UVM - What's Now and What's Next
UVM Working Group Update

Adam Sherer, Accellera
DVCon 2014
Agenda

- UVM working group history
- How to contribute to UVM
- Summary and next steps
Formation And Objective

Charter: define standard technology and/or methods to realize a modular, scalable, and reusable generic verification environment

- **Formed February 2008**
  - Co-Chairs: Hillel Miller (Freescale) and Tom Alsop (Intel)

- **Background**
  - IEEE 1800 SystemVerilog lacked standard for VIP creation and use
  - Many methods existed requiring expensive retraining and conversion costs

- **Objective**
  - VIP creation standards to lower verification costs and improve design quality throughout the industry
Key Standard Release Milestones

- **'08**: VIP Recommended Practices standard and OVM/VMM interoperability reference library
- **'09**: UVM 1.0 standard and reference library plus user guide
- **'10**: UVM 1.1 standard and reference library plus user guide (adds register layer)
- **'12**: UVM 1.2 reference library update release
- **'13**: UVM 1.1a early adopter reference library update release
- **'14**: UVM 1.1b reference library update release
- **'15**: UVM 1.1c reference library update release
- **'16**: UVM 1.1d reference library update release
UVM 1.1 Reference in Widespread Use

- Actual standard defines UVM APIs
  - Accellera reference library is commonly identified as “UVM”

- UVM 1.1 defines features that fulfill charter
  - Library includes component definition, factory, messaging system, register layer, and more
  - Enables VIP to scale from block to systems
  - Validated on multiple simulators

- Broadly adopted throughout industry
  - LinkedIn UVM community has 4300 members (600+ since Sept ’13)
  - UVM LinkedIn membership passed OVM April ‘13
UVMWorld Community

- Accellera members and non-members welcome
- Active forums provide fast answers from community of experts
- Links to technical material, tutorials from Accellera sponsored shows, tool/service/VIP providers and more
Contributing to the UVM

- **Employee of Accellera member companies**
  - Contribute your experience in weekly meetings and on reflector topics
  - Code solutions to Mantis items
  - Offer enhancements to the UVM standard and reference implementation

- **Everyone everywhere**
  - Use UVM!
  - Ask and answer question on UVMWorld forums
  - Contribute code, examples, tips, and more in UVMWorld contributions area
  - Report Mantis items – both bugs and enhancements
Summary and Next Steps

- **UVM is a well established industry standard**
  - Supported by multiple simulator providers
  - Extensive library of VIP from multiple providers
  - Service and training available worldwide

- **Active UVM support community**
  - UVMWorld forums and contributions areas
  - Supported by Accellera member and non-member experts

- **UVM is a living standard**
  - UVM 1.2 in development for delivery in 2013
  - Improving messaging, phasing, and many other features
Thank you
UVM - What's Now and What's Next
UVM Overview and Library Concepts

John Aynsley, Doulos
UVM Overview and Library Concepts

- What and Why?
- UVM Highlights
- The Big Picture
- Execution Phases
- The Agent
- Hello World Example
What is UVM?

- The Universal Verification Methodology for SystemVerilog
- An open source base class library
- Supports constrained random, coverage-driven verification
- A standard!
Why UVM?

- Best practice
  - Consistency, uniformity, don't reinvent the wheel, avoid pitfalls

- Reuse
  - Verification IP, verification environments, tests, people, knowhow
UVM Highlights

- Separation of tests from test bench
- Transaction-level communication (TLM)
- Factory and configuration
- Sequences (stimulus)
- End-of-test mechanism (objections)
- Message reporting
- Register layer
Test versus Testbench

Test writer

Verification specialist

Test1

Test2

Test3

Reusable verification environment

DUT

Tests define differences

Allows for unanticipated changes
The Big Picture

A consistent structure

Factory  config db

uvm_test

uvm_env

uvm_scoreboard  Virtual sequence

Register Layer

uvm_agent  uvm_agent  uvm_agent

DUT
Quasi-static vs Dynamic Objects

```verilog
class my_comp extends uvm_component;
`uvm_component_utils(my_comp)

function new (string name, uvm_component parent);
    super.new(name, parent);
endfunction
...
endclass

class my_tx extends uvm_sequence_item;
`uvm_object_utils(my_tx)

function new (string name = "");
    super.new(name);
endfunction
...
endclass
```
Execution Phases

- build
- connect
- end_of_elaboration
- start_of_simulation
- run
- extract
- check
- report
- final

- pre_reset
- reset
- post_reset
- pre_configure
- configure
- post_configure
- pre_main
- main
- post_main
- pre_shutdown
- shutdown
- post_shutdown

TL connections
Build and Connect Phases

class my_agent extends uvm_agent;
  `uvm_component_utils(my_agent)

  my_driver driv;
  my_sequencer seqr;
  my_monitor mon;
  ...

  function void build_phase(uvm_phase phase);
    driv = my_driver ::type_id::create("driv", this);
    seqr = my_sequencer::type_id::create("seqr", this);
    mon = my_monitor ::type_id::create("mon", this);
  endfunction

Port Export

Factory
Build and Connect Phases

class my_agent extends uvm_agent;
   `uvm_component_utils(my_agent)

my_driver   driv;
my_sequencer seqr;
my_monitor  mon;
...

