Deploying Customized Solution for Graphics Registers with UVM1.2 RAL

Roman Wang Nigel Wang Lunping Guo Robert Liu Jia Zhu
Advanced Micro Devices, Inc. Shanghai R&D Center
Roman.Wang@amd.com, Nigel.Wang@amd.com , Lunping.Guo@amd.com, Robert.liu@amd.com, Jimmy.Zhu@amd.com

Abstract- **In modular Universal Verification Methodology (UVM) test benches, the UVM Register Abstract Layer (RAL) is used to model the register behavior and facilitate productive functional verification of programmable hardware. The RAL model is attached on the bus interface UVM Verification Component (iUVC) through the RAL adapter and predictor. This paper illustrates a practical customized RAL solution to address special registers verification challenges in graphics and unify the easy-to-use and easy-to-understand RAL export to handle both general and special registers programming. We also present the RAL reuse considerations from blocks to the subsystem level which have register fabric. The solution is already adopted in teams and well proven to improve verification productivity greatly.**

I. INTRODUCTION

A. Motivation

UVM RAL is one of the most essential parts of the UVM testbench in IP or SoC verification, and it can handle the most common register access requirements. Special cases studies [3] [4] have been presented in previous technical conferences. In our graphics blocks, there are many sideband registers that represent different complex behaviors per specific application. For example, one register can be accessed with the same address, but it is really targeting an entry of a one- dimensional or a multi-dimensional array indexed by decoding kinds of IDs. As we see in the Fig. 1, this kind of special register has a local array with parameterized depth, and each entry of array represents the same type as its parent register. User can access one entry or all entries per application at a time. The common flow can only generated an RAL model for general registers, so the problem is how to handle this requirement without touching generate RAL model and still use the generic RAL methods. Another challenging area is about highly dependent special registers. For example, the access to fields depends on another field in the same register, or multiple registers have cross-dependencies. It is difficult to be handled by a typical RAL approach.

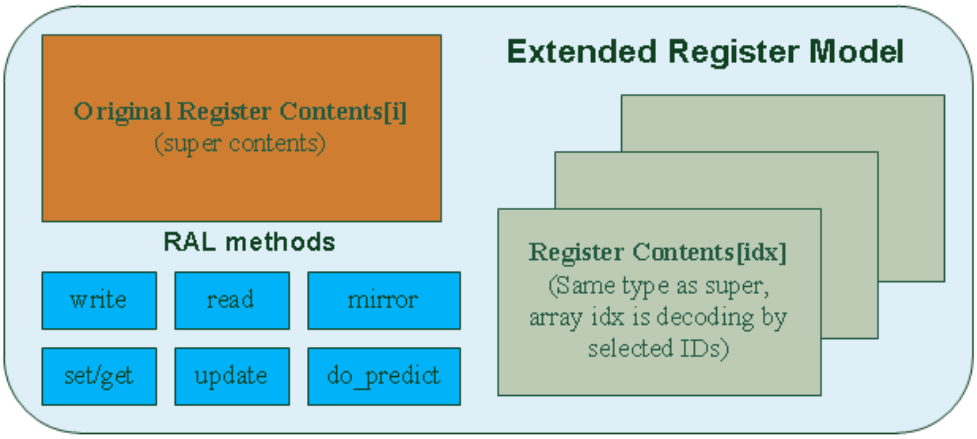

Fig. 1. Special Register Embed Dimensional Array.

Different verification teams may adopt different approaches to access registers by using RAL (examples in Fig. 2), and this may introduce duplicated work, a verification gap and additional maintenance efforts, especially reuse from IP to SoC. The most important is that the engineer must have a good understanding of RAL knowledge to reduce the verification gap between designer and verification engineer. The problem

may become more challenging across teams with overlapping project execution. The challenge raises a common question: why not unify the register access application program interface (API) to get everyone on the same page and hide the RAL knowledge to make verification life easier.

```
1.  Access APIs in uvm_reg
ip_top_ral.blk.reg_1.write(status, reg_data, .parent(this));

2. Access APIs in uvm_reg_block
reg_blk.write_reg_by_name (status, reg_name, reg_data, .parent(this));

3.  User defined task to execute on specific bus
axi4_reg_wr (ip_top_ral.blk.reg_1.get_address(), reg_data);
```

Fig. 2. Different methods to access registers using RAL.

