System-Level Register Verification and Debug

Utkarsh Bhiogade, Kautilya Joshi, Puneet Goel
IIIT Nagpur, IIIT Nagpur, Coverify Systems Technology
email: utkarsh@coverify.com, kautilya.joshi@ericsson.com, puneet@coverify.com

***Abstract*—The Register Abstraction Layer (RAL) forms the bulk of interface between *hardware design* and *systems software*. UVM reg layer package provides a convenient way to handle verification of RAL interface in a System on Chip (SoC) test environment. With the growing need for hardware-software *co-verification*, it becomes imperative to explore advanced verification techniques that can facilitate a *system-level* perspective and yet leverage the strengths of existing verification methodology standards. Embedded UVM (EUVM) [1] (earlier known as Vlang [2]) provides one such technical advancement. EUVM is a multicore-enabled opensource implementation of the latest IEEE 2020 Universal Verification Methodology (UVM) standard [3]. EUVM testbenches compile to produce native binaries that can be run directly on embedded systems, thus enabling a systems perspective to functional verification. In this article we explore the EUVM port of the System Verilog (SV) UVM reg-package from an RTL designer's, a device-driver developer's, and a system engineer's perspective.**

I. Introduction

A. UVM Reg Layer

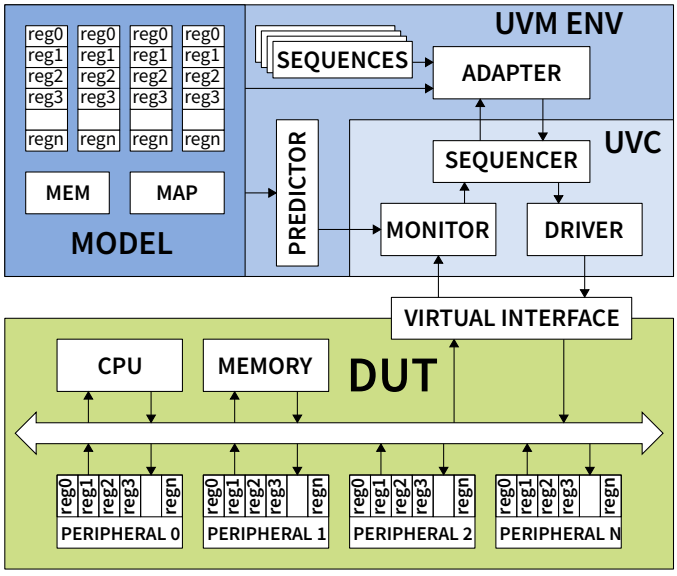

Fig. 1. UVM Reg Testbench Infrastructure

Complex SoCs may have scores of memory-mapped peripherals, integrated with one or more processors. Together these peripherals may account for tens of thousands of registers. The integration of such a large number of peripherals on the SoC bus fabric results in a complicated connectivity, which is difficult if not impossible to verify through a manual review of Register Transfer Language (RTL) code. A thorough functional verification of register accesses across the peripherals becomes imperative to debug and fix any connectivity issue that may have crept in during SoC integration of peripheral Intellectual Property (Core) (IP)s. But the functional verification of thousands of registers is itself a humongous task. With continuously evolving design specifications, adding a few registers in a peripheral may result in a shift in address space of a large number of registers. Additionally, most configuration and status registers consist of multiple fields with varying bit-positions, access restrictions and functionality, thus denying the possibility of a generic testbench architecture.

The UVM reg-package adds a higher abstraction layer to model register accesses in order to make verification process manageable. Figure 1 illustrates a typical UVM reg layer verification infrastructure. A reg *model* captures the complexity of all the registers down to individual fields. Each register is modelled by creating an individualized Object Oriented Programming (OOP) class with different fields laid out as instances therein. In turn, each field instance captures its bit-position, reset value, access restriction and other functionality traits. The *model* also maps the registers to memory-mapped addresses, enabling it to make read/write accesses using generic hierarchical names of the registers (or fields) rather than hard-coded addresses and bit offsets. When an access is made to a particular field, the UVM reg-package makes an access to the relevant register address and masks out the bits that do not correspond to the field being accessed. Consequently, any change in the address (or bit position) specification of a register field is automatically managed by the generated *model*, completely shielding the testbench code from tedious changes. This aspect of the UVM reg package is very useful for the testbench code meant for Design Under Test (DUT) configuration.

