Leveraging IEEE 1800.2-2017 UVM for improved RAL modelling

Vikas Sharma - (Mentor, A Siemens Business)
Manoj Manu - (Mentor, A Siemens Business)
Ankit Garg - (Mentor, A Siemens Business)
Agenda

- UVM RAL
  - Requirement
  - Challenges
- IEEE 1800.2 UVM RAL
  - Offerings
  - Solution
- Summary
- QA
Universal Verification Methodology (UVM)

• UVM, Stands always, Just like a true friend
• Guides & Supports everybody, and it could be
  – At times when executing, a verification test
  – At times when debugging, what went wrong
  – At times when pursuing, an unlawful timeline
• Offers many rational things & thoughts to be relied upon
• Register Abstraction Layer (RAL) is one of them, and stands out for
  – Being the most important one (automatic mirroring & predictions)
  – Being the most customizable one (implementation specific)
Register Abstraction Layer (RAL)

**Sequence Abstraction**
- Immune to changes in DUT
- Enables vertical reuse

**Register Model Abstraction**
- Imitates DUT registers specification
- Auto generated or hand written

**Address Map**
- Address space offset
- Specifies bus agent
- Specific bus adaptor

**Adaptor**
- Generic-to-Specific
- Specific-to-Generic

Image Courtesy: Verification Academy and UVM Cookbook
Requirements : UVM-RAL

• Dynamic address map
  – When address maps are required to be changed during the simulation
    • Addition/Deletion and Reconfiguration of registers and memories
  – When any new interrupt or hot-join occurs during simulation

• Multiple address map
  – When there are multiple masters in a system
    • Separate address map and adaptor for every bus interface
  – When there are multiple view (set of address offset) to the same set of registers
    • Separate address map for separate set of address offsets
Challenges : UVM-RAL

• Static address map
  – Allows only static locking with `lock_model()` at build time
  – It checks `is_locked()`, and if found locked then no further structural changes can be made
  – If really wanted to do, then extend base and rewrite many functionalities
  – No standard mechanism is available to unlock the register maps

• Single address map
  – Doesn’t allow a register to be added to multiple address map
  – Doesn’t allow a memory to be added to multiple address map
  – Doesn’t allow a sub-map to be added to multiple address map
  – Provides restriction of the same parent block

```c
// Once locked, no further structural changes, such as adding registers or memories, can be made.
function void uvm_reg_block::lock_model();

if (is_locked())
    return;

locked = 1;

foreach (regs[rg_]) begin
    uvm_reg rg = rg_
    rg.Xlock_modelX();
end

foreach (mems[mem_]) begin
    uvm_mem mem = mem_
    mem.Xlock_modelX();
end

foreach (blks[blk_]) begin
    uvm_reg_block blk=blk_
    blk.lock_model();
end
```
IEEE 1800.2 UVM-RAL

• Accellera in conjunction with IEEE launched 800.2 UVM standard
• IEEE 1800.2 standard comes with
  – Bug fixes,
  – Modified API’s (backward compatible)
  – New API’s and features
  – Deprecated APIs (activated again by using +UVM_ENABLE_DEPRECATED_API )
• Proposes the solution for the problems in traditional UVM-RAL
  – Dynamic and multiple address maps
• Allows removal or addition of registers and memories
• Allows an association to different address maps
Offerings : IEEE 1800.2 UVM-RAL

• New API `unlock_model()`
  – Unlocks the register model, bringing the register model to the state before `lock_model()`
  – Recursively unlocks the entire register model and sub-blocks
  – Sets the locked bit to 0

• New API `wait_for_lock()`
  – Blocks until `lock_model()` completes

```cpp
// Brings back the register mode to a state before lock_model() so that a subsequent lock_model() can be issued
virtual function void unlock_model();

bit s[uvm_reg_block]=m_roots;
m_roots.delete();

foreach (blks[blk_j])
  blk._unlock_model();

m_roots=s;
foreach(m_roots[b])
m_roots[b]=0;

locked=0;
endfunction

// Waits for the event to be triggered when locked using lock_model()
virtual task wait_for_lock();
  @m_uvm_lock_model_complete;
endtask
```
Offerings : IEEE 1800.2 UVM-RAL

• New API **set_lock()**
  – Sets the lock mode for the current register block and all its sub blocks

• New API **unregister()**
  – Unregisters recursively all content from map
  – Removes all knowledge of map from current block, registers, memories, and virtual registers
  – Enables the reuse of the map content and objects with a fresh map instance by using the add_** APIs
  – If used **this.unregister(map)** then full register map will get unregistered
  – If used **map.unregister(reg/mem)** then only specific register or memory will get unregistered

```cpp
virtual function void set_lock(bit v);
  locked=v;
  foreach(blks[idx])
    idx.set_lock(v);
endfunction

// Brings back the register mode to a
// state before lock_model() so that a
// subsequent lock_model() can be issued
//
virtual function void unlock_model();

bit s[uvm_reg_block]=m_roots;
  m_roots.delete();

  foreach (blks[blk_])
    blk_.unlock_model();

  m_roots=s;
  foreach(m_roots[b])
    m_roots[b]=0;
  locked=0;
endfunction
```
Solution: IEEE 1800.2 UVM-RAL

