UVM Register Modelling: Advanced Topics

Steve Holloway

Principal Verification Engineer

Dialog Semiconductor

...personal
...portable
...connected

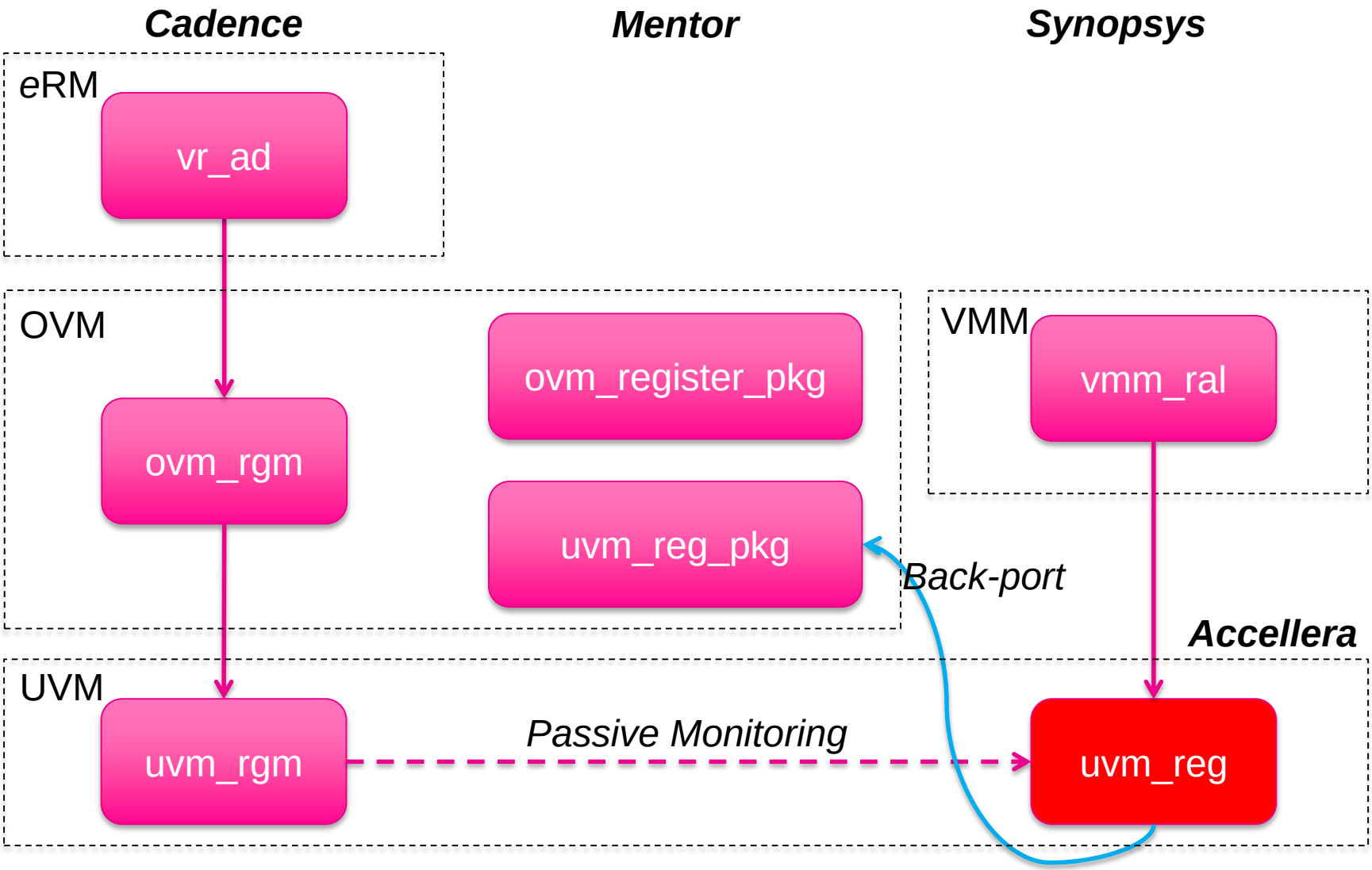
Agenda

- Introduction to the Register Layer
- Backdoor Access
- Using Multiple Address Maps
- Adding Bus Extensions
- Register Coverage
- Further Sources of Information

