



AMD ADAPTIVE COMPUTING CUSTOMER TRAINING

Getting Started with the Versal
Adaptive SoC Platform
Lab Workbook

ver-getting-started-2025.1-wkb-lab-rev1

Getting Started with the Versal Adaptive SoC Platform Lab Workbook 2025.1



together we advance_

DISCLAIMER AND ATTRIBUTIONS

The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18

© 2025 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, UltraScale+, Versal, Vitis, Vivado, Zynq, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere. Linux is the registered trademark of Linus Torvalds in the U.S. and other countries. Ubuntu and the Ubuntu logo are registered trademarks of Canonical Ltd. Other product names used in this publication are for identification purposes only and may be trademarks of their respective owners. Certain AMD technologies may require third-party enablement or activation. Supported features may vary by operating system. Please confirm with the system manufacturer for specific features. No technology or product can be completely secure.

Lab FAQ

- Where can I get the files for the labs?
 - <https://www.amd.com/en/training/customer/adaptive-computing/downloads.html>
 - These are original files and do not contain any work that you may have performed.
 - Labs were developed using version 2025.1 of the tools. Later versions may work and will likely require you to update various pieces of IP. See the "Updating IP" topic in the *Lab Reference Guide* for instructions on how to update IP (Vivado Design Suite Operations > Vivado IP Integrator Operations > Updating IP).
 - These labs were validated using a virtual machine that hosts the Ubuntu® Linux® OS. Many labs can be performed using the Windows OS; however, not all AMD tools are supported under Windows (such as QEMU and PetaLinux).
 - Please use the latest version of the lab and ensure that the lab instruction version matches the lab file set version.
- What if I cannot answer a question in the lab?
 - Do your best! The questions are meant to stimulate thought not to test your knowledge. After you have considered a question for a bit, you can find the answer at the end of each lab.
- Where can I get more detailed information on a topic?
 - The *Lab Reference Guide* is a collection of "how to" topics for commonly performed tasks categorized by the tool (Vivado™ Design Suite, Vitis™ Unified IDE, etc.) and subdivided into major areas within the tool.
 - The *Lab Reference Guide* is available from the lab files download page: <https://www.amd.com/en/training/customer/adaptive-computing/downloads.html>.
- How do the instructions work?
 - The instructions are provided in three layers:
 - Steps: These are the major/broadest aspects of solving the problem that the lab poses (X).
 - Instructions: These represent the significant instructions towards solving the issue outlined by the step (X-X).
 - Tasks: These are the finest granularity items that (when combined with the other tasks) solve the instruction to which they are subordinate (X-X-X).

Creating a New Project

Step 1

A step covers the process of achieving a major milestone towards completing the lab

Each instruction describes a major component of the step


Each task represents some action; many tasks refer to a figure for additional clarity

Launch the Vivado Design Suite and create a new project using the New Project Wizard.

The New Project Wizard creates a project file (.xpr). This project file organizes the design files and tracks the status of each phase in the development process from design entry through device programming and debug.

Here are two ways to open the Vivado Design Suite.

1-1. Open the Vivado Design Suite.

1-1-1. Click the Vivado icon () from the taskbar.



The step explanation describes what will be accomplished in this step

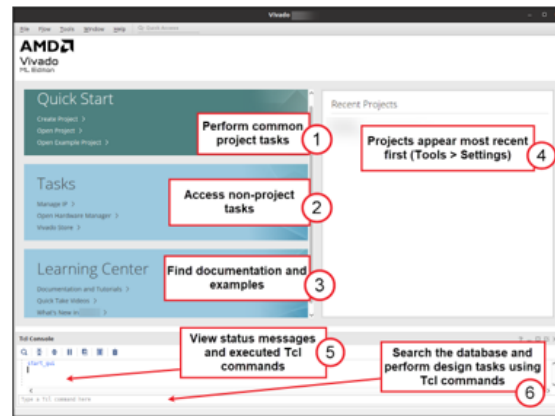
Figure 1-2: Launching the Vivado Design Suite from the Taskbar

Note: It takes a few moments to launch. The order of the icons in your environment may be different.

Alternatively, open the Linux terminal window (<Ctrl + Alt + T>) and enter the following:
`source /opt/amd/Vivado/ /settings64.sh; vivado`

The tool opens with a Welcome window. From here you can create a new project, open an existing project, enter Tcl commands, and access documentation and examples.

1-1-2. [Optional] Maximize the window as there is a lot of information to see.



Additional explanations may be present for instructions and tasks

Figure 1-3: Vivado Design Suite Welcome Screen

Table of Contents

Lab 1: NoC Introduction and Concepts	3
Lab 2: System Simulation.....	33

Lab 1: NoC Introduction and Concepts

2025.1

Abstract

The network on chip (NoC) forms the communication backbone of AMD Versal™ devices. Knowing how to access this valuable resource enables you to quickly and efficiently move data among the major blocks as well as to and from DDR memory. In this lab, you will access the NoC using the AMD Vivado™ Design Suite IP integrator tool.

This lab should take approximately 45 minutes.

CloudShare Users Only

You are provided with three attempts to finish a lab, where the time allotted to complete each lab is twice the expected completion time. Once the timer starts, you cannot pause the timer. Each lab attempt resets the previous attempt—your work from previous attempts is not saved.

Objectives

After completing this lab, you will be able to:

- Instantiate the correct form of the NoC IP
- Configure the ports of the NoC IP
- View the output of the NoC compiler

Introduction

The programmable network on chip (NoC) is the communication backbone for the Versal devices. This dedicated resource provides high-performance data movement among the principal blocks within the Versal devices.

One of the ways to configure NoC is using the Vivado™ Design Suite IP integrator tool. Each piece of NoC IP represents a logical portion of the overall physical NoC structure.

Designers instantiate and configure these pieces of NoC IP to connect a master (transaction initiator) to a slave (transaction reactor).

Data is moved through the NoC structure in the form of packets that share the physical routing resources (pathways). There are fewer routing resources than potential connections that can be made, which necessitates this sharing.

Each pathway is assigned a bandwidth target and a traffic class that determines the quality of service, or more simply, the priority of one packet over another.

Each NoC packet switch (NPS) arbitrates the packets that pass through it according to a packet's priority. Higher-priority traffic passes while lower-priority packets are buffered. Note that this lab only addresses the process of configuring these pathways and not the underlying rationale, strategy, or mechanisms.

Here you will create a basic system that connects the CIPS IP (which contains the processing system) to a DDR memory controller via the NoC. This is a very common configuration for moving instructions and data from the off-device DDR memory to the processor cores.

The NoC will handle an additional load generated by a pair of AXI traffic generators, which is representative of any data-generating IP. The NoC will route the output of these generators through the NoC to block RAM, which is simply a data consumer.

The goal of this lab is to:

- Illustrate the process of instantiating and configuring logical NoCs
- Demonstrate how to connect the CIPS block to DDR memory as this process represents a significant change from the way that the Zynq™ UltraScale+™ MPSoC PS connects to DDR memory

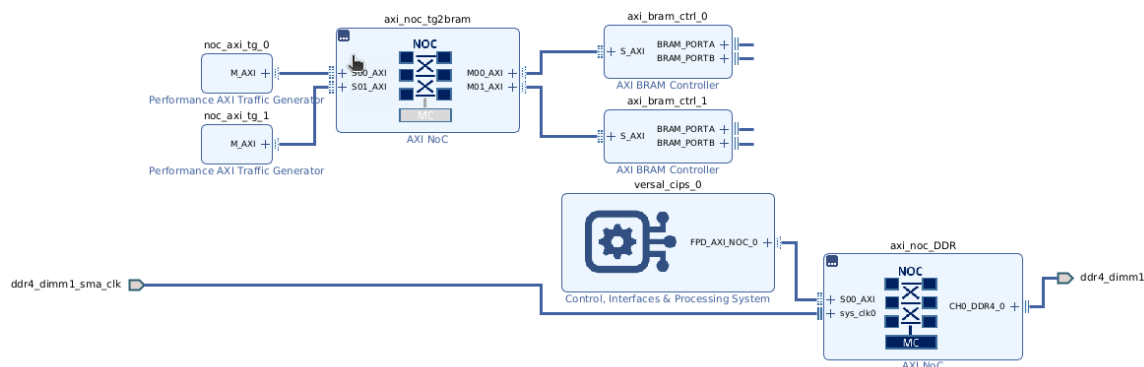


Figure 1-1: Block Design in Addressing View

Here is a simplified list of the connections:

Master	Slave	Address Base	Address Range
CIPS	DDR	0x0	2G
AXI_TG0	BRAM	0x201_0000_0000	8K
AXI_TG1	BRAM	0x201_4000_0000	8K

Understanding the Lab Environment

The labs and demos provided in this course are designed to run on a Linux® platform.

One environment variable is required: `TRAINING_PATH`, which points to the location of the lab files. This variable comes configured in the CloudShare/CustEd_VM environments.

Some tools can use this environment variable directly (that is, `$TRAINING_PATH` is recognized and automatically expanded), and some tools require manual expansion (`/home/amd/training` for the CloudShare/CustEd_VM environments). The lab instructions describe what to do for each tool. Other environments require the definition of this variable for the scripts to work properly.

The Vivado™ Design Suite and the Vitis™ Unified IDE offer a Tcl environment used in many labs. When either tool is launched, it starts with a clean Tcl environment with none of the procs or variables remaining from any previous launch of the tools.

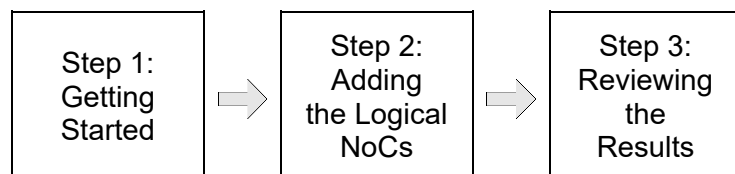
If you sourced a Tcl script or manually set any Tcl variables and you closed the tool, when you reopen the tool, you will need to re-source the Tcl script and set any variables that the lab requires. This is also true of terminal windows—any variable settings will be cleared when a new terminal opens.

Nomenclature

Formal nomenclature is used to explain how different arguments are used. The following are some of the more commonly used symbols:

Symbol	Description	Example	Explanation
<text>	Indicates a field	cd <dir>	<dir> represents the name of the directory. The < and > symbols are NOT entered. If the directory to change to is XYZ, then you would enter cd XYZ into the environment.
[text]	Indicates an optional argument	ls [more]	This could be interpreted as ls <Enter> or ls more <Enter> . The first instance lists the files in the current Linux directory, and the second lists the files in the current Linux directory, but additionally runs the output through the more tool, which paginates the output. Here, the pipe symbol () is a Linux operator.
	Indicates choices	cmd <ZCU104 VCK190>	The cmd command takes a single argument, which could be ZCU104 OR VCK190 . You would enter either cmd ZCU104 or cmd VCK190 .

General Flow



Getting Started

Step 1

You will begin the lab by creating a new Vivado Design Suite project. You will then create a block design and add supporting IPs by using a Tcl script.

Here are two ways to open the Vivado Design Suite.

1-1. Open the Vivado Design Suite.

1-1-1. Click the **Vivado** icon () from the taskbar.

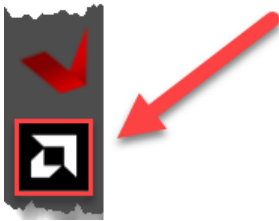


Figure 1-2: Launching the Vivado Design Suite from the Taskbar

Note: It takes a few moments to launch. The order of the icons in your environment may be different.

Alternatively, open the Linux terminal window (<Ctrl + Alt + T>) and enter the following:

```
source /opt/amd/2025.1/Vivado/settings64.sh; vivado
```

Note: This installation path is valid for the CustEd VM and CloudShare environments. Use the proper path for your environment.

The tool opens with a Welcome window. From here you can create a new project, open an existing project, enter Tcl commands, and access documentation and examples.

1-1-2. [Optional] Maximize the window as there is a lot of information to see.

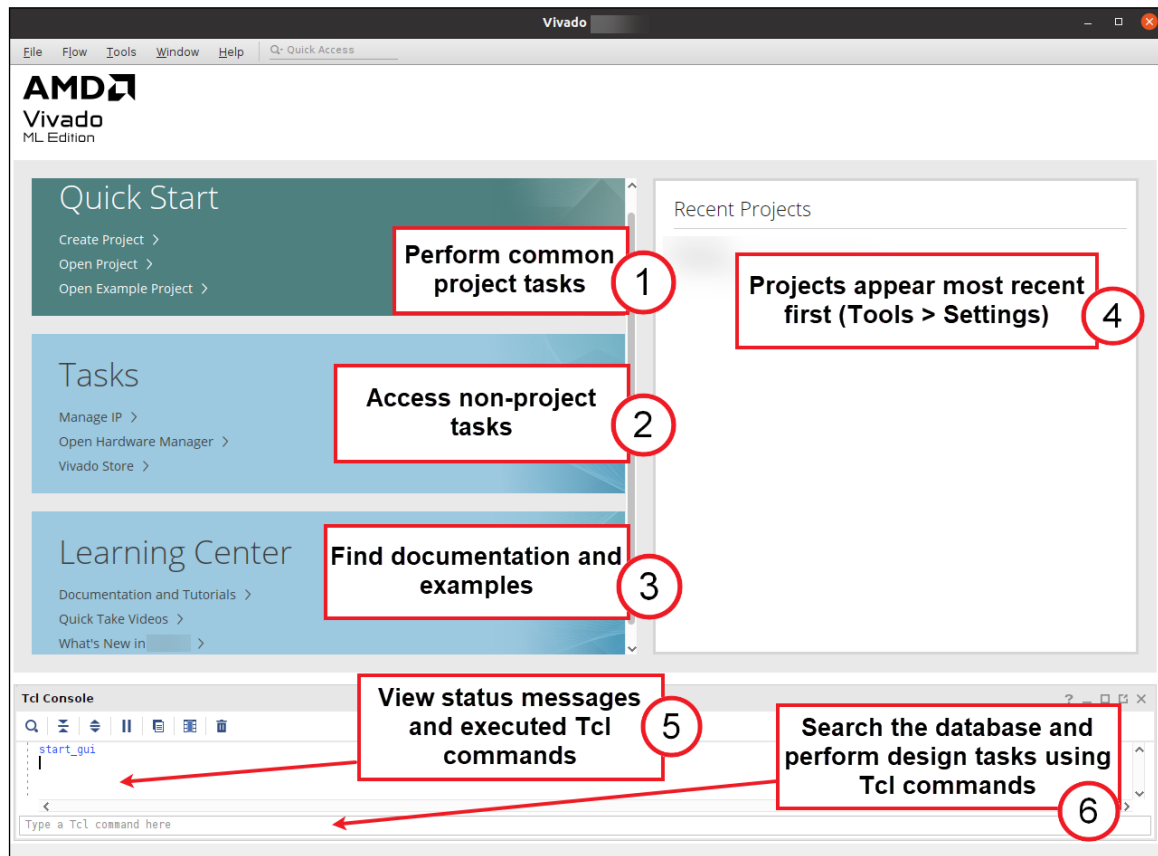


Figure 1-3: Vivado Design Suite Welcome Screen

Hint: If the Tcl Console is not visible, double-click the **Tcl Console** tab to make it visible.

"Create Project" is the starting point for all designs. Projects contain sources, settings, graphics, IP, and other elements that are used to build a final bitstream and analyze a design. The Create New Project Wizard in the Vivado Design Suite allows you to specify HDL and other project resource files that will be included in the project.

1-2. Create a new, blank Vivado Design Suite project.

1-2-1. Click **Create Project** to begin the process (1).

