UVM Register Abstraction Layer Generator User Guide

M-2017.03, March 2017

Contents

1

Code Generation

Once a description of the available registers and memories in a design is available, `ralgen` can automatically generate the UVM RAL abstraction model for these registers and memories. Test cases, firmware, device drivers and DUT configuration code use this model to access the registers and memories through an object-oriented abstraction layer. The predefined tests also use this model to verify the functional correctness of the registers and memories.

Generating a RAL Model

To generate the RAL model, use the following command:

```
% ralgen [options] -t topname  -I dir -uvm {filename.ralf}
```

Where:

`-t topname`

The name of the top-level block or system description in the RALF file that entirely describes the design under verification.

-uvm

Specifies UVM as the implementation methodology for the generated code. The RAL model for the entire design is generated in either a file named `ral_topname.sv` in the current working directory.

`-I dir`

An optional list of directories that `ralgen` searches for sourced Tcl files.

`*filename*.ralf`

The name of the files containing the RALF description. Although, the `.ralf` extension is not required, Synopsys recommends you specify it. However, for multiple files, you specify one top-level RALF file, which should include (source) all the other files through the `include` Tcl option. For example, in the top RALF file, you will have, *`source bottom`*`.ralf.`

Options

The following options are available:

`-all_fields_rand`

Allows you to configure all the writable fields as `rand` (`is_rand_bit` is set to 1) without requiring the constraint block to be specified. The generated code only marks fields, which have constraints defined with them as `rand`, by default.

`-b`

Generate the back-door access code for those registers and memories where a complete `hdl_path` has been specified.

`-c a`

Generate the “Address Map” functional coverage model. The `-c` option may be specified multiple times.

`-c b`

Generate the “Register Bits” functional coverage model. The `-c` option may be specified multiple times.

`-c f`

Generate the “Field Values” functional coverage model. The `-c` option may be specified multiple times.

`-c s`

Generate separate bins for read-only bits to read both 1 and 0 for read-only registers for ralgen-generated bit-level coverage. Specifying the `-c b` option alongside the `-c s` option will result in an error.

`-e`

Generate empty constraint blocks for every abstract class.

`-f <filename>`

Specifies all the ralgen options within a file.

`-gen_html`

Generates the RAL model and its HTML UVM document. The related files are dumped in the `ral_top_path_name_doc` directory. An error appears, if this option is specified without `-uvm` option.

`-top_macro <string_which_overrides_default_macro>`

Allows you to use a different macro instead of *`name_TOP_PATH`* to specify the absolute path to the instance of the DUT that corresponds to the RAL model.

Embedding Enums in Field Classes

If an enum is present in a field, ralgen creates a class containing that enum, but that class is not used and the enum is not embedded in the field class.

Ralgen is enhanced and the `-embed_enum_in_flds` option is provided to allow embedding an enum in the field class itself. If this option is used, ralgen embeds an enum in a class extending from the `uvm_reg_field` class and creates the corresponding field instance using this `uvm_reg_field` extension class.

For example, for the following RALF code:

```
block b1 {
    bytes 1;
    register r {
        bytes 1;
            field WDT_EN @'h5 {
                bits 1;
                reset 'h0;
                access rw;
                enum { ENABLE = 1, DISABLE = 0 };
            }

    }
}
```

The field class is extended from the `uvm_reg_field` class and used in a register as follows:

```
class ral_fld_b1_r_WDT_EN extends uvm_reg_field;
    `uvm_object_utils(ral_fld_b1_r_WDT_EN)

    function new(string name = "WDT_EN");
super.new(name);
    endfunction : new
    typedef enum bit[0:0] {
        ENABLE = 1,
        DISABLE = 0
    } WDT_EN_values;
endclass : ral_fld_b1_r_WDT_EN

class ral_reg_b1_r extends uvm_reg;
    rand ral_fld_b1_r_WDT_EN WDT_EN;

    function new(string name = "b1_r");
      super.new(name, 8,build_coverage(UVM_NO_COVERAGE));
    endfunction: new

virtual function void build();
    this.WDT_EN =
ral_fld_b1_r_WDT_EN::type_id::create("WDT_EN",,get_full_na
me());
    this.WDT_EN.configure(this, 1, 5, "RW", 0, 1'h0, 1, 1,
1);
```

```
endfunction: build

     `uvm_object_utils(ral_reg_b1_r)

endclass : ral_reg_b1_r
```

Splitting Model into Separate Files

Instead of generating a complete model into a single file named ral_topblkname.sv, the generated register model can be optionally split into individual files, one per block. The individual files can then be used in the verification environment of the individual blocks.

Command-Line Option

```
-F
```

When specified, saves the generated model in separate files.

Specification

For the top-level block, it instantiates each sub-block, which is specified in the input RALF file(s). The corresponding register abstraction model is generated in a separate file named "ral_blkname.sv".

The file contains all the class declarations implied by the block specification. This includes register type, register file type and the block type classes.

Register and register file types specified outside of a block specification is surrounded by pre-processor guard statements.

The `include statements include the files containing the generated model for the blocks instantiated in the block for the current file will be generated at the top of the block model file. The entire content of the file shall be embedded in a pre-processor guard.

Example

The following (partial) RALF file:

```
register R0 {…};
block B1 {
   register R0;
   register R1 {…};
}
block B2 {
   register R0;
   register R1 {…};
}
system sys {
bytes 16;
block B1;
block B2
}
ralgen -uvm -t sys sys.ralf -F
```

It generates the required (partial) class declarations in the following files:

ral_B1.sv:

```
`ifndef RAL_B1__SV
`define RAL_B1__SV

   `ifndef RAL_R0__TYP
   `define RAL_R0__TYP
      class ral_R0 …;
   `endif

   class ral_B1_R1 …;
   class ral_B1 …;
`endif
ral_B2.sv:
`ifndef RAL_B2__SV
`define RAL_B2__SV

   `include "ral_B1.sv"

   `ifndef RAL_R0__TYP
   `define RAL_R0__TYP
      class ral_R0 …;
   `endif

   class ral_B2_R1 …;

   class ral_B2 …;
`endif
```

Splitting Model into Separate Packages

Instead of generating a complete model into a single file named ral_topblkname.sv, the generated register model can be optionally split into individual files, one per block, with all block-related type declarations encapsulated in a package. The individual files can then be compiled separately and used in the verification environment of the individual blocks.

Command-Line Option

`-P`

When specified, saves the generated model in separate packages and in separate files.

Specification

- A warning appears if `-F` option is specified with the `-P` option.
- The `-P` option takes the precedence.
- For the top-level block, and each sub-block it instantiates, specified in the input RALF file(s), the corresponding register abstraction model is generated in a separate packaged named `ral_blkname_pkg` in a separate file named "ral_blkname_pkg.sv".
- The package contains all the class declarations implied by the block specification. This includes register type, register file type and the block type classes.
- Register and register file types specified outside of a block specification is included in the package without special treatment.
- Import statements to import the declarations from the packages containing the generated model for the blocks instantiated in the block for the current package shall be generated at the top of the package.
- The entire content of the file is embedded in a pre-processor guard.
- The list of vlogan commands required to compile the packages in the correct order is generated into a file name "ral_topblkname.compile" and duplicated on the standard output.

Example

The following (partial) RALF file:

```
register R0 {…};
block B1 {
   register R0;
   register R1 {…};
}
block B2 {
   register R0;
   register R1 {…};
}
system sys {
bytes 16;
     block B1;
     block B2;
}
ralgen -uvm -t sys sys.ralf -P
```

Generates the required (partial) class and package declarations in the following files:

ral_B1_pkg.sv:

```
`ifndef RAL_B1_PKG__SV
`define RAL_B1_PKG__SV

package ral_B1_pkg;
   import uvm_pkg::*;

   class ral_R0 …;
   class ral_B1_R1 …;
   class ral_B1 …;
endpackage

`endif
ral_B2_pkg.sv:
`ifndef RAL_B2_PKG__SV
`define RAL_B2_PKG__SV

package ral_B2_pkg;
   import uvm_pkg::*;
   import ral_B1_pkg::*;

   class ral_R0 …;
   class ral_B2_R1 …;
   class ral_B2 …;
endpackage

`endif
ral_B2.compile:
vlogan -sverilog ral_B1_pkg.sv
vlogan -sverilog ral_B2_pkg.sv
```

Generating Model for Top-Level Block Only

Instead of generating a complete model which includes all the blocks instantiated in the top-most block, only the model for the top-most block can be optionally generated. It is assumed that the register models for the instantiated blocks are previously generated. This allows the SoC team to generate the register model for the SoC only and inherit the register models for the instantiated IPs from the IP-level verification environments.

Note:Individually generated register models must use the same packaging approach to be compatible. For example, the model for a block generated without a package (using the `-T` option) is not compatible with the model for a block generated within a package (using the `-P` option).

If you need to generate a register model for multiple blocks, each can be generated through a separate invocations of ralgen, with each block specified as the topmost block in turn.

Command-Line Option

`-T`

When specified, generates the register model for the specified top-most block.

Specification

- It is necessary to specify `-F` option or `-P` option when specifying `-T` option.
- Only the file corresponding to the specified top-most block is generated.

Example

The following (partial) RALF file:

```
register R0 {…};
block slave1 {
   register R0;
   register R1 {…};
}


block slave2 {
   register R0;
   register R1 {…};
}
system sys {
bytes 16;
      block slave1;
      block slave2;
}
```

Generates the required (partial) class and package declarations in the following files:

```
ralgen -uvm -t sys slave_two_block.ralf -F -T
    will generate ral_sys.sv
ralgen -uvm -t slave1 slave_two_block.ralf -F -T
    will generate ral_slave1.sv
ralgen -uvm -t slave2 slave_two_block.ralf -F -T
    will generate ral_slave2.sv
```

Prunable Register Model

Large SoCs contain multiple thousands of registers. Depending on the way the registers are modeled, this may cause the generation of multiple thousands of classes declarations and the instantiation of several tens of thousands of class instances.

Most SoC-level tests do not need a complete register model. They only need the portions of the model that are used by the testcase. To reduce compile-time and run-time resources, it should be possible to prune the register model of all unnecessary block-level register models.

Command-Line Option

```
+prunable
```

When specified, helps the generated model to include directives to optionally prune the register model at compile time.

Specification

- It is necessary to specify the `-F` option or the `-P` option when specifying the `+prunable` option.
- For each block instantiated in the register model contained in the generated file, the corresponding `include or import directive shall be embedded in a pre-processor guard.
- For each block instantiated in the register model contained in the generated file, the corresponding block instance class property shall be embedded in the same pre-processor guard that guards the `include or import statement for that block.

- Any statement referring to a block instance class property is embedded in the same pre-processor guard that guards the `include or import statement for that block.
- The pre-processor symbol guarding the `include or import statement and class property for an instantiated block is named `RAL_PRUNE_BLKNAME`.
- By default, none of the pre-processor guard symbols is defined.

Example

The following (partial) RALF file:

```
register R0 {…};
block B1 {
   register R0;
   register R1 {…};
}

block B2 {
   register R0;
   register R1 {…};

}
system sys {
bytes 16;
     block B1;
     block B2;
}
ralgen -uvm -t sys sys.ralf +prunable -P
```

Generates the required (partial) prunable class and package declarations in the following files:

ral_B1_pkg.sv:

```
`ifndef RAL_B1_PKG__SV
`define RAL_B1_PKG__SV
```

```
package ral_B1_pkg;
   import uvm_pkg::*;

   class ral_R0 …;
   class ral_B1_R1 …;
   class ral_B1 …;
endpackage

`endif
ral_B2_pkg.sv:
`ifndef RAL_B2_PKG__SV
`define RAL_B2_PKG__SV

package ral_B2_pkg;
   import uvm_pkg::*;
   `ifndef RAL_PRUNE_B1
      import ral_B1_pkg::*;
   `endif

   class ral_R0 …;
   class ral_B2_R1 …;
   class ral_B2 extends uvm_reg;
      rand ral_R0    R0;
      rand ral_B2_R1 R2;
      `ifndef RAL_PRUNE_B1
      rand ral_B1    b1;
      `endif
      ...
      function void build();
         `ifndef RAL_PRUNE_B1
         b1 = ...;
         `endif
      endfunction
   endclass
endpackage

`endif
```

Understanding the Generated Model

The generated abstraction model is a function of the RALF description used to generate it. Therefore, understanding how the generation process works will help you use the generated model based on the knowledge of the RALF description.

The generated abstraction model is described using a bottom-up approach, in the order in which the classes are generated and then compiled. If you prefer to read a top-down description, simply read the following sections ("Fields" , "Registers" , "Register Files" , "Virtual Registers" , "Memories" , "Blocks" , and "Systems") in the reverse order.

Note:

> By default, the data in the register or memory access routines is two-state data. The `UVM_REG_4STATE_DATA` macro allows you to get four-state data in the register or memory access routines and controls the definition of the `uvm_reg_data_t` data type.

Fields

No abstraction class is generated for a field definition. Instead, each field is modeled by an instance of the `uvm_ral_field` class.

The instance of that class is stored in a property of the class modeling the register that instantiates it and the block that instantiates the register.

Registers

An abstraction class is generated for each register definition. For each:

- Independently defined register named `regnam`, there is a class named `ral_reg_regnam`
- Register named `regnam` defined inline in the specification of a block named `blknam`, there is a class named `ral_reg_blknam_regnam`
- Register named `regnam` defined inline in the specification of a register file named `filnam` in a block named `blknam`, there is a class named `ral_reg_blknam_filnam_regnam`

In all cases, the register abstraction class is derived from the `uvm_reg` class.

All virtual methods defined in the `uvm_reg` class are overloaded in the register model class. Each virtual method is overloaded to implement register-specific behavior of the register as defined in the RALF description. No new methods are added to the register abstraction class.

As shown in Example 1-1, the register abstraction class contains a class property for each field it contains. The name of the property is the name of the field. There are no properties for unused or reserved fields.

Example 1-1 Register Model Class for Register in Example A-5

```
class ral_reg_CTRL extends uvm_ral_reg;
   uvm_ral_field TXE;
   uvm_ral_field RXE;
   uvm_ral_field PAR;
   uvm_ral_field DTR;
   uvm_ral_field CTS;
   ...
endclass: ral_reg_CTRL
```

Instances of this class are found in the block abstraction class for the blocks instantiating this register.

Arrays

If a register contains any field array, the class property for the field array is declared as a fixed sized array in the corresponding register abstraction class.

Example 1-2   Array Specifications and Corresponding Model

```
register r {
      bytes 1;
field f[8] {
bits 1;
  }
}
```

Corresponding abstraction model:

```
class ral_reg_b_r extends uvm_ral_reg;
   rand uvm_ral_field f[8];
   ...
endclass: ral_reg_b_r
```

Register Files

An abstraction class is generated for each register file definition. For each register file named `filnam` defined inline in the specification of a block named `blknam`, there is a class named `ral_regfile_blknam_filnam`. The register abstraction class is not derived from the `uvm_blk_filnam` base class and is a container for the registers instantiated in the register file.

The register file container class contains a class property for each register it contains.

Example 1-3 Register File Specification and Corresponding Model

```
block dma_ctrl {
   regfile chan {
      register src {
         field addr { ... }
      }
      register dst {
         field addr { ... }
      }
      register count {
         field n_bytes { ... }
      }
      register ctrl {
         field TXE { ... }
         field BSY { ... }
      }
   }
}
```

Corresponding abstraction model:

```
class ral_regfile_dma_ctrl_chan;
   ral_reg_dma_ctrl_chan_src src;
   uvm_ral_field             src_addr;

   ral_reg_dma_ctrl_chan_dst dst;
   uvm_ral_field             dst_addr;

   ral_reg_dma_ctrl_chan_count count;
   uvm_ral_field               n_bytes, count_n_bytes;
   uvm_ral_field               TXE, ctrl_TXE;
   uvm_ral_field               BSY, ctrl_BSY;
   ...
endclass: ral_reg_dma_ctrl_chan
```

Instances (usually arrays of instances) of this class are found in the block abstraction class for the blocks instantiating this register file.

Virtual Registers

An abstraction class is generated for each virtual register array definition. For each independently defined virtual register array named `vregnam`, there is a class named `ral_vreg_vregnam`. For each virtual register array named `vregnam` defined inline in the specification of a block named `blknam`, there is a class named `ral_vreg_blknam_vregnam`. In both cases, the virtual register array abstraction class is derived from the uvm_vreg class. A single abstraction class is used for all virtual registers in the array.

All virtual methods defined in the uvm_ral_vreg class are overloaded in the virtual register array abstraction class. Each virtual method is overloaded to implement register-specific behavior of the virtual register array as defined in the RALF description. No new methods are added to the virtual register array abstraction class.

As shown in Example 1-4, the virtual register array abstraction class contains a class property for each virtual field it contains. The name of the property is the name of the field. There are no properties for unused or reserved fields, and unlike register arrays, a single instance of the virtual register array abstraction class is used to model the complete virtual register array.

Example 1-4   Virtual Register Abstraction Class

```
block blk1 {
   memory ram0 { ... }

   virtual register dma[256] ram0@0x0000 {
      field len { ... }
      field bfrptr { ... }
      field ok { ... }
   }
}
```

Corresponding abstraction model:

```
class ral_vreg_blk1_dma extends uvm_ral_vreg;
   uvm_ral_vfield len;
   uvm_ral_vfield bfrptr;
   uvm_ral_vfield ok;
   ...
endclass: ral_vreg_blk1_dma

class ral_block_blk1 extends uvm_reg_block;
   uvm_ral_mem       ram0;
   ral_vreg_blk1_dma dma;
   ...
endclass: ral_block_blk1
```

A single instance (not an array of instance) of this class is found in the block abstraction class for the blocks instantiating a virtual register array.

Memories

An abstraction class is generated for each memory definition. For each independently defined memory named `memnam`, there is a class named `ral_mem_memnam`. For each memory named `memnam` defined inline in the specification of a block named `blknam`, there is a class named `ral_mem_blknam_memnam`.

In both cases, the memory abstraction class is derived from the `uvm_ral_mem` class.

All virtual methods defined in the `uvm_ral_mem` class are overloaded in the memory abstraction class. Each virtual method is overloaded to implement memory-specific behavior of the memory as defined in the RALF description. No new methods are added to the memory abstraction class.

As shown in Example 1-5, the memory abstraction class contains no additional class properties.

Example 1-5   Memory Abstraction Class for Memory in Example A-10

```
class ral_mem_ROM extends uvm_ral_mem;
   ...
endclass: ral_mem_ROM
```

Instances of this class are found in the block abstraction class for the blocks instantiating this memory.

Blocks

An abstraction class is generated for each block definition. For each independently defined block named `blknam`, there is a class named `ral_block_blknam`. For each block named `blknam` defined inline in the specification of a system named `sysnam`, there is a class named `ral_block_sysnam_blknam`. In both cases, the block abstraction class is derived from the `uvm_reg_block` class.

All virtual methods defined in the `uvm_reg_block` class are overloaded in the block abstraction class. Each virtual method is overloaded to implement block-specific behavior of the block as defined in the RALF description. No new methods are added to the block abstraction class.

As shown in Example 1-6 and Example 1-7, the block abstraction class contains a class property for each register and register file it contains. The name of the register or register file property is the name of the register or file. The block abstraction class also contains one or two class properties for each field it contains. The name of each field property is the name of the field (if unique within the register) and the name of the register concatenated with the name of the field, respectively. There are no properties for unused or reserved fields.

In certain situations, it may be desirable to,

- Not generate any field properties outside the register class.

  or

- Generate field properties with register name pre-fixed to field name, while leaving out field properties with unique names.

You can control the field generation using the following command line option:

```
-flds_out_reg all | none |no_uniq
```

where,

`all` - This is default. It implies the current behavior.

`none` - This specifies that no field handles are generated outside the register class.

`no_uniq` - This specifies that no field handles are generated for unique names outside the register class.

Important:

It is preferable that field names be unique across blocks. Therefore, each field has a property with the same name in the block abstraction class that instantiates them. If you move the field to another physical register, you can use this uniquely-named field property to reduce testbench maintenance. If you use the name that is prefixed with the register name, you must modify testbenches if the field is relocated to another physical register.

Example 1-6 Block Abstraction Class for Block in Example A-11

```
class ral_block_uart extends uvm_reg_block;
   ral_reg_CTRL  CTRL;
   uvm_ral_field TXE, CTRL_TXE;
   uvm_ral_field RXE, CTRL_RXE;
   uvm_ral_field PAR, CTRL_PAR;
   uvm_ral_field DTR, CTRL_DTR;
   uvm_ral_field CTS, CTRL_CTS;

   ral_mem_tx_bfr tx_bfr;
   ...
endclass: ral_block_uart
```

Example 1-7 Block Abstraction Class for Block in Example A-13

```
ral_block_bridge extends uvm_reg_block;
   ral_reg_flags     pci_flags;
   uvm_ral_field     pci_flags_cts;
   uvm_ral_field     pci_flags_dtr;

   ral_reg_data_xfer to_ahb;
   uvm_ral_field     to_ahb_data;

   ral_reg_data_xfer frm_ahb;
   uvm_ral_field     frm_ahb_data;

   ral_reg_flags     ahb_flags;
   uvm_ral_field     ahb_flags_cts;
   uvm_ral_field     ahb_flags_dtr;

   ral_reg_data_xfer to_pci;
   uvm_ral_field     to_pci_data;
   ral_reg_data_xfer frm_pci;
   uvm_ral_field     frm_pci_data;
   ...
endclass: ral_block_bridge
```

Instances of this class are found in the system model class for the systems instantiating this block.

Arrays

If a block contains a register array or register file array, the class property for the register array or register file array is declared as a fixed sized array to the corresponding register abstraction class or register file container class. Similarly, the field properties for the fields contained in the register array are declared as a fixed sized array of `uvm_ral_field` classes.

Example 1-8   Array Specifications and Corresponding Model

```
block b1 {
   register r1[32] {
      field f1 { ... }
   }
   regfile  rf[16] {
      register r1 {
         field f1 { ... }
      }
      register r2[4] {
         field f1 { ... };
      }
   }
}
```

Corresponding abstraction model:

```
class ral_regfile_b1_rf;
   ral_reg_b1_rf_r1 r1;
   uvm_ral_field    r1_f1;

   ral_reg_b1_rf_r2 r2[4];
   uvm_ral_field    f2_f1[4];
   ...
```

```
endclass: ral_regfile_b1_rf

class ral_block_b1 extends uvm_reg_block;
   ral_reg_b1_r1 r1[32];
   uvm_ral_field f1[32], r1_f1[32];

   ral_regfile_b1_rf rf[16]
   ...
endclass: ral_block_b1
```

Systems

An abstraction class is generated for each system definition. For each independently defined system named `sysnam`, there is a class named `ral_sys_sysnam`. For each subsystem named `subnam` defined inline in the specification of a system named `sysnam`, there is a class named `ral_sys_sysnam_subnam`.

In both cases, the system abstraction class is derived from the `uvm_reg_block` class as in the case of the abstraction classes generated for each 'block' definition (for details, see "uvm_reg_block" in the UVM1.0 Reference Guide).

All virtual methods defined in the `uvm_reg_block` class are overloaded in the system abstraction class. Each virtual method is overloaded to implement system-specific behavior of the system as defined in the RALF description. No new methods are added to the system abstraction class.

As shown in Example 1-9 and Example 1-10, the system abstraction class contains a class property for each block and subsystem it contains. The name of the block or subsystem property is the name of the block or system. For blocks with multiple domains, the name of the blocks and subsystems are also available prefixed with the domain name.

Example 1-9 System Abstraction Class for Example A-14

```
class ral_sys_SoC extends uvm_reg_block;
   ral_block_uart uart0;
   ral_block_uart uart1;
   ...
endclass: ral_sys_SoC
```

Example 1-10 System Abstraction Class for Example A-15

```
class ral_sys_SoC extends uvm_reg_block;
   ral_block_uart uart0, ahb_uart0;
   ral_block_uart uart1, ahb_uart1,
   ral_block_bridge ahb_br;
   ral_block_bridge pci_br;
   ...
endclass: ral_sys_SoC
```

Inserting User-Defined Code Inside the Generated RAL Model Classes

You can insert user-defined code to the generated RAL model classes using the `user_code` construct, as follows,

```
user_code lang=sv [(new)]
{
  // Any kind/syntax of user code can be added here
}
```

If the optional (new) argument is specified, it indicates that the `user_code` be appended/inlined to/in the corresponding (RAL Model) class body of the build method. In case of a virtual register, the `user_code` shall be inlined in virtual register constructor.

If the new argument is not specified, user code will not be inlined in the build method code. By default, it will be located outside the constructor body (but inside the class), thus, opening up options of adding new data members, functions or task definitions and so on in the generated RAL model classes.

You can also use the `user_code` construct as an optional property in the definition or specification of a RALF register, virtual register, memory, regfile, block or system as shown in the example below.

Note:

> You can specify more than one RALF `user_code` in the definition or specification of a RALF register, virtual register, memory, regfile, block or system. The contents will be concatenated in the generated RAL model class in the same sequence as their corresponding occurrence in the RALF file.

