2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

SimpleLink™ MCU Platform: IP-XACT to UVM Register Model - Standardizing IP and SoC Register Verification

"UVM is a perfect start"

Jasminka Pasagic, Verification Engineer, Texas Instruments, Freising, Germany (j-pasagic@ti.com)

Frank Donner, Verification Engineer, Texas Instrumetns, Freising, Germany (f-donner@ti.com)

Keywords - UVM, Register-Model, Magillem,UVM Preferences, UVM_REG

Abstract

The register and memory model verification is stirring away from a manual development to an auto-generated process where IP-XACT specification is converted to UVM Register Model. Emerging design and verification technologies over the past period made it possible to standardize the process.

The data collected by Wilson Research Group Study from North American design engineers is revealing continuing trend going back at least ten years a specifications errors rate of ~45% due to incorrect/incomplete specifications. Moreover, 32% of those designs respins result in specifications changes.

In general, UVM based register model covers all verification elements like covergroups, coverpoints, coverbins and illegal bins where user can specify arbitrary hierarchical paths for blocks, register files, registers, register array and memories. IP-XACT is an XML format that defines and describes electronic components and their designs in this case register and memory map of design.

This paper explains current verification practices and issues of verifying register models. It gives details how to convert IP-XACT to UVM Register Model using Magillem UVM generator. It illustrates the verification productivity gains by systematizing the process of IP-XACT to UVM Register Model and risk reduction in introducing logic/functional flaws. It articulates the steps and issues that must be dealt with to have plug-and-play verification component in this case UVM Register Model.

It concludes to remain competitive, organizations need to link Register Verification Methodology to device development strategy, be sensitive to internal and external changes. The platforms with mix abstraction levels are necessary to keep the verification environment stimulus flexible, configurable and reproducible. This is collaborative work between different domain developers.

I. PROBLEM STATEMENT

Throughout the development cycle the IP-XACT specification change regularly. Respectively IP-XACT change is channeled to verification of register and memory model. Manual process of porting specifications changes to register model verification is time consuming and introduces logic/functional flaws. Additionally, every so often register access type is company specific and as such needs custom set of rules and regulations on how to handle them.

The platforms with mix abstraction levels are necessary to keep the verification environment stimulus flexible and configurable across multi-sites. The verification of register and memory model needs to ensure specification consistency, well-timed and free of logic/functional flaws development cycle.

II. VERIFICATION ENVIRONMENT

The process of solving the problem was done in two phases. The phase I of solution process involved accelerated integration and verification of standard interfaces (e.g.: APB/AHB) for register access. During this phase advanced UVM register model including predictor, adapter, map and basic sequences were developed. The phase II was looking into solution

on generating, standardizing the UVM preferences and developing scripts to pull-push IP-XACT to UVM Register Model for IP and SoC view.

A. Scope of Work

The work of this paper was embedded into a world-wide platform project with a high amount of reuse and standardization. All IP and SoC register layouts were centrally defined within IP-XACT and reused by different domains and sites. Examples for domain usage are IP verification, SoC verification, validation, design and design documentation as well as software development. The project was a multi-site project involving more than 5 countries and 4 time zones.

The scope of the work was to make use of the centralized IP-XACT data inputs and generate a common UVM register model for IP and SoC usage. The standardization of input data had been ensured by a common guideline document. Based on the common guidelines the description of the complete register layout had been entered into a Magillem database using the Magillem tool chain by the systems team. Majority of the data followed the IEEE-1685 standard. But there have been company specific data as well.

Foremost motivation was to establish a flow to push-pull IP-XACT to UVM register model and provide register model components to interact with rest of verification environment. The work included finding generator that converts the XML input data of design automatically to equivalent register model in SystemVerilog code and enable potential cycles update. The model contained both the standard IEEE-1685 data and vendor-specific data and provided automatic checks as well as coverage information.

A TI DMA device with APB standard bus interface and IP-XACT register description was selected to prototype the solution. The template testbench and test suite with advanced UVM register model including predictor, adapter, map and basic sequences where developed.

