2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE OCTOBER 26-27, 2021

A Novel Approach to Reuse Firmware for Verification of Controller based Sub-Systems using PSS

Vishnu Ramadas, Simranjit Singh, Ashwani Aggarwal
Samsung Semi-Conductors India R&D, Bangalore, India
v.ramadaska@samsung.com, simranjit.s@samsung.com, ashwani.a@samsung.com

Woojoo Space Kim , Seonil Brian Choi
Samsung Electronics, 1-1, Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do, Korea
space.kim@samsung.com, seonilb.choi@samsung.com

***Abstract*—Portable Stimulus Standard (PSS) is a methodology that captures the design verification intent and generates tests using it. The scenarios it captures can be reused across various design levels (Semiconductor Intellectual Property (IP), Block, System) as well as across multiple environments like simulation, emulation, virtual prototypes, and Silicon. A PSS model captures the test intent using an abstract representation of the design under test using high level attributes. However, the PSS model creation is a time intensive activity and increases in complexity with more sophisticated devices. The development of a PSS model for complex CPU based sub-systems involves significant time investment for both its creation and validation. In this paper, we demonstrate an approach to minimize the development time for such a complex block by making use of pre-existing firmware for the sub-system CPU written in C language to drive both emulation tests and RTL simulation tests**

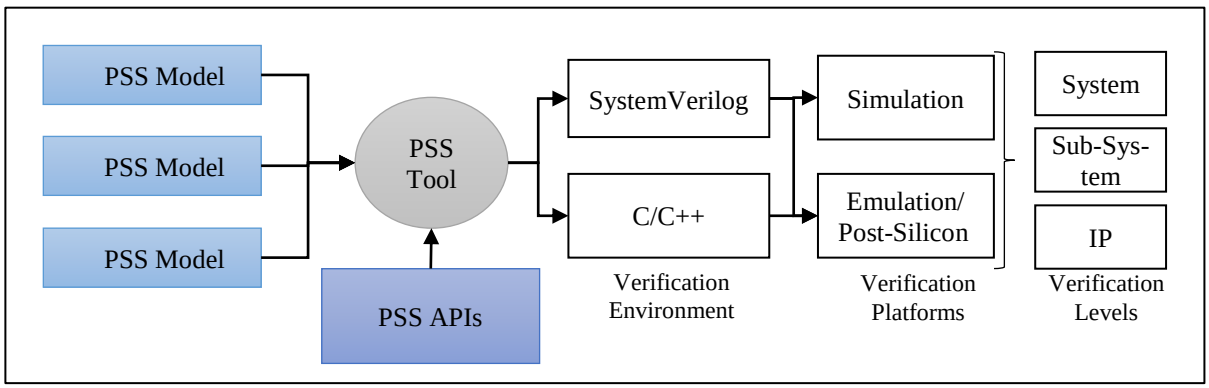
Keywords—Portable Stimulus Standard, Simulation, Emulation, Functional C Model, Functional Model, PSS APIs

I INTRODUCTION

Verification is an important part of a product's lifecycle. Over the years, considerable investments have been made to verification flow/methodology to make the process of verifying IPs and Systems on Chips (SoC) less time consuming and more complete. However, the chips getting more complex every year presents newer verification challenges at each stage of the design cycle. A significant effort goes into creating and executing tests with similar verification intent at various levels (at the IP level, SoC level, Silicon level) with little to no reuse. Additionally, porting these tests from one SoC to another is time consuming and prone to errors. Using Portable Stimulus Standard, a recently introduced methodology, this burden on verification engineers can be reduced significantly.

Portable Stimulus Standard is an Accellera standard that was released in 2018[1][2]. PSS is a methodology that helps with verification of any design across platforms. Figure 1 captures the PSS flow and its reuse across various verification environments, platforms, and levels [3].

Figure 1. PSS methodology flow



PSS helps describe test intents at an abstract, test-implementation agnostic level. This test intent, also called a scenario, can be used later to run the same or extended configuration on various platforms like simulation,

emulation, bare-metal, virtual prototypes, etc. The same scenarios can also be reused to run the same configuration across different levels in a system on chip. These levels include IP specific tests, sub system level tests, or the entire chip. Using various EDA tools, these scenarios will result in tests for all these targets. Once these targets are used to generate tests in either C or System Verilog (SV), they need to be compiled along with the right API code to be able to be used in a test environment. The API code is responsible for ensuring that the Device Under Test (DUT) is configured correctly.

