Improving Efficiency, Productivity, and Coverage Using SystemVerilog OVM Registers

Rich Edelman, Allan Crone, Mark Peryer, Raghu Ardeishar
Mentor Graphics

Mogens Lyngsie
Nokia Modem IP

This paper describes efficiency, productivity and coverage improvements to functional verification attainable by using a SystemVerilog OVM Register package.
Register packages have been used in verification for a number of years in a variety of ways. Verification teams use various register packages and tools today: homegrown, proprietary EDA, open-source. Design and verification teams have adopted a number of these tools and solutions, most non-standard; some moving to standardization.

A subset of the currently used solutions will be described in this paper, followed by a list of register verification requirements, a description of some register testing styles and modes, along with a review of coverage modes and coverage goals and finally a short overview of such tools in use and the issues identified. The SystemVerilog [1] OVM [2] Register Package will be used to illustrate various techniques for verification and its API described listed in the appendix.

REGISTER BACKGROUND

Register packages come in many forms. A collection of open source code (VMM RAL [3], OVM Register Package [4], arv [5]), an input language with generators for many formats (SystemRDL [6], IPXACT [6]) or specialized languages (specman vr_ad). Furthermore, many users have invented their own register source code, input language and code generators.

Most of the solutions share common modeling ideas: a register is a collection of fields or bits. A register file is a collection of registers and a register map is a collection of register files. Tests are written testing specific registers or combinations of registers. Automated tests are also used to test collections of registers. Generally, registers raise the abstraction level of the tests being written, and allow for reuse of tests as system address maps evolve during design and verification.

VMM RAL

The RAL exists today as an open source kit contained in the VMM download. The RAL contains a TCL based specification language, a binary generator and a collection of base classes and built-in tests. The built-in classes include system, block, register and field. The built-in tests may be customized to suit verification needs, and include reset, bit_bash, reg_access, mem_walk, mem_access and shared_access.

Specman vr_ad

The vr_ad has been available as a proprietary solution for a number of years, and has been widely used. There is limited documentation available to the authors on the details, but a few users have related an overall similarity to RAL and the OVM Register Package.

OVM Register Package

The OVM Register Package is an open source kit which evolved from a specification and implementation in the AVM. The OVM Register Package contains base classes which include a register map, register file and registers with fields. Registers are parameterized classes and use analysis ports to publish changes to interested subscribers. The register data is a SystemVerilog packed struct, and the packed struct bit fields are the register bit fields. Constraints can be specified between register fields and between registers. Both automatic and custom coverage can be generated and collected easily.
SystemRDL and IPXACT

SystemRDL and IPXACT are currently under standardization by the SPIRIT Consortium, and are used by a variety of code generators as the register description input language.

Although not a base class library providing simulation time register functionality, these input languages serve as a basis or standard for requirements that should be implemented in a base class library.

Other Register Packages

A variety of other open source and commercial packages exist, which all reflect the basic structure of register testing — a register map, a register file, a register and a field.

Additionally, most verification teams have built internal register verification packages which include scripting, automatic test generation, robust reuse and design specific additions.

REGISTER REQUIREMENTS

The requirements for a register package include many items, but the three most important are standardization of implementation, open source, and agreement on the API.

Standardization of implementation is a strong requirement, but will allow behavior to match among simulator vendors, while at the same time supporting the second requirement — open source.

An open source register implementation allows the benefits that go along with open source — shared implementation, transparency and safety from proprietary software. The register testing layer is a relatively small layer in the scheme of verification, and is best seen as a way to add value to existing verification solutions. As such it doesn’t warrant the place of “proprietary software”.

Agreement on the API follows naturally from the first two requirements, but is perhaps the hardest to satisfy, since there are existing implementations available today which have their strong supporters.

The API should be clean and small, with a clear data model underneath, and should be easy to extend and enhance for customer specific issues.

