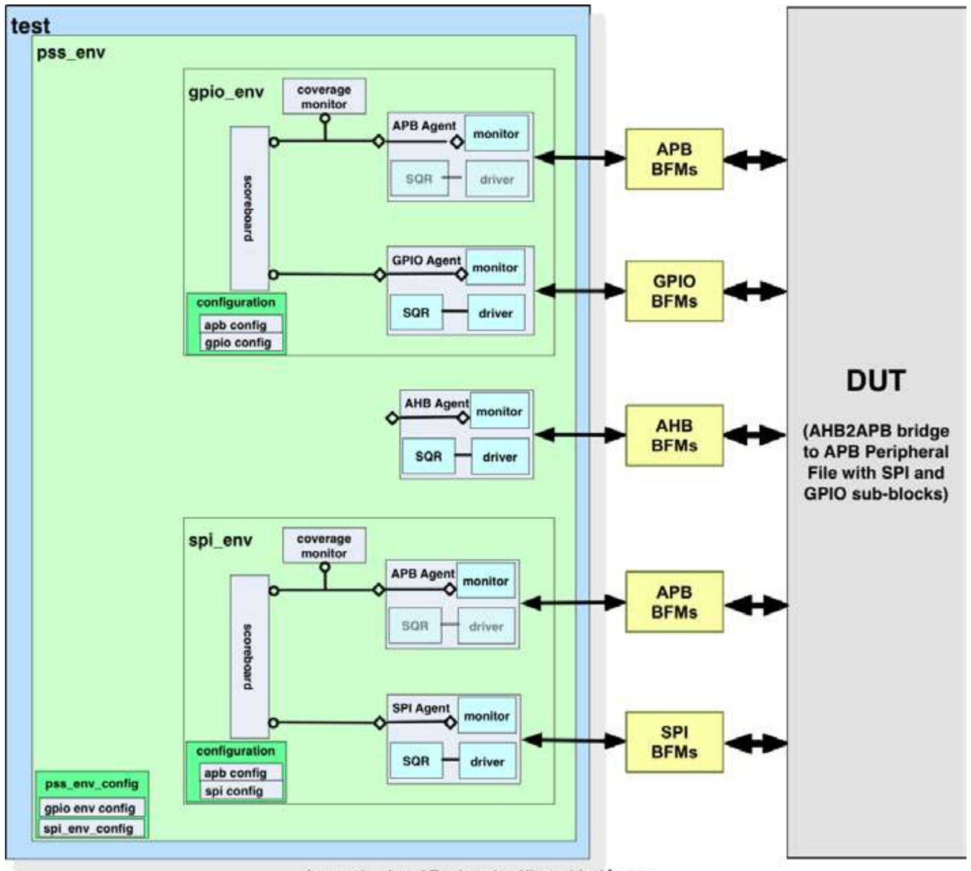
Integration-Level Testbench

This testbench example is one that takes two block level verification environments and shows how they can be reused at a higher level of integration. The principles that are illustrated in the example are applicable to repeated rounds of vertical reuse.

The example takes the SPI block level example and integrates it with another block level verification environment for a GPIO DUT. The hardware for the two blocks has been integrated into a Peripheral Sub-System (PSS) which uses an AHB to APB bus bridge to interface with the APB interfaces on the SPI and GPIO blocks. The environments from the block level are encapsulated by the pss_env, which also includes an AHB agent to drive the exposed AHB bus interface. In this configuration, the block level APB bus interfaces are no longer exposed, and so the APB agents are put into passive mode to monitor the APB traffic. The stimulus needs to drive the AHB interface and register layering enables reuse of block level stimulus at the integration level.



Integration Level Testbench – Hierarchical Layers

We shall now go through the testbench and the build process from the top down, starting with the two top level testbench modules.

