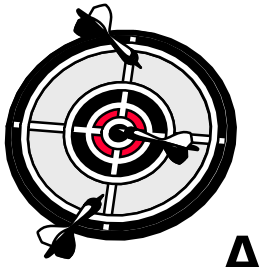
Agenda: Day 3

DAY 3

9	UVM Advanced Sequence/Sequencer
10	UVM Phasing and Objections 
11	UVM Register Abstraction Layer (RAL)
12	Summary 

Unit Objectives

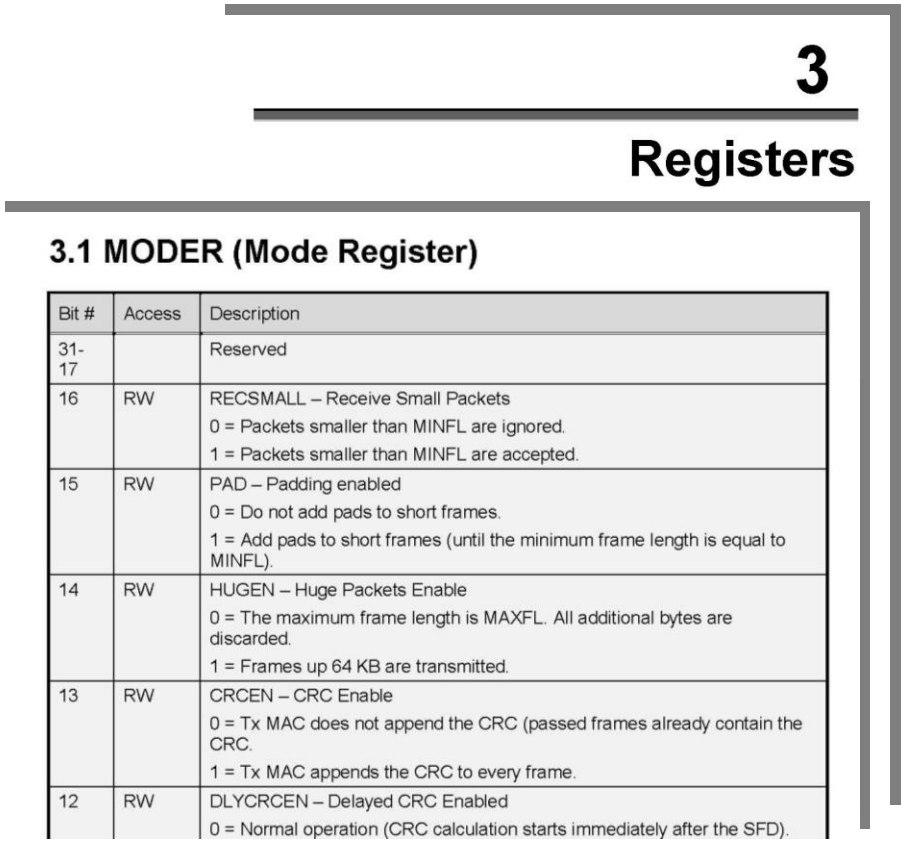


After completing this unit, you should be able to:

- **Create ralf file to represent DUT registers**
- **Use ralgen to create UVM register classes**
- **Use UVM register in sequences**
- **Implement adapter to pass UVM register content to drivers**
- **Run built-in UVM register tests**

Register & Memories

- **Every DUT has them**
- **First to be verified**
  - Reset value
  - Bit(s) behavior
- **High maintenance**
  - Modify tests
  - Modify firmware model



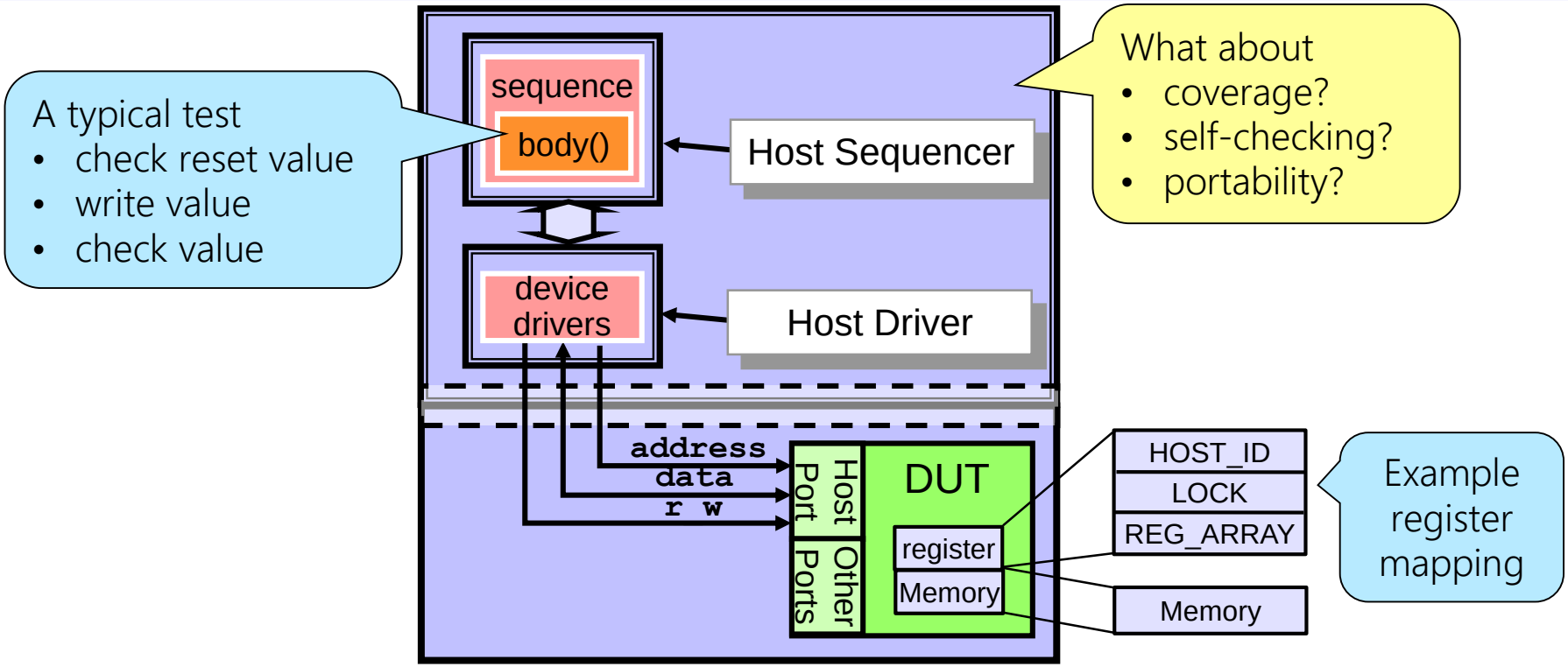
3

Registers

3.1 MODER (Mode Register)

Bit #	Access	Description
31-17		Reserved
16	RW	RECSMALL – Receive Small Packets 0 = Packets smaller than MINFL are ignored. 1 = Packets smaller than MINFL are accepted.
15	RW	PAD – Padding enabled 0 = Do not add pads to short frames. 1 = Add pads to short frames (until the minimum frame length is equal to MINFL).
14	RW	HUGEN – Huge Packets Enable 0 = The maximum frame length is MAXFL. All additional bytes are discarded. 1 = Frames up 64 KB are transmitted.
13	RW	CRCEN – CRC Enable 0 = Tx MAC does not append the CRC (passed frames already contain the CRC. 1 = Tx MAC appends the CRC to every frame.
12	RW	DLYCRCEN – Delayed CRC Enabled 0 = Normal operation (CRC calculation starts immediately after the SFD).

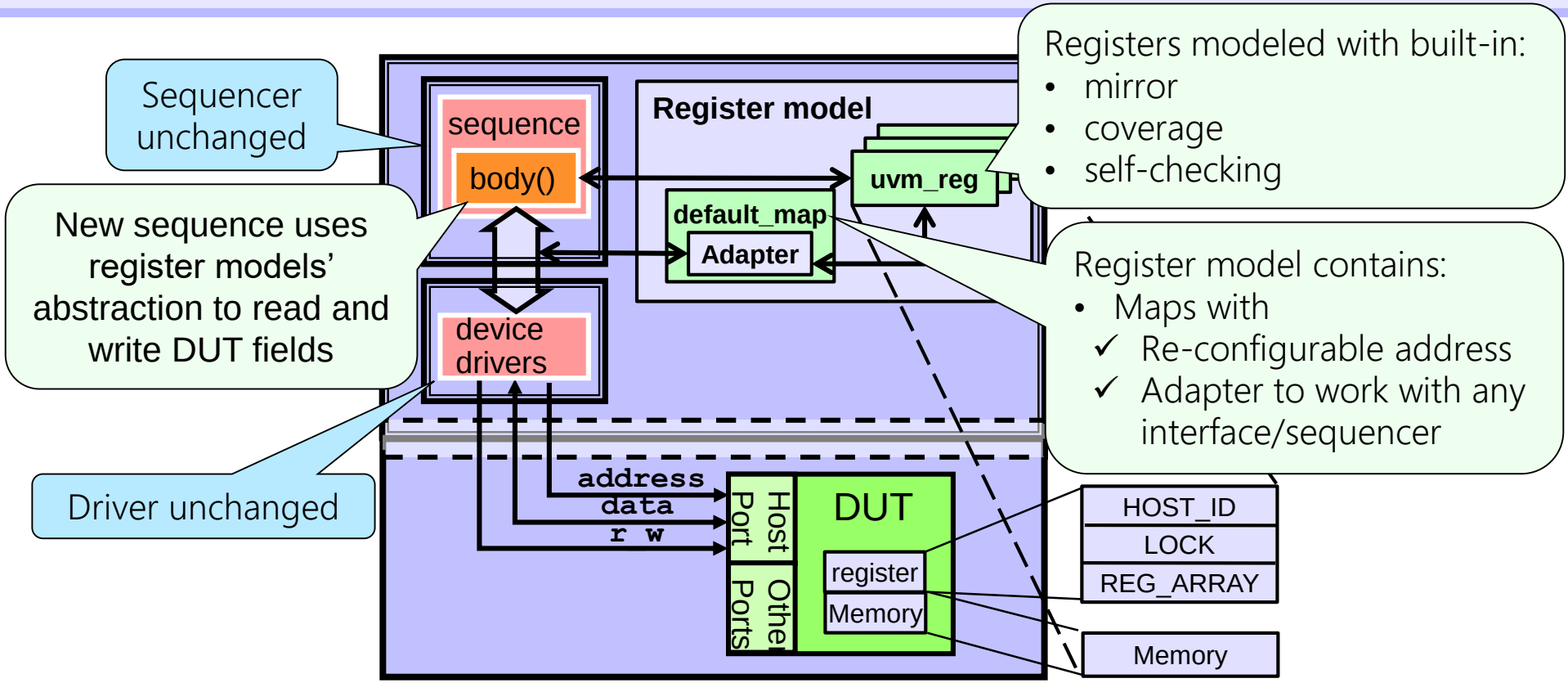
Testbench without UVM Register Abstraction




```
class host_sequence extends uvm_sequence #(host_data);  ...;
  virtual task body();
    `uvm_do_with(req, {addr=='h100; data=='1; kind==UVM_WRITE;});
    `uvm_do_with(req, {addr=='h1025; kind==UVM_READ;});  ...;
  endtask
endclass
```

Address hardcoded!
Field name unknown!
Status of execution unknown!
Front door access only!

Testbench with UVM Register Abstraction




```
class host_sequence extends uvm_reg_sequence#(uvm_sequence#(host_data));
  virtual task body(); uvm_status_e status; uvm_reg_data_t data;
    regmodel.HOST_ID.read(status, data, UVM_BACKDOOR, .parent(this));
    regmodel.RAM.write(status, 25, '1, UVM_FRONTDOOR, .parent(this));
  endtask
en
```

Access via reg/mem names
Status of execution returned

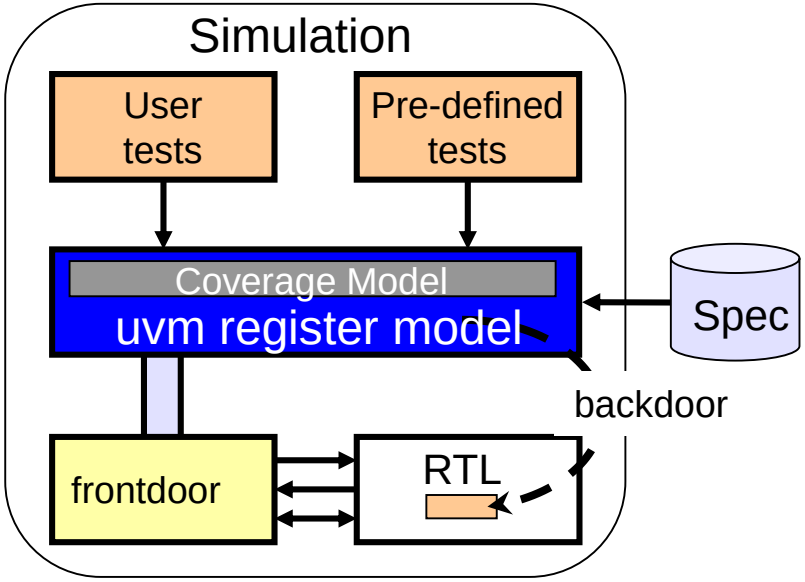
memory access specifies offset address

Access via front or back door

For debuggin

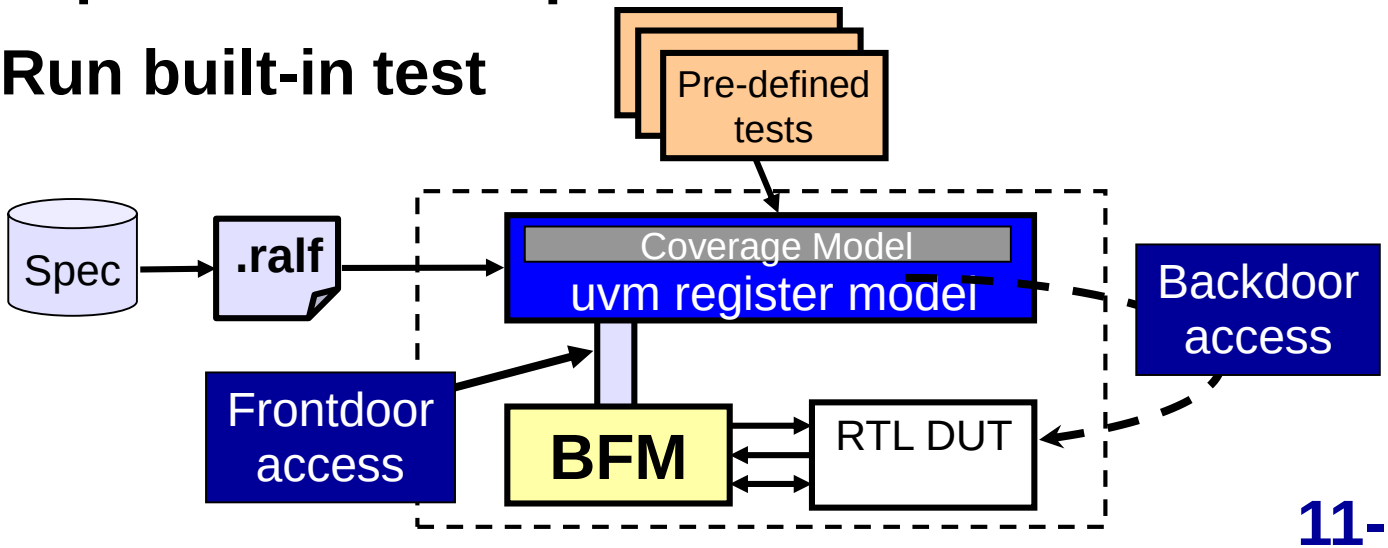
UVM Register Abstraction

- Abstracts reading/writing to configuration fields and memories
- Supports both front door and back door access
- Mirrors register data
- Built-in functional coverage
- Hierarchical model for ease of reuse
- Pre-defined tests exercise registers and memories

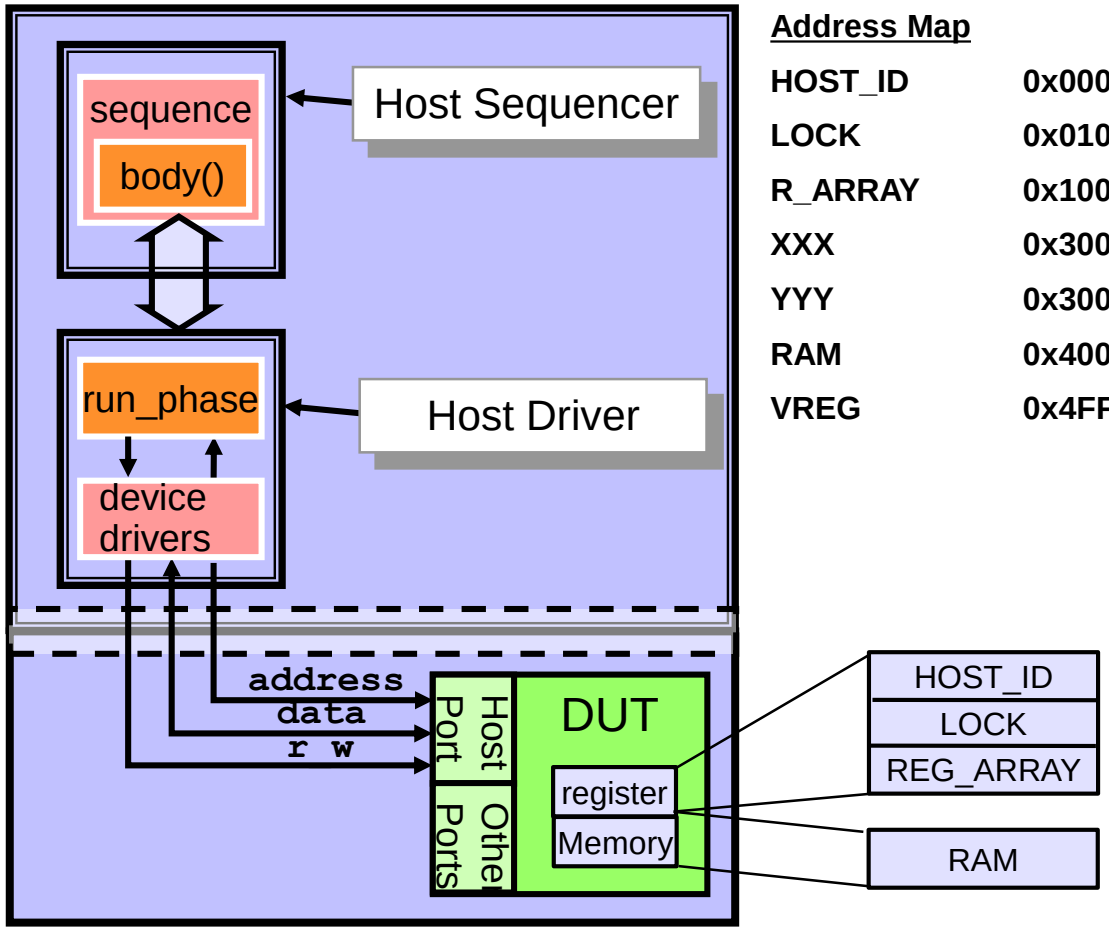


Implement UVM Register Abstraction

- Step 1: Verify frontdoor without UVM register abstraction
- Step 2: Describe register fields in .ralf file
- Step 3: Use ralgen to create UVM register abstraction
- Step 4: Create UVM register abstraction adapter
- Step 5: Add UVM register abstraction in environment
- Step 6: Write and run UVM register abstraction sequence
- Optional: Implement mirror predictor
- Optional: Run built-in test



Example Specification



Address Map

HOST_ID	0x0000
LOCK	0x0100
R_ARRAY	0x1000-0x10FF
XXX	0x3000
YYY	0x3001
RAM	0x4000-0x4FFF
VREG	0x4FF0-0x4FFF

HOST_ID Register		
Field	CHIP	REV
Bits	15-8	7-0
Mode	ro	ro
Reset	0x5A	0x03

LOCK Register	
Field	LOCK
Bits	15-0
Mode	w1c
Reset	0xffff

R_ARRAY[256] Registers	
Field	H_REG
Bits	15-0
Mode	rw
Reset	0x0000

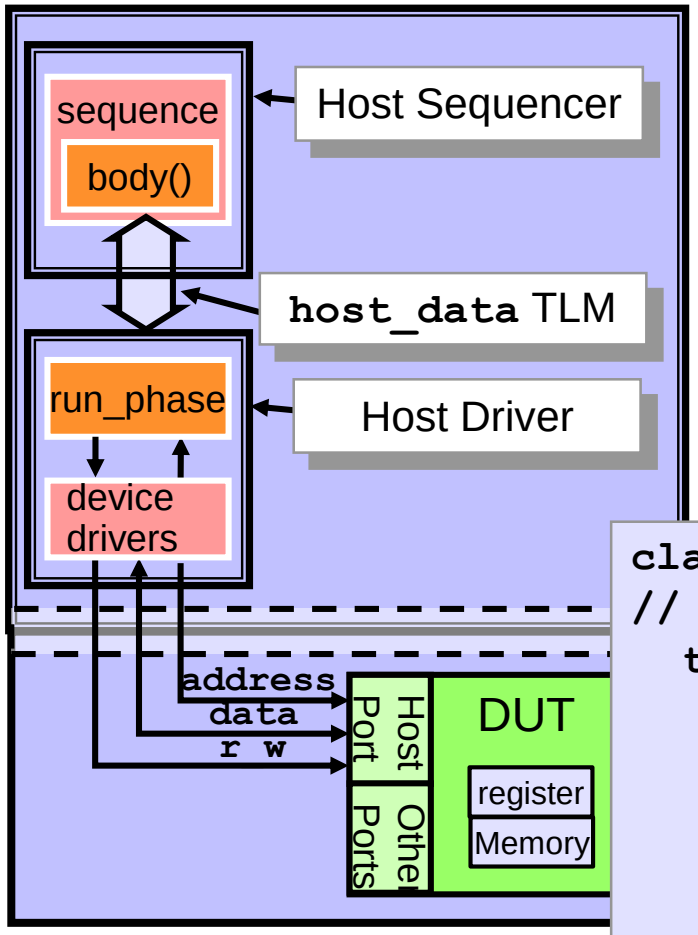
Register XXX

Register YYY

RAM (4K)	
Bits	15-0
Mode	rw

VREG[16]	
Bits	15-0

Step 1: Create Host Data & Driver



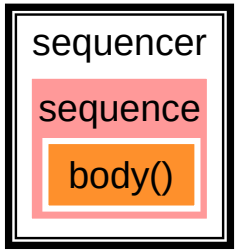

```
class host_data extends uvm_sequence_item;
// constructor and utils macro not shown
  rand uvm_access_e kind;
  uvm_status_e      status;
  rand bit[15:0]    addr, data;
endclass: host_data
```

