Modeling a Hierarchical Register Scheme with UVM

Joshua Hardy
Pensar Development
Washington, USA
joshua.hardy@pensardevelopment.com

1 Introduction:

UVM's register model is a powerful tool for verifying various register designs. Simple register designs can be verified with the basic features of the register model. More complex register designs often require the use of the register model's hooks and/or callbacks. A large amount of information regarding advanced register verification techniques can be found in industry literature including [4] and [5]. However, documentation regarding verification techniques for hierarchical registers seems lacking despite being a fairly common architecture.

This paper will describe what is meant by "hierarchical registers" and a solution to create and integrate a UVM register model for them. Verification engineers can then create UVM sequences and analysis components that manipulate and verify these types of registers.

This paper assumes the reader is familiar with UVM and its register model. Information on these base topics is widely available including [1], [2], and [3].

2 Hierarchical Registers:

This paper defines Hierarchical Registers as registers that meet the following conditions:

- Registers are replicated multiple times per the design hierarchy.
- Each copy of a register is referenced by the same address.
- Individual register copies are enabled/disabled at a higher level of hierarchy.
- Any combination of enables is allowed.

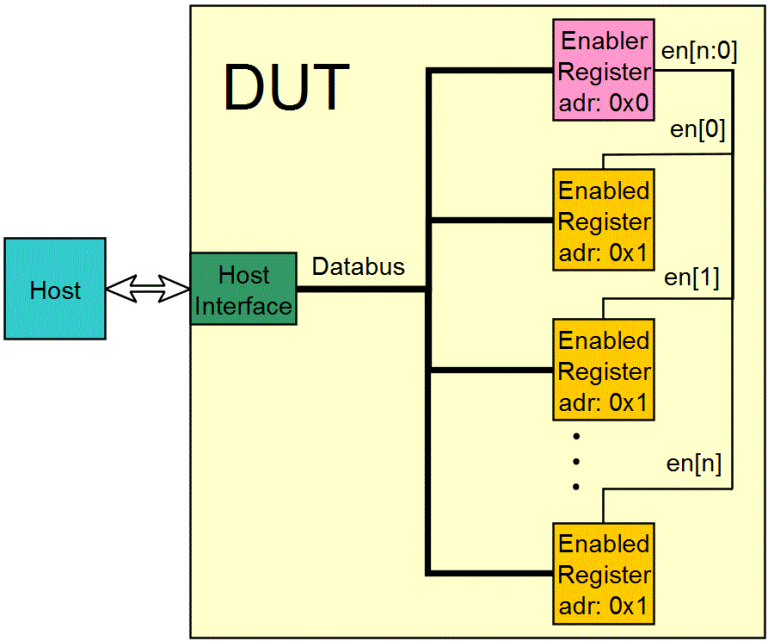

Figure 1 - DUT with hierarchical registers

This paper refers to these registers as ‘enabled’ and ‘enabler’ registers. With multiple levels of hierarchy, it is possible that some registers will qualify as both an ‘enabled’ and ‘enabler’ register. The ‘enabled’ registers are also often referred to as ‘replicated’ registers.

Hierarchical registers reduce the number of required register addresses and allow parallel access of lower levels of hierarchy. This maximizes system bandwidth when lower levels of hierarchy must be manipulated in an identical manner (e.g. broadcast writes). Note that system bandwidth is reduced when lower levels of hierarchy must be uniquely manipulated because users must first execute an operation to enable a unique lower level before manipulating it. Designers must weigh these pros and cons when considering hierarchical registers.

3 UVM Register Model Challenge

Hierarchical registers use an enabling scheme, and methods to verify enabled registers are described in industry literature. For example, see section III.B of [4]. However, hierarchical registers also present the challenge of having multiple unique registers that exist at the same address and can be accessed in parallel. This paper will discuss these challenges in further detail.

4 UVM Register Model Solution

This paper presents a solution that involves some advanced UVM Register Model techniques. The highlights of the necessary customizations are listed here and then discussed in further detail.

