Documentation Report on

IP Integration in IP-XACT Format using Kactus2

By-

Aditya Dhiman	Abhijeet Chauhan
2022597	2022017
aditya22597@iiitd.ac.in	abhijeet22017@iiitd.ac.in

Introduction

In modern electronic design workflows, managing and integrating Intellectual Property (IP) cores efficiently is critical for reducing development time and ensuring design consistency. The **IP-XACT standard**, defined by the IEEE 1685 specification, provides a structured XML-based format for describing IP and its metadata, enabling better interoperability and reusability across tools and design environments.

This project leverages **Kactus2**, an open-source tool designed for IP management and system design, to facilitate the integration of IP components within a streamlined design process. By utilizing Kactus2, we aim to exploit its capabilities for creating, editing, and integrating IP-XACT compliant metadata, thereby improving design modularity and enabling seamless integration into larger systems-on-chip (SoC) architectures.

This documentation provides a comprehensive overview of the methodologies and processes employed in this project, focusing on the integration workflow, challenges encountered, and the solutions implemented to ensure compliance with IP-XACT standards. It also highlights how Kactus2 has been utilized to enhance productivity and design integrity within the context of this IP integration initiative.

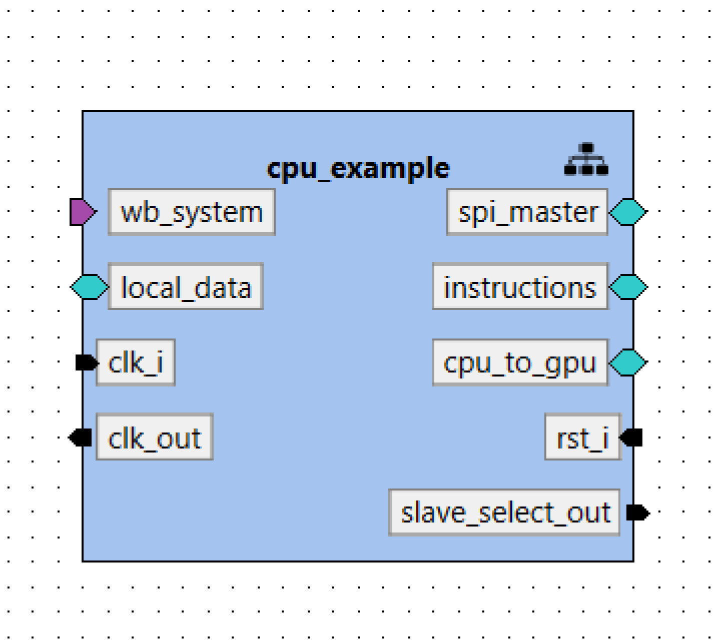
Goal

To construct a complicated system-on-a-chip (SoC) design that uses bus interconnects, multiple CPUs, main memory, and memory controller.

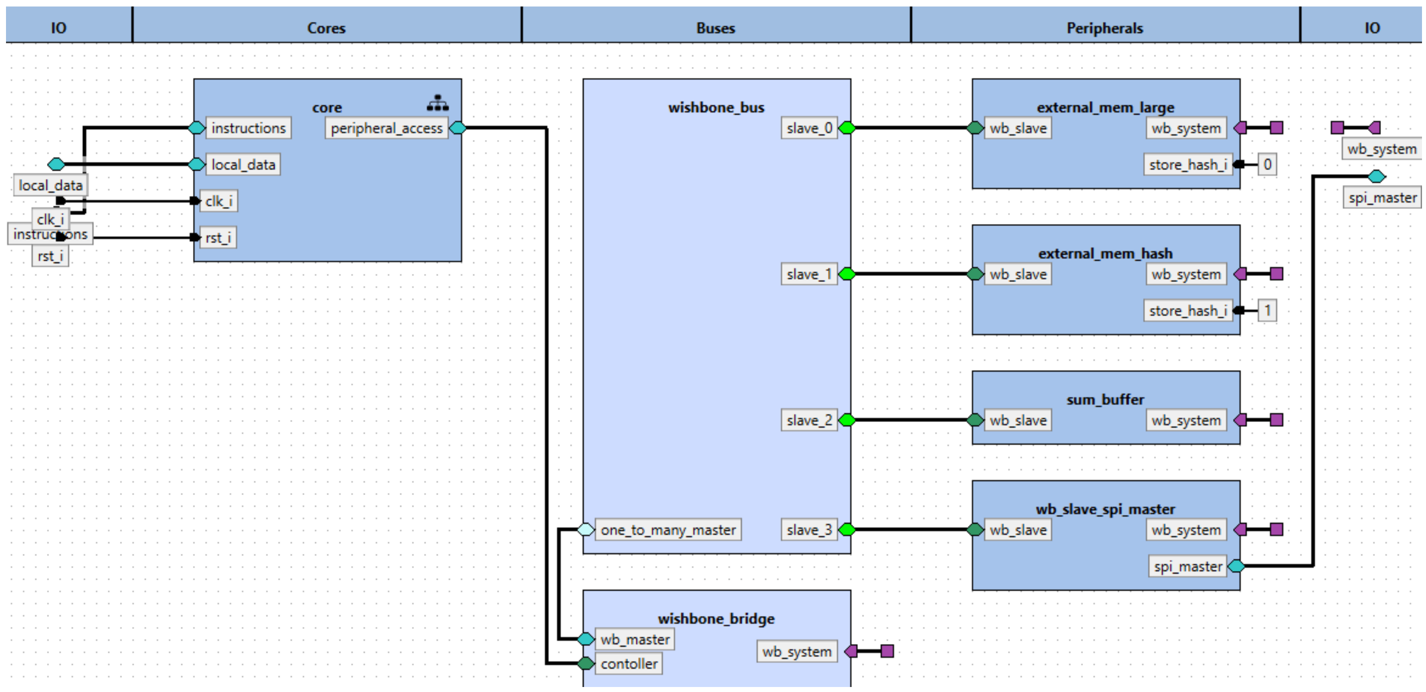
Components Used

1. CPU example:

Place the library from https://github.com/kactus2/ipxactexamplelib into the Kactus2 directory. It includes a hardware component called the CPU example, which features ports such as *reset*, *clock*, *slave_select_out*, and bus interfaces like *spi_master* and *instructions*, among others.

