Advanced UVM Register Modeling
There’s More Than One Way to Skin A Reg

Mark Litterick
Verilab GmbH, Munich, Germany

Copyright © 2014 Verilab & DVCon
Introduction

• UVM register model **overview**
  – structure, integration, concepts & operation
  – field modeling, access policies & interaction
  – behavior modification using hooks & callbacks

• Modeling **examples**
  – worked examples with multiple solutions illustrated
  – field access policies, field interaction, model interaction

• Register model **performance**
  – impact of factory on large register model environments
Register Model Structure

- Register model (or register abstraction layer)
  - models memory-mapped behavior of registers in DUT
  - topology, organization, packing, mapping, operation, ...
  - facilitates stimulus generation, checks & coverage

- Diagram showing register model, registers, fields, and interactions with memory and CPU.
• Set of **DUT-specific** files that extend *uvm_reg* base
• Instantiated in *env* alongside bus interface UVCs
  – *adapter* converts generic *read/write* to bus transactions
  – *predictor* updates model based on observed transactions

**NORMALLY AUTO-GENERATED DUE TO REGULAR STRUCTURE AND LARGE SIZE**
Register Model Concepts

- Normal **front-door** access **via** bus transaction & I/F
  - sneaky **backdoor** access **via** `hdl_path` - no bus transaction

- **Volatile** fields modified by non-bus RTL functionality
  - model updated using **active monitoring** **via** `hdl_path`
Active & Passive Operation

- Model must tolerate active & passive operations:
  1. **active** model read/write generates items via adapter
  2. **passive** behavior when a sequence does not use model
  3. **passive** behavior when embedded CPU updates register
• Use-case can be register or field-centric
  – **constrained random** stimulus typically register-centric
    e.g. reg.randomize(); reg.update();
  – **directed** or higher-level **scenarios** typically field-centric
    e.g. object.randomize(); field.write(object.var.value);
Register Field Modeling

- **Field access policy**
  - self-contained operations on this register field

- **Field interaction**
  - between different register fields

- Register **access rights** in associated memory map

- Model functional **behavior of DUT for volatile** fields
Field Access Policies

- **Comprehensive pre-defined field access policies**

<table>
<thead>
<tr>
<th></th>
<th>NO WRITE</th>
<th>WRITE VALUE</th>
<th>WRITE CLEAR</th>
<th>WRITE SET</th>
<th>WRITE TOGGLE</th>
<th>WRITE ONCE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NO READ</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>WO</td>
<td>WOC</td>
<td>WOS</td>
<td>-</td>
<td>W01</td>
</tr>
<tr>
<td><strong>READ VALUE</strong></td>
<td>RO</td>
<td>RW</td>
<td>WC</td>
<td>WS</td>
<td>W1T</td>
<td>W1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>W1C W0C</td>
<td>W1S W0S</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>READ CLEAR</strong></td>
<td>RC</td>
<td>WRC</td>
<td>-</td>
<td>WSRC</td>
<td>W1SRC W0SRC</td>
<td></td>
</tr>
<tr>
<td><strong>READ SET</strong></td>
<td>RS</td>
<td>WRS</td>
<td>WCRS</td>
<td>W0CRS</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Just defining access policy is **not enough**! Must also implement special behavior!

- **User-defined field access policies** can be added

```cpp
local static bit m = uvm_reg_field::define_access(“UDAP”);
if(!uvm_reg_field::define_access(“UDAP”)) `uvm_error(...)```

```
Hooks & Callbacks

• Field base class has empty virtual method hooks
  – implement in derived field to specialize behavior

```
class my_reg_field extends uvm_reg_field;
    virtual task post_write(item rw);
    // specific implementation
    endtask
```

• Callback base class has empty virtual methods
  – implement in derived callback & register it with field

```
class my_field_cb extends uvm_reg_cbs;
    function new(string name, ...);
    virtual task post_write();
    // specific implementation
    endtask

my_field_cb my_cb = new("my_cb", ...);
    uvm_reg_field_cb::add(regX.fieldY, my_cb);
```

most important callback for passive operation is `post_predict`
Hook & Callback Execution

- Field method **hooks** are **always** executed
- **Callback** methods are **only** executed if **registered**

```plaintext
task uvm_reg_field::do_write(item rw);
...
  rw.local_map.do_write(rw);
  ...
  post_write(rw);
for (uvm_reg_cbs cb=cbs.first();
  cb!=null;
  cb=cbs.next())
  cb.post_write(rw);
...
endtask
```

