verification with Python

Ilia Greenblat

August 9, 2016

1 Introduction

Over the years Asic and Fpga verification became a discipline by itself. Verification experts are in demand. Increasingly systemVerilog is used for this task. But here is the catch: systemVerilog is pretty bad foundation for verification. The syntax, structures and dynamics are just too many and too complicated to write, to use and to debug. This is the main motivation for the following article. We want to make verification simpler and more maintainable. The scarcity of verification experts is a function, among other things, of long learning curve and high effort to maintain the ever-changing designs.

2 What is wrong with SystemVerilog?

SV (systemVerilog) started out as enhancements to address weaknesses of regular verilog. During it's evolution though, it became bloated and complex. systemVerilog is in most parts a direct translation of C++ to verilog. So it happens that on top of simple but antiquated verilog syntax, myriad of syntax constructs are added to mimic C++ in somewhat vaguely resembling verilog way.

The problem is complexity and pickiness of the resulting mongrel language. Also because it so huge, no two implementations completely agree on exact meaning of everything. Our main problem is with parts of SV, used as foundation of UVM.

Designers spend inappropriate amount of time on working on the language itself, not on the design. Often the holy book of standard is consulted and the meaning of founding fathers is fathomed. Also error messages during compilation are many times cryptic and not very helpful. Bottom line it is just too complex to use for the task. Still wide acceptance and wrong notion that it is the future feeds back to increase it"s market share.

By the way, if C++ is to your liking, it is better to use full fledged C++ environment. For one such option lookup simPLus from Shmuel Amrusi.

"E" is rumoured to be simpler and better , but is in steady decline due to commercial issues. Still, it has hardcore enthusiastic followers, especially in companies with deeper pockets. Another weakness of "E" , i guess, it"s limited scope and development options.

And many years ago , i accidentally spoke with Yoav Hollander, then main guy at Verisity - it appears they thought about using Python, instead of inventing a language. It didn't happen because with Python there was nothing to sell. Which explains why using python is such a good idea.

UVM is a good idea. The verification built from recognisable components. It was long recognised that most verification tasks share a great deal. UVM formalises the shared parts into a set of standard components. It's main problem is that it is built on top of bad basis.

One may argue that the design language is not supposed to be pretty. However navigating between ugly looking classes, modports, defines and interfaces is taking too much effort and time on the part of the designers. It is easy to get lost amidst inheritance chains and other ropes given to You to hang oneself.

3 Are there alternatives?

So what is our offering? Let's start by agreeing that the language in which the design itself is expressed, doesn't have to be the one that verification is built (see E). Moreover if the language is different, recompilation time of the DUT is spared. Verilog code compilation time is much shorter, because original verilog compiled very fast.

Our language of choice for verification is Python. Combining Python with Verilog opens up an opportunity to use free tools throughout the whole process. It may save money, some grey hair and allow to see your kids growing. It also allows easy switch between commercial simulators, because the simulator sees just the DUT, which by most part is plain RTL. The connection between simulator and Python is currently implemented using VPI. see Figure1.

One obstacle to adopt Python as a verification vehicle is issue of standard. Many verificators are currently grinding UVM. So to facilitate the transition, we need to write up guidelines similar to UVM. Using Python is not equal to abandoning principles of UVM. Guidelines from UVM can be used to organise the python classes and their names. But because python is much more concise, it is not a big job. It will not require deep inheritance classes.

Another valid argument against python centric approach to verification is issue of VIP. verification IP is very important, but based on our experience , replacing commercial VIP with python version is not that hard. It worked on more than one instance. So maybe it is not that bad.

---

4 Why Python

There are two requirements from a verification language. One is just basic good powerful language. Any code we write should be concise and maintainable. The other aspect is "How

