# Python HDL interaction for simulation

We will use Python as the AI language, therefore we need to integrate our AI verification models to the existing UVM/SV verification modules and to find ways to call SystemVerilog or VHDL from Python and vice versa. We will not only assign or get data and parameters at the signal levels but also we will ne able to call classes and functions from within the SV packages and vice versa.

There are a number of solutions out there on the web. But not all of them are able to fulfill our needs. We will present them in the following sections. You will need to filter out which ones cater to your needs yourself based on your own project nature.

If you need other interaction needs between different languages (TCL, etc) and verification methodologies (OVM, UVVM, cocotb) and automation scripts (Makefile, TCL/do, bash, bat, perl etc), we have also our own solutions, but these solutions will not be independently presented, because we will focus on the Python/UVM/SV/vcs verification systems in our efforts to optimize the verification process based on Artificial Intelligence and Machine Learning. We will spend more time on AI/ML and leave the other options to others who actually implement the AI/ML based hardware verification speedup and improvement strategies for commercial usage.

# How python can interact with VHDL?

## PyHDL-IF

"PyHDL-IF: An Easy-to-Use Python/HDL Cross-Calling Interface" - Matt Ballance (Latch_2024)

## PyVHDL

https://github.com/GeezerGeek/PyVHDL
http://pyvhdl-docs.readthedocs.io/en/latest/

## Python subprocess

https://peerdh.com/blogs/programming-insights/integrating-vhdl-simulation-with-python-for-automated-testing-1

# How python can interact with SystemVerilog / UVM?

Python can interact with SystemVerilog UVM through several mechanisms, primarily leveraging the SystemVerilog Direct Programming Interface (DPI) or by employing frameworks designed for this purpose.

**1. SystemVerilog DPI-C with a C intermediary:**

Indirect Communication:

SystemVerilog cannot directly communicate with Python. Instead, SystemVerilog uses DPI to call C functions, which then interact with Python.

Process:

SystemVerilog calls a C function via DPI-C.
The C function then calls a Python script or embeds a Python interpreter to execute Python code.
Data transfer between C and Python can be handled using techniques like MessagePack or shared memory.
The results from Python are passed back to C, and then returned to SystemVerilog.

**2. File I/O with $system task:**

Simple Interaction:

SystemVerilog can execute external commands using the $system task, allowing it to invoke Python scripts.

Process:

SystemVerilog writes input data to a file.
It then calls a Python script using $system("python your_script.py input_file output_file").
The Python script processes the input and writes results to an output file.
SystemVerilog reads the output file to retrieve the results.

**3. Using Frameworks like cocotb or uvm-python:**

cocotb:

This framework allows you to write testbenches in Python that directly interact with your HDL design, including SystemVerilog. It provides a robust environment for creating and running tests using Python's features.

uvm-python:

This is a Python port of the SystemVerilog UVM 1.2, built on top of cocotb. It enables the use of UVM concepts and methodology within a Python-based verification environment, offering an API similar to the original SV-UVM. This allows for leveraging Python's strengths for UVM-based verification, potentially simplifying testbench development and maintenance.

**4. Inter-Process Communication (IPC) via Sockets:**

Client-Server Model:

Python and SystemVerilog can communicate as client and server applications using sockets.

Process:

One side (e.g., Python) acts as a server, listening on a specific port.

The other side (e.g., SystemVerilog via DPI-C and a C intermediary) acts as a client, connecting to the server's socket.

Data is exchanged over the established socket connection using a common protocol (e.g., TCP).

The choice of method depends on the complexity of the interaction and the specific needs of the verification environment. For deep, programmatic control and integration, DPI with a C intermediary or frameworks like cocotb are preferred. For simpler, file-based interactions, the $system task can be sufficient.

# DPI-C RELEVANT

## DPI-C

**Direct-Programming-Interface**
https://vlsiweb.com/direct-programming-interface/

## cocotb-BSHL: Reuse of SystemVerilog Verification IPs in cocotb / pyuvm : DPI

https://github.com/Infineon/cocotb-BSHL

Enable Reuse of SystemVerilog Verification IPs in cocotb/pyuvm

Integrating established SystemVerilog Verification IPs (SV-VIPs) utilizing the cocotb and pyuvm framework

Leveraging the Direct Programming Interface (DPI-C) and the ctypes library, our method ensures seamless integration between Python testbenches and SV-VIPs.