Example 1-11   RALF Description with User Code

```
block b {
    bytes 1
    user_code lang=SV {
    // Any block level code can come here.
    }
    register r {
        bytes 1
        field f
        user_code lang=SV (new) {
     // Any register level build method code can come here.
        }
    }
}
```

This will generate the following RAL classes with `user_code` inlined appropriately.

```
class ral_reg_b_r extends uvm_reg;
        rand uvm_reg_field f;
        function new(string name = "b_r");
                super.new(name,
8,build_coverage(UVM_NO_COVERAGE));
        endfunction: new
   virtual function void build();
this.f =
uvm_reg_field::type_id::create("f",,get_full_name());
     this.f.configure(this, 1, 0, "RW", 0, 1'h0, 0, 0, 1);

                //////
                // User Code - Begin
                //////
              // Any register level build method code can
come here.
//////
                // User Code - End
                //////
   endfunction: build

        `uvm_object_utils(ral_reg_b_r)

endclass : ral_reg_b_r

class ral_block_b extends uvm_reg_block;
rand ral_reg_b_r r;
        rand uvm_reg_field r_f;
        rand uvm_reg_field f;

        //////
        // User Code - Begin
        //////
        // Any block level code can come here.
           //////

         // User Code - End
       //////
       function new(string name = "b");
                super.new(name,
build_coverage(UVM_NO_COVERAGE));
         endfunction: new

   virtual function void build();
      this.default_map = create_map("", 0, 1,
UVM_LITTLE_ENDIAN, 0);
```

```
      this.r =
ral_reg_b_r::type_id::create("r",,get_full_name()
);
      this.r.configure(this, null, "");
      this.r.build();
      this.default_map.add_reg(this.r,
`UVM_REG_ADDR_WIDTH'h0, "RW", 0);
                this.r_f = this.r.f;
                this.f = this.r.f;
   endfunction : build

        `uvm_object_utils(ral_block_b)
   endclass : ral_block_b
```

Arrays

If a system contains a block array or subsystem array, the class property for the block array or subsystem array is declared as a fixed sized array of the corresponding block abstraction class or system abstraction class.

Example 1-12   System Abstraction Class with Block Array

```
class ral_sys_SoC extends uvm_reg_block;
   ral_block_uart uart[2]
   ...
endclass: ral_sys_SoC
```

2

Register and Memory Specification

The Register Abstraction Layer File (RALF) is used to specify all the registers and memories in the design under verification. It is used to generate the object-oriented register and memory high-level abstraction layer. The first step in a project is to create a RALF description. Appendix - "RALF Syntax" contains detailed syntax and documentation for the RALF description.

As you add and modify fields, registers, and memories, you can update the RALF description many times during a project. You can then regenerate the abstraction layer multiple times without requiring modifications to the existing environment or tests.

Systems, Blocks, Registers, and Fields

In RAL, a design is a block or a system of blocks. The smallest functional unit that can be verified is a block. Systems are designs composed of blocks. Systems can also be composed of smaller systems of blocks, called subsystems.

There must be at least one block in a RALF description. The top-level construct describing the design under verification can be a `block` or `system` construct. The top-level block is identified when the RAL code is generated, therefore, a single RALF description may contain descriptions of multiple blocks and systems. The following example shows the RALF description of a design block:

Example 2-1 RALF Description of a Design Block

```
block blk_name {
   ...
}
```

Systems are composed of subsystems or blocks. Blocks are composed of registers, memories and sub-blocks. There can be no registers or memories directly in a system. If a design has system-wide registers or memories, they should be described in a block named, for example, `system_wide`. The following example shows the RALF description of a system:

Example 2-2 RALF Description of a System

```
system sys_name {
   ...
   block blk_name ...
   system subsys_name ...
}
```

Registers are composed of fields. Fields are concatenated to form a register, with optional unused bits between fields. A register must contain at least one field. The following example shows the RALF description of registers and memories in a block:

Example 2-3 RALF Description of Registers and Memories in a Block

```
block blk_name {
   ...
   register reg_name ...
   register reg_name ...
   ...
   memory mem_name ...
}
```

The field is the basic unit of the RAL. Fields are accessed atomically, independently of their location within a register or other fields. Therefore, fields can be moved within or across registers without having to modify the code that uses them. The following example shows the RALF description of fields in a register:

Example 2-4 RALF Description of Fields in a Register

```
register reg_name {
   ...
   field fld_name ...
   field fld_name ...
}
```

Reusability and Composition

RALF descriptions are intended to describe designs that can be arbitrarily combined and reused to create larger designs. There is no need for a RALF description of a block or subsystem to be aware of the context in which the block or subsystem is going to be used. In RALF descriptions, blocks and subsystems are described as stand-alone designs.

Although a RALF can describe an entire design inline, as in Example 2-5, a description can also instantiate blocks, registers and fields as required. The granularity of the description is arbitrary and you should plan for it to maximize reuse.

Example 2-5 Inlined RALF Description

```
system sys_name {
   ...
   block blk_name {
      ...
      register reg_name {
         ...
         field fld_name {
            ...
         }
      }
      ...
      memory mem_name {
         ...
      }
   }
}
```

RALF descriptions can include other RALF descriptions of smaller designs. Included descriptions can be reused and instantiated to compose the description of a larger design. The following example illustrates how this can be done:

Example 2-6 Hierarchical RALF Description

```
field fld_name {
   ...
}

register reg_name {
   ...
   field fld_name ;
}

memory mem_name {
   ...
}

block blk_name {
   ...
   register reg_name;
   memory mem_name;
}

system sys_name {
   ...
   block blk_name;
}
```

Naming

The names of fields, registers, memories, blocks, and systems are very important because these names are used to identify their corresponding abstraction class in the RAL abstraction model.

The following naming conventions apply to the names elements within a RALF description:

- Names must not be OV or SV reserved keywords

  These names are used as the name of abstraction classes in the generated OV or SV code. Therefore, they cannot be the same as reserved keywords in OV or SV.

- Field names should be unique within a block

Each block abstraction class contains a class property for each field contained in all of its registers, regardless of the specific register where it is located. If unique, the name of the field class property within the block abstraction class is the name of the field. In this case, fields can be moved within or across physical registers without affecting the verification environment or tests. Regardless of field name uniqueness, the block abstraction class contains another field class property referring to each field using the concatenation of the register and field name. See "Registers" for additional information.

Example 2-7 Field Class Properties in a Block Abstraction Class

```
block blk_name {
   register reg_name {
      field fld1;
      field fld2;
   }
   register xyz {
      field fld2;
   }
}

Yields:

class ral_block_blk_name extends uvm_reg_block;
   ...
   uvm_ral_field fld1, reg_name_fld1;
   uvm_ral_field reg_name_fld2;
   ...
   uvm_ral_field xyz_fld2
endclass
```

- Register names must be unique within a block and should be unique from field names.

  Each block abstraction class contains a class property for each register it contains. The name of the register class property within the block abstraction class is the name of the register and must, therefore, be unique and should be different from field names.

Example 2-8 Register Abstraction Classes in a Block Abstraction Class

```
block blk_name {
   register reg_name {
      field fld1;
      field fld2;
   }
}

Yields:

class ral_block_blk_name extends uvm_reg_block;
   ral_reg_blk_name_reg_name reg_name;
   uvm_ral_field             fld1, reg_name_fld1;
   uvm_ral_field             fld2, reg_name_fld2;
endclass
```

- Memory names must be unique within a block and unique from register names and should be unique from field names.

  Each block abstraction class contains a class property for each memory it contains. The name of the memory class property within the block abstraction class is the name of the memory and must, therefore, be unique and different from register names. It should also be different from field names.

Example 2-9 Memory Abstraction Classes in a Block Abstraction Class

```
block blk_name {
   register reg_name {
      field fld1;
      field fld2;
   }
   memory mem_name;
}

Yields:

class ral_block_blk_name extends uvm_reg_block;
   ral_reg_blk_name_reg_name reg_name;
   uvm_ral_field             fld1, reg_name_fld1;
   uvm_ral_field             fld2, reg_name_fld2;
   ral_mem_blk_name_mem_name mem_name;
```

```
endclass
```

- Block and subsystem names must be unique within a system.

  Each system abstraction class contains a class property for each block and subsystem it contains. The name of the block and subsystem class property within the system abstraction class is the name of the block or subsystem. Therefore, block and subsystem names must be unique.

- Independently defined names of registers, memories, blocks, and systems must be, respectively, globally unique within a RALF description.

  Each independently defined RALF element corresponds to a generated abstraction class in the RALF model (see "Understanding the Generated Model"). The names of these elements are used to generate the name of the corresponding class. Class names must be globally unique in SystemVerilog and OpenVera. Therefore, the names of independently defined registers, memories, blocks, and systems must be globally unique, otherwise they will generate identical abstraction class names.

  This requirement does not apply to elements defined inline within another definition.

Note:

Instantiated fields, registers, memories, blocks and subsystems can be renamed. With all of these naming requirements, it would be very difficult to have reusable RALF descriptions. Descriptions would need to know of their contexts to ensure uniqueness. Nor would it be possible to describe a design that contains multiple instances of the same block. Fortunately, any element of a RALF description can be renamed when instantiated to ensure uniqueness.

Example 2-10 Renaming RALF Elements

```
block blk_name {
   ...
}

system sys_name {
   ...
   block blk_name=blk1;
   block blk_name=blk2;
}
```

Hierarchical Descriptions and Composition

A RAL description can have independently specified registers, memories, blocks, and subsystems. You can instantiate these elements in higher level elements to create complete design descriptions.

When you specify registers, memories, blocks and subsystems, you also independently and explicitly specify their physical width as a number of bytes. Therefore, a block can be composed of registers and memories of smaller or larger width. Similarly, systems can be composed of blocks of smaller or larger width.

If you instantiate an element in a wider element, the value of the narrower element is justified to the least-significant bit and the most significant bits are padded with zero or truncated.

If you instantiate an element in a narrower element, the value of the wider element is split into the minimum number of narrower values.

You can specify splitting as:

- **Big Endian -** The most-significant bits are split into the lower addresses in the narrower address space. A 5-byte wide value of 0x1234567890 would be split into three 2-byte narrower values at increasing addresses in the following order: 0x0012, 0x3456 and 0x7890.
- **Little Endian -** The least-significant bits are split into the lower addresses in the narrower address space. A 5-byte wide value of 0x1234567890 would be split into three 2-byte narrower values at increasing addresses in the following order: 0x7890, 0x3456 and 0x0012.
- **Big FIFO -** All split values are accessed at the same physical address in the narrower address space. The most-significant bits are accessed first. A 5-byte wide value of 0x1234567890 would be split into three consecutive 2-byte narrower values at the same address in the following order: 0x0012, 0x3456 and 0x7890.
- **Little FIFO -** All split values are accessed at the same physical address in the narrower address space. The least-significant bits are accessed first. A 5-byte wide value of 0x1234567890 would be split into three consecutive 2-byte narrower values at the same address in the following order: 0x7890, 0x3456 and 0x0012.

Arrays and Register Files

Many designs have identical registers or groups of registers located in consecutive memory locations. These registers could be described by explicitly specifying each register, ignoring the fact that they are identical.

Example 2-11 Explicit specification of register arrays

```
register reg_name {
   ...
}
block blk_name {
   ...
   register reg_name=reg_0;
   register reg_name=reg_1;
   ...
   register reg_name=reg_7;
}
```

The repetitive process could be simplified by using the TCL *for-loop* command. Using the for-loop only simplifies the syntactical requirements of the specification. It does not change the RAL model that will be ultimately generated.

Example 2-12 Iterated explicit specification of register arrays

```
register reg_name {
   ...
}
block blk_name {
   ...
   for {set n 0} {$n < 8} {incr n} {
      register reg_name=reg_$n;
   }
}
```

The problem with explicitly enumerating consecutive registers is that they have unique names. It will not be possible to randomly index or iterate over their RAL model when writing SystemVerilog or OpenVera code that uses these consecutive registers.

Specifying consecutive registers using a register array will result in an array being available to be indexed or iterated on at runtime, not just at specification time. See "Arrays" for more details on the code generation process for arrays.

Example 2-13   Specification of register arrays

```
register reg_name {
   ...
}
block blk_name {
   ...
   register reg_name[8];
   register regX[5] {
      ...
   }
}
```

A sequence of register arrays will locate them in consecutive memory locations. For example, the specification in Example 2-13 will result in the following address map: `reg_name[0]`, `reg_name[1]`, `...` `reg_name[7]`, `regX[0]`, `regX[1]`, `...` `regX[4]`. If sequences of register groups, or interleaved register arrays are required, then you should a register file array. The specification in Example 2-14 will yield the following address map: `reg[0].reg_name`, `reg[0].X`, `reg[1].reg_name`, `...` `reg[4].reg_name`, `reg[4].X`.

Example 2-14   Specification of register file arrays

```
register reg_name {
   ...
}
block blk_name {
      ...
      regfile reg[5] {
         register reg_name;
         register X {
            ...
         }
      }
   }
```

Support for Different Reset Values in Register Arrays for RALF

RALF syntax supports the case where you have different reset values. The following property allows you to selectively change the reset value:

```
change_reset register_or_field_instance_to_reset new_reset_value
```

The register or field instance specified cannot be an array itself, it can only be an array element or a simple instance.

This property is supported inside a regfile or block to override the reset value for a specific register or field. This property can also be used for a simple register or field instance which is not an array element.

The following example shows the usage which effectively changes the reset value of the 11th element of the COUNTERS array:

```
block slave {
   register COUNTERS[256](COUNTERS[%d]) @'h0400 {
      field value {
         bits 32;
         access ro;
         reset 'h0;
      }
   }
change_reset COUNTERS[10] 'h1
}
```

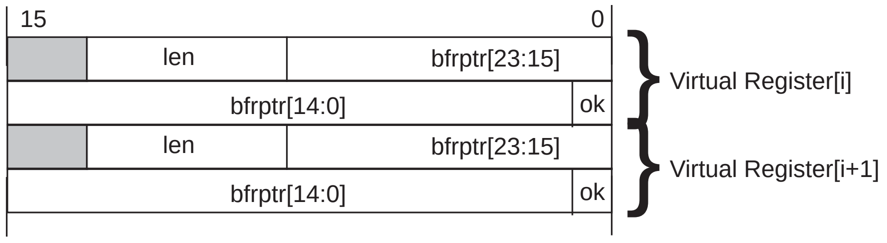
Virtual Fields and Virtual Registers

By default, fields and registers are assumed to be implemented in individual, dedicated hardware structures with a constant and permanent physical location such as a set of D flip-flops. In contrast, virtual fields and virtual registers are implemented in memory or RAM. Their physical location and layout is created by an agreement between the hardware and the software, not by their physical implementation.

Virtual fields and registers can be modelled using RAL by creating a logical overlay on a RAL memory model that can then be accessed as if they were real physical fields and registers. The RAL model of the memory itself remains available for directly accessing the raw memory without regard to any virtual structure it may contain.

Virtual fields define continuous bits in one or more memory locations and can span a memory location boundary. Virtual fields are contained in virtual registers. Virtual registers define continuous whole memory locations. They can span multiple memory locations but are always composed of entire memory locations, never fractions of memory locations.

Figure 2-1 Virtual Field and Virtual Register Structure



Virtual registers are always arrays because the usual reason they are virtual is that there are a large number of them and implementing them in a RAM instead of individual flip-flops is most efficient. Arrays of virtual registers are associated with a memory. The association of a virtual register array with a memory can be static (for example, specified in the RALF file) or dynamic (for example, specified at runtime through user code).

Static virtual registers are associated with a specific memory and are located at specific offsets within that memory. The association is specified in the RALF file and is created by the code generator. This association is permanent and cannot be broken at runtime.

Example 2-15   Static virtual register array

```
block MAC {
   ...
   memory DMABFRS { ... }
   ...
   virtual register CHANNEL[1024] DMABFRS@0 {
      field {...};
      ...
   }
}
```

Dynamic virtual registers are dynamically associated with a user-specified memory and are located at user-specified offsets within that memory at runtime. The dynamic allocation of virtual register arrays can also be performed randomly by a Memory Allocation Manager instance. The structure of the virtual registers is specified in the RALF file, but the number of virtual registers in the array and its association with a memory is specified in the SystemVerilog or OpenVera code and must be correctly implemented by the user. Dynamic virtual registers arrays can be relocated or resized at runtime.

Example 2-16   Dynamic virtual register specification

```
block MAC {
   ...
   memory DMABFRS { ... }
   ...
```

```
      virtual register CHANNEL {
         field {...};
         ...
      }
   }
```

Example 2-17   Implementing dynamic virtual registers

```
ral_model.MAC.CHANNEL.implement(1024,
                                ral_model.MAC.DMABFRS,
                                0);
```

Example 2-18   Randomly implementing dynamic virtual registers

```
ral_model.MAC.CHANNEL.allocate(1024,
                           ral_model.MAC.DMABFRS.mam);
```

Because virtual fields and virtual registers are implemented in memory, their content is not mirrored by the RAL model.

Multiple Physical Interfaces

Some designs may have more than one physical interface, each with accessible registers or memories. Some registers or memories may even be accessible via more than one physical interfaces and be shared.

A physical interface is called a domain. Only blocks and systems can have domains. Domains contain registers and memories. If a block or system has only one physical interface, there is no need to specify a domain for that interface.

For example, the block "bridge" shown in Example 2-19 specifies a block with two physical interfaces and a register accessible from both interfaces at offset 0 in their respective address spaces.

Example 2-19   Specification for a two-domain block

```
register xfer {
   bytes 4;
   field data {
      access rw;
   }
   shared (xfer_reg);
}

block bridge {
   domain apb {
      bytes 4;
      register xfer;
   }
   domain ahb {
      bytes 4;
      register xfer;
   }
}
```

Some physical interfaces may have different transactions used for configuration than the transactions used for normal operations. For example, PCI interfaces have `configuration write` transactions that are different from normal `write` transactions. Configuration transactions are typically used to set a base address and other decoding information required by normal transactions. Because configuration transactions are used separately from normal transactions, and normal transactions cannot occur until the DUT has been suitably configured using configuration transactions, configuration and normal transactions on the same physical interface must be modelled as separate physical interfaces.

Systems with multiple domains can instantiate blocks with a single domain. A domain must be entirely instantiated within a system domain, that is, a block-level or subsystem-level domain cannot be split between two system-level domains. Different block-level or subsystem-level domains can be instantiated in the same system-level domain but in different address offsets.

When instantiating a multiple-domain block or sub-system in a multiple-domain system, the same name and `hdl_path` must be used for all instances. This creates a single instance of the block or subsystem with its various domains instantiated in different domains.

Example 2-20 shows a specification of a multiple-domain instantiation. Notice how the same instance name "br" and HDL path are used in both cases. Example 2-21 shows the corresponding abstraction model of the system. Notice how domains do not create an additional abstraction scope.

Example 2-20 Instantiating a two-domain block in a two-domain system

```
system amba {
   domain apb {
      bytes 4;
      block bridge.apb=br (amba_bus.bridge);
   }
   domain ahb {
      bytes 4;
      block bridge.ahb=br (amba_bus.bridge);
   }
}
```

Example 2-21 Model of a two-domain block in a two-domain system

```
class ral_block_bridge extends uvm_reg_block;
   ral_reg_xfer xfer;
   ...
endclass

class ral_sys_amba extends uvm_reg_block;
   ral_block_bridge br;
   ...
endclass
```

Example 2-22 Instantiating a Domain Array Block

```
block blk {
   domain dom[8] {
      ...
   }
   ...
}

system sys {
      domain sys_dom[8] {
           bytes 4;
           block blk.dom[*]=blk @'h1000;
           ...
      }
      ...
}
```

Special Registers

The UVM register library presumes all registers and memories are average registers and memories, they are accessible at a known, constant, unique physical address(es), their behavior is constant throughout the simulation regardless of the physical interface used to access them, and they contain a single value.

Special register behavior can be modeled using any number of extension capabilities provided in the UVM register and field abstraction classes. Pre- and post-read/write callback objects, virtual callback methods, user-defined front-doors, and user-defined back-doors may be used to extend the behavior of the base library.

This section discusses the following topics that are supported by UVM Ralgen:

- "Indirect Indexed Registers"

- “Unimplemented Registers”
- “Aliased Registers”
- “Banked Registers”
- “Shared Registers”
- “Sparse Register Arrays”
- “Modeling Non-Standard Behavior of Registers”
- “Accessing Registers Based on Index in a regfile”

Indirect Indexed Registers

RALF provides support for indirect indexed registers. By default, the entire address space of registers and memories is assumed to be linearly mapped into the address space of the block that instantiates it. Each register or location in a memory corresponds to a unique address in the block.

However, some registers are not directly accessible through a dedicated address. Indirect access of an array of such registers is accomplished by first writing an "index" register with a value that specifies the array's offset, followed by a read or write of a "data" register to obtain or set the value for the register at that specified offset. This indexing mechanism allows to access a large memory in a limited address space.

In UVM-RAL, "data" register is modeled by extending the pre-defined `uvm_reg_indirect_data` class. The "data" register must not contain any fields. The "index" and "array" registers must be built first, as "index" and "array" registers are specified when the "data" register is configured using the `uvm_reg_indirect_data::configure()` method. The indirect register array, "index", and "data" registers are added as members of the containing block.

Since the registers in indirect indexed register array are not accessible via dedicated addresses, they are not added to the map. Only, the "index" and "data" registers are added to a map in the containing block.

For details on indirect indexed registers, see section "Indirect Indexed Registers" in *Universal Verification Methodology (UVM) 1.1 User's Guide*.

The following is the specification for each of the three registers used for implementing UVM-RAL Indirect Indexed Register functionality:

- **Indirect Index Register**

  Indirect Index register is specified like a regular register. It does not have a special syntax.

  ```
  register idx_reg … {
  …
  }
  ```

- **Indirect Array Register**

  Indirect Array register in UVM is not mapped to any address, hence the offset of an indirect array register needs to be mentioned as "none".

  For example:

  ```
  register array_reg @none {
    field  f1 {}
    …
  ```

```
}
```

- **Indirect Data Register**

  Indirect data register conveys the information regarding which registers will be used as an index register and unmapped array register. You can use the "`indirect_data`" property to provide this information. Also, an indirect data register shall not contain any fields. An error message is issued if this register contains any fields.

  In certain cases, you might have to index a group of registers that are essentially not register arrays. To index a non-array register, use the `regfile` instead of the register array as the `regfile` enables you to queue all the registers specified in the register file.

  Inside the regfile, the `regs` queue property contains all the register handles present inside this regfile. The order of specification of register in the register file determines the index location and this queue is used to index each of the register specified inside.

  Syntax for using indirect data registers:

  ```
  indirect_data
  <(indirect_reg[=arr_inst_name])|(indirect_regfile[=regf
  ile_inst_name])> indirect_idx=idx_inst_name;
  ```

  Example,

  ```
  register r1 {
  ...
  }

  register r2 {
  ...
  }
  ```

```
regfile rf {
        register r1=r1 @none;
        register r2=r2  @none;
}

register ind {
    field f {
        bits 8;
    }
}

system slave {
    bytes 4;

    block B1 @0  {
        bytes 4;
        regfile rf=rf1 @none;
        register ind=index @0x1;
        register data1 @0x0 {
            indirect_data rf=rf1 ind=index;
        }

    }
}
```

The generated SystemVerilog code includes the following `configure` call to use `regs` as indirect array:

```
this.data1.configure(this.index, this.rf1.regs, this,
null);
```

The *Indirect Array* and *Indirect Index* registers are extended from `uvm_reg` class. However, the *Indirect Data* register would be extended from the `uvm_reg_indirect_data` class.

For example:

```
class ral_b1_data_reg extends uvm_reg_indirect_data.
...
```

```
endclass
```

The call to configure method involves specifying *Indirect Array* and *Indirect Index* instances:

```
data_reg.configure(idx_reg, array_reg, this, null);
```

Apart from this, there will be no mapping done for indirect array register.

Limitations

The following are the limitations with this feature:

- Indirect_data rule is valid only while generating UVM code.
- An error will be indicated if an Indirect data register contains any fields.
- Indirect array register must be an array.
- Indirect index register must be a non-array register.

Limitations with regfile

- The register file pointed should be specified as `@none` (unmapped). All the registers inside this register file should be unmapped (either unspecified or as `@none`).
- Different regfile instances cannot be used as both indirect indexed array and normal register file.

Unimplemented Registers

Ralgen supports unimplemeted registers. The UVM register model can model registers that are specified but have not yet been implemented in the DUT. This allows the verification environment and testcases to make use of these registers before these are available.

Since these registers are unimplemented, there is nothing to actually read or write inside the DUT. The mirror in a register abstraction class provides a faithful model of the expected behavior of the respective register, thus, it can be used to provide a read back value. A yet-to-be-implemented register is thus modeled by writing to and reading from the mirror.

An unimplemented register can be modelled by providing a user-defined front-door and back-door that access the mirrored value instead of performing bus transactions.

RALF specification for un-implemented register:

```
{register regName @offset ?bankGroupName.bankModeName
?unimplemented …}
```

The unimplemented attribute can be used either at instantiation time or declaration time. In case it is used at declaration time, then all instances of that register will be treated as unimplemented.

Only one frontdoor and one backdoor extension is generated for a specification containing one or more unimplemented registers. The frontdoor and backdoors shall access the mirrored value instead of performing bus transactions.