B. Flow of Standardazing IP-XACT to UVM Register Model

1) Standardizing Verification Components of Register Model

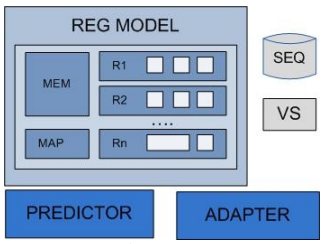
Collectively, new project verification development cycle challenge is simulatable TB framework in shortest amount of time that ensures flexibility to incremental development and work toward functional milestones. The work lead to standardizing our organizational UVM Register Model development, the look and feel of framework including the register model, sequences and sequencer, file and class naming, including the place holders for source files.

A TI DMA device with APB standard bus interface and IP-XACT register description was selected to prototype the solution. The template testbench and test suite of register model including predictor, adapter, map and basic sequences where developed. This work enabled us to set the guidelines and framework for developing the UVM register model across IPs and SoC. Further it enabled us to develop template for predictor, adapter, map and basic sequences of UVM register model. The register model components templates are inputs to custom developed script that post-processed it for IP or SoC naming and base address setting. Furthermore flow looked into the ways on how to handle custom field register polices or user-define register access types and their verification.

a) UVM Register Modeling

In a verification context, a register model is a set of classes that model the memory mapped behavior of registers and memories in the DUT in order to enable stimulus generation and functional checking (and optionally some aspects of functional coverage). The UVM provides a set of base classes that can be extended to implement comprehensive register modeling capabilities.

At first the simple versions of IP-XACT register model was described using Magillem tool. Its ‘generator’ was used to automatically generate the equivalent register model in SystemVerilog code. Further the register layer components to interact with rest of verification environment were developed as shown in Figure 1.



uvm_reg_adapter converts between register model read and write methods and the interface-specific transactions

uvm_reg_predictor updates the register model based on observed transactions published by a monitor

uvm_reg_block register model that instantiates and builds the register model

uvm_sequence #(uvm_sequence_item) base sequence and common sequences

uvm_sequencer

Figure 1: Verification Components of Register Model

After the structure is built, register access API methods like write() and read() called from sequence sent through sequencer, then driver are used to inject stimuli to the DUT. The monitor picks up activity and sends it back to predictor. The predictor sends data to the adapter where the bus data is converted to register model format for register model value to be updated through a predict() method call.

Through template testbench we standardized register model components look, basic content and framework for extending the classes and building IP/SoC exceptions. Moreover we decided on register sequence set that included the sequence for read reset values in random order, write value 0x5A, write value 0xA5, write all 1's, write all 0's and read. Additionally we standardize the name of class's, instance and file names as elaborated in below table and figures.

The benefit of this work was prepared register model components, available integration steps enabling head start for test cases development. Moreover all components are reproducible per IP or SoC and reusable for further integrations based on the specific needs. This work resulted in templates development for register model components. The templates were post-processed per IP or SoC IP-XACT input and registers model components per IP or SoC generated as summarized in Table 1.

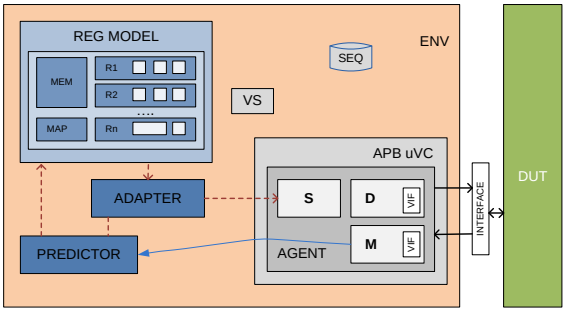


Figure 2: Block Diagram of Template Register Model [1]

