Open-source Framework for Co-emulation using PYNQ

Ioana-Cătălina Cristea
Amiq Consulting
Bucharest, Romania
ioana.catalina.cristea@amiq.com

Dragoș Dospinescu
Amiq Consulting
Bucharest, Romania
dragos.dospinescu@amiq.com

***Abstract* - Functional verification using co-emulation has seen a growing trend due to its main advantage: testbench acceleration. Co-emulation requires two main things: (1) a connection between the host machine running the testbench and the hardware platform where the design is synthesized, and (2) a software component for interacting with the design. Most currently available solutions for achieving a complete co-emulation environment are proprietary. This paper describes an Open-source Framework for Co-emulation (OFC) used for communication between a UVM-SystemVerilog testbench and a design emulated on the FPGA logic of a PYNQ board. The OFC framework is split into two main components: a TCP socket-based client-server connection and a Python component that interacts with the FPGA using the API provided by Xilinx for the PYNQ board. Owing to its modular implementation, the two components can be used either together or separately, depending on the user's needs.**

+ expensive e.g:- Cadence Palladium. {Emulating Engine}

Keywords - testbench acceleration, co-emulation, SystemVerilog, UVM, Python, PYNQ, TCP sockets, DPI-C, DMA, DUT, OFC

I. Introduction

Using co-emulation involves migrating a design under test (DUT), together with parts of the verification testbench logic, onto a hardware platform (typically an FPGA). One interesting use case for co-emulation is verification testbench acceleration as this can achieve shorter runtimes for test scenarios, verification of RTL features which depend on long simulation times, realistic performance benchmarking of the DUT, stress testing etc.

When migrating the design under test onto the FPGA board, the driving and the monitoring logic must be redefined, since the testbench can no longer interact directly with the DUT. Instead, the testbench connects to the hardware platform in order to pass over the generated stimulus for the intended test scenario. Subsequently, the hardware platform takes over the responsibility for controlling and reacting to the DUT interfaces by introducing new synthesizable monitoring and driving components.

Basically, PS handles Drivers & Monitors.

The solution presented in this paper is that of an open-source framework for co-emulation (OFC) using the PYNQ hardware platform. As part of the Zynq family, the PYNQ board has a "Processing System" side (PS) and a "Programmable Logic" side (PL). As such, the OFC framework provides two main components for interacting with the FPGA platform:

1. OFC SV-Python: connects an UVM-SystemVerilog testbench to a Python component (here, the PS side of a PYNQ board), through DPI-C and TCP sockets[1]
2. OFC Python-FPGA: connects the PS side and the PL side of the PYNQ board (via the API for PYNQ provided by Xilinx[2])

For debugging purposes, the OFC framework also provides logging capabilities. → Check Pynq Docs for This

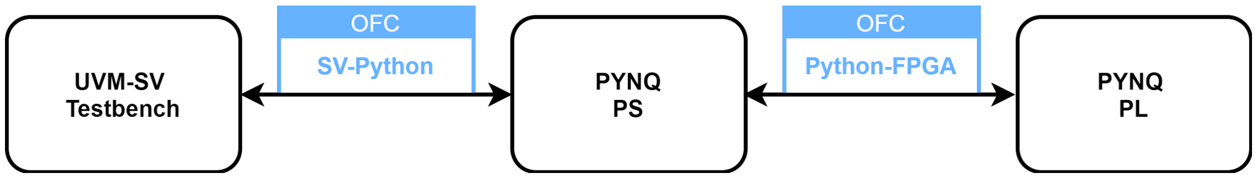


Figure 1. Overview of the OFC components.