Example 2-23

```
class ral_unimplemented_fd extends uvm_reg_frontdoor;
        virtual task body();
                uvm_reg R;
                $cast(R, rw_info.element);
             if(rw_info.kind == UVM_READ) rw_info.value[0]
= R.get();
                R.predict(rw_info.value[0], -1,
                        (rw_info.kind == UVM_READ)?
UVM_PREDICT_READ: UVM_PREDICT_WRITE,
                        rw_info.path, rw_info.map);
        endtask : body
endclass: ral_unimplemented_fd

class ral_unimplemented_bd extends uvm_reg_backdoor;
        virtual task read(uvm_reg_item rw);
                uvm_reg R;
                $cast(R, rw.element);
                do_pre_read(rw);
rw.value[0] = R.get();
             R.predict(rw.value[0], -1, UVM_PREDICT_READ,
rw.path, rw.map);
                do_post_read(rw);
        endtask : read
        virtual task write(uvm_reg_item rw);
                uvm_reg R;
                $cast(R, rw.element);
             R.predict(rw.value[0], -1, UVM_PREDICT_WRITE,
rw.path, rw.map);
        endtask : write
endclass: ral_unimplemented_bd
```

The user-defined frontdoor and backdoor shall be set during the build method of the corresponding block, since the environment is not accessible during generation.

```
virtual function void build();
   ...
   begin
     ral_unimplemented_fd R2_fd;
     ral_unimplemented_bd R2_bd;
```

```
      R2_fd = new;
      R2_bd = new;
      R2.set_frontdoor(R2_fd);
      R2.set_backdoor(R2_bd);
   end
endfunction
```

Note:

The “unimplemented” attribute shall be valid only when generating the UVM code. Ralgen issues an error message if you use the “unimplemented” attribute in RALF while generating VMM code.

Aliased Registers

Ralgen provides support for UVM Aliased registers. Aliased registers are accessible from multiple addresses in the same address map. These are different from shared registers as the latter are accessible from multiple address maps. Typically, the fields in aliased registers have different behavior depending on the address used to access them. For example, the fields in a register may be readable and writable when accessed using one address, but read-only when accessed from another address.

Modeling aliased registers in UVM involves more than simply mapping the same register at two different addresses.

In a UVM register model, each instance of a `uvm_reg` class must be mapped to a unique address in an address map and for aliased registers, a register class instance for each address is required. This is enabled using a specific register instance to access the aliased register via a specific address.

Each register instance must be of a register type that models the behavior of the register and the field it contains of its corresponding alias. For example, a register that contains a field that is RW when accessed via one address, but RO when accessed via another address would require two register types: one with a RW field and another with a RO field, and both using the same field names.

The aliasing functionality must be provided in a third class that links the two register type instances. The aliasing class can make use of pre-defined registers and field callback methods to implement the aliasing functionality. It may also make use of additional APIs or functionality created by the register model generator in the different register types that model each alias of the register. The aliasing class should be based on `uvm_object` to be factory-enabled. The required reference to the various register instance aliases shall be supplied via a `configure()` method.

RALF Extensions

In order to generate the corresponding alias framework, you need to specify the following:

- Grouping of registers which need to be aliased together
- Access policy of each of the field of these registers

The `alias` keyword is added as a block level property, as follows:

```
alias alias_name Reg1 Reg2 Reg3 ..
  {
    Reg1.F1 = Reg3.F1
    Reg2.F1 = Reg3.F1
  }
```

Where,

alias_name specifies the identifier for this alias.

Reg1 Reg2 Reg3 are the registers which require aliasing.

Field level mapping is specified within the curly brackets that allows the user to provide synchronization information. The field instance specifies the access policy.

For example, the user has 3 interfaces for a register, where first instance has `w1c` policy for a field, second instance has `w1s` policy for a field, and third instance has `ro` policy for the field. In case the user wants to update third register's field whenever a write operation is performed using first or second instance, mapping will allow to specify the pair of register fields to be synchronized. The callback based update mechanism is generated based on the policy of both the registers.

Example

To synchronize the `ro` field of a register with field `w1c` policy, whenever write operation is performed on the register.

When user specifies the alias property, an additional code will be generated. The generated code will contain the following:

- Extensions to the `uvm_reg_cbs` class which implements field level synchronization based on the access policy. The generated class will have the following prototype:

```
class <policy_one>_to_<policy_two>_update extends
uvm_reg_cbs;
    local uvm_reg_field m_toF;
    function new(uvm_reg_field toF);
        m_toF = toF;
    endfunction

   virtual function void post_predict(uvm_reg_field fld,
        uvm_reg_data_t previous,
        inout uvm_reg_data_t value,
        input uvm_predict_e kind,
        uvm_path_e path,
        uvm_reg_map map);
//Value transformation logic as per the policy pair
     endfunction
endclass
```

The following class will be generated if a field having `ro` access policy is synchronized with the filed having `w1c` policy:

```
class w1c_to_ro_update extends uvm_reg_cbs;
    local uvm_reg_field m_toF;
    function new(uvm_reg_field toF);
        m_toF = toF;
    endfunction

   virtual function void post_predict(uvm_reg_field fld,
        uvm_reg_data_t previous,
        inout uvm_reg_data_t value,
        input uvm_predict_e kind,
        uvm_path_e path,
        uvm_reg_map map);
        bit res;
        if (kind != UVM_PREDICT_WRITE) return;
        res = m_toF.predict(((~value) & previous), -1,
UVM_PREDICT_READ, path, map);
     endfunction
endclass
```

Only one class will be generated for each distinct 'access policy' pair. Further classes will only be generated for the policy pairs present in user's alias block.

For example, if it is required to alias a field having `w1c` policy to a field with `ro` policy, and there is another alias needed with the same policy pair (`w1c` for one field and `ro` for another field), then only one class will be generated and it will be instanced twice. Also, no other class will be generated corresponding to the other policy pair (since user's aliasing doesn't need them).

- A wrapper aliasing class which instantiates the synchronization class(es) described in the preceding step for each different type of register.

```
class ral_alias_<block_name>_<alias_name> extends
uvm_object;
    protected <register_type1> m_<register_name1>;
    protected <register_type2> m_<register_name2>;

`uvm_object_utils(ral_alias_<block_name>_<alias_name>)

    function new(string name = "<alias_name>");
        super.new(name);
    endfunction: new

    function void
configure(<register_type1><register_name1>,
<register_type2> <register_name2);
          <policy_one>_to_<policy_two>_update
<field_one>2<field_two>;

  //Other aliasing class instances as needed
    m_<register_name1> = <register_name1>;
    m_<register_name2> = <register_name2>;
    <register_name1>_<field_name1>_to_<register_name2>_
<field_name2>= new(<register_name2>.<field_name2>);
    uvm_reg_field_cb::add(<register_name1>.<field_name1
>,
<register_name1>_<field_name1>_to_<register_name2>_<fie
ld_name2>);

//Other instantiations and callback registration as
needed
endfunction : configure
endclass : ral_alias_<block_name>_<alias_name>
```

So, if two registers are aliased with fields named F in both the registers having access policies `w1c` and `ro` in the respective instances, the following class will be generated:

```
class ral_alias_b1_alias1 extends uvm_object;
```

```
    protected ral_reg_reg2_w1c m_reg2;
    protected ral_reg_reg3_ro m_reg3;
    `uvm_object_utils(ral_alias_b1_alias1)

    function new(string name = "alias1");
        super.new(name);
    endfunction: new

    function void configure(ral_reg_reg2_w1c reg2,
ral_reg_reg3_ro reg3);
          w1c_to_ro_update reg3_F_to_reg2_F;
          m_reg2 = reg2;
          m_reg3 = reg3;
          reg3_F_to_reg2_F = new(reg3.F);
        uvm_reg_field_cb::add(reg2.F, reg3_F_to_reg2_F);
    endfunction : configure
endclass : ral_alias_b1_alias1
```

- The corresponding block shall instantiate the aliasing class and glue the actual instances.

  So, for the application of synchronizing `ro` field with `w1c`, the following additional code is generated in the build method of the block:

```
      begin
          ral_alias_b1_alias1 alias1;
alias1 = ral_alias_b1_alias1::type_id::create("alias1",
get_full_name());
          alias1.configure(reg2, reg3);
      end
```

Limitations

- Aliasing is not supported with `-P` and `F` switches of ralgen.
- Aliasing currently requires one of the field of the field pair to have policy type `RO` or `RW`.

- Aliasing of field is not supported for field having policies of type `W1` or `WO1.`

Banked Registers

Ralgen provides support for Banked registers which are the registers that share a common address but are individually selected based on an external condition, such as the value of an external signal, or the value of a field in another register.

Some points to note about Banked Registers:

- Only one banked register can be selected at a given time.
- There can be more than two banked registers residing at the same address.
- It is possible that none of the banked registers are selected.
- Only front-door accesses are affected by the banking. Backdoor access to banked registers is not affected by the banking.
- If a banked register is accessed via the frontdoor and it is not currently selected,
    - An error will be issued.
    - The register model will not attempt to cause the accessed banked register to be selected.
    - The front-door access will access whatever banked register mapped at the same address is currently selected.
    - The register model mirror shall update the mirrored value of the banked register that is actually selected (if any).

- If a banked register is accessed via the backdoor and it is not currently selected,
  - No error or warning is issued.
  - The access is performed normally.
- The user is responsible to update the identity of the bank currently selected.
  - If the identity of the selected bank is different from the actual bank selection state in the DUT,

    -The register model may not be able to detect this condition.

    -The register model may not update the mirror values correctly.

    -No error or warning messages will be issued.

The register bank group must be specified using a `<snps:bankGroup>` element in a `<snps:memoryMap>` element in the `<spirit:memoryMap>` element. The name and a set of bank modes are then specified using the `<snps:name>` and `<snps:bankMode>` elements.

There can be more than one `bankGroup` in the same address map. The names of `bankGroups` within an address map must be mutually unique. A `bankGroup` is composed of two or more `bankModes`. The names of a `bankMode` must be unique within a `bankGroup`.

Example 2-24

```
<spirit:memoryMap>
   <spirit:vendorExtensions>
      <snps:memoryMap>
         <snps:bankGroup>
            <snps:name>bankGroupName</snps:name>
            <snps:bankMode>
               <snps:name>bankModeName</snps:name>
            </snps:bankMode>
         </snps:bankGroup>
      </snps:memoryMap>
   </spirit:vendorExtensions>
</spirit:memoryMap>
```

The Banked registers are associated with a specific mode by using a `<snps:bankGroupRef>` and a `<snps:bankModeRef>` element in the `<snps:register>` or `<snps:registerFile>` elements. The specified `bankGroupName` must be the name of a `bankGroup` defined in the address map containing the register or register file. The specified `bankModeName` must be the name of a `bankMode` defined in the specified `bankGroup.`

The register or register file will be accessible only if the specified `bankGroup` is currently in the specified `bankMode`. The method to specify a currently active `bankMode` of a specific `bankGroup` at run-time is outside the scope of IP-XACT.

If a register file is specified as banked, all of the registers and register files it contains are similarly banked. It is an error to have an explicitly banked register or register file in a banked register file.

Example 2-25

```
<spirit:registerFile>
   <spirit:vendorExtensions>
      <snps:registerFile>
         <snps:bankGroupRef>
            bankGroupName
         </snps:bankGroupRef>
         <snps:bankModeRef>bankModeName</snps:bankModeRef>
      </snps:registerFile>
   </spirit:vendorExtensions>
</spirit:registerFile>
<spirit:register>
   <spirit:vendorExtensions>
      <snps:register>
         <snps:bankGroupRef>
```

```
            bankGroupName
         </snps:bankGroupRef>
         <snps:bankModeRef>bankModeName</snps:bankModeRef>
      </snps:register>
   </spirit:vendorExtensions>
</spirit:register>
```

To specify register banking in the RALF description, the following additions are required:

- In a single-domain block or a domain element:

  ```
  {bankgroup bankGroupName bankModeName bankModeName …}
  ```

- In a register and register file instantiations:

  ```
  {register regName @offset ?bankGroupName.bankModeName …}
  ```

Use Model

The use of banked registers in a UVM register model will require the use of following UVM library classes:

- **snps_uvm_reg_bank_group**

  This class specifies the modes available in a `bankGroup` and the current mode in which the `bankGroup` is present. The extensions of this class specify a `bankGroup` type, with all of the valid `bankModes` in that group.

  All instances of this class are put in the `uvm_resource_DB#(snps_uvm_reg_bank_group)` resource database under the full hierarchical name of the group.

The user is responsible for retrieving the appropriate instance of this class from the resource database and calling a method on it to specify the current `bankMode` (whenever it changes). By default, a `bankGroup` is in its first declared `bankMode`.

- **snps_uvm_reg_banked**

  This class is an extension of the `uvm_reg` class for banked registers. Banked registers are modeled by extending `snps_uvm_reg_banked` class instead of the `uvm_reg` class.

  The `configure()` method of this class specifies the `bankMode` that must be selected in the specified `bankGroup` class instance for the banked register to be selected. It also returns the content of banked registers mapped to the same address.

- **snps_uvm_reg_bank_set**

  This class is an extension of the `uvm_reg` class that encapsulates all the banked registers mapped at the same address.

- **snps_uvm_reg_predictor**

  This class is a modified version of `uvm_reg_predictor::write()` to handle the case where `uvm_reg_map::get_reg_by_offset()` returns an instance of `snps_uvm_reg_bank_set` instead of a `uvm_reg`.

  Instances of this predictor must be used instead of the `uvm_reg_predictor` class in environments where banked registers are used.

- **snps_uvm_reg_map**

This class detects the banked registers mapped to the same physical address and wrap them in an instance of `snps_uvm_reg_bank_set` class. All registers mapped to the same physical address must be banked registers and must be in the same `bankGroup` and be mutually exclusively selected.

Shared Registers

Shared registers are accessible through more than one physical interfaces and be shared. You can specify the access policy while instantiating a shared register, as shown in the following example:

```
register data_xfer {
    bytes 4;
    field data {
        bits 32;
    }
    shared;
}

block bridge {
    domain pci {
        bytes 4;
        register data_xfer write;
    }
    domain  ahb {
bytes 4;
        register data_xfer write;
  }
}
```

Also, while generating backdoor paths for DPI-based tasks, generation of the `add_hdl_path` calls is supported for shared registers.

So, if in the above example, `hdl_path` is specified in register declaration as follows,

```
register data_xfer {
    bytes 4;
    field data {
        bits 32;
    }
    shared (data_xfer);
}
```

The generated code contains `add_hdl_path` as follows,

```
this.data_xfer.add_hdl_path
('{
   '{"data_xfer", -1, -1}
});
```

Sparse Register Arrays

Ralgen provides support to specify sparse register arrays in RALF. Sparse register arrays help you to optimize the performance when a large register array is present in RTL, but only some of the elements are accessed during simulation.

In case of a simple register array, register instances are created corresponding to each of the element in the array when register model is built. If only some of the locations are accessed in a large array, it may not be desirable to allocate instances for all the locations. A sparse register array addresses this need and does not create instances upfront. It creates register instances dynamically during the simulation based on the locations that are accessed.

For example, if only ten locations from a register array of size 1024 are accessed during simulation, the sparse register array only creates ten instances during the course of simulation.

Use Model

To specify to ralgen that sparse register array model needs to be generated for a particular register array, the `*sparse*` keyword must be specified in the first dimension as follows:

```
register sp_arr[100 *sparse*];
```

With the above specification, ralgen generates the RAL model where `sp_arr` is an object of a class extended from `uvm_reg_array` instead of `uvm_reg`. The instantiation and binding to the rest of the model is taken care by ralgen.

To compile the generated code along with the testbench, you can use `+define+UVM_SPARSE_ARRAY` that includes the base code changes necessary for the sparse array implementation.

To access an element in `UVM_SPARSE_ARRAY`, the read/write calls need to be executed along with the index specification. For example, to write to the third element of `sp_arr`, the write call is as follows:

```
model.sp_arr.write(status, 2, data1, .parent(this));
```

Note:

To use sparse register array functionality, UVM library shipped with VCS needs to be used.

Limitations

The following are the limitations with this feature:

- Only frontdoor access is supported, backdoor access is not supported.
- Coverage model is not supported.

- Sparse register array is supported only for simple registers. Special registers like indirect registers are not supported.

Modeling Non-Standard Behavior of Registers

To model a non-standard behavior of a register, you can embed the callback classes in the generated `ralgen` code. This enables you to easily embed the non-standard register behavior and reduce the changes needed to the generated model. The syntax is a follows:

```
register_cb_class <class_name> {
  var_declarations {
  }
  new_method (args) {
  }
  pre_read_method {
  }
  post_read_method {
  }
  pre_write_method {
  }
  post_write_method {
  }
}
```

This results in generating the callback class extending from `uvm_reg_cb` with the code placed in appropriate methods in the SystemVerilog code as follows:

```
class <class_name> extends uvm_reg_cb {
    function new (…);
    endfunction
    }

   function void pre_write (…);
  endfunction
  …
endclass
```

These declarations are allowed only at global level and you cannot declare them inside systems or blocks. Furthermore, to ease the addition of callback, the following rules are applicable inside the register/regfile/block definition.

- In the register definition, the syntax to add callback is as follows:

```
add_reg_cb <cb class name> "(" new_method_args ")".
```

- Inside the block and the regfile definition, you can attach the callback to a particular register instance as follows:

```
add_reg_cb <register instance name> <cb class name> "("
new_method_args ")".
```

The generated code of the corresponding class inside a build is as follows:

```
begin
    <cb class name> reg_cb = new(new_method_args);
    uvm_reg_cb::add(<register instance name>, reg_cb);
end
```

Note:

- In the case of callback addition specified inside the register definition, 'this' keyword will be used as `<register_instance_name>`
- Ralgen automatically adds the argument, `string name=<callback_class_name>` to the method and calls `super.new(name)` as the first implementation.

Accessing Registers Based on Index in a regfile

A regfile is generated with an additional property to queue all the registers present in the regfile. The regfile extended class has the following property declared in it:

```
 uvm_reg regs[$];
```

You can use this queue to access registers based on the index.

For example,

```
model.block.regfile.regs[0].write(.status(status),
.value('habcd_1234), .path(UVM_FRONTDOOR), .parent(this));
```

Each scalar register is added to the queue. In case of a register array, individual register of the array is added.

3

Generated Back-doors

Automatically generated back-door mechanisms are associated with their corresponding register or memory abstraction class when the RAL model containing these registers and memories is instantiated. However, in order to enable the automatic generation of back-door access, it is necessary to specify the hierarchical path to the HDL structures that implement the register or memory. This is accomplished by using the `hdl_path` attributes in “field” , “register” , “regfile” , “memory” , “block” and “system” instantiations of the RALF specification.

The generated backdoor simply concatenates the path elements specified in the individual `hdl_path` attributes to form the complete path to the target register or memory. For example, the RALF file shown in Example 3-1 would yield the path `S1_TOP_PATH.b1_i.dec.r1_reg` to the register `r1`.

Example 3-1   RALF Description with hdl_path Specifications

```
system s1 {
   ...
   block b1 (b1_i) @'h1000 {
     ...
      register r1 dec.r1_reg {
         ...
      }
   }
}
```

For a path to be well-formed, a RALF “regfile” , “block” or “system” must correspond to a design module or entity instance. For example, the (partial) RTL code shown in Example 3-2 represents the structure of the design matching the specification in Example 3-1.

Example 3-2   RTL Structure

```
module b1(...);
   ...
   always @ (posedge clk)
   begin: dec
      reg [7:0] r1_reg;
      if (rst) r1_reg <= 0;
      else if (...) r1_reg <= ...;
   end
   ...
endmodule

module s1(...);
   ...
   b1 b1_i(...);
   ...
endmodule

module tb_top;
   ...
   s1 dut(...);
   ...
endmodule
```

The absolute path to the instance of the DUT that corresponds to the RAL model is specified by defining the `name_TOP_PATH` symbol where `name` is the uppercase name of the top-level block or system in the RAL model. Using the structure shown in Example 3-2, the `S1_TOP_PATH` symbol must be defined to `tb_top.dut`, as shown in the following:

```
% vcs ... +define+S1_TOP_PATH=tb_top.dut ... \
      ral_s1.sv ...
```

Arrays

If the RALF specification contains arrays of “system” , “block” , “regfile” , “register” or “field” instances (see “Arrays and Register Files” for more details on arrays of instances), the `hdl_path` attribute must contain a `%d, [%d]` or `[%g]` format specifier.

Example 3-3 shows the key differences between them.

Example 3-3   A RALF Description Using Different Types of Array Backdoor

```
system s1 {
   bytes 1
   block b1[2] (b1_i%d) {
      bytes 1
      register r1 (dec.r1_reg) {
         field f
      }
   }
   block b2 (blk2) {
      bytes 1
      register r2[2] (r2_array[%d]) {
         field f
      }
   }
   block b3[2] (b3_gen_array[%g].blk) {
      bytes 1
```

```
         register r3 (dec.r3_reg) {
            field f
         }
      }
   }
```

%d Format Specifier

You should use the`%d` format specifier when the corresponding backdoor RTL implementation of the RALF array is not really an array, rather a series of similarly named non-array signals, for example, block `b1` array in Example 3-3.

Example 3-4 shows that the generated backdoor path of such an RALF array does not have any array in it.

Example 3-4 Generated Code During the %d Format Specifier Usage

```
class ral_reg_s1_b1_r1_bkdr extends
uvm_ral_reg_backdoor;
   int b1;

   function new(string name);
      super.new(name);
      this.b1 = b1;
   endfunction

   virtual task read(uvm_reg_item rw);
      do_pre_read(rw);
      case (b1)
         0:
           Rw.value[0]= `S1_TOP_PATH.b1_i0.dec.r1_reg;;
         1:
           Rw.value[0]= `S1_TOP_PATH.b1_i1.dec.r1_reg;;
      endcase
      rw.status = UVM_IS_OK;
      do_post_read(rw);
   endtask
```

```
      virtual task write(uvm_reg_item rw);
         do_pre_write(rw);
         case (b1)
           0: `S1_TOP_PATH.b1_i0.dec.r1_reg = rw.value[0];
           1: `S1_TOP_PATH.b1_i1.dec.r1_reg = rw.value[0];
         endcase
         rw.status = UVM_IS_OK;
         do_post_write(rw);
      endtask
   endclass
```

An example of the RTL implementation of a RALF array, which is not an array in the backdoor/RTL, rather a series of similarly named non-array signals is as follows:

```
   module s1(...);
      ...
      b1 b1_i0(...);
      b1 b1_i1(...);
      ...
   endmodule
```

[%d] Format Specifier

You should use the `[%d]` format specifier only when the end signal/ variable of the corresponding backdoor RTL path/implementation is actually an array, but not a generated instance array. For example, block `r2` as shown in Example 3-3.

The advantage of having a normal array instead of generated instance array is that you can access an each array element by indexing with a variable as shown in Example 3-5.

Example 3-5 Generated Code During the [%d] Format Specifier Usage

```
   class ral_reg_s1_b2_r2_bkdr extends
   uvm_ral_reg_backdoor;
```

```
      int r2;

      function new( string name);
         super.new(name);
         this.r2 = r2;
      endfunction

      );
    virtual task read(uvm_reg_item rw);
         do_pre_read(rw);
         rw.value[0] = `S1_TOP_PATH.blk2.r2_array[r2];
         rw.status = UVM_IS_OK;
         do_post_read(rw);
      endtask


         virtual task write(uvm_reg_item rw);
         `S1_TOP_PATH.blk2.r2_array[r2] = rw.value[0];
         rw.status = UVM_IS_OK;
      endtask
   endclass
```

[%d:%d] Format Specifier

You can use the `[%d:%d]` format specifier to specify the register arrays modeled in RTL using a single-dimension packed array in the UVM flow. Ralgen considers `[%d:%d]` as an indication that the corresponding representation in RTL is of a packed register. It generates the `add_hdl_path` as follows:

```
this.r[J].add_hdl_path('{
                       '{$psprintf("r[%0d:%0d]",
                       J*(size_of_register),
                       (J*size_of_register)+
                       (size_of_register-1)), -1, -1}
                      })
```

The backdoor paths are changed accordingly. Consider the following RALF code:

```
block bar {
  bytes 1;
  register r[4] hdl_path=(r1[%d:%d]) {

    backdoor_xor_mask 'hff

    field f1 {
      bits 4;
      reset 0;
      access rw;
    }

    field f2[4]  {
      bits 1;
      reset 0;
      access rw;
    }
  }
}
system foo {
  bytes 1;
  block bar (bar);
}
```

For this, the XMR based backdoor paths are generated as follows,

```
virtual task read(uvm_reg_item rw);
        do_pre_read(rw);
      case (r)
      0:  rw.value[0] = `FOO_TOP_PATH.bar.r1[0:7];
      1:  rw.value[0] = `FOO_TOP_PATH.bar.r1[8:15];
      2:  rw.value[0] = `FOO_TOP_PATH.bar.r1[16:23];
      3:  rw.value[0] = `FOO_TOP_PATH.bar.r1[24:31];
      endcase

      rw.value[0] = rw.value[0] ^ 'hff;
      rw.status = UVM_IS_OK;
      do_post_read(rw);
endtask
```

```
virtual task write(uvm_reg_item rw);
      rw.value[0] = rw.value[0] ^ 'hff;
      do_pre_write(rw);
      case (r)
      0: `FOO_TOP_PATH.bar.r1[0:7]   = rw.value[0];
      1: `FOO_TOP_PATH.bar.r1[8:15]  = rw.value[0];
      2: `FOO_TOP_PATH.bar.r1[16:23] = rw.value[0];
      3: `FOO_TOP_PATH.bar.r1[24:31] = rw.value[0];
        endcase

      rw.status = UVM_IS_OK;
      do_post_write(rw);
endtask
```

The corresponding `add_hdl_path` will be as follows,

```
this.r[J].add_hdl_path('{'{$psprintf("r1[%0d:%0d]", J*8,
J*8+7), -1, -1}});
```

[%g] Format Specifier

Use the `[%g]` format specifier when the corresponding backdoor RTL implementation of the RALF array is a generated instance array. For example, block `b3` array, as shown in Example 3-6.