A register package must be able to map a register name to an address, and back. It must be able to retain the expected contents of the actual register — acting as a shadow. It must be able to handle register fields which have certain properties (READABLE, READ-ONLY, WRITEABLE, WRITE-ONLY, etc), as well as different access types supported for different registers (16-bit, 32-bit, burst access for groups of registers, posted and non-posted writes, etc). It must be able to handle relationships between fields of the same register and fields of different registers. It must be possible to extend the set of supported register properties to include company specific register types.

Although the discussion involves registers, the solution should also consider memories — access to and testing of memories. The memories may be directly embedded in the address space, or may be hidden behind a collection of access registers — so called “indirect memory”.

The register package should not only be usable for automated register testing, but should also contain features that aid in writing functional test cases. The package should function as a general register
interface and in support of this it should contain features like protection of registers, backdoor access, coverage bins, and different access methods to the same register as well as user specific features.

It should be easy to update the register package during the lifetime of the project. The register package should take its input from the same tools as are used normally to describe register layout. Some standard intermediate format may be used. However, it is important that this format is open and accepted as an industry standard. IPXACT is a good example of such a register description format and is supported by most commercial tools.

**REGISTER ACCESS**

Registers exist in the simulated DUT as signals, wires or other simulation constructs. These registers in the DUT are the “real” registers. They operate as part of the operation of the DUT. A register package is used to abstract these physical registers into a higher level which can provide useful verification hooks.

An abstract register package can provide a “shadow” of the DUT registers. The shadow registers reflect the current DUT register values — and must be kept synchronized with the DUT registers. The shadow register doesn’t do this directly — a monitor might monitor bus access, and do the shadow updates. The shadow register used this way is like a database of the DUT registers. It can be used to collect coverage and detect errors in expected values.

The abstract register package can also be used as configuration registers — a DUT can be configured by setting a collection of registers to certain values, and then iterating through those registers, writing the register values into the DUT registers. (See Section VB Configuration and randomization below)

This act of reading or writing DUT registers can be performed in two ways — using a front-door access or using a back-door access. A front-door access is the access that the in-system environment uses. It is the normal way that the system reads and writes registers within the address space. When a front-door access occurs, a bus transfer is initiated, pins are wiggled, and simulation time elapses.

A back-door access is an access made possible because a simulation is running, and a register can be referenced using a name in the simulation like “top.eth01.reg3”. A back-door access is a direct access, poking into the actual simulation database. Back-door access takes no simulation time, and causes little simulation activity. Using back-door access is a fast way to read or write all the registers in a system. It is sometimes used to pre-load the DUT registers to a known state, and start normal simulation.

Below is an automatically generated back-door access for a ‘csr’ register.

```perl
function bit[31:0] backdoor_read_csr();
    bit [31:0] v;
    v = top.sw2.sw.csr;
    return v;
endfunction
```

As can be seen in the code above, this back-door access uses the HDL path in simulation to reference the register (“top.sw2.sw.csr”). This is problematic, since these instance names are likely to change.
frequently. By automatically generating these backdoor routines, or by using some simple text substitution macros, the HDL simulation path name problem can be avoided.

Another approach to back-door access is to use a PLI routine to dynamically find the HDL path

```verilog
define bit[31:0] backdoor_read_csr();
    bit[31:0] v;
    $backdoor_read("top.sw2.sw.csr", v);
    return v;
endfunction
```

This solution has the advantage that the HDL path name can be a simulation variable, but has the disadvantage that the library is now dependent on PLI C code.

Although these backdoor access solutions appear straightforward there are other complications that have caused other approaches to be adopted, including callbacks and connections to automatically generated interfaces with backdoor read and write access routines. Further discussion of backdoor access solutions is beyond the scope of this paper.

**REGISTER TEST STYLES**

Register tests can be used in several ways useful for verification. Some of the uses are listed here.

The first stage in register testing is to check that the defined registers can be reached — i.e. that they can be read from and written to correctly. Only after this basic capability has been established, can other tests based on the register functionality be expected to run correctly.

The register test environment will cause either bus transactions or back-door accesses which will result in the DUT registers being read or written.