Top Level Testbench Modules

As with the block level testbench example, two top level modules are utilized. The hdl_top instantiates the DUT, instantiates the BFM interfaces and connects the pin interfaces to the DUT and the BFM interfaces. Virtual Interface handles referencing the BFM interfaces are placed into the configuration space and clocks and resets are generated. The main differences between this code and the block level testbench code are that there are more interfaces and that there is a need to bind to some internal signals to monitor the APB bus. Another differences is that driver BFMs are not instantiated if the agent is going to be used in passive mode. The DUT is wrapped by a module which connects its I/O signals to the interfaces used in the UVM testbench. The internal signals are bound to the APB interface using the binder module:

```
module top_tb;

import uvm_pkg::*;
import pss_test_lib_pkg::*;

// PCLK and PRESETn
//
logic HCLK;
logic HRESETn;

//
// Instantiate the pin interfaces:
//
apb_if APB(HCLK, HRESETn);
// APB interface - shared between passive
agents
ahb_if AHB(HCLK, HRESETn);
// AHB interface
spi_if SPI();
// SPI Interface
...
// Additional pin interfaces

//
// Instantiate the BFM interfaces:
//
apb_monitor_bfm APB_SPI_mon_bfm(

    .PCLK    (APB.PCLK),
    .PRESETn (APB.PRESETn),

    .PADDR    (APB.PADDR),

    .PRDATA   (APB.PRDATA),

    .PWDATA   (APB.PWDATA),

    .PSEL     (APB.PSEL),

    .PENABLE (APB.PENABLE),

    .PWRITE   (APB.PWRITE),

    .PREADY   (APB.PREADY)

 );

 apb_monitor_bfm APB_GPIO_mon_bfm(

    .PCLK    (APB.PCLK),
    .PRESETn (APB.PRESETn),

    .PADDR   (APB.PADDR),
    .PRDATA  (APB.PRDATA),
          .PWDATA   (APB.PWDATA),
          .PSEL     (APB.PSEL),
          .PENABLE  (APB.PENABLE),
          .PWRITE   (APB.PWRITE),
          .PREADY   (APB.PREADY)
      );
        apb_driver_bfm APB_GPIO_drv_bfm(
          .PCLK     (APB_dummy.PCLK),
          .PRESETn (APB_dummy.PRESETn),
          .PADDR     (APB_dummy.PADDR),
          .PRDATA    (APB_dummy.PRDATA),
          .PWDATA    (APB_dummy.PWDATA),
          .PSEL      (APB_dummy.PSEL),
          .PENABLE   (APB_dummy.PENABLE),
          .PWRITE    (APB_dummy.PWRITE),
          .PREADY    (APB_dummy.PREADY)
      );
      ...
      // Additional BFM interfaces
      // Binder
      binder probe();

      // DUT Wrapper:
      pss_wrapper wrapper(.ahb(AHB),
                          .spi(SPI),
                          .gpi(GPI),
                          .gpo(GPO),
                          .gpoe(GPOE),
                          .icpit(ICPIT),
                          .uart_rx(UART_RX),
                          .uart_tx(UART_TX),
                          .modem(MODEM));
      // UVM initial block:
      // Virtual interface wrapping
      initial begin
         import uvm_pkg::uvm_config_db;
uvm_config_db  #(virtual apb_monitor_bfm)     ::set(null, "uvm_test_top", "APB_SPI_mon_bfm", APB_SPI_mon_bfm);
uvm_config_db  #(virtual apb_monitor_bfm)     ::set((null, "uvm_test_top", "APB_GPIO_mon_bfm", APB_GPIO_mon_bfm);
uvm_config_db #(virtual ahb_monitor_bfm) ::set(null, "uvm_test_top", "AHB_mon_bfm", AHB_mon_bfm);
uvm_config_db #(virtual ahb_driver_bfm)  ::set(null, "uvm_test_top", "AHB_drv_bfm", AHB_drv_bfm);
uvm_config_db #(virtual spi_monitor_bfm) ::set(null, "uvm_test_top", "SPI_mon_bfm", SPI_mon_bfm);
uvm_config_db #(virtual spi_driver_bfm)  ::set(null, "uvm_test_top", "SPI_drv_bfm", SPI_drv_bfm);
...
//Additional uvm_config_db::set() calls
end
```

```
 //
 // Clock and reset initial block:
 //
 initial begin
   HCLK = 1;
   forever #10ns HCLK = ~HCLK;
 end
 initial begin
   HRESETn = 0;
   repeat(4) @(posedge HCLK);
   HRESETn = 1;
end

// Clock assignments:
assign GPO.clk = HCLK;
assign GPOE.clk = HCLK;
assign GPI.clk = HCLK;

endmodule: hdl_top
```

The hvl_top module remains largely the same as in the block level example. It now imports the pss_test_lib_pkg so that the definition of the tests are known. Otherwise, the same functionality is present.

```
module hvl_top;

import uvm_pkg::*;
import pss_test_lib_pkg::*;

// UVM initial block:
initial begin
  run_test();
end

endmodule: hvl_top
```

The Test

Like the block level test, the integration level test should have the common build and configuration process captured in a base class that subsequent test cases can inherit from. As can be seen from the example, there is more configuration to do and so the need becomes more compelling.

