

# **Register Assistant UVM User Manual**

Release v2020.4

Unpublished work. © Siemens 2020

This document contains information that is confidential and proprietary to Mentor Graphics Corporation, Siemens Industry Software Inc., or their affiliates (collectively, "Siemens"). The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the confidential and proprietary information.

This document is for information and instruction purposes. Siemens reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Siemens to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Siemens products are set forth in written agreements between Siemens and its customers. **End User License Agreement** — You can print a copy of the End User License Agreement from: [mentor.com/eula](http://mentor.com/eula).

No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Siemens whatsoever.

SIEMENS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

SIEMENS SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, CONSEQUENTIAL OR PUNITIVE DAMAGES, LOST DATA OR PROFITS, EVEN IF SUCH DAMAGES WERE FORESEEABLE, ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF SIEMENS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

**LICENSE RIGHTS APPLICABLE TO THE U.S. GOVERNMENT:** This document explains the capabilities of commercial products that were developed exclusively at private expense. If the products are acquired directly or indirectly for use by the U.S. Government, then the parties agree that the products and this document are considered "Commercial Items" and "Commercial Computer Software" or "Computer Software Documentation," as defined in 48 C.F.R. §2.101 and 48 C.F.R. §252.227-7014(a)(1) and (a)(5), as applicable. Software and this document may only be used under the terms and conditions of the End User License Agreement referenced above as required by 48 C.F.R. §12.212 and 48 C.F.R. §227.7202. The U.S. Government will only have the rights set forth in the End User License Agreement, which supersedes any conflicting terms or conditions in any government order document, except for provisions which are contrary to applicable mandatory federal laws.

**TRADEMARKS:** The trademarks, logos and service marks ("Marks") used herein are the property of Siemens or other parties. No one is permitted to use these Marks without the prior written consent of Siemens or the owner of the Marks, as applicable. The use herein of third party Marks is not an attempt to indicate Siemens as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A list of Siemens' trademarks may be viewed at: [www.plm.automation.siemens.com/global/en/legal/trademarks.html](http://www.plm.automation.siemens.com/global/en/legal/trademarks.html) and [mentor.com/trademarks](http://mentor.com/trademarks).

The registered trademark Linux® is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

Support Center: [support.sw.siemens.com](http://support.sw.siemens.com)

Send Feedback on Documentation: [support.sw.siemens.com/doc\\_feedback\\_form](http://support.sw.siemens.com/doc_feedback_form)

# Table of Contents

---

<b>Chapter 1</b>	
<b>Register Assistant UVM</b>	<b>5</b>
Introduction	5
Register Assistant UVM Abstract Flow	6
Using Register Assistant UVM	7
Register Data Hierarchy	7
Blocks, Blocks Maps, and Registers/Memories	8
Blocks	8
Block Maps	10
Block Hierarchy	11
Relationship to UVM Generation	12
Additional Block and Block Map CSV Columns	12
Summary – Registers, Blocks, and Block Maps	13
 <b>Chapter 2</b>	
<b>Register Assistant UVM Inputs</b>	<b>17</b>
Importing Data From CSV Files	17
Setting Constraints	21
Auto-Instancing	22
 <b>Chapter 3</b>	
<b>Checking Imported Registers</b>	<b>27</b>
Introduction	27
Default Checks	27
 <b>Chapter 4</b>	
<b>Register Assistant UVM Output</b>	<b>29</b>
UVM Output	30
Generated UVM File	30
Supporting Coverage Models for Blocks	33
Supporting Coverage Models for Fields	37
Supporting Simple “Quirky” Registers	40
Supporting Back Door Access	41
Word Addressable Output	46
Applying Word Addressing in Register Assistant UVM	46
 <b>Chapter 5</b>	
<b>Customization</b>	<b>53</b>
Parameters	53
Using Parameters	53

**Chapter 6**  
**Register Assistant UVM in GUI Mode ..... 57**  
    Using Register Assistant UVM in GUI Mode ..... 57

**Appendix A**  
**Using Register Assistant in Batch/Command-Line Mode..... 61**  
    Running Register Assistant UVM ..... 61

**Appendix B**  
**Examples ..... 63**  
    UVM Output Example ..... 63  
    UVM Word Addressable Output Example ..... 73

**Third-Party Information**  
**End-User License Agreement**  
**with EDA Software Supplemental Terms**

# Chapter 1

## Register Assistant UVM

---

Register Assistant UVM is a register management tool that allows you to make changes to register specifications in a single place and automatically generate and update a UVM package.

<b>Introduction</b> .....	<b>5</b>
<b>Register Assistant UVM Abstract Flow</b> .....	<b>6</b>
<b>Using Register Assistant UVM</b> .....	<b>7</b>
<b>Register Data Hierarchy</b> .....	<b>7</b>
<b>Blocks, Blocks Maps, and Registers/Memories</b> .....	<b>8</b>
Blocks .....	8
Block Maps .....	10
Block Hierarchy .....	11
Relationship to UVM Generation .....	12
Additional Block and Block Map CSV Columns .....	12
Summary – Registers, Blocks, and Block Maps .....	13

## Introduction

A typical modern device contains a rich mix of hardware, firmware, and software. Communication between these domains is provided by software-addressable registers whose locations are specified along with memories in a block.

Specifying registers and managing changes is typically a manual, laborious, and error prone task. What is required is a single repository to describe registers and memories for each component, sub-system, and system from which the output can be generated.

This is not a new problem; project teams have been finding creative ways to address this challenge for many years and, therefore, use a variety of formats for describing register and memory information. Register Assistant UVM can import register and memory specifications from spreadsheets (CSV) into a cohesive, extensible data model describing a hierarchy of blocks, sub-blocks, maps, registers, fields, and memories.

DRC checks are available to ensure the consistency of data. The current release of Register Assistant UVM generates UVM register package SystemVerilog code for verification.

## Register Assistant UVM Abstract Flow

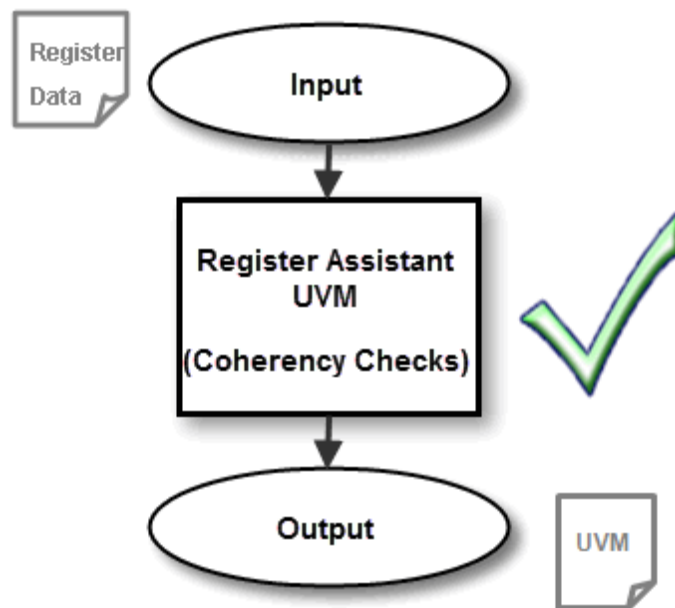
Register Assistant UVM operates and produces output based on a specific flow. The main stages that comprise the Register Assistant UVM flow are as follows:

- **Importing Register Data** — In this stage, you use Register Assistant UVM to import your register definitions. Register Assistant UVM imports from CSV files. Refer to [“Register Assistant UVM Inputs”](#) on page 17.
- **Checking the Imported Data** — Register Assistant UVM runs a number of coherency checks to verify the correctness of the imported data. For example, Register Assistant UVM ensures that at least one top register block is available.

Register Assistant UVM provides a number of built-in default checks. Refer to [“Checking Imported Registers”](#) on page 27 for details.

- **Generating Output** — The UVM generator in Register Assistant UVM uses the imported register data to create UVM register output for verification engineers. Refer to [“Register Assistant UVM Output”](#) on page 29.

The commands related to the above stages are run either through batch commands or through a wizard. Refer to [“Register Assistant UVM in GUI Mode”](#) on page 57.



---

### Tip

**i** You can refer to *Register Automation for UVM Workbook* for information on the usage flow of Register Assistant UVM. The workbook is available on the path `<install_directory>/registerassistantuvm/examples/labs/RUVM_Flow/Training_workbook.pdf`.

---

## Using Register Assistant UVM

Register Assistant UVM is either used in batch mode, or through a graphical interface wizard. Refer to “[Register Assistant UVM in GUI Mode](#)” on page 57 for more information.

## Register Data Hierarchy

To be able to extract data from the imported register definitions, Register Assistant UVM assumes that your register definitions are designed based on a certain hierarchy or model. The basic units of the hierarchy are registers, memories, register blocks, and block maps.

