February 28 – March 1, 2012

Register This! Experiences Applying UVM Registers

by

Kathleen Meade

Verification Solutions Architect

Cadence Design Systems

cādence™

Design & Verification Conference & Exhibition

Register Package: Motivation

- Almost all devices have registers
  - Hundreds (even thousands) of registers is not uncommon
- In verifying a DUT, one needs to control, observe and check register behavior
  - Randomize a configuration and initialize register values
  - Execute transactions to write/read registers and memories
  - Check registers and compare to a reference model
  - Collect coverage of device modes
- Much of the DUT configuration is done at the register level!

DVCon 2012 Design & Verification Conference & Exhibition

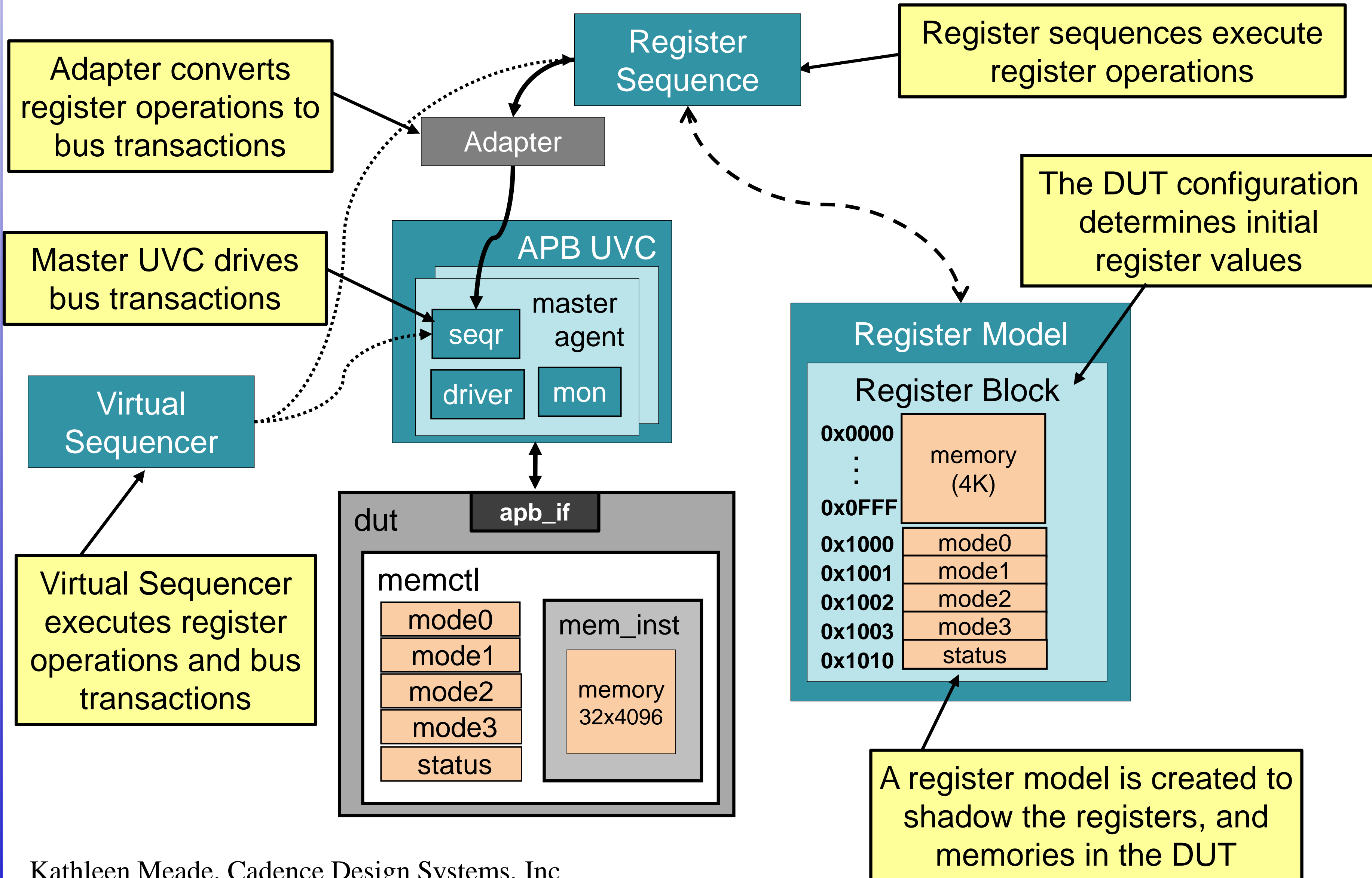
UVM_REG

- UVM_REG is the register and memory package that is included in UVM
  - Streamlines and automates register-related activities
  - Used to model registers and memories in the DUT

- Features
  - Built on top of UVM base classes
  - Access APIs – to write, read, update, peek, get reg values
  - front-door and back-door access to registers and fields
  - Hierarchical architecture
  - Built-in sequences for common register operations

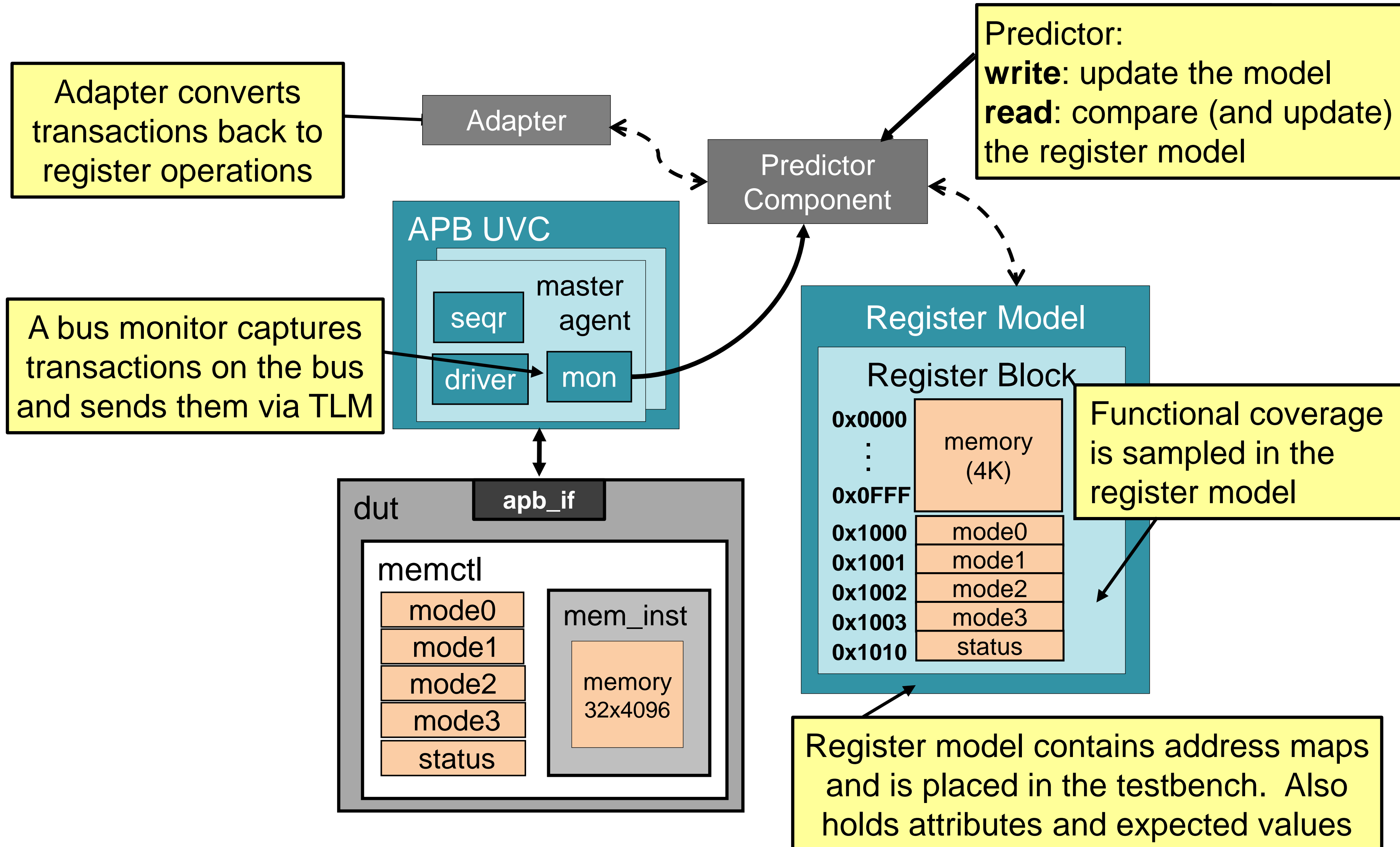
DVCon 2012
Design & Verification Conference & Exhibition

