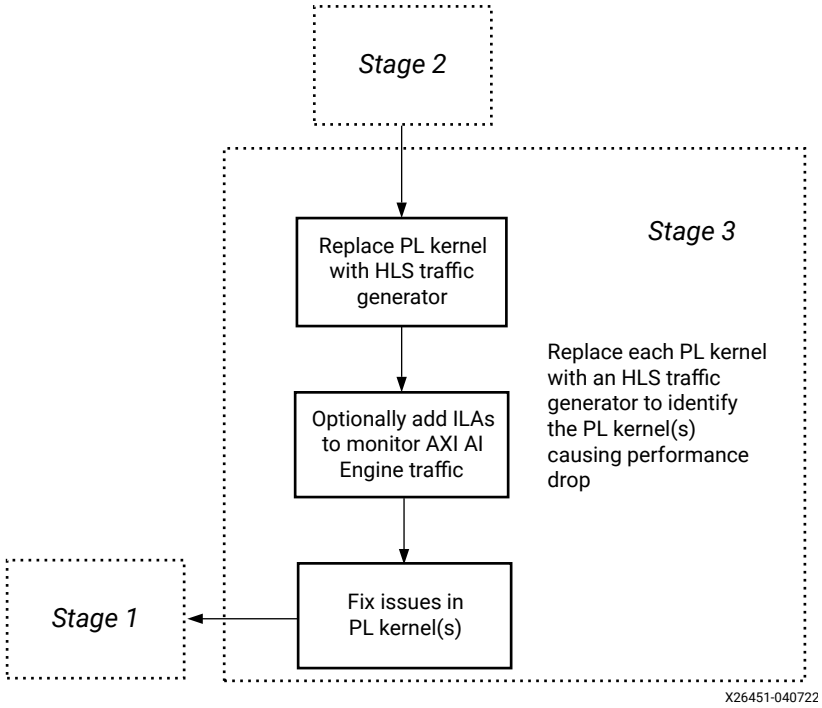
Next Stage:

- Proceed to stage 3 if you determine that a PL kernel is causing the performance drop. In stage 3, you can identify the exact PL kernel(s) with the sub-par performance.
- Proceed to stage 4 if you determine that an AI Engine kernel is causing the throughput drop.

Stage 3: PL Kernel Analysis

The goal of this stage is to determine the exact PL kernel(s) causing a throughout drop.

Figure 53: **PL Kernel Analysis**



The sections below list the different techniques available in this stage.

Profiling Using PL Profile Monitors

You can insert PL profile monitors using the v++ link command. This allows you to monitor active, stalled cycles and bytes transferred on specific PL-AI Engine interfaces. This can be enabled along with event tracing in the AI Engine to minimize the build time. This will allow you to identify specific PL kernel(s) causing a performance drop. For more information on the option for adding PL profile monitors, see --profile Options.

Replacing PL Kernels

You can replace each of the PL kernel that you suspect might be contributing a performance drop with non-throttling PL kernels. This allows you to determine if the PL kernel is responsible for the performance drop.

Inserting ILA(s) to Monitor Specific AXI Interfaces

You can insert one or more ILAs to monitor specific PL AXI interfaces to help identify exactly where and when a throughput drop occurs. It will also help you identify how frequently a throughput drop occurs. For details on the option to insert ILAs using the v++ command line, see Enabling Kernels for Debugging with Chipscope.

Next Stage: After you determine the cause of throughput drop and fix the issue, proceed to stage 1 to rerun the design.

Debugging During Hardware Execution

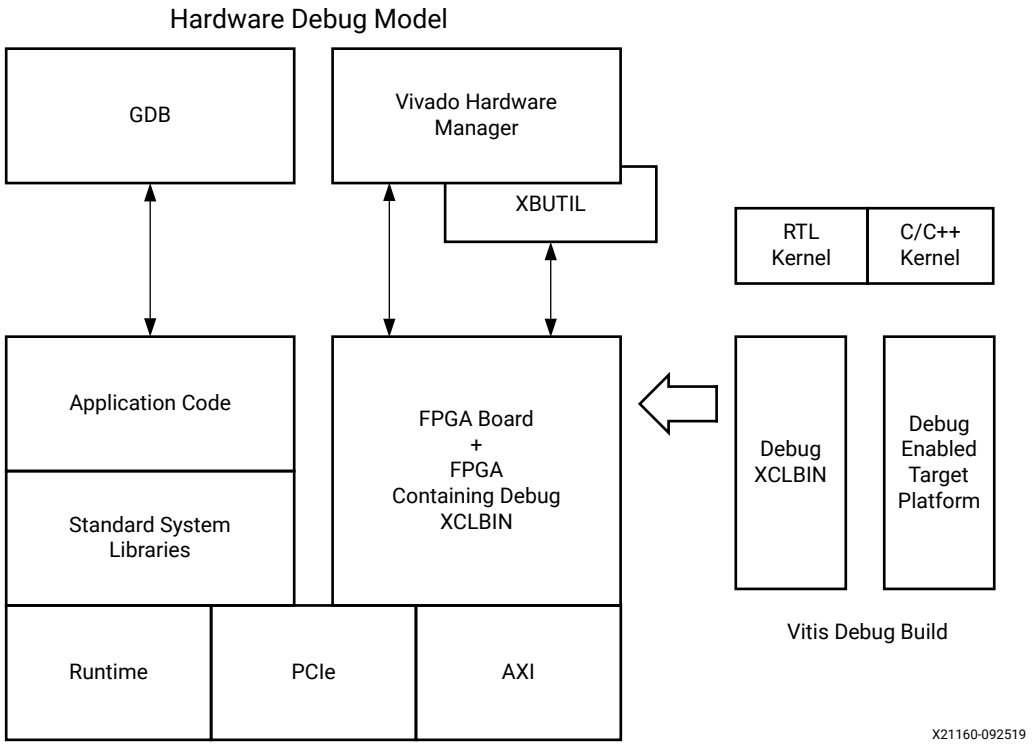
IMPORTANT! *The following steps describe debugging from the command line. However, you are strongly encouraged to debug command-line projects in the Vitis unified IDE by opening them in a workspace as described in Getting Started with Vitis in the Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400), and debugging them as described in Debugging the System Project and AI Engine Components in the Vitis Reference Guide (UG1702).*

During hardware execution, the actual hardware platform is used to execute the kernels, and you can evaluate the performance of the host program and kernels by running the application. However, debugging the hardware build requires additional logic to be incorporated into the application. This will impact both the FPGA resources consumed by the kernel and the performance of the kernel running in hardware. The debug configuration of the hardware build includes special ChipScope debug cores, such as Integrated Logic Analyzer (ILA) and Virtual Input/Output (VIO) cores, and AXI performance monitors for debug purposes.

TIP: *The additional logic required for debugging the hardware should be removed from the final production build.*

The following figure shows the debug process for the hardware build, including debugging the host code using GDB using the Vivado hardware manager with waveform analysis, kernel activity reports, and memory access analysis to identify and localize hardware issues.

Figure 54: **Hardware Execution**



With the system hardware build configured for debugging, the host program running on the CPU and the Vitis kernels running on the AMD device can be confirmed to be executing correctly on the actual hardware of the target platform. Some of the conditions that can be identified and analyzed include the following:

- System hangs caused by protocol violations:
  - These violations can take down the entire system.
  - These violations can cause the kernel to obtain invalid data or to hang.
  - It is hard to determine where or when these violations originate.
  - To debug this condition, you should use an ILA triggered off of the AXI protocol checker, which needs to be configured on the Vitis target platform.
- Problems with the hardware kernel:
  - Problems sometimes caused by the implementation: timing issues, race conditions, and bad design constraints.
  - Functional bugs that hardware emulation does not reveal.
- Performance issues:
  - The frames per second processing that are not what you expected.

- Examining data beats and pipelining.
- Using an ILA with trigger sequencer, you can examine the burst size, pipelining, and data width to locate the bottleneck.

Checking the FPGA Board for Hardware Debug Support

Supporting hardware debugging requires the platform to support several IP components, most notably the Debug Bridge. Talk to your platform designer to determine if these components are included in the target platform. If an AMD platform is used, debug availability can be verified using the `platforminfo` utility to query the platform. Debug capabilities are listed under the `chipscope_debug` objects.

For example, to query the a platform for hardware debug support, the following `platforminfo` command can be used:

```
$ platforminfo --json="hardwarePlatform.extensions.chipscope_debug"
xilinx_u250_gen3x16_xdma_4_1_202210_1
{
    "debug_networks": {
        "user": {
            "bar_number": "0",
            "supports_jtag_fallback": "false",
            "name": "User Debug Network",
            "supports_microblaze_debug": "true",
            "pcie_pf": "1",
            "axi_baseaddr": "0x000001C00000",
            "is_user_visible": "true"
        },
        "mgmt": {
            "bar_number": "0",
            "supports_jtag_fallback": "true",
            "name": "Management Debug Network",
            "supports_microblaze_debug": "true",
            "pcie_pf": "0",
            "axi_baseaddr": "0x01F60000",
            "is_user_visible": "false"
        }
    }
}
```

The response shows that the target platform contains `user` and `mgmt` debug networks, supports debugging a MicroBlaze™ processor, and also supports JTAG fallback for the Management Debug Network.

Enabling Kernels for Debugging with Chipscope

System ILA

The key to hardware debugging lies in instrumenting the kernels with the required debug logic. The following topic discusses the `v++` linker options that can be used to list the available kernel ports, enable the System Integrated Logic Analyzer (ILA) core on selected ports, and enable the AXI Protocol Checker debug core for checking for protocol violations.

The ILA core provides transaction-level visibility into an instance of a kernel running on hardware. AXI traffic of interest can also be captured and viewed using the ILA core. The ILA provides custom event triggering on one or more signals to allow waveform capture at system speeds. The waveforms can be analyzed in a viewer and used to debug hardware, finding protocol violations, or performance issues. It can also be crucial for debugging difficult situation like application hangs.