To address above the challenges, we propose a customized RAL solution for our graphics register verification. It has been well qualified in a real project, and different teams are already aligned on the same page. We are presenting our successful story and total solution experience in this paper.

B. Paper Organization

Section II attempts to dissect the proposed customized RAL solution, including the parameterized RAL template, customized RAL adapter, special register callback library, RAL export and customized built-in sequences.

Section III presents the reuse consideration. Finally, this paper concludes in Section V by sharing how we benefit from this solution and discusses a few limitations.

II. PROPOSED SOLUTION

A. Two approaches to handle the special registers in graphics

1) Explicit Approach: customized and parameterized RAL template

a) Modeling Special Register

To resolve the challenge of a special register with an embedded one-dimensional or multi-dimensional array, the idea is to create the parameterized template base class (shown in Fig. 2). To simplify the array modelling, we flatten the potential multi-dimensional array into a general array.

It has three parameters:

- **R**: User extension object type which is for decoding the index to access reg array.
- **T**: It is the type of data inside reg array, so it is supposed to always be the uvm_reg.
- **SIZE**: the depth of the array.

In the Fig. 2, **Trick1**: We do not use the factory method by calling type_id::create(), because we have to override the register at the test layer.

We make use of the **extension** argument in RAL generic methods like write/read, and decode the selected IDs into a user defined type (**R**) of an extension object before calling the methods. Eventually, we can get the extension object in a customized RAL adapter and generate a bus transaction item with IDs fields. In the base template class, we override several generic RAL APIs (in Fig. 3) to implement the ready-to-use code for the special. There are several tricks to resolve the issue during overriding, we will present in the following sections. To improve the debugging, we implement the report() function to dump all register entries of the array into a log file. We defined several extension functions (listing in A - Fig. 4) for the user to implement the differentials and decode the extension object into the following:

- One index, it means only one uvm_reg inside T_inst with a specific index will be touched.
- All_index, it means all uvm_reg entries inside T_inst will be touched.

These user defined virtual functions will be automatically called in overridden APIs in Fig. 3. To decouple the complexity, we also define virtual functions (listing in B - Fig. 4) for the user to implement and assist the decoding.

```
class cust_ral_tmpl_base #(type R=uvm_object, type T=uvm_reg, int SIZE=1) extends  T;
  `uvm_object_param_utils(cust_ral_tmpl_base#(R, T, SIZE))
  ……
  static const string type_name = {"cust_ral_tmpl_base#(",R::type_name, T::type_name, "_", $psprintf("%0d", SIZE), ")"};
  T T_inst[SIZE];   // register contents
  function new(string name = "cust_ral_tmpl_base");
    super.new(name);
```

```
    foreach ( T_inst[i] ) begin
      string name = $sformatf("%s_%0d", T::get_type_name(), i);
      T_inst[i] = new(name);  //[Trick1]
    end
  endfunction: new

  virtual function void build ();
      super.build();
      foreach ( T_inst[i] ) begin
        T_inst[i].build();
        T_inst[i].set_parent(super.get_parent(), null);
      end
  endfunction: build
  // Override APIs
```

Fig. 2. The Template Base Class.

```
function void reset (string kind = "HARD");
function void set (uvm_reg_data_t  value,   string    fname = "",  int   lineno = 0);
function uvm_reg_data_t get (string    fname = "",   int   lineno = 0);
function uvm_reg_data_t get_mirrored_value (string    fname = "",  int       lineno = 0);
virtual task write (output uvm_status_e   status, input  uvm_reg_data_t  value, ……)
virtual task read (output uvm_status_e   status, output  uvm_reg_data_t  value, ……)
virtual task update (output uvm_status_e   status, input  uvm_path_e path = UVM_DEFAULT_PAHT, ……)
virtual task mirror (output uvm_status_e   status, input  uvm_check_e  check = UVM_NO_CHECK, ……)
function void do_predict (uvm_reg_item  rw,  uvm_predict_e   kind = UVM_PREDICT_DIRECT,  uvm_reg_byte_en_t be = -1);
function void report();   // dump T_inst in the log file.
```