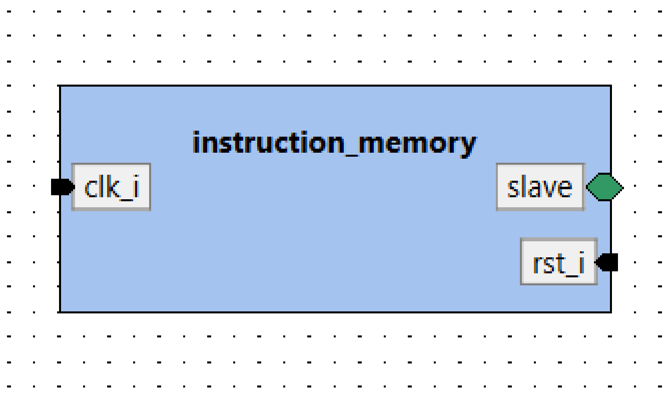


Internal hardware design of CPU example:



The CPU Example IPs can be utilized in any quantity necessary during the design of an architecture to create a processing stream. In the implemented architecture, 15 CPU examples have been incorporated within a processing cluster. The local data bus interface is connected to the data memory, and the instructions bus interface is connected to the instruction memory block, the wishbone system bus interface is present in the CPU system to have a clock and reset signals only, and lastly spi_master bus interface is present to give an output data to the surrounding blocks and receive an input to process it as well.

2. Instruction Memory:



Purpose:

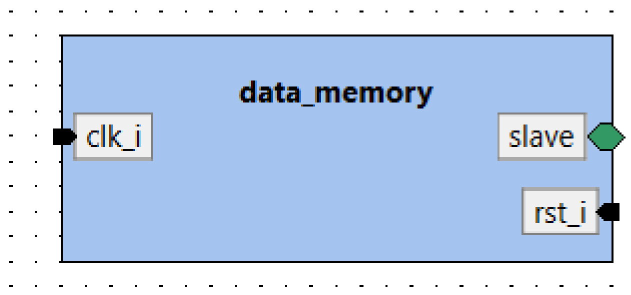
- Stores the program instructions that the CPU needs to execute.
- Typically read-only or less frequently modified during operation (e.g., when loading a new program).

Importance:

- **Facilitates program execution**: Without instruction memory, the processor wouldn't know what operations to perform.
- **Isolates code from data**: Separating instructions from data improves system organization and reliability.
- **Supports program loading**: Allows multiple programs to be loaded and executed sequentially.
- **Optimized for speed**: Designed for fast read access since the CPU fetches instructions in real-time during execution.

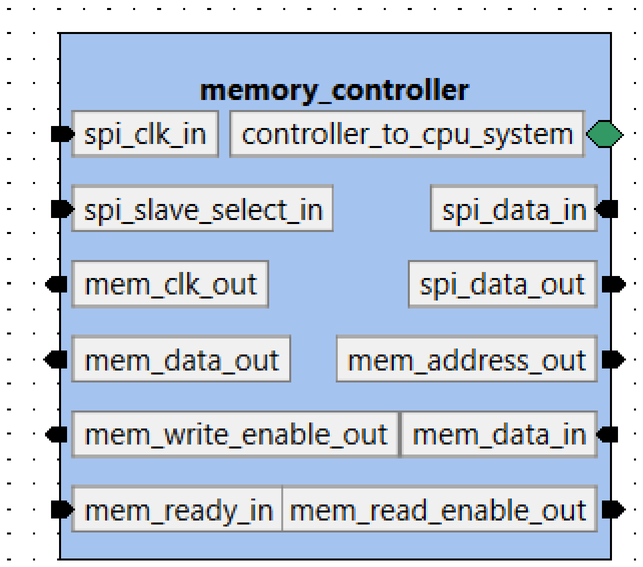
This Slave bus(Local memory abstraction bus) is here so that it can take care of the operations like read data and address, which means it can help CPU read data from a particular address from cpu example IP, and can provide necessary instructions that are to be performed by the cpu system.

3. Data Memory



1. **Purpose**:
    - Stores variables, temporary data, and the results of computations during program execution.
    - Read/write memory allows data to be dynamically updated.
2. **Importance**:
    - **Enables data storage**: Programs need a place to store and retrieve data as they execute operations.
    - **Supports dynamic operations**: Data memory allows for real-time adjustments and computations, which are crucial for applications like gaming, simulations, and AI.
    - **Separates concerns**: Isolating data memory from instruction memory enhances security and stability.
    - **Temporary storage**: A workspace for ongoing computations (e.g., registers, stack, and heap).

4. Memory Controller:



This hardware component was generated using 2 files ie. verilog and XML of the memory controller.

Prompt to generate a Verilog implementation of a memory controller on gpt "Design a memory controller that connects a CPU system to the main memory. The interface between the CPU system and the memory controller should use an SPI bus. The memory controller must support basic memory read and write operations and should be compatible with standard memory modules. Provide a detailed description of the architecture and functionality.". This Verilog file was fed into the EDA Utilis tool named verilog2ipxact for XML extraction.

Path of the Verilog and XML files in kactus2:

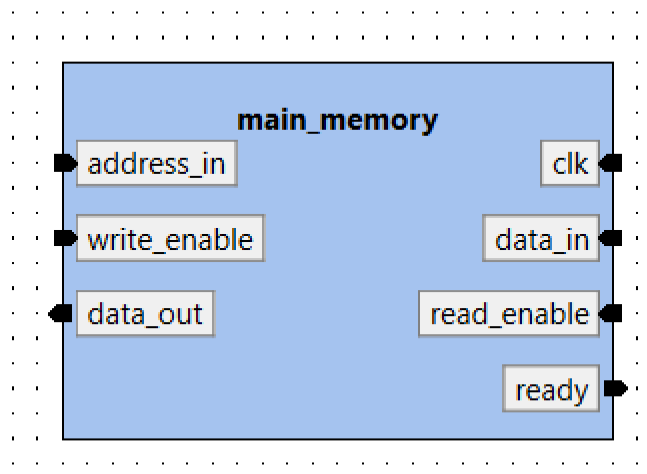
Vendor: tut.fi

Library: communication.template

Name: memory_controller

Version: 1.0

4. Main Memory:



Prompt given for the main memory was “Design a main memory system that interfaces with a memory controller. The memory system should support basic read and write operations and be compatible with an SPI-based memory controller. Provide a detailed description of its architecture, functionality, and the protocols it uses for seamless communication with the controller.”

