Sept 25-26, 2014
Hotel Park Plaza
Bangalore, India

Reusable UVM_REG Backdoor Automation

Balasubramanian G, Senior Design Verification Methodology Engineer, PMC-Sierra India Pvt Ltd., Bangalore, India (balasubramanian.g@pmcs.com)

Allan Peeters, Technical leader, Design Verification Methodology, PMC-Sierra Inc., Burnaby, Canada (allan.peeters@pmcs.com)

Bob Blais, Senior Manager, Design Verification Methodology, PMC-Sierra Inc., Burnaby, Canada (bob.blais@pmcs.com)

Abstract— **IPXACT standard provides methodology independent meta-data and tool independent mechanism for accessing data inside a design. This paper brings out the limitation of IP-XACT standard to capture the backdoor paths of programmable registers present in a design. It explains an approach to automate UVM_REG backdoor access. Custom backdoor script use SV UVM_REG methods to setup HDLPATH corresponding to register instances and register bit fields. It also manifests the idea of reusing this automation from a block level to Device or Top Level verification.**

I. Introduction

System on Chip designs have become modular and its sub-design IP has programmable registers embedded in it. When these small design IPs get integrated in an SOC environment, the configurable registers also get carried over. Programming a register through a BUS protocol (like APB, AXI, etc.) consumes considerable simulation time. Hence, backdoor access becomes increasingly valuable. SystemVerilog UVM_REG utility has been used in this verification.

II. MOTIVATION

A. Definitions, Acronyms & Abbreviations

IP-XACT – Standard structure for packaging, integrating, and reusing IP within tool flows.

UVM – Universal Verification Methodology

SV – SystemVerilog

RTL – Register Transfer Level

XML – Extensible Markup Language

EDA – Electronic Design Automation

SoC – System on Chip

AXI – Advanced Extensible Interface

APB – AMBA Peripheral Bus

B. Limitations of IP-XACT and vendor tool automation

IP-XACT XML and iregGen utility (hereafter referred as vendor toolset) have been used to automate the UVM_REG register generation. The vendor toolset parses the IP-XACT XMLs and converts them into UVM_REG register files. It also parses vendorExtensions tag present in IP-XACT XML to capture array of registers. It sets up backdoor hdlpaths corresponding to the array of registers defined. Though vendorExtensions are outside IP-XACT standard, they capture array dimensions appropriately. The following limitations were encountered while capturing backdoor hdlpaths in IP-XACT XML and automating the backdoor hdlpaths.

- The hdlpaths of sub-fields of a register cannot be captured in IP-XACT XML and the vendor tool (iregGen) doesn't support such customizations. A register field in IP-XACT XML can be defined as follows:

```
<spirit:field>
     <spirit:name>RW_REG_16BITS1</spirit:name>
     <spirit:bitOffset>16</spirit:bitOffset>
     <spirit:bitWidth>16</spirit:bitWidth>
     <spirit:access>read-write</spirit:access>
     <spirit:vendorExtensions>
        <vendorExtensions:hdl_path>abc</vendorExtensions:hdl_path>
     </spirit:vendorExtensions>
</spirit:field>
```

As per the 16-bit register field defined above, hdl_paths of all 16-bits needs to be captured in a 16-bit vector ('abc') in RTL. In typical cases, the 16-bit vector might be split into individual bits inside RTL. Figure 1 illustrates such bit fields. The individual bit based paths of a register field cannot be captured in IP-XACT XML format. The vendor toolset does not support such customization.