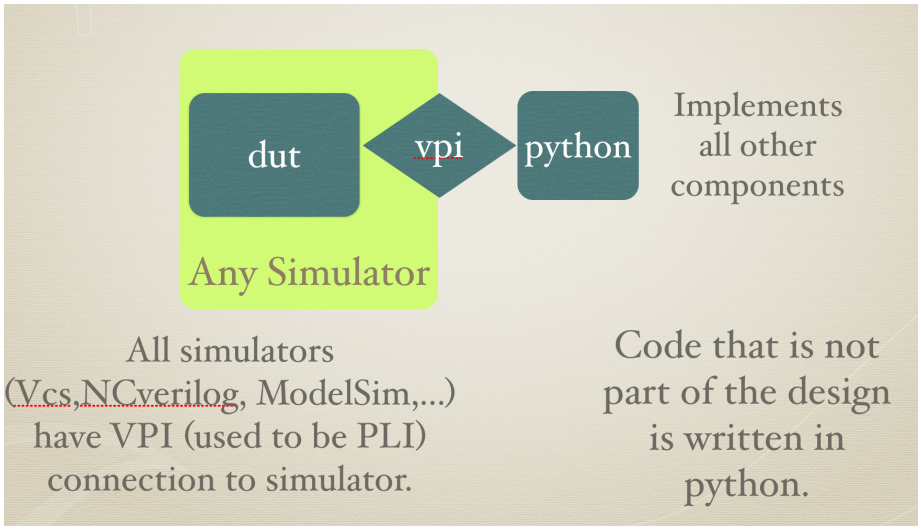

Figure 1: The connection

well it is suited for verification?" How well the verification tasks can be implemented using Python?

4.1 Python as a language

1. python is structured organised object-oriented scripting language. Easy to write and easy to understand what You or another designer wrote later. Everything is first class variables.
2. Widely used for many different tasks and for many years.
3. easy to learn. www.python.org is great starting point for learning it. Surely, faster and more fruitful than SV.
4. it is open source, platform agnostic and everything is downloadable.
5. Is very powerful and high level and very fast. High level is very important, but discussion about it is outside of scope.
6. has wealth of libraries written by many people.
7. Has GUI modules, in case graphical representation of values, states is desirable.
8. VPI interface is written in C once and effectively hidden from the user.

4.2 Python as a verification language

1. like all scripting languages, much work can be done with strings. State names, comm Packets, whole bunch of objects can be generated / read from files / manipulated as strings.

2. lists can be used as queues, scoreboards and much more. List items can be complex objects or mere integers.

3. directories (associative lists) can serve as scoreboards and holders of different objects.

4. rich random generator package works as constraints and generator of stimuli.

5. all test bench components are easily implemented. This statement is backed by our experience of over 10 years.

6. Solid object-oriented design facilitates scalability to all levels of chip verifications. from individual module to the whole deal.

7. ready made libraries cover almost anything imaginable. Many popular standards and algorithms have golden model already written in python libs.

8. for example MPEG encoders/decoders, GZIP algorithms, encryption algorithms, hash and signatures, various CRCs, maths and more.

9. reading/writing external setup/scenario files is done in straightforward fashion.

10. we had done golden models of custom processors, interlaken, ethernet, jtag, slow serials, cordics and more.

There are more python-verilog connections. one can be found at "http://tsheffler.com/software/python". The guy there takes a bit different approach, but is very convincing why the connection makes sense.

Perl can be used as replacement of Python. But to me, Python has distinct advantage - it is very ordered language. And ,until something changes, we need to decide on one standard way. If You know some programming language, You probably know great deal of python.

4.3 What using python as verification language misses?

Python is a general purpose language, it encompasses great deal, but was not designed with particular application in mind. For example, one may argue that there is no built-in constraint solver. Frankly i never encountered a design that needs one. Not sure if it is a problem at all. It is not that constraints solvers are not used. They are. It is just i never

saw example of significant use. In all real cases simple random package could do the job. For the rare exceptions, there are several python libraries that do constraint solving.

Important aspect of verification is coverage. What is known as "coverpoints, bins and crosscover". Indeed it is possible to use this mechanism as-is. Also it is a minor effort (and we have done it) to make it in Python and keep it as standard.

Control of future timing events is not trivial. That is, if You want to stop in the middle of a function and wait for event or delay. This may look bad, but

1. there are workarounds.
2. we doing fine without it.
3. it forces more robust code.

It is true, that it takes some adjustment to work effectively with Python as a verification language, but the results usually include increased productivity and shorter cycles. Now consider the popular notion of reuse. It is important to reuse, if re-inventing something is expensive. On the other hand there always is cost associated with reuse. Using powerful language eases some the pressure to reuse, because it is so easy to recreate.

