Using UVM Register Access Layer Hooks and Callbacks to Support Unconventional Register Structures

Elihai Maicas

Intel

Haifa, Israel

www.intel.com

ABSTRACT

The UVM Register Abstraction Layer (RAL) is used for modeling registers and memories of a DUT. It provides models for many types of simple registers such as: ReadWrite, ReadOnly and more complex ones such as Write1Toggle and ReadSet. However, there are other types of register structures that are not supported by the UVM RAL natively. Register structures composed of multiple registers or those that rely on additional DUT signals are not part of the standard UVM RAL. An example of such a structure is a register-lock mechanism, which prevents a register from being updated, if the value of other register field is set to one (i.e. locked), despite the fact that the write transaction completed successfully.

This paper will provide a complete solution for modeling such a structure, and others, using a set of UVM Register Callbacks and hooks, combined with SystemVerilog events triggered by internal DUT scenarios.

Table of Contents

1. Introduction 3
2. UVM RAL Overview 3
    2.1 Registers access API 3
        2.1.1 Read()/Write() 3
        2.1.2 Register callbacks 4
3. Modeling Complex Register Structures 5
    3.1 Command (Non-Sticky) Register 5
    3.2 Register Write Lock Mechanism 6
    3.3 Register Shadow-Update Mechanism 7
    3.4 Register Indirect Access Mechanism 9
4. Prevent back-to-back Transactions 10
5. Conclusions 11
6. References 11

Table of Figures

Figure 1. Intel® RealSense™ 3D Camera 3
Figure 2. Register Access Simplified Flow 4
Figure 3. Register Callback Class 4
Figure 4. Register Field - Calling post_predict callback 5
Figure 5. Command Register Schema 5
Figure 6. Callback for Command Register - define 5
Figure 7. Callback for Command Register - create and assing 6
Figure 8. Register Write Lock Schema 6
Figure 9. Register Write Lock Callback 7
Figure 10. Register Shadow-Update - HW event update 7
Figure 11. Register Shadow-Update Callback 8
Figure 12. Register Shadow-Update - utility function 8
Figure 13. Register Shadow-Update - Immediate update 8
Figure 14. Register Indirect Access Schema 9
Figure 15. Register Indirect Access Callback 9
Figure 16. Back-to-back Transaction Schema 10
Figure 17. Back-to-back Transaction Prevention Callback 10
Figure 18. Back-to-back Transaction Callback Usage 10

1. Introduction

Intel's 3D camera (RealSense™) is a complex system-on-board with various components: IR camera, color camera, IR laser projector etc. One of these main components is the Imaging ASIC. The Imaging ASIC essentially converts raw data from the sensors to high rate color and depth frames, which would be delivered to the host, for further processing by a designated application.

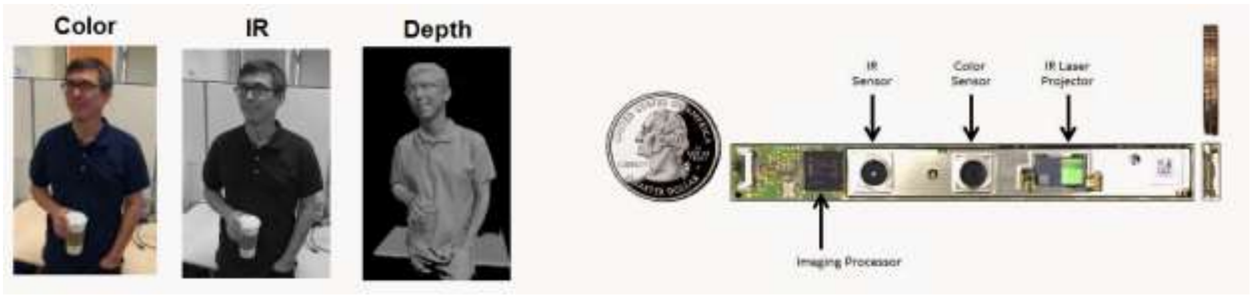


Figure 1. Intel® RealSense™ 3D Camera

In order to render a depth image of the scene, the camera is using an IR laser projector. The return light is caught by the IR sensors, and used to create the 3D image. Using laser beams can be hazardous, particularly for the eye, mostly because they can have high optical intensities even after propagation over relatively long distances. There are a lot of logic mechanisms in the SOC, which were built to eliminate any risk, and to shut-off the laser operation in case of the slightest chance for risk potential. For example, in order for all the logic elements of the data path to be aligned, on-the-fly configuration done by firmware/driver, does not affect immediately, but only when the laser if off between frames.

2. UVM RAL Overview

Note: The following sections provide a high-level overview on UVM RAL. For more detailed information refer to the UVM user-guide.

The UVM register layer classes are used to create a high-level, object-oriented model for memory-mapped registers and memories in a design under test (DUT). It provides a straightforward way for tracking and accessing the DUT registers, and reflects the HW-SW registers blueprint, used by the development teams: architecture, design and verification.