There is another perspective to the UVM reg layer, wherein it offers a standard mechanism to verify register address decoding and access functionality at block and system level. The UVM reg package contains a set of standard *sequences* that test the RTL design implementation of register address decoding and register access logic in a comprehensive manner. This is made possible by a *predictor* that monitors and interprets every register transaction on the memory-mapped bus, and figures out its outcome based on a precise reg *model*. A testbench coder is thereby fended off from the tedious effort that would otherwise be required to code a reference model of the register logic.

Further, the reg layer offers a means to abstract out the intricacies of the underlying memory-mapped bus protocol.

This is affected by the *adapter* block that needs to be coded manually. Functionally, the *adapter* block translates a generic `uvm_reg_item` transaction to the underlying memory-mapped bus protocol specific transaction and vice versa.

In practice, the register *model* is automatically generated from a specification maintained on an Excel sheet, or coded in a SystemRDL file. Normally, the same specification file is used to generate RTL code for the registers. Note that at architecture level, the reg *model* replicates the structure and functionality of the RTL implementation of register logic.

B. Reg Layer Verification Challenges

It may be noted that the benefits of UVM reg package are limited to RTL functional verification. With the ever-increasing hardware complexity at module and system level, there are new challenges that require a relook at reg-verification from various perspectives, as illustrated in the following scenarios:

1) Hardware-Software Coverification Perspective: Development of software drivers for complex hardware blocks require intricate maneuvers, wherein hundreds of registers need to be configured reflecting the desired mode of operation. In many cases registers are required to be configured in a specific order, failing which the device is rendered completely dysfunctional. Moreover, time-to-market pressure on the project timeline results in a cycle that requires parallel development of hardware and software. Section II describes a reg layer based technique that can enable coverification of RTL with device-driver software on a Qemu based development platform.

2) Design Debug Perspective: Continued advancement of deep sub-micron technology has resulted in very complex IP level designs that run into millions of gates. During the design-development cycle, a designer may need to experiment a lot with the various reg layer configurations, just to ensure the right mode of operation. But a testcase based reg layer approach may not be quite debug friendly, since a singular change in the register configuration code requires the designer to recompile and re-elaborate the testbench and the design, a process that takes significant amount of time. Section III expounds upon a testbench infrastructure that brings dynamism into the reg layer testbench, making it possible to read and write into registers at the designer's will via a user-friendly MS Excel based interface.

3) System Level Perspective – SoCFPGA: FPGA-based hardware accelerators have gained a substantial share in the processor domain. It is envisaged that in future even the server-end processors would have huge programmable FPGA resources available on chip. This brings a systems perspective to the IP-level design cycle. Note that the basic interaction between the processor and FPGA mapped blocks is via memory mapped register accesses, thus bringing a larger perspective to reg layer at application layer software. Section IV explores an EUVM-powered reg layer infrastructure that validates register access functionality directly on an SoCFPGA platform.

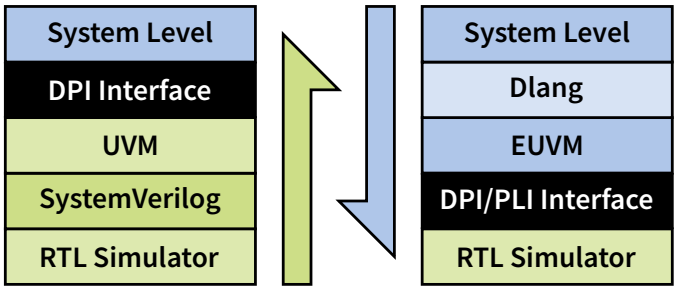

Fig. 2. UVM Reg Testbench Infrastructure

C. A brief introduction to EUVM

EUVM is a multicore enabled UVM implementation coded in The D Programming Language (Dlang) [4]. Like SV, EUVM offers sophisticated constrained-random stimulus generation and an automatic garbage collector.

