UVM Register Abstraction Layer Generator User Guide

G-2012.09-SP1
March 2013

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of
>
> _____ and its employees. This is copy number_____."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/Company/Pages/Trademarks.aspx.
All other product or company names may be trademarks of their respective owners.

Synopsys, Inc.
700 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Contents

A RALF Syntax

B Limitations in Code Generation for UVM Register Model

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:

`-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.

`-e`

Generate empty constraint blocks for every abstract class.

`-f <filename>`

Specifies all the ralgen options within a file.

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.

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.

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' definiton(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
```

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 1-11 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 A, "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 and memories. 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 {
         ...
      }
   }
}
```

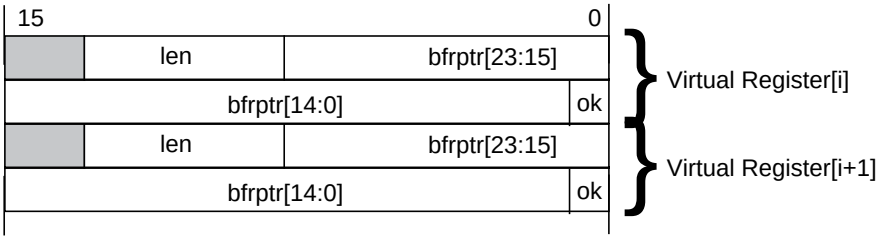
Virtual Fields and Virtual Registers

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
```

3

Generated Backdoors

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
```

[%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.

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 3-7 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 3-8 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 3-9 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 3-10, can be implemented using the RTL code shown in Example 3-11. The *reg* named `r1_reg` is used to implement fields `f1` and `f2`.

Example 3-10 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 3-11 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 3-12, can also be implemented using the RTL code shown in Example 3-11.

Example 3-12 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 3-13, can be implemented using the RTL code shown in Example 3-14. The *regs* named `f1_reg` and `f2_reg` are used to implement fields `f1` and `f2` respectively. Additionally, both Example 3-7 and Example 3-8 show an example of a register implemented using separate constructs for separate read-only fields.

Example 3-13 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 3-14 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 3-15, can be implemented using the RTL code shown in Example 3-16. The *regs* named `f2a_reg` and `f2b_reg` are used to implement field `f2`.

Example 3-15 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 3-16 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 3-17, can be implemented using the RTL code shown in Example 3-18. The *reg* named `m1_reg` is used to implement the entire memory.

Example 3-17 Memory Specification

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

Example 3-18 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.

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 3-19 will error out.

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

Example 3-19

```
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 3-20

```
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 3-19), except the fact that RALF reserved keywords checking will be disabled when the `hdl_path` syntax is used for specifying backdoor HDL path.

Support for Separate Compile

To compile cross module references (XMRs) with VCS separate compile, you need an XMR configuration file which will have a list of all the XMRs in a pre-defined format. In case of ralgen generated backdoor code, this configuration file should have a list of all XMRs implementing RAL backdoor signals.

You use the `-sep_cmp` and `-top_xmr_path <top_xmr_path>` `ralgen` command line options to generate the XMR configuration file.

User Interface

Use the `ralgen` command line option, `-sep_cmp`, to invoke the generation of the XMR configuration file when using the `-b` option for the generation of RAL backdoor access code. The `-sep_cmp` option will need the command line option, `-top_xmr_path <top_xmr_path>`.

When using the `-top_xmr_path` option, you need to specify the same top-level XMR path which defines the compile time macro `<top>_TOP_PATH` when compiling the generated RAL model classes. RAL top path/xmr information is required by `ralgen` because, RALF does not capture the information which is required by `ralgen` for building up the absolute top-level XMR path.

You can also use the `ralgen` command line option, `-sep_cmp` and `-top_xmr_path <top_xmr_path>` with other `ralgen` command line options without changing the functionality or meaning of the options.

The name of the generated XMR configuration file is, `ral_<top>.xmr` and will be located in your current working directory.

Example 3-21 Usage example

```
vgamddual100> cat test.ralf
block b {
   bytes 1
   register r1 (reg1) {
      field f
   }
   register r2 (regf.r2) {
      field f
   }
}
vgamddual100> ralgen -q -l sv -t b -b -sep_cmp -
top_xmr_path blk_top test.ralf
Synopsys UVM Register Abstraction Layer Code Generator
Copyright (c) 2006-2009 by Synopsys, Inc.
All Rights Reserved.
vgamddual100>
```

Generated XMR configuration file is as follows:

```
vgamddual100> cat ral_b.xmr
xmr {blk_top} {reg1};
xmr {blk_top.regf} {r2};
vgamddual100>
```

The format of the above shown separate compile XMR configuration file is the format required by VCS D-2009.12 and later versions.

For more details on this format and use-model of separate compile XMR configuration files, see VCS separate compile documentation of their respective releases.

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 ...
```

Even though the generated RAL model may 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.

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 specifing 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 specifing 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 4-1 will create three bins, named "AA", "BB" and "CC", each corresponding to field values 0, 1 and 15 respectively.

Example 4-1 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 4-2 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 4-2 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
     }
   }
```

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 labelled, it is possible to use that cross-coverage point in another cross-coverage point.