- Create a unique *uvm_reg* instance in the UVM register model for each design register instance.
- Create a field in the enabler registers for each lower level of hierarchy.
- Use addresses outside the design's address range for the replicated registers in the design. This avoids the problem of multiple uvm_reg instances with the same address.
- Ensure the sequence item address size matches the register model address size.
- Ensure the driver or BFM address size matches the DUT's address size.
- Extend the *uvm_reg_predictor* to predict the multiple copies of the enabled registers.
- Use callbacks to implement the behavior of the enabled and enabler registers. E.g. a register's mirrored value is only updated when the register is selected.

4.1 Create Unique uvm_reg Instances

For each replicated enabled register in the design we have a corresponding unique instance of *uvm_reg* (actually, the register's extension of *uvm_reg*) in our register model. This way each design register can benefit from the features of UVM's Register Model such as mirrored values and peek/poke ability. We accomplish this by creating a hierarchy of register blocks that mimics the DUT design to give us a unique register block for each replicated section or module of the DUT. The unique register blocks will often be implemented as arrays of register blocks in a higher level register block(s).

4.2 Create Enabler Register Fields

Many DUT designs will likely have enabler registers where each bit of the register selects (enables) one replicated lower level of hierarchy. In this case we must create a *uvm_reg_field* for each bit of the enabler register(s). These individual fields will be used to properly predict the enabled registers as discussed later.

4.3 Register Model Addresses

UVM register models require each register instance to have a unique address. This presents a challenge because each replicated DUT register utilizes the same address. Therefore, we must assign addresses to these registers that do not match their DUT address. A simple way to overcome this is for our register model to use an address space that is larger than the DUTs.

For example, suppose we have a DUT that uses 16 bits of address and has a set of hierarchical registers that are replicated 4 times. In that case our register model can use 18 bits of address and bits [17:16] can be used to identify each unique replicated register. This way, the 16 LSBs of the replicated register addresses are identical and match the address used by the DUT. As we will see, this type of register model addressing scheme becomes useful when we predict our register values and perform frontdoor operations.

4.4 Sequence Item Address Size

As with many UVM Register Model implementations, our register model will be connected to an agent and use explicit prediction. Therefore, we must ensure that the address in our sequence item class used by the adaptor and predictor is large enough to match the address size of our register model. It is not sufficient for the sequence item's address size to be large enough to match the DUT's address size. This will become clear when we discuss extending the *uvm_reg_predictor*.

4.5 Driver/BFM Address Size

If the register model addresses are assigned as discussed above, then their sizes do not match the DUT's address size. Therefore, the driver or BFM must ignore the MSBs of the address in the sequence item when presenting transactions to the DUT.

This allows sequences to perform frontdoor register operations on any of the registers in the register model despite the fact that their addresses sizes do not match the corresponding DUT address size. Note that frontdoor operations will affect only the DUT registers which are selected. Therefore, if a sequence performs a frontdoor operation on a replicated register then the corresponding DUT register may not be exercised and/or other replicated registers may be exercised.

If backdoor paths are utilized, then sequences and analysis components can use peek and poke operations to manipulate each register copy individually.

4.6 Extend uvm_reg_predictor

A single frontdoor operation to a DUT address corresponding to replicated registers will exercise zero, one, or many of the DUT's register replications depending on current state of the enabler registers. Therefore, our register model must execute prediction for multiple registers despite only one frontdoor operation with a single address. We accomplish this by extending the *uvm_reg_predictor*.

Our extended *uvm_reg_predictor* makes multiple predict calls, one to each replicated register, whenever our monitor writes a transaction to the predictor with an address that corresponds to replicated registers. It does this by overriding the base predictor's *write* task to call *super.write* multiple times, each time with a new address corresponding to a different register copy. It is for this reason that the sequence item's address size must be large enough to match the address size in our register model.