Captured data can be accessed through the Xilinx virtual cable (XVC) using the Vivado tools. See the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) for complete details.

The ILA core can be added to an existing RTL kernel to enable debugging features within that design, or it can be inserted automatically by the `v++` compiler during the linking stage. The `v++` command provides the `--debug` option as described in --debug Options to attach System ILA cores at the interfaces to the kernels for debugging and performance monitoring purposes.

IMPORTANT! *ILA debug cores require system resources, including logic and local memory to capture and store the signal data. Therefore they provide excellent visibility into your kernel, but they can affect both performance and resource utilization.*

The `--debug` option to enable ILA IP core insertion has the following syntax:

```
--debug.chipscope <kernel_instance_name>[:<interface_name>]>
```

TIP: *The `<interface_name>` is optional, and if not specified all ports on the kernel instance will be analyzed. You can use the `--debug.list_ports` option to return the interface names on the kernel to use with `--debug` options.*

In case of a flattened design or any design where there would be multiple debug bridges in master mode, the flow will not pick one to stitch the debug cores, a constraint is needed to define the connectivity. For example in a Samsung Smart SSD U.2 flat shell, there is no partitioning between the static and dynamic regions while generating the kernels with the debug (ILA) options enabled. It is required to specify the connectivity of the kernel AXI ports that needs to be under debug to the user debug bridge in the dynamic region.

To specify the connectivity, you must provide the option below in the `v++` command line:

```
--advanced.paramcompiler.userPostDebugProfileOverlayTcl=<path to
post_dbg_profile_overlay.tcl >
```

Inside the `post_dbg_profile_overlay.tcl`, the file must call the XDC file with the connect debug core command and mention its processing order.

For example, the contents in the `post_dbg_profile_overlay.tcl` file are given below.

```
read_xdc < path to the connect_debug_core.xdc file>
set_property used_in_implementation TRUE [get_files <path to the
connect_debug_core.xdc file>]
set_property PROCESSING_ORDER EARLY [get_files <path to the
connect_debug_core.xdc file>]]
```

In the `connect_debug_core.xdc` file, you have to specify the `connect_debug_cores` constraint.

For example:

```
connect_debug_cores -master [get_cells -hierarchical -filter {NAME =~
*debug_bridge_xsdbm/inst/xsdbm}]
-slaves [get_cells -hierarchical -filter {NAME =~ level0_i/ulp/
system_ila_0}]
```

AXI Protocol Checker

The AXI Protocol Checker core monitors AXI interfaces. When attached to an interface, it actively checks for protocol violations and provides an indication of which violation occurred. You can assign it for all kernel instances in the design, or for specific kernel instances and ports.

The `--debug` option to enable AXI Protocol Checker insertion has the following syntax:

```
 --debug.protocol all
```

The protocol checker can be specified with the keyword `all`, or the `<cu_name>:<interface_name>`.

***Note*:** The `--debug.list_ports` option can be specified to return the actual names of ports on the kernel to use with `protocol` or `chipscope`.

An example flow you could use for adding ILA or protocol checkers to your design is outlined below:

1. Compile the kernel source files into an XO file, using the `-g` option to instrument the kernel for debug features:

   ```
   v++ -c -g -k <kernel_name> --platform <platform> -o <kernel_xo_file>.xo
   <kernel_source_files>
   ```

2. After the kernel has been compiled into an XO file, use `--debug.list_ports` to cause the `v++` compiler to print the list of valid kernel instances and port combinations for the kernel:

   ```
   v++ -l -g --platform <platform> --connectivity.nk <kernel_name>:<kernel
   instances>:<kernel_nameN>
   --debug.list_ports <kernel_xo_file>.xo
   ```

3. Add the ILA or AXI debug cores on the desired ports by replacing `list_ports` with the appropriate `--debug.chipscope` or `--debug.protocol` command syntax:

```
v++ -l -g --platform <platform> --connectivity.nk
<kernel_name>:<kernel_instances>:<kernel_nameN>
--debug.chipscope <kernel_instances>:<interface_name> <kernel_xo_file>.xo
```

TIP: *The `--debug` option can be specified multiple times in a single `v++` command line, or configuration file to specify multiple kernel instances and interfaces.*

When the design is built, you can debug the design using the Vivado hardware manager as described in Debugging with ChipScope.

Adding Debug IP to RTL Kernels

IMPORTANT! *This debug technique requires familiarity with the Vivado Design Suite, and RTL design.*

You can also enable debugging in RTL kernels by manually adding ChipScope debug cores like the ILA and VIO in your RTL kernel code before packaging it for use in the Vitis development flow. From within the Vivado Design Suite, edit the RTL kernel code to manually instantiate an ILA debug core, or VIO IP from the AMD IP catalog, similar to using any other IP in Vivado IDE. Refer to the HDL Instantiation flow in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) to learn more about adding debug cores to your design.

The best time to add debug cores to your RTL kernel is when you create it. However, debug cores consume device resources and can affect performance, so it is good practice to make one kernel for debug and a second kernel for production use. The `rtl_vadd_hw_debug` of the RTL Kernels examples on GitHub shows an ILA debug core instantiated into the RTL kernel source file. The ILA monitors the output of the combinatorial adder as specified in the `src/hdl/krnl_vadd_rtl_int.sv` file.

```
    // ILA monitoring combinatorial adder
    ila_0 i_ila_0 (
        .clk(ap_clk),              // input wire         clk
        .probe0(areset),           // input wire [0:0]   probe0
        .probe1(rd_fifo_tvalid_n), // input wire [0:0]   probe1
        .probe2(rd_fifo_tready),   // input wire [0:0]   probe2
        .probe3(rd_fifo_tdata),    // input wire [63:0]  probe3
        .probe4(adder_tvalid),     // input wire [0:0]   probe4
        .probe5(adder_tready_n),   // input wire [0:0]   probe5
        .probe6(adder_tdata)       // input wire [31:0]  probe6
    );
```

You can also add the ILA debug core using a Tcl script from within an open Vivado project, using the Netlist Insertion flow described in *Vivado Design Suite User Guide: Programming and Debugging* (UG908), as shown in the following Tcl script example:

```
create_ip -name ila -vendor xilinx.com -library ip -version 6.2
-module_name ila_0
set_property -dict [list CONFIG.C_PROBE6_WIDTH {32} CONFIG.C_PROBE3_WIDTH
{64} \
CONFIG.C_NUM_OF_PROBES {7} CONFIG.C_EN_STRG_QUAL {1}
CONFIG.C_INPUT_PIPE_STAGES {2} \
CONFIG.C_ADV_TRIGGER {true} CONFIG.ALL_PROBE_SAME_MU_CNT {4}
CONFIG.C_PROBE6_MU_CNT {4} \
CONFIG.C_PROBE5_MU_CNT {4} CONFIG.C_PROBE4_MU_CNT {4}
CONFIG.C_PROBE3_MU_CNT {4} \
CONFIG.C_PROBE2_MU_CNT {4} CONFIG.C_PROBE1_MU_CNT {4}
CONFIG.C_PROBE0_MU_CNT {4}] [get_ips ila_0]
```

After the RTL kernel has been instrumented for debug with the appropriate debug cores, you can analyze the hardware in the Vivado hardware manager as described in Debugging with ChipScope.

Enabling ILA Triggers for Hardware Debug

To perform hardware debug of both the host program and the kernel code running on the target platform, the application host code must be modified to let you set up the ILA trigger conditions *after* the kernel has been programmed into the device, but *before* starting the kernel.

Pausing the Host Application Using GDB

If you are running GDB to debug the host program at the same time as performing hardware debug on the kernels, you can also pause the host program as needed by inserting a breakpoint at the appropriate line of code. Instead of making changes to the host program to pause the application as needed, you can set a breakpoint prior to the kernel execution in the host code. When the breakpoint is reached, you can set up the debug ILA triggers in Vivado hardware manager, arm the trigger, and then resume the host program in GDB.

Debugging with ChipScope

You can use the ChipScope debugging environment and the Vivado hardware manager to help you debug your host application and kernels quickly and more effectively. These tools enable a wide range of capabilities from logic to system-level debug while your kernel is running in hardware. To achieve this, at least one of the following must be true:

- Your Vitis application project has been designed with debug cores, using the `--debug.xxx` compiler switch, as described in Enabling Kernels for Debugging with Chipscope.
- The RTL kernels used in your project need to be instantiated with debug cores (as described in Adding Debug IP to RTL Kernels).

Running XVC and HW Servers

The following steps are required to run the Xilinx virtual cable (XVC) and HW servers, host applications, and also trigger and arm the debug cores in the Vivado hardware manager.

1. Add debug IP to the kernel as discussed in Enabling Kernels for Debugging with Chipscope.
2. Modify the host program to pause at the appropriate point as described in Enabling ILA Triggers for Hardware Debug.
3. Set up the environment for hardware debug, using an automated script described in Automated Setup for Hardware Debug, or manually as described in Manual Setup for Hardware Debug.
4. Run the hardware debug flow using the following process:
   a. Launch the required XVC and the `hw_server` of the Vivado hardware manager.
   b. Run the host program and pause at the appropriate point to enable setup of the ILA triggers.
   c. Open the Vivado hardware manager and connect to the XVC server.
   d. Set up ILA trigger conditions for the design.
   e. Continue execution of the host program.
   f. Inspect kernel activity in the Vivado hardware manager.
   g. Rerun iteratively from step b (above) as required.