In Figure 1 the solid arrow labeled ‘F’ is a frontdoor access. A test is issuing a command to the driver which will cause a bus transaction to occur on the bus. The bus monitor, labeled ‘MON’, monitors the bus, and reports bus transactions to the scoreboard or coverage collector or other interested subscriber.
The dashed arrow labeled ‘B’ is a backdoor access. A test is accessing the register directly using some HDL path scheme. Backdoor accesses need to be mirrored in the shadow, since they are not visible to a bus monitor. Application of automatic (built-in) tests to all user defined registers may be limited under certain circumstances. For example, the automated tests have limited knowledge of the actual register behavior. There are block and system related functionality to account for aside from the register definitions and tests.

In Figure 1 the shadow registers may be accessed by any of the other components (MON, DRIVER Scoreboard, test), depending on architectural design and verification team preference.

A built-in test may need to first initialize the DUT to a known good point before tests can begin. The initialization may require a reset signal and running the clocks, or something more sophisticated.

A built-in test needs to be aware of registers which should not be read or written, due to their special behavior. Reading or writing certain registers may trigger hardware activity — for example, writing to a register may trigger a calculation and change the internal DUT state. These registers should not participate in most built-in tests.

Simple automatic tests such as resetting all the values, reading and writing to all registers, and checking masking in registers is a powerful and useful technique, but most useful register testing is done by knowing how the registers operate together and designing constraints or directed tests which exercise collections of functionality.

Connectivity Checks

The easiest kind of register test is a connectivity check. In a connectivity check the goal is to make sure each register can be read and written. More complex connectivity checks can involve the field values — making sure to write different combinations of the fields, while adhering to any field constraint relationships.

Some registers cannot be completely tested with connectivity checks, since they may be read-only, or when written may cause some activity to take place which will corrupt the state of the registers. Special tests for these types of registers should be written to augment the normal automatic connectivity checks.

The register package should contain pre-defined functions that should be overloaded when doing special test coding. This ensures a coherent look of the extensions. For example, the package should have a function which returns all the registers of a specific type. Then using that function a test could select only registers of a specific type, and operate on them.

An example connectivity check might be the pseudo code and following example, which just tries to make sure all registers have their reset value:

```vhdl
foreach register (registerList) begin
    // Reset the shadow value.
    register.reset();
    // Construct and issue a bus transfer, reading the register.
    ...
```

Using SystemVerilog OVM Registers
// Compare the results.
if (register.compare(response.data))
    error("Register mismatch");
end

The example below is an ovm_sequence — specifically a derived class of a bus_sequence from the OVM Register Package example code.

This code is simple — it fetches the register map, ‘rm’, and then iterates through all the registers in the register map. For each register, it makes sure the shadow contains the reset value by calling ‘r.reset()’. (This test is run when the device has been reset — so the shadow should reflect this).

Then it looks up the register address and constructs a 'bus_transaction'. A read transaction is constructed and issued. The results of the read transaction are compared against the shadow value using 'r.bus_read32()'.

class reset_all_sequence #(type T = int)
    extends bus_sequence#(T);

    task body();
        ovm_register_base r, registers[];
        ovm_register_map rm;
        bit valid_address;

        // 'T' is going to be a my_bus_transaction
        // or some derivative.
        T bus_req, bus_rsp;

        bus_req = new();
        bus_rsp = new();
        rm = ovm_register_map ::
            ovm_register_get_register_map();

        // Get the list from 'rm'.
        registers = rm.get_register_array();
        foreach (registers[i]) begin
            // Go through the list and create
            // a bus transaction.
            r = registers[i];

            // Make sure the shadow value is
            // the reset value.
            r.reset();

            ovm_report_info("RSQR",
                $psprintf("Register %s, resetVal = %0h",
                    r.get_fullname(), r.get_data32()));

            // Build a bus request to send.
            bus_req.addr =
                rm.lookup_register_address_by_name("
Using SystemVerilog OVM Registers

Configuration and Randomization

Collections of registers represent the state of a device or the state of the system. The current register values represent the current state of the system. A device or system can be set to a new state by writing a new set of registers.