A register block may contain instances of registers, memories, or sub-blocks. Each register contains any number of fields, which mirror the values of the corresponding elements in hardware.

A register block can have more than one block map, but there should be at least one map.

It is important to note that Register Assistant UVM checks the structure of the imported register data to make sure that at least one top register block is available, and if not found, a failure occurs.

## Blocks, Blocks Maps, and Registers/Memories

This section explains block descriptions, defining a block map, specifying block hierarchy, how the UVM generator utilizes the block and block map information, and the relationship between registers, blocks, and block maps using a simple example.

<b>Blocks .....</b>	<b>8</b>
<b>Block Maps .....</b>	<b>10</b>
<b>Block Hierarchy .....</b>	<b>11</b>
<b>Relationship to UVM Generation .....</b>	<b>12</b>
<b>Additional Block and Block Map CSV Columns .....</b>	<b>12</b>
<b>Summary – Registers, Blocks, and Block Maps .....</b>	<b>13</b>

### Blocks

Blocks are optional. If you have a “flat” description for your registers/memories, you can use the Auto-Instance feature. This feature automatically creates a top-level block based on your register/memory descriptions provided that the address information is supplied for each register/memory definition.

For more information, refer to “[Auto-Instancing](#)” on page 22.

Alternatively you can explicitly instantiate one or more registers or memories in a named Block.

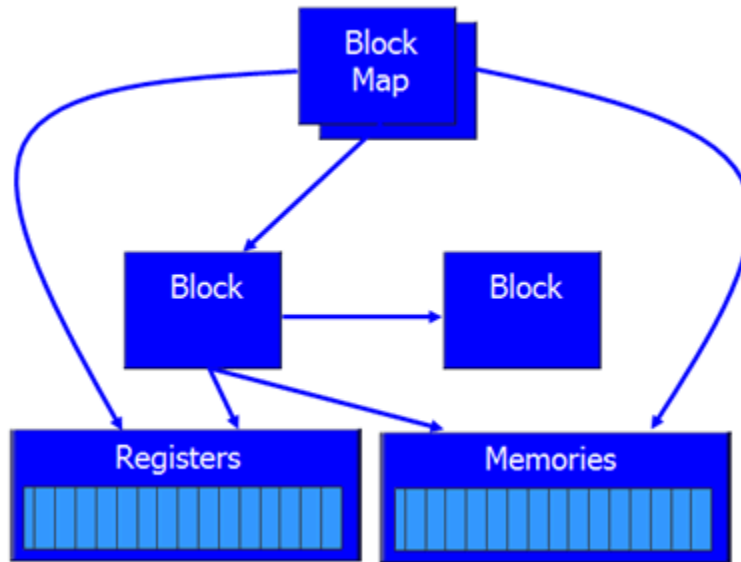
Blocks can be used:

- To instance a set of registers/memories that is used repeatedly throughout a design. For example, to represent each channel in a 32 channel DMA.
- To represent design structure (blocks can be instanced within other blocks in a hierarchical manner).
- To identify contiguous sections within the overall memory map.

Each block that you specify must also have one or more associated address map interfaces for the content of the block. This is known as a Block Map. A given block map specifies which registers/memories/sub-blocks are accessible together with their addresses as seen through the block map. Block maps allow multiple processors to access different combinations of the same registers/memories within a block but at different locations in the memory map.

It is common to define registers/memories in one CSV file and to specify blocks and block maps in separate CSV files. [Figure 1-1](#) shows the relationship between block maps, blocks, and register/memory descriptions.



**Figure 1-1. Block Maps, Blocks and Registers/Memories**

Each block description requires three columns to be defined in your CSV input:

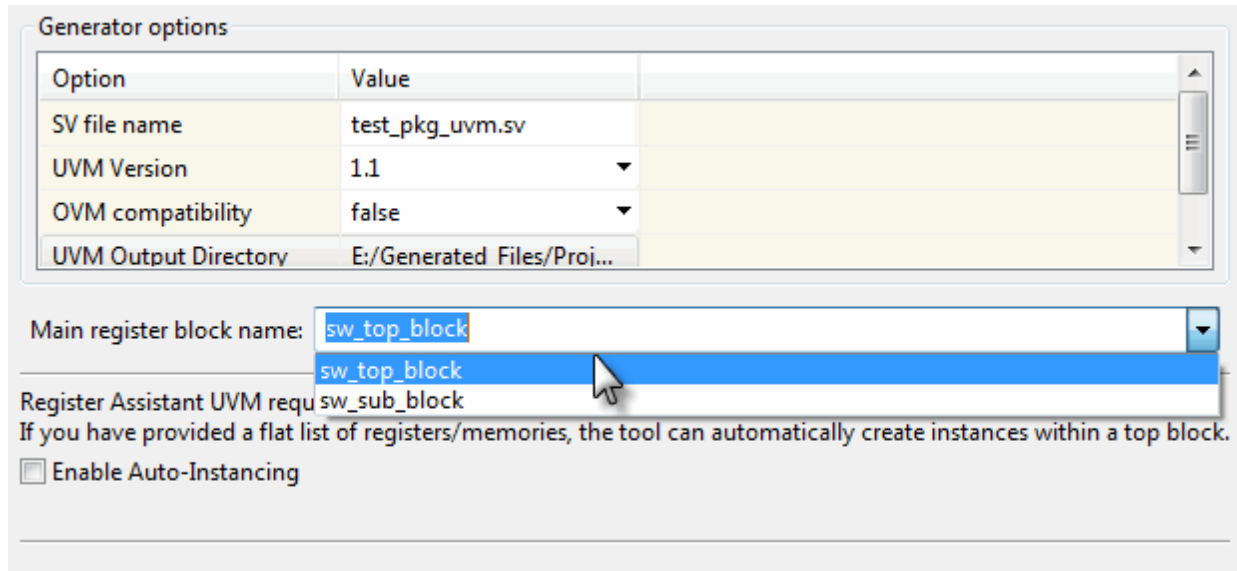
- **Block Name** — a name for the block. For each row of the block description, this name repeats.
- **Block Component Name** — the name of the register, memory, or sub-block that is instantiated in this block. This name must match the register/memory/sub-block name that you defined in your register/memory description columns. For example, if you define a register called `status_reg`, the entry in the Block Component Name column is `status_reg`.
- **Block Instance Name** — a unique identifier for the register/memory/sub-block in the context of this block. Similar to instantiating a component on a schematic, each register/memory/sub-block instance within a block must have a name. For example, if you want to add the `status_reg` to the block definition, you can name this register `status_reg_h`. You can instance the same register/memory/sub-block multiple times using different block instance names.

In this example, there are three registers and one memory grouped in a block named `reg_block`:

Block Name	Block Component Name	Block Instance Name	Block Instance Type
reg_block	status_reg	status_reg_h	register
reg_block	RegA	RegA_h	register
reg_block	RegB	RegB_h	register
reg_block	small_mem	small_mem_h	memory

You can define as many blocks as you require. However, when you define more than one block, Register Assistant UVM requires you to specify which block is considered the top block. When you run the tool interactively, you will see a list of blocks that the tool has determined from your

input files. Select the block that you want to designate as the top block from the dropdown list. In the following example, there are two blocks available and `sw_top_block` is the top block:



## Block Maps

To use a block description, you need to define a block map. The block map defines a starting address for each register/memory within the block.

Block maps require four columns:

- **Block Name** — the name of the block that you have defined. This name repeats for each register/memory/sub-block instance within the block that is accessible via this block map.
- **BlockMap Name** — a name for the block map that you are defining. This name repeats for each register/memory/sub-block entry in the block map.
- **BlockMap Instance Name** — the instance name that you defined for the register/memory/sub-block instance (using the Block Instance Name column).
- **BlockMap Instance Address** — the starting address for the register/memory/sub-block instance.

In this example, we define the address space for the `reg_block` block:

Block Name	BlockMap Name	BlockMap Instance Name	BlockMap Instance Address
reg_block	bus_map	status_reg_h	0x0
reg_block	bus_map	RegA_h	0x2
reg_block	bus_map	RegB_h	0x4
reg_block	bus_map	small_mem_h	0x10

## Block Hierarchy

You can specify levels of block hierarchy, or sub-blocks, within your block specification.

In this example block description there is a top-level block called `sw_top_block` that contains two instances of a block called `sw_sub_block`:

Block Name	Block Component Name	Block Instance Name	Block Instance Type
sw_top_block	sw_sub_block	sw1	block
sw_top_block	sw_sub_block	sw2	block
sw_top_block	stopwatch_counter	vreg1	register
sw_top_block	stopwatch_counter	vreg2	register
sw_top_block	stopwatch_counter	vreg3	register
sw_top_block	my_reg	my_reg1	register
sw_top_block	my_mem	mem1	memory
sw_sub_block	stopwatch_value	stopwatch_value_reg	register
sw_sub_block	stopwatch_reset_value	stopwatch_reset_value_reg	register
sw_sub_block	stopwatch_upper_limit	stopwatch_upper_limit_reg	register
sw_sub_block	stopwatch_lower_limit	stopwatch_lower_limit_reg	register
sw_sub_block	stopwatch_csr	stopwatch_csr_reg	register
sw_sub_block	stopwatch_memory	stopwatch_memory_reg	register

The example shows the use of the Block Instance Type of value block to indicate the two sub-blocks whose instance names are `sw1` and `sw2`. Those two instances reference a block called `sw_sub_block` which is defined to contain six registers (that are defined in a separate CSV file), starting on line 10.

