electronics

MDPI

Article

Open-Source HW/SW Co-Simulation Using QEMU and GHDL for VHDL-Based SoC Design

Giorgio Biagetti, Laura Falaschetti, Paolo Crippa *, Michele Alessandrini and Claudio Turchetti

Department of Information Engineering, Università Politecnica delle Marche, Via Brecce Bianche 12, 60131 Ancona, Italy; g.biagetti@univpm.it (G.B.); l.falaschetti@univpm.it (L.F.); m.alessandrini@univpm.it (M.A.); c.turchetti@univpm.it (C.T.)

* Correspondence: p.crippa@univpm.it

Abstract: Hardware/software co-simulation is a technique that can help design and validate digital circuits controlled by embedded processors. Co-simulation has largely been applied to system-level models, and tools for SystemC or SystemVerilog are readily available, but they are either not compatible or very cumbersome to use with VHDL, the most commonly used language for FPGA design. This paper presents a direct, simple-to-use solution to co-simulate a VHDL design together with the firmware (FW) that controls it. It aims to bring the power of co-simulation to every digital designer, so it uses open-source tools, and the developed code is also open. A small patch applied to the QEMU emulator allows it to communicate with a custom-written VHDL module that exposes a CPU bus to the digital design, controlled by the FW emulated in QEMU. No changes to FW code or VHDL device code are required: with our approach, it is possible to co-simulate the very same code base that would then be implemented into an FPGA, enabling debugging, verification, and tracing capabilities that would not be possible even with the real hardware.

Keywords: HW/SW co-simulation; QEMU; VHDL; FPGA; SoC

Citation: Biagetti, G.; Falaschetti, L.; Crippa, P.; Alessandrini, M.; Turchetti, C. Open-Source HW/SW Co-Simulation Using QEMU and GHDL for VHDL-Based SoC Design. *Electronics* **2023**, *12*, 3986. https://doi.org/10.3390/electronics12183986

Academic Editors: Alexander Barkalov, Larysa Titarenko, Dariusz Kania and Remigiusz Wiśniewski

Received: 11 August 2023
Revised: 14 September 2023
Accepted: 15 September 2023
Published: 21 September 2023

1. Introduction

At the core of many modern digital electronics devices there is a complex combination of hardware (HW), firmware (FW), and software (SW), often combined within one system-on-chip (SoC) circuit, or even a multitude of different SoCs, each specialized for a particular function. Design tools aim at making the realization of an SoC, from conception to production, as fast and as straightforward as possible, by facilitating reuse of pre-built blocks, commonly referred to as intellectual properties (IPs), and their integration. But the verification and testing of such complex devices remain a challenging task [1], as is the creation of new IPs.

Nowadays, it is ever more common for even critical hardware functionalities to be defined and controlled by software (or firmware); thus, the traditional approach of verifying the HW functionality separately, through digital simulators, and then developing the FW/SW and testing it on a HW prototype of the embedded system does not really work. Moreover, as the development of the three components is not parallelized, it is not compatible with increasingly stringent time-to-market requirements.

Prototyping the HW using reconfigurable logic devices such as FPGAs can certainly help. Indeed, implementing a digital design on a device like an FPGA using pre-built IPs and/or custom VHDL models is an almost completely automated process (timing closure problems aside). But it is still a very slow procedure, taking a considerable amount of time (many minutes), even for relatively simple projects that contain just a simple embedded CPU with a few basic peripherals, making the common edit/compile/verify development cycle quite unpractical.

Of course, instead of testing on an FPGA, it is in principle possible to simulate the whole design, including the CPU, running the desired FW. But simulating a CPU at the

RTL level is again an exceedingly slow and resource-consuming process, suitable only for simple and small FW codebases, and produces often unnecessarily detailed results. In many cases, especially during the initial phases of FW development, the timing details of the code execution are not yet available and can often be disregarded. Considering the tendency of progress to keep making faster processors, it can be considered good practice to develop an FW whose operation does not depend on the exact speed of the CPU it runs on but is able to operate at any speed above a certain minimum requirement. With this consideration in mind, it should be possible to emulate the FW using a much faster instruction-set simulator (ISS) or other even faster emulation techniques (such as real-time binary translation) to run the FW at the highest speed permitted by the host machine, as well as to rely on its interaction with the HW for (approximate) timing.

HW/SW co-simulation, e.g., the technique by which a simulator of the HW part of the design is somehow coupled to an emulator or simulator that runs the SW/FW component, is in principle exactly what is needed to solve all these problems. This would allow for the development of the HW and the SW/FW to proceed simultaneously, even when each one is hardly operable without the other, and the verification of both without relying on hardware prototypes. This would open up enormous possibilities in terms of verification, debugging, and even co-design capabilities, as will be discussed in the reminder of this paper. Unfortunately, despite the topic being well studied and investigated [2–4], setting up a co-simulation environment is often a challenging task in itself, hampering its widespread adoption at all levels of digital design, especially the smaller ones for which the complexity of the setup is not amortized.

Specifically, to the best of Authors' knowledge, currently available solutions mostly target higher-level system description and models written in SystemC, or hardware description languages (HDL) such as Verilog or SystemVerilog, but there are no ready-to-use tools for co-simulating VHDL-based designs. This is a little surprising because such a tool would be very desirable for designers, and, according to a recent study conducted in 2022 by the Wilson Research Group and Siemens EDA [5], VHDL is the most commonly used language for FPGA design and verification and has even gained some popularity over the previous 2018 study.

So, we decided to fill this gap by developing a turnkey solution for HW/SW co-simulation using open-source tools, to maximize its adoption possibilities amongst digital designers at all levels of complexity, including relatively simple SoC projects or small embedded systems for which the conventional approach of going through SystemC could be seen as too cumbersome. It uses QEMU [6] as a fast CPU emulator (it is based on real-time binary translation) to run the FW, and GHDL [7] to simulate the HW. Communication channels are also set up to allow the whole system to interact with SW running natively on the host system. The aim is to maximize simplicity because co-simulation should be seen by the designer as an aid and time-saving tool while focusing on the design and development, not as another complex tool to learn. By developing a few (or just a couple as the bare minimum) simple-to-use modules and making them publicly available [8] under permissive licenses (in general, the host code for the co-simulation framework is released under the GPL-2.0-or-later license, while the VHDL part is released under the more permissive Apache-2.0 license) we believe we have achieved the goal of making the co-simulation of VHDL-based designs simple and readily available to any designer. The purpose of this paper is to describe how this was achieved and to present a case study to exemplify its usage and applicability. To this end, also the examples are released as open source code, the FW being under the BSD-3-Clause license, and the sample HW under the CERN-OHL-W-2.0 license to facilitate its reuse, see `LICENSE.txt` and the `licenses` folder in the Supplementary Materials for the full texts and the SPDX identifiers in each file for further details.

This paper is structured as follows. Section 2 presents a few related works that tackled similar problems, and a brief survey of the available literature on the subject. Section 3 explains the proposed architecture, describes how the whole system was built using just

open-source tools, and explains how they were extended to implement a working co-simulation environment. Section 4 reports on a few application examples to demonstrate the effectiveness of the proposed solution, whose benefits and limitations are further discussed in Section 5. Section 6 concludes the work.

2. Related Works

As briefly mentioned in the Introduction, several studies have been performed on co-simulation using other HDL than VHDL. For instance, ref. [9] presents a co-simulation framework that targets SystemVerilog. It connects a virtual machine backed by QEMU in either virtualization mode (allowing for fast execution of the operating system, device driver, and application software) or emulation mode (allowing for better timing accuracy at the cost of very slow emulation) with a commercial HDL simulator. The interconnection is based on the PCIe protocol, emulating a server system with a PCIe-connected FPGA.

In [10], a meet-in-the-middle methodology was used to model and simulate heterogeneous smart devices. From a set of components belonging to different design domains and expressed in a heterogeneous set of abstraction levels, the proposed approach exploited automatic translation, abstraction, and integration to reconcile the heterogeneous set of component models into a single homogeneous system-level model or used multi-language commercial simulators to synchronize separate models.

SystemC [11] is also a very common target language for co-simulation [12–14]. Although it is not strictly an HDL, it can be used as such, and being a set of C++ classes, it is widely versatile, lending itself to many different simulation scenarios, including embedded analog and mixed-signal (AMS) systems [15,16] and allowing easy access to the underlying C language (and hence also all the operating system) communication facilities to interact with other simulators.

In [17] an emulation framework based on QEMU and SystemC is proposed. The connection between SystemC and the SW emulator can be made either using TLM channels or using RTL interfaces, according to the current level of design refinement. Several constraints still exist in the QEMU-SystemC interaction, e.g., it is incapable of cycle-accurate synchronization between the CPU and the HW. To overcome these constraints, in [18] a fast, cycle-accurate instruction set simulator (CA-ISS) based on QEMU and SystemC is proposed. This was obtained by making QEMU play the role of an ISS, sending information such as the instructions executed and fetched, the condition code flags, and the memory addresses to SystemC. QEMU was also used in [19] as an instruction-accurate instruction set simulator (IA-ISS) and interfaced with SystemC to facilitate co-design of hardware described in SystemC at the electronic system level (ESL) level.

More recently, SystemC was also used in [20], where a platform for fast co-simulation of FPGA-based SoCs has been presented. The platform uses QEMU to emulate the software and achieve a reasonable simulation speed, a transaction level modeling TLM 2.0 to simulate communications, and SystemC to describe the hardware. In particular, a design comprising CPU, two PLLs, RAM, and an ADC was developed, and the authors noted that using SystemC instead of VHDL provided a significant speed advantage, taking the same time to simulate the complete design in SystemC as just a single RTL module in VHDL.

Another line of research about co-simulation starts from the redefinition of the design stage. System Python (SysPy) is indeed a public domain design tool that uses Python to facilitate all stages of prototyping processor-centric SoCs for FPGAs. In [21], it was shown how HW/SW co-design and the verification of SoCs for FPGAs are facilitated by SysPy, if used as an architecture description language (ADL), which helps designers make decisions about key architectural features early in the design phase. The tool also supports the algorithmic joint modeling of hardware and software elements of an SoC (processing and control logic) using popular Python libraries such as SciPy and NumPy following a Matlab-like syntax.

Finally, a few attempts to somehow include VHDL designs in a co-simulation environment have been attempted, though they are not as mature as SystemC-based ones.