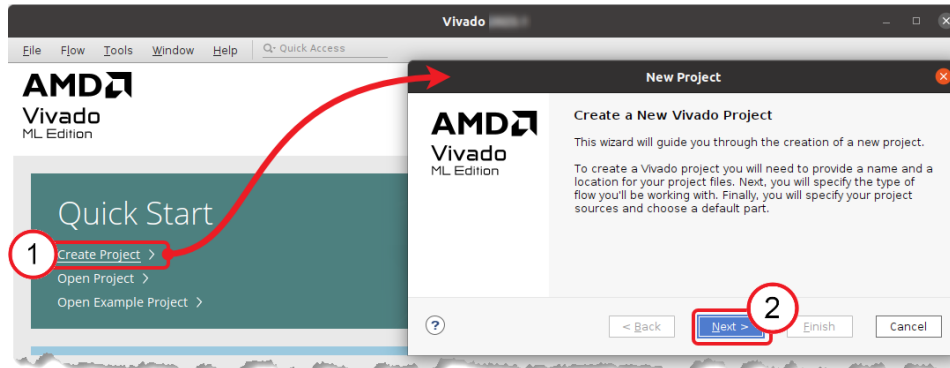


Figure 1-4: Creating a New Vivado Design Suite Project

This will launch the New Project Wizard.

1-2-2. Click **Next** to exit the introductory dialog box and begin entering project-specific information (2).

1-3. Describe the various aspects of the project.

1-3-1. Enter **NoCIntro** in the Project name field (1).

1-3-2. Enter the following location in the Project location field (2):

`$TRAINING_PATH/NoCIntro/lab`

Important: The Vivado Design Suite is capable of expanding variables when running under both the Linux and Windows environments. The available environment variables (regardless of the OS) must be predicated with the '\$' symbol.

Alternatively, you can use the browse feature to navigate to where you want the project to reside.

1-3-3. Uncheck the **Create Project Subdirectory** option if it is selected (3).

Leaving this checked will create an unnecessary level of hierarchy for the lab.

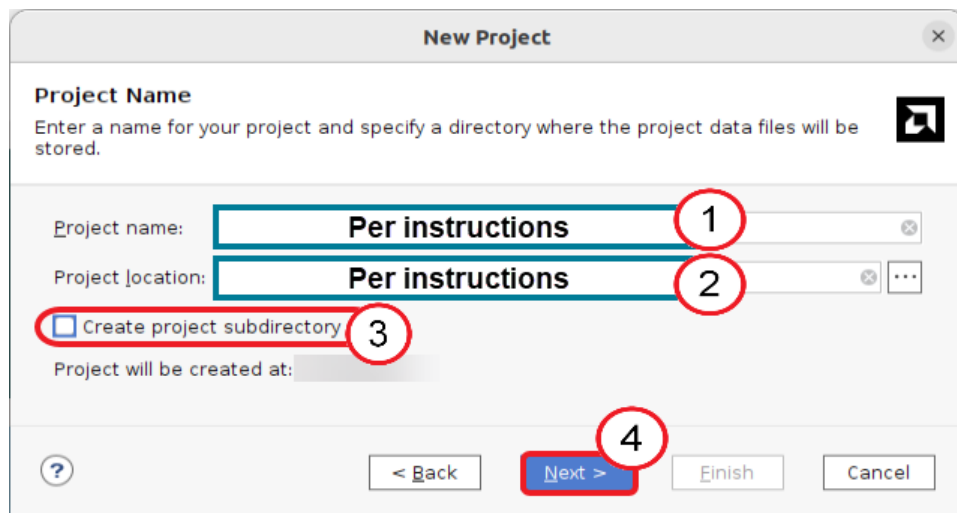


Figure 1-5: Entering the Project Name and Location

1-3-4. Click **Next** to advance to the next dialog box (4).

Here you will specify your project type as either an RTL project or a post-synthesis project. An RTL project enables you to add or create new HDL files and synthesize them, whereas the post-synthesis project requires pre-synthesized files. When an empty design is created, an RTL project is used.

1-3-5. Select **RTL Project** (1).

1-3-6. Select **Do not specify sources at this time** to instruct the Vivado tool to create a blank project (2).

While existing sources could be entered at this time, you will enter them later so that you can move through this portion of the project creation process more quickly.

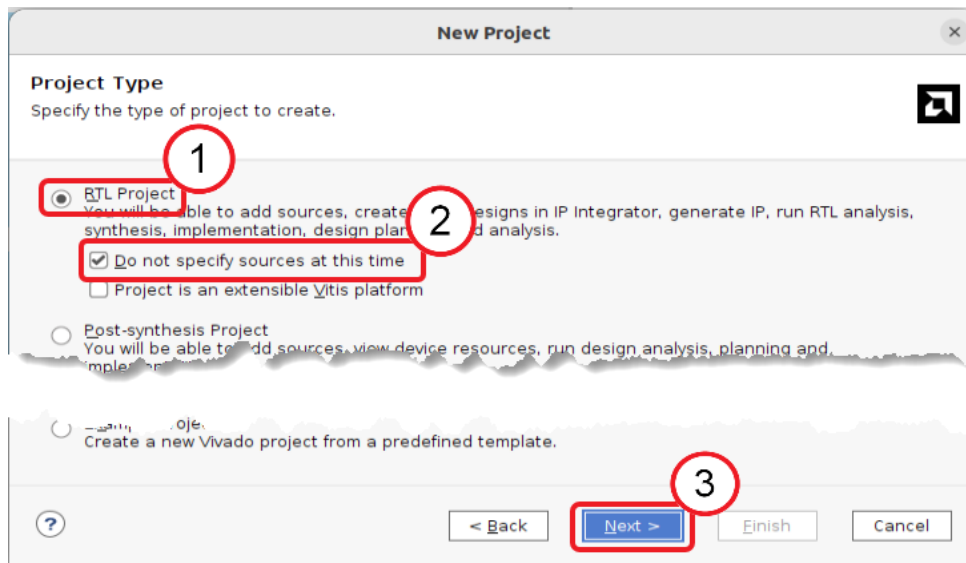


Figure 1-6: Specifying Project Options

1-3-7. Click **Next** to advance to the target device/platform selection (3).

1-4. Select the target part by first filtering by the board name. If you are not targeting a supported board, you will need to filter by the part.

1-4-1. Click **Boards** from the *Default part* area to filter by the board type rather than by the specific part (1).

1-4-2. Select **All** from the Vendor drop-down list in the Filter area (2).

This limits the number of boards seen to those manufactured by the specified vendor.

1-4-3. Select the **Versal VCK190 Evaluation Platform** board from the list.

If you accidentally click the hyperlink, a web page will open for that board. You can close the browser page.

If the board you want to use is not immediately visible, click the **Refresh** button to update the board catalog.

Note: While the web page contains important information and resources for the board, these details are not needed to complete this lab.

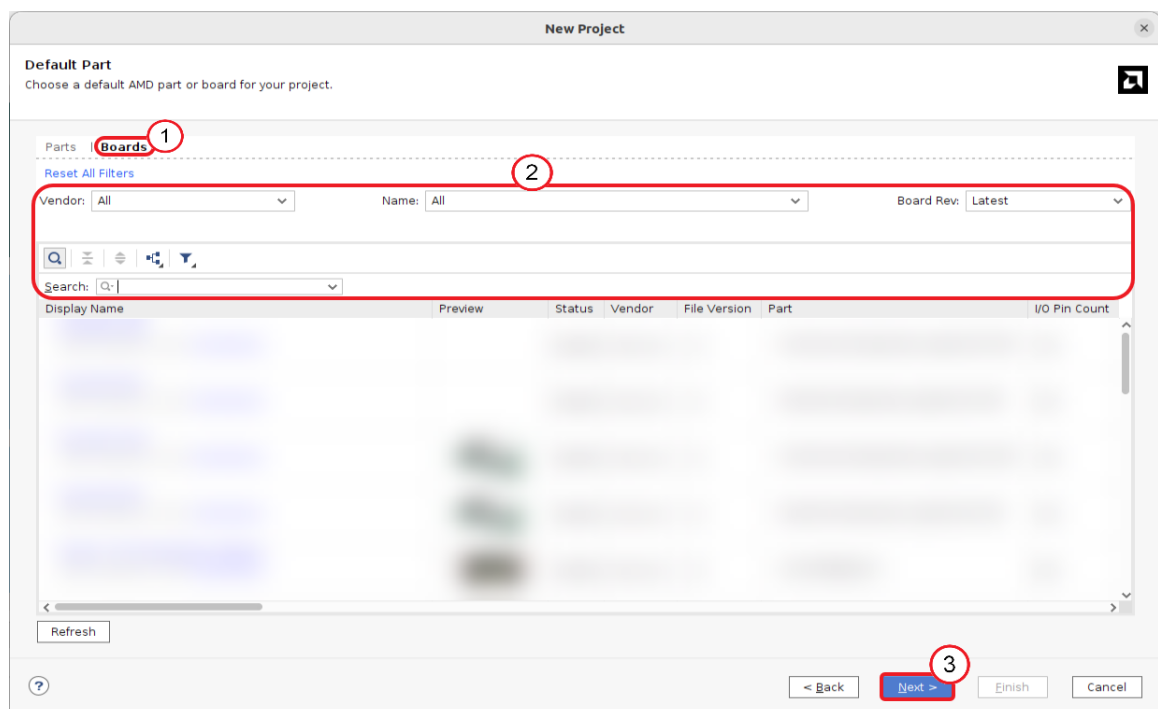


Figure 1-7: Selecting the Board for the Project

1-4-4. Click **Next** to advance to the summary (3).

A summary of your project is displayed. If you want to change any of the information that you entered, you can do so now by clicking **Back** until you reach the correct dialog box. After the project is created, the project properties can still be edited.

1-4-5. Click **Finish** to accept these settings and build the project.

Your project is constructed and leaves you in the operational portion of the Vivado Design Suite GUI.

The Vivado IP integrator is a graphical tool that assists you in "stitching" together various pieces of IP. This tool can be used for both embedded and non-embedded designs.

1-5. Create a Vivado IP integrator block diagram.

1-5-1. Expand **IP INTEGRATOR** in the Flow Navigator if necessary (1).

1-5-2. Click **Create Block Design** to start creating a new IP subsystem (2).

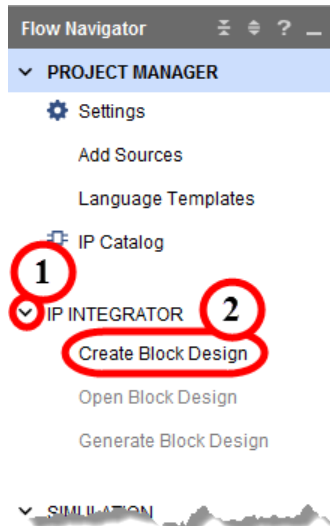


Figure 1-8: Launching the IP Integrator

1-5-3. Name the design **blkdsgrn** when the Create Block Design dialog box opens (1).

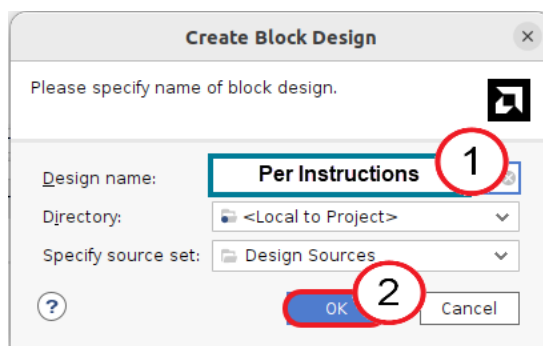


Figure 1-9: Creating an IP Integrator Block Design

1-5-4. Click **OK** to open a new, blank IP integrator canvas (2).

The IP integrator workspace opens with a note in the canvas area inviting you to begin adding IP.

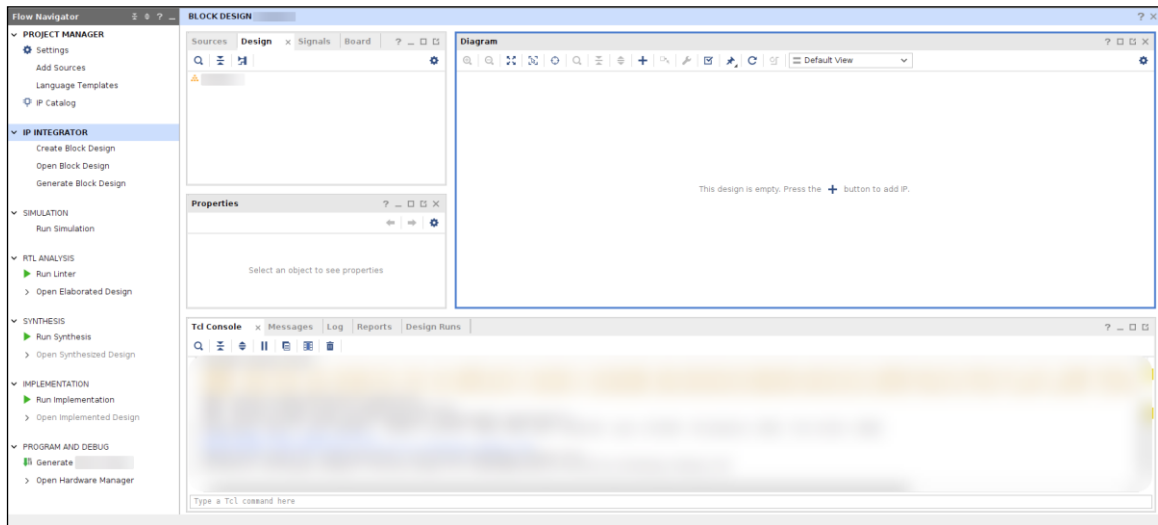


Figure 1-10: Initial View of the IP Integrator Tool

The Tcl script that you are about to load contains many *procs* that will aid you in this lab. A proc is the Tcl equivalent of a function.

The Vivado Design Suite offers both GUI and scripted control. Tcl provides scripted control. These Tcl commands can be entered directly into the tool one at a time (or block copied), or an entire Tcl script can be loaded and executed (sourced).

1-6. Run a Tcl script.

1-6-1. Select the **Tcl Console** tab to view the console.

Hint: If the Tcl Console is minimized, clicking the tab will partially expand the view.

1-6-2. Locate the Tcl command line entry.

The command line entry can be found either on the Welcome page before a project is opened, or once a project has been opened.

From the Welcome screen:

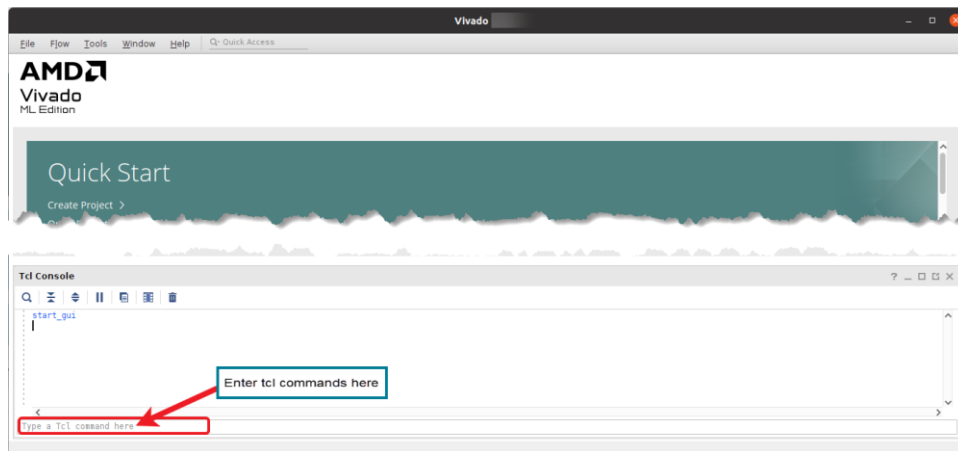


Figure 1-11: Accessing the Tcl Console from the Getting Started Page

You can always click the Tcl tab to maximize/restore it.

From the Tcl Console tab in an open project:

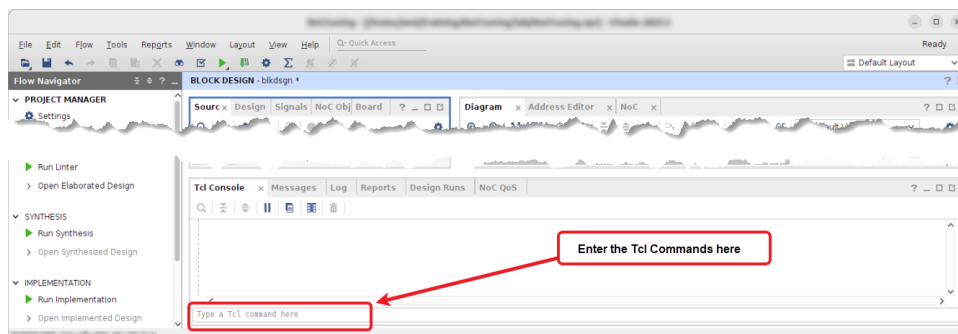


Figure 1-12: Entering Commands into the Tcl Console from an Open Project

Initially, the directory location begins within the tool installation directory.

