Guaranteed Vertical Reuse – C Execution In a UVM Environment

Rachida EL IDRISSI, ST-Ericsson, Rabat, Morocco,

Phone: 00 212 5 37 67 86 71
Email: rachida.el-idrissi@stericsson.com

Alain GONIER, Mentor Graphics, France

Phone: 00 33 1 40 94 7452

Email: alain_gonier@mentor.com

Introduction

At ST-Ericsson Rabat, Morocco, we develop hardware design IP and related driver software for wireless mobile platforms created elsewhere within the company. Our verification team has to achieve high levels of quality and provide verification deliverables that can be reused by integration teams to make their work more effective.

We recently adopted the Universal Verification Methodology (UVM) since it provides an open-source, vendor-independent, consistent architecture that allows us to share verification components and stimulus throughout the company. This paper describes a C-API package built on top of the UVM that allows us to develop software early in the life of a design IP. Our results are twofold: first, we are able to deliver software tests that can be reused by hardware integration and verification teams; second, we are able to deliver fully functional driver code to software developers early in the overall project cycle.

Project background

We developed the C-API package during development of new design IP for the MIPI Low Latency Interface (LLI). The LLI allows two chips to share resources over a point-to-point high-speed interface.

The LLI architecture is described in figure 1.

Our UVM verification environment was used to meet an exhaustive verification plan for the LLI design IP using sequence-based stimulus. The environment is comprehensive and allowed us to use multi-channel constrained random stimulus combined with scoreboarding and functional coverage analysis that checked each physical and logical layer of the LLI protocol.

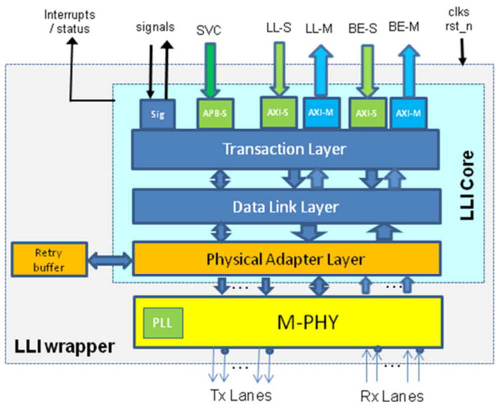


Figure 1- LLI logical layers

One of our required deliverables was a set of test cases that could be run as software by platform integration teams. ST-Ericsson has a SoC software framework that defines how these test cases should work and their interfaces. We realised that we could reuse the UVM environment to develop these test cases by developing an API package based on the use of the SystemVerilog Direct Programming Interface (DPI).

The LLI UVM verification architecture

The LLI DUV has several standard interfaces, some of which are design specific. The standard interfaces are the AMBA APB3 and AXI3, together with the MIPI LLI physical interface. The design-specific interfaces concern low-level control and interrupts.

The APB3 interface is used to program the registers within the LLI subsystem. A UVM register model was developed to abstract sequence-based register stimulus.

The AXI interfaces are used for data transfer between the two sides of the LLI. The LLI subsystem has four physical channels that are used for this purpose: a low latency (LL) master interface; a LL slave interface; a best effort (BE) master interface; and a BE slave interface. Each interface was connected to a master or slave AXI3 Questa Verification IP (QVIP) in order to source or sink data transfer across the LLI link as shown in figure 2

