DESIGN & VERIFICATION DVCON 2014 CONFERENCE & EXHIBITION

Advanced UVM Register Modeling

There's More Than One Way to Skin A Reg

Mark Litterick
Verilab GmbH, Munich, Germany

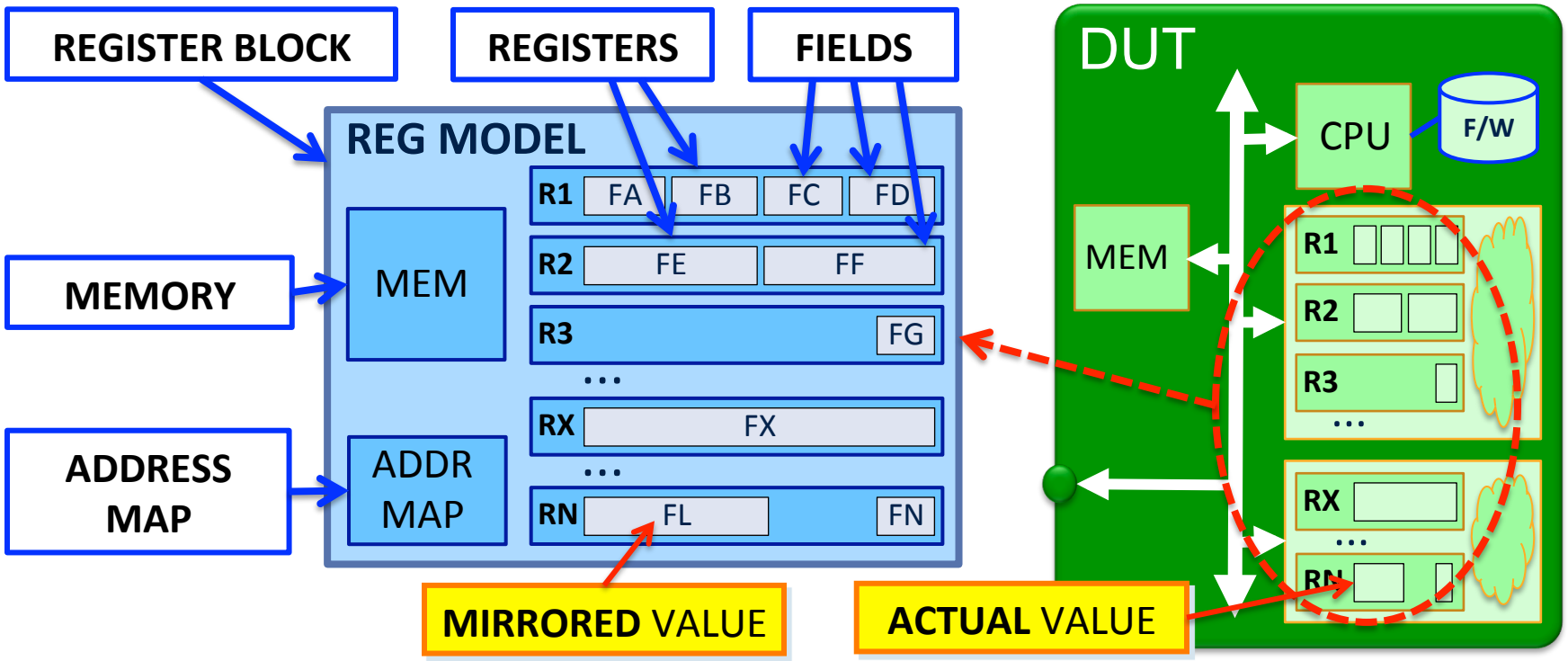
verilab

Introduction

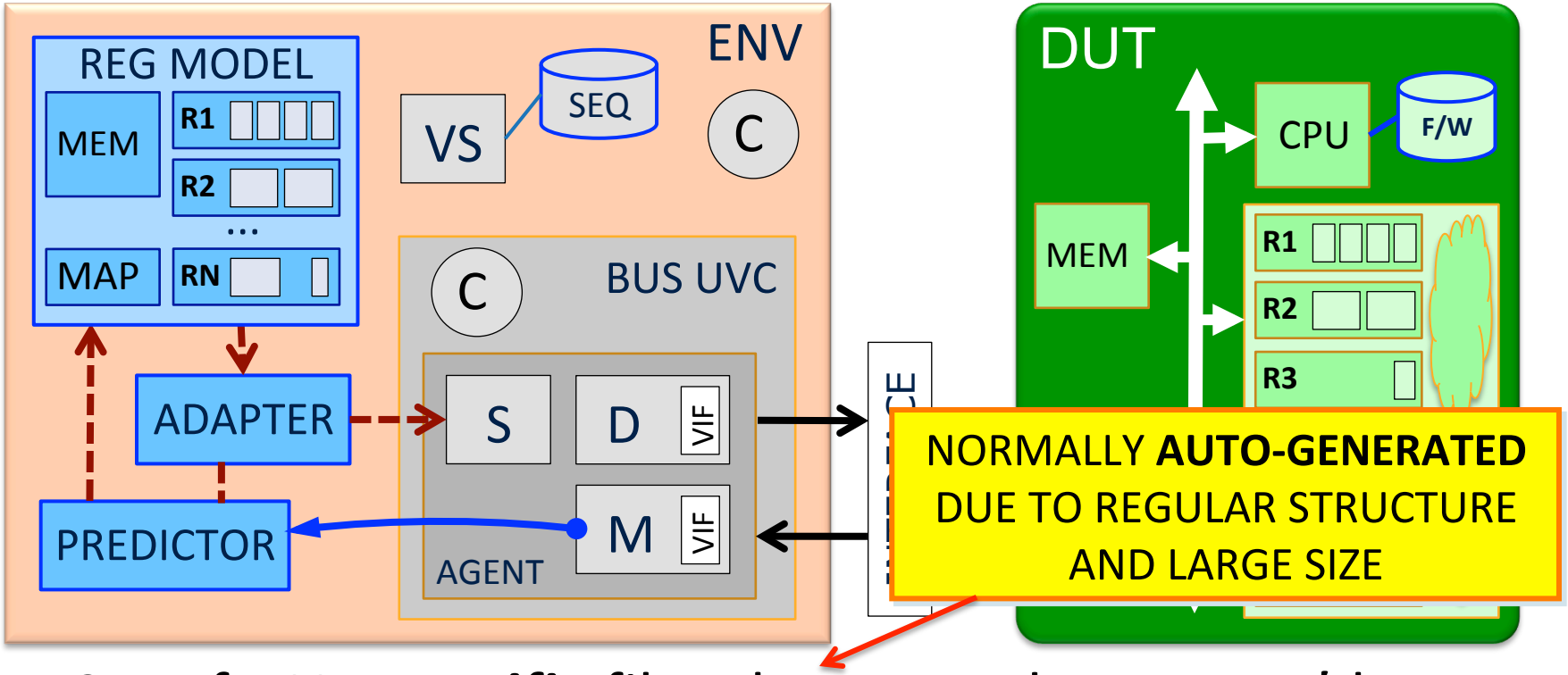
- UVM register model **overview**
  - structure, integration, concepts & operation
  - field modeling, access policies & interaction
  - behavior modification using hooks & callbacks
- Modeling **examples**
  - worked examples with multiple solutions illustrated
  - field access policies, field interaction, 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**