The configuration object for the pss_env contains handles for the configuration objects for the spi_env and the gpio_env. In turn, the sub-env configuration objects contain handles for their agent sub-component configuration objects. The pss_env is responsible for un-nesting the spi_env and gpio_env configuration objects and setting them in its configuration table, making any local changes necessary. In turn, the spi_env and the gpio_env put their agent configurations into their configuration table.

The pss test base class is as follows:

```
//
// Class Description:
//
//
class pss_test_base extends uvm_test;

// UVM Factory Registration Macro
//
`uvm_component_utils(pss_test_base)

//------------------------------------------
// Data Members
//------------------------------------------

//------------------------------------------
// Component Members
//------------------------------------------
// The environment class
pss_env m_env;
// Configuration objects
pss_env_config m_env_cfg;
spi_env_config m_spi_env_cfg;
gpio_env_config m_gpio_env_cfg;
apb_agent_config m_spi_apb_agent_cfg;
apb_agent_config m_gpio_apb_agent_cfg;
ahb_agent_config m_ahb_agent_cfg;
spi_agent_config m_spi_agent_cfg;
...
// Additional configuration object handles

// Register map
pss_register_map pss_rm;

// Interrupt Utility
intr_util ICPIT;

//------------------------------------------
```

```
// Methods
//------------------------------------------
// Standard UVM Methods:
extern function new(string name = "spi_test_base", uvm_component parent = null);
extern function void build_phase( uvm_phase phase);
extern virtual function void configure_apb_agent(apb_agent_config cfg, int index,
logic[31:0] start_address, logic[31:0] range);

extern task run_phase( uvm_phase phase );

endclass: pss_test_base

function pss_test_base::new(string name = "spi_test_base", uvm_component parent = null);
  super.new(name, parent);
endfunction

// Build the env, create the env configuration
// including any sub configurations and assigning virtual interfaces
function void pss_test_base::build_phase(uvm_phase phase);
   virtual intr_bfm temp_intr_bfm;

   m_env_cfg = pss_env_config::type_id::create("m_env_cfg");

   // Register model
   // Enable all types of coverage available in the register model
   uvm_reg::include_coverage("*", UVM_CVR_ALL);
   // Register map - Keep reg_map a generic name for vertical reuse reasons
  pss_rb = pss_reg_block::type_id::create("pss_rb");
  pss_rb.build();
  m_env_cfg.pss_rb = pss_rb;

  // SPI Sub-env configuration:
  m_spi_env_cfg = spi_env_config::type_id::create("m_spi_env_cfg");
  m_spi_env_cfg.spi_rb = pss_rb.spi_rb;

  // apb agent in the SPI env:

  m_spi_apb_agent_cfg = apb_agent_config::type_id::create("m_spi_apb_agent_cfg");
  configure_apb_agent(m_spi_apb_agent_cfg, 0, 32'h0, 32'h18);
  if (!uvm_config_db #(virtual apb_monitor_bfm)::get(this, "", "APB_SPI_mon_bfm",
  m_spi_apb_agent_cfg.mon_bfm))
    `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface APB_SPI_mon_bfm from
  uvm_config_db. Have you set() it?")
```

```
// if (!uvm_config_db #(virtual apb_driver_bfm) ::get(this, "", "APB_SPI_drv_bfm",
m_spi_apb_agent_cfg.drv_bfm))
// `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface APB_SPI_drv_bfm from
uvm_config_db. Have you set() it?")
m_spi_apb_agent_cfg.active = UVM_PASSIVE;
m_spi_env_cfg.m_apb_agent_cfg = m_spi_apb_agent_cfg;

// SPI agent:
m_spi_agent_cfg = spi_agent_config::type_id::create("m_spi_agent_cfg");
if (!uvm_config_db #(virtual spi_monitor_bfm)::get(this, "", "SPI_mon_bfm",
m_spi_agent_cfg.mon_bfm))
  `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface SPI_mon_bfm from uvm_config_db.
Have you set() it?")
  if (!uvm_config_db #(virtual spi_driver_bfm) ::get(this, "", "SPI_drv_bfm",
m_spi_agent_cfg.drv_bfm))
  `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface SPI_drv_bfm from
uvm_config_db. Have you set() it?")
m_spi_env_cfg.m_spi_agent_cfg = m_spi_agent_cfg;
m_env_cfg.m_spi_env_cfg = m_spi_env_cfg;
uvm_config_db #(spi_env_config)::set(this, "*", "spi_env_config", m_spi_env_cfg);

// GPIO env configuration:
m_gpio_env_cfg = gpio_env_config::type_id::create("m_gpio_env_cfg");
m_gpio_env_cfg.gpio_rb = pss_rb.gpio_rb;
m_gpio_apb_agent_cfg = apb_agent_config::type_id::create("m_gpio_apb_agent_cfg");
configure_apb_agent(m_gpio_apb_agent_cfg, 1, 32'h100, 32'h124);