Path of the Verilog and XML files in kactus2:

Vendor: tut.fi

Library: GPU

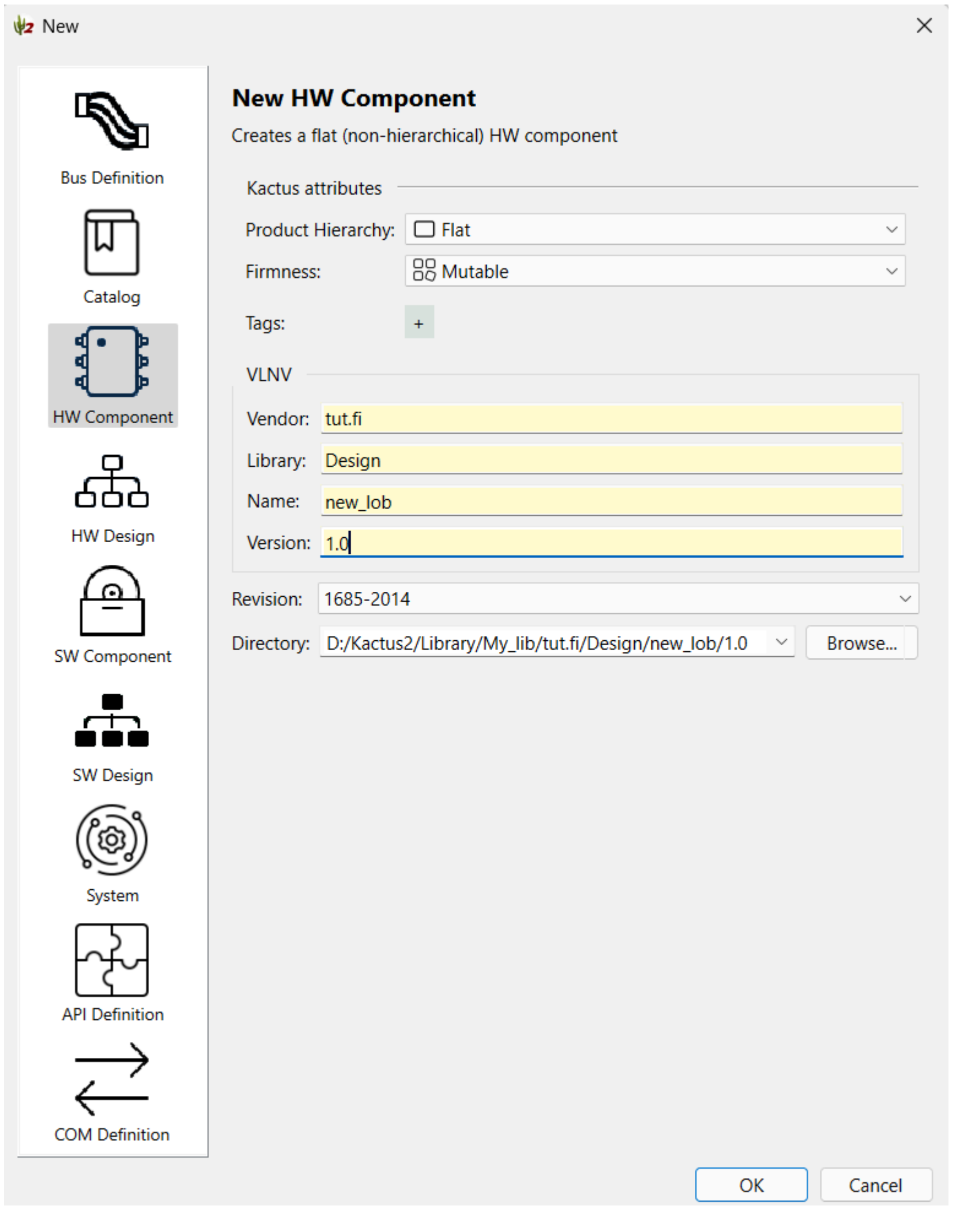
Name: memory_controller

Version: 1.0

Here all the logical ports of SPI bus interface are used MISO(Master In Slave Out), MOSI(Master Out Slave In), SS(Slave Select)

Creating hardware Design:

Step 1. Create a new Hardware component and describe the VLNV for the same



New HW Component

Creates a flat (non-hierarchical) HW component

Kactus attributes

Product Hierarchy: Flat

Firmness: Mutable

Tags: +

VLNV

Vendor: tut.fi

Library: Design

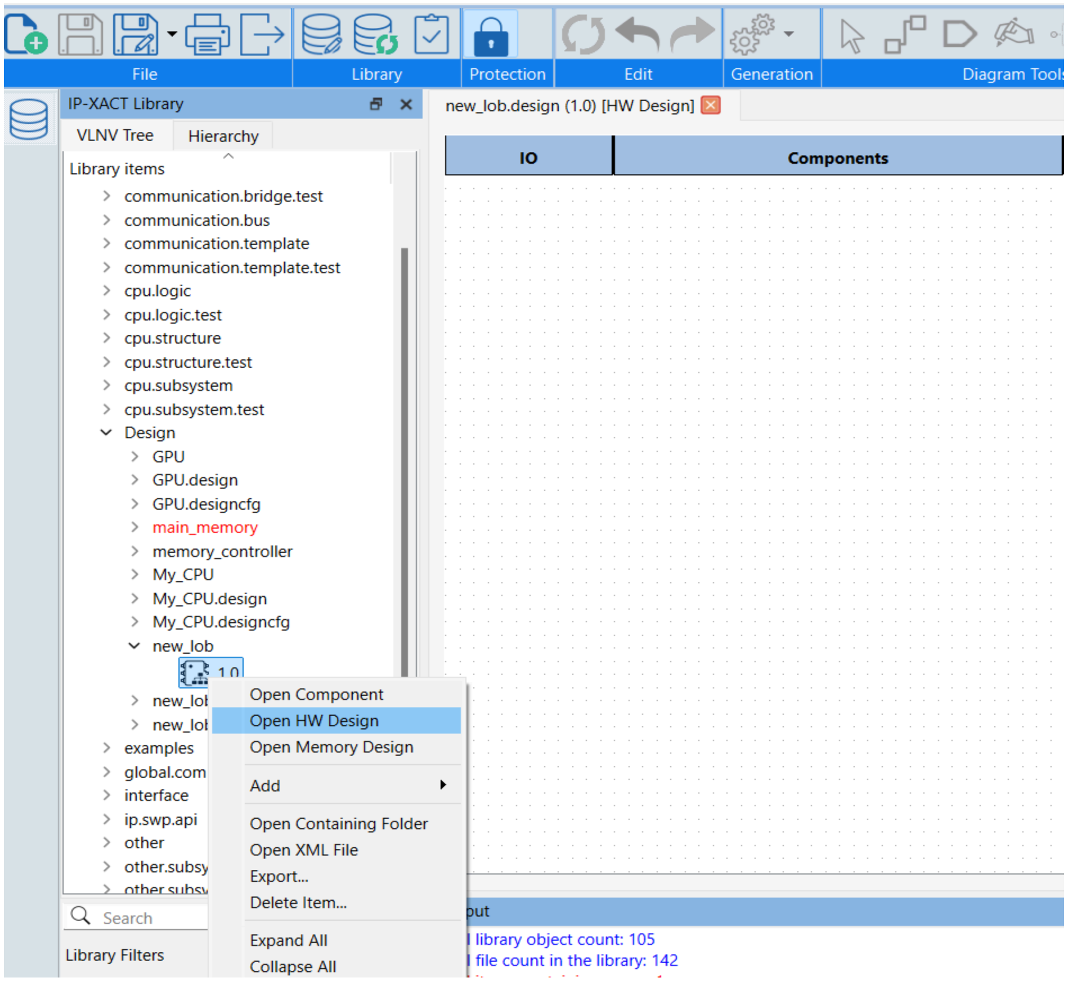
Name: new_lob

Version: 1.0

Revision: 1685-2014

Directory: D:/Kactus2/Library/My_lib/tut.fi/Design/new_lob/1.0

Step 2. Add a new hardware design with the same VLNV (Vendor, Library, Name, Version) as the hardware component. Remove any protection or restrictions applied to the design to enable modifications. Once unprotected, add the necessary components by dragging them to the hardware design.

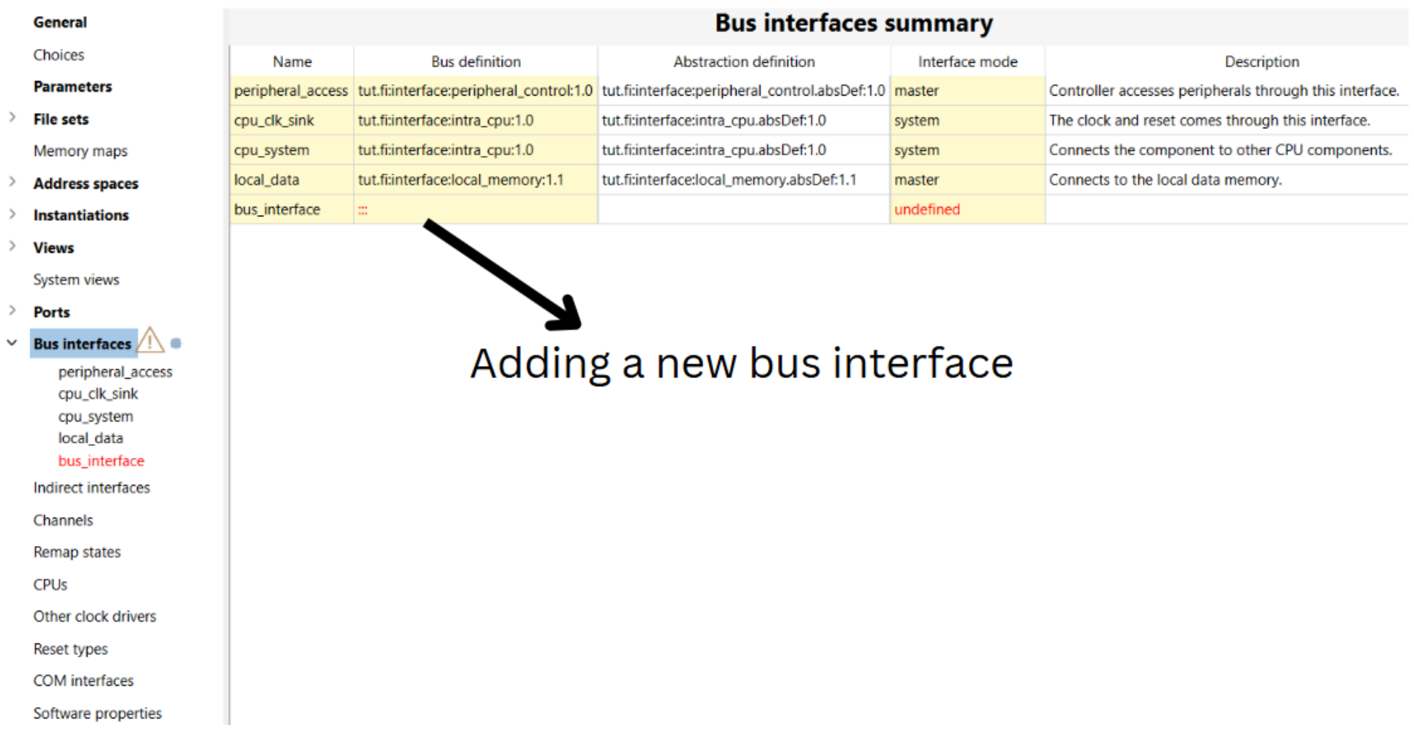


```
l library object count: 105
l file count in the library: 142
```

Establishing a bus interface :

Step 1. Open the hardware component of the required IP and open the protection.