Transaction specifies operation, address, data and status

```
class host_driver extends uvm_driver #(host_data);
// constructor and utils macro not shown
  task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      data_rw(req); // call device driver
      seq_item_port.item_done();
    end
  endtask
// device drivers not shown;
endclass: host_driver
```

Step 1: Create Host Sequence

- **Implement host sequence**




```
class host_bfm_sequence extends uvm_sequence #(host_data);
  // utils macro, constructor, pre/post_start not shown
  task body();
    `uvm_do_with(req, {addr=='h000; kind==UVM_READ;});
    `uvm_do_with(req, {addr=='h100; data=='1; kind==UVM_WRITE;});
    `uvm_do_with(req, {addr=='h1025; kind==UVM_READ;});
  endtask
endclass
```

Sequence hardcodes register addresses
Access DUT through front door

Verify Frontdoor Host is Working

- **Follow UVM guideline and complete test structure**
- **Then, compile and run simulation**

```
class host_agent extends uvm_agent; // support code not shown
  typedef uvm_sequencer #(host_data) host_sequencer;
  host_driver drv; host_monitor mon; host_sequencer sqr;
endclass
```

```
class host_env extends uvm_env; // other support code not shown
 host_agent h_agt;
 function void build_phase(uvm_phase phase);
  // super.build_phase and construction of components not shown
  uvm_config_db#(uvm_object_wrapper)::set(this,"h_agt.configure_phase",
                  "default_sequence", host_bfm_sequence::get_type());
 endfunction
endclass
```

```
class test_base extends uvm_test;    // other support code not shown
 host_env h_env; function void build_phase(uvm_phase phase);
  // super.build_phase and construction of components not shown
   uvm_config_db#(virtual host_io)::set(this, "h_env.h_agt", "vif",
                                        router_test_top.host_io);
 endfunction
endclass
```

Step 2: Create .ralf File Based on Spec

```
register HOST_ID {
  field REV_ID {
    bits   8;
    access ro;
    reset  'h03;
  }
  field CHIP_ID {
    bits   8;
    access ro;
    reset  'h5A;
}}
```

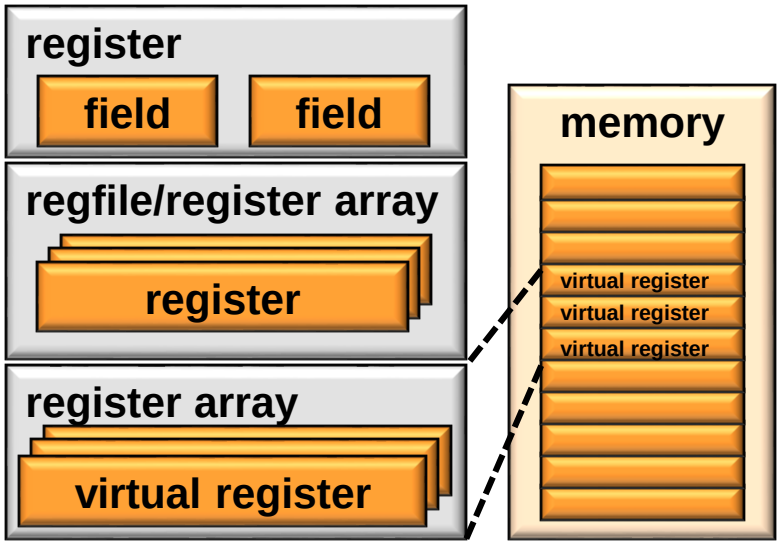
```
register LOCK {
  field LOCK {
    bits 16;
    access w1c;
    reset 'hffff;
}}
```

```
register R_ARRAY {
  field H_REG {
    bits 16;
    access rw;
    reset 'h0000;
}}
```

```
regfile REG_FILE {
  register XXX (xxx) @'h0 {…}
  register YYY (yyy) @'h1 {…}
}
```

RTL register name

```
memory RAM {
  size   4k;
  bits   16;
  access rw;
}
```



HOST_ID Register		
Field	CHIP_ID	REV_ID
Bits	15-8	7-0
Mode	ro	ro
Reset	0x5A	0x03

LOCK Register	
Field	LOCK
Bits	15-0
Mode	w1c
Reset	0xffff

R_ARRAY[256] Registers	
Field	H_REG
Bits	15-0
Mode	rw
Reset	0x0000

Register XXX

Register YYY

RAM (4K)	
Bits	15-0
Mode	rw

VREG[16]	
Bits	15-0

Step 2: Create .ralf File Based on Spec

.ralf block emulates RTL block module level
Specifies all content of block

Register name in block

Address

```
block host_regmodel {
  bytes 2;
  register HOST_ID        (chip_id)          @'h0000;
  register LOCK           (lock)             @'h0100;
  register R_ARRAY[256]   (host_reg[%d])     @'h1000;
  register nested         (subblk.nested)    @'h2000;
  regfile  REG_FILE                          @'h3000;
  memory   RAM             (ram)             @'h4000;
  virtual register VREG[16]                  RAM @'h0FF0 {
    field VREG { bits 16; access rw; }
  }
}
```

Address Map

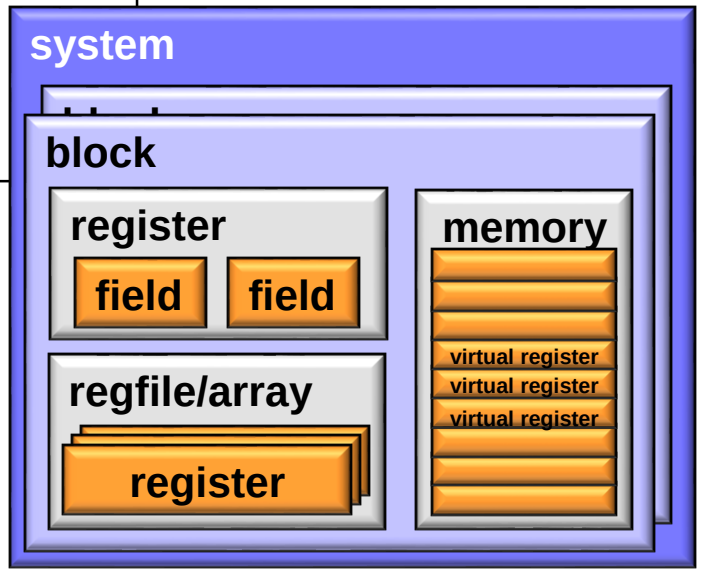
HOST_ID	0x0000
LOCK	0x0100
R_ARRAY	0x1000-0x10FF
XXX	0x3000
YYY	0x3001
RAM	0x4000-0x4FFF
VREG	0x4FF0-0x4FFF

Module instance name in DUT

Offset

```
system dut_regmodel {
  bytes 2;
  block host_regmodel       (host)   @'h0000;
  block other_regmodel      (other)  @'h8000;
}
```

.ralf system emulates upper layer module
which instantiates the block module

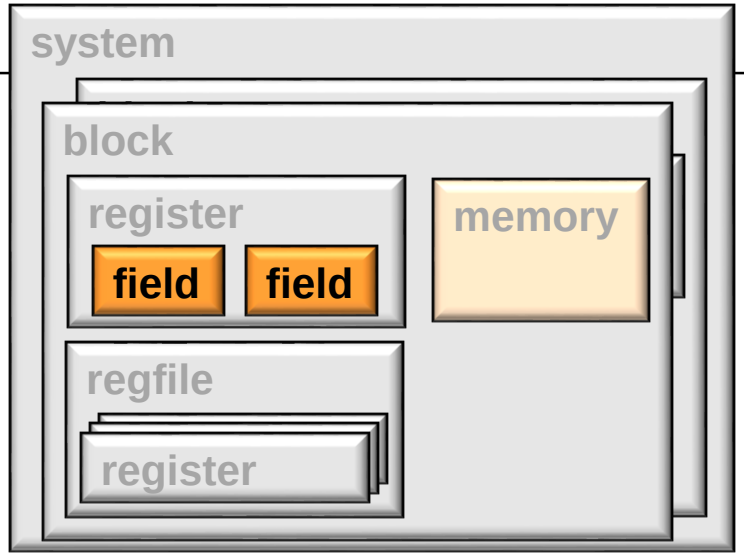


UVM Register Abstraction File (.ralf) Syntax

- **Field**
  - Contains Bits, Access, Reset, Constraints and Coverage

```
field field_name {
  bits n;
  access rw|ro|wo|w1|w0c|w1c|rc|...;
  reset value;
  [constraint name { <expressions> }]
  [enum { <name[=val],> }]
  [cover <+|- b|f>]
  [coverpoint {<bins name[[[n]]] = {<n|[n:n],>} | default>}]
}
```

```
field REV {
  bits   8;
  access ro;
  reset  'h03;
}
```



UVM Register Abstraction: Field

RAL field can have any of the following specifications

bits	Number of bits in the field
access	See next slide
reset	Specify the hard reset value for the field
constraint	Constraint to be used when randomizing field
enum	Define symbolic name for field values
cover	Specifies bits in fields are to be included (+b) in or excluded (-b) from the register-bit coverage model. Specifies coverpoint is a goal (+f). If not (-f) coverpoint weight will be zero.
coverpoint	Explicitly specifies the bins in coverpoint for this field

Field Access Types

RW	read-write
RO	read-only
WO	write-only; read error
W1	write-once
WO1	write-once; read error
W0C/S/T	write a 0 to bitwise-clear/set/toggle matching bits
W1C/S/T	write a 1 to bitwise-clear/set/toggle matching bits
RC/S	read clear /set all bits
WC/S	write clear /set all bits
WOC/S	write-only clear/set matching bits; read error
WRC/S	read clear/set all bits
WSRC [WCRS]	write sets all bits; read clears all bits [inverse]
W1SRC [W1CRS]	write one set matching bits; read clears all bits [inverse]
W0SRC [W0CRS]	write zero set matching bits; read clears all bits [inverse]
NOACCESS	no effect for write and read

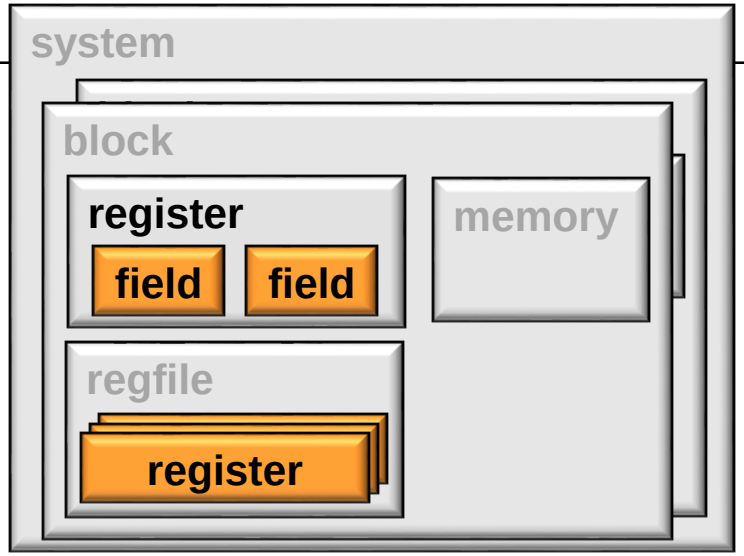
UVM Register Abstraction File (.ralf) Syntax

- **Register**
  - Contains fields

```
register reg_name {
  field name[=rename] [[n]] [@bit_offset[+incr]];
  field name[[n]] [(hdl_path)] [@bit_offset] {<properties>}
  [bytes n;]
  [left_to_right;]
  [<constraint name {<expression>}>]
  [shared [(hdl_path)];]
  [cover <+|- a|b|f>]
  [cross <cross_item1> … <cross_itemN>][{label <cross_label_name>}]
}
```

```
register HOST_ID {
  field REV_ID;
  field CHIP_ID;
}
```

```
register LOCK {
  bytes 2;
  field LOCK {
    access w1c;
    reset 'hffff;
}}
```



UVM Register Abstraction: Register

RAL registers can have any of the following specifications

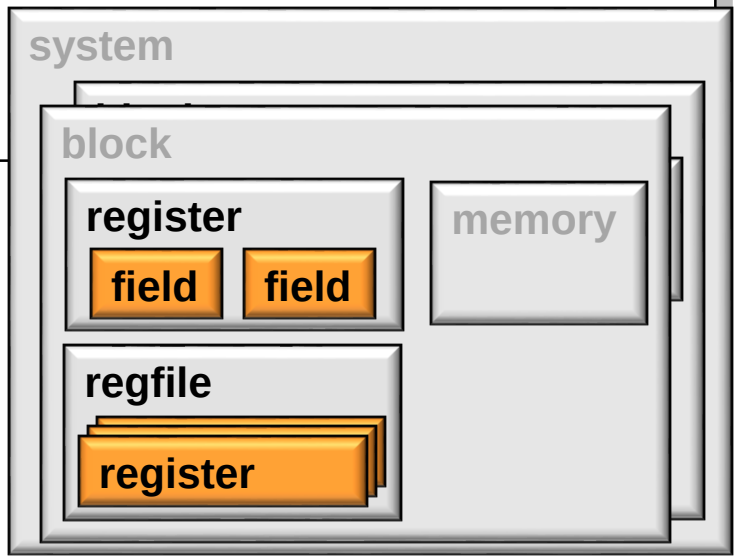
bytes	Number of bytes in the register
left_to_right	If specified, fields are concatenated starting from the most significant side of the register but justified to the least-significant side
[n]	Array of fields
bit_offset	Bit offset from the least-significant bit
incr	Array offset bit increment
hdl_path	Specify the module instance containing the register (backdoor access)
shared	All blocks instancing this register share the space
cover	Specifies if address of register should be excluded (-a) from the block's address map coverage model.
cross	Specifies cross coverage of two or more fields of coverpoints. The cross coverage can have a label.

UVM Register Abstraction File (.ralf) Syntax

- **Register File**
  - Contains register(s)

```
regfile regfile_name {
  register name[=rename][[n]][(hdl_path)][@offset];
  register name[[n]] [(hdl_path)] [@offset] {<property>}
  [<constraint name {<expression>}>]
  [shared [(hdl_path)];]
  [cover <+|- a|b|f>]
}
```

```
regfile REG_FILE {
  register XXX (xxx) @'h0 {
    field xxx {
      bits 16;
      access rw;
      reset 'h0000;
  }}
  register YYY (yyy) @'h1 {
    ...;
}}
```

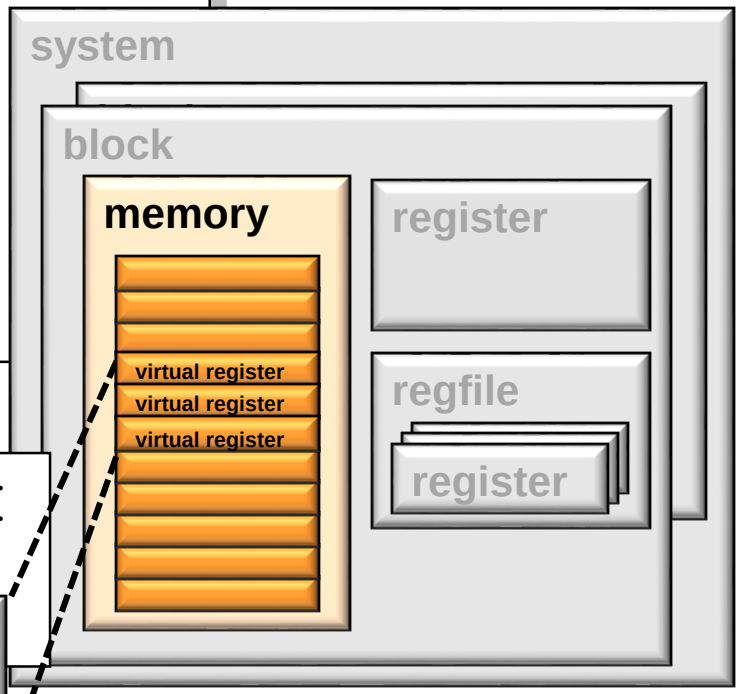


UVM Register Abstraction File (.ralf) Syntax

- **Memory**
  - Leaf level definition

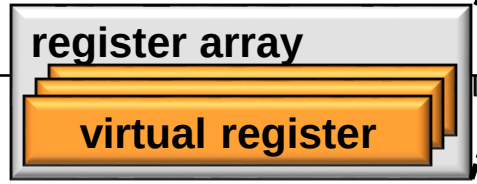
```
memory mem_name {
  size m[k|M|G];
  bits n;
  access rw|ro;
  [addr|literal[++|--];]
  [shared [(hdl_path)];]
}
virtual register reg_name {
  field field_name {
    bits n;  access rw|ro;
} }
```

```
memory RAM {
  size   4k;
  bits   16;
  access rw;
}
```



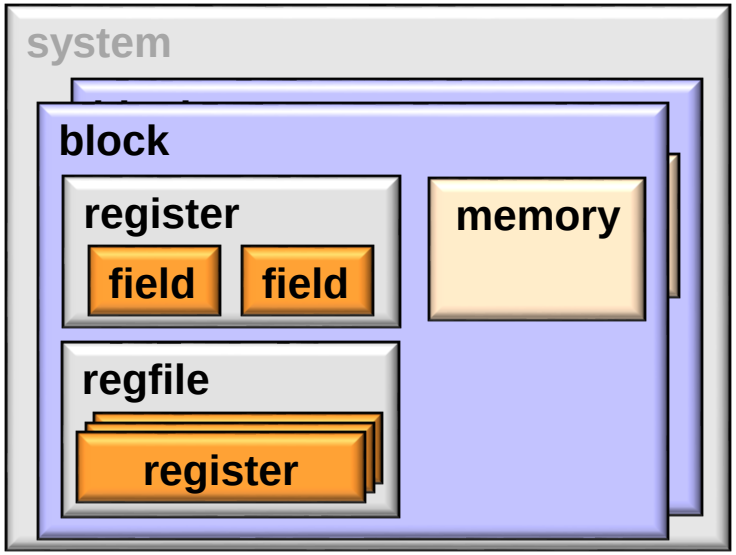
  - Emulated registers in memory

```
virtual register VREG[16] RAM @'h0FF0 {
  field VREG { bits 16; access rw; }
}
```



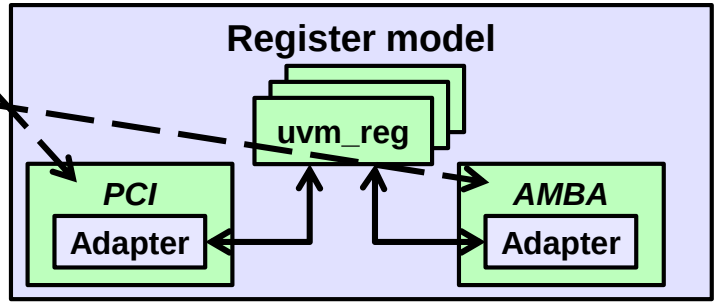
UVM Register Abstraction File (.ralf) Syntax

- **Blocks**
  - Defines content of block layer module
  - Can contain domains that define multiple interfaces accessing the same register (minimum of two if specified)
  - Can be instantiated in system




```
block blk_name {
   <property>
}
```

```
block blk_name {
   domain PCI {
      <property>
   }
   domain AMBA {
      <property>
   }
}
```



UVM Register Abstraction: Block

- **Contains register or memory elements of module**

```
block blk_name {
  bytes n;
  [endian no|little|big|fifo_ls|fifo_ms;]
  [<register name[=rename] [[n]][(hdl_path)] [@offset];>]
  [<register name[[n]][(hdl_path)][@offset] {<property>}]
  [<regfile name[=rename] [[n]] [(hdl_path)] [@offset] [+incr];>]
  [<regfile name[[n]][(hdl_path)][@offset] [+incr] {<property>}]
  [<memory name[=rename] [(hdl_path)] [@offset];>]
  [<memory name [(hdl_path)] [@offset] {<property>}>]
  [<constraint name {<expression>}>]
}
```

```
block host_regmodel {
  bytes 2;
  register HOST_ID                    @'h0000;
  register LOCK          (lock)       @'h0100;
  register R_ARRAY[256]  (host_reg[%d]) @'h1000;
  regfile  REG_FILE                   @'h3000;
  memory   RAM           (ram)        @'h4000;
  virtual register VREG[16]           RAM @'h0FF0 {
    field VREG { bits 16; access rw; }
} }
```

Must add index

Signal name in module

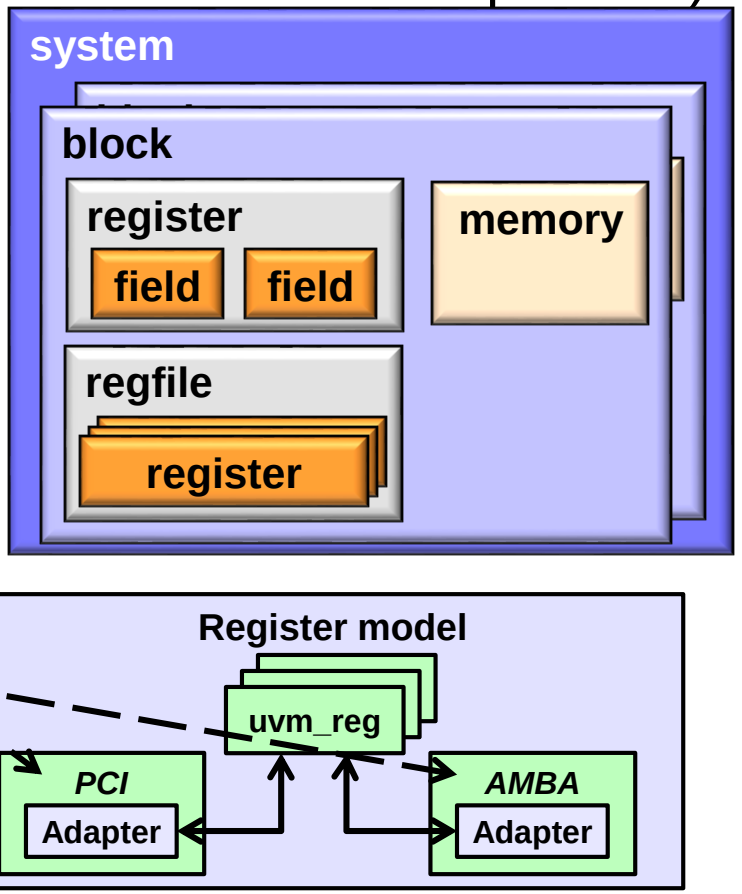
Register array

UVM Register Abstraction File (.ralf) Syntax

- **Top Level or subsystem**
  - Can contain domains that define multiple interfaces accessing the same register (minimum of two if specified)

```
system sys_name {
  <property>
}
```

```
system sys_name {
  domain PCI {
    <property>
  }
  domain AMBA {
    <property>
  }
}
```