II. Overview

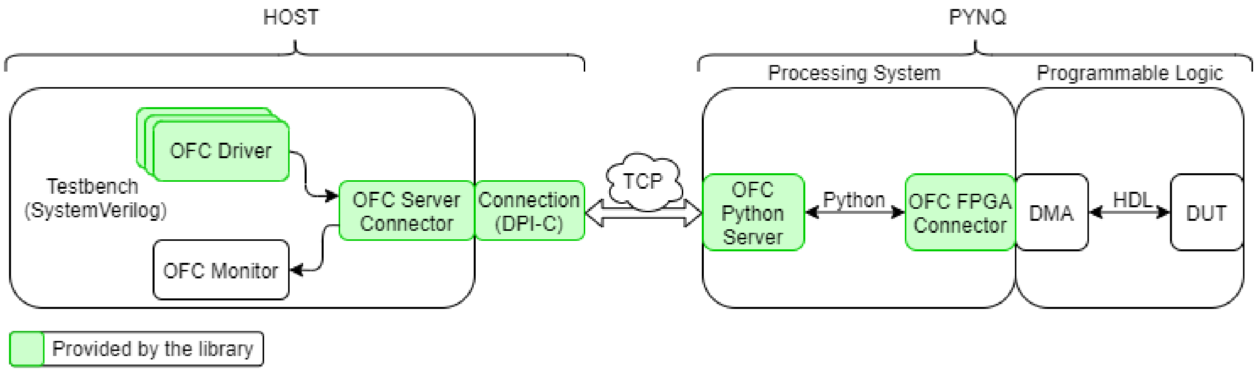


Figure 2. Overview of the connection between the testbench and the DUT.

The client communication API is defined in a DPI-C layer and is responsible for transferring messages containing stimuli to the Python server located on the PS side of the PYNQ board.

The interaction between the synthesized hardware and the server is done through Direct Memory Access (DMA) operations. The DMA IP is provided by Xilinx and can be integrated through the Vivado environment[3]. The manipulation of the DMA modules is done using the Pynq API.