For each block and sub-block, a corresponding block map is needed:

Block Name	BlockMap Name	BlockMap Description	BlockMap is default	BlockMap Instance Name	BlockMap Instance Address
sw_top_block	SW_MAP2	SW top block map	TRUE	sw1.SW_MAP	0x1000
sw_top_block	SW_MAP2			sw2.SW_MAP	0x2000
sw_top_block	SW_MAP2			vreg1	0x3000
sw_top_block	SW_MAP2			vreg2	0x3004
sw_top_block	SW_MAP2			vreg3	0x3008
sw_top_block	SW_MAP2			my_reg1	0x300C
sw_top_block	SW_MAP2			mem1	0x4000
sw_sub_block	SW_MAP	SW sub block map	TRUE	stopwatch_value_reg	0x04
sw_sub_block	SW_MAP			stopwatch_reset_value_reg	0x08
sw_sub_block	SW_MAP			stopwatch_upper_limit_reg	0x0C
sw_sub_block	SW_MAP			stopwatch_lower_limit_reg	0x10
sw_sub_block	SW_MAP			stopwatch_csr_reg	0x14
sw_sub_block	SW_MAP			stopwatch_memory_reg	0x34

In this example, we define the top-level block map in `SW_MAP2` and the instance addresses for each item in the block. Notice that in this case we explicitly reference the block map `SW_MAP` for both instances of the sub-block using the path “`sw1.SW_MAP`” for the BlockMap Instance Name. As each block can have any number of associated Block Maps, if you specify the instance name alone (for example, `sw1`) it will reference the default map for that sub-block. If

you wish to reference a specific map, you simply add the map name to the instance name (for example, “sw1.SW\_MAP”).

The full CSV files for the preceding example can be found in `<install_location>/examples/uvm/CSV`.

## Relationship to UVM Generation

The UVM generator utilizes the block and block map information that you provide.

The UVM generator creates a UVM block class for each Block, with instances for each register/memory instance. Declares the associated maps with the appropriate address information from the Block Map.

## Additional Block and Block Map CSV Columns

You can specify several optional CSV columns relating to blocks. For information on supported CSV columns, refer to `<installation_folder>\registerassistantuvm\docs\pdfdocs\RUVM_reference\CSV_columns.pdf`.

Commonly used optional columns include the following:

- **Block Description and Block Instance Description** — add comments and documentation about the blocks and instances within those blocks.
- **Block Coverage** — allows you to specify one of the built-in coverage models for a block. By adding a coverage model, such as UVM\_CVR\_ADDR\_MAP, Register Assistant UVM generates cover groups and cover points for the block.
- **Block Backdoor** — allows you to specify a relative backdoor path for the registers/memories within the block.
- **Block Instance Backdoor** — allows you to specify the starting instance name that the backdoor path references.
- **Block Instance Dimension** — enables you to specify how many instances of the register/memory or sub-block to add in the block.

---

### Note



When you explicitly set the Block Instance Dimension as “1”, Register Assistant UVM generates this instance as an array of single element. Yet, when a value is not specified for this column, Register Assistant UVM generates one regular instance by default (not an array).

---

- **Block Instance Type** — specifies the kind of instance in the block: register, memory, or another block (sub-block). The default value is register.

- **Block Replication Offset** — if sub-block instances are not contiguous and you want each instance of a sub-block to start on a specific boundary, you can specify a Replication Offset. This allows you to specify the overall width of the block instance (that is, the actual block size plus padding).
- **Parameters** — allows you to specify your own parameters to use in your own generator. You can specify a block level parameter and value or an instance-level parameter for each instance within a block.

You can specify several optional CSV columns relating to block maps. Commonly used optional columns include:

- **BlockMap Description** — allows you to add comments and documentation about the block map.
- **BlockMap is Default** — indicates whether a particular block map is the default address map for a block. If you only have a single block map, you do not need this column.
- **BlockMap Address Offset** — the address offset for the map (default is 0x0).
- **BlockMap Instance Access** — the software access mode for each register/memory instance in the block map.
- **Parameters** — allows you to specify your own parameter to use in your own generator.

## Summary – Registers, Blocks, and Block Maps

This section summarizes the relationship between registers, blocks, and block maps using a simple example.

We start by declaring three 32-bit, read-write registers at consecutive addresses:

Register Name	Register Address	Register Width	Register Access	Register Reset Value
reg1	0x0	32	RW	0x0
reg2	0x4	32	RW	0x0
reg3	0x8	32	RW	0x0

The address information is required if we use the auto-instance mechanism, but for this example we will explicitly instance these registers in a block so we add an instance of each register in a block called sub\_block:

Block Name	Block Component Name	Block Instance Name	Block Instance Type
sub_block	reg1	reg1_inst	register
sub_block	reg2	reg2_inst	register
sub_block	reg3	reg3_inst	register

We have now described the contents of a block called `sub_block` but we have not defined how a bus interface can access these register instances or which of these instances are visible to that specific interface. This address map information is specified using a Block Map.

In this simple case, we define a single map called `SUB_MAP` with corresponding address values for each instance in the block:

Block Name	BlockMap Name	BlockMap is default	BlockMap Instance Name	BlockMap Instance Address
<code>sub_block</code>	<code>SUB_MAP</code>	TRUE	<code>reg1_inst</code>	<code>0x20</code>
<code>sub_block</code>	<code>SUB_MAP</code>		<code>reg2_inst</code>	<code>0x24</code>
<code>sub_block</code>	<code>SUB_MAP</code>		<code>reg3_inst</code>	<code>0x28</code>

Since we have associated the map `SUB_MAP` with the block `sub_block`, the addresses specified in the map will be used rather than those specified in the register definitions. This means that address map would be as follows for Register Assistant UVM:

#### Address Map:

Physical Address	Hierarchical Name	Width (Bits)	Dimension
<code>0x20</code>	<a href="#"><code>sub_block.reg1_inst</code></a>	32	1
<code>0x24</code>	<a href="#"><code>sub_block.reg2_inst</code></a>	32	1
<code>0x28</code>	<a href="#"><code>sub_block.reg3_inst</code></a>	32	1

If we want to create hierarchy in our address map (perhaps to reuse the block multiple times) we simply instance it in another block:

Block Name	Block Component Name	Block Instance Name	Block Instance Type
<code>sub_block</code>	<code>reg1</code>	<code>reg1_inst</code>	register
<code>sub_block</code>	<code>reg2</code>	<code>reg2_inst</code>	register
<code>sub_block</code>	<code>reg3</code>	<code>reg3_inst</code>	register
<code>top_block</code>	<code>sub_block</code>	<code>sub_block_inst</code>	block

Here we have instanced block `sub_block` in a new parent block called `top_block` and specified the instance type as “block”. We also need to make a corresponding map for the new block:

Block Name	BlockMap Name	BlockMap is default	BlockMap Instance Name	BlockMap Instance Address
<code>sub_block</code>	<code>SUB_MAP</code>	TRUE	<code>reg1_inst</code>	<code>0x20</code>
<code>sub_block</code>	<code>SUB_MAP</code>		<code>reg2_inst</code>	<code>0x24</code>
<code>sub_block</code>	<code>SUB_MAP</code>		<code>reg3_inst</code>	<code>0x28</code>
<code>top_block</code>	<code>TOP_MAP</code>	TRUE	<code>sub_block_inst.SUB_MAP</code>	<code>0x100</code>

In the instance name we have explicitly stated that the instance of sub\_block in the block top\_block is to use the SUB\_MAP map and have specified the instance of the block starts at address “100 hex”. Therefore, from the perspective of the TOP\_MAP the following overall address map would be as follows for Register Assistant UVM:

**Address Map:**

Physical Address	Hierarchical Name	Width (Bits)	Dimension
0x100	<a href="#">.top_block.sub_block_inst</a>	-	1
0x120	<a href="#">.top_block.sub_block_inst.reg1_inst</a>	32	1
0x124	<a href="#">.top_block.sub_block_inst.reg2_inst</a>	32	1
0x128	<a href="#">.top_block.sub_block_inst.reg3_inst</a>	32	1

The offset addresses of the instances within SUB\_MAP have been added to the address of the parent map TOP\_MAP to show the absolute addresses for the overall design.





# Chapter 2

## Register Assistant UVM Inputs

---

Register Assistant UVM receives certain inputs (register definitions and optional block definitions in CSV format), runs a number of checks to verify that the input is correct, and then produces the required output (UVM files).

