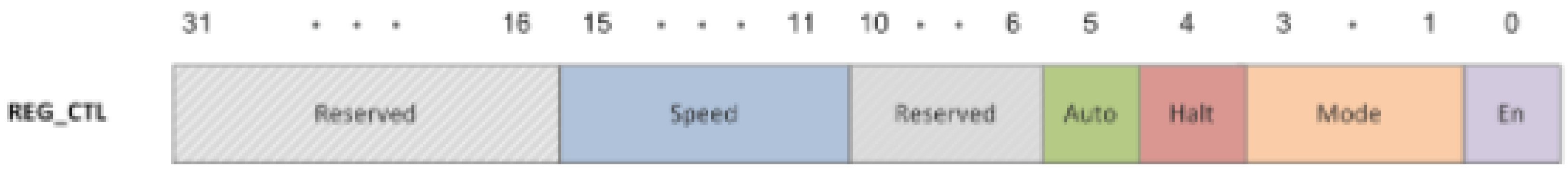
Register Abstraction Layer

What is the UVM register layer?

- These classes are used to create a high-level, object-oriented model for memories and memory-mapped registers in DUV
- The register layer defines many base classes that can be extended to abstract read and write operations to the DUV

What are registers?

- Most digital design blocks have software controllable registers that can be accessed via a peripheral bus.
- These registers allow the hardware to behave in certain ways when programmed with certain values.

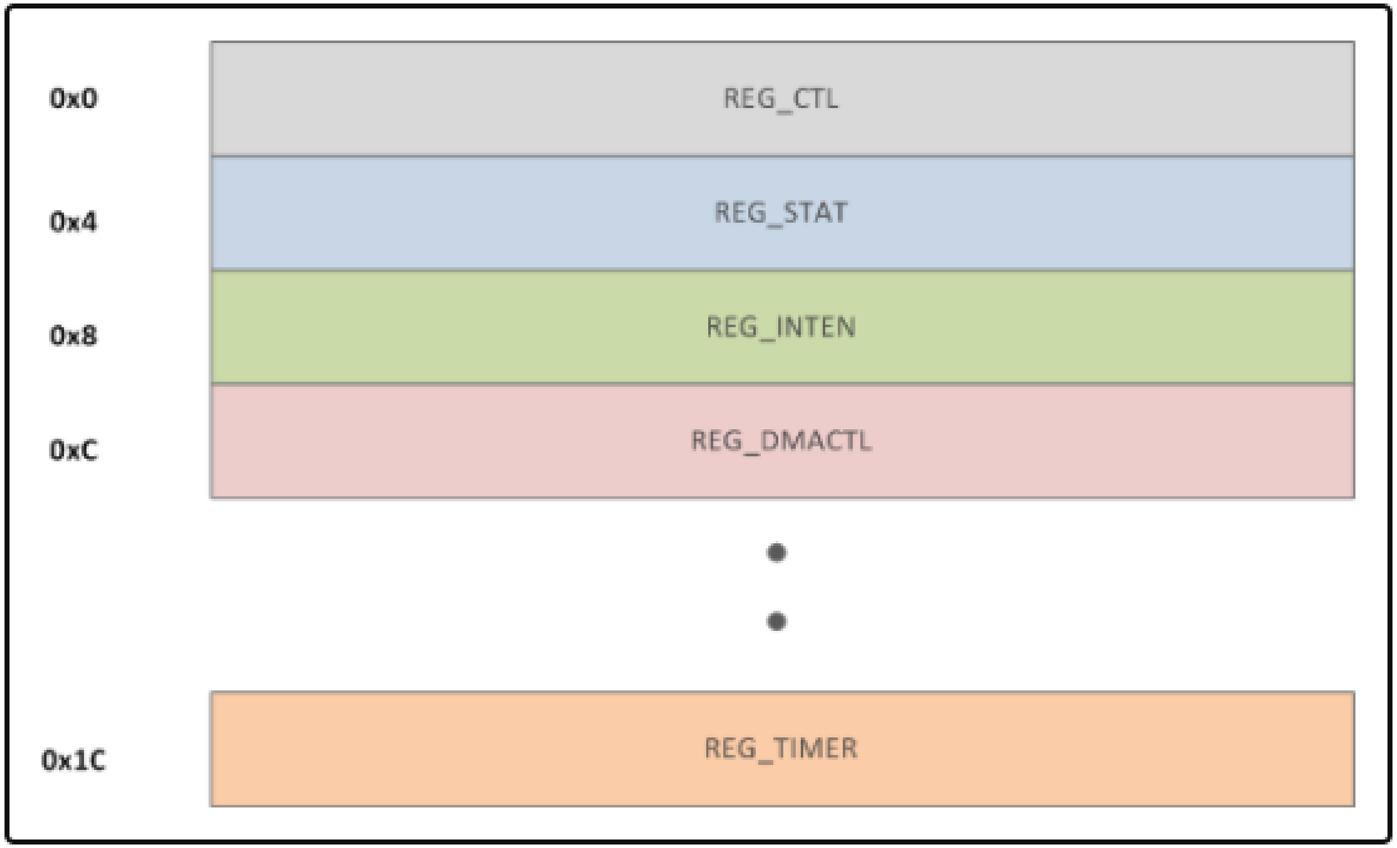


Bit Offset	Name	Read/Write	Width	Description
0	En	RW	1	Enable. Set this bit to 1 to turn on the entire module, set it to 0 to disable
1	Mode	RW	3	Supports the following modes : • Low Energy : 000 • Medium Energy : 001 • High Speed : 010 • Auto Adjust : 011 • Cruise Mode : 100 • Balanced : 101
4	Halt	RW	1	Halt. Set this bit to 1 to stop this module temporarily and set it to 0 to resume
5	Auto	RW	1	Auto shutdown mode. Set this bit to 1 to start auto shutdown after 5 min
6	Reserved	RO	5	Reserved for future additions
11	Speed	RW	5	Set a value to control the speed from 100 rpm to 2000 rpm.
16	Reserved	RO	16	Reserved for future additions

```
// RTL representation

module reg_block (...);
    wire [31:0]     reg_ctl;

    // Declare reg variables to store register field values
    reg             ctl_en;      // Enable for the module
    reg [2:0]       ctl_mode;    // Mode
    reg             ctl_halt;    // Halt
    reg             ctl_auto;    // Auto shutdown
    reg [4:0]       ctl_speed;   // Speed control

    assign reg_ctl = {16'b0, ctl_speed, 5'b0, ctl_auto, ctl_halt, ctl_mode, ctl_en};

    // Logic for getting individual fields from the bus
    ...
endmodule
```

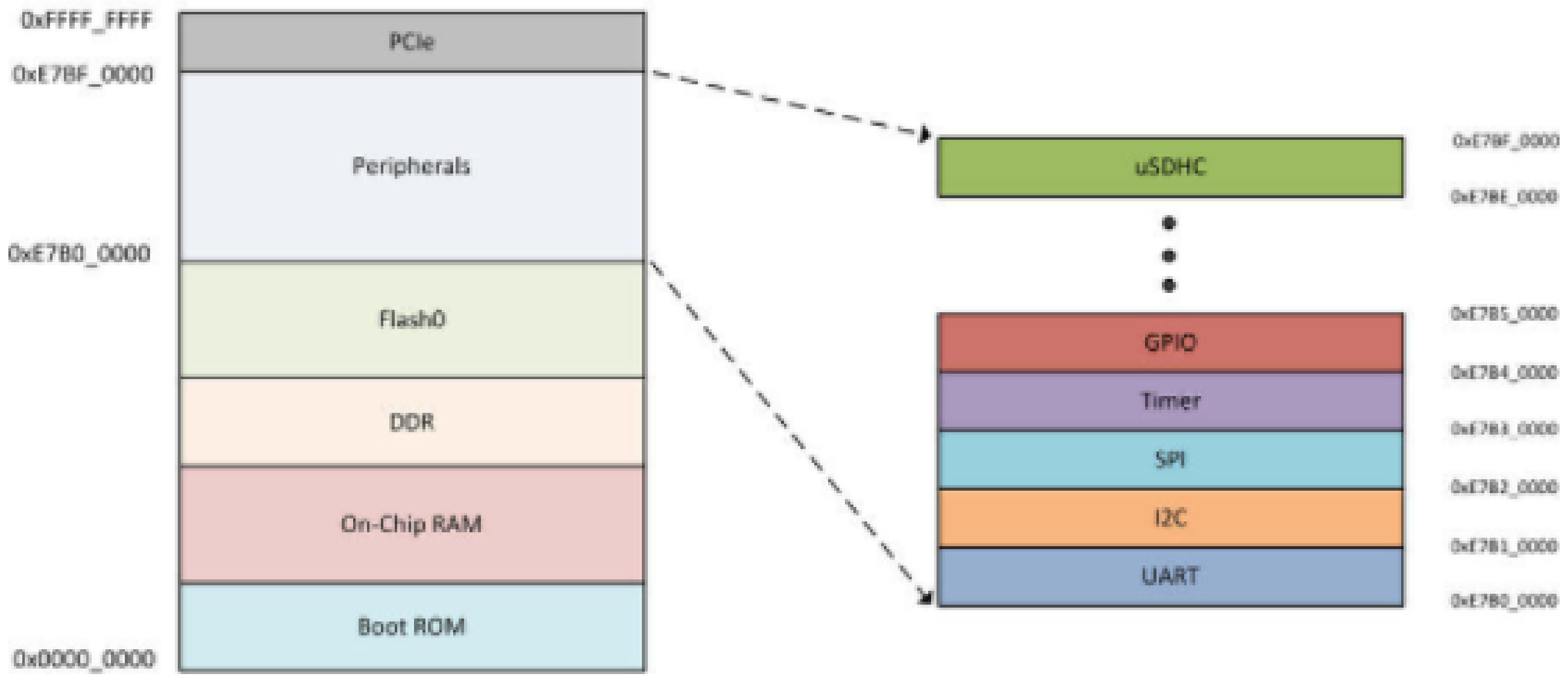
What is a register block?

- Collection of registers, with each register having a different set of fields and configurations.



What is a memory map?

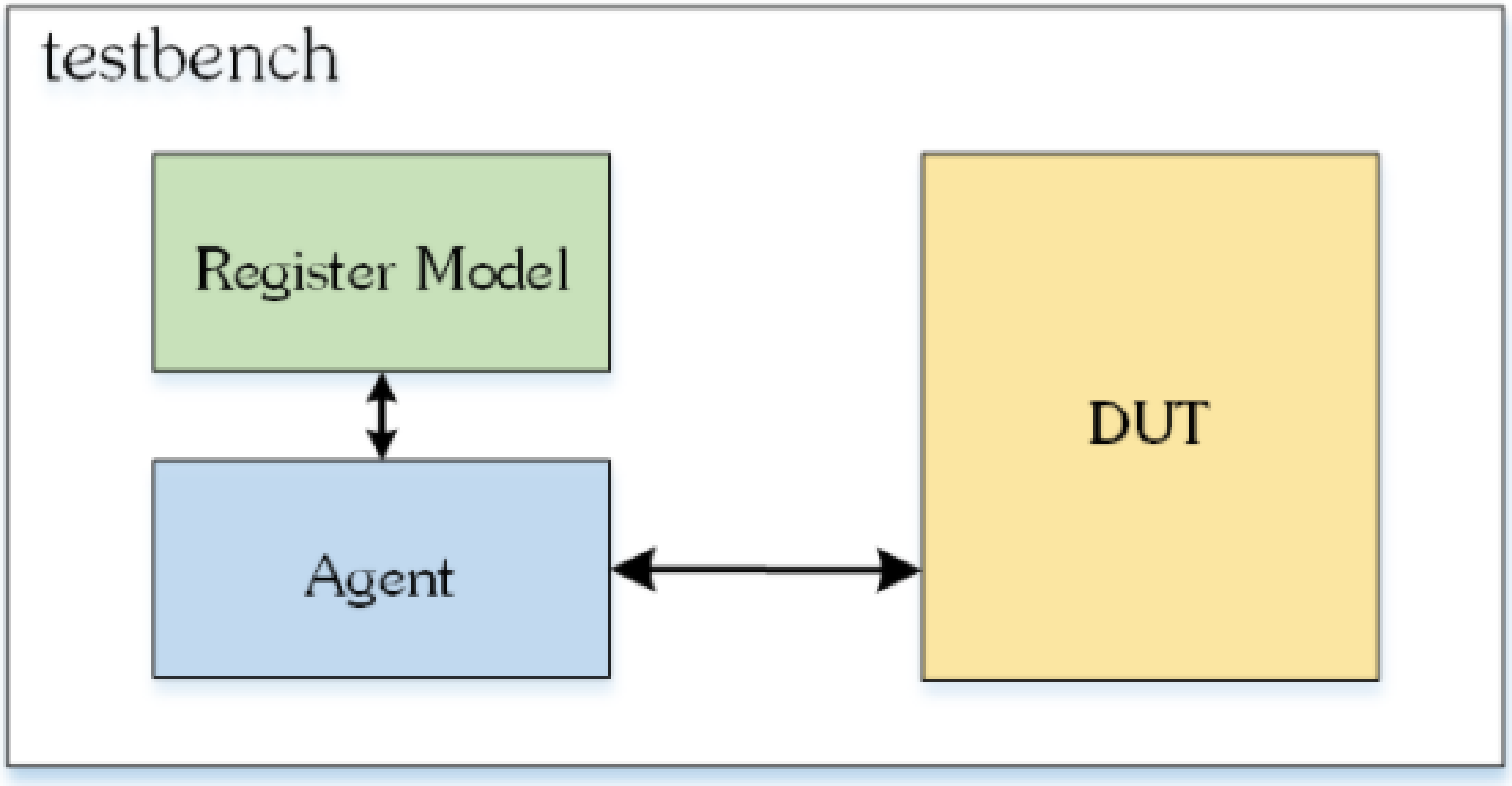
- A SoC normally has one or more processor cores, DMA engines, bus network interconnects and many peripheral modules.
- Each peripheral module would have an associated register block and they’re all laid out in memory map.