Note:

Because the generated instance array cannot be indexed using any variable, numeric constants are used for indexing them.

Example 3-6 Generated Code During the [%g] Format specifier Usage

```
class ral_reg_s1_b3_r3_bkdr extends
uvm_ral_reg_backdoor;
   int b3;

   function new(string name);
      super.new(name);
      this.b3 = b3;
   endfunction
```

```
      virtual task read(uvm_reg_item rw);
      do_pre_read(rw);
      case (b3)
         0: rw.value[0] =
`S1_TOP_PATH.b3_gen_array[0].blk.dec.r3_reg;
         1: rw.value[0] =
`S1_TOP_PATH.b3_gen_array[1].blk.dec.r3_reg;
      endcase
      rw.status = UVM_IS_OK;
   endtask


      virtual task write(uvm_reg_item rw);
      case (b3)
        0: `S1_TOP_PATH.b3_gen_array[0].blk.dec.r3_reg
= rw.value[0];
        1: `S1_TOP_PATH.b3_gen_array[1].blk.dec.r3_reg
= rw.value[0];
      endcase
      rw.status = UVM_IS_OK;
   endtask
endclass
```

You should use the `[%g]` format specifier when the backdoor RTL path has an array which is not the end signal/variable of that backdoor RTL path. For example, block `b3` array shown in Example 3-6.

Note:

You cannot index an XMR using variable, unless it is an end signal/variable. In this case you use numeric constants to index them.

Backdoor Support for VHDL or Mixed Language Designs

For UVM, XMR based backdoor access generation is supported for VHDL designs, or registers or memories located in a portion of the design where the end signal has been implemented or described using VHDL.

The XMR based backdoor aids in better simulation performance as it avoids having to provide read or write capabilities on the design hierarchy for enabling DPI based accesses. You need to use `vhdl_path` to specify backdoor RTL paths ending in a VHDL signal or a variable.

Example 3-7 RAL Backdoor Support for VHDL Signals

```
system top {
  bytes 1
  block blk vhdl_path=(dut_blk) {
    bytes 1;
    register reg1 vhdl_path=(reg1) {
      bytes 1;
      field f vhdl_path=(f) {
        bits 8;
      }
    }
  }
}
```

In Example 3-7, the final synthesized or concatenated backdoor access path for register `reg1` is `dut_blk:reg1`. For a given RALF construct, you can either specify a `vhdl_path` or a verilog `hdl_path` (not both).

If the RALF description of the top-level block or system has `vhdl_path` specified for it or any of its descendents and ralgen command line options `-b` and `-top_path` are used, then ralgen will create a `ral_vhdl_bkdrs_<top>.v` file. The ralgen generated file, `ral_vhdl_bkdrs_<top>.v` will have the definition of a module called `ral_<top>_vhdl_bkdr_connector`.

For each VHDL signal for which backdoor access path needs to be generated using `$hdl_xmr,` this module will have the corresponding Verilog side `reg` definition (of appropriate size) for use or mapping with `$hdl_xmr` and an initial block which would have the corresponding `$hdl_xmr` connection commands.

For a read/write VHDL implemented register, one Verilog register definition for reading and one for writing needs to be created. These intermediate Verilog registers are named after their corresponding synthesized/concatenated hierarchical/absolute `(v)hdl_path` names only after replacing ':' or '.' with '__' and adding a '__' prefix to it. To indicate the direction of data flow or assignment, a '__ip' or '__op' suffix is also added.

An example of a module for the RALF described above is shown in Example 3-8.

Example 3-8 Generated Connector Module

```
module ral_top_vhdl_bkdr_connector;
        reg[6:0] __dut_blk__reg1__f__ip;
        reg[6:0] __dut_blk__reg1__f__op;

        initial begin
                $hdl_xmr("<top_path>.dut_blk.reg1.f",
"__dut_blk__reg1__f__ip");
                $hdl_xmr("__dut_blk__reg1__f__op",
"<top_path>.dut_blk.reg1.f");
                $hdl_xmr("<top_path>.dut_blk.reg1.f",
"__dut_blk__reg1__f__op");
```

```
end
endmodule
```

Here, the `<top_path>` is the absolute top-level XMR path of the VHDL design instance which must be provided with the new ralgen command line option `-top_path` or `-p`. For generating RAL VHDL backdoor code, you should specify this option apart from `-b` ralgen command line option.

Take an instance of this ralgen generated module and specify the absolute top-level XMR path of this instance in the compile time macro `<top>_VHDL_BKDR_CONNECTOR`. By doing this, the RAL backdoor implementation classes can access the VHDL DUT registers or memories through this connection module instance as shown in Example 3-9.

Example 3-9   Backdoor Connection

```
class ral_reg_top_blk_reg1_bkdr extends uvm_reg_backdoor;

        function new(string name);
                super.new(name);
        endfunction
        virtual task read(uvm_reg_item rw);
                wait (this.b2b == 0);
                do_pre_read(rw);
                begin
                     rw.value[0] = `UVM_REG_DATA_WIDTH'h0;
                               rw.value[0][6:0] =
`TOP_VHDL_BKDR_CONNECTOR.__dut_blk__reg1__f__ip;
                       end
                rw.status = UVM_IS_OK;
                do_post_read(rw);
        endtask

        virtual task write(uvm_reg_item rw);
                do_pre_write(rw);
                begin
`TOP_VHDL_BKDR_CONNECTOR.__dut_blk__reg1__f__op =
rw.value[0][6:0];
                end
                rw.status = UVM_IS_OK;
                do_post_write(rw);
        endtask

        local int b2b;
endclass
```

Scoping Backdoor Classes to a SV Package

Ralgen currently provides two options for generating register/ backdoor memories.

- Cross-module reference based backdoors: ralgen option `-b` generates a set of classes with `write()` and `read()` APIs to access register/memory constructs using cross-module references (XMRs).
- DPI-C based: The default backdoor option with ralgen. Here DPI-C is used to deposit or read values from register/memory constructs.

  The generated backdoor classes with the first option cannot be packaged into a SystemVerilog package as the HDL XMRs are a part of the generated classes. The backdoors generated using second option can be packaged into a SystemVerilog package but in this case, the check for the existence of backdoor paths happens during simulation. This enhancement aims to generate a backdoor infrastructure that brings the best of both the above options using a virtual interface instance to access the HDL XMRs.

Command-Line Option

`-gen_vif_bkdr`

When specified, helps the generated model to have a register model free of HDL XMRs in its backdoor classes.

Specification

- It is necessary to specify the `-b` option when specifying the `-gen_vif_bkdr` option.
- A SystemVerilog interface is generated (in a separate file) additional to the register model itself which you need to pass to the compile command-line. Additional setup is not required.
- It is necessary to specify `-gen_vif_bkdr` when `-b` is specified with `-P`.

Example

The following (partial) RALF file:

```
block host_regmodel {
  bytes 2;
  register HOST_ID (host_id)   @'h0000;
}
```

Generates the following interface with tasks to access HDL XMRs and register backdoor classes (partial) calling these functions using a virtual interface handle:

ral_host_reg_model_interface.sv:

```
interface ral_host_regmodel_intf;
```

```
     import uvm_pkg::*;

     initial
uvm_resource_db#(virtual ral_host_regmodel_intf)::set("*",
"uvm_reg_bkdr_if", interface::self());

     task ral_host_regmodel_HOST_ID_bkdr_read(uvm_reg_item
rw, int index = 0);
          rw.value[0] = `HOST_REGMODEL_TOP_PATH.host_id;
     endtask
     task
ral_host_regmodel_HOST_ID_bkdr_write(uvm_reg_item rw, int
index = 0);
          `HOST_REGMODEL_TOP_PATH.host_id[31:0] =
rw.value[0][31:0];
     endtask
endinterface
```

ral_host_reg_model.sv:

```
class ral_reg_host_regmodel_HOST_ID_bkdr extends
uvm_reg_backdoor;
     virtual ral_host_regmodel_intf __reg_vif;
     function new(string name);
          super.new(name);
          uvm_resource_db#(virtual
ral_host_regmodel_intf)::read_by_name(get_full_name(),
"uvm_reg_bkdr_if", __reg_vif);
     endfunction

     virtual task read(uvm_reg_item rw);
          do_pre_read(rw);

__reg_vif.ral_host_regmodel_HOST_ID_bkdr_read(rw);
          rw.status = UVM_IS_OK;
          do_post_read(rw);
     endtask

     virtual task write(uvm_reg_item rw);
          do_pre_write(rw);

__reg_vif.ral_host_regmodel_HOST_ID_bkdr_write(rw);
          rw.status = UVM_IS_OK;
```

```
            do_post_write(rw);
        endtask
    endclass
```

Target Structures

The automatically-generated back-door access code must make certain assumptions about the nature of the HDL code used to implement the register and memory being accessed.

Although there are almost unlimited ways you can implement a register, there are only a few styles that are supported by the back-door access generator. It is important that, when implementing registers and memories in RTL code, a suitable coding style be used.

The following guidelines outline the restrictions on RTL structures used to implement registers and memories to enable automatic generation of their back-door access. Some of these restrictions may be removed in the future as the capabilities of the back-door access generator are improved.

If the target structures do not meet the requirements for automatic generation of back-door access, a user-defined back-door access mechanism must be created.

Writable Fields and Memories Must be Implemented Using "reg"

When performing a back-door write operation, a blocking procedural assignment is used. This requires that the target of the assignment be a *reg*.

Read-Only Fields May be Implemented Using Wire, Parameter or Boolean Expression

Such structures cannot be written to, therefore, only the read back-door access to a read-only field is generated. Attempting a back-door write to a read-only field will result in an error.

Example 3-10 Read-only Field Implemented Using an Expression

```
always @ (*)
begin
   if (wr) rdat = ’Z;
   else case (addr)
     ...
     16’h0010: rdat = {fifo_fl, fifo_mt};
     ...
   endcase
end
```

Example 3-11 RALF Description for Read-only Field

```
register r1 @’h0010{
   bytes 2;
   field mt (fifo_mt) {
      bits   1;
      reset  1;
      access ro;
   }
   field fl (fifo_fl) {
      bits   1;
      reset  0;
      access ro;
   }
}
```

Example 3-12 Alternative RALF Description for Read-only Field

```
register r1 (fifo_fl, fifo_mt) @’h0010{
   bytes 2;
   field mt {
      bits   1;
      reset  1;
   }
   field fl {
      bits   1;
      reset  0;
   }
}
```

A Register May Implement All of its Fields in a Single "reg"

A register may be composed of more than one field. All these different fields may be implemented in the same *reg* that implements the overall register.

This implies that all bits in the register, up to the most-significant bits of the most-significant field, are implemented and there are no reserved or unused bits between fields. In that case, no *hdl_path* should be specified in field instantiations in the register specification.

For example, the register specified using the *register* definition shown in Example 3-13, can be implemented using the RTL code shown in Example 3-14. The *reg* named `r1_reg` is used to implement fields `f1` and `f2`.

Example 3-13   Register with Multiple Fields

```
register r1 (r1_reg) @'h0010{
   bytes 2;
   field f1 {
      bits   4;
      reset  4'hA;
   }
   field f2 {
      bits   8;
      reset  8'h55;
   }
}
```

Example 3-14   Single-reg Implementation of Register with Multiple Fields

```
reg [11:0] r1_reg;
always @ (posedge clk)
begin
   if (rst) r1_reg <= {8'h55, 4'hA};
   else if (wr) case (addr)
      ...
      16'h0010: r1_reg <= wdat;
      ...
   endcase
end

always @ (*)
begin
   if (wr) rdat = 'Z;
   else case (addr)
     ...
     16'h0010: rdat = r1_reg;
     ...
   endcase
end
```

If per-field peek()/poke() operations are required (not yet supported), each field instance should have its respective bit slice specified in its *hdl_path* attribute. For example, the register specified using the *register* definition shown in Example 3-15, can also be implemented using the RTL code shown in Example 3-14.

Example 3-15   Register with Multiple Fields

```
register r1 @'h0010{
   bytes 2;
   field f1 (r1_reg[3:0]) {
      bits   4;
      reset  4'hA;
   }
   field f2 (r1_reg[11:4]) {
      bits   8;
      reset  8'h55;
   }
}
```

A Register May Implement its Fields in Separate "reg"

A register may be composed of more than one field. All these different fields may be implemented in different *regs* that each implement one field. The register is the concatenation of these individual *regs*. This implementation allows reserved or unused bits between fields. In that case, the *hdl_path* must be specified in field instantiations in the register specification.

For example, the register specified using the *register* definition shown in Example 3-16, can be implemented using the RTL code shown in Example 3-17. The *regs* named `f1_reg` and `f2_reg` are used to implement fields `f1` and `f2` respectively. Additionally, both Example 3-10 and Example 3-11 show an example of a register implemented using separate constructs for separate read-only fields.

Example 3-16 Register with Multiple Fields

```
register r1 @'h0010{
   bytes 2;
   field f1 (f1_reg) {
      bits   4;
      reset  4'hA;
   }
   field f2 (f2_reg) @8 {
      bits   4;
      reset  4'h5;
   }
}
```

Example 3-17 Multiple-reg Implementation of Register with Multiple Fields

```
reg [3:0] f1_reg, f2_reg;
always @ (posedge clk)
begin
   if (rst) begin
      f1_reg <= 4'hA;
      f2_reg <= 4'h5};
   end
   else if (wr) case (addr)
      ...
      16'h0010: begin
         f1_reg <= wdat[3:0];
         f2_reg <= wdat[11:8];
      end
      ...
   endcase
```

```
end

always @ (*)
begin
   if (wr) rdat = 'Z;
   else case (addr)
     ...
     16'h0010: rdat = {f2_reg, 4'h0, f1_reg};
     ...
   endcase
end
```

A Field May be Implemented Using Multiple "reg"

Like registers, a field may be implemented as separate *`regs`*. For example, the register specified using the *`register`* definition shown in Example 3-18, can be implemented using the RTL code shown in Example 3-19. The *`regs`* named `f2a_reg` and `f2b_reg` are used to implement field `f2`.

Example 3-18 Field Implemented with Multiple regs

```
register r1 @'h0010{
   bytes 2;
   field f1 (f1_reg) {
      bits   4;
      reset  4'hA;
   }
   field f2 (f2a_reg, f2b_reg) @8 {
      bits   4;
      reset  4'h5;
   }
}
```

Example 3-19 Multiple-reg Implementation of a Fields

```
reg [3:0] f1_reg, f2a_reg, f2b_reg;
always @ (posedge clk)
begin
   if (rst) begin
      f1_reg <= 4'hA;
      {f2a_reg, f2b_reg} <= 4'h55};
   end
   else if (wr) case (addr)
      ...
      16'h0010: begin
         f1_reg <= wdat[3:0];
         {f2a_reg, f2b_reg} <= wdat[11:4];
      end
      ...
   endcase
end

always @ (*)
begin
   if (wr) rdat = 'Z;
   else case (addr)
     ...
     16'h0010: rdat = {f2a_reg, f2b_reg, f1_reg};
     ...
   endcase
end
```

A Register May Have a Mix of Read-Only and Writable Fields

Read-only fields cannot be written to, even with a backdoor. A register containing a mix of read-only and writable fields will skip the read-only fields during a back-door write operation.

A Memory Must be Implemented Using a Single Unpacked Array

A memory is accessed using the offset of the memory as the index of the array storing its content. Two memories cannot be modeled using the same array nor can a memory be implemented using the concatenation of multiple arrays (either bit-wise or address-wise).

For example, the memory specified using the *memory* definition shown in Example 3-20, can be implemented using the RTL code shown in Example 3-21. The *reg* named `m1_reg` is used to implement the entire memory.

Example 3-20 Memory Specification

```
memory m1 (m1_reg) @'h1000{
   size 1k;
   bits 16;
}
```

Example 3-21 Implementation of Memory with Unpacked Array

```
reg [15:0] m1_reg[1024];
always @ (posedge clk)
begin
   if (wr) casex (addr)
      ...
      16'b0001_00xx_xxxx_xxxx: m1_reg[addr[9:0]] <= wdat;
      ...
   endcase
end

always @ (*)
begin
   if (wr) rdat = 'Z;
   else casex (addr)
     ...
     16'b0001_00xx_xxxx_xxxx: rdat = m1_reg[addr[9:0]];
     ...
   endcase
end
```

Note:

Automatic generation of back-door access to memories modeled using DesignWare models is not yet supported.

Support for Value Inversion in Backdoor Code

Value inversion is supported in the backdoor code using the following property inside the field:

```
backdoor_xor_mask <bit_by_bit_inversion_mask>
```

Based on the bit mask information, ralgen transforms the backdoor value appropriately. The `bit_by_bit_inversion_mask` field can be specified in decimal, binary, octadecimal or hexadecimal format.

For example,

```
backdoor_xor_mask 'b010;
```

This modifies the backdoor task as follows:

```
   rw.value[0] = <backdoor_path> ^ 'b010;
```

Support for Field Association with Register Backdoor Specification

Current RALF syntax does not provide a way to accurately specify the register backdoors when the register backdoor is composed of a concatenation of multiple HDL paths. The proposal is to provide a way to associate HDL paths with the specific fields of any given register while instantiating the register. The style would be similar to the named port mapping style used in Verilog.

Command-Line Option

None

This implementation involves support for additional syntax in RALF. Ralgen option is not required. The proposed syntax works with both the DPI-C based backdoors (default) and the HDL XMR based backdoors (ralgen option `-b`).

Specification

- Supporting simple HDL path specification using field name association

```
register reg=reg_inst (.fld1(reg.fld1), .fld2(reg.fld2),
….);
```

`reg_inst` is an instance of register of type `reg`, `fld1 and fld2` are fields defined within `reg`. This syntax generates backdoor `write()`/`read()` tasks that assign/sample values to/from `hdl_path1`and `hdl_path2` from/to the model value that correspond to the position of fields `fld1 and fld2` respectively.

- Supporting HDL path concatenation within a named association

```
register reg=reg_inst (.fld1(hdl_path1),
.fld2(hdl_path2_high, hdl_path2_low, …));
```

The first specification holds here, but the HDL path specification of `fld2` here is a concatenation of two or more HDL paths. Backdoors generated using option `-b` do a simple Verilog concatenation while assigning/sampling the model values that correspond to field `fld2`. For DPI-C based backdoors, this style uses `hdl_path2_high` ignoring "hdl_path2_low , …" because it does not provide adequate information to accurately map the HDL paths into the correct offset positions of the model values corresponding to the field `fld2`.

- Format specifiers is used as a part of HDL paths to specify the arrays. The current support does not include the specification of more than one format specifier in the HDL path i.e., specifying backdoor HDL path for an array of fields within an array of registers cannot be done as shown below. This would be an enhancement for the future.

  ```
  register reg=reg_array [16]
  (.fld1(top.reg[%d].fld1[%d]),
  .fld2(top.reg[%d].fld2[%d]), ….);
  ```

- If a format specifier is specified, either the register or the associated field must be an array. An error appears if both of them are defined as a non-array type.

Examples

The following RALF file:

```
register HOST_ID {
  field REV_ID  {
    bits 32;
    access rw;
    reset 'h03;
  }
  field CHIP_ID  {
    bits 8;
    access rw;
    reset 'h5A;
  }
}

block host_regmodel {
  bytes 2;
  register HOST_ID(.REV_ID(rev_id), .CHIP_ID(chip_id));
}
```

Generates the following `write()`/`read()` tasks within the register backdoor classes:

ral_host_reg_model.sv:

```
class ral_reg_host_regmodel_HOST_ID_bkdr extends
uvm_reg_backdoor;

     virtual task read(uvm_reg_item rw);
          do_pre_read(rw);
          rw.value[0][31:0] =
`HOST_REGMODEL_TOP_PATH.rev_id;
          rw.value[0][39:32] =
`HOST_REGMODEL_TOP_PATH.chip_id;
          rw.status = UVM_IS_OK;
          do_post_read(rw);
     endtask

     virtual task write(uvm_reg_item rw);
          do_pre_write(rw);
          `HOST_REGMODEL_TOP_PATH.rev_id =
rw.value[0][31:0];
          `HOST_REGMODEL_TOP_PATH.chip_id =
rw.value[0][39:32];
          rw.status = UVM_IS_OK;
          do_post_write(rw);
     endtask
endclass
```

The following RALF file:

```
register HOST_ID {
  field REV_ID  {
    bits 32;
    access rw;
    reset 'h03;
  }
  field CHIP_ID  {
    bits 8;
    access rw;
    reset 'h5A;
  }
}

block host_regmodel {
  bytes 2;
  register HOST_ID(.REV_ID(rev_id), .CHIP_ID(chip_id_high,
chip_id_low));
}
```

Generates the following `write()`/`read()` tasks within the register backdoor classes:

ral_host_reg_model.sv:

```
class ral_reg_host_regmodel_HOST_ID_bkdr extends
uvm_reg_backdoor;

         virtual task read(uvm_reg_item rw);
         do_pre_read(rw);
         rw.value[0][31:0] =
`HOST_REGMODEL_TOP_PATH.rev_id;
         rw.value[0][39:32] =
{`HOST_REGMODEL_TOP_PATH.chip_id_high,

`HOST_REGMODEL_TOP_PATH.chip_id_low};
         rw.status = UVM_IS_OK;
         do_post_read(rw);
     endtask

     virtual task write(uvm_reg_item rw);
         do_pre_write(rw);
         `HOST_REGMODEL_TOP_PATH.rev_id =
rw.value[0][31:0];
         {`HOST_REGMODEL_TOP_PATH.chip_id_high,
 `HOST_REGMODEL_TOP_PATH.chip_id_low} = w.value[0][39:32];
         rw.status = UVM_IS_OK;
         do_post_write(rw);
     endtask
endclass
```

The following RALF file:

```
register HOST_ID {
  field REV_ID  {
    bits 32;
    access rw;
    reset 'h03;
  }
  field CHIP_ID  {
    bits 8;
    access rw;
    reset 'h5A;
  }
}

block host_regmodel {
  bytes 2;
 register HOST_ID (.REV_ID(rev_id), .CHIP_ID((chip_low, 0, 
3),
                                         (chip_high, 4, 
7)))
}
```

Generates the following `write()`/`read()` tasks within the register backdoor classes:

ral_host_reg_model.sv:

```
class ral_reg_host_regmodel_HOST_ID_bkdr extends 
uvm_reg_backdoor;

     virtual task read(uvm_reg_item rw);
          do_pre_read(rw);
          rw.value[0][31:0] = 
`HOST_REGMODEL_TOP_PATH.rev_id;
          rw.value[0][35:32] = 
`HOST_REGMODEL_TOP_PATH.chip_low;
          rw.value[0][39:36] = 
`HOST_REGMODEL_TOP_PATH.chip_high;
          rw.status = UVM_IS_OK;
          do_post_read(rw);
     endtask

     virtual task write(uvm_reg_item rw);
          do_pre_write(rw);
          `HOST_REGMODEL_TOP_PATH.rev_id = 
rw.value[0][31:0];
          `HOST_REGMODEL_TOP_PATH.chip_low = 
rw.value[0][35:32];
          `HOST_REGMODEL_TOP_PATH.chip_high = 
rw.value[0][39:36];
          rw.status = UVM_IS_OK;
          do_post_write(rw);
     endtask
endclass
```

You can specify the width for concatenated field paths by specifying LSB and MSB positions, as follows:

```
field fld2 ((0:1)(data1_out), (2:3)(data2_out))
   {
     bits 4;
     reset 0;
     access rw;
    }
```

In case of register instantiation, the path is specified as follows:

```
register reg=reg_inst (.fld1(reg.fld1),
                        .fld2((0:1) reg.data1_out),
                        (2:3) reg.data2_out),
                         …)
                      );
```

With the additional information coming from LSB and MSB positions, you shall be able to generate `add_hdl_path` with the required information. This will also allow you to set the path for particular bits.

Note:

- The LSB and MSB positions are with respect to the field. Ralgen automatically uses the field offset to adjust the LSB position at the register level.
- During register definition, HDL path specification with LSB and MSB positions is not allowed at the register level. The LSB and MSB positions should rather be declared at the field level in such a case. For example,

```
register r1 {
   bytes 4;
   field f2 ((0:7)path1, (8:15)path2) {
     bits 16;
   }
   field f1 (f2_path) {
     bits 16;
 }
}
```

- If `hdl_path` is specified during register definition at field level, and at register instantiation level, then ralgen uses the path specified during register instantiation.

  -You must either specify all the paths for a particular field with LSB/MSB or none of the paths should contain LSB and MSB.

  -In case multiple paths are provided without LSB and MSB, then ralgen continues to use only the first path with DPI-C based backdoors.

Support for Register Array Index in Field Paths

By default, Ralgen looks for `%d` or `[%d]` type of format in `hdl_path`, when the corresponding RALF object is an array and does not process the format specifier when it is not an array.

In a scenario where a register in RALF is merely a logical representation and there is no physical representation, this implies that the register array in RALF would probably be modeled as individual field arrays in RTL. So, if backdoor access is required, it would need to use the register index in field paths.

Command-Line Option

```
-use_reg_idx_in_fld_path
```

When specified, ralgen substitutes `%d` or `[%d]` format specifier with the register array index in the field path for XMR-based backdoors.

Specification

The register index can be used in the field path with the following conditions:

- All the fields in a register must contain an extra index.
- The path specification must be consistent across the fields if multiple field specifiers are present.