- callbacks registered with field using **add**
- multiple callbacks can be **registered** with field
- callback methods executed in **cbs queue order**
Write-to-Reset Example

- Example user-defined field access policy
  - pre-defined access policies for Write-to-Clear/Set (WC, WS)
  - user-defined policy required for Write-to-Reset (WRES)

```
_uvm_reg_field::define_access("WRES")
```

- Demonstrate three possible solutions:
  - `post_write hook` implementation in derived field
  - `post_write` implementation in callback
  - `post_predict` implementation in callback
WRES Using *post_write* Hook

```vhdl
class wres_field_t extends uvm_reg_field;
...
virtual task post_write(uvm_reg_item rw);
  if (!predict(rw.get_reset())) `uvm_error
endtask

DERIVED FIELD
IMPLEMENT *post_write* TO SET MIRROR TO RESET VALUE

FIELD CREATED IN REG::BUILD
REGISTER CREATED IN BLOCK::BUILD

REG NOT PASSIVE
```

```vhdl
class wres_reg_t extends uvm_reg;
rand wres_field_t wres_field;
...
function void build();
  // wres_field create()/configure(.."WRES"..)

USE DERIVED FIELD

class my_reg_block extends uvm_reg_block;
rand wres_reg_t wres_reg;
...
function void build();
  // wres_reg create()/configure()/build()/add_map()

REG NOT PASSIVE
```

reg/block *build() is not* component *build_phase()"
WRES Using post_write Callback

```cpp
class wres_field_cb extends uvm_reg_cbs;
...
virtual task post_write(uvm_reg_item rw);
if (!predict(rw.get_reset()))
  uvm_error(...)
endtask
```

**DERIVED CALLBACK**

```cpp
class wres_reg_t extends uvm_reg;
rand uvm_reg_field wres_field;
...
function void build();
  // wres_field create()/configure("WRES")
```

**NOT PASSIVE**

```cpp
class my_reg_block extends uvm_reg_block;
rand wres_reg_t wres_reg;
...
function void build();
  // wres_reg create()/configure()/build()/add_map()
  wres_field_cb wres_cb = new("wres_cb");
  uvm_reg_field_cb::add(wres_reg.wres_field, wres_cb);
```

**USE BASE FIELD**

**IMPLEMENT post_write TO SET MIRROR TO RESET VALUE**

**CONSTRUCT CALLBACK**

**REGISTER CALLBACK WITH REQUIRED FIELD**
WRES Using `post_predict` Callback

```vhd
class wres_field_cb extends uvm_reg_cbs;
  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
IMPLEMENT post_predict TO
SET MIRROR VALUE TO RESET STATE
```

```vhd
PASSIVE OPERATION
```

```vhd
class wres_reg_t extends uvm_reg;
  rand uvm_reg_field wres_field;
  ... function void build();
    // wres_field create() /configure() /build() /add_map()
  wres_field_cb wres_cb = new("wres_cb");
  uvm_reg_field_cb::add(wres_reg.wres_field, wres_cb);
endfunction
```

*post_predict* is only available for fields *not* registers

```vhd
if we use this callback with a register we get silent non-operation!
```

```vhd
if(kind==UVM_PREDICT_WRITE) value = fld.get_reset();
```

```vhd
virtual function void build();
  // wres_field create() /configure() /build() /add_map()
endfunction
```

```vhd
PASSIVE OPERATION
```

```vhd
class my_reg_block extends uvm_reg_block;
  rand wres_reg_t wres;
  ... function void build();
    // wres_reg create() /configure() /build() /add_map()
endfunction
```
Lock/Protect Example

- Example register field interaction
  - protected field behavior based on state of lock field, or
  - lock field operation modifies behavior of protected field

- Demonstrate two possible solutions:
  - post_predict implementation in callback
  - dynamic field access policy controlled by callback
  - (not bad pre_write implementation from UVM User Guide)
class prot_field_cb extends uvm_reg_cbs;
    local uvm_reg_field lock_field;

function new (string name, uvm_reg_field lock);
    super.new (name);
    this.lock_field = lock;
endfunction

virtual function void post_predict(.
previous,
value);
    if (kind == UVM_PREDICT_WRITE)
        if (lock_field.get())
            value = previous;
endfunction

class my_reg_block extends uvm_reg_block;
    prot_field_cb prot_cb = new("prot_cb", lock_field);
    uvm_reg_field_cb:::add prot_field, prot_cb);