Class Type	Class Name	Class File name
uvm_reg_block	<IP/SoC>_rm	<IP/SoC>_rm.sv
uvm_reg_block	<IP/SoC>_rm_model	<IP/SoC>_rm_model.sv
uvm_sequence	<name>_seq	<IP/SoC>_rm_seq_lib.sv
uvm_sequencer	< IP/SoC>_sequencer	<IP/SoC>_rm_sequencer.sv

Table 1: Summary of Register Model Components Standardization

```
class <IP/SoC>_env extends uvm_env;
 ........
 // declare instances of register model
 <IP/SoC>_rm_model                              rm;
 <IP/SoC>_rm_adapter                            rm_adapter;
 uvm_reg_predictor #(<BUS_TYPE>          )      rm_predictor;

 virtual function void build();
  .........
  // create instances of Register Model
  rm = <IP/SoC>_rm_model::type_id::create("rm",this);
  rm.build();
  rm.lock_model();
  uvm_config_db# (<IP/SoC>_rm_model)::set(null, "*", "rm", rm);
  rm.default_map.set_check_on_read(1);
  rm_adapter   = <IP/SoC>_rm_adapter                    ::type_id::create("rm_adapter",   this);
  rm_predictor = uvm_reg_predictor #(cba4_vbus_xfer #(32))::type_id::create("rm_predictor", this);
  .........
endfunction
.........
function void connect_phase(uvm_phase phase);
  .........
  // connect the register instances
  // <USER IP VIRTUAL SEQUENCER INSTANCE or LOCAL REGISTER SEQUENCER>
  // <USER PATH TO BUS MASTER AGENT INSTANCE>
  // <USER PATH TO BUS AGENT MONITOR PORT>
  ip_vir_seqr0.rm_seqr.rm   = rm;
  rm_predictor.adapter      = rm_adapter;
  rm_predictor.map          = rm.default_map;
  rm.default_map.set_sequencer(env.mst0_agent.sequencer, rm_adapter);
  env.mst0_agent.monitor.beat_ap.connect  (rm_predictor.bus_in);
  .........
 endfunction
endclass : <IP/SoC>_rm_model
```

Figure 4: Register Model Verification Instantiation Basics

```
...
class rm_por_seq extends reg_base_seq;
 `uvm_object_utils(rm_por_seq)
...
 task body;
   uvm_reg       regs[$];
   uvm_reg_data_t ref_data;

   super.body();
   rm.get_registers(regs);
   regs.shuffle();

   foreach(regs[i])  begin
      ref_data = regs[i].get_reset();
      regs[i].read(status, data, .parent(this));
   end
 endtask
endclass
```

Figure 3: Register Model Sequence Library Basics

```
class <IP/SoC>_rm_sequencer extends uvm_sequencer;
 `uvm_component_utils(<IP/SoC>_rm_sequencer)

 <IP/SoC>_rm_model rm;

 // new - constructor
 function new (string name, uvm_component parent);
  super.new(name, parent);
 endfunction : new
endclass
```

Figure 5: Register Model Sequencer Basics

Additionally, we looked into the bus protocols for accessing registers used across the platform. This work resulted in developing uvm_reg_adapter classes for each bus protocol to support function of converting register model transactions to the lower level bus transactions and vice versa. This enabled easy bus connectivity to register model. The verification work focus changed to creating predictor of bus type in build_phase() and in connect_phase() to enable bus sequencer for register map sequencer and predictor to bus monitor port in verification environment.

The Figure 6 demonstrates schematics of two kinds of bus protocols and their associated register models. In rear cases the developed adapters had to be IP specific and in these cases local adapters respecting the naming conventions were developed.