III. OFC SV-Python Connection

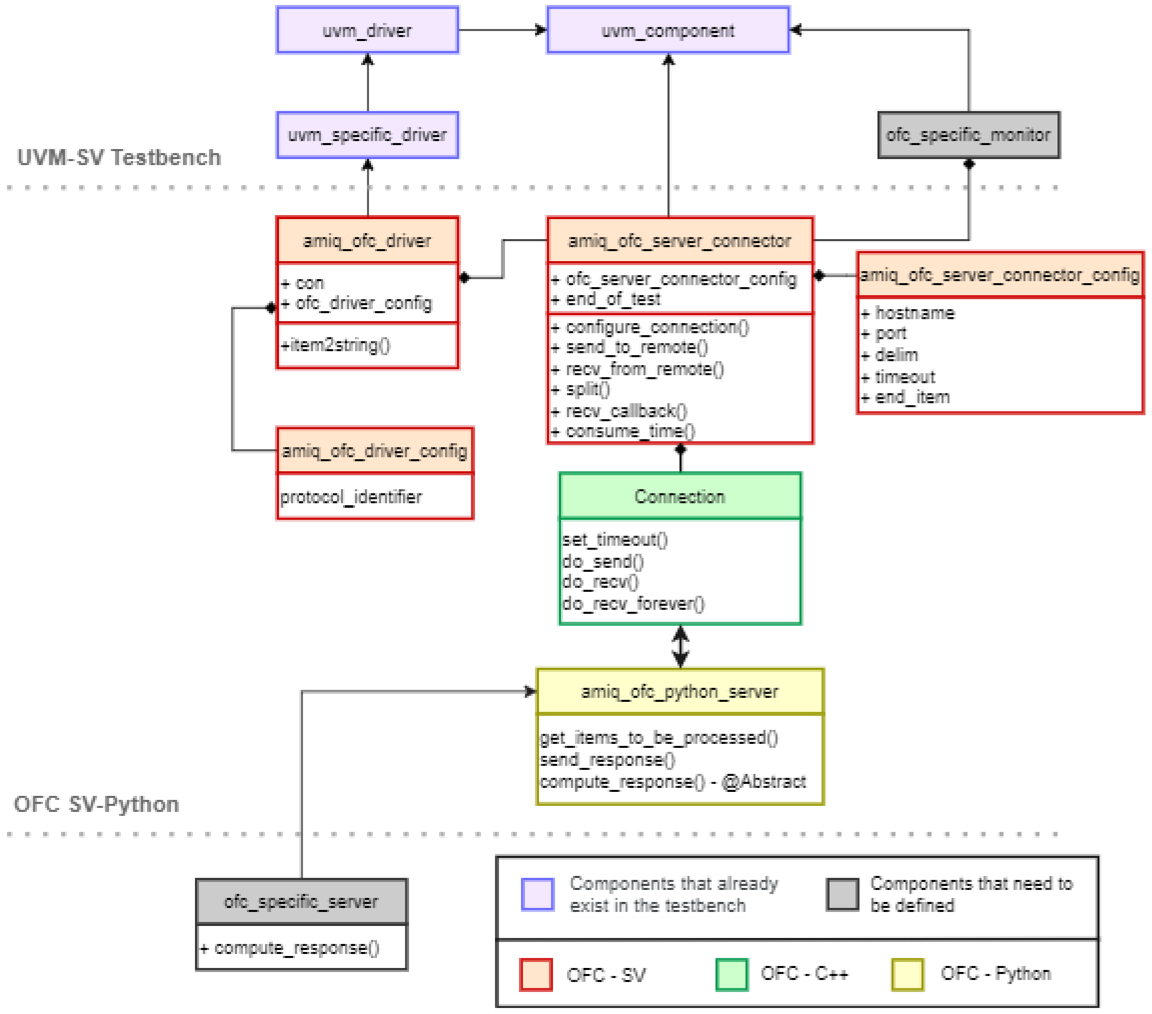


Figure 3. OFC SV-Python Class Diagram.

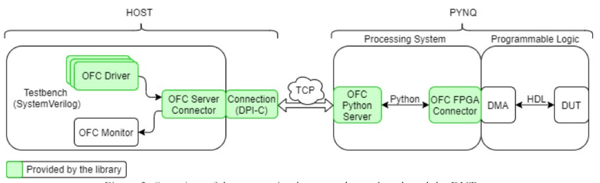


Figure 2. Overview of the connection between the testbench and the DUT.

The first component of the OFC framework is the one connecting a testbench based on UVM and SystemVerilog with the Processing System side of the PYNQ hardware platform, or, if needed, with another python component.
To integrate this component, the user must define the following:
- the behavior of the Python Server when a message is received on the Python side
- the behavior of the OFC specific monitor when a message is received by the testbench

} → TCP server

A more detailed description of the OFC SV-Python functionality is presented below.

1. OFC Driver

In a verification environment, the DUT is normally stimulated using UVM drivers. The OFC SV-Python Framework provides a driver that can be used for the co-emulation process. The role of the OFC driver is to send items to the OFC Python Server instead of directly stimulating the DUT.

In terms of implementation, the client-server communication is performed using strings. As a result, each item sent by the OFC driver requires a string conversion operation. The item2string() function takes care of this aspect by first packing the item into a list of bytes (using UVM's pack_bytes() function) and then converting each byte to the corresponding ASCII character. If the Design Under Test has multiple input interfaces/protocols this means that the SV-Testbench side has multiple active agents generating inbound traffic. When OFC is used, all agents drive data to a singular OFC connector inside the Python Server and then the OFC connector sends the received data to the proper DUT interface.

The OFC connector can receive items from multiple testbench sequencers. In this case, the user is responsible for providing a way to differentiate among various item sources. The OFC connector uses this source information to further drive the items to the corresponding DUT interface. The user is responsible for providing this item source information or he/she can use an existing mechanism from the OFC framework. This existing mechanism inserts the interface/protocol name, defined within the driver configuration object, into the communication string/item. The OFC connector will then extract this string and identify the correct interface to which the item should be driven.

```
 function string item2string(uvm_object req);
        byte unsigned  p_bytes [];
        string                  item_string = "";
        string                  string_id;
        if(!req.pack_bytes(p_bytes))
              `uvm_error(get_name(), "Could not pack item!")
        foreach(p_bytes[j]) begin
               string byte_string;
               $sformat(byte_string, "%h", p_bytes[j]);
               item_string = {item_string, byte_string};
        end
        string_id = ofc_driver_config.protocol_identifier;
        item_string = {string_id, item_string};
        return item_string;
endfunction: item2string
```

Through Host Computer
Sends strings to the
PS, PS Convert Them into
Transcations using which
DUT is Driven....

