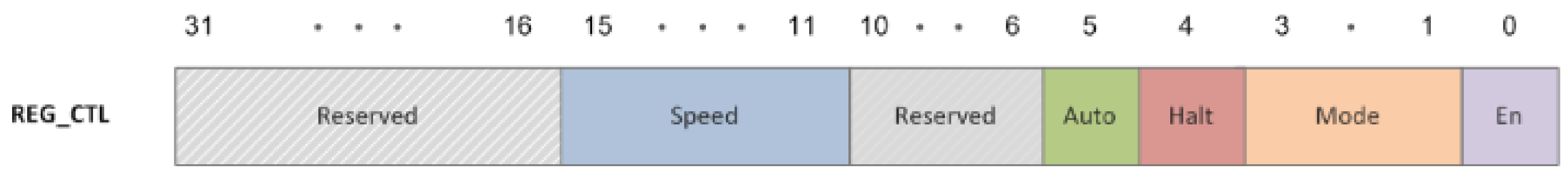
UVM Register Model

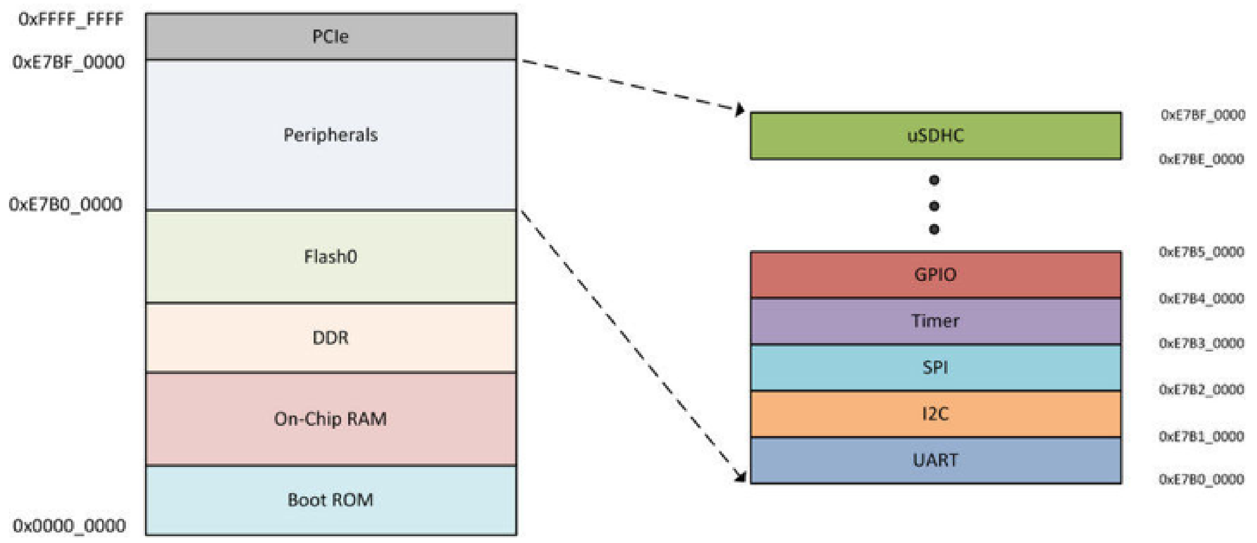
GitHub: samlyu

Register Structure

- Field: Part of a register, represents a particular feature.
- Register: Data storage, allows the hardware to behave in certain ways when programmed with certain values.

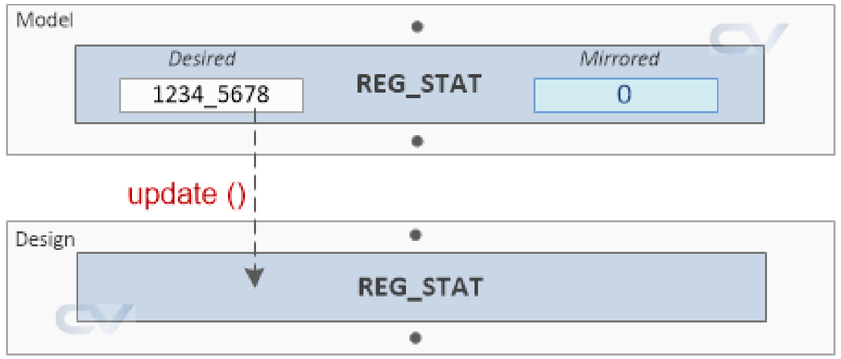


- Block: A collection of registers, each reg has different fields and configurations.
- Memory Map: A table that defines address ranges for devices.