  For example, if one field path contains `%d` as the first specifier and `[%d]` as the second one, then all other field paths should also have the same pattern, that is, `%d` must be the first specifier in all the field paths.

Example

Consider the following RALF specification:

```
register foo_bar {
        bytes 4;
        field foo (mod.foo_%d) {
                access rw;
                bits 16;
                hard_reset 'h0000;
        }
        field bar (mod.bar_%d) {
                access rw;
                bits 16;
                hard_reset 'h0000;
        }
}

block regs {
        bytes 4;
        register foo_bar=foo_bar[2] @'h0;
}
```

With the `-use_reg_idx_in_fld_path` option, the generated backdoor code uses the array index of `foo_bar` for substitution in the field paths. The generated backdoor code snippet is as follows:

```
class ral_reg_regs_foo_bar_bkdr extends uvm_reg_backdoor;
…
     virtual task read(uvm_reg_item rw);
          do_pre_read(rw);
          case (foo_bar)
          0:       begin
             rw.value[0] = `UVM_REG_DATA_WIDTH'h0;
             rw.value[0][15:0] = `REGS_TOP_PATH.mod.foo_0;
             rw.value[0][31:16] = `REGS_TOP_PATH.mod.bar_0;
          end

          1:       begin
             rw.value[0] = `UVM_REG_DATA_WIDTH'h0;
             rw.value[0][15:0] = `REGS_TOP_PATH.mod.foo_1;
             rw.value[0][31:16] = `REGS_TOP_PATH.mod.bar_1;
          end
          endcase
          rw.status = UVM_IS_OK;
          do_post_read(rw);
     endtask
     virtual task write(uvm_reg_item rw);
          do_pre_write(rw);
          case (foo_bar)
          0:       begin
             `REGS_TOP_PATH.mod.foo_0 = rw.value[0][15:0];
             `REGS_TOP_PATH.mod.bar_0 = rw.value[0][31:16];
          end

          1:       begin
             `REGS_TOP_PATH.mod.foo_1 = rw.value[0][15:0];
             `REGS_TOP_PATH.mod.bar_1 = rw.value[0][31:16];
          end
          endcase
          rw.status = UVM_IS_OK;
          do_post_write(rw);
     endtask
endclass
```

Support for Active Monitoring Logic

For active monitoring, an additional watch task with case statement is generated to keep the testbench register instance up-to-date (mirror) with RTL register. This is a part of the same backdoor class as only one backdoor class can be set using the `set_backdoor` function. This uses the same interface redirection to continue to make the code reusable. Thus, it makes use of local (relative to where the top interface is bound) hierarchical references counting on hierarchy never changing and being existent. Other classes for specific actions, for example, invalidate caches, initialization cleanup, backdoor access to dcache tag/data can be enabled to provide `user_code` addition when using ralgen.

Command-Line Option

```
-auto_mirror
```

When specified, generates the active monitoring logic within the backdoor infrastructure of the register model.

Specification

- It is necessary to specify the `-b` option when specifying the `-auto_mirror` option.
- A SystemVerilog interface is generated (in a separate file) additional to the register model itself which you need to pass to the compile command line. Additional setup is not required.
- It can be used with `–gen_vif_bkdr` switch, in which case the generated interface file has additional functions to facilitate active monitoring.

Example

The following (partial) RALF file:

```
block host_regmodel {
  bytes 2;
  register HOST_ID (host_id)   @'h0000;
}
```

Generates the following additional tasks in the register backdoor classes (partial):

ral_host_reg_model.sv:

```
    virtual task wait_for_change(uvm_object element);
      uvm_reg rg;
      uvm_status_e status;
      $cast(rg, element);
      @(`HOST_REGMODEL_TOP_PATH.rev_id or
           `HOST_REGMODEL_TOP_PATH.chip_id);
      rg.mirror(status, , UVM_BACKDOOR);
    endtask

     virtual function bit is_auto_updated(uvm_reg_field
field);
       case (field)
       "REV_ID" : return 1;
       "CHIP_ID" : return 1;
       endcase
     endfunction
```

The option `–gen_vif_bkdr` generates the following additional tasks in the generated interface (partial) and the tasks within the backdoor classes is modified appropriately:

ral_host_reg_model.sv:

```
class ral_reg_host_regmodel_HOST_ID_bkdr extends
uvm_reg_backdoor;
```

```
     virtual ral_host_regmodel_intf __reg_vif;
     virtual task wait_for_change(uvm_object element);
       uvm_reg rg;
       uvm_status_e status;
       $cast(rg, element);

__reg_vif.ral_host_regmodel_HOST_ID_wait_for_change();
       rg.mirror(status, , UVM_BACKDOOR);
     endtask
endclass
```

ral_host_reg_model_interface.sv:

```
interface ral_host_regmodel_intf;
     import uvm_pkg::*;

     initial
uvm_resource_db#(virtual ral_host_regmodel_intf)::set("*",
"uvm_reg_bkdr_if", interface::self());

     task ral_host_regmodel_HOST_ID_wait_for_change();
          @(`HOST_REGMODEL_TOP_PATH.rev_id or
            `HOST_REGMODEL_TOP_PATH.chip_id);
     endtask
endinterface
```

Reserved RALF Keywords in Backdoor Path

`ralgen` will error out if any RALF reserved keyword is found in any RALF backdoor HDL path specification. That means, the following RALF description in Example 3-22 will error out.

Note: There are two RALF keywords, `block` and `register` in the HDL path of register `reg`.

Example 3-22

```
register reg (block.register) {
```

```
          ...

}
```

If any of your RALF description has got RALF reserved keywords used in any of its backdoor HDL path specification, then use the following RALF syntax for specifying your RALF backdoor HDL path:

Example 3-23

```
register reg hdl_path = (block.register) {

          ...

}
```

The semantics of `hdl_path` usage is functionally/completely equivalent to the original HDL path specification style (used in Example 3-22), except the fact that RALF reserved keywords checking will be disabled when the `hdl_path` syntax is used for specifying backdoor HDL path.

4

Functional Coverage Model

Optionally, you can generate a RAL model with one or more predefined functional coverage models to measure how thoroughly the various host-accessible elements are exercised by your functional verification suite.

The default generated RAL model does not contain any functional coverage model. To generate a coverage model, `ralgen` must be invoked with the `-c` option. The argument to the `-c` option determines which coverage model is included in the RAL model:

Use `-c` b to generate the register bits coverage model.

-c a

Generate the address map coverage model.

-c f

Generate the field value coverage model.

Multiple functional coverage models can be generated in the same RAL model by specifying the `-c` option multiple times or specifying multiple arguments to a single `-c` option. For example, the following commands are equivalent:

```
   % ralgen -c b -c a ...
   % ralgen -c ba ...
```

Although the generated RAL model might contain one or more functional coverage models, they are not enabled by default. This is necessary in order to reduce the memory footprint of a RAL model, as some functional coverage models can be significant in size, and to improve the runtime performance of simulations as the collection of coverage metrics and the writing of functional coverage databases incurs a significant overhead. Therefore, it is necessary to explicitly enable a functional coverage model when a RAL model is first constructed.

To include coverage model in various block, register or memory abstract class instances, call the `include_coverage` class before building the RAL model.

For Example:

```
uvm_reg::include_coverage("*", UVM_CVR_ALL);
```

To enable implicit sampling, add the `set_coverage` call before starting the sequence.

Example 4-1   Enabling Implicit Sampling

```
virtual task run_phase(uvm_phase phase);
…
            env.regmodel.reset();
      void'(env.regmodel.set_coverage(UVM_CVR_ALL));
begin
```

```
          uvm_reg_sequence seq;

        seq = uvm_reg_bit_bash_seq::type_id::create("seq");
          seq.model = env.regmodel;
          seq.start(env.bus.sqr);
…
endtask
```

Note:

Ralgen automatically generates a corresponding HVP file when it is generating UVM RAL code with coverage enabled. The HVP file gets generated with the name `uvmp_<top>.hvp` and it can be included in your top level verification plan. The plan in HVP can be used as a sub-plan.

Predefined Functional Coverage Models

The following functional coverage models are available to be generated in the RAL model. Different models target a different perspective of the register verification process and should be used when appropriate.

Because functional models can be large in size and significantly impact runtime performance, they should be used carefully, at the right level of design granularity and only when their coverage points are targeted. Once filled to satisfaction, functional coverage models should no longer be generated—although their metrics should be preserved and continued to be reported.

Register Bits

This model is generated using the `-c b` command-line option for every register specified with a "+b" cover attribute. The coverage model is constructed by specifying the `uvm_reg::REG_BITS` symbol.

This model is designed to confirm that every specified bit in a RAL model has been thoroughly exercised and is implemented as specified. This functional model can be quite large and is, therefore, best used at the block level.

This functional coverage model is implemented by instances of `ral_cvr_reg_regname::reg_bits` coverage groups. In a block, there is one coverage group instance per register, for each domain instantiating the register. There is a coverage point for every field defined in the register and a bin to measure whether each individual bit of a field has been read and written through the domain physical interface as a 0 and a 1, respectively. For field arrays, a coverage point will be generated for each and every field in the field array and those coverpoints will be named in the <field_name>_<array_index> format where, <array_index> will range from 0 to field array size - 1.

This model does not measure backdoor accesses. The coverage model does not include unused or reserved bits.

Address Map

This model is generated using the `-c a` command-line option for every register and memory specified with a "+a" cover attribute. The coverage model is constructed by specifying the `UVM_CVR_ADDR_MAP` symbol.

This model is designed to confirm that the address map of a design has been thoroughly exercised. It is best used at the top-level.

Address map coverage is implemented at the block level and supports address coverage of registers (including any registers in register files) and memories. Because fields cannot be physically accessed, they are not considered in the address map coverage. Virtual registers, being a logical structure imposed on a memory, are not included in the address map coverage either: it is assumed that if the address map coverage model of the memory containing the virtual registers is covered, the address map coverage model for the virtual registers can be considered covered as well.

The address map functional coverage model is composed of the ral_cvr_block_<block_name>::[<domain_name>_]addr_map coverage groups. For each block, there is one coverage group instance per domain in each block instance. In each coverage group (i.e. domain), there is a coverage point for each register (including each registers in register arrays and register files) and a coverage point for each memory in the block.

A register coverage point contains only one bin named "accessed". The bin is covered whenever the register is accessed using a read or a write operation.

A memory coverage point contains three bins. The first bin, named "first_location_accessed", is covered when the first location in the memory is accessed using a read or a write operation. The second bin, named "last_location_accessed", is covered when the last location in the memory is accessed using a read or a write operation. The third bin, named "other_locations_accessed", is covered when anyone of the remaining locations in the memory is accessed using a read or write operation.

Address map coverage measurement happens automatically during any front door read or write operation. Back-door accesses do not contribute toward the address map functional coverage.

Field Values

This model is generated using the `-c f` command-line option for every register specified with a "+f" cover attribute. The coverage model is constructed by specifying the `UVM_CVR_FIELD_VALS` symbol.

This model is designed to confirm that every configuration of a design has been verified. It is best used at the top-level.

Field value coverage model is implemented at the register level and supports value coverage of all fields and cross coverage between fields and other cross coverage points within the same register. Field value coverage is not supported for virtual fields/registers.

The field value functional coverage model is composed of the ral_reg_<reg_name>::field_values coverage groups. There is one coverage group instance per register instance. In each coverage group, there is a coverage point for each field in the register, except for "unused" and "reserved" fields. For field arrays, a coverage point will be generated for each and every field in the field array, and those coverpoints will be named in the <field_name>_<array_index>_value format where, <array_index> will range from 0 to field array size - 1.

By default, if the size of a field is 4 bits or less, the corresponding coverage point contains a bin for each possible value of that field. If the size of the field is greater than 4 bits, the corresponding coverage point contains three bins: the first bin, named "min", corresponds to the minimum value of that field (or ’0); the second bin, named "max", corresponds to the maximum value of that field (or ’1); and the third bin, named "others" corresponds to all other values of that field. The weight of a coverpoint is equal to the number of bins in that point.

You can sample field value coverage by using the `sample_field_values()` function within the RAL registers.

By using this method, you will be able to sample field values within the RAL register itself, which would sample field coverage for all the fields within the register by calling `field_values.sample()` for the register.

User-Defined Field Value Coverage Bins

If the default field value bins are not suitable, there are many ways coverage bins can be defined for a coverage corresponding to a field value. In all cases, the weight of the coverage point will be equal to the number of bins.

If symbolic values are defined for a field using the "enum" property, a bin is implicitly defined for each symbolic value. The field specification shown in Example 4-2 will create three bins, named "AA", "BB" and "CC", each corresponding to field values 0, 1 and 15 respectively.

Example 4-2   Defining implicit coverage bins via symbolic field values

```
field f2 {
     bits 8;
     enum { AA, BB, CC=15 }
   }
```

User-defined bins can be explicitly specified using the "coverpoint" attribute. Example 4-3 illustrates how multiple coverage bins and bin arrays can be defined using numerical as well as symbolic field values, sets of values and ranges of values. The semantics of the bin specification is identical to the equivalent bin specification in SystemVerilog, as specified in the section named “*Defining coverage points*” in the 1800-2009 SystemVerilog Language Reference Manual.

Example 4-3   Defining explicit coverage bins

```
field f2 {
     bits 8;
     enum { AA, BB, CC=15 }
     coverpoint {
         bins AAA     = { 0, 12 }
         bins BBB []  = { 1, 2, AA, CC }
         bins CCC [3] = { 14,15, [ BB : 10 ] }
         bins DDD     = default
     }
   }
```

The `coverpoint` attribute supports user-defined bins corresponding to ignore and illegal value sets, that is, both `ignore_bins` and `illegal_bins` are supported along with `bins` for representing ignore and illegal value sets correspondingly.

The bin specification will be the same as bins. For the following example:

```
field f1 {
 bits 4;
 coverpoint {
   bins A = { 0, 12 }
   ignore_bins B = { 13 }
   illegal_bins C = { 14 }
 }
}
```

In UVM flow, the generated code for covergroup will be as follows,

```
covergroup cg_vals ();
  option.per_instance = 1;
  f1_value : coverpoint f1.value {
     bins A = { 0, 12 };
     ignore_bins B = { 13 };
     illegal_bins C = { 14 };
     option.weight = 3;
  }
endgroup : cg_vals
```

Specifying Wildcard in Coverpoint Specification

The generated SystemVerilog code includes the `wildcard` keyword prefixed to bin, as shown in the following example:

Input RALF:

```
register reg1 {
        field value (reg1) {        bits 7;
                        reset 7'b0;
                        access rw;
                        coverpoint {
                       wildcard bins wval = {8'b1???????};
                          bins v[] = {[3'h0:3'h7]};
                          }
                    }
        field bit_7 (reg1_bit_7) {   bits 1;
                        reset 1'b0;
                        access rw;
                    }

}
```

In UVM flow, the generated code for covergroup is as follows:

```
covergroup cg_vals ();
                option.per_instance = 1;
                value_value : coverpoint value.value {
                     wildcard bins wval = { 8'b1??????? };
                        bins v[] = { [3'h0:3'h7] };
                        option.weight = 2;
                }
               bit_7_value : coverpoint bit_7.value[0:0] {
                        option.weight = 2;
                }
        endgroup : cg_vals
```

User-Defined Cross Coverage Specification

A cross coverage point between different field values within the same register can be specified using the "cross" attribute. If a user-defined cross-coverage point is labeled, it is possible to use that cross-coverage point in another cross-coverage point.

Example 4-4   User-defined cross-coverage point

```
register r {
    field f1 {...}
    field f2 {...}
    field f3 {...}

    cross f1 f2 {
        label xyz;
    }
    cross xyz f3;
}
```

RALF Cover Attribute

By default, all applicable elements in a RAL models are included in the address map and register bits coverage models and all are excluded from the field value coverage model. The "cover" attribute can be used to specify the portions of the RAL model that should be included in or excluded from a coverage model.

All elements in a RAL model can be specified with a "cover" attribute to specify whether it and all of the sub-elements it contains are to be included in or excluded from a particular coverage mode. The address map, register bits and field value coverage models are identified by the letters "a", "b" and "f" respectively. A model element is included in or excluded from a coverage model by prefixing its identifying letter with a "+" or a "-’ respectively. For example, the attribute "cover +a+b-f" specifies that this element is included in the address map and register bits coverage model but not in the field values coverage model.

The coverage attribute for a RAL element are automatically inherited from the higher-level element. If a coverage model is not specified in a "cover" attribute, the inclusion or exclusion for that model is inherited from the higher level. For example, the attribute "cover +f" specified that this element (and all of its lower-level elements) are to be included in the field value coverage model but it does not say anything about the inclusion or exclusion of this element with respect to the other coverage models.

It is important to note that, unless a system, block, register file or register contains a "cover +f" attribute, no field value coverage model will be generated.

Example 4-5   Inherited cover attributes

```
system top {
    block b {
        cover -a+f            #-a+b+f
        …
        register r1 {
            cover -f          #+a+b-f
        }
        register r2 {
            cover -b          #+a-b+f
        }
    }
    system sub {
        cover +f              #+a+b+f
        …
    }
}
```

If a "cover" attribute is specified outside the "domain" attribute of a multi-domain block or system, it applies to all domains specified in that block or system. A "cover" attribute specified inside a "domain" attribute applies to all registers and memories instantiated in that domain.

5

Randomizing Field Values

A RAL model can specify constraints on field values. If a field is specified with a `constraint` attribute, its value can be randomized. If a field is specified with no `constraint` attributes, it is a constant field that is never randomized. If you require an unconstrained field that can be randomized, specify the field with an empty `constraint` attribute. For example, fields `f1` and `f2` in Example 5-1 are randomized but field `f3` is not.

Within a field specification, the constraints specify the valid values for the field independently of any other field value. Within a register specification, the constraints specify constraints on field values based on the register where the field is instantiated or other field values within the register. Within a block or system specification, the constraints specify constraints on field values based on the block or system where the field is instantiated or other field values within the block or system.

Example 5-1   Field Constraints

```
field f1 {
   bits 8;
   constraint spec {
      value <= ’h80;
   }
}

register r {
   field f1;
   field f2 {
      bits 8;
      constraint consistency {
         f1.value == f2.value;
      }
   }
   field f3 {
      bits 2;
   }
}
```

Example 5-2   RAL Model for Example 5-1

```
class ral_r1 extends uvm_ral_reg;
   rand uvm_ral_field f1;
   rand uvm_ral_field f2;

   constraint f1_spec {
      f1.value < ’h80;
   }
   constraint consistency {
      f1.value == f2.value;
   }
   constraint user_defined;
}
```

Field constraints are inlined in the register class that instantiates the field to minimize the possibility of randomly selecting inconsistent field values. Constraints declared in a `field` property in the RAL description are not visible in the field abstraction class because they are inlined in the register class that instantiates the field and not in the field itself. If a field descriptor is directly randomized, it is therefore unconstrained. Therefore, do not directly randomize field descriptors. To randomize the content of fields subject to their constraints, the register, block, or system descriptor must be randomized. Once randomized, the field values can be written or updated into the DUT.

Example 5-3 Improperly Randomizing Fields

```
ral_model.r1.f1.randomize();
```

Example 5-4 Properly Randomizing Fields

```
ral_model.r1.randomize();
```

The content of memories cannot be randomized.

6

Generating RALF and UVM Register Model from IP-XACT

The registers and memories in the design under verification are usually described in a RALF file for UVM RAL. You create this description based on your design register specification. The register specification is part of an architecture/design document usually created in a format such as FrameMaker, Microsoft Word, or a spreadsheet. Since there is no common standard text format that is used in the industry, every user has slightly different variations in describing the register specifications. IP-XACT is becoming a standard for describing register specifications.

After the register specification is converted to a common meta-data model, such as the IP-XACT schema, you can use the **`ralgen`** utility to automatically create a RALF file description. As discussed in "RALF File Description Mechanism" , the RALF model is used by **`ralgen`** to generate the corresponding RAL model for verification.

Definition of IP-XACT Schema

IP-XACT is a standard specification for eXtensible Markup Language (XML) meta-data and tool interfaces that is an industry intermediate specification format.

The IP-XACT standard specification is a mechanism to document and exchange information about design IP, its characteristics and its required configuration and integration. The memory and register specification is also described using the IP-XACT schema. The IP-XACT meta-data was conceived by the SPIRIT consortium.

The IP-XACT XML description is generated by the user from the original register specification using a user-supplied conversion script.

RALF File Description Mechanism

The default generated RALF model maps the XML specification file to generic RALF syntax format. To generate the RALF file from an IP-XACT file, **`ralgen`** is invoked with the **`-ipxact2ralf`** option.

For example, the following command can be used to generate a RALF model for cpu_regs registers, if the cpu_reg.xml file exists:

```
% ralgen -ipxact2ralf cpu_regs.xml
```

The generated file is named cpu_regs.ralf, which contains RALF descriptions of the registers. Example 6-1 shows the register description in IP-XACT schema, and its equivalent RALF format.

Example 6-1 Generating RALF from IP-XACT

cpu_regs.xml:

```
...
<spirit:register>
      <spirit:name>r2</spirit:name>
      <spirit:addressOffset>0x8</spirit:addressOffset>
      <spirit:size>64</spirit:size>
      <spirit:access>read-write</spirit:access>
   <spirit:field>
      <spirit:name>f2</spirit:name>
      <spirit:bitOffset>0</spirit:bitOffset>
      <spirit:bitWidth>1</spirit:bitWidth>
      <spirit:access>read-write</spirit:access>
   </spirit:field>
   ...
</spirit:register>
```

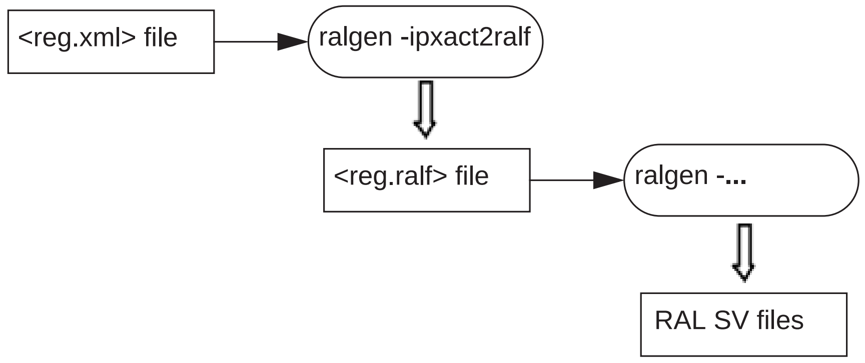
cpu_regs.ralf:

```
...
register r2 @'h8 {
   field f2 {
      bits 1;
      access rw;
   }
...
}
```

In the above, only a RALF file is generated. The next step is to generate all the necessary RAL files by invoking **`ralgen`** a second time, with appropriate switches, using the generated RALF file.

Figure 6-1 shows the steps involved in this process.

Figure 6-1   RALF Generation and RAL Generation



For IP-XACT 1.5, use `ralgen -uvm -ipxact <ipxact-file> <other RALF options>` to invoke ralgen.

For example, the following command is used to generate UVM RAL from IP-XACT 1.5:

```
ralgen -ipxact -uvm -t top mycpu.xml
```

The generated file, `mycpu.xml` contains RALF descriptions of the registers, which acts as an input to ralgen with `-uvm` to generate the SV UVM RAL model in a `ral_top.sv` file.

The `-ipxact_files` option is used to support multiple IP-XACT files as input. This option provides the list of files to be translated. The leaf-level components are dumped as standalone blocks, and the rest of the blocks are composed on the top of these blocks.

To generate the RALF file, use the following command line:

```
ralgen -ipxact_files <file_containing_ipxact_file_list>
```

To generate the RAL model directly, use the following command line:

```
ralgen -ipxact -ipxact_files <file_containing_ipxact_file_list>
```

Supported IP-XACT Schema

The ralgen utility accepts IP-XACT schema version 1.5 descriptions for the registers and memories with a few limitations. The conversion utility supports the XSD schema as this is the schema used for IP-XACT descriptions. Support for IPXACT DIM to RALF mapping is added.

Generic RALF Features and IP-XACT Mapping

Table 6-1 lists the generic IP-XACT features and their RALF equivalents supported by this conversion utility.

Table 6-1 RALF Equivalents of IP-XACT Features

Spirit IP-XACT 1.4 Description	**RALF Generic Feature**
<spirit:**name**>***name***</spirit:**name**>	name
<spirit:**description**>***description***</spirit:**description**>	description, doc
<spirit:**access**>*access_mode*</spirit:**access**>	access
<spirit:**reset**> ... </spirit:**reset**>	reset, hard_reset
<spirit:**value**>***reset_value***</spirit:**value**>	reset_value

Table 6-2 lists the generic IP-XACT access modes and their RALF equivalents supported by this conversion utility.

