UVM based Hardware/Software Co-Verification of a HW Coprocessor using Host Execution Techniques

François Cerisier, CEO, AEDVICES Consulting, Grenoble, <francois.cerisier@aedvices.com>
Arnaud Grasset, Research Engineer, Thales Research & Technology, Palaiseau, <arnaud.grasset@thalesgroup.com>
Christian Rivier, Design & Verification Consultant, <christian.rivier@aedvices.com>
Andrea Battistella, Virtual Platforms and System Consultant, <andrea.battistella@aedvices.com>

Abstract—While UVM generalized adoption provides solutions for metric driven verification at IP and subsystem level, there is still a need to co-verify hardware subsystems together with the associated low level software drivers and hardware abstraction layers. If virtualization techniques, ISS as well as emulators address this need, they can be expensive, complex to setup or requiring engineering skills that are not always present within mid-size and small design and verification teams. This often leads to either project schedule shift, a lack of hardware/software verification or of a lack of verification of the top-level integration.

This paper presents a generic hardware/software co-verification approach allowing to run low level software without the needs for ISS based SystemC virtual platforms nor other virtualization tools such as QEMU. We present how we can run low level C code on the host machine and connect it to any UVM verification IP in order to develop system level tests that are reusable on the physical board, while taking the benefits of random stimulus generation, scoreboard checking, assertions and functional coverage brought from SystemVerilog and UVM.

This technique has been applied to a hardware accelerator which integrates a soft-core processor, DMA IPs controlled by the vendor software drivers and a matrix of digital signal computation units and MACs that serves as a coprocessor for the soft core.

Keywords— UVM, SystemVerilog, C, Hardware-Software co-verification, host code execution, TLM, DPI

I. INTRODUCTION

The verification of subsystems that are controlled by software always raise the question of how much is actually performed by the software. When the process is highly coupled with the software, the co-verification of the hardware together with the software drivers is required to ensure that not only the hardware works as expected, but that the software works with the same expectations. Additionally, co-verification techniques further enables test reuse of the tests across platforms, allowing tests to run on actual physical boards without synthesizable testbench nor test rework.

Hw/Sw codesign techniques such as seamless environments have long been introduced for this purpose. However, most of these techniques requires either a processor being simulated, an instruction set simulator (ISS) or the use of virtualization software such as QEMU[2]. Another approach [10] is to run the software directly on the host and use memory and register access APIs to perform read and write bus accesses to the hardware. If these techniques have been questioned whether or not they should be maintained [1] due to the availability of ISS and virtualization platforms, they remain easier to set up and still allow a software test reuse across platforms.

On another side, hardware/software co-verification do not only focus on the software. As a matter of facts, the hardware may require advanced verification techniques such as coverage driven verification, assertions and scoreboards where a UVM approach will bring the benefits of VIP reuse as well as using the UVM and SystemVerilog skills present in today’s verification team.
In the approach described here, we show how to extend a typical UVM environment to complement it with a software driven test approach, allowing software driver co-design and co-verification, enabling test reuse while keeping the benefits of SystemVerilog/UVM and a coverage driven approach.

The proposed approach is using a two-step generic link from the software to the UVM components:

- a generic DPI2UVM library to map software accesses and interrupts to any UVM. This provides read and write APIs to the software and a generic UVM TLM blocking port to the sequencer. Integration of this library will only have to implement a conversion of a uvm_tlm_generic_payload transaction to the given VIP transaction class in the similar way the uvm_reg_adapter converts register accesses to the VIP sequence item.

- A generic host software execution library (relying on host code execution techniques) to provide seamless integration of the DPI layer without the need to modify the software nor use a specific API. Such techniques consist of catching pointer accesses to memory locations and routing them to the given API.

II. TESTBENCH ARCHITECTURE

A. Overview of the DUT

The DUT is part of a HW accelerator for image processing applications (Figure 1). The accelerator is built around a processing engine controlled by a processor. Data transfers between the processing engine and the rest of the system (e.g. DDR memory) are fully controlled by the software executed on a soft core. The software is in charge of sequencing the data transfers and controlling the operations realized by the processing engine. The design of the platform including the DMA controller and the AXI interconnect is mostly based on available IPs.