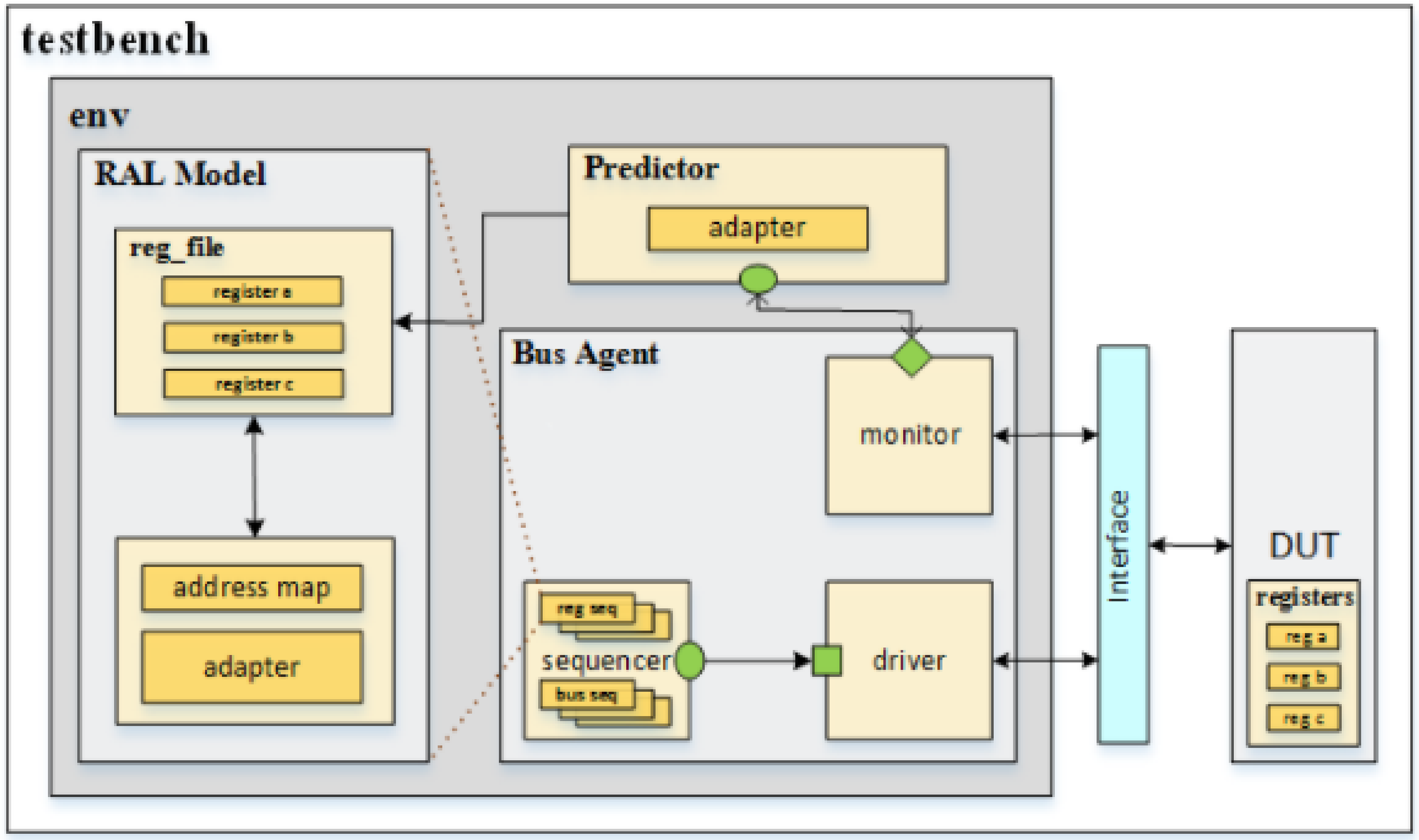
Advantages of UVM RAL

- Provides high-level abstraction for reading and writing DUT registers. i.e, registers can be accessed with its names
- UVM provides a register test sequence library containing predefined test cases these can be used to verify the registers and memories
- Register layer classes support front-door and back-door access
- Design registers can be accessed independently of the physical bus interface. i.e by calling read/write methods
- The register model can be accessed from multiple concurrent threads. It internally serializes the access to the register.
- Reusability, RAL packages can be directly reused in other environments
- Uniformity, Defines the set of rules or methodology on register access, which can be followed across the industry
- Automated RAL model generations, Tools or open-source scripts are available for RAL Model generation

RAL in Verification Testbench

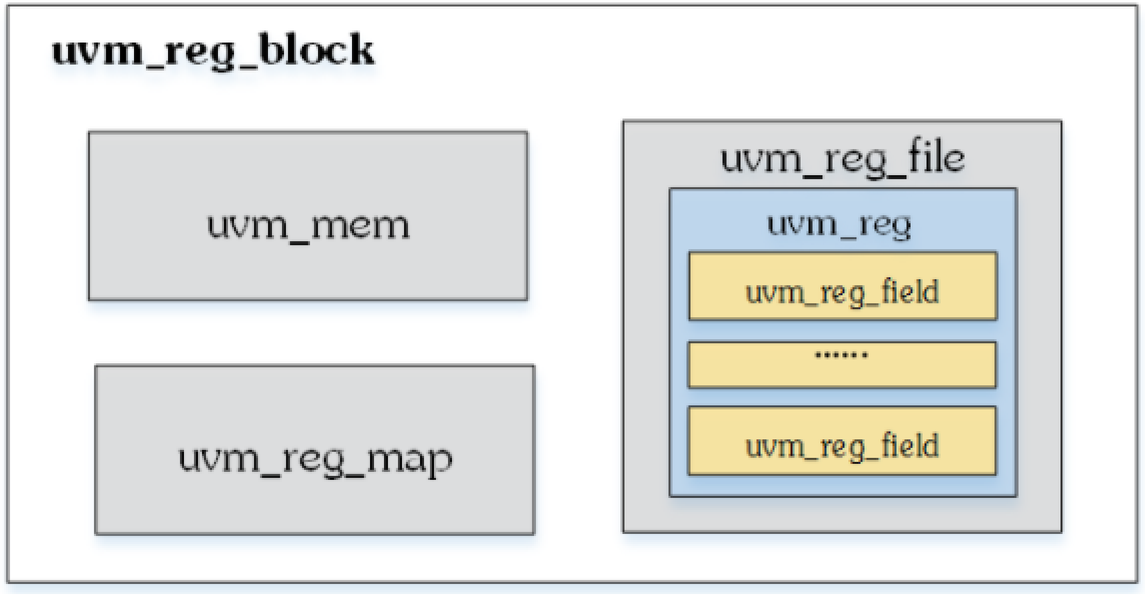


RAL in Verification Testbench

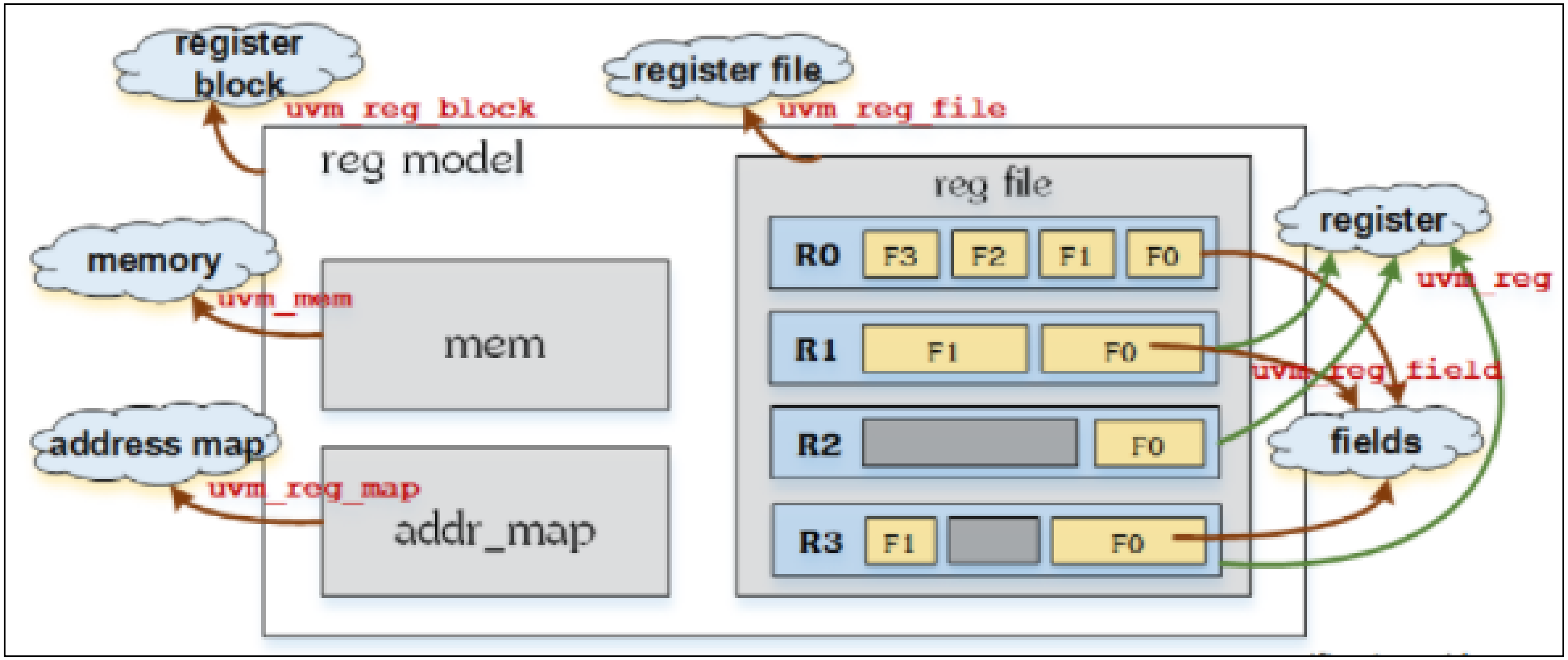


UVM Register Model Overview

- Register model consists of a hierarchy of blocks that map to the design hierarchy.
- Blocks can contain
  - Registers
  - Register files
  - Memories
  - Other blocks



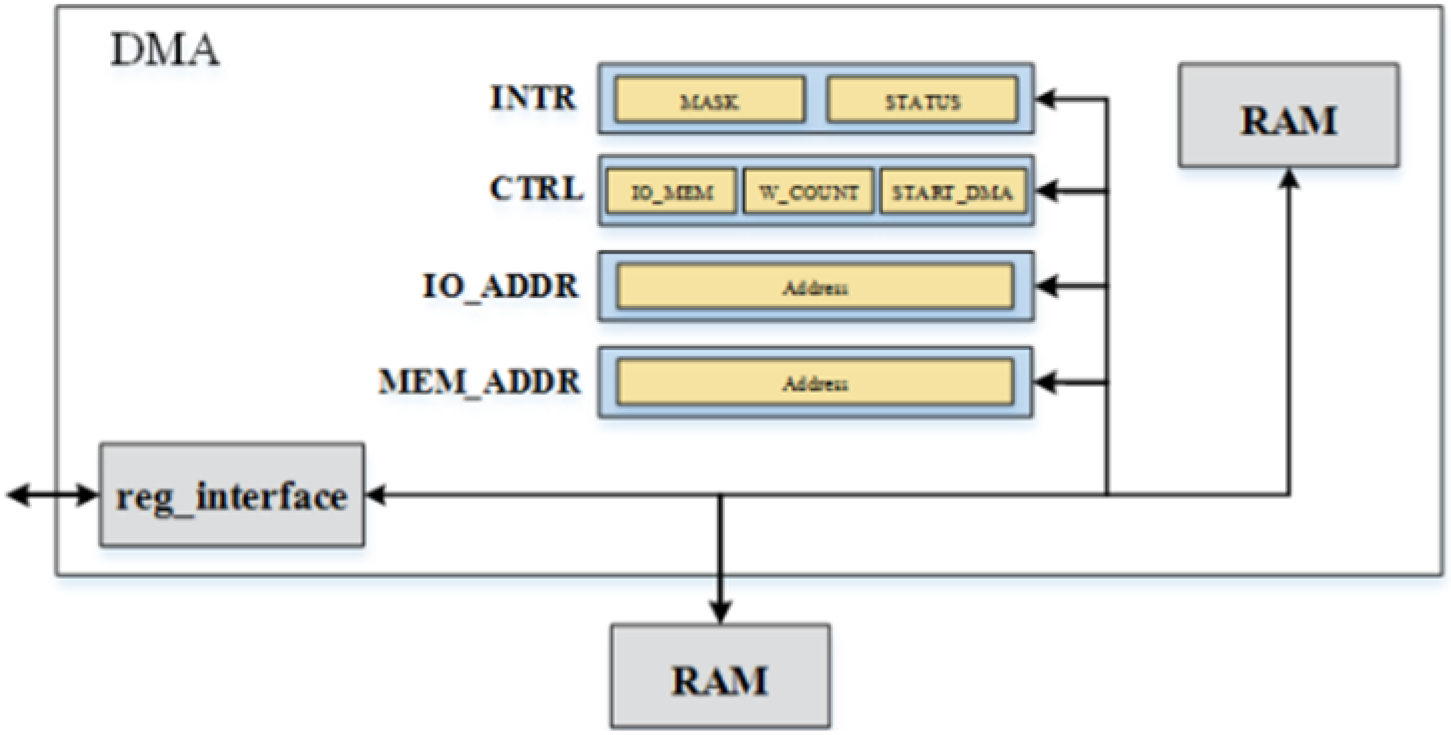
Mapping of Register Model Components



- The block diagram shows the mapping of register model components to the environmental components.
- Due to a large number of registers in design, this specialization shall be done by a register model generator.

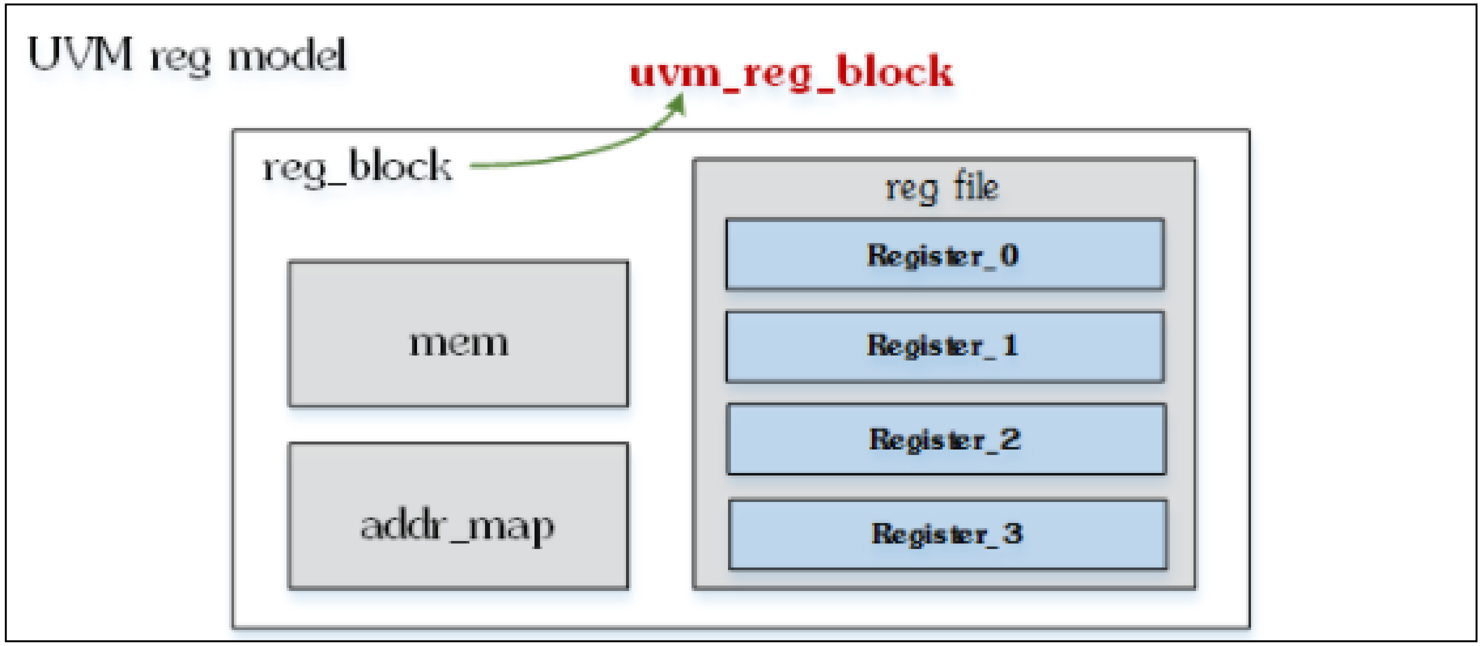
UVM RAL Usage Model

- uvm_reg_block represents each design module
- uvm_reg_file groups the module registers
- uvm_reg implements the design register
- uvm_reg_field type is used for field implementation