Snippet 1. The item2string() function.

Each item converted to a string (and where appropriate with the protocol label attached) is sent to the OFC Server Connector.

2. OFC Server Connector

The connection between SystemVerilog and the OFC Python Server is achieved through the OFC Server Connector. The OFC Server Connector uses C++ functions, defined in the DPI-C layer and exported in the SystemVerilog code, to create and manipulate the TCP socket used to communicate with the OFC Server Connector. The interaction between a UVM-SystemVerilog testbench and the OFC Server connector is done through two tasks:
- **send_item()**: to send items from the testbench to the OFC Python Server
- **recv_item()**: to receive items processed by the OFC Python Server

When initializing the OFC Server Connector, a few parameters need to be provided via the OFC Server Connector Configuration Object:

- **hostname** and **port**: the IP/hostname and port number of the Python Server
- **delim**: the character used for delimiting the items within a message (the default value for the delimiter character is *endline*)
- **timeout**: the number of milliseconds that should be waited when sending or receiving an item
- **end_item**: the item that signals the end of the simulation

Instead of UVM-SV based Testbench, if we use CoCoTb Based Testbench Then most of the Blocks can be Removed & System Block will be more easier

The OFC Server Connector has four main functionalities:

a. Configuring the connection to the OFC Python Server
The OFC Server Connector is responsible for initiating a connection with the OFC Python Server. This is done by calling the configure() DPI-C function and giving as arguments the server IP address and the port number. The connection, once established, is kept alive throughout the entire simulation time.

```
function void setup_connection();
       // Create connection to server
       if(configure(ofc_server_connector_config.hostname,
                    ofc_server_connector_config.port) != 0)
            $error("Could not establish connection!");
       // Set how many milliseconds to wait for
        // socket events when reading/writing to it
       set_timeout(ofc_server_connector_config.timeout);
endfunction: setup_connection
```

Snippet 2. The setup_connection() function.

b. Sending items to the OFC Python Server
Items received through the send_mbox mailbox are sent to the OFC Python Server using the function send_data(). Each item is concatenated with the delimiter character so that on the other side the Python Server can reconstruct and distinguish each item within a message.

```
task send_to_remote();
        int send_rsp;
        string item_str;
        send_mbox.get(item_str);
        item_str = {item_str, ofc_server_connector_config.delim};
        do begin
               send_rsp = send_data(item_str, item_str.len());
               if (send_rsp > 0)
                    // While only part of the message was sent to the server
                    // save the other part so it can be sent at next iteration
                    item_str = item_str.substr(send_rsp, item_str.len()-1);
        //exit loop when entire message was sent
        end while (send_rsp != item_str.len()) ;
endtask
```

Snippet 3. The send_to_remote() function.

c. Receiving items from the OFC Python Server
Each time a message is received by the Connection structure, a notification (receive_from_server) is sent to the SystemVerilog testbench. The entire message is split into string items based on the delimiter and each item sent through the recv_mbox mailbox.

```
task recv_from_remote();
      string items[];
      // Wait for the notification from the receive thread
      @receive_from_server;
      // consume all transactions in queue
      while (msgs_q.size() > 0) begin
            // split message into items
            split(msgs_q.pop_front(),
                  ofc_server_connector_config.delim,
                  items);
            foreach (items[i]) begin
                  recv_mbox.put(items[i]);
                  // end of test mechanism
                  if(items[i] == ofc_server_connector_config.end_item)
                        end_of_test = 1;
            end
      end
endtask
```

Snippet 4. The recv_from_remote() function.

d. Signaling the end of the simulation
When the end_item configured in the OFC Server Connector Configuration class is received, the end_of_test flag is set, as seen in the code of recv_from_remote(). The OFC Server Connector provides a task that waits for the above flag to be set. This functionality can be used to end the simulation.

```
task wait_for_end_item();
      wait(end_of_test == 1);
endtask
```

Snippet 5. The wait_for_end_item() function.