HANDLE TO LOCK FIELD
REVERT TO PREVIOUS VALUE IF LOCK ACTIVE
CONNECT LOCK FIELD
REGISTER CALLBACK WITH PROTECTED FIELD
class lock_field_cb extends uvm_reg_cbs;

local uvm_reg_field prot_field;

function new (string name, uvm_reg_field prot);
  super.new (name);
  this.prot_field = prot;
endfunction

virtual function void post_predict(...);
  if (kind == UVM_PREDICT_WRITE)
    if (value)
      void'(prot_field.set_access("RO"));
    else
      void'(prot_field.set_access("RW"));
  endfunction

class my_reg_block extends uvm_reg_block;
  lock_field_cb lock_cb = new("lock_cb", prot_field);
  uvm_reg_field_cb::add(lock_field, lock_cb);

Buffered Write Example

- Example *register field interaction*
  - *trigger* field *operation* effects *buffered* field behavior
- Demonstrate two possible solutions:
  - *overlapped register* implementation with *callback*
  - *derived buffer field* controlled by multiple *callbacks*
Buffered Write Using 2 Registers

```plaintext
class trig_field_cb extends uvm_reg_cb {
  local uvm_reg_field current, buffer;

  function new (string name, uvm_reg_field current, uvm_reg_field buffer);
  ...}

virtual function void post_predict (...);
if (kind == UVM_PREDICT_WRITE) begin
  uvm_reg_data_t val = buffer.get_mirrored_value();
  if (!current.predict(val))
    uvm_error(...);
}
```

```plaintext
class my_reg_block extends uvm_reg_block {
  ...

default_map.add_reg(cur_reg, 'h10, "RO");
default_map.add_reg(buf_reg, 'h10, "WO");

trig_field_cb trig_cb = new("trig_cb", cur_reg.cur_field, buf_reg.buf_field);
trig_field_cb::add(trig_field, trig_cb);
```

- **copy from buffer to current on write to trigger**
- **ro & wo register at same address**
  - all writes go to WO buffer
  - all reads come from RO current
- **register callback with trigger field**
- **cannot share address again**
- **complicated to generate**
- **confusing map for user**

Handles to both current & buffer fields
Buffered Write Using Derived Field

class buf_reg_field extends uvm_reg_field;
  uvm_reg_data_t buffer;
  virtual function void reset(string kind);
      super.reset(kind);
      buffer = get_reset(kind);
  endfunction

class buf_field_cb extends uvm_reg_cbs;
  local buf_reg_field buf_field;
  virtual function void post_predict(...); // if write
      buf_field.buffer = value;
      value = previous;
  endfunction

class trig_field_cb extends uvm_reg_cbs;
  local buf_reg_field buf_field;
  virtual function void post_predict(...); // if write
      buf_field.predict(buf_field.buffer);
  endfunction

buf_field_cb buf_cb = new("buf_cb", buf_field);
buf_reg_field_cb::add(buf_field, buf_cb);
trig_field_cb trig_cb = new("trig_cb", buf_field);
buf_reg_field_cb::add(trig_field, trig_cb);

ADD BUFFER TO DERIVED FIELD
RESET BUFFER TO FIELD RESET VALUE
_post_predict callback required for passive
SET BUFFER TO VALUE ON WRITE TO FIELD, SET MIRROR TO PREVIOUS (UNCHANGED)
COPY BUFFER TO MIRROR ON WRITE TO TRIGGER
REGISTER CALLBACKS WITH BUFFERED & TRIGGER FIELDS
Register Side-Effects Example

• Randomize or modify registers & reconfigure DUT
  – what about UVC configuration?
    • update from register sequences
    • snoop on DUT bus transactions
    • implement post_predict callback

`side_effect_cb`
```c
if (field.write(val))
    cfg.set_var(val);
```

`callback` registered with model field

access UVC config via a `handle`

not passive
not backdoor
passive & backdoor
class reg_cfg_cb extends uvm_reg_cbs;
  my_config cfg;

function new (string name, my_config cfg);
  super.new (name);
  this.cfg = cfg;
endfunction

virtual function void post_predict(...);
  if (kind == UVM_PREDICT_WRITE)
    cfg.set_var(my_enum_t'(value));
endfunction

class my_env extends uvm_env;
...
  uvc = my_uvc::type_id::create(...);
  reg_model = my_reg_block::type_id::create(...);
...
  reg_cfg_cb cfg_cb = new("cfg_cb", uvc.cfg);
  uvm_reg_field_cb::add(reg_model.reg.field, cfg_cb);
Performance