function void build_phase(uvm_phase phase);
   driv = my_driver ::type_id::create("driv", this);
   seqr = my_sequencer::type_id::create("seqr", this);
   mon  = my_monitor ::type_id::create("mon", this);
endfunction

function void connect_phase(uvm_phase phase);
   driv.seq_item_port.connect( seqr.seq_item_export );
endfunction

my_driver::type_id::set_type_override(alt_driver::get_type());
Agent Architecture

Checking and Coverage

Constrained random stimulus generation

Agent-per-interface

DUT

Monitor

Driver

Sequencer

Sequence

Env

Agent

Test

Test

© 2014 Accellera Systems Initiative, Inc.
Sequencer-Driver Communication
interface dut_if;
    logic clock, reset;
    logic cmd;
    logic [7:0] addr;
    logic [7:0] data;
endinterface

data 'uvm_pkg::*;

class my_agent extends uvm_agent;
    `uvm_component_utils(my_agent)
    ...
endclass

class my_env extends uvm_env;
    `uvm_component_utils(my_env)
    ...
    my_agent m_agent;
    function void build_phase(uvm_phase phase);
        m_agent = my_agent::type_id::create(...);
    endfunction
endclass

class my_test extends uvm_test;
    `uvm_component_utils(my_test)
    ...
    function void build_phase(uvm_phase phase);
        m_env = my_env::type_id::create("m_env", this);
    endfunction
endclass

task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    #10;
    `uvm_info("", "Hello World", UVM_MEDIUM)
    phase.drop_objection(this);
endtask
endclass
endpackage: my_pkg

module top;
    import uvm_pkg::*;
    import my_pkg::*;
    dut_if dut_if1 ();
    dut dut1 ( .dif(dut_if1) );
    initial
        begin
            run_test("my_test");
        end
endmodule

module dut(dut_if dif);
endmodule

module top;
    import uvm_pkg::*;
    import my_pkg::*;
    dut_if dut_if1 ();
    dut dut1 ( .dif(dut_if1) );
    initial
        begin
            run_test("my_test");
        end
endmodule
module dut(dut_if dif);
    import uvm_pkg::*;

    always @(posedge dif.clock)
    begin
        `uvm_info("",
            $sformatf("DUT received cmd=%b, addr=%d, data=%d", dif.cmd, dif.addr, dif.data), UVM_MEDIUM)
    end
endmodule

module dut(dut_if dif);
    import uvm_pkg::*;
    dut_if dut_if1 ();
    dut dut1 ( .dif(dut_if1) );
    initial
    begin
        run_test("my_test");
    end
endmodule
Hello World Source Code

```verilog
module top;

    import uvm_pkg::*;
    import my_pkg::*;

    dut_if dut_if1 ();

    dut dut1 ( .dif(dut_if1) );

    initial begin
        uvm_config_db #(virtual dut_if)::set(
            null, "*", "dut_if", dut_if1);
        run_test("my_test");
    end

endmodule
```

```verilog
module dut(dut_if dif);
endmodule
```

```verilog
package my_pkg;
    import uvm_pkg::*;
endpackage: my_pkg
```

```verilog
interface dut_if;
endinterface
```
package my_pkg;

import uvm_pkg::*;

class my_agent extends uvm_agent;
  `uvm_component_utils(my_agent)

  my_driver   driv;
  my_sequencer seqr;
  my_monitor  mon;

  function void build_phase(uvm_phase phase);
    driv = my_driver  ::type_id::create("driv", this);
    seqr = my_sequencer::type_id::create("seqr", this);
    mon  = my_monitor  ::type_id::create("mon", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    driv.seq_item_port.connect( seqr.seq_item_export );
  endfunction

endclass

class my_env extends uvm_env;
  `uvm_component_utils(my_env)

  my_agent m_agent;

  function void build_phase(uvm_phase phase);
    m_agent = my_agent::type_id::create(...);
  endfunction

endclass

class my_test extends uvm_test;
  `uvm_component_utils(my_test)

  function void build_phase(uvm_phase phase);
    m_env = my_env::type_id::create("m_env", this);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    #10;
    `uvm_info("", "Hello World", UVM_MEDIUM)
    phase.drop_objection(this);
  endtask

endclass

module top;
  import uvm_pkg::*;
  import my_pkg::*;

  dut_if dut_if1 ();
  dut dut1 ( .dif(dut_if1) );

  initial
  begin
    run_test("my_test");
  end

endmodule

interface dut_if;
endinterface

module dut(dut_if dif);
endmodule
package my_pkg;

class my_env extends uvm_env;
    `uvm_component_utils(my_env)