2.1 Registers access API

Register and register-fields have several access methods that can be used to get or set actual HW register values.

2.1.1 Read()/Write()

Calling a read or write method, will invoke a protocol transaction access to the HW register via one of the UVM bus agents associated with the regmodel. At the end of this transaction, the RAL register values are being updated.

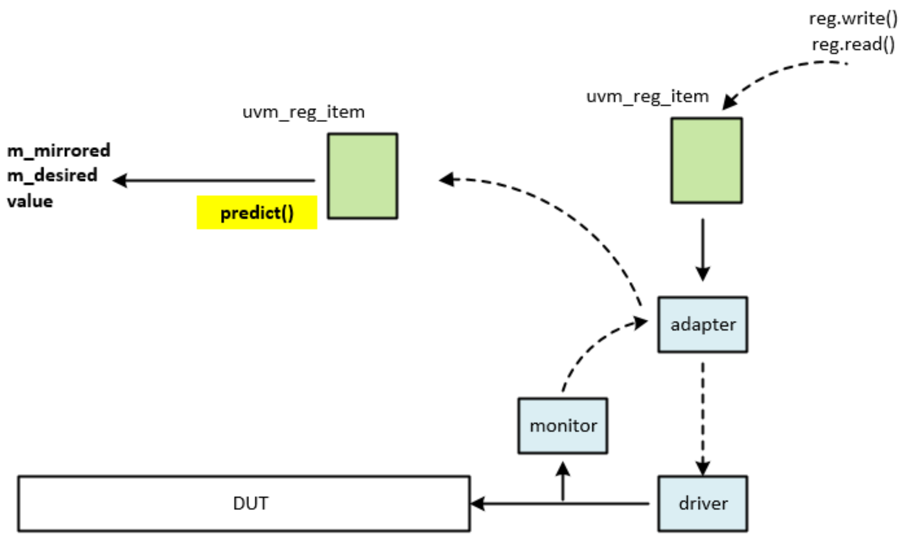
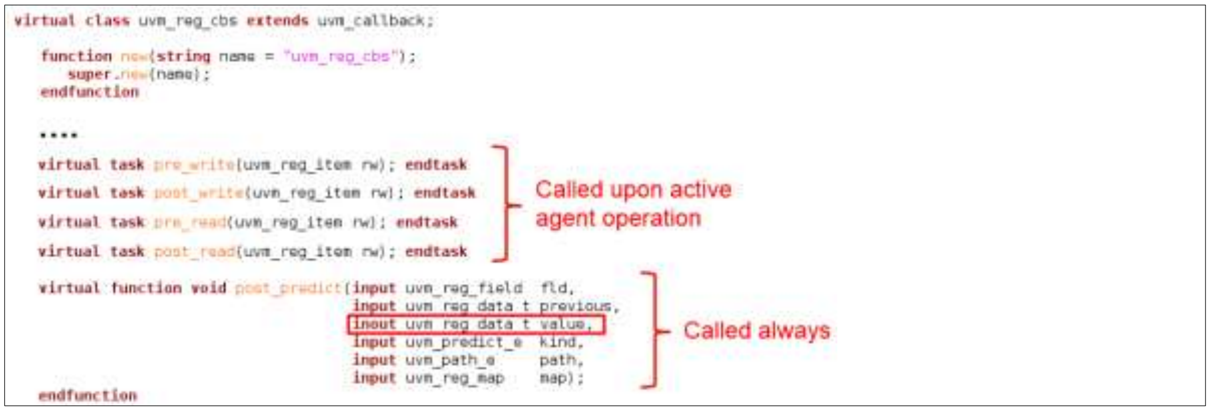


Figure 2. Register Access Simplified Flow

The `predict` method is called in the last step of the read/write flow. After its operation, the predicted (calculated) register field value will be updated. The user may also call the `predict` method, indicating it's a direct call, and update the register value in a backdoor fashion.

2.1.2 Register callbacks

One of the UVM register layer classes is the `uvm_reg_cbs` class. This is a virtual class that has a few empty method declared, that may be extended by the user in order manipulate the behavior of the supplied register access methods.