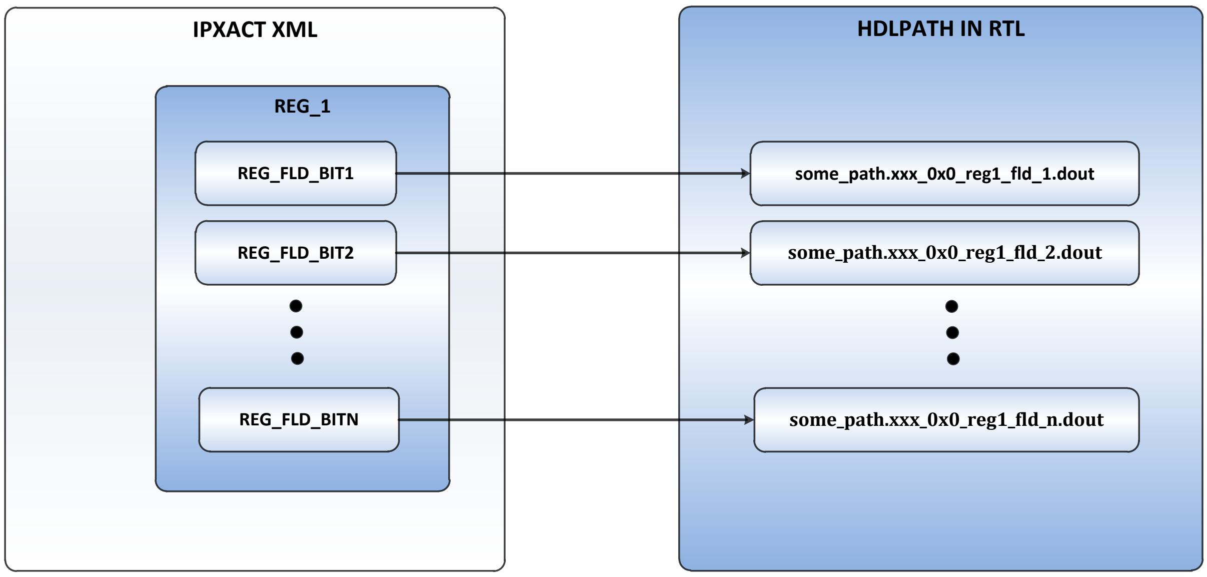

IPXACT XML
REG_1
REG_FLD_BIT1
REG_FLD_BIT2
REG_FLD_BITN
HDLPATH IN RTL
some_path.xxx_0x0_reg1_fld_1.dout
some_path.xxx_0x0_reg1_fld_2.dout
some_path.xxx_0x0_reg1_fld_n.dout

Figure 1 Typical HDLPATHs of register fields

- The hdlpaths of single or multidimensional array of registers cannot be captured effectively in IP-XACT XML and the vendor tool (iregGen) does not have automation support for such customizations. Consider an array of registers defined in IP-XACT XML as vendorExtensions:

```
<spirit:vendorExtensions>
  <vendorExtensions:array
xmlns:vendorExtensions="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5">
  <vendorExtensions:x_from>0</vendorExtensions:x_from>
  <vendorExtensions:x_to>7</vendorExtensions:x_to>
  <vendorExtensions:offset_calc>0x120+(x*4)</vendorExtensions:offset_calc>
  <vendorExtensions:hdl_pattern>("xyz[%0d]", x)</vendorExtensions:hdl_pattern>
  </vendorExtensions:array>
</spirit:vendorExtensions>
```

The array of registers defined above expects us to define each instance of a register in some design hierarchy. As per above definition, reg[0] should be in a hierarchy 'xyz[0].' followed by individual field hdlpaths. In typical cases, The RTL implementation of a register or an array of registers need not be modular. ie., reg[0] does not need to have hdlpath path in design hierarchy (it may just be direct fields). reg[0] need not have any relation with array index defined as per hdl_pattern tag. Refer figure 2 to understand the limitation.