Using this ability to set device or system state, a collection of registers can be “randomized” together, and then written to the device or system. Once written, the device is in the defined state, and can begin processing from that point. By constraining and randomizing in interesting ways, various states can be reached directly which may have been hard to reach by other means.

The code below writes the value of the shadow to the bus.

class configure_sequence #(type T = int)
    extends bus_sequence#(T);
    task body();
        ovm_register_base r, registers[];
        ovm_register_map rm;
        bit valid_address;

        // 'T' is going to be a my_bus_transaction
        //   or some derivative.
        T bus_req, bus_rsp;

        bus_req = new();
        bus_rsp = new();
        rm = ovm_register_map::
            ovm_register_get_register_map();

        // Get the list from 'rm'.
        registers = rm.get_register_array();
        foreach (registers[i]) begin
            r = registers[i];

            ovm_report_info("RSQR",
                $psprintf("Register %s, value = %0h", 
                    r.get_fullname(), r.get_data32()));

            r.get_fullname(),
                valid_address);
        bus_req.data = r.get_data32();
        bus_req.rw   = READ_op;

        // Send it. Get the response.
        send_to_bus(bus_req, bus_rsp);

        r.bus_read32(bus_rsp.data);
    end
endtask
endclass
Using SystemVerilog OVM Registers

Directed Tests

Automated and random configurations are powerful and useful, but directed tests are just as powerful using the register abstraction. A directed test in this case is made up of a collection of register writes and reads which taken together cause some activity which may have been hard to achieve by other means.

Directed tests might be created as a collection of function calls each of which is a “useful” work unit. For example a function to read the status register could be combined with a function to increment the address pointer register. These two functions could be used to check that the overflow bits are working properly. Various possibilities for writing directed tests exist, depending on style and situation. The underlying implementation of each may turn out to be the same or similar — three kinds are listed below.

A ‘write()’ routine in a register class:

```plaintext
eth_reg reg = lookup("eth0.reg1");
reg.write(data);
```

A utility write() routine taking an argument:

```plaintext
write("eth0.reg1", data);
```

A lower level implementation generating bus transactions directly:

```plaintext
ovm_register_base r;
r = rm.lookup_register_by_name("eth0.reg1");

// Build a bus request to send.
bus_req.addr =
    rm.lookup_register_address_by_name(
        r.get_fullname(),
        valid_address);

bus_req.data = r.get_data32();
bus_req.rw   = W_op;

// Send it. Get the response.
send_to_bus(bus_req, bus_rsp);
```
Directed tests can also be used to “set up” a random test. A sequence of directed register writes are performed. After the writes are finished, the DUT is in a certain state from which random testing can commence.

**Reusing Register Tests**

Once a register test is written for a block or device it may be desirable to reuse that test in a larger test. For example, a device configuration test — which puts a device in a certain state, might be usable from a larger test in which all devices are being set to some starting configuration.

Since tests are written using register names instead of specific addresses, they are more reusable as the design changes and as the test is reused in larger blocks and the system.

**COVERAGE**

Collecting coverage on registers helps understand how well the tests are exercising different register values and relationships. Additionally, register files and register maps can be covered to check how well the address space has been sampled.

**Coverage for Registers and Fields**

Register coverage is first answering the question “has the register been read or written in any form”. Next comes checking if each field has been read or written. Covering fields is obvious, each value should be written and read, but some fields are not writable or readable, and some values are harder to check than others (like error status bits or interrupt bits).

Additionally, covering all bits of all fields of all registers is likely to overload the simulation or the reports. It is important to plan coverage for individual fields carefully.

