Blog

UVM Gotchas: UVM Register Layer Prediction Modes

July 17, 2018

by Robin Hotchkiss, Senior Verification Consultant

What are we talking about?

This post talks about the difference between implicit and explicit prediction with front-door accesses in a UVM register model. These differences can cause intermittent false-positive failures due to hard-to-debug thread execution order issues in the simulator. This issue can surface when developing product specific tests or even when running the UVM pre-defined sequences.

What’ s the problem?

The UVM user guide has a section that describes integrating a register model that touches on three modes of prediction. Prediction in this context is the mechanism of how the register model is kept in sync with the design under test.

- Implicit Prediction

    - Active use of the model is when the test calls model class tasks to perform a transfer to interact with the design. These tasks automatically update the model before the task completes with the latest values. For example, on a write action the transfer is initiated and, after it completes the model, is updated to match the data that was sent. A read action updates the model with the data that was returned when the transfer completes.
- Explicit Prediction
    - In this mode the active use of the model does not directly perform prediction during the execution of the tasks. Instead a separate passive monitor is used to feed a UVM predictor class instance to update the model state. In this way the model is still triggering the read and write transfers to the DUT, but the passive components are keeping the model in sync with the design.
- Passive Prediction
    - This mode isn' t really any different from the explicit prediction description except it implies there is no active use of the model – just the passive update. This means that other parts of the design or environment may be causing changes to the design state, and the passive components above are being used to keep the register model in sync with the design.

The description of the different modes, provided above, isn' t that much different from those that can be found in the UVM user guide, so what' s the problem? The problem stems from how these modes are executed. With implicit prediction, a single execution thread is used to trigger the transfer, wait for it to complete, and then update the model state. When explicit prediction is used, the active tasks execute in a separate thread of execution to that of the components performing the passive prediction. If you have worked with SystemVerilog for a while, or any multi-threaded language, talk of concurrent threads working on a single resource should start to raise some concerns.

In the following basic example, the code is performing a basic read and then using a "get" call to obtain the value of a field in the register. While this code doesn' t look multi-threaded, using explicit prediction has that effect.

```
m_reg.read(.status (l_status), .value (l_value));
l_reg_status_flag = m_reg.status_flag.get();
if (l_reg_status_flag)
  // flag is true
else
  // flag is false
```

With implicit prediction, the register and field values will be updated before the "**read**" call completes, so the value returned from the "**get**" call will match the value returned by the transfer. However, with explicit prediction things can be a bit more uncertain depending how the simulator has decided to schedule and switch between executing threads. For example:

1. **Read** completes -> **prediction** completes -> **get** completes
2. **Read** completes -> **get** completes -> **prediction** completes

In the first order example, the code will behave the same as the implicit prediction in all cases. The simulator decided to switch to the predictor thread on the return of the read call. All is good with the world. However, the second order example isn' t as fortunate. Are you thinking this is a contrived example? Yes, it is, but code similar to this is not uncommon and even exists in the UVM pre-defined sequence used to perform the bit bash test.

What' s the solution?

Cheapest and dirtiest? Add a delay after the access but before accessing the model value. You may have seen code that uses #0 to attempt to encourage a simulator to reschedule and allow other threads to execute. However, the #0 delay doesn’t dictate any thread ordering behaviour, and the simulator can choose the same thread to continue so it won’t solve anything. The delay needs to be sufficient to ensure the prediction has completed. Not nice, not pretty, but it works around the immediate issue and allows things to progress. I said it was cheap and dirty, and it is. Any solution like this is not going to be robust, so I strongly recommend you find a better solution. Fragile code, as such a delay can be, can open your code up to other language/tool pitfalls that may cause you or others in the team to waste more time and effort later in the project.

So how do we provide a cleaner fix? You will need to resort to the parts the SystemVerilog language offers to enforce ordering. One solution I have used in the past employed a generic UVM callback to synchronise the active accesses with the passive prediction using semaphores. I created a simple function that attached the callback onto all the registers in the model. The helper function was called after the model was created and was given the model as a parameter. The synchronisation between the passive and active operations means that the explicit prediction mimics the behaviour of the implicit prediction and solves the observed problem. I’ve provided the code below, so you can take a look at one possible solution.