UVM Register Layer Features

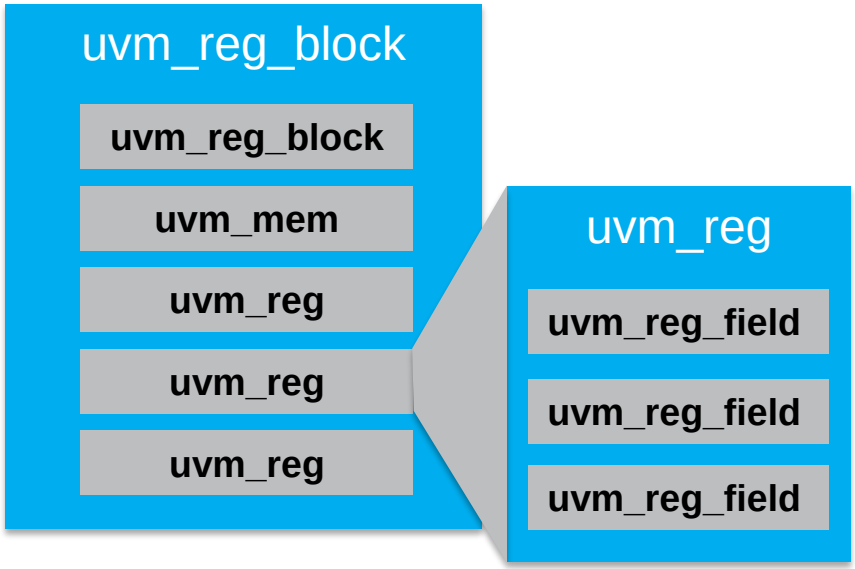
- A ***standard*** modeling approach
- A means to control, check and cover DUT registers
- A register / memory block hierarchy
  - Portable from Module – Chip level
- A register access API
  - Reusable with different bus agents
- Ability to model non-standard "quirky" registers
  - Callbacks
- Passive update of register model
  - Re-use in other (e.g. directed) environments
- It does **not** provide automated generation
  - Could be 1000s of registers in complex SoC designs
  - Use vendor-specific register generator tool