Automated Setup for Hardware Debug

1. Set up your Vitis core development kit as described in Setting Up the Vitis Environment.
2. Use the `debug_hw` script to launch the `xvc_pcie` and `hw_server` apps as follows:

   ```
   debug_hw --xvc_pcie /dev/xfpga/xvc_pub.<driver_id> --hw_server
   ```

   The `debug_hw` script returns the following:

   ```
   launching xvc_pcie...
   xvc_pcie -d /dev/xfpga/xvc_pub.<driver_id> -s TCP::10200
   launching hw_server...
   hw_server -sTCP::3121
   ```

   TIP: *The `/dev/xfpga/xvc_pub.<driver_id>` driver character path is defined on your machine, and can be found by examining the `/dev` folder.*

3. Modify the host code to include a pause statement *after* the kernel has been created/downloaded and *before* the kernel execution is started, as described in Enabling ILA Triggers for Hardware Debug.
4. Run your modified host program.

5. Launch Vivado Design Suite using the `debug_hw` script:

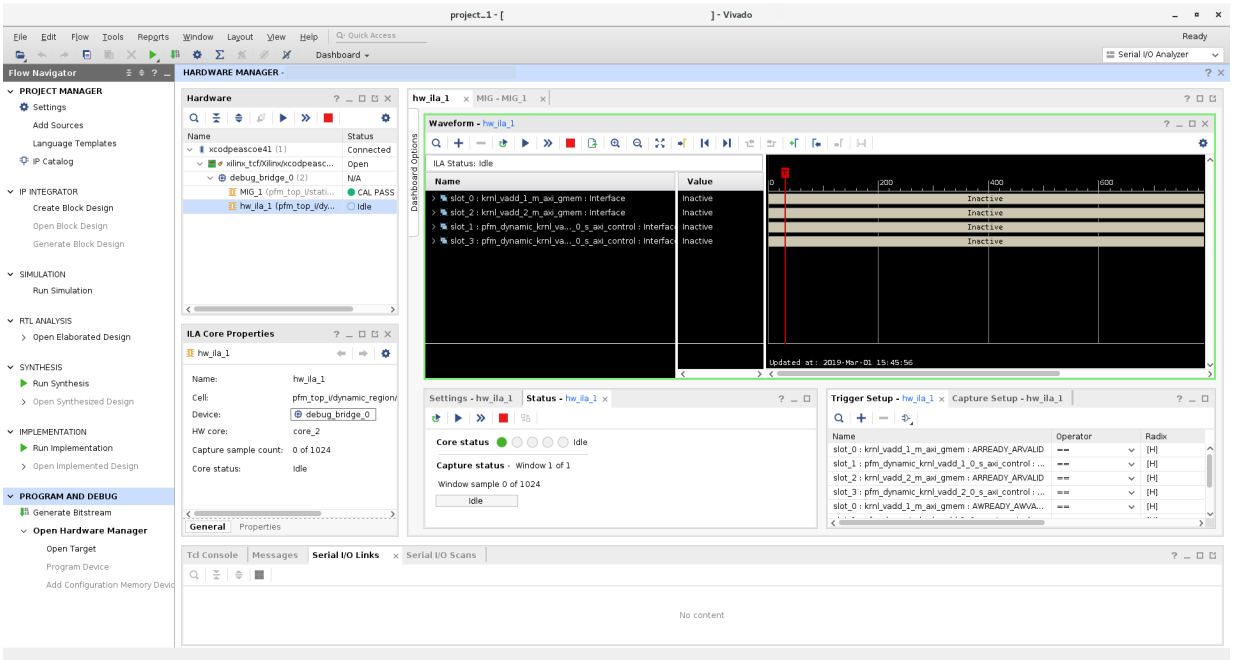
```
debug_hw --vivado --host <host_name> --ltx_file ./_x/link/vivado/vpl/prj/
prj.runs/impl_1/debug_nets.ltx
```

> **TIP:** *The `<host_name>` is the name of your system.*

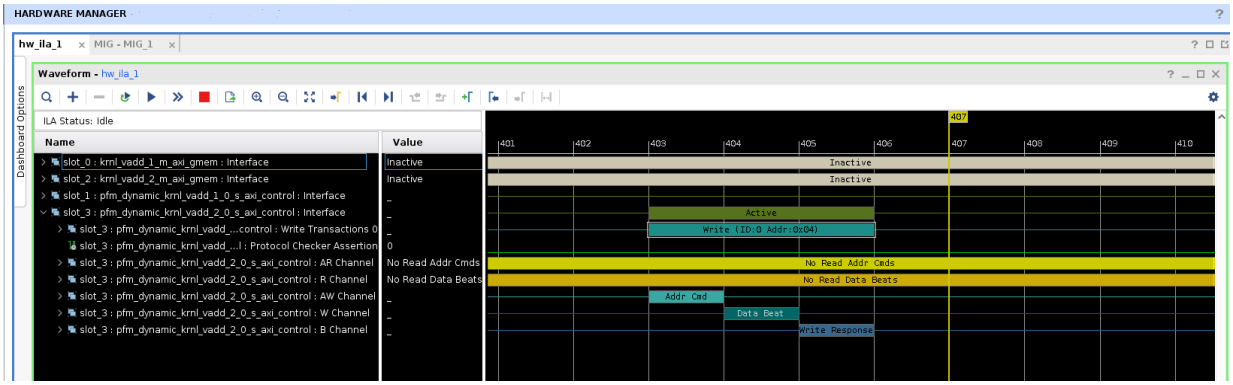
As an example, the command window displays the following results:

```
launching vivado... ['vivado', '-source', 'vitis_hw_debug.tcl', '-
tclargs',
'/tmp/project_1/project_1.xpr', 'workspace/vadd_test/System/
pfm_top_wrapper.ltx',
'host_name', '10200', '3121']

****** Vivado v2019.2 (64-bit)
  **** SW Build 2245749 on Date Time
  **** IP Build 2245576 on Date Time
    ** Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.

start_gui
```

6. In Vivado Design Suite, run the ILA trigger.



7. Press **Enter** to continue running the host program.
8. In the Vivado hardware manager, see the interface transactions on the kernel instance slave control interface in the Waveform view.



Manual Setup for Hardware Debug

TIP: *The following steps can be used when setting up Nimbix and other cloud platforms.*

There are a few steps required to start the debug servers prior to debugging the design in the Vivado hardware manager.

1. Set up your Vitis core development kit as described in Setting Up the Vitis Environment.
2. Launch the `xvc_pcie` server. The file name passed to `xvc_pcie` must match the character driver file installed with the kernel device driver, where `<driver_id>` can be found by examining the `/dev` folder.

```
>xvc_pcie -d /dev/xfpga/xvc_pub.<device_id>
```

TIP: *The `xvc_pcie` server has many useful command line options. You can issue `xvc_pcie -help` to obtain the full list of available options.*

3. Start the `hw_server` on port `3121`, and connect to the XVC server on port `10201` using the following command:

```
>hw_server -e "set auto-open-servers xilinx-xvc:localhost:10201" -e "set
always-open-jtag 1"
```