UVM Register Abstraction: System

- **Top Level or subsystem**
  - Contains other systems or blocks

```
system sys_name {
bytes n;
endian no|little|big|fifo_ls|fifo_ms;]
[<block name[[.domain]=rename][[n]][(hdl_path)]@offset[+incr];>]
[<block name[[n]] [(hdl_path)] @offset [+incr] {<property>}>]
[<system name[[.domain]=rename][[n]][(hdl_path)]@offset[+incr];>]
[<system name[[n]][(hdl_path)] @offset [+incr] {<property>}]
[<constraint name {<expression>}>]
}
```

```
system dut_regmodel {
   bytes 2;
   block host_regmodel=HOST0 (blk0) @'h0000;
   block host_regmodel=HOST1 (blk1) @'h8000;
}
```

Step 3: Create UVM Register Abstraction Model

```
ralgen -uvm -t dut_regmodel host.ralf
```

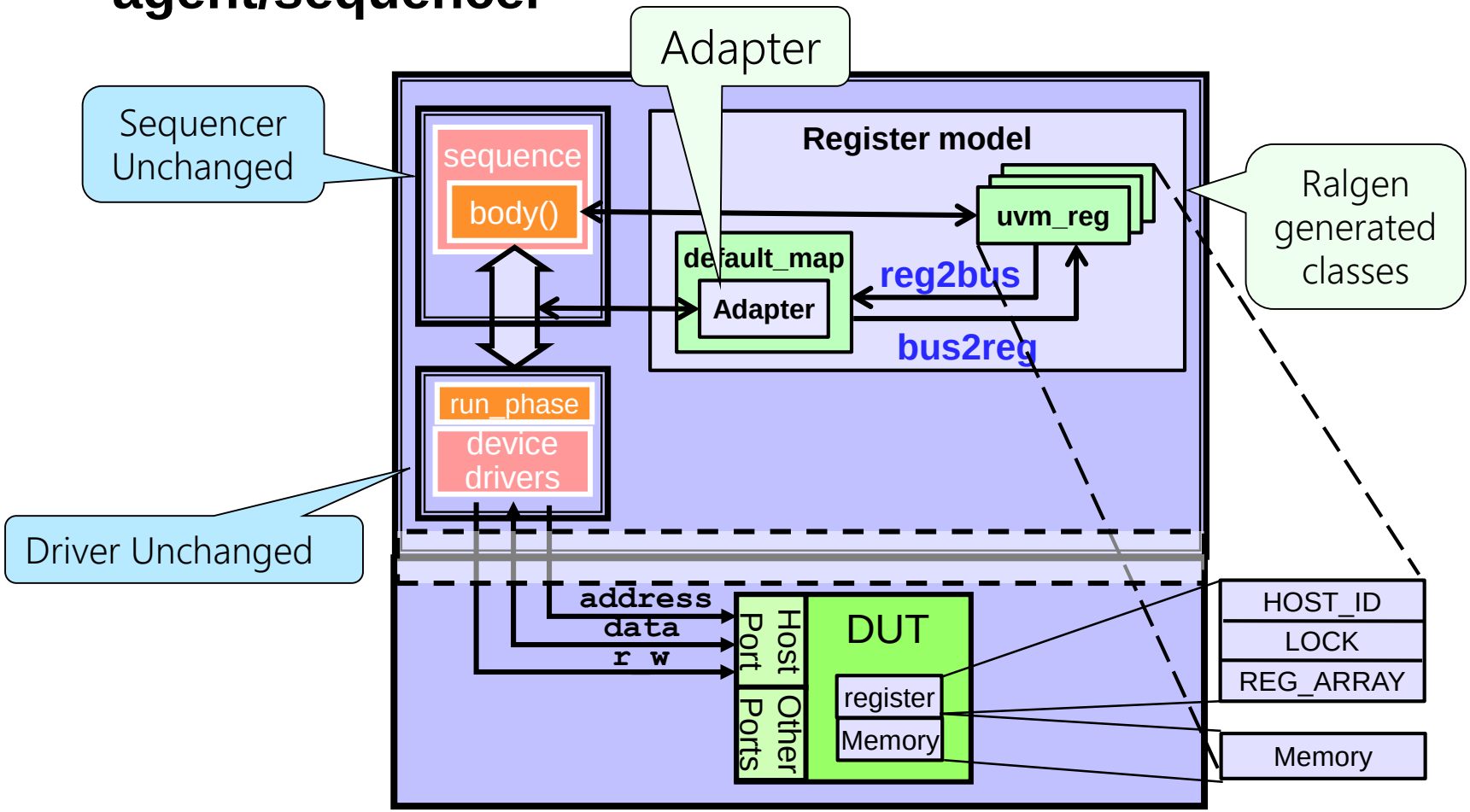
```
// host.ralf
register HOST_ID {
  field REV_ID  {…}
  field CHIP_ID {…}
}
register LOCK {
  field LOCK {…}
}
register R_ARRAY {
  field H_REG {…}
}
regfile REG_FILE {…}
memory RAM {…}
block host_regmodel {…}
system dut_regmodel {…}
```

```
// ral_dut_regmodel.sv
class ral_reg_HOST_ID extends uvm_reg;
  uvm_reg_field REV_ID;
  uvm_reg_field CHIP_ID;
  ...
endclass : ral_reg_HOST_ID
class ral_reg_LOCK extends uvm_reg;
class ral_reg_R_ARRAY extends uvm_reg;
class ral_regfile_REG_FILE extends uvm_regfile;
class ral_mem_RAM extends uvm_mem;
class ral_block_host_regmodel extends uvm_reg_block;
  rand ral_reg_HOST_ID       HOST_ID;
  rand ral_reg_LOCK          LOCK;
  rand ral_reg_R_ARRAY       R_ARRAY[256];
  rand ral_regfile_REG_FILE  REG_FILE;
  rand ral_mem_RAM           RAM;
  ...
endclass : ral_block_host_regmodel
class ral_sys_dut_regmodel extends uvm_reg_block;
  rand ral_block_host_regmodel HOST0;
  rand ral_block_host_regmodel HOST1;
  ...
endclass : ral_sys_dut_regmodel
```

UVM RAL Classes

Step 4: Create UVM Register Adapter

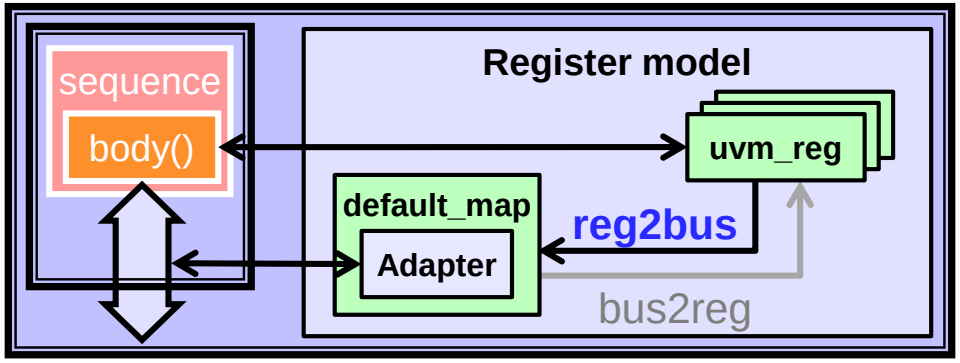
- **Environment needs adapters to convert UVM register data to bus transactions for each agent/sequencer**



Sequencer Adapter Class (1/2)

- **UVM abstracted registers need to be translated to what the driver can understand (`reg2bus`)**

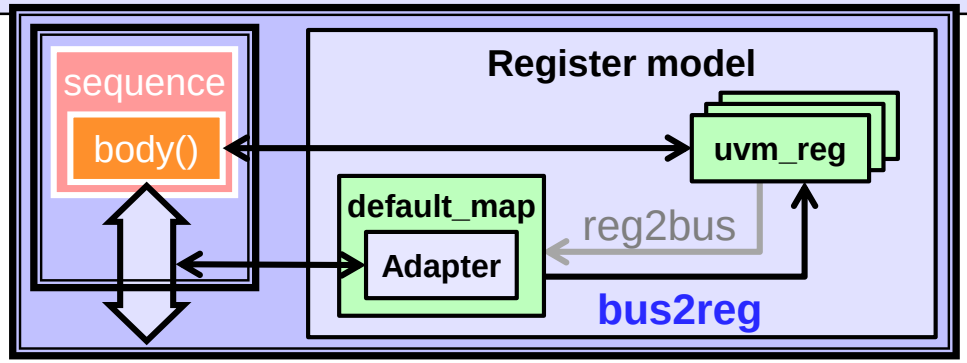
```
class reg_adapter extends uvm_reg_adapter;
 virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
   host_data tr;
   tr = host_data::type_id::create("tr");
   tr.addr = rw.addr;
   tr.data = rw.data;
   tr.kind = rw.kind;
   return tr;
 endfunction
// Continued on next slide
```



Sequencer Adapter Class (2/2)

- **bus2reg** translates bus transaction back to RAL

```
// Continued from previous slide
  virtual function void bus2reg(uvm_sequence_item bus_item,
                                ref uvm_reg_bus_op rw);
    host_data tr;
    if (!$cast(tr, bus_item)) `uvm_fatal(…);
    rw.addr   = tr.addr;
    rw.data   = tr.data;
    rw.kind   = tr.kind;
    rw.status = tr.status;
  endfunction
endclass
```



Optional Feature in Adapter Class

- **There is an optional field in the access methods (`write`, `read`, `peak`, `poke`) called `extension`**
  - Can be used to add additional information for populating the transaction to be passed on to the driver

```
task write(…, input uvm_object extension = null, …)
```

Example:

```
typedef enum { by_byte, by_word } host_access_t;
class host_ext extends uvm_object; // code left off
  rand host_access_t access_mode;
  constraint mode { soft access_mode == by_word; }
endclass
```

```
task ral_sequence::body();  host_ext ext = new();
  ext.randomize() with { access_mode == by_byte; };
  regmodel.R0.write(status, data, .parent(this), .extension(ext));
endtask
```

```
function uvm_sequence_item ral_adapter::reg2bus(...); ...//
  host_ext extension = host_ext'(get_item().extension);
  if (extension != null)
    host_data.access_mode = extension.access_mode;
 endfunction
```

Step 5: Instantiating UVM Register Model

- **Allow register model to be self constructed or passed in via configuration database**

```
class host_env extends uvm_env; // Other code not shown
 host_agent                h_agt;
 ral_block_host_regmodel   regmodel;
 virtual function void build_phase(uvm_phase phase); // Code simplified
  uvm_config_db#(ral_block_host_regmodel)::get(this, "",
                                               "regmodel", regmodel));
  if (regmodel == null) begin
   regmodel = ral_block_host_regmodel::type_id::create("regmodel", this);
   regmodel.build();
   regmodel.lock_model();

   uvm_config_db#(ral_block_host_regmodel)::set(this, "h_agt",
                                                "regmodel", regmodel);
  end
 endfunction: build_phase  // Continued on next slide
```

Create UVM register hierarchy. Not build_phase()!

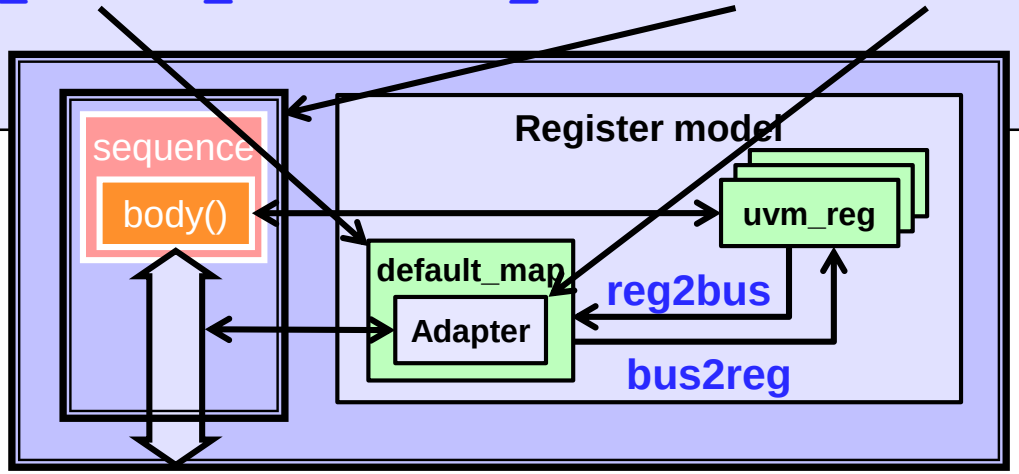
Lock register hierarchy and create address map

Pass register model to agent for sequence to pick up

Tie Sequencer Adapter to Register Map

- **Register each sequencer with the associated adapter in the UVM register model**

```
  virtual function void connect_phase(uvm_phase phase);
    reg_adapter adapter = reg_adapter::type_id::create("adapter", this);
    super.connect_phase(phase);
    regmodel.default_map.set_sequencer(h_agt.sqr, adapter);
  endfunction
endclass
```



- **A map within a register model represents an interface**
  - Registers with only one interface uses default_map
  - Registers with multiple interfaces use map names specified by user (domain name in ralgen)

UVM Register Abstraction Sequence

```
class bfm_sequence extends uvm_sequence#(host_data);
  // other code not shown
  virtual task body();
    `uvm_do_with(req, {addr=='h0; kind==UMV_READ;});
    `uvm_do_with(req, {addr=='h4009; data=='1; kind==UVM_WRITE;});
  endtask
endclass
```

Becomes

```
class ral_sequence extends uvm_reg_sequence #(uvm_sequence#(host_data));
  ral_block_host_regmodel regmodel;     // other code not shown
  virtual task pre_start();
    super.pre_start();
    uvm_config_db#(ral_block_host_regmodel)::get(
          get_sequencer().get_parent(), "", "regmodel", regmodel))

  endtask
  virtual task body(); uvm_status_e status;  uvm_reg_data_t data;
    regmodel.HOST_ID.read(status, data, .parent(this)); // can specify .path
    regmodel.RAM.write(status, 9, '1, .parent(this));    // defaults to frontdoor
  endtask                                                // if not specified
endclass
```

Retrieve regmodel from database via agent

Abstracted and self-documenting code

Run RAL Sequence Implicitly or Explicitly

```
class test_ral_implicit extends test_ral_base; // Support code not shown
 virtual function void build_phase(uvm_phase phase); // Other code
  uvm_config_db#(uvm_object_wrapper)::set(this,"*.sqr.configure_phase",
                    "default_sequence", host_ral_seqeuence::get_type());
 endfunction
endclass
```

Execute RAL sequence implicitly

```
class test_ral_explicit extends test_ral_base; // Support code not shown
  // Not shown-in start_of_simulation_phase:set default_sequence to null
  virtual task configure_phase(uvm_phase phase);
    host_ral_sequence h_seq;    super.configure_phase(phase);
    phase.raise_objection(this);
    h_seq = host_ral_sequence::type_id::create("h_seq", this);
    h_seq.start(env.h_agt.sqr);
    phase.drop_objection(this);
  endtask
endclass
```

Or, execute RAL sequence explicitly

Optional: Backdoor Access

- **Two ways to generate the backdoor access:**
  - Via SystemVerilog Cross Module Reference (XMR)
  - Via SystemVerilog DPI call
- **Both implementations allow register model to be part of SystemVerilog package**
- **XMR implementation is faster, but requires user to compile one additional file and at compile-time provide top level path to DUT**
- **DPI implementation is slower, but no additional file is needed and top level path to DUT can be provided at run-time via user implemented configuration settings**

ralgen XMR Backdoor Access (1/3)

- **-b -gen_vif_bkdr option generates an interface accessing register via Verilog XMR**

```
// host.ralf
  ...
  register HOST_ID (host_id) @'h0000;
```

```
ralgen -b -gen_vif_bkdr -uvm -t dut_regmodel host.ralf
```

```
// ral_host_regmodel_intf.sv
interface ral_host_regmodel_intf;
  // other code not shown
  initial
    uvm_resource_db#(virtual ral_host_regmodel_intf)::set("*",
                        "uvm_reg_bkdr_if", interface::self());
  task ral_host_regmodel_HOST_ID_bkdr_read(uvm_reg_item rw);
    rw.value[0] = `HOST_REGMODEL_TOP_PATH.host_id;
  endtask
endinterface
```

Self stored into database (VCS only – see note)

Access method uses XMR

Interface must be compiled

```
vcs ral_host_regmodel_intf.sv \
    +define+HOST_REGMODEL_TOP_PATH=router_test_top.dut ...
```

XMR path from harness to DUT must be specified

ralgen XMR Backdoor Access (2/3)

- **Generated backdoor class uses interface in to access registers**

Backdoor access class is generated by ralgen

```
// ral_dut_regmodel.sv
class ral_reg_host_regmodel_HOST_ID_bkdr extends uvm_reg_backdoor;
 // other code not shown
 virtual ral_host_regmodel_intf __reg_vif;
 function new(string name);
  super.new(name);
  uvm_resource_db#(virtual ral_host_regmodel_intf)::read_by_name(get_full_name(),
                                                  "uvm_reg_bkdr_if", __reg_vif);
 endfunction

 virtual task read(uvm_reg_item rw);
  do_pre_read(rw);
  __reg_vif.ral_host_regmodel_HOST_ID_bkdr_read(rw);
  rw.status = UVM_IS_OK;
  do_post_read(rw);
 endtask
endclass
```

XMR encapsulated in package-able interface

Interface retrieved via resource database (see note)

Register access method make use of interface access method

ralgen XMR Backdoor Access (3/3)

- XMR access method flow:

```
regmodel.HOST_ID.read(status, data, UVM_BACKDOOR, .parent(this));
```

```
task uvm_reg::read(…);  ...
  XreadX(status, value, path, map, parent, prior, extension, fname, lineno);
endtask: read
```

```
task uvm_reg::XreadX(…);  ...
  do_read(rw);
endtask
```

```
task uvm_reg::do_read(uvm_reg_item rw);  ...
  case (rw.path)
    UVM_BACKDOOR: begin
      uvm_reg_backdoor bkdr = get_backdoor();
      if (bkdr != null)  bkdr.read(rw);
      else backdoor_read(rw);
  ...
endtask
```

Uses backdoor class generated by ralgen

```
task ral_reg_host_regmodel_HOST_ID_bkdr::read(uvm_reg_item rw);
  do_pre_read(rw);
  __reg_vif.ral_host_regmodel_HOST_ID_bkdr_read(rw);
  rw.status = UVM_IS_OK;
  do_post_read(rw);
endtask
```

Uses XMR built-in to ralgen generated interface

ralgen DPI Backdoor Access (1/2)

- No additional switch on the ralgen or vcs command

```
// host.ralf
  ...
  register HOST_ID (host_id) @'h0000;
```

```
ralgen -uvm -t dut_regmodel host.ralf
```

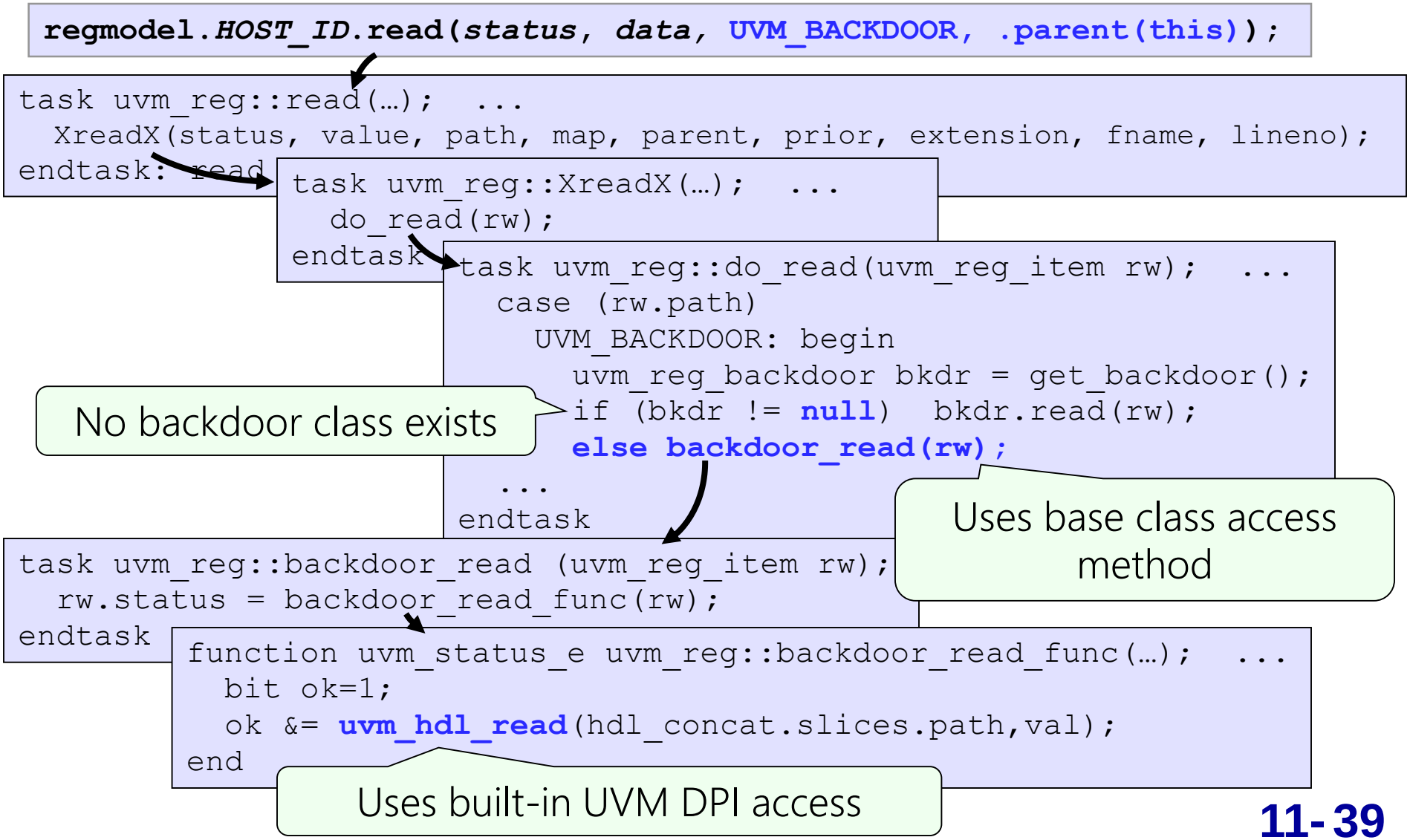
- No specialized backdoor interface generated
- Need to add top level path at run-time via database:

```
virtual function void test_ral_base::build_phase(uvm_phase phase);
  uvm_resource_db #(string)::set(env, "hdl_path","router_test_top.dut",
                                                              this);
endfunction
```

```
function void host_env::build_phase(uvm_phase phase);
  if (!uvm_resource_db#(string)::read_by_name(get_full_name(),
                                  "hdl_path", hdl_path, this))
    regmodel.set_hdl_path_root(hdl_path);
  else `uvm_fatal("host_regmodel", "top path is not set!");
endfunction
```

ralgen DPI Backdoor Access (2/2)

