The UVM Register Layer
Introduction, Experiences and Recipes

Steve Holloway – Principal Verification Engineer

Dialog Semiconductor

Overview

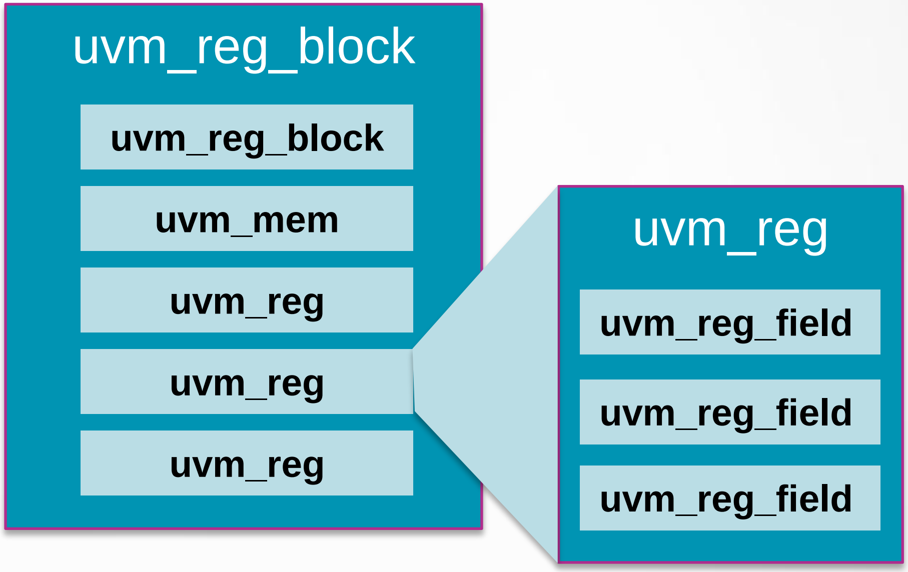
- Register Model Requirements
- UVM Register Layer
- Creating the Register Model
- Register Modeling Recipes
- Conclusions

Register Model Requirements

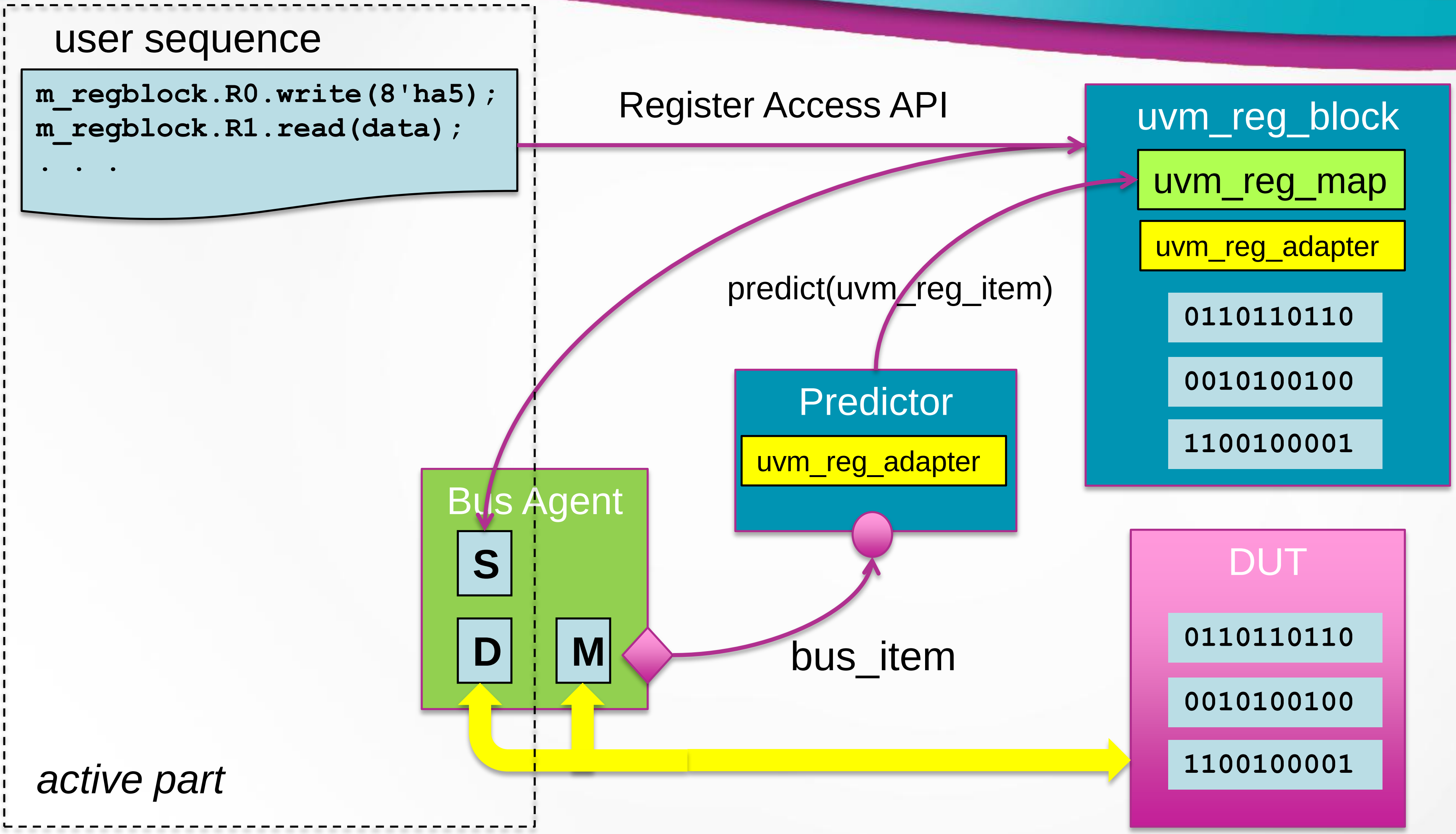
- *A **standard** modeling approach*
  - Previous use of internal register models and OVM_RGM
  - Transition to UVM within Dialog
- *Distributed register blocks*
  - Register block per IP
  - Access & update on a per-field basis
- *Non-standard "quirky" registers – e.g.*
  - Locking - dependent on bus master
  - "Snapshot" - coherency between registers
  - Interrupt / event generation
- *Passive update of register model*
  - Re-use in other (directed) environments
- *Automated generation*