Listed below is a simple register definition with a covergroup. It counts certain values of the fields that are the legal values.

```vhdl
typedef struct packed {
  bit [3:0] addr1; // Legal values
  bit [3:0] addr2; // - 0, 1, 2, 3.
  enum {WRITE, READ} rw;
} r_t;

class my_r extends ovm_register #(r_t);
  covergroup r_cv;
  ...
  addr1_val: coverpoint data.addr1 {
    bins even [] = { 0, 2 };
    bins odd [] = { 1, 3 };
  }
  addr2_val: coverpoint data.addr2 {
    bins even [] = { 0, 2 };
    bins odd [] = { 1, 3 };
  }
  rw_val: coverpoint data.rw;
endgroup
```

*Using SystemVerilog OVM Registers*
function void sample();
        r_cv.sample();
endfunction

Coverage for Collections of Fields

Although field coverage as outlined above can be easily automated, it is not as useful as coverage on
collections of fields and their relationships. Collections of fields describe a “state” or “configuration”,
and should be covered if those states are important — for example a counter with an increment of 2,
counting up.

A covergroup needs to be built to represent a functional relationship — i.e. named bins to catch a
particular configuration or result. Crossing the contents of different fields of different registers creates
a set of orthogonal configurations or states. These collections of functional relationships taken together
describe the functionality or configuration of the hardware under test. In order to compute coverage,
the covergroups need to sample the current state of the registers. An abstract register model allows you
to do this by reference more easily than accessing the internal registers of the DUT. Proper sampling of
the covergroups is critical to producing correct coverage results.

The collection and coverage of functional relationships is hard to automate, but a register package
which offers a simple to use API and is extendable will allow writing custom coverage extensions.
Listed below is a register file which contains two registers that have relationships between their fields.
The crosses c1, c2 and c3 represent the “interesting” functionality between registers.

class rf extends ovm_register_file;
    rand my_r r1;
    rand my_r r2;

    // Some interesting relationship
    covergroup rf_cx;

    r1_addr1: coverpoint r1.data.addr1 {
        bins even [] = { 0, 2 };
        bins odd [] = { 1, 3 };
    }
    r2_addr2: coverpoint r2.data.addr2 {
        bins even [] = { 0, 2 };
        bins odd [] = { 1, 3 };
    }
    r1_rw: coverpoint r1.data.rw;
    r2_rw: coverpoint r2.data.rw;

    c1 : cross r1_addr1, r2_rw;
    c2 : cross r2_addr2, r1_rw;

    c3 : cross r1_rw, r2_rw;
endgroup
function void sample();
    rf_cx.sample();
endfunction

Coverage for Register Files and Register Maps

A register file represents the registers in a device. Those registers have addresses in the register file. A
register can have more than one address. A register map represents a collection of devices or a sub-
system. A register map is made up of many register files, with many addresses.

The register map should have coverage for both mapped and unmapped addresses are used, and that
certain addresses close to the mapped address boundary have been tested.

An easily automated covergroup using a cross between register addresses, and READ/WRITE can
catch gaps in tests. Listed below is a register map with a covergroup named ‘rf_addr2’ which will
check that each mapped address is covered. Detailed discussion of the covergroup design is beyond the
scope of this paper, but the biggest problem faced is that a covergroup needs to have a set of values at
compile-time, and mapped addresses may be generated late in simulation.

class my_register_map ...;
    ...
    // Must be guaranteed to only return 0 to n.
    //  - 0 to n-1 are for mapped addresses.
    //  - n is for unmapped addresses.
    function int map(T address);
    ...
endfunction

covergroup rf_addr2 (int n)
    with function sample(T address, op_t rw);
    ...
    coverpoint map(address) {
        bins needed [] = { [ 0 : n-1 ] };
        bins unmapped = { n };
    }
    cr1: cross rw, address;
endgroup
    ...
function new();
    ...
    n = 4;
    rf_addr2 = new(n); // Pass n to covergroup
    // constructor.
endfunction
endclass

class test;
    ovm_register_base register_list[];
    ovm_register_map rm;
    ...
    task run();
foreach ( register_list[s] ) begin
    // Generate a new register value.
    assert(register_list[s].randomize());