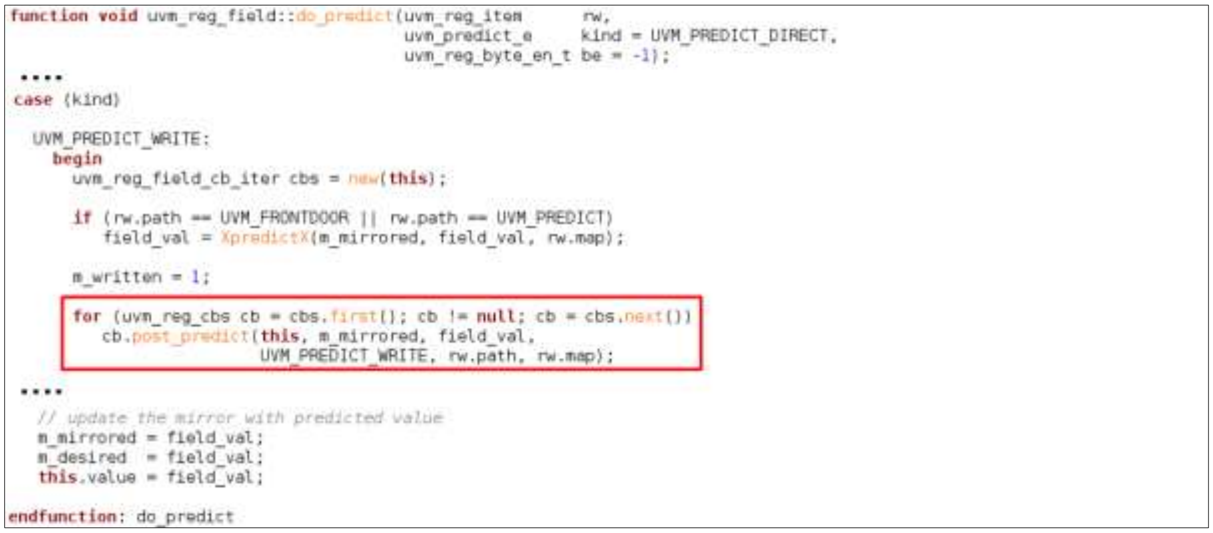


```
virtual class uvm_reg_cbs extends uvm_callback;

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

   ....

   virtual task pre_write(uvm_reg_item rw); endtask
   virtual task post_write(uvm_reg_item rw); endtask
   virtual task pre_read(uvm_reg_item rw); endtask
   virtual task post_read(uvm_reg_item rw); endtask

   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);
   endfunction
```

Figure 3. Register Callback Class

All the `pre_*` and `post_*` tasks defined in the callback class are called when using an active protocol agent. In addition, the `post_predict` function is called when a passive agent is used. An agent is active when registers are accessed by environment and test sequences. An agent is passive when only the monitor is operating and registers are accessed by internal HW logic such as an embedded micro-controller.

As mentioned above, the `predict` function is being called in the last step of the read/write flow. If there was a callback class registered with the field, the `post_predict` function implemented within is being called, and may alter the final value calculated for the field.



```
function void uvm_reg_field::do_predict(uvm_reg_item      rw,
                                        uvm_predict_e     kind = UVM_PREDICT_DIRECT,
                                        uvm_reg_byte_en_t be = -1);
 ....
case (kind)

  UVM_PREDICT_WRITE:
    begin
      uvm_reg_field_cb_iter cbs = new(this);

      if (rw.path == UVM_FRONTDOOR || rw.path == UVM_PREDICT)
         field_val = XpredictX(m_mirrored, field_val, rw.map);

      m_written = 1;

      for (uvm_reg_cbs cb = cbs.first(); cb != null; cb = cbs.next())
         cb.post_predict(this, m_mirrored, field_val,
                         UVM_PREDICT_WRITE, rw.path, rw.map);
 ....
   // update the mirror with predicted value
   m_mirrored = field_val;
   m_desired  = field_val;
   this.value = field_val;

endfunction: do_predict
```

Figure 4. Register Field - Calling post_predict callback

3. Modeling Complex Register Structures

During development, we have had few register types which were not natively supported by RAL specification. This part will describe our implementation using available RAL constructs.

3.1 Command (Non-Sticky) Register

In command (non-sticky) register, the value written to it is self-cleared in the next clock cycle, creating a pulse (command) on the bit field that was written to '1'.

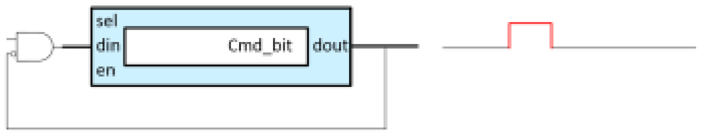
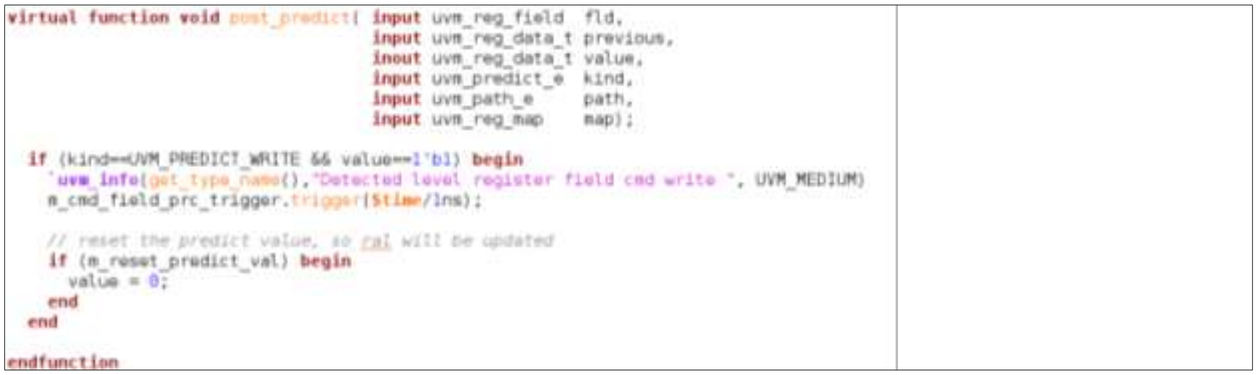