Unlike the SV implementation of UVM which is tightly integrated with RTL simulation, EUVM offers a system-centric top-down approach. An SV interface to the system-level has to go through the DPI layer, which may offer a significant performance penalty [5].

Another advantage of the EUVM implementation of UVM is its portability across the various operating systems and machine architectures. EUVM testbenches can run on Windows, Android, Mac OS and most versions of Linux. Additionally, EUVM testbenches can be cross-compiled for embedded system platforms, making it possible to port RTL testbenches to embedded systems with a completely portable stimulus.

Under the hood, EUVM uses a rudimentary constraint solver to handle elementary constraints. It uses a Binary Decision Diagram (BDD) based solver to handle constraints of medium complexity. Hard constraints that can not be handled by BDD, are delegated to the Z3 [6] library that takes satisfiability approach to predicate solving.

II. Qemu/Reg Layer Coverification

Qemu [7] enables a cross-platform virtual machine environment that emulates the machine's processor through dynamic binary translations. It also lets the developer model hardware devices in order to facilitate development of hardware device drivers. Since such virtual models of hardware devices may not accurately reflect the actual functioning of hardware design, it is prudent to co-verify the device-driver code against an RTL simulation of the device. This section explores a simple setup to interface Qemu with an RTL design simulation. With this setup in place, when a developer executes his driver-code on Qemu virtual machine, it results in register access transactions in a simulation running on the host machine. On completion, the outcome of the transactions is reflected back to the device-driver code running on Qemu.

Since Qemu and RTL simulation run as separate Linux processes, a pair of Linux *fifos* handle interprocess communication between the two. One *fifo* manages register access commands from the qemu to the simulation and the other one is used to

communicate status/read-data back to Qemu. As outlined in [8], Qemu provides command-line options to enable sharing of *fifos* between the host machine and the Qemu virtual machine.

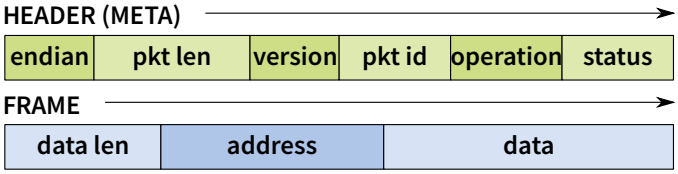

Fig. 3. Reg Command Packet Frame

Note that a Linux fifo is a serial device. Data exchanges via these devices take place as a stream of byte transfer without any underlying packet structure. A protocol (or a packet structure) is therefore required for the transaction of register commands and status requests. In addition to *data* and *address*, we need to add some meta information to the packet in order to take care of variations in machine architectures and to facilitate consolidation of structured data frames on the host and the Qemu virtual machine platforms. A typical command/status frame structure is outlined in figure 3. A header is prefixed with every frame of transaction made across the *fifos*.

Correspondingly, in the testbench code, the *header* and the *frame* structures are laid out in form of packed structs ...

```
1  struct tr_header {
2    align(1):
3    ubyte endian;
4    uint pkt_len;
5    ushort ver;
6    uint id;
7    kind_e op;
8    uint status;
9  }
10
11 struct tr_frame {
12   align(1):
13   uint length;
14   ulong addr;
15   uint data;
16 }
```

Fig. 4. Reg Header and Frame Structs in EUVM

Note that the pragma *align* on line number 2 and 12, is the D way of telling the compiler that the fields in the structs are packed (aligned at single byte boundaries).