<b>Importing Data From CSV Files</b> .....	<b>17</b>
<b>Setting Constraints</b> .....	<b>21</b>
<b>Auto-Instancing</b> .....	<b>22</b>

## Importing Data From CSV Files

To import CSV files, first you have to prepare the CSV files, which contain register data and optional block data.

To define your register data in CSV files, you have to use the columns described in [Table 2-1](#). Make sure you use the same column names below.

**Table 2-1. CSV Columns**

Column Name <sup>1</sup>	Description
<b>Register Columns:</b>	
Register Name*	Name of the register.
Register Description	Description for the register.
Register Address*	Address offset for the register.
Register Width*	Width in bits for the register.
Register Access*	Software Access for the register.
Register Reset Value*	Reset value for the register.
Register Constraints	Constraints for the register.
Register Custom Type	Custom register class from which this register can be extended. For example, “my_reg_type”.
<b>Field Columns:</b>	
Field Name*	Name of the field.
Field Description	Description of the field.
Field Offset*	Address offset for the field,
Field Width*	Width in bits for the field.

**Table 2-1. CSV Columns (cont.)**

<b>Column Name<sup>1</sup></b>	<b>Description</b>
Field Access*	Software Access for the field.
Field Reset Value*	Reset value for the field.
Field is Covered	Indicates whether to generate functional coverage.
Field is Reserved	Indicates if the field is reserved.
Field is Volatile	Indicates if the field is volatile.
Field Constraints	Constraints for the field.
Field Backdoor	Simulator field path name for a given field in the register.
<b>Memory Columns:</b>	
Memory Name*	Name of the memory.
Memory Description	Description of the memory.
Memory Address*	Address offset for the memory.
Memory Range*	Size of the memory.
Memory Width*	Width of the memory,
Memory Access*	Software Access for the memory.
Memory Constraints	Constraints for the memory.
<b>Block Columns:</b>	
Block Name*	Name of the block. Required to create blocks and add children to it.
Block Description	Description of the block.
Block Component Name*	The definition name of the child to be added to the block (register, memory, sub-block, and so on).
Block Coverage	Indicates the coverage type of the block. The default is (UVM_NO_COVERAGE).
Block Backdoor	HDL backdoor path to access the block when used as top block.
Block Instance Name*	The instance name of the child.
Block Instance Type	Type of the instance to add (block, register, memory). The default is "register".
Block Instance Description	Description of the instance.
Block Instance Dimension	Array dimension of the instance. The default is 1.
Block Instance Backdoor	Simulator instance path name for a given instance in the block.

**Table 2-1. CSV Columns (cont.)**

Column Name <sup>1</sup>	Description
Block Instance No Reg Tests	Disable all register auto-tests.
Block Instance No Reg Access Test	Disable register access auto-test.
Block Instance No Reg Shared Access Test	Disable register shared access auto-test.
Block Instance No Reg Bit Bash Test	Disable register bit bash auto-test.
Block Instance No Reg HW Reset Test	Disable register hardware reset auto-test.
Block Instance No Mem Tests	Disable all memory auto-tests.
Block Instance No Mem Access Test	Disable memory access auto-test.
Block Instance No Mem Shared Access Test	Disable memory shared access auto-test.
Block Instance No Mem Walk Test	Disable memory walk auto-test.
Block Replication Offset	Specifies the overall width for instances of the block if not the exact size of the block (that is, the size of block plus padding).
<b>Block Map Columns:</b>	
BlockMap Name*	Name of the address map within a block.
BlockMap Description	Description of block address map.
BlockMap is default	Indicates whether this is the default address map for the block.
BlockMap Instance Name*	Name of the instance of a component in the block.
BlockMap Instance Address*	Offset address for the given instance.
BlockMap Address Offset	Address offset for the map itself. The default is 0x0.
BlockMap Instance Access	Optional software access for the instance in this map. For example: RW
BlockMap Address Mode	Indicates whether byte addressing or word addressing should be used by Register Assistant UVM.
BlockMap Word Bytes	Number of bytes in a Word. This is used when the generation of word-addressable output is enabled.

**Table 2-1. CSV Columns (cont.)**

Column Name <sup>1</sup>	Description
BlockMap Endian	Indicates the endianness setting.
<b>Project Columns:</b>	
Project Extra Imports	Verilog package/s to import into the generated UVM package. For example: mypkg::*,mypk2::*

1. The asterisk “\*” signifies that the column is required.

For information on the supported CSV columns, refer to *<installation\_folder>\registerassistant\docs\pdfdocs\RUVM\_reference\CSV\_columns.pdf*. For information on the supported software access modes, refer to *<installation\_folder>\registerassistant\docs\pdfdocs\RUVM\_reference\SW\_Access\_Modes.pdf*.

Note the following regarding the required register columns:

- You only need to specify a Register Address if you are using [Auto-Instancing](#) (that is, you are not specifying a BlockMap Instance Address).
- You only need to specify the Register Access and Register Reset Value if you do not have the Field Access and Field Reset Value specified for every field in the register.
- You only need to specify the Field Access if it is different from the Register Access (or if you did not specify the Register Access). Also, you only need to specify the Field Reset Value if you did not specify a Register Reset Value for the register as a whole.

General Notes:

- Column names are case insensitive and also white space insensitive. For example, “register name”, “RegisterName”, “REGISTER NAME” will all match the column “Register Name”.
- When using the Block Instance Backdoor column in the CSV input files, if you want to set a backdoor path for an array with a different path for each instance in the array, you can use the %(DIM) variable.

If used, the following code is generated (in this example the backdoor path is set as “data\_small\_mem\_h\_array\_%(DIM)\_local”):

```
foreach ( small_mem_h[i] ) begin
small_mem_h[i] =
small_mem::type_id::create($psprintf("small_mem_h[%0d]", i));
small_mem_h[i].configure(this, null,
($psprintf("data_small_mem_h_array_%0d_local", i)));
small_mem_h[i].build();
end
```

If not used, the following code is generated (in this example the backdoor path is set as “data\_small\_mem\_h\_array\_local”):

```
foreach ( small_mem_h[i] ) begin
small_mem_h[i] =
small_mem::type_id::create($psprintf("small_mem_h[%0d]", i));
small_mem_h[i].configure(this, null,
"data_small_mem_h_array_local");
small_mem_h[i].build();
end
```

For information on the %(DIM) variable, refer to the UVM Variables table available on [<installation\\_folder>\registerassistant\docs\pdfdocs\RUVVM\\_reference\UVM\\_Variables.pdf](#).

- Specifying the software access mode for the register is important to determine the behavior of the register. Note the following:
  - If the access mode of the register is not detected, Register Assistant UVM will use the default access mode “read-write”.
  - You can also specify the access mode for the fields within the registers. Note that if the field’s access mode is not specified, Register Assistant UVM will use the value of the register’s access mode.
  - If you do not explicitly define fields within the register in the input files, Register Assistant UVM creates a single field by default in the generated output covering the entire register. This default field takes the same access mode of the register.
- If there is a row in a CSV file that does not have a Register Name or that is completely empty, then the whole row will be ignored by Register Assistant UVM. If only the Register Name is available in the row, then it will be added to the project.
- Before generation, make sure that any instance of any object in the input files has a definition. Otherwise, an error is raised during generation if the definition is missing.
- When defining register input, be informed that Register Assistant UVM defaults the addressable width to 8 bits. So for a 32 bit register width, the addresses increment by 4 Hex. e.g. 00, 04, 08, 0C, and so on.
- The Field Is Reserved property enables you to define reserved fields within the register. Reserved fields accept non-unique field names.

## Setting Constraints

You can set constraints on your imported register definitions. For example, you may need to make sure that value “X” related to object “Y” is not greater than “10”. These constraints are SystemVerilog constraints.

Any added constraints are inserted in the generated UVM output. Later when running simulation, the simulator takes the constraints in the generated UVM file into consideration to ensure they are fulfilled.

When importing from CSV files, you will be able to apply constraints to fields, registers, and memories. Constraints cannot be applied to blocks or block maps.

## Procedure

1. Add a column in the CSV file for the required object (registers, fields or memories) using the following default column names respectively: Register Constraints, Field Constraints, or Memory Constraints.
2. Write the code of the constraints in the column using SystemVerilog language. This code is added as it is in the generated UVM output later.

For example:

```
constraint my_constraint
{x < 5;}
```

Constraints can be spread over multiple lines as in the following example:

```
Constraint my_constraint {
    x > 5 ;
    y > 3;
}
```

# Auto-Instancing

Auto-Instancing automatically instances registers/memories in a new or existing top-level block and creates a corresponding block map. This provides a rapid way to create the required UVM block structure from a “flat” list of registers and memories. Auto-instancing requires register addresses to be specified with the register/memory definitions.