For instance, ref. [22] describes a method that uses Xilinx's fork of QEMU. This fork includes a device called "Remote Port" that can stream I/O accesses to external simulators, but they only provide a SystemC-TLM implementation for the receiving end. To drive a VHDL design, another bridging and wrapping layer was needed, adding to the complexity of the environment, and it only worked on a very specific proprietary simulator. In [23], on the other hand, the authors describe a method that couples their VHDL BFMs to a C++ ISS for the RISC-V open-source architecture.

3. Materials and Methods

Although the proposed solution should be general enough to be applied to SoC design, embedded system designs, and FPGA designs, we will focus our attention on the latter for the sake of simplicity. In particular, the scenario of an embedded system developer, who needs to design some custom peripherals using the programmable logic (PL) in the FPGA, will be considered. These peripherals are directly controlled by some FW running on a CPU, also known as the programmable system (PS). The CPU can either be implemented as a soft core using a portion of the PL or be already present as a hard core inside the FPGA; it does not really matter at this level of abstraction. Given the popularity of the ARM ecosystem, it is nowadays increasingly common that the interface between the PS and the PL consists of some variant of the AMBA AXI bus [24], even when the core is not an ARM CPU itself, so we will also focus on that. It should be trivial to extend the technique to other buses provided an appropriate VHDL BFM implementation exists (or can be written). Finally, provisions are also made for the embedded system to be able to be connected to a host PC for debugging and/or control purposes.

We will focus on using GHDL [7] as the HW simulator for the PL, and QEMU [6] to emulate the FW running in the PS, while the SW part will run natively on the host system. Development was done using QEMU version 8.0.0 and GHDL version 3.0, hosted on an Ubuntu 22.04 LTS Linux system, but should be easily portable to other platforms.

The basic idea is to use QEMU to emulate the embedded CPU so that it can execute an unmodified copy of the FW image, while intercepting accesses to peripherals. These accesses are then serialized over an IPC channel and conveyed to the GHDL side, where the said HW peripherals are simulated, with replies and interrupt requests (IRQs) sent back to QEMU.

The main contribution of this work is thus the development of a few modules to make the interaction between QEMU, GHDL, and application SW simple and straightforward. There are two main modules that make this co-simulation possible, implementing the IPC between QEMU and GHDL, plus a third optional utility entity that eases direct HW-SW communication. In detail, the three modules are:

- **RTLbridge**: a QEMU device module that can be added to custom machines and that exposes a user-configurable memory-mapped I/O region to the HW simulator and routes IRQs back from the HW simulator to the emulated CPU. It uses the generic QEMU "chardev" abstraction to communicate with the external simulator. Written in C and licensed under the GPL-2.0 as it needs to be compiled within QEMU itself.
- **CPUemu**: this is a VHDL entity that speaks the RTLbridge protocol over UNIX named pipes and exposes an AXI bus manager interface to the HW domain (it also takes care of handling clocks and reset signals). It is written in VHDL-2008 and licensed freely under the Apache-2.0 license so as not to interfere with the rest of the digital design. It uses the facilities provided by the UVVM framework [25] as helper functions to implement the AXI BFM.
- **PTYemu**: this is an optional VHDL entity that can help couple the HW simulator to the application software or device driver running on the host. It exposes a UART BFM to the HW side and a PTY to the host and is written partly in VHDL (for the BFM) and partly in C (for the PTY-related code), coupled by means of GHDL's VHPIDIRECT interface.

Figure 1 shows a high-level overview of the interaction of all these modules, which will be described in detail in the following.

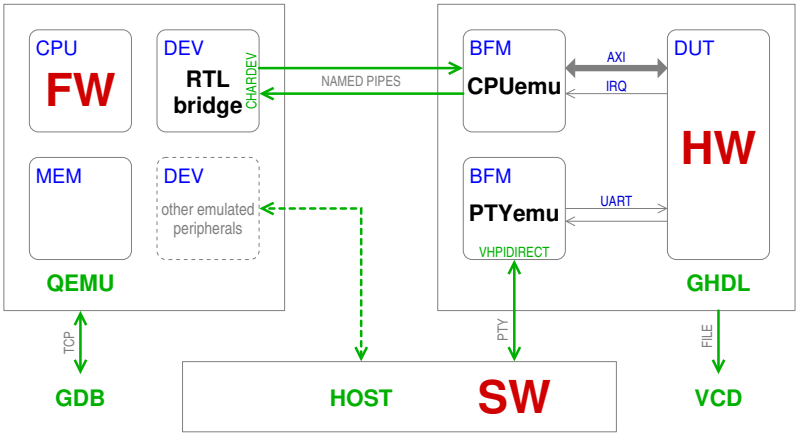

Figure 1. Overview of the modules and of their interactions in a full HW/FW/SW co-simulation environment: The VHDL HW model is simulated by GHDL. The top-level design must include the DUT itself and our CPUemu entity to connect it to the AXI bus of the emulated CPU, where the RTLbridge virtual device takes care of handling the IPC protocol between QEMU and GHDL and synchronizes the two simulations. The DUT can also connect to our PTYemu entity, which emulates a serial port routed to the host by means of a PTY, for direct HW-SW interaction not mediated by QEMU. Standard QEMU-emulated peripherals can also be added to the virtual machine for other FW-SW communication channels not captured in the VHDL model. In general, no changes are required in the DUT HW model, FW code, and SW code between the co-simulation environment and a real hardware implementation.

3.1. QEMU Architecture and Modifications

QEMU is "a generic and open source machine emulator and virtualizer" [6] capable of emulating an entire PC and many of its peripherals even on a system with a completely different architecture. It can be used in "virtualization mode", whereas the emulated machine has the same architecture as the host, and in this case it can take advantage of virtualization extensions implemented on modern CPUs to reach near-native speeds. But, and this is our use case, it can emulate a different CPU by means of real-time binary code translation, the so called TCG (which stands for tiny code generator). It converts blocks of code intended to be run on the emulated CPU into instructions for the host CPU and then executes them, setting up the host memory management system so that accesses to designated memory ranges are trapped and can be emulated by calling device drivers compiled into QEMU itself [26].

This feature was leveraged to implement the RTLbridge module. From the QEMU point of view, it is just a device that maps the entirety of the I/O space of the CPU. But instead of emulating the behaviour of the device inside QEMU, it relays read and write requests to an external simulator, like GHDL.

The difficulty in doing this is that the emulated CPU must wait (think of read operations) for the reply, and so must be stopped, while the external simulator "computes" the reply. QEMU uses multiple threads to perform its tasks: there is one main thread, one I/O thread (for I/O operations towards the host), and at least one CPU thread that runs the emulated (translated) code (it can be configured to use one CPU thread per emulated processor). The problem is that this threading mechanism was added after much of QEMU had already been developed, and parallelism has not yet been fully exploited. In particular, when the CPU thread is blocked, the I/O thread is also stalled on the so-called big QEMU lock (BQL), making it impossible for the CPU thread to receive notifications from the external simulator.

To overcome this BQL problem, another thread must be created to communicate with the HW simulator. This will take care of handling the communications, using blocking read and write operations, and of awakening the CPU thread when the transaction results are ready. But there is another problem.

QEMU has, amongst other mechanisms, a generic abstraction layer to allow the virtual machine it emulates to communicate with the host it runs on, the so-called `chardev`. A `chardev` transports a stream of data, like an ordinary Unix character device, hence the name, and it is split into a front-end (facing the virtual machine) and a back-end (facing the host). Different back-end implementations are provided, such as PTYs, sockets, files, and named pipes. In principle, the abstraction provides generic methods callable by the front-end to perform operations such as reads, writes, and receiving notifications of incoming data, independently of the actual back-end used. But in the QEMU version 8.0.0 implementation, only the socket back-end supports the "read" operation, which is needed by our communication thread. And sockets cannot be accessed natively by a VHDL model (VHDL was never intended to be a general-purpose language, unlike SystemC, which is based on C++ and thus inherits all of its underlying capabilities). The `textio` module of VHDL-2008 can instead easily access named pipes (they act like files), so one further modification of QEMU was necessary to also implement reads for the pipe backend. Alternatively, a traditional utility such as `socat` could be used to "convert" sockets to pipes, but it would add another process in between so that two IPCs would be involved per transaction, potentially slowing things down and certainly complicating the execution of the co-simulation.

So, in summary, from the QEMU side these modifications are necessary to enable co-simulation:

- The development of the RTLbridge device driver that creates the IPC thread and implements the PS side of the communication protocol with the HW simulator;
- The implementation of the missing "read" methods for the chosen `chardev` backends ("pipe" in our case);
- The definition of a custom machine type that allows the insertion of the RTLbridge device and specifies the processor to emulate.

As can be seen, the changes to the QEMU source tree are intentionally quite small to ease porting to future versions. All the required patches and newly written modules are provided as Supplementary Materials to this paper, and possible future updates can be found online at the repository https://github.com/giorby/cosim, together with some examples.

In the end, a typical invocation of the QEMU emulator modified to co-simulate some HW with an ARM embedded processor looks as follows:

```
1  qemu-system-arm -monitor stdio \
2    -gdb tcp::1234 \
3    -machine fpga -m 256 \
4    -icount shift=3,sleep=on \
5    -chardev pipe,id=rtllink,path=/tmp/test/fifo \
6    -device RTL-bridge,chardev=rtllink,base=0xE0000000 \
7    -device loader,file=/tmp/test/code.elf
```

where line 2 enables the GDB interface for FW debugging and line 3 selects our custom machine type (called "`fpga`") and specifies the amount of processor local memory (256 MiB). Line 4 enables instruction counting for synchronization, emulating a single-clock-per-instruction processor running at 125 MHz (the 2^3 ns clock period, as defined by the `shift` parameter). Line 5 declares the `chardev` that will be used to communicate with the VHDL side, in this case using the pipe backend (the two pipes `/tmp/test/fifo.in` and `/tmp/test/fifo.out` must be created separately before invoking QEMU; we provide a script that takes care of these details). Finally, line 6 instantiates the RTLbridge device, associating it with the pipe previously declared. Optionally, the base address and aperture of the simulated I/O space can be specified. Line 7 then loads the firmware.