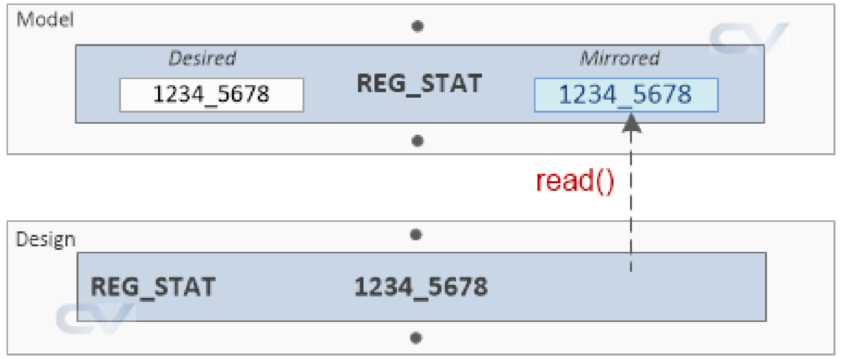


UVM Register Abstraction Layer

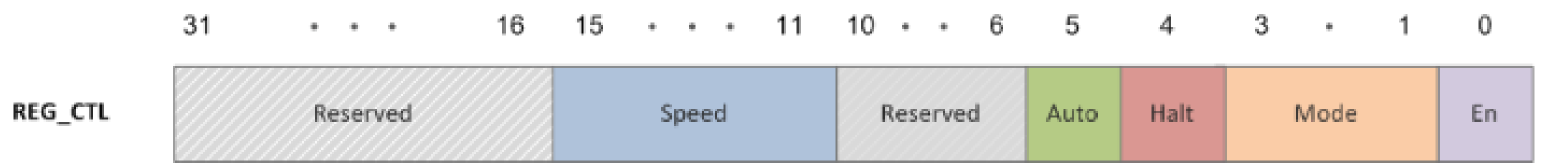
- Register Model: An entity that encompasses and describes the hierarchical structure of class objects for each register and its individual fields. **Every register in the model corresponds to an actual hardware register in the design.**
- Desired Value: The value to be updated later to the design from the reference model.



- Mirrored Value: Last known value in the reference model from the design.