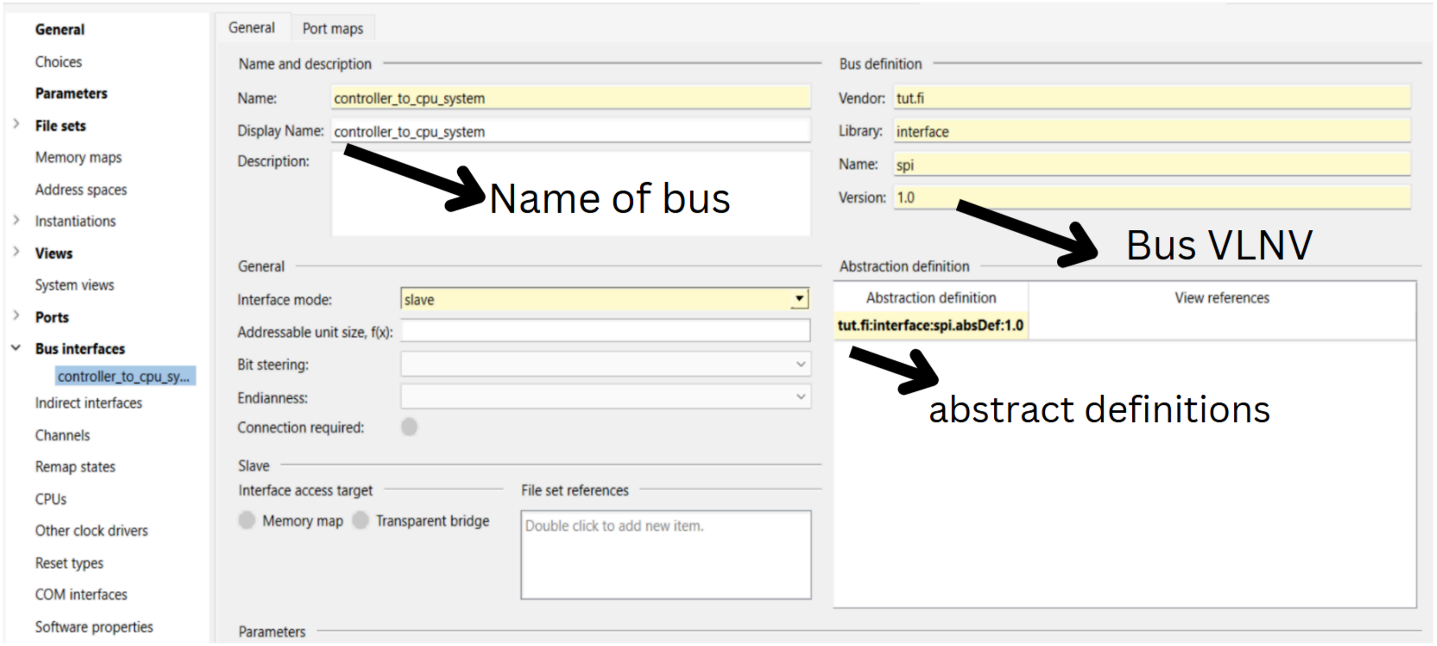
Step 2. Go to the bus interface and double-click on the bus interface summary to add a row



Bus interfaces summary

Name	Bus definition	Abstraction definition	Interface mode	Description
peripheral_access	tut.fi:interface:peripheral_control:1.0	tut.fi:interface:peripheral_control.absDef:1.0	master	Controller accesses peripherals through this interface.
cpu_clk_sink	tut.fi:interface:intra_cpu:1.0	tut.fi:interface:intra_cpu.absDef:1.0	system	The clock and reset comes through this interface.
cpu_system	tut.fi:interface:intra_cpu:1.0	tut.fi:interface:intra_cpu.absDef:1.0	system	Connects the component to other CPU components.
local_data	tut.fi:interface:local_memory:1.1	tut.fi:interface:local_memory.absDef:1.1	master	Connects to the local data memory.
bus_interface	:::		undefined	

Step 3. Go to the general properties of the specific bus and provide the VLNV (Vendor, Library, Name, Version) for the same, also the VLNV for the abstract definition.



Name and description

Name: controller_to_cpu_system

Display Name: controller_to_cpu_system

Description:

Bus definition

Vendor: tut.fi

Library: interface

Name: spi

Version: 1.0

General

Interface mode: slave

Addressable unit size, f(x):

Bit steering:

Endianness:

Connection required:

Abstraction definition

Abstraction definition	View references
tut.fi:interface:spi.absDef:1.0	

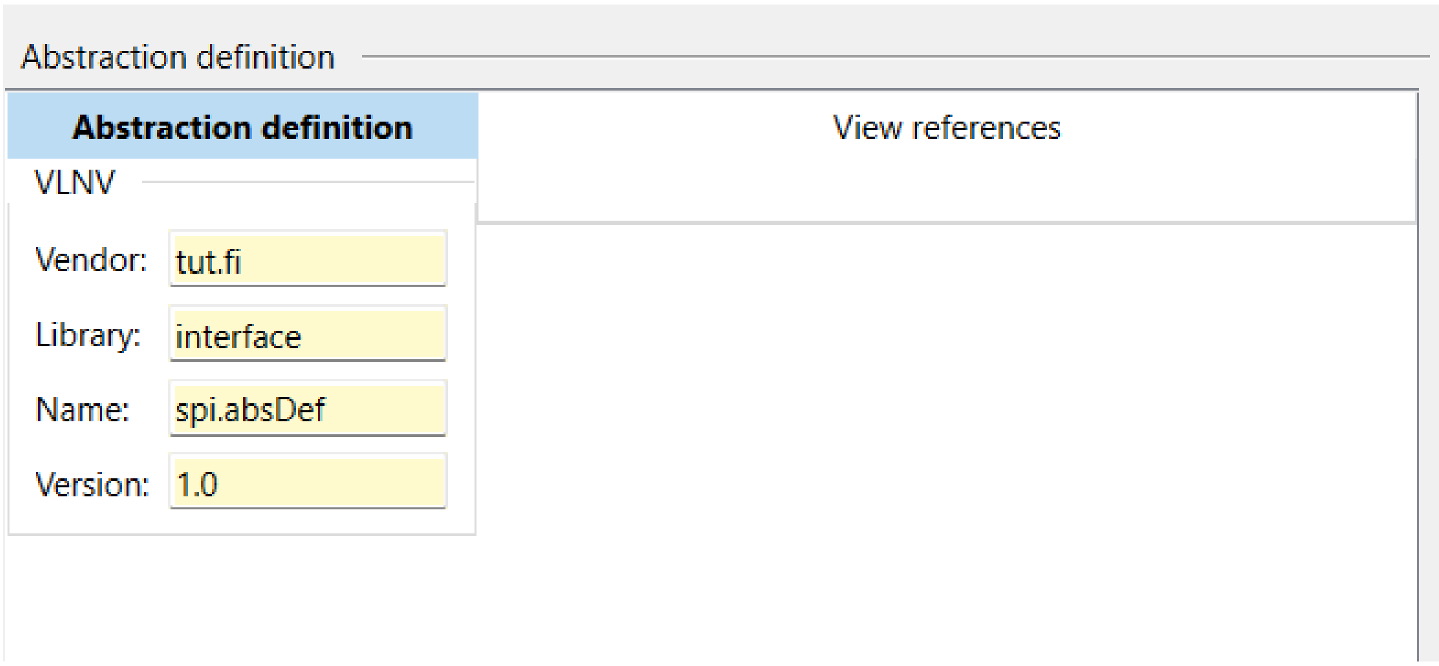
Slave

Interface access target

Memory map Transparent bridge

File set references

Parameters



Abstraction definition

Abstraction definition	View references

VLNV

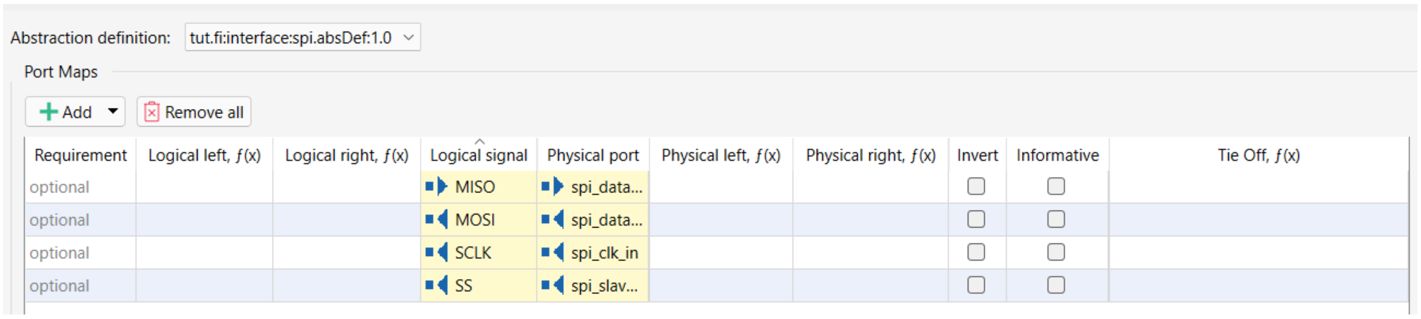
Vendor: tut.fi

Library: interface

Name: spi.absDef

Version: 1.0

Step 4. Open the ports map, add a row and describe the logical signals and physical ports according to the compatibility of the IP's in which you want to add the bus.

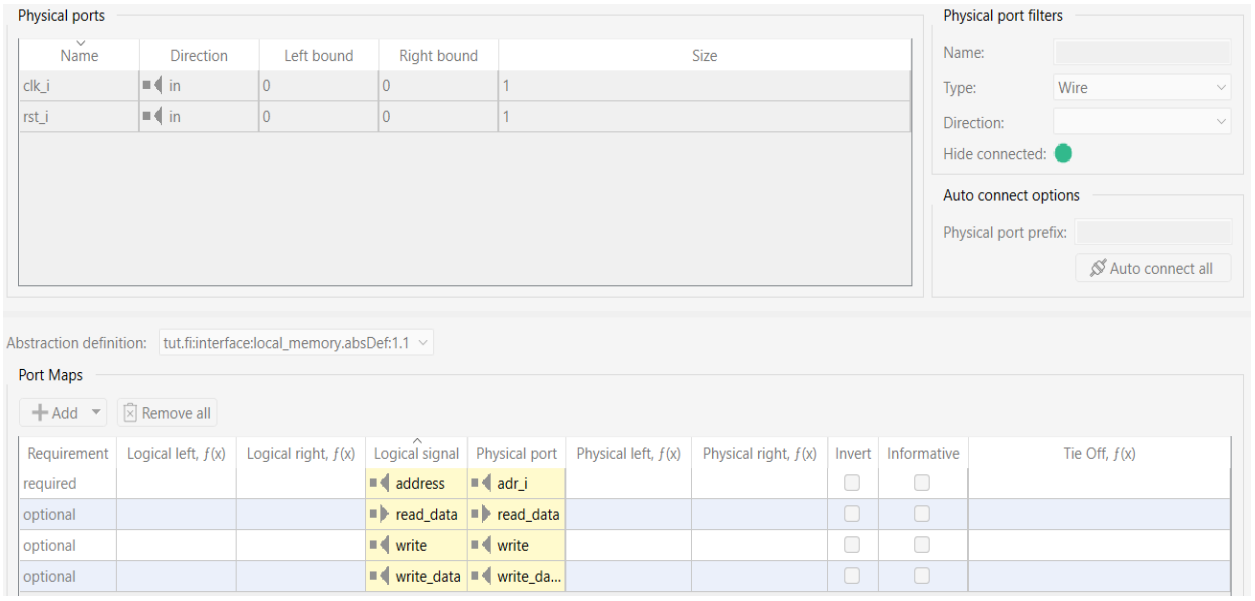


Abstraction definition: tut.fi:interface:spi.absDef:1.0

Port Maps

Requirement	Logical left, ƒ(x)	Logical right, ƒ(x)	Logical signal	Physical port	Physical left, ƒ(x)	Physical right, ƒ(x)	Invert	Informative	Tie Off, ƒ(x)
optional			MISO	spi_data...			☐	☐	
optional			MOSI	spi_data...			☐	☐	
optional			SCLK	spi_clk_in			☐	☐	
optional			SS	spi_slav...			☐	☐	

This is the bus interface's ports of the data memory component:



Physical ports

Name	Direction	Left bound	Right bound	Size
clk_i	in	0	0	1
rst_i	in	0	0	1

Physical port filters

Name:

Type: Wire

Direction:

Hide connected: ●

Auto connect options

Physical port prefix:

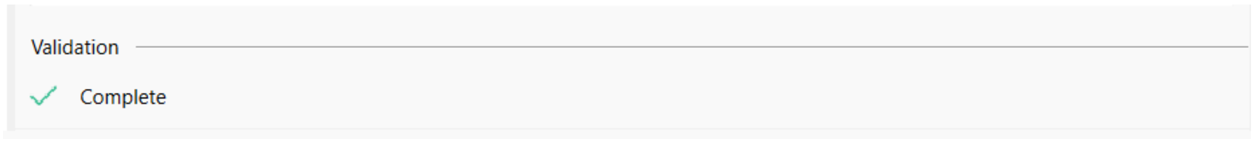
Abstraction definition: tut.fi:interface:local_memory.absDef:1.1

Port Maps

Requirement	Logical left, ƒ(x)	Logical right, ƒ(x)	Logical signal	Physical port	Physical left, ƒ(x)	Physical right, ƒ(x)	Invert	Informative	Tie Off, ƒ(x)
required			address	adr_i			☐	☐	
optional			read_data	read_data			☐	☐	
optional			write	write			☐	☐	
optional			write_data	write_da...			☐	☐	

This Slave bus here uses a local memory abstraction definition bus which carries out 4 ports named address input, read data, write, write data.