## Procedure

1. To automatically create instances, this can be done using one of the following methods:
2. Through setting the auto-instancing option in the Register Assistant UVM wizard if you are running Register Assistant UVM in the graphical user interface mode.

See “[Register Assistant UVM in GUI Mode](#)” on page 57.

3. Through using the following command line option when running Register Assistant UVM in batch mode:

```
-autoinstance
```

See “[Register Assistant UVM in GUI Mode](#)” on page 57.

4. The register/memory instances are created in the top block which is specified either through the Register Assistant UVM wizard if you are using GUI mode (refer to [“Register Assistant UVM in GUI Mode”](#) on page 57) or through -block command line option if you are using batch mode (refer to [“Using Register Assistant in Batch/Command-Line Mode”](#) on page 61).
5. If the block name specified does not exist in the imported definitions, Register Assistant UVM will create a block with the same name in the generated output and make it the top block, and then create instances of the register definitions within that block.

## Examples

The following example is an excerpt of UVM output generated for CSV input that only contained the definitions of registers. A new top block my\_top\_Block is automatically created (this block was not found in the input files) and it contains instances of the imported registers.

```
/* BLOCKS */
//-----
// Class: my_top_Block
//-----

class my_top_Block extends uvm_reg_block;
`uvm_object_utils(my_top_Block)

rand stopwatch_value inst_stopwatch_value; // Current value
rand stopwatch_reset_value inst_stopwatch_reset_value; // Reset value
rand stopwatch_upper_limit inst_stopwatch_upper_limit; // Upper limit
rand stopwatch_lower_limit inst_stopwatch_lower_limit; // Lower limit
rand stopwatch_memory inst_stopwatch_memory; // Memory register
rand stopwatch_csr inst_stopwatch_csr; // Control Status Register
rand stopwatch_counter inst_stopwatch_counter; // Stop Watch Counter

uvm_reg_map my_top_Block_map;

// Function: new
//
function new(string name = "my_top_Block");
super.new(name, UVM_NO_COVERAGE);
endfunction

// Function: build
//
virtual function void build();
inst_stopwatch_value =
stopwatch_value::type_id::create("inst_stopwatch_value");
inst_stopwatch_value.configure(this);
inst_stopwatch_value.build();

inst_stopwatch_reset_value =
stopwatch_reset_value::type_id::create("inst_stopwatch_reset_value");
inst_stopwatch_reset_value.configure(this);
inst_stopwatch_reset_value.build();

inst_stopwatch_upper_limit =
stopwatch_upper_limit::type_id::create("inst_stopwatch_upper_limit");
inst_stopwatch_upper_limit.configure(this);
inst_stopwatch_upper_limit.build();

inst_stopwatch_lower_limit =
stopwatch_lower_limit::type_id::create("inst_stopwatch_lower_limit");
inst_stopwatch_lower_limit.configure(this);
inst_stopwatch_lower_limit.build();

inst_stopwatch_memory =
stopwatch_memory::type_id::create("inst_stopwatch_memory");
inst_stopwatch_memory.configure(this);
inst_stopwatch_memory.build();

inst_stopwatch_csr =
stopwatch_csr::type_id::create("inst_stopwatch_csr");
inst_stopwatch_csr.configure(this);
inst_stopwatch_csr.build();
```



```
inst_stopwatch_counter =  
stopwatch_counter::type_id::create("inst_stopwatch_counter");  
inst_stopwatch_counter.configure(this);  
inst_stopwatch_counter.build();  
  
my_top_Block_map = create_map("my_top_Block_map", 'h0, 4,  
UVM_LITTLE_ENDIAN, 1);  
default_map = my_top_Block_map;  
  
my_top_Block_map.add_reg(inst_stopwatch_value, 'h0, "RW");  
my_top_Block_map.add_reg(inst_stopwatch_reset_value, 'h4, "RW");  
my_top_Block_map.add_reg(inst_stopwatch_upper_limit, 'h8, "RW");  
my_top_Block_map.add_reg(inst_stopwatch_lower_limit, 'hc, "RW");  
my_top_Block_map.add_reg(inst_stopwatch_memory, 'h10, "RW");  
my_top_Block_map.add_reg(inst_stopwatch_csr, 'h30, "RW");  
  
lock_model();  
endfunction  
endclass
```



# Chapter 3

## Checking Imported Registers

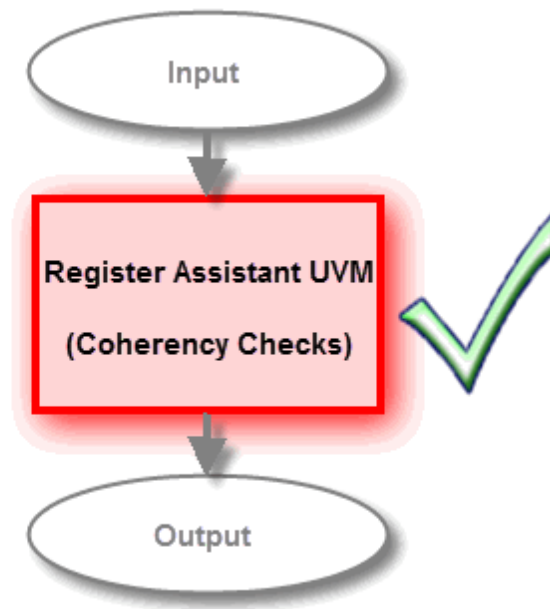
---

This chapter explains how Register Assistant can apply coherency checks to imported register definitions.

<b>Introduction</b> .....	<b>27</b>
<b>Default Checks</b> .....	<b>27</b>

### Introduction

After importing the register definition files, Register Assistant UVM automatically performs a number of built-in coherency checks on registers to identify any incorrect data. These checks are used to validate different aspects, such as ensuring that mandatory information on each register is provided.



### Default Checks

Register Assistant UVM includes a number of built-in checks that run by default on running the tool.

The following are examples of checks that Register Assistant UVM supplies:

- **Important Attributes** — Checks that mandatory data is provided correctly for the registers such as the Register Name, Register Address Offset and Field Name. This also applies to the Memory Name, Address Offset and Range.
- **Address Overlap** — Checks that no registers in the top level map overlap in addresses. This also applies to memories.
- **Reset Values** — Checks that all registers have valid reset values defined, whether on the registers or fields, and that the reset values completely fill the registers. This also applies to memories.

---

**Note**



You do not have to specify register reset values if all the fields have reset values and vice-versa.

---

- **Top Block Availability** — Checks that at least one top register block is available, and if not found, a failure occurs.
- **Unique Names** — All definition names should be unique: registers, memories, sub-blocks, and blocks. Within a block or a sub-block, all instance names should be unique. Within a Register, all field names should be unique. Within a register block, all map names should be unique.
- **Object Names** — Checks that instances have valid definitions and that object names are valid strings and valid identifiers.
- **Field Value Width** — Verifies that field values can fit within their defining fields.
- **Register Field Access Mismatch** — Checks that the register access and its field access are compatible. This check fails in the following conditions:
  - Register RO, field RW or WO
  - Register WO, field RW or RO

---

**Note**



The checks operate on instances, so if there is a violation in the definition and it is instantiated more than once, the same error can be reported many times. If a definition is not instantiated, no violations will be reported.

---

The generator runs the checks, but if a violation occurs, the flow does not stop (except with the Top Block Availability check). Any errors are displayed in the log.

# Chapter 4

## Register Assistant UVM Output

---

Register Assistant UVM imports the register data provided in CSV files and accordingly generates UVM files.

<b>UVM Output</b> .....	<b>30</b>
Generated UVM File .....	30
Supporting Coverage Models for Blocks .....	33
Supporting Coverage Models for Fields .....	37
Supporting Simple “Quirky” Registers .....	40
Supporting Back Door Access .....	41
<b>Word Addressable Output</b> .....	<b>46</b>
Applying Word Addressing in Register Assistant UVM .....	46

## UVM Output

---

The UVM verification environment is composed of several components, among which is the UVM register package. This package can be automatically generated using Register Assistant UVM. The generated UVM file should contain the description of the imported registers.

To generate the UVM register package, refer to “[Register Assistant UVM in GUI Mode](#)” on page 57 for detailed information on the procedure.

<b>Generated UVM File .....</b>	<b>30</b>
<b>Supporting Coverage Models for Blocks .....</b>	<b>33</b>
<b>Supporting Coverage Models for Fields .....</b>	<b>37</b>
<b>Supporting Simple “Quirky” Registers .....</b>	<b>40</b>
<b>Supporting Back Door Access .....</b>	<b>41</b>

## Generated UVM File

Register Assistant UVM generates a SystemVerilog file containing the description of your registers, memories, blocks as extracted from the imported source.