4.7 Implement Enabled Register Prediction

Our enabled registers must be predicted appropriately based on the state of the enabler registers. We implement enabled register behavior using the post_predict callback. We borrow the technique discussed in section III.B.2 of [4] regarding Locked/Protected Fields (referred to as enabler/enabled fields in this paper) and make some modifications to address additional challenges.

The first challenge is that with multiple levels of hierarchy, our enabled fields may need to consider multiple enabler fields. Therefore, we pass a queue of enabler fields to the callback's constructor and the callback checks all enabler fields in the *post_predict* function.

The second challenge regards frontdoor read operations because the values in the DUT's register copies may be different. Therefore, the DUT's behavior for reads when multiple registers are selected must be considered. A common behavior is for the DUT to perform a bitwise OR or AND of all selected registers during a read. With behaviors like this, we should not update the mirrored value of registers during frontoor read operations because it cannot be known which registers have which values. The exception, of course, is when only one of the enabled register copies is selected. In that case we can safely update the mirrored value of the selected register.

The *post_predict* callback needs information in addition to its enabler fields to check if the enabled register is the only one selected. It also needs to know the state of the other enabler fields that correspond to the other enabled registers in the other register blocks. Therefore, we pass a queue of the non-corresponding enabler fields to the callback's constructor and the callback checks the corresponding and non-corresopnding enabler fields in the *post_predict* function.

5 Example – 3 Levels of Hierarchy:

A simple design example is a DUT with 3 levels of hierarchy: high, medium, and low. A RW debug register exits at each copy of each level of hierarchy. Enabler registers exist at the higher levels of hierarchy to enable the lower levels of hierarchy.

Our example design has 4 medium levels of hierarchy and 4 low levels of hierarchy in each medium level. The high-level module contains a 'medium-level enable' register at address 0x2. The 'medium-level enable' register is 4 bits wide and each bit selects (enables) one of the medium-level modules. Each medium-level module contains a 'debug' register at address 0x100 and a 'low-level enable' register at address 0x102. Each 'low-level enable' register is 4 bits wide and each bit selects (enables) one of the low-level modules in the medium-level module. Each low-level module contains a 'debug' register at address 0x1000. See Figure 2 for a block diagram of the design.