Figure 5. Command Register Schema

Although it's very simple, this behavior does not exist in the RAL pre-defined access policies that can be used as-is. In addition, the pulse exact timing might be required for the Testbench-DUT synchronization.

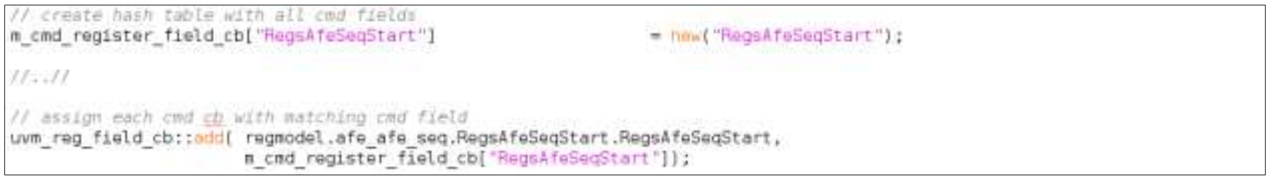
To model this register, we have used the `post_predict` method that resides in the `uvm_regs_cb` class as follows:



```
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 && value==1'b1) begin
    `uvm_info(get_type_name(),"Detected level register field cmd write ", UVM_MEDIUM)
    m_cmd_field_prc_trigger.trigger($time/1ns);

    // reset the predict value, so ral will be updated
    if (m_reset_predict_val) begin
      value = 0;
    end
  end

endfunction
```

Figure 6. Callback for Command Register - define

Each register field is assigned with a callback class, which includes an instance of an event class:



```
// create hash table with all cmd fields
m_cmd_register_field_cb["RegsAfeSeqStart"]                    = new("RegsAfeSeqStart");

//..//

// assign each cmd cb with matching cmd field
uvm_reg_field_cb::add( regmodel.afe_afe_seq.RegsAfeSeqStart.RegsAfeSeqStart,
                       m_cmd_register_field_cb["RegsAfeSeqStart"]);
```

Figure 7. Callback for Command Register - create and assing

Schematic flow:

1. A command register write is being performed by a test sequence
2. The protocol monitor is snooping the lines, and `uvm_reg_item` is being created
3. After the `predict` method calculates final field value, the `post_predict` callback is called
4. If the final value is '1', an event is being triggered, and the final value is reverted to '0'

Any component with access to the event handle can sync to this command and the RAL is synced with the DUT register value.

3.2 Register Write Lock Mechanism

To prevent an unauthorized firmware code to manipulate configurations related to the laser operation (by mistake or intentionally), there are a few groups of registers that get locked after the system completes booting up. The mechanism is as followes: each group has a designated register field bit that locks it. If the bit value is set to 1, all the group is locked, and can be accessed only via a protected transaction from designated micro-controller. The lock bit itself is also part of the same group it locks, which means that it cannot be re-set to 0 with a regular transaction.

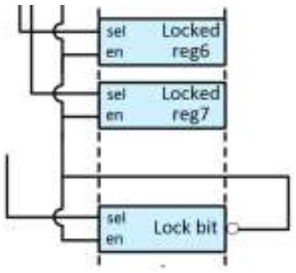
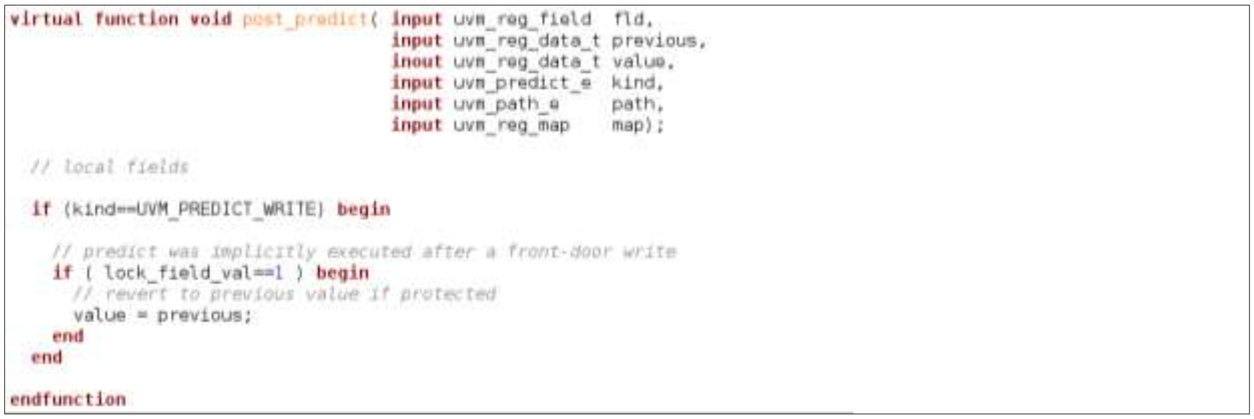