UVM Register Layer (UVM_REG)

- Abstract model for registers and memories in DUT
  - Maintains a "mirror" of the DUT registers
- Hierarchy analogous to DUT:
  - Register Block
  - Register File
  - Memory
  - Register
  - Field
- Standardised register access API
  - Address-independent instance/string names
- Address maps
  - model access via a specific interface / bus master



Register Model Components



```
m_regblock.R0.write(8'ha5);
m_regblock.R1.read(data);
. . .
```

Register Access API

write()

- Generate a physical write to the DUT register

read()

- Generate a physical read from the DUT register

set()

- Set the *desired* value of the register in the model

get()

- Get the *desired* value of the register from the model

update()

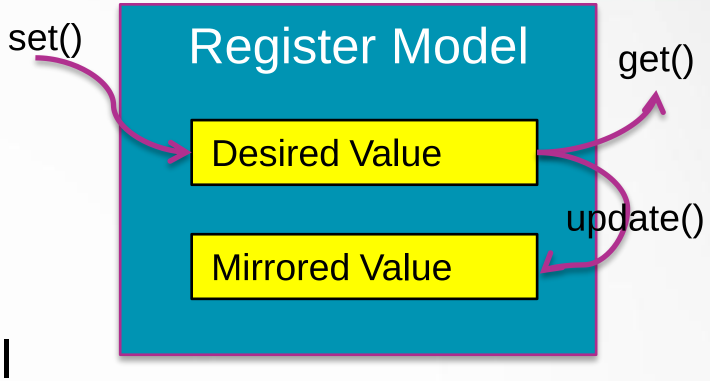
- Update the DUT with the desired value in the model

mirror()

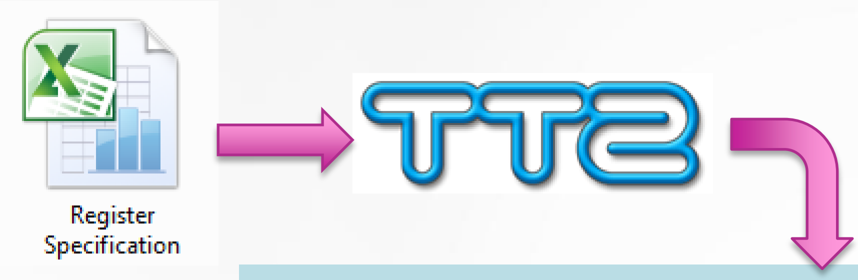
- Read the DUT register and check / update the model value

predict()