- 1-6-3.** Enter the following command to change the current working directory to where the Tcl script is located:

```
cd $::env(TRAINING_PATH)/NoCintro/support
```

Note: The Tcl proc `env` reaches out to the operating system and returns the value of the environment variable given by `TRAINING_PATH`. The `$` indicates that the value of that variable should be returned and the two colons (`::`) indicate from which namespace the information should be pulled. Because there is nothing in front of the `::`, the global or base layer of the namespace is to be used.

- 1-6-4.** Verify that you are now where you want to be by entering the following into the Tcl command line:

```
pwd
```

This should show that you are in the `support` directory under the `NoCintro` directory in the `training` directory.

Note that the `training` directory can be anywhere in the system. What is essential is that you are currently working from the `support` directory under `NoCintro`.

- 1-6-5.** Run the following Tcl command to run the helper Tcl script for this lab:

```
source NoCintro.tcl
```

The Tcl script is run as though you typed each command included in the Tcl script into the Tcl command line. You can follow the execution of the script and monitor for any errors or warnings in the Tcl Console.

With the Tcl script now loaded, you will run the provided `addStuff` procs to fill the canvas with the IP elements that will be connected to the NoC.

1-7. Run the procs to set up the basics of the design.

- 1-7-1.** Enter the following command to add the supporting IP that you will connect to the NoC in the next step:

```
addStuff
```

Note that many of the elements that are added by this proc can be automatically generated when using the block automation capabilities for the NoCs. These elements are "manually" added here to help you get a sense of how things will connect and give you a better understanding of the design rather than just having the IP magically appear later.

With the canvas now set, you will continue to the next step where you will add and configure the two NoC IP blocks.

Adding the Logical NoCs

Step 2

Now you will add two pieces of NoC IP. This will illustrate how the NoC compiler can merge the logical NoC instantiations into a single physical NoC resource. The first logical NoC makes the crucial connection between the CIPS block (which contains the processor cores) and the DDR memory.

This task is an important aspect of virtually all embedded designs.

The second logical NoC ties the two traffic generators to two block RAM controllers. The traffic generators are surrogates for any IP that generates data and drives (that is, masters or initiates) data into the NoC. Conversely, the block RAM controllers represent any IP that sinks or consumes the data.

First, add the logical NoC that will connect the CIPS IP to the DDR memory.

2-1. Open the IP catalog.

2-1-1. The IP catalog can be opened in several ways:

- If the design is empty, add IP by clicking either of the '+' icons.

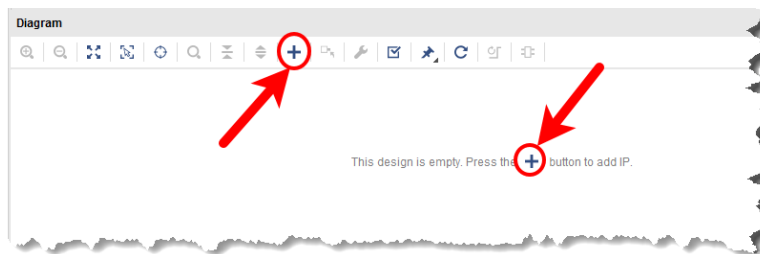


Figure 1-13: Opening the IP Catalog from the Initial Information Bar Display

- Right-click any background space in the workspace and select **Add IP**.

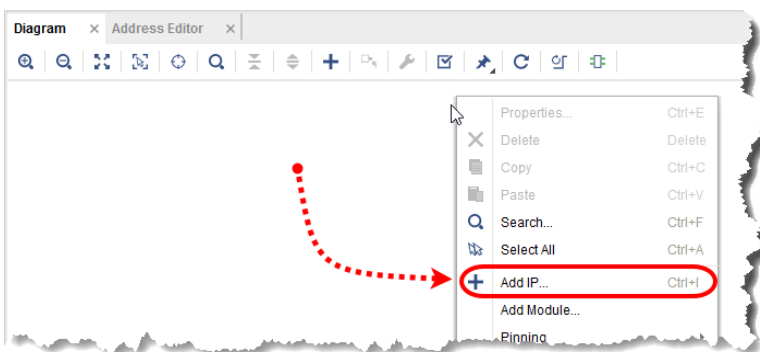


Figure 1-14: Opening the IP Catalog by Right-Clicking the Workspace

- Press <Ctrl + I> to open the IP catalog.

Important Note: Using Windows > IP Catalog will add the new IP to the top level of the design's hierarchy—it will NOT add the IP to the diagram! It is, however, possible to float this window and drag-and-drop IP into the Block Design canvas.

2-2. Once the IP catalog opens, search for **AXI NoC** and add it to the canvas.

2-2-1. Type all or part of the IP name into the Search field.

2-2-2. Locate **AXI NoC**.

Hint: You can scroll with the mouse or use the scroll bar on the right side of the list. The more of the IP name that you provide, the fewer items that you will need to manually scroll through.

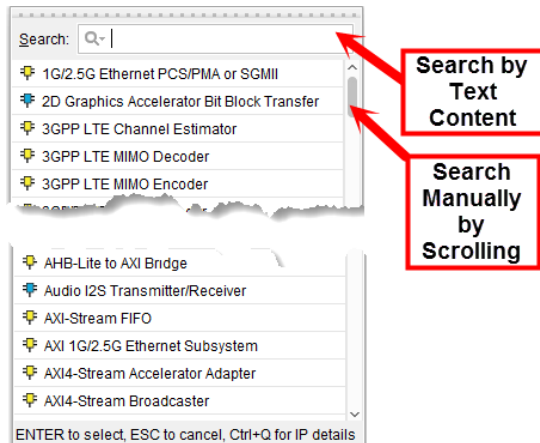


Figure 1-15: Two Mechanisms to Search for IP

2-2-3. Double-click the name of the IP to add it to the design (if you use this method, the IP catalog will close after the IP has been added to the design).

or

Drag-and-drop the IP into the workspace.

The IP catalog remains open once the IP has been added to the design. This is a convenient way to quickly add multiple pieces of IP. Pressing the **<Esc>** key or clicking an open space in the canvas will close the IP catalog.

2-2-4. Select the newly added NoC IP by clicking it once.

The properties for the block will appear in the Block Properties view.

2-2-5. Rename the IP by typing the following into the Name field in the Block Properties window:

axi_noc_DDR

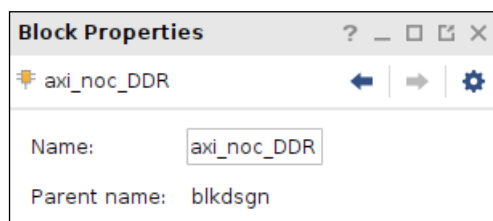


Figure 1-16: Renaming the NoC IP

2-2-6. Press **<Enter>** to register the name change.

This will help you identify its purpose as you will be adding another NoC momentarily.

2-3. Configure the NoC to make an exclusive connection to the DDR memory.

2-3-1. Double-click the **axi_noc_DDR** block to open the Re-customization Wizard.

Alternatively, you can right-click the block and select **Customize Block**.

The Re-customize IP window opens for this NoC block. You will notice several tabs across the top.

2-3-2. Select the **General** tab to access the high-level features for this logical NoC (1).

2-3-3. Under the AXI Interfaces section, verify that **1** is selected from the Number of AXI Slave Interfaces drop-down list to enable one slave port (2).

This will enable the CIPS to master the NoC to reach the DDR.

2-3-4. Select **0** from the Number of AXI Master Interfaces drop-down list to disable the master ports.

This NoC is only being used to connect the CIPS to the DDR so no master ports are needed to drive any logic or peripheral connections.

Since there is a slave interface, it will need to be clocked.

2-3-5. Select **1** from the Number of AXI Clocks drop-down list to enable a single AXI clock (2).

2-3-6. Under the Memory Controllers - DDR4/LPDDR4 section, select **Single Memory Controller** from the Memory Controller drop-down list (3).

Here, the specifics of the DDR that you are connecting to are irrelevant. Since the VCK190 board was specified, the tools were able to automatically populate the necessary fields with the correct information.

Typically, you would only need to define the timing parameters and other DDR configurations based on the specifics of your board or if the memory was changed. This is done by copying the parameters given in the memory datasheet into the appropriate field in the Re-customization Wizard.

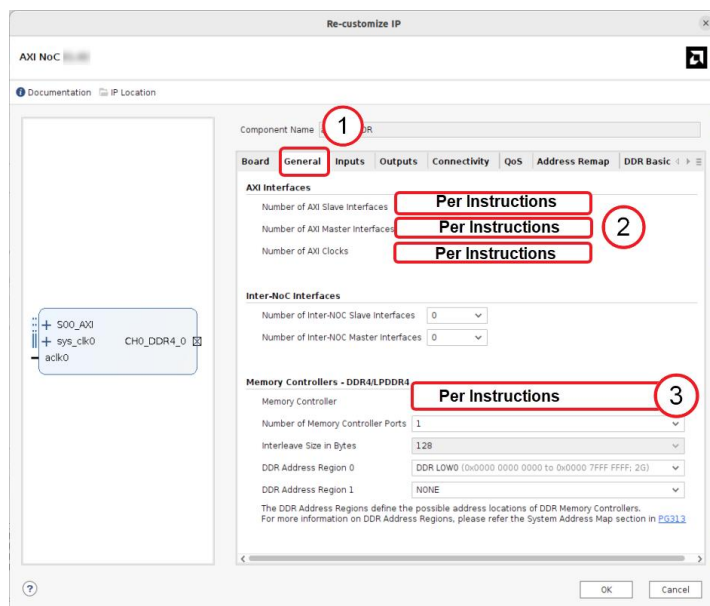


Figure 1-17: Setting Parameters for the AXI NoC under the General Tab

With the basic information captured, you will now progress through the other tabs, refining your selections.

- 2-3-7.** Select the **Inputs** tab to specify that the NoC input will be coming from the CIPS (1).

Note: More details regarding the fields available in this tab will be discussed shortly.

- 2-3-8.** Select **PS Non-Coherent** under the Connected To column (2).

This specifies that the type of port is coming from the processing system and does not have to be kept coherent with the caches.

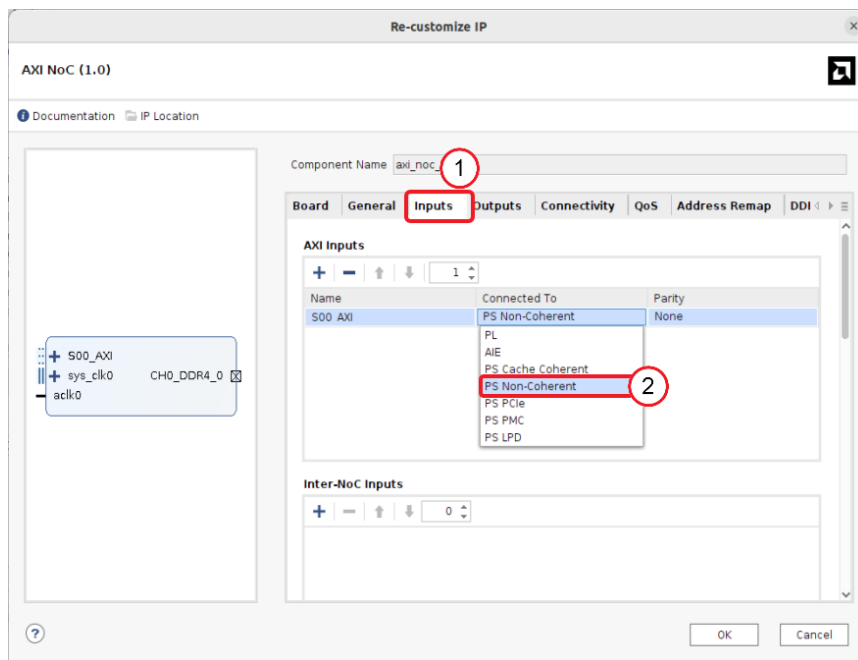


Figure 1-18: Selecting a Non-Coherent Port Type

Note: Since the output of the NoC is tied to the DDR memory controller, the Outputs tab is empty and should be left that way.

- 2-3-9.** Select the **Connectivity** tab to map the input port to the output port (1).

Note that there is only one option to be made: Connect the input from the slave port (which is connected to the CIPS block) to the Memory Controller port (which is an input into the Memory Controller and regulates traffic to the DDR and is indicated by the column heading "MC Port 0").

- 2-3-10.** Select the check box for S00_AXI ps_nci row under the MC Port 0 column to make the connection (2).

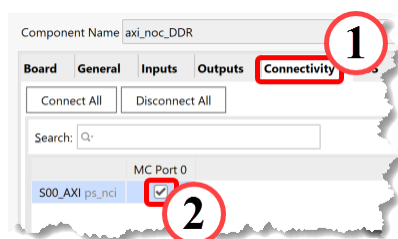


Figure 1-19: Connecting the Input Port to the Output Port

Note: The remaining tabs can be ignored for now as anything having to do with the DDR parameters has been automatically populated. Since there is only one connection from the CIPS block to the memory controller, there is not going to be any contention for time on the network and the QoS value can be set to anything.

2-3-11. Click **OK** to accept the modifications and close the Re-customization Wizard.

2-4. **Connect the CIPS to the NoC and the NoC to the DDR pins.**

Begin this process by running block automation on the NoC DDR IP.

2-4-1. Click **Run Connection Automation** in the info bar.

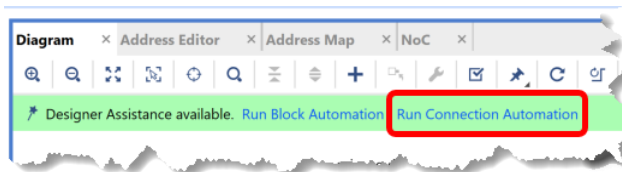


Figure 1-20: Selecting Run Connection Automation

This will open the Run Connection Automation dialog box. At this point in the design, you are only connecting the configured NoC IP to the DDR.

2-4-2. Select *only* the **axi_noc_DDR** entry under All Automation on the left.

This will connect:

- The output port of the NoC to the package pins that connect to the DDR memory
- The clock for the DDR
- The AXI connection to the CIPS - FPD_AXI_NOC_0

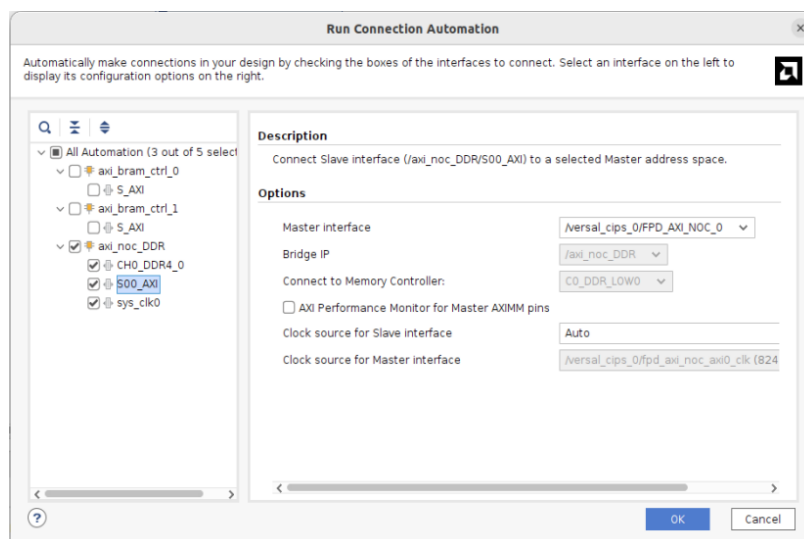


Figure 1-21: Running Connection Automation on the AXI NoC DDR IP

Note: You will be adding another NoC IP to manage the connections to the block RAMs, which is why you are omitting these connections now.

2-4-3. Click **OK** to run the connection automation.

2-5. Add a second logical NoC that will connect the two AXI traffic generators to the block RAM controllers.