Although reuse of tests across platforms is seamless and straightforward, adoption of PSS is not without its challenges. Proprietary IPs would not be part of a reusable standard library and will require both significant time and effort to develop. This increases with the complexity of the IP. The PSS model will need to contain a list of attributes that describe the configuration space of the IP and capture a set of constraints that model the dependence of these attributes on one another. Using the PSS models, a test scenario can be created by specifying the configuration of each model and its control and data flow. Using a PSS Electronic Design Automation (EDA) tool, the test code is generated for the given scenario. The test code can be in either C or SV language depending on which verification platform it needs to be run on. As the test-code is agnostic to verification platform, it needs a test realization layer which maps the test code to the target verification platform. This realization is done using a set of Application Programming Interfaces (API) which is responsible for configuring the device-under-test and the test-bench components. Therefore, these APIs act as driver code for the DUT. The complexity of this code depends on the complexity of the device under test. In the case of CPU based sub-system, there exists an additional layer of sophistication due to the presence of a dedicated CPU core. The APIs for these sub-systems might require multiple state machines where different programming sequences are executed based on the requirement. This paper presents the process of designing and deploying a PSS model, the challenges in developing these for CPU based sub-systems (a video processing sub-system in particular), how exploiting pre-existing firmware can reduce these difficulties, and how this will streamline testing across various levels.

II PSS Model Development

A PSS Model has three main components. It consists of a set of attributes, constraints, and APIs. The attributes are used to capture various configurations of the DUT that correspond to test intents for verification. PSS randomizes the values of these attributes, unless specified to pick a particular one, to generate scenarios that can cover various use-cases of the device under test. To ensure that randomization doesn't lead to illegal scenarios, constraints are modelled between the attributes in the PSS model. These constraints describe the relation between various attributes and, thus, restrict the set of PSS generated scenarios to be within the device's legal configuration space. The PSS model also consists of a set of APIs that are responsible for using the attribute values generated to configure the DUT and execute the test.

The attributes and constraints in a PSS model are dependent only on the device under test and are straightforward to develop. The APIs, on the other hand, need to be written keeping in mind the target platform. This is because emulators and RTL simulators differ fundamentally in how they access both memory and register space. While the address can be referenced directly in the case of the former, specialized read and write functions written in SV are needed for the latter. The effort here can, however, be reduced by the use of Direct Programming Interfaces (DPIs). With DPIs the same API can be used to access memory and registers in both cases as shown in Figure 2. Based on what compiler is used and which flag is passed, the correct implementation is used. These APIs may also perform verification checks by handling the interrupts raised during processing or verifying output images in multimedia devices. Developing these APIs is time intensive and requires detailed knowledge of the DUT. With increasing complexities of IP designs the APIs get more sophisticated. While certain IPs only require a set of register writes before triggering them to start, others might require a certain level of decision making. In case of complex IPs such as video processors, the APIs are required to keep a track of the number of frames processed, the results of previous operations, and also maintain state machines that are responsible for driving the hardware at various stages of the decoding pipeline. Developing PSS models for devices similar to the latter are challenging and require a lot of effort during development and validation. It could also be possible that the PSS model developer is not wholly familiar with the requirements such a code might have. The authors of this paper worked on

developing a PSS model for a CPU based sub-system used for such a video processing application. Developing the APIs needed was not a straightforward task. The challenges involved in the task were overcome by reusing test bench driver code that was already in use with a functional C model of the sub system, with the PSS model. This generic & reusable approach and the benefits it provides are discussed in greater detail in the following sections.