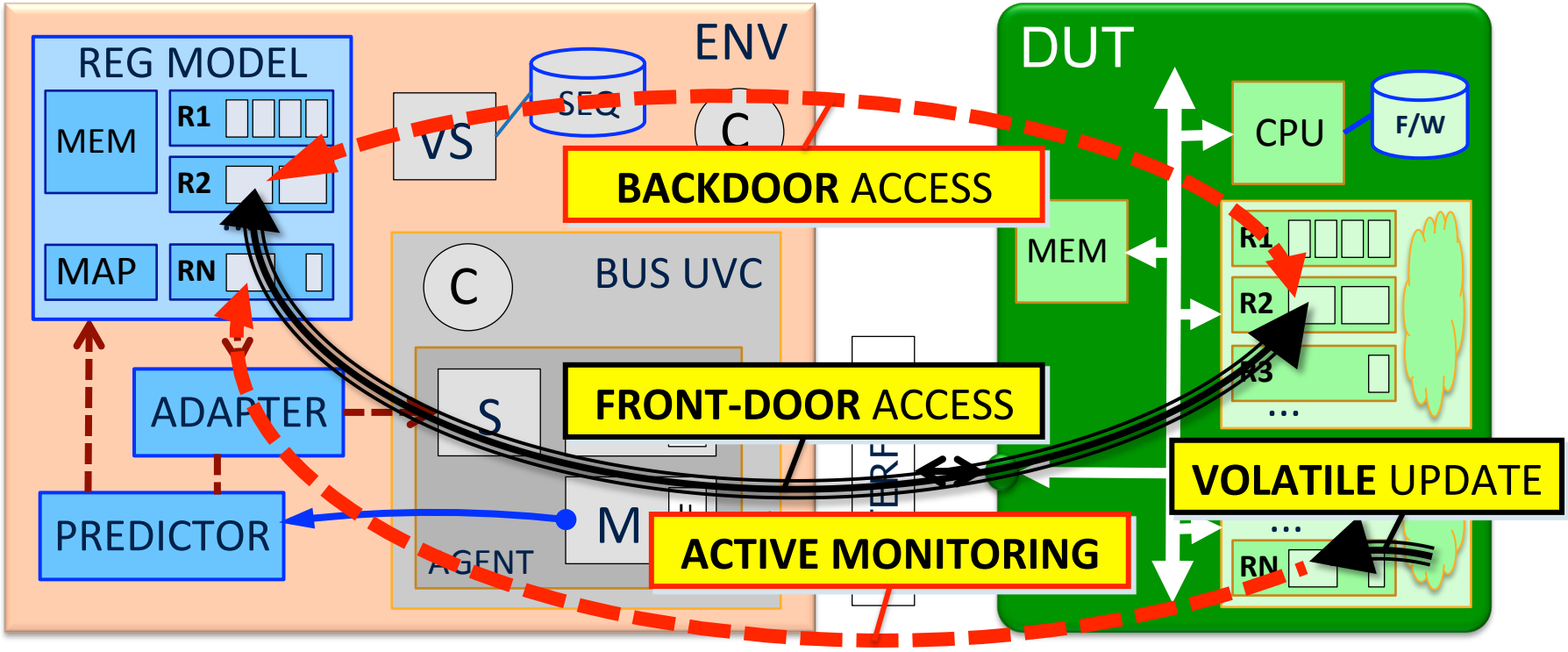


Register Model Integration



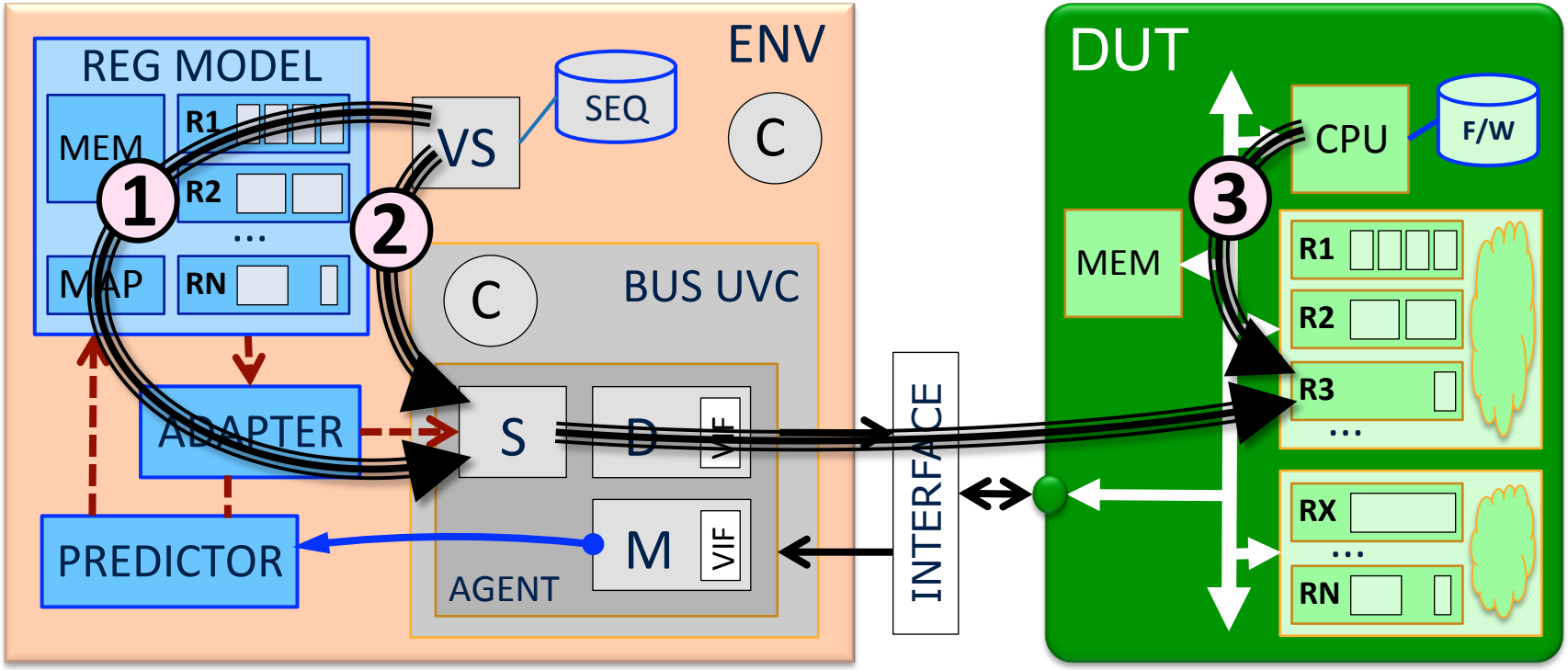
- Set of **DUT-specific** files that extend ***uvm_reg**** base
- Instantiated in *env* alongside bus interface UVCs
  - ***adapter*** converts generic *read*/*write* to bus transactions
  - ***predictor*** updates model based on observed transactions

