TWEPP 2021
Topical Workshop on
Electronics for Particle Physics
(20-24 September 2021)

Easy and structured approach for software and firmware co-simulation for bus centric designs

Michał Kruszewski, M.Kruszewski@elka.pw.edu.pl, Institute of Electronic Systems, Warsaw University of Technology

Introduction

Software and firmware co-simulation can save a lot of time and money, as it leads to a lower number of HDL project builds and reduces the number of test iterations with the real hardware.

Despite its obvious advantages the co-simulation is still relatively rare to see in FPGA designs. There are at least four reasons for such a situation. The first one is that setting up a co-simulation framework requires knowledge of multiple computing areas. The second one is that it might be time-consuming. The third one is that ready-to-use frameworks sometimes do not support some of the desired HDL features, for example handling compound types such as records or arrays. The fourth one is that ready-to-use frameworks, such as cocotb, are strictly coupled with a single programming language. This work presents the modular approach that tries to be in line with the Unix philosophy.

Concept

The co-simulation framework consists of the following mandatory elements:

1. software co-simulation interface,
2. HDL co-simulation interface,
3. HDL BFM (Bus Functional Model),
4. test runner.

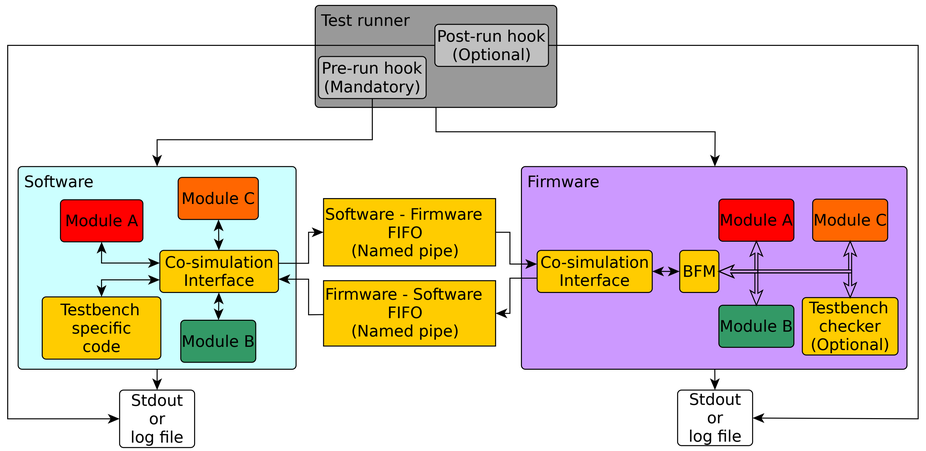
The framework blocks are loosely coupled, and each of them can be easily replaced. The whole test bench additionally consists of the project's software and firmware code. The co-simulation interfaces are relatively short and straightforward, and once written, they can be reused for different tests within the project. If other software languages are used for the prototype and target implementation phases, it is also easy to write a co-simulation interface for the new language and reuse the co-simulation framework from the prototyping phase. The BFM can be custom or taken from a library such as OSVVM or UVVM. The idea is based on the assumption that all communication is done via the bus. Not only the regular data is transferred via the bus, but also the test bench specific data. Such an approach is immune to the lack of support for compound types. What is more, the bus infrastructure is tested by the way.

The proposed approach is not free of drawbacks. The first one is that the firmware design must have some kind of bus. This should not be a problem as almost all complex FPGA designs have some kind of bus, Wishbone and AXI being probably the most popular. The second one is that precise timing checking between signals is hard to achieve solely within the test bench software. It requires a firmware checker accessible via the bus. Another approach is using PSL (Property Specification Language) or SVA (SystemVerilog Assertions).

Implementation

The concept has been successfully implemented. A simplified example showing a co-simulation for an adder is available on [5]. The FuseSoc and fsva tools have been used as the test runner. The AGWB tool is used for the registers generation. Wishbone has been chosen as the bus. The firmware bus infrastructure comes from the General-Cores library. The Wishbone BFM comes from the UVVM library. Although all listed components are necessary, it is worth noting that the whole concept is agnostic to the chosen components. They all depend on the project and personal preferences.

The most important parts are the co-simulation interfaces. Although the control and data can be sent via them in an arbitrary way, they should resemble the interface used for interacting with the actual hardware as much as possible. Such an approach helps to avoid maintaining differences, which leads to less cognitive complexity of the codebase.



Scheme of the co-simulation framework concept.