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

    my_agent m_agent;

    function void build_phase(uvm_phase phase);
        m_agent = my_agent::type_id::create("agent", this);
    endfunction

endclass

class my_test extends uvm_test;
    `uvm_component_utils(my_test)

    function void build_phase(uvm_phase phase);
        m_env = my_env::type_id::create("m_env", this);
    endfunction

    task run_phase(uvm_phase phase);
        phase.raise_objection(this);
        #10;
        `uvm_info("", "Hello World", UVM_MEDIUM)
        phase.drop_objection(this);
    endtask

endclass

dpath dut_if;

module dut(dut_if dif);

module top;
import uvm_pkg::*;
import my_pkg::*;
dut_if dut_if1 ();
dut dut1 ( .dif(dut_if1) );
initial
begin
    run_test("my_test");
end
endmodule
Hello World Source Code

```verilog
interface dut_if;
endinterface

module dut(dut_if dif);
endmodule

module top;
import uvm_pkg::*;
import my_pkg::*;
dut_if dut_if1 ();
dut dut1 ( .dif(dut_if1) );
initial
begin
run_test("my_test");
end
endmodule

class my_test extends uvm_test;
`uvm_component_utils(my_test)

function void build_phase(uvm_phase phase);
    m_env = my_env::type_id::create("m_env", this);
endfunction

task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    #10;
    `uvm_info("", "Hello World", UVM_MEDIUM)
    phase.drop_objection(this);
endtask
endclass
```
Hello World Source Code

```verilog
interface dut_if;
endinterface

module dut(dut_if dif);
endmodule

module top;

import uvm_pkg::*;
import my_pkg::*;

dut_if dut_if1 ();
dut dut1 (.dif(dut_if1));

initial
begin
  run_test("my_test");
end
endmodule

package my_pkg;

import uvm_pkg::*;

class my_agent extends uvm_agent;
  `uvm_component_utils(my_agent)
  ...
endclass

class my_env extends uvm_env;
  `uvm_component_utils(my_env)
  my_agent m_agent;
  function void build_phase(uvm_phase phase);
    m_agent = my_agent::type_id::create(...);
    endfunction
endclass

class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  ...
  function void build_phase(uvm_phase phase);
    m_env = my_env::type_id::create("m_env", this);
    endfunction
  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    #10;
    `uvm_info("", "Hello World", UVM_MEDIUM)
    phase.drop_objection(this);
    endtask
  endclass
endpackage: my_pkg
```

Always Remember

- **Best practice**
  - Consistency, uniformity, don't reinvent the wheel, avoid pitfalls

- **Reuse**
  - Verification IP, verification environments, tests, people, knowhow
Thank you
Some O.O.P. Concepts

- Part of a testbench...

**Verilog**
```verilog
reg [31:0] addr;
reg [31:0] data[];

function void print(
    bit[31:0] addr,
    bit[31:0] data[]);
```

**System Verilog**
```systemverilog
class transaction_item;
    bit [31:0] addr;
    bit [31:0] data[];

    function void print();
endclass
```

Objects travel across these arrows. These are not method calls!

```systemverilog
// System Verilog
xact.print();
```
An Example Sequence Item

class bus_transaction extends uvm_sequence_item;

typedef enum { READ, WRITE } type_e;

`uvm_object_utils_begin(bus_transaction)
  `uvm_field_enum(type_e, type, UVM_ALL_ON | UVM_NOPACK)
  `uvm_field_int(         addr, UVM_ALL_ON | UVM_NOPACK)
  `uvm_field_int(       length, UVM_ALL_ON | UVM_NOPACK)
  `uvm_field_array_int(   data, UVM_ALL_ON)
`uvm_object_utils_end

rand type_e type;
rand bit[31:0] addr;
rand int unsigned length;
rand bit[ 7:0] data[];

constraint c_len { data.size() == length;  length < 256; }

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

endclass

Macros build methods for print, compare, etc.

Properties of a transaction are kept together
Stimulus

- UVM sequences give a flexible and procedural mechanism for creating “constrained random” (or directed) stimulus.
- A coverage plan directs the verification effort.
- Sequences are “objects” that contain a method to produce the desired stimulus, often a stream of “sequence_items.”
An Example Sequence

```verbatim
class bus_rw_sequence extends uvm_sequence #(bus_transaction);
`uvm_object_utils(bus_rw_sequence)
...
virtual task body();
  uvm_config_db#(int)::get(
    get_sequencer(), "", "seq_len", seq_len);
  repeat (seq_len) begin
    `uvm_create("req")
    start_item(req); // sequencer.wait_for_grant(...);
    if (!req.randomize()) `uvm_fatal(...)
    finish_item(req); // sequencer.send_request(...);
    // sequencer.wait_for_item_done(...) 
  end 
endtask
endclass
```

Sequences are template classes

Often provide "knobs" for customization
class bus_rw_sequence extends uvm_sequence #(bus_transaction);
`uvm_object_utils(bus_rw_sequence)
...
virtual task body();
  uvm_config_db#(int)::get(null, get_full_name(), "len", len);
  repeat (len) begin
    // create a new bus transaction, randomize it, send it ...
    `uvm_do_with(req, { type == WRITE; })
    addr = req.addr;
    // create a new bus transaction, randomize it, send it ...
    `uvm_do_with(req, { type == READ; addr == local::addr; })
  end
endtask
endclass

A variety of sequence macros simplify tasks
Sequence Macros

- Create, Randomize and Send a sequence_item or sub-sequence:

  ```
  `uvm_do()
  `uvm_do_with() // add a constraint block
  `uvm_do_on()  // performs on the specified sequencer
  `uvm_do_pri() // supplies a priority
  ```

- Manual creation and sending of sequence_item or sub-sequence:

  ```
  `uvm_create() // just build the object
  `uvm_send()   // "start" the sequence
  ```

- Use of these macros is optional. Alternatively...

  ```
  start_item()  // arbitrate at sequencer
  finish_item() // blocks until item is done
  ```
Sequencers