Table 6-2 RALF Equivalents of IP-XACT Access Modes

IP-XACT Definition access_mode	**RALF Register Access Mode**
read-write	rw
read-only	ro
write-only	wo

field

`field` *`name`* `[{`*`properties`*`}]`

Spirit IP-XACT Equivalent	**RALF Feature**
<spirit:**field**> ... </spirit:**field**>	field
<spirit:**bitOffset**>***field_bit_offset***</spirit:**addressOffset**>	*@field_bit_offset*
<spirit:**bitWidth**>***number_of_bits_in_field***</spirit:**bitWidth**>	bits
<spirit:**access**>***access_mode***</spirit:**access**>	access

Note:

If the name of a field is `unused` or `reserved`, IP-XACT to RALF translation treats it as a special case and it is dumped as an unused or reserved field. Only the bits information is translated from IP-XACT to RALF. The rest of the fields are ignored.

register

`register` *`name {properties}`*

Spirit IP-XACT Equivalent	**RALF Feature**
<spirit:**register**> ... </spirit:**register**>	register
<spirit:**addressOffset**>*register_bit_offset*</spirit:**addressOffset**>	*@'register_bit_offset*
<spirit:**size**>width_of_register</spirit:**size**>	bytes

<spirit:**reset**><spirit:**value**>reset_value</spirit:**value**>
 	
Optional:
<spirit:**mask**>mask_value</spirit:**mask**></spirit:**reset**>
	The reset value for a field is specified at register level. The mask defines which bit has the reset value.
	For example:
	field rdata_msb @'h4 {
	.....
	hard_reset 'ha;
	}

<spirit:**typeIdentifier**>type_name</spirit:**typeIdentifier**>	Ralgen generates single definition for multiple registers in same description, if spirit:typeIdentifier tag is specified in each of these registers with the same type name.

registerFile

`registerFile` *`name {properties}`*

Spirit IP-XACT Equivalent	**RALF Feature**
<spirit:**registerFile**> ... </spirit:**registerFile**>	regfile
<spirit:**name**>***status***</spirit:**name**>	name
<spirit:**description**>Status register</spirit:**description**>	doc
<spirit:**addressOffset**>*regfile_bit_offset*</ spirit:**addressOffset**>	*@'regfile_bit_offset*
<spirit:**range**>*range*</spirit:**range**>	range
<spirit:**size**>width_of_register</spirit:**size**>	bytes
<spirit:**access**>***access_mode***</spirit:**access**>	access

block

`block` *`name {property}`*

Spirit IP-XACT Equivalent	**RALF Feature**
<spirit:**addressBlock**> ... </spirit:**addressBlock**>	block
<spirit:**baseAddress**>'*block_start_address*</ spirit:**baseAddress**>	@'*block_start_address*

memory

`memory` *`name {property}`*

Spirit IP-XACT Equivalent	**RALF Feature**
<spirit:**usage**>memory</spirit:**usage**>	memory
<spirit:**baseAddress**>'*memory_start_offset* </spirit:**baseAddress**>	@'*memory_start_offset*
< spirit:**size**>*number_of_rows*</spirit:**size**>	[size]
<spirit:**bitWidth**>*number_of_bits_in_each_row* </spirit:**bitWidth**>	bits, bytes

memoryRemap

`memoryRemap` *`name {property}`*

Spirit IP-XACT Equivalent	**RALF Feature**
<spirit:**memoryRemap**> ... </spirit:**memoryRemap**>	domain
<spirit:**addressBlock**> ...</spirit:**addressBlock**>	block
<spirit:**name**>*name*</spirit:**name**>	name
<spirit:**baseAddress**>'*block_start_address*</ spirit:**baseAddress**>	@'*block_start_address*
<spirit:**range**>*range*</spirit:**range**>	range
<spirit:**usage**>*memory*</spirit:**usage**>	usage
<spirit:**access**>***access_mode***</spirit:**access**>	access

register array

`register array`

Spirit IP-XACT Equivalent	**RALF Feature**
<spirit:**register**> ... </spirit:**register**>	register
<spirit:**baseAddress**>’*array_start_offset* </spirit:**baseAddress**>	@’*array_start_offset*
< spirit:**size**>*width_of_register*</spirit:**size**>	bytes
<spirit:**dim**>number_of_array_elements</spirit:**dim**>	[dim]

system

`system` *`name`* `{`*`property`*`}`

Spirit IP-XACT Equivalent	**RALF Feature**
<spirit:memoryMap> ... </spirit:memoryMap>	system

bank

bank *`name`* `{`*`property`*`}`

Spirit IP-XACT Equivalent	**RALF Feature**
<spirit:bank > ... </spirit:bank >	system

serial/parallel bank

Spirit IP-XACT Equivalent	**RALF Feature**
<spirit:bankAlignment="serial"></spirit:bankAlignment>	serial specifies that the first item is located at the bank's base address
<spirit:bankAlignment="parallel"></spirit:bankAlignment>	parallel specifies that each item is located at the same base address with different bit offsets

Constraints

IP-XACT provides an option to describe a set of constraint values on the register fields using `writeValueConstraint`, which is converted to an equivalent SystemVerilog constraint.

The `writeValueConstraint` provides the following three ways for specifying the constraint values:

- `minimum`, `maximum`
- `useEnumeratedValues`
- `writeAsRead`

minimum, maximum

It specifies the range of minimum to maximum values for a field to be written with.

```
<spirit:writeValueConstraint>
<spirit:minimum>0x0</spirit:minimum>
<spirit:maximum>0x2</spirit:maximum>
<spirit:writeValueConstraint>
```

The above IP-XACT specification provides the following constraint block output:

```
constraint writeValueConstraint {
       value inside { ['h0:'h3] };
}
```

useEnumeratedValues

If the value of `useEnumeratedValues` is true, it implies that the legal values to write to a field are the ones specified in `enumeratedValues` element for this field.

For example:

```
<spirit:enumeratedValues>
  <spirit:enumeratedValue spirit:usage="read-write">
  <spirit:name>oddParity</spirit:name>
  <spirit:value>0<spirit:value>
</spirit:enumeratedValue>
  <spirit:enumeratedValue spirit:usage="read-write">
  <spirit:name>evenParity</spirit:name>
  <spirit:value>1</spirit:value>
</spirit:enumeratedValue>
</spirit:enumeratedValues>
<spirit:writeValueConstraint>
  <spirit:useEnumeratedValues>true<spirit:useEnumeratedVal
ues>t
</spirit:writeValueConstraint>
```

It generates a constraint block as shown below:

```
enum { oddParity = 0, evenParity = 1 }
constraint writeValueConstraint {
       value inside { 0, 1 };
}
```

writeAsRead

If the value of `writeAsRead` is true, it implies that the only values which can be written to this field are the ones which were previously read.

Currently, this is not supported.

Access Types

The following access types are supported by IP-XACT, which are complaint to UVM RAL.

- read-write - RW
- read-only - RO
- write-only - WO
- read-writeOnce - W1
- writeOnce - W01

The following tables provides the IP-XACT mapping for the access types.

Table 6-3 IP-XACT Mapping for access==read-write

access==read-write				
modifiedWriteValuer	**readAction**			
	Unspecified	**clear**	**set**	**modify**
Unspecified	RW	WRC	WRS	User-defined
oneToClear	W1C	n/a	W1CRS	User-defined
oneToSet	W1S	W1SRC	n/a	User-defined

Table 6-3 IP-XACT Mapping for access==read-write

onetoToggle	W1T	n/a	n/a	User-defined
zeroToClear	W0C	n/a	W0CRS	User-defined
zeroToSet	W0S	W0SRC	n/a	User-defined
zeroToToggle	W0T	n/a	n/a	User-defined
clear	WC	n/a	WCRS	User-defined
set	WS	WSRC	n/a	User-defined
modify	User-defined	User-defined	User-defined	User-defined

Table 6-4 IP-XACT Mapping for access==read-only

access==read-only				
modifiedWriteValuer	**readAction**			
	Unspecified	**clear**	**set**	**modify**
Unspecified	RO	RC	RS	User-defined
All others	n/a	n/a	n/a	n/a

Table 6-5 IP-XACT Mapping for access==write-only

access==write-only				
modifiedWrite Valuer				
	Unspecified	**clear**	**set**	**modify**
Unspecified	WO	n/a	n/a	n/a
clear	W0C	n/a	n/a	n/a
set	W0S	n/a	n/a	n/a
All others	n/a	n/a	n/a	n/a

Table 6-6 IP-XACT Mapping for access==read-writeOnce

access==read-writeOnce				
modifiedWrite Valuer				
	Unspecified	**clear**	**set**	**modify**
Unspecified	W1	n/a	n/a	n/a
All others	n/a	n/a	n/a	n/a

Table 6-7 IP-XACT Mapping for access==writeOnce

access==writeOnce				
modifiedWrite Valuer	**readAction**			
	Unspecified	**clear**	**set**	**modify**
Unspecified	W01	n/a	n/a	n/a
All others	n/a	n/a	n/a	n/a

The following are the access types: RO, RW, RC, RS, WRC, WRS, WC, WS, WSRC, WCRS, W1C, W1S, W1T, W0C, W0S, W1SRC, W1CRS, W0SRC, W0CRS, WO, W0C, W0S, W1, W01.

The optional elements `modifiedWriteValue` and `readAction` are newly introduced for IP-XACT and can be used to specify the remaining access types of UVM RAL.

For `modifiedWriteValue`:

- oneToClear - W1C
- oneToSet - W1S
- oneToToggle - W1T

- zeroToClear - W0C
- zeroToToggle - W0T
- clear - WC
- set - WS

For `readAction`:

- clear - RC
- set - RS

The combination of access types, `modifiedWriteValue` and `readAction` is used to specify the remaining access types: WSRC, WCRS, W1SRC, W1CRS, W0SRC, W0CRS, W0C, W0S.

For example:

```
<spirit:field>
    <spirit:name>interrupt</spirit:name>
    <spirit:bitOffset>0</spirit:bitOffset>
    <spirit:bitWidth>1</spirit:bitWidth>
    <spirit:access>read-write</spirit:access>
    <spirit:modifiedWriteValue>oneToSet</
     spirit:modifiedWriteValue>
    <spirit:readAction>clear</spirit:readAction>
</spirit:field>
```

The above specification results in the UVM access type `W1SRC`. Similarly, a combination of the three inputs covers all the access types defined by UVM. Any illegal combinations can be filtered out and default it to read-write.

Reserved and Parameters Attributes

The parameters attribute is supported as it does not have a respective context on the UVM REG side, and hence it is ignored with a warning. The reserved attribute is not supported.

Reset/Mask for Register

The reset values along with the mask for a field are specified only at the register level in IPXACT as shown in the following example. The mask defines which bit of the register has a known reset value.

IP-XACT specification:

```
<spirit:register>
     <spirit:name>srd_reg</spirit:name>
     <spirit:addressOffset>0xb</spirit:addressOffset>
     <spirit:size>8</spirit:size>
     <spirit:access>read-only</spirit:access>
     <spirit:reset>
          <spirit:value>0xa5</spirit:value>
          <spirit:mask>0xff</spirit:mask>
     </spirit:reset>
     <spirit:field>
          <spirit:name>rdata_msb</spirit:name>
          <spirit:bitOffset>4</spirit:bitOffset>
          <spirit:bitWidth>4</spirit:bitWidth>
     </spirit:field>
</spirit:register>
```

Equivalent RALF Specification

```
        register srd_reg @'hb {
            bytes 1;
            field rdata_msb @'h4 {
                bits 4;
                access ro;
                hard_reset 'ha;
            }
        }
```

Resetting the Values of the Field Using <ipxact:resets>

You can use the `<ipxact:resets>` construct to describe the reset values of the field. Each reset element defines the reset value for a given reset type. The two values of `resetTypeRef` supported by ralgen are: “`HARD`” and “`SOFT`”. If `resetTypeRef` is omitted in `<ipxact:reset>`, the default value of `HARD` is considered.

The following is an example of IP-XACT specification for `<ipxact:resets>`:

ipxact_resets.xml:

```
<ipxact:register>
  <ipxact:name>BasicRegister</ipxact:name>
  <ipxact:addressOffset>0x4</ipxact:addressOffset>
  <ipxact:size>8</ipxact:size>
  <ipxact:volatile>true</ipxact:volatile>
  <ipxact:access>read-writeOnce</ipxact:access>
  <ipxact:field>
    <ipxact:name>F1</ipxact:name>
    <ipxact:bitOffset>0</ipxact:bitOffset>
    <ipxact:resets>
    <ipxact:reset>
    <ipxact:value>0x0</ipxact:value>
    </ipxact:reset>
    <ipxact:reset resetTypeRef="SOFT">
    <ipxact:value>0xf</ipxact:value>
    <ipxact:mask>0xa</ipxact:mask>
    </ipxact:reset>
    </ipxact:resets>
    <ipxact:bitWidth>4</ipxact:bitWidth>
  </ipxact:field>
  <ipxact:field>
    <ipxact:name>F2</ipxact:name>
    <ipxact:bitOffset>0</ipxact:bitOffset>
    <ipxact:bitWidth>4</ipxact:bitWidth>
  </ipxact:field>
</ipxact:register>
```

The preceding IP-XACT code results in the following RALF:

ipxact_resets.ralf:

```
register BasicRegister @0x4 {
    bytes 1;
    field F1 @'h0 {
        bits 4;
        access rw;
        volatile 1
        hard_reset 'h0;
        soft_reset 'ha;
    }
    field F2 @'h0 {
        bits 4;
        access rw;
        volatile 1
    }
}
```

Volatile Construct

The `ralgen` utility supports the `<ipxact:volatile>` construct and it maps to the `volatile` construct field in RALF.

You can add the `<ipxact:volatile>` construct at field/register/ block level. If nothing is specified at block level, `false` value is considered. In case, nothing is specified at the register/field level, the value of the container block/register is inherited respectively. If the value is specified at register/field level, it overrides the inherited value.

You can see the final computed volatility at the field level as the value of volatile construct/bit is generated in the RALF/SystemVerilog code by `ralgen`.

Following is an example of IP-XACT specification for `<ipxact:volatile>`:

ipxact_volatile.xml:

```
<ipxact:addressBlock>
  <ipxact:name>VolatileAddressBlock</ipxact:name>
  <ipxact:baseAddress>0x0</ipxact:baseAddress>
  <ipxact:usage>register</ipxact:usage>
  <ipxact:volatile>true</ipxact:volatile>
  <ipxact:register>
    <ipxact:name>VolatileRegister</ipxact:name>
    <ipxact:addressOffset>0x4</ipxact:addressOffset>
    <ipxact:size>32</ipxact:size>
    <ipxact:field>
      <ipxact:name>NonVolatileField</ipxact:name>
      <ipxact:bitOffset>0</ipxact:bitOffset>
      <ipxact:bitWidth>16</ipxact:bitWidth>
      <ipxact:volatile>false</ipxact:volatile>
    </ipxact:field>
    <ipxact:field>
      <ipxact:name>VolatileField</ipxact:name>
      <ipxact:bitOffset>15</ipxact:bitOffset>
      <ipxact:bitWidth>16</ipxact:bitWidth>
    </ipxact:field>
  </ipxact:register>
  <ipxact:register>
    <ipxact:name>NonVolatileRegister</ipxact:name>
    <ipxact:addressOffset>0x8</ipxact:addressOffset>
    <ipxact:size>32</ipxact:size>
    <ipxact:volatile>false</ipxact:volatile>
    <ipxact:field>
      <ipxact:name>NonVolatileField</ipxact:name>
      <ipxact:bitOffset>0</ipxact:bitOffset>
      <ipxact:bitWidth>16</ipxact:bitWidth>
    </ipxact:field>
    <ipxact:field>
      <ipxact:name>VolatileField</ipxact:name>
      <ipxact:bitOffset>15</ipxact:bitOffset>
      <ipxact:bitWidth>16</ipxact:bitWidth>
      <ipxact:volatile>true</ipxact:volatile>
    </ipxact:field>
  </ipxact:register>
</ipxact:addressBlock>
```

The preceding IP-XACT code results in the following RALF:

ipxact_volatile.ralf

```
block VolatileAddressBlock @0x0 {
    register VolatileRegister @0x4 {
        bytes 4;
        field NonVolatileField @'h0 {
            bits 16;
            access rw;
        }
        field VolatileField @'hf {
            bits 16;
            access rw;
            volatile 1
        }
    }
    register NonVolatileRegister @0x8 {
        bytes 4;
        field NonVolatileField @'h0 {
            bits 16;
            access rw;
        }
        field VolatileField @'hf {
            bits 16;
            access rw;
            volatile 1
        }
    }
}
```

As illustrated in this example, the `VolatileAddressBlock` address block has explicit volatile specification set to `true`. Both `VolatileRegister` and `NonVolatileRegister` registers inherit `true` value, unless and until an explicit volatile tag is specified with the `false` value. In this case, the `NonVolatileRegister` register has such a specification.

Similarly, the fields inside `VolatileRegister` and `NonVolatileRegister` inherit volatile value as `true` and `false` respectively, unless an explicit specification is provided.

The `NonVolatileField` field inside `VolatileRegister` and `VolatileField` inside `NonVolatileRegister` explicitly specify the volatile value to override the inherited values.

Vendor Extensions

All Synopsys vendor extensions are located in the `<spirit:vendorExtensions>` element.

All Synopsys vendor extensions are further contained in a `<snps:spirit-element>` container. For example, all Synopsys vendor extensions to the `<spirit:register>` element are contained in a `<snps:register>` container element.

All other vendor extensions are ignored.

The vendor extensions make an IP-XACT specification inherently vendor-specific. If an IP-XACT specification contains vendor extensions from another vendor that specify the same information, it is necessary to first translate these third-party extensions into the equivalent Synopsys extensions.

Any IP-XACT-compliant tool should ignore the Synopsys extensions, the same way the Synopsys tools ignore the non-Synopsys extensions.

It is recommended that the equivalent Synopsys vendor extensions be added to the `<spirit:vendorExtensions>` element, thus maintaining the compatibility of the IP-XACT specification with its original third-party tool environment.

Example:

```
<spirit:vendorExtensions>
     <3rdparty:extension>
     ...
     </3rdparty:extension>
     <snps:extension>
     ...
     </snps:extension>
</spirit:vendorExtensions>
```

Reset Values

The reset override of a particular register in a register array or a particular field is specified using the `<snps:changeResets>` element inside the `addressBlock` and `regfile` tags.

Example:

```
<spirit:vendorExtensions>
<snps:changeResets>
<snps:changeReset>
<snps:name>COUNTERS[10]</snps:name>
<snps:value>0x1</snps:value>
</snps:changeReset>
</snps:changeResets>
</spirit:vendorExtensions>
```

HDL Paths

The HDL path added by a IP-XACT structural element is specified using the `<snps:hdl_path>` element.

Example:

```
<spirit:register>
<spirit:name>srd_reg</spirit:name>
<spirit:addressOffset>0xb</spirit:addressOffset>
<spirit:size>8</spirit:size>
<spirit:access>read-only</spirit:access>
<spirit:vendorExtensions>
<snps:register>
<snps:hdl_path>srd_reg</snps:hdl_path>
</snps:register>
</spirit:vendorExtensions>
<spirit:reset>
<spirit:value>0xa5</spirit:value>
<spirit:mask>0xff</spirit:mask>
</spirit:reset>
<spirit:field>
<spirit:name>rdata_lsb</spirit:name>
<spirit:bitOffset>0</spirit:bitOffset>
<spirit:bitWidth>4</spirit:bitWidth>
<spirit:vendorExtensions>
<snps:field>
<snps:hdl_path>rdata_lsb</snps:hdl_path>
</snps:field>
</spirit:vendorExtensions>
</spirit:field>
</spirit:register>
```

Exclusion from Pre-Defined Tests

To exclude a register from all the tests, all read tests or all write tests, the `<snps:csrSetting>` element is specified inside the `<snps:register>` element in the corresponding register. The content of this element must be `NO_CSR_TEST, NO_CSR_R_TEST` or `NO_CSR_W_TEST` respectively.

The mapping from the `NO_CSR_*TEST` to the UVM pre-defined sequences are as follows:

NO_CSR_TEST	NO_RAL_TESTS 1
NO_CSR_R_TEST	NO_BIT_BASH 1, NO_MEM_ACCESS 1, NO_MEM_WALK 1, NO_REG_ACCESS 1, NO_SHARED_ACCESS 1, NO_HW_RESET 1
NO_CSR_W_TEST	NO_BIT_BASH 1, NO_MEM_ACCESS 1, NO_MEM_WALK 1, NO_REG_ACCESS 1, NO_SHARED_ACCESS 1

Example:

```
<spirit:vendorExtensions>
<snps:register>
<snps:csrSetting>NO_CSR_TEST</snps:csrSetting>
</snps:register>
</spirit:vendorExtensions>
```

Non-Zero Start Array Index

If the `<sprit:dim>` element is used, an array indexed from `0` to `dim-1` is inferred. If a different starting index is desired, specify the bounds of the array index using the `<snps:x_from>` and `<snps:x_to>` elements within a `<snps:register>` element. The start and end index must be static and known at model generation time.

Example

```
<spirit:vendorExtensions>
<snps:register>
<snps:x_from>1</snps:x_from>
<snps:x_to>5</snps:x_to>
</snps:register>
</spirit:vendorExtensions>
```

User-Defined Array Address Stride

By default, the consecutive registers in a register array are located in consecutive address offsets to minimize the overall address space taken by the entire array and avoid unused addresses. If a different address stride is required, it can be specified using the `<snps:incr>` element within a `<snps:register>` element. The address stride value must be static and known at model-generation time.

This extension is only valid in `<snps:register>` element with a `<spirit:dim>` element or a `<snps:x_from>` and `<snps:x_to>` extensions.

Example:

```
<spirit:vendorExtensions>
<snps:register>
<snps:incr>4</snps:incr>
</snps:register>
</spirit:vendorExtensions>
```

Attributes

The RALF attributes are specified using the `<snps:attribute>` element within a `<snps:attributes>` extension as follows:

```
<spirit:vendorExtensions>
  <snps:attributes>
    <snps:attribute>
     <snps:name></snps:name>
     <snps:value></snps:value>
     <snps:type></snps:type>
    </snps:attribute>
    <snps:attribute>
    ….
    </snps:attribute>
  …
  </snps:attributes>
</spirit:vendorExtensions>
```

Where,

- `<snps:attribute>` can occur once or multiple times and each of the attribute container comprises of the following elements:
    - `<snps:name>` specifies the attribute name.
    - `<snps:value>` specifies the value to be dumped.
    - `<snps:type>` is an optional element and can accept one of the following three strings:

      `-string`

      `-integer`

      `-bit`

      If `<snps:type>` is not specified, it is assumed to be of `bit` type.

This vendor extension is supported inside `field`, `register` and `addressBlock`. The IP-XACT `vendorExtensions` map to the RALF attributes as follows:

```
attributes
{
name1 value1, name2, value2 …
}
```

Example:

```
<spirit:register>
    <spirit:name>PC</spirit:name>
    <spirit:addressOffset>0x8</spirit:addressOffset>
    <spirit:size>64</spirit:size>
    <spirit:access>read-write</spirit:access>
    <spirit:vendorExtensions>
           <snps:attributes>
               <snps:attribute>
                   <snps:name>NO_REG_TESTS</snps:name>
                   <snps:value>1</snps:value>
                   <snps:type>bit</snps:type>
               </snps:attribute>
           </snps:attributes>
</spirit:vendorExtensions>
...
</spirit:register>
```

The preceding IP-XACT code results in the following RALF:

```
register PC @0x8 {
     bytes 8;
     …
     attributes {
          NO_REG_TESTS 1
     }
}
```

User Code

You can use the `user_code` vendor extension to add SystemVerilog properties and code inside the generated RAL classes. For more details about how to use the `user_code` construct, see section "Inserting User-Defined Code Inside the Generated RAL Model Classes" .

You can specify user code using the `<snps:user_code>` element inside `addressBlock`, `register` and `regfile` tags.

The RALF construct is supported in IP-XACT via the following vendor extensions in address block/regfile/register:

```
<spirit:vendorExtensions>
<snps:user_codes>
<snps:user_code>
<snps:scope>new</snps:scope>
<snps:body>body_of_the_user_code</snps:body>
</snps:user_code>
</snps:user_code>
</spirit:vendorExtensions>
```

Using `<snps:scope>` is optional, if unspecified, it defaults to the scope of the class itself. Currently, `<snps:scope>` only accepts "`new`" as the scope name. If you specify this, then the corresponding code is placed in the build methods.

Register CallBack Class

The `register_cb_class` RALF construct is supported in IP-XACT via the following vendor extension. This extension is valid only in the top-level container tag of the xml (`ipxact:component/ipxact:memoryMaps`) tag.

```
<snps:register_cbs>
   <snps:register_cb>
     <snps:name>class_name <snps:name>
         <snps:var_declarations>
     Additional Class properties
         </snps:var_declarations>
         <snps:new_method>
           <snps:args>
          </snps:args>
           <snps:body>
          </snps:body>
         </snps:new_method>
         <snps:pre_write_method>
<snps:body>
Content for pre write method
```

```
            </snps:body>
        </snps:pre_write_method>
        <snps:post_write_method>
<snps:body>
Content for post write method
            </snps:body>
        </snps:post_write_method>
      <snps:post_read_method>
<snps:body>
Content for post read method
            </snps:body>
        </snps:post_read_method>
      <snps:pre_read_method>
<snps:body>
Content for pre read method
            </snps:body>
        </snps:pre_read_method>
          <snps:register_cb>
</snps:register_cbs>
```

This translates to the following code in RALF:

```
register_cb <class_name> {
  var_declarations {
}
  new_method {arg_list} {
//code to be exexcuted in new method
 }

pre_write_method {
//code to be exexcuted in pre_write method
}
post_write_method {
…
}

pre_read_method {
…
}
post_read_method {
…
}

}
```

Adding Register Callback

The `add_reg_cb` RALF construct is supported using the following vendor extensions. This extension is valid only in `ipxact:register`, `ipxact:addressBlock`, and `ipxact:registerFile` tags.