UVM Reg Class

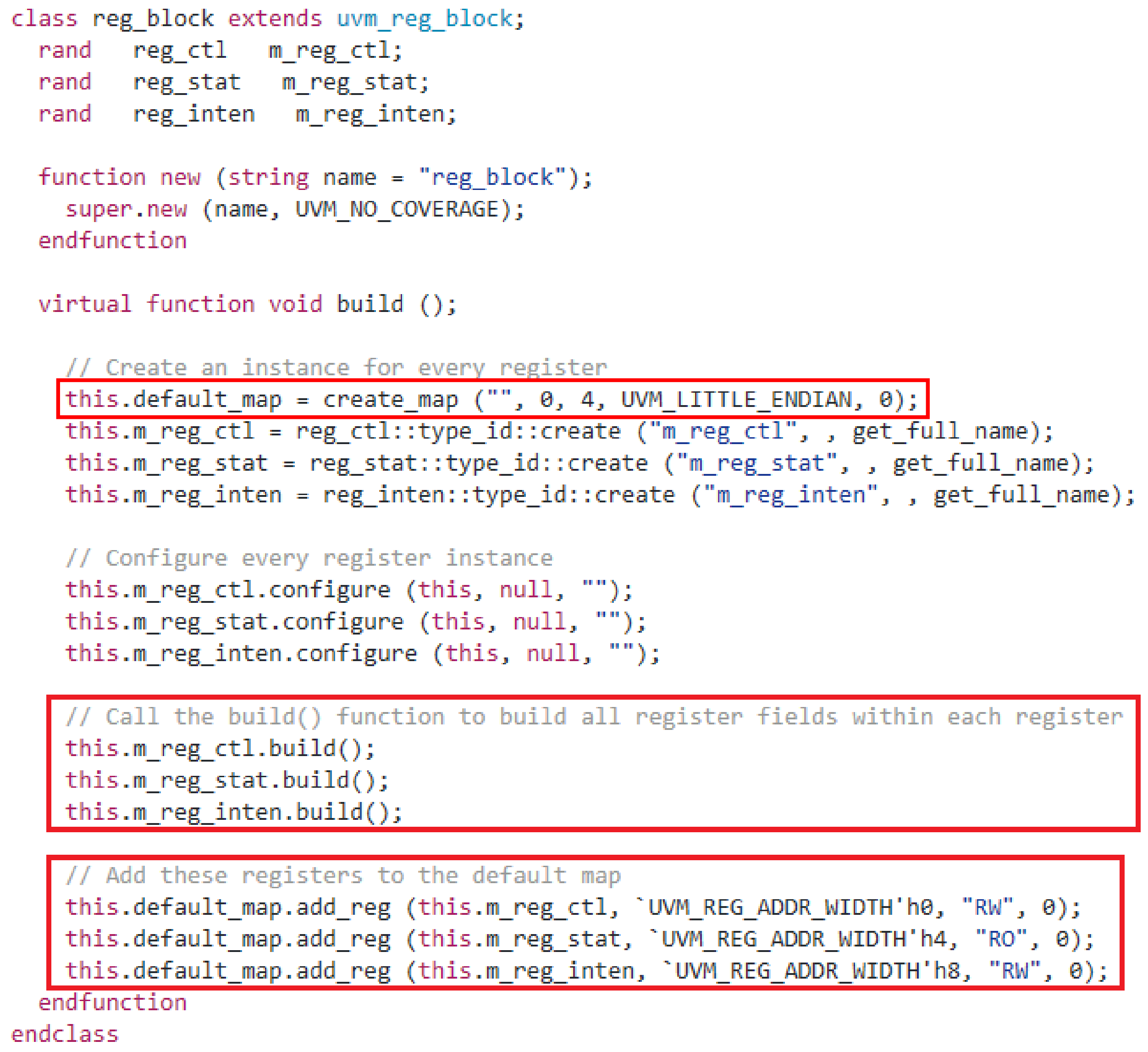


- Class extends from uvm_reg:
- New function for the reg:
- Build function for each field in the reg:

```
class reg_ctl extends uvm_reg;
  rand uvm_reg_field  En;
  rand uvm_reg_field  Mode;
  rand uvm_reg_field  Halt;
  rand uvm_reg_field  Auto;
  rand uvm_reg_field  Speed;

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

  virtual function void build ();

    // Create object instance for each field
    this.En     = uvm_reg_field::type_id::create ("En");
    this.Mode   = uvm_reg_field::type_id::create ("Mode");
    this.Halt   = uvm_reg_field::type_id::create ("Halt");
    this.Auto   = uvm_reg_field::type_id::create ("Auto");
    this.Speed  = uvm_reg_field::type_id::create ("Speed");

    // Configure each field
    this.En.configure (this, 1, 0, "RW", 0, 1'h0, 1, 1, 1);
    this.Mode.configure (this, 3, 1, "RW", 0, 3'h2, 1, 1, 1);
    this.Halt.configure (this, 1, 4, "RW", 0, 1'h1, 1, 1, 1);
    this.Auto.configure (this, 1, 5, "RW", 0, 1'h0, 1, 1, 1);
    this.Speed.configure (this, 5, 11, "RW", 0, 5'h1c, 1, 1, 1);
  endfunction
endclass
```

UVM Reg Block

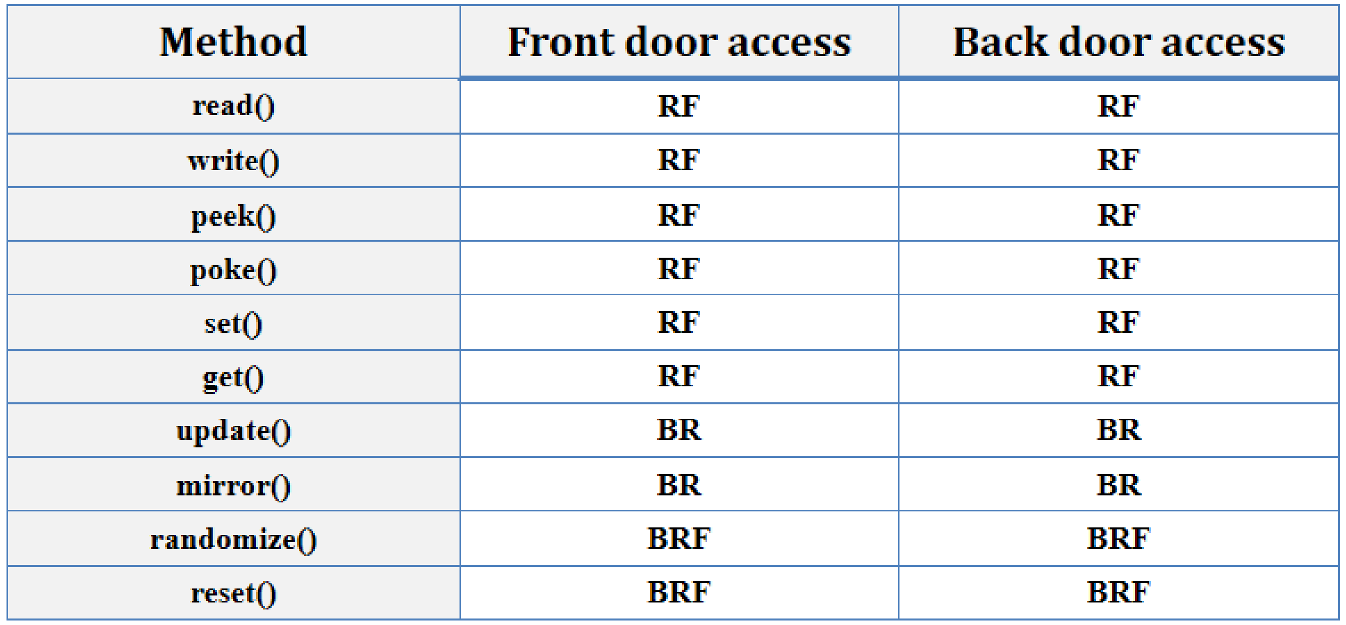
- Similar style to create a collection of UVM reg classes, need to call build() for each reg and add to map:



```
class reg_block extends uvm_reg_block;
  rand   reg_ctl    m_reg_ctl;
  rand   reg_stat    m_reg_stat;
  rand   reg_inten    m_reg_inten;

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

  virtual function void build ();

    // Create an instance for every register
    this.default_map = create_map ("", 0, 4, UVM_LITTLE_ENDIAN, 0);
    this.m_reg_ctl = reg_ctl::type_id::create ("m_reg_ctl", , get_full_name);
    this.m_reg_stat = reg_stat::type_id::create ("m_reg_stat", , get_full_name);
    this.m_reg_inten = reg_inten::type_id::create ("m_reg_inten", , get_full_name);

    // Configure every register instance
    this.m_reg_ctl.configure (this, null, "");
    this.m_reg_stat.configure (this, null, "");
    this.m_reg_inten.configure (this, null, "");

    // Call the build() function to build all register fields within each register
    this.m_reg_ctl.build();
    this.m_reg_stat.build();
    this.m_reg_inten.build();

    // Add these registers to the default map
    this.default_map.add_reg (this.m_reg_ctl, `UVM_REG_ADDR_WIDTH'h0, "RW", 0);
    this.default_map.add_reg (this.m_reg_stat, `UVM_REG_ADDR_WIDTH'h4, "RO", 0);
    this.default_map.add_reg (this.m_reg_inten, `UVM_REG_ADDR_WIDTH'h8, "RW", 0);
  endfunction
endclass
```