• Big register models have **performance impact**
  – full SoC can have >10k fields
• Register model & RTL typically auto-**generated**
  – made-to-measure for each device derived

**REGISTER DESCRIPTION**
(TEXT, XML, YAML, etc.)

**GENERATOR TOOL/SCRIPTS**
Life Without The Factory

- Example **SoC** with **14k+ fields in 7k registers**
  - many **register classes** (most fields are base type)
  - **not using factory** overrides – **generated** on demand

<table>
<thead>
<tr>
<th>MODE</th>
<th>FACTORY TYPES</th>
<th>COMPILATE TIME</th>
<th>LOAD TIME</th>
<th>BUILD TIME</th>
<th>DISK USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO REGISTER MODEL</td>
<td>598</td>
<td>23</td>
<td>9</td>
<td>1</td>
<td>280M</td>
</tr>
<tr>
<td>+REGISTERS USING FACTORY</td>
<td>8563</td>
<td>141</td>
<td>95</td>
<td>13</td>
<td>702M</td>
</tr>
<tr>
<td>+REGISTERS NO FACTORY</td>
<td>784</td>
<td>71</td>
<td>17</td>
<td>1</td>
<td>398M</td>
</tr>
</tbody>
</table>

- Register model still **works without** the **factory**
  - **do not use** `uvm_object_utils` **macro** for fields & registers
  - **construct** registers using `new` instead of `type_id::create`
\texttt{\textbackslash uvm\_object\_utils}

```vhdl
\`define uvm\_object\_utils(T) \ 
  `m_uvm\_object\_registry\_internal(T,T) \ 
  class \texttt{my\_reg} extends \texttt{uvm\_reg}; 
  `uvm\_object\_utils(my\_reg) 
  endclass 
```

```vhdl
class \texttt{my\_reg} extends \texttt{uvm\_reg}; 
  typedef \texttt{uvm\_object\_registry} \#(\texttt{my\_reg},"my\_reg") \texttt{type\_id}; 
  static function \texttt{type\_id} get\_type(); 
  \texttt{return} \texttt{type\_id::get();} 
endfunction 
function \texttt{uvm\_object} create (\texttt{string} name=""); 
const static \texttt{string} \texttt{type\_name} = "my\_reg"; 
\texttt{virtual function \texttt{string} get\_type\_name ()}; 
\texttt{return \texttt{type\_name;}} 
endfunction 
```
class uvm_object_registry
    #(type T, string Tname) extends uvm_object_wrapper

typedef uvm_object_registry #(T,Tname) this_type;

local static this_type me = get();

static function this_type get();
    if (me == null) begin
        uvm_factory f = uvm_factory::get();
        me = new;
        f.register(me);
    end
    return me;
endfunction

virtual function uvm_object create_object(...);

static function T create(...);

static function void set_type_override(type, replace);

static function void set_inst_override(type, inst, parent);
endclass

• thousands of registers means thousands of proxy classes are constructed and added to factory when files loaded
• do not need these classes for register generator use-case!
type_id::create

```plaintext
reg= my_reg::type_id::create("reg",,get_full_name());
```

**class uvm_object_registry #(T, Tname) extends uvm_object_wrapper;**

```plaintext
... static function T create(name,parent,contxt="");
  uvm_object obj;
  uvm_factory f = uvm_factory::get();
  obj = f.create_object_by_type(get(),contxt,name,parent);
  if (!$cast(create, obj)) uvm_report_fatal(...);
endfunction

virtual function uvm_object create_object (name,parent);
  T obj;
  obj = new(name, parent);
  return obj;
endfunction
... endclass
```

**uvm_object_registry #(my_reg,"my_reg")**

- **static create function**
- **request factory create based on existing type overrides (if any)**
- **return handle to object**
- **search queues for overrides**
- **constructs actual object**

- **create** and factory **search** takes time for **thousands of registers**
  during the **pre-run phase** for the environment (**build time**)  
- **no need** to search for overrides for **register generator** use-case!
Conclusions

• There’s more than one way to skin a reg...
  – but some are better than others!
  – consider: passive operation, backdoor access, use-cases,...
• Full-chip SoC register model performance impact
  – for generated models we can avoid using the factory
• All solutions evaluated in UVM-1.1d & OVM-2.1.2
  – updated uvm_reg_pkg that includes UVM-1.1d bug fixes
    (available from www.verilab.com )

mark.litterick@verilab.com