Fig. 3. The Overridden APIs.

```
A.  Issue fatal message if not override by user
virtual function void write_extension(input R extension,output int reg_idx,output int reg_idx_all);
virtual function void update_extension(input R extension,input int lineno, output int reg_idx, output int reg_idx_all);
virtual function void mirror_extension(input R trans, output int reg_idx);
virtual function void do_predict_extension(input uvm_reg_item rw, input uvm_reg_data_t reg_value, output int reg_idx, output int
reg_idx_all);

B.  Issue warning message if not override by user
virtual function int get_index_by_trans (ref R trans);
virtual function int get_index_by_value (uvm_reg_data_t value);
virtual function int get_all_index_by_trans (ref R trans);
virtual function int get_all_index_by_value (uvm_reg_data_t value);
```

Fig. 4. User Extension APIs.

- ✓ Case study of set API

The code snippet is shown in Fig. 5. We use the **lineno** argument to indicate the index of array or the flag of idx_all which means set all entries as the same value. If the lineno is equal to the array SIZE, it means idx_all.

Trick2: we do not update the super, because it is only one copy and cannot be updated times in idx_all case.

```
function void set(uvm_reg_data_t  value,   string    fname = "",   int       lineno = 0);
    int reg_idx = 0; int reg_idx_all = 0;
    if(lineno == SIZE)
      reg_idx_all = 1;
    else
      reg_idx = lineno;
   if (!reg_idx_all) begin
      T_inst[reg_idx].set(value[31:0], fname, lineno);
   …..
   // super.set(value, fname, lineno);   //[Trick2]
```

Fig. 5. Customized Set.

- ✓ Case study of get API

The idea is similar to set API. The code snippet is shown in Fig. 6. We use the **lineno** argument to index the array. If the **fname** argument is equal to "super", it results in a super reg, otherwise it results in the entry of array.

```
 function uvm_reg_data_t get(string    fname = "",   int       lineno = 0);
    uvm_reg_data_t reg_val;
    // Range  check
    if(lineno < 0 || lineno >= SIZE) begin
       `uvm_error(get_type_name(), $psprintf("lineno should be inside [0:%d], assign lineno to 0 by default", SIZE-1))
```

```
      lineno = 0;
    end
    if(fname == "super")
      reg_val= super.get();
    else
     reg_val= T_inst[lineno].get();
```

Fig. 6. Customized Get.

- ✓ Case study of get_mirrored_value API

The idea is the same as get API and it just calls the super.get_mirrored_value() or T_inst[lineno]. get_mirrored_value().

- ✓ Case study of write API

The code snippet is shown in Fig. 7. The extension is expected to be passed down outside.

Trick3: we do not directly call super.write(), because the super has the set(value) which will call the overridden set function and update all idx to entry0. We copy the code from super.write and refine it here.

```
virtual task write(     output uvm_status_e     status, … )
          int reg_idx = 0;  int reg_idx_all = 0;
          uvm_reg_item rw_super;
          R trans;
          if(!$cast(trans, extension) || extension == null)
              trans = R::type_id::create("trans");
          write_extension(trans, reg_idx, reg_idx_all);
          if (!reg_idx_all  )  T_inst[reg_idx].set(value[31:0]);
          else ….
          XatomicX(1);   // Refine Begin
          if(reg_idx == 0)
            set(value);   //[Trick3]
          rw_super = uvm_reg_item::type_id::create("write_item",,get_full_name());
          rw_super.element      = this;
          ……
          rw_super.extension    = trans;
          super.do_write(rw_super);
          …
          XatomicX(0);     // Refine End
```

Fig. 7. Customized Write.