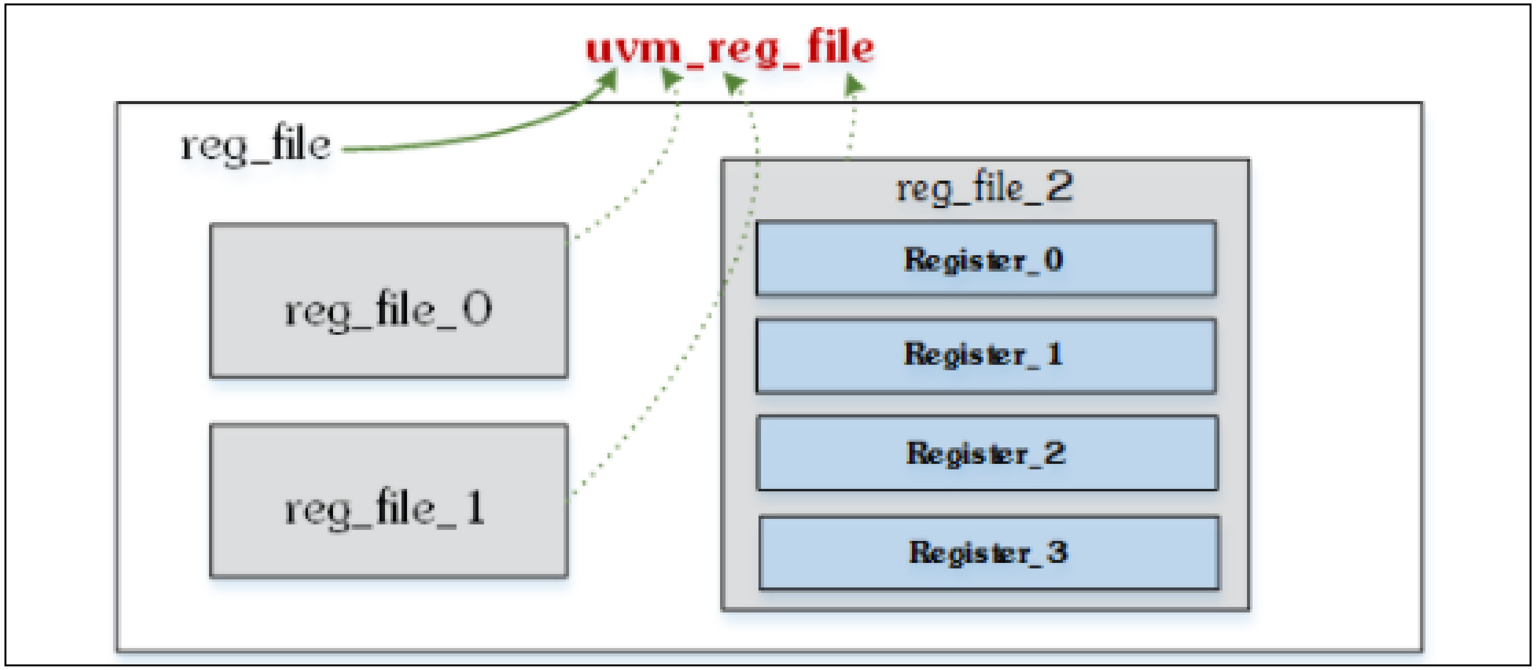
RAL Building Blocks –Register Block

- Extends the **uvm_reg_block**
- A **block** is a design component/hierarchy with its own interface(s), registers, register files, memories and sub-blocks.
- A **register model** is an instance of a **register block**



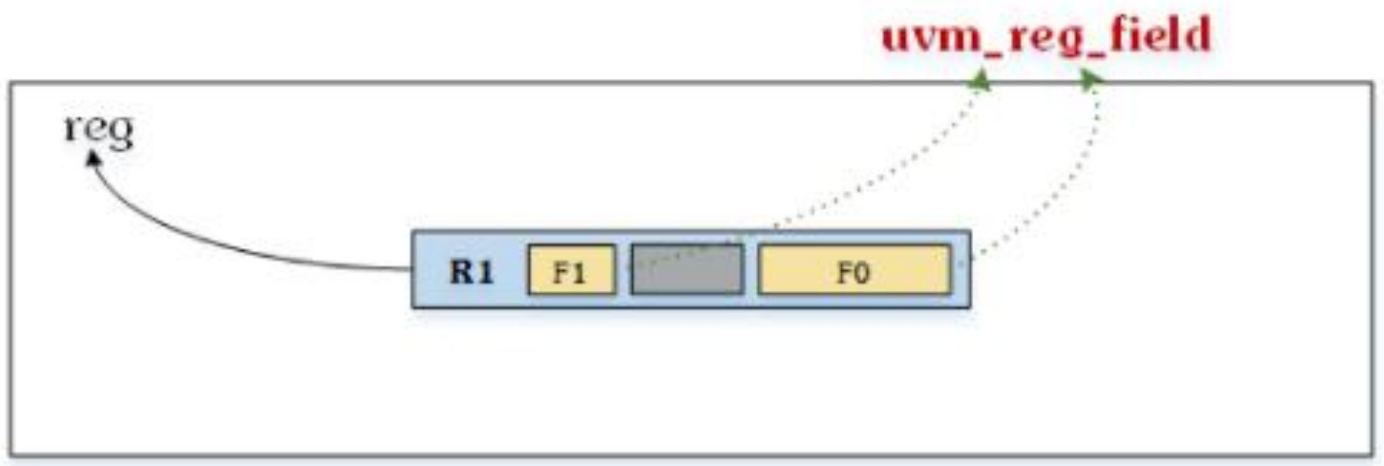
RAL Building Blocks – Register File

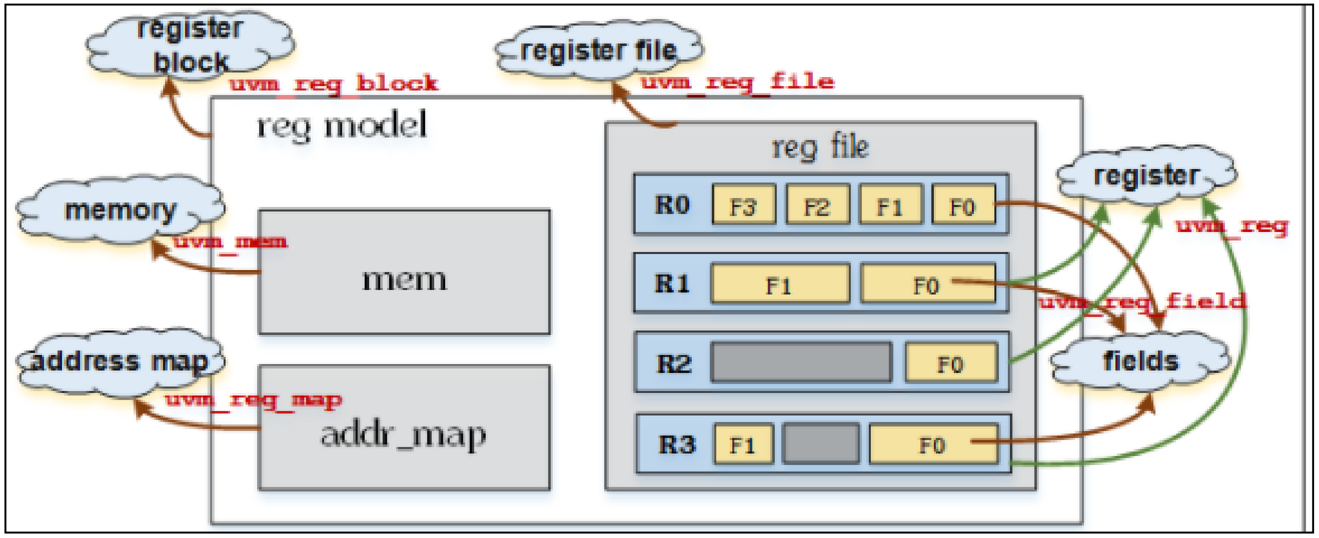
- Extends the uvm_reg_file
- Used to group the number of registers or register files.