You can specify multiple `snps:add_reg_cb` tags inside `snps:add_reg_cbs`. Only one `snps:add_reg_cbs` for each register/registerFile/addressBlock is read.

The following is the syntax:

```
<snps:add_reg_cbs>
<snps:add_reg_cb>
<snps:cb_name>Callback class name</snps_cb_name>
<snps:args>new method args</snps:args>
</snps:add_reg_cb>
<snps:add_reg_cb>
....
</snps:add_reg_cb>
</snps:add_reg_cbs>
```

Inside the `ipxact:register` tag, the `reg_inst_name` keyword is not required, and the generated code uses the “`this`” keyword.

This translates to the following code in RALF:

```
add_reg_cb [register_instance_name] <cb_name> '(' args_to_new ')'
```

The `register_instance_name` is not generated for the `add_reg_cb` vendor extension placed inside the `ipxact:register` tag.

Limitations of IP-XACT to RALF Feature Mapping

The following are the limitations of IP-XACT to RALF feature mapping:

- The **`ralgen`** utility has no mapping for the IP-XACT memory schema features or syntax listed in Table 6-8.

Table 6-8 IP-XACT Memory Schema Features with No RALF Mapping

reserved
ref: parameters
values: value/name/description
suspaceMap
masterRef
nameGroup
coverage
endianess
virtual register

- There are some RALF features with no direct equivalence as yet in the IP-XACT 1.4 memory/registers schema. Table 6-9 lists the RALF syntax items that are not available in IP-XACT 1.4 syntax.

Table 6-9 RALF Features with No Direct IP-XACT 1.4 Equivalent

domain
initial
initial_value
reset_type
soft_reset
little

Table 6-9 RALF Features with No Direct IP-XACT 1.4 Equivalent

big
fifo_ls
fifo_ms
endian
endian_value

- Nested registerFile is not supported.

7

UVM Register C++ Interface

The UVM register C interface allows firmware and application-level code to be developed and debugged on a simulation of the design. For runtime performance reasons, only the lower layers of an application are simulated.

You can access the fields, registers, and memories included in a UVM register model in C code through C API. The C code is executed natively on the same workstation that is running the SystemVerilog simulation, eliminating the need for an instruction set simulator or a RTL model of the processor. You can compile the same C code later for the target execution processor.

The C++ interface is made visible by including the following file in any C++ source file accessing registers in your design:

```
#include "snps_reg_rw_api.h"
```

The API defines the following set of functions to read and write registers, overloaded for different register sizes.

```
namespace snps_reg {
   inline volatile uint8 regRead(volatile uint8  *addr);
   inline volatile uint16 regRead(volatile uint16 *addr);
   inline volatile uint32 regRead(volatile uint32 *addr);

  inline volatile void regWrite(volatile uint8 *addr, uint8 
val);
   inline volatile void regWrite(volatile uint16 *addr, 
uint16 val);
   inline volatile void regWrite(volatile uint32 *addr, 
uint32 val);
}
```

There are two versions of the UVM register C++ API that can be used. One is designed to interface to the UVM register model running in the SystemVerilog simulator using the Direct Programming Interface. The other is pure stand-alone C++ code and is designed to be compiled on the target processor in the final application. This allows the firmware and application-level code to be verified against a simulation and then used, unmodified, in the final application. The version of the C++ API that is used is determined at compile time by including the `snps_reg_rw_api.h` file from one of the two directories.

To compile your C++ code for execution on the target processor, use pure C++ code API by specifying the following compile-time options:

```
% g++ -c -I$UVM_HOME/include/pureC …
```

To compile your C++ code for execution on the host computer and co-simulation with the UVM register model, use DPI C++ code API by specifying the following compile-time options:

```
% g++ -c -I$UVM_HOME/include/uvmC …
```

The types `uint8`, `uint16` and `uint32` are of course machine-dependent and must be defined before including the `snps_reg_rw_api.h` file. For your convenience, a set of default type definitions are provided in the file `$UVM_HOME/include/snps_reg_uints.h`.

C++ Register Model

Ralgen creates a hierarchical model of the registers found in the design and accessible through a specific address map.

```
% ralgen …
```

A class is defined for every structural component in the register specification. Each class contains instances of lower-level structural components and a method of returning the address of every register it contains. If a field within a register is the sole field in its byte lane, a method returning the address of that field also exists. To specify the address of the register or field to access, call its corresponding method through a hierarchical reference in the register model.

```
reqs = snps_reg::regRead(usbdev.status());
snps_reg::regWrite(usbdev.intrMask(), 0xFFFF);
```

The device driver code should be written in functions accepting a reference to the register model corresponding to the device. The register model is then used to identify the registers to be accessed.

```
int
usb_dev_isr(usbdev_t &dev)
{
   int reqs = snps_reg::regRead(dev.status());
   regWrite(dev.status(), reqs);
   if (reqs & 0x0001) usb_dev_tx_rdy(dev);
   if (reqs & 0x0002) usb_dev_rx_rdy(dev);
}
```

The C++ register model is limited to registers that can be accessed using a single read or write operation with a 32-bit data bus, which means that registers are limited to 32 bits. If the architecture of the processor and implementation of the device supports byte-level access, individual bytes and words are accessible as fields within a register.

For example, the following register specification

```
block comp1 {
   bytes 4;

   register regA @ 0x00 {
      field data { bits 32; }
   }

   register regB @ 0x04 {
      field fldA { bits 8; }
      field fldB { bits 8; }
      field fldC { bits 16; }
   }
   register regC @ 0x08 {
      field fldA { bits 16; }
      field fldD { bits 8; }
   }
}
```

yields the following C++ register model:

```
class comp1_t
{
   public:
      inline volatile uint32 *regA();
      inline volatile uint32 *regB();
      inline volatile uint8  *fldA();
      inline volatile uint8  *fldB();
      inline volatile uint16 *fldC();
      inline volatile uint32 *regC();
      inline volatile uint16 *regC_fldA();
      inline volatile uint8  *fldD();
}
```

Instantiating the Register Model

Before the device driver code can be invoked, an instance of the register model must exist. The register model being instantiated depends on whether the device driver code is called by the target application or by the UVM simulation.

When using the device driver code in the target application, the target application code must instantiate the register model, specifying the base address of the device in question. Multiple register models may be instantiated.

```
usbdev_t usb0("usb0", 0x100000);
usbdev_t usb1("usb1", 0x110000);

int
main(char **argv, int argc)
{
   ...
}
```

When using the device driver code from the UVM simulation, it is necessary for the C++ code to be called by the simulation to be executed. The application software's main() routine must be replaced by one or more entry points known to the simulation through the DPI interface. The DPI-C entry point creates an instance of the register model based on the context specified by the UVM simulation.

```
extern "C" int
usb_dev_isr_entry(int context)
{
   usbdev_t usb(context);
   return usb_dev_isr(usb);
}
```

The C++ code can then be called from UVM simulation by calling its corresponding entry point and specifying the context of the register model. You must also include the `snps_reg.svh` file that is shipped with UVM libraries present in VCS installation directory.

```
‘include "vcs/snps_reg.svh"
...
import "DPI-C" function int usb_dev_isr_entry(int ctxt);
...
ral_sys_soc soc = new("soc", 'h10000);
soc.build();
...
usb_dev_isr_entry(snps_reg::create_context(soc.usb0));
```

Retrieving IDs of All the Registers in a Block/System

In uvmC and pureC flow, you can use the following API to retrieve the list of all the registers present in a block or a system:

```
int getRegisters(snps_reg::reg_add**regs, int hier = 1)
```

Here,

`regs:` It is allocated and populated with the register information by the API. Each element of this array corresponds to a register and can be used directly as an argument to register access functions.

`hier:` If specified as 1, it recursively includes the registers in sub-blocks. If specified as 0, it limits the results returned to the block on which it is invoked.

Note:

> This API function is added in the `regmodel` base class, and the functionality is guarded under the `SNPS_REG_ENABLE_REG_ITER` pre-processor directive. To enable this functionality, you must use the `-DSNPS_REG_ENABLE_REG_ITER` option for C compilation.

Following is an example that illustrates the usage of these functions:

```
comp1_t dev(context);
snps_reg::reg_addr * regs;
int count;

count = dev.getRegisters(&regs);

for (int ind = 0; i < count; i++) {
    regWrite(regs[ind], <some_value>);
}
```

Co-Simulation Execution Timeline

When executing with a simulation of the design, all C++ code executes atomically. It is unlike the real application code running as object code on a real processor, where the execution of the code happens concurrently with other processing in the neighboring hardware.

When the C++ code executes, only the code performs any form of processing and the simulation of the rest of the design is frozen. The only way for the design simulation to proceed, is for the C++ code to return, or for the C code to perform a read or write operation through the register model. In the latter case, once the read or write operation completes and the control is returned back to the C code, the simulation is again frozen.

The entire execution timeline in the C++ code thus occurs in zero-time in the simulation timeline. This has an important impact on runtime performance of how the C++ code interacts with the design.

If a polling strategy is used, the simulation will have the opportunity to advance only during the execution of the repeated polling read cycles. It would likely require many hundreds of such read cycles for the design to reach a state that is relevant and significant for the application software. With a physical device, this can happen in less than a microsecond. However, in a simulation, this would require a lot of processing for simulating essentially useless read cycles and exchanging data between the C++ world and the simulation world.

If an interrupt-driven strategy is used, the simulation will proceed until something of interest to the application software has happened before transferring control to the C++ code and only the necessary read and write operations needs to be performed. Therefore, it is important that you use a service-based approach as much as possible.

It is also very important that the execution of the C++ code not be blocked by an external event such as waiting for user input or a file to be unlocked as it prevents the simulation from moving forward while it is blocked. If the application software requires such synchronization, it should similarly use an asynchronous interrupt-driven approach.

A

RALF Syntax

A RALF description is a Tcl 8.5 file. Therefore, it is possible to use programming constructs such as loops and variables to rapidly and concisely construct large register sets and memory definitions. You can also use the Tcl `source` command to perform multiple and hierarchical register specification management. Also, you can use Tcl expressions to specify register offset values, base values and register names.

The semi-colon is used as a separator and is not necessary immediately after or before a closing curly brackets.

This appendix contains the following topics:

Grammar Notation

The following notations are used to specify the exact syntax of RALF descriptions:

`normal`	Literal items
italics	User-specified identifiers
[...]	Optional items
<...>	Repeated items, 1 to *N* times
[<...>]	Optional repeated items, 0 to *N* times
...|...	A choice of items

This section contains the following topic:

- "Reserved Words"

Reserved Words

In addition to the SystemVerilog and OpenVera reserved words, the following words are reserved and cannot be used as user-defined identifiers:

access	field	regfile
bits	hard_reset	register
block	hdl_path	reset
bytes	initial	shared
constraint	left_to_right	size
doc	memory	soft_reset
domain	noise	system
endian	read	virtual write
		write

Useful Tcl Commands

Considering a RALF description is a Tcl file, the full power of the Tcl language becomes available. The following Tcl commands are likely to be useful:

`#`*`comment`*

Indicates single-line comments with characters following a # considered as comments.

`set` *`name value`*

Sets the specified variable to the specified value. Allows the use of variable names as mnemonics, using Tcl syntax to set and get variable values.

`source` *`filename`*

Includes the specified Tcl file. Inclusion of files enable hierarchical RALF descriptions. The filename can have an absolute path or relative path.

```
for {set i 0} {$i < 10} {incr i} {
      ...
   }
```

For loops can be used to concisely create multiple fields, registers, memories and blocks specifications. Any RALF property value can be based on the value of the loop index variable or other variables.

```
if {$var} {
      ...
   }
```

Conditionally interprets Tcl statements or RALF specifications. Allows the selection or exclusion of elements in a RALF description.

You can view a complete list of available Tcl commands by visiting the following web address:

http://www.tcl.tk/man/tcl8.5/TclCmd/contents.htm

This section contains the following topic:

- "Tcl Syntax and FAQ"

Tcl Syntax and FAQ

The Tcl syntax rules can be found by visiting the following web address:

http://www.tcl.tk/man/tcl8.5/TclCmd/Tcl.htm

Note that ralgen preprocesses the RALF file to escape some of its syntax elements that have special meaning in Tcl. For example, the `[` and `]` used to specify arrays are properly escaped to avoid command substitution.

Whitespace

It is important to note how Tcl breaks a command into separate words on whitespaces, quoted (") and bracketed ({ and }) text. Therefore, a RALF file is sensitive to whitespace. Do not use whitespace in your code if none is shown in this appendix. Where a whitespace is shown, at least one must be present. For example, the following syntax is invalid because the `{` is considered as part of the *`field`* command's second argument and not a separate token:

```
## This is wrong
   field REVISION_ID @2{
      bits 8;
   }
```

This example is valid because a space is required to separate the `{` from the preceding Tcl command argument:

```
## This is right
   field REVISION_ID @2 {
      bits 8;
   }
```

Trailing Comments

A common mistake occurs when trying to add a trailing comment to a RALF construct using the following (erroneous) syntax:

```
register my_reg {
      ...
   } # my_reg
```

Considering that Tcl commands terminate at the end-of-line, the trailing comment is considered part of the register command. To have the trailing comment be properly interpreted as a comment, the previous Tcl command should be explicitly terminated with a semicolon, as shown in the following (correct) syntax:

```
register my_reg {
      ...
   }; # my_reg
```

RALF Construct Summary

- `field`
- `register`
- `regfile`
- `memory`
- `virtual register`
- `block`
- `system`

field

A field defines an atomic set of consecutive bits. Fields are concatenated into registers.

Syntax

```
field name [{
   <properties>
}]
```

Defines a field with the specified name. If you specify the name `unused` or `reserved`, it specifies unused or reserved bits within a register and you can specify only the `bits` property. Unused bits are assumed to be read-only and have a permanent value of zero. If another behavior is expected of unused or reserved bits, such as a different read-back value, you must specify an explicit field for them.

Properties

The following properties can be used to specify the field;

`[bits n;]`

Specifies the number of bits in the field. If not specified, defaults to 1. This property can only be specified once.

```
[access    rw|ro|wo|w1|w1c|rc|rs|wrc|wrs|wc|ws|
           wsrc|wcrs|w1s|w1t|w0c|w0s|others...
```

Specifies the functionality of all the bits in the field when the field is written or read.

By default, a field is writeable (`rw`).

A field can be,

rw	read/write
ro	read-only
wo	write-only
w1	write-once
w1c	write a 1 to bitwise-clear
rc	clear on read
rs	Read Sets All
wrc	Write Read Clears All
wrs	Write, Read Sets All
wc	Write Clears All
ws	Write Sets All
wsrc	Write Sets All, Read Clears All
wcrs	Write Clears All, Read Sets All
w1s	Write 1 to Set If the bit in the written value is a '1', then the corresponding bit in the field is set to 1. Otherwise, the field bit is not affected.
w1t	Write 1 to Toggle If the bit in the written value is a '1', then the corresponding bit in the field is inverted. Otherwise, the field bit is not affected.

w0c	Write 0 to Clear If the bit in the written value is a '0', then the corresponding bit in the field is set to 0.Otherwise, the field bit is not affected.
w0s	Write 0 to Set If the bit in the written value is a '0', then the corresponding bit in the field is set to 1. Otherwise, the field bit is not affected.
w0t	Write 0 to Toggle If the bit in the written value is a '0', then the corresponding bit in the field is inverted. Otherwise, the field bit is not affected.
w1src	Write 1 to Set, Read Clears All If the bit in the written value is a '1', then the corresponding bit in the field is set to 1. Otherwise, the field bit is not affected.
w1crs	Write 1 to Clear, Read Sets All If the bit in the written value is a '1', then the corresponding bit in the field is set to 0. Otherwise, the field bit is not affected.
w0src	Write 0 to Set, Read Clears All If the bit in the written value is a '0', then the corresponding bit in the field is set to 1. Otherwise, the field bit is not affected.
w0crs	Write 0 to Clear, Read Sets All If the bit in the written value is a '0', then the corresponding bit in the field is set to 0. Otherwise, the field bit is not affected.
woc	Write Only Clears All
wos	Write Only Sets All
wo1	Write Only, Once Changed to written value if this is the first write operation after a hard reset. Otherwise has no effect.

`[reset|hard_reset` *`value`*`;]`

Specifies the hard reset value for the field. By default, a value of 0 is used.

Supports unknown (x or X) and high-impedance (z or Z) bits in *value*. However, such bits are eventually converted to 0 in the RAL Base Class because the reset *value* in the RAL Base Class is a 2-state value.

`[soft_reset` *`value`*`;]`

Specifies the soft reset value for the field. By default, a field is not affected by a soft reset.

Supports unknown (x or X) and high-impedance (z or Z) bits in *value*. However, such bits are eventually converted to 0 in the RAL Base Class because the soft reset *value* in the RAL Base Class is a 2-state value.

```
[<constraint name [{
     <expressions>
}]>]
```

Specifies constraints on the field value when it is randomized. The constraints are not interpreted by the generation script and must be valid SystemVerilog or OpenVera expressions. The identifier `value` is used to refer to the value of the field.

If a `constraint` property is not specified, the field cannot be randomized. If an unconstrained but random field is required, simply specify an empty constraint block.

```
[enum { <name[=val],> }]
```

Defines symbolic names for field values. If a value is no explicitly specified for a symbolic name, the value is the value of the previous name plus one—or zero if it is the first name.

```
[cover <+|- b|f>
```

Specifies if the bits in this fields are to be included (+b) in or excluded (-b) from the register-bit coverage model.

Specifies if the field value coverage point for this field is an explicit goal (+f), in which case its weight will be equal to the number of specified or implicit bins. If it is specified as an implicit goal (-f) as part of a cross-coverage point, its coverage point weight will be equal to zero.

```
[<coverpoint {
   <bins name [[[n]]] = { <n|[n:n],> } | default>
}>]
```

Explicitly specifies the bins in the field value coverpoint for this field. The semantics of the bin specification is identical to the SystemVerilog coverage bin specification, as defined in the section named “*Defining coverage points*” in the 1800-2009 SystemVerilog Language Reference Manual.

Example

Example A-1   1-bit read/write Field

```
field tx_en;
```

Example A-2   2-bit Randomizable Field

```
field PAR {
   bits 2;
   reset 2’b11;
   constraint valid {
      value != 2’b00;
   }
}
```

Example A-3   Explicitly specified coverage bins

```
field f2 {
     bits 8;
     enum { AA, BB, CC=15 }
     coverpoint {
         bins AAA     = { 0, 12 }
         bins BBB []  = { 1, 2, AA, CC }
         bins CCC [3] = { 14,15, [ BB : 10 ] }
         bins DDD     = default
      }
   }
```

register

A register defines a concatenation of fields. Registers are used in register files and blocks.

Syntax

```
register name {
   <properties>
}
```

Defines a register with the specified name.

Properties

The following properties can be used to specify the register.

```
[attributes {
      <name> <value>[, ...]
   }]
```

Specifies a value for the specified user-defined attribute. Multiple attributes may be specified by separating each attribute-value pair with a comma. If the value contains white spaces, it must be included between double quotes.

```
[bytes n;]
```

Specifies the number of bytes in the register. The total number of bits in the fields in this register cannot exceed this number of bytes. If this property is not specified, the width of the register is the minimum integral number of bytes necessary to implement all fields contained in the register.

`[left_to_right;]`

By default, fields are concatenated starting from the least-significant bit of the register. If this property is specified, fields are concatenated starting from the most-significant side of the register, but justified to the least-significant side. When using a left-to-right specification style, the first field cannot have a bit offset specified: the offset of the first field will depend on the size of and spacing between the other fields.

```
[<field name[=rename][[n]] [(hdl_path)]
   [@bit_offset[+incr]];
```

```
[<field name [[n]] [(hdl_path)]
   [@bit_offset[+incr]] {
    <field properties>
   }>]
```

Defines and instantiates the specified field in this register. The first form specifies an instance of a previously-defined field description. The second form defines a new field description and instantiates it in the register file.

Fields separated by unused or reserved bits can be separated by specifying a field named `unused` or `reserved` of the appropriate width or by using a bit offset. A bit offset, from the least-significant bit in the register can be specified. If no bit offset is specified, the field is located immediately to the left (or right if the `left_to_right` property is specified) of the previously instantiated field. If the numerical index n is specified, an array of fields is instantiated.

Field array elements are located at consecutive offsets in the register, starting with `field[0]`, separated by a specified offset increment. The offset increment is only valid when instantiating a field array.

Instantiating an array of fields is logically equivalent to explicitly instantiating all the individual fields. The only difference is that they will be accessible as an array in the generated SystemVerilog code.

By default, the location of the low-index field will be in the LSB (least significant bit) position. If the `left_to_right` attribute is specified for the instantiating register, the low-index field is in the MSB (most significant bit) position.

A field array is generated into a fixed-sized array of `uvm_ral_field` instances in the `uvm_ral_reg` and `uvm_reg_block` class extensions using the same naming convention as a regular field. The array is populated with individual `uvm_ral_field` class instances, one per array element, appending `[%0d]` to the field name (where`%0d` is replaced with the field index). Each instance is registered with the parent register abstraction class as if they were individually-specified fields.

Arrays of fields can be interspersed with other arrays of fields or regular fields, as long as the field themselves do not overlap.

The optional `(hdl_path)` is the hierarchical reference, within the register, to the HDL structure implementing the field. If an `(hdl_path)` is specified, direct hierarchical access to the field can be automatically generated by concatenating it with the `(hdl_path)` of the enclosing register. The `(hdl_path)` can be an expression and it must be enclosed between parentheses.

By default, the bit offset represents the position of the least-significant bit of the field with respect to the least-significant bit of the register. A value of 0 indicates a field starting in the least-significant bit of the register. If the `left_to_right` property is specified, the bit offset is specified as the offset of the most-significant bit of the field from the most-significant used bit in the register. The position of the most-significant used bit in the register, is a function of the size of, and spacing between all specified fields as fields are always left-justified, even when specifying a left-to-right order.

You must specify at least one `field` property.

Any gap in the register before and after fields is assumed to be made of unused bits that are read-only and have a permanent value of zero. If another behavior is expected from unused or reserved bits, an explicit field must be specified for them.

```
[<constraint name [{
      <expression>
   }]>]
```

Specifies constraints on the value of the fields it contains when it is randomized. The constraints are not interpreted by the generation script and must be valid SystemVerilog or OpenVera expressions. The identifier `fieldname.value` refers to the value of a field.

```
[noise ro|rw|no;]
```

Specifies if and how this register can be accessed during normal operations of the design without affecting the configuration or functional correctness of the device. By default, a register can be read at any time (`ro`). If `rw` is specified, this register can also be written. If `no` is specified, this register cannot be accessed in any way during normal operations. Currently unsupported.

`[shared [(`*`hdl_path`*`)];]`

Specifies that this register is physically shared by all domains in a block that instantiates it. This property can only be used in a stand-alone register specification.

The `(hdl_path)` specifies the hierarchical access path to the physical register. It is used instead of the `(hdl_path)` specified in the block instantiating it. If an `(hdl_path)` is specified, direct hierarchical access to the shared register can be automatically generated by concatenating it with the `(hdl_path)` of the enclosing block. The `(hdl_path)` must be enclosed in parentheses.

`[cover <+|- a|b|f>`

Specifies if the address of this register should be excluded (-a) from the block’s address map coverage model.

Specifies if the bits in this register are to be included (+b) in or excluded (-b) from the register-bit coverage model.

Specifies if the fields in this registers should be included (+f) in or excluded (-f) from the field value coverage model.

cross <cross_item1> <cross_item2> [<cross_item3> …

<cross_itemN>] [{

label <cross_label_name>

}]

Specifies a cross coverage point of two or more fields or of any previously defined cross coverage point. To use a previously defined cross coverage point in another cross coverage specification, the specification of the former cross coverage point must have a label, so that it can be referenced in a later cross coverage specification, if needed by using that label.

<cross_itemN> can be, either a previously defined nonarray field name or a previous defined <cross_label_name>. For field arrays, <cross_itemN> will need to specify the exact field (array element) to be used for calculating the cross, using <fieldarray-name>[<index>] syntax, where <index> will range from 0 to field array size - 1.

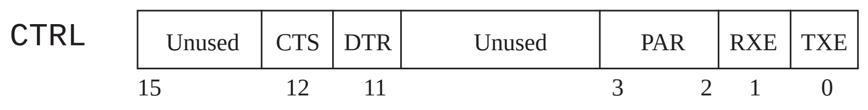
Example

Example A-4 Attribute specification for a register

```
register R {
   ...
   attributes {
      NO_RAL_TESTS 1,
      RETAIN       1
   }
}
```

The following examples are different ways to specify the register illustrated in Figure A-1.

Figure A-1 Register Specification



Example A-5 Specification for Register in Figure A-1

```
register CTRL {
   field TXE {}
   field RXE {}
   field PAR {
      bits 2;
      reset 2’b11;
   }
   field DTR @11 {
      access rw;
   }
   field CTS {
      access rw;
      reset 1;
   }
}
```

Example A-6 Specification for register in Figure A-1

```
source Example A-2
register CTRL {
   bytes 2;
   left_to_right;
   field CTS {
      access rw;
      reset 1;
   }
   field DTR {
      access rw;
   }
   field unused {
      bits 7;
   }
   field PAR;
   field RXE {}
   field TXE {}
}
```

Example A-7   User-defined cross-coverage point

```
register r {
    field f1 {...}
    field f2 {...}
    field f3 {...}

    cross f1 f2 {
        label xyz;
    }
    cross xyz f3;
}
```

regfile

A register file defines a collection of consecutive registers. Register files are used in blocks.