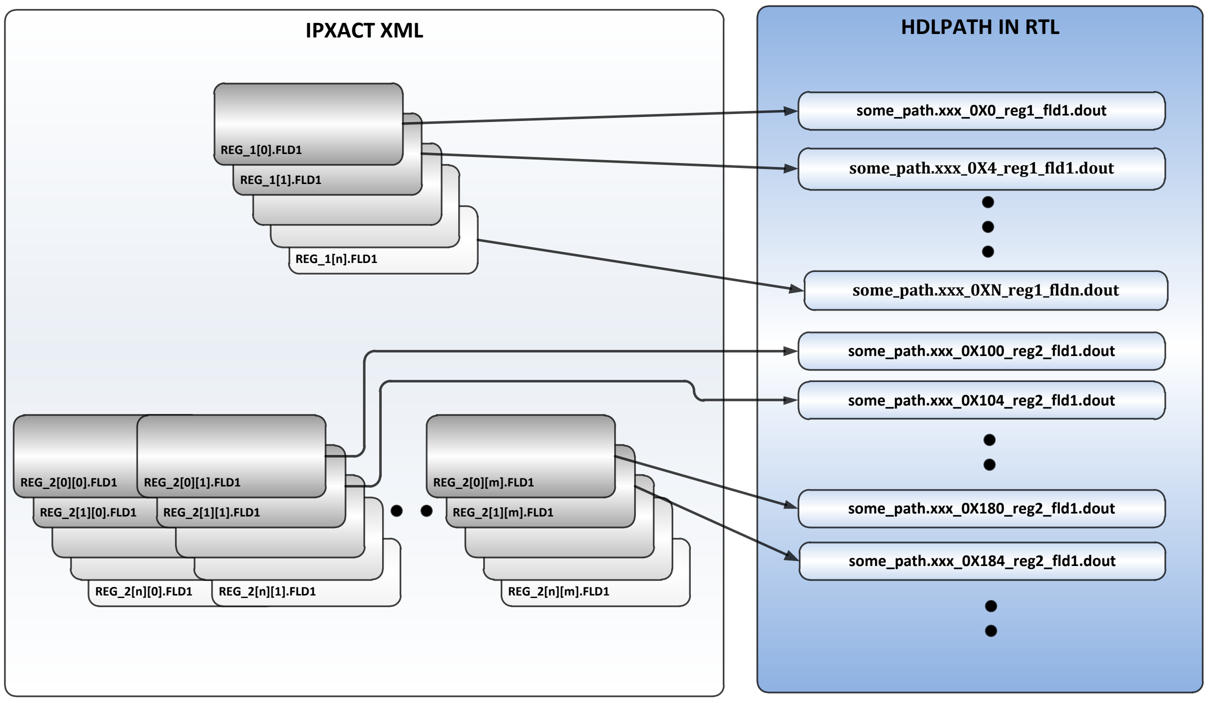

IPXACT XML
HDLPATH IN RTL
REG_1[0].FLD1
REG_1[1].FLD1
REG_1[n].FLD1
some_path.xxx_0X0_reg1_fld1.dout
some_path.xxx_0X4_reg1_fld1.dout
some_path.xxx_0XN_reg1_fldn.dout
some_path.xxx_0X100_reg2_fld1.dout
some_path.xxx_0X104_reg2_fld1.dout
some_path.xxx_0X180_reg2_fld1.dout
some_path.xxx_0X184_reg2_fld1.dout
REG_2[0][0].FLD1
REG_2[1][0].FLD1
REG_2[n][0].FLD1
REG_2[0][1].FLD1
REG_2[1][1].FLD1
REG_2[n][1].FLD1
REG_2[0][m].FLD1
REG_2[1][m].FLD1
REG_2[n][m].FLD1

Figure 2 HDLPATHs of an array of registers

- A combination of sub-fields and arrays cannot be captured in IP-XACT XML and the vendor toolset does not support such customizations.
- Building IP-XACT xmls with composite register fields and setting hdl_paths to them poses further challenges. ie., hdlpaths of some register field exists in cluster and hdlpaths of some register fields exists as individual bits.
- Handling or splitting hdlpath instantation hierarchy inside IP-XACT xml poses a challenge when there are insufficient hdlpath hierarchy for some register implementation in design.

C. Custom Backdoor Automation Solution

Due to the above said limitations, we chose to capture register hdl_paths in a separate XML. Figure 3 represents the architecture of the backdoor automation solution.