4. Launch Vivado Design Suite and open the hardware manager:

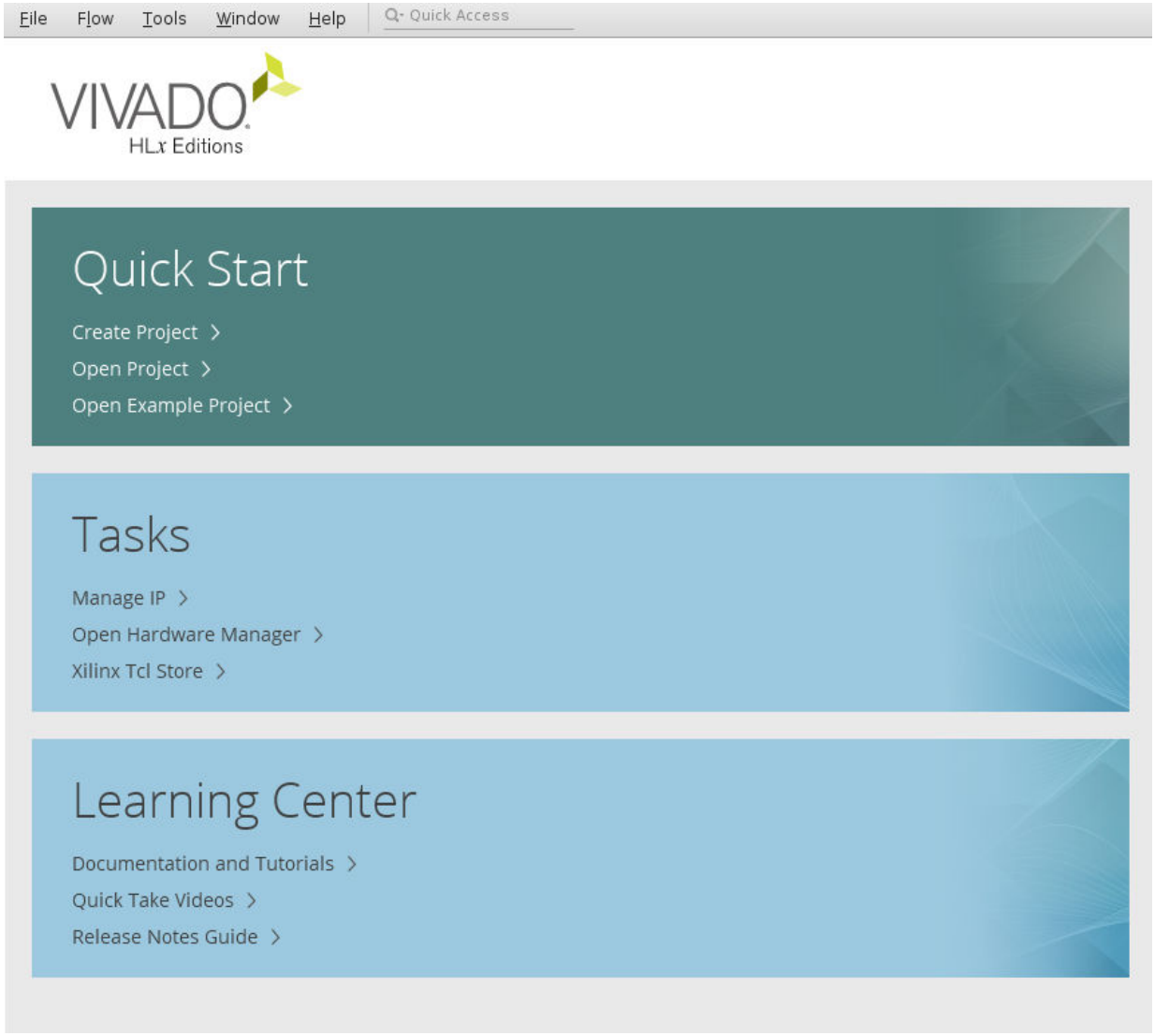
```
vivado
```

Debugging Designs Using Vivado Hardware Manager

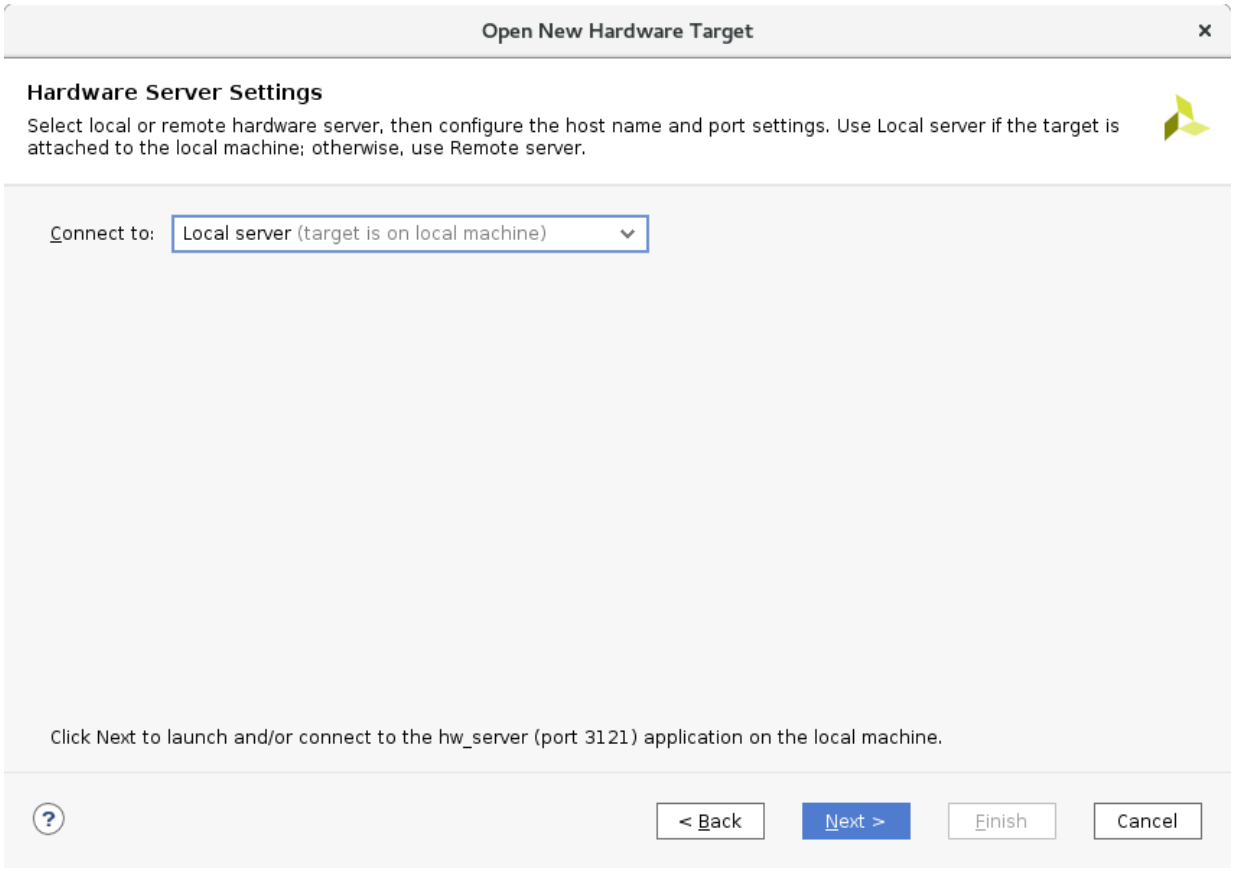
Traditionally, a physical JTAG connection is used to perform hardware debug for AMD devices with the Vivado hardware manager. The Vitis unified software platforms also makes use of the Xilinx virtual cable (XVC) for hardware debugging on remote accelerator cards. To take advantage of this capability, the Vitis debugger uses the XVC server, an implementation of the XVC protocol that allows the Vivado hardware manager to connect to a local or remote target device for debug, using the standard AMD debug cores like the ILA or the VIO IP.

The Vivado hardware manager, from the Vivado Design Suite or Vivado debug feature, can be running on the target instance or it can be running remotely on a different host. The TCP port on which the XVC server is listening must be accessible to the host running Vivado hardware manager. To connect the Vivado hardware manager to XVC server on the target, the following steps should be followed on the machine hosting the Vivado tools:

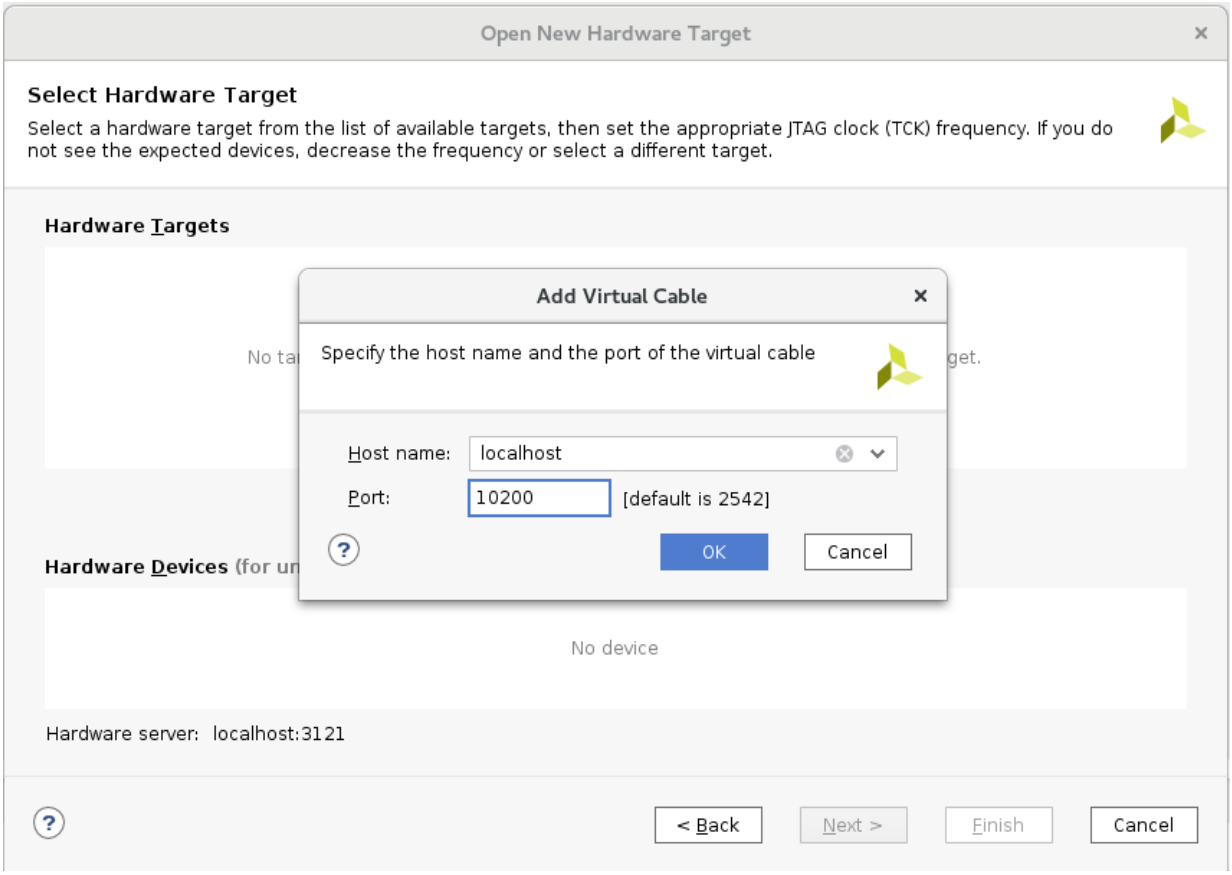
1. Launch the Vivado debug feature, or the full Vivado Design Suite.
2. Select **Open Hardware Manager** from the Tasks menu, as shown in the following figure.



3. Connect to the Vivado tools `hw_server`, specifying a local or remote connection, and the **Host name** and **Port**, as shown below.