3.2. Communication with an External Simulator

Each time the emulated CPU initiates a transaction to access the VHDL-modeled memory region, either the "read" or "write" method of the virtual device is called. These methods are passed a 64-bit physical address a of the start of the transfer, the transfer size s (either 1, 2, or 4 bytes for a 32-bit CPU), and the data d to be written in case of writes, or must return the data d in case of reads. QEMU does not yet support outstanding (pending) transactions, and this eases the serialization of the requests. These have been encoded as simple ASCII commands so that they can be easily parsed and formatted from the VHDL side using the standard `textio` library. Scheme 1 describes the IPC protocol we defined. Each "packet" is terminated by the standard CR-LF sequence, and for a 32 bit CPU they are all either 12 B long or 24 B long in case of write requests, simplifying buffering and parsing in case non-boundary-preserving IPC I/O systems are used.

request (QEMU → VHDL)	**reply** (VHDL → QEMU)
`W:` a `<=` d `|` m	`W=` `OK` [1]
`R:` a	`R=` d
`T:` Δt	`T=` t
`X:` v	`X=` r
(asynchronous event)	`I=` IRQ
`···` → HEX number	`···` → ASCII string

a → address
d → data
t → time (µs)
$m = (2^s - 1) \cdot 2^{a \bmod 4}$
$v \in \{\texttt{RESET}, \texttt{STOP}\}$
$r \in \{\texttt{RESET}, \texttt{RUNNING}\}$

[1] Actually, bus accesses can fail, and thus an error code could be reported. But in the current implementation of the BFM we used (from UVVM), these error conditions raise alerts but are not propagated to the VHDL caller. So, for now the only response code that is ever transferred through IPC is actually "OK" as errors cause the simulation to stop. This is not normally a problem, except for FW portions that actually probe the HW to auto-detect peripherals or to perform similar tasks, for which a different BFM implementation is needed.

Scheme 1. The text-based IPC protocol used between QEMU and VHDL.

There, a and d are the main data being exchanged and are hex-encoded in 8 characters (32 bits), m is the 4-bit write mask to activate individual byte lines for short writes, computed from the write size s and address alignment as shown beside the table; t is the current VHDL simulation time (in microseconds); Δt is the amount of time to let the VHDL simulation advance; v is a verb to request special actions (like resetting the HW or stopping the simulation); and r is its corresponding status code.

Asynchronous events, such as interrupts, are special in that they can happen at any time outside of the control of QEMU. They are intercepted by our communication thread but cannot be acted upon from there due to the usual BQL. So, another (internal, unnamed) pipe pair is created to allow these events to be queued from the communication thread, while the read end of the pipe is added to QEMU's list of sources to monitor for events. This way, the event can be read back from the main I/O thread, where the IRQ line of the emulated CPU can actually be asserted. More details will be given in Section 3.4.

The PL side of this protocol is implemented in the CPUemu VHDL module, which decodes it and performs the requested bus access using an appropriate BFM, as will be shown below.

3.3. VHDL Bus Functional Models and Control

The traditional method of verifying a VHDL design (or any other design captured in an HDL in general) is through a test bench that exercises the DUT with proper stimuli. Given the difficulty of directly writing these stimuli, verification frameworks that ease this task for commonly used buses, and add many convenience functions to aid in the checking of results and in the execution of complex tests, have emerged.

The two most known for VHDL are OSVVM [27] and UVVM [25]. We chose to use the latter because of its simplicity and smaller footprint, especially in the "Light" version, to emulate the manager side (the one connected to the CPU) of the AXI bus. UVVM has many modules, but the one that matters the most for our project is the BFM set. For each bus being emulated, it provides convenient parameterized types to help declare all the

required signals, and simple procedures to execute transactions on the bus. Performing a write operation on an AXI Lite bus is as simple as calling a VHDL procedure:

```
axilite_write(addr, data, mask, "CPUemu", clk, axi_if);
```

where the signals in the call have an obvious meaning, and reads are even easier:

```
axilite_read(addr, data, "CPUemu", clk, axi_if);
```

But instead of writing a test bench that performs a sequence of transactions, these calls have been inserted inside a process that reads, from the communication pipe, the commands described in Scheme 1 and performs the corresponding operations. "`W`" and "`R`" correspond to the calls just mentioned, "`T`" is just a `wait on irq for` Δt `* 1 us;`, and "`X`" either asserts the reset line or terminates the simulation. After each transaction or task is finished, a separate process (to avoid deadlocks with blocking I/O) writes the proper reply back to the pipe, and monitors the IRQ line to asynchronously send notifications when it changes state. A clock and reset signal generator completes the essence of the `CPUemu.vhdl` module.

Besides the AXI-Lite BFM to enable HW-FW interaction, we also developed the `PTYemu.vhdl` module (coadiuvated by some foreign functions defined in `PTYemu.c`) to allow for direct HW-SW interaction, with the SW running natively on the host system and communicating with the HW through a basic UART serial interface. In this case, we did not use the UVVM-provided UART BFM but rolled our own since we needed to detect and process some special events such as breaks and an idle line. The data passing through this module are then forwarded to and from the host through a pseudo terminal, whose master end is opened and handled by the C code linked to the VHDL model, and its slave end is available for the SW to access as an ordinary serial port.

3.4. Interrupts and Time Synchronization

As previously mentioned, keeping two independent simulators synchronized is a difficult task. And, in many circumstances, it should also not always be needed—modern CPUs do not have easily predictable execution timings due to caches, speculative execution, branch prediction, etc.—so well-designed firmware should not rely on instruction timing for anything but the most time-critical tasks (for which an upper time limit can be necessary, but in general not a lower limit) and should use HW timers and events for general timekeeping.

So, we let the emulated CPU run at the maximum speed allowed by QEMU but enabled the instruction count feature of the TCG so that it would be possible to have an idea of the amount of emulated time spent by the CPU. The VHDL simulation is instead advanced "on request" by QEMU. Bus transactions advance the simulated time by the minimum required number of clock cycles that are needed to complete the transaction. This is possible because the VHDL process that implements the PL side of the communication protocol (whose PS side is implemented in the RTLbridge QEMU device driver) blocks on reads from the communication pipe, thus pausing the VHDL simulation until a command is received.

This works well for performing fast bus accesses but inhibits progress of the digital simulation when the FW is not performing transactions. There are two typical scenarios in which the FW might not carry out transactions for a prolonged amount of time: it is either busy doing some intensive computations (or a busy wait, though it is not good practice), or it is idle and has stopped the CPU with a kind of "wait for interrupt" instruction (e.g. "WFI" for ARM processors, or the "SLEEP" MicroBlaze pseudo-instruction). To handle these cases, the RTLbridge device sets up a timer using QEMU's virtual clock (`QEMU_CLOCK_VIRTUAL`).

This virtual clock is meant to mimic the simulated time in the emulated CPU, and it is advanced by 2^n ns for each increment of the instruction counter, with n being configurable through the `shift` parameter to the `icount` option. Actually, the exact mechanics of virtual clock handling are quite complicated: for efficiency reasons it is not really incremented after each emulated instruction but only in batches, but with safeguards in place to ensure timers are fired without too much slack.

So, in principle, every time this timer fires, it should be possible to advance the VHDL simulation to bring the VHDL idea of simulated time in sync with QEMU's. This could

work well for busy loops but fails for wait instructions. When the CPU is halted, as a matter of fact there are no instructions being executed to count, so QEMU instead advances the virtual clock at the same rate as the real time clock in the hope of making real-time full system emulation work at the proper speed. Unfortunately, in most cases of practical interest, a VHDL simulation is in itself much slower than real time, making it impossible for VHDL to catch up with QEMU while a "wait" instruction is being executed.

To avoid this problem, we chose not to try and keep the two simulators' idea of time in sync. After all, since it is not provided by our RTLbridge module, the FW has no way to access QEMU's idea of time, and if it needs a time reference, it should query the HW side that has the correct notion. So, the VHDL simulation is just advanced by a fixed quantum of time (which can even be shortened by HW events or interrupts if need be) for every firing of the QEMU timer, independently of how much time QEMU thinks has elapsed. This way, the period of the timer only affects simulation performance and not the overall function of the co-simulation. Shorter periods lead to more QEMU to VHDL communication, and thus more overhead, while larger periods may increase the interrupt processing latency during busy loops or delay the detection by the FW of HW events if the FW uses polling. Still, these delays only affect the FW "perceived" amount of time passed (number of instrunctions or loops being executed), but the FW is not supposed to rely on that method to gain knowledge of the passage of time. No problems should arise if the FW timing is based on reading any HW-based timer or RTC.

In any case, this parameter can be adjusted from the command line (it is the `sync` parameter to the `RTL-bridge` device) so that it can be easily tweaked if there is the need to optimize co-simulation speed. We found experimentally that values between 1 µs, which is the same as the VHDL quantum of time, and 1 ms tend to work well with relatively simple HW models such as those presented later, and its effect on simulation speed is often quite modest so an accurate tuning is not necessary. In general, higher should be safer, so the default was set to 1 ms.

3.5. Running the Co-Simulation

With all this framework in place, performing a co-simulation becomes very easy and straightforward. The Supplementary Materials provide compilation scripts that automate downloading, patching, configuring, and compiling the correct versions of both QEMU and GHDL. If the user wants to simulate designs that contain low-level primitives often found on FPGAs (DSP slices, BRAM, etc.), then of course the vendor-specific libraries are also needed and, due to licensing restrictions from their vendor, must be installed separately (and before compiling GHDL). The `ghdl/compile` compilation script tries to detect the presence of the Xilinx Vivado 2023.1 simulation libraries and, if found, also compiles them for GHDL use. The `qemu/compile` script, on the other hand, prepares the QEMU emulator.

Once these two scripts have been succesfully run, the system is ready to perform co-simulation. Just compile the FW in a format supported by QEMU (ELF usually works well as it also allows for debugging and is the default output format of GCC), provide a suitable VHDL testbench that instantiates the `CPUemu` module and connects it to the HW being tested, compile the VHDL code, and run the two simulators as detailed previously. To facilitate the task, the QEMU compilation script also automatically generates a run control script (under `/tmp/test/run` by default) that creates the pipes and passes all the required options to QEMU before launching the VHDL simulation, so running a co-simulation is usually just a matter of invoking this script with the FW and VHDL executables as arguments.

Please refer to the `README.txt` file in the Supplementary Materials for further instructions, and to the `Makefiles` in the various subdirectories within `examples` for guidelines on how to compile the design-specific FW and HW components.

4. Application Examples

In this section, a few examples showing how the modules we developed can be applied in a real-world practical design are reported. Although the actual target CPU is a Xilinx

MicroBlaze, in the examples it has been replaced by an ARM processor as it has the most readily available FW development toolchain, easing the reproducibility of our examples. In the end, only a few lines of platform-specific FW code (basically just the interrupt enable/disable/wait instructions) should be changed between the two architectures, and since both expose an AXI bus, no changes at all are required in the HW description of the peripherals that will be implemented.

