Unified Test Writing Framework for Pre and Post Silicon Verification

Rahulkumar Patel
Pablo Cholbi
Sivasubrahmanya Evani
Raman K
Agenda

- DV Environment overview
- The Journey
- Challenges Faced
- Solutions
- Results
DV Environment Overview

- Top Level view of verification environment
Problem Statement

- Why is this framework needed?
  - P1: Test writing in UVM for non digital folks is a challenge. How do we get them on board (especially for mixed signal simulation)?
  - P2: How can we reuse all/most of the pre silicon infrastructure developed to enable faster post silicon debug?
The Journey

- Problem 1: How do we get more people on board?

  - Pros
    - Simple txt interface, easy to write.
    - Saves Compilation time

  - Cons
    - Easy to write but difficult to maintain
    - Vulnerable to regMap changes
    - Difficult to port across products
The Journey

- Problem 1: How do we get more people on board?

  - C wrapper on text based interface
  - Bit field Centric
  - Pros
    - Easy to write/understand
    - Portable
    - Allows to develop layers of functions
  - Cons
    - Unidirectional, no conditional loops/branches

```c
#include "basic_op.h"
#include "reglayers.h"
#include <stdio.h>

int main()
{
    setup(); // Configure
    spi.spi_initfaceconfig.dem = 1;
    spi_reg_update();
    force_digital("DIGITAL_TOP muxout_in", CLRV1);
    wait_state("VPD1", 0, 70);
    delay_ms(1);
    nisc.nisc_muxoutmuxoutsel = 1;
    dev_reg_update();
    dev_reg_read_explore(adc_15ll_adc11_stat_addr, ADC55555);
    cleanup(); // Finish gracefully
    return(0);
}
```

`test.c`
The Journey

- Key Feature Set
  - MMR read/write
  - Memory read/write
  - Analog node voltage probing
  - Force/Release/Wait on digital nodes
  - Waiting for specific voltage on analog node
  - Delay
The Journey

- Recap:
  - Unidirectional communication

How to mimic the host processor?
Direct Programming Interface (DPI)
  • Import of the C functions and export the SV functions
The Journey

Taped Out...!!!

Silicon Arrived...!!!

Cont...
The Journey

- Problem 2: How can we reuse the pre-silicon infrastructure for post-silicon?

  **Eval. Engineers:**
  Develop the test cases in Python

  **DV Engineers:**
  Develop the test cases in C/SV/UVM

- Python being very popular language and lot of open source communities working on it, makes it well suited to communicate with lab equipment

- C is well suited in communicating with the simulators, it is not that well suited to communicate with lab equipment
Challenges

- Python <=> SV
  - SV can’t communicate directly with Python
Solution

- API-DPI Cascade Bridge
  - *Direct Programming Interface (DPI)* for SystemVerilog–C
  - *Python-C API* for Python–C

- API provides access to the Python interpreter from C code.
  - **Embedding Python**: Inserting calls to Python interpreter into your C application and Calling Python code at specific time
  - **Extending Python**: Python interpreter loads the set of C functions as part of import statement
Solution

Cont...

Functions in Python

Python

test.py

Functions in C

C

PY/C API

DPI

SV

main_py()

command_main()

uvm_test

mem_write()

sv_mem_write()

mem_write_seq
Embedding Python: Call python method `main_py` of `test.py` from C

```c
/* 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
```

Extending Python: Create a module which contain set of C functions

```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);
}
```
Byproduct