UVM Reg methods

- read() and write(): reads and writes values to DUT reg, frontdoor or backdoor
- get() and set(): reads and writes directly to the desired values
- peek() and poke(): reads and writes to DUT reg using backdoor access, regardless of the reg status
- update(): writes to DUT reg after set(), if the desired values and mirrored values are different
- mirror(): reads the updated DUT reg
- randomize()
- reset()



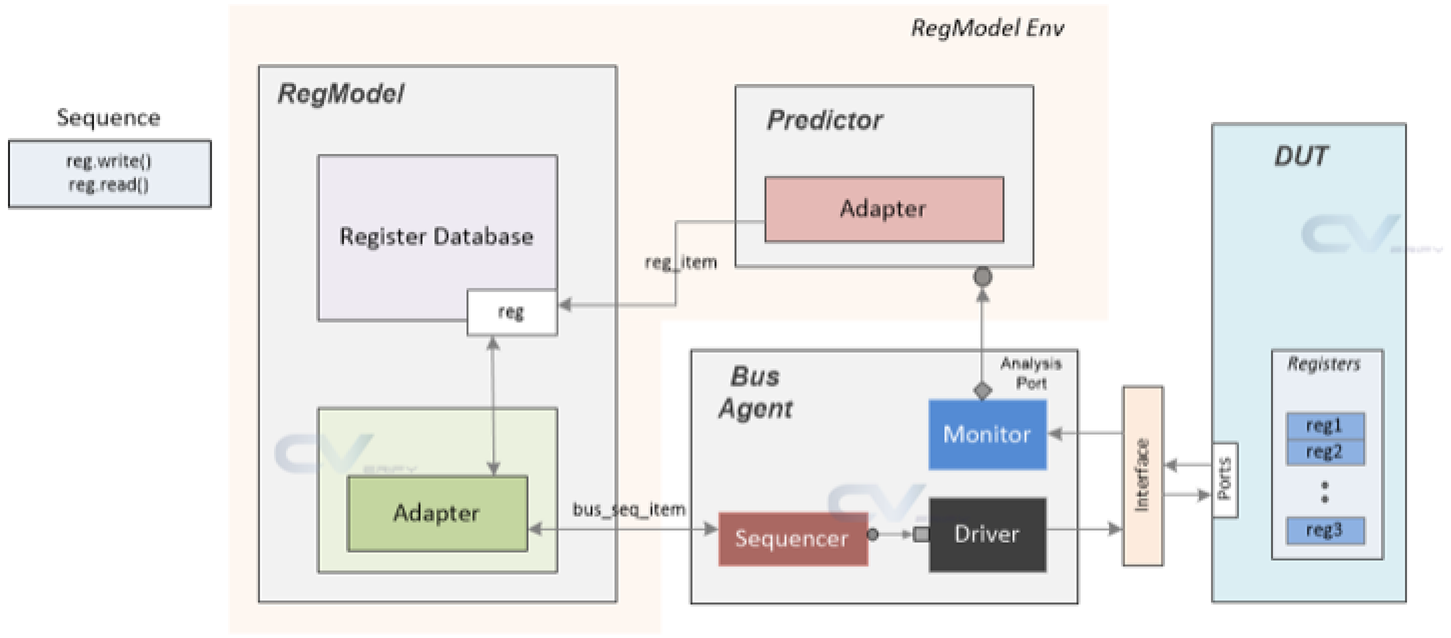
Method	Front door access	Back door access
read()	RF	RF
write()	RF	RF
peek()	RF	RF
poke()	RF	RF
set()	RF	RF
get()	RF	RF
update()	BR	BR
mirror()	BR	BR
randomize()	BRF	BRF
reset()	BRF	BRF

register access methods at a different level

* B - Block R - Register F - Field

Environment Setup

- Adapter: Reg R/W to bus transactions (vice versa) (between reg model and sequencer)
- Predictor: Bus transactions to reg R/W (monitor to reg model)