- ✓ Case study of read API

The read is simple and just calls super.read(status,value,UVM_FRONTDOOR,.extension(trans));

- ✓ Case study of update API

The code snippet is shown in Fig. 8.

Trick4: We do not use the super.update() for SET_UPDATE command, because the set() function does not update the super and the update will be bypassed due to failure of super.update.

We do not use the super.write(), because the write() task calls the set(value) and it will always update the index zero of array.

```
task update (output uvm_status_e      status, ….)
  R trans ;    int reg_idx = 0;   int reg_idx_all = 0;    uvm_reg_data_t upd;   uvm_status_e sta = UVM_IS_OK;
  uvm_reg_item rw_super;
  ……..
  if(!$cast(trans, extension))
      trans = R::type_id::create("trans");
  update_extension(trans,lineno,reg_idx,reg_idx_all);

  //RAND_UPDATE is only for reg_idx by now
  if(fname == "RAND_UPDATE") begin
    if(!reg_idx_all)
      super.update(status, UVM_FRONTDOOR, map, parent, prior, extension, fname, lineno);
  end //rand_update
  else begin // SET_UPDATE
  if(!reg_idx_all)
        if(T_inst[reg_idx].needs_update()) begin
              rw_super = uvm_reg_item::type_id::create("write_item",,get_full_name());
              rw_super.value[0]      =  T_inst[reg_idx].get();
              ……
              rw_super.extension    = trans;
              super.do_write(rw_super);  //[Trick4]
```

```
    else …….
```

Fig. 8. Customized Update.

- ✓ Case study of mirror API

The code snippet is shown in Fig. 9.

Trick5: We refer to the super.mirror to handle the map and map_info for T_inst[idx], then the entry of the array can be added into the map. It is necessary to call the map.Xinit_address_mapX which is to resolve the map when the block is locked.

```
task mirror (output uvm_status_e      status, ….)
   R trans ;    int reg_idx = 0;     uvm_reg_map       _map;    uvm_reg_map_info  _map_info;
   …..
   mirror_extension(trans,reg_idx);
   _map = super.get_local_map(map, "read()");  // [Trick5]
   if(_map == null) begin
     status = UVM_NOT_OK;     return;
   end
  _map_info = _map.get_reg_map_info(super);
   if (_map.get_reg_map_info(T_inst[reg_idx],0) == null) begin
      _map.add_reg(T_inst[reg_idx], _map_info.offset, _map_info.rights , _map_info.unmapped, _map_info.frontdoor );
      _map.Xinit_address_mapX();
   end
   T_inst[reg_idx].mirror(status, check, path, map, parent, prior, extension, fname, lineno);
```

Fig. 9. Customized Mirror.

- ✓ Case study of do_predict API

The code snippet is shown in Fig. 10. We firstly predict the super and then predict the array.

```
  function void do_predict(uvm_reg_item   rw,   uvm_predict_e     kind = UVM_PREDICT_DIRECT, uvm_reg_byte_en_t be = -1);
      int reg_idx = 0;    int reg_idx_all = 0;

      uvm_reg_data_t reg_value = rw.value[0];
     super.do_predict(rw, kind, be);
     do_predict_extension(rw, reg_value,reg_idx,reg_idx_all);
     if (!reg_idx_all)
        T_inst[reg_idx].do_predict(rw, UVM_PREDICT_DIRECT, be);
    else …..
```

Fig. 10. Customized Do_predict.

b) User extending template class

The code snippet is shown in Fig. 11. The user may implement the selected decoding function and must implement all of the extension functions.

```
class IP1_RAL_TMPL #(type T=uvm_reg, int SIZE=8) extends  cust_ral_tmpl_base#(ip1_user_ext_trans,T,SIZE);
 …….
 function int get_index_by_trans (ref ip1_user_ext_trans trans);
    …… // To implement other decoding function if necessary.
 virtual function void write_extension(input R extension,output int reg_idx,output int reg_idx_all);
        if (extension != null ) begin
            reg_idx      = get_index_by_trans(extension);
            reg_idx_all = get_all_index_by_trans(extension);
       end
  …….
 endfunction
 …… // To implement other *_extension functions.
```