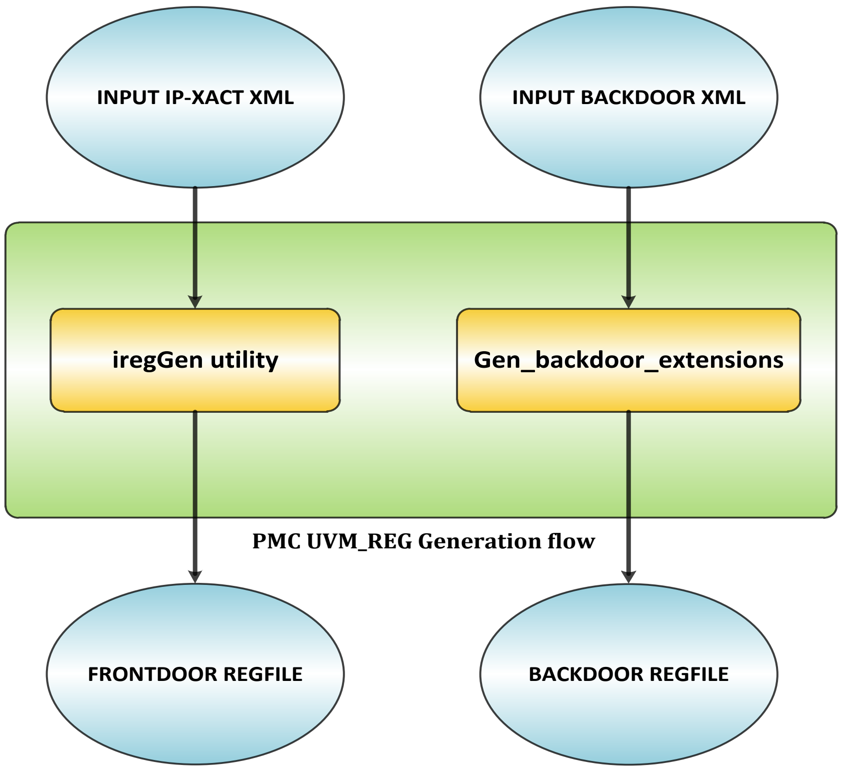


Figure 3 Custom backdoor automation

The programmable registers of a particular design need to be captured in IP-XACT XML. Cadence iregGen toolset was used to generate a UVM_REG front door register file based on the IP-XACT XML.

An INPUT BACKDOOR file was created with parameters like register_name, instance_number, register_field name, hdl_path corresponding to each and every register field present in the design. (The backdoor file was provided by design team)

Backdoor automation involves the creation of a derived class that inherits the top-level class of the front door register file. Figure 4 represents the class hierarchy. A script parses the backdoor XML and creates a build method in the derived class to set up backdoor paths of each individual register bit (based on instance number) using add_hdl_path_slice() method. The add_hdl_path_slice() method helps us to configure single bits or multiple bits in a register to some HDLPATH in design. i.e, The add_hdl_path_slice() method takes in HDLPATH string, bit position, bits size (number of bits connected in this HDLPATH) and an abstraction level (which is RTL in this case). Using add_hdl_path_slice() method, HDLPATH of various bits defined in a register can point to different HDLPATHs hierarchy in RTL.

Example #1, single bit in a register can be programmed to hdlpath as:

```
rsvd.SAMPLE_REG1.add_hdl_path_slice("xcbi_nregs_m0.singlebitr1_0x0.dout", 29, 1, “RTL”);
```

Example #2, multiple bits in a register can be programmed to vector hdlpath as:

```
rsvd.SAMPLE_REG1.add_hdl_path_slice("xcbi_nregs_m0.multibitrw0_0x0", 25, 4, “RTL”);
```

Example #3, individual bits of a register in an array of registers can also be configured to different HDLPATH hierarchy in RTL:

```
rsvd.SAMPLE_REG2[0].add_hdl_path_slice(“xcbi_nregs_m0.singlebitr1_0x100.dout”,29,1,“RTL”);
rsvd.SAMPLE_REG2[1].add_hdl_path_slice(“xcbi_nregs_m0.singlebitr1_0x104.dout”,29,1,“RTL”);
```

Example #4, multiple bits of a register in an array of registers can also be configured to different vector HDLPATH hierarchy in RTL:

```
rsvd.SAMPLE_REG2[0].add_hdl_path_slice(“xcbi_nregs_m0. multibitrw0_0x100”, 25, 4, “RTL”);
rsvd.SAMPLE_REG2[1].add_hdl_path_slice(“xcbi_nregs_m0. multibitrw0_0x104”, 25, 4, “RTL”);
```

Examples #1, #2, #3 and #4 show that HDLPATH could be configured to registers with composite register fields.