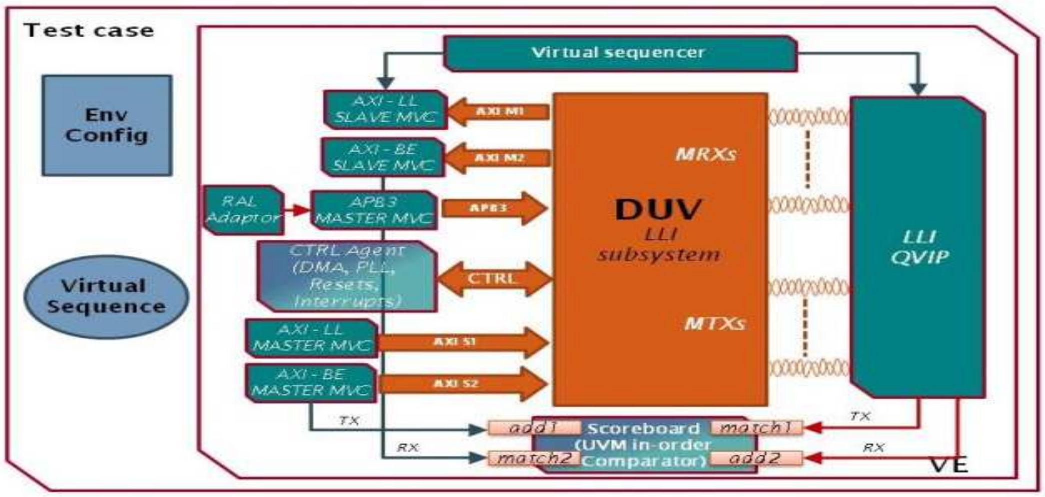


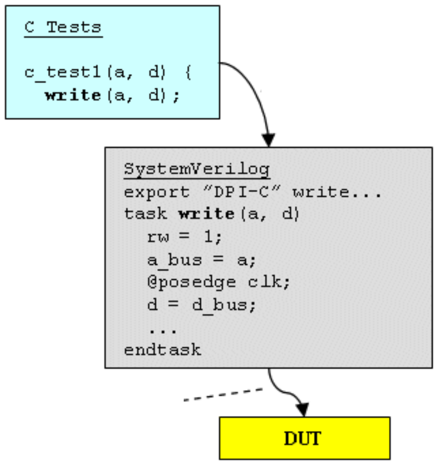
Figure 2- UVM Testbench architecture

A LLI QVIP connected to the PHY signals was used to represent the behaviour of a partner LLI device. The QVIP supports a transactional interface that makes it straightforward to configure and send different types of LLI traffic, including sourcing and sinking LL and BE data transfers. Both the LLI subsystem and the LLI QVIP could be configured as a master or a slave LLI device.

Scoreboards and functional coverage monitors were developed that used the different transaction types available within the environment to check for correct device behaviour and to collect functional coverage.

C-API integration with the UVM environment

To support the use of a C-API, an optional layer was created for the UVM testbench. This was implemented as the lli_c_api SystemVerilog package. The C-API layer is implemented using the SystemVerilog DPI, which allows a number of SystemVerilog tasks and functions to be made available to C programs and executed as depicted in the Figure 3- SystemVerilog DPI mechanism.



```
C Tests

c_test1(a, d) {
  write(a, d);
```

```
SystemVerilog
export "DPI-C" write...
task write(a, d)
  rw = 1;
  a_bus = a;
  @posedge clk;
  d = d_bus;
  ...
endtask
```

Figure 3- SystemVerilog DPI mechanism

Furthermore, you can refer to Mentor Graphics Verification Academy located at http://verificationacademy.com/uvm-ovm/CBasedStimulus for a detailed description of C/UVM tests integration principles.

In our environment, the exposed API provides access to four main areas of functionality as shown in Figure 4 below.

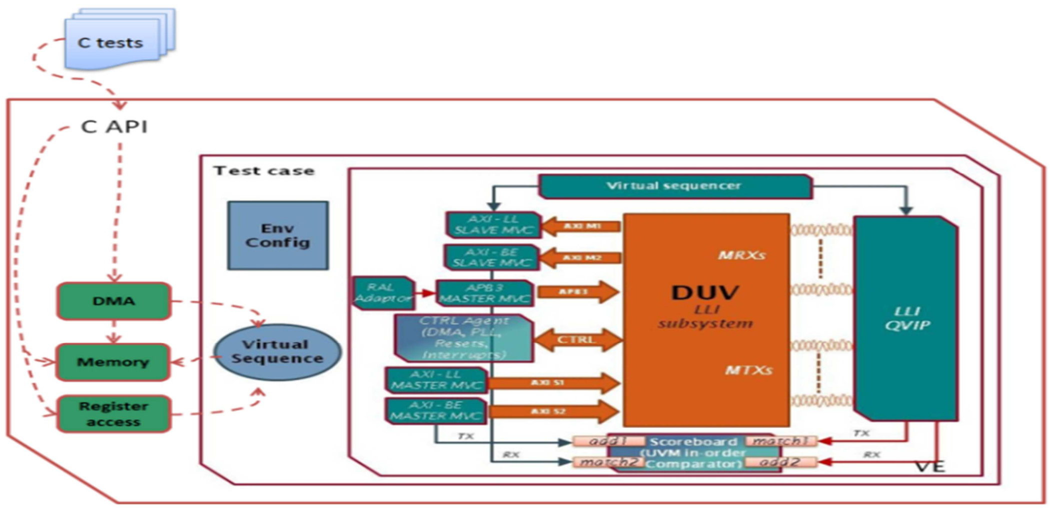


Figure 4- C API access block diagram

The package implements the testbench side of the API as either DUT-centric (by configuring hardware registers via an APB3 QVIP and sending data traffic using DMA via the AXI3 QVIPs), or as LLI QVIP-centric (by configuring the QVIP and streaming data traffic through it to emulate DMA transfers). C test cases were developed using this API using a runtime parameter to determine which side of the test bench the stimulus was directed to.