Fig. 11. User Extending Template Class.

c) The customized UVM RAL adapter

In the general register, the UVM RAL adapter plays a role to translate uvm_reg_item into bus item per a specific protocol in the reg2bus function or the process RAL prediction in bus2reg. However, the original challenge was that we could not use uvm_reg_bus_op (which is a UVM built-in struct) to carry the side effect of IDs. However, uvm_reg_item provides another opportunity which has an extension field. It is usually a null handle in a general register but we can use it for the special register. When the RAL APIs executes in uvm_reg_map.svh, the UVM source code will execute set_item with uvm_reg_item (rw) on an adapter. In user RAL adapter, [**Trick5**] we can use the get_item() function to get uvm_reg_item, and then identify its extension to the general process and special registers.

The code snippet is shown in Fig. 12. The workflow from IDs decoding, extension, and adapter is shown in Fig. 13.

```
In the uvm_reg_map.svh
task uvm_reg_map::do_bus_write
  ……
  adapter.m_set_item(rw);
  bus_req = adapter.reg2bus(rw_access);

In the user cusztomized RAL adapter
class ra_reg_gfx_adapter extends uvm_reg_adapter;
  ……
  function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
    bus_trans_item      b_tr;
    user_ext_item       user_ext_tr;
    uvm_reg_item        temp_tr;
    b_tr = new("b_tr");
    user_ext_tr = new("user_ext_tr");
    temp_tr = get_item();  //  [Trick5] the extension value should be got by get_item function
    if(!$cast(user_ext_tr, temp_tr.extension))
      `uvm_fatal(ID, "Get the expected trans info from extension from read")
    b_tr.addr = rw.addr;
    b_tr.id1   = user_ext_tr.u_id1;
    ……..
```

Fig. 12. Customized RAL Adapter.

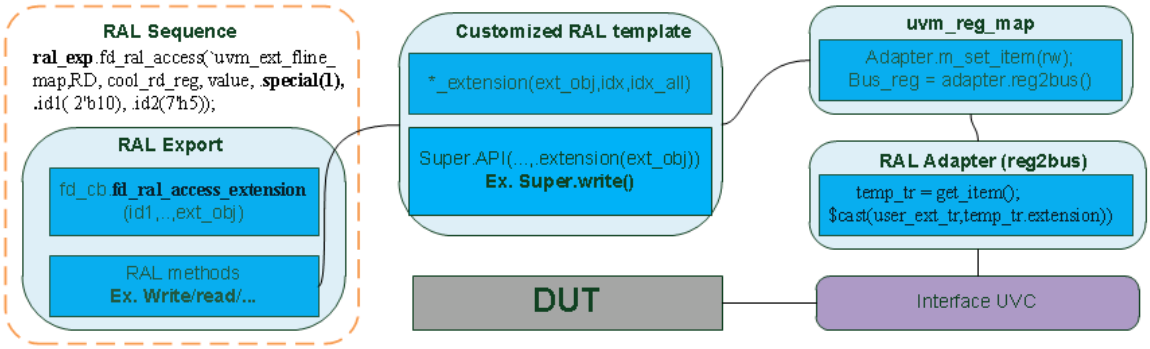

Fig. 13. Workflow of IDs to extension in special register access.

d) Factory override in UVM test

In the IP base test layer, we use the factory override method to replace the original register with the parameterized template, e.g. IP1_RAL_TMPL. The code snippet is shown in Fig. 14.

```
function void ip1_test_base :: build_phase ( uvm_phase phase );

  IP_S_reg::type_id::set_type_override(IP1_RAL_TMPL #( IP_S_reg,16)::get_type(), 1);
  IP_Q_reg::type_id::set_type_override(IP1_RAL_TMPL #( IP_Q_reg,32*2)::get_type(), 1);
……
```

Fig. 14. UVM REG Factory Override.

e) The built-in functional coverage

In the base template, we also implement the built-in functional covergroup to cover every index in array and idx_all flag if it's pre-configured. The sample of covergroup happens after every extension function is called.