2-5-1. Add another piece of NoC IP using the same procedure as described above in step 2-2.

2-5-2. Select the newly added NoC IP.

The properties for the block will appear in the Block Properties view.

2-5-3. Rename the IP to a meaningful name by typing the following into the Name field in the Block Properties window:

axi_noc_TG2BRAM

TG2BRAM indicates that this IP is the NoC that connects the traffic generator to the block RAMs.

2-5-4. Press <Enter> to register the name change.

This will help you differentiate between the two NoCs in this design: the Traffic Generator to Block RAM (TG2BRAM) and the CIPS to DDR (DDR) logical NoC IPs.

2-6. Configure the TG2BRAM NoC to make the connections between the traffic generators and the block RAM controllers.

2-6-1. Double-click the **axi_noc_TG2BRAM** block to open the Re-customization Wizard.

Alternatively, you can right-click the block and select **Customize Block**.

The Re-customize IP window opens for this NoC block.

2-6-2. Select the **General** tab to access the high-level features for this logical NoC (1).

2-6-3. Under the AXI Interfaces section, enable **2** AXI slave interfaces (2).

This is where the data from the traffic generators will enter the NoC.

2-6-4. Enable **2** AXI master interfaces (2).

This is where the data will exit the NoC and be routed to the block RAM controllers.

2-6-5. Since the same clock is used for both masters and slaves, only **1** AXI clock is required (2).

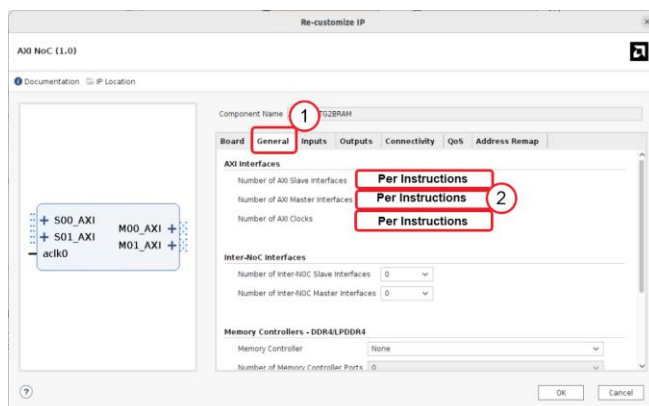


Figure 1-22: Configuring the General Tab for the NoC

Since you have the Re-customization Wizard still open, let's look at the next two tabs to understand what they do even though you will not be making any changes to them.

2-6-6. Select the **Inputs tab.**

There are two fields here: AXI Inputs and Inter-NoC Inputs.

The AXI Inputs show the number of slave ports on the device. This was populated when you selected two inputs under the General tab. Here, you can make adjustments, such as indicating the connections type and identifying the port's source clock. The default is a PL connection, which you will use since the AXI traffic generators are present in the PL as well as the block RAMs.

The Inter-NoC inputs are used to connect two logical NoCs. For example, you may choose to keep two groupings of circuitry visually or conceptually separate, but some information must flow between the two logical NoCs. Instead of manually merging the two NoC IPs into a single NoC IP, you can specify an input port that is sourced by another NoC. The connection is made in the usual fashion using the IP integrator tool (using a wire bundle).

Since the default connection type (PL) is appropriate for receiving input from the programmable logic, no changes are necessary for this tab.

2-6-7. Select the **Outputs tab.**

Just as you were able to specify where the inputs connected to, you can do the same for the output destinations.

Since the block RAMs and their controllers all reside in the PL section, no changes to the default settings are needed.

2-6-8. Select the **Connectivity tab to make connections between the input and output ports.**

The table here represents how data flows from the slave ports (the inputs) to the master ports (the outputs). By selecting the appropriate box, you form a connection between the slaves and the masters.

The tools automatically populate the S00_AXI:M00_AXI box. This means that data arriving on S00 will be routed through to M00. The tools do not populate the remainder of the table, resulting in no paths between the arriving data on S01 and either master port.

2-6-9. Connect S01 to both master ports by selecting the appropriate boxes.

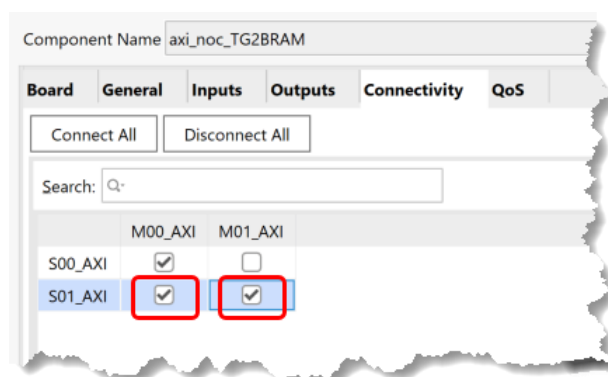


Figure 1-23: Connecting S01 to Both the M00 and M01 Ports

This allows data arriving on S01 to be directed to M00 or M01 based on the addressing. Data arriving on S00 only flows to M00.

Note: Since no memory controller was specified, the DDR tabs are absent.

2-6-10. Click **OK** to accept the modifications and close the Re-Customization Wizard.

While many connections can be made using the Connection Automation capability, you will need to make some connections manually.

2-7. Connect the TG2BRAM NoC to the traffic generators and block RAM controllers.

2-7-1. Click the **Regenerate Layout** icon (🔄) to re-arrange the IP in a more organized fashion.

2-7-2. Click-and-drag a connection from `noc_axi_tg_0.M_AXI` to `axi_noc_TG2BRAM.S00_AXI` to drive data from the traffic generator into the NoC's S00_AXI port.

Hint: Feel free to move the components to make this connection easier. You can always regenerate the layout at any time.

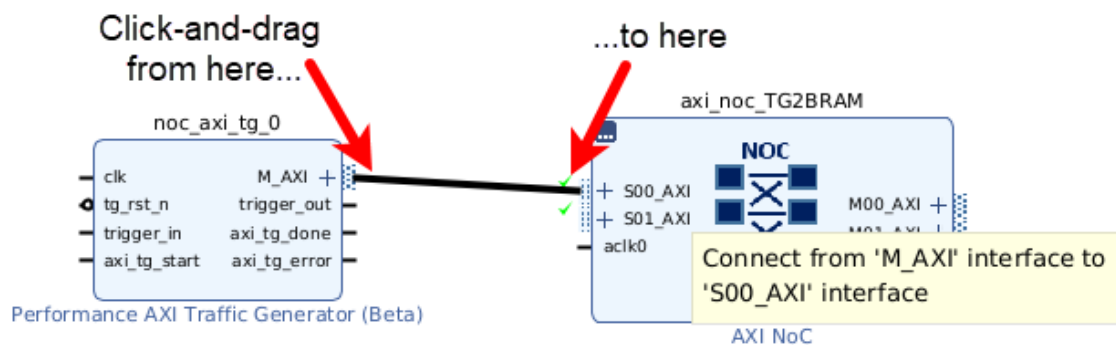


Figure 1-24: Connecting the Performance AXI Traffic Generator's Output to the NoC

2-7-3. Repeat this action to establish a connection between the traffic generator's output (`noc_axi_tg_1.M_AXI`) to the NoC that manages access to the block RAMs (`axi_noc_TG2BRAM.S01_AXI`).

2-7-4. Create a connection from the NoC's output to the BRAM Controller's input by click-dragging from `axi_noc_TG2BRAM.M00_AXI` to the `axi_bram_ctrl_0.S_AXI` port.

2-7-5. Repeat this action to establish a connection between `axi_noc_TG2BRAM.M01_AXI` to the `axi_bram_ctrl_1.S_AXI` port.

2-7-6. Click the **Regenerate Layout** icon (🔄) to clean up the display.

The layout shown here is generated using the Interfaces view, which is selected from the View selection.

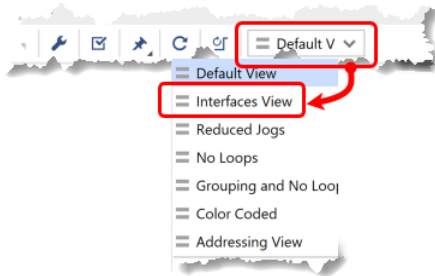


Figure 1-25: Selecting the Interfaces View for Regenerating the Block Design View

This part of the diagram should be similar to the following figure.

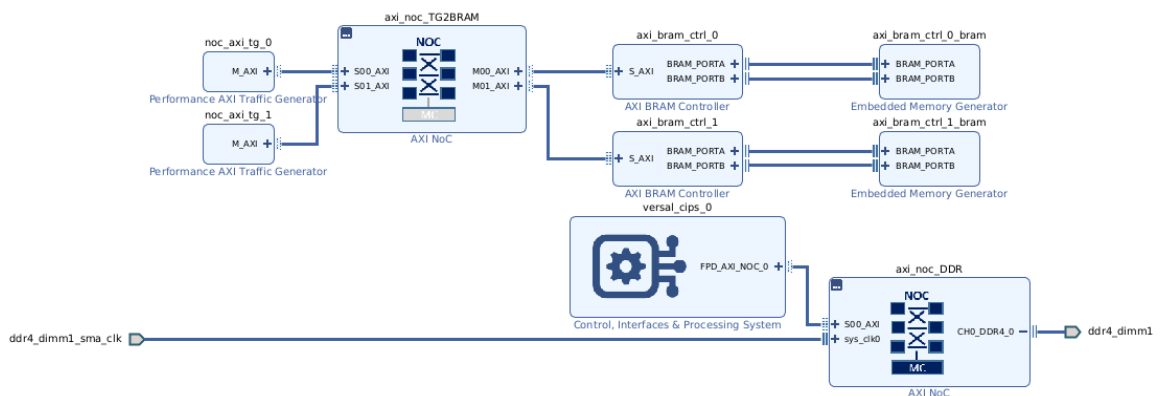


Figure 1-26: NoC Connected to the Traffic Generators and the Block RAM Controllers

2-7-7. Once ready, make certain that **Default Layout** is selected from the view selection drop-down list in the main toolbar.

2-8. **Run the Connection Automation to complete the TG2BRAM NoC's connections.**

2-8-1. Click **Run Connection Automation** in the info bar.

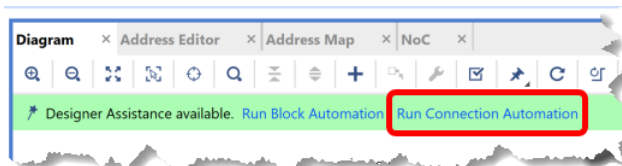


Figure 1-27: Selecting Run Connection Automation

2-8-2. Click the **Expand All** (⏏) icon to expand the tree fully.

2-8-3. Select **All Automation**.

This will establish the connections for all the clocks that have not yet been made.

When selected (not necessarily checked, but selected), each leaf node will display the options for that element.


Since this design is only for exercising the tools and not implementing in silicon, the specific clock selected in the "Select Board Part Interface" is of no concern. For your edification, view the available clocks by clicking the pull-down menu. This field is intelligently populated by the board selection that was done for you when the project was created.

- 2-8-4.** Click **OK** to run the connection automation.

2-9. Finish the connections.

The previous connection automation exposes the need for additional connections. You will now re-run the Connection Automation to complete the design.

- 2-9-1. Click **Run Connection Automation** from the info bar.
- 2-9-2. Select **All Automation** to ensure that the reset to the CIPS is created.
- 2-9-3. Click **ext_reset_in** under All Automation > rst_versal_cips_0_333M to show its options in the right-hand panel.
- 2-9-4. Select **/versal_cips_0/pl0_resetn (ACTIVE_LOW)** from the Select Reset Source pull-down menu.
- 2-9-5. Click **OK** to run the connection automation.

All that remains is enabling the traffic generators. While you could instantiate a constant block and set its output to "high" and then make the four connections to the two traffic generators, a Tcl script is provided to speed you through this simple process.
- 2-9-6. Enter **tgEnable** in the Tcl Console bar to make the connections that enable and start the traffic generators.
- 2-9-7. Click the **Regenerate Layout** icon () to clean up the display.

Remember that you may be in the Interfaces view. You can return to the normal view by selecting **Default View** from the View selection.

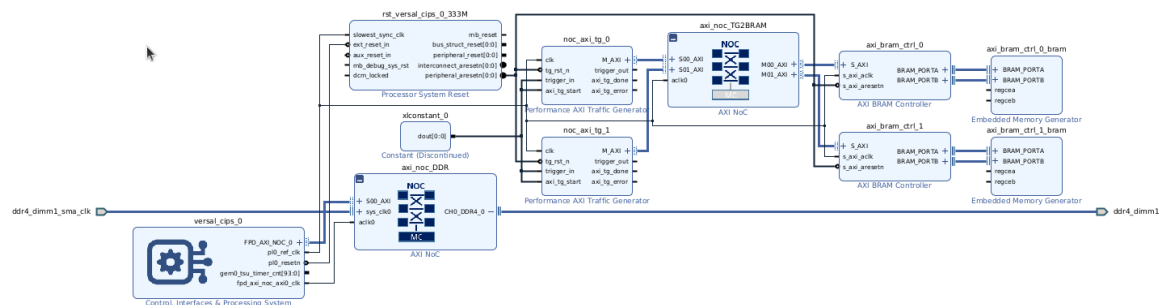


Figure 1-28: Design Default View

2-10. Configure the addresses so that the NoC can properly map memory.

2-10-1. Select the **Address Editor** tab to open the address mapping view.

2-10-2. Click the **Expand All** (☰) icon to ensure that the table is fully expanded.

Note that there are several unassigned regions. These must be assigned before you can move forward.

2-10-3. Click the **Assign All** (⇩) icon to automatically assign addresses to the unassigned elements.

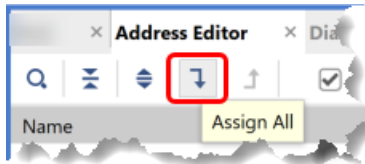


Figure 1-29: Using the Assign All Icon

Alternatively, you can right-click anywhere to open the context menu and select **Assign All**.

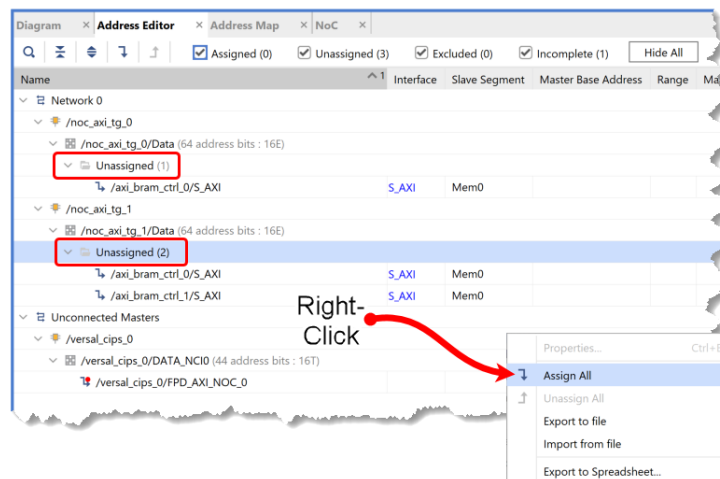


Figure 1-30: Generating Addresses for Unassigned Elements

2-10-4. Click **OK** to close the dialog box showing that all address assignments have been completed successfully.

2-11. Validate the design to both check the design for issues as well as run the NoC compiler.

2-11-1. From the menu bar, select **Tools > Validate design**.

Alternatively, you can press <F6> to validate the design.

The design is now complete. It is now time to review the results and explore what the NoC compiler produced.

2-11-2. Click the **OK** button to close the pop-up window that reports the completion of validation.

Reviewing the Results

Step 3

With the design now complete, let's look at the results.

3-1. Reorganize the block diagram by addresses.

This is especially useful when using NoC IPs as this view helps with seeing what is present in the different address ranges.

3-1-1. From the Diagram tab (1), click the drop-down arrow next to the Optimize Routing (🔗) icon (2).

3-1-2. Select **Addressing View** (3).