Configuring the DUT with uvm_reg



Kathleen Meade, Cadence Design Systems, Inc

Register Monitoring and Checking



The Register Model

- Defines fields, registers, register blocks and memories for the DUT
  - Includes attributes for registers and register fields
    (size, reset value, compare mask and access)
  - Also tracks the expected values for checking

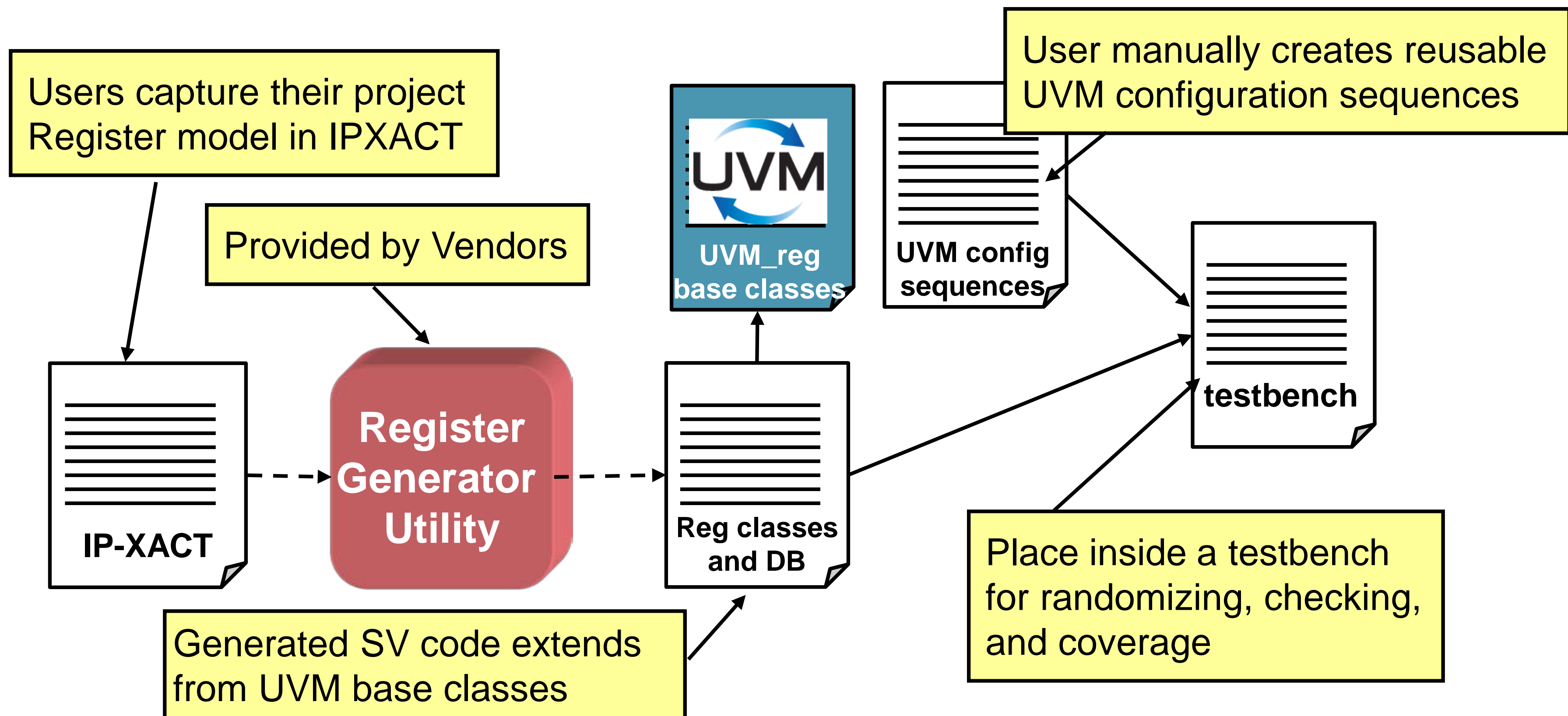
- Register operations are used for register-related stimulus
  - Separating register operations from bus protocols
    - Don't need to learn protocol-specific details
    - Can easily change underlying protocol