2) Implicit approach: special register callback library

The customized and parameterized RAL template can model and handle the special register itself by overriding the original uvm_reg; however, another kinds of special registers are highly dependent on others (Such as the registers or fields). The register access may depend on another register or a specific field, or the register access may depend on its specific field. To resolve these challenges, we adopt the callback approach [5] to implicitly handle inflight.

B. The RAL access export

RAL Access Export (**RAE**) acts as a proxy, it is a general SystemVerilog (SV) class as a singleton pattern. Before we start to talk the RAE, it is necessary to understand a few prerequisites first.

B.1 *Prerequisites*

- *The ID2EXT hook base class*.
  The ID2EXT hook base class is an abstract class (ral_fd_hook_base) and provides a chance for the user to perform customized decoding from IDs to user defined extension types during special register access. The user should extend the base class and implement the fd_ral_access_extension as its pure virtual type.

```
virtual class ral_fd_hook_base;
  ……
  // User defined API to decode the extension by GRBM IDs.
  pure virtual function void fd_ral_access_extension( input bit [`MAX_ID1_W-1:0]   id1 = 0,
                                                      input bit [`MAX_ID2_W-1:0]   id2 = 0,
                                                      …..
                                                      output uvm_object ext_extension);
  pure virtual function void vector2extension( input bit [`MAX_VECTOR_W-1:0]   vector = 0,
                                               output uvm_object ext_extension);
```

Fig. 15. ID2EXT Hook Class.

- *Optional atomic protect*
  The atomic control class is a general SV class as a singleton pattern. It is optionally used in the unique register front door access API (fd_ral_access) to avoid potential unexpected deadlock. In the ral_export, we also provide the enable and disable function for the user to configure inflight.

```
class ral_access_atomic;
  local static   semaphore   m_atomic = new(1);
  local process  m_process;
  static local ral_access_atomic m_inst;
  ……
  task ato_lock(bit on);
    process m_sqr_process;
    m_sqr_process=process::self();

    if (on) begin
      if (m_sqr_process == m_process)
        return;
      m_atomic.get(1);
      m_process = m_sqr_process;
    end
    else begin
       void'(m_atomic.try_get(1)); // Maybe a key was put back in by a spurious call to reset()
       m_atomic.put(1);
       m_process = null;
    end
  endtask: ato_lock

  function void reset();
     void'(m_atomic.try_get(1));
     m_atomic.put(1);
     m_process = null;
  endfunction: reset
  ……
const ral_access_atomic ral_atomic = ral_access_atomic::get();
```

Fig. 16. RAL Atomic Class.

- *Enhanced UVM Message Macros*
  There is a debug overhead when we call the SV task (which has a debugging message inside) in different places. These UVM messages do always display as the file line of the task but not exactly display where it is called. We enhance the UVM message into several uvm_ext_* macros like `uvm_ext_into and provide `uvm_ext_fline_map and `uvm_ext_fline_decl to ease user coding.

```
// Example of enhanced UVM message marco.
`define uvm_ext_info(ID, MSG, VERBOSITY,FILE,LINE) \
 begin \
```

```
    if (uvm_report_enabled(VERBOSITY,UVM_INFO,ID)) \
      uvm_report_info (ID, MSG, VERBOSITY,FILE ,LINE, "", 1); \
  end

  `define uvm_ext_fline_map \
        `uvm_file \
       ,`uvm_line

  `define uvm_ext_fline_decl \
              string filename, \
              int    line,
```

Fig. 17. Enhanced UVM Message Macro.

B.2 *Details in RAE*

- The singleton instance

```
class ral_export;
  static local ral_export m_inst;
  ral_fd_cb_base   fd_cb;
  …..
  static function ral_export get();
    if (m_inst == null) begin
      m_inst = new(m_name);
    end
    return m_inst;
  endfunction
  ……..
  task automatic fd_ral_access( ….. );
endclass: ral_export

const ral_export  ral_exp = ral_export::get();
```

Fig. 18. RAL Export Class.