RAL Building Blocks – Register Field

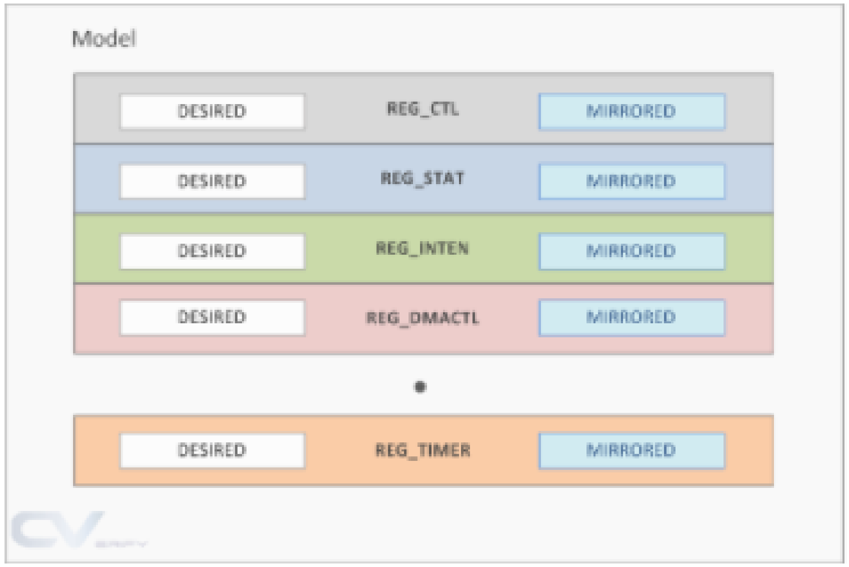
- Declared with type uvm_reg_field

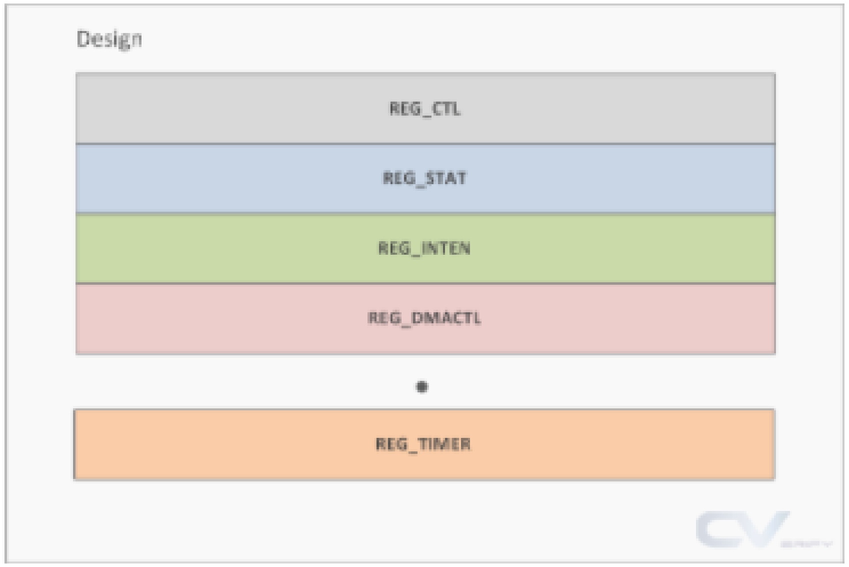




UVM Register Model

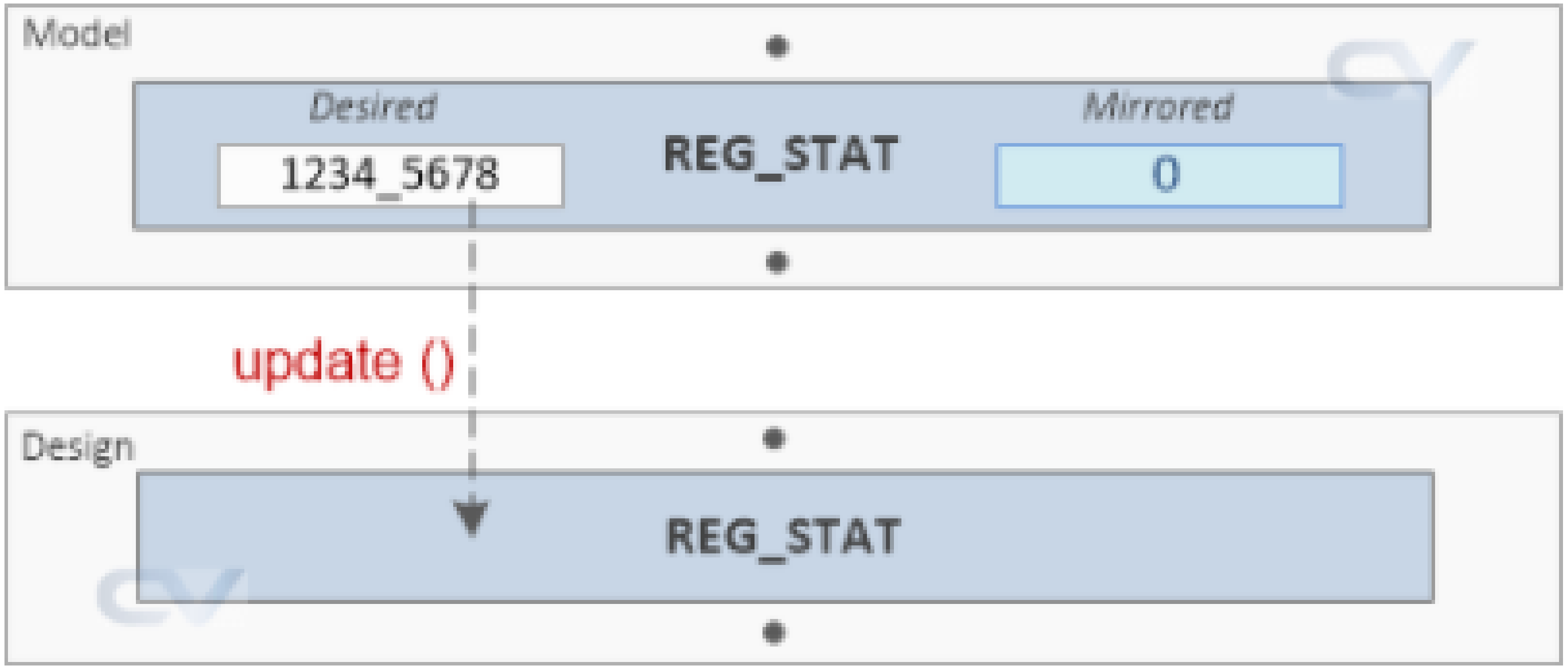
- Desired value
- Mirrored value





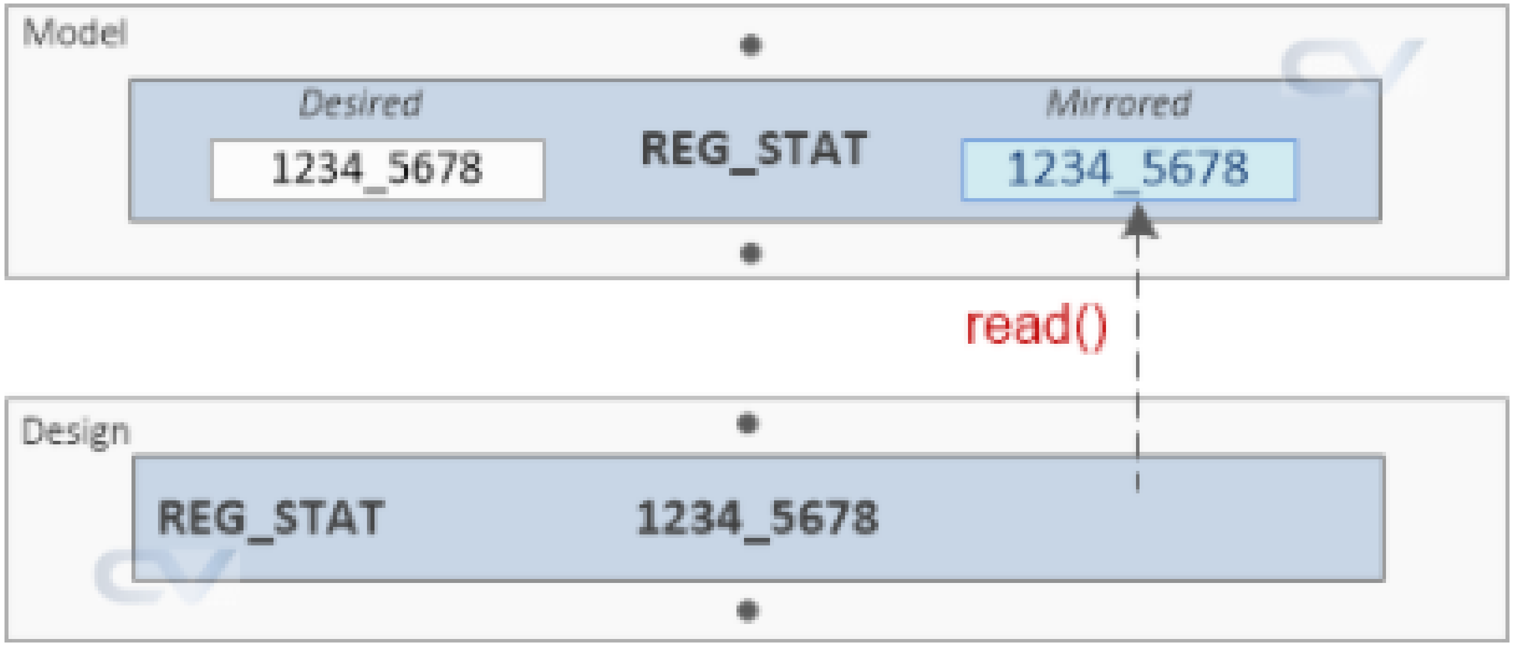
Desired Value

- This is the value we would like the design to have.
- In other words, the model has an internal variable to store a *desired value* that can be *updated* later in the design



Mirrored Value

- Every time a read or write operation occurs on the design, the *mirrored* values for that particular register will be updated.
- Hence, the mirrored value in the model is the latest known value in the design.

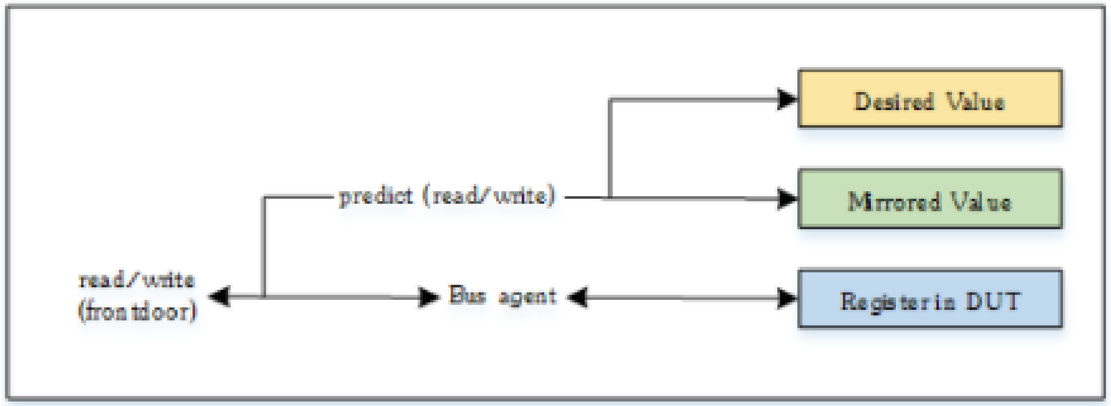


UVM RAL Methods

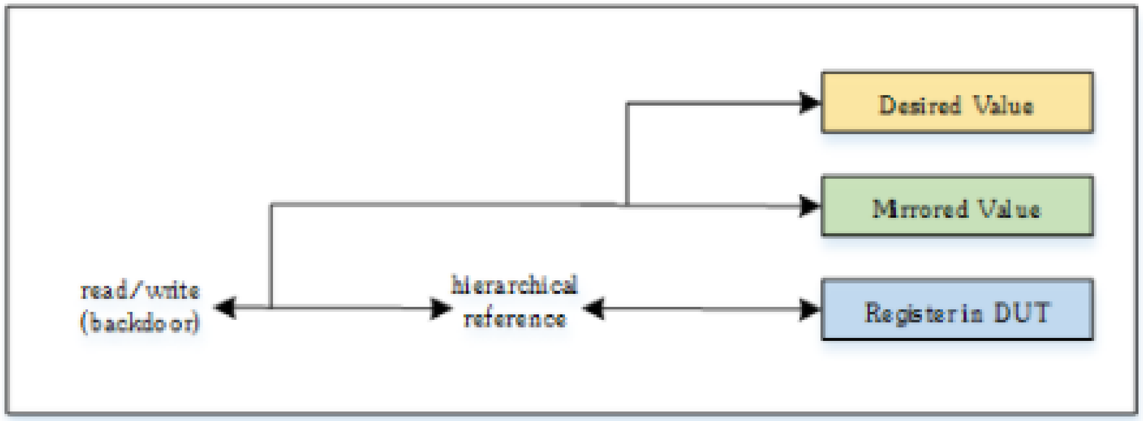
- **Register Access Methods**
  - Built-in methods used for accessing the registers
- **APIs**
  - Methods to read, write, update and mirror DUT registers and register field values
- **Front door access**
  - Involves using bus interface and is associated with timing
- **Back door access**
  - Uses simulator database access routines and happens in zero simulation time

API Methods – read and write

- read() returns and updates the value of the DUT register
- write() writes and updates the value of the DUT register



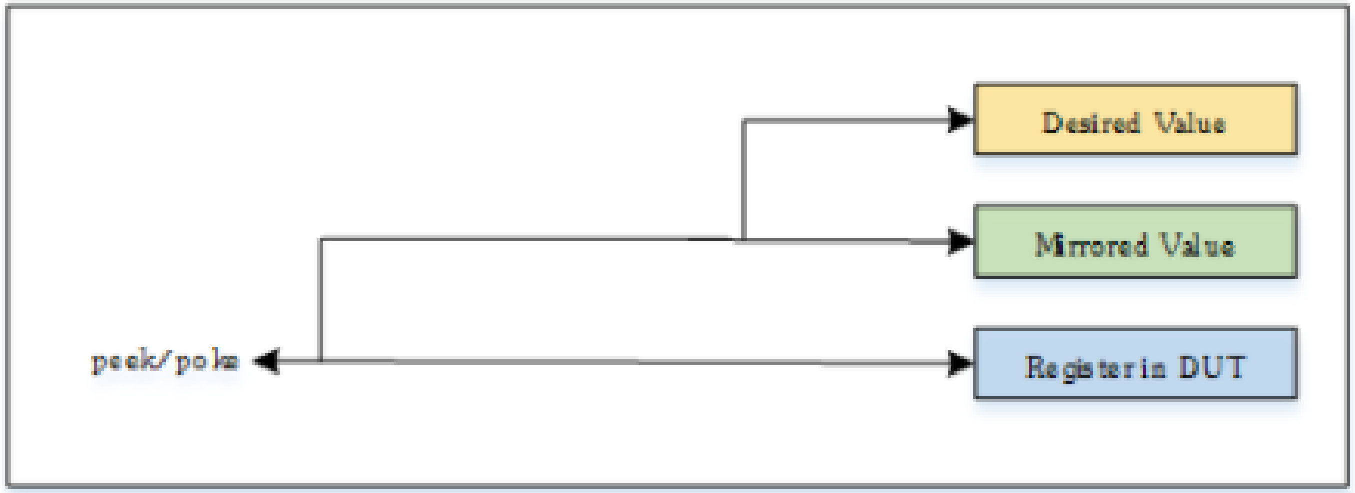
UVM RAL FrontDoor Access



UVM RAL BackDoor Access

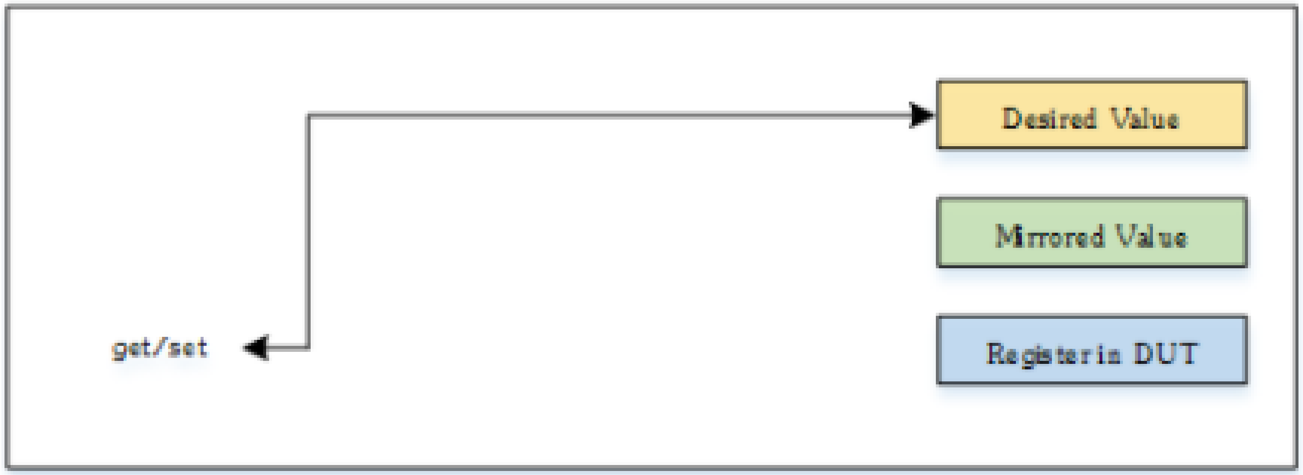
API Methods – peek and poke

- peek() reads the DUT register value using a backdoor
- poke() writes a value to DUT register using backdoor



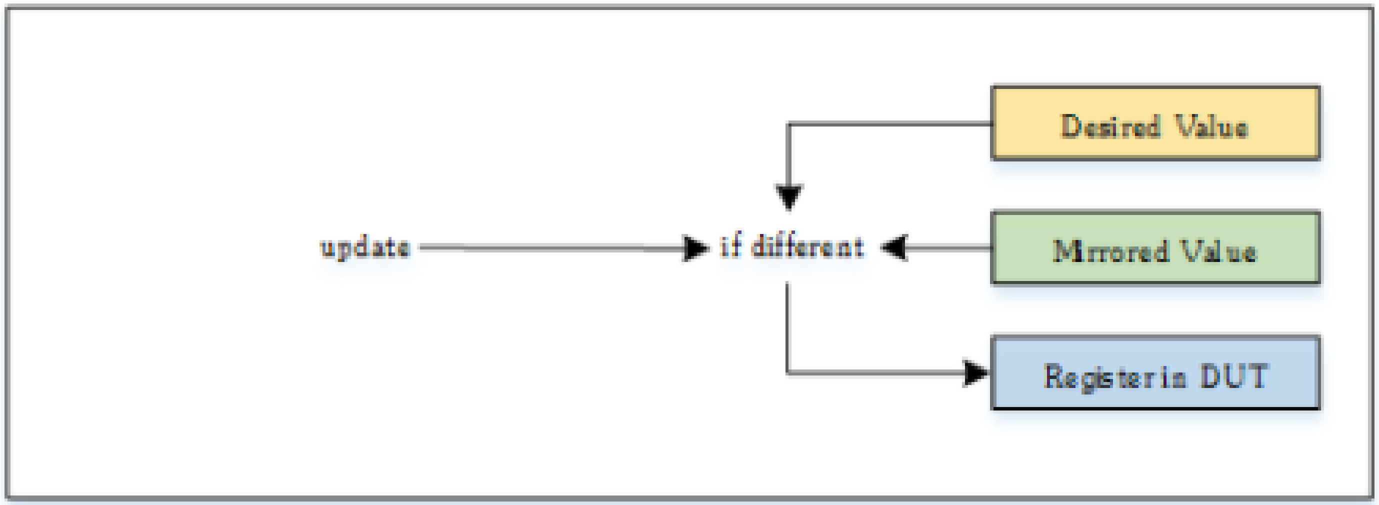
API Methods – set and get

- set() and get() methods operates on the register model desired value, not accesses to DUT register value.
- The desired value can be updated to the DUT using the update method.



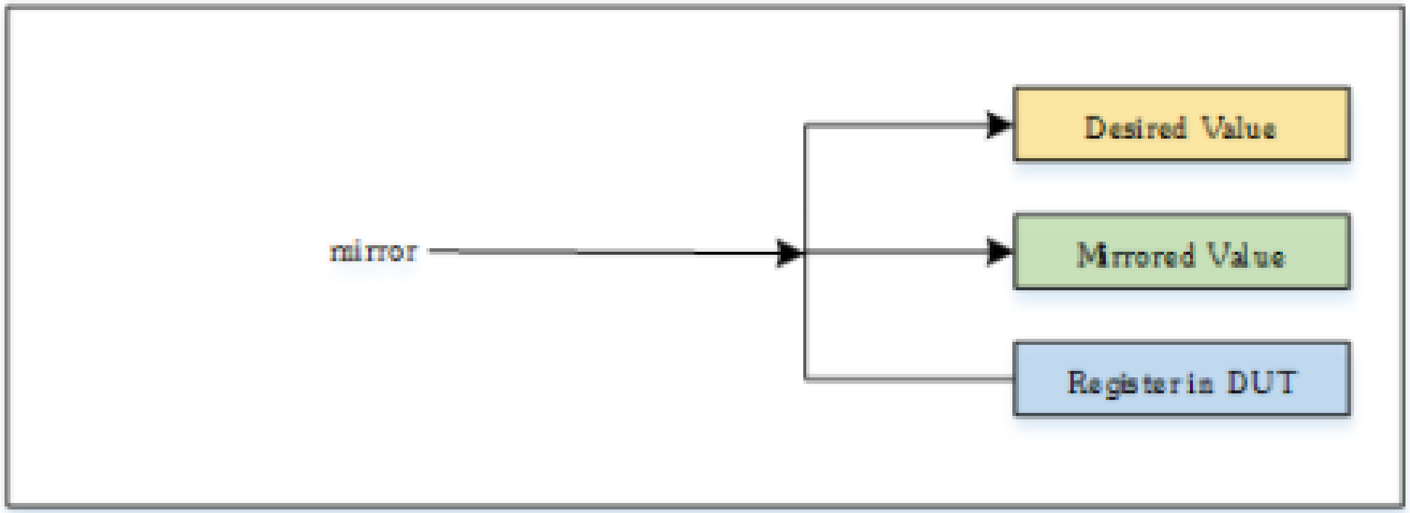
API Methods – update

- If there is a difference between desired value and mirrored value, update() will initiate a write to register
- Update method can be used after the set method



