2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Extending the UVM register model generation and integration flow to support user-defined scenarios and register mask values

Shuang Han, Kees van Kaam, Martin Barnasconi
NXP Semiconductors,
Eindhoven, Netherlands
(shuang.han@nxp.com, kees.vankaam@nxp.com, martin.barnasconi@nxp.com)

***Abstract*—This paper describes a methodology and flow which extends the UVM register model generation and integration flow to support user-defined scenarios and register mask values. It presents the usage of IP-XACT to generate UVM register model and extra register mask values. This generated register and register mask is then integrated together with the adapter and predictor functionality in a UVM-based verification environment. By means of an example, the methodology and flow are demonstrated to explain the practical usage for user-defined scenarios and register mask values accordingly.**

Keywords—UVM, Register-Model, IP-XACT, Multiple User Scenarios, Multiple register mask values

I. INTRODUCTION

The UVM register abstraction layer provides a convenient way to verify the implementation of registers in a design under verification (DUV). Due to the large number of registers in a DUV and the numerous small details involved in properly configuring the UVM register layer classes, it is common practice to automatically generate the UVM register model by using a code generator, to guarantee that the register is fully identical to the one in the DUV. This improves the test bench quality and avoids making human errors.

The generated register model will contain a pre-defined register field access policy for each register field. Normally this is enough for the design of registers behavior. But sometimes, a register could be designed to have different access policies in different user-defined scenarios, like writing the same value into a register in configuration mode, user mode or test mode, the final result written into the register is varying by the different register mask value associated with that specific running mode. How to use model generator to generate those extra access policies (register mask values) and how to apply them in the UVM verification environment is a challenge.

Therefore, a methodology has been developed which extends the UVM register model generation and integration flow to support verifying registers design with multiple access policies in different user-defined scenarios.

This paper is organized as follows: Section II will present the register verification requirements, which are used as starting point for the methodology and flow development described in section III. The flow implementation is presented in Section IV, and explains the register model and mask generation and implementation. Section V presents the results. The conclusions are given in section VI.

II. REGISTER VERIFICATION REQUIREMENTS

When using a UVM register model to verify the register implementation in the DUV, it should reflect all the registers in the design and should be structured as a set of hierarchically nested register blocks following the design hierarchy of the DUV. The UVM register abstraction layer should support front-door access (via a bus interface) and back-door access to quickly set or get a register value without the overhead to communicate over the actual bus interface. Furthermore, to communicate with bus-specific UVC's, the register model should implement a register adapter to translate between register transaction and bus sequences and should offer implicit or explicit prediction to update register mirrored values, etc.

Besides these general requirements, we have the following additional requirements for our register verification:

- Verify the register behaviour within n different user scenarios and 2*n register mask values (each user scenarios associated with 2 register mask values, one for read mask and one for write mask).
- Automatic UVM register model and mask generation flow based on the register description captured in an Excel sheet.
- Implement project-specific register access interface adapter to pass extra information regarding user scenarios with generic *uvm_reg_bus_op*
- Extend UVM predictor to update mirrored value in the register model with different register mask values.

Our register description example in excel sheet is illustrated in Figure 1. It contains register names, field names inside of one register, register address, fields offset, register reset values, all the register mask values in different user scenarios, etc. This excel sheet is owned by the architect; RTL designers and verification engineers are involved to review it. The register description is used as the single source for registers RTL code generation, UVM register model generation and software development in our project team.