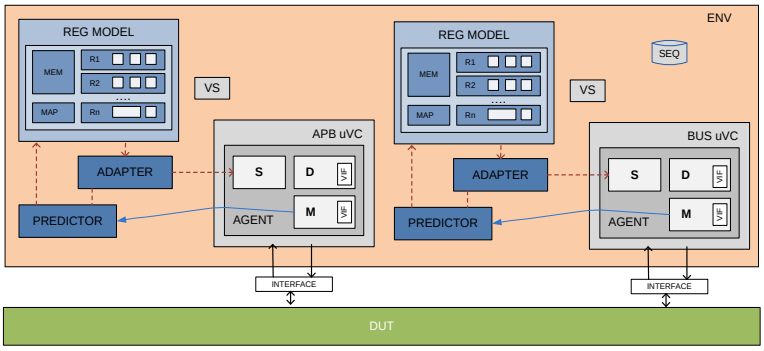


Figure 6: Block Diagram of Register Model Environment with Two Kinds of Bus Protocols and Register Models

Further, it was often the case where a register model can be accessed by one or more masters that have different addresses. Solution to this case was handled in register block in build_phase() by assigning the register address offsets according to the bus interface as shown in Figure 7 and Figure 8 .

The benefit of the approach comes from the high level of abstraction provided. As demonstrated the bus protocols for accessing registers can change but register model components for verification of the registers doesn't have to.

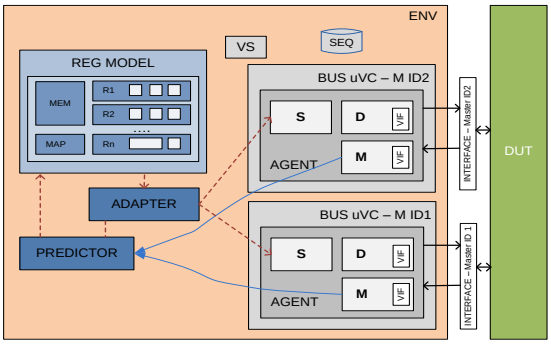


Figure 7: Block Diagram of Register Model with Two Masters

```
virtual function void build();
    super.build();
    uart_reg = uart_reg_block::type_id::create("uart_reg",, get_full_name());
    uart_reg.configure(this, "");
    uart_reg.build();
    uart_reg.lock_model();

    bus_M_ID1_map.add_submap(uart_reg.default_map, 32'h4000000);
    bus_M_ID1_map.set_check_on_read(1);

    bus_M_ID2_map.add_submap(uart_reg.cpu_map, 32'h30000000);
    bus_M_ID2_map.set_check_on_read(1);
endfunction : build
```

Figure 8: Template Code of reg_block for build_phase() for Register Model with Two Masters

b) UVM Custom Field Access Policies

After developing the register model components and standardizing naming conventions and basic content we had to look into how to handle the custom field access polices in IP-XACT description and also UVM domain. In UVM domain implementing custom policies can be done by defining the specific behavior for each register through extensions and customization of the register itself and by defining a new access policy.

To illustrate the solution explored, we use an example of register which holds two fields. One field is RW, and the other has custom access policy:

field1	standard register access type RW
field2	field sets itself if a write access of zero occurs

The solution includes class development of uvm_reg_field class and uvm_reg_cbs class associated to the custom access policy as shown in below figures. The field2 was described as type zeroToSet_reg_field and having the RW0S access policy. We added a callback to this field to actually implement the access policy. This is the only way UVM RAL allows us to define custom field access policies. We've had to split our code into three sections: the callback class, the register field class and the register class. This work opened question how to describe the custom field access policy in IP-XACT and it's post-

processing by UVM Generator to output SystemVerilog code. Furthermore the SystemVerilog output then had to be post-processed to describe custom class register field name and enable register callback.

```
class zeroToSet_cbs extends uvm_reg_cbs;
  `uvm_object_utils(zeroToSet_cbs)

  function new(string name = "zeroToSet_cbs");
    super.new(name);
  endfunction

  virtual function void post_predict(input uvm_reg_field  fld,
                                     input uvm_reg_data_t previous,
                                     inout uvm_reg_data_t value,
                                     input uvm_predict_e  kind,
                                     input uvm_path_e     path,
                                     input uvm_reg_map    map);
    if (kind == UVM_PREDICT_WRITE && fld.get_access() == "RWOS" && value == 0)
      value = 1;
  endfunction
endclass
```