The 4 main areas are:

1. Register reads and writes using an address map implemented in the package.
   a. DPI_HAL_READ32 ()
   b. DPI_HAL_WRITE32 ()
2. DMA Controller API, used for setting up a DMA transfer between the DUV and the LLI QVIP on LL and BE channels. The list of functions used are below
   a. DPI_dmac_drv_enable
   b. DPI_dmac_drv_disable
   c. DPI_dmac_drv_is_dma_enabled
   d. DPI_dmac_drv_get_dma_enabled_channels
   e. DPI_dmac_drv_start_channel
   f. DPI_dmac_drv_is_channel_done
   g. DPI_dmac_drv_clear_channel_done
   h. …

A code snippet below shows how one function is implemented:

function automatic byte ***DPI_dmac_drv_is_dma_enabled*** *(device_type_e device);*
return DMAC[device].get_dma_enable();
endfunction: ***DPI_dmac_drv_is_dma_enabled***

It is then exported in the lli C API package as shown below and thus callable by the C test

// DMA Functionality DPI Method Exports:
export "DPI-C" function DPI_dmac_drv_enable;
export "DPI-C" function DPI_dmac_drv_disable;
export "DPI-C" function DPI_dmac_drv_is_dma_enabled;
….
…

3. Memory block functions, used for setting up and checking areas of memory used by the DMA controller.
   a. DPI_init_mem
   b. DPI_get_mem

A code snippet below shows how one function is implemented:

```
function automatic int DPI_get_mem(device_type_e device, int addr);
`uvm_info("get_mem",$sformatf("%s get_mem at @:%08x START ... \n",device,addr), UVM_MEDIUM)
if( (addr >= `LMI_MEMORY_BASE_ADDRESS) && (addr <= (`LMI_MEMORY_BASE_ADDRESS +
`LMI_MEM_RANGE)))
return LMI[device].get(addr);//local LMI memory init
else if     ((addr >= `EMI_MEMORY_BASE_ADDRESS) && (addr <= (`EMI_MEMORY_BASE_ADDRESS +
`EMI_MEM_RANGE)))
return EMI[device].get(addr);//local EMI memory init
else//anything else is just stored temporarly into a "virtual" memory
return mem[device].get(addr);
endfunction: DPI_get_mem
```

4. Delay functions.

the main goal of those functions is to insert some delays in ns or us and also to establich the synchronisation between the C and SV domains.

a. wait_n_ns
b. wait_n_us
c. esw_sync

A code snippet below shows how one function is implemented:

```
  task automatic wait_n_ns(int n);
 time p;
 p = 1ns;
 p = p * n;
 #p;
endtask: wait_n_ns
```

To link the DPI functions/tasks to the C test, during compilation a header file is automatically generated declaring all exported/imported functions prototypes to be use either from SV to C or from C to SV. In our case the generated file is called lli_c_api.h. and contains the few functions detailed previously.

C test execution example

We will take a concrete example to illustrate a C test/UVM testbench interaction. As an example, one of our tests generates LL channel traffic concurrently on both the DUT and TB side to check that the link is working properly. Note, except during the initial setup process, both the DUT and the TB side are generally running the same SW.

In that test, on the DUT and TB side, a call to the API will be done to request an LL channel transfer. The C test will call the API function DPI_HAL_READ32 through the DPI, requesting a 32-bit read access in the range of the LL channel to happen in the UVM TB.

A code snippet of that API function is shown below:

```
task automatic DPI_HAL_READ32(input device_type_e device, input int address, output int data);

`uvm_info("DPI_HAL_READ32",$sformatf("%s: HAL READ %0h STARTED ********************\n", device,
address),UVM_LOW);
// MF Address range:
if( ( device==DUT && ((address >= `mf_base) && (address <= (`mf_base + `mf_length))) )//DUT READ ALL MF
ADDRESSES
|| ( device==TB && (address == `ALL_DONE_STATUS_OFFSET || address == `OUTPUT_DONE_OFFSET) ) //TB
ONLY READ MF STATUS)
do_mf_read(device,address, data);
// AXI Address range:
else if((address >= `LLI_LL_BASE_ADDRESS) && (address <= `LLI_LL_BASE_ADDRESS + `LLI_MEM_RANGE))
do_ll_be_read(device, LL, 32, address, data);
else if((address >= `LLI_BE_BASE_ADDRESS) && (address <= `LLI_BE_BASE_ADDRESS + `LLI_MEM_RANGE))
do_ll_be_read(device, BE, 32, address, data);
else   begin
```

```
// Anything else:
if(device == DUT)
                    apb_read(device, address[15:0], data);
else begin
//decode QVIP read access to local/remote registers
….
          end
`uvm_info("DPI_HAL_READ32",$sformatf("%s: HAL READ %0h @ %0h COMPLETED ********************\n",
device, data, address),UVM_LOW);
endtask: DPI_HAL_READ32
```

As we can see in the code above, the call to the API function by the DUT translates into a call to do_ll_be_read with parameter device=DUT, channel=LL, size=32 and address equal to the address location to be read; data will return the read value. Whilst a call by the TB will call the same function but with the device parameter set to TB.