	A	B	C	D	E	F	G	H	I	J
1	Register Name	Field Name	Address	Offset	Reset_Value_h	M1_Read_Mask_h	M1_Write_Mask_h	M2_Read_Mask_h	M2_Write_Mask_h	M3_Re
2	REG_1		0x0000		0x00000000	-----rrrrrrrrrrrrrrrrrrrrrrrrrrrrrr	----------	-----rrrrrrrrrrrrrrrrrrrrrrrrrrrrrr	----------	-----rrrrrrrrrrrr
3		Field1		0						
4		rfu		27						
5	REG_2		0x0004		0x0000	----rrrrrrrrrrrr	----wwwwwwwwwww	----rrrrrrrrrrrr	----wwwwwwwwwww	--
6		Field1		0						
7		rfu		11						
8	REG_3		0x0006		0x00	r-----rr	w-----ww	r-----rr	w-----ww	
9		Field1		0						
10		Field2		1						
11		rfu		2						
12		Field3		7						
13										

Figure 1. Register description excel sheet example

III. PROPOSED METHODOLOGY AND FLOW

Based on the register verification requirements described in section II, we proposed a new methodology which extends existing UVM register model generation and integration flow to fulfil our special requirements.

In the normal generated UVM register model, the multiple access policies for one register field will not be generated. So first, we need to define a data structure to save all the extra register mask information listed in Figure 1, then extending model generator function to automatically generate extra register mask information in this defined data structure.

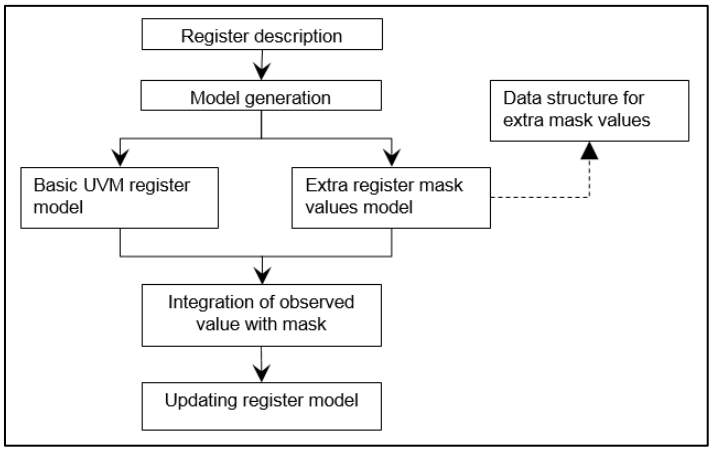

Figure 2. Proposed methodology

When integrating the UVM register model into the UVM test environment, both UVM register model and extra register mask information should be created and can be accessed by the unique register name. The observed register value from the register access interface should be integrated together with the register mask value before updating register model. Proposed methodology is illustrated in Figure 2.

A flow for register model and mask generation and integration has also been developed according to the methodology we discussed above. The register model and mask generation flow is illustrated in Figure 3.