Register Model Concepts



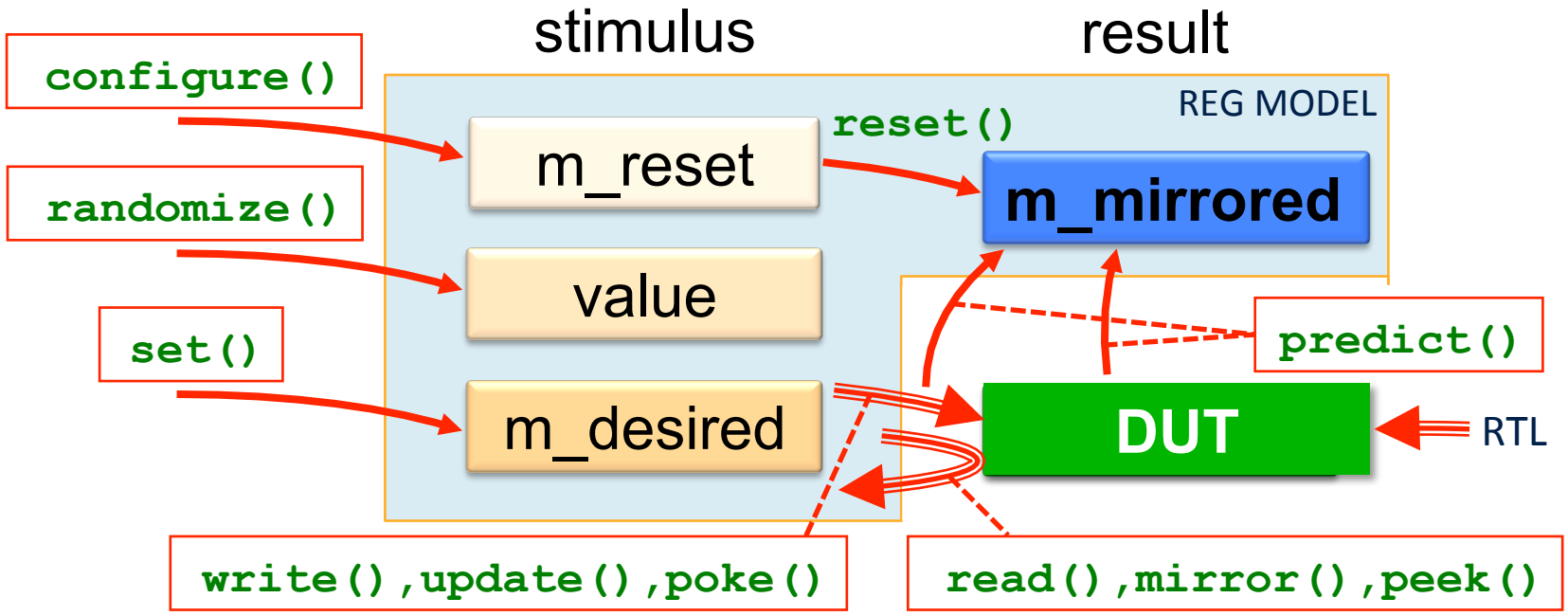
- Normal **front-door** access **via bus transaction** & **I/F**
  - sneaky **backdoor** access via ***hdl_path*** - no bus transaction
- **Volatile** fields modified by non-bus RTL functionality
  - model updated using **active monitoring** via ***hdl_path***