As a first example, a very simple peripheral, i.e., a timer/PWM module, just to obtain an idea of how the co-simulation framework puts things together and to analyze possible timekeeping/synchronization issues, is presented. Then, a relatively more advanced but still quite simple HW design—a UART transceiver—will be discussed. This was chosen because of its almost universal usefulness as a basic communication subsystem and because it is a fundamental part of the final design. Indeed, as a final application, a complete SoC comprising an AXI crossbar to connect the previously analyzed PWM modules; the UART transceiver; some shared (dual port) memory with its SRAM controller; an interrupt controller; and the DAQ controller will be described. It is meant to be used as a digital data acquisition system for capturing data sampled by external high-speed (relative to UART speed) ADC converters, storing them in memory, and then transferring them over the serial port.

4.1. QEMU-GHDL Synchronization Example

In Section 3.4, it was mentioned that the proposed framework does not attempt to reach cycle-accurate synchronization between FW and HW—doing so is also beyond the possibilities of QEMU, as it is designed to be fast, not time-accurate—so it can be useful to provide an example of what it means from a FW development point of view.

To this end, a simple timer/PWM peripheral is considered. The HW side is trivial, comprising just three AXI-accessible 32-bit registers: `count`, `period`, and `value`. The first is a read-only counter that is incremented at each clock cycle and is reset to zero when it reaches the value stored in `period`. An IRQ is raised when `count` is reset and cleared when read. The last register is double-buffered, and the PWM output is brought high whenever `count` $<$ `value`, so that 0% PWM can be achieved with `value` $= 0$ and 100% PWM with `value` $=$ `period` $+ 1$ or greater. The complete VHDL sources are in `examples/PWM/hw/pwm.vhdl`.

Figure 2 shows a code snippet of how this module can be used to generate an approximation of a sine wave.

```
30  // 12-point "sinusoidal" waveform, amplitude = 100, offset = 125:
31  static const uint8_t pwm_values[] = {125, 174, 211, 224, 211, 174, 125, 76, 39, 26, 39, 76};
32  static const uint8_t num_values   = sizeof pwm_values / sizeof *pwm_values;
33
34  void timer_isr (void)
35  {
36      static unsigned phase = 0;
37      uint32_t time = tmr->count;
38      tmr->value = pwm_values[phase++];
39      if (phase >= num_values) phase = 0;
40      if (count_events(time) > 26) event_set_nolock(TMR_SEQUENCE_FINISHED);
41  }
42
43  int main (void)
44  {
45      tmr->period = 250 - 1;
46      wait_for_event(TMR_SEQUENCE_FINISHED);
47      return 0;
48  }
```

Figure 2. Example code to test the synchronization between QEMU and GHDL for an HW-driven process such as IRQ-based PWM generation. Excerpt from `test-pwm.c`; see the Supplementary Materials for the complete sources. The `count_events` function was meant to compute statistics of the ISR entry latency, besides counting the events, of course. The entry latency was found to be 2 clock cycles—the time required to setup an AXI read transaction—because for this short code QEMU is much faster than GHDL, and so from the VHDL point of view no time has elapsed at all between IRQ assertion and the reading of the `count` register.

To interact with IRQs, a few platform-specific helper functions have been defined. The `wait_for_event` call at line 46 is basically a wrapper around the `WFI` ARM instruction, within a loop that awaits for the assertion of the specified event flag. Such event flags are asserted by a call to `event_set_nolock` (or just `event_set` if called with interrupts enabled). Line 40 does this after 26 IRQs (a little more than two complete cycles), to stop the simulation.

For these examples, we set the HW clock to 100 MHz, so a `period` of 249 makes the PWM run at 400 kHz. After running the co-simulation and saving the HW-side signals to a VCD file, it is possible to obtain the waveforms shown in Figure 3.

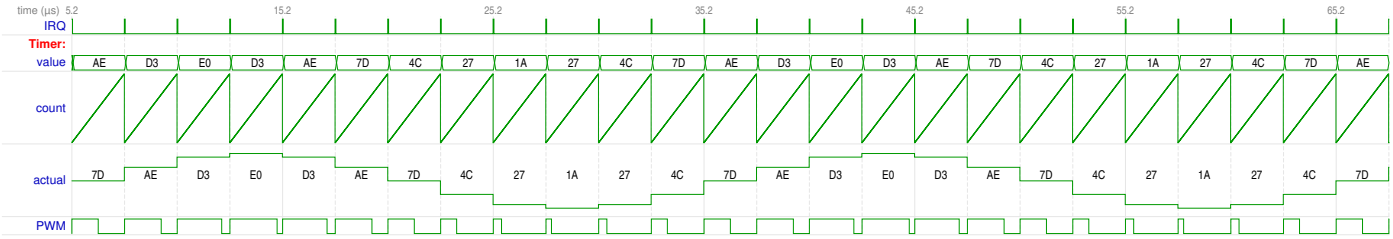

Figure 3. Waveforms resulting from co-simulating the FW shown in Figure 2. `value` is the PWM value written by the FW, while `actual` is its double-buffered counterpart updated when `count` is reset to 0.

As can be seen, the IRQ processing latency is very low, actually too low: after IRQ is asserted, it is acknowledged in two clock cycles, and then `value` is immediately updated. This is because from the VHDL point of view, FW execution takes no time at all. It is as if the CPU runs at infinite frequency: the VHDL simulation is advanced in "batches" when the CPU is idle or has executed a certain number of instructions, and such a short ISR code does not trigger the instruction count limit (the QEMU timer does not expire). Still, the FW works as expected, and we believe that a well designed FW should not misbehave if run on a faster CPU than originally planned, so this should not be a problem in most scenarios.

The only cases in which a working FW could misbehave in simulation are if it tries to time operations by means other than HW timers, e.g., by using on-CPU time stamp counters or performance monitors, or by counting instructions. For example, if the "WFI" instruction in the wait loop is replaced by a register increment, on average the count reaches around $1.6 \cdot 10^6$ in the 67.5 µs the sequence supposedly lasts, i.e., $25 \cdot 10^9$ increments per second. This is due to the fact that the QEMU timer, by default, expires every 1 ms but only advances VHDL simulation by 1 µs, so the virtual CPU "appears" to run at 125 GHz instead of 125 MHz (there are five assembly instructions in the loop). These particular values have been chosen not to hog the host with IPC calls between QEMU and GHDL because of the fact that GHDL is usually much slower than QEMU, and can be easily tweaked if need be. But trying to synchronize two processes every 1 µs would waste most of the computational resources in IPC, while advancing the HW by 1 ms could create unacceptable latencies (in case the FW polls the HW instead of using interrupts), so a compromise must nevertheless be chosen.

4.2. High-Speed Deep FIFO-Based UART Transceiver

As a more practical application, the co-simulation tools we developed had been used to help design and verify a deep FIFO-based UART transceiver. The target for the design is a Xilinx Artix-7 FPGA `XC7A35T-1CPG236C`, included into a Digilent `Cmod A7-35T` FPGA module. The module also features 512 KiB of SRAM, 4 MiB of flash memory usable for FPGA configuration and FW image storage, and an FTDI `FT2232HQ` high-speed USB to UART and JTAG converter.

The final application will be an embedded data acquisition system built around the FPGA, using Xilinx proprietary MicroBlaze soft core as the CPU for the PS and the remaining PL for custom peripherals. There is also the need to stream the acquired data

to a controlling PC that runs the SW part for debug and control purposes, but since the Xilinx-provided mini-UART implementation has a maximum baud rate of 921,600 Bd on this FPGA, while the `FT2232HQ` can work at rates up to 12 MBd, we decided to design a high-speed UART transceiver capable of operating at 12 MBd with a bus clock of 120 MHz (the upper limit for the MicroBlaze CPU on the "`-1`" speed grade parts mounted on Digilent's modules). To ensure the firmware can keep up with such a throughput, deep (2048 9-bit characters) FIFOs have been implemented on the RX and TX channels using one 4.5 KiB BRAM tile (one half for each direction), and an advanced idle/special event detection and signaling mechanism was devised to try and minimize the number of IRQs sent to the CPU, using the 9th bit in each queued word to flag these special events.

For reference, a simplified block diagram of this peripheral is shown in Figure 4. It is essentially a standard design with the addition of the "special event counter" that can assert an interrupt whenever an event, like an idle condition or a break, is present in the FIFO, so that the FW can promptly process incoming data as soon as each burst ends. For completeness, a brief summary of the "`fastUART`" control register set we devised is reported in Figure 5 (for compactness, not all of the implemented flags are shown in the simplified block diagram, "line", e.g., sets or reports the line level during pauses, i.e., breaks and idle conditions; the remaining bits should be self-explanatory).

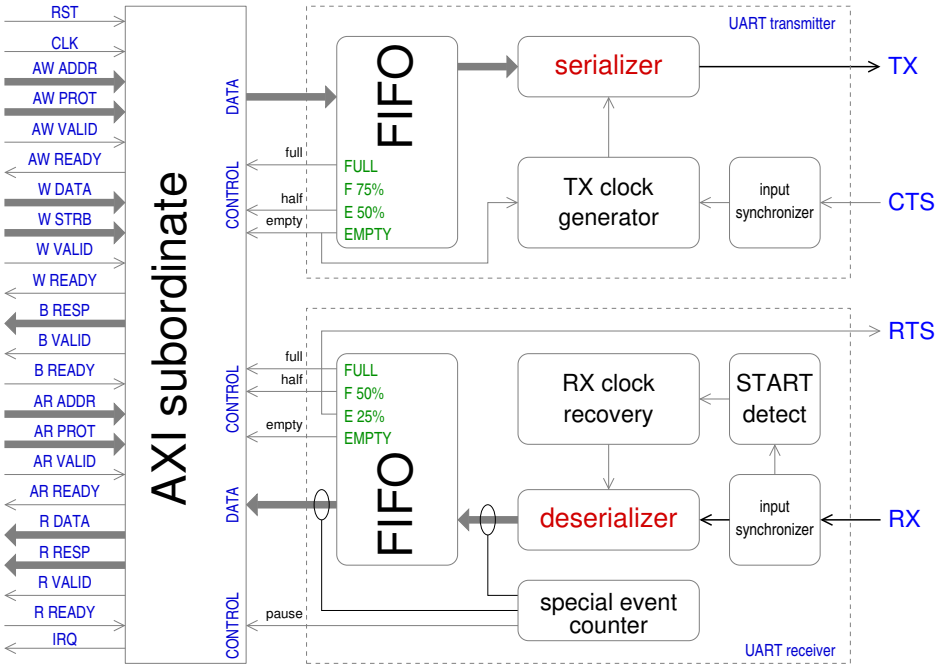

Figure 4. UART simplified block diagram.

