Using PSS-2.0 Hardware-Software Interface to validate 4G/5G Forward Error Correction Encoder/Decoder IP in Emulation & Silicon

Vinit Shenoy, Suresh Vasu, Nithin Venkatesh, Suhas Reddy, Joydeep Maitra
Intel Technology India Pvt Ltd, Bangalore
vinit.shenoy@intel.com, suresh.vasu@intel.com , nithin.venkatesh@intel.com, suhas.reddy@intel.com, joydeep.maitra@intel.com

Luis Campos
Intel Corporation, 2200 Mission College Blvd. Santa Clara, CA 95052
luis.campos@intel.com

***Abstract* - For effective validation, it is critical to have a test framework which can seamlessly scale from IP level to subsystem level to SoC level test cases. The PSS-2.0 standard introduces the Hardware Software Interface (HSI) as part of the core library, which provides a representation of registers. The HSI layer enables users to write a single implementation of device driver logic. In this paper, we discuss how we used the HSI layer to enable vertical reuse of test content across platforms.**

I. INTRODUCTION

One of the key driving factors in the development of the PSS[1] standard was the need to describe test content at a high abstraction level – enabling seamless execution across the different verification platforms.

There are multiple platforms that are used throughout verification: simulation, emulation, FPGA prototyping, Silicon to name a few, and each platform might use a different language. The scope of verification also varies across platforms, from IP-level to sub-system to SoC level scenarios. Based on current industry trends, the methodology of choice is UVM testbenches on IP/Sub-System level on simulation platforms and C-driver based tests on embedded core-based environments running on emulation or silicon. This creates a 'Portability Wall' as shown in Figure-1, as UVM sequences cannot be run on embedded cores, and C-driver code cannot natively drive UVM testbench.

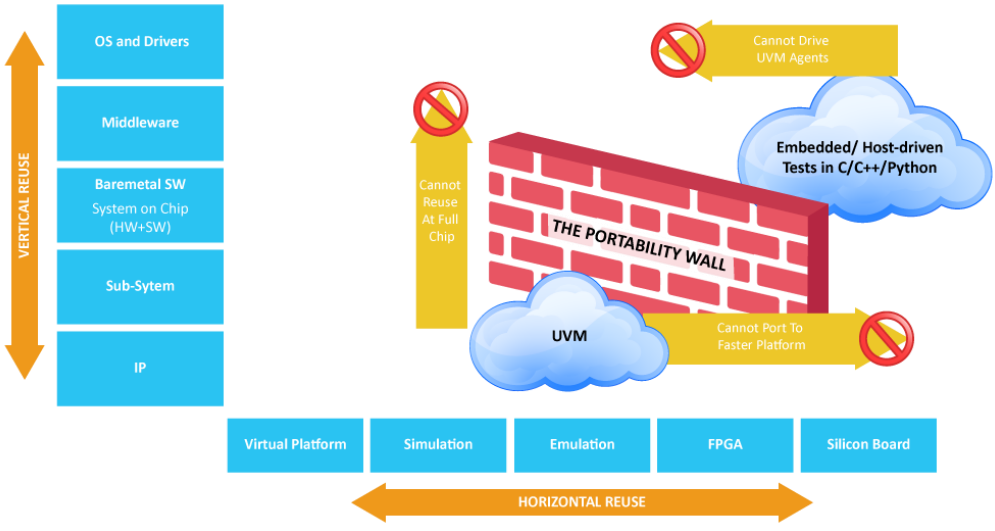

Figure 1 – Portability Challenge Across Platforms

The PSS-2.0 release introduces the Hardware-Software Interface as part of the core library. HSI abstracts the mechanism of lower-level register access. HSI provides much more than simple register abstraction, it provides a representation of memory, registers, interrupts, synchronization etc, enabling efficient way to implement driver logic. The HSI sequences can be mapped to different target implementations by the PSS tool. In this paper we provide an overview on how to adopt HSI into a PSS methodology.

II. PSS HSI DEVELOPMENT FLOW

Figure.2 captures the steps involved in generating a PSS test using HSI sequences.