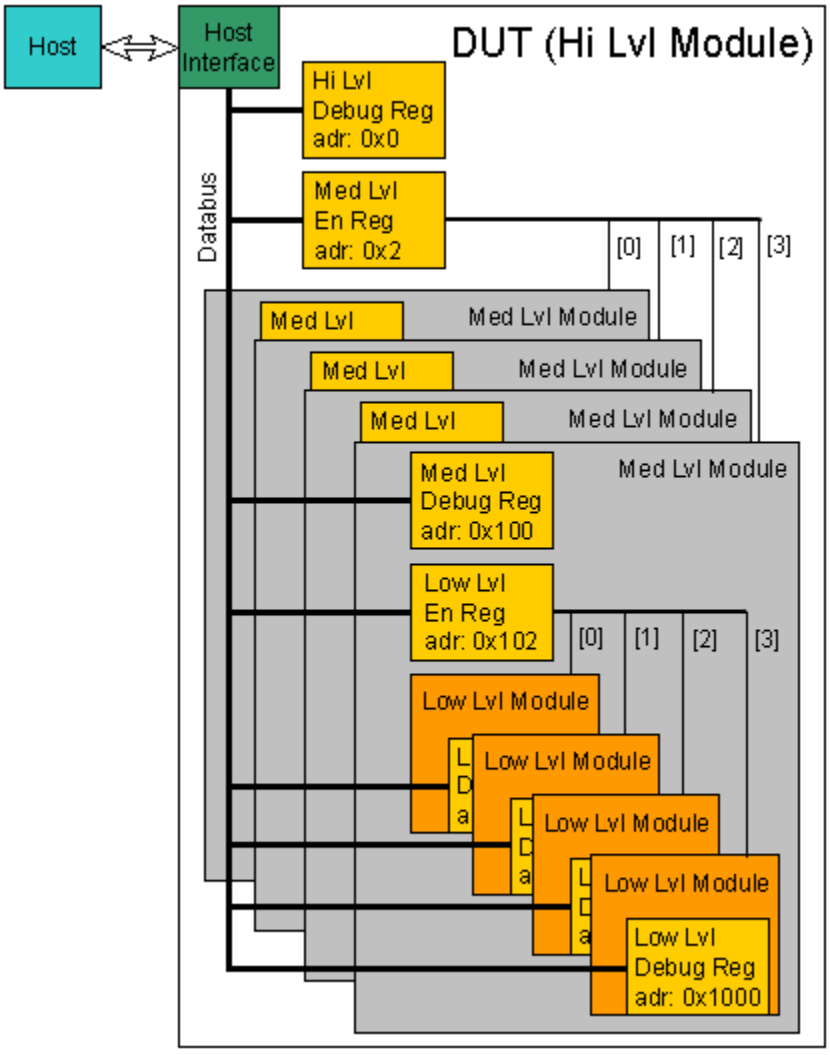

Figure 2 - Sample Design

Our example design implements a bitwise OR of all selected registers (registers that have their corresponding enabler bit(s) set) when the host performs a read.

Our example design uses 16 bits of address (0x0 – 0xFFFF). This becomes important when we assign our register model addresses.

Address	Register	Notes
0x0	HI_LVL_DBG	Debug register to test reads/writes.
0x2	MED_LVL_EN	4-bit register. Each bit enables/disables one copy of the lower level registers.
0x100	MED_LVL_DBG	Debug register to test reads/writes. This register is replicated in the design 4 times. Each replication responds to the address if selected by MED_LVL_EN at address 0x2.
0x102	LOW_LVL_EN	4-bit register. Each bit enables/disables one copy of the lower level registers in the medium-level module.
0x1000	LOW_LVL_DBG	Debug register to test reads/writes. This register is replicated in the design 4 times for each medium level module. Each replication responds to the address if selected by MED_LVL_EN at address 0x2 and the corresponding LOW_LVL_EN at address 0x102.

Table 5-1 - Example Design Registers

Note that **both** the corresponding MED_LVL_EN bit and LOW_LVL_EN bit must be set for a LOW_LVL_DBG register instance to respond to reads and writes.

The following sections illustrate each step of the register model solution.

5.1 Create Unique uvm_reg Instances

We create a register model with a hierarchy of register blocks that mimics the hierarchy of modules in the DUT.

Block	Block Component	Address	Block Instance Dimension	Block Replication Offset
hi_lvl_reg_block	hi_lvl_dbg_reg	0x0		
hi_lvl_reg_block	med_lvl_en_reg	0x2		
hi_lvl_reg_block	med_lvl_reg_block	Registers in this block start at 0x100	4	0x1_0000
med_lvl_reg_block	med_lvl_dbg_reg	0x0		
med_lvl_reg_block	low_lvl_en_reg	0x2		
med_lvl_reg_block	low_lvl_reg_block	Registers in this block start at 0x1000	4	0x10_0000
low_lvl_reg_block	low_lvl_dbg_reg	0x0		

Table 5-2 - Register Model Structure

Table 5-2 - Register Model Structure contains 21 total register blocks (1 hi_lvl_reg_block + 4 med_lvl_reg_block + 4 low_lvl_reg_block/med_lvl_reg_block) and 26 total registers (2 reg in hi_lvl_reg_block + 2 reg/med_lvl_reg_block + 1 reg/low_lvl_reg_block).

If we implement Table 5-2 correctly, then the low_lvl_dbg_reg instances will have addresses as shown in Table 5-3.

