2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Unified Test Writing Framework for Pre and Post Silicon Verification

Rahulkumar Patel, Analog Devices Inc., Bengaluru, India (*Rahulkumar.patel@analog.com*)

Pablo Cholbi, Analog Devices Inc., Bengaluru, India (*Pablo.Cholbi@analog.com*)

Sivasubrahmanya Evani, Analog Devices Inc., Bengaluru, India (*Sivasubrahmanya.Evani@analog.com*)

Raman K, Analog Devices Inc., Bengaluru, India (*Raman.k@analog.com*)

***Abstract*—Day by day there is an increasing need of integrating/reusing the infrastructure across all the stages of product development right from pre-silicon design verification to post-silicon test validation, evaluation and applications board validation. We propose a framework, built on UVM centric digital verification environment that not only enables analog designers/test writers to write tests without having to know the complexities of the underlying UVM but also opens up a common communication medium over which the design, test, evaluation and application can talk and exchange tests/high level functions. This framework is generic and can be used by any project as most of the infrastructure needed is being dumped from IP-XACT by custom generators.**

Keywords—Protable Stimulus; PyBasedTest; python

I. INTRODUCTION

All the digital verification environments today are predominantly in UVM because of the verification features it offers. While digital verification community is well versed with this methodology on the other hand designers (both digital and analog) see this complexity of infrastructure as a difficult task to begin with and this makes test writing a challenge for them. Having said that, once we have silicon in lab other challenge is how we can facilitate faster bring-up of silicon (on evaluation board) and minimize the design- evaluation round trip debug delays involved. Further part of this paper explains how the proposed framework solves the above-mentioned challenges.

II. MAIN IDEA

In complex System on Chip (SoC) projects, in top-level system verification is likely where all the design sub-blocks come together and interact. Complex configurations and use cases will be simulated in the context of top-level system verification, and a valid concern which can arise if how to address this challenge and close top-level verification with good coverage while being efficient and re-usable across both pre-silicon and post silicon. In system-level verification, the Device Under Test (DUT) is likely to be controlled in a similar manner in simulation (pre-silicon) as in test/evaluation/applications (post-silicon). Typically, there will be a relatively reduced number of well-defined interfaces between the chip and the external world. These interfaces may include communication interfaces (SPI, I2C, JTAG, etc.), General Purpose Input Output (GPIO), or specific purpose input-outputs and protocols. Likewise, there is also a delimited number of high-level actions which can be performed on these interfaces: read/write frame, wait on output event, drive input, wait for a predefined amount of time.

The interaction with the DUT is very similar at system-level between simulation verification and evaluation/test. The main difference is that in simulation verification it will be the Universal Verification Components (UVC) components which interact with the device, whereas in post-silicon it will be actual hardware which is connected to the DUT. But the interfaces and the actions on them remain the same. This presents an opportunity to implement a high-level test writing framework which is common across pre-silicon and post-silicon. Only at a lower level, when driving/monitoring the interfaces it is necessary to distinguish to a specific action on the interface is handled. An implementation such as the one proposed in Fig.1 present several advantages to the deployment of a chip such as:

1. Accelerate post-silicon testing by leveraging the already developed test library during simulation verification.
2. Having a common high-level framework and test format enables seamless exchange between post-silicon and pre-silicon. This is helpful when simulating tests/configurations which have resulted in buggy or unexpected results on the bench.

Therefore, the proposed framework may provide a significant reduction in the evaluating times and allows for a tightly coupled loop between simulation and evaluation.