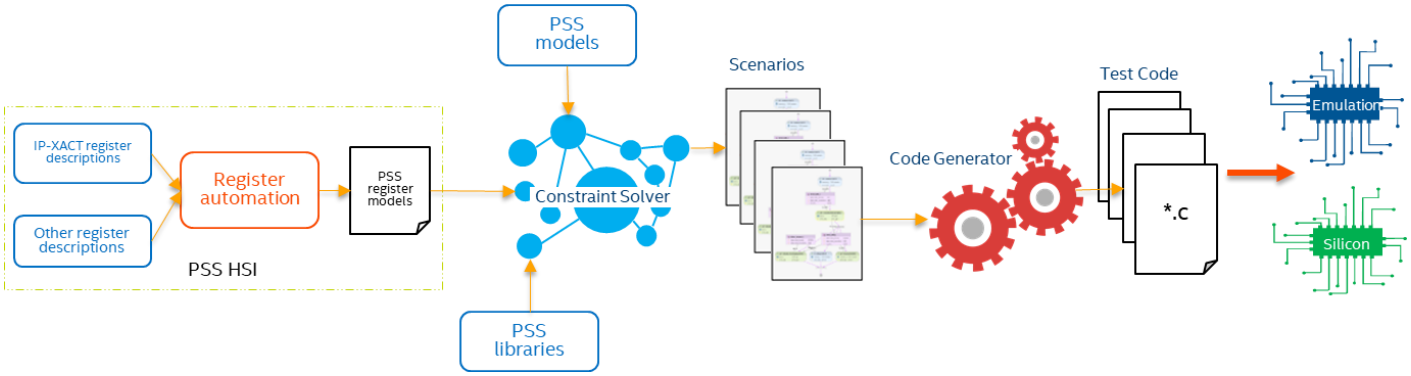

Figure 2: PSS HSI based development flow

A. PSS Register Definition

PSS tools may provide the register automation required to convert standard register description formats like IPXACT into PSS-HSI register models. At Intel, we use a custom XML format similar to IPXACT for register description, so a custom script was developed to convert this XML format into HSI register. Code snippet below shows a basic register description.

```
package ip_dma_MEM_regs_pkg {
 component ip_dma_MEM_c : reg_group_c {

  struct IP_DMA_CONFIG_0_REG_reg_s : packed_s<> {
     bit[30] RESERVED_0;
     bit[2] QUEUE_MODE;
  }

  reg_c<IP_DMA_CONFIG_0_REG_reg_s> IP_DMA_CONFIG_0_REG ;
     exec init {
          IP_DMA_CONFIG_0_REG.offset = 0x0;
     }
  }
}
```

Here the registers are defined under the component ip_dma_MEM_c. The reg_group_c component is the base type for specifying register groups. *reg_c* is a parameterized component base type for specifying the programmable registers of DUT.
Component *reg_c* is parameterized by:

a) A type *R* for the value (referred to as the *register-value type*) that can be read/written from/to the register, which can be:
   1) A packed structure type (that represents the register structure)
   2) A bit-vector type (*bit[N]*)

b) Kind of access allowed to the register, which by default is READWRITE

c) Width of the register (*SZ*) in number of bits, which by default equals the size of the register-valuetype R (rounded up to a multiple of 8)

The *reg_c* component provides *read()/read_val()/write()/write_val()* functions which may be called from the test-realization layer of a PSS model. Being declared as target functions, these need to be called in an *exec body* context.
Instantiating the register structs within the component prevents namespace collision.