When a reg command/status transaction is received from Qemu, first the header part of the packet, which is always of a fixed length, is cast to the `tr_header`. Consequently, based on the interpretation of total `pkt_len` in the header, the remainder of the frame is unpacked into a `tr_frame`. Figure 7 illustrates the process for a read command. Note that the flag `_swapRequired` is based on the interpretation of the *endian* field, to take care of the scenario where the *endianness* of Qemu virtual machine does not match that of the host machine.

```
1  tr_frame* read_data(File fifo, uint size) {
2    ubyte[] trSizeBuf;
3    trSizeBuf.length = size;
4    fifo.rawRead(trSizeBuf);
5
6    tr_frame* frame;
7
8    frame = cast(tr_frame*)trSizeBuf.ptr;
9
10   if (_swapRequired) {
11     frame.length = swapEndian(frame.length);
12     frame.addr = swapEndian(frame.addr);
13     frame.data = swapEndian(frame.data);
14   }
15
16   return frame;
17 }
```

Fig. 5. Mapping Read Data to Frame Structure

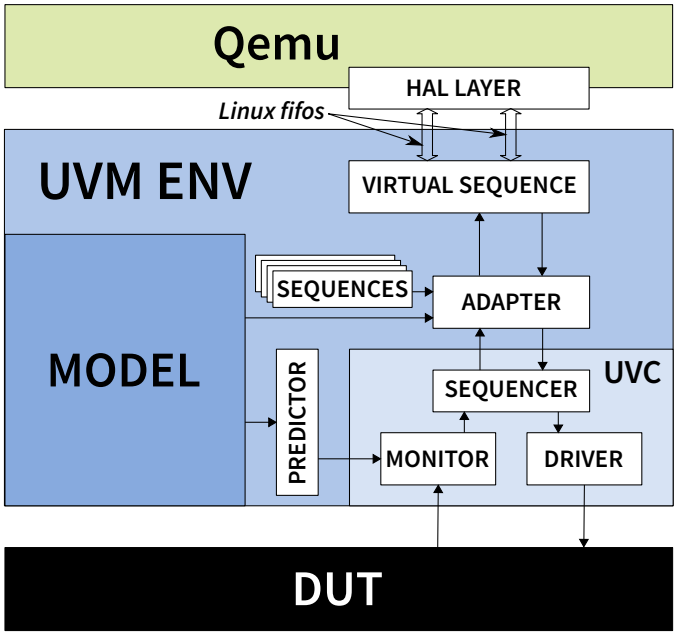

Fig. 6. Reg Layer and Qemu Interface

It is also pertinent to note that while the testbench executes on a discrete event simulator, the driver software runs on the Qemu virtual machine in a completely asynchronous manner. A mechanism is therefore required to synchronize the incoming reg access commands with the testbench. This is done via a virtual sequence that waits on packets from the Linux *fifo* interface. Once the sequence receives a command from the *fifo*, it interprets the command and converts it to a reg sequence item. These sequence items are then forwarded to the simulation via a UVM sequencer and a bus protocol driver. Figure 8 outlines the architecture and connectivity of the virtual sequence. Note that there is little involvement of the reg *model* in this scheme of things. It is the software driver code, running on Qemu, that maintains an address map for the registers it wishes to access.

```
override void body() {
  req =
    uvm_reg_item.type_id.create("req_"
                                ~ get_name);
  req.kind = _kind;
  req.addr = _addr;
  req.data = _data;

  start_item(req);
  finish_item(req);

  uvm_info("APB SEQ", format("\n%s",
           req.sprint()), UVM_DEBUG);

  _data = req.data;

} // body
```

Fig. 7. Virtual Sequence for reg-qemu interface

Note the complete absence of DPI like interface in the EUVM testbench infrastructure that integrates with Qemu. EUVM is fully ABI compliant with C/C++, thus allowing the verification engineer to seamlessly interface with any C/C++ code and make operating system calls at will.

Complete EUVM reg layer interface to Qemu is available for download from the EUVM Github account [9].