The verification of this design should ensure that the operations done by the processing engine are correct but also that the connection with the rest of the system works properly. Being able to sequence the operations of the processing engine in the testbench in a realistic way was mandatory for the verification of this design. To meet tight time-to-market requirements, the development of the SW has been done in parallel with the HW design. A functional model of the platform was thus available early in the design phase, as well as preliminary version of the SW.
B. The verification strategy

The verification strategy was based on the following considerations:

- **Use Cases**
  - The Hardware Accelerator is controlled by software using the DMA. There is then a strong dependency on the DMA integration with the design pipeline. The DUT which is considered is therefore the hardware subsystem consisting of
    - The Signal Processing HWA pipeline itself
    - The DMA
    - The interconnects between the DMA and the HWA
    - Glue logics
  - The HW Accelerator is controlled by software using DMA drivers. Both the verification and the software development will benefit from a verification strategy allowing to share the low level drivers.

- **AXI Protocol**
  - The AXI protocol is a relatively complex protocol. The verification of the design with AXI protocol variabilities at signal levels is mandatory to ensure the design works properly under stress conditions. Verification IPs from third vendors have been qualified on different projects and present the advantage to address the AXI protocol from on-the-shelf components, in our case, the Mentor Graphics QVIP for the AXI protocol was made available. The verification strategy applied takes advantage of this.

- **Algorithm C-Model availability:**
  - The full algorithm was available is C. This C-Model has been used to validate the use cases and the overall algorithm. It can therefore be considered as a “golden” C-Model. The verification can take advantage of this to compare the expected results.

Since the CPU core was only available in the FPGA and was not available neither in its RTL form nor as an ISS, we needed to drive directly the AXI interfaces of the design. However, to be able to benefit from the existing software as well as to allow reuse with the software team and the on-board validation, we planned for a software driven test approach.

We therefore planned for a software driven test approach on top of UVM using the DPI to control the VIPs from the C tests (Figure 2).
C. The DPI to UVM generic Agent

In order to facilitate the integration of C code to any types of UVM verification IPs, we have implemented a DPI to UVM library based on precedent work [8] to provide the following facilities:

- Connect C read/write functions to a generic UVM sequencer
- Connect C interrupt handlers to SystemVerilog events
- Address mapping to enable the connection to several UVM verification IPs
- Reports UVM_ERROR, UVM_FATAL and UVM_WARNING API to C.

On the C side, the library provides the read and write functions that are mapped to SystemVerilog using the DPI interface. The SystemVerilog then converts the address and data of the transfer into a UVM TLM Generic Payload sequence item and place this transaction into the UVM TLM blocking transport queue `uvm_tlm_b_transport_port`. This way, an agent can connect to this transport interface and drive the transaction through a VIP sequencer (Figure 3).

![Figure 3 Principle of DPI2UVM generic link to VIPs](image)

For genericity, we delegate the conversion of the read and write to the actual protocols to the user, in the similar way as the `uvm_reg_adapter` works. Users therefore have to inherit from the `dpi2uvm_tlm_conv_container` and implement the conversion function to the VIP sequence item (Figure 4).

In our case, we have converted the `uvm_tlm_generic_payload` transaction into the Mentor AXI QVIP™ sequence item and connected our `dpi2uvm_tlm_agent` to the QVIP sequencer.

This two-step link between the C to the final sequencers has the advantage to provide a generic library with the same API whatever verification IP is used, while deferring the conversion to a simple conversion function to the user. Therefore, it provides a generic approach to connect a C test to any UVM verification IP of the market. It also provides an easy to use C to VIP link without to reinvent the wheel each time a DPI call to a verification IP has to be made.
On the software side, as we were targeting to use the Xilinx DMA IP, we wanted to use the associated C driver functions from the Xilinx HAL libraries (Figure 5). Since Xilinx HAL is using read and write macros for the register accesses, we redefined these link them to the above dpi2uvxm_read and dpi2uvxm_write functions. This allowed us to use the Xilinx DMA C drivers without modification and program the DMA from C, with register accesses were performed through the Mentor QVIP.