The run_phase() of the OFC Server Connector consists of configuring the connection, starting a thread for receiving messages in the DPI-C layer and waiting for messages to be sent and received. The receive thread is given by an infinite loop in the DPI-C layer which waits for messages. Without this loop, messages from the Python Server cannot be received.

```
virtual task run_phase(uvm_phase phase);
        setup_connection(); //Initialize connection to server
        fork
                recv_thread(); //Start recv thread in DPI-C layer
        join_none
        fork
                forever send_to_remote();
                forever recv_from_remote();
        join
endtask
```

Snippet 6. The run_phase() of the OFC Server Connector.

For more information on how the OFC connector works, see "Non-Blocking Communication in SystemVerilog using DPI-C"[1].

$\longrightarrow$ System Verilog features.

3. OFC Python Server.
The PYNQ processing system contains the Python Server. This server processes items received in a string format based on the user's needs. The OFC SV-Python component provides a server class with an abstract function for computing the response (compute_response()) which receives a list of string items and processes them in order to be sent back to the client. This function is overridden in a child class by the user. In a co-

$$\begin{tikzcd}
\text{Input Strings from HOST} \arrow[r] & \text{(Dual Core A9) Received \& Forwarded via PS} \arrow[r] & \text{DMA} \arrow[r] & \text{(FPGA) PL}
\end{tikzcd}$$

emulation environment, its role is to capture requests received from the testbench and transfer them to the DUT through Direct Memory Access operations.

Every interface type has an associated Direct Memory Access module, and therefore the server is responsible for unpacking and mapping the stimulus to the protocol-specific Direct Memory Access block. Driving the item to the proper interface means that the Python Server triggers a DMA access for the decoded interface ID.

The communication with the Direct Memory Access IPs from the programmable logic side is achieved via the Python API provided by Xilinx for the PYNQ board and is part of the OFC Python-FPGA framework component, which will be described in the next chapter. $\rightarrow$ Pynq- DMA Tutorial.

4. Testbench

Compared to a simulation testbench, a co-emulation testbench does not connect directly to the DUT; instead, it interacts with the tested module via the PYNQ board using the TCP client API implemented in DPI-C.

None of the testbench components should depend on clock cycles or timing values, since the clock is generated in the FPGA and cannot be controlled from the testbench. Only the drivers and monitors need to change, assuming that the rest of the testbench does not depend on the clock signal.

a. Driving
To adapt a testbench for a co-emulation environment, the drivers should be overridden through the UVM factory mechanism with the driver provided by the OFC SV-Python Driver.

b. Monitoring
For the monitoring process, a new monitor has to be defined. All messages are received by the OFC Server Connector component from the OFC Python Server regardless of the protocol type of the item. Therefore, the new monitor should take the received items from the recv_mbox of the OFC Server Connector, convert them from string into specific items of the testbench and send them through the appropriate analysis ports.

IV. OFC Python-FPGA Interaction

The second component of the OFC framework is the one connecting Python to the Programmable Logic side of the PYNQ board. This is achieved using the PYNQ Overlay API provided by Xilinx.

$\rightarrow$ Overlays are Basically Loading Designs onto the FPGA from Pynq.

This component offers three functionalities:

1. Configuring the PL side of the PYNQ board.

The FPGA logic of the PYNQ board can be programmed using a bitstream file and a hardware description file, both of which can be generated through the Vivado environment.

The initialization of this class is also responsible for finding the DMA IPs within the design. These IPs will be used for transferring data between Python and the PL side. Each DMA has two channels: recvchannel and sendchannel. In addition, each channel has an associated buffer used for storing data after receiving or sending a transfer.

```
def __init__(self, bit_file):
     # Program the PL side using the bitstream file
     # The overlay also holds the design IP and hierarchies as attributes
     self.overlay = Overlay(bit_file)
     # ...
     # Extract DMA IPs from overlay
     self.dma = {}
     for ip in self.overlay.ip_dict.keys():
         if 'axi_dma' in self.overlay.ip_dict[ip]['type']:
             self.dma[ip] = getattr(self.overlay, ip)
     # ...
     self.buffers = {}
     for key in self.dma.keys():
          self.buffers[key] = amiq_ps_pl_connector_buffers()
```

Snippet 7. The initialization of the Python-FPGA connector.