III. Interactive Register Debug

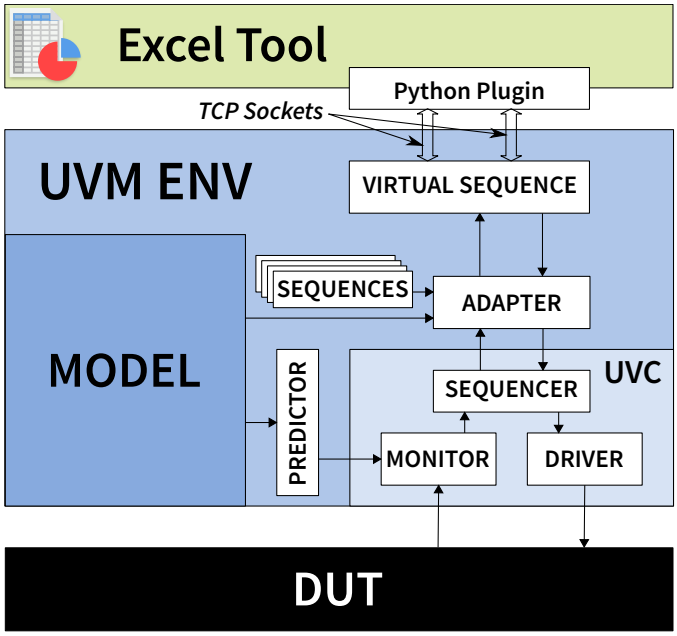

Fig. 8. reg layer and MS Excel Interface

As noted in section I-B2, during IP development, an RTL designer may need to experiment a lot with configuration of registers in order to correctly bring the design simulation up in a particular functional mode. For complex multi-million gate IP designs, a few mistakes in the configuration could mean hours of wasted time in repeat fix, compile and elaboration cycles.

In the previous section we explored how a simulation can be made to follow register access commands sent from another process running on the same Linux server that runs the simulation. To achieve interactive register debug, the same testbench architecture can be extended to interact with an Excel tool.

Gnumeric [10] is a Linux tool to edit Excel files, and allows the user to execute Python code that can be plugged in into the tool. More recent advancements [11] have made it possible to plugin Python code into the Microsoft Excel tool running on a Windows server.

Considering that MS Excel could be running on a different server than the design simulator, and that MS Excel would execute mainly on Windows servers, Linux *fifo*s offer little help. TCP/IP socket [12] interface offers the most reliable means of communication between applications running on different servers connected on an Ethernet LAN.

As was the case with Qemu, reg *model* is of little use with Excel driven simulations as well. A populated Excel sheet must have all the registers and fields laid out along with reset values, address, and bit positions. It is common for RTL designers to maintain the list registers in an Excel sheet in this manner, though a recent trend is to model the registers in SystemRDL [13]. The advantage of using SystemRDL is that it is an industry standard. There is also a wide availability of commercial as well as opensource SystemRDL [14] tools. Some SystemRDL tools also make it possible to convert the register specification into a CSV file that can be used to populate an Excel sheet with register specification.

Figure 9 lists the Python code snippet that enables reading the value stored in a register from the simulation. When a similar access is made on a reg-field instead, Python reg layer plugin first makes a read access on the corresponding register by creating a reg-command packet and sending it to the simulation via TCP socket interface. Later when the read-data is returned by the simulation, the Python plugin masks out the bits that do not belong to the specified field based on the bit positions of the specific field as listed in the Excel sheet.

The *virtual sequence* for TCP Socket interface needs a bit more attention since TCP payload arrives on the TCP sockets in chunks unlike a Linux *fifo* which behaves as a streaming serial device. When receiving a packet on a TCP socket, the virtual sequence code first buffers the ingress data and only when the buffer has sufficient number of bytes to interpret a reg header, it processes the header. Later, the reg-frame is similarly construed from the buffered data, based on the packet length information received as one of the fields in the reg-header packet structure.