Register	Address
hi_lvl_reg_block.med_lvl_reg_block[0].low_lvl_reg_block[0].low_lvl_dbg_reg	0x00_1000
hi_lvl_reg_block.med_lvl_reg_block[0].low_lvl_reg_block[1].low_lvl_dbg_reg	0x10_1000
hi_lvl_reg_block.med_lvl_reg_block[0].low_lvl_reg_block[2].low_lvl_dbg_reg	0x20_1000
hi_lvl_reg_block.med_lvl_reg_block[0].low_lvl_reg_block[3].low_lvl_dbg_reg	0x30_1000
hi_lvl_reg_block.med_lvl_reg_block[1].low_lvl_reg_block[0].low_lvl_dbg_reg	0x01_1000
hi_lvl_reg_block.med_lvl_reg_block[1].low_lvl_reg_block[1].low_lvl_dbg_reg	0x11_1000
hi_lvl_reg_block.med_lvl_reg_block[1].low_lvl_reg_block[2].low_lvl_dbg_reg	0x21_1000
hi_lvl_reg_block.med_lvl_reg_block[1].low_lvl_reg_block[3].low_lvl_dbg_reg	0x31_1000
hi_lvl_reg_block.med_lvl_reg_block[2].low_lvl_reg_block[0].low_lvl_dbg_reg	0x02_1000
hi_lvl_reg_block.med_lvl_reg_block[2].low_lvl_reg_block[1].low_lvl_dbg_reg	0x12_1000
hi_lvl_reg_block.med_lvl_reg_block[2].low_lvl_reg_block[2].low_lvl_dbg_reg	0x22_1000
hi_lvl_reg_block.med_lvl_reg_block[2].low_lvl_reg_block[3].low_lvl_dbg_reg	0x32_1000
hi_lvl_reg_block.med_lvl_reg_block[3].low_lvl_reg_block[0].low_lvl_dbg_reg	0x03_1000
hi_lvl_reg_block.med_lvl_reg_block[3].low_lvl_reg_block[1].low_lvl_dbg_reg	0x13_1000
hi_lvl_reg_block.med_lvl_reg_block[3].low_lvl_reg_block[2].low_lvl_dbg_reg	0x23_1000
hi_lvl_reg_block.med_lvl_reg_block[3].low_lvl_reg_block[3].low_lvl_dbg_reg	0x33_1000

Table 5-3 - low_lvl_dbg_reg Addresses

5.2 Create Enabler Register Fields

We create a *uvm_reg_field* for each bit of the enabler registers. Each field is named according to which DUT module the bit enables. The code for the `med_lvl_en_reg` is illustrated here:

```
class med_lvl_en_reg extends uvm_reg;
   `uvm_object_utils(med_lvl_en_reg)

   rand uvm_reg_field med_lvl03;
   rand uvm_reg_field med_lvl02;
   rand uvm_reg_field med_lvl01;
   rand uvm_reg_field med_lvl00;


   function new(string name = "med_lvl_en_reg");
      super.new(name, 4, UVM_NO_COVERAGE);
   endfunction


   virtual function void build();
      med_lvl03 = uvm_reg_field::type_id::create("med_lvl03");
      med_lvl02 = uvm_reg_field::type_id::create("med_lvl02");
      med_lvl01 = uvm_reg_field::type_id::create("med_lvl01");
      med_lvl00 = uvm_reg_field::type_id::create("med_lvl00");

      med_lvl03.configure(this, 1, 3, "RW", 0, 1'b0, 1, 1, 0);
      med_lvl02.configure(this, 1, 2, "RW", 0, 1'b0, 1, 1, 0);
      med_lvl01.configure(this, 1, 1, "RW", 0, 1'b0, 1, 1, 0);
      med_lvl00.configure(this, 1, 0, "RW", 0, 1'b0, 1, 1, 0);
   endfunction
endclass
```

5.3 Register Model Addresses

The desired register addresses are illustrated in Table 5-2 and Table 5-3. Note that the register model requires a 22-bit address space which is larger than the DUT's 16-bit address space.

We could have chosen different block replication offsets to reduce the required number of address space bits. For example, the block replication offset for low_lvl_reg_block could be 0x4_0000 but using 0x10_0000 allows us to use the hex digits for bits [21:20] and [19:16] to more easily identify which replication the address corresponds to.