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

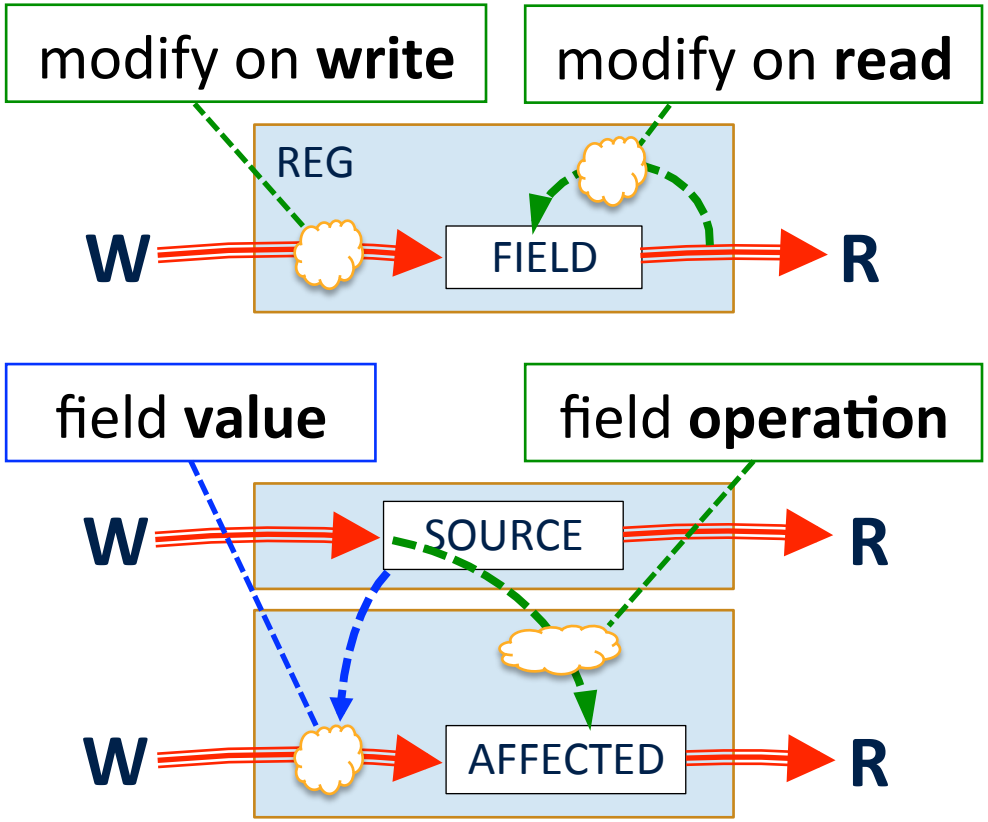
Register Access API



- 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**

	NO WRITE	WRITE VALUE	WRITE CLEAR	WRITE SET	WRITE TOGGLE	WRITE ONCE
NO READ	-	WO	WOC	WOS	-	WO1
READ VALUE	RO	RW	WC W1C W0C	WS W1S W0S	W1T W0T	W1
READ CLEAR	RC	WRC	-	WSRC W1SRC W0SRC		
READ SET	RS	WRS	WCRS W1CRS W0CRS	-		

Just **defining** access policy is ***not enough***!

Must also **implement special behavior**!

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

```
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
```

pre_write
post_write
pre_read
post_read

- **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_wr
    // specific implem
  endtask
```

most **important** callback for **passive** operation is ***post_predict***

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

pre_write
post_write
pre_read
post_read
post_predict
encode
decode

Hook & Callback Execution

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

```
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
```

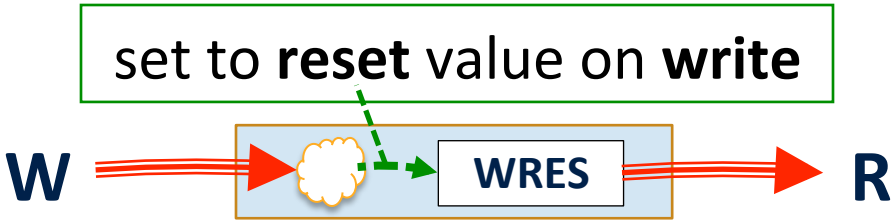
ACTUAL **WRITE**

HOOK METHOD

CALLBACK METHOD
FOR ALL **REGISTERED CBS**

- **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

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

DERIVED **FIELD**

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

NOT PASSIVE

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

USE **DERIVED** FIELD

FIELD CREATED IN **REG::BUILD**

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

REGISTER CREATED IN **BLOCK::BUILD**

reg/block **build()** is **not** component **build_phase()**

WRES Using *post_write* Callback

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

DERIVED **CALLBACK**

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

NOT PASSIVE

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

USE **BASE** 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()
  wres_field_cb wres_cb = new("wres_cb");
  uvm_reg_field_cb::add(wres_reg.wres_field, wres_cb);
```

CONSTRUCT CALLBACK

REGISTER CALLBACK WITH REQUIRED **FIELD**

WRES Using *post_predict* Callback

```
class wres_field_cb extends uv
 ...
 virtual function void post_predict(..,fld,value,..);
  if(kind==UVM_PREDICT_WRITE) value = fld.get_reset();
```

IMPLEMENT *post_predict* TO SET MIRROR VALUE TO RESET STATE

PASSIVE OPERATION

```
class wres_reg_t exte
 rand uvm_reg_field w
 ...
 function void build(
  // wres_field creat
```