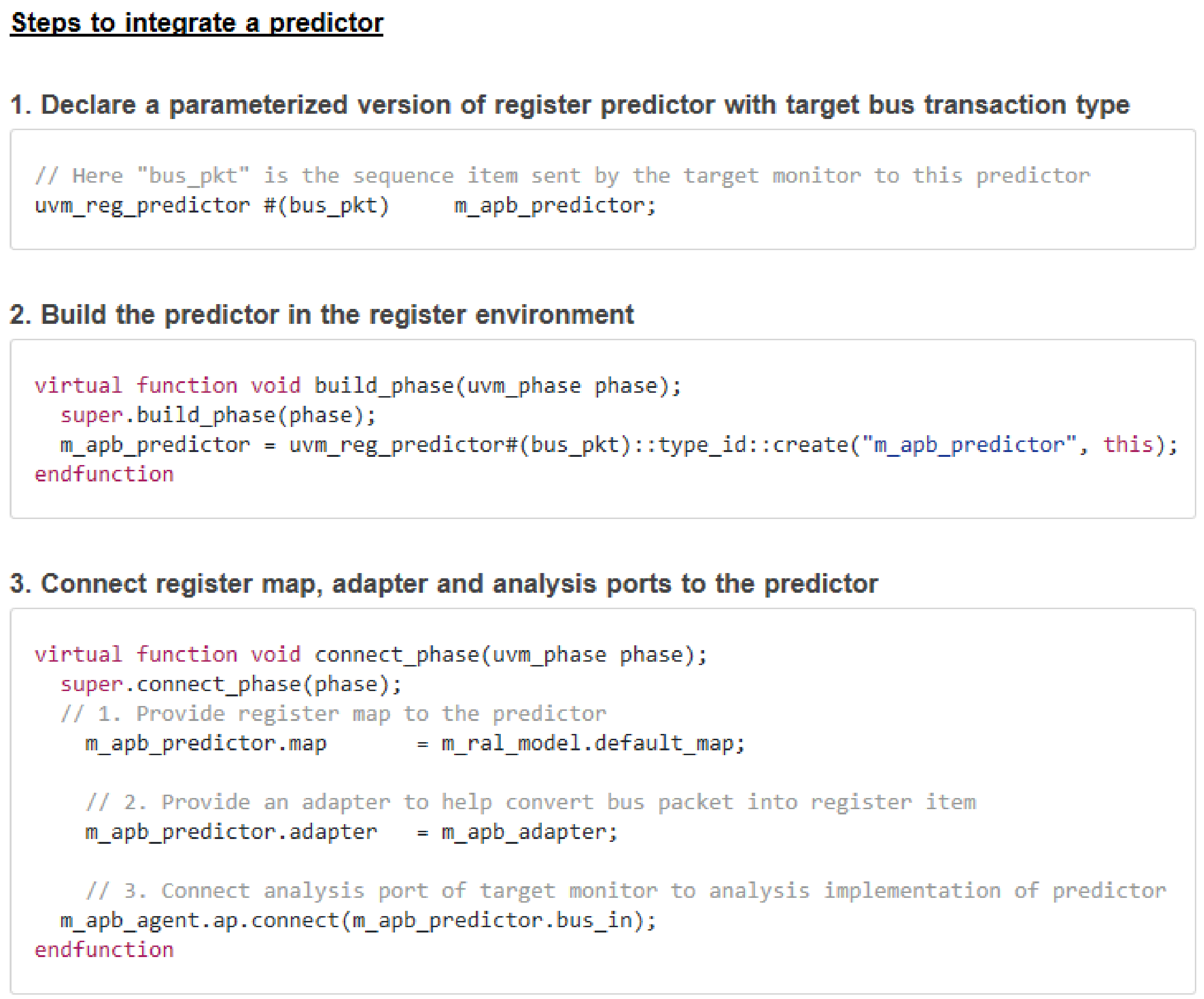
Adapter

- New function (enable byte or responses):
- Convert reg statements to bus transactions:
- Convert bus transactions to reg statements:

```
// apb_adapter is inherited from "uvm_reg_adapter"
class reg2apb_adapter extends uvm_reg_adapter;
   `uvm_object_utils (apb_adapter)

   // Set default values for the two variables based on bus protocol
   // APB does not support either, so both are turned off
   function new(string name="apb_adapter");
      super.new(name);
      supports_byte_enable = 0;
      provides_responses = 0;
   endfunction
  // This function accepts a register item of type "uvm_reg_bus_op" and assigns
  // address, data and other required fields to the bus protocol sequence_item
   virtual function uvm_sequence_item reg2bus (const ref uvm_reg_bus_op rw);
      bus_pkt pkt = bus_pkt::type_id::create ("pkt");
      pkt.write = (rw.kind == UVM_WRITE) ? 1: 0;
      pkt.addr  = rw.addr;
      pkt.data  = rw.data;
      return pkt;
   endfunction

    // This function accepts a bus sequence_item and assigns address/data fields to
    // the register item
     virtual function void bus2reg (uvm_sequence_item bus_item, ref uvm_reg_bus_op rw);
        bus_pkt pkt;

        // bus_item is a base class handle of type "uvm_sequence_item" and hence does not
        // contain addr, data properties in it. Hence bus_item has to be cast into bus_pkt
        if (! $cast (pkt, bus_item)) begin
           `uvm_fatal ("reg2apb_adapter", "Failed to cast bus_item to pkt")
        end

        rw.kind = pkt.write ? UVM_WRITE : UVM_READ;
        rw.addr = pkt.addr;
        rw.data = pkt.data;
        rw.status = UVM_IS_OK;  // APB does not support slave response
     endfunction
  endclass
```

Predictor

- Ways to update the reference model in sync with the DUT value:
  - read() the value from sequencer to adapter
  - Use predictor to get value from monitor



Steps to integrate a predictor

1. Declare a parameterized version of register predictor with target bus transaction type

```
// Here "bus_pkt" is the sequence item sent by the target monitor to this predictor
uvm_reg_predictor #(bus_pkt)     m_apb_predictor;
```