Complete EUVM and Python plugin code for the Excel interface to reg layer is available for download from the EUVM Github account [9].

```
def reg_read(cell_range):
  ar=get_field(cell_range)
  #checking if field is valid
  if ar[0] == None:
    return 'INVALID FIELD'
  else:
    if ar[0].lower()[0:8] == 'register':
      field_ar=get_fields(ar)
      a='WO' in str(field_ar)
      if a == True:
        return 'WO reg, unable to read'
      else:
        addr=int(str(ar[2]),16)
        read_data=read_val(addr)
        return 'DATA: ' +
            str(int(read_data,16))
    elif ar[0].lower()[0:5] == 'field':
      a='WO' in str(ar)
      if a == True:
        return 'WO field, unable to read'
      else:
        fg=True
        i=n1-1
        while fg == True:
          cell=s[m1+2,i]
          val=cell.get_value()
          reg_cell=s[m1,i]
          reg_name=reg_cell.get_value()
          if (val != None and
            reg_name.lower() == 'register'):
            addr=int(val,16)
            fg=False
          else:
            i -= 1
            read_data=read_val(addr)
            read_data=
            "{0:032b}".format(int(read_data,
                                  16))
        if len(str(ar[4])) <= 4 :
          end=32 - int(ar[4])
          temp_data=list(read_data)
          data=temp_data[end-1]
          read_data="".join(data)
          read_data=int(read_data,2)
          return 'DATA: '+str(read_data)
        else:
          st1=(ar[4].strip('[]')).split(':')
          start=32 - int(st1[0])
          end=32 - int(st1[1])
          temp_data=list(read_data)
          data=temp_data[start-1:end]
          read_data="".join(data)
          read_data=int(read_data,2)
          return 'DATA: '+ str(read_data)
    else:
      return 'INVALID FIELD'
```

Fig. 9. Python Plugin Code to infer a Read Access

IV. REG LAYER TESTBENCH FOR SOCFPGA

An SoCFPGA [15] integrates FPGA resources with a CPU on the same ASIC. The ability to program an FPGA and its tight integration with the CPU enables an end user to implement hardware accelerator designs on the FPGA. Unlike the hard ASIC blocks that are mainly handled by device drivers, FPGA based hardware accelerators interface with the software at user or application level. Also note that in an SoCFPGA, the CPU mainly interacts with the FPGA via memory mapped bus protocols.

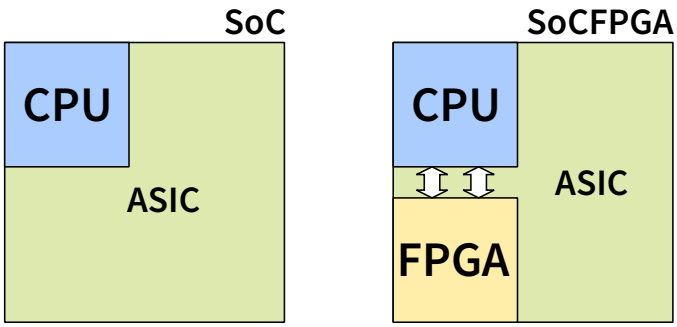

Fig. 10. SoCFPGA Overview

The advent of SoCFPGAs, brings forth an interesting verification perspective. When an IP designer hands off an RTL or a bitfile to a software engineer, there needs to be a way to ensure that the RTL design has been integrated and mapped flawlessly on the SoCFPGA. Since a hardware accelerator integrates tightly with software, there is also a need to pragmatically ensure that the logic mapped on the FPGA meets the functional requirements listed in the specification document.

Since the interaction between the accelerator and the CPU is limited to memory mapped accesses, reg layer plays an important role in the verification of SoCFPGA systems. The ability to cross-compile EUVM testbenches for embedded CPU architectures ensures that we can run EUVM reg layer testcases directly on the SoCFPGA systems.

Unlike a normal testbench meant to verify RTL designs, an EUVM testbench running on SoCFPGA would not need a driver because an SoCFPGA maps the memory-mapped protocol driver directly to the hardware logic. In fact, the testbench can be made completely protocol-agnostic with `uvm_reg_item` transactions directly getting processed by software reads and writes.

It is pertinent to note that before we can start making access requests to the design mapped on the FPGA, we need to map the register address space to virtual memory. This is done by making a call to Linux `mmap` function as shown in the code listing in Figure 11.

A complete golden reference testbench that runs reg layer tests complete with reg *model* on a Cyclone V board can be downloaded from [9].