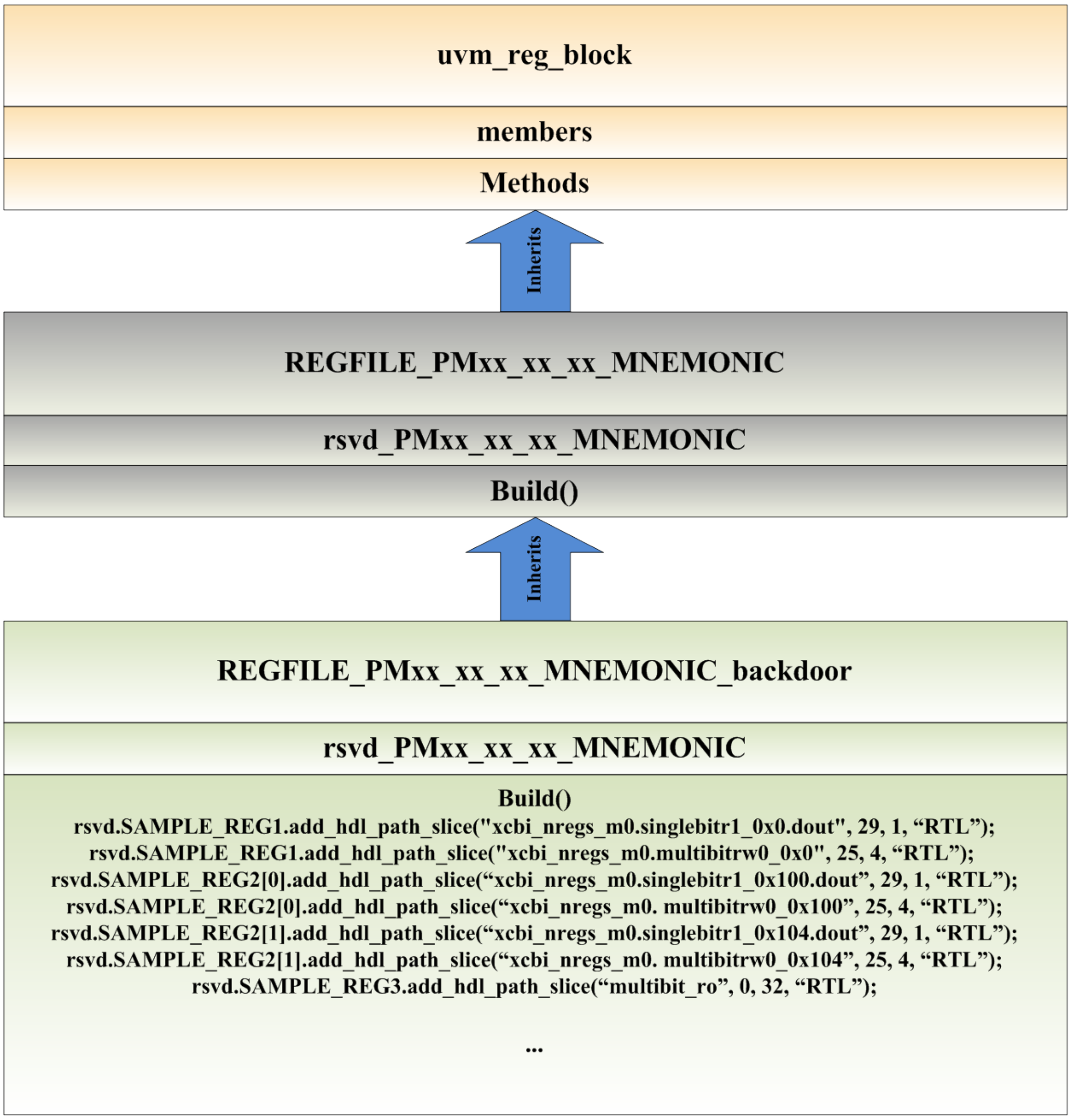
Example #5, insufficient HDLPATH hierarchy for some registers can be captured using add_hdl_path_slice command as:

```
rsvd.SAMPLE_REG3.add_hdl_path_slice(“multibit_ro”, 0, 32, “RTL”);
```