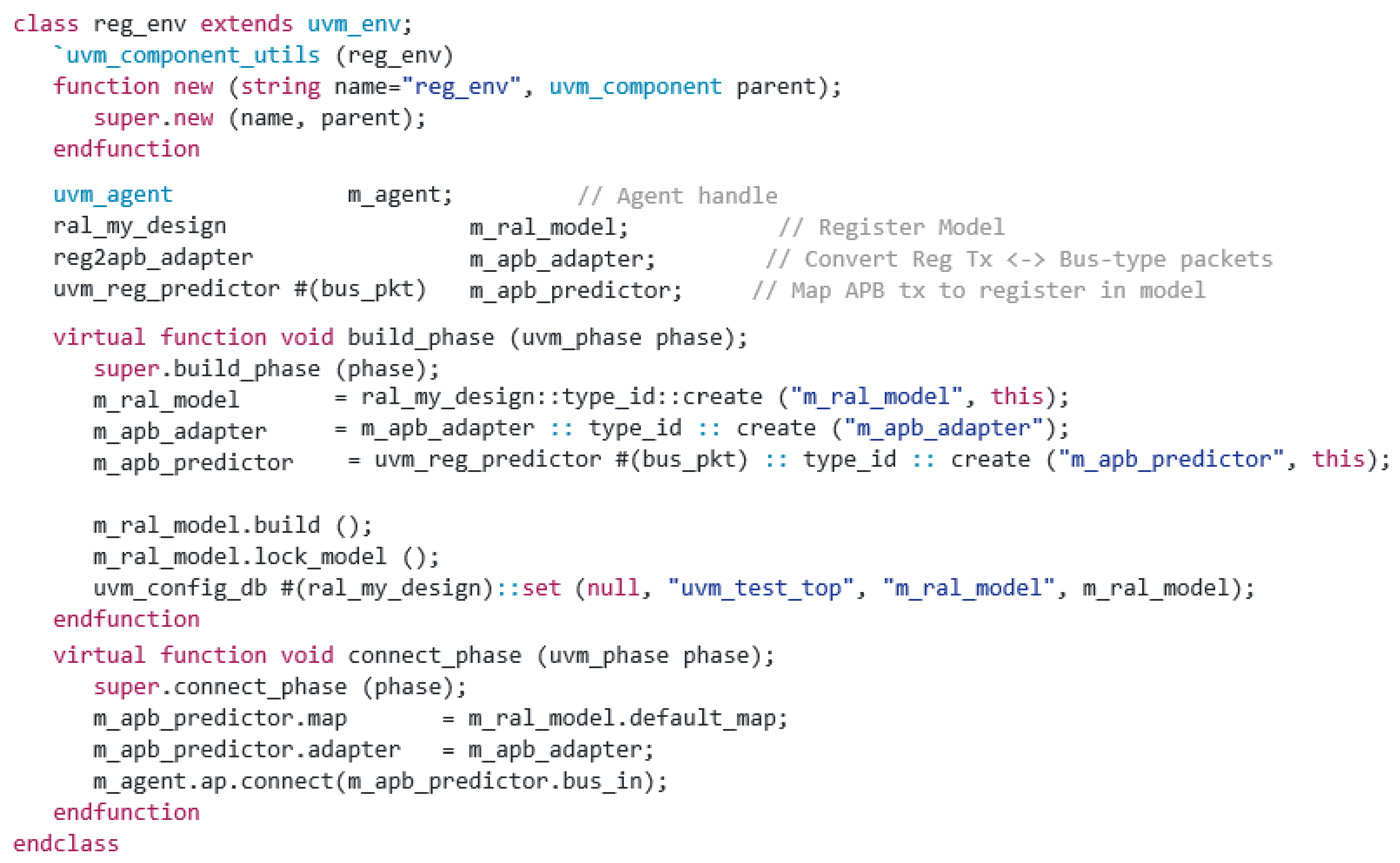
2. Build the predictor in the register environment

```
virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  m_apb_predictor = uvm_reg_predictor#(bus_pkt)::type_id::create("m_apb_predictor", this);
endfunction
```

3. Connect register map, adapter and analysis ports to the predictor

```
virtual function void connect_phase(uvm_phase phase);
  super.connect_phase(phase);
  // 1. Provide register map to the predictor
    m_apb_predictor.map       = m_ral_model.default_map;

    // 2. Provide an adapter to help convert bus packet into register item
    m_apb_predictor.adapter   = m_apb_adapter;

    // 3. Connect analysis port of target monitor to analysis implementation of predictor
  m_apb_agent.ap.connect(m_apb_predictor.bus_in);
endfunction
```

Reg_env Integeration



```
class reg_env extends uvm_env;
   `uvm_component_utils (reg_env)
   function new (string name="reg_env", uvm_component parent);
      super.new (name, parent);
   endfunction

   uvm_agent                m_agent;         // Agent handle
   ral_my_design                     m_ral_model;           // Register Model
   reg2apb_adapter                   m_apb_adapter;        // Convert Reg Tx <-> Bus-type packets
   uvm_reg_predictor #(bus_pkt)    m_apb_predictor;     // Map APB tx to register in model

   virtual function void build_phase (uvm_phase phase);
      super.build_phase (phase);
      m_ral_model       = ral_my_design::type_id::create ("m_ral_model", this);
      m_apb_adapter     = m_apb_adapter :: type_id :: create ("m_apb_adapter");
      m_apb_predictor    = uvm_reg_predictor #(bus_pkt) :: type_id :: create ("m_apb_predictor", this);

      m_ral_model.build ();
      m_ral_model.lock_model ();
      uvm_config_db #(ral_my_design)::set (null, "uvm_test_top", "m_ral_model", m_ral_model);
   endfunction

   virtual function void connect_phase (uvm_phase phase);
      super.connect_phase (phase);
      m_apb_predictor.map       = m_ral_model.default_map;
      m_apb_predictor.adapter   = m_apb_adapter;
      m_agent.ap.connect(m_apb_predictor.bus_in);
   endfunction
endclass
```