The chosen block replication offsets also allow us to more easily increase the number of replications if necessary in the future.

We are careful to properly set the address offsets of our *uvm_reg_map* instances for our register blocks as shown here:

```
class hi_lvl_reg_block extends uvm_reg_block;
   ...
   rand med_lvl_reg_block med_lvl_reg_block_h[4];
   uvm_reg_map HI_LVL_MAP;
   ...
   virtual function void build();
      ...
      HI_LVL_MAP = create_map("HI_LVL_MAP", 'h0, 2, UVM_LITTLE_ENDIAN, 1);
      default_map = HI_LVL_MAP;

      foreach(med_lvl_reg_block_h[i]) begin
         HI_LVL_MAP.add_submap(med_lvl_reg_block_h[i].MED_LVL_MAP, (i * ('h10000)) + ('h100));
      end
      ...
   endfunction
endclass


class med_lvl_reg_block extends uvm_reg_block;
   ...
   rand low_lvl_reg_block low_lvl_reg_block_h[4];
   uvm_reg_map MED_LVL_MAP;
   ...
   virtual function void build();
      ...
      MED_LVL_MAP = create_map("MED_LVL_MAP", 'h0, 2, UVM_LITTLE_ENDIAN, 1);
      default_map = MED_LVL_MAP;

      foreach(low_lvl_reg_block_h[i]) begin
         MED_LVL_MAP.add_submap(low_lvl_reg_block_h[i].LOW_LVL_MAP, (i*('h10_0000))+('hf00));
      end
      ...
   endfunction
endclass
```

Some readers may use a tool from their EDA vendor which automatically generates register model code from a spreadsheet or other entry from. When using such a tool, the same care must be taken to ensure the address offsets are correct.

5.4 Sequence Item Address Size

Our register model design requires the address in our sequence item to be at least 22 bits wide.

5.5 Driver/BFM Address Size

Our DUT uses a 16-bit address space. Therefore, our driver or BFM must ignore the 6 MSBs of the address in the sequence item when presenting transactions to the DUT.

Reads and writes to any one of the replicated registers will cause the same operation on the interface/BFM connected to the DUT. For example, a write to

`hi_lvl_reg_block.med_lvl_reg_block[2].low_lvl_reg_block[3].low_lvl_dbg_reg`

will cause the interface/BFM to send a write operation to the DUT with an address of 0x1000 (not 0x32_1000).

5.6 Extend uvm_reg_predictor

Our register predictor checks each transaction to see if it corresponds to a replicated register. If so, it makes a predict call for each replicated register with the appropriate address. The code for our predictor is illustrated here:

```
class enabled_reg_pred #(type BUSTYPE) extends uvm_reg_predictor #(BUSTYPE);
   `uvm_component_param_utils(enabled_reg_pred#(BUSTYPE))

   function new (string name = "enabled_reg_pred", uvm_component parent);
      super.new(name, parent);
   endfunction : new


   function void write(BUSTYPE tr);
      // If we are accessing a hi level register, then we predict as normal
      if (tr.adr < 'h100) begin
            super.write(tr);

      // If we are accessing a lower level register, then we predict for each copy.
      // We will use a register callback to determine if we should ignore the predict
      // operation based on the Enabler regiser(s)
      end else begin
         int unsigned low_lvl_loop_max = 1;

         for (int med_lvl_index = 0; med_lvl_index < 4; med_lvl_index++) begin
            if (tr.adr[15:0] >= 'h1000) begin
               low_lvl_loop_max = 4;
            end

            for (int low_lvl_index = 0; low_lvl_index < low_lvl_loop_max; low_lvl_index++) begin
               // Call base predictor write function
               tr.adr = tr.adr[15:0] + (med_lvl_index * 'h1_0000) + (low_lvl_index * 'h10_0000);
               super.write(tr);
            end
         end
      end
   endfunction : write

endclass : enabled_reg_pred
```

5.7 Implement Enabled Register Prediction

We register a post_predict callback for each field of each enabled register in our register model. During reads, our DUT performs a bitwise OR of all selected registers. Therefore, we cannot accurately update the mirrored value for our registers unless only one register is selected so our callback receives the two register field queues discussed earlier.