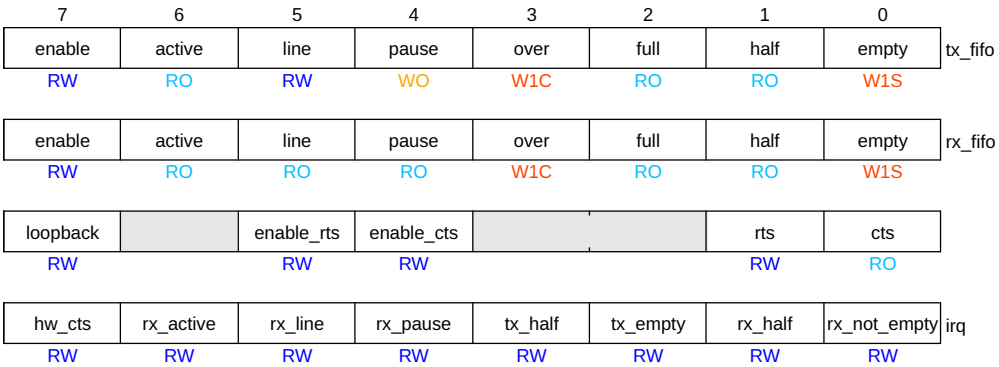

Figure 5. UART control register bit assignment. From top (LSB) to bottom (MSB): TX and RX FIFO status and control, flow control, and individual IRQ enable flags. RW: read/write; RO: read only; WO: write only; W1C: write 1 to clear; W1S: write 1 to set (clears the FIFOs).

The data register allows writes of any length (8, 16, or 32 bits) to ease FW development, and each written byte is queued in little-endian order. Reads return one byte at a time, with one extra flag bit to signal special events. We indeed chose not to use DMA because in the Authors' experience it is seldom useful for the asynchronous protocols typically used over UARTs and would offer little advantage over a deep FIFO but at a considerable cost in terms of FPGA resource usage.

The full, synthesizable VHDL model of this serial interface can be found in the `examples/fastUART/hw` folder on the project repository, where the complete description and device drivers are also included. In the following, a couple of FW examples, the first one to test and validate the HW, the second one using the simulated HW to help develop the FW, will be examined.

4.3. Co-Simulating the HW and FW

In this section, a sample C code to exercise the different registers is examined. Figure 6 shows an excerpt that writes data to the TX FIFO with different access widths and alignments, sends some special characters by manipulating the control register bits, and enables the hardware loopback feature (that internally connects the RX line to the TX line) to read back the written data through the RX FIFO. After that, an interrupt-driven transmission is also tested (started by the `uart_post` function). The data read back will be examined in memory; hence, the buffer variables have been declared `volatile` as a quick way to ensure they are accessible from the debugger and do not get optimized away by the compiler.

```
17  static volatile char    rxbuf[256];
18  static volatile uint8_t rxcnt;
19
20  void loopback_test (void)
21  {
22      // enqueue some test signals on the TX FIFO:
23      uart_data->word = 0x78563412;  // basic 32-bit access
24      uart_data->half = 0xCDAB;      // basic 16-bit access
25      uart_data->byte = 0xEF;        // basic  8-bit access
26      uart_data->test = 0x9291;      // high-half 16-bit access (no need to use, just a test)
27      uart_data->bytes[0] = 0xB0;    // individual byte lane access (unneeded... just a test)
28      uart_data->bytes[1] = 0xB1;    // individual byte lane access (unneeded... just a test)
29      uart_data->bytes[2] = 0xB2;    // individual byte lane access (unneeded... just a test)
30      uart_data->bytes[3] = 0xB3;    // individual byte lane access (unneeded... just a test)
31      uart_control->tx = send_error; // special symbol: writes 0xF0, queues 0x1FE.
32      uart_control->tx = send_break; // special symbol: writes 0x90, queues 0x1C0.
33      uart_control->tx = send_idle;  // special symbol: writes 0xB0, queues 0x1C1.
34
35      uart_control->loopback       = 1;
36      uart_control->irq.tx_empty   = 1;
37      uart_control->rx_fifo.enable = 1;
38      wait_for_event(uart_tx_done);
39
40      // read back the characters received so far:
41      for (uint16_t value; (value = uart_data->read) != uart_fifo_empty; )
42          rxbuf[rxcnt++] = value; // also stores control characters!
43
44      // now start interrupt-based transmission test:
45      uart_recv(true);         // enables rx_half and rx_pause interrupts
46      uart_post("Hi\r\n", 4); // enables tx_half interrupt, actual loading of FIFO is in ISR
47      wait_for_event(uart_rx_ready);
48
49      // and read the remaining received characters:
50      for (uint16_t value; (value = uart_data->read) != uart_fifo_empty; )
51          if (value < 0x100) rxbuf[rxcnt++] = value;
52  }
```

Figure 6. Example code to test the interaction between C register access and VHDL bus transactions. Excerpt from `test-bfm.c`; see the Supplementary Materials for the complete sources.

We also need a VHDL test bench that instantiates the `UART_interface` module and connects it to the emulated CPU (through `CPUemu`). Since we plan to perform further tests beyond a simple loopback, it also connects the serial lines to an emulated serial port on the host through `PTYemu`. An excerpt of this VHDL code, provided to illustrate how easy it is to setup the environment, is reported in Figure 7: the designer only needs to connect

the three modules, no further code is necessary (besides the mandatory library declaration, omitted for brevity from the figure).

```
architecture structural of testbench is
    -- CPU interface:
    signal clk      : std_logic;
    signal rst      : std_logic;
    signal irq      : std_logic;
    signal axi      : t_axilite_if( ... omissis ... );
    -- serial lines:
    signal host_rx  : std_logic;
    signal host_tx  : std_logic;
begin

    dut : entity UART_interface
    port map (
        S_AXI_•     => clk, rst, axi, (compacted)
        uart_tx     => host_rx,
        uart_rx     => host_tx,
        irq         => irq
    );

    cpu : entity CPUemu
    generic map ( fifo_path => "/tmp/test/fifo" )
    port map (
        M_AXI_•     => clk, rst, axi, (compacted)
        M_IRQ_LEVEL => irq
    );

    host : entity PTYemu
    generic map ( pty_path => "/tmp/test/pty" )
    port map ( rx => host_rx, tx => host_tx );

end architecture;
```

Figure 7. Skeleton of a VHDL test bench that instantiates our modules to allow co-simulation. Some connections are compacted for conciseness of explanation; the "•" symbol stands for a placeholder for all the 21 signals that actually constitute the AXI-Lite bus as shown in Figure 4 (see the `cosim_tb.vhdl` file in the supplementary materials for the complete source code).

Performing the co-simulation is just a matter of launching QEMU as shown in Section 3.1 and the VHDL simulator (order does not matter; each one waits for the other to connect to the pipes). But in order to examine the results, it can be useful to attach a debugger to the FW so that it is possible to control program execution and examine memory. To this end, the `-S` option to QEMU could be added so that the CPU is not started until it is so instructed by the debugger. An example of a debugging session is reported in Figure 8.

As can be seen, after running the code `rxcnt` equals 20, i.e., the 16 bytes queued by the direct accesses in lines 23–33, plus the 4 bytes posted from line 46. Indeed, it is possible to check that the first 16 bytes of the `rxbuf` array contain the same data that were queued in transmission. All write accesses have been correctly serialized in little-endian ordering, and the special events "error", "break", and "idle" have been detected as NAK, EOT, and NUL characters, respectively (they would have had the 9th bit set too, but it was not stored in `rxbuf`). The end of the RX FIFO can be detected by checking the "empty" flag or, more conveniently, reading from an empty FIFO just returns a flagged EM character (0x119) to save extra accesses to the control register. The last 4 bytes also correctly contain the string `"Hi\r\n"` that was last queued—in this case, stripped of special events as per line 51. Continuing the execution causes the `main` function to return and consequently the simulation to end. Actually, to terminate the simulation, the platform-specific code overrides the `exit` function with a write to a special register, implemented in our RTLbridge module, which causes the emulation to terminate. This is done only for the sake of presentation and to ease tests like this as normally a real FW image never terminates.

But the true power of co-simulation lies in the fact that the "electrical" effects of the FW can also be inspected so that its interaction with the HW can be verified and validated. During simulation, GHDL can save a VCD file with traces of all the digital signals of interest. A few examples of these, regarding the AXI bus accesses and serial lines, are reported in

Figure 9. For better visualization, the simulation was performed with a 100 MHz clock so that the signals align with the decimal time grid.

```
giorgio@i7-11700k:~/Research/fpga/hwsw$ gdb-multiarch
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
[ ... ]
(gdb) file /tmp/test/code.elf
Reading symbols from /tmp/test/code.elf...
(gdb) target remote :1234
Remote debugging using :1234
vector_irq () at platform.c:79
79              __asm("b _start");
(gdb) break main
Breakpoint 1 at 0x8020: file test-bfm.c, line 23.
(gdb) continue
Continuing.

Breakpoint 1, main () at test-bfm.c:57
57              loopback_test();
(gdb) next
58              return 0;
(gdb) print (int) rxcnt
$1 = 20
(gdb) x /21bx rxbuf
0x18858 <rxbuf>:        0x12    0x34    0x56    0x78    0xab    0xcd    0xef    0x91
0x18860 <rxbuf+8>:      0x92    0xb0    0xb1    0xb2    0xb3    0x15    0x04    0x00
0x18868 <rxbuf+16>:     0x48    0x69    0x0d    0x0a    0x00
(gdb) x /s rxbuf+16
0x18868 <rxbuf+16>:     "Hi\r\n"
(gdb) continue
Continuing.
[Inferior 1 (process 1) exited normally]
(gdb) quit
```

Figure 8. GDB session connected to the QEMU system emulator showing the looped-back serial data stored in memory by the emulated CPU after running the code listed in Figure 6.