- **DPI access flow**



UVM Register Test Sequences

- **Some of test sequences depend on mirror to be updated when backdoor is used to access DUT registers**
  - You need to call **`set_auto_predict()`** in test or implement **`uvm_reg_predictor`**

Sequence Name	Description
uvm_reg_hw_reset_seq	Test the hard reset values of register
uvm_reg_bit_bash_seq	Bit bash all bits of registers
uvm_reg_access_seq	Verify accessibility of all registers
uvm_mem_walk_seq	Verify memory with walking ones algorithm
uvm_mem_access_seq	Verify access by using front and back door
uvm_reg_mem_built_in_seq	Run all reg and memory tests
uvm_reg_mem_hdl_paths_seq	Verify hdl_path for reg and memory
uvm_reg_mem_shared_access_seq	Verify accessibility of shared reg and memory

Execute RAL Test Sequence

- **Make sure RAL model is set to the correct prediction mode**

```
class test_ral extends test_base; // support code left off
  // code identical to auto predict test is left off
  virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this, "Starting reset tests");
    rst_seq = virtual_reset_sequence::type_id::create("rst_seq", this);
    rst_seq.start(env.v_reset_sqr);
    clp.get_arg_value("+seq=", seq_name);
    $cast(selftest_seq, uvm_factory::get().create_object_by_name(seq_name));
    env.regmodel.default_map.set_auto_predict(1);



    selftest_seq.model = env.regmodel;
    selftest_seq.start(env.h_agt.sqr);
    phase.drop_objection(this, "Done with register tests");
  endtask
endclass
```

0 – Manual prediction
1 – Auto prediction

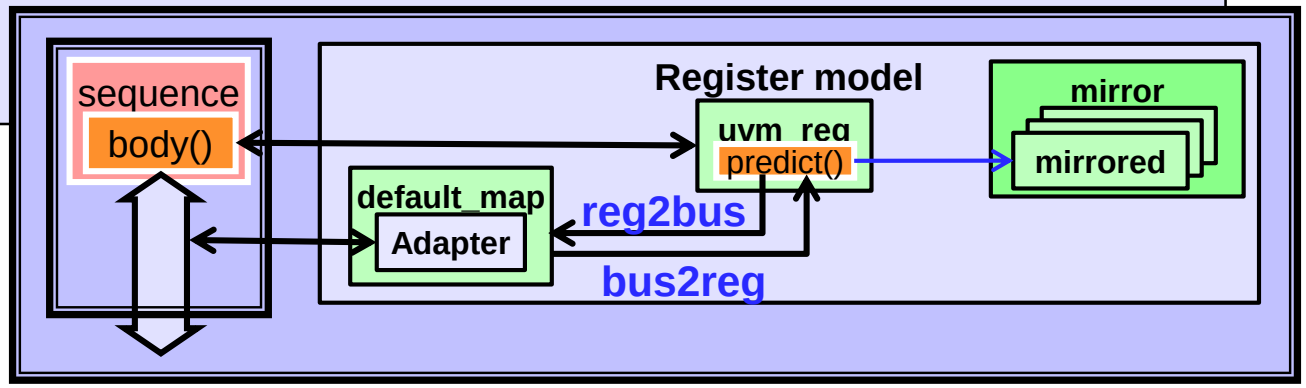
Select sequence at run-time

```
simv +UVM_TESTNAME=test_ral +seq=uvm_reg_hw_reset_seq
```

Enabling Auto Mirror Prediction

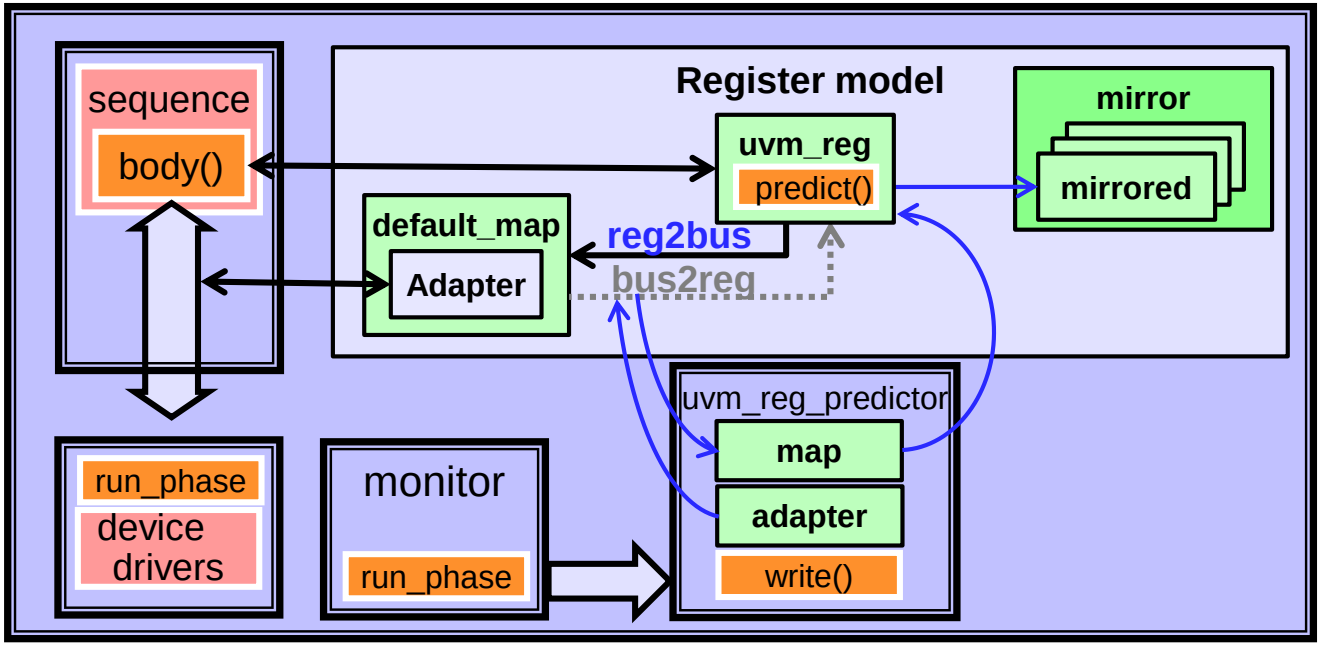
- **To enable automatic mirror update when registers are written and read: call `set_auto_predict(1)`**
  - Advantage:
    - Simple: Read, write method calls automatically updates mirror. No other user code required.
  - Drawbacks:
    - Timing of FRONTDOOR update is not cycle accurate
    - Changes internal to DUT is not reflected in mirror
    - Not a good choice if mirror value is needed in user tests

```
class router_env extends uvm_env; ...;
  virtual function void connect_phase(uvm_phase phase); ...;
    regmodel.default_map.set_auto_predict(1);
  endfunction
endclass
```



Manual (Explicit) Mirror Prediction

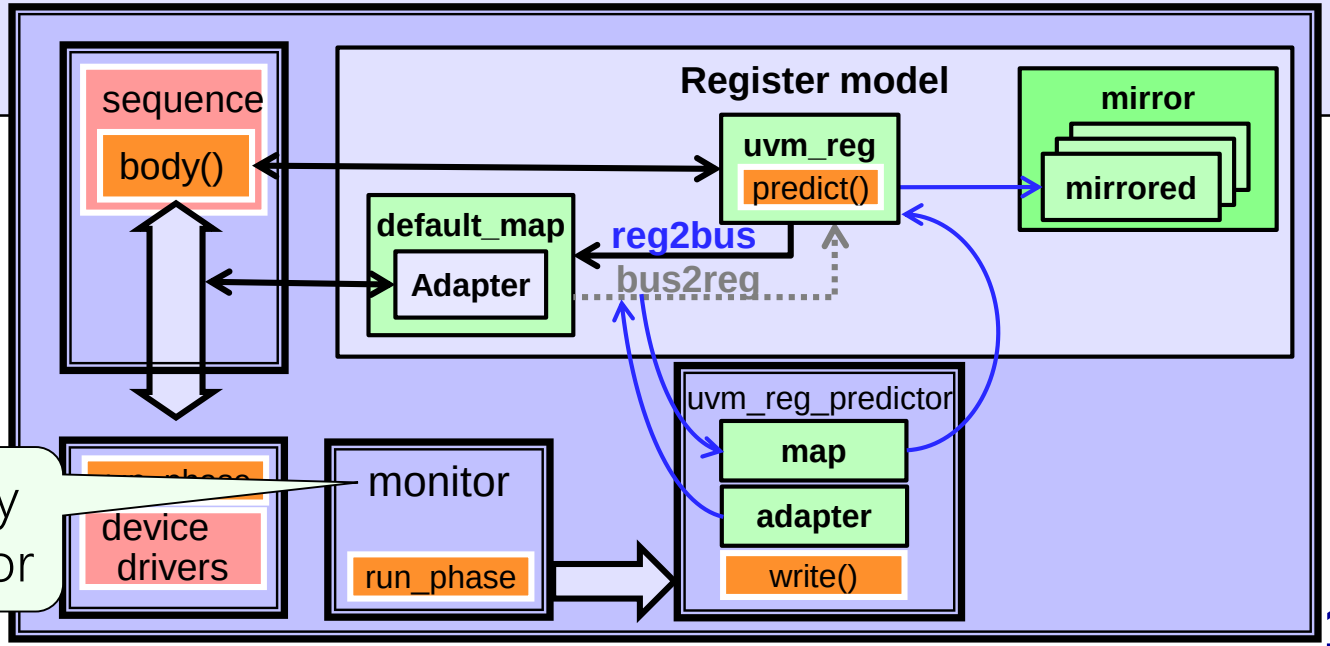
- **Mirror updates when monitor observe register changes**
  - Advantage:
    - Mirror is updated based on observation of bus protocol
    - Changes internal to DUT can be monitor and reflected in mirror
    - Correct choice if mirror value is needed in user tests
  - Drawbacks:
    - More complex to set up. Requires a monitor.



Connecting Explicit Mirror Predictor

```
class host_env extends uvm_env; // Other support code not shown
  typedef uvm_reg_predictor #(host_data) hreg_predictor;
  hreg_predictor hreg_predict;
  virtual function void build_phase(uvm_phase phase); // Other code
    hreg_predict = hreg_predictor::type_id::create("hreg_predict", this);
  endfunction
  virtual function void connect_phase(uvm_phase phase); // Other code
    hreg_predict.map = regmodel.get_default_map();
    hreg_predict.adapter = adapter;
    regmodel.default_map.set_auto_predict(0);
    h_agt.analysis_port.connect(hreg_predict.bus_in);
  endfunction
endclass
```

Disable auto predict



Complexity is in monitor

Unit Objectives Review

Having completed this unit, you should be able to:

- **Create ralf file to represent DUT registers**
- **Use ralgen to create UVM register classes**
- **Use UVM register in sequences**
- **Implement adapter to pass UVM register content to drivers**
- **Run built-in UVM register tests**

Appendix

UVM Register Modes

UVM Memory Modes

ralgen Options

UVM Register Class Tree

UVM Register/Memory Class Members

UVM Register Callbacks

Changing Address Offsets of a Domain

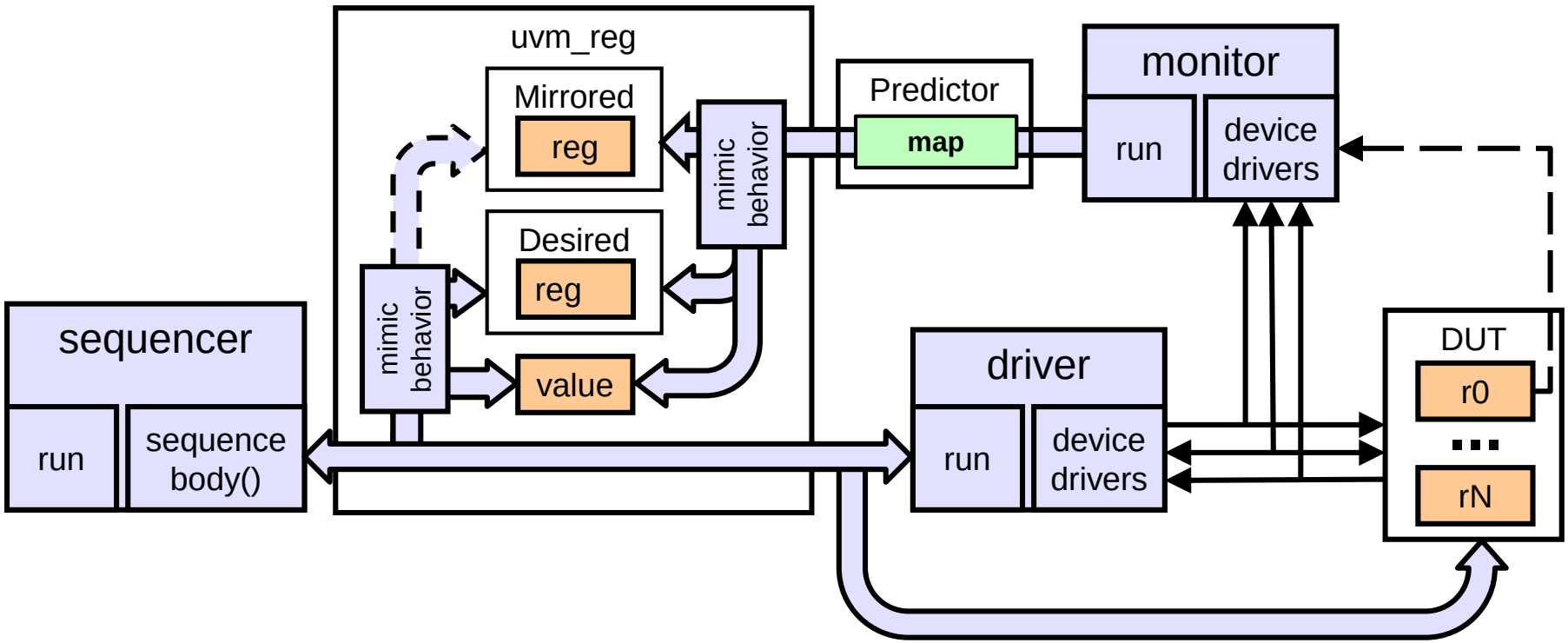
Detailing RAL Register Access

UVM Register Modes

UVM Register Modes

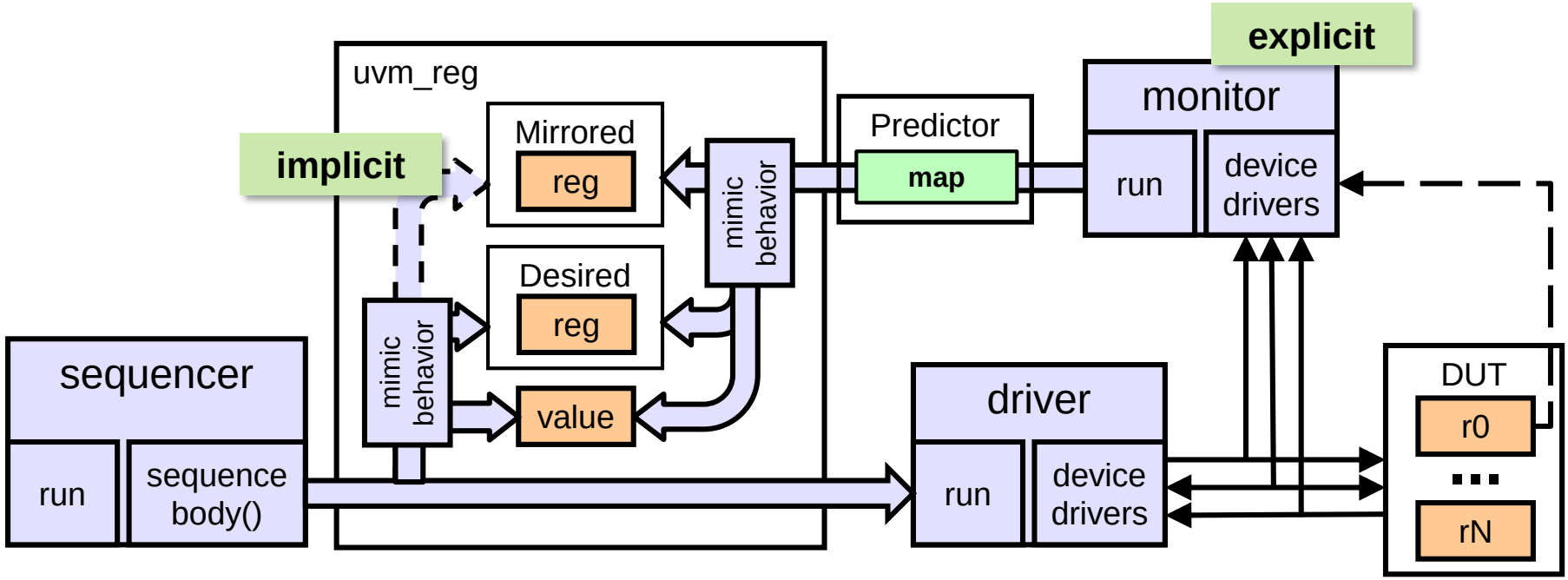
- **Frontdoor read/write**
- **Backdoor read/write/peek/poke**
- **Mirror set/get/update**

Memory is not mirrored



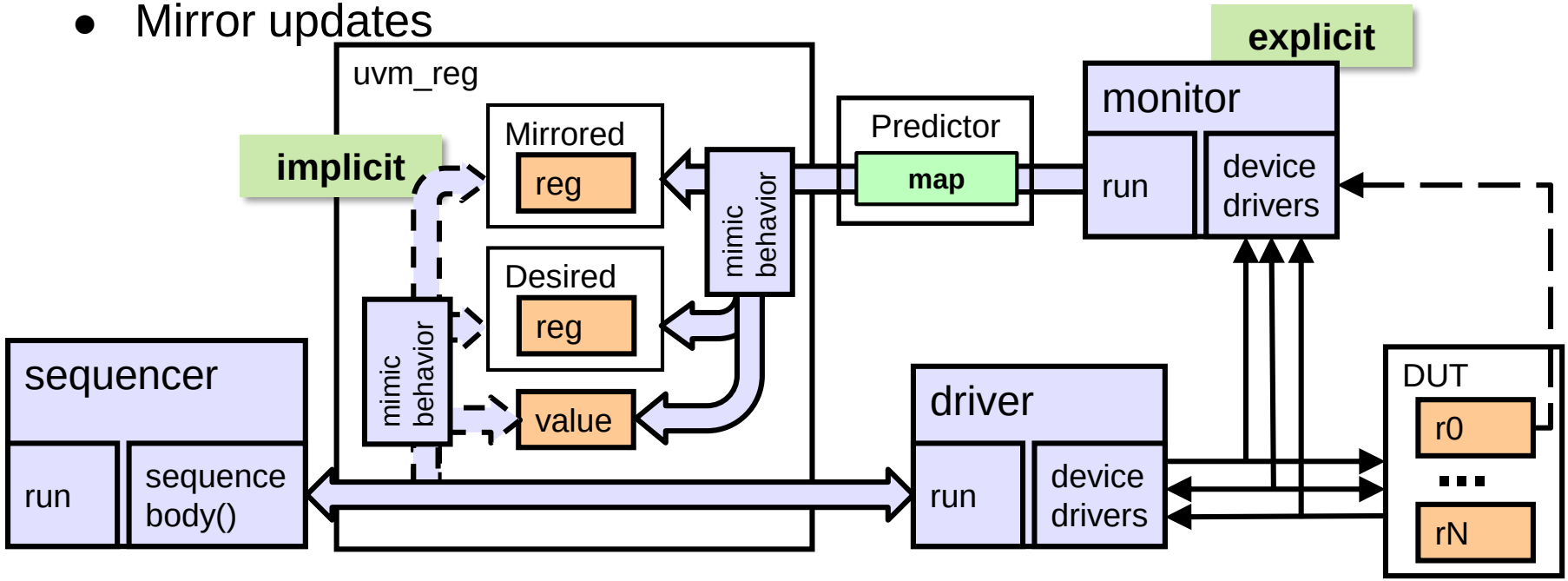
Register Frontdoor Write

- ***model.r0*.write(*status*, *value*, [UVM_FRONTDOOR], .parent(this));**
- **write_reg(*model.r0*, *status*, *value*, [UVM_FRONTDOOR]);**
  - Sequence sets uvm_reg with value
  - uvm_reg content is translated into bus transaction
  - Driver gets bus transaction and writes DUT register
  - Mirror is updated either implicitly or explicitly