4. Connect to the target instance Virtual JTAG XVC server.

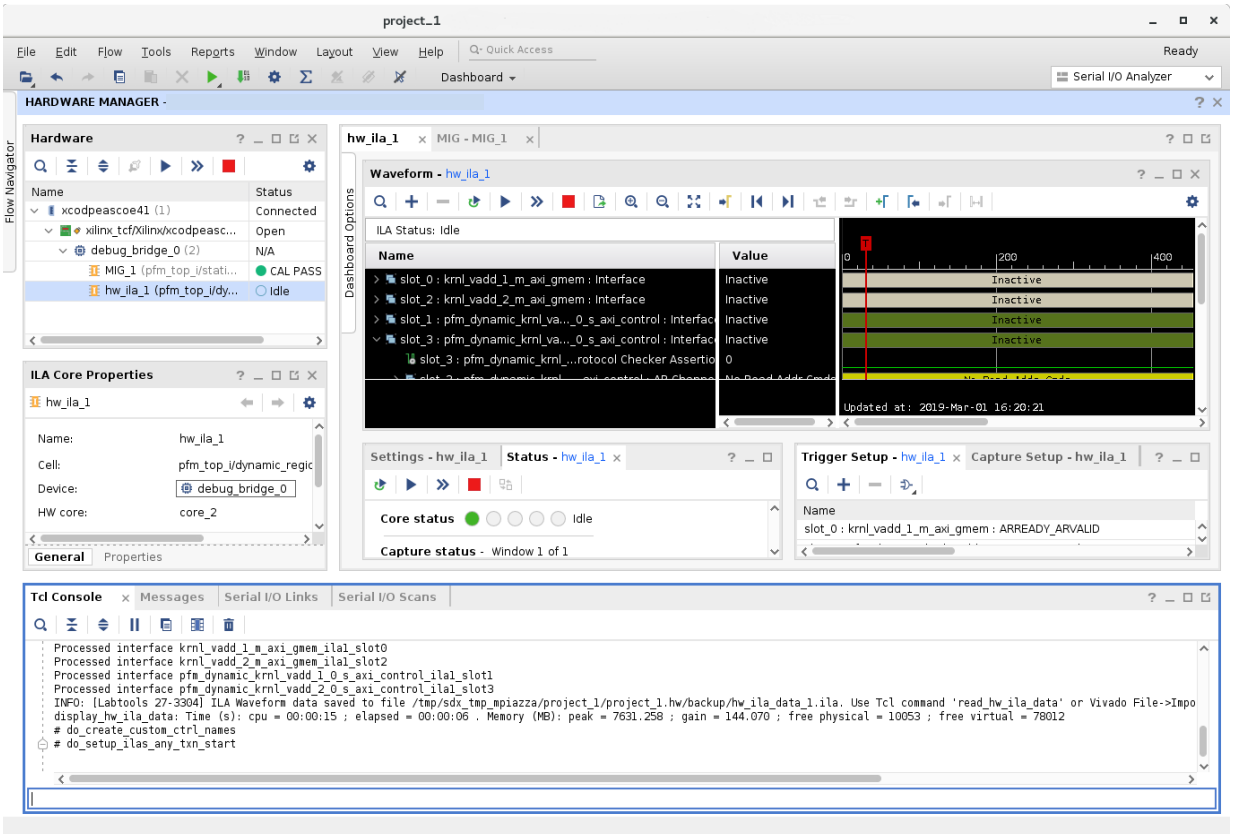


5. Select the `debug_bridge` instance from the Hardware window in the Vivado hardware manager.

   Specify the probes file (`.ltx`) for your design adding it to the **Probes → File** entry in the Hardware Device Properties window. Adding the probes file refreshes the hardware device, and Hardware window should now show the debug cores in your design.

   > **TIP:** *If the kernel has debug cores as specified in Enabling Kernels for Debugging with Chipscope, the probes file (`.ltx`) is written out during the implementation of the kernel by the Vivado tool.*

6. The Vivado hardware manager can now be used to debug the kernels running on the Vitis software platform. Arm the ILA cores in your kernels and run your host application.



TIP: *Refer to the Vivado Design Suite User Guide: Programming and Debugging (UG908) for more information on working with the Vivado hardware manager to debug the design.*

Utilities for Hardware Debugging

In some cases, the normal Vitis IDE and command line debug features are limited in their ability to isolate an issue. This is especially true when the software or hardware appears not to make any progress (hangs). These kinds of system issues are best analyzed with the help of the utilities mentioned in this section.

Using the Linux dmesg Utility

Well-designed kernels and modules report issues through the kernel ring buffer. This is also true for Vitis technology modules that allow you to debug the interaction with the accelerator board on the lowest Linux level.

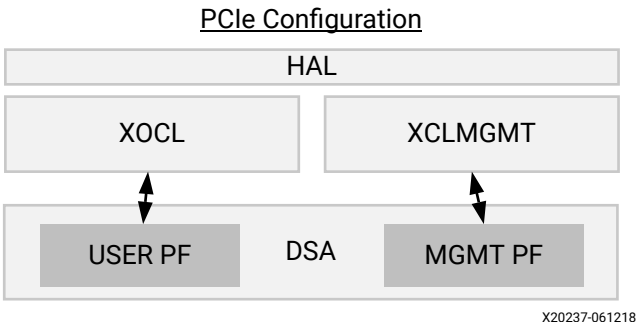
The `dmesg` utility is a Linux tool that lets you read the kernel ring buffer. The kernel ring buffer holds kernel information messages in a circular buffer. A circular buffer of fixed size is used to limit the resource requirements by overwriting the oldest entry with the next incoming message.

TIP: *In most cases, it is sufficient to work with the less verbose `xrt-smi` feature to localize an issue. Refer to Using the xrt-smi Utility for more information on using this tool for debug.*

In the Vitis technology, the `xocl` module and `xclmgmt` driver modules write informational messages to the ring buffer. Thus, for an application hang, crash, or any unexpected behavior (like being unable to program the bitstream, etc.), the `dmesg` tool should be used to check the ring buffer.

The following image shows the layers of the software platform associated with the target platform.

Figure 55: **Software Platform Layers**



To review messages from the Linux tool, you should first clear the ring buffer:

```
sudo dmesg -c
```

This flushes all messages from the ring buffer and makes it easier to spot messages from the `xocl` and `xclmgmt`. After that, start your application and run `dmesg` in another terminal.

```
sudo dmesg
```

The `dmesg` utility prints a record shown in the following example:

Figure 56: **dmesg Utility Example**

```
[ 9902.316729] xclmgmt: AXI Firewall 2 has tripped. Status: 0x80000
[ 9902.316874] xclmgmt: xclmgmt_killall_processes
[ 9902.317007] xclmgmt: Killing pid: 19891
[ 9902.317501] xocl:xdma_xfer_submit: xfer 0xffff8801c1be1018,268435456, s 0x1 timed out, ep 0x10000000.
[ 9902.317911] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0000) = 0x1fc00006 (id).
[ 9902.318410] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0040) = 0x00000001 (status).
[ 9902.318895] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0004) = 0x00f83e1f (control)
[ 9902.319370] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e4080) = 0xa7a30000 (first_desc_lo)
[ 9902.319848] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e4084) = 0x00000000 (first_desc_hi)
[ 9902.320336] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e4088) = 0x0000000f (first_desc_adjacent).
[ 9902.320802] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0048) = 0x00000000 (completed_desc_count).
[ 9902.321279] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0090) = 0x00f83e1e (interrupt_enable_mask)
[ 9902.321759] xocl:engine_status_dump: SG engine 0-H2C0-MM status: 0x00000001: BUSY
[ 9902.322233] xocl:transfer_abort: abort transfer 0xffff8801c1be1018, desc 240, engine desc queued 0.
[ 9902.322752] [drm:xdma_migrate_bo [xocl]] *ERROR* DMA failed to device addr 0x0, tid 19897, channel 0
[ 9902.323232] [drm:xdma_migrate_bo [xocl]] *ERROR* Dumping SG Page Table
```

In the example shown above, the AXI Firewall 2 has tripped, which is better examined using the `xrt-smi` utility.

Using the xrt-smi Utility

The Xilinx board utility (`xrt-smi`) is a powerful standalone command line utility that can be used to debug lower level hardware/software interaction issues. A full description of this utility can be found in xrt-smi Utility in the *Vitis Reference Guide* (UG1702).

With respect to debugging, the following `xrt-smi` options are of special interest:

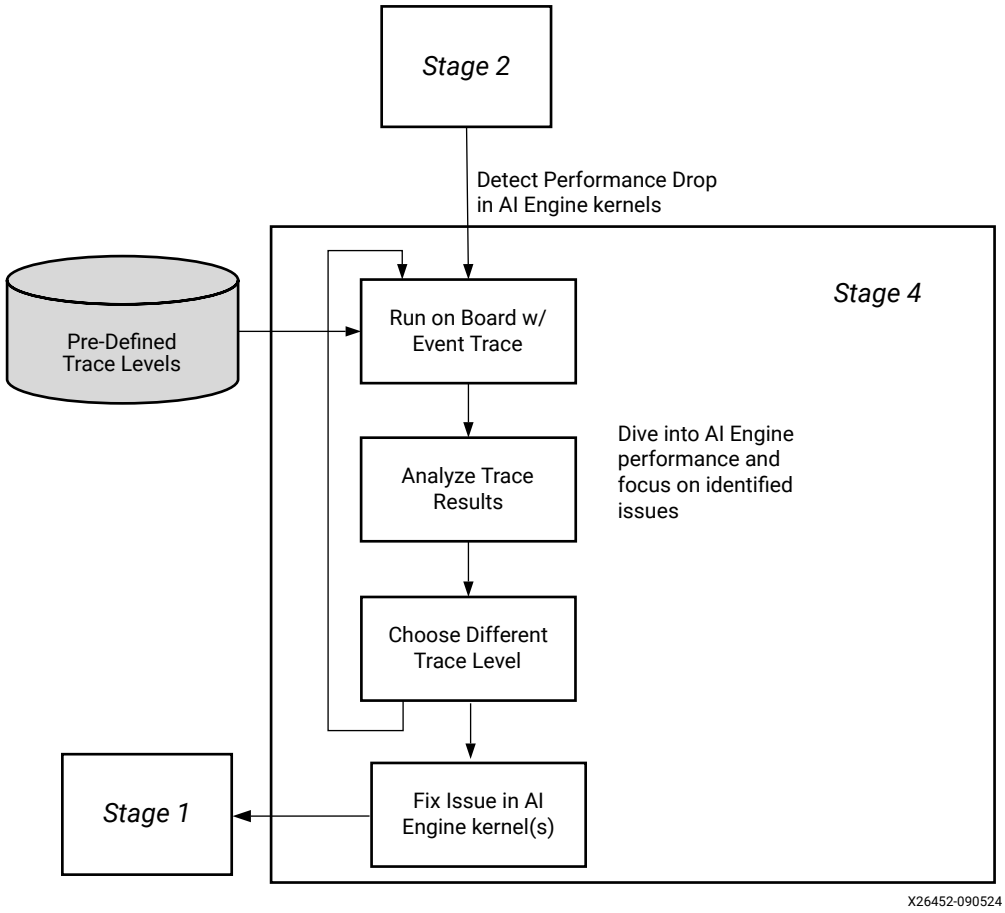
- **`examine`:** Provides an overall status of a card including information on the kernels in card memory.
- **`program`:** Downloads a binary (`xclbin`) to the programmable region of the AMD device.
- **`xrt-smi exmaine -r debug-ip-status -d <BDF>`:** Extracts the status of the Performance Monitors (`aim` and `asm`) and the Lightweight AXI Protocol Checkers (`lapc`).