- Let’s suppose that there is a test requirement where:
  - State-1 and State-2 specify different register structures and address maps as explained in the above figure.
  - The simulation starts from the initial model and moves the system from State-1 to State-2, and after some time the system returns to State-1 as explained in the figure below.
  - At every stage, user needs to reconfigure the model, but it cannot be done because the system is already locked during initial model setup.
Solution: IEEE 1800.2 UVM-RAL

- **DUT register structure and Standard Modelling**
  - **Build Register Model**
    - `add_reg()` and `add_mem()`
  - **Use lock_model()** to lock the register model
  - **Sequences to move states**
  - **State transitions happening**
  - **Interrupt triggers the address map change**
  - **wait_for_lock()**
  - **Use unlock_model()** to unlock model during simulation
  - **Use unregister()** to delete traces of all or specific ones
  - **Add new registers, memory and maps**

- **State require change in address map**
  - **YES**
  - **NO**
Solution: IEEE 1800.2 UVM-RAL

- **map_as_per_states()** function
  - Handles the register mapping based on the states of the system
  - Called every time the system state changes
  - Unlocks the register model and unregisters the register maps
  - Calls function **map_state_1()** and **map_state_2()** when in **STATE_1** and **STATE_2** respectively

```cpp
// Remapping the register model as per states
function void map_as_per_states(state_e state);

unlock_model();  // Unlocks the model
unregister(map_A);  // Unregisters the address map_A
unregister(map_B);  // Unregisters the address map_B
map_A = null;      // Delete the address map_A
map_B = null;      // Delete the address map_B

case(state)
  STATE_1 : map_state_1();
  STATE_2 : map_state_2();
endcase

// Locks the model
lock_model();
endfunction: map_as_per_states
```
function void mapState2()
{
    reg_A = ctrl_status_reg::type_id::create("reg_A");
    reg_A.configure(this, null, ";");
    reg_A.build();
    reg_B = ctrl_status_reg::type_id::create("reg_B");
    reg_B.configure(this, null, ";");
    reg_B.build();
    reg_C = ctrl_status_reg::type_id::create("reg_C");
    reg_C.configure(this, null, ";");
    reg_C.build();
    reg_D = ctrl_status_reg::type_id::create("reg_D");
    reg_D.configure(this, null, ");");
    reg_D.build();
    mem_A = mem_model::type_id::create("mem_A");
    mem_A.configure(this, ");");
    mem_B = mem_model::type_id::create("mem_B");
    mem_B.configure(this, ");");
}

function void mapState1()
{
    reg_A = ctrl_status_reg::type_id::create("reg_A");
    reg_A.configure(this, null, ";");
    reg_A.build();
    reg_B = ctrl_status_reg::type_id::create("reg_B");
    reg_B.configure(this, null, ";");
    reg_B.build();
    reg_C = ctrl_status_reg::type_id::create("reg_C");
    reg_C.configure(this, null, ");");
    reg_C.build();
    reg_D = ctrl_status_reg::type_id::create("reg_D");
    reg_D.configure(this, null, ");");
    reg_D.build();
    mem_A = mem_model::type_id::create("mem_A");
    mem_A.configure(this, ");");
    mem_B = mem_model::type_id::create("mem_B");
    mem_B.configure(this, ");");
}

// Add memories and registers to the map_A
map_A = create_map("map_A", "\x06", 4, UVLITTLE_ENDIAN, 1);
map_A.add_reg_reg A, 32\x0000_0000, "RW"
map_A.add_reg_reg B, 32\x0000_0000, "RW"
map_A.add_reg_mem A, 32\x0000_0000, "RW"

// Add memories and registers to the map_B
map_B = create_map("map_B", "\x06", 4, UVLITTLE_ENDIAN, 1);
map_B.add_reg_reg A, 32\x0000_0000, "RW"
map_B.add_reg_reg B, 32\x0000_0000, "RW"
map_B.add_reg_mem B, 32\x0000_0000, "RW"

endfunction: mapState2

endfunction: mapState1

Changing the address map of map_A from State-1 to State-2

Changing the address map of map_B from State-1 to State-2
Summary

- Identified the problem of static nature of the RAL model in traditional UVM
- Introduced new IEEE 1800.2 UVM RAL model
- Proposed solutions targeting dynamic address mapping
- Illustrated an example changing the register model dynamically using system states
- Demonstrated the usage of new APIs provided in the IEEE 1800.2 UVM RAL model
Questions