- Registers and blocks can be reused within and between projects
  - Configuration sequences can be packaged and reused

Creating the UVM_REG Model

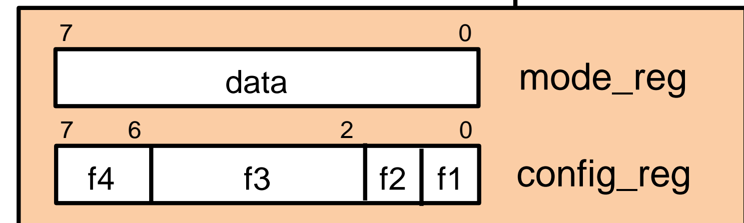
Following the IP-XACT Hierarchy

- IP-XACT is the Accellera XML standard format to capture the register model (driven by the IPXACT sub-committee)



IP-XACT Register Format (XML)

```
<spirit:register>     <!– CONFIG REGISTER -->
    <spirit:name>config_reg</spirit:name>
     <spirit:addressOffset>0x0010</spirit:addressOffset>
     <spirit:size>8</spirit:size>
     <spirit:reset> <spirit:value>0x00</spirit:value>
    <spirit:mask>0xff</spirit:mask> </spirit:reset>
     <spirit:field>     <!– FIELD DEFINITIONS -->
          <spirit:name>f1</spirit:name>
          <spirit:bitOffset>0</spirit:bitOffset>
          <spirit:bitWidth>1</spirit:bitWidth>
        <spirit:access>read-write</spirit:access>
    </spirit:field>
   <spirit:field> <spirit:name>f2</spirit:name>
    <spirit:bitOffset>1</spirit:bitOffset>
        <spirit:bitWidth>1</spirit:bitWidth>
        <spirit:access>read-only</spirit:access>
   </spirit:field>
. . .
</spirit:register>
```



Vendor extensions can be used in the XML file to capture additional register and field dependencies (backdoor path, constraints or coverage info)

Generated Register Definition

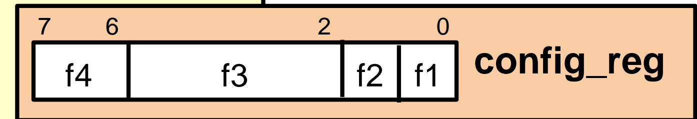
```
class config_reg_c extends uvm_reg;
  rand uvm_reg_field  f1;
  rand uvm_reg_field  f2;
  rand uvm_reg_field  f3;
  rand uvm_reg_field  f4;

  virtual function void build();
    f1 = uvm_reg_field::type_id::create("f1");
    f1.configure(this,  1,  0,  "RW",  0,  'h0,  1,  1,  1);
    f2 = uvm_reg_field::type_id::create("f2");
    f2.configure(this, 1,  1,  "RO",  0,  'h0,  1, 1, 1);
    f3 = uvm_reg_field::type_id::create("f1");
    f3.configure(this, 4,  2,  "RW",  0,  'h0,  1, 1, 1);
    f4 = uvm_reg_field::type_id::create("f2");
    f4.configure(this, 2,  6,  "WO",  0,  'h0,  1, 1, 1);
endfunction

  `uvm_register_cb(config_reg_c, uvm_reg_cbs)
  `uvm_set_super_type(config_reg, uvm_reg)
  `uvm_object_utils(config_reg_c)

  function new (input string name="config_reg_c");
    super.new(name, 8, UVM_NO_COVERAGE);
  endfunction : new
endclass : config_reg_c
```