- Set the value of the register in the model



Creating the Register Model



field configuration

```
function void configure(uvm_reg          parent,
                        int unsigned     size,
                        int unsigned     lsb_pos,
                        string           access,
                        bit              volatile,
                        uvm_reg_data_t   reset,
                        bit              has_reset,
                        bit              is_rand,
                        bit individually_accessible)
```

```
class SYS_CTRL_0 extends uvm_reg;

  rand uvm_reg_field SYS_CONF0;
  rand uvm_reg_field SYS_CONF1;
  rand uvm_reg_field SYS_STATUS0;

  virtual function void build();
    SYS_CONF0 = uvm_reg_field::type_id::create("SYS_CONF0");
    SYS_CONF0.configure (this, 1,  0, "RW",  0, 1'h0, 1, 1, 1);

    . . .

  endfunction

  function new(string name = "SYS_CTRL_0");
    super.new(name,16,build_coverage(UVM_NO_COVERAGE));
  endfunction

  `uvm_object_utils(SYS_CTRL_0)

endclass: SYS_CTRL_0
```

hierarchical build() of regmodel

Handling "Quirky" Registers

- UVM_REG gives simple model for "free"
  - "Dumb" storage subject to 1 of 25 pre-defined access policies:
  - {"RW", "RC", "RS", "WRC", . . .}
  - Most of our registers are more *quirky* than this
- More complex behaviour is handled by callbacks
  - Pre-defined "hook" methods called during model update
    - pre_read()
    - post_read()
    - pre_write()
    - post_write()
    - post_predict()
- Using "passive" prediction, important to use post_predict() hook

Simple Callback – "Control" field

- Set

  Write 0 1 → 0 1

- Clear

  Write 1 0 → 0 0

```
class control_reg_field_cbs 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);

    if (kind == UVM_PREDICT_WRITE)
      value = (value === 2'b01) ? 2'b01 :
              (value === 2'b10) ? 2'b00 : previous;

  endfunction

endclass: control_reg_field_cbs
```

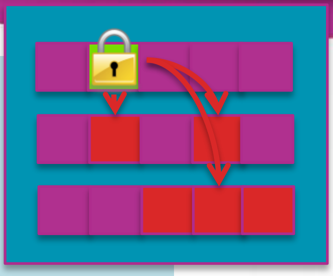
After predictor has observed R/W

value before predict()

value to be set after predict()

UVM_PREDICT_READ
UVM_PREDICT_WRITE
UVM_PREDICT_DIRECT

Callback for Locking Behaviour



```
class lock_reg_field_cbs extends uvm_reg_cbs;

  string m_lock_name;

  `uvm_object_utils(lock_reg_field_cbs)

  function new(string name = "lock_reg_field_cbs", string lock_name = "");
    super.new(name);
    m_lock_name = lock_name;
  endfunction: new

  virtual function void post_predict(. . .);
    . . .
    if (kind == UVM_PREDICT_WRITE) begin
      lock_fld = m_reg_block.get_field_by_name(m_lock_name);
      if (lock_field.get())
            value = previous;
  endfunction: post_predict
```

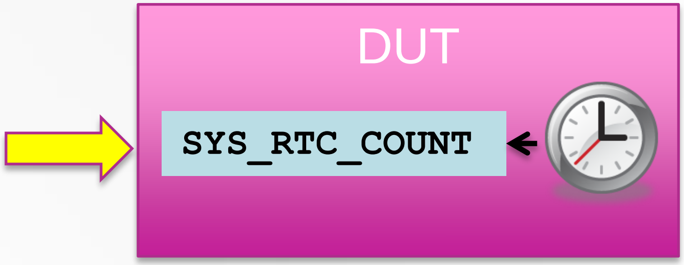
Locking field name in constructor

Extract the lock field from block

If locked, revert to previous

Modeling Status Registers

- RTC Counter Field declared as "RO"



```
task my_scoreboard::update_rtc_count;
  int unsigned cnt = 0;

  forever begin
    @(timer_event_e);
    void'(m_regmodel.SYS_RTC_COUNT.predict(cnt++));
  end

endtask: update_rtc_count
```