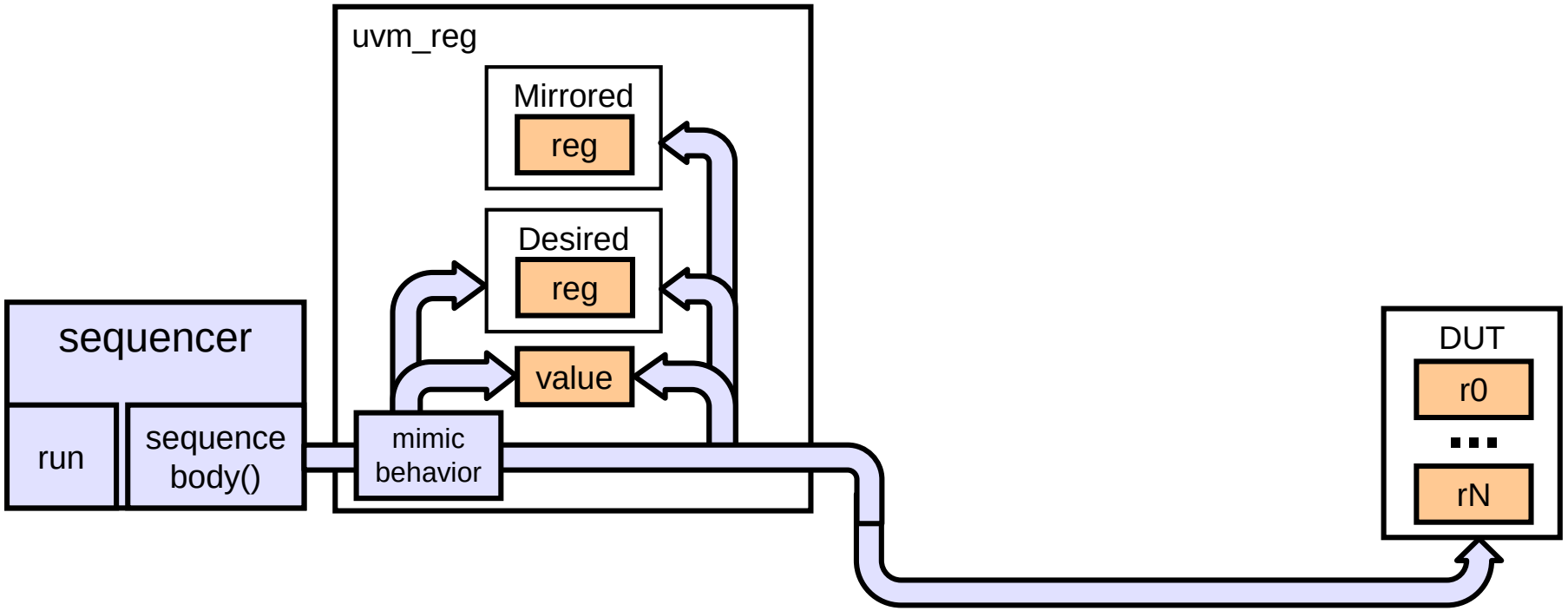
Register Frontdoor Read

- ***model.r0*.read(*status*, *value*, [UVM_FRONTDOOR], .parent(this));**
- **read_reg(*model.r0*, *status*, *value*, [UVM_FRONTDOOR]);**
  - Sequence executes uvm_reg READ
  - uvm_reg READ is translated into bus transaction
  - Driver gets bus transaction and read DUT register
  - Read value is translated into uvm_reg data and returned to sequence
  - Mirror updates



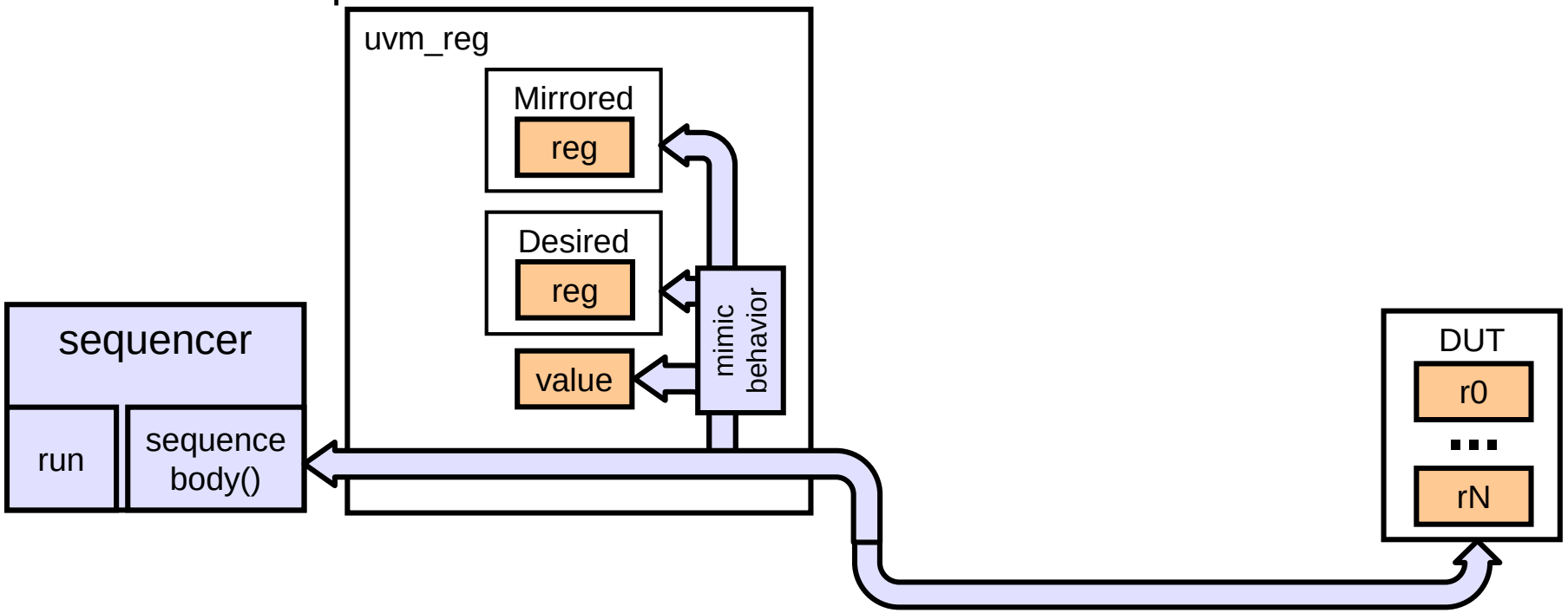
Register Backdoor Write

- ***model.r0*.write(*status*, *value*, UVM_BACKDOOR, .parent(this));**
- **write_reg(*model.r0*, *status*, *value*, UVM_BACKDOOR);**
  - Sequence write to uvm_reg with value mimicking register access policy (wc clears register)
  - uvm_reg uses DPI/XMR to set DUT register with value
  - Mirror is updated implicitly



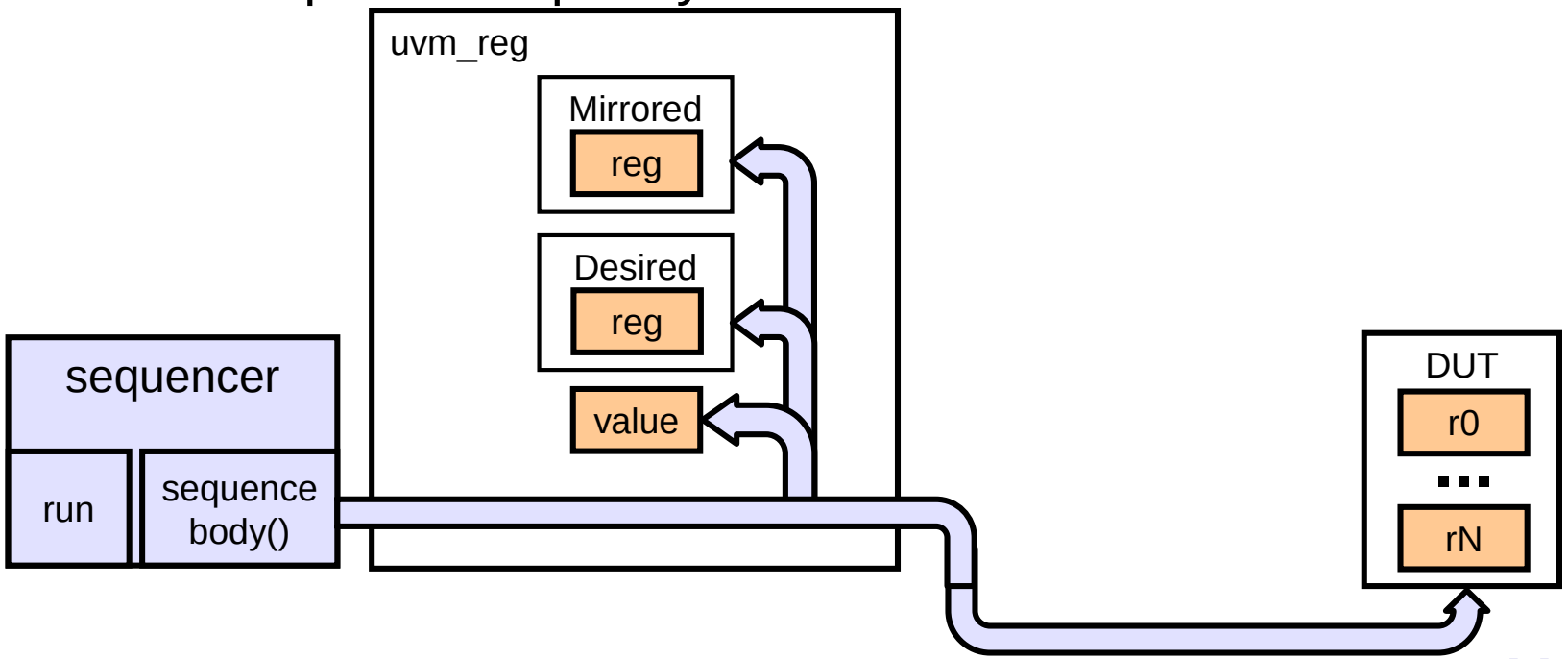
Register Backdoor Read

- ***model.r0*.read(*status*, *value*, UVM_BACKDOOR, .parent(this));**
- **read_reg(*model.r0*, *status*, *value*, UVM_BACKDOOR);**
  - Sequence executes uvm_reg READ
  - uvm_reg uses DPI/XMR to get DUT register value
    - Modifies DUT register according to register policy – rc clears register
  - Mirror is updated



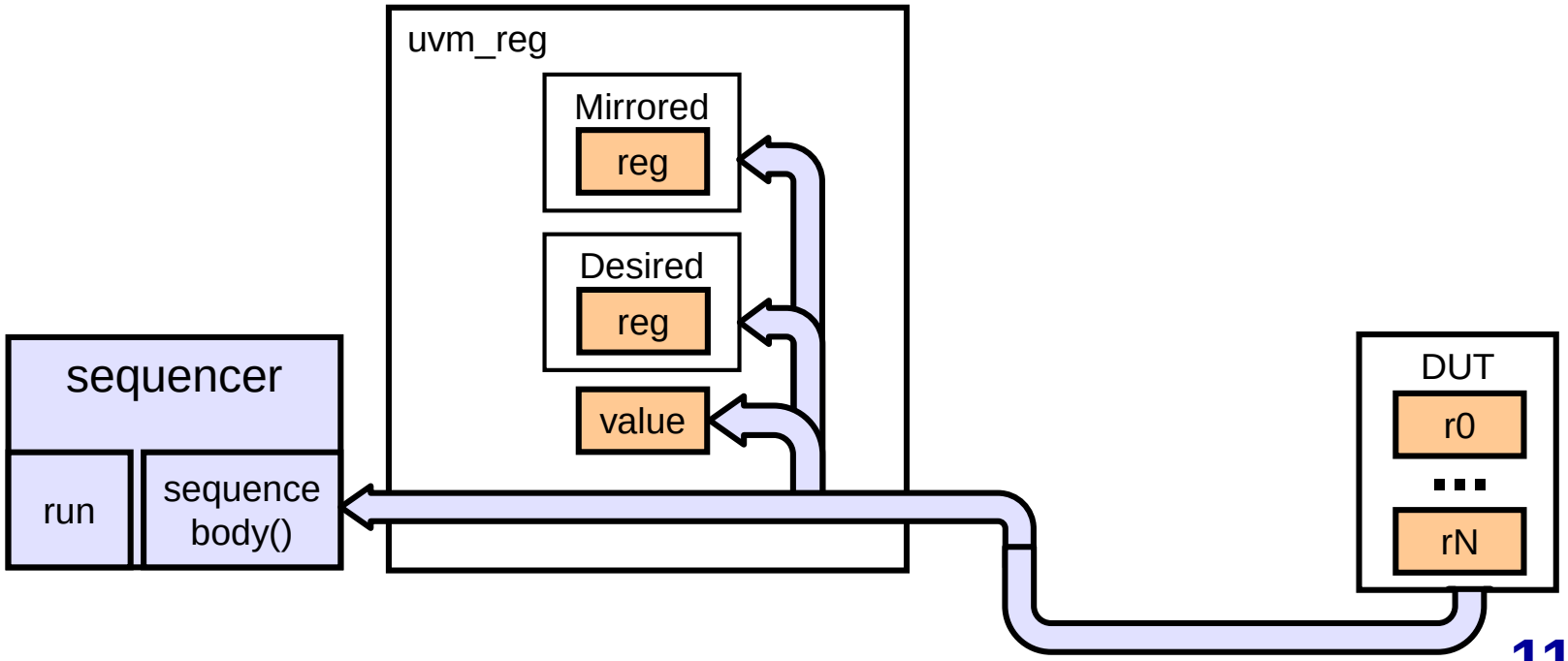
Register Backdoor Poke

- ***model.r0*.poke(*status*, *value*, .parent(this));**
- **poke_reg(*model.r0*, *status*, *value*);**
  - Sequence write to uvm_reg with value as is
    - Does not mimic register access policy (wc, etc.)
  - uvm_reg uses DPI/XMR to set DUT register with value
  - Mirror is updated implicitly



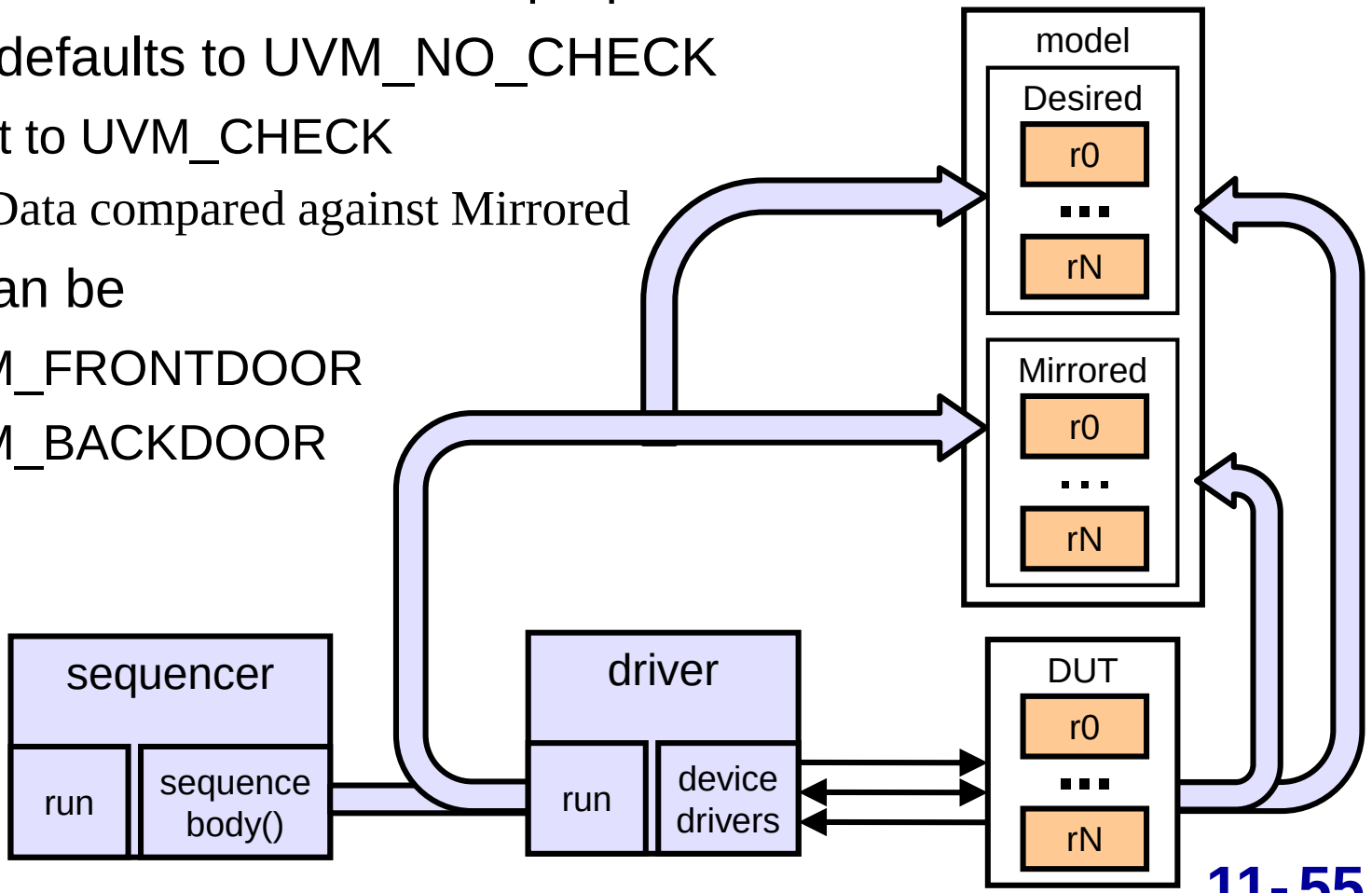
Register Backdoor Peek

- ***model.r0*.peek(*status*, *value*, .parent(this));**
- **peek_reg(*model.r0*, *status*, *value*);**
  - Sequence executes uvm_reg PEEK
  - uvm_reg uses DPI/XMR to get DUT register value as is
    - Does not mimic behavior (rc, etc.)
  - Mirror is updated implicitly



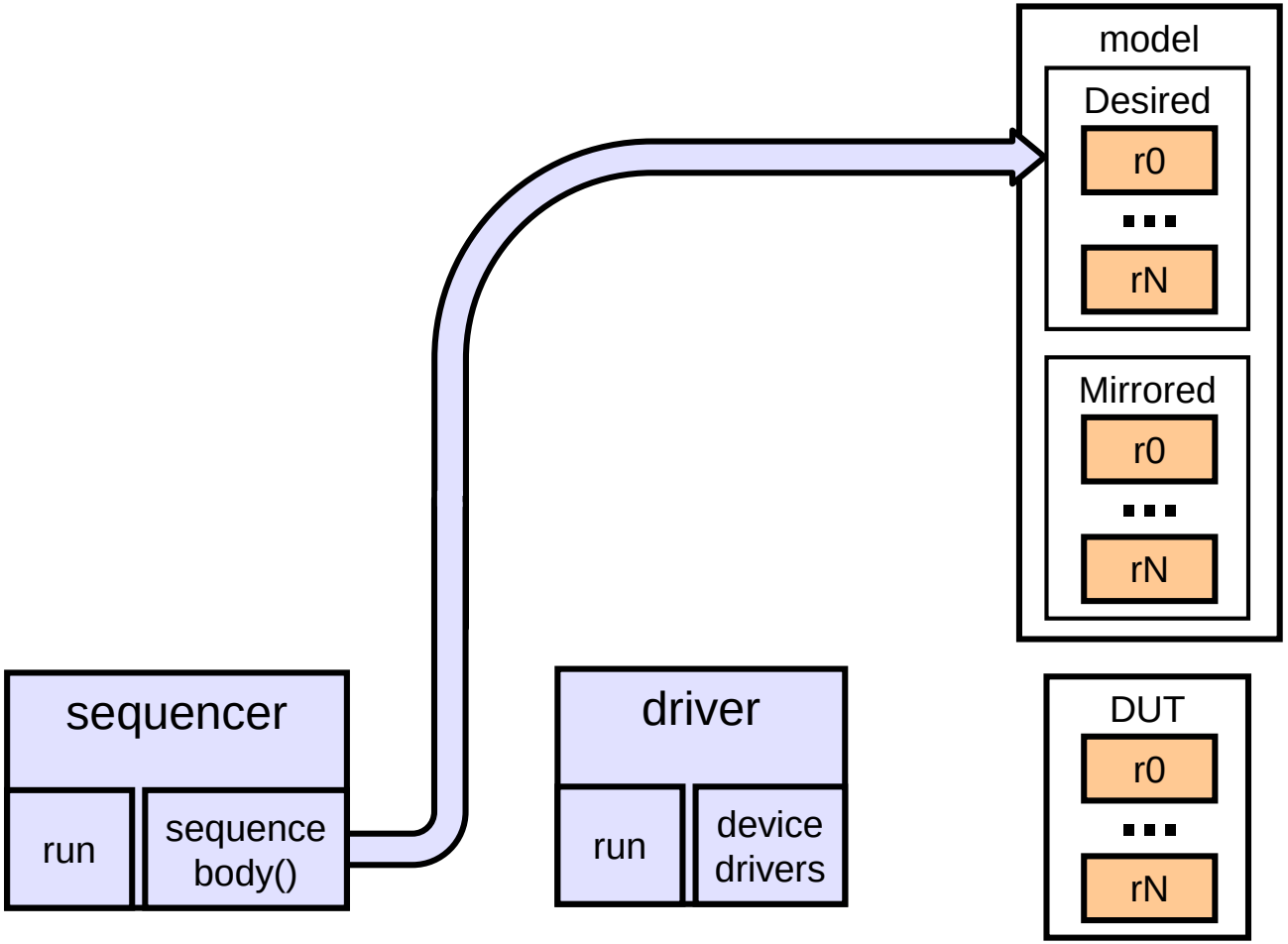
Mirrored & Desired Property Update

- ***model.r0*.mirror(*status*, [`check`], [`path`], .parent(this));**
- **mirror_reg(*model.r0*, *status*, [`check`], [`path`]);**
  - Update mirrored and desired properties with DUT content
  - **`check`** defaults to UVM_NO_CHECK
    - If set to UVM_CHECK
      - Data compared against Mirrored
  - **`path`** can be
    - UVM_FRONTDOOR
    - UVM_BACKDOOR