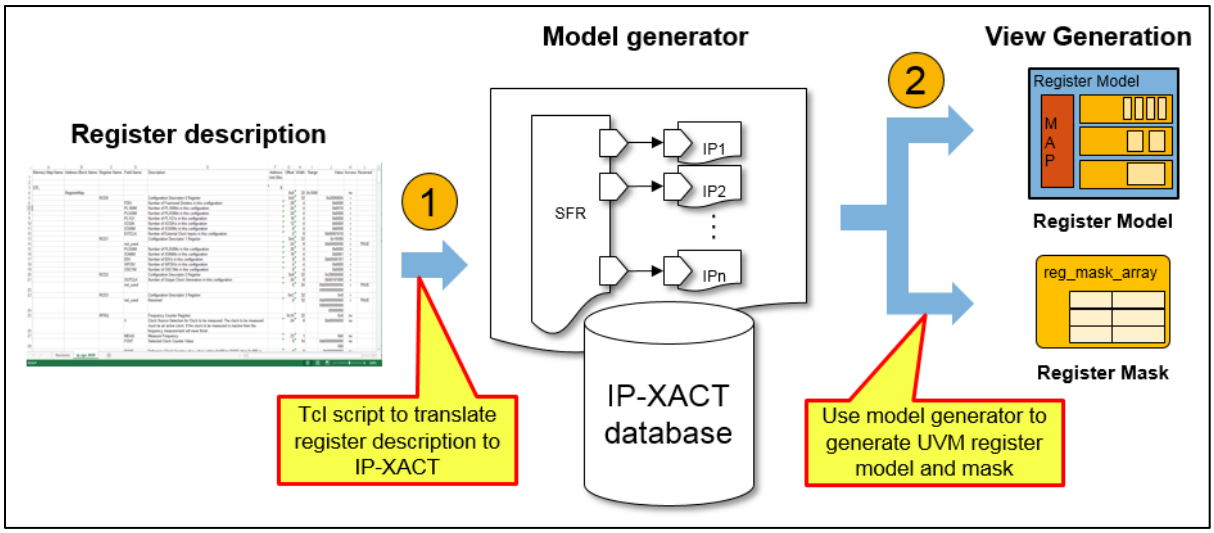

Figure 3. Register model and mask generation flow

Generated register model and mask integration flow is illustrated in Figure 4.

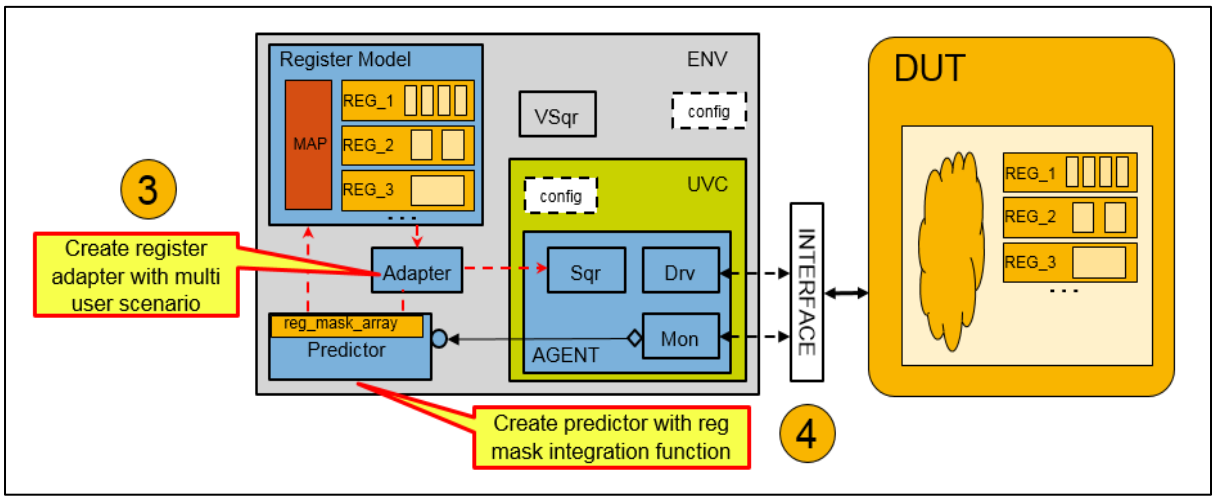

Figure 4. Register model and mask integration flow

IV. FLOW IMPLEMENTATION

As illustrated in section III, the flow is divided into 4 steps. We will use example implementation to show how the flow is implemented in details.

A. Proposed data structure to save all register mask values

To save all extra register mask values and use them in the UVM test environment, a generic *reg_mask* class and a *reg_mask_block* class are implemented. The *reg_mask* class contains members of all the 2*n (n is the number of mode) register masks and provides basic function like *add_mask* and *get_mask* based on read/write and user scenarios. The *reg_mask_block* class instantiated an associated array of *reg_mask*. A generated file called "*myblock_reg_mask.sv*" is also included in *build()* function to fill the *reg_mask_array* with all 2*n register mask

```
class reg_mask extends uvm_object;
  uvm_reg_data_t m1_r_mask  ;
  uvm_reg_data_t m1_w_mask  ;
  ...
  // Constructor
  fucntion new (string name = "reg_mask");
    super.new(name);
  endfunction

  function void add_mask (uvm_reg_data_t m1_r , uvm_reg_data_t m1_w ,...);
  ...
  endfunction

  function uvm_reg_data_t get_mask ( bit wr, user_scenario_e user_scenario);
  ...
  endfunction
endclass
```

Figure 5. Example of reg_mask class snippet

values. The *add_reg_mask* and *get_reg_mask* function are also implemented in *reg_mask_block* class. Figure 5 shows the example implementation snippet of the *reg_mask* class.