The callback code is shown here. Notice how the callback checks the field queues as appropriate for the write and read operations. The callback contains a *uvm_info* message that verification engineers may want to use when a read is performed with multiple registers selected. Also notice how the enabler queue check is performed for frontdoor operations but not backdoor. Therefore, the state of the enabler fields does not affect peeks and pokes of the enabled registers.

```
class enabled_reg_cbs extends uvm_reg_cbs;
   // Handle(s) to the field(s) that enable this copy of this enabled field.
   local uvm_reg_field m_enabler_reg_field_h[$];
   // Handle(s) to the field(s) that enable the other copies of this enabled field.
   local uvm_reg_field m_other_enabler_reg_field_h[$];

   function new(string name="enabled_reg_cbs",
                uvm_reg_field enabler_reg_field_h[$],
                uvm_reg_field other_enabler_reg_field_h[$]);
      super.new(name);
      m_enabler_reg_field_h = enabler_reg_field_h;
      m_other_enabler_reg_field_h = other_enabler_reg_field_h;
   endfunction : new

   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
   );
      // Reminder! - post_predict is not called if kind is UVM_PREDICT_DIRECT

      if (path inside {UVM_FRONTDOOR, UVM_PREDICT}) begin
         bit is_enabled = 1;

         // If any enabler field is not set then we do not update the
         // mirrored value for this enabled field
         foreach(m_enabler_reg_field_h[i]) begin
            if (!m_enabler_reg_field_h[i].get_mirrored_value()) begin
               value = previous;
               is_enabled = 0;
               break;
            end
         end

         if (kind == UVM_PREDICT_READ) begin
            // If any enabler field for other copies of this field is set then
            // we do not update the mirrored value for this enabled field
            foreach(m_other_enabler_reg_field_h[i]) begin
               if (m_other_enabler_reg_field_h[i].get_mirrored_value()) begin
                  value = previous;
                  if (is_enabled) begin
                     `uvm_info("EVENT_REG_CBS", $sformatf("Frontdoor read made with multiple
                         fields enabled including %s! Mirrored values not updated",
                         fld.get_full_name()), UVM_LOW)
                  end
                  break;
               end
            end
         end
      end
   endfunction : post_predict
endclass : enabled_reg_cbs
```