Figure 9: Custom Field Access Policy - Call Backs Example

```
class zeroToSet_reg_field extends uvm_reg_field;

  `uvm_object_utils(uvm_reg_field_ext)

  local static bit m_rwos = define_access("RWOS");

  //Cunstructor
  function new(string name = "zeroToSet_reg_field");
    super.new(name);
  endfunction
endclass
```

Figure 10: Custom Field Access Policy - UVM Register Field defining RWOS policy Example

```
class <IP/SoC>_reg extends uvm_reg;

  `uvm_object_utils(zts)

  rand uvm_reg_field          field1;
  rand zeroToSet_reg_field    field2;

  //Cunstructor
  function new(string name = "<IP/SoC>_reg");
    super.new(name);
  endfunction

  virtual function void build();
    field1 = uvm_reg_field::type_id::create("field1");
    field2 = zeroToSet_reg_field::type_id::create("field2");

    field1.configure(this,  16, 16, "RW",   0, 0, 1, 1, 0);
    field2.configure(this,  16,  0, "RWOS", 0, 0, 1, 1, 0);

    // register callback
    zeroToSet_cb field2_cbs = new("rwos_cbs");
    uvm_reg_field_cb::add(field2, rwos_cbs);
    ...
  endfunction

endclass
```

Figure 11: UVM_REG Memory Model - Example of Register Description with Custom Field Access Policy

2) Standardizing IP-XACT Description

The systems team of TI uses Magillem tool to describe IP-XACT register models. The work involved consolidating the PSD requirements and IP-XACT custom field access polices between verification and system team.

Platform included numerous IP-XACT descriptions. This indicates the scale of consolidations and work put behind to achieve reproducible, reusable and automated register modelling.

a) Register Requirements for IP's

To achieve a high degree of standardization TI put requirements in place that each IP had to follow to achieve the goal of automation and providing a common set of definitions and look and feel on the register layout. Therefore common naming rules, instance names, fields, register and register usage had been predefined and provide for implementation. Standardization of functional register classes was provided to further streamline the usage. Examples are power, debug, reset, interrupt and clock control. Having all of those standards in place supports the integration of standard set of verification coverage when generating the UVM_REG classes. The reiteration of rules on IP's allowed the definitions of rule parameters that allow automated checked of them on reserved area, verification of reset values, and verification of CRC calculation or retention. Overall TI developed 22 parameter rules for this automated checking. Adding them into the UVM_REG implementation by a generator supports common checking methods, standard update and enhancements of checks in a centralized manner and increased the quality of the checker over time.

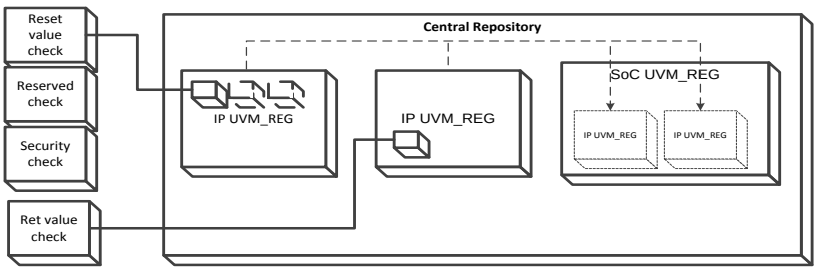


Figure 12: Overview of checking methods

b) Consolidating IP-XACT Custom Field Access Policies

The custom register access polices required implementation of database classes for uvm_reg_field and uvm_reg_cbs detailing the specifics of each custom register access policy. The script was implemented to post-process output of SystemVerilog code to replace the register field type of custom register access and all associated instances.

c) MRV Generator

As mentioned earlier platform involves multiple IPs and SoC and as such hundreds if not thousands of registers and tens of thousands of register fields. Manually trying to write SystemVerilog code to represent those registers and register fields would be a dreadful task. For this reason we looked into SystemVerilog generators whose job is to take the register specifications of a design and automatically 'generate' the equivalent register model in SystemVerilog code. The system team used Magillem tool to describe register specifications and for this reason we looked into Magillem tool generator - UVM Register Model SystemVerilog generator. This generator was used to auto-generate IP-XACT register description to UVM register model. The generator preferences were provided and were project specific. The UVM generation preferences were identified and captured in template file. This template was used to create TCL script for its execution.