Figure 2.

```
// C source code
void dut_write_register (unsgined int a_address, unsigned int a_data)
{
#ifdef   EMULATION
   *(volatile unsigned int*)(a_address) = a_data;
#elif SIMULATION
   sv_write_register(a_address, a_data);
#endif
   . . .
}

 sv_write_register(unsigned int a_address, unsigned int a_data)
{
   ...
   invoke_write_reg(a_address, data);
   ...
}

//SV source code
export “DPI-C” task invoke_write_reg;
task automatic invoke_write_reg(input unsigned int a_address, input unsigned int a_data)
     ...
     write_reg(a_address, a_data);
     ...
endtask

task automatic write_reg(input unsigned int a_address, input unsigned int a_data)
   ...
   tb.axi.do_write(a_address, a_data);
   ...
endtask
```

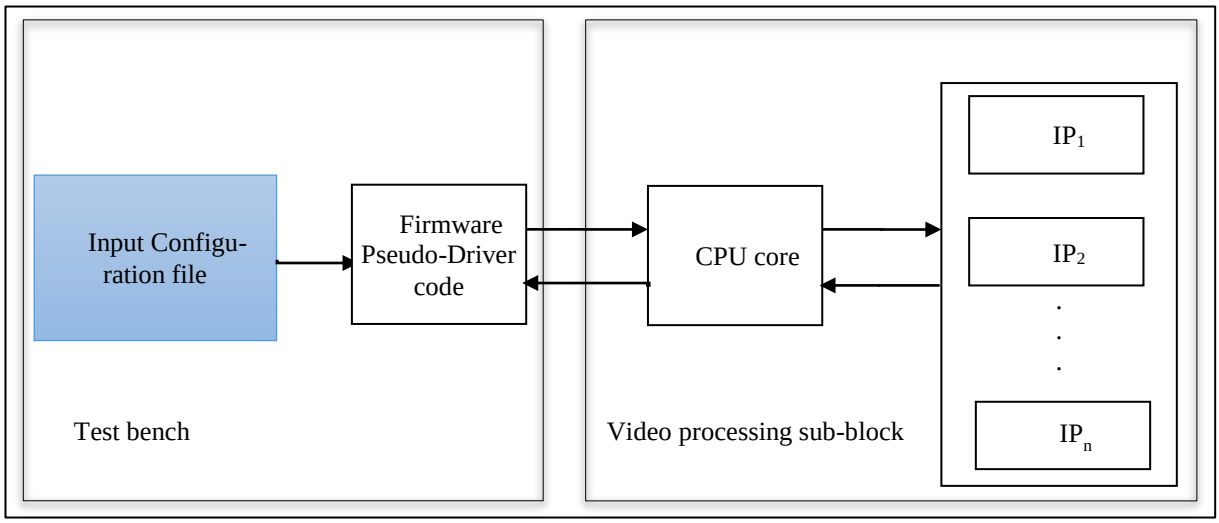
III PSS API DEVELOPMENT

The video processing sub system that the PSS model was developed for is shown in Figure 3. This sub system consists of a CPU core that interacts with a few IPs that handle various stages in the encoding/decoding pipeline. The test bench is responsible for driving the CPU core. The attributes in the PSS model represent the parameters needed for fine tuning the encoding/decoding of the input video.

III.A Usage of Firmware code in C Model

A functional C model of a device is a high level model that is designed to be functionally accurate. It provides a reference as to what the core operations/algorithmic implementations of the device are supposed to be. This model also serves as a way of generating reference output files for the RTL model/Silicon verification engineers. Such output files serve as a very good test of fidelity of the core algorithm in the RTL design to what is needed. Such functional models usually consist of two parts – the C language model of the hardware and an accompanying test bench or driver firmware.

Figure 3. CPU based sub-system functional model flow



The functional C model of the video processor shown in figure 3 accurately captures the algorithms used in the IPs that make up the sub system. This model uses a set of input configuration files to generate the encoded/decoded image that matches the output of the actual device. To run this C model, two dependencies are to be met. First, a CPU firmware binary is needed to interact with the C model. Second, a pseudo-driver firmware code is needed to read the configuration files and interact with the CPU. While the CPU firmware source code is the same for all platforms as it is not run outside the core, the pseudo-driver firmware is a feature that is solely used with the C model setup. This driver code parses through the configuration files and extracts information it needs to drive the CPU. On the other hand, the CPU firmware is available early-on due to its importance in interacting with hardware. In simulation or emulation platforms, the test bench used is different and is developed keeping in mind the specific test intent, while the CPU firmware is used as it is. Figure 4 is an example configuration file that is used by the C model. Figure 5 is a snippet of the parser that reads and understands this configuration file and figure 6 is a snippet of the code that interacts with the registers in the CPU core