Figure 8. Register Write Lock Schema

The challenge was to verify of the mechanism correctness. A straight forward write to a locked register completes successfully with no error indication, but the DUT register value is not changed. The monitor snooping the write transaction sends an item to the regmodel class, and the new value is being updated to the class register instance, via the predict method.

Using the `post_predict` method, it is possible to check whether the new value should be updated, and later check that the DUT acted the same:



```
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);

  // local fields

  if (kind==UVM_PREDICT_WRITE) begin

    // predict was implicitly executed after a front-door write
    if ( lock_field_val==1 ) begin
      // revert to previous value if protected
      value = previous;
    end
  end

endfunction
```

Figure 9. Register Write Lock Callback

Schematic flow:

1. A lockable register write is being performed by a test sequence
2. The protocol monitor is snooping the lines, and `uvm_reg_item` is being created
3. After the `predict` method calculates final field value, the `post_predict` callback is called
4. The lock field related to this register is checked. If the register is not supposed to be modified, the final value is being reverted to the previous value

Additionally, prior to updating the final value, more conditions may be checked, such as value of a designated debug strap that used to override the lock mechanism for debug purposes, and the identity of the master that performed the write access. As described above, there is a designated micro-controller with higher access level and it can modify locked registers. This indication is being propagated to all callback class instances.

3.3 Register Shadow-Update Mechanism

In order to keep all the data manipulation pipeline and TX-RX parts in sync, on-the-fly registers configurations performed by the micro-controller does not take effect immediately. The new values are kept in shadow registers, and sampled to the active registers on the next pre-defined trigger, for example end-of-frame indication.

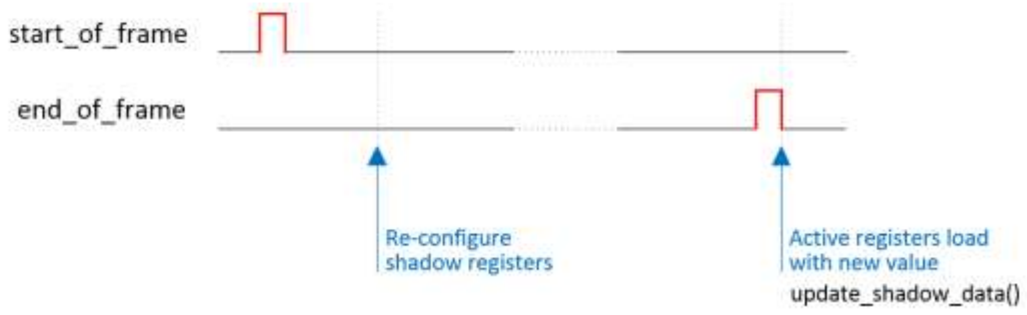
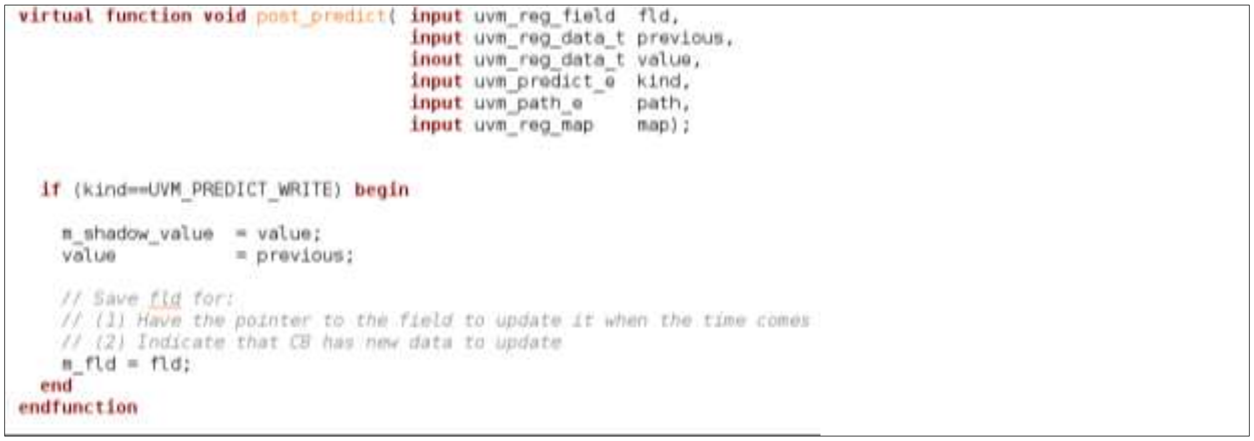