Example software interface

```
class CosimInterface:
    def __init__(
        self, write_fifo_path, read_fifo_path,
        delay_function=None, delay=False
    ):
        """Create co-simulation interface.
        Parameters:
        -----------
        write_fifo_path - Path to software -> firmware named pipe.
        read_fifo_path  - Path to firmware -> software named pipe.
        delay_function  - Reference to the function returning random
                          value when delay is set to 'True'.
        delay - If set to 'True' there is a random delay between
                any write or read operation. Useful for modelling
                real access times.
        """

    def _make_fifos(self):
        """Create named pipes for inter-process communication."""

    def _remove_fifos(self):
        """Remove named pipes."""

    def write(self, addr, val):
        """Write register.
        Parameters
        ----------
        addr - Register address.
        val  - Value to be written.
        """

    def read(self, addr):
        """Read register.
        Parameters
        ----------
        addr - Register address.
        """

    def wait(self, time_ns):
        """Wait in the simulator for a given amount of time.
        Parameters
        ----------
        time_ns - Time to wait in nanoseconds.
        """

    def end(self, status):
        """End a co-simulation with a given status.
        Parameters:
        -----------
        status - Status to be returned by the simulation process.
        """
```

The code snippet, attached on the left, presents minimal, viable software interface implementation (function bodies have been removed for brevity). The Python language has been chosen for the example as it is very readable, thus is suitable for presenting concepts.

The `__init__()` and `end()` functions are used to set up the inter-process communication and end the co-simulation in a controlled way. As checking, if valid contracts are met during the co-simulation, may occur in the software as well as in the firmware, there is a need to inform the firmware running in the simulator what the exit status is so that it can exit with the same status. This is crucial in the case of running co-simulation as regression test in the CI/CD (Continuous Integration / Continuous Delivery) pipelines. As it is the firmware side that is run by the test runner directly, it must exit with a failure when a failure on the software side is detected. Otherwise, a failed test would be missed by the CI.

The `write()` and `read()` methods are respectively for writing and reading a single register. In the case of more complex interfaces, one can add methods for writing particular bit fields (read-modify-write operation) or methods for grouping operations and dispatching them in order to increase performance or decrease latency.

The whole `CosimInterface` implementation, with the documentation, logging, and formatted with the black code formatter, takes 142 lines.

Example co-simulation outputs

Software side stdout:

```
tb_cosim:INFO:Starting adder cosimulation
cosim_interface:INFO:Removing FIFOs
cosim_interface:INFO:Making FIFOs
tb_cosim:INFO:Generating random numbers
tb_cosim:INFO:a = 62338616, b = 823730966
tb_cosim:INFO:Expected sum = 886069582
cosim_interface:INFO:Waiting for 825 ns
cosim_interface:INFO:Writng address 0x00000002, value 62338616 (0x03b73638)
cosim_interface:INFO:Waiting for 500 ns
cosim_interface:INFO:Writng    address    0x00000003,    value    823730966
(0x31192316)
cosim_interface:INFO:Waiting for 975 ns
cosim_interface:INFO:Reading address 0x00000004
cosim_interface:INFO:Read value 886069582 (0x34d0594e)
tb_cosim:INFO:Read correct value
cosim_interface:INFO:Waiting for 250 ns
tb_cosim:INFO:Ending cosimulation
cosim_interface:INFO:Ending with status 0
cosim_interface:INFO:Removing FIFOs
```

Firmware side stdout:

```
UVVM: ID_POS_ACK  450.0 ns  TB seq.       check_value_in_range() => OK, for
time 25000000 fs. 'checking clk period is within requirement.'
UVVM: ID_BFM        465.0 ns   WISHBONE  BFM wishbone_write(A:x"00000002",
x"346DFE0D") completed. 'cosim interface'
UVVM: ID_POS_ACK 1400.0 ns  TB seq.       check_value_in_range() => OK, for
time 25000000 fs. 'checking clk period is within requirement.'
UVVM: ID_BFM       1415.0 ns   WISHBONE  BFM wishbone_write(A:x"00000003",
x"60182672") completed. 'cosim interface'
UVVM: ID_POS_ACK 2200.0 ns  TB seq.       check_value_in_range() => OK, for
time 25000000 fs. 'checking clk period is within requirement.'
UVVM: ID_BFM       2215.0 ns   WISHBONE  BFM wishbone_read(A:x"00000004")=>
x"9486247F". 'cosim interface'
simulation finished @2465ns
```

Real use case

The proposed approach has been used for testing of DAQ (Data Acquisition) system for the CBM (Compressed Baryonic Matter) experiment that is being prepared at FAIR (Facility for Antiproton and Ion Research) in Darmstadt.

References

1. cocotb, https://github.com/cocotb/cocotb.
2. OSVVM, https://github.com/OSVVM/OSVVM.
3. Tallaksen, E., "UVVM - Universal VHDL Verification Methodology. Setting a standard for VHDL testbenches," (Apr. 2018). https://indico.esa.int/event/232/contributions/2159/.
4. UVVM, https://github.com/UVVM/UVVM.
5. FuseSoc Cosimulation Example, https://github.com/m-kru/FuseSoc-Cosimulation-Example.
6. Kindgren, O., "Invited Paper: A Scalable Approach to IP Management with FuseSoC", https://osda.gitlab.io/19/kindgren.pdf.
7. fsva, https://github.com/m-kru/fsva.
8. Zabołotny, M. W., et al. "Automatic management of local bus address space in complex FPGA-implemented hierarchical systems", 6 November 2019, https://doi.org/10.1117/12.2536259.
9. AGWB, https://github.com/wzab/agwb.
10. General-Cores, https://ohwr.org/project/general-cores.