3) Standardizing the Placeholder for IP-XACT and UVM Register Model(s) Sources

a) Repository for IP-XACT Files

The IP-XACT repository is Bitbucket. It is organized per projects and each IP or SoC is a single project and a working area for systems to deliver the IP-XACT description. The file naming convention was established and IP-XACT TI internal standard requirements had been followed. To make use of them from bitbucket into our design environment we developed a solution to integrate GIT repositories into DesignSync area and have direct access to a single source repository from GIT.

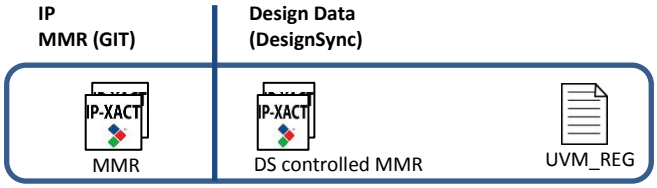


Figure 13: IP-XACT Repository Overview

The versioning of IP-XACT input data is controlled by reusing the configuration management system (DesignSync) versioning mechanism and never risk outdated input data. Whenever an update happens on systems side also a release need to go with it and these releases need to be selected by the uvm_reg generator. After generation also the UVM_REG classes for the whole SoC get released and tagged to continue development without distracting ongoing verification work.

b) Repository for UVM Register Model Files

The DesignSync data management was used to store the various components of register modelling including the scripts, IP-XACT, RAL source files and SystemVerilog classes for custom register access polices. The module imported bitbucket ports for all relevant IP and SoCs. This enabled released versions of IP-XACT from BitBucket each time the central repository is updated and ensured the data consistency from systems to verification.

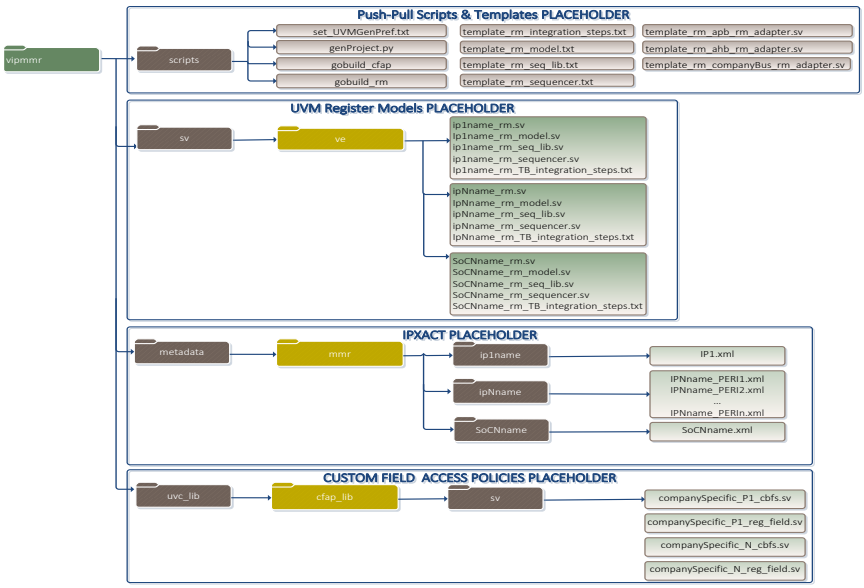


Figure 14: File Organization of IP-XACT to UVM_REG Register Model

4) Custom Developed Scripts to Pull-Push IP-XACT to UVM Register Model

The flow of IP-XACT to UVM Register model was enabled by developing several scripts and enabling several modes of auto-generation of UVM Register Model components as show in Table 2. Master script is "gobuild_rm" and has role of selecting the IP-XACT file, generating the TCL file with UVM generator preference settings, post-processing template files and producing IP or SoC SV files ready for TB further use or integration and post-processing of generated SystemVerilog code for custom register access polices.