The code below shows the function do_ll_be_read:

```
task automatic do_ll_be_read(input device_type_e device,input rw_seq_channel_t channel, input int size, input
int address, output int data);

lli_ll_be_seq_base          ll_rd_seq;//base class for LL/BE transfer
lli_ll_be_read_DUT_seq   ll_rd_seq_DUT= lli_ll_be_read_DUT_seq::type_id::create("ll_rd_seq_DUT");
lli_ll_be_read_TB_seq     ll_rd_seq_TB= lli_ll_be_read_TB_seq::type_id::create("ll_rd_seq_TB");
//specialized sequence depending on initiator
case(device)
          DUT: $cast(ll_rd_seq,ll_rd_seq_DUT);
TB    $cast(ll_rd_seq,ll_rd_seq_TB);
endcase
//setup transaction -> AXI request if DUT and QVIP LL request if TB
//        constrain transfer size to "size" bits
if        (!ll_rd_seq.randomize() with
{
ll_rd_seq.addr      == address;
transfer_byte_size          == (local::size/8);
}
)
……..
ll_rd_seq.transfer_channel            = channel;
ll_rd_seq.start(v_sqr);
……
          data = ll_rd_seq.data[0];
endtask: do_ll_be_read
```

Eventually, the C test call to DPI_HAL_READ32 is translated into a UVM sequence generating the appropriate read transaction. If the function is called by the DUT, it will translate into a sequence generating an LL read transaction of the LL AXI interface.

The code below shows the DUT LL sequence:

```
// DUT->TB LL/BE READ REQ
class lli_ll_be_read_DUT_seq extends lli_ll_be_DUT_seq_base;
……
task body;
```

```
axi_atomic_read_seq read_seq = axi_atomic_read_seq::type_id::create("read_seq");

super.body();
if          (!read_seq.randomize() with
{
addr == local::this.addr;
transfer_byte_size==local::this.transfer_byte_size;
}
)
….
read_seq.start(transfer_channel_sqr[transfer_channel]);
//return data and response
data = new[read_seq.data.size()];
foreach(data[i])
            data[i] = read_seq.data[i];
resp = new[read_seq.resp.size()];
foreach(resp[i])
resp[i] = read_seq.resp[i];

endtask: body
endclass: lli_ll_be_read_DUT_seq
```

In the case of the TB, it will translate into a sequence generating an LL read transaction but on the QVIP LLI LL interface.

The code below shows the sequence generating the TB LL sequence:

```
// TB->DUT LL/BE READ REQ
class lli_ll_be_read_TB_seq  extends lli_ll_be_TB_seq_base;
…..
task body();

lli_atomic_read_t            read_seq= lli_atomic_read_t::type_id::create("read_seq");
int j=0;

super.body();

if          (!read_seq.randomize() with
{
Addr == local::this.addr;
transfer_byte_size== local::this.transfer_byte_size;
}
)
case(transfer_channel)
LL: read_seq.m_ch_id_user = MGC_LLI_CH_LL_REQ;
BE: read_seq.m_ch_id_user = MGC_LLI_CH_BE_REQ;
endcase
read_seq.start(v_sqr.lli_sqr);
….
//fill in data
data[j]   = '0;
foreach(read_seq.m_read_data[i])
begin
data[j] = data[j] | (read_seq.m_read_data[i]<< ( (i % (AXI_LL_MASTER_PARAMS::AXI_WDATA_WIDTH/8))*8));
if((i+1) % (AXI_LL_MASTER_PARAMS::AXI_WDATA_WIDTH/8) == 0)
begin
j++;//go to next data word
```

```
data[j] = '0;//initialize data word
end
end
endtask: body
endclass: llih_ll_be_read_TB_seq
```

In order to start the C program, DPI tasks need to be called from SystemVerilog start_DUT_c_code() and start_TB_c_code();

```
task run_phase(uvm_phase phase);
…
phase.raise_objection(this);
//specific C test sequences to be launched

//launch both SW and response sequence for QVIP LL/BE requests
fork
start_DUT_c_code();
start_TB_c_code();
be_slave_seq.start(m_env.v_sequencer.axi_ll_slave_sqr);//slave sequence to get QVIP->DUT BE req LLI->AXI
ll_slave_seq.start(m_env.v_sequencer.axi_be_slave_sqr);//slave sequence to get QVIP->DUT LL req LLI->AXI
join
phase.drop_objection(this);
endtask: run_phase
```