If there isn't any error in the hardware design then it will show a green tick on the validation option as shown below:

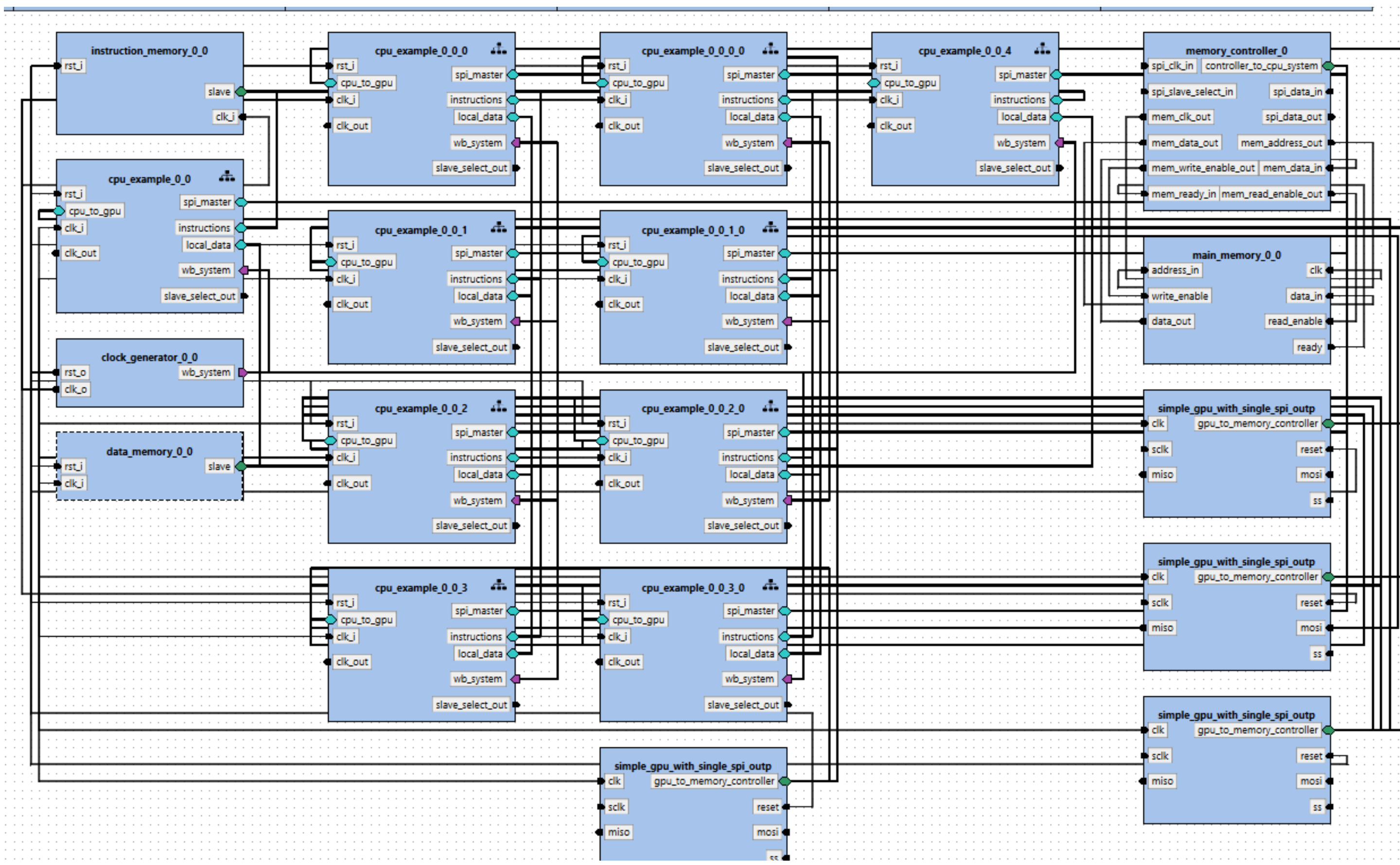


Validation

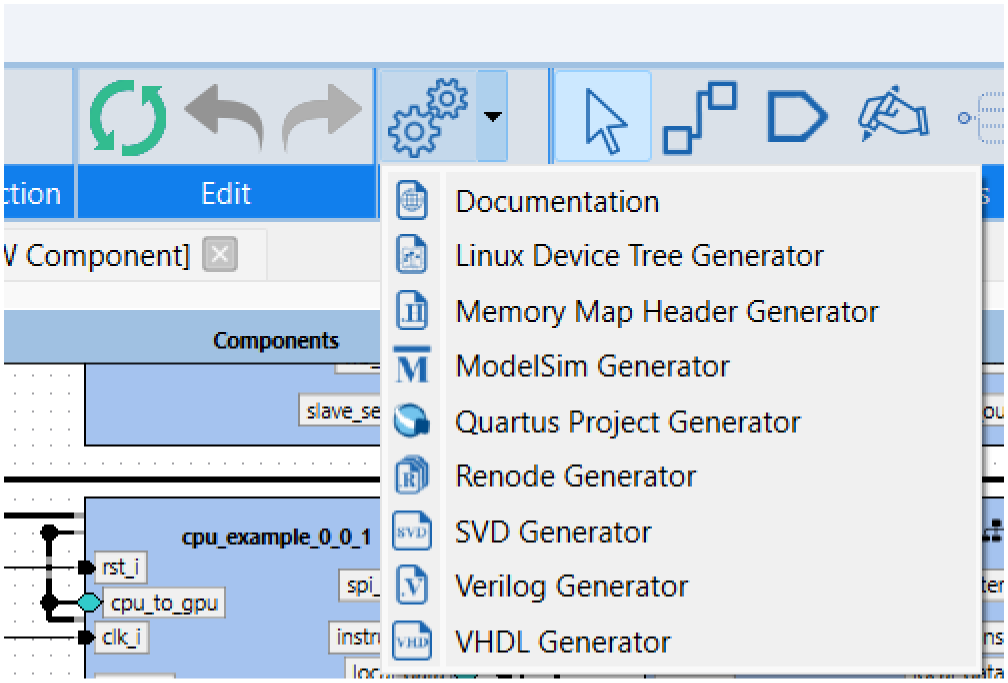
✓ Complete

The Final Design:

The final design looked like this after all the ports are connected :



After creating the HW design of the SoC, you can create the RTL by clicking on the Verilog generator.