2. Sending items through DMA to the PL side.

Sending items to the PL side can be done using the send_to_dma() function. This function takes as arguments an item or a list of items and the name of the DMA to be used for the transfer. This transfer is done in a blocking manner so as to ensure that every item reaches the DUT interfaces.

Each item should be a number with a size corresponding to the DMA sendchannel size, as the DMA operates with a list of bits. The HDL driver should know how to interpret the integer number and send it onto the DUT interfaces.

Sending a list of items with a size not equal to the buffer used for the DMA transfer will result in an error unless the force_send flag is set.

Sending entire lists of items at a time will result in back-to-back transactions stimulating the emulated design. If a fine granularity for delays between packets is required, then the delay measured in clock cycles can be packed within the item and processed by the HDL driver. This is an advantage of using FIFO modules within the HDL side.

```
def send_to_dma(self, to_send, dma_name, force_send = False):
    # ...
    dma_buffer = self.buffers[dma_name].send
    np.copyto(dma_buffer, to_send)
    self.dma[dma_name].sendchannel.transfer( dma_buffer )
    self.dma[dma_name].sendchannel.wait()
    # ...
```

Snippet 8. The send_to_dma() function.

3. Receiving items from the PL side through DMA.

Receiving items from the PL side can be done through the recv_from_dma() function. This function takes as argument the name of the DMA from which the transfer is required and stores the received item in the receive buffer associated with said DMA. This transfer is done in a non-blocking manner, so that if no item is monitored, the program does not block.

Each item received is a number with a size corresponding to the DMA recvchannel size.

```
def recv_from_dma(self, dma_name):
     # ...
     dma_buffer = self.buffers[dma_name].recv
     # Indicate the buffer which will store the received item
     # and wait for transfer to complete
     self.dma[dma_name].recvchannel.transfer(dma_buffer)
     self.dma[dma_name].recvchannel.wait_async()
     # ...
     return dma_buffer
```

Snippet 9. The recv_from_dma() function.

V. FPGA

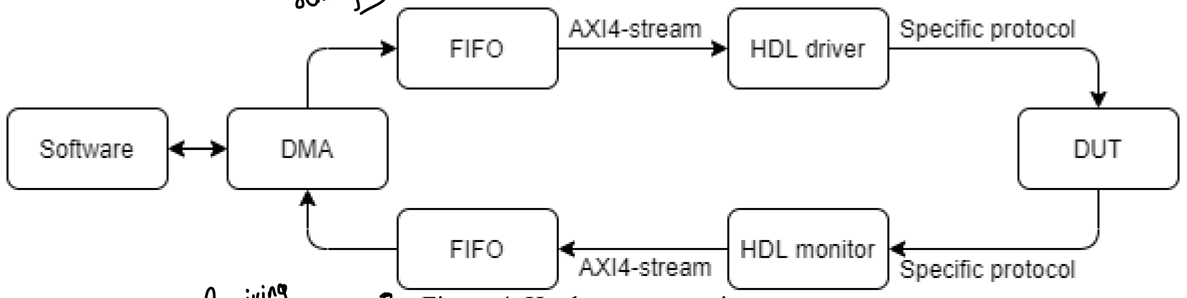
The hardware side is responsible for transferring items received from the software to the device under test and then transferring the results back to the software for processing. To accomplish this connection, a Direct Memory Access IP is used for each of the DUT interfaces.

The Direct Memory Access modules communicate with the emulated design through the synthesizable drivers and monitors, which are called hereafter HDL components. The HDL components implement two interfaces:

1. One for communicating with the Direct Memory Access module
2. One for communicating with the device under test.

What Type of Driver ?

Sending Buffers ?