## Format

The UVM file is basically composed of several sections as shown in the following figure:

Figure 4-1. UVM Output

```

package sw_pkg_uvm;
import uvm_pkg::*;
`include "uvm_macros.svh"

class stopwatch_lower_limit extends uvm_reg;
  `uvm_object_utils(stopwatch_lower_limit)
  rand uvm_reg_field F;
  // Function: new
  function new(string name = "stopwatch_lower_limit");
    super.new(name, 32, UVM_NO_COVERAGE);
  endfunction
  // Function: build
  virtual function void build();
    F = uvm_reg_field::type_id::create("F");
    F.configure(this, 32, 0, "RW", 1, 32'h00000000, 1, 1, 1);
  endfunction
endclass

class sw_sub_block extends uvm_reg_block;
  `uvm_object_utils(sw_sub_block)

  rand stopwatch_value stopwatch_value_reg; // value instance
  rand stopwatch_reset_value stopwatch_reset_value_reg; // Reset value
  ...
  rand stopwatch_memory stopwatch_memory_reg[8]; // MEM instances

  uvm_reg_map SW_MAP; // SW sub block map
  ...
endclass

class sw_top_block extends uvm_reg_block;
  `uvm_object_utils(sw_top_block)

  sw_sub_block sw1; // Block1 instance 1
  sw_sub_block sw2; // Block1 instance 2
  rand stopwatch_counter vreg1; // Counter instance 1
  rand stopwatch_counter vreg2; // Counter instance 2
  ...
  my_mem mem1; // Memory instance
  uvm_reg_map SW_MAP2; // SW top block map
  ...
  // Function: build
  virtual function void build();
    ...
    sw1 = sw_sub_block::type_id::create("sw1");
    sw1.configure(this);
    sw1.build();
    ...
    vreg1 = stopwatch_counter::type_id::create("vreg1");
    vreg1.configure(this, null, "f_vreg1");
    vreg1.build();
    ...
    SW_MAP2 = create_map("SW_MAP2", 'h0, 4, UVM_LITTLE_ENDIAN);
    default_map = SW_MAP2;

    SW_MAP2.add_submap(sw1.SW_MAP, 32'h00001000);
    ...
    SW_MAP2.add_reg(vreg1, 32'h00003000, "RW");
    ...
    uvm_resource_db #(bit)::set({"REG::", this.vreg1.get_full_name()})
    lock_model();
  endfunction
  ...
endclass
endpackage
  
```

Defining the Register Package

Defining Register Objects

Defining Register Blocks and Register Hierarchy



- **Defining the Register Package** — Contains the definition of the register package and the required imports.
- **Defining Register Objects** — Contains the definitions of registers.

Registers are defined by extending the class `uvm_reg` or the alternative base class specified using the Register Custom Type CSV column.

```
class stopwatch_csr extends uvm_reg;
```

Subsequently, fields are defined within the register, constructors are created determining the name and size of the register and whether it has coverage or not, a build method is implemented to actually create the instances in the class, and then each field is configured using a configure method.

This is repeated for each register.

- **Defining Register Blocks and Register Hierarchy** — Contains the definitions of register blocks. It also contains the register hierarchy and the block maps.

This section defines sub-blocks by extending from the `uvm_reg_block` class. In this section, an instance is created for each register or memory in the block.

## Example

Refer to “[UVM Output Example](#)” on page 63 to view an example UVM register package generated by Register Assistant UVM.

## Supporting Coverage Models for Blocks

Register Assistant UVM enables you to generate coverage models within the UVM register package. You can specifically generate coverage models for instances in a block.

Register Assistant also supports UVM field coverage; refer to “[Supporting Coverage Models for Fields](#)” on page 37 for information.

To generate coverage models for blocks, you have to set the coverage type in your input files. This is achieved by doing the following:

1. In the CSV files which contain your register definitions, make sure you have a column in your blocks file titled “Block Coverage”.
2. Use the value of this column to specify the coverage type identifier. The only supported identifier is “UVM\_CVR\_ADDR\_MAP”. This identifier checks the addresses read or written in a block map. If you use any other value, then this is interpreted as no coverage required, that is, it will be interpreted as “UVM\_NO\_COVERAGE”.

By setting the coverage option in your input files, the generated UVM register package will include a new defined class called `<block name_coverage>`. This class will be instantiated

within the class of the block itself, and the instance will be called within the “sample” and “build” methods. See the following generated UVM register package example.

## Example

The following generated code excerpt shows an example of a block and its corresponding coverage class.

```
//-----
// Class: sw_sub_block_SW_MAP_coverage
//
// Coverage for the 'SW_MAP' in 'sw_sub_block'
//-----

class sw_sub_block_SW_MAP_coverage extends uvm_object;
    `uvm_object_utils(sw_sub_block_SW_MAP_coverage)

    covergroup ra_cov(string name) with function sample(uvm_reg_addr_t
addr, bit is_read);

        option.per_instance = 1;
        option.name = name;

        ADDR: coverpoint addr {
            bins stopwatch_value_reg = {'h4};
            bins stopwatch_reset_value_reg = {'h8};
            bins stopwatch_upper_limit_reg = {'hc};
            bins stopwatch_lower_limit_reg = {'h10};
            bins stopwatch_csr_reg = {'h14};
            bins stopwatch_memory_reg[8] = {'h34,
                                           'h38,
                                           'h3c,
                                           'h40,
                                           'h44,
                                           'h48,
                                           'h4c,
                                           'h50};
        }

        RW: coverpoint is_read {
            bins RD = {1};
            bins WR = {0};
        }

        ACCESS: cross ADDR, RW;

    endgroup: ra_cov

    function new(string name = "sw_sub_block_SW_MAP_coverage");
        ra_cov = new(name);
    endfunction: new

    function void sample(uvm_reg_addr_t offset, bit is_read);
        ra_cov.sample(offset, is_read);
    endfunction: sample

endclass: sw_sub_block_SW_MAP_coverage

//-----
// Class: sw_sub_block
//
// Sub_block for the stopwatch design
//-----

class sw_sub_block extends uvm_reg_block;
```

```
`uvm_object_utils(sw_sub_block)

    rand stopwatch_value stopwatch_value_reg; // Value instance
    rand stopwatch_reset_value stopwatch_reset_value_reg; // Reset Value
instance
    rand stopwatch_upper_limit stopwatch_upper_limit_reg; // Upper Limit
instance
    rand stopwatch_lower_limit stopwatch_lower_limit_reg; // Lower Limit
instance
    rand stopwatch_csr stopwatch_csr_reg; // CSR instance
    rand stopwatch_memory stopwatch_memory_reg[8]; // MEM instances

    uvm_reg_map SW_MAP; // SW sub block map
    sw_sub_block_SW_MAP_coverage SW_MAP_cg;

// Function: new
//
function new(string name = "sw_sub_block");
    super.new(name, build_coverage(UVM_CVR_ALL));
endfunction

// Function: build
//
virtual function void build();

    if(has_coverage(UVM_CVR_ADDR_MAP)) begin
        SW_MAP_cg =
sw_sub_block_SW_MAP_coverage::type_id::create("SW_MAP_cg");
        void'(set_coverage(UVM_CVR_ADDR_MAP));
    end
    stopwatch_value_reg =
stopwatch_value::type_id::create("stopwatch_value_reg");
    stopwatch_value_reg.configure(this);
    stopwatch_value_reg.build();

    stopwatch_reset_value_reg =
stopwatch_reset_value::type_id::create("stopwatch_reset_value_reg");
    stopwatch_reset_value_reg.configure(this);
    stopwatch_reset_value_reg.build();

    stopwatch_upper_limit_reg =
stopwatch_upper_limit::type_id::create("stopwatch_upper_limit_reg");
    stopwatch_upper_limit_reg.configure(this);
    stopwatch_upper_limit_reg.build();

    stopwatch_lower_limit_reg =
stopwatch_lower_limit::type_id::create("stopwatch_lower_limit_reg");
    stopwatch_lower_limit_reg.configure(this);
    stopwatch_lower_limit_reg.build();

    stopwatch_csr_reg =
stopwatch_csr::type_id::create("stopwatch_csr_reg");
    stopwatch_csr_reg.configure(this);
    stopwatch_csr_reg.build();

    foreach ( stopwatch_memory_reg[i] ) begin
```

```

        stopwatch_memory_reg[i] =
stopwatch_memory::type_id::create($psprintf("stopwatch_memory_reg[%0d]",
i));
        stopwatch_memory_reg[i].configure(this, null,
($psprintf("f_stopwatch_memory_reg_%0d", i)));
        stopwatch_memory_reg[i].build();
    end

    SW_MAP = create_map("SW_MAP", 'h0, 4, UVM_LITTLE_ENDIAN, 1);
    default_map = SW_MAP;

    SW_MAP.add_reg(stopwatch_value_reg, 'h4, "RW");
    SW_MAP.add_reg(stopwatch_reset_value_reg, 'h8, "RW");
    SW_MAP.add_reg(stopwatch_upper_limit_reg, 'hc, "RW");
    SW_MAP.add_reg(stopwatch_lower_limit_reg, 'h10, "RW");
    SW_MAP.add_reg(stopwatch_csr_reg, 'h14, "RW");
    foreach(stopwatch_memory_reg[i]) begin
        SW_MAP.add_reg(stopwatch_memory_reg[i], (i * ('h4)) + ('h34),
"RW");
    end

    lock_model();
endfunction

// Function: sample
//
function void sample(uvm_reg_addr_t offset, bit is_read, uvm_reg_map
map);
    if(get_coverage(UVM_CVR_ADDR_MAP)) begin
        if(map.get_name() == "SW_MAP") begin
            SW_MAP_cg.sample(offset, is_read);
        end
    end
endfunction: sample

endclass

```

See “[UVM Output Example](#)” on page 63 to view the full example.