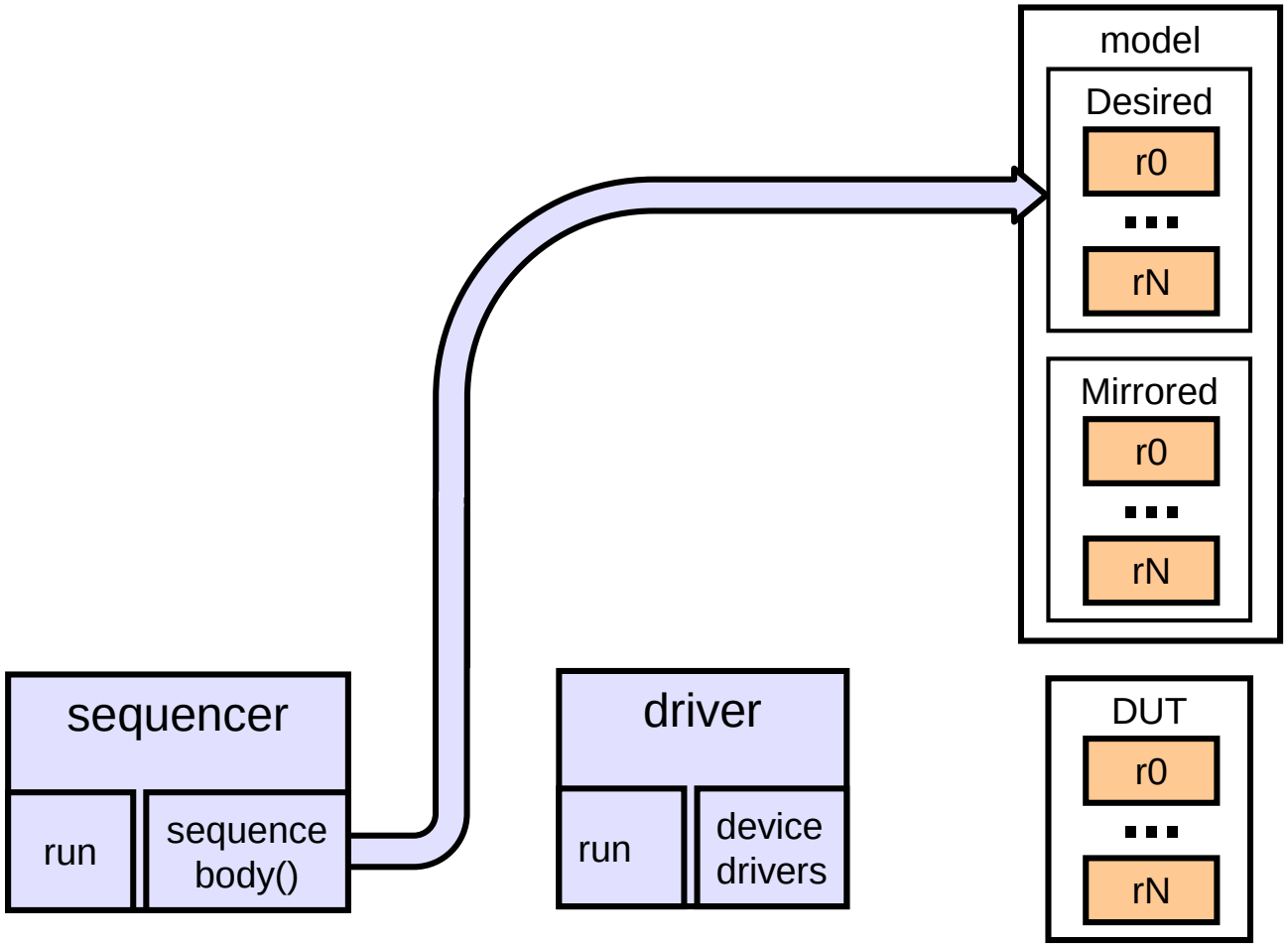
UVM Register Desired Property Write

- ***model.r0*.set(*value*);**
  - Set method set value in desired property



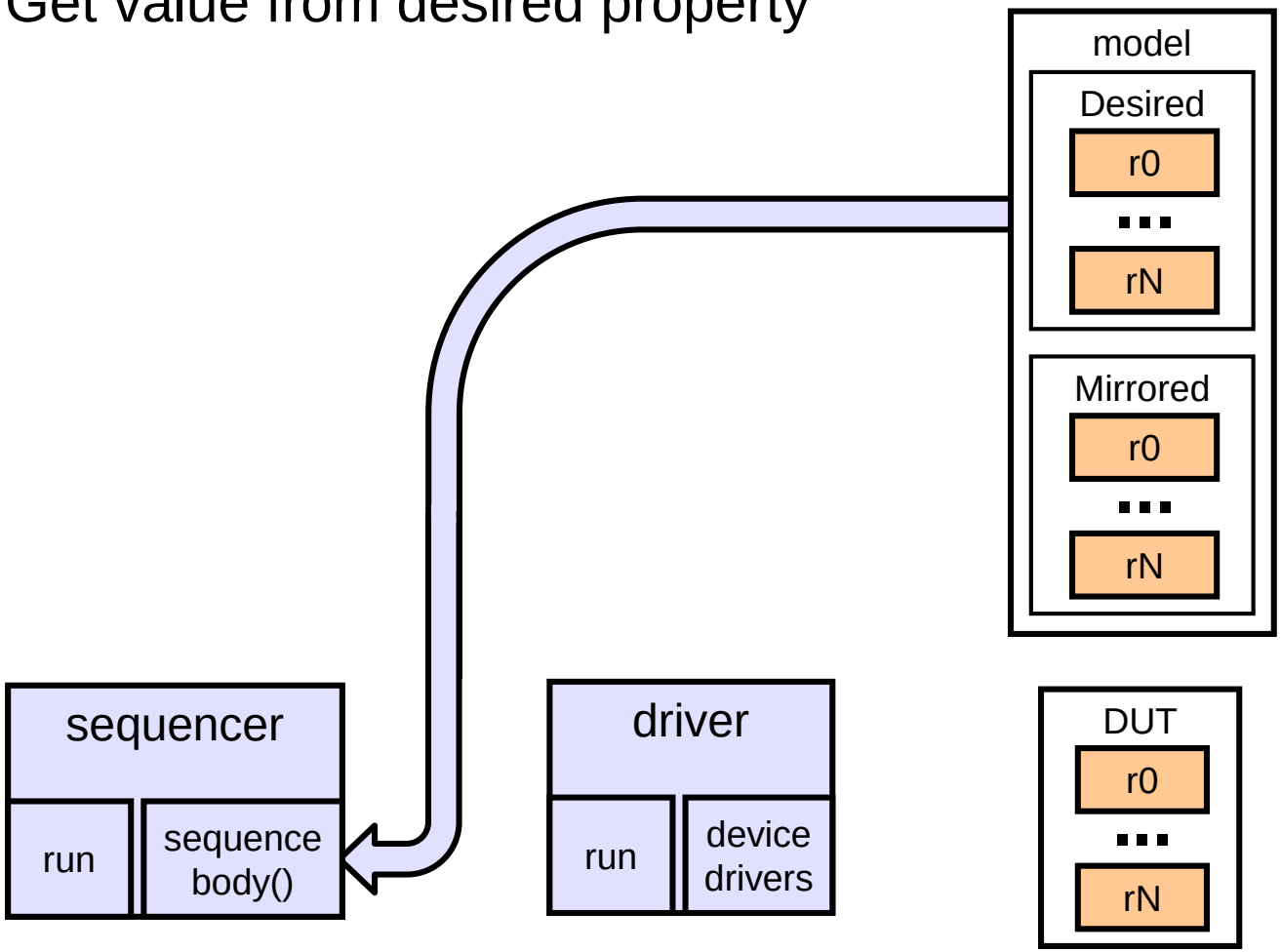
Randomize UVM Register Desired Property

- ***model*.randomize();**
  - Populate desired property with random value



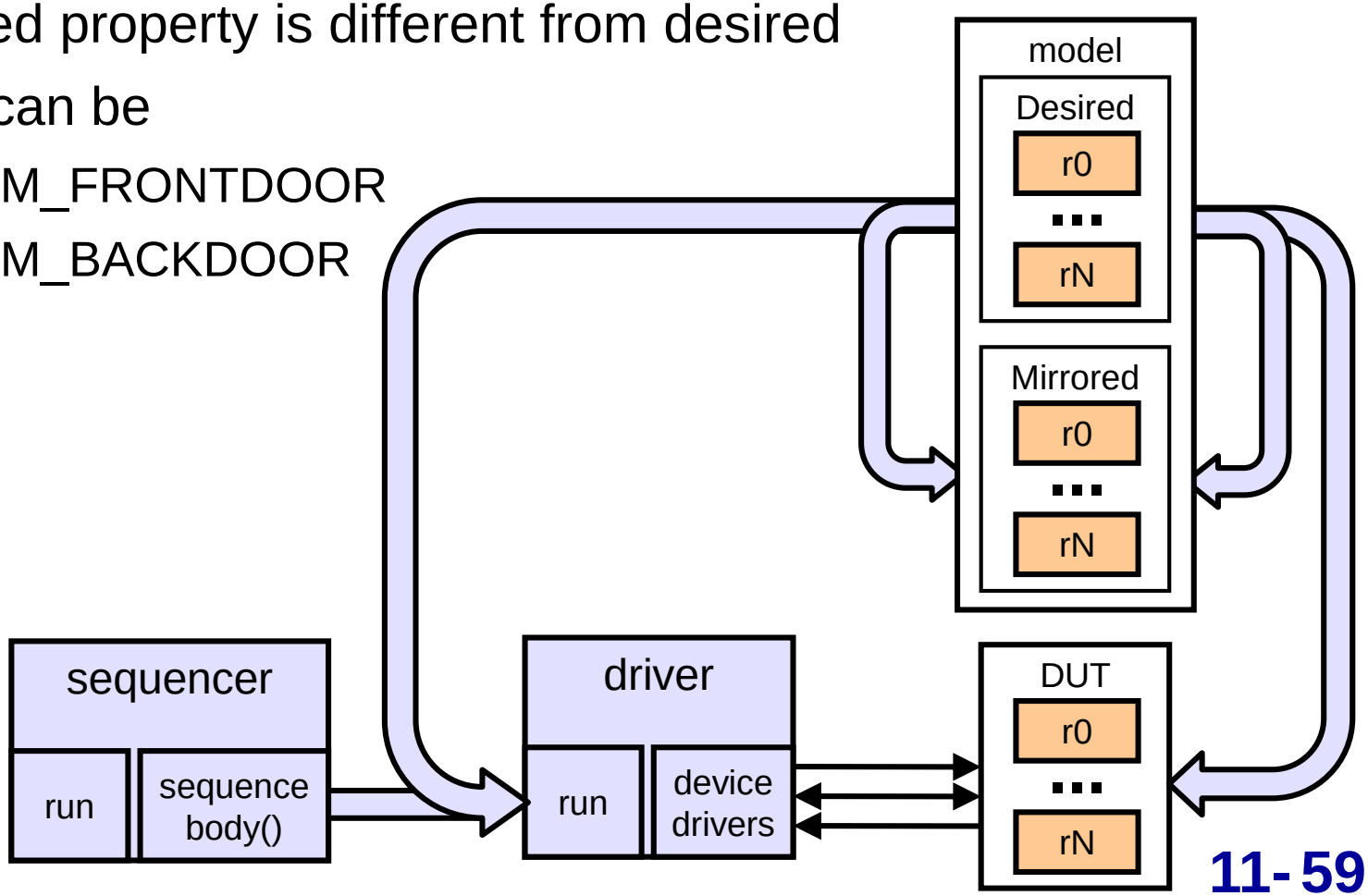
UVM Register Desired Property Read

- ***value* = *model*.*r0*.get();**
  - Get value from desired property



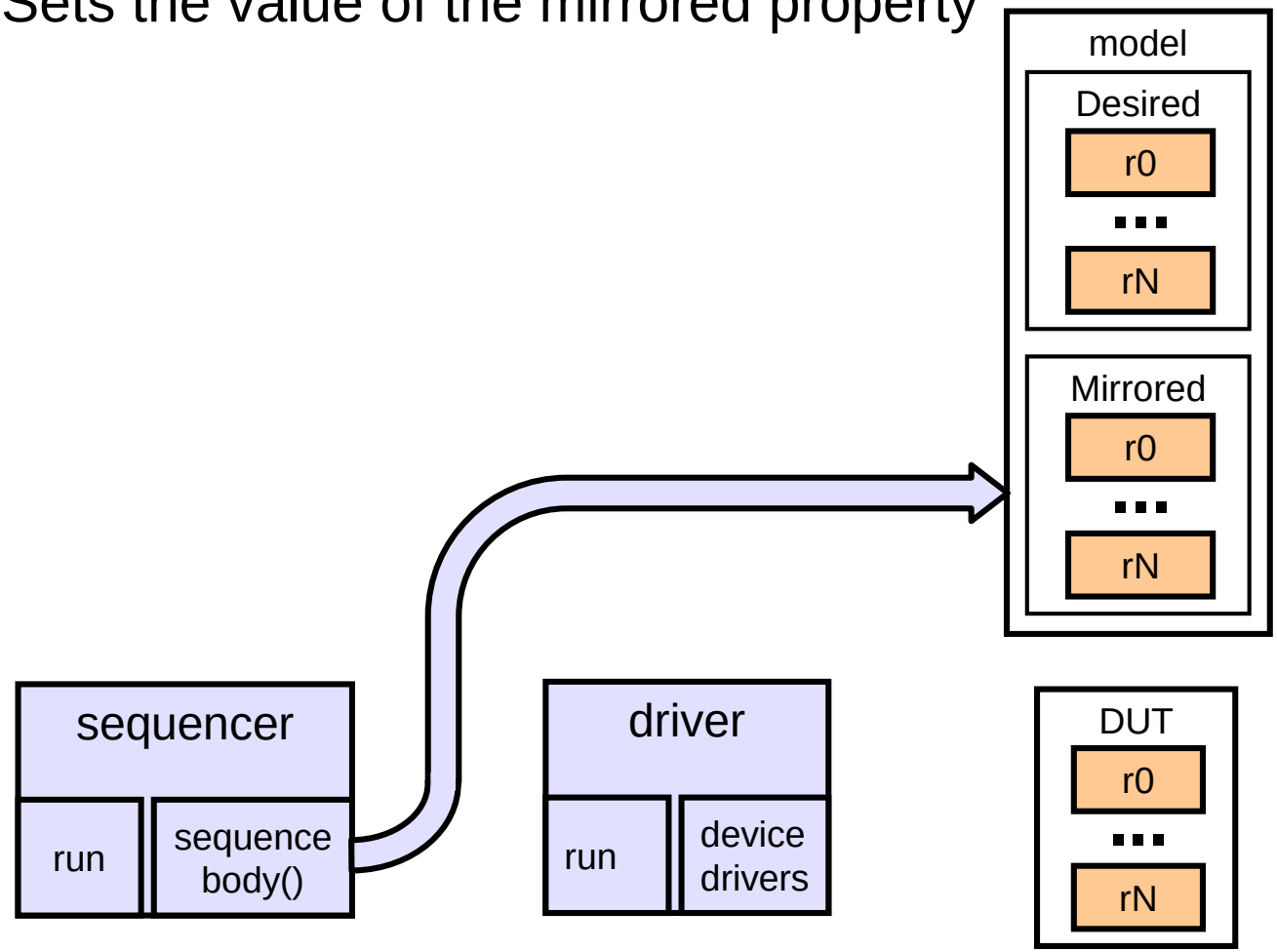
Mirrored & DUT Value Update

- ***model*.update(*status*, [`path`], .parent(this));**
- **update_reg(*model*, *status*, [`path`]);**
  - Update DUT and mirrored property with desired property if mirrored property is different from desired
  - **`path`** can be
    - UVM_FRONTDOOR
    - UVM_BACKDOOR



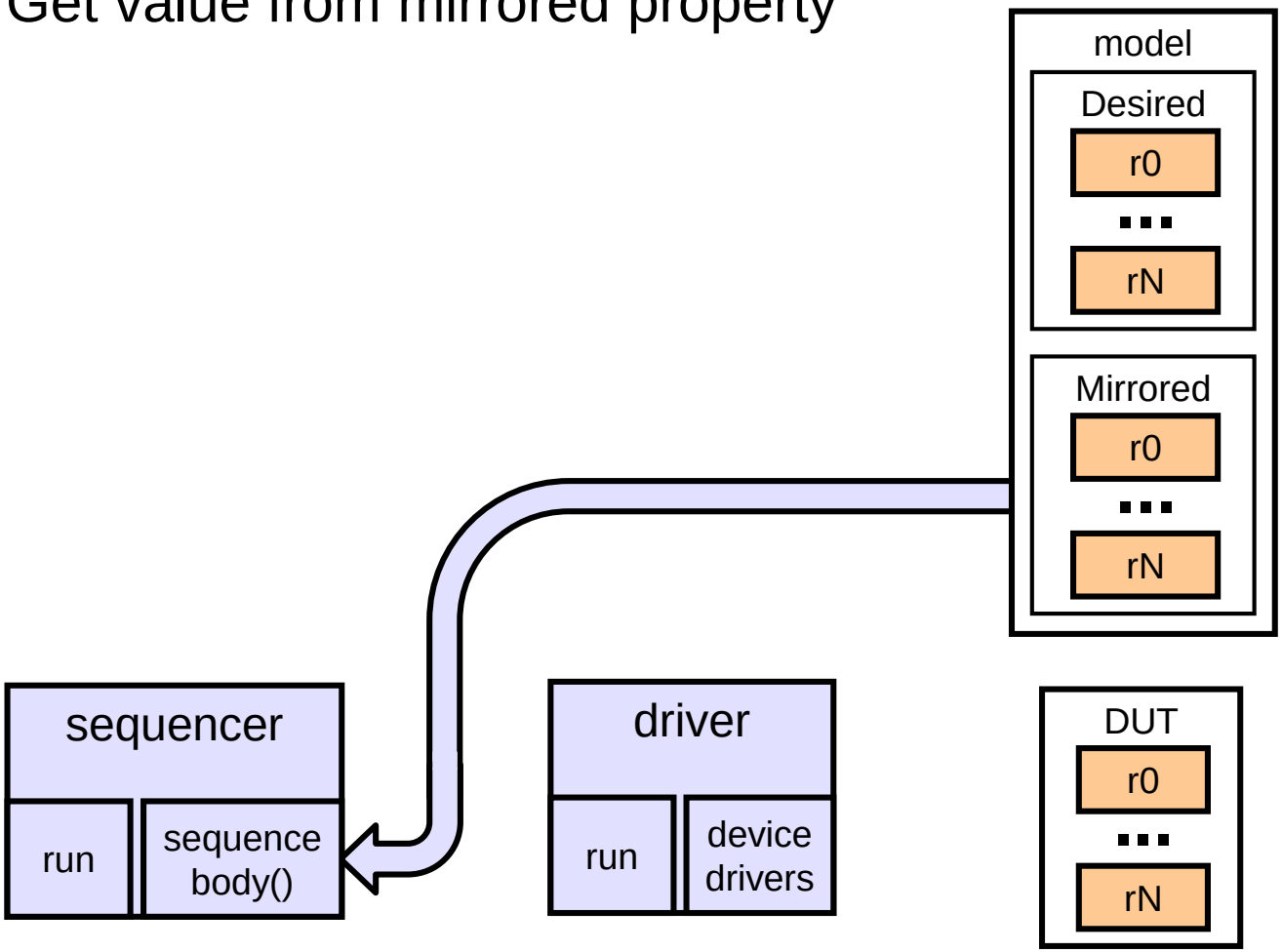
Writing to uvm_reg Mirrored Property

- ***model.r0*.predict(*value*);**
  - Sets the value of the mirrored property



Reading uvm_reg Mirrored Property

- ***value* = *model.r0*.get_mirrored_value();**
  - Get value from mirrored property

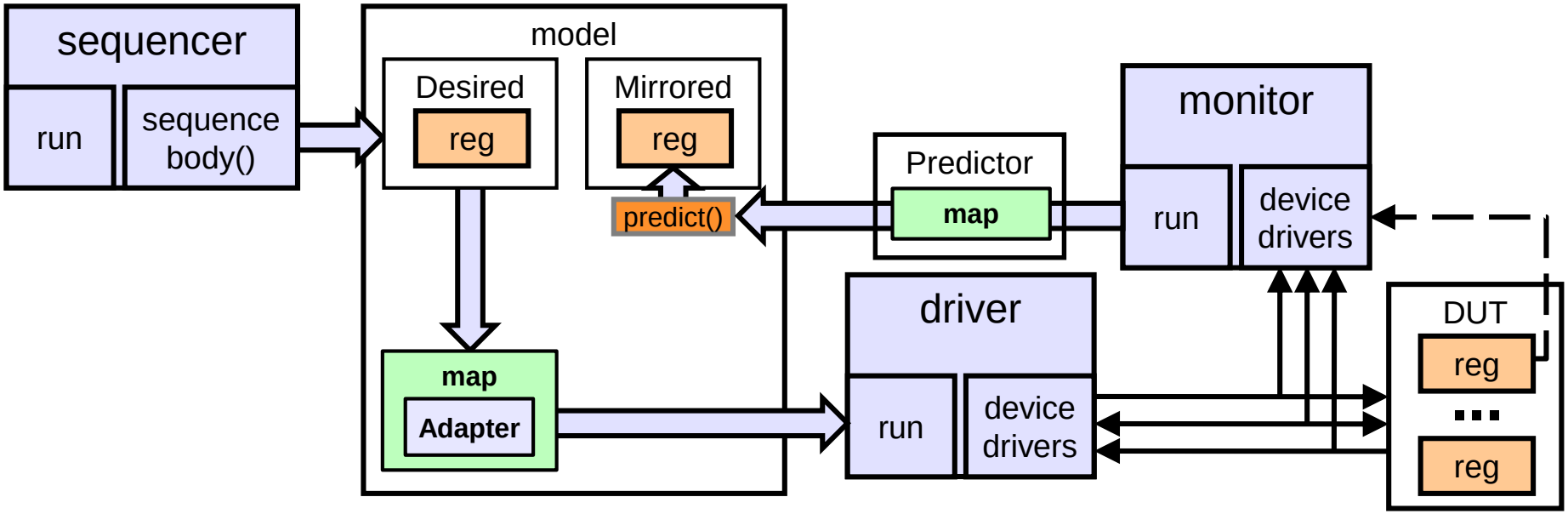


Typical use of uvm_reg predict() method (1/2)

Example for explicit predictor implementation:

```
typedef uvm_reg_predictor #(host_data) hreg_predictor;
hreg_predictor hreg_predict;
// other code left off see slides on predictor
h_agt.analysis_port.connect(hreg_predict.bus_in);
regmodel.default_map.set_auto_predict(0);
```

- Monitor pass observed transaction to predictor
- Predictor calls predict() method to update mirrored property