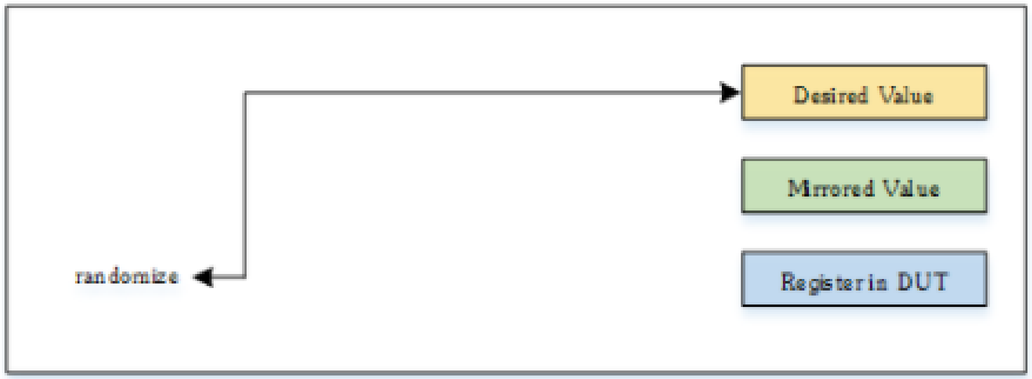
API Methods - mirror

- mirror() reads the updated DUT register values.
- The mirroring can be performed in the front door or back door(peek())



API Methods – randomize, reset

- randomize() randomizes register or field values with or without constraints as per the requirement
- Register values can be modified in post_randomize()
- After randomization, update() can be used to update the DUT register values



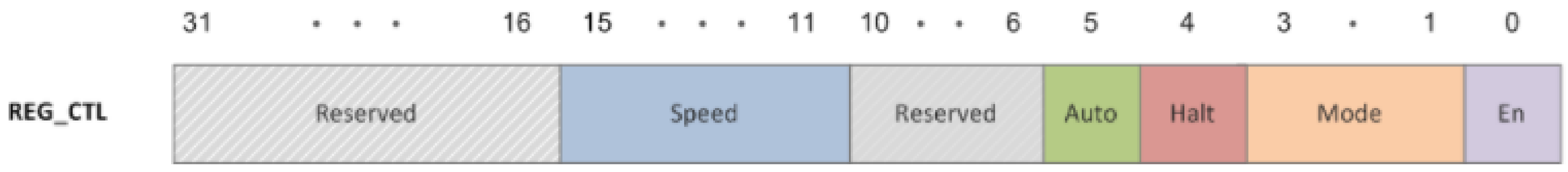
- reset() sets the register desired and mirrored value to the pre-defined reset value

Method	Front door access	Back door access
read()	RF	RF
write()	RF	RF
peek()	RF	RF
poke()	RF	RF
set()	RF	RF
get()	RF	RF
update()	BR	BR
mirror()	BR	BR
randomize()	BRF	BRF
reset()	BRF	BRF

register access methods at a different level

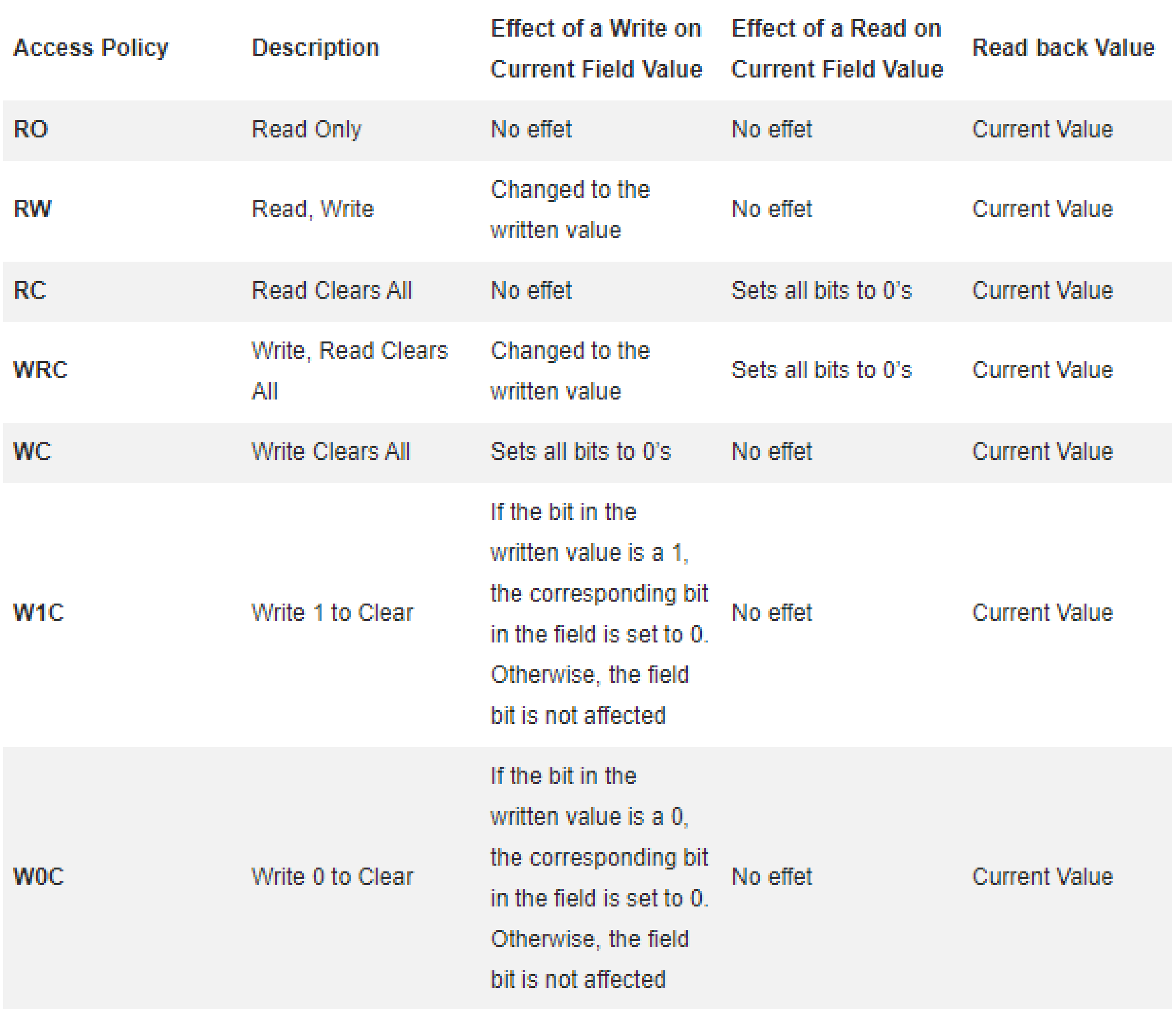
* **B** – Block **R** – Register **F** – Field

Creating a register class – Part 1



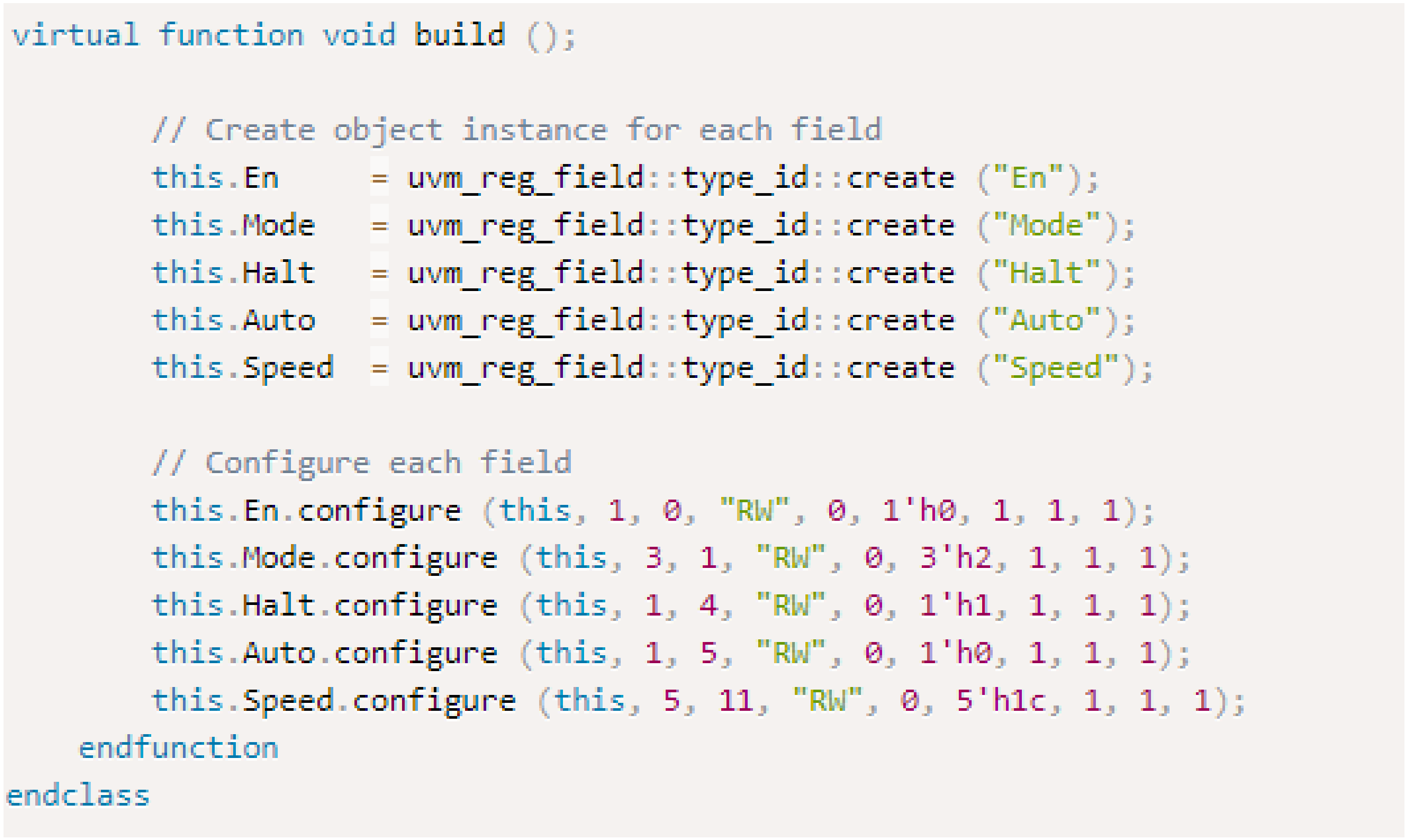
```
class reg_ctl extends uvm_reg;
    rand uvm_reg_field  En;
    rand uvm_reg_field  Mode;
    rand uvm_reg_field  Halt;
    rand uvm_reg_field  Auto;
    rand uvm_reg_field  Speed;
```

```
function new (string name = "reg_ctl");
    super.new (name, 32, UVM_NO_COVERAGE);
endfunction
```



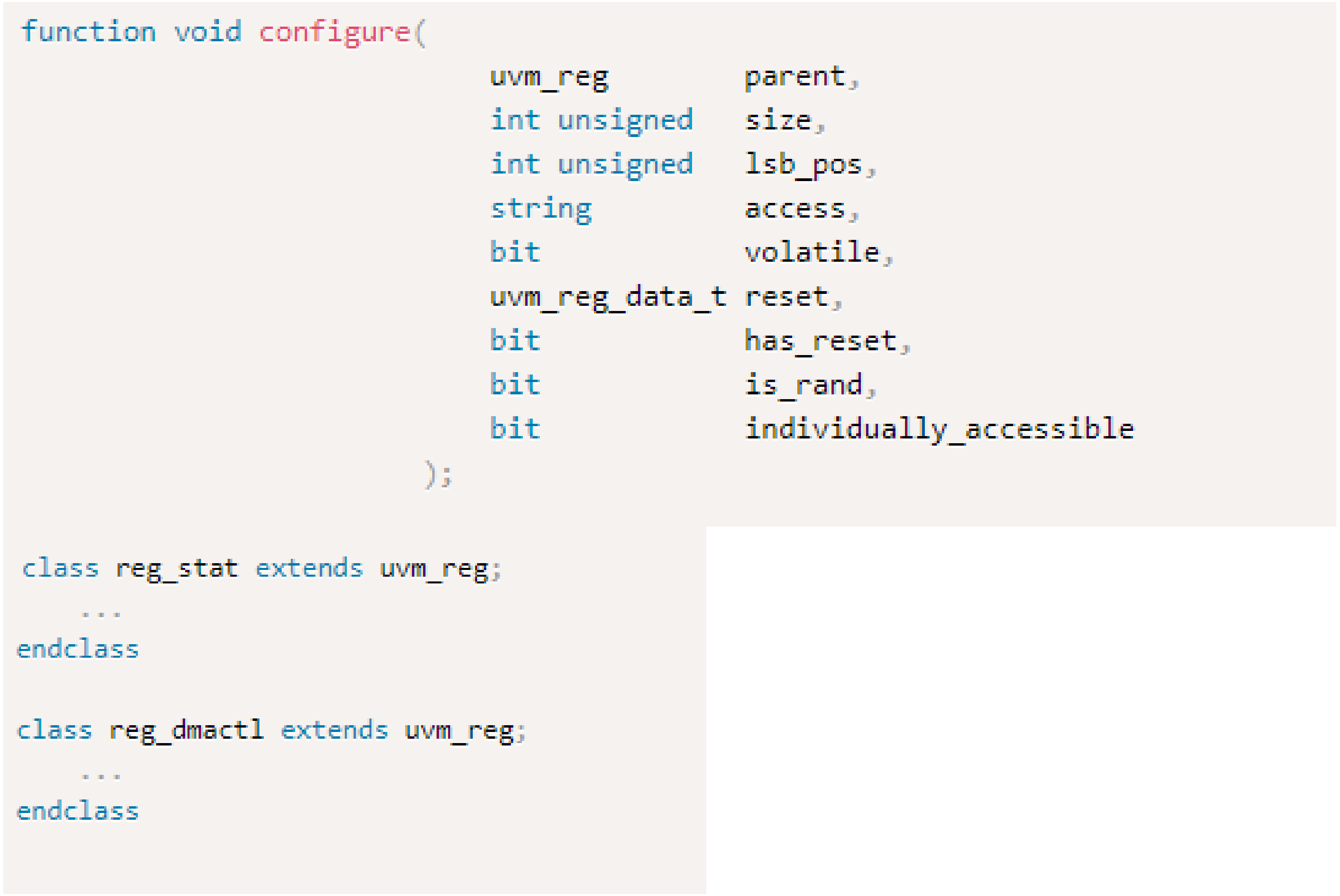
Access Policy	Description	Effect of a Write on Current Field Value	Effect of a Read on Current Field Value	Read back Value
RO	Read Only	No effet	No effet	Current Value
RW	Read, Write	Changed to the written value	No effet	Current Value
RC	Read Clears All	No effet	Sets all bits to 0's	Current Value
WRC	Write, Read Clears All	Changed to the written value	Sets all bits to 0's	Current Value
WC	Write Clears All	Sets all bits to 0's	No effet	Current Value
W1C	Write 1 to Clear	If the bit in the written value is a 1, the corresponding bit in the field is set to 0. Otherwise, the field bit is not affected	No effet	Current Value
W0C	Write 0 to Clear	If the bit in the written value is a 0, the corresponding bit in the field is set to 0. Otherwise, the field bit is not affected	No effet	Current Value