    // 1. Issue a bus transaction -
    //      Wiggle pins.
    register_list[s].sample();

    // Collect coverage on the "shadow"
    // register.
    register_list[s].sample();

    // Collect coverage on the "shadow"
    // register map
    rm.sample();
    rm.rf_addr2.sample(i, bus_trans.rw);
end
endtask
endclass

USING A REGISTER PACKAGE

The ovm_register register package has been applied in a larger GSM/WCDMA modem project. The data base functionality was used in register testing as well as in functional testing. The register database was created within the ovm_env environment and stored in a configuration object. In this way it is easily accessible from all parts of the testbench.

Default value and connectivity checking of all registers was made with very little effort as described in the above section, Register Test Styles.

Three customizations were made to the register package. These are the description of the special ID registers (Appendix I — An ID Register), the description of the memory access registers and the addition of semaphores in all registers. (See below — nve_register definition).

In all modules in the model, the first address location is used for module ID information. Special functions for doing automated testing of these were added.

Many of the blocks in the design contain embedded memories for internal use. Most of these memories are also accessible through some dedicated register via the processor bus. A special custom type of register was used to describe this type of register in the register package.

Access of memory locations can be done in a number of different ways. A simple method for doing simple read/writes was used by the test cases. Within the register class these methods will then result in one of a number of register read/write patterns that implement the desired function. Storing these methods within the register package hides these different access patterns from the test case and makes it easy to use the same tests with different access methods. E.g. the memory load method is randomly selected for each access or on a test case basis.
The `ovm_register` package used in the project did not contain support for semaphore protection of registers. This is needed when different tests need to access the same register. With a semaphore it is possible to make a read-modify-write operation on a register without the two parts of the tests interfering with each other.

The source code below shows the very simple code used to add the semaphore functionality. This is then available for all registers in the design.

```verilog
//Base class for register declarations at Nokia
virtual class nve_register#(type T = int)
    extends ovm_register #(T);

protected semaphore access_right;

function new(string l_name = "registerName",
    ovm_component p = null, T l_resetVal = 0);
    super.new(l_name, p);
    access_right = new(1);
endfunction

task put_access_right();
    access_right.put(1);
endtask

task get_access_right();
    access_right.get(1);
endtask

function int try_get_access_right();
    if (access_right.try_get(1) == 0)
        return 0;
    else
        return 1;
endfunction
endclass: nve_register;
```

All the additions to the basic `ovm_register` package were easily done because all source code is openly available.

**CONCLUSION**

An introduction to register test writing has been presented along with requirements for such a package, some suggested use models and a discussion of usage of such a package.

Using registers for verification is not new. Registers are already used widely and in varied ways. The OVM register API outlined by this paper is a register package which is open-source, OVM based, with a small API and flexible use-model. Most importantly this package has been successfully deployed in
various locations around the world in a variety of use-models. Its flexibility and simplicity allows customization in the form of user specific added-value functionality.

Since this package is flexible and customizable, it has been easy for verification teams to replace home-grown register solutions as they transition to a supported register approach that is based on a standard language and standard tools.

REFERENCES


This paper appeared in Proceedings of DVCon 2009, San Jose, CA, pp. 166–173.
APPENDIX I — AN ID REGISTER

// The ID register is a read only register where
// each read cycles through 8 states where each
// state reveals a new read value.
// Immediately after reset, the ID register returns
// the read value for location 0, the next read
// returns read value for location 1 etc.
class id_register extends ovm_register #(logic[31:0]);
    logic[31:0] id_values[];
    int id_ptr;
    int id_width = 32;

covergroup id_reads(string name);
    option.name = {name, "_id_reads"};
    option.comment = "ID register locations read";
    option.per_instance = 1;
    cp_id_reads : coverpoint id_ptr {
        bins ptr_vals[8] = {[0:7]};
    }
endgroup: id_reads