However, this approach requires the use of read and write macros in the software. If this generally not a constrained for verification engineers, it was however a strong constraint to the software engineers, limiting them in their ability to use direct register structure pointers in their API.

This was a strong limitation in the ability to reuse software drivers developed for this project.

D. The Host Software Execution to DPI Library

As actual software is not necessarily using read and write functions but rather use address pointers, the DPI2UVM library described above is not enough to run a software without any modification. To avoid complex virtualization techniques or the use of expensive ISS based virtual platform, we opted to develop host code
execution techniques to actually catch memory accesses and redirect them to the dpi2uvm_read() and dpi2uvm_write() functions through a similar API (Figure 6).

The approach was to use Linux memory protection directives and implement a dedicated memory access handler. Since Linux is a common OS for most of RTL simulators, this is not perceived as a limitation. Also, should we need to port this to Windows, we could use similar mechanisms from the Windows API.

The technique consists in protecting the memory area that is used by the software using the mmap() Linux function. Any pointer access to this area will therefore enter in dedicated handler in which we then have the leisure to call whichever function and in our case to call our dedicated DPI functions.

```c
// API Functions
void hse_init_memory_map(hse_address_t addr);
void hse_memory_handler();

// Handler links to external read/write functions
static void hse_memory_handler()
{
    uint8_t temp = 0x0;
    hse_read8(hhs.g_addr, &temp);
}

void hse_read8(int addr, char* data)
{
    if (hse_func.read8 != NULL && hse_init_done != 0) {
        hse_func.read8(addr, data);
    }
}
```

The API is tightly coupled with the DPI2UVM API and both libraries easily map to each other, so that a simple pointer access from C will be mapped to a VIP transaction read or write (Table 1).

In the end, the host execution library is made available as part of the DPI 2 UVM package so that users will only have one API and will not care about this function mapping between the hse_read/write and dpi2uvm_read/write. We kept the HSE library as a separate file though so that we can reuse this bit of the code in other SystemC headless environments.
### SOFTWARE TESTS AND REUSE

By using the above techniques, combining host code software execution to a generic DPI 2 UVM link to the actual AXI VIP, we were able to write C tests as if they were running on the actual processor. This fully enabled both vertical and horizontal reuse of the tests. Vertical reuse from the subsystem level to the system level as the tests running at IP level through the DPI could be reused without modification to be compiled to target the system processor. Horizontal reuse from simulation to the FPGA board as the same tests were used in both platforms. The developed tests further use third vendor HAL and software drivers such as the Xilinx DMA driver without more difficulties than using the software in a software context.

As the model of the algorithm was also available in C, we opted to embed the model into the test, rather than using it as a reference model within a UVM scoreboard. Although not ideal for debugging, this allowed the reuse of the tests onto the FPGA board. As such, it actually preserved the pass/fail criteria on the result comparison without the need for further post processing of the final data from the FPGA memories.

On the timing side, the software which executes on the simulation using the host software execution library does not meet the same timing as on the FPGA. In fact, this software driven approach does not guarantee any timings related to the software execution itself. As this could be seen as a problem, it however guarantees that the software is built properly using the adequate semaphores, wait-for-interrupts and register polling. Additionally, the UVM part of the testbench was dealing with random timings in the execution of the software, so that the software did not have to care about the timing except those related to timing rendez-vous of the interrupts.

Additionally, in order to enable some sort of random testing, an additional API has been created to generate constrained random configurations in the SystemVerilog code (Figure 7).
Since such randomization is no longer reusable at System or the FPGA board level, we could record such generated configurations as configuration files to be able to replay the same test on a FPGA board. Although possible, we didn’t have that need and did not implement this last configuration portability.