Figure 10. Register Shadow-Update - HW event update

To model this register structure, we have used a combination of the post_predict method and few utility methods:



```
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) begin

    m_shadow_value  = value;
    value           = previous;

    // Save fld for:
    // (1) Have the pointer to the field to update it when the time comes
    // (2) Indicate that CB has new data to update
    m_fld = fld;
  end
endfunction
```

Figure 11. Register Shadow-Update Callback

Schematic flow:

1. A register write is being performed by a test sequence
2. The protocol monitor is snooping the lines, and `uvm_reg_item` is being created
3. After the `predict` method calculates final field value, the `post_predict` callback is called
4. The final value is stored in an internal variable marked as shadow_value and reset to the current held by the field

Now both the DUT active register and its RAL value still has the previous value. Later in the simulation when the dedicated HW event is triggered (e.g. end-of-frame indication), the shadow value is being copied into the RAL register using the `predict` method. The `update_shadow_data` method is also implemented in the `uvm_regs_cb` class, so it has access to all class attributes:

```
virtual function void update_shadow_data ();
  if (m_fld!=null && m_shadow_ready) begin
    m_fld.predict(.value(m_shadow_value), .kind(UVM_PREDICT_DIRECT));
  end
  m_shadow_ready = 0;
endfunction : update_shadow_data
```

Figure 12. Register Shadow-Update - utility function

Additionally, the `update_shadow_data` method may be called in other scenarios when the micro-controller desires to bypass the mechanism and update the new values to the active register promptly, usually after the initial configuration sequence.

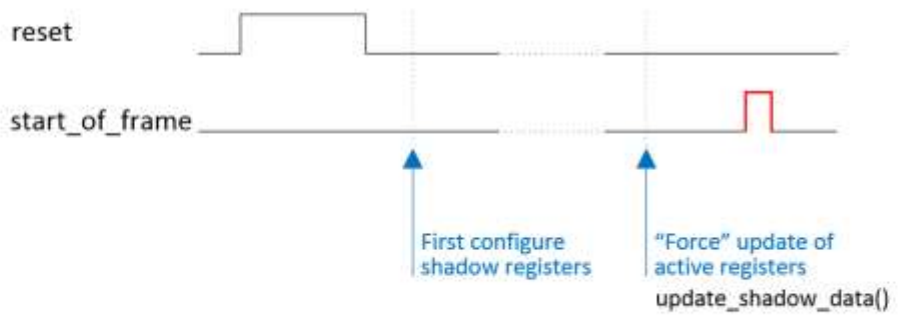

reset
start_of_frame
First configure shadow registers
"Force" update of active registers
update_shadow_data()

Figure 13. Register Shadow-Update - Immediate update

3.4 Register Indirect Access Mechanism

In few 3rd party IPs, there is a use of indirect register/mem access, in order to preserve address space. In its simple implementation, this mechanism has 4 interface registers: `addr_reg`, `data_reg`, `cmd_reg` and `status_reg`.