- Pre-Silicon Test suite developed is still valid ...!!!
Framework Overview

- Most of the framework auto dumped form YODA through custom script
- Test cases / functions are reused across all the stage of product development
Sample test Case

```python
test.py
import sys
from Module import *
import time
import Bench
import time

def main_py():
    print("You passed this Python program from C! Congratulations!")
    dur B Bench.chips.ADA150x()
    * Initialize the PLL
    dur A ADFPPLLdsp1_pll_fast_init_gen(5000, 0, 1, 10000)
    delay ns(1000)
    Register write
    dur MISC_MISC_FILTER_CTRL.FILTER_DECIM_RATIO = 0x20
    dur MISC_MISC_FILTER_CTRL.FILTER_OUTPUT_BITWIDTH = 0x1
    * Register Read
    dur comm.read_expect(UInt16_Address(dur MISC_MISC_SCRATCHPAD_0), 0x30, 0x04)
    Memory write
    dur comm.write(UInt16_Address(DCCON_STRADD0=0x5000), 0x12345670, 0x04)
    Memory Read
    read_data = dur comm.read(UInt16_Address(DCCON_STRADD0=0x5000), 0x04)
    if (read_data != 0x87654321):
        read_data = dur comm.read(UInt16_Address(DCCON_STRADD0=0x5000), 0x04)
        print (read_data)
    Force/Wait/Release
    force_digital1("DIGITAL_TOP.muxout_in", 0x01)
    # wait for expected value on specified signal
    wait_time("ND10[1]", 1.10
    # release the signal
    release("DIGITAL_TOP.muxout_in")
    # Generate a random value
    random_data = gen_random_data(1, 10)
    dur MISC_MISC_SCRATCHPAD_1.SCRATCHPAD_1 = random_data
    dur der_reg_update()
    # Same function can be used for all instead of AFE, RX
    for RX_name, AFE_name in {["RX0", "AFE0"], ["RX1", "AFE1"], ["RX2", "AFE2"], ["RX3", "AFE3"]}:
        RX = Util.get_subsystems_by_instance_name(dur RX_name)
        AFE = Util.get_subsystems_by_instance_name(dur AFE_name)
        RX.channel_init()
        AFE.afe_channel1_init()
        return 0
```
Extended Solution

- Automated Checks for the connectivity from digital register bits to the relevant analog nodes in mixed signal simulation.
Extended Solution Cont...

```python
import sys
from Module import *
from chips import Util
import Bench
import time

def main_py():
    dut = Bench.chips.ADA690x()

    comment("Testing afe0_adc_bias_ctrl0.adc_dac23_ncs_ctrl")
    # local_hier[] = "digimic.die.rx_section.rxchannel0.afe>adc_dac23.ncs_ctrl"
    signal="DIGIMIC.DIE.RX_SECTION.RXCHANNEL0.AFE>ADC_DAC23_NCS_CTRL"
    comment("signal="+str(signal)) # FOUND HIERARCHY OF AFE0.ADC_BIAS_CTRL0.ADC_DAC23_NCS_CTRL

    # Bus check
    for i in range(2):
        dut.AFE0.ADC_BIAS_CTRL0.ADC_DAC23_NCS_CTRL = 0X1 << i; dut.dev_reg_update();
        dut.comms.read_expect(Uutil.Address(dut.AFE0.ADC_BIAS_CTRL0), 0x1<<[i+24],0x04);
        probe_bus_expect(signal, 0x1 << i);

    dut.AFE0.ADC_BIAS_CTRL0.ADC_DAC23_NCS_CTRL = 0X0; dut.dev_reg_update();
    dut.comms.read_expect(Uutil.Address(dut.AFE0.ADC_BIAS_CTRL0), 0x0,0x04);
    probe_bus_expect(signal, 0x0);

    comment("Testing afe0_adc_bias_ctrl0.adc_amp3_stgl_bias")
    # local_hier[] = "digimic.die.rx_section.rxchannel0.afe>adc_amp3_stgl_bias"
    signal="DIGIMIC.DIE.RX_SECTION.RXCHANNEL0.AFE>ADC_Amp3_STGL_BIAS"
    comment("signal="+str(signal)) # FOUND HIERARCHY OF AFE0.ADC_BIAS_CTRL0.ADC_Amp3_STGL_BIAS

    # Bus check
    for i in range(2):
        dut.AFE0.ADC_BIAS_CTRL0.ADC_Amp3_STGL_BIAS = 0X1 << i; dut.dev_reg_update();
        dut.comms.read_expect(Uutil.Address(dut.AFE0.ADC_BIAS_CTRL0), 0x1<<[i+24],0x04);
        probe_bus_expect(signal, 0x1 << i);

    dut.AFE0.ADC_BIAS_CTRL0.ADC_Amp3_STGL_BIAS = 0X0; dut.dev_reg_update();
    dut.comms.read_expect(Uutil.Address(dut.AFE0.ADC_BIAS_CTRL0), 0x0,0x04);
    probe_bus_expect(signal, 0x0);

    return 0
```
Other benefits

- Functions and test re-use across digital and mixed signal simulation.
- Enables the verification infrastructure of complex data-paths present in the system
  - Using power of Python packages like numpy to compute FFTs etc.
- Enabled system boot-up infrastructure with minimal effort.
- Enabled designers to get involved in test development.
- Accelerating debug, evaluation and demo creation.
Results

- Eval and Apps support
  - Easy debug of configurations issues from Evaluation and Applications
- Tester support
  - UltrFlex tester was used which has a Visual Basic (VB) Front End and all the chip configurations that were developed in Py/C were translated as VB code with minimal effort.