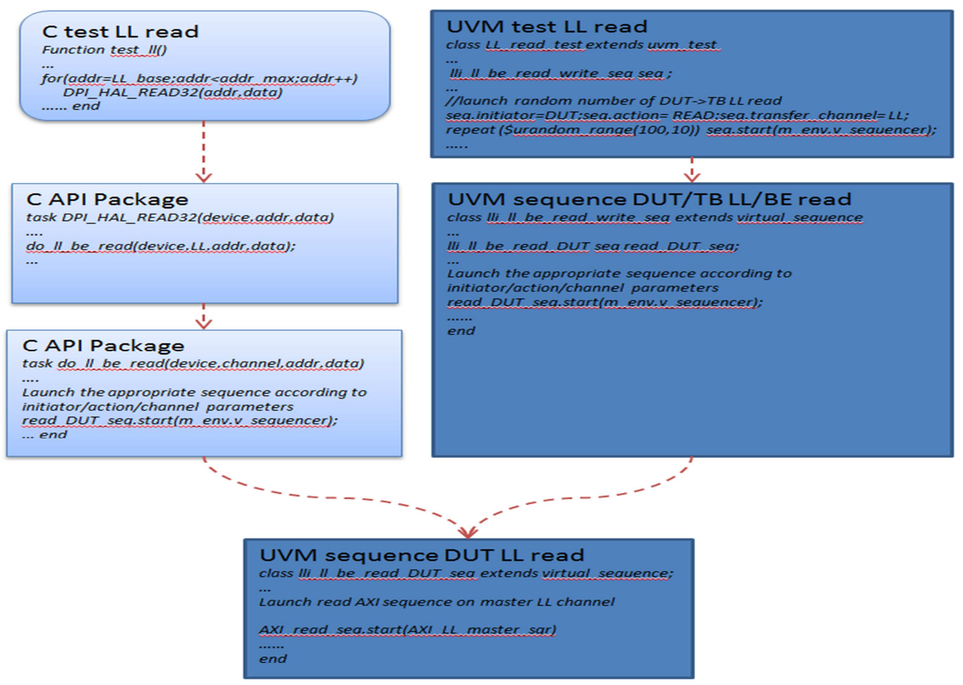
In the C code, we need to implement a function call with the same name,

```
int start_DUT_c_code()
{
set_hal_mode(DUT);
//-- Initialize LLI Driver


lli_test_initialize(&lli_local, INSTANCE_NAME, 0, INSTANCE_MPHY, INSTANCE_REF_CLK);
//-- Run Test
run_test(&lli_local);
printf(" ************ %s Test Done ! ************\n", INSTANCE_NAME);

return(0);
}


int start_TB_c_code()
{
set_hal_mode(TB);
//-- Initialize LLI Driver
lli_test_initialize(&lli_local,INSTANCE_NAME, LLI_SVC_BASE_ADDRESS,INSTANCE_MPHY, INSTANCE_REF_CLK);
//-- Run Test
run_test(&lli_local);

printf(" ************ %s Test Done ! ************\n", INSTANCE_NAME);
return(0);
}
```

The UVM testbench should call the above functions during an active UVM phase such as the run_phase; in our case, in order to start C execution.

Horizontal reuse across UVM tests and C tests

The overhead to execute a C test is limited, only residing in writing the C API package defining the DPI tasks. The UVM sequences described in the previous section will be reused for writing pure SV tests and generate traffic on the different interfaces. The result, depicted below with an example of C test and SV test both initiating LL read transfer on the DUT side, is easier test development and maximized reuse and maturity of the TB.



```
C test LL read
Function test_ll()
...
for(addr=LL_base;addr<addr_max;addr++)
    DPI_HAL_READ32(addr,data)
...... end
```

```
C API Package
task DPI_HAL_READ32(device,addr,data)
....
do_ll_be_read(device,LL,addr,data);
...
```

```
C API Package
task do_ll_be_read(device,channel,addr,data)
....
Launch the appropriate sequence according to
initiator/action/channel  parameters
read_DUT_seq.start(m_env.v_sequencer);
... end
```

```
UVM test LL read
class LL_read_test extends uvm_test
...
 lli_ll_be_read_write_seq seq ;
...
//launch random number of DUT->TB LL read
seq.initiator=DUT;seq.action= READ;seq.transfer_channel= LL;
repeat ($urandom_range(100,10)) seq.start(m_env.v_sequencer);
.....
```

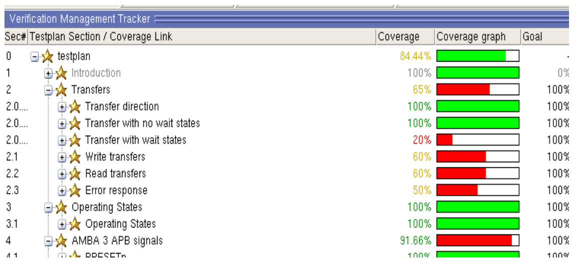
```
UVM sequence DUT/TB LL/BE read
class lli_ll_be_read_write_seq extends virtual_sequence
...
lli_ll_be_read_DUT seq read_DUT_seq;
...
Launch the appropriate sequence according to
initiator/action/channel  parameters
read_DUT_seq.start(m_env.v_sequencer);
......
end
```

```
UVM sequence DUT LL read
class lli_ll_be_read_DUT_seq extends virtual_sequence;
...
Launch read AXI sequence on master LL channel

AXI_read_seq.start(AXI_LL_master_sqr)
......
end
```