- Sequencers are components that select and “run” sequences.
- Within an agent, sequencers typically send data directly to a uvm_driver via a built-in TLM export (and collect responses).
- Sequencer components are often used without subclassing.

```verbatim
class bus_agent extends uvm_agent;
  uvm_sequencer #(bus_transaction) seqr;
  ... 

class my_sequencer extends uvm_sequencer
  port_configuration cfg;
  subsequence subseq;
  ... 
```
Virtual Sequences

- No connection to a uvm_driver
- Typically launches sequences on other sequencers
  - “layered” or “hierarchical” sequences

```verilog
class bus_clash_sequence extends uvm_sequence;
    uvm_sequencer#(bus_transaction) mseqr[2];
    bus_transaction_sequence seq[2];
...
virtual task body();
    fork
        `uvm_do_on_with(seq[0], mseqr[0], { addr == 8'h23; })
        `uvm_do_on_with(seq[1], mseqr[1], { addr == 8'h23; })
    join
endtask
endclass
```

These two sequences run in parallel on different sequencers
Starting Sequences

- A user can manually start sequences, typically within a test.

```cpp
virtual task my_test::main_phase(uvm_phase phase);
    stim_seq  seq;
    ...
    seq = stim_seq::type_id::create("seq", this);
    seq.starting_phase = phase;
    seq.start(env.agent[3].sequencer);
```

- A user can start a sequence from within another sequence.

```cpp
virtual task top_seq::body();
    stim_seq  seq;
    `uvm_do_on(seq, env.agent[3].sequencer)
endtask
```

- A sequencer starts a “default_sequence” for each phase.

```cpp
virtual function my_test::build_phase(uvm_phase phase);
    ...
    uvm_config_db#(uvm_object_wrapper)::set(this, "env.agent.bus_sequencer.configure_phase",
        "default_sequence", reg_cfg_seq::type_id::get());
```
Objection Management

- You often want to keep the phase from progressing until your sequence has finished.
- You must either...
  - Manually manage objections
  - Build sequences that manage objections
- Many different approaches are in wide use for as many reasons.

```cpp
class interesting_sequence extends uvm_sequence#(data_item);
...
virtual task pre_start(); // callback method
  if ((get_parent_sequence() == null) &&
      (starting_phase != null)) // UVM 1.1 only
    starting_phase.raise_objection(this, "Running seq...");
endtask
...
```

Callback methods are invoked by start() method
UVM Tests

- Typically a uvm_test will construct the testbench environment.
  - One uvm_test is created per simulation.
  - That single test forms the uppermost-level of the UVM component hierarchy.

```vhdl
class base_test extends uvm_test;
  `uvm_component_utils(base_test)

  env   env_h;

  virtual function void build_phase(uvm_phase phase);
    env_h = env::type_id::create("env_h", this);
  endfunction
endclass
```

- Users typically extend a “base_test,” choosing and configuring different sequences to complete the verification plan.
Example UVM Test

```verilog
class bus_long_test extends base_test;
  `uvm_component_utils(bus_long_test)
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    set_type_override_by_type(bus_transaction::get_type(),
                              big_bus_transaction::get_type());
  uvm_config_db#(uvm_object_wrapper)::set(this,
                         "env.agent.bus_sequencer.main_phase",
                         "default_sequence", bus_basic_seq::get_type());
  uvm_config_db#(int)::set(this, "env.agent.bus_sequencer.*", "seq_len", 1000);
endfunction
endclass
```

“env” built by the base class

Substitute “big” transactions

Run this sequence

Long test, 1000 transactions...
It All Starts With a UVM Test

- An initial block, typically in the “test_top” will launch everything!

```verilog
module test_top;
    import uvm_pkg::*;
    dut dut(...);
    bus_if bus_if(...);
    initial begin
        uvm_config_db#(virtual bus_if)::set(uvm_root::get(),
            "uvm_test_top.env.bus_agent", "vif", bus_if);
        run_test(); // global task, imported from uvm_pkg
    end
endmodule: test_top
```

Give interface pins to the testbench agent

Builds the test object, then starts phasing

- Choose a test to run on the command line:

```bash
%> simv +UVM_TESTAME=bus_long_test
%> simv +UVM_TESTAME=another_test
```
Thank you
UVM - What's Now and What's Next
UVM Register Layer
Tom Fitzpatrick, Mentor Graphics
UVM Registers are Layered

- UVM Register Layer provides protocol-independent register-based layering

Device specific
```
cfg.write(0xDE);
```

Bus specific
```
wr(0xAF, 0xDE);
```
UVM Register Use Models

- **Stimulus Generation**
  - Abstraction of stimulus:
    - i.e., Set this bit in this register rather than write x to address y
  - Stimulus reuse
    - If the bus agent changes, the stimulus still works
  - Front and Back Door access:
    - Front door is via an agent
    - Back door is directly to the hardware via the simulator database

- **Configuration**
  - Register model reflects hardware programmable registers
  - Set up desired configuration in register model then dump to DUT
    - Randomization with configuration constraints

- **Analysis 'Mirror'**
  - Current state of the register model matches the DUT hardware
  - Useful for scoreboards and functional coverage monitors
Register Information Model