if (!uvm_config_db #(virtual apb_monitor_bfm)::get(this, "", "APB_GPIO_mon_bfm",
m_gpio_apb_agent_cfg.mon_bfm))
  `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface APB_GPIO_mon_bfm
from uvm_config_db. Have you set() it?")
// if (!uvm_config_db #(virtual apb_driver_bfm) ::get(this, "", "APB_GPIO_drv_bfm",
m_gpio_apb_agent_cfg.drv_bfm))
// `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface APB_drv_bfm from
uvm_config_db. Have you set() it?")
m_gpio_apb_agent_cfg.active = UVM_PASSIVE;
m_gpio_env_cfg.m_apb_agent_cfg = m_gpio_apb_agent_cfg;
m_gpio_env_cfg.has_functional_coverage = 1;
// Register coverage no longer valid

// GPO agent
m_GPO_agent_cfg = gpio_agent_config::type_id::create("m_GPO_agent_cfg");
if (!uvm_config_db #(virtual gpio_monitor_bfm)::get(this, "", "GPO_mon_bfm",
m_GPO_agent_cfg.mon_bfm))
```

```
  `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface GPO_mon_bfm from
uvm_config_db. Have you set() it?")
// if (!uvm_config_db #(virtual gpio_driver_bfm) ::get(this, "",
"GPO_drv_bfm", m_GPO_agent_cfg.drv_bfm))
// `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface SPI_drv_bfm from
uvm_config_db. Have you set() it?")
m_GPO_agent_cfg.active = UVM_PASSIVE;
// Only monitors
m_gpio_env_cfg.m_GPO_agent_cfg = m_GPO_agent_cfg;

// GPOE agent
m_GPOE_agent_cfg = gpio_agent_config::type_id::create("m_GPOE_agent_cfg");
if (!uvm_config_db #(virtual gpio_monitor_bfm)::get(this, "", "GPOE_mon_bfm",
m_GPOE_agent_cfg.mon_bfm))
  `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface GPOE_mon_bfm from
uvm_config_db. Have you set() it?")
// if (!uvm_config_db #(virtual gpio_driver_bfm) ::get(this, "",
"GPOE_drv_bfm", m_GPOE_agent_cfg.drv_bfm))
// `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface SPI_drv_bfm
from uvm_config_db. Have you set() it?")
m_GPOE_agent_cfg.active = UVM_PASSIVE;
// Only monitors
m_gpio_env_cfg.m_GPOE_agent_cfg = m_GPOE_agent_cfg;

// GPI agent - active (default)
m_GPI_agent_cfg = gpio_agent_config::type_id::create("m_GPI_agent_cfg");
if (!uvm_config_db #(virtual gpio_monitor_bfm)::get(this, "", "GPI_mon_bfm",
m_GPI_agent_cfg.mon_bfm))
  `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface GPI_mon_bfm from
uvm_config_db. Have you set() it?")
if (!uvm_config_db #(virtual gpio_driver_bfm) ::get(this, "", "GPI_drv_bfm",
m_GPI_agent_cfg.drv_bfm))
  `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface SPI_drv_bfm from
uvm_config_db. Have you set() it?")
m_gpio_env_cfg.m_GPI_agent_cfg = m_GPI_agent_cfg;

// GPIO Aux agent not present
m_gpio_env_cfg.has_AUX_agent = 0;
m_gpio_env_cfg.has_functional_coverage = 1;
m_gpio_env_cfg.has_out_scoreboard = 1;
m_gpio_env_cfg.has_in_scoreboard = 1;
m_env_cfg.m_gpio_env_cfg = m_gpio_env_cfg;
uvm_config_db #(gpio_env_config)::set(this, "*", "gpio_env_config", m_gpio_env_cfg);
```

```
// AHB Agent
m_ahb_agent_cfg = ahb_agent_config::type_id::create("m_ahb_agent_cfg");
if (!uvm_config_db #(virtual ahb_monitor_bfm)::get(this, "", "AHB_mon_bfm",
m_ahb_agent_cfg.mon_bfm))
  `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface AHB_mon_bfm from
uvm_config_db. Have you set() it?")
if (!uvm_config_db #(virtual ahb_driver_bfm) ::get(this, "", "AHB_drv_bfm",
m_ahb_agent_cfg.drv_bfm))
  `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface AHB_drv_bfm from
uvm_config_db. Have you set() it?")
m_env_cfg.m_ahb_agent_cfg = m_ahb_agent_cfg;

// Add in interrupt line
ICPIT = intr_util::type_id::create("ICPIT");
if (!uvm_config_db #(virtual intr_bfm)::get(this, "", "ICPIT_bfm", temp_intr_bfm))
  `uvm_fatal("VIF CONFIG", "Cannot get() interface ICPIT_bfm from uvm_config_db.
Have you set() it?")
ICPIT.set_bfm(temp_intr_bfm);
m_env_cfg.ICPIT = ICPIT;
m_spi_env_cfg.INTR = ICPIT;

uvm_config_db #(pss_env_config)::set(this, "*", "pss_env_config", m_env_cfg);
m_env = pss_env::type_id::create("m_env", this);
endfunction: build_phase

//
// Convenience function to configure the apb agent
//
// This can be overloaded by extensions to this base class
function void pss_test_base::configure_apb_agent(apb_agent_config cfg, int index,
logic[31:0] start_address, logic[31:0] range);
    cfg.active = UVM_PASSIVE;
    cfg.has_functional_coverage = 0;
    cfg.has_scoreboard = 0;
    cfg.no_select_lines = 1;
    cfg.apb_index = index;
    cfg.start_address[0] = start_address;
    cfg.range[0] = range;
endfunction: configure_apb_agent

task pss_test_base::run_phase( uvm_phase phase );

endtask: run_phase
```