IV. CONSIDERED ALTERNATIVES AND COMPARISONS

As host code execution techniques is often used in a SystemC context and as the DPI usage to perform reads and writes is also in the verification engineer toolbox, it’s important to note the differences here.

- **Custom DPI calls**

  When developing DPI links between C and SystemVerilog, users usually find a path from a global scope or a package down to a specific verification IP instance. They then convert the address and data from the C to the required transaction. These are usually not-reusable and at best reusable only to address a specific VIP.

  By using the uvm_tlm_generic_payload and the uvm_tlm_b_transport_port we are able to provide a generic approach to map to any UVM verification IP sequencer, only providing a conversion function and the path to the required sequencer. The API further allows to address multiple masters using a user defined address map if required.

  Custom DPI calls also don’t have the capability to map a C pointer access to an actual read or a write. When this is required, a virtualization technique needs to be implemented.

- **SystemC / TLM virtualization**

  Virtualization on top of SystemC is also possible. This would however either require to buy external virtual platforms or to develop a platform in SystemC. Cost driven project and time to market
pressures did not allow this, while we had a few lines of code available in C and in SystemVerilog to address our needs which was just to drive the AXI bus of the RTL simulation from C. SystemC engineers may also not be available within the division and other alternatives need to be taken.

- QEMU platforms

QEMU platforms are good alternatives to the proposed approach. They further add the advantage of being much more precise in the simulation time. QEMU however relies on a list of known processors that have been modeled and which may not be the ones we have. It requires cross compilation of the software to the targeted platform. QEMU is also a platform that requires some time to apprehend its complexity when you are not a pure software engineer. In [2], the author presented a very interesting approach to address a similar need. However, this approach was tight to the author AXI bus functional model and lacks genericity in its concepts. The configuration of a given QEMU platform also seems not straightforward.

- ISS

Using an ISA Simulator would probably be the most beneficial ways of resolving the hardware/software co-verification of such designs. It would require to be integrated into the SystemVerilog environment and mapped to the DPI calls. This seems reasonable on the technical side. However, ISS may not be available for all hard-macros or CPU provided in the FPGA libraries. They may be bought separately from third providers in which case it becomes a business decision. The approach presented here is straight forward and does not have the internal complexity of an ISS, leading to a cheaper solution in both the development and the integration and better performances.

V. RESULTS

Hardware bugs were found by comparing the expected results from the embedded model with the actual results from the hardware. These would have been found using a more conventional UVM scoreboard and reference model.

The software driven verification approach however uncovered additional bugs that were strongly linked to the hardware / software interactions. One of them was related to the synchronization between the interrupt received from the software and the state machine responsible of writing to the output buffer (Figure 8). As the software was receiving the interrupt too early, it was able to read the output buffer while the hardware was not completely done, resulting in reading non-initialized data from the software.

![Figure 8 End of processing early interrupt bug](image)

Another issue which was found was related to the actual software application running on the FPGA board. At some point the software application developed by the software team did not give the same results as the tests which were developed using the same configuration. But running the verification tests onto the FPGA was giving
the same results than from the simulation platform. Since we had a testbench which was able to rerun the bare metal software without a big effort, we were able to rerun part of the software in simulation to actually debug the case and finally found an issue in the application software.

VI. CONCLUSION

This paper described two techniques, the developed dpi2uvvm link which enables software API mapping to any Verification IPs using UVM TLM transport and conversion functions, and the host software execution library which enables a given software to map pointer accesses to read and write functions. These two techniques, when combined together allow to develop a software driven test approach on top of UVM to address both hardware/software co-verification and hardware/software codesign aspects. Such reuse is both vertical from IP to SoC and horizontal from simulation to the physical board. These techniques are cost effective to setup and do not require the development or the purchase a costly ISS nor a SystemC platform and provides simpler setup than virtualization platforms such as QEMU. In the end, these techniques are not tight to any specific processors and therefore do not rely on the availability of a given virtual processor model.

REFERENCES

[8] Complex Signal Processing Verification under DO-254 Constraints, François Cerisier, Verification Horizon Article on MENTOR newsletter, Sept 2017