Figure 6 shows the example implementation snippet of reg_mask_block class.

```
class reg_mask_block extends uvm_object;
  `uvm_object_utils(reg_mask_block)
  reg_mask reg_mask_array[string];
  // Constructor
  fucntion new (string name = "reg_mask_block");
    super.new(name);
  endfunction

  function void build();
    `include "myblock_reg_mask.sv"
  endfunction

  function void add_reg_mask (string reg_name, uvm_reg_data_t m1_r, uvm_reg_data_t m2_w, ...);
  ...
  endfunction

  function uvm_reg_data_t get_reg_mask ( string reg_name, bit wr, user_scenario_e user_scenario);
    if (reg_mask_array.exist(reg_name))
      return reg_mask_array[reg_name].get_mask(wr,user_scenario);
    else
      `uvm_error("REG_MASK", $sformat("Register %s can't be found in reg_mask_array", reg_name))
  endfunction
endclass
```

Figure 6. Example of reg_mask_block class snippet

B. Step 1: Converts the register description in XLS/CSV into IP-XACT

Many third-party tools generate the IP-XACT XML format from other sources (such as a specification or spreadsheet). These third-party register tools may also auto-generate an UVM register model. So, depending on which tool you are using, the generated IP-XACT file and final UVM register model may differs a bit. Here we will use our flow to show the basic idea to deal with our multiple user scenarios and multiple register mask values issue.

In our practice, we developed a Tcl based tool to convert register XLS/CSV descriptions into IP-XACT register descriptions, then powered by Magillem generator, the UVM register model will be generated. To deal with our issue, all the extra register mask values will be read out from excel sheet and converted into *parameter* together with normal register description in converted IP-XACT file, following IEEE Std 1685-2009 [2]. Generated IP-XACT file is illustrated in Figure 7.

C. Step 2: Automatically generate register model and mask by model generator

As shown in above Figure, all 10 mask values for each register are recorded as *parameter* together with normal register description in the IP-XACT file. Via extending the tcl script used for generating normal UVM register model, a register mask file which named "*myblock_reg_mask.sv*" will be generated together with normal UVM

```
<spirit:register>
    <spirit:name>REG_1</spirit:name>
    <spirit:description>Register 1 </spirit:description>
    <spirit:addressOffset>0x0000</spirit:addressOffset>
    <spirit:size>32</spirit:size>
    .
    .
    .
    <spirit:parameters>
        <spirit:parameter>
            <spirit:name>M1_Read_Mask_h</spirit:name>
            <spirit:value>-----rrrrrrrrrrrrrrrrrrrrrrrrrrrr</spirit:value>
        </spirit:parameter>
        <spirit:parameter>
            <spirit:name>M1_Write_Mask_h</spirit:name>
            <spirit:value>-------------------------------</spirit:value>
        </spirit:parameter>
        .
        .
        .
    </spirit:parameters>
</spirit:register>
```

Figure 7. Generated IP-XACT file with extra register mask values

register model. The generated register mask file is included in the *reg_mask_block* class shown in the proposed ata structure , containing the function call of *add_reg_mask* for all the registers in excel sheet. An generated example of register mask file is shown Figure 8.

```
add_reg_mask("REG_1",
   32'b00000111111111111111111111111111,
   32'b00000000000000000000000000000000,
   ...
);
add_reg_mask("REG_2",
   16'b0000111111111111,
   16'b0000111111111111,
   ...
);
add_reg_mask("REG_3",
   8'b10000011,
   8'b10000011,
   ...
);
```

Figure 8. Generated myblock_reg_mask.sv

D. Step 3: Create register adapter with multi-user scenarios information

Normally adapter is implemented by extending the *uvm_reg_adapter* class and implementing the *reg2bus()* and *bus2reg()* function. It will convert between generic *uvm_reg_bus_op* and bus *sequence_item*. But normal *uvm_reg_bus_op* only contains generic bus operation information, like kind of access, the bus address, the data to write, number of bits being transferred, etc.