5 First Example

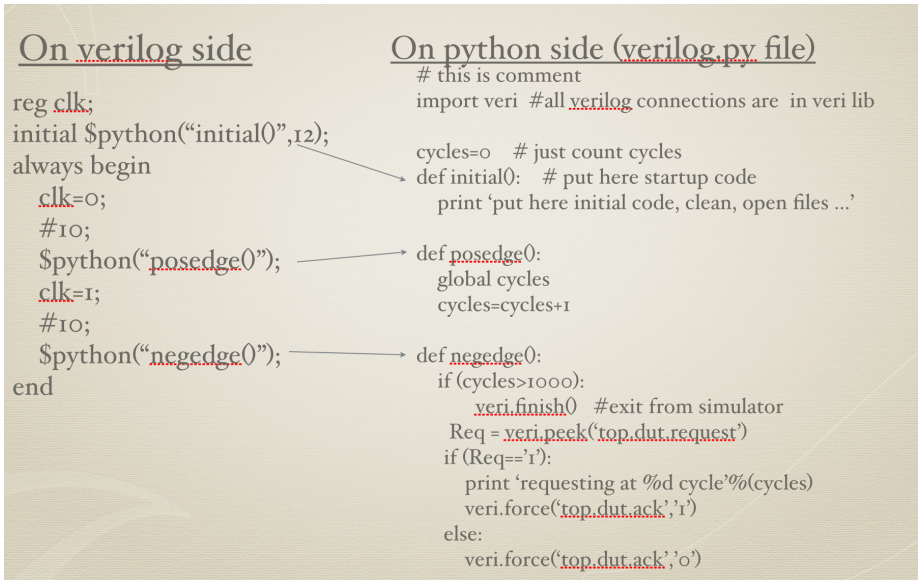

Figure 2: An introduction example

From verilog side, You call the python routines through `$python()` system task. see Figure2. This task is defined in the connection shared library. The python routines are defined in python source code. This source code is preloaded from python specified file during simulator starting up. On python side, importing veri module enables the python code to access and manipulate simulator variables.

6 What can we keep from systemVerilog

Some constructs from SystemVerilog are used in RTL. Since they are not part of verification, our discussion does not relate to them. Another section usable is coverage. Although it is doable our way too, it is relatively simple and has nice tool chain.

7 The connection

The connection uses VPI to connect Python to verilog simulator. It doesn't try to imitate VPI routines with python wrapper routines. Rather minimal set of useful procedures is implemented. It is designed to be as lean as possible, while keeping all the power deemed necessary. To use it, special line in invocation of the simulator should be added. for example in VCS it should look like `vcs +v2k -full64 +acc+1 -load /home/ilia/software/vpi/vpi5_nc_27.so:vpit`

In NCVERILOG use "+vpiload=" instead. Other simulators use similar lines. Currently You need python2.7 installed on the Linux system for this connection to work. On windows, different compilation path is used, and proved to work too.

The code of the shared library ".so" is written in C. It is compiled only when the code changes, which is rare. Same shared library (dynamically loaded) works as-is on many different Linux versions. The only hard constraint is 64bit vs 32bit operating system. Working in 64bit system and simulator is the optimal choice.

The library serves two purposes. Facilitate invocation of python routines from verilog side and access verilog variables from python.

8 Verilog Side

On verilog side, we have just two system tasks. `$import("vsrc/topVerilog.py");` import system task will preload python code. The filename for `$import` is either relative pathname from the current directory or an absolute path. It may also include setenv variables. If ":" character is present in the filename, it is treated as `$PATH` is treated — search alternatives.

The system also checks for "verilog.py" file in current directory. If found it preloads it also. It is best to use either method, but not both. There might be unwanted name space clashes otherwise. all python preloads end up as "from topVerilog import *". So all functions end up in same shared namespace.

The main workhorse is `$python()` system task. As most tasks in verilog, It can be invoked inside always statements. first parameter is a string with name of any preloaded python function. for example: `always @(negedge clk) $python("negedge()");`

This will result in invoking "def negedge()" python function on every negedge of "clk". `$python` can have more parameters. `$python("do_something()",counter[4:0],somewire,...);`

All parameters are passed as binary strings. In python side they will come something like "0010101010". The length of the string is the width of the original expression or wire or reg.

9 Python Side

To access verilog from Python, first `import veri` needs to be done. this *veri* library has number of functions to connect and manipulate verilog simulation.