config_reg fields



In the **build()** method, each field is created, and then configured

Callback and derivation support

Constructor specifies width and coverage options

Register File/Model Declaration

```
class memctl_rf_c extends uvm_reg_block;

  rand mode_reg_c      mode_reg;
  rand config_reg_c    config_reg;

  virtual function void build();
    mode_reg = mode_reg_c::type_id::create("mode_reg", get_full_name());
    config_reg = config_reg_c::type_id::create("config_reg", get_full_name());
    mode_reg.configure(this,  null, "ctl.mode_reg");
    mode_reg.build();
    config_reg.configure(this, null, "ctl.config_reg");
    config_reg.build();
    // define address mappings
    default_map = create_map("default_map", 0, 1, UVM_LITTLE_ENDIAN);
    default_map.add_reg(mode_reg, 'h0, "RW");
    default_map.add_reg(config_reg, 'h10, "RW");
endfunction

`uvm_object_utils(memctl_rf_c)

  function new (input string name="memctl_rf_c");
    super.new(name, UVM_NO_COVERAGE);
  endfunction
endclass
```

Registers (can also contain fields)

In the **build()** method, each register is created, and then configured

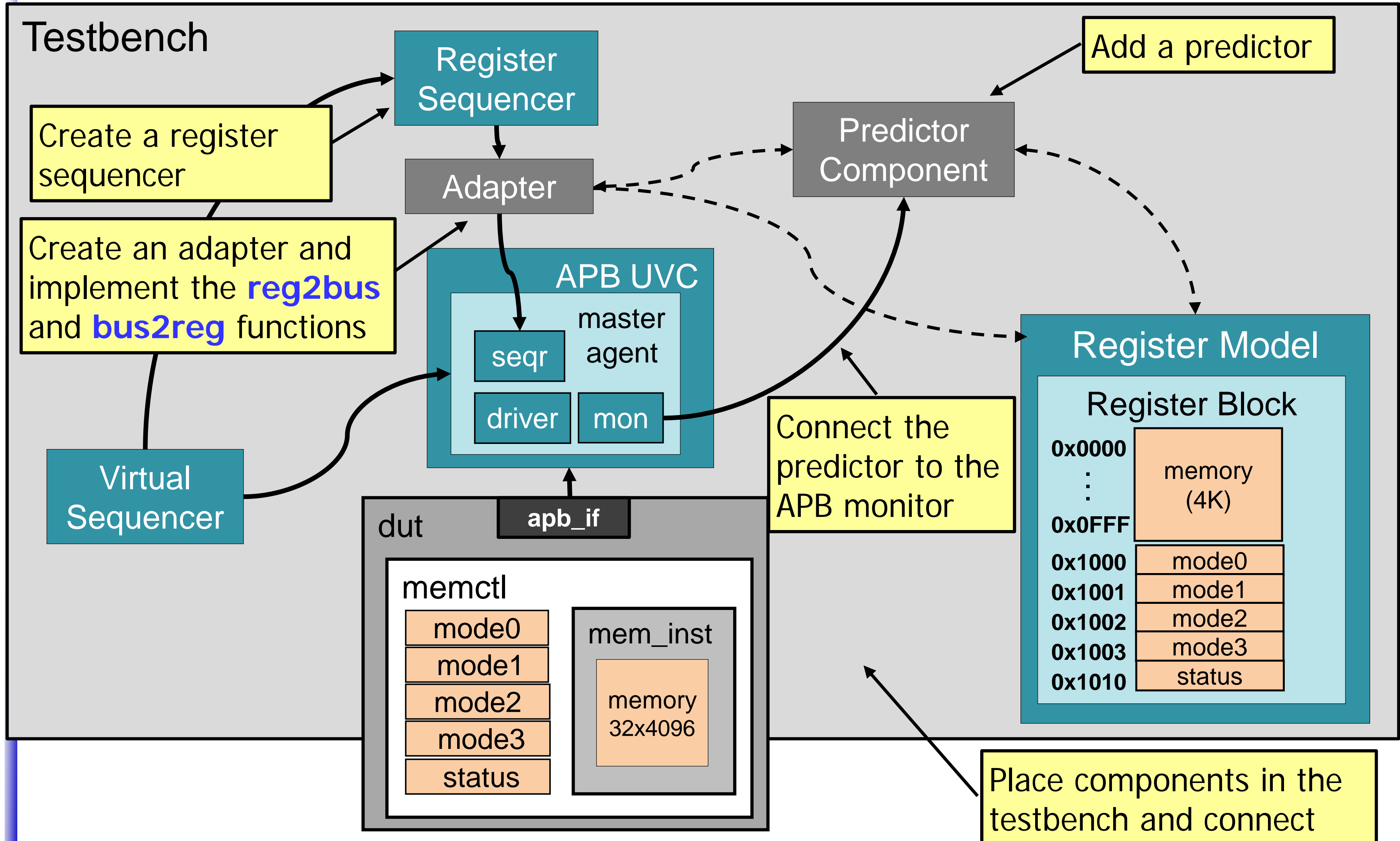
hdl_path for backdoor access

A default map is created with address offset information