For extra information for a register transaction, like register is accessed in which user scenarios, with or without parity, those information can not passed through the generic *uvm_reg_bus_op*. To solve this, you need to declare an extra class from uvm_object containing all extra information you want to pass with your generic bus transaction. The defined user scenarios include mode 1 (M1), mode 2 (M2), mode 3 (M3), etc. Figure 9 shows an example for this extra information class.

```
typedef enum {M1, M2, M3, M4, M5} user_scenario_e;

class extra_bus_info extends uvm_object;

  `uvm_object_utils(extra_bus_info)

  rand user_scenario_e m_user_scenario;
  rand bit             m_parity;
  rand int             m_gap;
```

Figure 9. Extra information class declaration snippet

```
class base_vseq extends uvm_sequence;
  // This set up is required for child sequences to run
  task body;
    if (!uvm_config_db #(myblock_reg_block)::get(p_sequencer, "", "reg_block", reg_block) )
       `uvm_fatal("CONFIG_LOAD", "Cannot get() register model myblock_reg_block from uvm_config_db.")
  endtask: body

  task write_reg_with_mode(string reg_name, int value, user_scenario_e user_scenario=M1, bit parity=0, int gap=0);
    ...
  endtask

  task read_reg_with_mode(string reg_name, output int value, input user_scenario_e user_scenario=M1, bit parity=0, int gap=0);
    uvm_status_e          status;
    uvm_reg               tmpreg;
    extra_bus_info       extra_info;

    if(!reg_block.get_reg_by_name(reg_name)) begin
     `uvm_error(get_type_name(), $sformatf("Can't find register %s in reg_block %s",reg_name, reg_block.get_name()))
     return;
    end
    else tmpreg = reg_block.get_reg_by_name(reg_name);

    extra_info = extra_bus_info::type_id::create("extra_info");
    extra_info.m_user_scenario = user_scenario;
    extra_info.m_parity        = parity;
    extra_info.m_gap           = gap;
    tmpreg.read(.status(status), .value(value), .extension(extra_info));
  endtask
```

Figure 10. Register read/write task with user scenarios example

Then pass all extra information in above class instance into register read/write access method call via the *extension* argument. You can either call register read/write access in a register sequence, or in our case, we created two task (read and write) to encapulate all register access information, and put them in the base virtual sequence where you can call in all extended virtual sequences. Figure 10 shows an example of base virtual sequence implementation.

When user call the read/write task above, the register.read method calls XreadX() method of the uvm_reg class, then a uvm_reg_item object will be created, which contains all the information for bus sequence. During the following internal calls, reg2bus() method in uvm_reg_adapter will be called to translate register access into bus sequence. Figure 11 illustrates how extra information can be passed into bus sequence in the adapter .

```
class myblock_reg_adapter extends uvm_reg_adapter;
  ...
  virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
    myblock_seq_item   myblock_op;
    extra_bus_info     extra_info; //to get extra info
    uvm_reg_item       item;

    myblock_op = myblock_seq_item::type_id::create("myblock_op");
    // to get reg_item extension info from reg.write/reg.read call
    item = get_item(); //this is available only in reg2bus function
    if (item.extension != null) begin
      if (! $cast(extra_info, item.extension))
        `uvm_error(get_name(), "item.extension type is not extra_bus_info.")
        myblock_op.m_user_scenario = extra_info.m_user_scenario;
        myblock_op.m_parity        = extra_info.m_parity;
        myblock_op.m_gap           = extra_info.m_gap;
    end
    else begin
        myblock_op.m_user_scenario = M1;
        myblock_op.m_parity        = 0;
        myblock_op.m_gap           = 0;
    end

    myblock_op.m_direction = (rw.kind == UVM_READ) ? READ : WRITE;
    myblock_op.m_addr = rw.addr;
    myblock_op.m_data = rw.data;

    return myblock_op;
  endfunction: reg2bus
```