1. `veri.force(Sig,Val)` deposit value to reg.
2. `veri.force_mem(Arr,Ind,Val)` deposit value to member of array.
3. `veri.force_3d(Arr,Ind0,Ind1,Val)` deposit value to member of 3d array.
4. `Val = veri.peek(Sig)` get current value of net/reg.
5. `Val = veri.peek_mem(Sig,Ind)` get current value of array member.
6. `Val = veri.peek_3d(Sig,Ind0,Ind1)` get current value of 3d array member.
7. `veri.force_hard(Sig,Val)` really force net or reg.
8. `veri.release(Sig,Val)` release net, not sure what Val is used for.
9. `Time  = veri.stime()` return current sim time (32bit)
10. `veri.finish()` finish simulation and exit.
11. `veri.listing(Path,Depth,Filename)` list to a file all nets/hierarchies from Path to Depth levels. 0 depth is all.
12. `veri.register(Path,Routine)` activate Routine every time net Path changes.
13. `veri.register(Delay,Routine)` activate Routine once after Delay. Delay is strings like "1000". the time unit according to timescale.

In all veri.force functions, the Value is always a string. It may be a binary string, like "0b0010101010xz". It starts with 0b. Then the string is made out of just four characters 0,1,x,z. The length of the string should be the verilog variable width. If the string starts

with "0x", the rest is treated as hexadecimal characters. If it doesn't start with 0x or 0b, it is assumed decimal value. `veri.force('top.dut.count',str(counter))`

You cannot force or peek partial variable, like `bus[15:0]`. No `[:]` is allowed. In most cases the path is hierarchical, like `tb_top.dut0.cpu.pc`.

The peek values are always binary string. It comes without "0b", just series of characters 0,1,x,z .

`veri.listing` will travel the hierarchy from `path` down and will write all variable names and it's current value into `Filename`. Depth limits the travel depth down. Value of 0 is equal to infinity.

10 Starting out

10.1 Installation

First, make sure that python2.7 is installed in the system. It should better be 2.7 and latest one from www.python.org. But previous versions may be ok too.

The Python installation should also include development stuff. Best if libpython2.7.so is present. libpython2.7.a can be used, but will require recompilation. Make sure that `libpython2.7.so` path is in `$LD_LIBRARY_PATH`.

You should obtain from me file like: `vpi5_64_27.so` . Where 5 stands for major release, 64 stands for 64bit system and 27 indicates python2.7 The path to this file is written explicitly in the NCVERILOG or VCS or others compilation of the test. So no need for setting more variables. It works as is on Ubuntu and RedHat. If You need to compile it yourself, ask me for the source code.

create file `test.v` like this:

```
module top;
initial begin
    $python("dosomething()");
end

endmodule
```

create locally also a file called verilog.py:

```
import os,sys,string
import veri   # import connection back to verilog

def dosomething():
    print 'do nothing'
```

now run Your preferred simulator. Don't forget to add +loadvp / -load to compilation script. You should see "do nothing" printed on terminal.

10.2 More examples

coming soon

10.3 Some real life use cases

1. VIP of encryption and hash signatures. Existing standard python libraries were used as golden model of encryption rtl.

2. Virtual tester. Patterns sent to test (first Credence, later J750) were read by python and driven to chip gatelevel. Saved many iterations with the test house.

3. VIP of interlaken and other serial protocols. Python classes embedded into thin encapsulations used to debug interlaken rtl.

4. Golden model of embedded processor. Special instruction-set cpu was debugged and verified this way. Among other things this cpu had math functions like sinus, cosinus and atan. All were easily verified with python math.

5. Verification processor. This python class imports instructions file and drives and checks the DUT. The syntax of instructions file is half way between assembler and high level language. It was used to implement AXI (and OCP) master and slave. It can drive signals, wait for external conditions or just wait some number of clocks, ask questions and change flow accordingly.

6. Interactive testing. in Python second thread is opened. It executes command loop. All veri and python commands are available. The simulation runs unhindered. But at any time, the user can change it's flow by issuing commands.

7. continious dsp vector generation. If the golden model of a DSP pipe is written in Python, there is additional benefit. The generator can produce vectors non stop, without the need to buffer them in huge files. it can generate checker lines on the fly as well. In this manner the simulation can run few weeks or until few bugs.

10.4 UVM in Python

work in progress