Avoiding Errors Caused By XML File:

```
<ipxact:busType vendor="tut.fi" library="interface" name="spi" version="1.0"/>
```

The bus definition specifies the type using four elements: vendor, library, name, and version. Provide the bus definition corresponding to the folder or directory path where the IP component will be stored.

```
<ipxact:abstractionRef vendor="tut.fi" library="interface" name="spi.absDef" version="1.0"/>
```

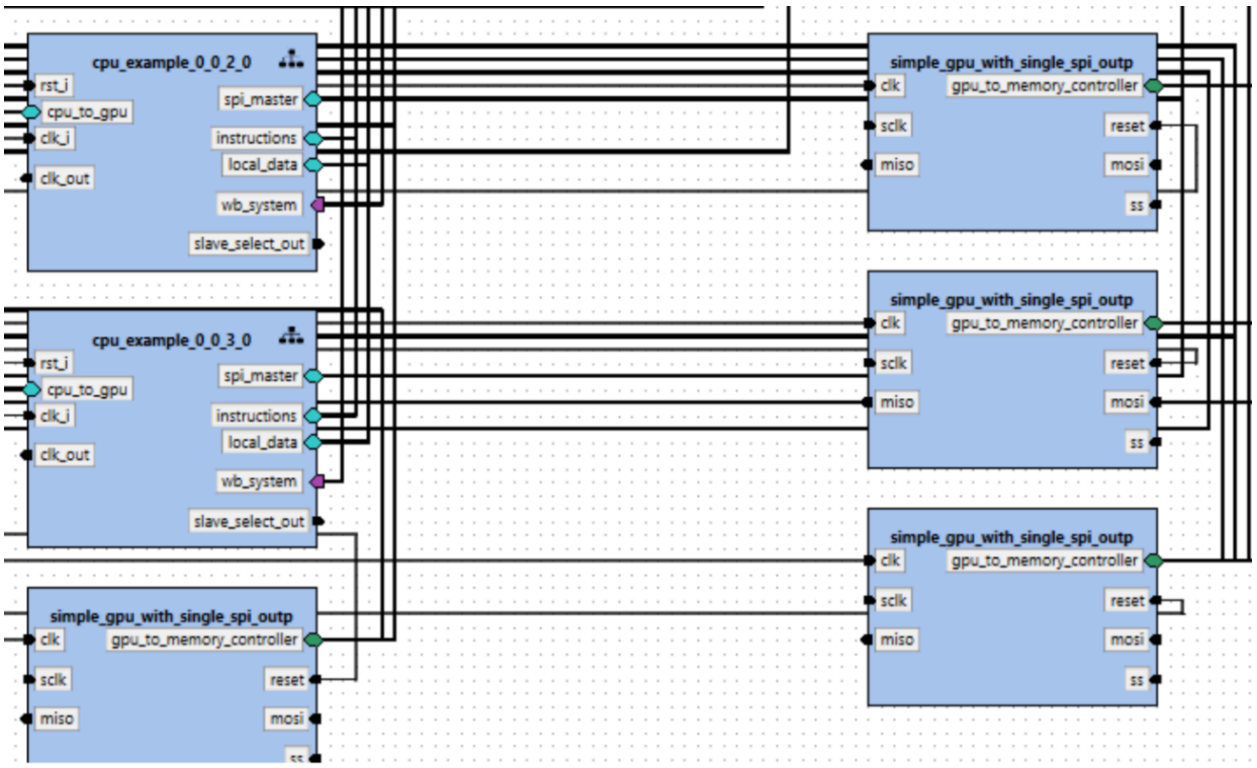
This is the bus abstraction definition This IP-XACT description starts with the container element abstractionDefinition. Subsequently, the identifier for the type is specified using the four elements: vendor, library, name, and version.

The postfix ".absDef" is typically used in the name to indicate that the abstraction definition is a signal-level representation.

It is advisable to give the abstraction definition and bus definition of the bus interface that is available in the library.

Modify the Vendor and Library settings of the bus definition to align it with the directory containing the bus interface templates and communication templates for that specific bus interface protocol, ensuring proper functionality in case of any unexpected errors.

A GPU IP is added which performs basic algorithms, which a GPU can perform, like:



1. Vertex Processing

- **What it does**: Converts 3D vertex data into 2D screen space coordinates using a transformation matrix (e.g., model-view-projection matrix).
- **Functionality in this design**:
    - Accepts 3D vertices as input.
    - Applies a 4x4 transformation matrix to transform the vertices into 2D screen space.
    - Enables rendering of basic 3D objects as a 2D image.

2. Rasterization

- **What it does**: Converts transformed 2D geometric primitives (e.g., triangles) into pixel fragments within a bounding box.
- **Functionality in this design**:
    - Calculates a bounding box around the transformed triangle vertices.
    - Generates pixel coordinates within the bounding box.
    - This creates the basis for fragment generation.

3. Pixel Shading

- **What it does**: Computes the final color of each pixel using simple shader logic.
- **Functionality in this design**:
    - Implements a basic shading algorithm, where the color is derived from pixel coordinates.

    - Demonstrates how shaders influence pixel appearance, albeit with simplified logic.

4. Texture Mapping

- **What it does**: Applies a texture (image) to geometric shapes by fetching and mapping texture colors to corresponding pixels.
- **Functionality in this design**:
    - Integrates a small texture memory (256 entries).
    - Fetches texture colors based on pixel coordinates and combines them with shading logic.
    - Simulates how textures enhance visual richness in rendered images.