Figure 11. Register adapter with multi-user scenarios example

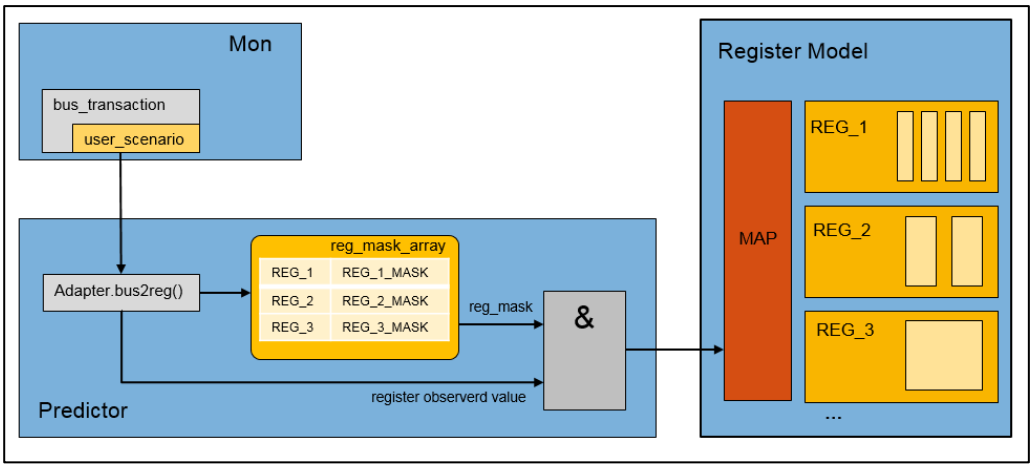

Figure 12. Predictor glue logic for mirrored values update

E. Step 4: Predictor integrate register observed value with register mask value

By default, the register model updates its mirror copy of the register values implicitly, so no need for a predictor. Every time a register is read or written through the register model, its mirror value is updated. However, if other agents on the bus interface perform read and write transactions to DUT registers without the register model, then the register model must learn of these bus operations to update its mirrorred value accordingly [1]. To use a bus monitor to observe all the transaction on the bus interface, then using predictor to get bus transaction from connected bus monitor to explicitly update the register model, doing so can guarantee all register accesses via bus interface will not be missed. In our case, we used an explicit predictor to integrate register observed value from register access interface with register mask value to update mirrorred vaules in the register model. Figure 12 shows this process.

Example predictor implementation is illustrated in Figure 13.

```
class reg_predictor extends uvm_reg_predictor#(myblock_seq_item);
  myblock_reg_mask_block    mask;
  ...
  virtual function void write(BUSTYPE tr);
    uvm_reg           rg;
    uvm_reg_bus_op    rw;
    bit               wr;
    user_scenario_e   user_scenario;

     // get user scenario info from monitored transaction
     user_scenario = tr.m_user_scenario;

     adapter.bus2reg(tr,rw);
     rg = map.get_reg_by_offset(rw.addr, (rw.kind == UVM_READ));
     ...
     if (rg != null) begin
       ...
       foreach (map_info.addr[i]) begin
       if (rw.addr == map_info.addr[i]) begin
         found = 1;
         reg_item.value[0] |= rw.data << (i * map.get_n_bytes()*8);
         predict_info.addr[rw.addr] = 1;
         if (predict_info.addr.num() == map_info.addr.size()) begin

         // integration starts from here
         wr = (reg_item.kind == UVM_READ) ? 0 : 1;
         reg_mask = mask.get_reg_mask(rg.get_name(), wr, user_scenario);
         // integrate register observed value with register mask
         reg_item.value[0] = reg_item.value[0] & reg_mask;
         ...
         rg.do_predict(reg_item, predict_kind, rw.byte_en);
         ...
  endfunction
```