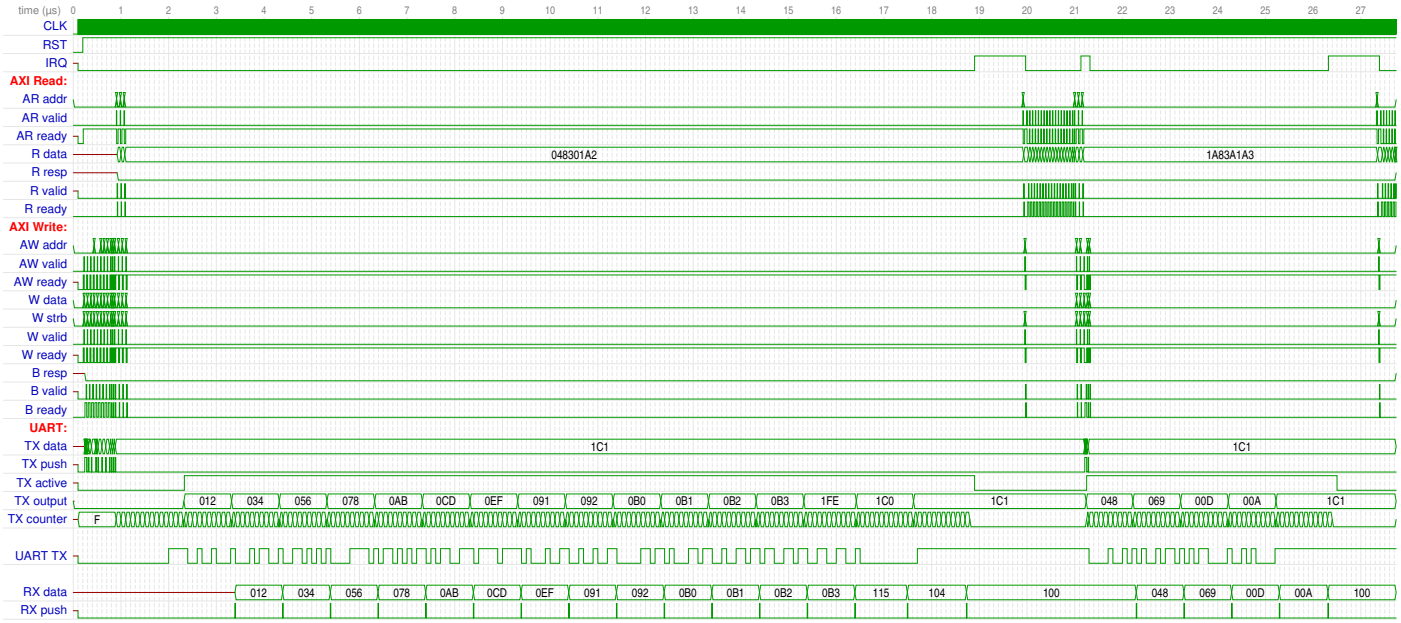

Figure 9. Digital waveforms saved by GHDL during co-simulation of the loopback test. This overview spans the entire duration of the test and reports all the main signals: AXI-Lite bus lanes, UART internal control signals (see text for description), and the serial data waveform (UART TX) synthesized by the DUT being co-simulated. The full VCD file is available in the Supplementary Materials.

The different phases of the FW are clearly visible in this overview: as soon as the RST line is de-asserted, the FIFO is primed with the 16 test bytes; then, three read-modify-write

transactions occur to manipulate the control bits (lines 35–37). After this, the CPU is put to sleep, but the VHDL simulation proceeds. Data are being pulled from the TX FIFO (`TX output` trace) and shifted to the `UART TX` output. Being a loopback test, the `UART TX` is also being de-serialized into `RX data`, which is pushed to the RX FIFO after each byte (`RX push`) or special event. When the transmission is finished (`TX active` goes low), the `IRQ` line is asserted. This causes QEMU to resume execution of the FW. It acknowledges the interrupt, reads back the data received, and prepares for the next transmission. This is actually started by the next IRQ, which causes the ISR to enqueue the posted data; then, the CPU sleeps again until an idle line is detected and the final IRQ causes the FW to read back the rest and terminate.

To better appreciate what is going on on the HW side of the simulation, details of the BFM transactions that occur at the beginning and end are reported in Figure 10.

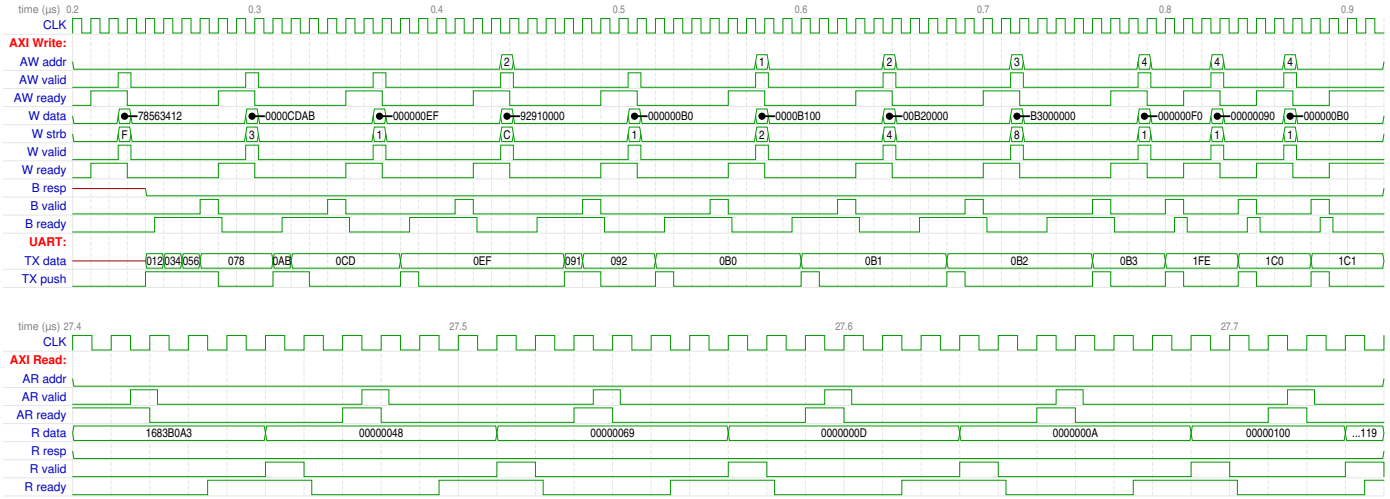

Figure 10. Details of the bus transactions that occur at the beginning (top panel) and at the end (bottom panel) of the overall simulation shown in Figure 9. The first part of the test was designed to test the correct implementation of all the possible bus access patterns and widths, as shown in the code excerpt. As can be seen from the "write strobe" (`W strb`) trace, the different C data types are correctly mapped to the corresponding access widths. Byte lanes are also correctly shifted (transactions at time 0.44 µs for the upper 16-bit and 0.58, 0.65, and 0.72 µs for non-32-bit aligned 8-bit accesses).

There, it is actually possible to see the AXI ready/valid handshake taking place, and although it is not a full conformance test (there exists specialized tools for that), it is still an indication of the correctness of its implementation in our `fastUART` module. The multi-byte writes to the data register being serialized into the TX FIFO can also be seen at the very beginning, taking 7 clock cycles per 32 bits (the implementation is not optimized for speed; it is just an example "optimized" for clarity), while the 8-bit accesses at the very end take 4 clock cycles each. Finally, the reads of the last 4 bytes from the RX FIFO are reported in the bottom panel, with the 0x119 code (flagged EM) that signals that the FIFO had been emptied at the very end.

4.4. Interactive FW-HW-SW Co-Simulation Example

In many cases, if the design being developed is a peripheral of some sort, not only does the HW need to interact with the FW within the embedded processor but the whole system also needs to communicate with some SW or device driver running on the host PC. The `PTYemu` module makes this possible without requiring modifications to the FW because it attaches to the HW side. It also makes it possible to interact with the FW in real time to validate its higher-level functionalities.

The `test-pty.c` example described here does just that. It is a (simplified) command line interpreter with very basic line editing functionalities implemented over a serial line,

and it is meant to be interacted with by the user through a terminal emulator. Figure 11 shows a little excerpt that detects the connection of the terminal emulator (through the special events break/idle) and sends a welcome message when it connects. The code then proceeds to handle terminal input, echoing it back and processing a few control characters such as CTRL+C, CTRL+D, CTRL+E, and DEL.

```
110  int main (void)
111  {
112      uart_control->rx_fifo.enable = 1;
113      uart_control->tx_fifo.enable = 1;
114
115      int escape = 0;
116      bool connected = false;
117      while (true) {
118          uart_recv(true);
119          wait_for_event(uart_rx_ready);
120          for (uint16_t value; (value = uart_data->read) != uart_fifo_empty; ) {
121              if (value == uart_recv_break) connected = false;
122              if (value == uart_recv_idle && !connected) {
123                  show_welcome_msg();
124                  connected = true;
125              }
```

Figure 11. Fragment of C code that interacts with a terminal emulator: initial connection detection. Excerpt from `test-pty.c`; see the Supplementary Materials for the complete sources.

An example of an interactive session using `picocom` on the host is reported in Figure 12, showcasing the possibility of real-time interaction with the emulated FW and simulated HW.

```
giorgio@i7-11700k:~/Research/fpga/hwsw$ picocom -q /tmp/test/pty 2>/dev/null
   Welcome to the FastUART FPGA Demo Interface

Firmware version: 2023-07-31T19:59:59+02:00

> help
This is the help message
that should be written to guide the user
on what the available commands are.
> ^E
ID      FastUART FPGA Demo Interface
FW      2023-07-31T19:59:59+02:00

> ^D

Bye!
giorgio@i7-11700k:~/Research/fpga/hwsw$
```

Figure 12. Example of an interactive terminal session (through the utility `picocom`) talking to the FW being emulated in QEMU by means of a PTY being simulated by GHDL. The command "help" and the characters ENQ (CTRL+E) and EOT (CTRL+D) after the ">" prompt had been interactively typed on the keyboard and caused the FW to reply (and terminate simulation after processing EOT). Complete source code of the example is in the `test-pty.c` file.

4.5. Shared-Memory Data Acquisition System

Finally, a complete SoC co-simulation is demonstrated in this section. The system designed is a digital data aquisistion system (DAQ) built to record data captured by external (to the FPGA) ADCs. The DAQ has a 128-bit parallel input data bus (which was meant to be used for 4 channels at 32 bit per channel or 8 channels at 16 bit per channel, or a combination thereof) and should be able to operate from 100 kS/s to 1 MS/s.

The acquired data are stored in an internal 64 KiB memory that is also accessible from the CPU (hence, we called it shared), built using dual-port BRAM. From there, it can be moved to a larger external memory or sent over the serial line, according to the application needs. A block diagram of the DUT, together with the added external modules to complete the testbench, is shown in Figure 13.