Registers contain bits & fields

Reg Blocks contain registers & memories

Blocks contain Address Maps

Reg Blocks correspond to design registers

Blocks are hierarchical

One map per interface

---

Register Blocks contain registers & memories

Address Maps are hierarchical

Each interface has a unique address map

Reset Value: 0x00000000

Table 6: Control and Status Register

<table>
<thead>
<tr>
<th>Bit #</th>
<th>31:14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>R</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>Name</td>
<td>Reserved</td>
<td>ASS</td>
<td>IE</td>
<td>LSB</td>
<td>Tx_Neg</td>
<td>Rx_Neg</td>
<td>Go_Bsy</td>
<td>Rsvr</td>
<td>Char_Len</td>
</tr>
</tbody>
</table>

---

© 2014 Accellera Systems Initiative, Inc
class csr_reg extends uvm_reg;
`uvm_object_utils(csr_reg)

uvm_reg_field reserved;
rand uvm_reg_field char_len;

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

virtual function void build();
  char_len = uvm_reg_field::type_id::create("char_len");
  char_len.configure(this, 7, 0, "RW", 0, 7'h7f, 1, 1, 1);
endfunction
endclass
Registers, Blocks & Maps
Registers, Blocks & Maps

class spi_reg_block extends uvm_reg_block;
  `uvm_object_utils(spi_reg_block)

  rand csr_reg csr;
  ...
  uvm_reg_map APB_map; // Block map

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

  virtual function void build();
  endfunction: build

endclass: spi_reg_block

Address Map points to Registers

Register Block contains Maps

One Map per physical interface
class spi_reg_block extends uvm_reg_block;
`uvm_object_utils(spi_reg_block)

virtual function void build();
    csr = csr_reg::type_id::create("csr");
    csr.configure(this, null, "");
    csr.build();
    csr.add_hdl_path_slice("csr", 0, 7);
    csr.add_hdl_path_slice("csr_dff.q", 0, 7, "GATES");

    APB_map = create_map("APB_map", 'h800, 4,
                           UVM_LITTLE_ENDIAN);
    APB_map.add_reg(csr, 32'h0000014, "RW");
    add_hdl_path("top.DUT", "RTL");
    add_hdl_path("top.board.DUT", "GATES");
    lock_model();
endfunction: build

class spi_reg_block extends uvm_reg_block;
`uvm_object_utils(spi_reg_block)
Register Blocks are Hierarchical

class soc_block extends uvm_reg_block;
`uvm_object_utils(soc_block)

spi_reg_blk spi_regs;
wsh_reg_blk wsh_regs;
function new(string name = "soc_block");
    super.new(name, UVM_NO_COVERAGE);
endfunction

virtual function void build();
    default_map = create_map("", 'h0, 4,
        UVM_LITTLE_ENDIAN);
    spi_regs = spi_reg_blk::type_id::create(
        "spi_regs", get_full_name());
    spi_regs.configure(this, "spi_regs");
    spi_regs.build();
    default_map.add_submap(spi_regs.default_map, 'h0000);
    ...
    default_map.add_submap(wsh_regs.default_map, 'h1000);
endfunction: build

datatype

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Reserved</td>
</tr>
<tr>
<td>R/W</td>
<td>ASS</td>
</tr>
<tr>
<td>R/W</td>
<td>IE</td>
</tr>
<tr>
<td>R/W</td>
<td>LSB</td>
</tr>
<tr>
<td>R/W</td>
<td>TxNeg</td>
</tr>
<tr>
<td>R/W</td>
<td>RxNeg</td>
</tr>
<tr>
<td>R</td>
<td>GoBsy</td>
</tr>
<tr>
<td>R</td>
<td>Rrv</td>
</tr>
<tr>
<td>R/W</td>
<td>Char_Len</td>
</tr>
</tbody>
</table>

Address Map points to Registers

Base Address

Blocks are hierarchical
class spi_env extends uvm_env;
    `uvm_component_utils(spi_env)

    function void build_phase(uvm_phase phase);
        if(!uvm_config_db #(spi_env_config)::get(this, "", "spi_env_config", m_cfg))
            begin `uvm_error("build_phase", "Failed to find spi_env_config") end

    endfunction:build_phase

    function void connect_phase(uvm_phase phase);
        if(m_cfg.m_apb_agent_cfg.active == UVM_ACTIVE) begin
            reg2apb = reg2apb_adapter::type_id::create("reg2apb");
            if(m_cfg.spi_rm.get_parent() == null) begin
                m_cfg.spi_rm.APB_map.set_sequencer(
                    m_apb_agent.m_sequencer, reg2apb);
                m_cfg.spi_rm.APB_map.set_auto_predict(0); //default
            end
        end

    endfunction:connect_phase
endclass

Setting Up the Register Map
How Do Register Accesses Work?

- When an explicit register access method is called
  - The register layer uses a generic register command:
    - Register.[Read / Write](data)

- The register transaction is passed to the address map
  - The map’s adapter (extended from uvm_reg_adapter) converts it to a bus transaction

- This is then sent through a layering to the target bus agent
How Do Register Accesses Work?