V. CONCLUSION

Since the UVM reg layer relates to register and memory accesses which almost always originate from software code running on a CPU, it plays an important role in hardware-software coverification which has been largely ignored. From a systems perspective, the tight integration of SV UVM with RTL simulators poses a hindrance in using SV UVM

```
void map_registers() {
    fd = open("/dev/mem",
              O_RDWR | O_SYNC);
    if (fd < 0) {
      assert(false,
             "Failed to open /dev/mem");
    }
    mem = mmap(null, HPS_TO_FPGA_LW_SPAN,
               PROT_READ | PROT_WRITE,
               MAP_SHARED, fd,
               HPS_TO_FPGA_LW_BASE);
    if (mem == MAP_FAILED) {
      close(fd);
      assert(false, "Can't map memory");
    }
    regs = cast(uint*) mem;
  }
}

override void final_phase(uvm_phase phase) {
  super.final_phase(phase);
  munmap(mem, HPS_TO_FPGA_LW_SPAN);
  close(fd);
}
```

Fig. 11. EUVM code to map FPGA registers physical addresses to Virtual Memory

reg-package for system-level verification. EUVM provides a complete port of IEEE UVM 2020 standard along with the reg layer. EUVM testbenches can be cross-compiled and directly run on embedded systems, thus providing a practical path to system level reg layer verification and validation.

Since EUVM is build on Dlang, a systems programming language, it provides a convenient way to integrate EUVM testbenches directly with systems software development platforms like Qemu.

VI. Acronyms

Dlang The D Programming Language
RAL Register Abstraction Layer
EUVM Embedded UVM
ABI Application Binary Interface
API Application Programming Interface
BDD Binary Decision Diagram
DPI Direct Programming Interface
DSL Domain Specific Language
DUT Design Under Test
GC Garbage Collector
IP Intellectual Property (Core)
OOP Object Oriented Programming
OS Operating System
PLI Programming Language Interface
RTL Register Transfer Language
SV System Verilog
SoC System on Chip
UVM Universal Verification Methodology
VPI Verilog Procedural Interface

References

[1] Embedded UVM Home Page. [Online]. Available: http://uvm.io

[2] P. Goel and S. Adhikari, "Introduction to Next Generation Verification Language - Vlang," 2014.

[3] "IEEE Standard for Universal Verification Methodology Language Reference Manual," *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*, pp. 1–458, 2020.

[4] W. Bright, A. Alexandrescu, and M. Parker, "Origins of the D Programming Language," *Proc. ACM Program. Lang.*, vol. 4, no. HOPL, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3386323

[5] A. S. Parag Goel and H. V. Balisetty, ""C" you on the faster side: Accelerating SV DPI based co-simulation," *Proceedings of Design Verification Conference, San Jose*, 2014.

[6] L. De Moura and N. Bjorner, "Z3: An Efficient SMT Solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[7] X. Bian, "Implement a virtual development platform based on QEMU," in *2017 International Conference on Green Informatics (ICGI)*, 2017, pp. 93–97.

[8] How to setup QEMU output to console and automate using shell script. https://fadeevab.com/how-to-setup-qemu-output-to-console-and-automate-using-shell-script/. Accessed: 2021-07-26.

[9] Embedded UVM Github Repositries. Accessed: 2021-07-26. [Online]. Available: https://github.com/euvm

[10] O. Frommel, "Working with Gnome's Gnumeric Spreadsheet," *Linux Magazine*, 2006.

[11] Xlwings Documentation. [Online]. Available: https://docs.xlwings.org/

[12] M. Donahoo and K. Calvert, *TCP/IP Sockets in C: Practical Guide for Programmers*, ser. TCP/IP Sockets in C Bundle. Elsevier Science, 2009. [Online]. Available: https://books.google.co.in/books?id=dmt_mERzxV4C

[13] "SystemRDL 2.0, Register Description Language," 2018. [Online]. Available: https://accellera.org/images/downloads/standards/systemrdl/SystemRDL_2.0_Jan2018.pdf

[14] Free and Opensource SystemRDL tools. Accessed: 2021-07-26. [Online]. Available: https://github.com/SystemRDL

[15] What is an SoC FPGA? Accessed: 2021-07-26. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ab/ab1_soc_fpga.pdf