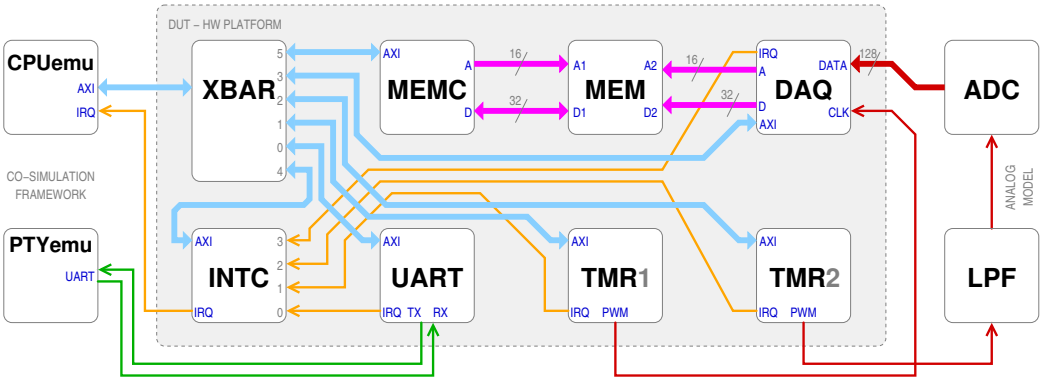

Figure 13. VHDL testbench for the DAQ system. Besides the CPUemu and PTYemu modules necessary to make the HW talk to the FW and SW, respectively, something to create the signal to be acquired by the DAQ must also be included in the testbench. The "analog model" serves this purpose and includes a low-pass filter and an "ideal" ADC to feed the DAQ with the PWM signal from TMR2, while TMR1 generates its sampling clock.

The modules that will be implemented into the FPGA are those in the shaded area named DUT:

- An AXI crossbar (XBAR), i.e., in this case an "AXI multiplexer" that has one upstream subordinate port and 6 downstream manager ports to which different peripherals can be connected. It decodes the addresses on the bus and connects the appropriate peripheral according to a configurable memory map. For our example, we used the VHDL module provided by Truestream AB (Linköping, Sweden) as part of their open-source offering "HDL_MODULES" [28].
- A memory controller (MEMC), to map the internal BRAM blocks to an AXI address space and adjust the signaling and handshake generation. It drives one port of the dual-port RAM.
- The internal memory (MEM), just 64 KiB of dual-port BRAM.
- The data acquisition controller (DAQ), which "serializes" its 128-bit input data into 32-bit writes to MEM, using its second port, and communicates to the CPU for trigger and status signals.
- An interrupt controller (INTC), to combine all the IRQ lines from the different modules into the single line to the CPU, with individual IRQ line enable flags, and of course IRQ status bits.
- The previously analyzed fast UART module (UART), for host communication.
- Two timer/PWM modules (TMR); one is used to generate the sampling clock for the DAQ, and the other is used to simulate an analog signal being acquired.

To perform the co-simulation, the DUT can be used unmodified as in the FPGA implementation, but the testbench must include our CPUemu module to drive the XBAR and receive interrupts from the INTC, as well as our PTYemu module to haul the UART data to the host. Plus, to provide a stimulus to the DAQ that could be controlled, so that it would be possible to verify its operation, we added a couple of "analog" modules, a low-pass filter (LPF) to turn the PWM signal into a continuous one, and an ADC "ideal" model that just quantizes the "analog" signal. Indeed, the LPF is a two-part filter: first the PWM signal is passed through a first-order filter and sub-sampled to 100 kHz; then it is further filtered with a 4th order discrete-time elliptical filter, designed in Matlab, and created with its HDL coder to demonstrate how easy it is to incorporate third-party models in our framework.

Performing the co-simulation is just a matter of connecting all the modules together in the testbench, writing a sample FW code to initialize the system, and compiling and executing the code. For reference, the main function of the FW is reported in Figure 14.

```c
int main (void)
{
    uart_control->tx_fifo.enable = 1;
    uart_control->rx_fifo.enable = 1;
    intc->enable = irq_uart | irq_tmr2 | irq_daqc;
    tmr1->period = 1000 - 1; // 100 kHz -- DAQ sampling rate
    tmr1->value  =  500;     // 50% duty cycle
    tmr2->period =  250 - 1; // 400 kHz -- PWM frequency
    daqc->reg    = 0xB0;     // single buffer capture
    wait_for_event(daq_capture_done);
    wait_for_pty_connection();
    uart_post((void *) mem, 64 << 10);
    wait_for_event(uart_tx_done);
    return 0;
}
```

Figure 14. Example FW code to test the DAQ system. See `test-daq.c` in the Supplementary Materials for register definitions, ISRs, and all the other details.

It just initializes the control registers of all the peripherals and waits for the completion of the data acquistion (the DAQ can raise an IRQ after it has filled half of the internal memory, for continuous acquisitions, or the whole of it, for single-shot acquisitions as in this example). Then, it waits for the SW client to connect (the `wait_for_pty_connection` function simply reads from the UART until an idle-condition special event is found, meaning the the other end of the PTY has been connected), and then it posts the whole memory content on the serial port. The timer ISR is very similar to the one in Figure 2, with the exception that the PWM period is now doubled after 8 cycles (just to verify that timing and delays work as expected).

However, in order to obtain the results from the co-simulation, the data that are being sent through the UART need to be extracted. The test FW, for simplicity and speed, just transfers the whole content of MEM in binary. To receive it, a simple host-side SW application that sets up the serial port, processes escaped bytes (for breaks and error notifications, see the PARMRK termios flag documentation for more details), and saves the results in a more convenient way for visualization is necessary. This can be found in the `examples/DAQ/sw/` folder in the Supplementary Materials. The final application will incorporate a graphical user interface for immediate visualization and control of all the DAQ settings, and the ability to perform a three-way co-simulation between the HW, FW, and this SW application will certainly aid its development. For now, a plot of the initial portion of the acquired data is shown in Figure 15, from which the delay of the LPF can clearly be seen both at the beginning and at the middle when the PWM frequency is abruptly halved.

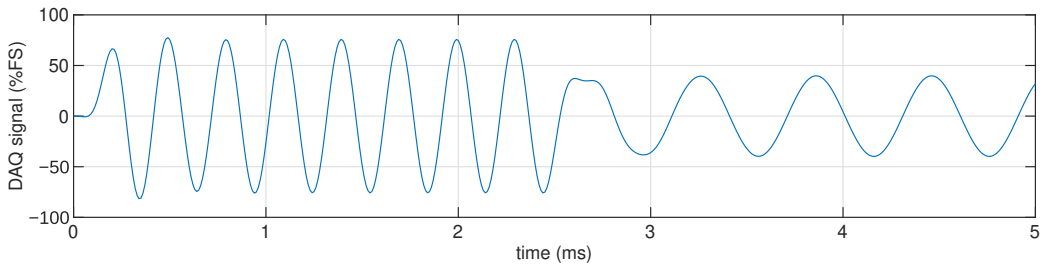

Figure 15. Data saved from the SW-side of the co-simulation of the DAQ system when the FW feeds the 400 kHz PWM with a 120-point sampled sine wave, resulting in a 3.333 kHz frequency, for the first 2.4 ms; then, the PWM frequency is abruptly halved. Halving the frequency also results in halving the amplitude because the waveform lookup table remains the same. The LPF has a cut-off frequency of 5 kHz so the transition is smoothed (and there is some ringing).

To ascertain the co-simulation performance, a further experiment was performed. This DAQ system, having its dedicated shared memory, is in principle able to operate autonomously, without a CPU, provided its internal registers are set "in HW" at the appropriate values. The same holds true for the PWM modules. It was thus possibile to remove the CPUemu and PTYemu modules from the testbench and just simulate the HW

alone with minimal modifications to its reset logic. To simulate 40.950 ms (the time to fill MEM at 100 kHz sampling frequency), GHDL took 29.5 s of CPU time on an Intel i7-11700k processor with 32 GiB of RAM. The HW/FW co-simulation (without the SW part, as serial transmission would have added extra time not included in the HW-only simulation), on the other hand, took 33.8 s of CPU time to co-simulate 40.960 ms (the FW exit is slightly delayed by a few microseconds). Emulating the CPU and adding the co-simulation modules to the testbench resulted in less than a 15% increase in simulation times w.r.t. an HW-only simulation. Of course, these figures largely depend on the relative complexity of the HW and the FW and cannot be considered universal for other designs but are still an indication that this framework is not very taxing CPU-wise.

5. Discussion

As seen in the previous section, the framework we developed allows for co-simulation to be performed with very minimal effort on the part of the designer. They only need to instantiate a couple more entities in the testbench—one to connect to the FW and (optionally), the other to connect to the SW—and run QEMU on the FW image while simulating the VHDL model.

This simplicity had been the major driving factor in the design of our co-simulation framework, to lower the barriers to the widespread adoption of this powerful technique, currently hindered by the lack of ready-to-use VHDL co-simulation tools.

As shown, we succesfully applied the proposed framework to the design of a fast UART transceiver to be later implemented on FPGA. Without it, trying to obtain the details of the interaction between FW, HW, and SW right would have been a terribly tedious and time-consuming process, with each iteration of synthesizing the design, although simple, taking significant fractions of an hour. Running the co-simulation takes fractions of a second. Moreover, using real HW makes it very difficult to obtain results such as those shown in Figure 9 because HW trace buffers are inherently limited in capacity, and their addition may also limit the whole design speed, preventing it from reaching timing goals.

Of course, the digital waveforms could have been generated by a VHDL simulator alone, provided a proper test bench is written to create the appropriate stimuli. This is undoubtedly required to obtain the finer timing details right, but for the overall functional test, this would have been a duplication of effort as in the final application the stimuli are ultimately decided by the FW running on the embedded processor. It would require a lot of effort to replicate in the VHDL code the functionality, e.g., of the interrupt handler that makes background transmission work.

Moreover, having the additional possibility of interacting with the FW from the host, while the FW is being developed and without involving configuring and programming the real HW, will greatly simplify and speed up the development process. This will allow us to start developing the PC-side device driver even before the real HW is available.

Besides all these advantages, the simplicity of the proposed framework comes with a few shortcomings that must be mentioned. Actually, they all derive from the lack of perferct synchronization between QEMU and the VHDL simulator. QEMU was never intended to be a cycle-accurate simulator, so, despite the several attempts that have been made in the literature to improve the synchronization, it will never be possible to achieve 100% accuracy without a complete redesign of QEMU itself, at which point one might be better off using an ISS or simulating the CPU at the RTL level. Again, there are not many tools capable of achieving the former, and the latter can become painfully slow quite quickly.