- Predefined Operation Commands (ral_fd_cmd_t)
  - **WR**: Write one register
  - **WR_COMP**: Write one register and read compare
  - **RD**: Read one register with return data
  - **RD_COMP**: Read with compare
  - **MIRROR_W_CHK**: Read without return data and checking with RAL mirror value
  - **MIRROR_WO_CHK**: Read without return data and no checking with RAL mirror value
  - **SET_UPDATE**: Set the desired value before calling this command
  - **RAND_UPDATE**: atomic operation including reg.randomize() then reg.update
  - **RD_M_WR**: Read modify write
  - **RD_M_WR_COMP**: Read modify write with compare
  - **POLL**: Polling a register times
- Register Front Door Access API
  It is the unique register front door access API and is easy-to-use for user. It can handle both the general and special register. The code snippet is shown in Fig. 19.
  [**Trick5**]: release ral_atomic.ato_lock if it is POLL. It's to avoid the POLL operation which may lock the bus.
  Here is the description of some arguments:
  - ✓ **`uvm_ext_fline_decl**, it helps to pass down the original file/line calling the fd_ral_access.
  - ✓ **reg_access_type**, it is one of the access command defined in ral_fd_cmd_t
  - ✓ **rg**, it is the reference of uvm_reg you want to access
  - ✓ **rwdata**, the read or write data. It is a reference, so you may declare it as "logic [63:0]" ahead.
  - ✓ **special**, it indicates the special register access or not, 0 means general regiser by default. The user must set the fd_cb if special is set.
  - ✓ **id1, …, idn**, they are customized id fields for special decoding.
  - ✓ **poll_num**, it is the number of the polling operation, the default is 2.
  - ✓ **poll_interval**, it is the duration between two polling operations, the default is 10ns.
  - ✓ **timeout**, the register access time for every single access, the default is 5us.

```
task automatic fd_ral_access  (          `uvm_ext_fline_decl
```

```
                                        ral_fd_cmd_t reg_access_type,
                                        uvm_reg rg,
                                        ref logic [`MAX_REG_DATA_W-1:0] rwdata,
                                        input bit special = 0,
                                        bit [`MAX_ID1_W-1:0] id1 = 0,
                                        ……
                                        bit [`MAX_IDn_W-1:0] idn = 0,
                                        int poll_num = 2,
                                        real poll_interval = 10ns,
                                        uvm_sequence_base parent = null,
                                        uvm_reg_map map = null,
                                        real timeout = 5us);
  uvm_object ext_extension;
  process job[] = new [2];
  ……..
  if(atomic_protect_en)
    ral_atomic.ato_lock (1);
  …..
  if(special)
      fd_cb.fd_ral_access_extension(id1,…,idn, ext_extension);  // decoding from IDs to ext_extension
  …..
  if(atomic_protect_en && (reg_access_type==POLL))  //  [Trick6]
     ral_atomic.ato_lock(0);
  …..
    case (reg_access_type)
        WR:begin
            fork: block_WR
                begin
                    job[0] = process::self();
                    rg.write(status, reg_data, UVM_FRONTDOOR, .map(map), .parent(parent), .extension(ext_extension));
                end
                begin
                    job[1] = process::self();
                    delayer (timeout);   // delayer is a task to achieve the delay using uvm_tlm_time. timeout will issue fatal.
                end
            join_any
            foreach (job[j])
              if ( job[j].status != process::FINISHED )
                 job[j].kill();
       ……
  endcase
  if(atomic_protect_en && (reg_access_type!=POLL))
     ral_atomic.ato_lock(0);
```

Fig. 19. Unique RAL Access API.

- Model Trace Interface (MTI)
  Its intent is to dump register access into a file and help the user trace for debugging.