The example solution assumes a simple register operation. A model with more complex register behaviours may have other requirements that are not addressed by this code.

```
class ext_ral_predictor_sync_cb extends uvm_reg_cbs;

//------------------------------------------------------------------------------
// internal variables
//--------------------------------------------------------------------
semaphore         m_waiting_on_ext_predict;
semaphore         m_ext_predict_done;

//------------------------------------------------------------------------------
// UVM macros
//--------------------------------------------------------------------
`uvm_object_utils( ext_ral_predictor_sync_cb )

//------------------------------------------------------------------------------
// new
//--------------------------------------------------------------------
function new (string name = "ext_ral_predictor_sync_cb");
  super.new(name);
  m_ext_predict_done = new(0);
  m_waiting_on_ext_predict = new(0);
endfunction : new

//------------------------------------------------------------------------------
// pre_read
//--------------------------------------------------------------------
virtual task pre_read (uvm_reg_item rw);
  if (rw.path == UVM_FRONTDOOR)
  begin
    m_waiting_on_ext_predict.put();
  end
endtask : pre_read

//------------------------------------------------------------------------------
// pre_write
//--------------------------------------------------------------------
```

```
  virtual task pre_write (uvm_reg_item rw);
    if (rw.path == UVM_FRONTDOOR)
    begin
      m_waiting_on_ext_predict.put();
    end
  endtask : pre_write

  //----------------------------------------------------------------------------
  // post_read
  //------------------------------------------------------------------
  virtual task post_read (uvm_reg_item rw);
    if (rw.path == UVM_FRONTDOOR)
    begin
      m_ext_predict_done.get();
    end
  endtask : post_read

  //----------------------------------------------------------------------------
  // post_write
  //------------------------------------------------------------------
  virtual task post_write (uvm_reg_item rw);
    if (rw.path == UVM_FRONTDOOR)
    begin
      m_ext_predict_done.get();
    end
  endtask : post_write

  //----------------------------------------------------------------------------
  // post_predict
  //------------------------------------------------------------------
  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 (m_waiting_on_ext_predict.try_get())
    begin
      m_ext_predict_done.put();
    end
  endfunction : post_predict
  endclass : ext_ral_predictor_sync_cb

  //----------------------------------------------------------------------------
  // add_ral_ext_predictor_sync
  //------------------------------------------------------------------
  function void add_ral_ext_predictor_sync(uvm_reg_block p_reg_block);
    ext_ral_predictor_sync_cb l_pred_sync_cb;
    uvm_reg l_reg_list[$];
    uvm_reg_field l_field_list[$];
    string l_name;

    // Get a list of all registers
    l_reg_list.delete();
    p_reg_block.get_registers(.regs(l_reg_list));

    // step through all the registers
    foreach (l_reg_list[i])
    begin
      // get the first field of the register
      l_field_list.delete();
      l_reg_list[i].get_fields(.fields(l_field_list));

      // get the name of the register, to create a name for the callback
      l_name = {l_reg_list[i].get_name(), "_sync_cb"};

      // create the callback instance for this register
      l_pred_sync_cb = ext_ral_predictor_sync_cb::type_id::create( l_name );

      // add the callback to the first field of the register
      uvm_reg_field_cb::add( l_field_list[0], l_pred_sync_cb );
    end
  endfunction : add_ral_ext_predictor_sync
```

Related

UVM Reuse: How to use a non-UVM VIP in a UVM environment
August 14, 2018
In "XtremeTeam"

Portable Stimulus at a Minimum -- by Neil Johnson
June 6, 2018
In "XtremeTeam"

Continuous Improvement in XtremeEDA Project Teams
November 7, 2017
In "XtremeTeam"

Posted in XtremeTeam

Leave a Comment

Comments are closed.

More Articles

- **UVM Reuse: How to use a non-UVM VIP in a UVM environment**

  August 14, 2018

  by Ramprasad Chandrasekaran, Principal Verification Consultant What are we talking about? We as a verification community like to talk about creating […]

  Learn More

- **Video: Building An Integrated Verification Flow**

  July 31, 2018

  by Neil Johnson, Chief Technologist DAC2018 has come and gone. It was a great conference as usual with lots of […]

  Learn More

- **Open-Source RISC-V Verification for the Nation**

  July 10, 2018

By Jeremy Ralph, Principal Verification Engineer Many moons ago, as a new grad straight out of university, I started as […]

Learn More

- **Portable Stimulus at a Minimum — by Neil Johnson**

  June 6, 2018

  Portable stimulus is becoming the new hot topic in verification that only got hotter with the release of the early […]

  Learn More

XtremeEDA

- News
- Blog
- Contact

- Subscribe to our RSS Feed
- Facebook
- Twitter
- LinkedIn
- Glassdoor