We chose to use a very pragmatic approach to synchronization, by allowing the QEMU concept of simulated time to advance at its “natural” speed while trying to also advance the GHDL concept of simulated time at an approximately fixed rate w.r.t. QEMU’s (a simulation quantum per QEMU timer tick). It is the authors’ opinion that this is enough for the majority of the FW. The reported examples show that it was indeed enough for interrupt-based events, polling (done while checking the RX FIFO for the empty condition), and even real-time interaction through a terminal emulator.

Time-critical sections of the FW might be unsuitable to be verified using this approach, but a functional verification can still be carried out, putting off a much more time-consuming detailed timing analysis to later stages of the development process, when at least the functionality will already be tested and verified.

6. Conclusions

In this work, a co-simulation framework obtained by patching QEMU, to make it possible for it to delegate memory-mapped I/O accesses to an external HW simulator, like GHDL, is presented. A simple VHDL module, CPUemu, translates these accesses to logic signals that can drive a VHDL design ranging from a simple peripheral to a full SoC (minus the CPU, which is instead emulated in QEMU for efficiency reasons, which is the whole point of co-simulation). A lightweight synchronization scheme allows both simulators to proceed concurrently, exchanging bus transaction requests and results, and interrupt requests, through sockets or pipes. This allows the FW and HW to freely interact with each other.

Moreover, another utility module, PTYemu, was also developed to allow the HW to interact with some SW running on the host PC where the co-simulation is being performed. This addresses the quite common use case wherever the device being co-simulated is in itself connected to a PC, and its FW must interact with a device driver or SW running on the PC. It thus allows a three-way co-simulation between FW, HW, and SW.

Application examples ranging from a trivial timer/PWM module, to a full set of peripherals with their AXI crossbar and interrupt controller, and including a complex FIFO-based UART, show the effectiveness of the framework to help design both the HW and FW, and to even perform three-way co-simulations that can interact with the user in real time.

Indeed, by avoiding the highly time-consuming FPGA synthesis and place-and-route process, co-simulation can significantly reduce the debug iteration time. It also provides complete observability of the entire system by enabling the use of standard software debuggers and recording waveforms for all the hardware signals, thus potentially enhancing the productivity of both HW designers and FW/SW developers.

To help ease the adoption of co-simulation techniques among designers using VHDL for their projects, at all levels of design complexity and experience, and to the scientific community at large, the source code for the framework we developed is released under liberal open-source licenses and made available both as Supplementary Materials attached to this article and online [8].

Supplementary Materials: The following supporting information can be downloaded at https://www.mdpi.com/article/10.3390/electronics12183986/s1: complete source code of the developed framework, compilation scripts, and examples, and complete VCD traces of Figures 9 and 10. Please also see [8] for possible updates.

Author Contributions: Conceptualization, G.B., L.F., P.C., M.A. and C.T.; investigation, G.B.; methodology, G.B., P.C. and C.T.; project administration, P.C. and C.T.; software, G.B.; supervision, G.B., P.C. and C.T.; validation, M.A. and L.F.; visualization, G.B.; writing—original draft, G.B. and P.C.; and writing—review and editing, G.B., L.F., P.C., M.A. and C.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AXI	Advanced eXtensible Interface
BFM	Bus functional model
BRAM	Block RAM
DMA	Direct memory access
DSP	Digital signal processor
DUT	Device under test
FPGA	Field-programmable gate array
GHDL	G hardware description language
HDL	Hardware description language
IRQ	Interrupt ReQuest
ISR	Interrupt service routine
PTY	Pseudo terminal
PWM	Pulse width modulation
QEMU	Quick EMUlator
RTC	Real time clock
RTL	Register transfer level
SPDX	Software Package Data Exchange
VCD	Value change dump
VHDL	VHSIC hardware description language

References

1. Chen, W.; Ray, S.; Bhadra, J.; Abadir, M.; Wang, L.C. Challenges and Trends in Modern SoC Design Verification. *IEEE Des. Test* **2017**, *34*, 7–22. [CrossRef]
2. Díaz, E.; Mateos, R.; Bueno, E.J.; Nieto, R. Enabling Parallelized-QEMU for Hardware/Software Co-Simulation Virtual Platforms. *Electronics* **2021**, *10*, 759. [CrossRef]
3. Yeh, T.C.; Zin-Yuan, L.; Ming-Chao, C. A Novel Technique for Making QEMU an Instruction Set Simulator for Co-simulation with SystemC. In Proceedings of the International MultiConference of Engineers and Computer Scientists, Hong Kong, China, 16–18 March 2011; Volume 2188.
4. Kruszewski, M. Easy and structured approach for software and firmware co-simulation for bus centric designs. *arXiv* **2021**, arXiv:2110.10447.
5. Foster, H. The 2022 Wilson Research Group Functional Verification Study. 21 November 2022. Available online: https://blogs.sw.siemens.com/verificationhorizons/2022/11/21/part-6-the-2022-wilson-research-group-functional-verification-study/ (accessed on 7 July 2023).
6. QEMU: A Generic and Open Source Machine Emulator and Virtualizer. Available online: https://www.qemu.org/ (accessed on 7 July 2023).
7. GHDL: VHDL 2008/93/87 Simulator. Available online: https://github.com/ghdl/ghdl (accessed on 7 July 2023).
8. Biagetti, G. HW/SW Co-Simulation. Available online: https://github.com/giorby/cosim (accessed on 14 September 2023).
9. Cho, S.; Patel, M.; Chen, H.; Ferdman, M.; Milder, P. A Full-System VM-HDL Co-Simulation Framework for Servers with PCIe-Connected FPGAs. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18), Monterey, CA, USA, 25–27 February 2018; pp. 87–96. [CrossRef]
10. Lora, M.; Vinco, S.; Fummi, F. A unifying flow to ease smart systems integration. In Proceedings of the 2016 IEEE International High Level Design Validation and Test Workshop (HLDVT), Santa Cruz, CA, USA, 7–8 October 2016; pp. 113–120. [CrossRef]
11. Accellera Systems Initiative. SystemC: The Language for System-Level Design, Modeling and Verification. Available online: https://systemc.org/overview/systemc/ (accessed on 11 August 2023).
12. Benini, L.; Bertozzi, D.; Bruni, D.; Drago, N.; Fummi, F.; Poncino, M. Legacy SystemC co-simulation of multi-processor systems-on-chip. In Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, Freiburg, Germany, 16–18 September 2002; pp. 494–499. [CrossRef]
13. Wang, C.C.; Wong, R.P.; Lin, J.W.; Chen, C.H. System-level development and verification framework for high-performance system accelerator. In Proceedings of the 2009 International Symposium on VLSI Design, Automation and Test, Hsinchu, Taiwan, 28–30 April 2009; pp. 359–362. [CrossRef]
14. Lin, J.W.; Wang, C.C.; Chang, C.Y.; Chen, C.H.; Lee, K.J.; Chu, Y.H.; Yeh, J.C.; Hsiao, Y.C. Full System Simulation and Verification Framework. In Proceedings of the 2009 5th International Conference on Information Assurance and Security, Xi'an, China, 18–20 August 2009; Volume 1, pp. 165–168. [CrossRef]
15. Biagetti, G.; Giammarini, M.; Ballicchia, M.; Conti, M.; Orcioni, S. SystemC-WMS: Wave mixed signal simulator for non-linear heterogeneous systems. *Int. J. Embed. Syst.* **2014**, *6*, 277–288. [CrossRef]

16. *IEEE Std 1666.1-2016*; IEEE Standard for SystemC Analog/Mixed-Signal Extensions Language Reference Manual. IEEE Standards Association: New York, NY, USA, 2016. [CrossRef]
17. Monton, M.; Portero, A.; Moreno, M.; Martinez, B.; Carrabina, J. Mixed SW/SystemC SoC Emulation Framework. In Proceedings of the 2007 IEEE International Symposium on Industrial Electronics, Vigo, Spain, 4–7 June 2007; pp. 2338–2341. [CrossRef]
18. Yeh, T.C.; Tseng, G.F.; Chiang, M.C. A fast cycle-accurate instruction set simulator based on QEMU and SystemC for SoC development. In Proceedings of the Melecon 2010–2010 15th IEEE Mediterranean Electrotechnical Conference, Valletta, Malta, 26–28 April 2010; pp. 1033–1038. [CrossRef]
19. Yeh, T.C.; Chiang, M.C. On the interfacing between QEMU and SystemC for virtual platform construction: Using DMA as a case. *J. Syst. Archit.* **2012**, *58*, 99–111. [CrossRef]
20. Díaz, E.; Mateos, R.; Bueno, E. Virtual Platform of FPGA based SoC for Power Electronics Applications. In Proceedings of the 2019 IEEE 28th International Symposium on Industrial Electronics (ISIE), Vancouver, BC, Canada, 12–14 June 2019; pp. 1371–1376. [CrossRef]
21. Logaras, E.; Koutsouradis, E.; Manolakos, E.S. Python facilitates the rapid prototyping and hw/sw verification of processor centric SoCs for FPGAs. In Proceedings of the 2016 IEEE International Symposium on Circuits and Systems (ISCAS), Montreal, QC, Canada, 22–25 May 2016; pp. 1214–1217. [CrossRef]
22. REDS Institute. Zynq-7000 HW-SW Co-Simulation QEMU-QuestaSim. 20 May 2020. Available online: https://blog.reds.ch/?p=1180 (accessed on 7 July 2023).
23. Southwell, S. Co-Simulation with OSVVM. 8 February 2023. Available online: https://osvvm.org/archives/2159 (accessed on 7 July 2023).
24. ARM. AMBA AXI Protocol Specification. Available online: https://developer.arm.com/documentation/ihi0022 (accessed on 7 July 2023).
25. UVVM: Universal VHDL Verification Methodology. Available online: https://uvvm.org/ (accessed on 7 July 2023).
26. Bellard, F. QEMU, a Fast and Portable Dynamic Translator. In Proceedings of the 2005 USENIX Annual Technical Conference (ATEC'05), Anaheim, CA, USA, 10–15 April 2005; pp. 41–46.
27. OSVVM. Open Source VHDL Verification Methodology. Available online: https://osvvm.org/ (accessed on 7 July 2023).
28. Vik, L. HDL_MODULES: A Collection of Reusable, High-Quality, Peer-Reviewed VHDL Building Blocks. Available online: https://hdl-modules.com/ (accessed on 5 September 2023).