The uvm_reg Family Tree



UVM Register Layer

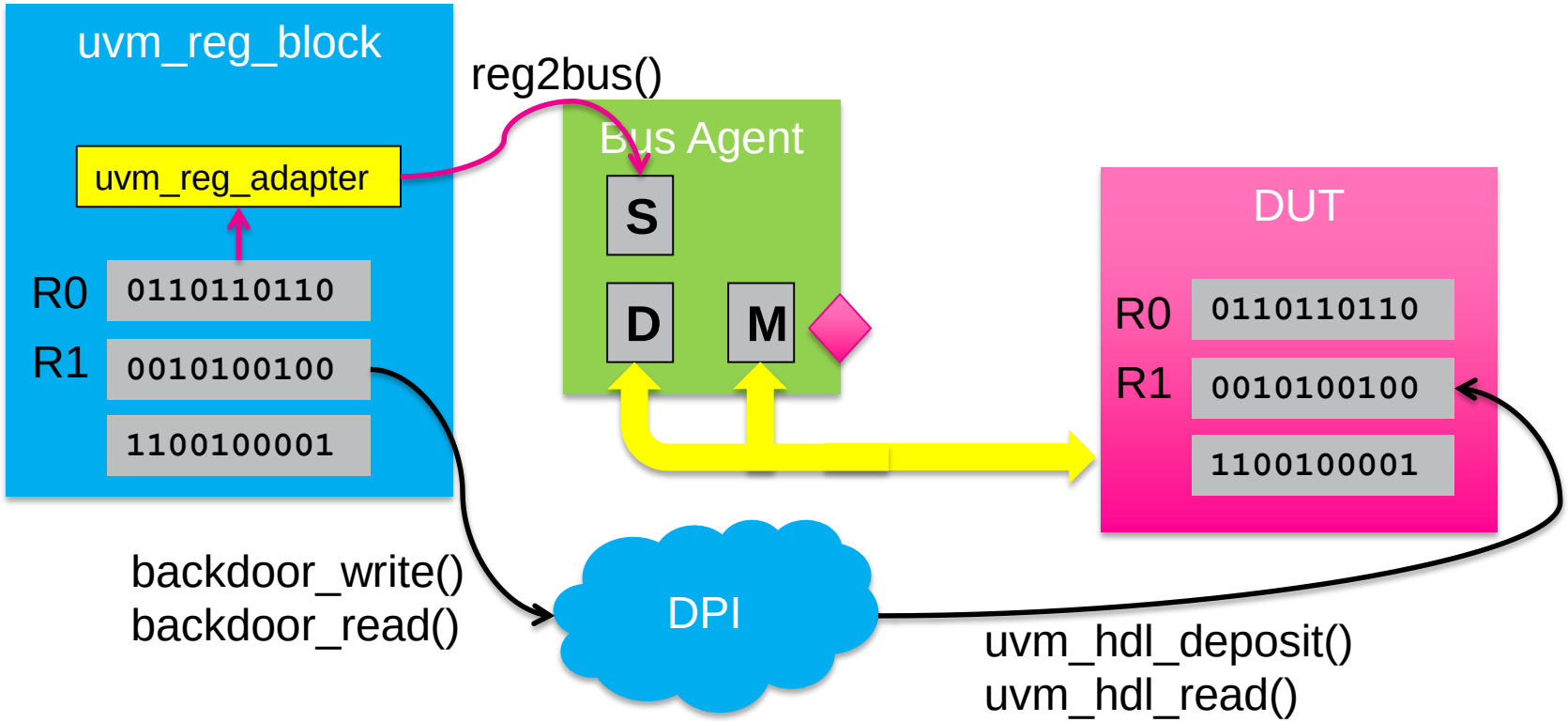
- Abstract model for registers and memories in DUT
  - Maintains a "mirror" of the DUT registers
- Hierarchy analogous to DUT:
  - Register Block
  - Register File
  - Memory
  - Register
  - Field
- Standardised register access API
  - Address-independent instance/string names
- Address maps
  - model access via a specific interface / bus master



Frontdoor or Backdoor Access?

user sequence

```
regblock.R0.write(8'ha5);
regblock.R1.write(8'h5a, .path(UVM_BACKDOOR));
. . .
```