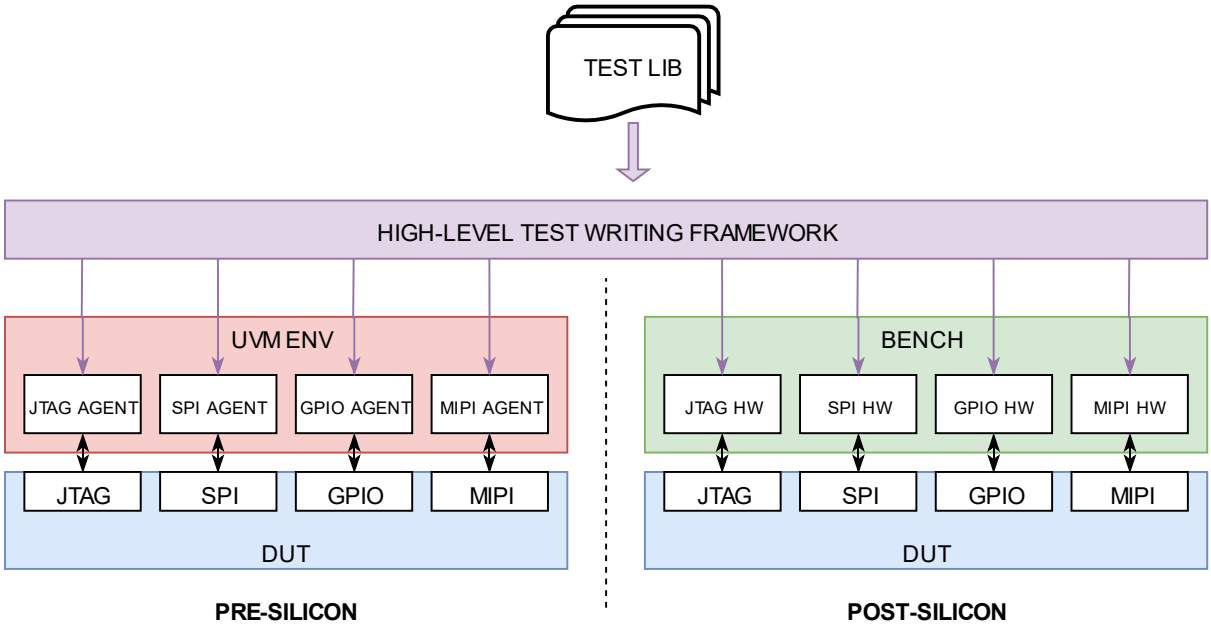

Fig. 1: Pre- and Post-Silicon Test Framework

In addition to the above, If the device will also require firmware (FW) to be developed, there is a potential opportunity for the functions developed in simulation verification to be used a guideline or example for the FW developers, if the FW development starts after top-level simulation verification starts. In this situation, there is a chance to also leverage the function writing effort and accelerate FW development to some extent.

In order to solve the first part of the problem, a bit-field centric C based infrastructure was developed. This infrastructure consists of:

1. A front-end software implemented in C on which tests are written. It provides the following features:
    a. C model of the chip register map which is auto-generated from the IP-XACT files which consist the registers information.
    b. Basic low-level instructions that define how to interact with the chip.
        i. MMR read/write
        ii. Memory read/write
        iii. Analog node voltage probing
        iv. Force/Release digital/analog nodes
        v. Waiting for specific value on analog/digital node
        vi. Delay
        vii. Random number generation
    c. Simplified access to the register map, allowing reading and writing bit-field or registers as if they were C data types while keeping track of the state of the register map and abstracting low-level details such as register addresses and bit-field offsets.
    d. Ability to use higher-level language constructs offered by C while letting the C compiler take care of the parsing.
    e. Write a library of functions to configure the chip developed by the design and verification engineers to speed up system-level test writing.

2. COSIM (analog-digital co-simulation) to run the simulation which provides the following features:
    a. Re-use of UVM setup and agents from Digital Design Verification, saving, avoiding duplicated effort and harnessing the power of a UVM.
    b. Analog simulations to execute analog models which are not SystemVerilog Real Number models.
    c. Make use of all the functionality lent by analog simulator for waveform post-simulation processing.