Creating a register class – Part 2



```
virtual function void build ();

        // Create object instance for each field
        this.En     = uvm_reg_field::type_id::create ("En");
        this.Mode   = uvm_reg_field::type_id::create ("Mode");
        this.Halt   = uvm_reg_field::type_id::create ("Halt");
        this.Auto   = uvm_reg_field::type_id::create ("Auto");
        this.Speed  = uvm_reg_field::type_id::create ("Speed");

        // Configure each field
        this.En.configure (this, 1, 0, "RW", 0, 1'h0, 1, 1, 1);
        this.Mode.configure (this, 3, 1, "RW", 0, 3'h2, 1, 1, 1);
        this.Halt.configure (this, 1, 4, "RW", 0, 1'h1, 1, 1, 1);
        this.Auto.configure (this, 1, 5, "RW", 0, 1'h0, 1, 1, 1);
        this.Speed.configure (this, 5, 11, "RW", 0, 5'h1c, 1, 1, 1);
    endfunction
endclass
```

Creating a register class – Part 3



```
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 reg_stat extends uvm_reg;
   ...
endclass

class reg_dmactl extends uvm_reg;
   ...
endclass
```

Creating a register block

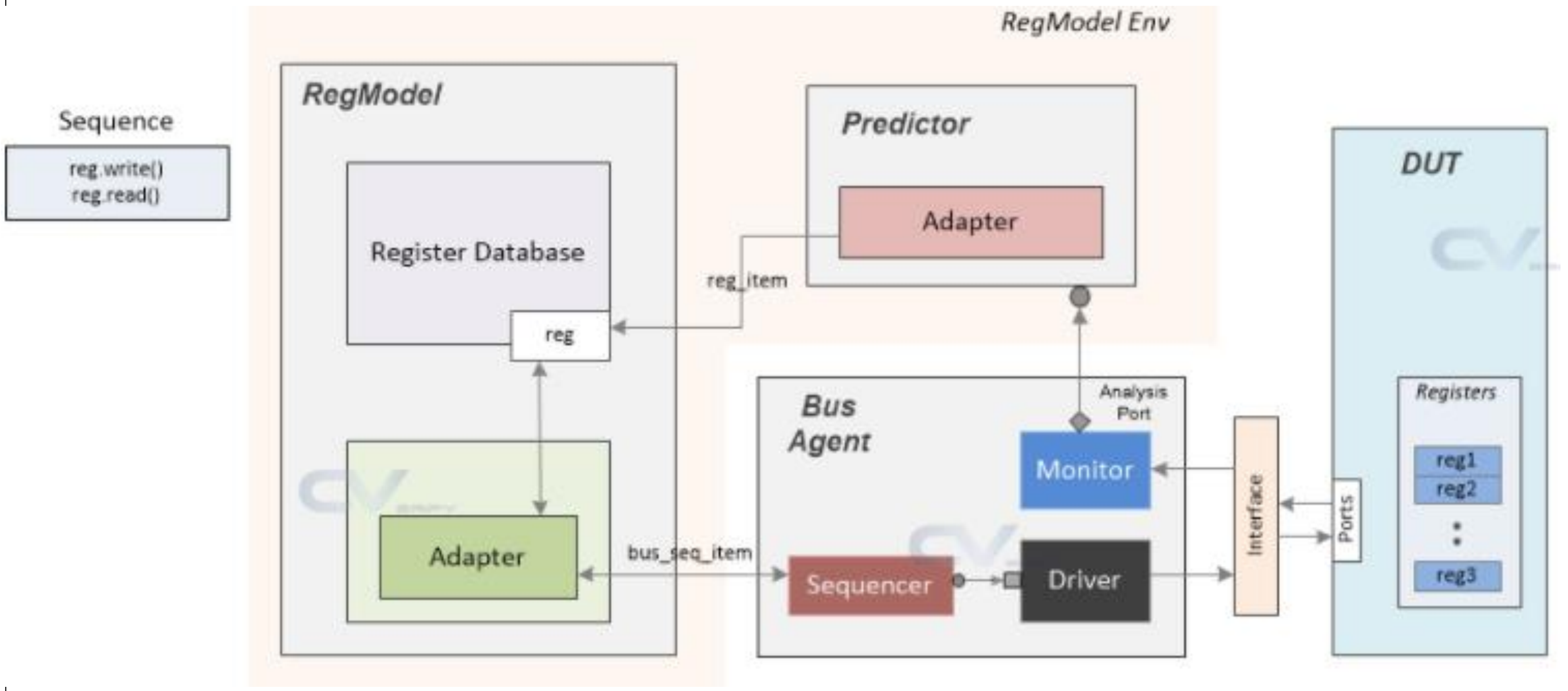
```
class reg_block extends uvm_reg_block;
   rand    reg_ctl     m_reg_ctl;
   rand    reg_stat    m_reg_stat;
   rand    reg_inten   m_reg_inten;

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

   virtual function void build ();

      // Create an instance for every register
      this.default_map = create_map ("", 0, 4, UVM_LITTLE_ENDIAN, 0);
      this.m_reg_ctl = reg_ctl::type_id::create ("m_reg_ctl", , get_full_name);
      this.m_reg_stat = reg_stat::type_id::create ("m_reg_stat", , get_full_name);
      this.m_reg_inten = reg_inten::type_id::create ("m_reg_inten", , get_full_name);

      // Configure every register instance
      this.m_reg_ctl.configure (this, null, "");
      this.m_reg_stat.configure (this, null, "");
      this.m_reg_inten.configure (this, null, "");

      // Call the build() function to build all register fields within each register
      this.m_reg_ctl.build();
      this.m_reg_stat.build();
      this.m_reg_inten.build();

      // Add these registers to the default map
      this.default_map.add_reg (this.m_reg_ctl, `UVM_REG_ADDR_WIDTH'h0, "RW", 0);
      this.default_map.add_reg (this.m_reg_stat, `UVM_REG_ADDR_WIDTH'h4, "RO", 0);
      this.default_map.add_reg (this.m_reg_inten, `UVM_REG_ADDR_WIDTH'h8, "RW", 0);
   endfunction
endclass
```

UVM Register Environment

Four components required for a register environment:

1. A register model based on UVM classes
2. An agent to drive actual bus transactions to the design
3. An adapter to convert the read and write statements from the model to protocol based bus transactions
4. A predictor to understand bus activity and update the register model to match the design contents

UVM Register Environment



Register Adpater

- Acts as a converter between RAL model and interface
- Converts transactions of RAL methods to Interface/Bus transactions
- The adapter converts between register model read, write methods and the interface-specific transactions
- The transaction adapter is implemented by extending uvm_reg_adapter class and implementing reg2bus() and bus2reg() methods.

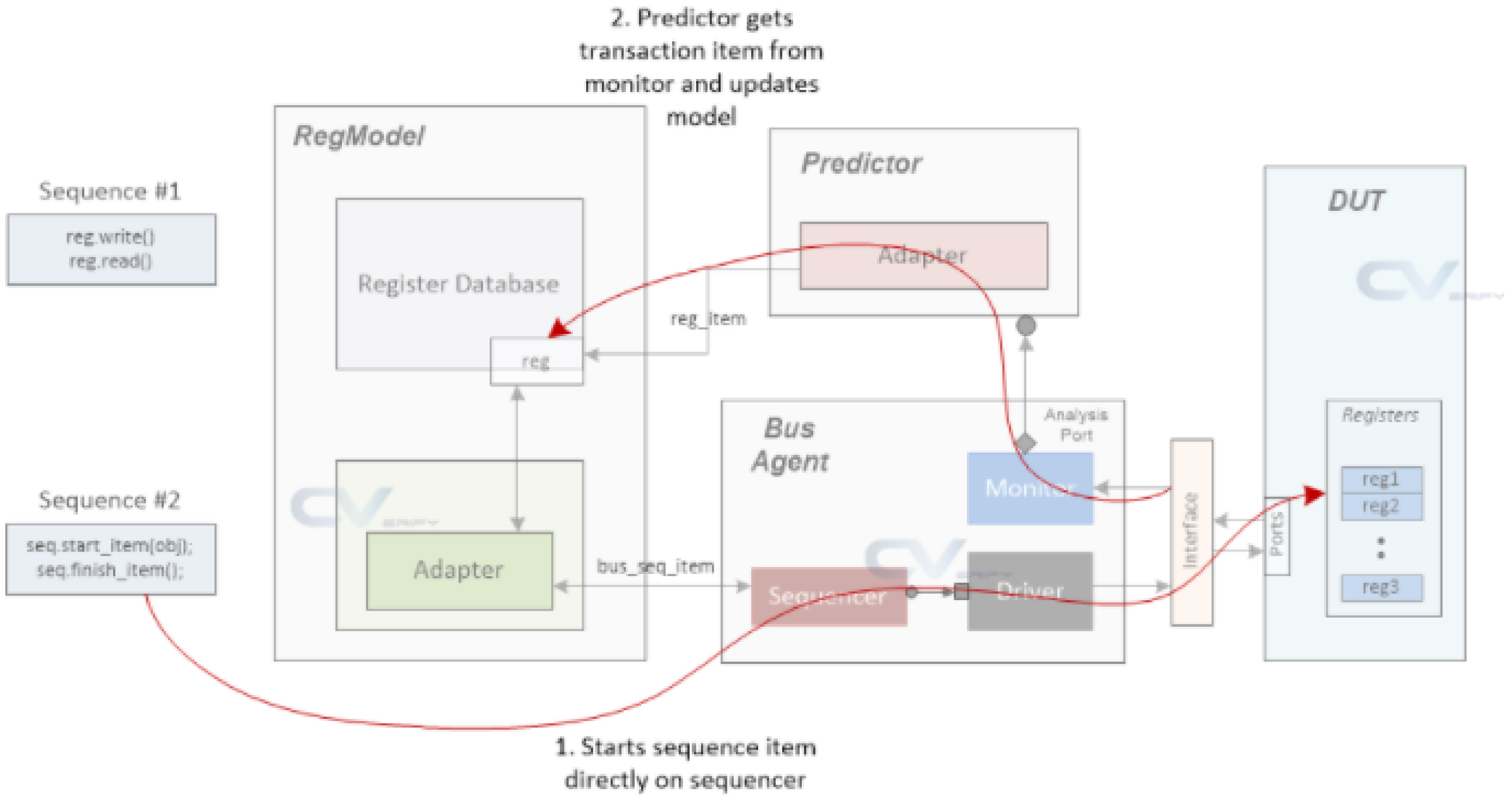
Register Adapter

```
class reg_ctl extends uvm_reg;
    ...
endclass

m_reg_ctl.write (status, addr, wdata);      // Write wdata to addr
m_reg_ctl.read  (status, addr, rdata);      // Read rdata from addr
```

```
typedef struct {

   uvm_access_e        kind;      // Access type: UVM_READ/UVM_WRITE
   uvm_reg_addr_t      addr;      // Bus address, default 64 bits
   uvm_reg_data_t      data;      // Read/Write data, default 64 bits
   int                 n_bits;    // Number of bits being transferred
   uvm_reg_byte_en     byte_en;   // Byte enable
   uvm_status_e        status;    // Result of transaction: UVM_IS_OK, UVM_HAS_X, UVM_NOT_OK

} uvm_reg_bus_op;
```

Register Predictor

- The UVM RAL predictor predicts the register access done through the register model and updates the RAL model registers.
- It does this update based on observed transactions published by a monitor.



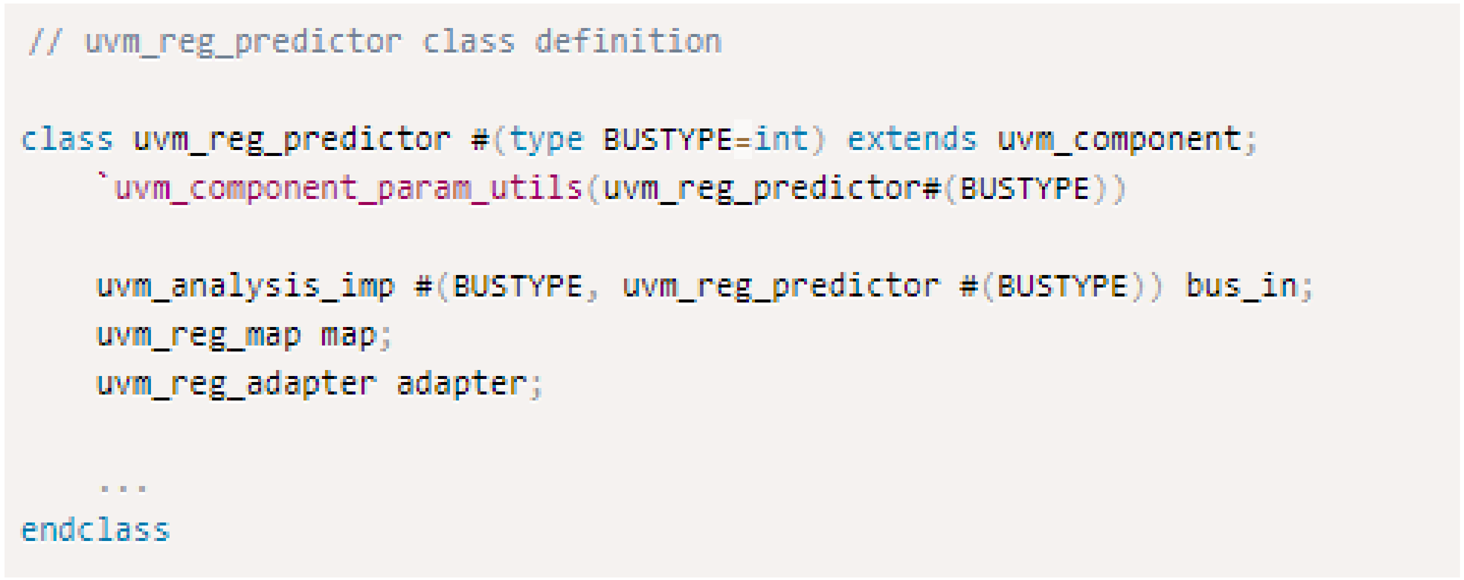
Register Predictor

Implicit Prediction

- Only requires the integration of the register model with one or more bus sequencers
- Updates to the mirror are predicted automatically (implicitly) by the register model after completion of each read, write, peek or poke

Explicit Prediction

- Requires the register model to be integrated with both bus sequencers and corresponding bus monitors
- In this mode, implicit prediction is turned off and all updates to the mirror are predicted externally (explicitly) by a uvm_reg_predictor component, one for each bus interface



```
// uvm_reg_predictor class definition

class uvm_reg_predictor #(type BUSTYPE=int) extends uvm_component;
    `uvm_component_param_utils(uvm_reg_predictor#(BUSTYPE))

    uvm_analysis_imp #(BUSTYPE, uvm_reg_predictor #(BUSTYPE)) bus_in;
    uvm_reg_map map;
    uvm_reg_adapter adapter;

    ...
endclass
```