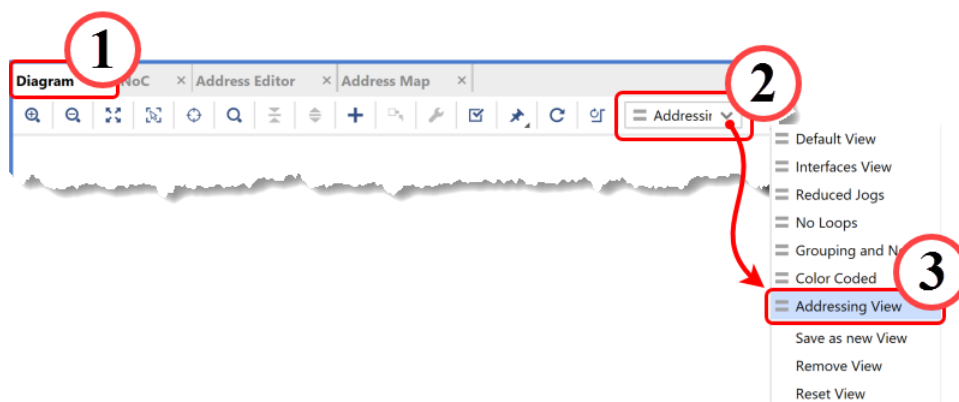


Figure 1-31: Organizing the Canvas by Address View

3-1-3. Click the **Regenerate Layout** icon (🔄).

This will reorganize the diagram into groups so that it is easier to see what addresses connect with which pieces of IP.

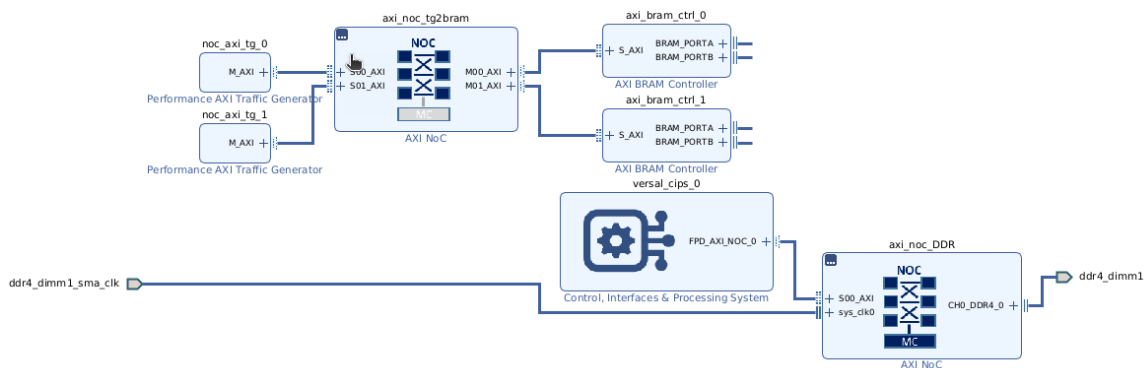


Figure 1-32: Block Design in Addressing View

3-2. View the results of the NoC compiler.

3-2-1. Select the **NoC** tab.

If the tab is not visible, select **Window > NoC**.

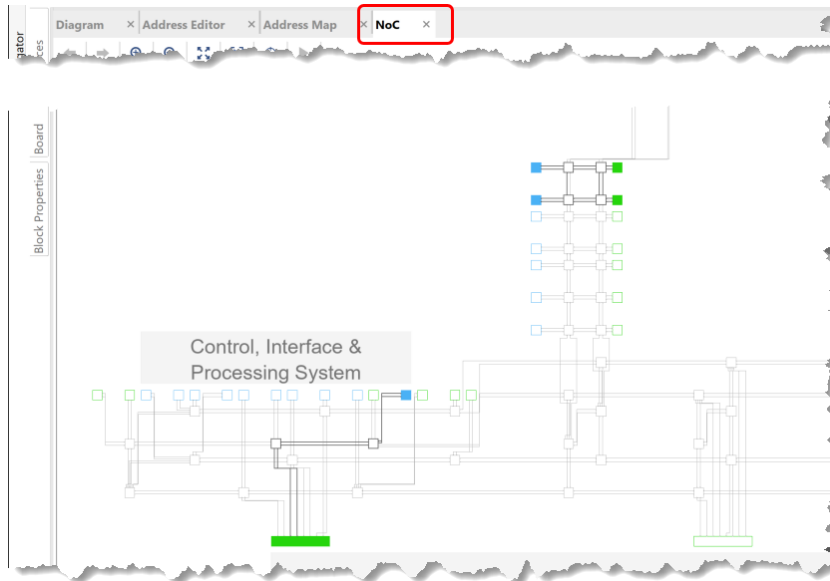


Figure 1-33: NoC View after Validation

3-2-2. Hover over any of the small boxes.

Notice that information specific to this node appears as a tool tip.

3-2-3. Click any small box to see its properties in the NoC Site Properties box.

3-2-4. Explore the properties by selecting each active block.

If the Properties view is not open, you can right-click an active block and select **NoC Site Properties**.

Question 1

What appears in the NoC Site Properties window?

Question 2

What do the different colors represent?

Question 3

What masters are shown?

Let's look at one more item before you finish the lab—the representation of how the addresses are mapped.

3-3. Review the address map.

3-3-1. Select the **Address Map** tab.

If the tab is not visible, select **Window > Address Map**.

3-3-2. Click Generate Address Map in the Address Map tab.

3-3-3. Modify the network selections at the top to see how the display changes (1).

Hint: You may need to click **Zoom Fit** icon to adjust the display (2).

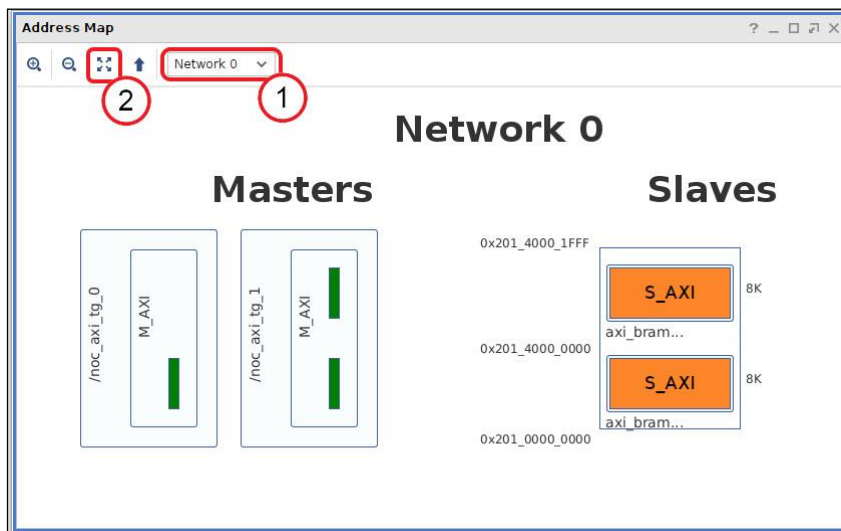


Figure 1-34: Default View for the Address Map

3-4. Close the Vivado Design Suite.

3-4-1. Select **File > Exit**.

The Exit Vivado dialog box opens.

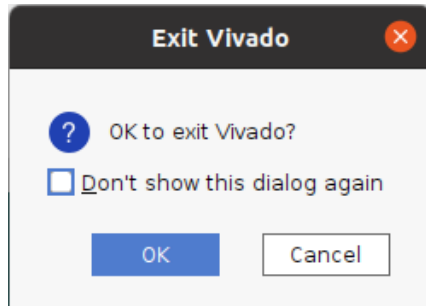


Figure 1-35: Exit Vivado Dialog Box

3-4-2. If you are asked to save the project or a portion of the project, select whichever elements of the project you want to save, then click **Save** to save the selected elements; otherwise, click **Don't Save**.

3-4-3. Click **OK** when you are asked to exit the Vivado Design Suite.

Note: You can choose to select the *Don't show this dialog again* option to avoid being asked for confirmation when exiting the Vivado Design Suite.

Some systems (particularly VMs) may be memory constrained. Removing the workspace frees a portion of the disk space, allowing other labs to be performed.

You can delete the directory containing the lab you just ran by using the graphical interface or the command-line interface. You can choose either mechanism. Both processes will recursively delete all the files in the `$TRAINING_PATH/NoCintro` directory.

3-5. [Optional] [Only for local VMs—not for CloudShare] Clean up the file system.

Using the GUI:

3-5-1. Navigate to `$TRAINING_PATH/NoCintro`.

3-5-2. Select **NoCintro**.

3-5-3. Press **<Delete>**.

-- OR --

[Linux users]: Using the command line:

3-5-4. Press **<Ctrl + Alt + T>** to open a terminal window.

3-5-5. Enter the following command to delete the contents of the workspace:

```
[host] $ rm -rf $TRAINING_PATH/NoCintro
```

Summary

You just completed building a basic system showing how the NoC can be used to connect two regions in the PL as well as connecting the CIPS to the NoC's DDR controller to provide the processors in the CIPS access to the DDR—something that almost every embedded design requires.

Answers

1. What appears in the NoC Site Properties window?

The unmodifiable name of the node, which includes the location information, the mode that the element represents (master, slave, packet switch, or route), and the type (NPS, PL_NSU, PL_NMU, and PS_CCI_NMU, for example).

2. What do the different colors represent?

Blue represents masters, green represents slaves, and dark outlines on the squares represent a utilized network packet switch. Dark lines represent utilized routing resources.

3. What masters are shown?

NOC_NMU512_X0Y6, which is of type PL_NMU.

NOC_NMU512_X0Y5, which is of type PL_NMU.

NOC_NMU128_X0Y5, which is of type PS_NCI_NMU.

Note: The position of the NoC may change when you run the lab.

Lab 2: System Simulation

2025.1

Abstract

This lab demonstrates a system design running on the AMD Versal™ adaptive SoC programmable logic (PL), processing system (PS), and AI Engines. The lab also illustrates how to validate a design running on these heterogeneous domains by performing hardware emulation (system simulation).

This lab should take approximately 60 minutes.

CloudShare Users Only

You are provided with three attempts to finish a lab, where the time allotted to complete each lab is twice the expected completion time. Once the timer starts, you cannot pause the timer. Each lab attempt resets the previous attempt—your work from previous attempts is not saved.

Objectives

After completing this lab, you will be able to:

- Compile a complete system-level design that includes HLS functions, adaptive data flow (ADF) graphs for AI Engines, and host code
- Run hardware emulation in a mixed AI Engine, RTL cycle-accurate, QEMU-based system simulator (RTL co-simulation)

Introduction

The AMD Versal device delivers programmable logic (PL), a processing system (PS), and AI Engines, all tightly coupled with customizable memory hierarchies and a network on chip (NoC) to efficiently move data among the processing elements.

The following is the system-level design that is used in this lab.

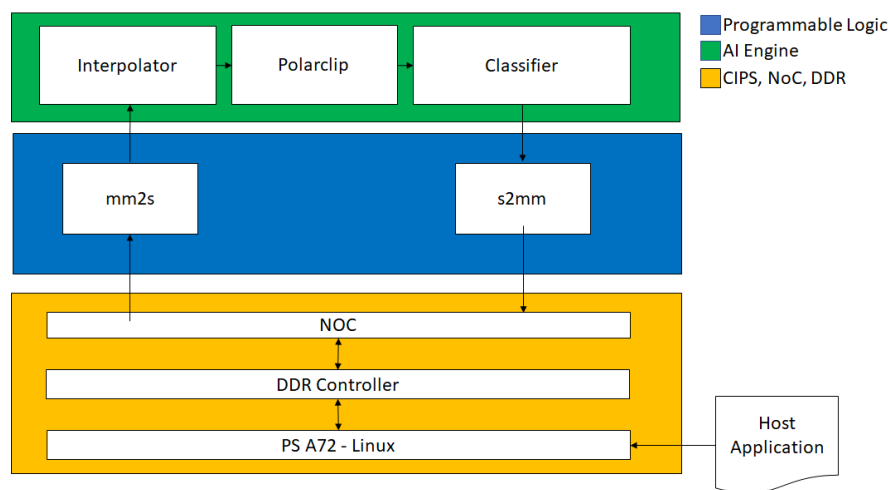


Figure 2-1: System Design

This system design has different kernels that run on the AI Engine, PS, and PL domains simultaneously. The AI Engine domain contains a simple graph consisting of three kernels. These kernels are connected by both buffers and streams. The PL domain contains two HLS-type data mover kernels that provide input and capture output from the AI Engine. The PS domain contains a host application that controls the entire system.

There are five kernels used in this design. The following table provides the details of each kernel.

Kernel	Kernel Type	Description
MM2S	HLS	Memory Map to Stream HLS kernel to feed input data from DDR to the AI Engine interpolator kernel via the PL DMA.
Interpolator	AI Engine	Half-band 2x up-sampling FIR filter with 16 coefficients. Its input and output are cint16 buffer interfaces and the input interface has a 16-sample margin.
Polar_clip	AI Engine	Determines the magnitude of the complex input vector and clips the output magnitude if it is greater than a threshold. The polar_clip has a single input stream of complex 16-bit samples and a single output stream whose underlying samples are also complex 16-bit elements.
Classifier	AI Engine	Determines the quadrant of a complex input vector and outputs a single real value depending on which quadrant. The input interface is a cint16 stream and the output is a int32 buffer.
S2MM	HLS	Stream to Memory Map HLS kernel to feed output result data from the AI Engine classifier kernel to DDR via the PL DMA.

The steps for building an AI Engine system design in the Vitis Unified IDE for use on a Versal device are as follows:

1. Launch the Vitis Unified IDE in a new workspace.
2. Create an AI Engine component with ADF graphs.
3. Create HLS components for the programmable logic region of the device.
4. Create an application component to run on an embedded processor.
5. Create a system project to link the different components.

In this lab, you will review a provided system project that was created using the above steps. You will then package the boot files by using the Vitis `v++ --package` command. Finally, you will run the system in hardware emulation.

Top-level System Project

The top Vitis Unified IDE project (the system project) will be divided into multiple sub-projects:

- Top-level system project
 - AI Engine
 - Programmable logic
 - Processing system
 - Hardware link

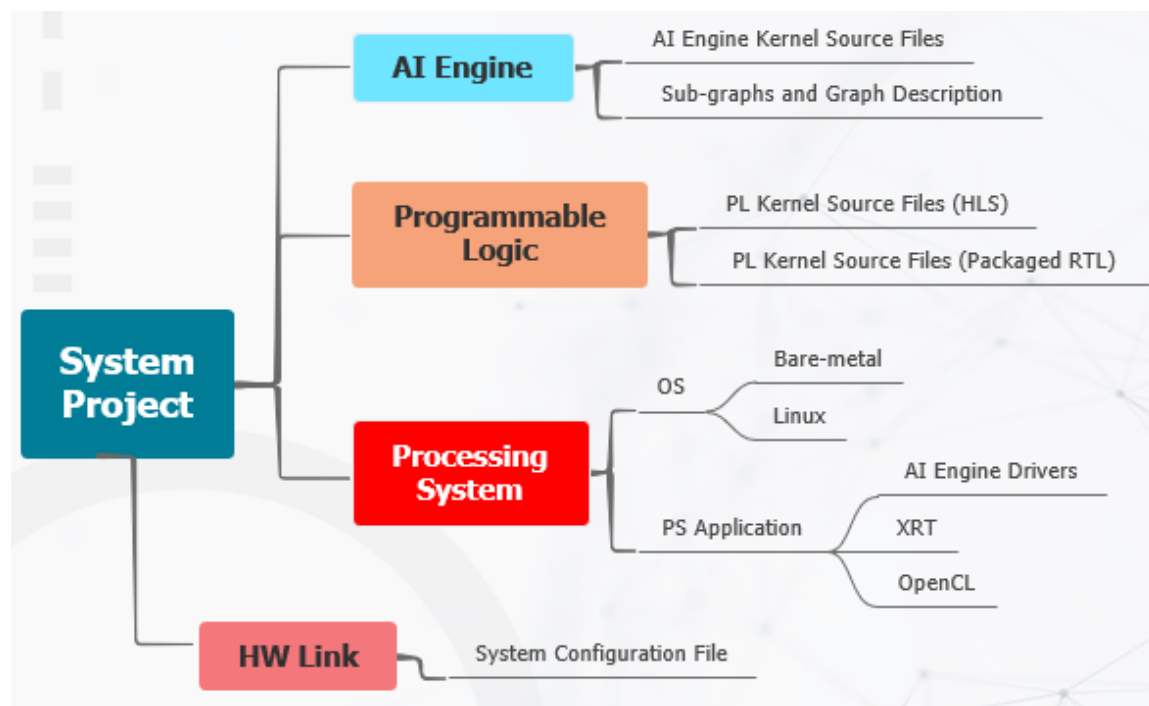


Figure 2-2: Vitis Unified IDE – System Project Details

During software emulation, the Vitis Unified IDE automatically builds the necessary structure to feed the AI Engine design and get back the data. In this hardware emulation flow, the origin of the data must be specified.