A script was written to parse the BACKDOOR XML and deduce the HDLPATHs corresponding to each register bit and each instance of a register. This script generated a target file containing add_hdl_path_slice() method calls configuring each and every instance of the register bits. The generated target file was used in conjunction with add_hdl_path() configuration to complete backdoor automation stitching.

```
rsvd.add_hdl_path(“top_testbench.dut.block_inst1", “RTL”);
```

add_hdl_path() command has been used to attach hierarchy to all registers defined in a register block. This enables reuse of backdoor information for register blocks from block level to device level verification.



```
rsvd.SAMPLE_REG1.add_hdl_path_slice("xcbi_nregs_m0.singlebitr1_0x0.dout", 29, 1, “RTL”);
rsvd.SAMPLE_REG1.add_hdl_path_slice("xcbi_nregs_m0.multibitrw0_0x0", 25, 4, “RTL”);
rsvd.SAMPLE_REG2[0].add_hdl_path_slice(“xcbi_nregs_m0.singlebitr1_0x100.dout”, 29, 1, “RTL”);
rsvd.SAMPLE_REG2[0].add_hdl_path_slice(“xcbi_nregs_m0. multibitrw0_0x100”, 25, 4, “RTL”);
rsvd.SAMPLE_REG2[1].add_hdl_path_slice(“xcbi_nregs_m0.singlebitr1_0x104.dout”, 29, 1, “RTL”);
rsvd.SAMPLE_REG2[1].add_hdl_path_slice(“xcbi_nregs_m0. multibitrw0_0x104”, 25, 4, “RTL”);
rsvd.SAMPLE_REG3.add_hdl_path_slice(“multibit_ro”, 0, 32, “RTL”);

...
```