Figure 5- C vs. UVM test reuse

C tests contribution to coverage

Since we are running the C tests on a SV UVM testbench containing our coverage model, we can measure the coverage contribution of our C test suite and more importantly, check which functionalities are covered and which are not by the test suite. It is then up to the verification team to choose to address the coverage holes using a C test or a SV test.



Verification Management Tracker

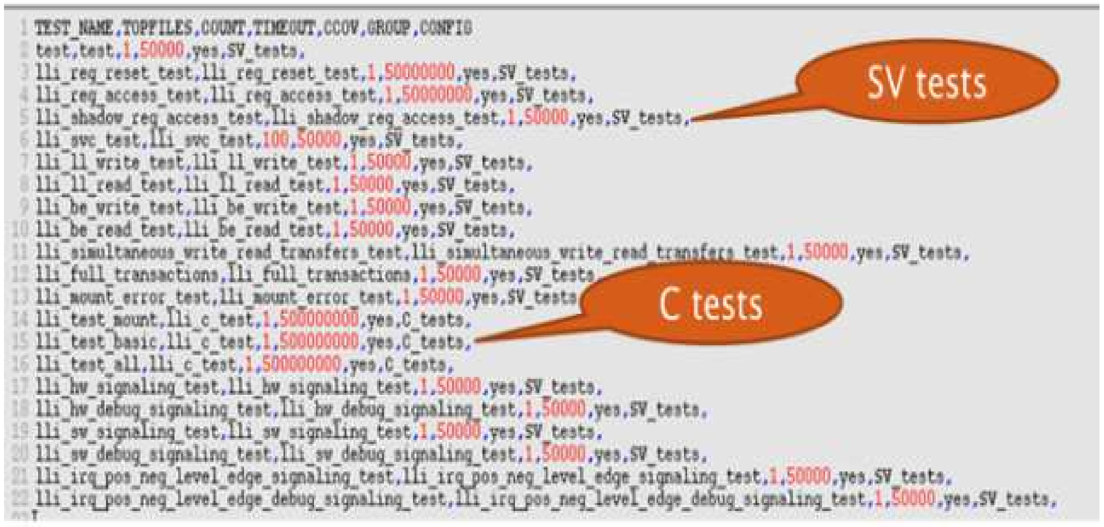
Sec#	Testplan Section / Coverage Link	Coverage	Coverage graph	Goal
0	testplan	84.44%	🟩	-
1	Introduction	100%	🟩	0%
2	Transfers	65%	🟥	100%
2.0....	Transfer direction	100%	🟩	100%
2.0....	Transfer with no wait states	100%	🟩	100%
2.0....	Transfer with wait states	20%	🟥	100%
2.1	Write transfers	60%	🟥	100%
2.2	Read transfers	60%	🟥	100%
2.3	Error response	50%	🟥	100%
3	Operating States	100%	🟩	100%
3.1	Operating States	100%	🟩	100%
4	AMBA 3 APB signals	91.66%	🟥	100%
4.1	PRESETn	100%	🟩	100%

Figure 6- Functional coverage snapshot

For instance, the snapshot above shows the coverage contribution of one C test on the APB3 interface protocol test plan. We can see that almost 85% of the protocol is covered running that single test. Other C tests—for instance setting LL/BE transfers—will contribute to the coverage of the AXI protocol as it is the interface used for the transfer.