Script Type	Script Name	Description
Text	set_UVMGenPref.txt	Consolidated UVM generator preferences settings
Python	genProject.py	Create Magillem project and import XML file(s)
Perl	gobuild_cfap	Post-processing of generated SV register map to ensure custom field access policies
Perl	gobuild_rm	Main script calling all other script to achieve pull-push IP-XACT to UVM Register Model

Table 3: Custom Developed Scripts to Pull-Push IP-XACT to UVM Register Model

```
================================================================================
WELCOME TO GOBUILD_RM: this script read's given XML file and produces SV register map.
================================================================================
================================================================================

NOTE: make sure you do not have active Magillem running in your workspace when executing this script!!!
The pitfall of having it active is that magillem_cl.sh commands in script will not be executed.

================================================================================
...Default defined UVM generator preferences used: <SimpleLink>/vipmmr/scripts/Set_UVMGenPref.txt
================================================================================
 USER COMMANDS

 NAME
    gobuild_rm - creates UVM_REG registar and memory map files
 SYNOPSIS
    gobuild_rm [OPTION] .. [FILE]...
 DESCRIPTION
    -ipxactfile          - give IPXACT file name <.xml>
    -uvmpref               - give preference file name and path </path/my_pref.txt>
    -all                 - SV register map will be created for all the IPXACT files in ~/vipmmr/metadata
subfolder
    -help                - this menu
 EXAMPLE
    gobuild_rm -ipxactfile <my.xml>
    gobuild_rm -ipxactfile <my.xml> -uvmpref </path/my_pref.txt>
    gobuild_rm -all
    gobuild_rm -help
```

Figure 15: gobuild_rm Developed Script to Pull-Push IP-XACT to UVM Register Model

III. APPLICATION

The technical contribution outlines the current complexity and problems of register model verification in multi-site project settings. It calls out for best practices and formalization of verification methodology to ease the use of data in automatic and straightforward way to ease the debug, optimize and enable multi-team collaboration. It illustrates process's involved in UVM register model that should be and can be standardized to achieve automation and gain time on verification implementation.

IV. RESULTS

The main result of solution described in this paper is achieved by consolidating PSD IP-XACT requirements, UVM generation preferences, templates of register model components and defining the source placeholder. Definition of source placeholder included IP-XACT, UVM register model and its components, SystemVerilog classes for custom access polices and custom developed scripts to enable pull-push IP-XACT to UVM Register Model for an IP and SoC view.

This work has enabled IP-XACT to UVM register model for any IP or SoC to be a work of script that executes in seconds and is ready to be integrated to TB. The TB integration steps are semi-manual verification work of few minutes before first successful simulatable test case. The speed factor from manual development to auto-generated development of register model is outstanding.

Example, assume IP has ten registers they are mix of standard and custom register access policies. The verification engineer work involved to arrive to first successful injection of register stimuli includes: writing UVM register description for 10 registers, implementing register model components, integrating the register model to interact with rest of verification environment, writing sequences and test cases, simulating and debugging. After the consolidation and automation of the flow the verification engineer work focus changed to sequences and test cases writing, adding advanced register checkers, simulating and debugging. The flow supports address aliasing using different addresses for the same physical register locations.

In exceptional cases incremental development or extension of existing UVM register model to address rare cases of register behavior still had to be done manually. The flow enabled reuse from IP to SoC or between SoCs by reusing input data of IP-XACT and flow to reproducible SystemVerilog output of register model and its components.