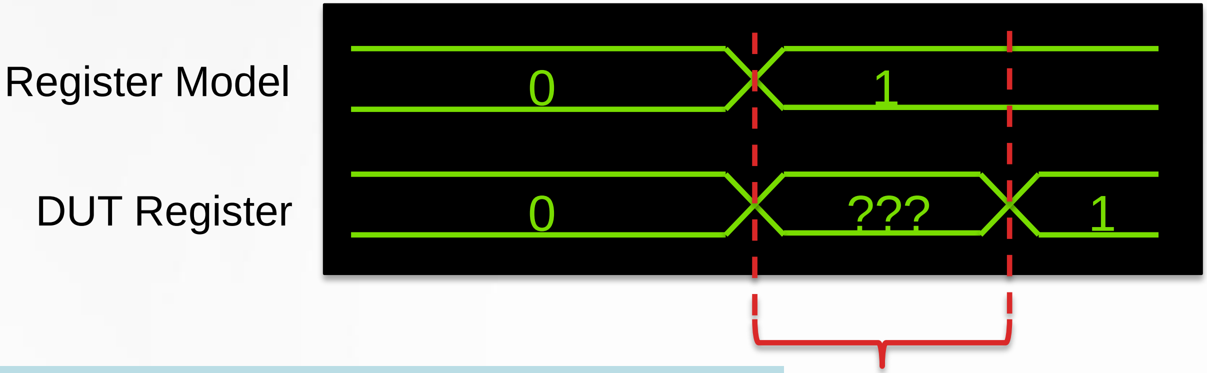
Does not use access policy "RO"

Possible to use access policy & callbacks

```
reg.predict(value, .kind(UVM_PREDICT_WRITE));
reg.predict(value, .kind(UVM_PREDICT_READ));
```

Handling DUT Uncertainty

- Sometimes we don't know exactly when a DUT register will change



```
task my_scoreboard::uncertainty_window;
  forever begin
    @(window_start_e);
    my_fld.set_compare(UVM_NO_CHECK);
    @(window_end_e);
    my_fld.set_compare(UVM_CHECK);
  end
endtask: uncertainty_window
```

Don't update if comparison is disabled

```
class my_reg_field extends uvm_reg_field;

function void do_predict(. . .);
  super.do_predict(rw, kind, be);
  if (kind == UVM_PREDICT_READ)
    if (get_compare() == UVM_NO_CHECK)
      value = previous;
endfunction: do_predict

endclass: my_reg_field
```

Using Extension Information

- Additional information can be attached to register access via "extension" object.

```
uvm_reg my_reg;
my_bus_info extra_info = new();
m_regmodel.get_reg_by_name("SYS_CTRL_0");
extra_info.master_id = HOST;
my_reg.write(status, data, .parent(this), .extension(extra_info));
```

Sending master_id with write()

- Adapter needs to use get_item() to access extension info

```
virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
    my_bus_info         extra_info;
    uvm_reg_item item = get_item();

    $cast(extra_info, item.extension);

    bus_trans.addr      = rw.addr;
    bus_trans.data      = rw.data;
    bus_trans.master_id = extra_info.master_id;

    return bus_trans;
endfunction: reg2bus
```

uvm_addr_map calls set_item()

extension is a uvm_object

Synchronising to Register Access

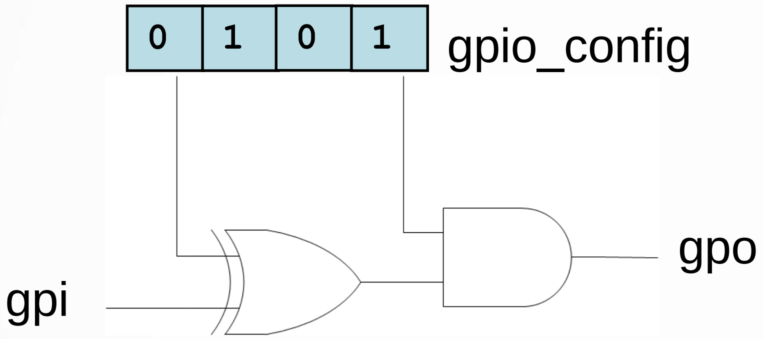
```
class trigger_reg_field_cbs extends uvm_reg_cbs;
  . . .
  virtual function void post_predict(. . .);
    uvm_event access_event;
    if (kind == UVM_PREDICT_WRITE) begin
      if (value != previous) begin
        access_event = uvm_event_pool::get_global($psprintf("%s_WRITE", fld.get_name()));
        access_event.trigger();
      end
    end
  endfunction
endclass: trigger_reg_field_cbs
```

Synchronise across env with uvm_event_pool

```
task my_scoreboard::model_timer();

  uvm_event ev_timer_start = uvm_event_pool::get_global("TIMER_START_WRITE");

  ev_timer_start.wait_trigger();

  . . .
endtask: model_timer
```