The design here takes a large number of input samples and sends them to PL and AI Engine kernels where the compute intensive part happens. Once the input samples are processed, they

are outputted. The PS controls this data movement using XRT APIs. You will run the system simulation and verify the output in the console in the final step.

Packaging the System

The Vitis `v++ --package` command generates SD card and other Flash images required for booting the system, as well as 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.

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.

In this lab, you will use the command line flow for the packaging process and learn about the required options for the Vitis `package` command.

Hardware Emulation

Hardware emulation simulates a complete Versal adaptive SoC system composed of the AI Engine, PS, and PL. Using the Vitis software platform, you can integrate blocks and functions targeting all three compute domains. The Vitis linker automatically generates a complete co-simulation setup involving RTL, the AIE simulator, and QEMU models:

- Embedded software code running on the PS is emulated using QEMU.
- Code running on the AI Engines is emulated using the AI Engine simulator.
- User PL kernels are simulated as RTL code.
- IP blocks in the hardware platform are simulated either as RTL or AIE simulator TLM, based on the available or selected types of models.

In this lab, you will perform the hardware emulation for the given design. Hardware emulation allows you to simulate the entire design and test the interactions between the PL, PS, and AI Engine prior to implementation. Because hardware emulation provides full debug visibility into all aspects of the application, it is easier to debug complex problems in this environment than in real hardware.

Understanding the Lab Environment

The labs and demos provided in this course are designed to run on a Linux® platform.

One environment variable is required: `TRAINING_PATH`, which points to the location of the lab files. This variable comes configured in the CloudShare/CustEd_VM environments.

Some tools can use this environment variable directly (that is, `$TRAINING_PATH` is recognized and automatically expanded), and some tools require manual expansion (`/home/amd/training` for the CloudShare/CustEd_VM environments). The lab instructions describe what to do for each tool. Other environments require the definition of this variable for the scripts to work properly.

The Vivado™ Design Suite and the Vitis™ Unified IDE offer a Tcl environment used in many labs. When either tool is launched, it starts with a clean Tcl environment with none of the procs or variables remaining from any previous launch of the tools.

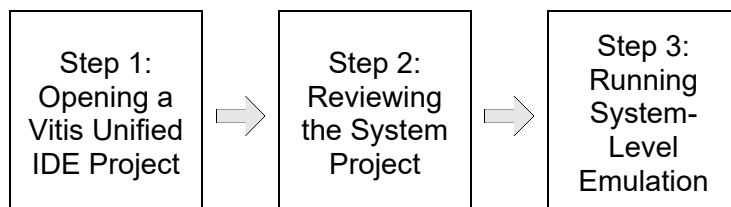
If you sourced a Tcl script or manually set any Tcl variables and you closed the tool, when you reopen the tool, you will need to re-source the Tcl script and set any variables that the lab requires. This is also true of terminal windows—any variable settings will be cleared when a new terminal opens.

Nomenclature

Formal nomenclature is used to explain how different arguments are used. The following are some of the more commonly used symbols:

Symbol	Description	Example	Explanation
<text>	Indicates a field	cd <dir>	<dir> represents the name of the directory. The < and > symbols are NOT entered. If the directory to change to is XYZ, then you would enter cd XYZ into the environment.
[text]	Indicates an optional argument	ls [more]	This could be interpreted as ls <Enter> or ls more <Enter>. The first instance lists the files in the current Linux directory, and the second lists the files in the current Linux directory, but additionally runs the output through the <code>more</code> tool, which paginates the output. Here, the pipe symbol () is a Linux operator.
	Indicates choices	cmd <ZCU104 VCK190>	The <code>cmd</code> command takes a single argument, which could be ZCU104 OR VCK190. You would enter either <code>cmd ZCU104</code> or <code>cmd VCK190</code> .

General Flow



Opening a Vitis Unified IDE Project

Step 1

You will begin this lab by opening the system design project that has been provided for you.

1-1. Launch the Vitis Unified IDE.


- 1-1-1. Click the **Vitis** icon () from the taskbar or desktop to launch the tool.



Figure 2-3: Launching the Vitis Unified IDE from the Taskbar

Alternatively, from a Linux terminal window (<Ctrl + Alt + T>), enter the following:

```
[host] $ source /opt/amd/2025.1/Vitis/settings64.sh; vitis
```

Note: This installation path is valid for the CloudShare and CustEd VM environments. Use the proper path for your environment.

It is important to note that not all tools included in the Vitis suite of tools are supported under Windows.

- 1-1-2. If not already maximized, click the Maximize icon in the upper-right corner of the tool window to see all the views.

1-2. Set the workspace.

Best practices recommend setting a workspace regardless of the flow you are following as it makes it easier to find the tool-generated logs and scripts.

Alternatively, you can immediately jump into one of the development flows, where you will eventually be asked to specify a workspace. This causes the log and script files to be placed in two different directories.

- 1-2-1. From the Vitis Components window, click the **Set Workspace** link.

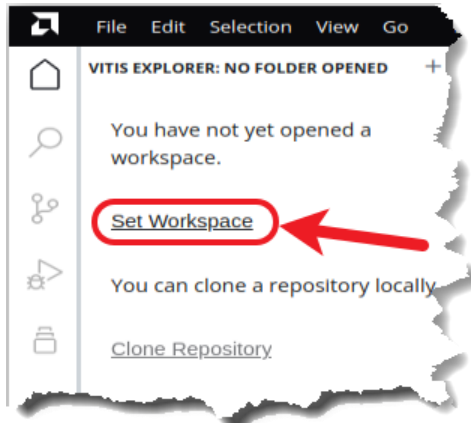


Figure 2-4: Setting the Workspace in the Vitis Unified IDE

- 1-2-2. Browse to the following location:

```
$TRAINING_PATH/system_simulation/lab/sys_project
```

- 1-2-3. Click **Open** to open the new workspace.

The tool relaunches using the new workspace.

- 1-2-4. Close the **Welcome** tab if it appears.

This tab opens various types of components, enables access to documentation and examples, quickly switches workspaces, etc. If you want to reopen the Welcome tab, select **Help > Welcome**.

1-3. Import a Vitis Unified IDE project.

1-3-1. Select **File > Import**.

The Import Workspace dialog box opens.

1-3-2. Click **Browse** next to the Import from Archive field and choose the **system_simulation.zip** file from the following directory (1):

\$TRAINING_PATH/system_simulation/support

1-3-3. Click **Select**.

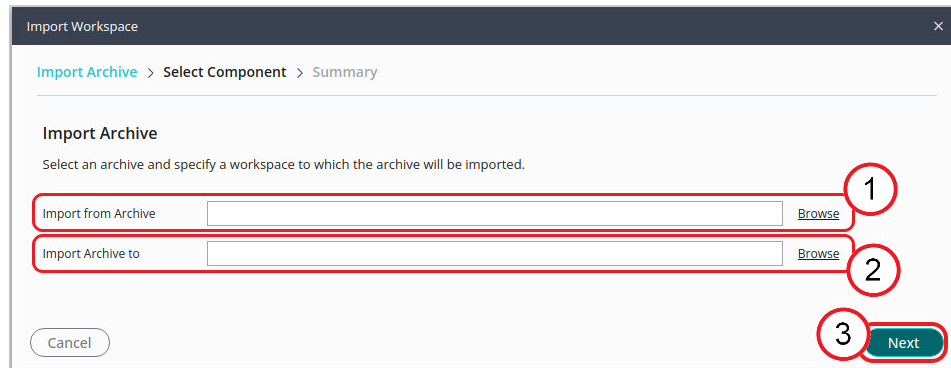


Figure 2-5: Selecting the Archive and Workspace Directories

The *Import Archive to* field will be set to the following directory by default (2):

\$TRAINING_PATH/system_simulation/lab/sys_project

1-3-4. Click **Next** (3).

1-3-5. Review the projects and components that are listed.

system_project should be selected by default.

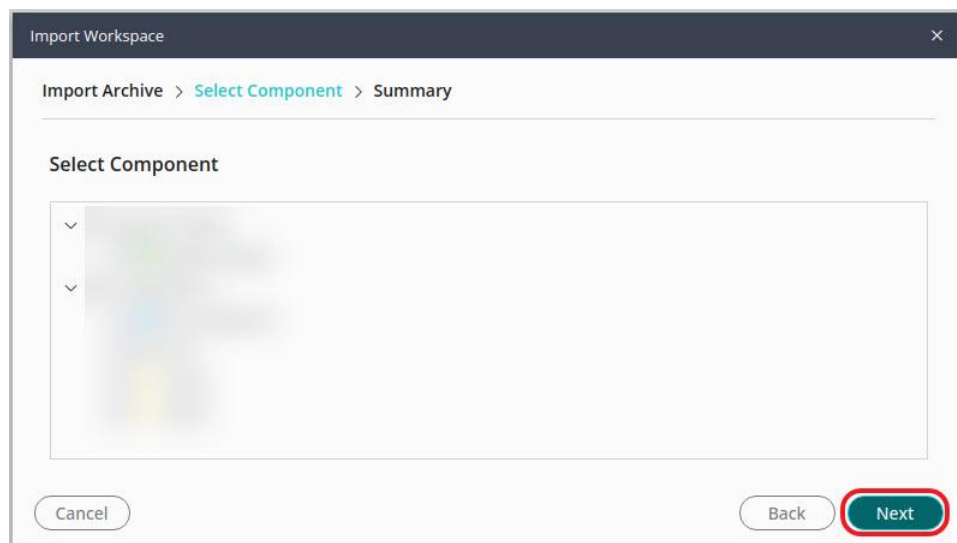


Figure 2-6: Selecting the Vitis Project Archive File

1-3-6. Click **Next**.

1-3-7. Review the Summary and click **Finish** to import the project.

Note: You may see a message stating that the components have absolute paths.

It may take a minute to complete importing the project. Once done, you should see the project files in the Vitis Unified IDE GUI.

Note: It may take about 5–6 minutes for the import to complete.

You should see a GUI similar to the one shown in the figure below.

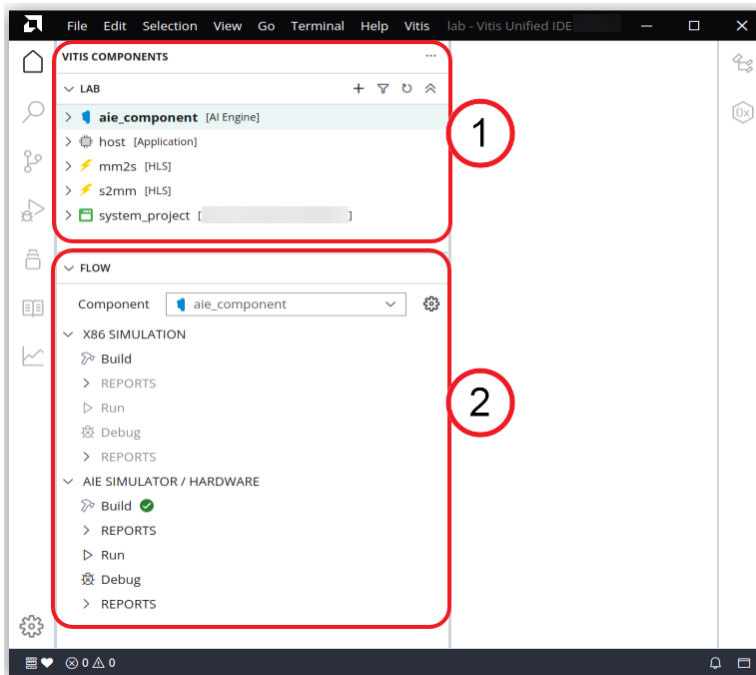


Figure 2-7: Vitis Unified IDE GUI for the Provided System Project

In the Vitis Components window, you can see the following components that have already been created (1):

- *aie_component*
 - This is an AI Engine component that contains the AI Engine kernel sources.
- *host*
 - This is an application component that contains sources for an application that runs on the processor (Arm® or x86 CPU) that loads and runs the device binary (.xclbin).
- *mm2s* and *s2mm*
 - These are HLS components for creating two PL kernels to load onto the device.
- *system_project*
 - This is a system project component where the different components that you have seen above (the AI Engine component, the application component, and the HLS components) are integrated into a single system.

The Flow window contains the tool flow options available for the corresponding component (2).

Reviewing the System Project

Step 2

You will now review the AI Engine system design for use on the VCK190 platform and Versal device.

2-1. Review the AI Engine component.

- 2-1-1. In the Vitis Components > Lab window, expand **aie_component [AI Engine]** > **Settings** (1).
- 2-1-2. Click the **vitis-comp.json** file to see the AI Engine component settings (2).
- 2-1-3. Review the component settings, such as the platform details, top-level file, and configuration file (3).

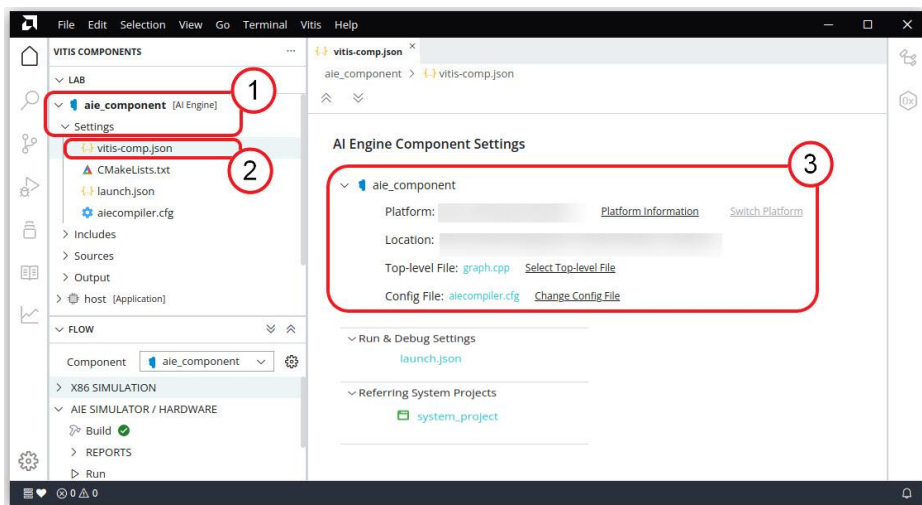


Figure 2-8: Reviewing the AI Engine Component Settings

- 2-1-4. In the Vitis Components window, expand **Sources > AIE > kernels** under the **aie_component [AI Engine]** folder.
- 2-1-5. Review the AI Engine kernel sources.

There are three AI Engine kernels used in this design:

- o interpolator
- o classifier
- o polarclip

Other than the kernel sources, the other two important files are `graph.h` and `graph.cpp`.

Filename	Type	Comment
graph.h	C++	Description of the AI Engine project graph. This graph contains two AI Engine kernels.
graph.cpp	C++	Test bench for AI Engine project testing.

Figure 2-9: Graph Files

2-2. Review the application component.

- 2-2-1. In the Vitis Components > Lab window, expand **host [Application]** > **Settings** (1).
- 2-2-2. Click the **vitis-comp.json** file to see the application component settings (2).
- 2-2-3. Review the platform details, domain, OS, selected processor, and sysroot path (3).