C/SV tests regression system

Our regression system is using a text file as an input to define the regression test suite to be run. By adding a new attribute "GROUP," it was adapted to allow users to choose between running UVM sequence based tests and/or C-based tests.



```
TEST_NAME,TOPFILES,COUNT,TIMEOUT,CCOV,GROUP,CONFIG
test,test,1,50000,yes,SV_tests,
lli_req_reset_test,lli_req_reset_test,1,50000000,yes,SV_tests,
lli_req_access_test,lli_req_access_test,1,50000000,yes,SV_tests,
lli_shadow_req_access_test,lli_shadow_req_access_test,1,50000,yes,SV_tests,
lli_svc_test,lli_svc_test,100,50000,yes,SV_tests,
lli_ll_write_test,lli_ll_write_test,1,50000,yes,SV_tests,
lli_ll_read_test,lli_ll_read_test,1,50000,yes,SV_tests,
lli_be_write_test,lli_be_write_test,1,50000,yes,SV_tests,
lli_be_read_test,lli_be_read_test,1,50000,yes,SV_tests,
lli_simultaneous_write_read_transfers_test,lli_simultaneous_write_read_transfers_test,1,50000,yes,SV_tests,
lli_full_transactions,lli_full_transactions,1,50000,yes,SV_tests,
lli_mount_error_test,lli_mount_error_test,1,50000,yes,SV_tests,
lli_test_mount,lli_c_test,1,500000000,yes,C_tests,
lli_test_basic,lli_c_test,1,500000000,yes,C_tests,
lli_test_all,lli_c_test,1,500000000,yes,C_tests,
lli_hw_signaling_test,lli_hw_signaling_test,1,50000,yes,SV_tests,
lli_hw_debug_signaling_test,lli_hw_debug_signaling_test,1,50000,yes,SV_tests,
lli_sw_signaling_test,lli_sw_signaling_test,1,50000,yes,SV_tests,
lli_sw_debug_signaling_test,lli_sw_debug_signaling_test,1,50000,yes,SV_tests,
lli_irq_pos_neg_level_edge_signaling_test,lli_irq_pos_neg_level_edge_signaling_test,1,50000,yes,SV_tests,
lli_irq_pos_neg_level_edge_debug_signaling_test,lli_irq_pos_neg_level_edge_debug_signaling_test,1,50000,yes,SV_tests,
```

Figure 7- Regression suite input

For compilation we have a top makefile that takes care of SV compilation and calls another dedicated makefile that handles C compilation (tests and driver).

The top makefile has a C_TESTNAME variable that can be overridden at invocation to define the C test name to be compiled. If we run an SV test, we can omit C_TESTNAME:

<command>if ((%GROUP%) == "C_tests") then</command>

<command>make -f (%MAKE_COMPILE%) all C_TESTNAME=(%TEST_NAME%) QUESTA_MVC_HOME=(%QUESTA_MVC_HOME%) </command>

<command>else </command>

<command>make -f (%MAKE_COMPILE%) all QUESTA_MVC_HOME=(%QUESTA_MVC_HOME%) </command>

<command>endif </command>

In the case of a C test, C_TESTNAME will be passed to the C compilation makefile and thus create the according .so files to be loaded at simulation.

In the simulation phase, to run a C test we need to load the shared object for both DUT and TB threads. We rely on the new attribute "GROUP" to add adequate options for the simulation, such as the .so file to be loaded according to the test we are running.

<command>if ((%GROUP%) == "C_tests") then</command>

<command>setenv ***VE_SIM_OPTS*** *"-mvchome (%QUESTA_MVC_HOME%) +UVM_TESTNAME=(%TOPFILES%) -L dut_lib -L tb_lib -L qvl_lib -t 1ps -64 -suppress 8683 -L qvl_lib -L mtiUvm -sv_lib $QUESTA_HOME/uvm-1.1a/linux_x86_64/uvm_dpi -sv_lib ./Tests_obj/main_(%TEST_NAME%)_DUT -sv_lib ./Tests_obj/main_(%TEST_NAME%)_TB "</command>*

<command>else </command>

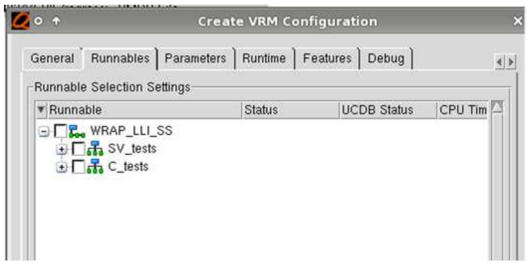
<command>setenv ***VE_SIM_OPTS*** *"-mvchome (%QUESTA_MVC_HOME%) +UVM_TESTNAME=(%TOPFILES%) -L dut_lib -L tb_lib -L qvl_lib -t 1ps -64 -suppress 8683 -L qvl_lib*

-L mtiUvm -sv_lib $QUESTA_HOME/uvm-1.1a/linux_x86_64/uvm_dpi "</command>

<command>endif </command>

For SV simulation, there is no need for shared objects.

In the regression tool, we also use the attribute "GROUP" to distinguish between C regression test and SV regression:

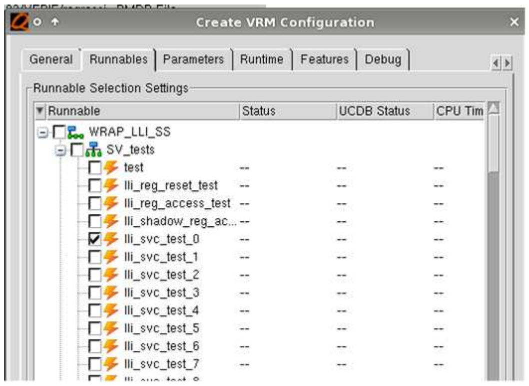


Runnable Selection Settings

Runnable	Status	UCDB Status	CPU Tim
WRAP_LLI_SS			
SV_tests			
C_tests			

Figure 8- Regression run cockpit snapshot

And the user can choose the relevant test to run as shown in the Figure 9.:



Runnable Selection Settings

Runnable	Status	UCDB Status	CPU Tim
WRAP_LLI_SS			
SV_tests			
test	--	--	--
lli_reg_reset_test	--	--	--
lli_reg_access_test	--	--	--
lli_shadow_reg_ac...	--	--	--
✓ lli_svc_test_0	--	--	--
lli_svc_test_1	--	--	--
lli_svc_test_2	--	--	--
lli_svc_test_3	--	--	--
lli_svc_test_4	--	--	--
lli_svc_test_5	--	--	--
lli_svc_test_6	--	--	--
lli_svc_test_7	--	--	--

Figure 9- Regression run cockpit snapshot

Summary and results

We achieved our goal of vertical reuse.

At the IP level, using the C-API package we were able to develop 10 test cases written in C that checked the main LLI functionality using the same UVM environment that we used to verify the design.

We have developed testcases only necessary for SoC integration and for the SW driver, as the exhaustive verification will be completed by SV scenarios.

We were also able to develop a software driver for the LLI IP block.

At the SoC level, the integration team was able to reuse our C tests with no modification and the software development team was able to use our LLI driver with only minimal changes.

The software reuse was made possible thanks to guidance (SW framework cookbook) agreed and deployed within STEricsson.

Modularity of this framework allows maximum reuse of existing code and portability between platforms with minimum coding and debugging effort. The software stack as described in figure 10, is made of four main elements:

- Common modules exportable and reusable through all platforms
- IPs modules, including IP, drivers, interfaces and tests
- Framework or platform-specific drivers and their interfaces
- Project-specific elements, including SoC test bench maps and global test runners

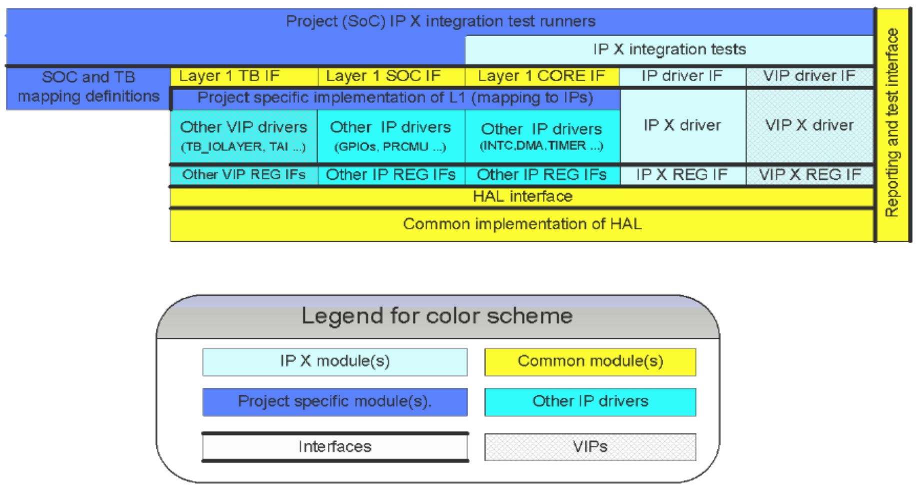

Project (SoC) IP X integration test runners
IP X integration tests
SOC and TB mapping definitions
Layer 1 TB IF
Layer 1 SOC IF
Layer 1 CORE IF
IP driver IF
VIP driver IF
Project specific implementation of L1 (mapping to IPs)
Other VIP drivers (TB_IOLAYER, TAI ...)
Other IP drivers (GPIOs, PRCMU ...)
Other IP drivers (INTC,DMA,TIMER ...)
IP X driver
VIP X driver
Other VIP REG IFs
Other IP REG IFs
Other IP REG IFs
IP X REG IF
VIP X REG IF
HAL interface
Common implementation of HAL
Reporting and test interface
Legend for color scheme
IP X module(s)
Common module(s)
Project specific module(s).
Other IP drivers
Interfaces
VIPs

Figure 10- Software modular architecture

In the light of our experience, we are now refining the technique to improve performance through the use of interrupts (instead of wait function) and reduce the size of the driver code.

References:

1. UVM 1.1a
2. ST-Ericsson Alliance Software framework – SW structure v1.5
3. http://verificationacademy.com/uvm-ovm/CBasedStimulus

Acknowledgements:

We would like to thank the following colleagues for their contributions to the development of the C-API technique: Jamal EL-HAITOUT and Youssef BAKRIM from ST-Ericsson; Mark PERYER, and Yassine LAHLOU from Mentor Graphics.