Stage 4: AI Engine Event Trace and Analysis

The goal of this stage is to determine the AI Engine kernel or graph construct causing design performance drop or stall, or causing a deadlock.

The following figure shows the tasks and techniques available in this stage.

Figure 57: **AI Engine Event Trace and Analysis**



The sections below list the different debug techniques available in this design stage.

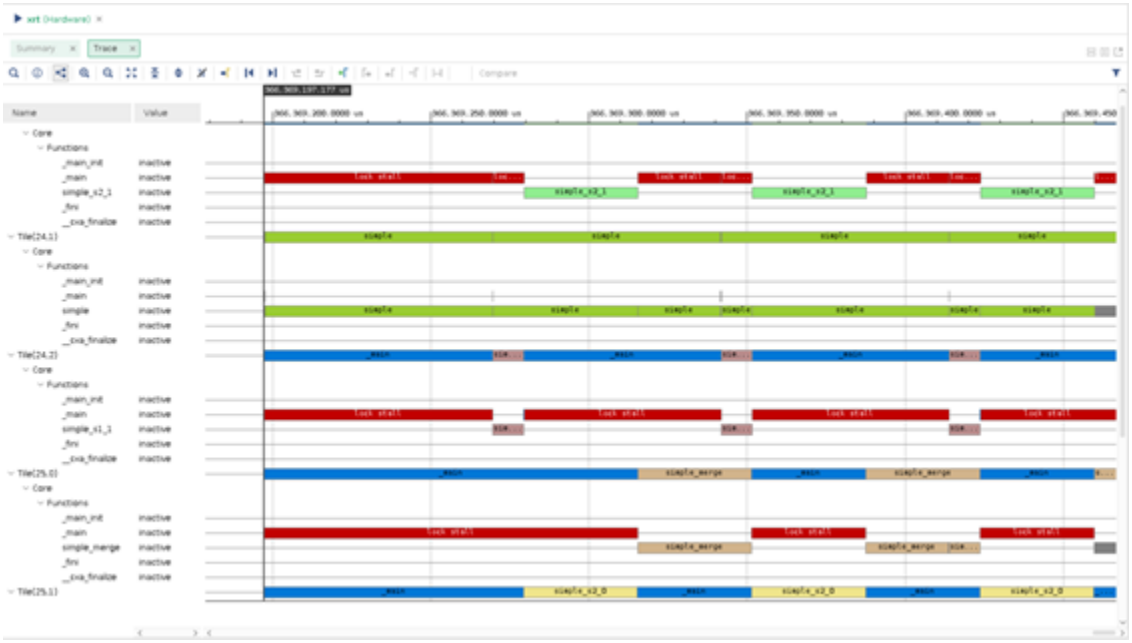
Running and Analyzing Runtime Trace Data Using AI Engine Event Trace Flow

The AI Engine Event Trace feature offers a comprehensive view of design trace data when running the design in hardware, which is a three step process to:

1. Compile the design with event trace enabled and other event trace related options.
2. Run the design in hardware and collect event trace data.
3. Open the trace summary file in the Vitis IDE, which provides a waveform view of the trace data collected above.

Event trace data lets you identify the AI Engine kernel that contributes to a stall, deadlock or throughput drop, and also view events prior to a stall/throughput drop in addition to other detailed trace information. Details on the event trace feature can be found in AI Engine Event Trace in Hardware.

Figure 58: **Event Tracing**



For detailed resolution to specific techniques encountered running event trace in hardware, see Troubleshooting Event Trace in Hardware in the *AI Engine Tools and Flows User Guide* (UG1076). The feature is limited by the event trace counters, streams, DDR memory and design resources available for event trace in the device.

Profiling Intra-Kernel Performance

You can also profile code blocks inside a specific kernel using `aie::tile::cycles()` API.

To get this value in hardware, you can write this value to memory or to an output stream. An example of writing to output stream is shown below. This stream of data can then be examined in the host application to read back the profile data.

```
// get the current tile
aie::tile tile=aie::tile::current();
unsigned long long time=tile.cycles(); //cycle counter of [SS1] the AI
Engine tile
writeincr(out,time);
{//loop to be profiled
}
time=tile.cycles();//cycle counter of the AI Engine tile
writeincr(out,time);
```

This is a very intrusive method of profiling kernel code. AMD recommends that you use this method to simulate the graph with the AI Engine Simulator. In addition, trace and profile data in simulation can also be used for this purpose.

For details on the `aie::tile::cycles()` API, see *AI Engine Kernel and Graph Programming Guide* (UG1079).

Vitis IDE Debugger

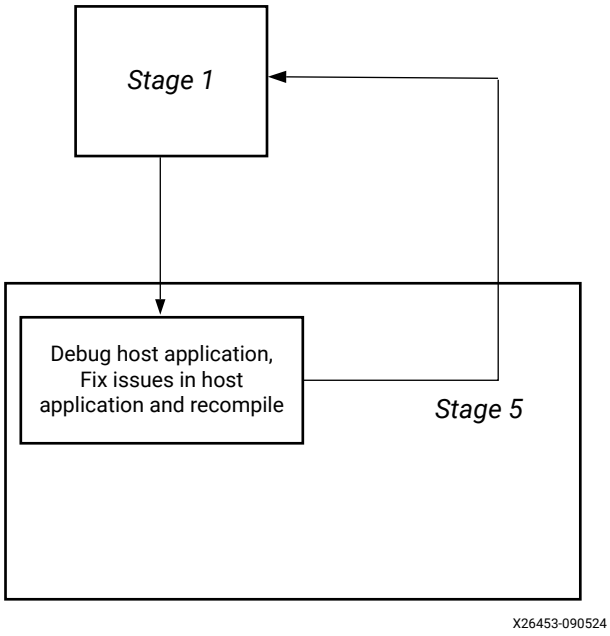
You can also use the Vitis IDE debugger to debug kernel source code. Details on the Vitis Debugger can be found in Debugging the System Project and AI Engine Components in the *Vitis Reference Guide* (UG1702).

Next Stage: After you determine the cause of throughput drop and fix the issue, proceed to stage 1 to rerun the design.

Stage 5: Host Application Debug

The goal of this stage is to debug the host application, and address application exceptions or crashes, if any exist.

Figure 59: **Host Application Debug**



The sections below list the different techniques available in this stage.

Vitis IDE Debugger

You can use the Vitis IDE debugger to debug host application source. Details on the Vitis Debugger can be found in Debugging the System Project and AI Engine Components in the *Vitis Reference Guide* (UG1702).

Printf

You can also use `printf()` statements in your host code to help debug the host application.

Next Stage: After you determine the cause of the failure and fix the issue, you can recompile the host application, and proceed to stage 1.

Techniques for Debugging Application Hangs

This section discusses debugging issues related to the interaction of the host code and the kernels. Problems with these interactions manifest as issues such as machine hangs or application hangs. Although the GDB debug environment might help with isolating the errors in some cases (`xprint`), such as hangs associated with specific kernels, these issues are best debugged using the `dmesg` and `xrt-smi` commands as shown here.

If the process of hardware debugging does not resolve the problem, it is necessary to perform hardware debugging using the ChipScope feature.

AXI Firewall Trips

The AXI firewall should prevent host hangs. This is why the AXI Protocol Firewall IP is included in all production Vitis platforms. When the firewall trips, one of the first checks to perform is confirming if the host code and kernels are set up to use the same memory banks. The following steps detail how to perform this check.

1. Use `xrt-smi` to program the FPGA:

   ```
   xrt-smi program -p <xclbin>
   ```

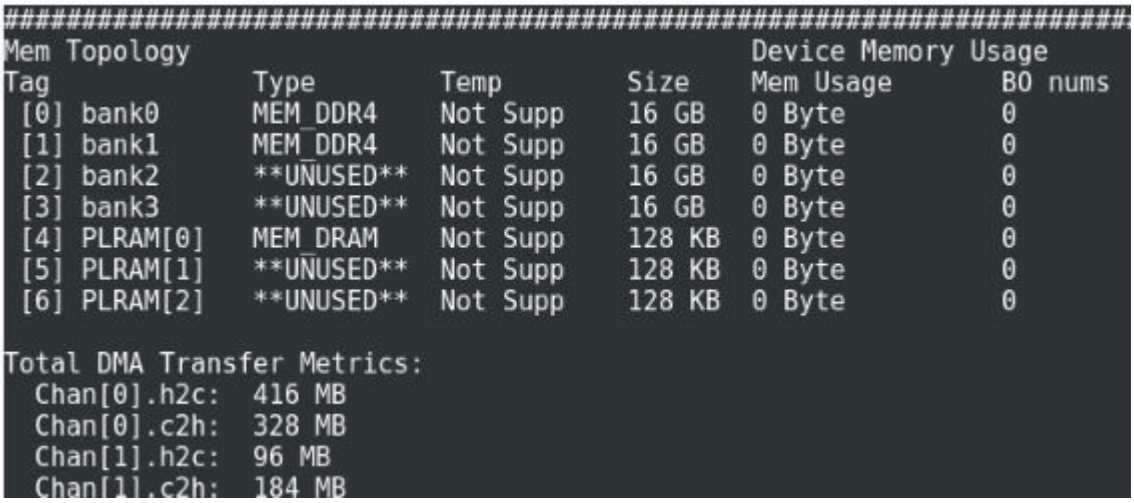
   

   TIP: *Refer to xrt-smi Utility in the Vitis Reference Guide (UG1702) for more information on `xrt-smi`.*

2. Run the `xrt-smi examine` option to check memory topology:

   ```
   xrt-smi examine -r memory -d <bdf>
   ```

   In the following example, there are no kernels associated with memory banks:



3. If the host code expects any DDR banks/PLRAMs to be used, this report should indicate an issue. In this case, it is necessary to check kernel and host code expectations.