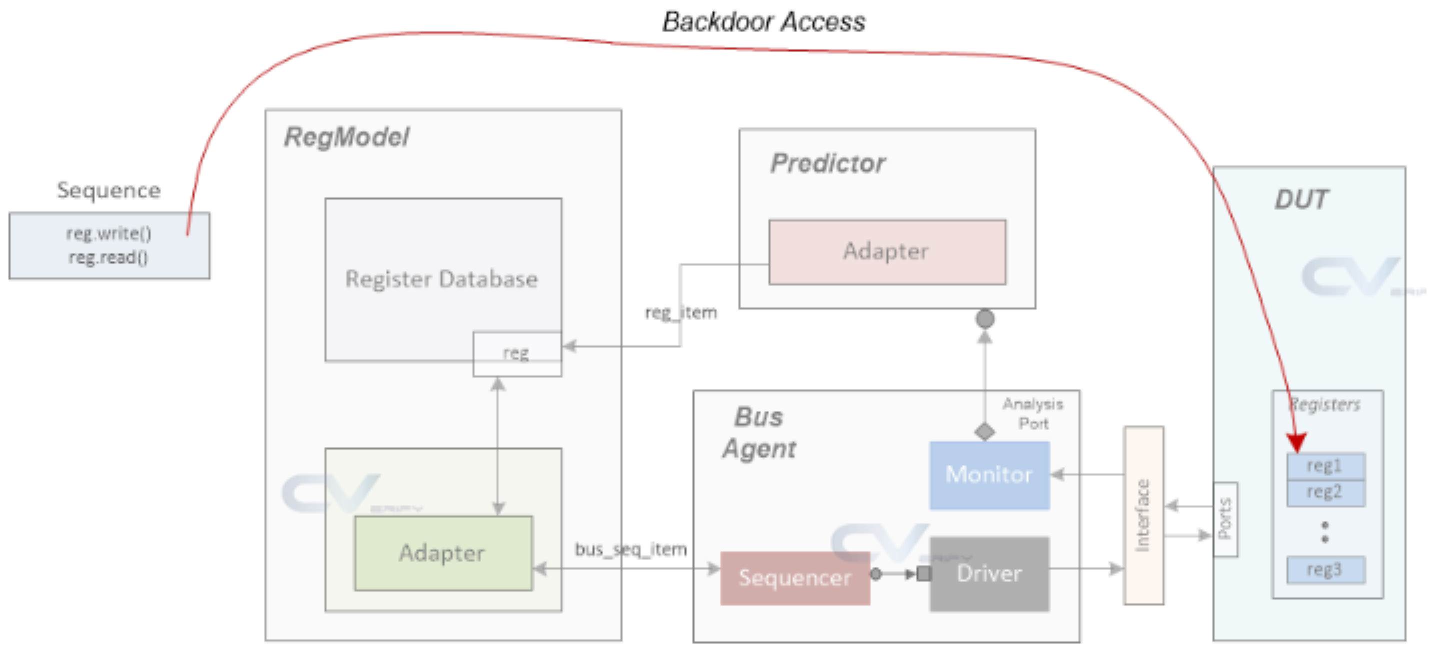
My_env Integration

- Define my_agent for driving transactions:

```
class my_env extends uvm_env;
   `uvm_component_utils (my_env)

   my_agent       m_agent;
   reg_env        m_reg_env;

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

   virtual function void build_phase (uvm_phase phase);
      super.build_phase (phase);
      m_agent = my_agent::type_id::create ("m_agent", this);
      m_reg_env = reg_env::type_id::create ("m_reg_env", this);
   endfunction

   virtual function void connect_phase (uvm_phase phase);
      super.connect_phase (phase);
      m_agent.m_mon.mon_ap.connect (m_reg_env.m_apb2reg_predictor.bus_in);
      m_reg_env.m_ral_model.default_map.set_sequencer (m_agent.m_seqr, m_reg_env.m_reg2apb);
   endfunction

endclass
```

Backdoor Access

- Directly access signals within the DUT (without bus control)
- Zero simulation time, no bus transactions, not recommended