Backdoor Access

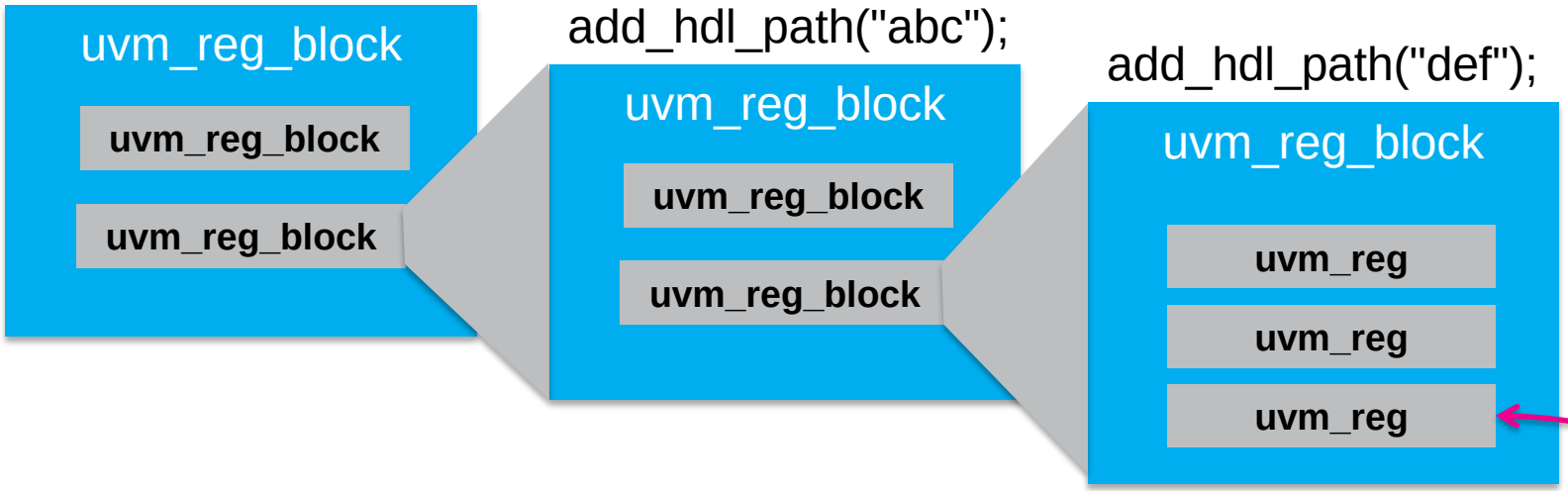
- Backdoor access is much faster than frontdoor – zero time!
  - Rapid DUT configuration possible
- Can uncover bugs hidden by the frontdoor access path
  - E.g. mangled memory addresses, data bus reversal
- Backdoor requires an hdl path to the register

```
class my_regmodel extends uvm_reg_block;

  rand reg_R0 R0;

  virtual function void build();
    R0 = reg_R0::type_id::create("R0");
    R0.configure(this, null, "reg_r0.q");
    R0.build();
  endfunction: build

endclass: my_regmodel
```

Relative to hierarchy of reg block

HDL Path Hierarchy

set_hdl_path_root("$root.tb.dut");



configure(…,"r0.q");

"$root.tb.dut.abc.def.r0.q"

```
class my_block_base_test extends uvm_test;

 function void end_of_elaboration_phase(uvm_phase phase);
  regmodel.set_hdl_path_root("$root.tb.u_dut.regs");
 endfunction

  ...

endclass : my_block_base_test
```

Modify in Chiplevel test class

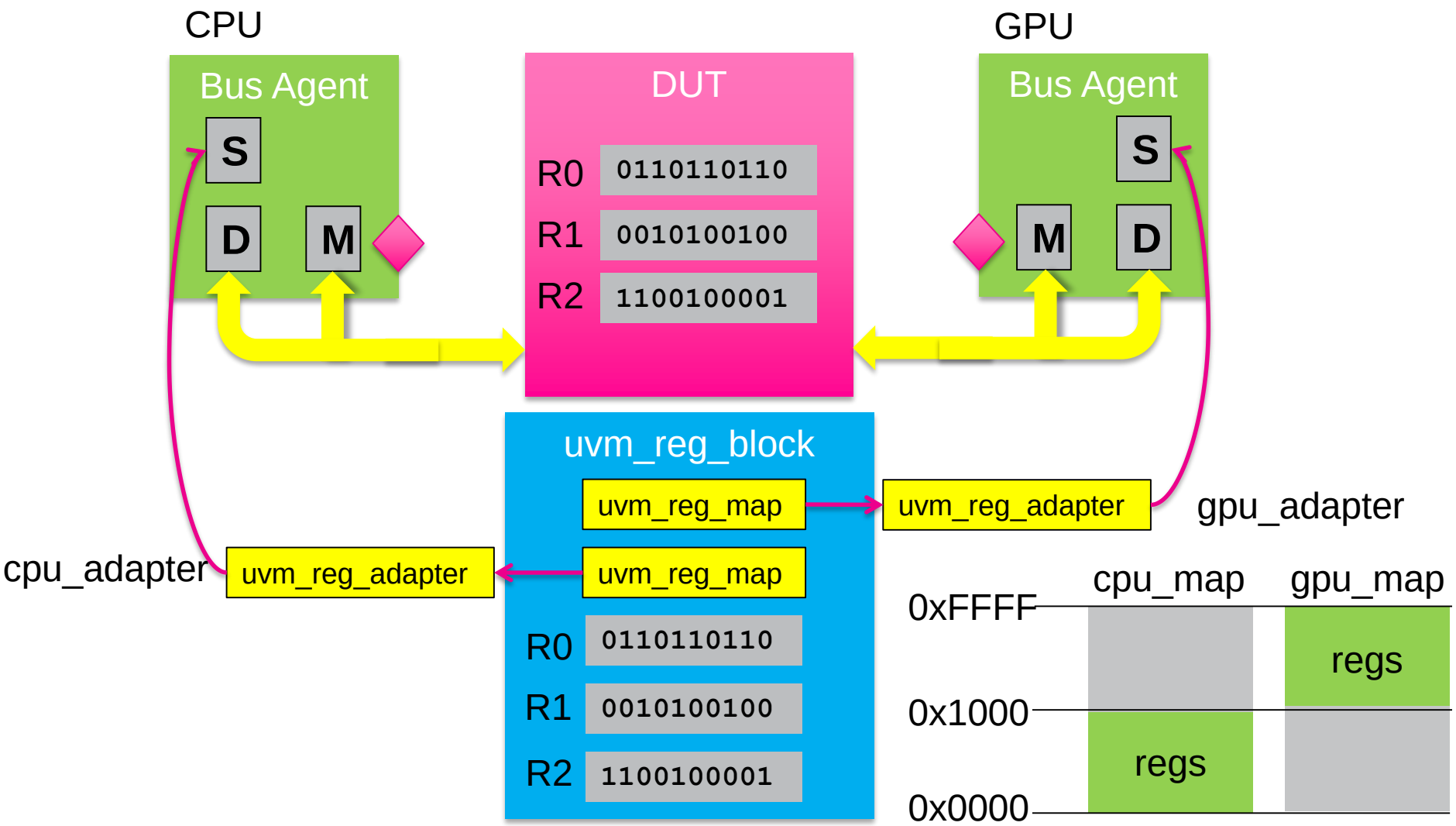
Useful Built-in Sequences for Backdoor Testing