- The predictor updates the value of the register model
  - Bus transaction (from monitor) converted back to Reg transaction
  - Write: Value that was written to DUT is reflected
  - Read: Value that was read from DUT is reflected

- The predictor then writes the register transaction out its analysis_port
  - Generic register requests to target bus sequence items
# Register Access Method Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>uvm_status_e</td>
<td>status</td>
<td>Indicates Access completed OK</td>
</tr>
<tr>
<td>uvm_reg_data_t</td>
<td>value</td>
<td>Data value transferred</td>
</tr>
<tr>
<td>uvm_path_e</td>
<td>path</td>
<td>Front or back door access</td>
</tr>
<tr>
<td>uvm_reg_map</td>
<td>map</td>
<td>Map to use for access</td>
</tr>
<tr>
<td>uvm_sequence_base</td>
<td>parent</td>
<td>Parent sequence</td>
</tr>
<tr>
<td>int</td>
<td>prior</td>
<td>Sequence priority on sequencer</td>
</tr>
<tr>
<td>uvm_object</td>
<td>extension</td>
<td>Transfer extension object</td>
</tr>
<tr>
<td>string</td>
<td>fname</td>
<td>Filename (For reporting)</td>
</tr>
<tr>
<td>int</td>
<td>lineno</td>
<td>Line number (For reporting)</td>
</tr>
</tbody>
</table>

- **Good news – most of these fields have defaults!**
- **A typical register access only needs a few of these:**

```c
spi_rm.ctrl.write(status, wdata, .parent(this));
```
Front-Door Access Modes

- **Consume time on the bus (default)**
  
  ```
  write_reg(model.write, status, wdata, UVM_FRONTDOOR); .parent(this));
  read_reg(model.read, status, rdata, UVM_FRONTDOOR); .parent(this));
  ```

- **Desired and Mirrored values updated at end of transaction**
  - Based on transaction contents and field access mode

- **Can access individual fields**
  - Only if hardware supports it
    - Field = byte lane
Back-Door Access Modes

- Consume no time on the bus
  - `write_reg(model.ctrl, status, wdata, UVM_BACKDOOR);`
  - `read_reg(model.ctrl, status, rdata, UVM_BACKDOOR);`
    - Must be specified explicitly

- Desired and Mirrored values updated at end of transaction
  - Based on transaction contents and field access mode

- Can only access full register via back door
# Back-Door Access Modes

- **Consume no time on the bus**
  - `poke_reg (model.ctrl, status, wdata);`
  - `peek_reg (model.ctrl, status, rdata);`

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

- **Peek/Poke work on fields**
Direct Access Modes

- Consume no time on the bus
- Access the Desired Value directly

Use `update()` method to update actual value

- via frontdoor: `update_reg(model.ctrl, status, UVM_FRONTDOOR);`
- or backdoor: `update_reg(model.ctrl, status, UVM_BACKDOOR);`

model.ctrl.set(wdata);
model.ctrl.randomize();
rdata = model.ctrl.get();

set + update = write/poke
Mirror Method

- Read register and update/check mirror value
  - via frontdoor
    
    ```
    mirror_reg(model.ctrl, status, UVM_CHECK, UVM_FRONTDOOR);
    ```
  - or backdoor
    
    ```
    mirror_reg(model.ctrl, status, UVM_CHECK, UVM_BACKDOOR);
    ```

- Can be called on field, reg or block
class reg2ahb_adapter extends uvm_reg_adapter;
`uvm_object_utils(reg2ahb_adapter)

function new(string name = "reg2ahb_adapter");
  super.new(name);
endfunction

virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
  ahb_seq_item ahb = ahb_seq_item::type_id::create("ahb");
  ahb.HWRITE = (rw.kind == UVM_READ) ? AHB_READ : AHB_WRITE;
  ahb.HADDR = rw.addr;
  ahb.DATA = rw.data;
  return ahb;
endfunction

endclass: reg2ahb_adapter
class reg2ahb_adapter extends uvm_reg_adapter;
`uvm_object_utils(reg2ahb_adapter)

function new(string name = "reg2ahb_adapter");
  super.new(name);
endfunction

virtual function void bus2reg(uvm_sequence_item bus_item,
  ref uvm_reg_bus_op rw);
  ahb_seq_item ahb;
  if (!$cast(ahb, bus_item)) begin
    `uvm_fatal("NOT_AHB_TYPE","Wrong type for bus_item")
  end
  rw.kind = (ahb.HWRITE == AHB_READ) ? UVM_READ : UVM_WRITE;
  rw.addr = ahb.HADDR;
  rw.data = ahb.DATA;
  rw.status = ahb.status ? UVM_IS_OK : UVM_NOT_OK;
endfunction

endclass: reg2ahb_adapter

Register Adapter Class Example

bus2reg() converts bus item to reg operation

UVM_HAS_X also legal
class spi_env extends uvm_env;

apb_agent m_apb_agent;
spi_env_config m_cfg;
reg2apb_adapter reg2apb;
uvm_reg_predictor #((apb_seq_item) apb2reg_predictor);

function void connect_phase(uvm_phase phase);
if(m_cfg.ss_rm == null) begin
`UVM_FATAL("spi_env", "No Register Model found in m_cfg")
end else begin
reg2apb = reg2apb_adapter::type_id::create("reg2apb");
m_cfg.ss_rm.TOP_map.set_sequencer(m_apb_agent.m_sequencer, reg2apb);
apb2reg_predictor.map = m_cfg.ss_rm.TOP_map;
apb2reg_predictor.adapter = reg2apb;
m_apb_agent.ap.connect(apb2reg_predictor.bus_in);
end
endfunction: connect_phase
Register Sequence Base Class

class uvm_reg_sequence#(type BASE=uvm_sequence#(uvm_reg_item))
    extends BASE;
    `uvm_object_param_utils(uvm_reg_sequence #(BASE))

    uvm_reg_block model;
    ...  
endclass
class blk_R_test_seq
  extends uvm_reg_sequence;
  `uvm_object_utils(blk_R_test_seq)

  reg_block_B model;

  function new(string name = "blk_R_test_seq");
    super.new(name);
  endfunction: new

  virtual task body();
    uvm_status_e status;
    uvm_reg_data_t data, rd_data;

    write_reg(model.R, status, data);
    read_reg (model.R, status, rd_data);
    ...
  endtask
endclass
class spi_bus_base_seq extends uvm_reg_sequence;