Kernel Hangs Due to AXI Violations

It is possible for the kernels to hang due to bad AXI transactions between the kernels and the memory controller. To debug these issues, it is required to instrument the kernels.

1. The Vitis core development kit provides two options for instrumentation to be applied during `v++` linking (`--link`). Both of these options add hardware to your implementation, and based on resource usage it can be necessary to limit instrumentation.

   a. Add Lightweight AXI Protocol Checkers (`lapc`). These protocol checkers are added using the `--debug.protocol` option, as explained in --debug Options. The following syntax is used:

   ```
   --debug.protocol <kernel_instance_name>:<interface_name>
   ```

   In general, the `<interface_name>` is optional. If not specified, all ports on the CU are expected to be analyzed. The `--debug.protocol` option is used to define the protocol checkers to be inserted. This option can accept a special keyword, `all`, for `<kernel_instance_name>` and/or `<interface_name>`.

   Note: Multiple `--debug.xxx` options can be specified in a single command line, or configuration file.

   b. Adding Performance Monitors (`am, aim, asm`) enables the listing of detailed communication statistics (counters). Although this is most useful for performance analysis, it provides insight during debugging on pending port activities. The Performance Monitors are added using the `--profile` option as described in --profile Options. The basic syntax for the `--profile` option is:

   ```
   --profile.data <krnl_name>|all:<kernel_instance_name>|
   all:<intrfc_name>|all:<counters>|all
   ```

Three fields are required to determine the specific interface to attach the performance monitor to. However, if resource consumption is not an issue, the keyword `all` lets you apply the monitoring to all existing kernels, their instances, and interfaces with a single option. Otherwise, you can specify the `kernel_name`, `kernel_instance_name`, and `interface_name` explicitly to limit instrumentation.

The last option, `<counters>|all`, allows you to restrict the information gathering to `counters` for large designs, while `all` (default) includes the collection of actual trace information.

***Note*:** Multiple `--profile` options can be specified in a single command line, or configuration file.

```
[profile]
dataernel1:cu1:m_axi_gmem0
dataernel1:cu1:m_axi_gmem1
dataernel2:cu2:m_axi_gmem
```

2. When the application is rebuilt, rerun the host application using the `xclbin` with the added AIM IP and LAPC IP.
3. When the application hangs, you can use `xrt-smi examine` to check for any errors or anomalies.
4. Check the AIM output:
   - Run the following command a couple of times to check if any counters are moving. If they are moving then the kernels are active.

     ```
     xrt-smi examine -d <bdf> -r debug-ip-status -e aim
     ```

     TIP: *Testing AIM output is also supported through GDB debugging using the command extension `xstatus aim`.*
   - If the counters are stagnant, the outstanding counts greater than zero might mean some AXI transactions are hung.
5. Check the LAPC output:
   - Run the following command to check if there are any AXI violations.

     ```
     xrt-smi examine -d <bdf> -r debug-ip-status -e lapc
     ```

     TIP: *Testing LAPC output is also supported through GDB debugging using the command extension `xstatus lapc`.*
   - If there are any AXI violations, it implies that there are issues in the kernel implementation.

Host Application Hangs When Accessing Memory

Application hangs can also be caused by incomplete DMA transfers initiated from the host code. This does not necessarily mean that the host code is wrong; it might also be that the kernels have issued illegal transactions and locked up the AXI.

1. If the platform has an AXI firewall, such as in the Vitis target platforms, it is likely to trip. The driver issues a `SIGBUS` error, kills the application, and resets the device. You can check this by running the following command:

   ```
   xrt-smi examine -d <bdf> -r firewall
   ```

   The following figure shows such an error in the firewall status:

   ```
   Firewall Last Error Status:
           0:         0x0     (GOOD)
           1:         0x0     (GOOD)
           2:         0x80000 (RECS_WRITE_TO_BVALID_MAX_WAIT).
                      Error occurred on Tue 2017-12-19 11:39:13 PST

   Xclbin ID:    0x5a39da87
   ```

   TIP: *If the firewall has not tripped, the Linux tool, `dmesg`, can provide additional insight.*

2. When you know that the firewall has tripped, it is important to determine the cause of the DMA timeout. The issue could be an illegal DMA transfer, or kernel misbehavior. However, a side effect of the AXI firewall tripping is that the health check functionality in the driver resets the board after killing the application; any information on the device that might help with debugging the root cause is lost. To debug this issue, disable the health check thread in the `xclmgmt` kernel module to capture the error. This uses common Unix kernel tools in the following sequence:

   a. `sudo modinfo xclmgmt`: This command lists the current configuration of the module and indicates if the `health_check` parameter is ON or OFF. It also returns the path to the `xclmgmt` module.

   b. `sudo rmmod xclmgmt`: This removes and disables the `xclmgmt` kernel module.

   c. `sudo insmod <path to module>/xclmgmt.ko health_check=0`: This re-installs the `xclmgmt` kernel module with the health check disabled.

      TIP: *The path to this module is reported in the output of the call to `modinfo`.*

3. With the health check disabled, rerun the application. You can use the kernel instrumentation to isolate this issue as previously described.

Typical Errors Leading to Application Hangs

The user errors that typically create application hangs are listed below:

- Read-before-write in 5.0+ target platforms causes a Memory Interface Generator error correction code (MIG ECC) error. This is typically a user error. For example, this error might occur when a kernel is expected to write 4 KB of data in DDR, but it produces only 1 KB of data, and then try to transfer the full 4 KB of data to the host. It can also happen if you supply a 1 KB buffer to a kernel, but the kernel tries to read 4 KB of data.

- An ECC read-before-write error also occurs if no data has been written to a memory location as the last bitstream download which results in MIG initialization, but a read request is made for that same memory location. ECC errors stall the affected MIG because kernels are usually not able to handle this error. This can manifest in two different ways:

  1. The kernel instance might hang or stall because it cannot handle this error while reading or writing to or from the affected MIG. The `xrt-smi` query shows that the kernel instance is stuck in a `BUSY` state and is not making progress.
  2. The AXI Firewall might trip if a PCIe® DMA request is made to the affected MIG, because the DMA engine is unable to complete the request. AXI Firewall trips result in the Linux kernel driver killing all processes which have opened the device node with the `SIGBUS` signal. The `xrt-smi` query shows if an AXI Firewall has indeed tripped and includes a timestamp.

  If the above hang does not occur, the host code might not read back the correct data. This incorrect data is typically 0s and is located in the last part of the data. It is important to review the host code carefully. One common example is compression, where the size of the compressed data is not known up front, and an application might try to migrate more data to the host than was produced by the kernel.

Defensive Programming

The Vitis compiler is capable of creating very efficient implementations. In some cases, however, implementation issues can occur. One such case is if a write request is emitted before there is enough data available in the process to complete the write transaction. This can cause deadlock conditions when multiple concurrent kernels are affected by this issue and the write request of a kernel depends on the input read being completed.

To avoid these situations, a conservative mode is available on the adapter. In principle, it delays the write request until it has all of the data necessary to complete the write. This mode is enabled during compilation by applying the following `--advanced.param` option to the `v++` compiler:

```
--advanced.param:compiler.axiDeadLockFree=yes
```

Because enabling this mode can impact performance, you might prefer to use this as a defensive programming technique where this option is inserted during development and testing and then removed during optimization. You might also want to add this option when the accelerator hangs repeatedly.

Hardware Debug for Embedded Processors

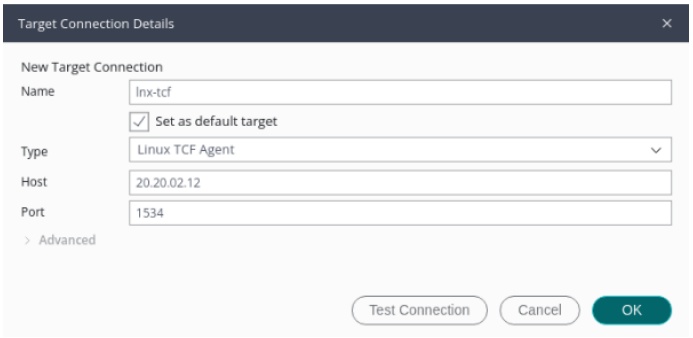
For hardware builds the setup involves the following steps:

1. Copy the contents of the `<project>/Hardware/sd_card/sd_card` folder to a physical SD card. This creates a bootable medium for your target platform.
2. Insert the SD card into the card reader of your embedded processor platform.