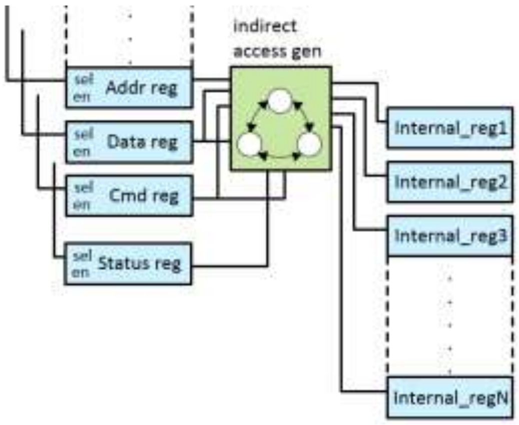


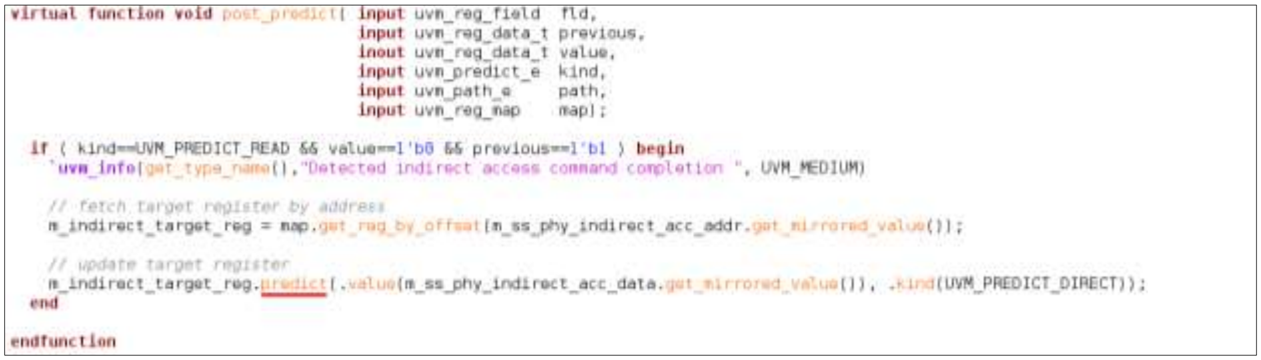
Figure 14. Register Indirect Access Schema

An example flow to perform a write of DATA_X to internal address REG_Y, will be:

1. Write address of REG_Y to addr_reg
2. Write DATA_X to data_reg
3. Write CMD_WR to cmd_reg
4. Poll `status_reg` till its value is "not busy"

The last write will trigger an internal logic mechanism, which will perform the write command to the target internal register.

In order to use the same regmodel for all the registers, and use same register access in all the verification environment, we have used the `post_predict` method to update the RAL registers values upon a "not busy" value read from the `status_reg`.



```
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_READ && value==1'b0 && previous==1'b1 ) begin
    `uvm_info(get_type_name(),"Detected indirect access command completion ", UVM_MEDIUM)

    // fetch target register by address
    m_indirect_target_reg = map.get_reg_by_offset(m_ss_phy_indirect_acc_addr.get_mirrored_value());

    // update target register
    m_indirect_target_reg.predict(.value(m_ss_phy_indirect_acc_data.get_mirrored_value()), .kind(UVM_PREDICT_DIRECT));
  end

endfunction
```

Figure 15. Register Indirect Access Callback

Schematic flow:

1. An indirect access sequence is performed
2. The `status_reg` is being read till its value is 0 – command completed
3. The target register is being fetched for the relevant map according to the address in the `addr_reg` interface register
4. The target register is being updated with the new value, using the `predict` method

4. Prevent back-to-back Transactions

In many on-chip interconnect protocols such as AHB and AXI, there is a concept of back-to-back (pipelined) transactions. What it means, is that a write/read access may start before a previous write/read access completes.

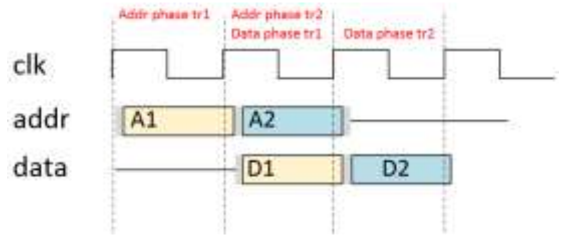
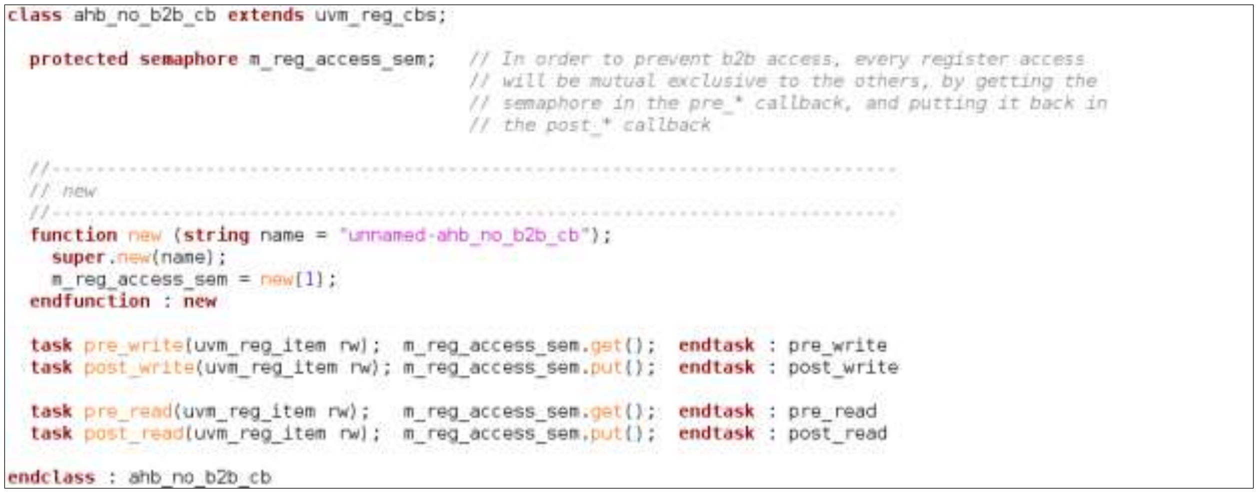


Figure 16. Back-to-back Transaction Schema

When using a protocol verification IP instead of the embedded micro-controller, starting more than one register access in the same simulation tick, may result in back-to-back transaction. In one of our projects, both the micro-controller and the DUT slaves, did not support such a transaction, but the used protocol verification IP did not have a way to disable its generation.