Figure 4 Class hierarchy diagram

III. RESULTS

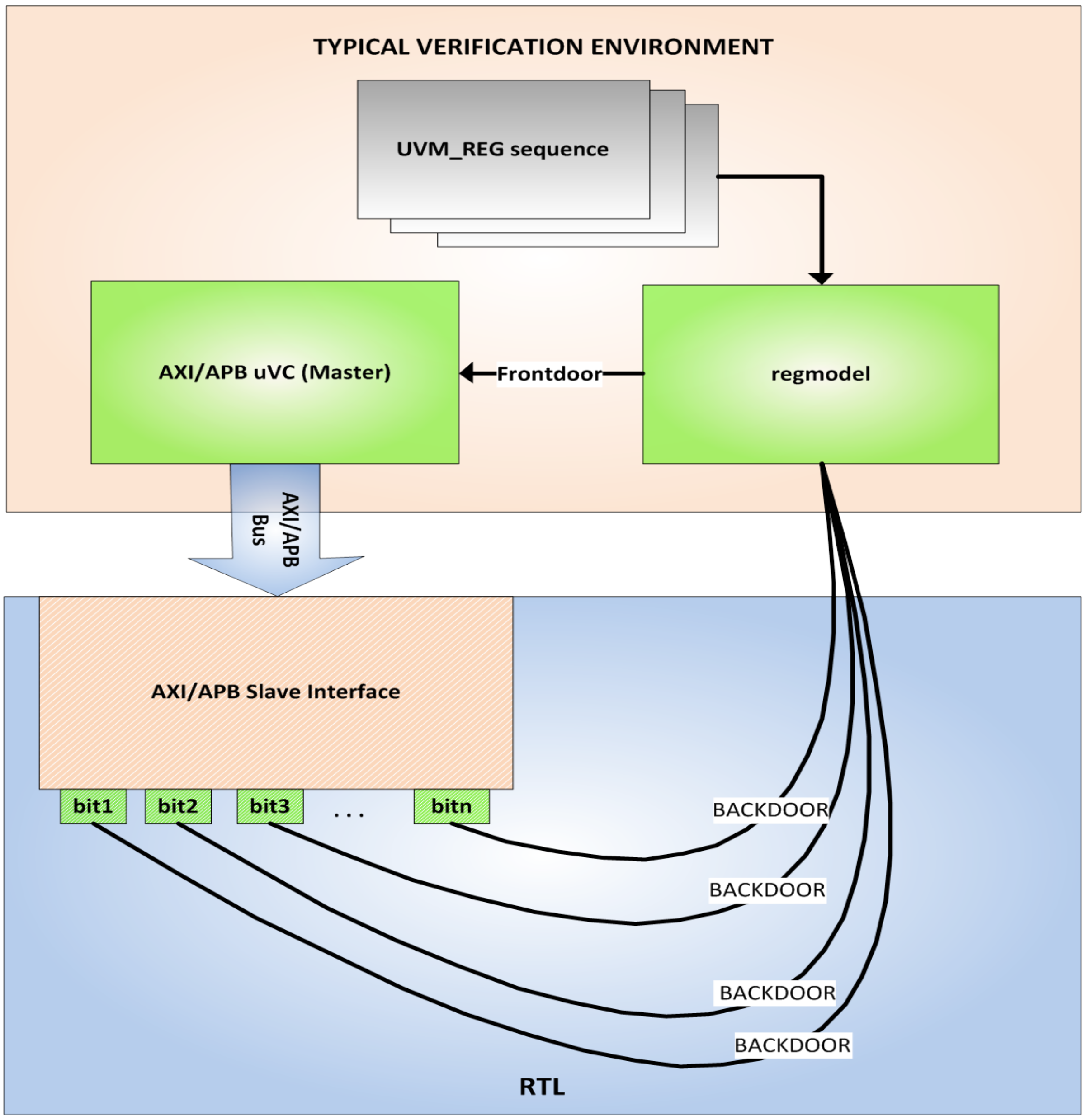


Figure 5 Typical Verification environment

Figure 5 explains the simple testbench used to test backdoor register automation. DUT was an AXI slave that contains registers inside. Each register present in design would need to be programmed through AXI BUS protocol (in front door access). Design team provided us an IP-XACT XML file containing register definitions and a backdoor XML file containing backdoor paths of each register present in design. We generated a front door and backdoor UVM_REG extensions file based on the methodology explained in this paper. The backdoor register file was instantiated in the verification environment as ‘regmodel’. The backdoor register file was connected to an adapter that converts UVM_REG data item to an AXI transaction item which gets driven to DUT using an AXI uVC. The front door register access takes considerable simulation time as a register write or read completes after AXI VALID/READY handshaking. Every register defined in UVM_REG_BLOCK gets connected to the corresponding HDLPATH in RTL. A UVM_REG backdoor write would deposit the values directly on HDLPATH connected to that register instance. Similarly, a UVM_REG backdoor read would read the net value corresponding to the HDLPATH connected to it. The backdoor access happens at zero simulation time as it does not go through VALID/READY handshaking. In order to test the sanity of HDLPATHs set to registers by automation, we ran a test sequence that performs front door write followed by back door read and vice versa.

If the hdlpaths are incorrect, read from backdoor registers would report UVM_ERROR as the data won't match the written value.

UVM_REG backdoor write followed by a backdoor read:

```
`uvm_info("UVMREG_SEQ", $psprintf("BACKDOOR WRITE to
SAMPLE_REG_1_PM40_40_322_BETA_NREGS_V3 : %8h", 32'hffffffff), UVM_LOW)
model.rsvd.SAMPLE_REG_1[0].write(status, 32'hffffffff, .parent(this),
.path(UVM_BACKDOOR));
model.rsvd.SAMPLE_REG_1[0].read(status, rdata, .parent(this), .path(UVM_BACKDOOR));
`uvm_info("UVMREG_SEQ", $psprintf("BACKDOOR READ to
SAMPLE_REG_1_PM40_40_322_BETA_NREGS_V3 : %8h", rdata), UVM_LOW)
```

Result in simulation:

```
UVM_INFO ../funcsim/tb/seq_lib//pm_18_xcbi_tb_virtual_seq_lib.sv(177) @ 212450000:
uvm_test_top.xcbi_sve.xcbi_vs@@seq1 [UVMREG_SEQ] BACKDOOR WRITE to
SAMPLE_REG_1_PM40_40_322_BETA_NREGS_V3 : ffffffff
UVM_INFO ../funcsim/tb/seq_lib//pm_18_xcbi_tb_virtual_seq_lib.sv(186) @ 232450000:
uvm_test_top.xcbi_sve.xcbi_vs@@seq1 [UVMREG_SEQ] BACKDOOR READ to
SAMPLE_REG_1_PM40_40_322_BETA_NREGS_V3 : 94800000
```

Using the above test bench, we performed a comparison between front door and backdoor access varying the number of UVM_REG register access. ie., Write followed by read to a register was tested in front door and backdoor mode varying the number of register access. Number of register access samples were 100, 1000, 10000, 100000. We printed the UNIX time before the start of the first register access and after the end of last register access. The difference between the UNIX times was considered as the CPU time taken. We tried to compare the CPU time taken for the above said test when register access was executed in front door and backdoor modes.