We create the enabler field queues and register the callbacks in the build phase of the test as shown here:

```
function void build_phase(uvm_phase phase);
   uvm_reg_field     med_reg_fields_h[$];

   ...
   ////////////////////////////////////
   // build register model
   ////////////////////////////////////
   m_hi_lvl_reg_block_h = hi_lvl_reg_block::type_id::create("m_hi_lvl_reg_block_h", this);
   m_hi_lvl_reg_block_h.build();
   m_hi_lvl_reg_block_h.reset();  // Set initial values for all registers

   m_hi_lvl_reg_block_h.med_lvl_en_reg_h.get_fields(med_reg_fields_h);
   foreach (med_reg_fields_h[f]) begin
      uvm_reg_field med_enabler_reg_field_h[$];
      uvm_reg_field other_med_enabler_reg_field_h[$];
      uvm_reg_field low_reg_fields_h[$];

      int med_lvl_index = med_reg_fields_h[f].get_name().substr(7,8).atoi();

      foreach (med_reg_fields_h[g]) begin
         if (f == g) begin
            med_enabler_reg_field_h.insert(0, med_reg_fields_h[g]);
         end else begin
            other_med_enabler_reg_field_h.insert(0, med_reg_fields_h[g]);
         end
      end

      // Set up the enabled registers in the med lvl block
      add_enabled_reg_cbs(m_hi_lvl_reg_block_h.med_lvl_reg_block_h[med_lvl_index],
                          med_enabler_reg_field_h, other_med_enabler_reg_field_h);


      m_hi_lvl_reg_block_h.med_lvl_reg_block_h[med_lvl_index].low_lvl_en_reg_h.get_fields(
         low_reg_fields_h);
      foreach (low_reg_fields_h[h]) begin
         uvm_reg_field low_enabler_reg_field_h[$] = med_enabler_reg_field_h;
         uvm_reg_field other_low_enabler_reg_field_h[$] = other_med_enabler_reg_field_h;

         int low_lvl_index = low_reg_fields_h[h].get_name().substr(7,8).atoi();

         foreach (low_reg_fields_h[i]) begin
            if (h == i) begin
               low_enabler_reg_field_h.insert(0, low_reg_fields_h[i]);
            end else begin
               other_low_enabler_reg_field_h.insert(0, low_reg_fields_h[i]);
            end
         end

         // Set up the enabled registers in the low lvl block
         add_enabled_reg_cbs(
m_hi_lvl_reg_block_h.med_lvl_reg_block_h[med_lvl_index].low_lvl_reg_block_h[low_lvl_index],
            low_enabler_reg_field_h, other_low_enabler_reg_field_h);
      end
   end

   m_env_cfg_h.hi_lvl_reg_block_h  = m_hi_lvl_reg_block_h;

endfunction : build_phase
```

```
// This function will attach an instance of the enabled_reg_cbs class to
// all register fields in the register block.
// This function assumes the block is an enabled block
function automatic void add_enabled_reg_cbs(
   uvm_reg_block uvm_reg_block_h,
   uvm_reg_field enabler_reg_field_h[$],
   uvm_reg_field other_enabler_reg_field_h[$]);
   uvm_reg regs_h[$];

   uvm_reg_block_h.get_registers(.regs(regs_h), .hier(UVM_NO_HIER));

   foreach (regs_h[i]) begin
      uvm_reg_field reg_fields_h[$];

      regs_h[i].get_fields(reg_fields_h);

      foreach(reg_fields_h[j]) begin
         string name;
         enabled_reg_cbs  enabled_reg_cbs_h;

         name = reg_fields_h[j].get_full_name();
         enabled_reg_cbs_h = new(.name(name), .enabler_reg_field_h(enabler_reg_field_h),
                                .other_enabler_reg_field_h(other_enabler_reg_field_h));
         uvm_reg_field_cb::add(reg_fields_h[j], enabled_reg_cbs_h);
      end
   end
endfunction : add_enabled_reg_cbs
```

6 Conclusion

This paper has described a solution to model hierarchal registers using the UVM register model. Perhaps the most significant aspect of this solution is creating a unique UVM register instance for each register instance in the DUT while the address of these two entities are not identical. This differs from typical UVM register implementations where the UVM registers addresses are identical to the DUT addresses. Multiple register instances in a DUT implementing hierarchical registers will have the same address, but each UVM register instance must have a unique address and this solution allows that rule to be followed.

Hopefully this solution has given readers ideas on how they can model their particular DUT(s) with hierarchical registers. Please feel free to contact me with any suggestions on how to improve this paper.

7 References

[1] Accellera, “UVM User Guide, v1.1”, www.uvmworld.org

[2] Accellera, “UVM Reference Guide, v1.1d” , www.uvmworld.org

[3] Mentor, “UVM Cookbook”, https://verificationacademy.com/cookbook/uvm

[4] M. Litterick and M. Harnisch (2014) “Advanced UVM Register Modeling - There’s More Than One Way to Skin A Reg” DVCon 2014 - http://www.verilab.com/files/litterick_register_final_1.pdf

[5] S. Holloway (2013) “UVM Register Modelling: Advanced Topics” DVClub 2013 - http://www.testandverification.com/DVClub/09_Sep_2013/DVClub-UVMRegistersAdvanced_Topics-SteveHolloway.pdf