Syntax

```
regfile name {
   <properties>
}
```

Defines a register file with the specified name.

Properties

The following properties can be used to specify the register file.

```
[<register name[=rename][[n]] [(hdl_path)]
   [@offset] [read|write];>]
```

```
[<register name[[n]] [(hdl_path)] [@offset] {
      <property>
   }]
```

```
[<register name[=rename][m:n]] [(hdl_path)]
   [@offset] [read|write];>]
```

```
[<register name[[m:n]] [(hdl_path)] [@offset] {
     <property>
   }]
```

The first form specifies an instance of a previously-defined register description. The second form defines a new register description and instantiates it in the register file. An inlined register description cannot contain the `shared` property. Access to a shared register can be further restricted to read or write in a particular instance.

A register may be instantiated at an explicit address offset within the register file. If not specified, the register is instantiated at the next available address, starting with 0. The number of addresses occupied by a register depends on the width of the register and the endian property of the block defining the register file. If a register is not mapped in the address space of the block, the offset may be specified as `@none` to indicate that the register does not consume any address locations.

If a numerical index is specified, an array of registers is instantiated. Register arrays are located at consecutive address offsets. Register array declared using notation `[n]` will be starting from address 0 and will have word addresses from 0 to n-1. Instantiating an array of register is logically equivalent to explicitly instantiating all of the individual registers explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

The optional (`hdl_path`) is the hierarchical reference, within the block, to the HDL structure implementing the register. If an (`hdl_path`) is specified, direct hierarchical access to the register can be automatically generated by concatenating it with the (`hdl_path`) of the enclosing system and any HDL expression specified in the register. The (`hdl_path`) can be an expression and it must be enclosed between parentheses. (`hdl_path`) can be a simple name or a Verilog port like expressions for associating to fields as shown in the following example:

```
register reg=reg_inst (.fld1(reg.fld1),
.fld2(reg.fld2), ….);
```

The (`hdl_path`) for a register array must include a %d placeholder that will be replaced with the decimal index of the register in the array.

If more than one register with the same name is instantiated in the same register file, it must be renamed to a unique name within the register file.

You must specify at least one register property.

```
[<constraint name [{
     <expression>
  }]>]
```

Specifies constraints on the value of the registers and fields it contains when it is randomized. The constraints are not interpreted by the generation script and must be valid SystemVerilog or OpenVera expressions.

```
[shared [(hdl_path)];]
```

Specifies that this register file is physically shared by all domains in a block that instantiates it. This property can only be used in a stand-alone register file specification.

The `(hdl_path)` specifies the hierarchical access path to the shared register file. For shared register files, this `(hdl_path)` is used, instead of the `(hdl_path)` specified, if any, while instantiating the register file in a block. If an `(hdl_path)` is specified, direct hierarchical access to the shared register file can be automatically generated by concatenating it with the `(hdl_path)` of the enclosing block. The `(hdl_path)` must be enclosed in parentheses.

All the registers instantiated inside a shared register file must also be shared.

```
[cover <+|- a|b|f>
```

Specifies if the registers in this register file are to be included (+) in or excluded (-) from the address map (a), register bits (b) or field value (f) coverage model.

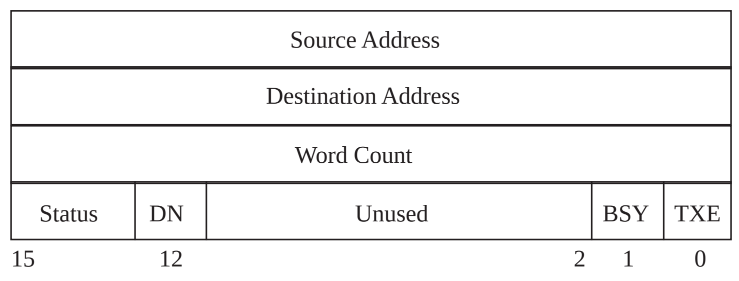
```
[doc {
      <text>
   }]
```

Specifies user documentation for the register file, using HTML formatting tags. Currently unsupported.

Example

The primary purpose of register files is to define arrays of groups of registers. For example, the register group illustrated in Figure A-2 is used to configure a DMA channel. The block RALF specification shown in Example A-8 illustrates how a 16-channel DMA controller might be described.

Figure A-2   DMC Channel Configuration Registers Specification



Example A-8   Specification for multi-channel DMA controller

```
block dma_ctrl {
   regfile chan[16] {
      register src {
         bytes 2;
         field addr {
            bits 16;
         }
      }
      register dst {
         bytes 2;
         field addr {
            bits 16;
         }
      }
      register count {
         bytes 2;
         field n_bytes {
            bits 16;
         }
      }
      register ctrl {
         bytes 2;
         field TXE {
            bits 1;
            access rw;
         }
         field BSY {
            bits 1;
            access ro;
         }
         field DN @12 {
            bits 1;
            access ro;
         }
         field status {
            bits 3;
            access ro;
         }
      }
   }
}
```

memory

A memory defines a region of consecutively addressable locations. Memories are used in blocks.

Syntax

```
memory name {
   <property>
}
```

Defines a memory with the specified name.

Properties

The following properties can be used to specify the memory.

```
[attributes {
      <name> <value>[, ...]
   }]
```

Specifies a value for the specified user-defined attribute. Multiple attributes may be specified by separating each attribute-value pair with a comma. If the value contains white spaces, it must be included between double quotes.

`size` *m*[k|M|G]`;`

Specifies the number of consecutive addresses in the memory where each location has the number of bits specified by the ***`bits`*** property. The size may also include a unit. In that case, the specified size is multiplied by:

- 1024 (k)
- 2^20 (M)
- 2^30 (G)

This property is required.

`bits` *`n`*`;`

Specifies the number of bits in each memory location. The total number of bits in the memory is the specified number of bits multiplied by the specified size. This property is required.

`[access rw|ro;]`

Specifies if the memory is a RAM (`rw`) or a ROM (`ro`). By default, a memory is a RAM.

`[initial x|0|1|addr|`*`literal`*`[++|--];]`

Specifies the initial content of the memory is to be filled with unknowns (`x`), filled with zeroes (`0`), filled with ones (`1`), set to the physical address value (`addr`), or set to a constant (*`literal`*), incrementing (*`literal`*`++`) or decrementing (*`literal`*`--`) literal value.

The content of the memory is initialized to the specified pattern when the `uvm_ral_mem::initialize()` method in its abstraction class is invoked. By default, a memory is initialized with unknowns (x).

Initialization requires that backdoor access to the memory content be available.

`[shared [(`*`hdl_path`*`)];]`

Specifies that this memory is physically shared by all domains in a block that instantiates it. Can only be used in a standalone memory specification.

The optional `(hdl_path)` specifies the hierarchical access path to the physical memory. It is used in lieu of the `(hdl_path)` specified in the block instantiating it. If an `(hdl_path)` is specified, direct hierarchical access to the shared memory can be automatically generated by concatenating it with the `(hdl_path)` of the enclosing block. The `(hdl_path)` must be enclosed between parentheses.

`[cover <+|- a>`

Specifies if this memory is to be included (+a) in or excluded (-a) from the address map coverage model.

Example

Example A-9   64 KB RAM

```
memory dma_bfr {
   bits 8;
   size  64k;
}
```

Example A-10 2 KB ROM

```
memory tx_bfr {
   bits    16;
   size    1024;
   access  ro;
   initial 0++;
}
```

virtual register

A `virtual register` defines a concatenation of virtual fields. Virtual registers are used in blocks.

Syntax

```
virtual register name {
   <properties>
}
```

Defines a virtual register with the specified name.

Properties

The following properties can be used to specify the virtual register.

`[bytes n;]`

Specifies the number of bytes in the register. The total number of bits in the fields in this register cannot exceed this number of bytes. The actual number of memory locations used by the virtual register is the minimum integral number of memory locations required to provide the specified number of bytes. If this property is not specified, the width of the register is the minimum integral number of memory locations necessary to implement all fields contained in the register.

`[left_to_right;]`

By default, fields are concatenated starting from the least-significant bit of the register. If this property is specified, fields are concatenated starting from the most-significant side of the register but justified to the least-significant side. When using a left-to-right specification style, the first field cannot have a bit offset specified: the offset of the first field will depend on the size of, and spacing between, the other fields.

```
[<field name[=rename] [@bit_offset];

[<field name [@bit_offset] {
     bits n;
     [doc {
        <text>
     }]
  }>]
```

Defines and instantiates the specified virtual field with the specified number of bits in this virtual register. The first form specifies an instance of a previously-defined field description where only the `bits` property is considered (all other properties are ignored). The second form defines a new field description and instantiates it in the register file.

Refer to the specification of the `field` property in the "register" construct for more details on how they are physically laid out.

At least one `field` property must be specified.

All bits in a virtual register, including unused and reserved bits have their access modes defined by the access mode of the underlying memory used to implement it and the domain used to access them.

block

A block defines a set of registers and memories. Registers are concatenated into blocks. A block can have more than one physical interface. Registers and memories can be shared across physical interfaces within a block.

Syntax

```
block name {
   <property>
}
```

Specifies a design block with the specified name and a single physical interface.

```
block name {
   domain name {
      <property>
   }
   <domain name {
      <property>
   }>
   [doc { <text> }]
}
```

Specifies a design block with the specified name and multiple physical interfaces. The name of each domain specifies the name of the corresponding physical interface. At least two domains must be specified. This form of the block specification can have a `doc` property outside of the `domain` specification.

The name of the block is used to generate block-specific unique identifiers.

Properties

The following properties can be used to specify the block and its domains.

```
[attributes {
      <name> <value>[, ...]
   }]
```

Specifies a value for the specified user-defined attribute. Multiple attributes may be specified by separating each attribute-value pair with a comma. If the value contains white spaces, it must be included between double quotes.

bytes *n*;

Specifies the number of bytes that can be accessed concurrently and uniquely addressed through the physical interface. This property is required.

[endian little|big|fifo_ls|fifo_ms;]

Specifies how wider registers and memories are mapped onto multiple accesses over the physical interface. See "Hierarchical Descriptions and Composition" for a description of the various mapping modes. By default, little endian is used.

```
[<register name[=rename][[n]] [(hdl_path)]
   [@offset] [read|write];>]
```

```
[<register name[[n]] [(hdl_path)] [@offset] {
      <property>
   }]
```

```
[<register name[=rename][m:n]] [(hdl_path)]
   [@offset] [read|write];>]
```

```
[<register name[[m:n]] [(hdl_path)] [@offset] {
      <property>
    }]
```

The first form specifies an instance of a previously-defined register description. The second form defines a new register description and instantiates it in the block. An inlined register description cannot contain the `shared` property. Access to a shared register can be further restricted to read or write in a particular instance.

A register may be instantiated at an explicit address offset within the block. If not specified, the register is instantiated at the next available address, starting with 0. The number of addresses occupied by a register depends on the width of the register and the endian property of the block. If a register is not mapped in the address space of the block the offset may be specified as `@none` to indicate that the register does not consume any address locations.

If a numerical index is specified, an array of register is instantiated. Register arrays are located at consecutive address offsets. Register array declared using notation [n] will be starting from address 0 and will have word addresses from 0 to n-1. If an increment value is specified, the offset of each register in the register array is incremented by the specified increment. Instantiating an array of register is logically equivalent to explicitly instantiating all of the individual registers explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

The optional (`hdl_path`) is the hierarchical reference, within the block, to the HDL structure implementing the register. If an (`hdl_path`) is specified, direct hierarchical access to the register is automatically generated by concatenating it with the (`hdl_path`) of the enclosing system and any (`hdl_path`) expression specified in the register. The (`hdl_path`) can be an expression and it must be enclosed between parentheses. (`hdl_path`) can be a simple name or a Verilog port like expressions for associating to fields as shown in the following example:

```
register reg=reg_inst (.fld1(reg.fld1),
.fld2(reg.fld2), ….);
```

The (`hdl_path`) for a register array must include a "%d" placeholder that will be replaced with the decimal index of the register in the array.

If more than one register with the same name is instantiated in the same block, it must be renamed to a unique name within the block.

You must specify at least one register or memory property.

Registers must have unique addresses, therefore, it is not possible to describe a block containing a read-only register and a write-only register sharing the same physical address. If it is not possible to avoid this implementation structure, specify a single register with a field of `other` bits.

`[<regfile` *`name`*`[=`*`rename`*`][`*`[n]`*`] [(`*`hdl_path`*`)] [`*`@offset`*`]`
`[+incr] [read|write];>]`

```
[<regfile name[[n]] [(hdl_path)] [@offset] [+incr]
   {
      <property>
   }]
```

The first form specifies an instance of a previously-defined register file description. The second form defines a new register file description and instantiates it in the block. An inlined register file description cannot contain the shared property. Access to a shared register file can be further restricted to read or write in a particular instance, which would essentially apply this restriction to all shared registers contained inside that shared register file.

If a numerical index is specified, an array of register files is instantiated. Register file arrays are located at consecutive address offsets, starting with register [0], separated by the specified offset increment. The offset increment is required and only valid when instantiating a `regfile` array. Instantiating an array of register files is logically equivalent to explicitly instantiating all of the individual register files explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

Register files are usually used to specify arrays of register groups. Arrays of register files yield a different address map than register arrays. See "Arrays and Register Files" for more details.

The optional (`hdl_path`) is the hierarchical reference, within the block, to the HDL structure implementing the register file. Direct hierarchical access to the register file can be automatically generated by concatenating the specified (`hdl_path`) with the (`hdl_path`) of the enclosing system and the (`hdl_path`) specified in the registers. The (`hdl_path`) must be enclosed between parentheses. The (`hdl_path`) for a register file array must include a "%d" placeholder that will be replaced with the decimal index of the register file in the array.

```
[<memory name[=rename] [(hdl_path)] [@offset]
      [read|write];>]
```

```
[<memory name [(hdl_path)] [@offset] {
      <property>
   }>]
```

The first form specifies an instance of a previously-defined memory description. The second form defines a new memory description and instantiates it in the block. An inlined memory description cannot contain the `shared` property. The access to a shared memory can be further restricted to read or write in a particular instance.

A memory may be instantiated at an explicit address offset within the block. If not specified, the memory is instantiated at the next available address, starting with 0. The number of addresses occupied by a memory depends on the size and width of the memory, and the endian property of the block. If a memory is not mapped in the address space of the block, the offset may be specified as `@none` to indicate that the memory does not consume any address locations.

The optional `(hdl_path)` is the hierarchical reference, within the block, to the HDL structure implementing the memory. If an `(hdl_path)` is specified, direct hierarchical access to the memory can be automatically generated by concatenating it with the `(hdl_path)` of the enclosing system. The `(hdl_path)` must be enclosed between parentheses.

If more than one memory with the same name is instantiated in the same block, it must be renamed to a unique name within the block.

At least one register or memory property must be specified.

```
[<virtual register name [=rename[n] mem@offset
  [+incr]];>]
```

```
[<virtual register name[[n] mem@offset [+incr]] {
     <property>
  }]
```

The first form instantiates an array of a previously-defined virtual register description in the block. The second form instantiates an array of a new virtual register description.

If a memory association is specified, the array of virtual register is statically implemented in the specified memory starting at the specified offset. If an increment value is specified, the implementation offset of each virtual register in the virtual register array is incremented by the specified increment. If a memory association is not specified, the virtual register is still instantiated in the block but must be dynamically associated with an implementation memory using the "uvm_reg_vreg::implement()" or "uvm_reg_vreg::allocate()" method before it can be used.

If more than one array of virtual registers with the same name is associated in the same block, it must be renamed to a unique name within the memory.

Because virtual registers are implemented in memory, it is possible to describe overlapping virtual register arrays.

```
[<constraint name [{
     <expression>
  }]>]
```

Specifies constraints used when the content of the registers in the block is randomized. The constraints are not interpreted by the generation script and must be valid SystemVerilog or OpenVera expressions. Constraints at this level should specify cross-register constraints.

Constraints cannot be used to constrain the content of memories or virtual registers.

```
[cover <+|- a|b|f>
```

Specifies if the registers and memories in this block are to be included (+) in or excluded (-) from the address map (a), register bits (b) or field value (f) coverage model. If specified inside a "domain", applies to that domain only.

```
[doc {
     <text>
  }]
```

Specifies user documentation for the block or domain, using HTML formatting tags.

Example

Example A-11   Block With Single Physical Interface

```
source Example A-6
source Example A-10
block uart {
   bytes 1;
   endian little;
   register CTRL;
   memory tx_bfr @’h00100;
}
```

Example A-12   Block With Register Array

```
block multi_chan {
   bytes 1;
   endian little;
   register CHAN_CTRL[32] @’h0200 {
      bytes 2;
      ...
   };
}
```

Example A-13   Block With Two Physical Interfaces

```
register data_xfer {
   bytes 4;
   field data {
      bits 32;
   }
   shared;
}
register flags {
   field cts {
      access rw;
      reset 1;
   }
   field dtr {
      access rw;
   }
}
block bridge {
   domain pci {
     bytes 4;
     register flags=pci_flags;
     register data_xfer=to_ahb write;
```

```
      register data_xfer=frm_ahb read;
    }
    domain ahb {
       bytes 4;
       register flags=ahb_flags;
       register data_xfer=to_pci write;
       register data_xfer=frm_pci read;
    }
 }
```

```
[<block name[[.domain]=rename][[n]] [(hdl_path)]
            @offset [+incr];>]
```

```
[<block name[[n]] [(hdl_path)] @offset [+incr] {
      <property>
   }]
```

The first form specifies an instance of a previously-defined sub-block description. The second form defines a new sub-block description and instantiates it in the block.

A sub-block must be instantiated at an explicit address offset within the block. If the base address of the sub-block is programmable, specify the default (after reset) base address.

If a numerical index is specified, an array of sub-blocks is instantiated. Sub-block arrays are located at consecutive address offsets, starting with sub-block[0], separated by the specified offset increment. The offset increment is required and only valid when instantiating a sub-block array. Instantiating an array of sub-blocks is logically equivalent to explicitly instantiating all of the individual sub-blocks explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

The optional (`hdl_path`) is the hierarchical reference, within the block, to the HDL structure implementing the sub-block. Direct hierarchical access to the registers and memories in the sub-block can be automatically generated by concatenating the specified (`hdl_path`) with the (`hdl_path`) of any enclosing block and the (`hdl_path`) to the registers and memories within the sub-block. The (`hdl_path`) must be enclosed between parentheses. The (`hdl_path`) for a sub-block array must include a "%d" placeholder that will be replaced with the decimal index of the sub-block in the array.

If more than one sub-block with the same name is instantiated in the same block, it must be renamed to a unique name within the block. A reference to a domain within a sub-block uses a composite name and must be renamed to a single name that is a valid SystemVerilog or OpenVera user-defined identifier.

system

A `system` defines a design composed of blocks or subsystems. A system can be used to create larger systems.

Syntax

```
system name {
   <property>
}
```

Specifies a system with the specified name and a single physical interface.

```
system name {
   domain name {
      <property>
   }
   <domain name {
      <property>
   }>
   [doc { <text> }]
}
```

Specifies a system with the specified name and multiple physical interfaces. The name of each domain specifies the name of the corresponding physical interface. At least two domains must be specified. This form of the `system` specification can have a `doc` property outside of the `domain` specification.

The name of the system is used to generate system-specific unique identifiers.

Properties

The following properties can be used to specify the system and its domains.

```
[attributes {
      <name> <value>[, ...]
   }]
```

Specifies a value for the specified user-defined attribute. Multiple attributes may be specified by separating each attribute-value pair with a comma. If the value contains white spaces, it must be included between double quotes.

`bytes` *`n`*`;`

Specifies the number of bytes that can be accessed concurrently and uniquely addressed through the physical interface. This property is required.

```
[endian little|big|fifo_ls|fifo_ms;]
```

Specifies how wider blocks and subsystems are mapped onto multiple accesses over the physical interface. See “Hierarchical Descriptions and Composition” for a description of the various mapping modes. By default, little endian is used.

```
[<block name[[.domain]=rename][[n]] [(hdl_path)]
          @offset [+incr];>]
```

```
[<block name[[n]] [(hdl_path)] @offset [+incr] {
     <property>
  }]
```

The first form specifies an instance of a previously-defined block description. The second form defines a new block description and instantiates it in the system.

A block must be instantiated at an explicit address offset within the system. If the base address of the block is programmable, specify the default (after reset) base address.

If a numerical index is specified, an array of blocks is instantiated. Block arrays are located at consecutive address offsets, starting with block[0], separated by the specified offset increment. The offset increment is required and only valid when instantiating a block array. Instantiating an array of blocks is logically equivalent to explicitly instantiating all of the individual blocks explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

The optional `(hdl_path)` is the hierarchical reference, within the system, to the HDL structure implementing the block. Direct hierarchical access to the registers and memories in the block can be automatically generated by concatenating the specified `(hdl_path)` with the `(hdl_path)` of any enclosing system and the `(hdl_path)` to the registers and memories within the block. The `(hdl_path)` must be enclosed between parentheses. The `(hdl_path)` for a block array must include a "%d" placeholder that will be replaced with the decimal index of the block in the array.

If more than one block with the same name is instantiated in the same system, it must be renamed to a unique name within the system. A reference to a domain within a block uses a composite name and must be renamed to a single name that is a valid SystemVerilog or OpenVera user-defined identifier.

At least one `block` or `system` property must be specified.

```
[<system name[[.domain]=rename][[n]] [(hdl_path)]
            @offset [+incr];>]
```

```
[<system name[[n]] [(hdl_path)] @offset [+incr] {
      <property>
   }]
```

The first form specifies an instance of a previously-defined subsystem description. The second form defines a new subsystem description and instantiates it in the system.

A subsystem must be instantiated at an explicit address offset within the system. If the base address of the subsystem is programmable, specify the default (after reset) base address.

If a numerical index is specified, an array of subsystems is instantiated. Subsystem arrays are located at consecutive address offsets, starting with `subsys[0]`, separated by the specified offset increment. The offset increment is required and only valid when instantiating a subsystem array. Instantiating an array of subsystems is logically equivalent to explicitly instantiating all of the individual subsystems explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

The optional `(hdl_path)` is the hierarchical reference, within the system, to the HDL structure implementing the subsystem. Direct hierarchical access to the registers and memories in the subsystem can be automatically generated by concatenating the specified `(hdl_path)` with the `(hdl_path)` of any enclosing system and the `(hdl_path)` to the registers and memories within the subsystem. The `(hdl_path)` must be enclosed between parentheses. The `(hdl_path)` for a subsystem array must include a "%d" placeholder that will be replaced with the decimal index of the subsystem in the array.

If more than one subsystem with the same name is instantiated in the same system, it must be renamed to a unique name within the system. A reference to a domain within a subsystem uses a composite name and must be renamed to a single name that is a valid SystemVerilog or Openvera user-defined identifier.

At least one *`block`* or *`system`* property must be specified.

```
[<constraint name [{
     <expression>
  }]>]
```

Specifies constraints used when the content of the registers and memories in the system is randomized. The constraints are not interpreted by the generation script and must be valid SystemVerilog or OpenVera expressions. Constraints at this level should specify cross-register constraints.

```
[cover <+|- a|b|f>
```

Specifies if the registers and memories in this block are to be included (+) in or excluded (-) from the address map (a), register bits (b) or field value (f) coverage model. If specified inside a "domain", applies to that domain only.

```
[doc {
      <text>
   }]
```

Specifies user documentation for the system or domain, using HTML formatting tags. Currently unsupported.

Example

Example A-14   System With Single Physical Interface

```
source Example A-11
system SoC {
   bytes 1;
   endian little;
   block uart[2] @'hF0000 +'h01000;
}
```

Example A-15   System With Two Physical Interfaces

```
source Example A-11
source Example A-13
system SoC {
   domain ahb {
      bytes 4;
      block uart[2] @'hF0000 +'h01000;
      block bridge.ahb=br @0;
   }
   domain pci {
      bytes 4;
      block bridge.pci=br @0;
   }
}
```

B

Limitations in Code Generation for UVM Register Model

Fields

Volatility

The `uvm_reg_field::configure()` takes an argument bit `volatile`, this can be specified from the RALF as shown below.

```
block controller @100 {

   bytes 2;

   register status @1 {
      field value {
           bits 16;
           volatile 1;
```

```
                access wrc;
            }
        }
    }

    If unspecified a default of 0 is assumed.
```

has_reset

The `uvm_reg_field::configure()` method has another argument bit `has_reset`. The generated `UVM_REG MODEL` will have “1” assigned to this argument whenever hard reset value is set by the user in RALF description, else this argument will be assigned with value “0”.

individually_accessible

RALF Specification cannot specify if the field is individually accessible or not.

soft_reset

The current `ralgen` neglects this option provided by the RALF Specification. However, there is the `set_reset()` method in `UVM_REG` model that updates the value provided by the `soft_reset` field from RALF Specification.

set_compare()

The `set_compare()` method helps you to disable the checks done on certain fields, the RALF Specification lacks this feature.

Memories

Coverage

The `ralgen` command generates no covergroups for memories.

Registers

UVM_REG_FIFOs

This special register models a DUT FIFO accessed through write/ read, where writes push to the FIFO and reads pop from it. But the RALF specification lacks features supporting the usage of `uvm_reg_fifo`.

REGISTER CALLBACKS

`REG_MODEL` generated by the `ralgen` command lacks Callback Register Macro "`` `uvm_register_cb ``" and "`` `uvm_set_super_type ``" etc.