Again, a test case that extends this base class would populate its run method to define a virtual sequence that would be run on the virtual sequencer in the env. If there is non-default configuration to be done, then this could be done bypopulating or overloading the build method or any of the configuration methods.

```
//
// Class Description:
//
//
class pss_spi_polling_test extends pss_test_base;

  // UVM Factory Registration Macro
  //
  `uvm_component_utils(pss_spi_polling_test)

  //------------------------------------------
  // Methods
  //------------------------------------------

  // Standard UVM Methods:
  extern function new(string name = "pss_spi_polling_test", uvm_component parent = null);
  extern function void build_phase(uvm_phase phase);
  extern task run_phase(uvm_phase phase);
endclass: pss_spi_polling_test

function pss_spi_polling_test::new(string name = "pss_spi_polling_test",
uvm_component parent = null);
  super.new(name, parent);
endfunction

// Build the env, create the env configuration
// including any sub configurations and assigning virtual interfaces
function void pss_spi_polling_test::build_phase(uvm_phase phase);
  super.build_phase(phase);
endfunction: build_phase

task pss_spi_polling_test::run_phase(uvm_phase phase);
  config_polling_test t_seq = config_polling_test::type_id::create("t_seq");

   t_seq.m_cfg = m_spi_env_cfg;
   t_seq.spi = m_env.m_spi_env.m_spi_agent.m_sequencer;

   phase.raise_objection(this, "Starting PSS SPI polling test");

   repeat(10) begin
     t_seq.start(null);
   end

   phase.drop_objection(this, "Finishing PSS SPI polling test");
   endtask: run_phase
```