covergroup id_writes(string name);
    option.name = {name, "_id_writes"};
    option.comment = "ID register locations read";
    option.per_instance = 1;
    cp_id_writes : coverpoint id_ptr {
        bins ptr_vals[8] = {[0:7]};
    }
endgroup: id_writes

function new(string l_name = "default_id_reg",
              ovm_named_component p = null,
              logic[31:0] id[] = '{0,0,0,0,0,0,0,0});
    super.new(l_name, p);
    id_reads  = new(l_name);
    id_writes = new(l_name);
    id_values = new[id.size()] (id);
    id_ptr = 0;
    data = id_values[id_ptr];
    register_type = "ID";
endfunction

function void sample();
    id_reads.sample();
endfunction: sample
// Simulates the effect of a read from the ID register
function void id_read(bytearray_t read_data);
    id_ptr++;
    if (id_ptr == id_values.size())
        id_ptr = 0;
    data = id_values[id_ptr];
endfunction: id_read

// Simulates the effect of a write to the ID register
function void id_write(int new_id_ptr_value);
    if (new_id_ptr_value <= id_values.size())
        begin
            id_ptr = new_id_ptr_value;
            data = id_values[id_ptr];
            id_writes.sample();
        end
    else
        ovm_report_error("ID_register",
            "Illegal ID pointer value written, value ignored");
endfunction: id_write

...
virtual class ovm_register_base
extends ovm_transaction;

function new(string l_name = "", 
             ovm_component p = null);

virtual function string get_full_name();
virtual function string get_name();
virtual function int   get_num_bits();

virtual function void set_data32( logic[31:0] bv);
virtual function logic [31:0] get_data32();

virtual function void set_data64( logic [63:0] bv);
virtual function logic [63:0] get_data64();

virtual function void set_byte_array( 
                         input bytearray_t i);
virtual function void get_byte_array( 
                         ref    bytearray_t o);

virtual function void sample();
virtual function void reset();

pure virtual function void bus_read32( 
                         input bit[31:0]   read_data_bv );
pure virtual function void bus_write32( 
                         input bit[31:0]   write_data_bv );
pure virtual function void bus_read( 
                         input bytearray_t read_data );
pure virtual function void bus_write( 
                         input bytearray_t write_data );
endclass

virtual class ovm_register #(type T = int)
extends ovm_register_base;

function new(string l_name = "registerName", 
             ovm_component p = null, T l_resetVal = 0);

virtual function void bus_read_bv( 
                         input bitvector_t read_data_bv );
virtual function void bus_write_bv( 
                         input bitvector_t write_data_bv );

virtual function T   read_without_notify();
virtual function void write_without_notify(T v);
virtual function T read();
virtual function void write(T v);

virtual function void poke(T v);
virtual function T peek();

virtual function bit compare_data(T new_data);
virtual function bit compare(this_type b);

virtual function void sample();
virtual function void reset();
endclass

virtual class ovm_register_map_base
    extends ovm_component;
    function new(string name, ovm_component p);

    virtual function register_list_t get_register_array();

    function void add_register(
        string name, offset_t offset,
        ovm_register_base register);

    function void add_register_in_range(
        string name,
        offset_t first_offset, offset_t last_offset,
        offset_t grid, ovm_register_base register);

    virtual function address_list_t
        lookup_register_addresslist_by_name(string name);

    virtual function address_t
        lookup_register_address_by_name (string name);

    virtual function ovm_register_base
        lookup_register_by_name     (string name);

    virtual function ovm_register_base
        lookup_register_by_address (offset_t offset);
endclass

class ovm_register_file extends ovm_register_map_base;
    function new(string name, ovm_component p);
virtual function register_list_t get_registers();
virtual function void report_version_numbers();
endclass

class ovm_register_map extends ovm_register_file;
function new(string name, ovm_component p);
function void add_register_map(
    ovm_register_map rm, address_t addr);
function void add_register_file(
    ovm_register_file rf, address_t addr);

static function ovm_register_map
    ovm_register_get_register_map(
        string config_name = "register_map");
endclass