## Supporting Coverage Models for Fields

Register Assistant UVM enables you to generate coverage for UVM register fields. Using this feature, you can easily map OVM designs with field coverage properties to the UVM related behavior.

### Procedure


1. To generate coverage models for fields:
2. Add an optional “Field is Covered” column to your CSV input files.

### Note

 The default value for the “Field is Covered” column is FALSE. This indicates that the default Register Assistant UVM behavior is not to generate coverage information for register fields.

- Set the column value to TRUE for fields that requires coverage.

### Note

 If you set the “Field is Covered” column value to TRUE for a reserved field, Register Assistant UVM issues a warning message during generation.

## Examples

In this example, register “stopwatch\_csr” fields “stride”, “updown”, “upper\_limit\_reached” and “lower\_limit\_reached” are to be covered. We therefore, set the “Field is Covered” flag value to TRUE for each of these fields in the CSV input file.

Register Name	Register Address	Register Width	Register Access	Register Reset Value	Field Name	Field Offset	Field Width	Field Access	Field Reset Value	Field is Covered	Field is Reserved
stopwatch_value	0x0	32	RW	0x0							
stopwatch_reset_value	0x04	32	RW	0x0							
stopwatch_upper_limit	0x08	32	RW	0x0							
stopwatch_lower_limit	0x0C	32	RW	0x0							
stopwatch_memory	0x010	32	RW	0x0							
stopwatch_csr	0x030	32	RW	0x0	padding	7	25	RO	0x0	FALSE	TRUE
stopwatch_csr	0x030	32	RW	0x0	stride	3	4	RW	0x0	TRUE	
stopwatch_csr	0x030	32	RW	0x0	updown	2	1	RW	0x0	TRUE	
stopwatch_csr	0x030	32	RW	0x0	upper_limit_reached	1	1	RO	0x0	TRUE	
stopwatch_csr	0x030	32	RW	0x0	lower_limit_reached	0	1	RO	0x0	TRUE	
stopwatch_counter	0x034	32	RW	0x0							

Examine the generated code for the Control Status Register and note the field coverage code added.

```
// Class: stopwatch_csr
//
// Control Status Register
//-----

class stopwatch_csr extends uvm_reg;
    `uvm_object_utils(stopwatch_csr)

    uvm_reg_field padding
    rand uvm_reg_field stride;
    rand uvm_reg_field updown;
    uvm_reg_field upper_limit_reached;
    uvm_reg_field lower_limit_reached;
```

By setting the field coverage option in your input files, the generated UVM register package will create a single covergroup called “cg\_vals”. For each non-reserved field whose “Field is Covered” flag is TRUE, a coverpoint with the name of the field is created.

```
// Function: coverage
//
covergroup cg_vals;
    stride          : coverpoint stride.value[3:0];
    updown          : coverpoint updown.value[0];
    upper_limit_reached : coverpoint upper_limit_reached.value[0];
    lower_limit_reached : coverpoint lower_limit_reached.value[0];
endgroup
```

In the function “new”, the coverage is set to UVM\_CVR\_FIELD\_VALS and the covergroup is constructed.

```
// Function: new
//
function new(string name = "stopwatch_csr");
    super.new(name, 32, build_coverage(UVM_CVR_FIELD_VALS));
    add_coverage(build_coverage(UVM_CVR_FIELD_VALS))
    if (has_coverage(UVM_CVR_FIELD_VALS)) begin
        cg_vals = new();
    end
endfunction
```

A function “sample\_values” is created to check whether the coverage is set to UVM\_CVR\_FIELD\_VALS and accordingly call the “sample” method for the created covergroup.

```
// Function: sample_values
//
virtual function void sample_values();
    super.sample_values();
    if (get_coverage(UVM_CVR_FIELD_VALS))
        cg_vals.sample();
endfunction

// Function: build
//
virtual function void build();
    padding = uvm_reg_field::type_id::create("padding");
    stride = uvm_reg_field::type_id::create("stride");
    updown = uvm_reg_field::type_id::create("updown");
    upper_limit_reached =
uvm_reg_field::type_id::create("upper_limit_reached");
    lower_limit_reached =
uvm_reg_field::type_id::create("lower_limit_reached");

    padding.configure(this, 25, 7, "RW", 0, 25'b0, 1, 0, 0);
    stride.configure(this, 4, 3, "RW", 0, 4'h0, 1, 1, 0);
    updown.configure(this, 1, 2, "RW", 0, 1'b0, 1, 1, 0);
    upper_limit_reached.configure(this, 1, 1, "RO", 0, 1'b0, 1, 0, 0);
    lower_limit_reached.configure(this, 1, 0, "RO", 0, 1'b0, 1, 0, 0);
endfunction
endclass
```

## Supporting Simple “Quirky” Registers

When generating the UVM register package, Register Assistant UVM depends on built-in packages to describe the registers. For example, when defining registers, Register Assistant UVM extends from the class `uvm_reg` available in its built-in package `uvm_pkg`. Through the support of “quirky” (custom) registers, you can rely on user-defined packages and, hence, extend from your own user-defined register classes.

This feature can be useful, for example, when you want to use a new access mode other than those available in Register Assistant’s package, so you will be able to define this access mode in your own quirky register class. This is applicable to any other register properties you need to customize.

### Procedure

1. To use quirky registers, you need to write the definition of your quirky register class in a package, and then you should do the following:
2. On the project level, add a CSV column for the import statement of your package. This is done through the column titled “Project Extra Imports”.
3. On the register level, add a CSV column to specify the corresponding user-defined register class which you will use to extend the register. This is done through the column titled “Register Custom Type”.

### Examples

The “Project Extra Imports” column is added in the CSV input file and given the value `mypkg::*,pkg2::*`. Hence, Register Assistant UVM adds the following imports at the beginning of the generated output:

```
import mypkg::*;  
import pkg2::*;
```

At the same time, the “Register Custom Type” column is added in the CSV input file and given the value `my_custom_reg`, specifically on the row of the register titled `stopwatch_counter`.



Hence, Register Assistant UVM extends from the class `my_custom_reg` for the register `stopwatch_counter` only (instead of extending from `uvm_reg`) as follows:

```
//-----  
// Class: stopwatch_counter  
//  
// Stop Watch Counter  
//-----  
  
class stopwatch_counter extends my_custom_reg;  
  `uvm_object_utils(stopwatch_counter)  
  
  rand uvm_reg_field F;  
  
// Function: new  
//  
  function new(string name = "stopwatch_counter");  
    super.new(name, 32, UVM_NO_COVERAGE);  
  endfunction  
  
// Function: build  
//  
  virtual function void build();  
    F = uvm_reg_field::type_id::create("F");  
    F.configure(this, 32, 0, "RW", 1, 32'h00000000, 1, 1, 1);  
  endfunction  
endclass
```

## Supporting Back Door Access

The UVM generator allows you to access registers through the regular bus cycles (“front door”) or directly via the hierarchical pathname used in the target simulator (“back door”). This backdoor approach allows registers/memories to be read and written in zero simulator time and, therefore, allows the registers/memories in the system being simulated to be put into a known state much faster than by performing the appropriate bus cycles.

It also enables the reading of the register/memory values by the testbench without moving simulator time forward and potentially changing the system state.

For maximum reuse, it is recommended to specify just the path segment for each level in the design hierarchy although you can specify multiple levels of the path to a given object if required.

Register Assistant UVM allows you to specify backdoor paths for blocks, instances within blocks (registers, memories or other blocks), and fields within registers.

## Procedure

1. You can specify backdoor paths through your CSV input files by doing the following:
  - If you have multiple fields within your registers, add the “Field Backdoor” column in the CSV file describing your registers. You can use this column to specify the backdoor path for each field.
  - In the CSV file describing your blocks, add the “Block Backdoor” and “Block Instance Backdoor” columns. These columns allow you to specify the simulation backdoor path for the block and for a given instance (register, memory or sub-block) within a block respectively.
2. Refer to the following example for further details.

## Examples

Consider the following example CSV input files and the resulting UVM generated output.