Figure 4. Sample configuration file

```
Codec = 0
InputFormat = 1
SourceHeight = 240
SourceWidth = 320
. . .
```

Figure 5. Configuration file parser code

```
void set_encoder_parameters (volatile encoder_conf_s * a_conf , char* a_param_name, int value)
{
    if (!strcmp(a_param_name, “Codec”))      a_conf->codec = value;
   else if (!strcmp(a_param_name, “InputFormat”))    a_conf->input_format = value;
   else if (!strcmp(a_param_name, “SourceHeight”))  a_conf->source_height = value;
   else if (!strcmp(a_param_name, “SourceWidth”))  a_conf->source_width = value;”
  …….
}
```

Figure 6. Pseudo-driver code for register access functions

```
void dut_write_register (unsgined int a_address, unsigned int a_data)
{
   write_to_register_model (a_address,a_data);
}

void write_to_register_model(unsigned int a_address, unsigned int a_data)
{
   switch(a_address)
   {
       case 0x0:    video_processor.reg0 = a_data; break;
       case 0x4:    video_processor.reg1 = a_data; break;
       . . .
   }
}

unsigned int dut_read_register (unsgined int a_address)
{
   return read_from_register_model (a_address);
}

unsigned int read_from_register_model(unsigned int a_address)
{
   switch(a_address)
   {
       case 0x0:    return video_processor.reg0 ;
       case 0x4:    return video_processor.reg1;
       . . .
   }
}
```

III.B Reuse of Pseudo-Driver Firmware Code in PSS

The PSS test environment is very different to that of the C model. First, the difference in the execution platform itself presents challenges to the usage of the C model. In addition to that, the format of input files itself is different, While C models require a configuration text file with parameters and values as listed in figure 4, PSS generates the scenario as either C or SV files. The C generation scheme was preferred over SV when developing the PSS APIs and validating it as the pseudo-driver code reused from the functional model was written in C. The generated scenario snippet is captured in figure 7

Figure 7. PSS generated input configuration

```
void init()
{
   . . .
     video_processor_inst_1.codec = 0;
     video_processor_inst_1.input_format = 0;
     video_processor_inst_1.source_height = 0;
     video_processor_inst_1.source_width = 0;
   . . .
}
```

Given that the functional model test bench is not directly reusable due to differences in platforms, changes are needed before it can be used. The CPU core firmware can be run unmodified and only needs to be loaded at the right address during test run. The pseudo-driver code, on the other hand, can be used as the APIs needed to run PSS generated tests after three modifications to its code. First, since the input file format is different, file parsing used

with configuration files must be replaced with the values from the generated structures themselves. This is captured in figure 8

Figure 8. Parser to extract attribute values from PSS generated code

```
void set_encoder_parameters (volatile encoder_conf_s * a_conf , volatile video_processor_conf_s *
a_pss_conf)
{
    a_conf->codec = a_pss_conf->codec;
    a_conf->input_format = a_pss_conf->input_format;
    a_conf->source_height = a_pss_conf->source_height;
     a_conf->source_width = a_pss_conf->source_width;”
    . . .
}
```

Second, the register accesses need to be redefined for use in various other platforms. With the C model, these accesses consist of directly updating fields in a structure that describes an IP. In emulation, the register address in memory needs to be updated while SV simulation has its own utility. These are captured in figure 9

Figure 9. Register access functions used with PSS

```
void dut_write_register (unsigned int a_address, unsigned int a_data)
{
#ifdef   EMULATION
   *(volatile unsigned int*)(a_address) = a_data;
#elif SIMULATION
   sv_write_register(a_address, a_data);
#else
   write_to_register_model (a_address,a_data);
#endif
   . . .
}
unsigned int dut_read_register (unsigned int a_address)
{
#ifdef  EMULATION
   return *(volatile unsigned int*)(a_address);
#elif SIMULATION
   unsigned int data;
   sv_read_register(a_address, &data);
   return data
#else
   return read_from_register_model (a_address);
#endif
}
```

Finally, the pseudo-driver code makes extensive use of memory management routines. These are used for two purposes. One, to create objects that will be used to store attribute information and other metadata. Two, to allocate spaces for image buffers in DRAM. As bare metal application environments do not support much memory management, these will have to be replaced by other procedures. One such way is to directly assign buffers and objects a region in memory. The approach used by the authors is listed in figure 10. Any similar memory management routine can also be used. This is, however, platform dependent as certain RTL simulation tools allow C/C++ memory management routines.