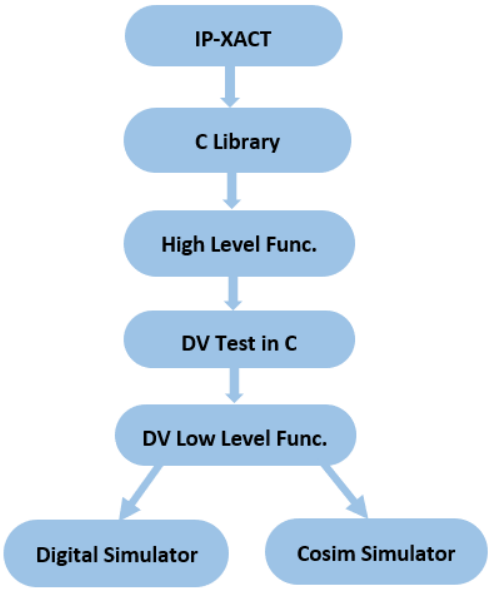

Fig. 3: C Based Framework

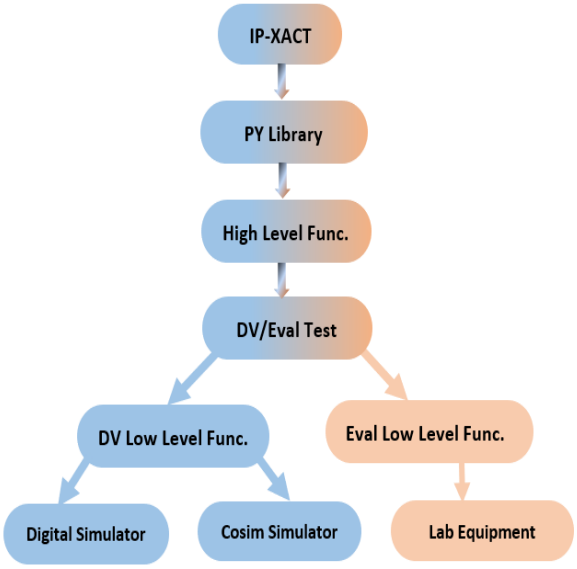

Fig. 4: Python Based Framework

With the above proposed methodology, we were able to extend all the complex features of digital verification like randomization, force/release/wait on internal digital design nodes etc. to the cosim infrastructure and also been able to get more analog designers on board to enable more cosim test writing. This first step enabled re-use of functions, tests and infrastructure across cosim, digital and firmware verification. This solves first part of the problem.

Since this framework enabled lot of re-use in pre-silicon verification efforts, we wanted to extend it for post-silicon verification as well. While C is well suited in communicating with the simulators, it is not that well suited to communicate with lab equipment as all the device drivers needed to talk to lab equipment have to be developed from scratch. Python being very popular language and lot of open source communities working on it seemed a possible solution to bridge this gap. Two hurdles on this path were bi-directional communication between Python and the simulator (SV) and all the existing tests suite which was already developed in C needs a huge migration effort. As Python can't communicate directly with SV, we had to build an intermediate bridge to solve this challenge. This intermediate bridge (as shown in Fig.5), which is transparent to the user, consists of DPIs imported/exported from SV to C and Py/C API. It is this bridge which enabled us to run the already existing test-suite without any change. However, we plan to fully migrate all the C tests to Python in long term to avoid/minimize infrastructure maintenance for both C and Py. This infrastructure is shown in the Fig. 5.

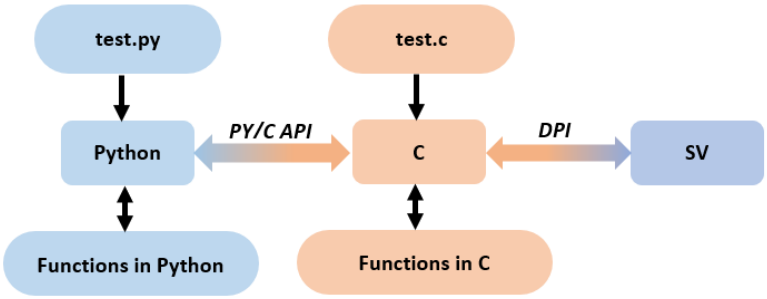

Fig. 5: Python-SV Simulation Flow

Direct Programming Interface (DPI) is basically interface between the SV and foreign programming language, in particular the C language. It allows to call the C functions from SV and to export the SV functions, so that they are called from C. Example code for SV-C communication through DPI shown in Fig. 6.

```
module block;
    import “DPI-C” context function void c_display();
    export “DPI-C” function sv_display; //No type or args

    initial c_display();

    function void sv_display();
        $display(“SV : block”);
    endfunction

endmodule : block
```

```
#include "svdpi.h"
// Imported from SystemVerilog
extern void sv_display();

void c_display()
{
  io_printf("C: c_display \n");
  sv_display();
}
```

Fig. 6: SV-C Communication through DPI

Python/C Application Programmer’s Interface (API) provides embedding and extending the Python. Embedding Python to insert the calls to Python interpreter into your C application and extending Python to load the set of C functions as part of import statement. Example code for the Py-C communication through API shown in Fig 7, Fig. 8. In this example, C function *command_main()* call the Python function *main_py()* (in test case file *command.py*) using the embedding the Python concept as shown in Fig. 7 and to call the C functions from Python require to build the module in C (*Cmodule* in this example) as shown in Fig. 8.