`uvm_object_utils(spi_bus_base_seq)

spi_rm model;
// SPI env config object (contains register model handle)
spi_env_config m_cfg;

// Properties used by the register access methods:
rand uvm_reg_data_t data; // For passing data
uvm_status_e status; // Return status

task body;
    m_cfg = uvm_config_db #(spi_env_config)::get(null, get_full_name(),
        "spi_env_config", m_cfg)

    model = m_cfg.spi_rm;
endtask: body

endclass: spi_bus_base_base_seq
Register Stimulus

class div_load_seq extends spi_bus_base_seq;

`uvm_object_utils(div_load_seq)

// Interesting divisor values:
constraint div_values {data[15:0] inside {16'h0, 16'h1, 16'h2, 16'h4, 16'h8, 16'h10, 16'h20, 16'h40, 16'h80};}

task body;
  super.body;
  // Randomize the local data value
  assert(this.randomize());
  // Write to the divider register
  write_reg(model.divider_reg, status, data);
  assert(status == UVM_IS_OK);
endtask: body

endclass: div_load_seq
class data_load_seq extends spi_bus_base_seq;

`uvm_object_utils(data_load_seq)

uvm_reg data_regs[]; // Array of registers

task body;
    super.body;
    // Set up the data register handle array
    data_regs = '{spi_rm.rxtx0_reg, spi_rm.rxtx1_reg,
                  spi_rm.rxtx2_reg, spi_rm.rxtx3_reg};

    // Randomize order
    data_regs.shuffle();
    foreach(data_regs[i]) begin
        assert(data_regs[i].randomize());
        update_reg(data_regs[i], status, UVM_FRONTDOOR);
    end
endtask: body

data_load_seq
Stimulus Reuse (Bridge Example)

- SPI master is integrated inside an AHB peripheral block
- Host bus sequences can be reused as is
- Testbench structure changes
UVM Register Summary

- Register model follows hardware structure
  - Fields, Registers, Blocks, Maps
  - Internal access – get(), set() etc.
    - Sets up desired value
  - External access – Front and Backdoor

- Access layered via model
  - Generic sequences adapted to target bus sequences
  - Sequence reuse straight-forward

- Use the convenience API
  - Extend uvm_reg_sequence
  - write_reg()/read_reg() vs. write()/read()
  - Don’t have to worry about .parent() argument

- Built-in sequences for sanity testing
Thank you
UVM – What’s Now and What’s Next
UVM 1.2 Introduction
Uwe Simm, Cadence Design Systems
UVM 1.2 in numbers

- ~90 Mantis items addressed (60 bugs/clarifications, 30 enhancements)
- UVM core is today ~64kLOC (29kLOC doc, 35kLOC code) from UVM 1.1d (26kLOC, 30kLOC)
- Git says: UVM 1.1d -> UVM 1.2 is 12kLOC added and 4kLOC removed
- ~15% more and improved developer tests
What can you expect from UVM 1.2

- Bug fixes, performance fixes and cleanups
- Enhancements
- Few API and semantic changes
- User guide cleanup and update
- Largely backward compatible but maybe not 100% for your project

Note: The success of your verification project typically does NOT depend upon the latest features. Please consult with your tool, VIP, and training supplier to see how the new features best fit into your environment.
Notable changes: Reporting

- Now object based with ability to add values/objects. Can record messages to some other storage.

- All UVM core messages now routed through uvm messaging

- New addon message macros uvm_*_begin, uvm_*_end

```c
uvm_report_message msg = uvm_report_message::type_id::create("msg");
`uvm_info_begin("MY_ID2", "My info message", UVM_LOW, msg)
    `uvm_message_add_tag("my_color", "red") // add string
    `uvm_message_add_int(my_int, UVM_HEX)
    `uvm_message_add_object(object)
`uvm_info_end

uvm_text_tr_database db = new("my_msg_log.txt");
uvm_tr_stream stream = db.open_stream("my_stream");
uvm_recorder rec rec = stream.open_recorder("my_recorder");
msg.record(rec);
```

Notable changes: Sequences

- `<someseq>.set_automatic_phase_objection(arg)` automatically performs a raise/drop of the objection before/after the sequence execution

```plaintext
function my_sequence::new(string name="unnamed");
  super.new(name);
  set_automatic_phase_objection(1);
endfunction
```

- `+uvm_set_default_sequence=<seqr>,<phase>,<type>` allows you to start a sequence from the command line

```plaintext
+uvm_set_default_sequence=uvm_test_top.seqr,main_phase,my_seq
```