Refer the table 1 and figure 6 given below:

Number of register access	**RESULTS**		
	CPU time (sec) taken for frontdoor access	***CPU time taken (sec) for backdoor access***	***Acceleration***
100	5	1	5x
1000	50	3	17x
10000	418	19	22x
100000	4567	208	22x

Table 1 Results

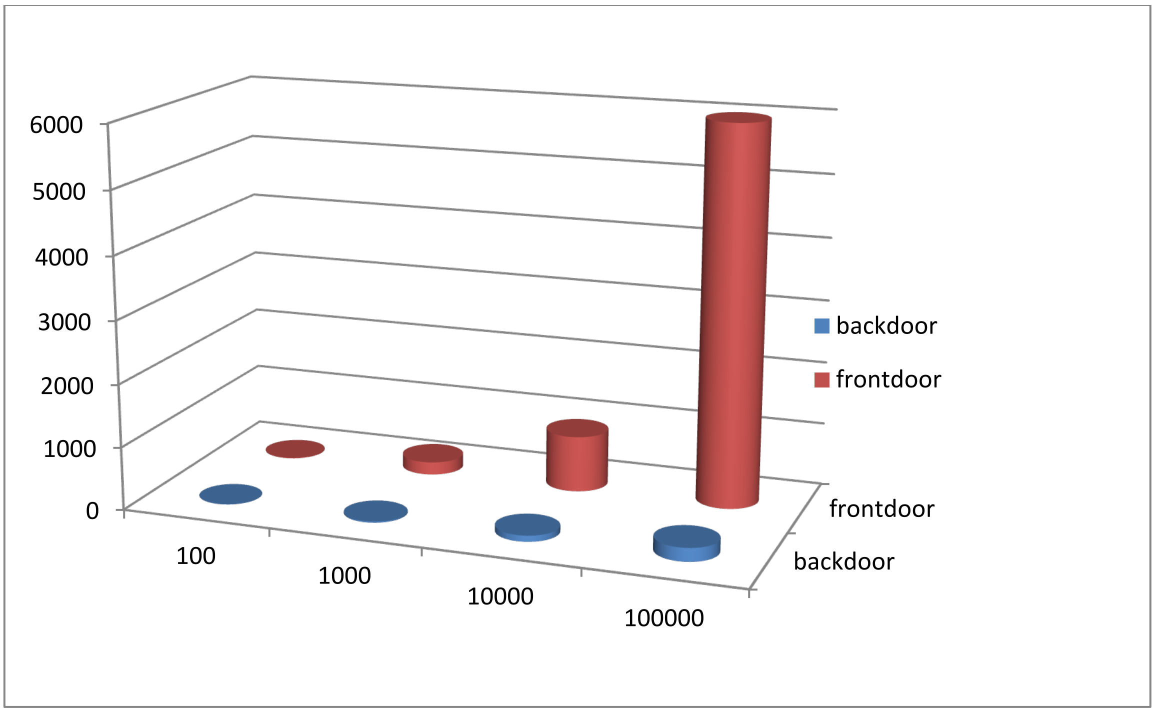


Figure 6 Graphical representation of results

The x-axis denotes the number of register access and the y-axis denotes the CPU time in seconds. Results indicate that backdoor is 20+ times faster than the front door access. In practice, the acceleration will always be better than reported here as the DUT used for testing was very simplistic. Hence backdoor access reduces simulation configuration time of an SOC.

References

[1] UVM users guide, http://www.accellera.org/downloads/standards/uvm/uvm_users_guide_1.1.pdf

[2] IPXACT XML standard, http://standards.ieee.org/findstds/standard/1685-2009.html

[3] Cadence IUS toolset (iregGen examples)