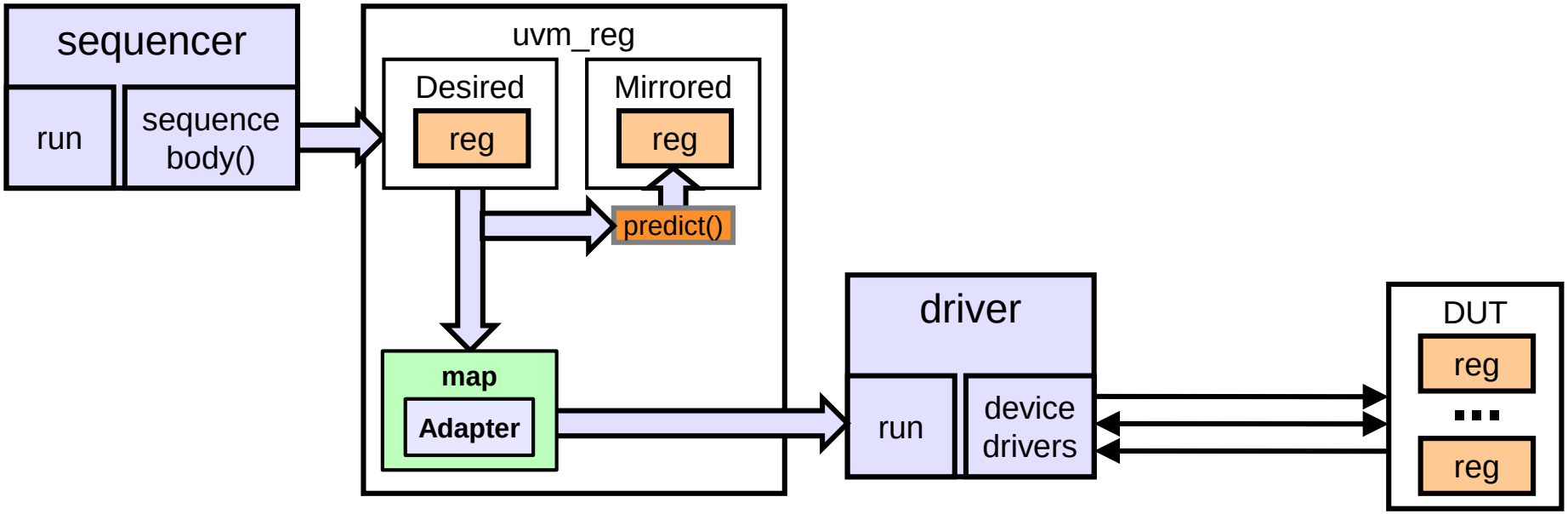


Typical use of uvm_reg predict() method (2/2)

Example for implicit predictor implementation:

```
regmodel.default_map.set_auto_predict(1);
```

- uvm_reg calls predict() method to update mirrored property directly on register activity rather observed physical changes



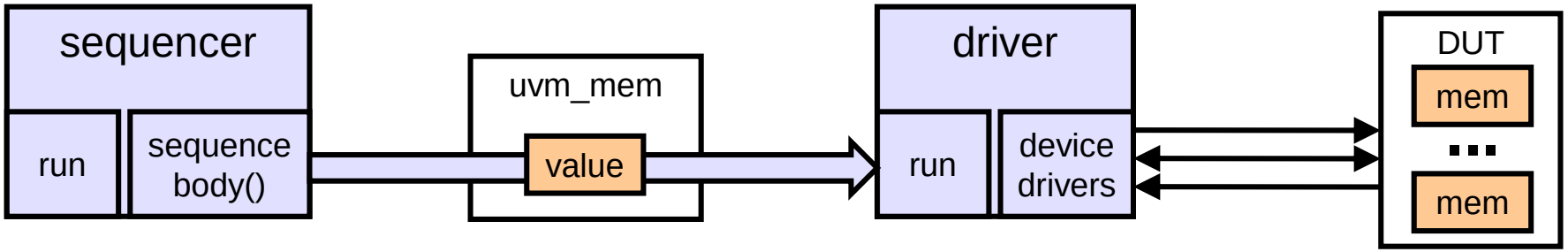
UVM Memory Modes

Memory Frontdoor Write

- ***model.mem*.write(*status*, *offset*, *value*, [UVM_FRONTDOOR], .parent(this));**
- ***model.mem*.burst_write(*status*, *offset*, *value*[], [UVM_FRONTDOOR], .parent(this));**

Provide array populated with values to write to memory

- **write_mem(*model.mem*, *status*, *offset*, *value*, [UVM_FRONTDOOR]);**
  - Sequence sets uvm_mem with value
  - uvm_mem content is translated into bus transaction
  - Driver gets bus transaction and writes DUT memory
  - Memory is not mirrored

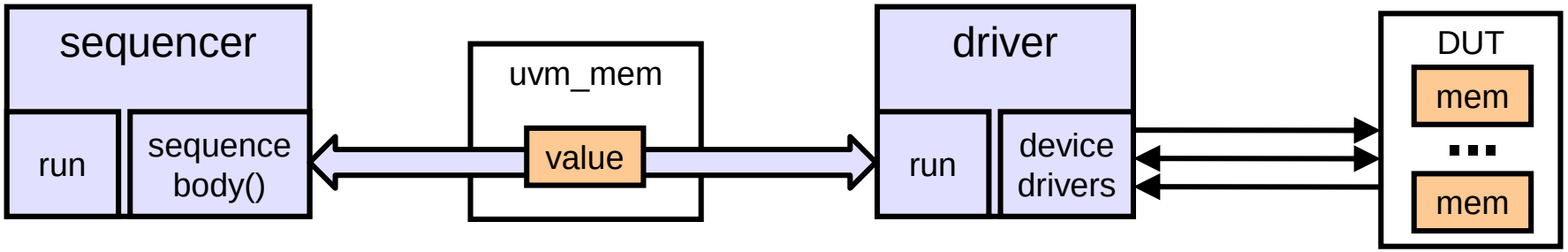


Memory Frontdoor Read

- ***model.mem*.read(*status*, *offset*, *value*, [UVM_FRONTDOOR], .parent(this));**
- ***model.mem*.burst_read(*status*, *offset*, *value*[], [UVM_FRONTDOOR], .parent(this));**

Provide array sized to number of values to read from memory

- **read_mem(*model.mem*, *status*, *offset*, *value*, [UVM_FRONTDOOR]);**
  - Sequence executes uvm_mem READ
  - uvm_mem READ is translated into bus transaction
  - Driver gets bus transaction and reads DUT memory
  - Read value is translated to uvm_mem format and returned to sequence
  - Memory is not mirrored

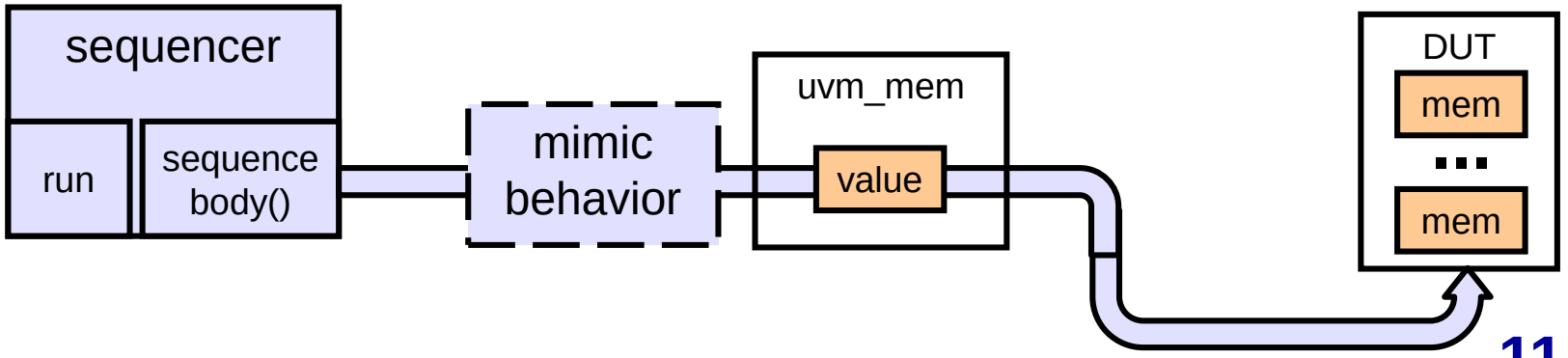


Memory Backdoor Write

- ***model.mem*.write(*status*, *offset*, *value*, UVM_BACKDOOR, .parent(this));**
- ***model.mem*.burst_write(*status*, *offset*, *value*[], UVM_BACKDOOR, .parent(this));**

Provide array populated with values to write to memory

- **write_mem(*model.mem*, *status*, *offset*, *value*, UVM_BACKDOOR);**
  - Sequence write to uvm_mem with value mimicking memory access policy (ro does not change memory value)
  - uvm_mem uses DPI/XMR to set DUT memory with value
  - Memory is not mirrored

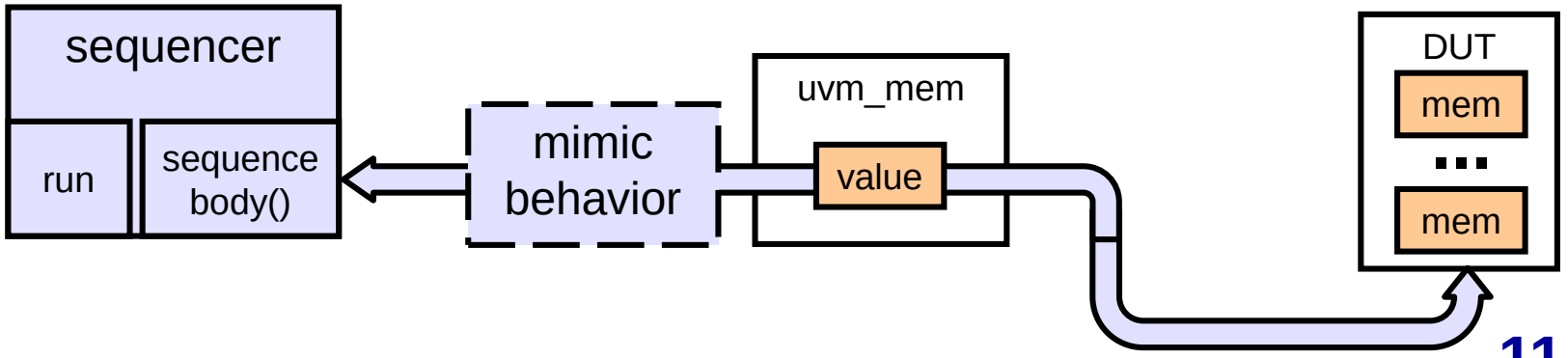


Memory Backdoor Read

- ***model.mem*.read(*status*, *offset*, *value*, UVM_BACKDOOR, .parent(this));**
- ***model.mem*.burst_read(*status*, *offset*, *value*[], UVM_BACKDOOR, .parent(this));**

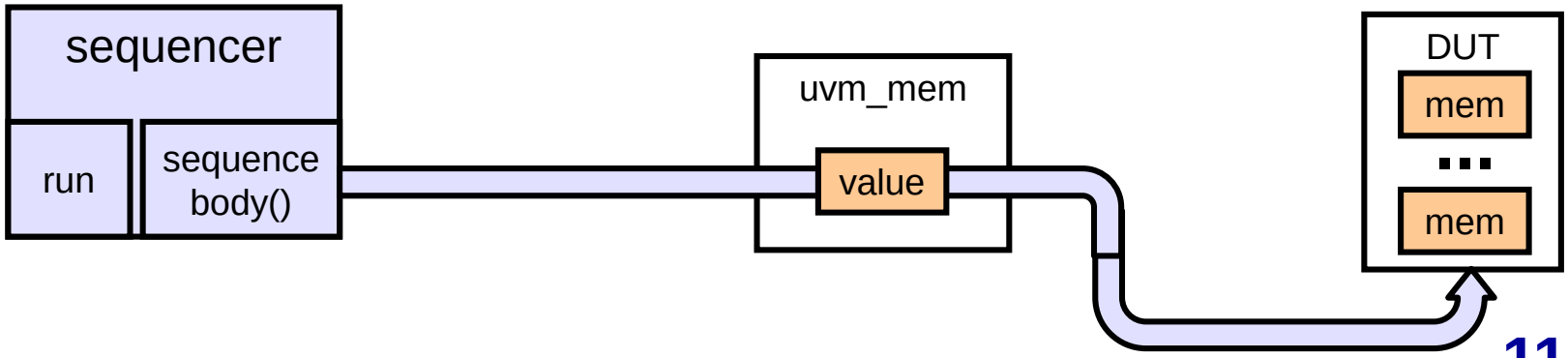
Provide array sized to number of values to read from memory

- **read_mem(*model.mem*, *status*, *offset*, *value*, UVM_BACKDOOR);**
  - Sequence executes uvm_mem READ
  - uvm_mem uses DPI/XMR to get DUT memory value mimicking memory access policy (wo does not return memory value)
  - Memory is not mirrored



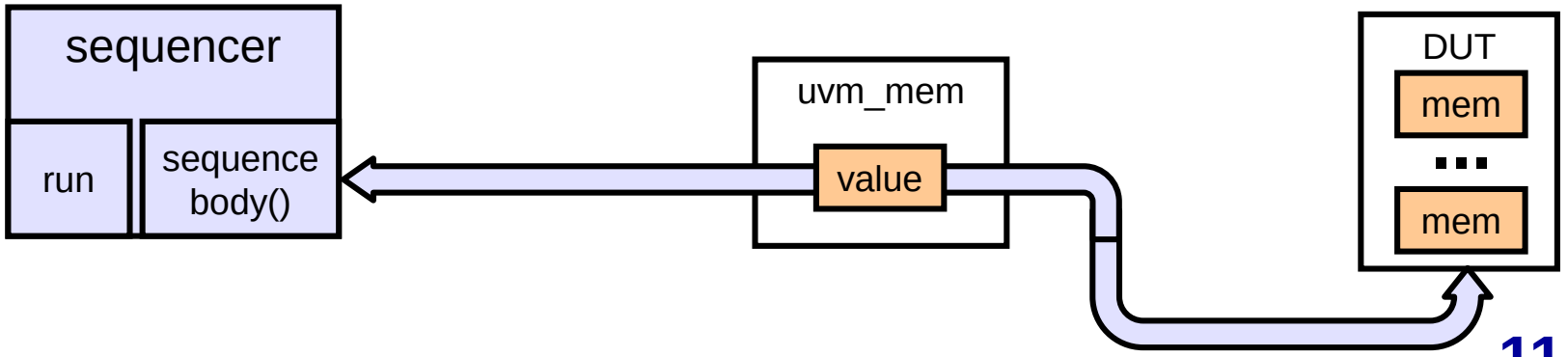
Memory Backdoor Poke

- ***model.mem*.poke(*status*, *offset*, *value*, UVM_BACKDOOR, .parent(this));**
- **poke_mem(*model.mem*, *status*, *offset*, *value*, UVM_BACKDOOR);**
  - Sequence writes to uvm_mem with value
    - Does not mimic memory access policy (ro memory WILL be modified)
  - uvm_mem uses DPI/XMR to set DUT memory with value
  - Memory is not mirrored



Memory Backdoor Peek

- ***model.mem*.peek(*status*, *offset*, *value*, UVM_BACKDOOR, .parent(this));**
- **peek_mem(*model.mem*, *status*, *offset*, *value*, UVM_BACKDOOR);**
  - Sequence executes uvm_mem PEEK
  - uvm_mem uses DPI/XMR to get DUT memory value
  - Memory is not mirrored



ralgen Options

ralgen Options: Common

```
ralgen [options] -uvm -t top_model file.ralf
```

- **Common options:**
  - **`-f file`**
    - Specify a file containing ralgen options
  - **`-o fname`**
    - Specify output file name. Default is ral_<top_model>
  - **`-c b|s|a|f|F`**
    - Generate functional coverage model
  - **`-P`**
    - Generate model in separate packages

ralgen Options: Advanced

- **Advanced options:**
  - **`-b`**
    - Generate XMR-based (non-DPI) back-door access code
    - **`-gen_vif_bkdr`**
      - Generate back-door access with virtual interface
    - **`-auto_mirror`**
      - Generate code for automatic mirror update
  - **`-B`**
    - Generate byte-base addressing
- **See uvm_ralgen_ug.pdf for other options**
  - Or, execute: ralgen -h

ralgen Address Granularity

```
register R {
  bytes 4;
  field C { bits 8; }
}
register S {
  bytes 4;
  field D { bits 8; }
}
register T {
  bytes 4;
  field A { bits 8; }
}
block BLK {
    bytes 4;
    register R;
    register S;
    register T;
}
```

```
regmodel = ral_block_BLK::type_id::create("regmodel", this);
regmodel.build();
regmodel.lock_model();
`uvm_info("ADDR_MAP", $sformatf("R addr = %0h", regmodel.R.get_address()), UVM_HIGH)
`uvm_info("ADDR_MAP", $sformatf("S addr = %0h", regmodel.S.get_address()), UVM_HIGH)
`uvm_info("ADDR_MAP", $sformatf("T addr = %0h", regmodel.T.get_address()), UVM_HIGH)
```

`ralgen -uvm -t BLK file.ralf`

```
UVM_INFO program.sv(11) @ 0: reporter [ADDR_MAP] R address = 0
UVM_INFO program.sv(12) @ 0: reporter [ADDR_MAP] S address = 1
UVM_INFO program.sv(13) @ 0: reporter [ADDR_MAP] T address = 2
```

`ralgen -uvm -t BLK -B file.ralf`

```
UVM_INFO program.sv(11) @ 0: reporter [ADDR_MAP] R address = 0
UVM_INFO program.sv(12) @ 0: reporter [ADDR_MAP] S address = 4
UVM_INFO program.sv(13) @ 0: reporter [ADDR_MAP] T address = 8
```

ralgen XMR Backdoor Access (1/2)

- **–b –gen_vif_bkdr option generates an interface accessing register via verilog XMR**

```
// host.ralf
  ...
  register HOST_ID (host_id) @'h0000;
```

```
ralgen -b -gen_vif_bkdr -uvm -t dut_regmodel host.ralf
```

```
// ral_host_regmodel_intf.sv
interface ral_host_regmodel_intf;
  // other code not shown
  initial
    uvm_resource_db#(virtual ral_host_regmodel_intf)::set("*",
                        "uvm_reg_bkdr_if", interface::self());
  task ral_host_regmodel_HOST_ID_bkdr_read(uvm_reg_item rw);
    rw.value[0] = `HOST_REGMODEL_TOP_PATH.host_id;
  endtask
endinterface
```

Self stored into database

Uses XMR

```
vcs ral_host_regmodel_intf.sv \
    +define+HOST_REGMODEL_TOP_PATH=router_test_top.dut ...
```

Interface must be compiled

XMR path from harness to DUT must be specified

ralgen XMR Backdoor Access (2/2)

- **Register model uses interface to access registers**
  - Can be located in a SystemVerilog package

```
// ral_dut_regmodel.sv
class ral_reg_host_regmodel_HOST_ID_bkdr extends uvm_reg_backdoor;
 // other code not shown
 virtual ral_host_regmodel_intf __reg_vif;
 function new(string name);
  super.new(name);
  uvm_resource_db#(virtual ral_host_regmodel_intf)::read_by_name(get_full_name(),
                                                  "uvm_reg_bkdr_if", __reg_vif);
 endfunction
 virtual task read(uvm_reg_item rw);
  do_pre_read(rw);
  __reg_vif.ral_host_regmodel_HOST_ID_bkdr_read(rw);
  rw.status = UVM_IS_OK;
  do_post_read(rw);
 endtask
endclass
```

ralgen Functional Coverage

```
// host.ralf
register HOST_ID {
  field REV_ID {…}
  field CHIP_ID {…}
}
```

```
ralgen -c b -uvm -t dut_regmodel host.ralf
```

```
class ral_reg_HOST_ID extends uvm_reg;  // other code not shown
  uvm_reg_field REV_ID;  uvm_reg_field CHIP_ID;
  covergroup cg_bits ();
    option.per_instance = 1;
    REV_ID: coverpoint {m_data[7:0], m_is_read} iff(m_be) {
      wildcard bins bit_0_wr_as_0 = {9'b???????00};
      wildcard bins bit_0_wr_as_1 = {9'b???????10};
```

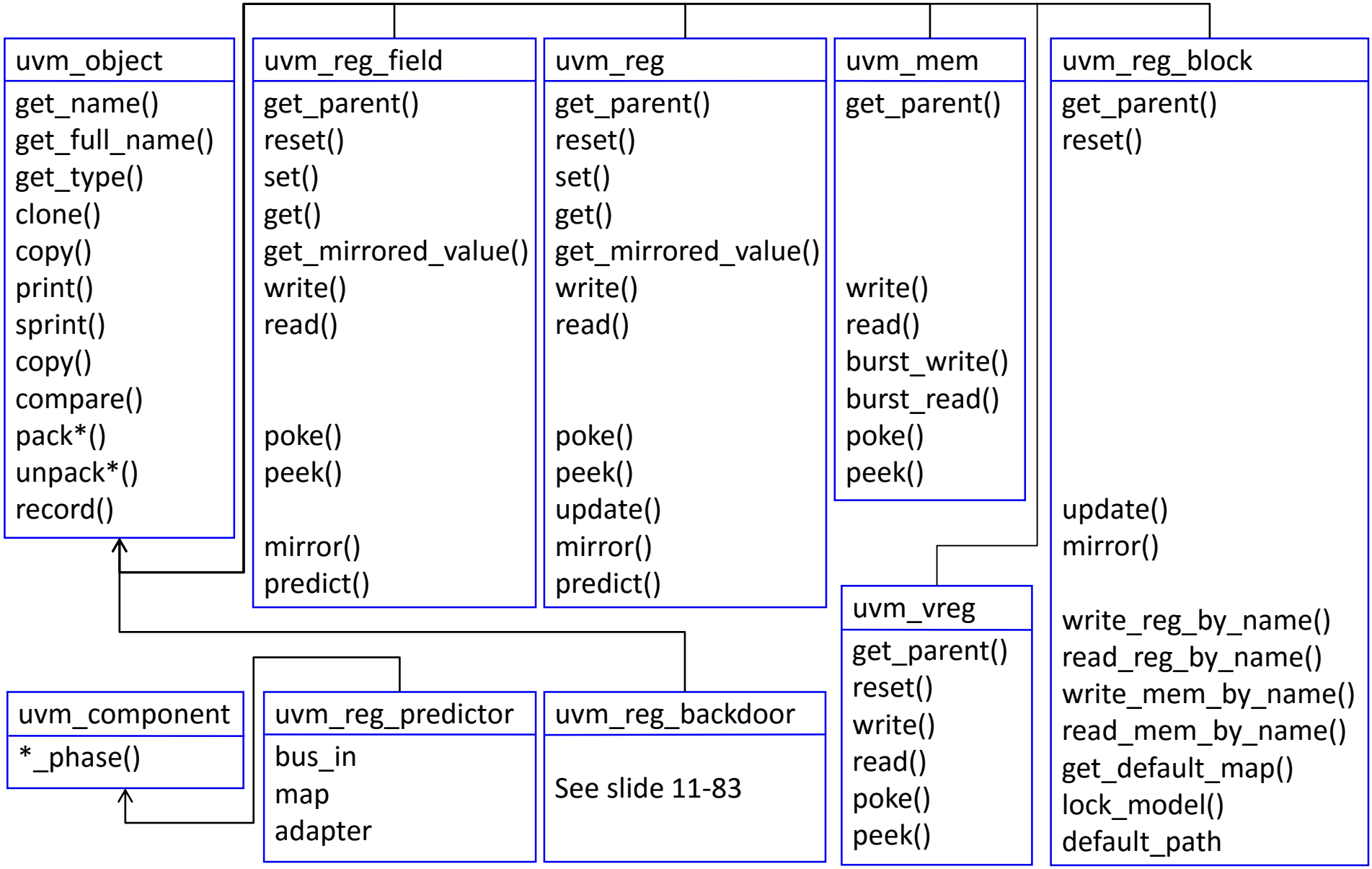
UVM RAL Classes

```
program automatic test; // other code not shown
  initial begin
    uvm_reg::include_coverage("*", UVM_CVR_ALL);
    run_test();
  end
endprogram
class test_coverage exte          _base; // other code not shown
  virtual function void          nd_of_elaboration_phase (uvm_phase phase);
    env.regmodel.set_coverage(UVM_CVR_ALL);
  endfunction
endclass
```