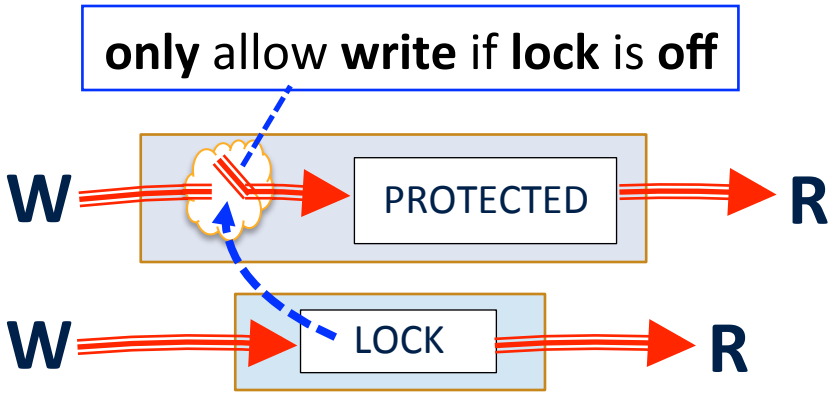
```
virtual function void post_predict(
  input  uvm_reg_field   fld,
  input  uvm_reg_data_t  previous,
  inout  uvm_reg_data_t  value,
  input  uvm_predict_e   kind,
  input  uvm_path_e      path,
  input  uvm_reg_map     map
```

```
class my_reg_bloc
 rand wres_reg_t
 ...
 function void bu
  // 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);
```

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

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

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*)

Lock Using *post_predict* Callback

```
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
```

HANDLE TO **LOCK** FIELD

REVERT TO **PREVIOUS VALUE** IF **LOCK ACTIVE**

```
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);
```

CONNECT LOCK FIELD

REGISTER CALLBACK WITH **PROTECTED** FIELD

Lock Using Dynamic Access Policy

```
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"));
 en
```

```
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);
```

HANDLE TO **PROTECTED** FIELD

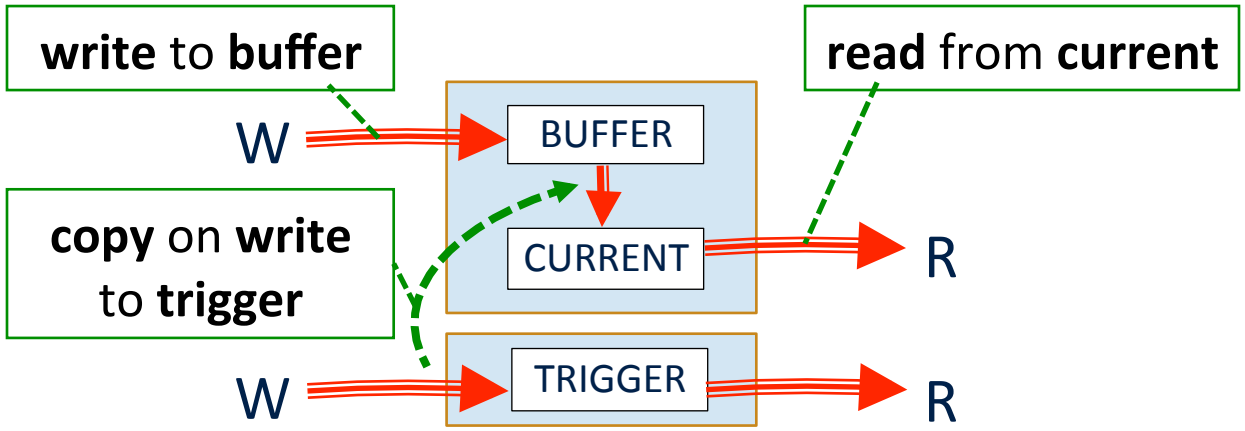
SET **ACCESS POLICY** FOR **PROTECTED** FIELD BASED ON **LOCK OPERATION**

prot_field.get_access() RETURNS CURRENT POLICY

REGISTER CALLBACK WITH **LOCK** FIELD

CONNECT PROTECTED FIELD

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

```
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_predi
  if (kind == UVM_PREDICT_WRITE) begin
   uvm_reg_data_t val = buffer.get_mirrored_value();
   if (!current.predict(val))
```

HANDLES TO **BOTH CURRENT** & **BUFFER** FIELDS

COPY FROM **BUFFER** TO **CURRENT** ON WRITE TO **TRIGGER**

```
class my_reg_block extends uvm
 ...
 default_map.add_reg(cur_reg, 'h10, "RO");
 default_map.add_reg(buf_reg, 'h10, "WO");

                                = ne
                                cur_field, buf_reg.buf_field);
                                trig_field, trig_cb);
```

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

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);
```

ADD **BUFFER** TO DERIVED FIELD

RESET BUFFER TO FIELD RESET VALUE

```
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;
```

***post_predict* callback** required for **passive**

SET **BUFFER** TO **VALUE** ON **WRITE** TO FIELD, SET **MIRROR** TO **PREVIOUS** (UNCHANGED)