Receiving Buffers

Figure 4. Hardware connections

The Direct Memory Access IP interfaces are based on the widely adopted AXI4-stream protocol. Therefore the HDL components will contain an AXI4-stream slave interface used to receive information from the Direct Memory Access modules (drivers) or an AXI4-stream master interface used to transfer information to the Direct Memory Access modules (monitors). The code for the AXI4-stream interface can be generated through Vivado. The other interface of the HDL components will be the targeted interface of the emulated design.

In order to be able to send and receive multiple items at the same time, a FIFO IP is also introduced between each DMA-HDL component. Using FIFOs increases the performance of the co-emulation project by decreasing the number of DMA accesses, but also allows the user to insert a specific delay between the items sent through a single DMA transfer.

Debugging designs implemented in FPGA logic can be quite difficult without waveforms. The following methods can be used for the debugging process:

- Instantiating a debug IP, for example: the Integrated Logic Analyzer (ILA) Core provided by Xilinx[4]
- Using the integrated log capabilities of the OFC Python-FPGA component
- Creating your own debug monitor within the HDL side

VI. Integration

To integrate the OFC framework for a co-emulation environment, the user should:

1. Replace the UVM drivers with the OFC driver through UVM factory override
2. Make sure that each item has pack and unpack functions
3. Create an OFC specific monitor which receives string messages from the OFC Server Connector, converts them into items and sends them through the appropriate analysis port
4. Extend the OFC Python Server in order to define the compute_response() function. The comupte_response() function receives a list of string items and should then:
    - unpack them for DMA accesses
    - use the OFC Python-FPGA Connector to send stimuli
    - use the OFC Python-FPGA Connector to receive monitored items
    - return packed monitored items
5. Create the HDL drivers and monitors and connect them to DMA modules within the PL side

VII. Limitations

This architecture implies some limitations since the testbench no longer has direct access to the signals of the DUT interfaces. This means that coverage, checks and any other logic that relies on the signal level changes have to be modified. For example, SystemVerilog Assertions will have to be synthesized and moved to the emulated design. Another example could be finely controlling the behavior of interface signals used to drive valid, backpressure, etc.

That behavior will also have to be implemented within the HDL drivers. However, these limitations are also valid for any co-emulation environment.

→ At the Later stages of Asic Flow, when Design is working At root Level, and/or when we need to do Functional Verification.

Co-emulation is best suited for high level scenarios where there is no need for fine control of the signals. It is also more advantageous to co-emulate a data processing block or streaming block than a forwarding block, such as an arbiter or a scheduler.

The solution provided by this framework uses Direct Memory Access transfers for each set of stimuli and monitored items. These transfers represent the main bottleneck of the described architecture. Therefore, the fewer Direct Memory Access operations made, the faster the co-emulation process will be.

VIII. Conclusions

We have developed a modular open-source framework for co-emulation split into two main components. The user can integrate these components in order to create a connection between a UVM-SystemVerilog testbench and a Python environment, and/or to create a link between the PS side and the PL side of a PYNQ board. Owing to its modular design, the OFC framework can be used either as a whole, for a full co-emulation environment, or partially, to achieve SV-Python/Python-FPGA communication only.

Integration of the OFC SV-Python component into a testbench requires the user to define an OFC monitor and the way in which the OFC Python Server processes transactions received from the testbench. This allows for maximum flexibility, as the user is not constrained in any way by the behavior of the framework. If the goal is a co-emulation process, then the OFC Python-FPGA component can be attached through the OFC Python Server.

Co-emulating the functional verification testbench implies splitting drivers and monitors into timed and untimed components and connecting them. The suggested connection is based on a client-server communication and the use of Direct Memory Access operations. This architecture uses a PYNQ FPGA board since it provides a Python API required for the Direct Memory Access manipulation.

→ No Data Related to the Speed and Improvement Compare with Software Based Verification

→ Implementing This framework for CoCoTb Testbench will be Interesting.

References

[1] Amiq Consulting, "Non-Blocking Socket communication in SystemVerilog using DPI-C "
[2] Xilinx, "Python Overlay API"
[3] Xilinx, "AXI DMA v7.1 - LogiCORE IP Product Guide"
[4] Xilinx, "Integrated Logic Analyzer v6.1 - LogiCORE IP Product Guide"