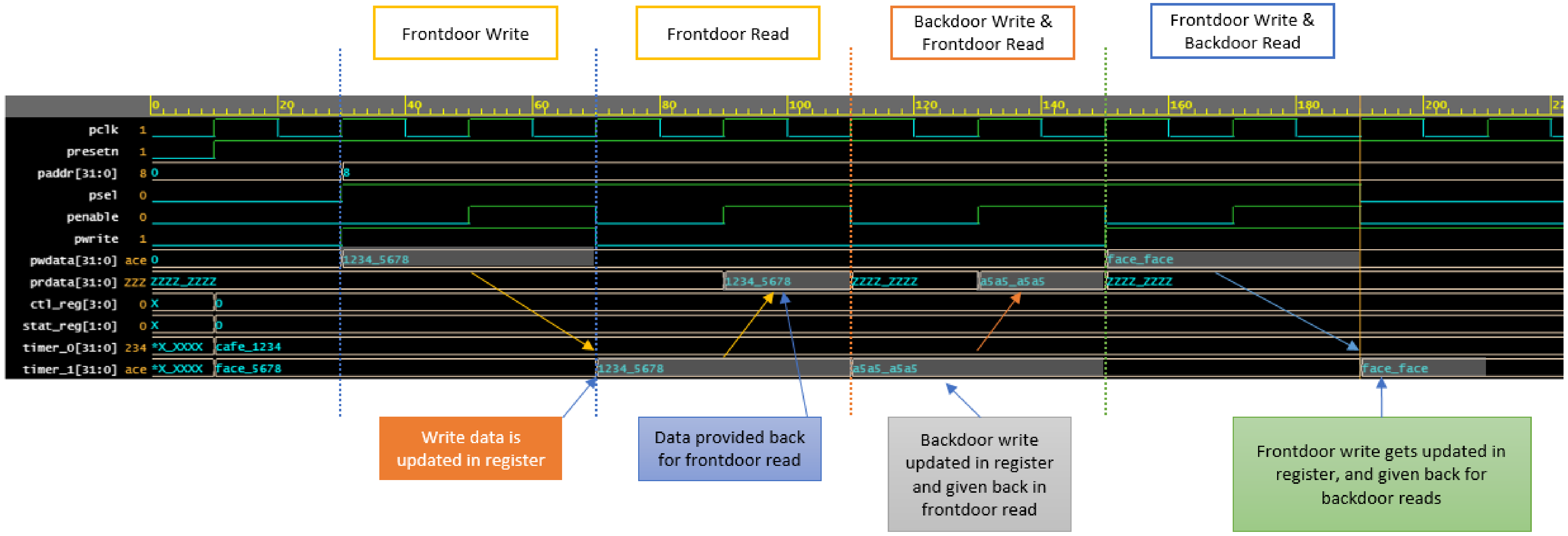
Backdoor Access Setup

```
// Assume that ral_cfg_ctl and other "uvm_reg" classes are already
// defined as in the frontdoor example

class ral_block_traffic_cfg extends uvm_reg_block;
  rand ral_cfg_ctl    ctrl;       // RW
  rand ral_cfg_timer  timer[2];   // RW
       ral_cfg_stat   stat;       // RO

  `uvm_object_utils(ral_block_traffic_cfg)

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

  virtual function void build();
    default_map = create_map("", 0, 4, UVM_LITTLE_ENDIAN, 0);
    stat = ral_cfg_stat::type_id::create("stat",,get_full_name());
    stat.configure(this, null, "");
    stat.build();

    // HDL path from DUT to the status register will now be
    // "tb.DUT.stat_reg" after the previous hierarchies are used
    // for path concatenation
    stat.add_hdl_path_slice("stat_reg", 0, stat.get_n_bits());
    default_map.add_reg(this.stat, `UVM_REG_ADDR_WIDTH'hc, "RO", 0);
    add_hdl_path("DUT");
  endfunction
endclass
```

```
class ral_sys_traffic extends uvm_reg_block;
  rand ral_block_traffic_cfg cfg;

  `uvm_object_utils(ral_sys_traffic)
  function new(string name = "traffic");
    super.new(name);
  endfunction

  function void build();
    default_map = create_map("", 0, 4, UVM_LITTLE_ENDIAN, 0);
    cfg = ral_block_traffic_cfg::type_id::create("cfg",,get_full_name());

    // Since registers exist at the DUT level in our design, configure
    // "cfg" class to have an HDL path called "DUT". So complete path to
    // DUT is now "tb.DUT"
    cfg.configure(this, "DUT");
    cfg.build();

    // Path to this top level regblock in our testbench environment is "tb"
    add_hdl_path("tb");
    default_map.add_submap(this.cfg.default_map, `UVM_REG_ADDR_WIDTH'h0);
  endfunction
endclass
```

Backdoor Access in Sequence

```
 // Perform a normal frontdoor access -> write some data first and then read it back
 m_ral_model.cfg.timer[1].write(status, 32'h1234_5678);
 m_ral_model.cfg.timer[1].read(status, rdata);
 // Perform a backdoor access for write and then do a frontdoor read
 m_ral_model.cfg.timer[1].write(status, 32'ha5a5_a5a5, UVM_BACKDOOR);
 m_ral_model.cfg.timer[1].read(status, rdata);
 // Perform a frontdoor write and then do a backdoor read
 m_ral_model.cfg.timer[1].write(status, 32'hface_face);
// Wait for a time unit so that backdoor access reads update value
#1;
 m_ral_model.cfg.timer[1].read(status, rdata, UVM_BACKDOOR);
```



Frontdoor Write
Frontdoor Read
Backdoor Write & Frontdoor Read
Frontdoor Write & Backdoor Read
pclk 1
presetn 1
paddr[31:0] 8
psel 0
penable 0
pwrite 1
pwdata[31:0] ace
prdata[31:0] zzz
ctl_reg[3:0] 0
stat_reg[1:0] 0
timer_0[31:0] 234
timer_1[31:0] ace
1234_5678
face_face
zzzz_zzzz
a5a5_a5a5
cafe_1234
face_5678
Write data is updated in register
Data provided back for frontdoor read
Backdoor write updated in register and given back in frontdoor read
Frontdoor write gets updated in register, and given back for backdoor reads

Reference

- https://www.chipverify.com/uvm/register-layer
- https://www.verificationguide.com/p/uvm-register-model.html