```
class trig_field_cb extends uvm_reg_cbs;
 local buf_reg_field buf_field;
 virtual function void post_predict(...
  buf_field.predict(buf_field.buffer);
```

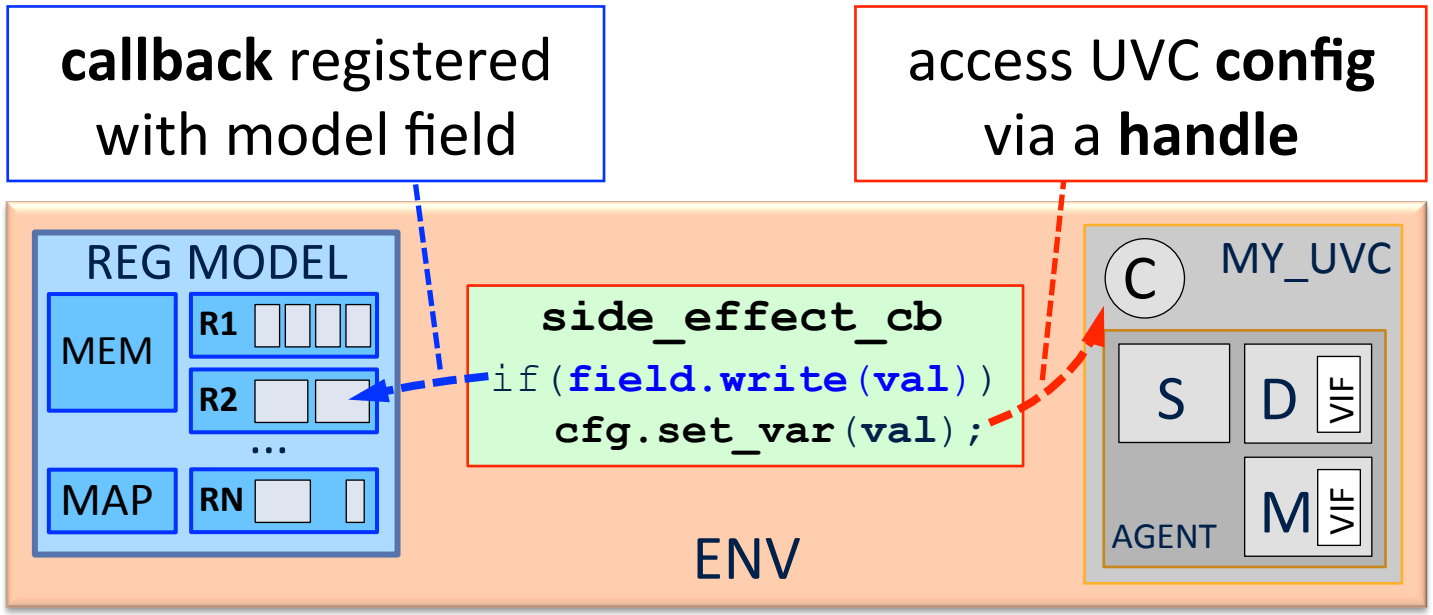
COPY BUFFER TO **MIRROR** ON **WRITE** TO **TRIGGER**

```
buf_field_cb buf_cb = new("buf_cb",b
uvm_reg_field_cb::add(buf_field, buf_
trig_field_cb trig_cb = new("trig_cb",buf_field);
uvm_reg_field_cb::add(trig_field, trig_cb);
```

REGISTER CALLBACKS WITH **BUFFERED** & **TRIGGER** FIELDS

Register Side-Effects Example

- Randomize or modify registers & reconfigure DUT
  - what about **UVC configuration?**
    - update from **register sequences** — not passive
    - **snoop** on DUT bus transactions — not backdoor
    - implement ***post_predict*** **callback** — passive & backdoor



Config Update Using Callback

```
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
```

HANDLE TO **CONFIG** OBJECT

SET **CONFIG** ON **WRITE** TO REGISTER FIELD (TRANSLATE IF REQUIRED)