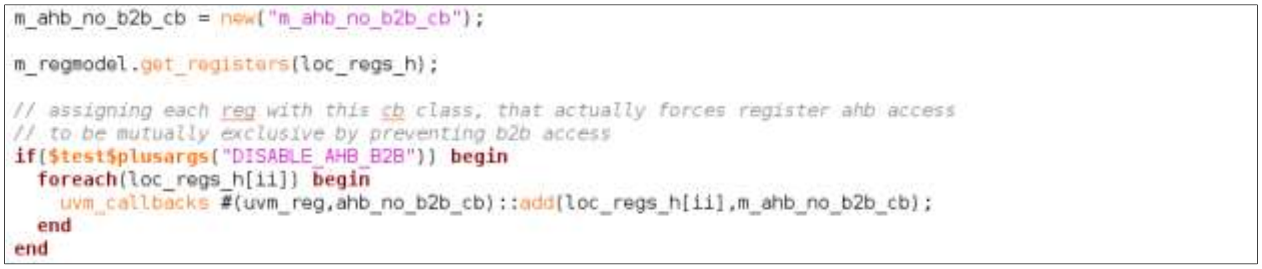
To overcome this drawback, we have used the callback class, with a simple semaphore that was grabbed before a read/write transaction, and released after the transaction was done. This way, we have forced all the transactions to be executed sequentially and not pipelined.



```
class ahb_no_b2b_cb extends uvm_reg_cbs;

  protected semaphore m_reg_access_sem;   // In order to prevent b2b access, every register access
                                          // will be mutual exclusive to the others, by getting the
                                          // semaphore in the pre_* callback, and putting it back in
                                          // the post_* callback

  //--------------------------------------------------------------------------------
  // new
  //--------------------------------------------------------------------------------
  function new (string name = "unnamed-ahb_no_b2b_cb");
    super.new(name);
    m_reg_access_sem = new(1);
  endfunction : new

  task pre_write(uvm_reg_item rw);  m_reg_access_sem.get();  endtask : pre_write
  task post_write(uvm_reg_item rw); m_reg_access_sem.put();  endtask : post_write

  task pre_read(uvm_reg_item rw);   m_reg_access_sem.get();  endtask : pre_read
  task post_read(uvm_reg_item rw);  m_reg_access_sem.put();  endtask : post_read

endclass : ahb_no_b2b_cb
```

Figure 17. Back-to-back Transaction Prevention Callback

Before the test is starting, all the registers were assigned with the same callback class instance.



```
m_ahb_no_b2b_cb = new("m_ahb_no_b2b_cb");

m_regmodel.get_registers(loc_regs_h);

// assigning each reg with this cb class, that actually forces register ahb access
// to be mutually exclusive by preventing b2b access
if($test$plusargs("DISABLE_AHB_B2B")) begin
  foreach(loc_regs_h[ii]) begin
    uvm_callbacks #(uvm_reg,ahb_no_b2b_cb)::add(loc_regs_h[ii],m_ahb_no_b2b_cb);
  end
end
```

Figure 18. Back-to-back Transaction Callback Usage

5. Conclusions

The UVM Register Abstraction Layer has a set of classes that can be used to model most of the common register types used in SOCs. In addition, it supplies a set of callbacks and hooks that enable the user to manipulate the original behavior and implement various alterations. We have demonstrated a few implementations of these callbacks, such as modeling write locked registers, shadow update registers and indirect access mechanism. Furthermore, we have used the callback methods to force the existing verification IP to execute sequential register access transactions, instead of pipelined transactions. The described implementations may be used as a starting point to implement more register behaviors that are not supported natively by the RAL access policies.

6. References

[1] IEEE Computer Society, "SystemVerilog--Unified Hardware Design, Specification, and Verification Language (1800-2012)," New York, 2013.

[2] Universal Verification Methodology (UVM) 1.1 User's Guide, 2011

[3] Synopsys, Inc., *VCS Mx / VCS MXi User Guide*, J-2014.12 ed., 2014.

[4] ReaLSense SDK Design Guidelines, version 2

[5] Shimizu, K. (2016, April 03). UVM Tutorial for Candy Lovers – 16. Register Access Methods. Retrieved May 24, 2017, from http://cluelogic.com/2013/02/uvm-tutorial-for-candy-lovers-register-access-methods/