B.3 Examples to adopt

In Fig. 20, we list several usages in different commands.

```
1.  General register READ
ral_exp.fd_ral_access(`uvm_ext_fline_map, RD, state_regs,value);

2.  General register WRITE
value = 64'habcd;
ral_exp.fd_ral_access(`uvm_ext_fline_map,WR, ctrl_regs, value);

3.  Special register READ
ral_exp.fd_ral_access(`uvm_ext_fline_map,RD, cool_rd_reg, value, .special(1), .id1( 2'b10), .id2(7'h5));

4.  Special register SET_UPDATE
cool_rd_reg.set(32'h12345);
ral_exp.fd_ral_access(`uvm_ext_fline_map,SET_UPDATE, cool_rd_reg, value, .special(1), .id1( 2'b10), .id2(7'h5));

5.  Special register POLL
reg_data=64'hxxxx_xxxx;
reg_data[15]=1'b1;   // polling till bit15 is 1.
```

```
ral_exp.fd_ral_access(`uvm_ext_fline_map, POLL, cool_state_reg, value, .special(1), .id1( 2'b10), .id2(7'h6),  .poll_num(10),
.poll_interval(1us));  // total polling 10 times and 1us interval between two polling operation.
```

Fig. 20. Example of RAL Export API Usage.

UVM RAL has several built-in sequences to test the registers automatically. However they are not suitable for our special registers based on a customized template. We rewrite the built-in sequence like uvm_reg_hw_reset_seq, embeds an uvm_pool and the user ID2EXT hook class reference inside. The uvm_pool saves the IDs vector indexed by a special register name. The idea is to audit every register name before doing a mirror operation, and it can process the IDs vector to an extension translation by calling vector2extension. The user should fill the pool and assign its reference to a customized built-in sequence in the test layer. Another usage is the same as the UVM REG built-in sequence. The code snippet is shown in Fig. 21.

```
ral_fd_cb_base   fd_cb;
uvm_object       ext_obj;
uvm_pool#(string, id_vector_t) spec_reg_pool;
……
protected virtual task do_block(uvm_reg_block blk);
    ……
    foreach (regs[i]) begin
        if(spec_reg_pool.exists(regs[i].get_name()))
           fd_cb.vector2extension(spec_reg_pool.get(regs[i].get_name()),ext_obj)
        regs[i].mirror(status, UVM_CHECK, UVM_FRONTDOOR, maps[d], this, .extension(ext_obj));
```

Fig. 21. Example of customized uvm_reg_hw_reset_seq.

III. REUSE CONSIDERATION

In graphics, there are some masters (internal or external) connecting with an internal register fabric which routes to different clients. The internal and external masters present different bus protocols. Especially, the external master port has few different IDs information comparing with the internal. When we reuse the RAL sequence on the master port, the high level UVM RAL adapters should have additional logic to remap the client level IDs into master IDs. The RAL prediction part should always be attached on the IP ports to ensure the exact prediction.

IV. CONCLUSION

We define and deploy the customized solution for graphics registers with UVM1.2 RAL. It can well handle both general and special registers. In real project, it is well proven to be practical, easy to understand, easy to adopt and good for debugging. Different team can work on the same methodology and write the RAL sequence in the same way. When a new team brings up the RAL, they can easily reference the successful story and take low effort to do. The current solution also has few limitations to be improved in near future.

- ✓ The customized template only supports front door register access.
- ✓ Some APIs can't support index all in the customized template.

ACKNOWLEDGMENT

We would like to thank Roman's wife (Liangliang Li) for her continued support and AMD graphics team (Nigel Wang, Lunping Guo, and Animisha Bisht) for executing this solution in a real project.

REFERENCES

[1] IEEE 1800.2-2017 UVM

[2] IEEE 1800-2009 SystemVerilog

[3] Huang Rui, Advanced Micro Devices, Inc, "one Stop Solution for DFT Register Modelling in UVM". DVCon USA 2017.

[4] Krishnan BalaKrishnan, Courtney F., Kaushal M., Analog Devices, "Improving the UVM register model: adding product feature based API for Easier Test Programming". DVCon USA 2016.

[5] Mark Litterick, Marcus Harnisch, Verilab, "Advanced UVM Register Modelling – There's More Than One Way to Skin A Reg". DVCon USA 2014.