B. Associating Register Groups with Address Regions

Before the read/write functions can be invoked on a register, the top-level register group (under which the register object has been instantiated) must be associated with an address region, using the *set_handle()* function in that register group. An example is shown in in the code snippet below.

```
extend component pss_top {
  import fec_pkg::*;

  /* Instantiate TIP Component */
  fec_c ip;

  exec init {
        // Allocate TIP DMA MMIO Space
        transparent_addr_region_s<> ip_dma_mmio_region;
        ip_dma_mmio_region.base_addr = 0x8F700000;
        sys_mem_map.add_nonallocatable_region(ip_dma_mmio_region);
        ip.dma_regs.set_handle(make_handle_from_region(ip_dma_mmio_region));
  }
} // pss_top
```

C. Writing an HSI Sequence

Once we have the register group instantiated in our PSS model, we can write our HSI register sequences. Code snippet below shows a simple HSI register write using the *write()* function.

```
action ip_reg_write {
  rand bit[2] qmode;

  // DMA register struct handle
  ip_dma_MEM_c::IP_DMA_CONFIG_0_REG_reg_s reg_inst;

  exec post_solve {
    reg_inst.QUEUE_MODE = qmode;
  }

  exec body {
    comp.dma_regs.IP_DMA_CONFIG_0_REG.write(reg_inst);
  }
 }
```

In the code snippet, reg_inst is a reference to the register structure we plan to write into, and it is updated with the randomized value in post_solve.

D. Target Code Implementation

When we talk about target code implementation of HSI sequences it is important to point out that the PSS standard does not impose specific rules on target code implementation. PSS tools are free to implement their own way of generating the target implementation code. The target code shown in this next section is specific to the PSS tool used at Intel.

1) HSI to UVM Target Implementation

The PSS tool generates C host code which drives transactions on the UVM agents using SV-DPI calls. The tool also generates required SV integration code to map these functions to the target. The generated C host code for our simple register write looks as follows.

```
ip_dma_MEM_c_IP_DMA_CONFIG_0_REG_reg_s_union tmp_var0;
{
 ip_dma_MEM_c_IP_DMA_CONFIG_0_REG_reg_s tmp___0 = {
  .RESERVED_0 = 0x0, .QUEUE_MODE = 0x3,
 };
 tmp_var0.inst = (tmp___0);
 write_scalar(0x8f700000,4,tmp_var0.val);
}
```

The PSS tool also generates an SV stub file which creates a task *write_scalar()* that is exported as an DPI-C call, and this task maps to the lower level UVM sequence. The code example below shows the SV integration.

```
export "DPI-C" task write_scalar;

task automatic write_scalar(input addr_data_type addr, int signed width, addr_data_type data);
  apb_master_agent master;
  string apb_master = {"*.",inst_id,".master"};
  master = find_apb_master_agent(apb_master);
  apb_write_scalar(master, addr, width, data);
endtask


task automatic apb_write_scalar( input apb_master_agent master, addr_data_type addr,
                                 int signed width, addr_data_type data);

  `uvm_info("APB_WRITE",
        $sformatf("APB WRITE ADDR: 0x%0x  WIDTH: %0d  DATA: 0x%0x\n", addr, width, data),
        UVM_LOW);
  case (width)
   4: begin
      apb_pkg::write_word_seq word_seq;
      word_seq = apb_pkg::write_word_seq::type_id::create("word_seq",master);
      word_seq.start_addr = addr[31:0];
      word_seq.write_data = data[31:0];
      word_seq.del = 0;
      word_seq.start(master.sequencer);
     end
   default:  `uvm_error("BAD_WIDTH",
                $sformatf("Invalid width in apb_write_scalar - %0d - expecting 4", width))

  endcase
  endtask
```

The host code is forked off from the run phase of the UVM test.

2) HSI to Embedded C Target Implementation

On core-based environments registers are accessed using direct memory pointer-based accesses as below.

```
{
   ip_dma_MEM_c_IP_DMA_CONFIG_0_REG_reg_s tmp___0 = {
      .RESERVED_0 = 0x0, .QUEUE_MODE = 0x3,
   };
      *(volatile unsigned int*)(0x8f700000) = *(unsigned int*)(&(tmp___0));
}
```

III. FORWARD ERROR CORRECTION DESIGN BLOCK

As communications service providers move from 4G to 5G networks, many are adopting virtualized radio access network (vRAN) architecture[2] for higher channel capacity and easier deployment of edge-based services and applications. vRAN solutions are ideally located to deliver low-latency services with the flexibility to increase or decrease capacity based on the volume of real-time traffic and demand on the network.