Figure 10. Memory allocation routine used in emulation

```
void mem_alloc (unsigned int a_address, unsigned int a_size)
    unsigned int   return_address = region_start_address;
    region_start_address = region_start_address + a_size

    if (region_start_address >= dram_end_address)
       region_start_address = dram_base_address;

   return   return_address;
}
```

The changes listed in this section took roughly three days to make on a source code of size 5MB with the most time taken to replace the input parser. The PSS model was first validated on the C model binary before it was successfully ported on to C based emulation and SV based RTL simulation platforms.

IV VALIDATION OF THE MODEL

In order to validate the PSS model, a scenario was first run with the C model binary. To do this, the generated scenario and APIs were compiled using the GNU C Compiler (GCC) and the output of the run matched with a reference output. After verifying that the PSS model was interacting with the CPU firmware binary as expected, the scenario was ported on to an emulation environment. Here, the scenario was compiled using a version of the ARM C Compiler compatible with the emulation setup. The scenario ran successfully on the emulator with minimal fuss and the output here was also found to match with the expected result. Finally, the scenario was ported on to an SV based simulation environment. Here too, the scenario completed successfully. Thus, the PSS model was ported seamlessly across three different platforms with no errors during execution

V RESULTS

Table 1 captures the number of days it took to develop a PSS model for the Video Processor Sub System the authors worked on. The effort for the development and validation of the APIs is listed based on the early estimates before reuse of pseudo-driver firmware began. As can be seen from the table, reuse helped save roughly a month's worth of effort and allowed its re-investment in other tasks.

Table I. Comparison of effort taken to develop PSS model of Video Processor with and without code reuse

Development task for Video Processor PSS Model	**Effort (in days)**	
	Without pseudo-driver code reuse	***With pseudo-driver code reuse***
Developing the attribute model	2	2
Developing the constraint model	3	3
Developing the driver code	~15 (estimate)	3
Validating the PSS model	~25 (estimate)	5 (for all input/output data formats)

VI CONCLUSION

PSS Models make verification easier by allowing for reuse of scenarios across platforms and levels. The development of such a model, however, is time consuming. In order to reduce this effort, pre-existing pseudo-driver firmware that is used for verification/validation at some stage in the development (in this case, from the post-silicon stage) can be reused in the PSS Model. This saves a lot of time that can now be invested elsewhere. Moreover, redundancies that creep in because of the maintenance of different test benches for different platforms/levels can also be removed by using the same model. However, in most cases, this firmware is ready at a much later stage. To

ensure maximum reuse of code across various levels, adoption of a left-shift in pseudo-driver code/post silicon test bench development is needed where it starts along with the development of the PSS model. This model can then be ported across all verification levels. The reusability of this PSS model across various levels/platforms will result in significant reduction in the time spent on design verification.

ACKNOWLEDGMENTS

We would like to thank Mr. Balajee Sowrirajan, Corporate Vice-President, SSIR and Dr. Manoj Choudhary, Sr. Director, SSIR, for their continuous support in adopting PSS methodology. We would also like to thank Abhilash Karthik B. V. and Prashanth Narayan Jalli, SSIR for their help in understanding the video processor pseudo-driver code used. We would finally like to thank Mr. Gnaneshwar Tatuskar from the Cadence India team for helping us with the PSS methodology and the models development.

REFERENCES

1 https://www.accellera.org/downloads/standards/portable-stimulus

2 https://www.accellera.org/images/downloads/standards/pss/Portable_Test_Stimulus_Standard_v10a.pdf

3 Simranjit Singh, Ashwani Aggarwal, Harshita Prabha, Vishnu Ramadas, Seonil Brian Choi, and Woojoo Space Kim, “Adopting Accellera’s Portable Stimulus Standard: Early Development and Validation using Virtual Prototyping.” Paper presented at DVCON US 2021.