## uvm_bridge

https://github.com/Dragon-Git/uvm_bridge

## Py-HPI, replaced by PyHDL-IF

**Python/Simulator integration using procedure calls**

https://github.com/fvutils/py-hpi

**Py-HPI: Applying Python for Verification**
https://bitsbytesgates.com/2019/06/16/py-hpi-applying-python-for-verification.html
https://bitsbytesgates.blogspot.com/2019/06/py-hpi-applying-python-for-verification.html

**Py-HPI: A Procedural HDL/Python Integration**
https://bitsbytesgates.com/2019/06/09/py-hpi-procedural-hdlpython-integration.html
https://bitsbytesgates.blogspot.com/2019/06/py-hpi-procedural-hdlpython-integration.html

## Pybfms

https://github.com/pybfms
https://github.com/pybfms/pybfms
https://github.com/pybfms/pybfms-docs

https://pybfms.readthedocs.io/en/latest/
https://pybfms.readthedocs.io/en/latest/index.html

http://bitsbytesgates.blogspot.com/2020/12/2020-nights-and-weekends-projects-in.html

# SystemVerilog <-> Python Bidirectional

## pyhdl-if

https://github.com/fvutils/pyhdl-if

https://fvutils.github.io/pyhdl-if/

**Status and Roadmap**

PyHDL-IF is still under active development. The information here attempts to capture the status of available features and a roadmap of planned future features.

**Platform Support**

| Linux | Windows (x86_64) | MacOS (x86_64) | MacOS (arm64) |
|---|---|---|---|
| Yes | Planned | Planned | Planned |

The native-compiled portions of PyHDL-IF are compiled according to the relevant Python version-specific requirements. If your favorite platform is not listed above and you would like PyHDL-IF to support, please file a

feature-request ticket and note the platform and the simulator you typically use on that platform.

### HDL Calling Python API

| DPI | FLI | VPI | VHPI |
|---|---|---|---|
| Yes | | Planned | Planned |

Supporting HDL calls to the Python API allows HDL to interact with Python as if it was a Python extension by calling the Python C API.

### Python calling HDL API

| DPI | FLI | VPI | VHPI |
|---|---|---|---|
| Yes | | Planned | Planned |

Support Python calls to an HDL API allows Python to interact with simulator APIs as if it were a shared library loaded by the simulator. For example, support for VPI allows a Python module to walk through the design design hierachy and implement system tasks that the HDL can call.

**DPI Isn't Enough: Making Python Part of Your SV Testbench**
https://bitsbytesgates.com/python/2024/11/04/DPI_Isnt_Enough_MakingPythonPartOfYourSVTB.html
https://bitsbytesgates.com/python/2024/11/04/DPI_Isnt_Enough_MakingPythonPartOfYourSVTB.html

**Easy Access to Python Libraries with a SystemVerilog Convenience API**
https://bitsbytesgates.com/python/2024/11/17/PyHDL_Convenience_API.html
https://fvutils.github.io/pyhdl-if/sv_api.html#systemverilog-api
PyHDL-IF convenience API

## tblink-rpc

https://tblink-rpc.github.io/
https://tblink-rpc.github.io/tblink-rpc-docs/index.html

https://github.com/tblink-rpc/
https://github.com/tblink-rpc/tblink-rpc-core
https://github.com/tblink-rpc/tblink-rpc-hdl
https://github.com/tblink-rpc/pytblink-rpc
**https://github.com/tblink-rpc/tblink-rpc-examples**

# TbLink-RPC: Simplifying the Multi-Language Testbench

https://bitsbytesgates.com/2022/03/27/tblink-rpc-simplifying-multi-language.html
http://bitsbytesgates.blogspot.com/2022/03/tblink-rpc-simplifying-multi-language.html

TbLink-RPC facilitates cross-language communication, including cross-calling between SystemVerilog and Python, within a co-simulation environment. It achieves this by providing a remote procedure call (RPC) mechanism that

allows functions and methods defined in one language to be invoked from another.

Here's how TbLink-RPC enables cross-calling between SystemVerilog and Python:

- **RPC Infrastructure:**

TbLink-RPC provides the core RPC infrastructure, including API implementations in both C++ and Python. This core enables the underlying communication and data serialization/deserialization between the different language environments.

- **HDL Simulator Integration:**

It includes specific integrations for Hardware Description Language (HDL) simulators, such as those for Verilog and SystemVerilog. These integrations allow the SystemVerilog environment to interact with the TbLink-RPC framework.

- **Defining and Registering APIs:**

    - **Python Side:** In Python, classes and methods intended for remote invocation from SystemVerilog are decorated with tblink_rpc.iftype and method decorators (including type annotations for parameters and return values). These decorators register the Python API with the TbLink-RPC infrastructure.
    - **SystemVerilog Side:** Corresponding SystemVerilog classes are created. These classes often combine a base class from the TbLink-RPC library (e.g., TbLinkLaunchUvmObj) with generated implementation classes that mirror the Python API definition.

