February 28 – March 1, 2012

Register This! Experiences Applying UVM Registers

by
Kathleen Meade
Verification Solutions Architect
Cadence Design Systems
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!
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
Configuring the DUT with uvm_reg

- Register Model
  - Register Block
    - 0x0000
    - .
    - 0xFFF
    - 0x1000
    - 0x1001
    - .
    - 0x1010
    - memory (4K)
    - mode0
    - mode1
    - mode2
    - mode3
    - status

- Adapter
  - Converts register operations to bus transactions

- Master UVC
  - Drives bus transactions

- Virtual Sequencer
  - Executes register operations and bus transactions

- DUT configuration determines initial register values

A register model is created to shadow the registers, and memories in the DUT.
Register Monitoring and Checking

Adapter converts transactions back to register operations

A bus monitor captures transactions on the bus and sends them via TLM

APB UVC
  master
  agent
  sequencer
  driver
  monitor

Register Model
  Register Block
    0x0000
    ...  
    0xFFFF
    0x1000
    0x1001
    0x1002
    0x1003
    0x1010
    memory
      (4K)
    mode0
    mode1
    mode2
    mode3
    status

Register model contains address maps and is placed in the testbench. Also holds attributes and expected values

Functional coverage is sampled in the register model

Predictor:
  write: update the model
  read: compare (and update) the register model

A bus monitor captures transactions on the bus and sends them via TLM

Adapter converts transactions back to register operations

Predictor Component

Kathleen Meade, Cadence Design Systems, Inc
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)

Users capture their project Register model in IPXACT

Provided by Vendors

IP-XACT

Register Generator Utility

Generated SV code extends from UVM base classes

User manually creates reusable UVM configuration sequences

UVM_reg base classes

UVM config sequences

testbench

Place inside a testbench for randomizing, checking, and coverage

Generated SV code extends from UVM base classes

Kathleen Meade, Cadence Design Systems, Inc
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 Definition

```plaintext
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

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

default function : new
endclass config_reg_c
```

#### Register Definition

- **`config_reg`** field types:
  - `f1`: RW, width 0, offset 0h0, 1 bit, 1 bit
  - `f2`: RO, width 0, offset 0h0, 1 bit, 1 bit
  - `f3`: RW, width 4, offset 0h0, 1 bit, 1 bit
  - `f4`: WO, width 2, offset 0h0, 1 bit, 1 bit

- In the `build()` method, each field is created, and then configured.

- **Callback and derivation support**:
  - `uvm_register_cb(config_reg_c, uvm_reg_cbs)`
  - `uvm_set_super_type(config_reg, uvm_reg)`
  - `uvm_object_utils(config_reg_c)`

- **Constructor** specifies width and coverage options:
  - `super.new(name, 8, UVM_NO_COVERAGE)`

---

Kathleen Meade, Cadence Design Systems, Inc

9 of 105
Register File/Model Declaration

```vhd
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

Kathleen Meade, Cadence Design Systems, Inc
Instantiation and Hook-up

Testbench

Create a register sequencer

Create an adapter and implement the **reg2bus** and **bus2reg** functions

Virtual Sequencer

Register Sequencer

Add a predictor

APB UVC

Adapter

Connect the predictor to the APB monitor

Register Model

Register Block

Place components in the testbench and connect

Kathleen Meade, Cadence Design Systems, Inc
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)

![Diagram of register model]

- config_reg
- mode_reg
- data
- f4 f3 f2 f1
- RO RW
## Access APIs for Registers, Fields and Memories

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

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
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

Configuration Sequences

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 in a sequence body and enable the check:

```c+
// 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
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!