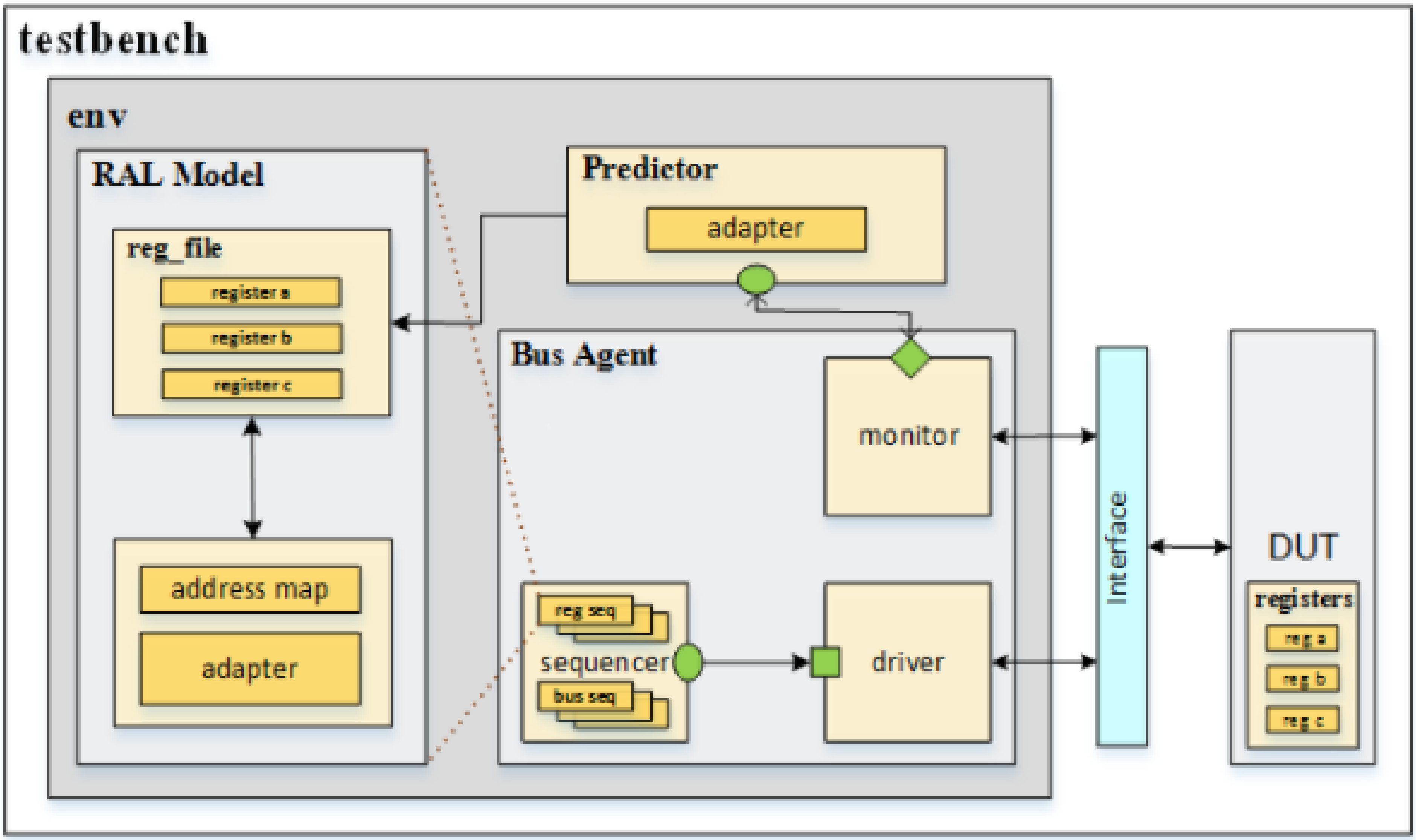
Register Environment Integration

```
class reg_env extends uvm_env;
   `uvm_component_utils (reg_env)
   function new (string name="reg_env", uvm_component parent);
      super.new (name, parent);
   endfunction

   uvm_agent                      m_agent;             // Agent handle
   ral_my_design                  m_ral_model;         // Register Model
   reg2apb_adapter                m_apb_adapter;       // Convert Reg Tx <-> Bus-type packets
   uvm_reg_predictor #(bus_pkt)   m_apb_predictor;     // Map APB tx to register in model

   virtual function void build_phase (uvm_phase phase);
      super.build_phase (phase);
      m_ral_model       = ral_my_design::type_id::create ("m_ral_model", this);
      m_apb_adapter     = m_apb_adapter :: type_id :: create ("m_apb_adapter");
      m_apb_predictor   = uvm_reg_predictor #(bus_pkt) :: type_id :: create ("m_apb_predictor", this);

      m_ral_model.build ();
      m_ral_model.lock_model ();
      uvm_config_db #(ral_my_design)::set (null, "uvm_test_top", "m_ral_model", m_ral_model);
   endfunction

   virtual function void connect_phase (uvm_phase phase);
      super.connect_phase (phase);
      m_apb_predictor.map       = m_ral_model.default_map;
      m_apb_predictor.adapter   = m_apb_adapter;
      m_agent.ap.connect(m_apb_predictor.bus_in);
   endfunction
endclass
```

Connecting register env

```
class my_env extends uvm_env;
   `uvm_component_utils (my_env)

   my_agent       m_agent;
   reg_env        m_reg_env;

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

   virtual function void build_phase (uvm_phase phase);
      super.build_phase (phase);
      m_agent = my_agent::type_id::create ("m_agent", this);
      m_reg_env = reg_env::type_id::create ("m_reg_env", this);
   endfunction

   virtual function void connect_phase (uvm_phase phase);
      super.connect_phase (phase);
      m_agent.m_mon.mon_ap.connect (m_reg_env.m_apb2reg_predictor.bus_in);
      m_reg_env.m_ral_model.default_map.set_sequencer (m_agent.m_seqr, m_reg_env.m_reg2apb);
   endfunction

endclass
```

Integrating RAL to Agent

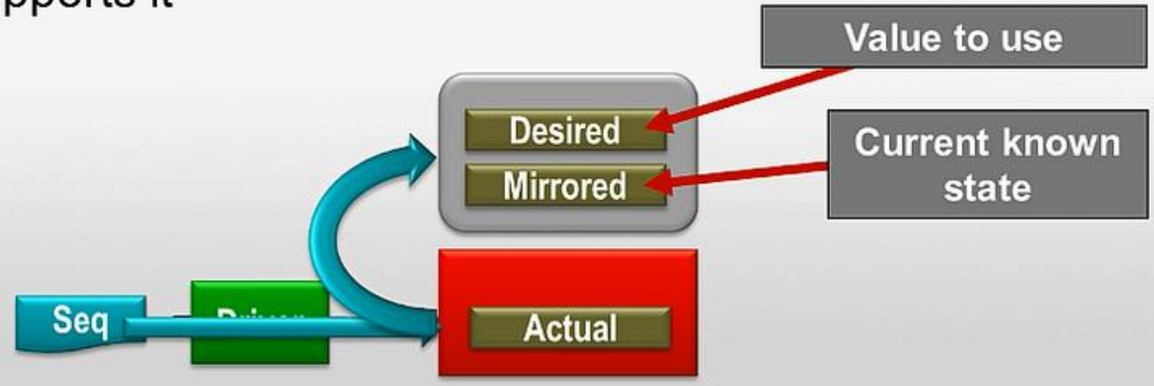


Front Door Access Modes

- **Consume time on the bus (default)**

```
write_reg(model.ctrl, status, wdata, UVM_FRONTDOOR);
read_reg (model.ctrl, status, rdata, UVM_FRONTDOOR);
```

- **Desired and Mirrored values updated at end of transaction**
  - Based on transaction contents and field access mode
- **Can access individual fields**
  - Only if hardware supports it
    - Field = byte lane

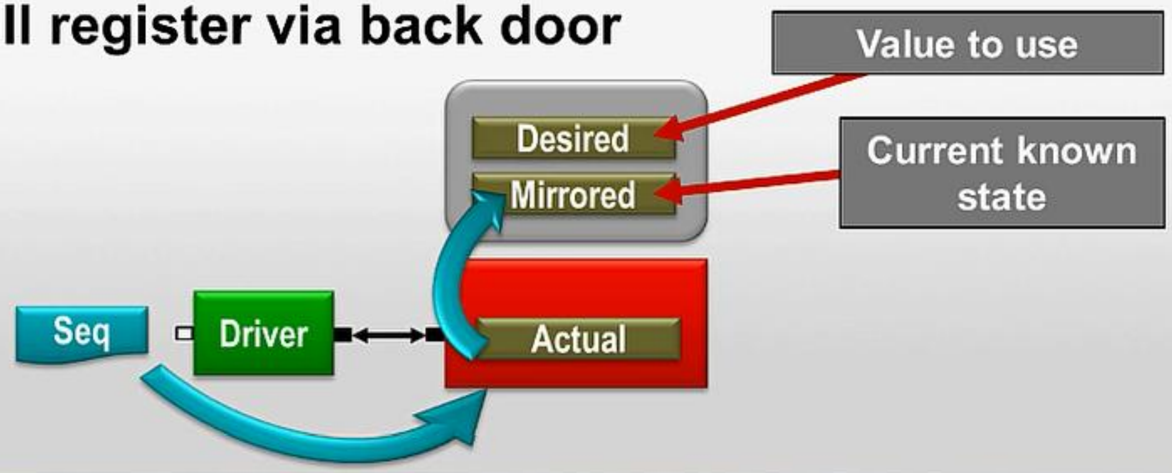


Back Door Access Modes

- **Consume no time on the bus**

```
write_reg(model.ctrl, status, wdata, UVM_BACKDOOR);
read_reg (model.ctrl, status, rdata, UVM_BACKDOOR);
```

  - must be specified explicitly
- **Desired and Mirrored values updated at end of transaction**
  - Based on transaction contents and field access mode
- **Can only access full register via back door**

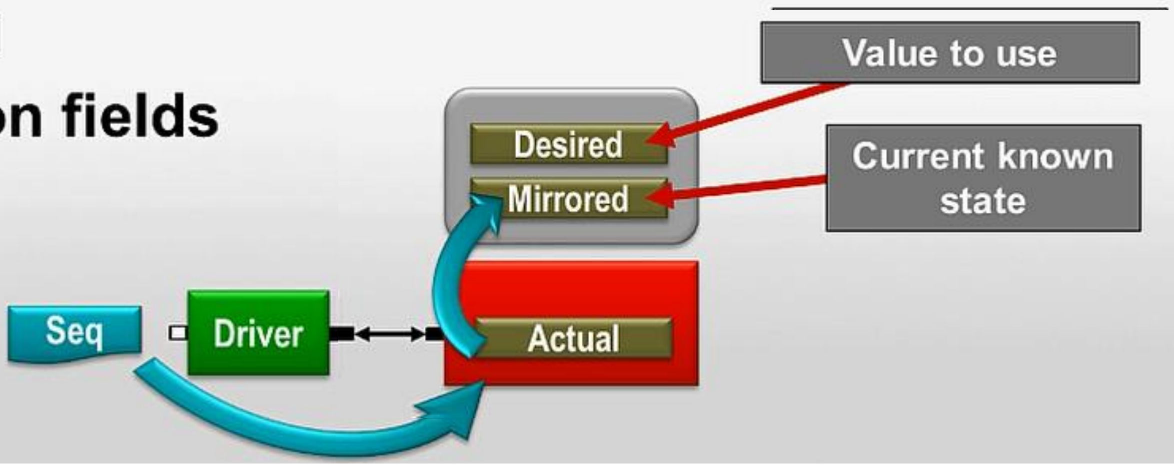


Back Door Access Modes

- **Consume no time on the bus**

```
poke_reg (model.ctrl, status, wdata);
peek_reg (model.ctrl, status, rdata);
```

- **Desired and Mirrored values updated directly at end of transaction**
  - Poke sets the actual register value
  - Peek samples the actual value, which is written to model
- **Peek/Poke work on fields**



Back Door Access

- Uncovers bugs that may be hidden since write and read cycles are performed using same access path.

- Improves efficiency of verifying registers and memories

Mirror Method

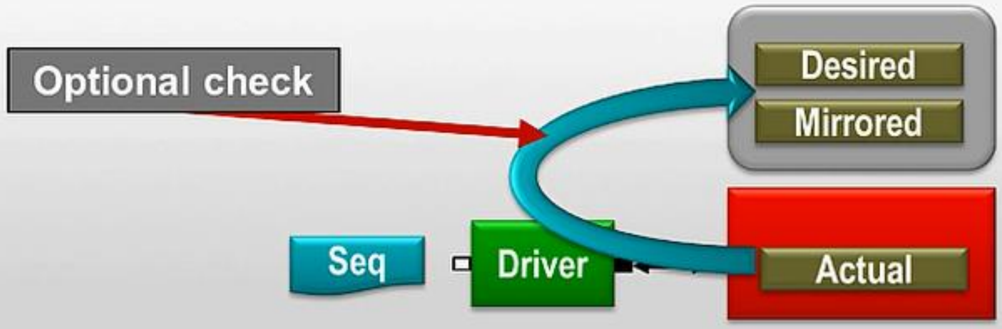
- **Read register and update/check mirror value**
  - via frontdoor

```
mirror_reg(model.ctrl, status, UVM_CHECK, UVM_FRONTDOOR);
```

  - or backdoor

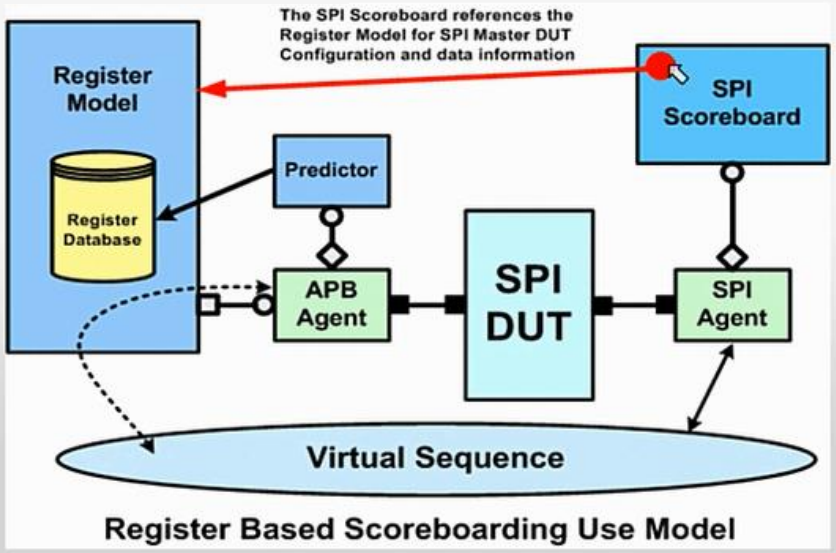
```
mirror_reg(model.ctrl, status, UVM_CHECK, UVM_BACKDOOR);
```

- **Can be called on field, reg or block**

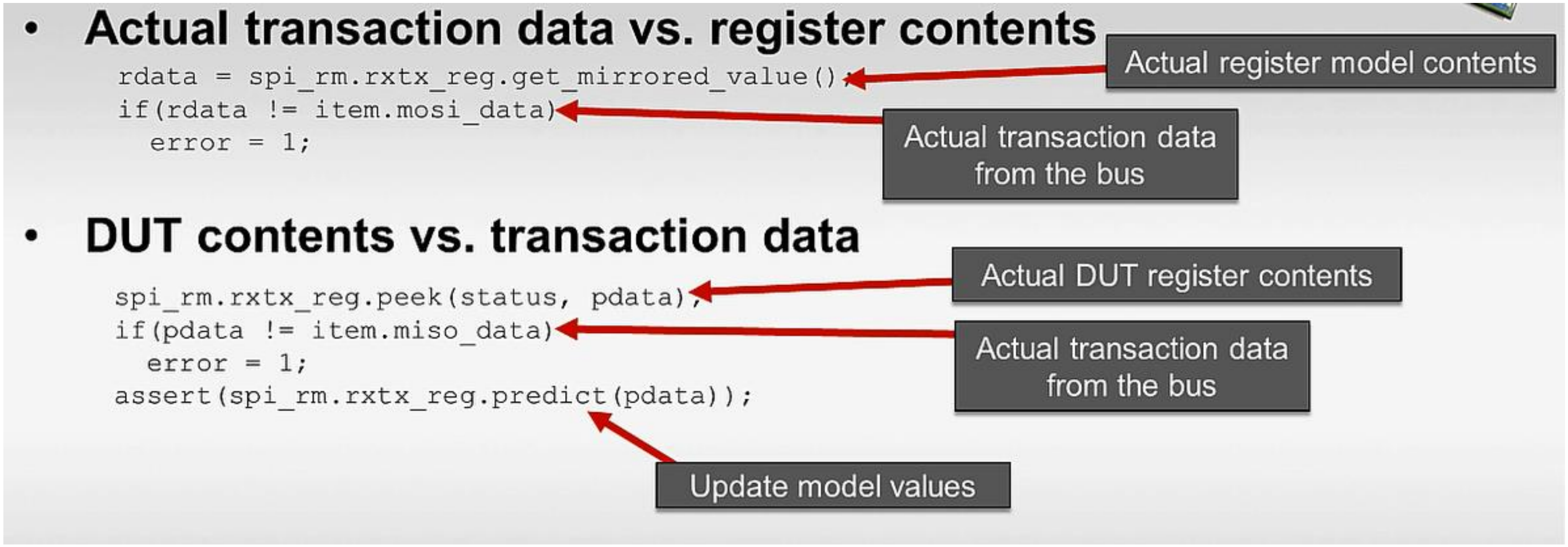


Register Scoreboard Guidelines

- **Scoreboard needs a handle to the register model**
- **Scoreboard accesses register values from model**
- **Scoreboard checks DUT register contents**
  - Compare observed data vs. register model contents
  - Compare DUT contents vs. expected
    - via peek access to DUT