- uvm_reg_mem_hdl_paths_seq
  - Check that the simulator can access all HDL paths
  - Run this first!

- uvm_reg_mem_generate_access_file_seq
  - Generate optimised HDL access file for backdoor paths
  - Pass into irun with –afile <access_file> option (Cadence only)

- uvm_reg_mem_access_seq
  - For all registers (with hdl paths):
    - Write frontdoor, read backdoor (and check)
    - Write backdoor, read frontdoor (and check)

Using Multiple Address Maps



Setting up the Maps

```
class my_regmodel extends uvm_reg_block;
  ...
  uvm_reg_map cpu_map;
  uvm_reg_map gpu_map;
  ...
  cpu_map = create_map("cpu_map", 'h0, 2, UVM_LITTLE_ENDIAN);
  cpu_map.add_reg(R0, 'h0, "RW"); // real address: 'h0
  cpu_map.add_reg(R1, 'h2, "RW"); // real address: 'h2
  cpu_map.add_reg(R2, 'h4, "RW"); // real address: 'h4

  gpu_map = create_map("gpu_map", 'h1000, 2, UVM_LITTLE_ENDIAN);
  gpu_map.add_reg(R0, 'h0, "RW"); // real address: 'h1000
  gpu_map.add_reg(R1, 'h2, "RO"); // real address: 'h1002
  gpu_map.add_reg(R2, 'h4, "RW"); // real address: 'h1004

  set_default_map(cpu_map); // map to use if none specified

endclass: my_regmodel
```

```
regblock.R0.write(status,8'ha5,.map(cpu_map),.parent(this));
regblock.R1.read(status,data,.map(gpu_map),.parent(this));
. . .
```

Hookup in the Env

```
function void my_env::build_phase(uvm_phase phase);
  ...
  regmodel = my_regmodel::type_id::create("regmodel");
  reg2apb  = reg2apb_adapter::type_id::create("reg2apb");
  reg2ahb  = reg2ahb_adapter::type_id::create("reg2ahb");

endfunction: build_phase
```