```
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);
```

ENVIRONMENT HAS **UVC** & **REG_MODEL**

CONNECT CONFIG

REGISTER CALLBACK

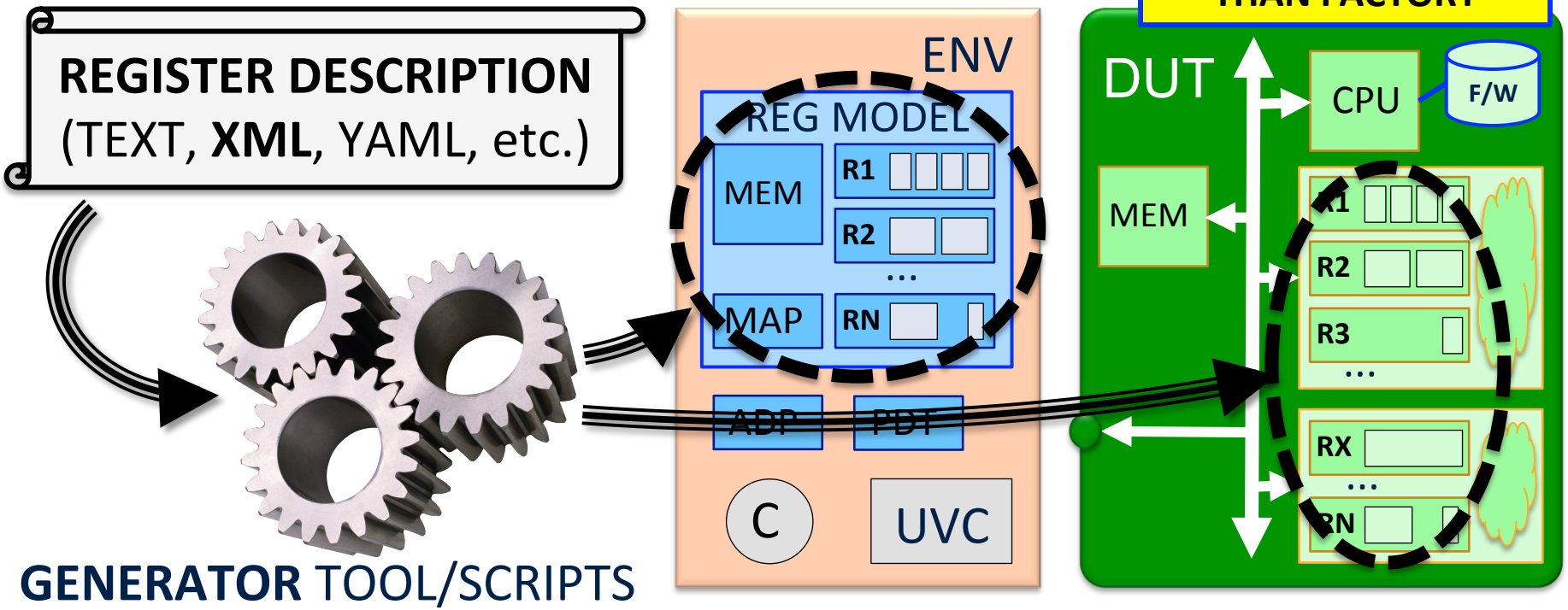
Performance

- Big register models have **performance impact**
  - full SoC can have >10k fields

MANY REGISTER CLASSES
(MORE THAN REST OF ENV)

- Register model & RTL typically auto-**generated**
  - **made-to-measure** for each device de

DIFFERENT USE-CASE THAN FACTORY



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

MODE	FACTORY TYPES	COMPILE TIME	LOAD TIME	BUILD TIME	DISK USAGE
NO REGISTER MODEL	598	23	9	1	280M
+REGISTERS USING FACTORY	8563	141	95	13	702M
+REGISTERS NO FACTORY	784	71	17	1	398M

COMPILE TIME x2
+1 min infrequently

LOAD + BUILD TIME x5
+1.5 min for every sim

- 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***

`uvm_object_utils

```
`define uvm_object_utils(T) \
  `m_uvm_object_registry_internal(T,T) \
```

```
class my_reg extends uvm_reg;
  `uvm_object_utils(my_reg)
endclass
```

```
`define m_uvm_object_registry_internal(T,S) \
```

```
class my_reg extends uvm_reg;
  typedef uvm_object_registry #(my_reg,"my_reg") type_id;
  static function type_id get_type();
    return type_id::get();
  endfunction
  function uvm_object create (strin
  const static string type_name = "my_reg";
  virtual function string get_type_name ();
    return type_name;
```

declare a **typedef** specialization of **uvm_object_registry** class

explains what **my_reg::type_id** is

but what about **factory registration** and **type_id::create ???**

declare some **methods** for factory API

uvm_object_registry

```
class uvm_object_registry
      #(type T, string Tname) extends uvm_
  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
  virtu
  stati
  stati
  stati
endcla
```

proxy type
lightweight substitute for real object

local **static proxy** variable calls **get()**

construct instance of proxy, not real class

register proxy with factory

registration is via **static initialization**
=> happens at **simulation load time**

```
function void uvm_factory::register (uvm_object_wrapper obj);
  ...
  // add to associative arrays
  m_type_names[obj.get_type_name()] = obj;
```

- **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

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

`uvm_object_registry #(my_reg,"my_reg")`

static **create** function

```
class uvm_object_registry #(T, Tname) extends uvm_object_wrapper;
  ...
  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(...);
  endfunc
  virtual function uvm_object create_object (name,parent);
    T obj;
    obj = new(name, parent);
    re
  end
  ...
end
```

```
... uvm_factory::create_object_by_type
  (type,context,name,parent);
```

request factory **create** based on existing type overrides (if any)

return **handle** to object

constructs **actual** object

search queues for **overrides**

- **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