- starting_phase is now guarded via get/set_starting_phase

```plaintext
virtual task body();
  uvm_phase phase = get_starting_phase();
  phase.raise_objection(this);
  repeat(10) `uvm_do(req)
  phase.drop_objection(this);
endtask
```
Notable changes: Registers

- VHDL backdoor support
- Ability to control transaction order when register access result in multiple bus transactions

```verbatim
class high_addr_first extends uvm_reg_transaction_order_policy;
virtual function void order(ref uvm_reg_bus_op q[$]);
    q.rsort with (item.addr);
endfunction
...
endclass

high_addr_first p = new(“policy”);
someregmap.set_transaction_order_policy(p);
```
Notable changes: Objects

- `uvm_objects` do require a constructor now unless `UVM_OBJECT_DO_NOT_NEED_CONSTRUCTOR` is set.

```plaintext
class obj extends uvm_object;
  `uvm_object_utils(obj)
  function new(string name = "obj");
    super.new(name);
  endfunction
endclass
```

- Component names are being checked for compliance. This avoids bad names such as “ “, ”…..”, ”😊” or “a.b.c.d”

- `uvm_integral_t (64b packed logic) type support in API’s [un]pack_field_int`
Notable changes: Phasing

- Removed objections from non-task-imps (because objections do not make sense with function phases)

  - `uvm_phase.get_objection_count()` can be used to retrieve pending objections for the phase

- Phase transitions callbacks

```cpp
class my_cb extends uvm_phase_cb;
virtual function void phase_state_change(uvm_phase phase,
                                          uvm_phase_state_change change);
    uvm_phase_state state = change.get_state();
    `uvm_info("CALLBACK",
                $sformatf("Detected phase state change %s for phase %s", 
                            state.name(), phase.get_name()), UVM_LOW);
endfunction

uvm_callbacks#(uvm_phase,uvm_phase_cb)::add(phaseinst,my_cb_inst);
```
Notable changes: uvm_config_db

- Meta characters / regex in field names disabled due to performance and semantic issues.

```cpp
uvm_config_int::set(this, "","/z?mycomplexint/",4);
uvm_config_string::set(this, "","/mycomplexint/","xx");
uvm_config_int::set(this, "","/my*int/",2);
uvm_config_int::set(this, "","/my_complex.*/",3);
```

- set_config_, get_config_* now deprecated. Be careful when converting *_config_object with clone semantic (=default). Migration script handles this.

```cpp
// normally
uvm_config_{int,string,object}::{set,get}(ccntxt,...)

// careful with clone arg
set_config_object(...,obj[,1]) -> uvm_config_object::set(...,obj.clone())
```
Notable changes: factory

- **uvm_pkg::factory** has been removed. You can retrieve the factory via **uvm_factory::get()** instead.

- May undo a factory override

```plaintext
set_type_override_by_type(someT::get_type(), someT::get_type());
```

- Can replace factory in order to trace or log factory calls or build a dynamic factory

```plaintext
class uvm_to_factory extends uvm_delegate_factory;
    virtual function void set_inst_override_by_type (uvm_object_wrapper original_type,
        uvm_object_wrapper override_type, string full_inst_path);
    `uvm_info("FACTORY", "...", UVM_NONE)
    delegate.set_inst_override_by_type(original_type,override_type,full_inst_path);
endfunction
endclass
```

```plaintext
uvm_coreservice_t cs= uvm_coreservice_t::get(); uvm_to_factory f = new();
f.delegate=uvm_factory::get(); cs.set_factory(f);
```
Notable changes: Objections

- `set_propagate_mode()` can be used to avoid rippling of objections through hierarchy
  - Performance gain in high-frequency raise/drop scenarios
Notable changes: Misc

- Data access policy objects (dap) provide controlled access to embedded objects (get-to-lock, set-before-get,..)

- Visitor pattern infrastructure added (uvm_visitor, uvm_structure_proxy, uvm_visitor_adapter)

- uvm_coreservice_t common container for package scope variables with set/get accessors

```cpp
uvm_factory f = uvm_factory::get(); // same as
uvm_coreservice_t cs = uvm_coreservice_t::get();
uvm_factory f = cs.get_factory();
```

- uvm_sequence_library now documented

- uvm_severity_type (int) deprecated -> uvm_severity (enum)
Notable changes: Misc

- **uvm_enum_wrapper converter**
  - Convert from string via `uvm_enum_wrapper#(T)::from_name("STR", ref T o)`
  - `uvm_field_enum` now scans string type resources in addition to enum,int during auto config -- e.g., one can set enum fields via string literals from cmdline

- Messages in DPI-C now routed back to UVM message facilities

- Separation of classes into abstract API and "_default_" implementation for `uvm_factory, uvm_report_server`

  ```
  class my_server extends uvm_default_report_server;
  ```

- Cleanup of package scope variables (factory; missing UVM_ prefix – e.g., “UVM_”SEQ_ARB_RANDOM )
Migration from UVM 1.1 to UVM 1.2

- Backward incompatible changes outlined in migration document
- release-notes.txt does have list of addressed Mantis items with marker for backward compatibility
- Migration scripts provided
  - ./bin/add_uvm_object_new.pl adds uvm_object ctor if missing
  - ./bin/uvm11-to-uvm12.pl may help to do the simple changes around starting_phase, set/get_config, reporting
  - bin/ovm2uvm.pl - the old OVM -> UVM10 script
Thank you