Address offset

The register model is similarly hierarchical and contains register files, memories and also registers

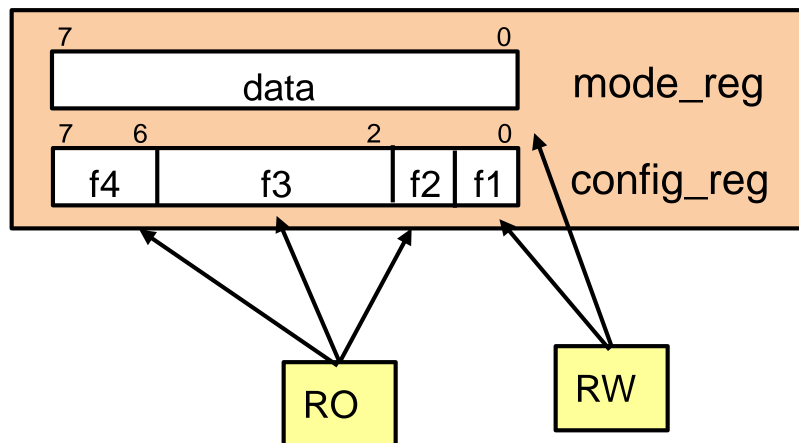
Instantiation and Hook-up



Accessing the Register Model

- Each ***register field*** holds three copies of data:
  - Mirrored: What we think is in the HW
  - **Value**: A value to be randomized
  - **Desired value**: A desired value for the field for reference and comparison

- Has an associated access policy (RW, RO, WO, W1C, etc)



DVCon 2012 Design & Verification Conference & Exhibition

Access APIs for Registers, Fields and Memories

write()/read()	Write/read immediate **value** to the DUT
set()/get()	Sets or gets **desired value** from the register model
randomize()	Randomizes and copies the randomized **value** into the **desired value** of the **mirror** (in post_randomize())
update()	Invokes the **write()** method if the **desired value** (modified using **set()** or **randomize()**) is different from the **mirrored** value
mirror()	Invokes the **read()** method to update the **mirrored** value based on the read back value. Can also compare the read back value with the current **mirrored** value before updating it. (for checking)