Anyone with basic knowledge on Python can easily get on board with this framework to write the test cases. A sample test case is shown in Fig.9 for reference.

```
#include <Python.h>
#include <stdlib.h>
extern void PyInit_CModule();
int command_main() {
  setenv("PYTHONPATH",".",1);

  /* Add a built-in module, before Py_Initialize */
  PyImport_AppendInittab("CModule", PyInit_CModule);

  /* Initialize the Python Interpreter */
  Py_Initialize();

  /*Get a reference to the command.main_py function*/
  PyObject *pFunc, *u_name, *module;
  PyObject *args;
  PyObject *kwargs;
  PyObject *result = 0;
  int retval;

  /*Get a reference to the command.main_py function*/
  u_name = PyUnicode_FromString("command");
  module = PyImport_Import(u_name);
  Py_DECREF(u_name);
  pFunc = PyObject_GetAttrString(module, "main_py");

  /* Make sure we own the GIL(global interpreter
lock) */
  PyGILState_STATE state = PyGILState_Ensure();

  /* Verify that func is a proper callable */
  if (!PyCallable_Check(pFunc)) {
          fprintf(stderr,"call_func:expected a
callable\n");
          goto fail;
  }
  /* Build arguments */
  args = Py_BuildValue("()");
  kwargs = NULL;
```

```
 /* Call the function */
  result = PyObject_Call(pFunc, args, kwargs);
  Py_DECREF(args);
  Py_XDECREF(kwargs);

  /* Check for Python exceptions (if any) */
  if (PyErr_Occurred()) {
          PyErr_Print();
          goto fail;
  }

  /* Verify the result is a int object */
  if (!PyLong_Check(result)) {
          fprintf(stderr,"call_func: callable
didn't return a Long\n");
          goto fail;
  }

  /* Create the return value */
  retval = PyLong_AsLong(result);
  Py_DECREF(result);

  /* Restore previous GIL state and return */
  PyGILState_Release(state);

  /* Done */
  Py_DECREF(pFunc);
  Py_Finalize();

  return 0;

fail:
  Py_XDECREF(result);
  abort();   // Change to something more
appropriate
}
```

Fig. 7: Embedding the Python

```c
//initCModule.c

#include <Python.h>
#include "basic_op.h"

/* This is a wrapper function for C function "mem_write". */
static PyObject* py_mem_write(PyObject* self, PyObject*
args)
{
    uint32_t addr;
    uint32_t data;
    PyArg_ParseTuple(args, "II", &addr, &data);
    // part of basic_op.h which call the SV
    mem_write(addr, data);
    return Py_BuildValue("");
}

/* This is a wrapper function for C function "mem_read". */
static PyObject* py_mem_read(PyObject* self, PyObject* args)
{
    uint32_t return_val;
    uint32_t addr;
    PyArg_ParseTuple(args, "I", &addr);
    // part of basic_op.h which call the SV
    return_val = mem_read(addr);
    return Py_BuildValue("I", return_val);
}

/* Bind Python function names to our C functions */
static PyMethodDef CModule_methods[] = {
    {"mem_write", py_mem_write, METH_VARARGS},
    {"mem_read", py_mem_read, METH_VARARGS},
        {NULL, NULL}
};
```

```c
#if PY_MAJOR_VERSION >= 3
static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT, /* m_base */
    "CModule",             /* m_name */
    NULL,                  /* m_doc */
    -1,                    /* m_size */
    CModule_methods     /* m_methods */
};
#endif

/* Python calls this to let us
initialize our module */
PyMODINIT_FUNC
PyInit_CModule(void)
{
    return PyModule_Create(&moduledef);
}
```