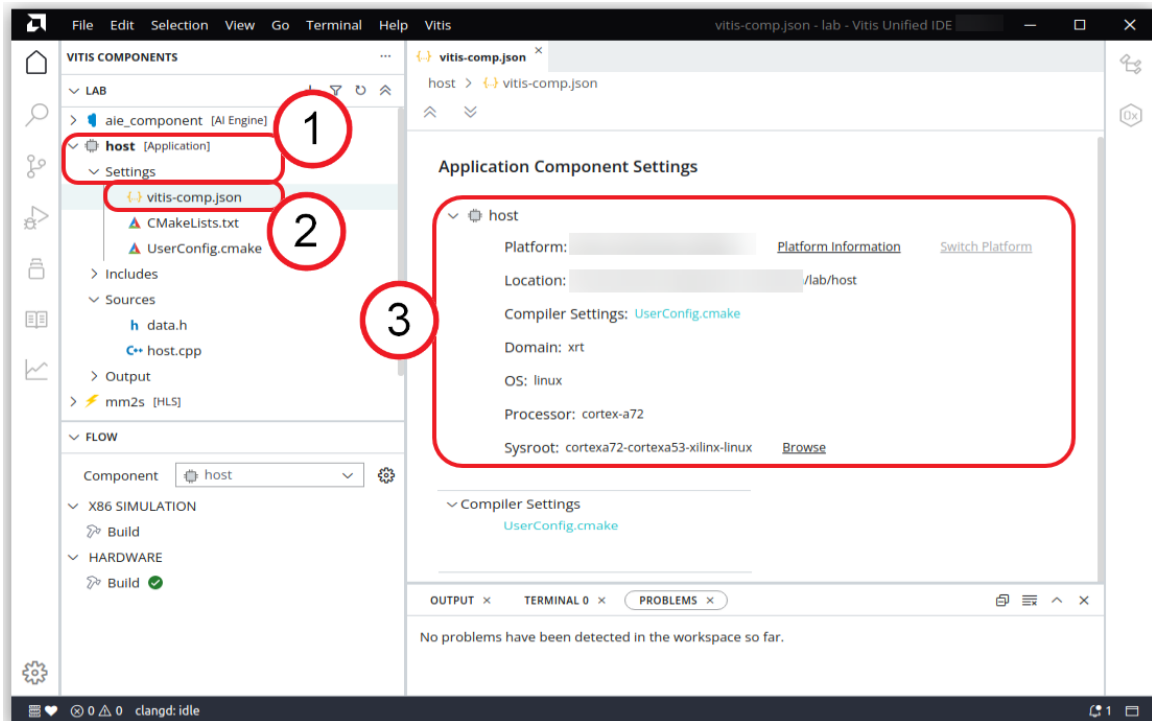


Figure 2-10: Reviewing the Application Component Settings

- 2-2-4. In the Vitis Components window, expand **Sources** under the host [Application] folder.
- 2-2-5. Click **host.cpp** to open the file.
- 2-2-6. Review the file and pay attention to the API calls and comments provided.
Note that this is the main PS application that controls the entire system.
- 2-2-7. Once you are ready, close the file.

2-3. Review the HLS components.

- 2-3-1. In the Vitis Components > Lab window, expand **mm2s [HLS]** > **Settings** (1).
- 2-3-2. Click the **mm2s_config.cfg** file to open this HLS configuration file in the editor tab (2).
- 2-3-3. Click the **Source Editor** icon to see the text form of the configuration file (3).
- 2-3-4. Review the contents of the configuration file (4).

This file currently contains the part (defined by the platform), flow target, package output, source CPP file, and other options that are defined for the HLS component.

The flow target is set to `vitis`, which will run the additional packing step to create `.xo` output as you will be using the HLS component in a system project and you will want a PL kernel (`.xo`) to be generated. Building this HLS component will generate the packaged RTL designs, which will have `.xo` extensions.

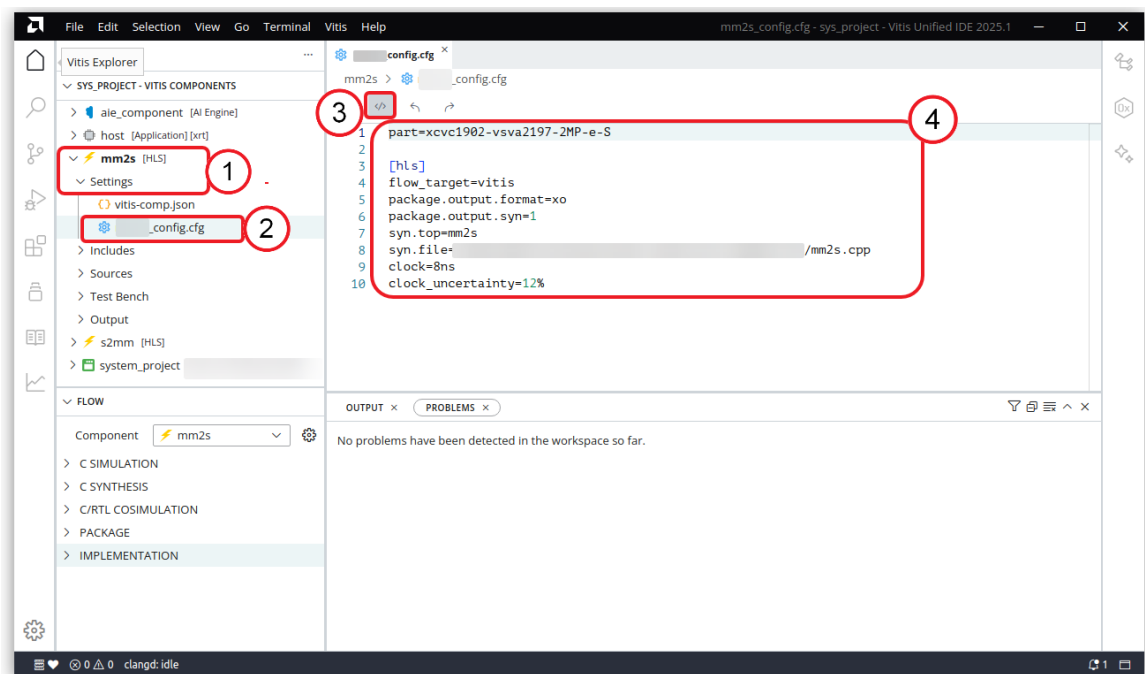


Figure 2-11: Reviewing the HLS Configuration File

- 2-3-5. Once you are ready, close the configuration file.
- 2-3-6. Similarly, review **s2mm [HLS]**, the other HLS component in the system.

2-4. Review the system project.

- 2-4-1. In the Vitis Components > Lab window, expand **system_project** [xilinx_vck190_base_202510_1] > **Settings** (1).
- 2-4-2. Click the **vitis-sys.json** file to see the system project settings (2).
- 2-4-3. Review the system project details, hardware linker settings, package settings, and component settings (3).

Note: You may already have these settings partially expanded in the Vitis Unified IDE GUI.

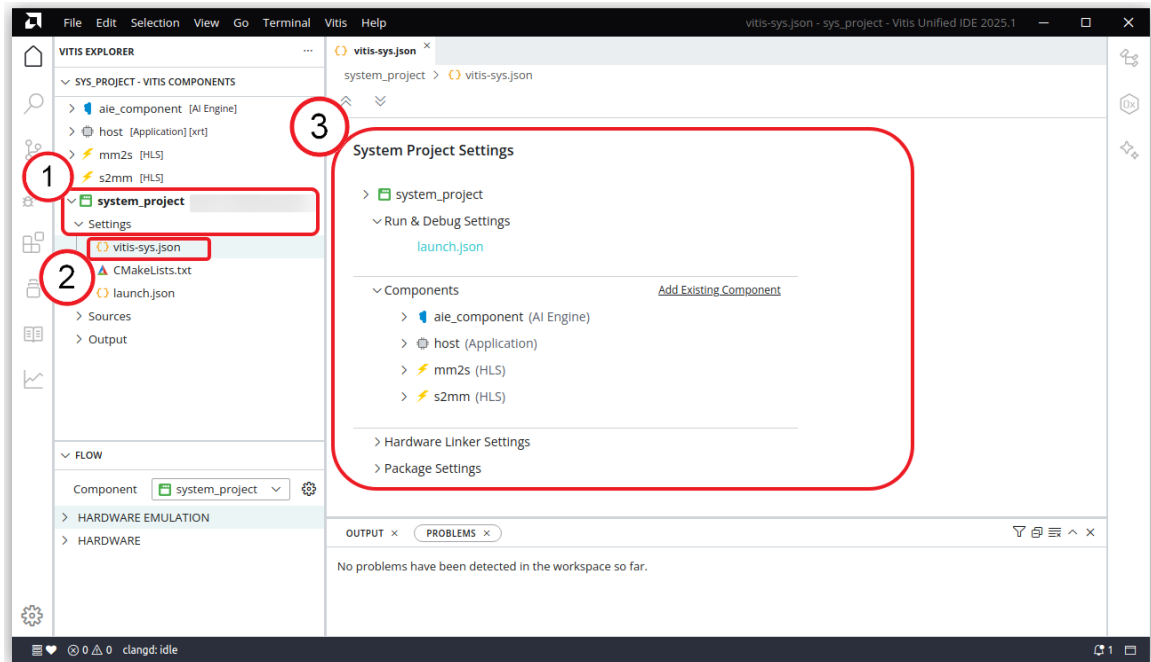


Figure 2-12: Reviewing the System Project Settings

2-5. Review the hardware linker settings.

- 2-5-1. Expand **Hardware Linker Settings** from the System Project Settings if necessary.
- 2-5-2. Expand **binary_container_1** under Hardware Linker Settings > BINARY CONTAINERS (1).
- 2-5-3. Observe that the HLS components (*mm2s* and *s2mm*) are added under KERNELS and that the AI Engine component (*aie_component*) is added under AIE GRAPHS (2).

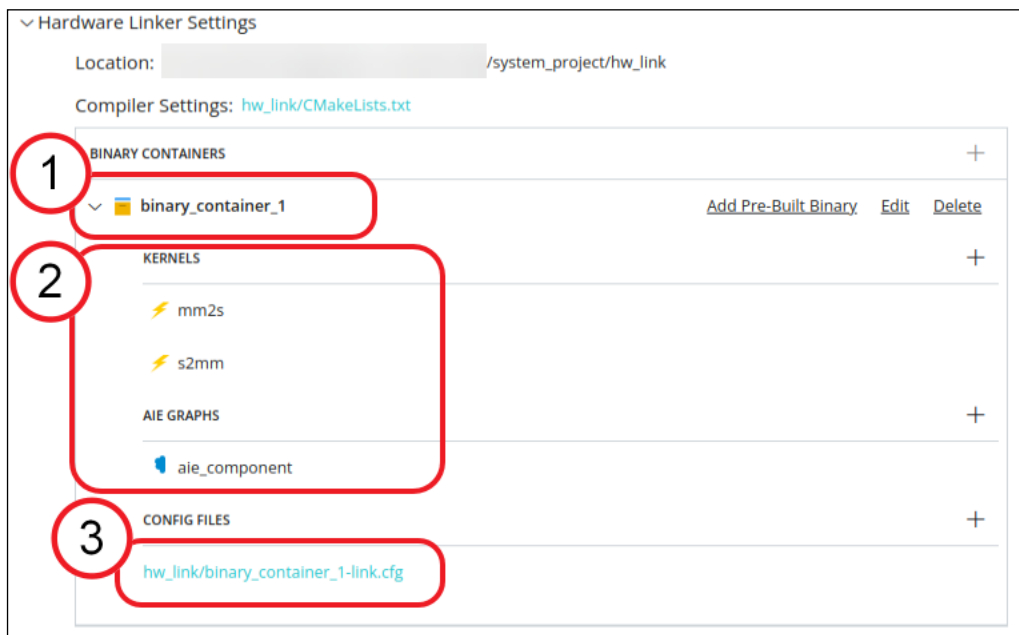


Figure 2-13: Reviewing the Hardware Linker Settings

- 2-5-4. Click the **hw_link/binary_container_1-link.cfg** link to open this configuration file in the editor tab (3).

This configuration file contains commands that determine how the hw_link process will run and how the device binary will be generated. The configuration file editor tab opens and displays the v++ linker settings. Look through the settings to see what is available.

- 2-5-5. Click the **Source Editor** icon to see the text form of the configuration file.
- 2-5-6. Review the connectivity lines as shown below:

```
[connectivity]
nk=mm2s:1:mm2s
nk=s2mm:1:s2mm
sc=mm2s.s:ai_engine_0.DataIn1
sc=ai_engine_0.DataOut1:s2mm.s
```

These configuration commands tell the linker how to connect the AI Engine array to PL and vice versa.

A complete description of the syntax can be found in the *AI Engine Tools and Flows User Guide* (UG1076).

Some of the switches that can be used for system configuration are shown below.

Switch	Description
nk	Number of kernels mm2s:2:mm2s_1:mm2s_2 means that the Vitis compiler should instantiate two mm2s kernel and name the instance 'mm2s_1' and 'mm2s_2'. By default, there is a single instance named 'mm2s_1'.
stream_connect or sc	How the kernels will connect to IPs, platforms, or other kernels. The output of the AI Engine compiler will tell you the interfaces that need to be connected. mm2s_1.s:ai_engine_0.DataIn1 means that the Vitis compiler should connect the port 's' of 'mm2s_1' to the port 'DataIn1' of AI Engine project 0. The name of the AI Engine port is the one that has been defined in the graph.cpp PLIO instantiation or in the graph.h if no PLIO has been used.

Figure 2-14: Connectivity Details

2-5-7. Once you are ready, close the **binary_container_1-link.cfg** file.

2-6. Review the package settings.

2-6-1. Expand **Package Settings** from the System Project Settings if necessary (1).

2-6-2. Review the compiler settings, common Linux kernel image, and root file system path.

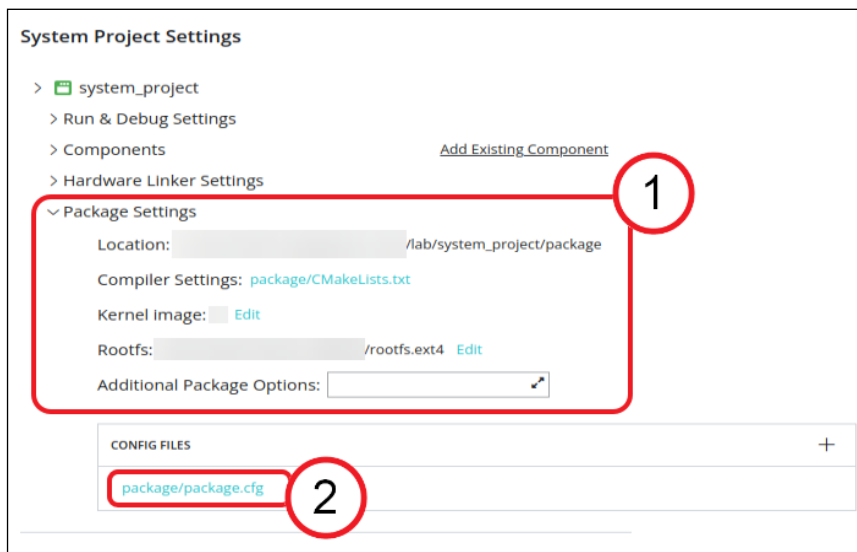


Figure 2-15: Reviewing the Packaging Settings

2-6-3. Click the **package/package.cfg** link to open this configuration file in the editor tab (2).

2-6-4. Click the **Source Editor** icon to see the text form of the configuration file.

2-6-5. Review the configuration details for the Linux boot and AI Engine settings.

Note: You will perform the packaging step using the command line interface in the next step. In that step, you will use a similar type of configuration settings.

2-6-6. Once you ready, close the **package.cfg** file.

2-7. Review the Components section.

- 2-7-1. Under the Components section, verify that the existing system components, such as *aie_component*, *host*, *mm2s*, and *s2mm*, are included.