The SPI Scoreboard references the Register Model for SPI Master DUT Configuration and data information
Register Model
Register Database
Predictor
SPI Scoreboard
APB Agent
SPI DUT
SPI Agent
Virtual Sequence
Register Based Scoreboarding Use Model

Scoreboard Checking



- **Actual transaction data vs. register contents**

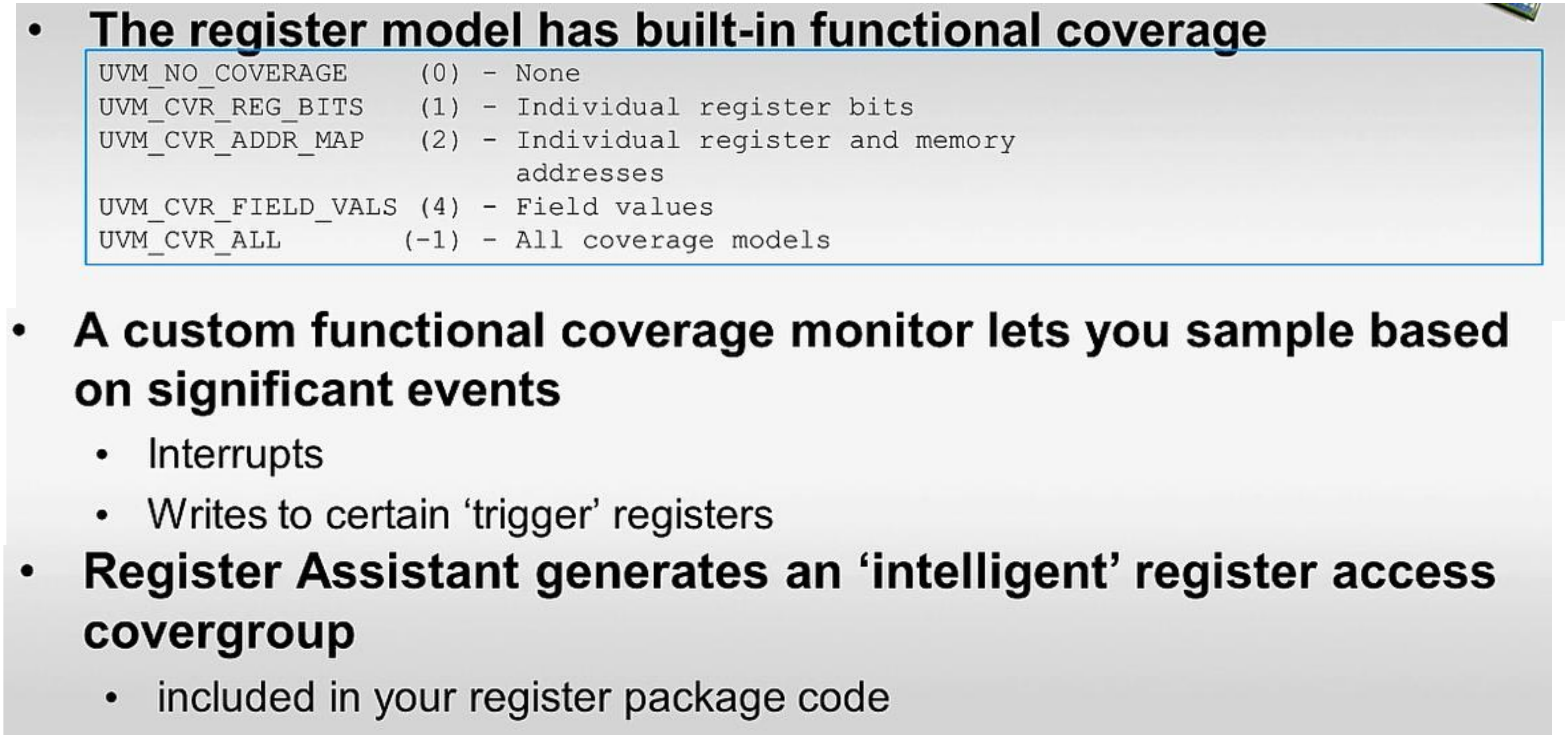
```
rdata = spi_rm.rxtx_reg.get_mirrored_value();
if(rdata != item.mosi_data)
  error = 1;
```

- **DUT contents vs. transaction data**

```
spi_rm.rxtx_reg.peek(status, pdata);
if(pdata != item.miso_data)
  error = 1;
assert(spi_rm.rxtx_reg.predict(pdata));
```

Actual register model contents
Actual transaction data from the bus
Actual DUT register contents
Actual transaction data from the bus
Update model values

Functional Coverage Monitors

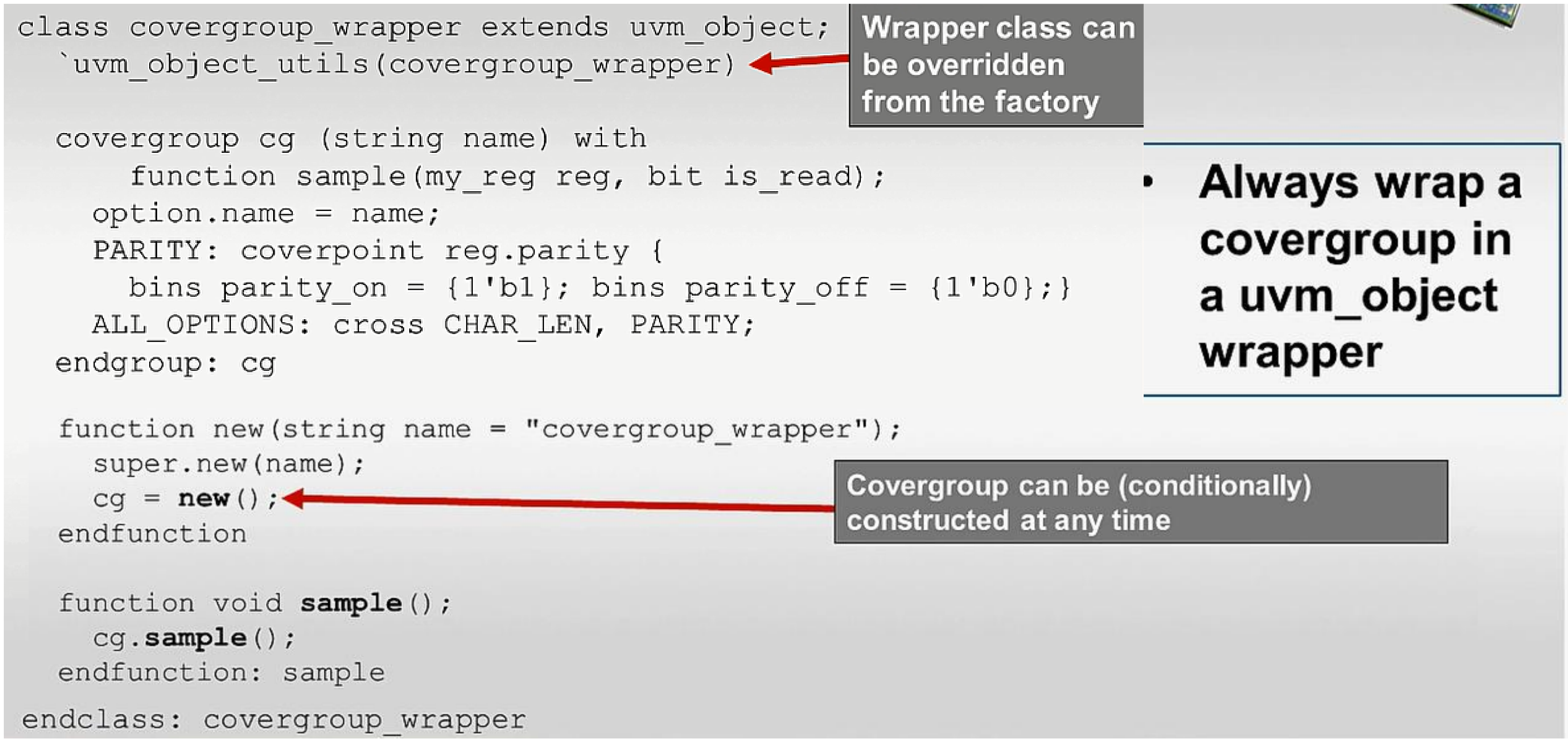


- **The register model has built-in functional coverage**

```
UVM_NO_COVERAGE     (0) - None
UVM_CVR_REG_BITS    (1) - Individual register bits
UVM_CVR_ADDR_MAP    (2) - Individual register and memory
                          addresses
UVM_CVR_FIELD_VALS (4) - Field values
UVM_CVR_ALL        (-1) - All coverage models
```

- **A custom functional coverage monitor lets you sample based on significant events**
  - Interrupts
  - Writes to certain ‘trigger’ registers
- **Register Assistant generates an ‘intelligent’ register access covergroup**
  - included in your register package code

Coding Guideline



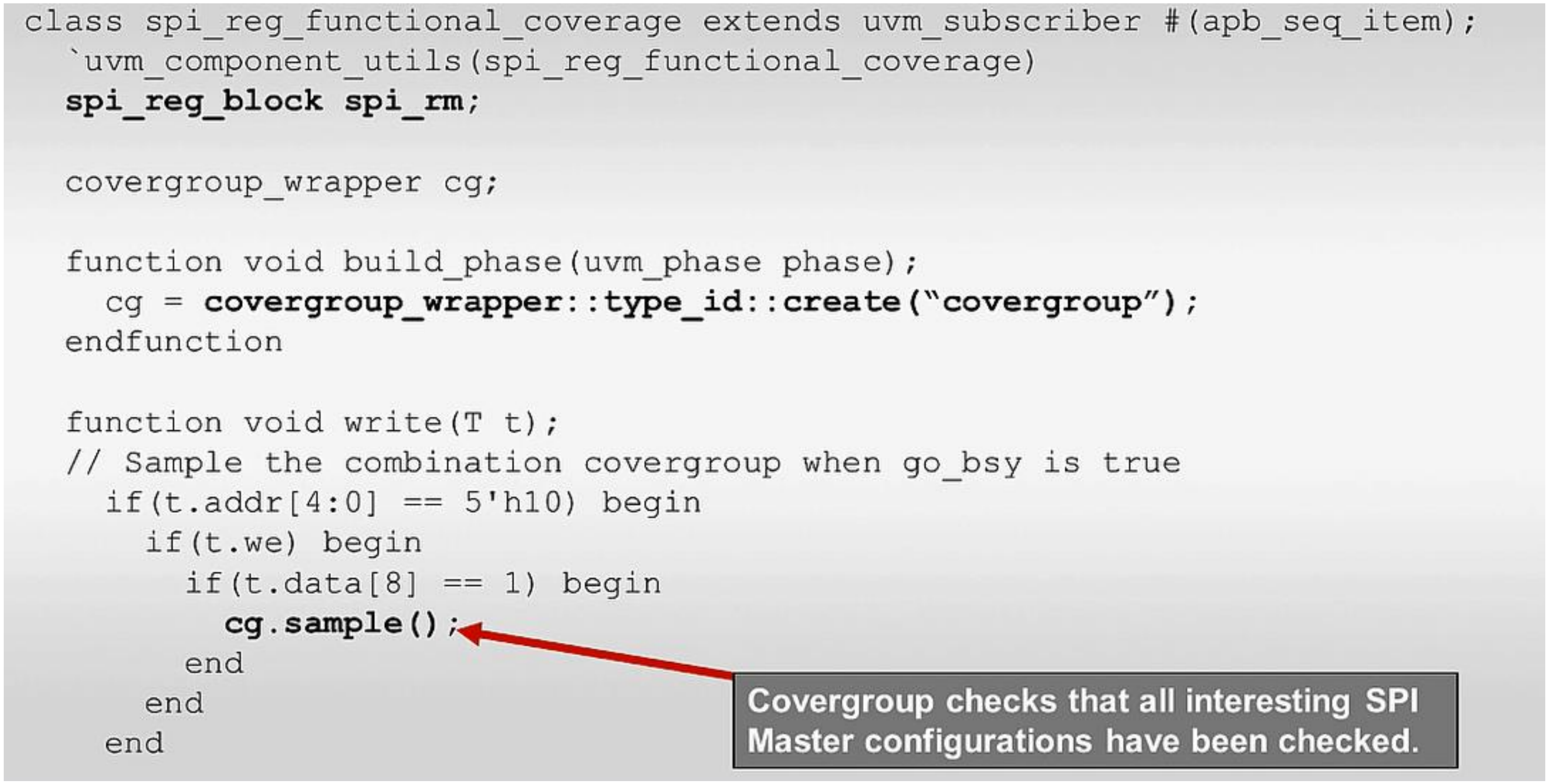
```
class covergroup_wrapper extends uvm_object;
  `uvm_object_utils(covergroup_wrapper)

  covergroup cg (string name) with
      function sample(my_reg reg, bit is_read);
    option.name = name;
    PARITY: coverpoint reg.parity {
      bins parity_on = {1'b1}; bins parity_off = {1'b0};}
    ALL_OPTIONS: cross CHAR_LEN, PARITY;
  endgroup: cg

  function new(string name = "covergroup_wrapper");
    super.new(name);
    cg = new();
  endfunction

  function void sample();
    cg.sample();
  endfunction: sample
endclass: covergroup_wrapper
```

Wrapper class can be overridden from the factory

- **Always wrap a covergroup in a uvm_object wrapper**

Covergroup can be (conditionally) constructed at any time

Monitor with Wrapped Covergroup



```
class spi_reg_functional_coverage extends uvm_subscriber #(apb_seq_item);
  `uvm_component_utils(spi_reg_functional_coverage)
  spi_reg_block spi_rm;

  covergroup_wrapper cg;

  function void build_phase(uvm_phase phase);
    cg = covergroup_wrapper::type_id::create("covergroup");
  endfunction

  function void write(T t);
  // Sample the combination covergroup when go_bsy is true
    if(t.addr[4:0] == 5'h10) begin
      if(t.we) begin
        if(t.data[8] == 1) begin
          cg.sample();
        end
      end
    end
```

Covergroup checks that all interesting SPI Master configurations have been checked.