The PSS env

The PSS env build process retrieves the configuration object and constructs the various sub-envs, after testing the various has_<sub-component> fields in order to determine whether the env is required by the test case. If the sub-env is to be present, the sub-envs configuration object is set in the PSS envs configuration table. The connect method is used to make connections between TLM ports and exports between monitors and analysis components such as scoreboards.

```
//
// Class Description:
//
//
class pss_env extends uvm_env;

   // UVM Factory Registration Macro
   //
   `uvm_component_utils(pss_env)

   //------------------------------------------
   // Data Members
   //------------------------------------------
   pss_env_config m_cfg;
   //------------------------------------------
   // Sub Components
   //------------------------------------------
   spi_env m_spi_env; gpio_env
   m_gpio_env; ahb_agent
   m_ahb_agent;

   // Register layer adapter
   reg2ahb_adapter m_reg2ahb;
   // Register predictor
   uvm_reg_predictor#(ahb_seq_item) m_ahb2reg_predictor;

   //------------------------------------------
   // Methods
   //------------------------------------------

   // Standard UVM Methods:
   extern function new(string name = "pss_env", uvm_component parent = null);
   // Only required if you have sub-components
   extern function void build_phase(uvm_phase phase);
   // Only required if you have sub-components which are connected
   extern function void connect_phase(uvm_phase phase);

endclass: pss_env
```

```
function pss_env::new(string name = "pss_env", uvm_component parent = null);
  super.new(name, parent);
endfunction

// Only required if you have sub-components
function void pss_env::build_phase(uvm_phase phase);
  if (!uvm_config_db #(pss_env_config)::get(this, "", "pss_env_config", m_cfg) )
    `uvm_fatal("CONFIG_LOAD", "Cannot get() configuration pss_env_config from
  uvm_config_db. Have you set() it?")

  uvm_config_db #(spi_env_config)::set(this, "m_spi_env*", "spi_env_config",
  m_cfg.m_spi_env_cfg); m_spi_env = spi_env::type_id::create("m_spi_env", this);

  uvm_config_db #(gpio_env_config)::set(this, "m_gpio_env*", "gpio_env_config",
  m_cfg.m_gpio_env_cfg); m_gpio_env = gpio_env::type_id::create("m_gpio_env",
  this);

  uvm_config_db #(ahb_agent_config)::set(this, "m_ahb_agent*", "ahb_agent_config",
  m_cfg.m_ahb_agent_cfg); m_ahb_agent = ahb_agent::type_id::create("m_ahb_agent",
  this);

  // Build the register model predictor
  m_ahb2reg_predictor = uvm_reg_predictor#(ahb_seq_item)::type_id::create
  ("m_ahb2reg_predictor", this); m_reg2ahb =
  reg2ahb_adapter::type_id::create("m_reg2ahb");
endfunction: build_phase

// Only required if you have sub-components which are connected
function void pss_env::connect_phase(uvm_phase phase);
  // Only set up register sequencer layering if the pss_rb is the top block
  // If it isn't, then the top level environment will set up the correct sequencer
  // and predictor
  if(m_cfg.pss_rb.get_parent() == null) begin
    if(m_cfg.m_ahb_agent_cfg.active == UVM_ACTIVE) begin
      m_cfg.pss_rb.pss_map.set_sequencer(m_ahb_agent.m_sequencer, m_reg2ahb);
    end
    //
    // Register prediction part:
    //
    // Replacing implicit register model prediction with explicit prediction
    // based on APB bus activity observed by the APB agent monitor
    // Set the predictor map:
    m_ahb2reg_predictor.map = m_cfg.pss_rb.pss_map;
    // Set the predictor adapter:
    m_ahb2reg_predictor.adapter = m_reg2ahb;
    // Disable the register models auto-prediction
    m_cfg.pss_rb.pss_map.set_auto_predict(0);
    // Connect the predictor to the bus agent monitor analysis port
    m_ahb_agent.ap.connect(m_ahb2reg_predictor.bus_in);

  end
endfunction: connect_phase
```

The rest of the testbench hierarchy

The build process continues top-down with the sub-envs being conditionally constructed as illustrated in the block level testbench example and the agents contained within the sub-envs being built as described in the agent example.

Further levels of integration

Vertical reuse for further levels of integration can be achieved by extending the process described for the PSS example. Each level of integration adds another layer, so for instance a level 2 integration environment would contain two or more level 1 envs and the level 2 env configuration object would contain nested handles for the level 1 env configuration objects. Obviously, at the test level of the hierarchy the amount of code increases for each round of vertical reuse, but further down the hierarchy, the configuration and build process has already been implemented in the previous generation of vertical layering.