Use UVM_BACKDOOR mode to directly access the DUT (via HDL path)

randomize(), update() and mirror() can be called on a container (block) too

UVM_REG write() API

- An example of the task signature for write is:

```
virtual task write( output uvm_status_e          status,
                    input  uvm_reg_data_t        value,
                    input  uvm_path_e            path = UVM_DEFAULT_PATH,
                    input  uvm_reg_map           map = null,
                    input  uvm-sequence_base     parent = null,
                    input  int                   prior = -1,
                    input  uvm_object            extension = null,
                    input  string                fname = "",
                    input  int                   lineno = 0 );
```

default is FRONTDOOR

- Usage of write() inside a sequence looks like this:

```
model.block.mode_reg0.write(status, 8'h34, UVM_BACKDOOR, .parent(this));
model.block.config_reg.write(status, 8'h20, UVM_FRONTDOOR, .parent(this));
```

bound by name instead of position

Configuring Your DUT Using uvm_reg

- Use the register database API to configure the DUT and update the register model
- To maximize automation and reuse use UVM sequences for configuration
- Advantages:
  - Registers are great candidates for vertical reuse
  - Sequences are the most natural way for UVM users
  - Leverage built-in sequence capabilities: grab, lock, priorities, etc
  - Easy system-level control via virtual sequences

Configuration Sequences

DVCon 2012 Design & Verification Conference & Exhibition

```
class config_wr_rd_seq extends uvm_reg_sequence;

  memctl_rf  model;

  rand logic [31:0] mode0, config;

  virtual task body();
    uvm_status_e status;
    model.mode0_reg.write(status, mode0, .parent(this));
    model.config_reg.write(status, config ,  .parent(this));
    model.mode0_reg.read(status, mode0, UVM_BACKDOOR, .parent(this));
    model.config_reg.read(status, config, UVM_BACKDOOR, .parent(this));
 endtask : body

  `uvm_object_utils(config_wr_rd_seq)
  function new ( string name=“config_wr_rd_seq“ );
    super.new(name);
  endfunction : new
 endclass
```

register model

The `uvm_do actions are hidden by the read/write routines

Frontdoor Writes

Backdoor Reads

What about Checking?

Checking for Correctness

- Register checking and coverage is useful
  - Register field values map to DUT operation modes and designers can observe the combinations of configurations that were exercised
- Consistency checking against a mirror/reference can identify errors regardless of the testbench implementation or DUT complexity
- Where do I place the monitoring logic? Directly in sequences where I want to check? or a passive monitor?
- UVM_REG supports two types of monitoring: implicit and explicit

Checking: Implicit Monitoring

- Implicit monitoring
  - The sequence automatically updates the desired value
  - Easy to set up but dangerous, not reusable (no support for passive), no support for other activity on the bus
  - To activate: my_reg_model.default_map.set_auto_predict(1)
- Use the mirror() method n a sequence body and enable the check:

```
// Read and check the mode register via back-door access
model.mode0_reg.mirror(status, UVM_CHECK, UVM_BACKDOOR, .parent(this));

model.mirror(status, UVM_CHECK, UVM_FRONTDOOR, .parent(this));
```

Can also **mirror()** the register model or any sub_container

DVCon 2012
Design & Verification Conference & Exhibition

Checking: Explicit Monitoring

- Explicit Monitoring:
  - The bus monitor and a predictor are used for monitoring and checking
  - Much safer and more reusable
  - Checking logic needs to be added to module UVC
- Separation of the injection and monitoring paths is one of the basic concepts of UVM
- We recommend passive monitoring – independent capture of transactions on the bus that can be recognized as bus operations.
- Note: It's OK to have a check in a sequence body if that is the purpose of the sequence, but want to be able to do independent checking too

Coverage Model in UVM_REG

- The register model can include functional coverage
  - Details of coverage points, bins are left to generator
  - Coverage model can be very large, so instantiate/cover only what needs to be covered
- UVM_REG pre-defined coverage models
  - Register bits (all bits have been read/written)
  - Address maps (addresses have been accessed)
  - Field values (specific values are covered)
- A register generator creates a coverage model for you.
  - Uses IP-XACT vendor-extensions to enable coverage at the field-level
  - Also allows command-line option to enable/disable register-level functional coverage generation

Questions?

Thank You!