3. Change the boot-mode settings of the platform to SD boot mode, and power up the board.
4. After the device is booted, enter the `mount` command at the command prompt to get a list of mount points. SD cards are typically mounted as `/run/media/mmcblk*`
5. Execute the following commands, for example:

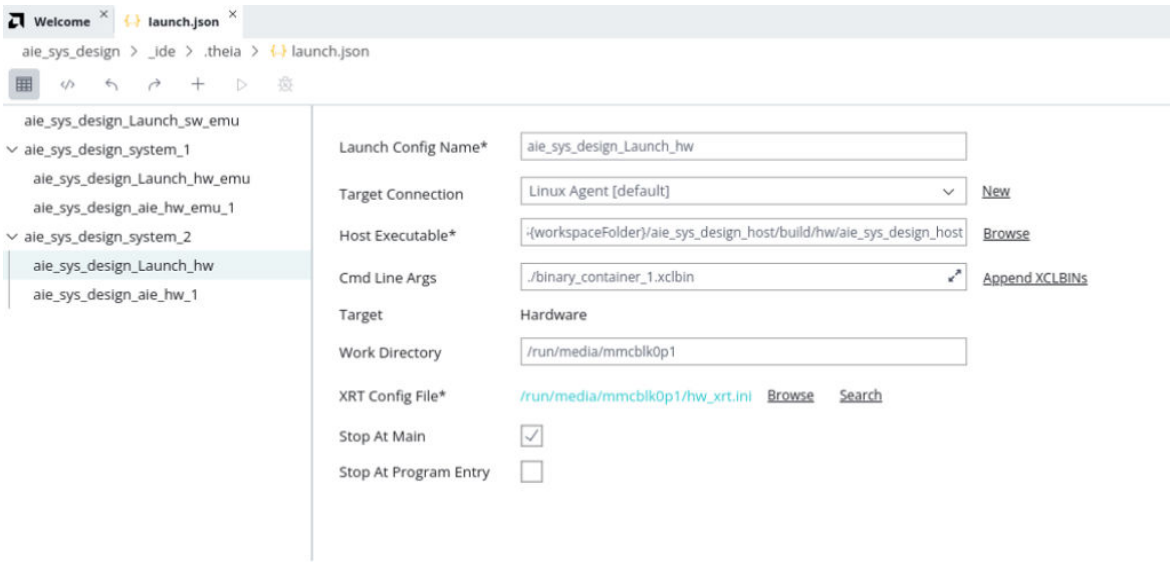
   ```
   cd /run/media/mmcblk0p1
   source init.sh
   cat /etc/xocl.txt
   ```

   The `cat` command will display the platform name `xilinx_vck190_base_202420_1` to let you confirm it is the same as your specified platform and that your setup is correct.
6. Run `ifconfig` to get the IP address of the target card. You will use the IP address to set up a TCF agent connection in the Vitis unified IDE to connect to the assigned IP address of the embedded processor platform.
7. Create a target connection to the remote accelerator card. Use the **Vitis → Target Connections** menu command to open the Target Connections dialog box.
8. Right-click the **Linux TCF Agent** and select the **New Target** command to open the New Target Connection dialog box.
9. 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 in an earlier step.



10. Click **OK** to close and continue.

11. In the Flow Navigator view, click the Open Settings command to open the Launch Configuration editor to create a new launch configuration for the hardware design



    Set the following fields on the **Main** tab of the dialog box:

    - **Name:** Specifies a name for your Hardware debug configuration.
    - **Target Connection:** Select a Linux TCF agent as previously configured.
    - **Host Executable:** Specify the location of the software application to drive the hardware.
    - **Cmd Line Args:** Specify any needed command line arguments for the host application, such as the `.xclbin` file to load.
    - **Work Directory:** Specifies the location where the system is run and output files will be written.
    - **XRT Config File:** Specifies the `xrt.ini` file to add to the hardware run as described in Enabling Profiling in Your Application.
    - **Stop at Main:** Puts a breakpoint in the host application to stop at the entry to the `main()` function to enable debug operations.
    - **Stop At Program Entry:** Places a breakpoint at the entry to the hardware program to enable debug operations.

12. Select **Debug** from the Flow Navigator to open the Debug view as described in Debug View in the *Vitis Reference Guide* (UG1702).

    This opens the Debug view in the Vitis unified IDE, and connects to the PS application on your hardware platform. The application automatically breaks at the `main()` function to let you set up and configure the debug environment.

Chapter 12

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. *Vitis Model Composer User Guide* (UG1483)
2. *Data Center Acceleration using Vitis* (UG1700)
3. *Vitis Software Platform Release Notes* (UG1742)
4. *Vitis Reference Guide* (UG1702)
5. *Introduction to FPGA Design with Vivado High-Level Synthesis* (UG998)
6. *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400)
7. *Vitis High-Level Synthesis User Guide* (UG1399)
8. *Versal Premium Series Data Sheet: DC and AC Switching Characteristics* (DS959)
9. *AI Engine Tools and Flows User Guide* (UG1076)
10. *PetaLinux Tools Documentation: Reference Guide* (UG1144)
11. *Versal Adaptive SoC System Software Developers Guide* (UG1304)
12. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
13. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
14. *Versal Adaptive SoC Design Guide* (UG1273)
15. *UltraFast Design Methodology Guide for FPGAs and SoCs* (UG949)
16. *Versal Adaptive SoC Clocking Resources Architecture Manual* (AM003)
17. *Versal Prime Series Data Sheet: DC and AC Switching Characteristics* (DS956)
18. *Versal AI Core Series Data Sheet: DC and AC Switching Characteristics* (DS957)
19. *Versal Adaptive SoC Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313)
20. *Clocking Wizard for Versal Adaptive SoC LogiCORE IP Product Guide* (PG321)
21. *Control, Interface and Processing System LogiCORE IP Product Guide* (PG352)
22. *AI Engine Kernel and Graph Programming Guide* (UG1079)
23. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)

24. *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137)
25. *Zynq 7000 SoC: Embedded Design Tutorial* (UG1165)
26. *Zynq UltraScale+ MPSoC: Embedded Design Tutorial* (UG1209)
27. *Bootgen User Guide* (UG1283)
28. *Embedded Design Tutorials: Versal Adaptive Compute Acceleration Platform* (UG1305)
29. *Versal Adaptive SoC Hardware, IP, and Platform Development Methodology Guide* (UG1387)
30. *AI Engine-ML Kernel and Graph Programming Guide* (UG1603)
31. *Vivado Design Suite User Guide: Design Flows Overview* (UG892)
32. *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894)
33. *Vivado Design Suite User Guide: Implementation* (UG904)
34. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
35. *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909)
36. *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947)
37. *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994)
38. *Versal AI Edge Series Data Sheet: DC and AC Switching Characteristics* (DS958)
39. *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
11/20/2025 Version 2025.2	
General Updates	Topics now includes an “Important” callout explicitly stating that if BDCs are used in extensible hardware platforms, you must follow Vivado’s BDC rules, with a direct link to the relevant guidance in *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994).
Chapter 8: Building and Running the System	Added reference to the newly detailed Validate NoC section in *Versal Adaptive SoC Hardware, IP, and Platform Development Methodology Guide* (UG1387).
VFS Objects	Added a section on how to convert varray back to np.array. Clarified code examples.
NoC Memory Configuration	Updated to replace invalid concatenated GMIO sp_tag examples with correct per-port syntax.
Packaging the System in HW Emulation	For hw_emu on EDF-enabled platforms (e.g., vrk160), run `v++ -p -t hw_emu` with `--advanced.param package.enableEdfHwEmu=1` because EDF changes packaging via its build/QEMU artifacts.

Section	Revision Summary
Building the System in HW Emulation	Clarified the differences in setup and run scenarios between `hw` and `hw_emu` flows.
General Updates	Clarified meaning of XSA acronyms.
Versal Hardware Platform using a CED Design	Clarified statement on CED usage.
Adding Hardware Interfaces	Added clarification on SP Tag ID.
07/16/2025 Version 2025.1	
General Updates	Editorial and link revisions.
Platform Support for Segmented Configuration Flow	Added new topic.
Special Considerations for Embedded Platform Creation	Hidden topic as contents need to be reworked for 2025.2.
05/29/2025 Version 2025.1	
Adding Hardware Interfaces	Updated AXI4-Stream Interfaces platform properties and added example.
AI Engine Event Trace in Hardware	Revised AI Engine events to add AI Engine metrics and added descriptions regarding AI Engine-ML and AI Engine-ML Gen2 events.
AI Engine Partitions and Runtime Control and Reload	New topic on reset and reload of AI Engine partitions when running design on HW.
Chapter 8: Building and Running the System	Added steps for choosing pre-built base platform or custom platform.
Chapter 6: Simulation and Verification in Vitis	New topic covering AIE graph and HLS kernel functional verification in Python or the MATLAB® environment.
Hardware Emulation and Hardware Run Profiling	Added new metrics `start_to_bytes_transferred` and `interface_tile_latency`.
Incremental Design Management	Added scenario for modifying Vitis Subsystem components.
Integrating the System	Updated keywords.
Interpreting Data in the Waveform Views	Removed deprecated reference to OpenCL.
Linking a VSS Component	Added description and more examples for hierarchical Vitis Subsystem.
Mapping Kernel Ports to Memory	Updated link to tutorial.
NoC Memory Configuration	New topic describing how Vitis can access NoC Memory configurations for DDR and LPDDR memories in the platform, and considerations for GMIO access to these memories.
Packaging the RTL Code as a Vitis XO	Added note clarifying that XRT controlled RTL IP requires packaged XO for kernel information.
Pre-built Base Platforms	Updated platform versions to 2025.1 and added VEK385 example.
SSI Technology Devices and Hardware Platforms	New topic on how to set up a Vivado platform that supports an SSI technology based device.
Running Traffic Generators in Python/C++	Removed deprecated reference to `sw_emu`. Clarified example for Python API.
Simulator Support in Hardware Emulation	Updated tool versions in examples.
Special Considerations for Embedded Platform Creation	New topic on recommendations for placing logic in platform versus kernel.
Specifying SLR Region for SSI Devices	New topic on specifying a SLR region through a property for SSI devices.
Using Traffic Generators for AI Engine Designs	Removed deprecated reference to `sw_emu`.

Section	Revision Summary
Using Traffic Generators in AI Engine Graphs	Removed deprecated reference to `sw_emu`.
VARRAY Supported Data Types	New topic on data type object used by tools such as Vitis functional simulation.
VFS Objects	New topic on Vitis functional simulation objects.
Vitis Functional Simulation Overview	New topic with content for Vitis functional simulation.
Vitis Key Concepts	Added Vitis subsystem and Vitis functional simulation terminology.
Vitis Subsystem Flow	New topic describing the Vitis subsystem.

Please Read: Important Legal Notices

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