Using SVA with the Register Model

- GPIO – a function of input & register settings

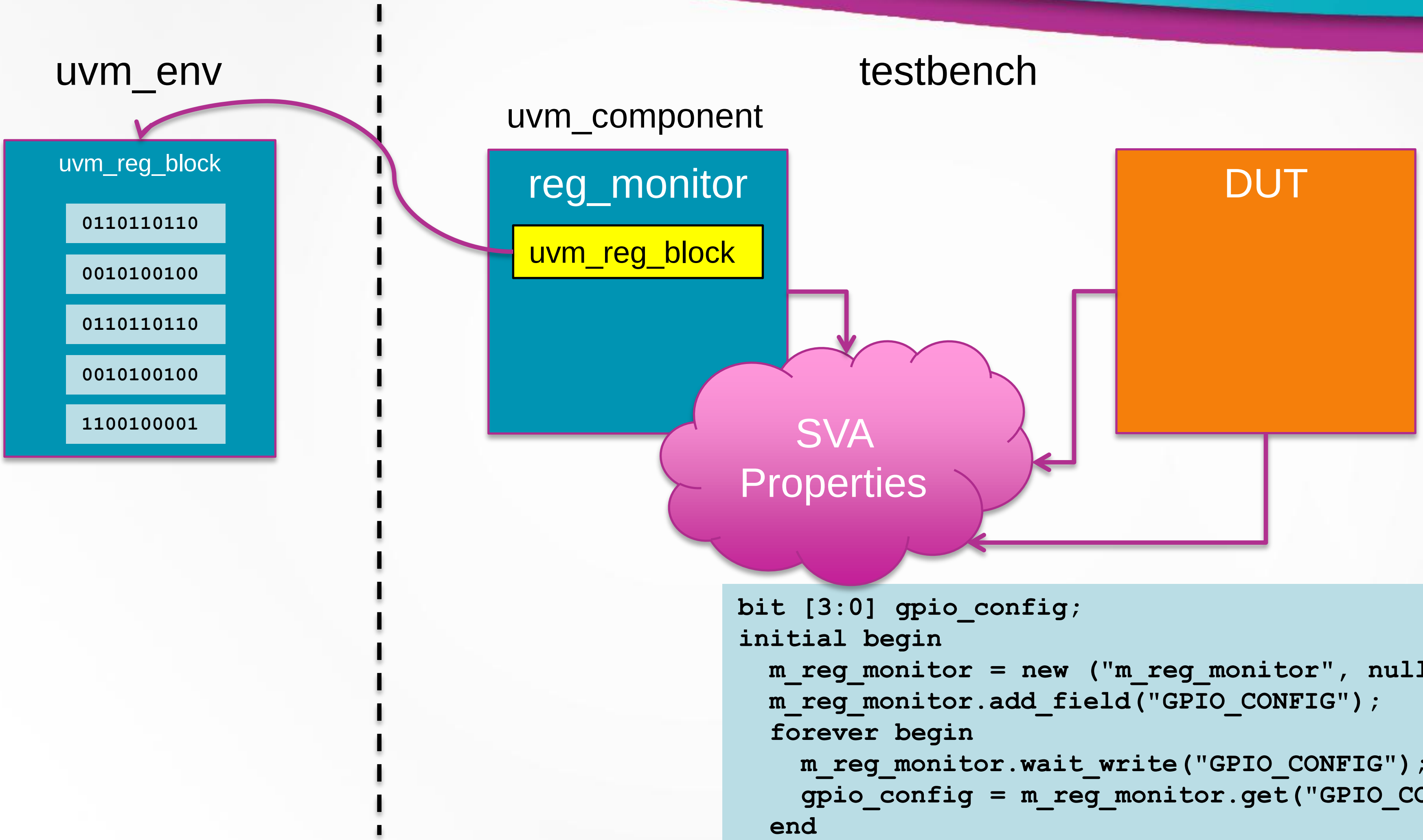


```
property gpio_control_p @(gpio_change_e)
   (gpo == (gpi ^ gpo_invert) & gpio_en);
endproperty
```

Can only live in module-based code

- *How can we "hook" the regmodel into a testbench?*

Hooking in the Regmodel



```
bit [3:0] gpio_config;
initial begin
  m_reg_monitor = new ("m_reg_monitor", null);
  m_reg_monitor.add_field("GPIO_CONFIG");
  forever begin
    m_reg_monitor.wait_write("GPIO_CONFIG");
    gpio_config = m_reg_monitor.get("GPIO_CONFIG");
  end
end
```

Conclusions

- Register Modeling Standard is key to UVM
  - Inconsistent approach with OVM
  - UVM_REG has all features necessary
- Early UVM 1.1 issues
  - No support for passive compare & update (Mantis #3540)
  - ~20 bug fixes in UVM-1.1a, several more in 1.1b; 1.1c
- Callbacks have to be used for "quirky" registers
  - VMM concept adopted by UVM
  - Can become complex if "layering" behaviours
- Good examples & boilerplate at http://www.verificationacademy.com

Energy Management Excellence

Any Questions ?