Example 4-3 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 4-4 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>
```

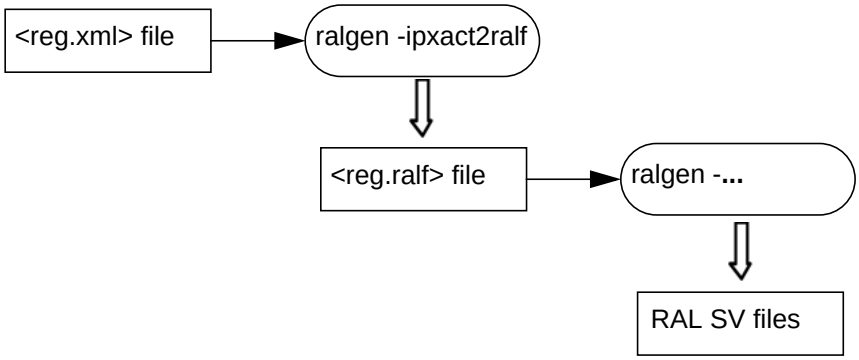
cpu_regs.ralf:

```
...
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>` 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.

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.

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

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*

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

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**>*number_of_array_elements*</spirit:**size**>	[size]
<spirit:**dim**>number_of_array_elements</spirit:**dim**>	[dim]
<spirit:**bitWidth**>*number_of_bits_in_each_element* </spirit:**bitWidth**>	bits / bytes

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

mask

Spirit IP-XACT Equivalent	RALF Feature
<spirit:**mask**>mask_value</spirit:**mask**>	defines which bit of the register has a known reset value

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.

```
<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, 'h2 };
}
```

It generates a constraint block as shown below:

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

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
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.

Access/Reset for Register

Access type should be specified only at the register level in IPXACT.

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;
    }
}
```

Vendor Extensions

Register Backdoors

A new vendor extension for the specification of backdoor paths is parsed and it is included while dumping the intermediate .ralf. The RALF is then generated with the backdoor structure. You are allowed to use the XMR based backdoor mechanism in which case `-b` is necessary on the ralgen command line. The default is the VPI based backdoors.

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>
     <SynopsysExtension:backdoor>srd_reg</
      SynopsysExtension:backdoor>
     <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>
         <SynopsysExtension:backdoor> rdata_lsb </
          SynopsysExtension:backdoor>
</spirit:field>
<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 (srd_reg) @'hb {
           bytes 1;
           field rdata_lsb (rdata_lsb) @'h0 {
               bits 4;
               access ro;
               hard_reset 'ha;
           }
           field rdata_msb @'h4 {
               bits 4;
               access ro;
               hard_reset 'h5;
           }
}
```

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
dim
values: value/name/description
suspaceMap
masterRef
nameGroup
coverage
endianess
regfile
virtual register
enum

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
hdl_path
reset_type

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

soft_reset
little
big
fifo_ls
fifo_ms
endian
endian_value

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.

```
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));
```

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

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

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

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 specificied 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.

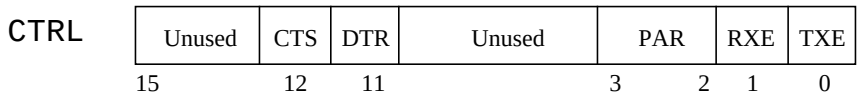
Example

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>
   }]
```

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, starting with register[0]. 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. 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.

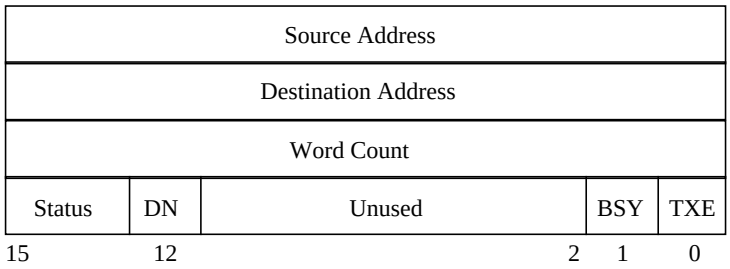
```
[doc {
      <text>
   }]
```

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 specificied 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 specificied 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] [+incr] [read|write];>]

[<register name[[n]] [(hdl_path)] [@offset] [+incr]
   {
      <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, starting with register [0]. 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. 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;
   }
}
```

---

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 specificied 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 block, it must be renamed to a unique name within the block. 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 same method takes another argument `bit has_reset`. The `UVM_REG MODEL` generated will have “1”assigned to this argument, because by default a reset value of 'h0 is assigned to the field. But the you might want this field to be unaffected when `uvm_reg_block::reset()` is called, the RALF specification cannot help in this case.

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`.

UVM_REG_INDIRECT_DATA

Used for indirectly accessing register arrays with the help of 2 mapped registers. This has to be done by specifying @none as the offset and has to be handled through user defined frontdoor.

REGISTER CALLBACKS

`REG_MODEL` generated by the `ralgen` command lacks Callback Register Macro "`` `uvm_register_cb `` and "`` `uvm_set_super_type `` etc.

Generated backdoors

`ralgen` doesn't generate the PLI based accesses for the backdoors presently and will be enhanced to handle that in future releases.