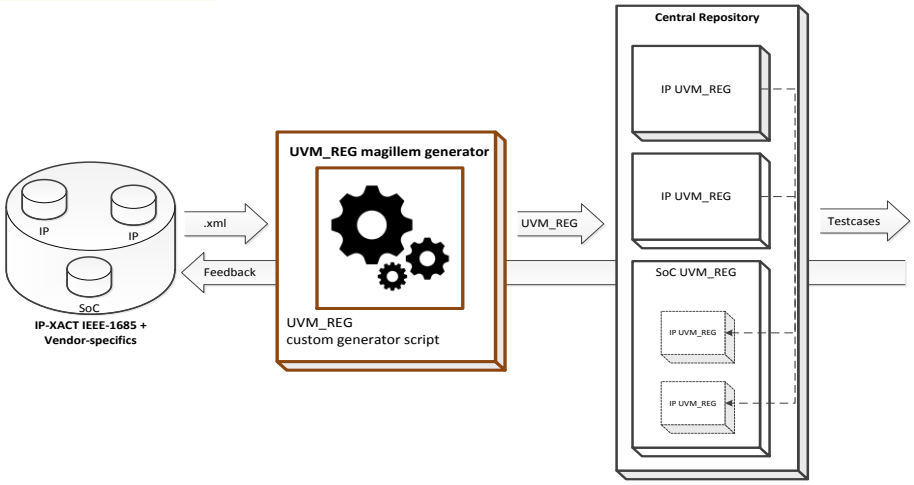


Figure 16: Consolidated Process's to Enable Pull-Push IP-XACT to UVM Register Model

V. CONCLUSIONS AND RELEVEANCE OF THE PAPER

In the past the UVM register map was written manually and updated manually per spec updates. If the IP or SoC contained custom register access polices the implemented classes were local to the IP verification development and as such introduced range of potential problems and flaws if reused or not updated per latest IP-XACT description. The process of standardizing the register verification eliminated many issues of manual and non-centralized development of UVM register map as well as the coordination and updates of the multi-site project.

It is concluded, the verification register and memory model needs to be auto generated to ensure specification consistency, well-timed and free of logic/functional flaws development cycle. To assure the auto generation of register model ensure various development phases are coordinated. In this case the architects, systems and verification. They all coordinated and implemented single source PSD requirements and ensured all involved development teams follow it. This work resulted in time reduction of implementation of UVM register map, faster cycle time to simulatable register framework, stable flexible and incremental development.

The significance of the paper is identification of reproducible verification elements of testbench in this case register map and identification of its links to input or output development cycles to achieve their optimization and best case automation. We reused standard industry practices to auto-generate register map from IP-XACT to SystemVerilog which was simple and straight forward but applied custom UVM preferences. Further we used UVM best practices to build verification SystemVerilog database of company specific register access policies description and centralized it for further reuse. Moreover we centralized the placeholder of the sources files for PSD requirements, IP-XACT, UVM register model and its components, SystemVerilog custom register access polices database, scripts to auto generate and achieved push-pull flow of IP-XACT to UVM register map for IP or SoC. Furthermore the custom developed scripts ensure automation and address UVM conformity. After standardizing mentioned components and flow from IP-XACT to UVM register map verification work focus changed from ensuring and locating latest IP-XACT description and manual developments and updates to sequence development, debugging, and optimizing the coverage and simulation performance. Further work remains as we plan to add checks for custom register access polices and custom field generator.

VI. ACKNOWLEDGMENT

The author would like to thank entire Texas Instruments Freising MCU systems, design, verification and management team for their support during the implementation of aforementioned solutions.

VII. REFERENCES

[1] M. Litterick and M. Harnisch, "Verilab," [Online]. Available: http://www.verilab.com/files/litterick_register_final_1.pdf.

[2] M. G. V. M. Team, Cookbook - Online Methodology DOcumentation from Mentor Graphics Verification Methodology Team, Menthor Graphics, 2012.

[3] H. Foster, "Wilson Research Group Functional Verification Study," 2012.

[4] K. A. Meade and S. Rosenberg, A Practical Guide to Adopting the Unversal Verification Methodology (UVM), second edition ed., San Jose, CA, 2013.

[5] "Accellera," UVM Library, [Online]. Available: http://accellera.org/downloads/standards/uvm.