1 adapter per bus type

```
function void my_env::connect_phase(uvm_phase phase);
  ...
  regmodel.cpu_map.set_sequencer(apb_agent.sequencer, reg2apb);
  regmodel.gpu_map.set_sequencer(ahb_agent.sequencer, reg2ahb);

  cpu_predictor.map = regmodel.cpu_map;
  cpu_predictor.adapter = reg2apb;
  apb_agent.monitor.ap(cpu_predictor.bus_in);

  gpu_predictor.map = regmodel.gpu_map;
  gpu_predictor.adapter = reg2ahb;
  ahb_agent.monitor.ap(gpu_predictor.bus_in);

endfunction: connect_phase
```

Changing Register Behaviour According to Map

- Model a register which cannot be modified by the GPU
  - Callback hook methods get the map used

```
class my_reg_field_cbs extends uvm_reg_cbs;

  virtual function void post_predict(input uvm_reg_field  fld,
                                     input uvm_reg_data_t previous,
                                     inout uvm_reg_data_t value,
                                     input uvm_predict_e  kind,
                                     input uvm_path_e     path,
                                     input uvm_reg_map    map);

    if (map.get_name() == "gpu_map")
      if (kind == UVM_PREDICT_WRITE)
        value = previous;

  endfunction

endclass: my_reg_field_cbs
```

Extension Information

- Register Access API is relatively simple
  - Read/write arguments: *status, value, path, map, prior*

priority passed to sequencer

- We may need additional information for a bus transfer
  - E.g. protected access, locking, bursts
- Additional argument *extension* of type *uvm_object*

```
ahb_info_c ahb_info = new();
ahb_info.privileged = 1;
. . .
R0.write(status, data, .parent(this), .extension(ahb_info));
```

Getting Extension Information in the Adapter

- Adapter needs to call get_item() to access extension info

can only be called here

```
virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);

  ahb_info_c ahb_info;
  uvm_reg_item item = get_item();
  ...
  bus_trans.addr = rw.addr;
  bus_trans.data = rw.data;

  if (item.extension != null) begin
    $cast(ahb_info, item.extension);
    bus_trans.ahb_info  = ahb_info;
  end

  return bus_trans;

endfunction: reg2bus
```

extension is a uvm_object

Coverage Considerations

- Auto-generated coverage model form 3rd party tool
  - Covers field/register/map *access* only
- We may also need to cover:
  - A register value when something interesting happens
  - A scoreboard variable when a register gets modified

```
task my_scoreboard::monitor_tx_request();

  uvm_reg_field fld;

  forever begin
      @(tx_request_ev);
      fld = regmodel.get_field_by_name("BUFFER_LEVEL");
      buffer_level = fld.get();
      buffer_status_cg.sample();
  end

endtask: monitor_tx_request
```

e.g. emitted by DUT monitor

regmodel handle in scoreboard

Sampling Coverage on Reg Access

```
class trigger_reg_field_cbs extends uvm_reg_cbs;
  event write_ev;
  . . .
  virtual function void post_predict(. . .);
    if (kind == UVM_PREDICT_WRITE) begin
      if (value != previous) begin
        -> write_ev;
      end
    end
  endfunction
endclass: trigger_reg_field_cbs
```

Add callback to field in regmodel

```
task my_scoreboard::monitor_timer();

  trigger_reg_field_cbs trigger_cb = new();
  uvm_reg_field_cb::add(regmodel.TIMER_START, trigger_cb);

  forever begin
    @(trigger_cb.write_ev);
    timer_status_cg.sample();
  end

endtask: monitor_timer
```

Further Sources of Information

- Accellera UVM World (Source code, User Guide, Forums)
  - http://www.accellera.org/community/uvm
- Cadence UVM Training Videos (YouTube)
  - http://www.youtube.com/user/CadenceDesign
- Mentor Verification Academy (UVM Cookbook, Training Videos)
  - http://verificationacademy.com
- A Practical Guide to Adopting the Universal Verification Methodology
  - Sharon Rosenberg, Kathleen Meade (Second Edition)

dialog SEMICONDUCTOR

The power to be...

...personal
...portable
...connected