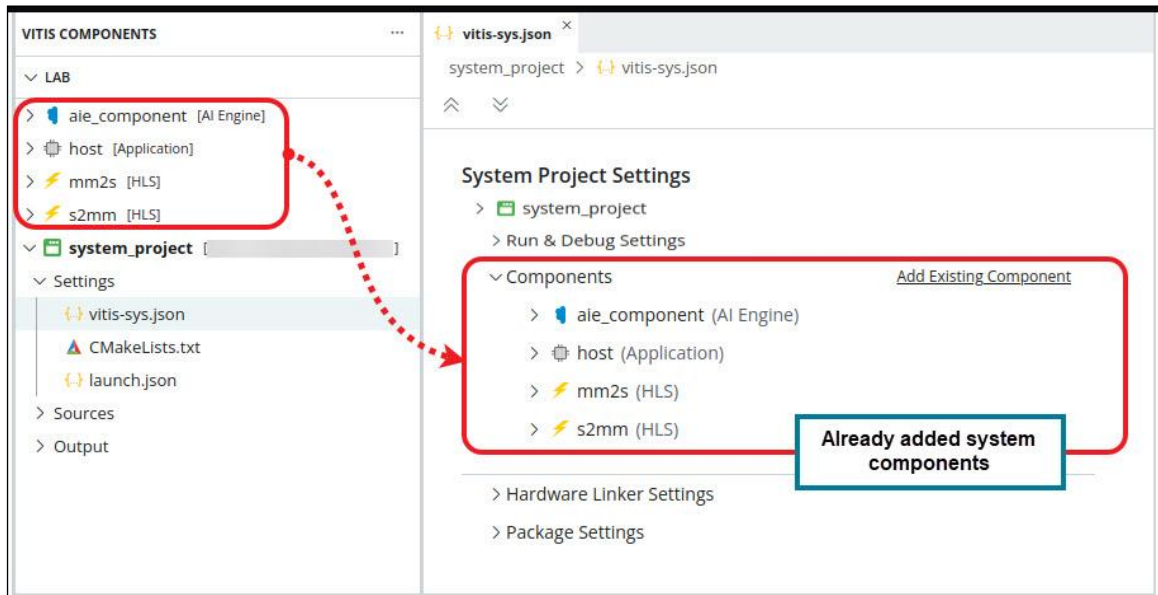


Figure 2-16: Reviewing the Added Components

2-8. Once you are ready, close the Vitis Unified IDE.

- 2-8-1. Select **File > Close Window** to close the tool.

Running the System-Level Hardware Emulation

Step 3

You will now run hardware emulation to verify the complete system by using prebuilt files.

Note: Compiling and building the whole system design will take around 1-2 hours and hence this is out of scope for this lab. You just reviewed the project files in the last step (without building), and you will now run the hardware emulation in this step by using prebuilt files.

The following prebuilt files are provided:

- AIE Engine kernels: `libadf.a`
- PL kernels: `binary_container_1.xsa`
- Host application: `host`

These files are in the following directory:

`$TRAINING_PATH/system_simulation/lab/prebuilt`

3-1. Set up the Vitis Unified IDE environment.

3-1-1. Press <Ctrl + Alt + T> to open a new terminal window.

3-1-2. Enter the following command to source the Vitis tools:

```
[host]$ source /opt/amd/2025.1/Vitis/settings64.sh
```

Note: The customer training environment (CustEd_VM) sets the Vitis tool install path to /opt/amd/2025.1/Vitis. If the tool is installed in a different location in your environment, use that install path.

3-2. Package the SD card with the required files by using the v++ package command (v++ -p or v++ --package).

After compiling and linking your kernel code to build the .xclbin, you need to package the device binary (along with any required supporting files) to build a package that can be run for software or hardware emulation or that can be booted and run on the hardware device.

The v++ --package step, or -p, packages the final product at the end of the v++ compile and link process. This is a required step for all Versal platforms, including AI Engine platforms and embedded processor platforms.

3-2-1. Enter the following command to change the directory to the prebuilt directory:

```
[host]$ cd $TRAINING_PATH/system_simulation/lab/prebuilt
```

3-2-2. Enter the following command to review the GeneratePackage.sh file:

```
[host]$ gedit GeneratePackage.sh
```

The --package command has a range of options for use with the different platforms and build targets supported by the Vitis tools. Here are the options used:

- o --config package.cfg

- This configuration file contains the following details:

- platform: Specifies the target board.
 - target: hw_emu (hardware emulation): Specifies the emulation [sw_emu | hw_emu | hw].
 - kernel_image: Path to the kernel image location.
 - rootfs: Path to the rootfs location.
 - out_dir: Output directory named package where the package command output will be generated.
 - sd_file: Host application.

- o --package.defer_aie_run

- This is to make the AI Engines enabled by the PS application. When unset, the AI Engines will be enabled during the PDI load.

- o libadf.a
 - Compiled output of the AI Engine kernels.
- o binary_container_1.xsa
 - Compiled output of all the PL kernels.
- o -o binary_container_2.xclbin
 - Specifies the output file generated by the v++ command.

3-2-3. Close the file.

3-2-4. Enter the following command to run the Vitis tool package command:

```
[host]$ ./GeneratePackage.sh
```

Note: This may take approximately 1-2 minutes to complete.

3-3. Run hardware emulation using the launch_hw_emu.sh generated script.

3-3-1. Enter the following to change the directory to the generated package directory:

```
[host]$ cd package
```

3-3-2. Enter the following command to launch hardware emulation with the Vivado simulator GUI:

```
[host]$ ./launch_hw_emu.sh -g
```

Note: Adding the -g option invokes the Vivado simulator GUI.

This will launch QEMU and, after some time, the Vivado simulator will launch.

Note: This will take approximately 2-3 minutes.

3-3-3. Go to the Vivado simulator.

3-3-4. Expand **vitis_design_wrapper_sim_wrapper > vitis_design_wrapper_i > vitis_design_i** in the Scope window on the left (1).

You may want to remove all the automatically added signals from the wave window.

3-3-5. Select the **Search** field in the Objects window and enter **ai** to find AI Engine interface signals (2).

3-3-6. Select all the AI Engine interface signals from the Objects window (3).

3-3-7. Right-click and select **Add to Wave Window** to move the signals to the waveform viewer area (4).

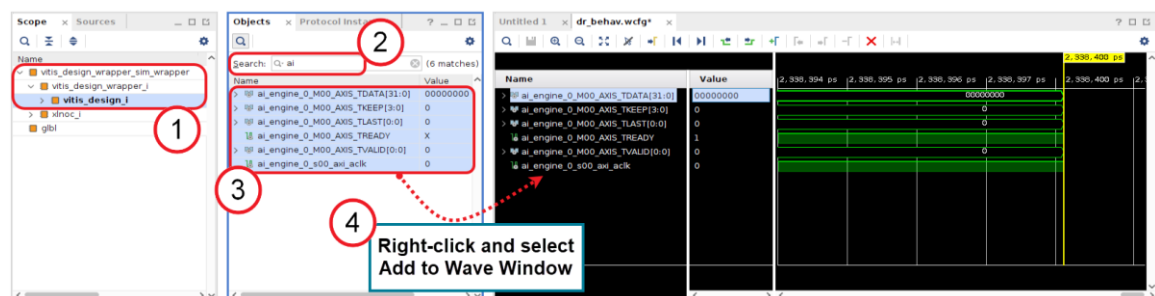


Figure 2-17: Adding the AI Engine AXIS Interface Signals to Wave Window

3-3-8. Set the simulation time to **160 us**.

3-3-9. Click the **Run For** icon to run the simulation.

The simulation in the Vivado Design Suite starts, and you will have to switch to the Vitis terminal window to see the boot log.

Note: If you do not get the desired output after 160 us as described in the steps below ("TEST PASSED" in the Vitis terminal window), run the Vivado simulation for a further 50 us to get the output.

This may happen due to a possible time lag (latency) in transferring files between the tools if your connection is not good enough.

3-3-10. Switch back to the Vitis terminal window and observe the log.

Notice that Linux booting starts in the terminal. It will take approximately 4-5 minutes to complete the Linux booting. Once the booting completes, you will need to run the host application.

3-3-11. Once booting is done, observe the command prompt in the Vitis terminal window.

3-3-12. Enter the following command to run the host application:

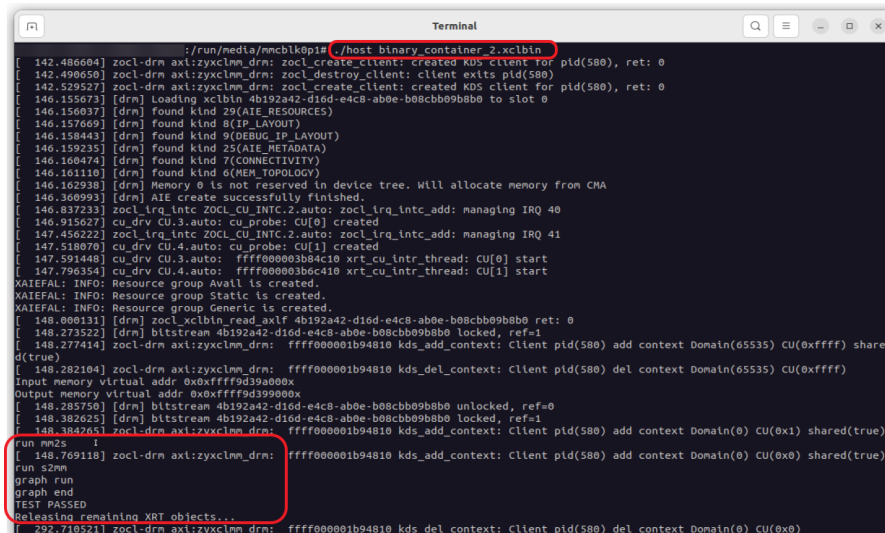
```
versal-rootfs-common-20251:/mnt# ./host
binary_container_2.xclbin
```

You should see that the host application starts running. Observe that the kernels are launched by the host program in the terminal.

You will notice that "Graph run" runs for longer.

This will take approximately 5-8 minutes to complete. You should see the "TEST PASSED" message as shown below.

Known issue: This lab requires more memory to run QEMU, the AIE emulator, and the Vivado simulator. In some situations, you may see the "TCF agent failed" or "Failed to run aie initialization script ..." messages. If so, stop the QEMU emulator, close the Vitis tool, and relaunch the tool again.



```

[ 142.486604] zocl-drm axl:zyxclmw_drm: zocl_create_client: created KDS client for pid(580), ret: 0
[ 142.490650] zocl-drm axl:zyxclmw_drm: zocl_destroy_client: client exits pid(580)
[ 142.529529] zocl-drm axl:zyxclmw_drm: zocl_create_client: created KDS client for pid(580), ret: 0
[ 146.155673] [drm] Loading xclbin 4b192a42-d16d-e4c8-ab0e-b08cbb09b8b0 to slot 0
[ 146.156037] [drm] found kind 29(AIE_RESOURCES)
[ 146.157069] [drm] found kind 8(IP_LAYOUT)
[ 146.158443] [drm] found kind 9(OEBUG_IP_LAYOUT)
[ 146.159235] [drm] found kind 25(AIE_METADATA)
[ 146.160474] [drm] found kind 7(CONNECTIVITY)
[ 146.161110] [drm] found kind 6(MEM_TOPOLOGY)
[ 146.162930] [drm] Memory 0 is not reserved in device tree. Will allocate memory from CMA
[ 146.360993] [drm] AIE create successfully finished.
[ 146.837233] zocl_irq_intc zocl_cu_intc.2.auto: zocl_irq_intc_add: managing IRQ 40
[ 146.915627] cu_drv CU.3.auto: cu_probe: CU[0] created
[ 147.450222] zocl_irq_intc zocl_cu_intc.2.auto: zocl_irq_intc_add: managing IRQ 41
[ 147.518070] cu_drv CU.4.auto: cu_probe: CU[1] created
[ 147.591448] cu_drv CU.3.auto: ffff000003b84c10 xrt_cu_intr_thread: CU[0] start
[ 147.796354] cu_drv CU.4.auto: ffff000003b84c10 xrt_cu_intr_thread: CU[1] start
XAIEFAL: INFO: Resource group Avail is created.
XAIEFAL: INFO: Resource group Static is created.
XAIEFAL: INFO: Resource group Generic is created.
[ 148.000131] [drm] zocl_xclbin_read_axlf 4b192a42-d16d-e4c8-ab0e-b08cbb09b8b0 ret: 0
[ 148.273522] [drm] bitstream 4b192a42-d16d-e4c8-ab0e-b08cbb09b8b0 locked, ref=1
[ 148.277414] zocl-drm axl:zyxclmw_drm: ffff000001b94810 kds_add_context: Client pid(580) add context Domain(65535) CU(0xffff) share
d(true)
[ 148.282104] zocl-drm axl:zyxclmw_drm: ffff000001b94810 kds_del_context: Client pid(580) del context Domain(65535) CU(0xffff)
Input memory virtual addr 0x0ffff9d39a000x
Output memory virtual addr 0x0ffff9d399000x
[ 148.285750] [drm] bitstream 4b192a42-d16d-e4c8-ab0e-b08cbb09b8b0 unlocked, ref=0
[ 148.382625] [drm] bitstream 4b192a42-d16d-e4c8-ab0e-b08cbb09b8b0 locked, ref=1
[ 148.384265] zocl-drm axl:zyxclmw_drm: ffff000001b94810 kds_add_context: Client pid(580) add context Domain(0) CU(0x1) shared(true)
run
[ 148.769118] zocl-drm axl:zyxclmw_drm: ffff000001b94810 kds_add_context: Client pid(580) add context Domain(0) CU(0x0) shared(true)
run s2mm
graph run
graph end
TEST PASSED
Releasing remaining XRT objects...
[ 292.710521] zocl-drm axl:zyxclmw_drm: ffff000001b94810 kds_del_context: Client pid(580) del context Domain(0) CU(0x0)

```

Figure 2-18: Hardware Emulation - Test Passed

Rerun the simulation if it fails because of a timeout error. This issue can sometimes occur as a result of high latency (lag) between the host and the target.

Observe in the Vivado simulator window that the test passes after ~62 us. This timing varies based on when the simulator starts and when the host application starts.

Note: If the Vivado simulator completes before the program runs completely, you may have to run the simulation for a further 50 us. Otherwise, the host application waits for the simulation to run.

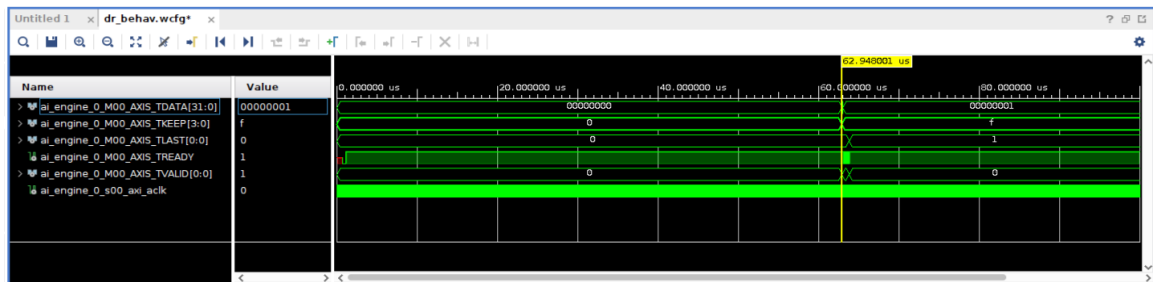


Figure 2-19: Waveform Showing the Interfaces of the AI Engine

Note: M00_AXIS_TDATA is the output from the classifier kernel.

In this way you can perform hardware emulation of a system design comprising of AI Engine, PS, and PL resources.

3-4. Close the tools.

3-4-1. Close the Vivado simulator and Vitis terminal window.

You may possibly face issues when trying to close the simulator. Make sure that you stop the simulation first if it is running at the back end.

Some systems (particularly VMs) may be memory constrained. Removing the workspace frees a portion of the disk space, allowing other labs to be performed.

You can delete the directory containing the lab you just ran by using the graphical interface or the command-line interface. You can choose either mechanism. Both processes will recursively delete all the files in the \$TRAINING_PATH/system_simulation directory.

3-5. [Optional] [Only for local VMs—not for CloudShare] Clean up the file system.

Using the GUI:

3-5-1. Navigate to \$TRAINING_PATH/system_simulation.

3-5-2. Select system_simulation.

3-5-3. Press <Delete>.

-- OR --

[Linux users]: Using the command line:

3-5-4. Press <Ctrl + Alt + T> to open a terminal window.

3-5-5. Enter the following command to delete the contents of the workspace:

```
[host] $ rm -rf $TRAINING_PATH/system_simulation
```

Summary

In this lab, you reviewed system design kernels running on the AI Engine and PL. You validated a host program (PS) targeting the Linux OS, which controls the system flow, and you validated the system design with hardware emulation.