One of the most compute-intensive 4G and 5G workloads is RAN layer-1 (L1) forward error correction (FEC), which resolves data transmission errors over unreliable or noisy communication channels. FEC technology detects and corrects a limited number of errors in 4G or 5G data without the need for retransmission. FEC is a very common function that is not differentiated across vendor implementations. Since the FEC acceleration transaction does not contain cell state information, it can be easily virtualized, enabling pooling benefits and easy cell migration.

Figure-3. shows a block diagram of the FEC IP.

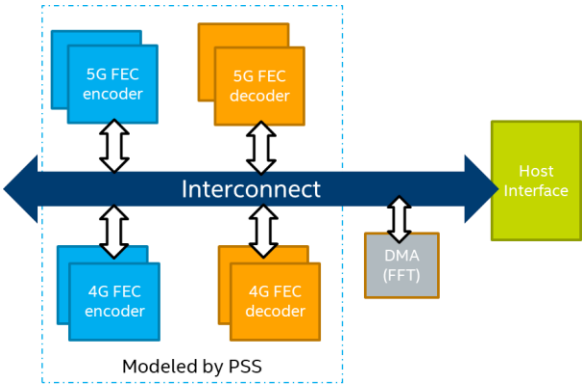

Figure 3: Indicative IP Integration Diagram

The IP is exposed as a PCIe device to host interface and provides multiple virtual functions (VFs) to the host interface. The IP also contains multiple instances of the 4G/5G encoder and decoder blocks. These multiple instances of design blocks basically imply multiple instances of register files, e.g. the PCIe configuration space for each of the virtual function, or the registers corresponding to each instance of 4G/5G encoder/decoder blocks. This is where PSS constraint randomization works very well with HSI sequences to provide an efficient way to write driver code. We shall discuss this in detail in the next section.

IV. OUR EXPERIENCE DEPLOYING PSS-HSI

In this section we elaborate on how we deployed PSS on our design, and the features of the language which were useful in creating test scenarios.

A. Register File Instantiation and Address Factorization

The first task is to instantiate the multiple instances of register files and set address map of each instance. The code snippet below shows the instantiation of few of the register blocks.

```
// Instantiate FEC-SS Register Files
pci_configreg_CFG_c pf_regs;
ip_fecdl4g_configreg_MEM_c dl_4g_regs[NUM_OF_4G_FEC_DL];
ip_fecdl5g_configreg_MEM_c dl_5g_regs[NUM_OF_5G_FEC_DL];
ip_fecul4g_configreg_MEM_c ul_4g_regs[NUM_OF_4G_FEC_UL];
ip_fecul5g_configreg_MEM_c ul_5g_regs[NUM_OF_5G_FEC_UL];
pci_configreg_CFG_c vf_cfg_regs[NUM_OF_VF];
```

To make the PSS code more re-useable IP configuration parameters like number of virtual functions can be factored out into const variables like NUM_OF_VF, so that the same model can be re-used in a different revision of the IP which has different configuration parameters. Similarly, the base addresses of the register blocks can be factored out into a CSV table.

table: ip_4g_fec_dl		
@package: ip_pkg	@size_const: NUM_OF_4G_FEC_DL	@struct: ip_core_info_s
#inst_id	#inst_name	#base_addr
0	"4G_FEC_DL_0"	0x2000_A000_0000
1	"4G_FEC_DL_1"	0x2000_B000_0000
2	"4G_FEC_DL_2"	0x2000_C000_0000

Capturing the address offsets in such a way helps with vertical re-use, the register offsets will be different on an IP-level platform compared to a SoC level environment. PSS helps in factoring out these configuration parameters.