- **Launching and Connecting:**
    - From the SystemVerilog testbench, an instance of the Python class is launched and connected with its corresponding SystemVerilog representation. This establishes the communication link.

- **Remote Procedure Calls:**

Once connected, the SystemVerilog code can make method calls on its local representation of the Python class. TbLink-RPC handles the marshalling of data, sending the request to the Python environment, executing the corresponding Python method, and returning the result to SystemVerilog.

**Example Use Cases:**

- **UVM Sequences in Python:**

Implementing UVM sequences in Python to leverage Python's capabilities for high-level test behavior, data manipulation, and access to external libraries.

- **Reference Models:**

Using Python to create reference models for design verification, allowing SystemVerilog testbenches to call these models to obtain expected results for comparison with DUT outputs.

# Calling SystemVerilog from Python

## PyMTL

PyMTL An Open-Source Python-Based Hardware Generation, Simulation, and Verification Framework

https://github.com/pymtl/pymtl3/

https://github.com/pymtl/pymtl3-quick-demo

https://github.com/cornell-brg/pymtl
https://github.com/cornell-brg/pymtl3

https://pymtl.github.io/

https://pymtl3.readthedocs.io/

**Cross-calling Python SystemVerilog with PyMTL3**

PyMTL3 provides capabilities for co-simulating external SystemVerilog (or Verilog) modules within the PyMTL3 framework. This means you can integrate existing SystemVerilog designs into your PyMTl3 simulation environment and test them using Python-based methodologies.

Here's a breakdown of how this process generally works:

**1. Using Verilator for SystemVerilog Import:**

Verilator: PyMTL3 leverages the open-source Verilator toolchain to compile your Verilog/SystemVerilog RTL models into C++ simulators.

Importing into PyMTL3: PyMTL3 then uses its import passes to bring these compiled SystemVerilog modules into the PyMTL3 environment as "black-box" models. These black-box models effectively act as placeholders that communicate with the external SystemVerilog/C++ simulation.

Benefits: This allows you to combine the familiarity of Verilog/SystemVerilog with the productivity and extensive testing capabilities of Python.

**2. Co-simulation Flow:**

PyMTL3 Testbench: You can create a PyMTL3 testbench that interacts with the imported SystemVerilog module.

Translation and Import: PyMTL3 can translate your PyMTL3 RTL design into SystemVerilog (if you started in PyMTL3) and then import it back for simulation. This allows you to test the generated SystemVerilog code using your existing PyMTL3 test harness and test cases.

Faster Simulation: This translation-and-import approach can even lead to faster RTL simulation speeds, acting like a Just-In-Time (JIT) compiler.

**3. Limitations and Considerations:**

SystemVerilog Calling Python: SystemVerilog itself cannot directly call Python. You would typically need to use DPI-C to call C code from SystemVerilog, and then use the Python C/C++ interface to interact with Python.

Data Exchange: If you need to exchange data between SystemVerilog and Python when initiating a call from SystemVerilog, you might use file I/O or the DPI-C interface as described above.

**In summary:**

PyMTL3's strength lies in its ability to call SystemVerilog from Python for co-simulation, enabling efficient testing and integration of external SystemVerilog IPs. While calling Python from SystemVerilog is possible, it requires more indirect methods involving intermediate languages like C and the Python C/C++ interface.

## Sockets: amiq_sv_c_python

Connect SystemVerilog with Python
https://github.com/amiq-consulting/amiq_blog/tree/master/amiq_sv_c_python_how_to_connect_systemverilog_with_python
https://www.consulting.amiq.com/2019/03/22/how-to-connect-systemverilog-with-python/

# Calling Python from SystemVerilog

## pysv

**pysv: Python SystemVerilog (Python SV)**
**Running Python Code in SystemVerilog**

https://github.com/Kuree/pysv
https://pysv.readthedocs.io/

## pysv-numpy

**Run Python functions in System-Verilog** with supporting the interplay between numpy and svOpenArrayHandle. Also optimize the runtime performance.

https://github.com/LIU-Yinyi/pysv-numpy

## pystim

**Call Python functions, use Python classes, and execute Python scripts within the SystemVerilog environment by embedding a Python interpreter**
https://pystim.dev/
https://pystim.dev/tutorial-call-python-functions-from-systemverilog/

**How PyStim Connects Python and SystemVerilog for RTL Design Verification ?**
https://dev.to/anl5736/how-pystim-connects-python-and-systemverilog-for-rtl-design-verification-epo