Fig. 8: Extending the Python

III. EXPERIMETAL RESULTS

The proposed infrastructure has been implemented in the product which is a 77GHz radar sensor chip in 28nm RF-CMOS technology. Some of the fruits of this effort are

During pre-silicon phase (C/Py-based infrastructure):

a. Automated test dumping for checking the connectivity from SPI to digital register bits in digital sims and the same approach extended to check for the right voltage levels at deep-down analog nodes in cosims. This was possible due to strict net-naming conventions followed in RTL and schematics.
b. Functions and test re-use: This enabled function re-use across digital and cosims.
c. Enables the verification infrastructure of complex data-paths present in the system by using power of Python packages like numpy to compute FFTs etc.
d. Enabled system boot-up infrastructure with minimal effort.
e. Enabled designers to get involved in test development.

During post-silicon phase (Py-based infrastructure):

a. Design shared the chip configuration library with Evaluation and Apps, accelerating debug, evaluation and demo creation.
b. Configurations from Evaluation and Applications which had issues could easily be read into the Design Verification simulation environment to debug the issue.
c. Ultra-Flex tester has a Visual Basic Front End and all the chip configurations that design has developed in Py/C were translated as VB code with minimal effort.

This idea was well received and well appreciated by design, evaluation and application teams.

```
import sys
from CModule import *
from chips import Util
import Bench
import time

def main_py():
   print ("You passed this Python program from C! Congratulations!")
   dut = Bench.chips.ADAR690x()
   # Initialize the PLL
   dut.ADC_ADPLL.adcpll_fast_init_gen(80e6, 0, 1, 500e3)
   # delay
   delay_ns(1000)

   # Register write
   dut.MISC.MISC_FILTER_CTRL.FILTER_DECIM_RATIO = 0x20
   dut.MISC.MISC_FILTER_CTRL.FILTER_OUTPUT_BITWIDTH = 0x1
   dut.dev_reg_update()

   ## Read the register and compare with excepted data
   dut.comms.read_expect(Util.Address(dut.MISC.MISC_SCRATCHPAD_0), 0x30, 0x04)

   ## Memory Write
   dut.comms.write(0x80000500, 0x12345678, 0x04)

   ## Memory Read
   read_data = dut.comms.read(0x80000500, 0x04)
   if(read_data != 0x87654321):
      read_data = dut.comms.read(0x80000500, 0x04)
   print (read_data)

   #force the signal
   force_digital("`DIGITAL_TOP.muxout_in",0x01)
   #wait for expected value on specificed signal
   wait_state("GPIO[1]",1,10)
   #release the signal
   release("`DIGITAL_TOP.muxout_in")

   #generate a random value
   random_data = gen_random_data(1,10)
   dut.MISC.MISC_SCRATCHPAD_1.SCRATCHPAD_1 = random_data
   dut.dev_reg_update()

   #Same function can be used for all instand of AFE,RX
   for RX_name, AFE_name in [['RX0','AFE0'],['RX1','AFE1'],['RX2','AFE2'],['RX3','AFE3']]:
      RX = Util.get_subsystems_by_instance_name(dut,RX_name)
      AFE = Util.get_subsystems_by_instance_name(dut,AFE_name)
      RX.channel_init()
      AFE.afe_channel_init()

   return 0
```

Fig. 9: command.py

IV. CONCLUSION

Early planning and consensus amongst the different sub-teams allowed us to specify and implement a framework which accelerated directed test development effort in pre- and post-silicon. Silicon evaluation timeline was shortened because the pre-silicon verification tests could be leveraged and used on the bench. In addition to this, problematic configurations on the bench were quickly and transparently sent back to simulation verification to debug the issue with increased observability.

This allowed for the total verification time of the project to be shortened and the time-to-market of the project was pulled-in, while most likely also improving silicon quality. This framework, and the underlying idea, helped meet the product-line deadlines and increases the chances of commercial success of the device. DUT debug was streamlines in all stages of top-level verification.

ACKNOWLEDGMENT

We would like to acknowledge Jaime Rustarazo, Purushottam Kumar from evaluation team for sharing ideas and helping us in trying this out in Lab.