User must enable coverage with include_coverage() and set_coverage()

UVM Register Class Tree

UVM Register Base/Library Class Hierarchy



UVM Register/Memory Class Members

UVM Register Class Key Properties

```
`ifndef UVM_REG_ADDR_WIDTH
 `define UVM_REG_ADDR_WIDTH 64
`endif
`ifndef UVM_REG_DATA_WIDTH
 `define UVM_REG_DATA_WIDTH 64
`endif
typedef  bit unsigned [`UVM_REG_DATA_WIDTH-1:0]  uvm_reg_data_t ;
typedef  bit unsigned [`UVM_REG_ADDR_WIDTH-1:0]  uvm_reg_addr_t ;
```

Caution: Not complete.
Only key members are shown.

```
class uvm_reg_field extends uvm_object;
  `uvm_object_utils(uvm_reg_field)
  rand uvm_reg_data_t value; // Public property for
                             // coverage and randomization
                             // mirrored in m_desired after randomize()
  local uvm_reg_data_t  m_mirrored;      // Shadows what is in the DUT
  local uvm_reg_data_t  m_desired;       // User field local
  local uvm_reg_data_t  m_reset[string]; // Value at reset
  local string          m_access;
  local bit             m_volatile;
endclass: uvm_reg_field
```

UVM Memory Class Key Properties

```
virtual class uvm_reg extends uvm_object;
  protected bit           m_maps[uvm_reg_map];
  protected uvm_reg_field m_fields[$];   // Fields in LSB to MSB order
  local uvm_reg_backdoor  m_backdoor;
endclass: uvm_reg
```

```
class uvm_mem extends uvm_object;
   local bit               m_maps[uvm_reg_map];
   local uvm_reg_backdoor  m_backdoor;
   local bit               m_vregs[uvm_vreg];
   uvm_mem_mam             mam; // Memory allocation manager
endclass: uvm_mem
```

```
class uvm_reg_backdoor extends uvm_object;
   virtual task write(uvm_reg_item rw);
   virtual task read(uvm_reg_item rw);
   virtual function void read_func(uvm_reg_item rw); // User method
   virtual function bit is_auto_updated(uvm_reg_field field);
   virtual local task wait_for_change(uvm_object element);
endclass: uvm_reg_backdoor
```

Caution: Not complete.
Only key members are shown.

uvm_reg_bus_op Definition

```
typedef struct {
  // Variable: kind - Can be: UVM_READ, UVM_WRITE, UVM_BURST_READ, UVM_BURST_WRITE
  uvm_access_e kind;
  uvm_reg_addr_t addr;
  uvm_reg_data_t data;

  // Variable: n_bits - The number of bits of <uvm_reg_item::value>
  // being transferred by this transaction.
  int n_bits;

  // Variable: byte_en - Enables for the byte lanes on the bus.
  // Meaningful only when the bus supports byte enables and the
  // operation originates from a field write/read.
  uvm_reg_byte_en_t byte_en;

  // Variable: status - Result can be: UVM_IS_OK, UVM_HAS_X, UVM_NOT_OK.
  uvm_status_e status;
} uvm_reg_bus_op;
```

UVM Register Callbacks

RAL Class Key Callback Members

Caution: Simplified code for illustration. Most code left off.

```
// See class reference document and source code files
// for actual code
virtual class uvm_reg extends uvm_object;
  `uvm_register_cb(uvm_reg, uvm_reg_cbs)
  virtual task pre_write(uvm_reg_item rw); endtask
  virtual task post_write(uvm_reg_item rw); endtask
  virtual task pre_read(uvm_reg_item rw); endtask
  virtual task post_read(uvm_reg_item rw); endtask
endclass: uvm_reg
task uvm_reg::write(…);
  set(value);
  do_write(rw);  // See next page
endtask
```

- **Two ways to implement callbacks**
  - Simple callback – extend uvm_reg class
  - UVM callback – extend uvm_reg_cbs class then register cb

RAL Class Key Callback Members

```
task uvm_reg::do_write(uvm_reg_item rw);
  uvm_reg_cb_iter cbs = new(this);
  foreach (m_fields[i]) begin
    uvm_reg_field_cb_iter cbs = new(m_fields[i]);
    uvm_reg_field f = m_fields[i];
    f.pre_write(rw);
    for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
      cb.pre_write(rw);
  end
  pre_write(rw);
  for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
    cb.pre_write(rw);
  // EXECUTE WRITE...
  for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
    cb.post_write(rw);
  post_write(rw);
  foreach (m_fields[i]) begin
    uvm_reg_field_cb_iter cbs = new(m_fields[i]);
    uvm_reg_field f = m_fields[i];
    for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
      cb.post_write(rw);
    f.post_write(rw);
  end
endtask: do_write
```

Caution: Simplified code for illustration only

Changing Address Offsets of a Domain

Changing the Address offsets of a Domain

- **`set_base_addr()` method allows user to modify the base address of a uvm register map**
  - Can be called before or after lock_model
  - If called after the model is locked, the address will be re-initialized

```
virtual function void build_phase(uvm_phase phase);
  ...; // other code left off
  model.build();
  model.APBD.set_base_addr('h2000_0000);
  model.WSHD.set_base_addr('h4002_0000);
  model.lock_model();
endfunction
```

Detailing RAL Register Access

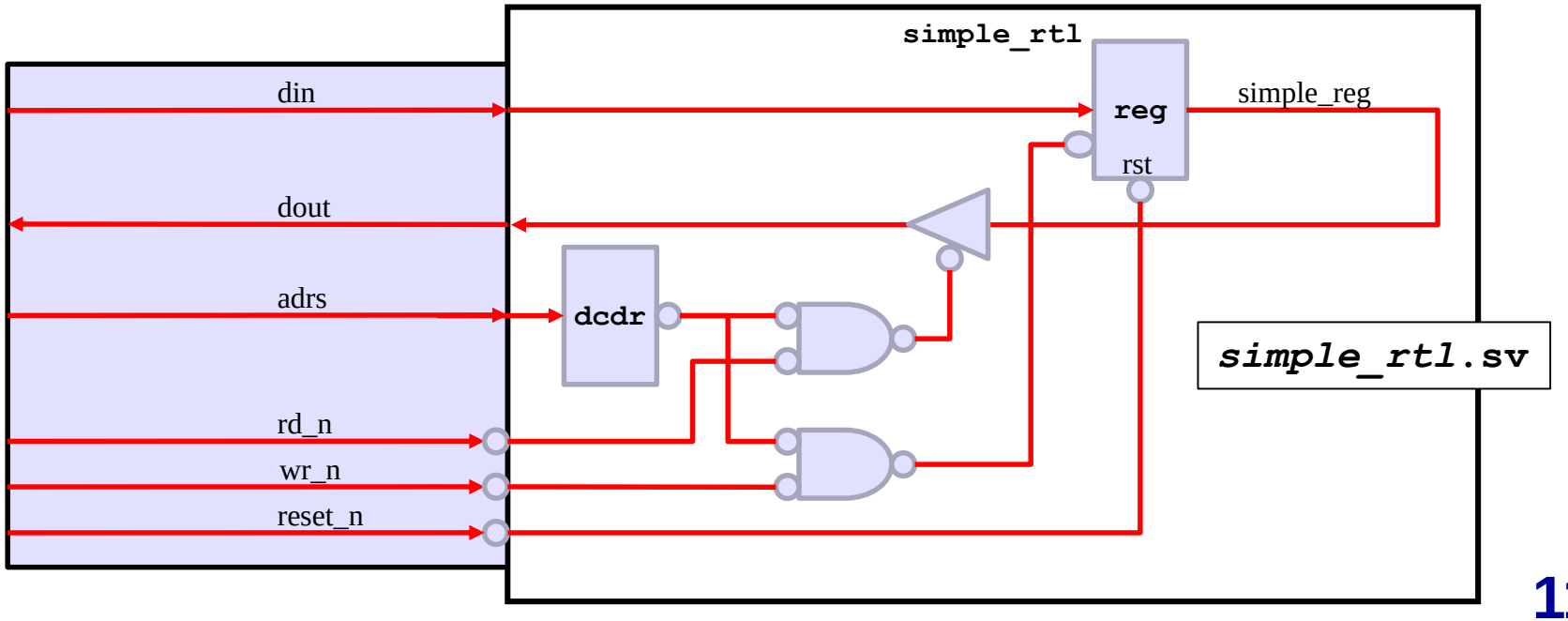
A ralf to UVM Conversion Example (1/4)

simple_rtl.ralf

```
register simple_reg {
  field simple_reg {
    bits 16;
    access rw;
    reset 'h0000;
  }
}
block simple_rtl_regmodel {
  bytes 2;
  register simple_reg   (simple_reg)  @16'h00ff; # Address in map
}
```

Address Map	
simple_reg	0x00ff

simple_reg Register	
Field	simple_reg
Bits	15-0
Mode	rw
Reset	0x0000



A ralf to UVM Conversion Example (2/4)

```
register simple_reg {
  field simple_reg {
    bits 16;
    access rw;
    reset 'h0000;
  }
}
```

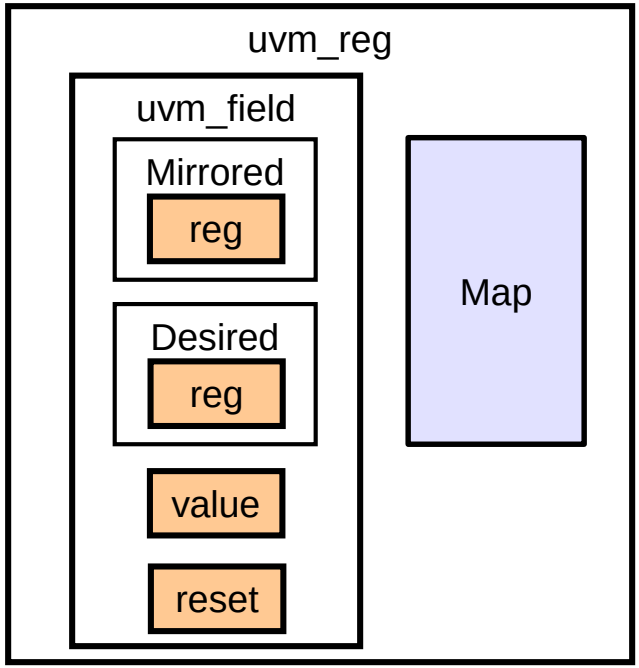
```
block simple_rtl_regmodel {
  bytes 2;
  register simple_reg (simple_reg) @16'h00ff;
}
```

```
ralgen -uvm -t simple_rtl_regmodel simple_rtl.ralf
```

```
class ral_reg_simple_reg extends uvm_reg;
  rand uvm_reg_field simple_reg;
  virtual function void build();
    simple_reg =  uvm_reg_field::type_id::create(
      "simple_reg",,get_full_name());
    simple_reg.configure(this,16,0,"RW",0,16'h0000,1,0,1);
  endfunction: build
endclass
```

A ralf to UVM Conversion Example (3/4)

```
class uvm_reg_field extends uvm_object;
   rand  uvm_reg_data_t  value;  // Used for randomization and coverage
   local uvm_reg_data_t  m_mirrored;      // What we think is in the HW
   local uvm_reg_data_t  m_desired;       // assigned by set()
   local uvm_reg_data_t  m_reset[string]; // reset value
   local string          m_fname;         // field name
   ...// other code not shown
endclass
```



A ralf to UVM Conversion Example (4/4)

```
register simple_reg {
  field simple_reg {
    bits 16;
    access rw;
    reset 'h0000;
  }
}
```

```
block simple_rtl_regmodel {
  bytes 2;
  register simple_reg (simple_reg) @16'h00ff;
}
```

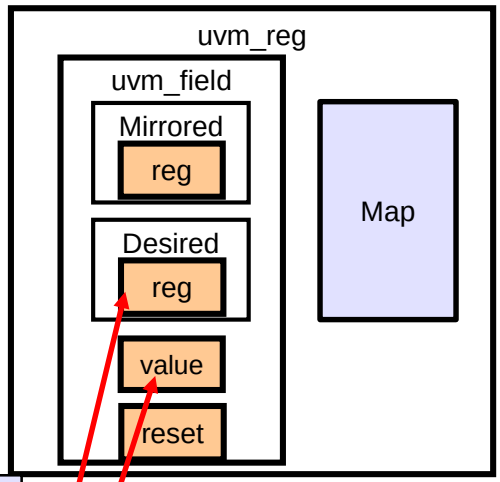
```
ralgen -uvm -t simple_rtl_regmodel simple_rtl.ralf
```

```
class ral_block_simple_rtl_regmodel extends uvm_reg_block;
  rand ral_reg_simple_reg simple_reg;
  rand uvm_reg_field simple_reg_simple_reg;
  virtual function void build();
    default_map = create_map("", 0, 2, UVM_LITTLE_ENDIAN, 0);
    simple_reg = ral_reg_simple_reg::type_id::create(
      "simple_reg",,get_full_name());
    simple_reg.configure(this, null, "");
    simple_reg.build();
    simple_reg.add_hdl_path('{'{"simple_reg", -1, -1}});
    default_map.add_reg(simple_reg,`UVM_REG_ADDR_WIDTH'hFF,"RW",0);
    simple_reg_simple_reg = this.simple_reg.simple_reg;
  endfunction
endclass
```

A Simple Write Example

```
class ral_sequence extends reg_sequence_base;
  ... // other code not shown

  virtual task body();
    uvm_status_e   status;
    uvm_reg_data_t data;
    regmodel.simple_reg.write(status, data, .parent(this));
    regmodel.simple_reg.read(status, data, .parent(this));
  endtask
endclass
```




```
task uvm_reg::write(…);
  set(data);
  rw = new();
  rw.map = this.map;
  rw.data = data;
  rw.map.do_write(rw);
endtask
```

```
task uvm_reg::set(…);
  foreach(field[i])
    field[i].set(data);
endtask
```

```
task uvm_reg_field::set(…);
  desired = data;
  value   = desired;
endtask
```

```
task uvm_reg_map::do_write(…);
  adapter   = get_adapter();
  sequencer = get_sequencer();
  do_bus_write();
endtask
```

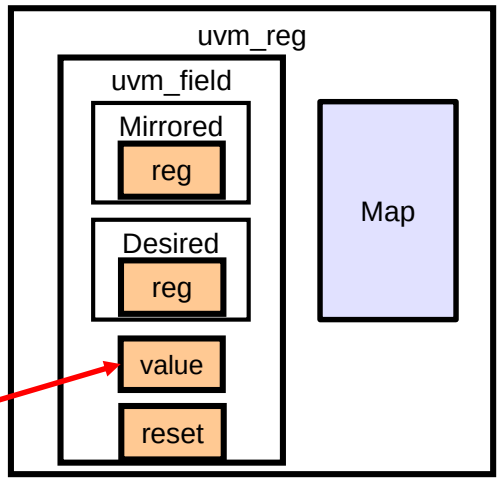
```
task uvm_reg_map::do_bus_write(…);
  seq_item = adapter.req2bus(rw);
  start_item(seq_item);
  finish_item(seq_item);
  adapter.bus2reg();
endtask
```

Caution: Pseudo code, not actual code

A Simple Read Example

```
class ral_sequence extends reg_sequence_base;
  ... // other code not shown

  virtual task body();
    uvm_status_e   status;
    uvm_reg_data_t data;
    regmodel.simple_reg.write(status, data, .parent(this));
    regmodel.simple_reg.read(status, data, .parent(this));
  endtask
endclass
```




```
task uvm_reg::read(…);
  rw = new();
  rw.map = this.map;
  do_read(rw);
  value = rw.value;
endtask
```

```
task uvm_reg::do_read(…);
  rw.map.do_read(rw);
endtask
```

```
task uvm_reg_map::do_read(…);
  adapter   = get_adapter();
  sequencer = get_sequencer();
  do_bus_read();
endtask
```

```
task uvm_reg_map::do_bus_read(…);
  seq_item = adapter.req2bus(rw);
  start_item(seq_item);
  finish_item(seq_item);
  adapter.bus2reg();
endtask
```

Caution: Pseudo code, not actual code

A Simple Backdoor ralgen Example

```
block simple_rtl_regmodel {
  bytes 2;
  register simple_reg (simple_reg) @16'h00ff;
}
```

```
ralgen -uvm -b -t simple_rtl_regmodel simple_rtl.ralf
```

```
class ral_reg_simple_rtl_regmodel_simple_reg_bkdr extends uvm_reg_backdoor;
  virtual task read(uvm_reg_item rw);
    //...
    rw.value[0] = `SIMPLE_RTL_REGMODEL_TOP_PATH.simple_reg;
  endtask
  virtual task write(uvm_reg_item rw);
    `SIMPLE_RTL_REGMODEL_TOP_PATH.simple_reg = rw.value[0];
  endtask
endclass
```

```
class ral_block_simple_rtl_regmodel extends uvm_reg_block;
  virtual function void build();
    // Other code same as shown previously
    ral_reg_simple_rtl_regmodel_simple_reg_bkdr bkdr = new(…);
    simple_reg.set_backdoor(bkdr);
  endfunction : build
endclass : ral_block_simple_rtl_regmodel
```

Need to specify path
At compilation

```
vcs +define+SIMPLE_RTL_REGMODEL_TOP_PATH=simple_test.dut ...
```

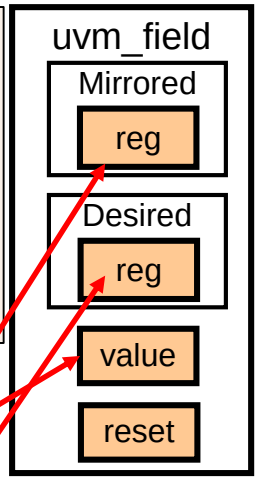
A Simple Backdoor Write Example

```
class backdoor_sequence extends reg_sequence_base;
  virtual task body();
    ...
    regmodel.simple_reg.write(status,data,UVM_BACKDOOR,.parent(this));
    regmodel.simple_reg.read(status, data,UVM_BACKDOOR,.parent(this));
  endtask
endclass
```

```
task uvm_reg::write(…);
  ... // same as previously shown
  rw.map.do_write(rw);
endtask
```

```
task uvm_reg_map::do_write(…);
  ...
  case (rw.path)
    UVM_BACKDOOR: begin
                    uvm_reg_backdoor bkdr = get_backdoor();
                    value = rw.data;
                    bkdr.read(rw);
                    // calculate operational mode result
                    rw.data = final_val;
                    bkdr.write(rw);
                    do_predict(rw);
                    value = final_val;
                  end
    ... // other code not shown
  endcase
endtask
```

```
task uvm_reg_field::do_predict(…);
  ...
  mirrored = final_val;
  desired  = final_val;
endtask
```



Caution: Pseudo code, not actual code

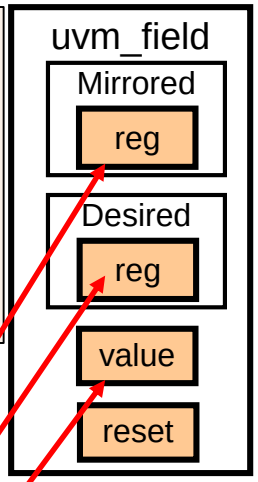
A Simple Backdoor Read Example

```
class backdoor_sequence extends reg_sequence_base;
  virtual task body();
    ...
    regmodel.simple_reg.write(status,data,UVM_BACKDOOR,.parent(this));
    regmodel.simple_reg.read(status, data,UVM_BACKDOOR,.parent(this));
  endtask
endclass
```

```
task uvm_reg::read(…);
  ... // same as previously shown
  do_read(rw);
  value = final_val;
endtask
```

```
task uvm_reg::do_read(…);
  ...
  case (rw.path)
    UVM_BACKDOOR: begin
                    uvm_reg_backdoor bkdr = get_backdoor();
                    value = rw.data;
                    bkdr.read(rw);
                    // calculate operational mode result
                    rw.data = final_val;
                    bkdr.write(rw);
                    do_predict(rw);
                  end
    ... // other code not shown
  endcase
endtask
```

```
task uvm_reg_field::do_predict(…);
  ...
  mirrored = final_val;
  desired  = final_val;
  value    = final_val;
endtask
```

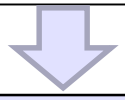


Caution: Pseudo code, not actual code

A Simple Mirror ralgen Example

```
block simple_rtl_regmodel {
  bytes 2;
  register simple_reg (simple_reg) @16'h00ff;
}
```

```
ralgen -uvm -b -auto_mirror -t simple_rtl_regmodel simple_rtl.ralf
```



```
class ral_reg_simple_rtl_regmodel_simple_reg_bkdr extends uvm_reg_backdoor;

  virtual task read(...);  // same as previously shown
  virtual task write(...); // same as previously shown

  virtual function bit is_auto_updated(...);
    if (field.get_name() == "simple_reg") begin return 1; end
  endfunction

  virtual task wait_for_change(...);
    @(`SIMPLE_RTL_REGMODEL_TOP_PATH.simple_reg);
    simple_reg.mirror(status, , UVM_BACKDOOR);
  endtask
endclass
```

Caution: May cause warning message – see note