B. Handling Non-Contiguous Register Arrays

The IP block contained multiple register arrays where the registers themselves were non-contiguous but had a fixed offset between each register. PSS allows adding a constraint on the register instantiation to handle these offsets.

```
reg_c<IP_INGRESS_AQ_reg_s> IP_INGRESS_AQ[INGRESS_QUEUE_SIZE];
  exec init {
    foreach(IP_INGRESS_AQ[i]) {
        IP_INGRESS_AQ[i].offset = i * 0x8;
    }
  }
```

Here every register index has an offset of 0x8, and the foreach block sets the correct offset for every index item.

C. Combining PSS Randomization with HSI Sequences

The randomized configuration generated by the PSS tool can be incorporated into the HSI sequences. Below code snippet shows one interesting register write.

```
comp.hi_vf_regs[vf_id].QMGR_INGRESS_AQ[queue_id].write_val(enqueue_data);
```

Here the eventual register write depends on two configuration parameters, vf_id and queue_id, and HSI provides us an efficient way to write such sequences.

D. Incorporating Interrupt Service Routines (ISRs) into HSI

HSI enables writing interrupt routines using the wait() and emit() APIs of ps_event. The code snippet below shows the way to write your interrupt routine in PSS.

```
extend component fec_ip_c {
// Define a PSS Event
ps_event irq;

// PSS ISR implemenation in a function
function void Isr () {
  bit[32] status;

  /* Write your ISR handler code here */

  // Unblock thread once ISR is complete
  irq.emit();
}

extend action execute_xfer {
  exec body {

    /* Write sequence to configure and initiate transfer */

    /* Wait for interrupt */
    comp.irq.wait();
}
}
```

The *wait()* function acts as an implicit yield statement on the calling thread, and the control flow switches to any other concurrent activity scheduled on the thread. The thread gets unblocked once the *emit()* function gets called from within the ISR.

E. HSI and Efficient Target Code Generation

While target code generation semantics is mainly left to the PSS tool provider, we felt it is worthwhile to discuss how PSS can generate efficient code. Consider the following code snippet.

```
// Write only enabled VF registers
foreach(comp.qmgr_regs.QMGR_FUNCTION_0_WEIGHT_RR_VF[i]) {
  if(i in vf_en_list) {
    comp.qmgr_regs.QMGR_FUNCTION_0_WEIGHT_RR_VF[i].write(reg_fn0_weight_rr_vf[i]);
  }
}
```

Here the generated code will only write to those virtual functions which are enabled in the scenario, which is captured in the variable vf_en_list. The generated code only contains very ‘directed’ code, which contains

programming sequences very specific to the randomized scenario. This helps in keeping the code footprint small and helps while debugging your test-case. Traditional test methodologies usually consist of driver code, and the test case is a separate entity which calls the driver APIs. HSI blends the driver and test-case into a single homogeneous test file.

Another HSI API we found very useful was *ps_event.sync()* which adds a synchronization point between concurrent threads. Adding a sync point between the IP configuration and data transfer phase helped us stress the design effectively.

IV. RESULTS

Adopting HSI into our PSS methodology enabled us to re-use test cases from core-less to core based environments, reducing the time taken to develop test content by 70% compared to previous methodologies. PSS also helped us generate a richer set of test scenarios, with more concurrency and stress added into the mix. It also enabled the Pre and Post-Si teams to collaborate on the test development, coming up with a common test plan. PSS also enabled quicker coverage closure compared to traditional methodologies.

Adopting PSS-HSI also helps users to get the most of each environment from an overall verification standpoint. The PSS model and HSI sequences can be developed on an IP level environment, which provides quicker turn-around time in the development phase. Shorter and basic functional test scenarios can be run on IP-level simulation environment, and more complex and stress scenarios were run on emulation and Post-Si.

IV. CONCLUSION

This paper provides a basic overview on how one can adopt HSI into their PSS methodology. We believe this can help accelerate the verification cycle while adding a lot of quality into the generated stimulus. We hope this paper encourages others to adopt PSS into their verification arsenal.

REFERENCES

[1] https://www.accellera.org/images/downloads/standards/Portable_Test_Stimulus_Standard_v20.pdf
[2] https://builders.intel.com/docs/networkbuilders/intel-vran-dedicated-accelerator-acc100-product-brief.pdf