Figure 13. Predictor example snippet

```
UVM_INFO my_virtual_seqs.sv(163) @ 880187500: uvm_test_top.m_env.m_v_sqr@@t_seq WRITE register [REG_3],  write_data: fd, user_scenario: M1, parity: 0, gap: 0
UVM_INFO my_monitor.sv(302) @ 880192500: uvm_test_top.m_env.my_agent::m_monitor [MONITOR:monitor_trans] received m_trans_item is:
...
UVM_INFO reg_predictor.sv(162) @ 880192500: uvm_test_top.m_env.reg_predictor [Predictor_Write] BEFORE MASK:register REG_3, reg_item.kind = 1, user_scenario is M1,
reg_item.value[0] = 00000000000000fd,  reg_mask = 00000000000000000000000000000000000000000000000000000000010000011

UVM_INFO reg_predictor.sv(165) @ 880192500: uvm_test_top.m_env.reg_predictor [Predictor_Write] AFTER MASK:register REG_3, reg_item.kind = 1, user_scenario is M1,
reg_item.value[0] = 0000000000000083,  reg_mask = 00000000000000000000000000000000000000000000000000000000010000011

 register is REG_3, mirrored_value = 83

UVM_INFO my_virtual_seqs.sv(188) @ 990192500: uvm_test_top.m_env.m_v_sqr@@t_seq READ register [REG_3],  read_data: 0, user_scenario: M1, parity: 0, gap: 0
UVM_INFO my_monitor.sv(248) @ 990197500: uvm_test_top.m_env.my_agent::m_monitor [MONITOR:monitor_trans] received m_trans_item is:
...
UVM_INFO reg_predictor.sv(162) @ 990197500: uvm_test_top.m_env.reg_predictor [Predictor_Write] BEFORE MASK:register REG_3, reg_item.kind = 0, user_scenario is M1,
reg_item.value[0] = 0000000000000083,  reg_mask = 00000000000000000000000000000000000000000000000000000000010000011

UVM_INFO reg_predictor.sv(165) @ 990197500: uvm_test_top.m_env.reg_predictor [Predictor_Write] AFTER MASK:register REG_3, reg_item.kind = 0, user_scenario is M1,
reg_item.value[0] = 0000000000000083,  reg_mask = 00000000000000000000000000000000000000000000000000000000010000011

 register is REG_3, write value = 00000000000000fd, mirrored_value_after_write = 0000000000000083, read back value = 83
```

Figure 14. Simulation log file snippet

V. RESULTS

A new methodology and flow has been developed which extends the UVM register model generation and integration flow to support verifying registers design with multiple access policies in different user-defined scenarios. The proposed methodology and flow are clear and easy to be understood. The implementation of the flow is well structured and easy to apply. Proposed data structure to save all register mask values is simple and fully aligned with register model structure. With single source, the extra register mask values together with other register information are automatically converted to IP-XACT and automatically generated UVM register model and mask. The integration process in predictor is fully aligned with register design behaviour. All the implementation has been tested in the real project by different blocks register verification and used by different verification engineer, all users found it easy to be understood and apply. Some simulation log file snippet is shown in Figure 14.

VI. CONCLUSION

The register model and mask generation and integration flow has been used in different block's UVM test environments by many verification engineers. The register model flow introduced in the paper is not something new, only extending the normal register model generation and integration flow to deal with register in multiple user scenarios with multiple register mask values issue. But maybe the idea of this glue logic can be used in other user cases.

REFERENCES

[1] Accellera, "Universal Verification Methodology (UVM) 1.2 User's Guide", www.uvmworld.org

[2] IEEE, "1685-2009 - IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows", https://standards.ieee.org/findstds/standard/1685-2009.html