### Input:

The CSV definition file for registers is as follows:

**Figure 4-2. Register Description.**

Register Name	Register Address	Register Width	Field Name	Field Offset	Field Width	Field IsReserved	Field Backdoor
status_reg	0x0	16	ctrl_bit	0	1		ctrl_bit_status_reg_h_local
status_reg			other_field	1	2		other_field_status_reg_h_local
status_reg			reserved	3	13	TRUE	
RegA	0x2	16	regAfield1	0	8		regAfield1_RegA_h_local
RegA			regAfield2	8	8		regAfield2_RegA_h_local
RegB	0x4	16	myField	0	16		
small_mem	0x10	16	field1	0	8		field1_small_mem_h
small_mem			field2	8	8		field2_small_mem_h

In this file, note the added column titled “Field Backdoor”. It contains the backdoor path on the level of each field.

The CSV definition file for blocks is as follows:

**Figure 4-3. Block Description**

Block Name	Block Backdoor	Block Component Name	Block Instance Name	Block Instance Dimension	Block Instance Type	Block Instance Backdoor
reg_block	top.dut	status_reg	status_reg_h		register	%(FIELDS)
reg_block		sub_block	sub_block_inst		block	sub_block_inst
reg_block		RegB	RegB_h		register	myField_RegB_h_local
reg_block		small_mem	small_mem_h	8	register	%(FIELDS)_array_%(DIM)_local
sub_block		RegA	RegA_h		register	%(FIELDS)

In this file, note the “Block Backdoor” and “Block Instance Backdoor” columns. The “Block Backdoor” column indicates the simulator path for a specific block. The “Block Instance Backdoor” column indicates the path or path segment for a given instance (register, memory or sub-block) within a block.

Note the following:

- You can leave a cell empty which indicates that no backdoor path is required.
- You can directly specify a backdoor path.
- You can use the `%(FIELDS)` variable which indicates that a backdoor path is required and that the tool can use the field paths specified for the field in the “Field Backdoor” column (see [Figure 4-2](#) and [Figure 4-3](#)).
- If the block instance has an array, you can use the variable `%(DIM)` to refer to each instance of the expanded array. For example, you can enter the following value in the cell: `%(FIELDS)_array_%(DIM)_local`

The overall backdoor path to a given instance (sub-block, register or memory) is determined during simulation by concatenating the paths for each level. In the above example, the path to the last register in the instance array would be:

`top.dut.field1_small_mem_h_array_7_local`

See the Output section below for more details.

---

**Note**

For arrays of instances, the `%(DIM)` variable can be combined with the `%(FIELDS)` in order to create the appropriate backdoor path expression.

For information on the `%(DIM)` and `%(FIELDS)` variables, refer to the UVM Variables table available on `<installation_folder>\registerassistantuvm\docs\pdfdocs\RUVm_reference\UVM_Variables.pdf`.

---

**Output:**

The UVM output generated by Register Assistant UVM will be affected by your backdoor access definitions as follows:

- For the “Field Backdoor”, the generated UVM output requires a “slice” to be specified for each register field along with its bit offset and width, which are provided through the field definition. By adding the “Field Backdoor” path, the information can be used for any instance of that register simply by adding the `%(FIELDS)` variable for the “Block Instance Backdoor”.

In the generated UVM, the HDL path argument in the “configure” method is left blank and add\_hdl\_path\_slice statements are added for each field as follows:

```
<reg_inst>.add_hdl_path_slice("<field_path", offset, width, 0,  
<kind>);
```

```
status_reg_h = status_reg::type_id::create("status_reg_h");  
status_reg_h.configure(this, null, "");  
status_reg_h.add_hdl_path_slice("other_field_status_reg_h_local", 1, 2);  
status_reg_h.add_hdl_path_slice("ctrl_bit_status_reg_h_local", 0, 1);  
status_reg_h.build();
```

- The “Block Backdoor” is typically used for the top-level block (see the example in [Figure 4-3](#)) and would contain the hierarchy path from the simulator root down to the top-level block, for example “top.dut”. Depending on the hierarchical structure of the design, “Block Backdoor” may be left blank for lower-level blocks.

In the example, reg\_block is the top-level block. Therefore, the “Block Backdoor” path is specified as “top.dut”.

The path is added for the block instance as follows:

```
virtual function void build();  
    add_hdl_path("top.dut");
```

- For the “Block Instance Backdoor”, there are several cases according to which the output will be affected:

a. Register Instance without Fields:

In case of a register without fields (or when there is only one field spanning the register width), the path is the name of the register signal being assigned within the flip-flop “always” block, not the label of the “always” block which may be shown in the simulator hierarchy window.

The path is added into the “configure” statement for the block instance as follows:

```
RegB_h = RegB::type_id::create("RegB_h");  
RegB_h.configure(this, null, "myField_RegB_h_local");  
RegB_h.build();
```

b. Register Instance with Fields:

If a “Block Instance Backdoor” path is required for a register instance containing fields, the information specified in the field definitions will be automatically inserted if you enter the %(FIELDS) variable for the “Block Instance Backdoor” column. See [Figure 4-3](#).

## c. Array of Registers:

For an array of register instances, the system variable %(DIM) can be used in the path name. For example: data\_small\_mem\_h\_array\_%(DIM)\_local

This will be automatically expanded in the generated UVM as follows:

```
foreach ( small_mem_h[i] ) begin
  small_mem_h[i] = small_mem::type_id::create($psprintf("small_mem_h[%0d]", i));
  small_mem_h[i].configure(this, null, "");
  small_mem_h[i].add_hdl_path_slice(($psprintf("field2_small_mem_h_array_%0d_local",
i)), 8, 8);
  small_mem_h[i].add_hdl_path_slice(($psprintf("field1_small_mem_h_array_%0d_local",
i)), 0, 8);
  small_mem_h[i].build();
end
```

## d. Sub-block Instance:

The path for a sub-block instance (if required) is the instance name in the block, in the current example it is called sub\_block. In the generated UVM output, this would appear in the “configure” statement as follows:

```
sub_block_inst = sub_block::type_id::create("sub_block_inst");
sub_block_inst.configure(this, "sub_block_inst");
sub_block_inst.build();
```

- Following the above example, the resulting UVM backdoor paths used in the simulator would be as follows:

```
/top.dut.other_field_status_reg_h_local
/top.dut.ctrl_bit_status_reg_h_local
/top.dut.myField_RegB_h_local
/top.dut.field2_small_mem_h_array_0_local
/top.dut.field1_small_mem_h_array_0_local
/top.dut.field2_small_mem_h_array_1_local
/top.dut.field1_small_mem_h_array_1_local
/top.dut.field2_small_mem_h_array_2_local
/top.dut.field1_small_mem_h_array_2_local
/top.dut.field2_small_mem_h_array_3_local
/top.dut.field1_small_mem_h_array_3_local
/top.dut.field2_small_mem_h_array_4_local
/top.dut.field1_small_mem_h_array_4_local
/top.dut.field2_small_mem_h_array_5_local
/top.dut.field1_small_mem_h_array_5_local
/top.dut.field2_small_mem_h_array_6_local
/top.dut.field1_small_mem_h_array_6_local
/top.dut.field2_small_mem_h_array_7_local
/top.dut.field1_small_mem_h_array_7_local
/top.dut.sub_block_inst.regAfield2_RegA_h_local
/top.dut.sub_block_inst.regAfield1_RegA_h_local
```

## Word Addressable Output

By default, Register Assistant UVM assumes all addresses in your register definition files are byte-addressable and generates any output accordingly. In addition to byte addressing, Register Assistant UVM also supports word addressing; you have the ability to control the address mode in your input files to be word-based, thus affecting address representation in the generated output.

To elaborate on the difference between byte and word addressing, a byte is regarded as 8 bits, whereas a word is typically the address width of the processor, for example, 8, 16, 32, 64, or 128 bits. When expressed as a multiple of bytes, a 32-bit address bus consists of 4 bytes, which is the quotient of 32 divided by 8. Similarly, a 64-bit bus consists of 8 bytes, which is the quotient of 64 divided by 8.

In the default byte addressing mode, Register Assistant UVM assumes that all addresses are 1 byte (8 bits) apart. Hence, for a 32-bit bus, each entry in the address maps is 4 bytes apart, which means that address values increment by 4 to access the start of the next register. For example, the first register might be at address 0x0, the second at 0x4, the third at 0x8 and so on.

In the word addressing mode, address values simply increment by 1 word each time given that the word size is defined as multiples of 8-bit bytes. Using the 32-bit bus example, the first register might be at address 0x0, the second at 0x1, the third at 0x2 and so on.

See the following table:

**Table 4-1. Byte versus Word Addressing**

Word	Word #1				Word #2				Word #3			
Byte Addressing	0	1	2	3	4	5	6	7	8	9	10	11
Word Addressing	0				1				2			

As shown in the table, for byte addressing, the first word starts at address 0x0, the second word at address 0x4, whereas for word addressing (in this 32-bit example) all 4 bytes of the first word are at address 0x0, all 4 bytes for the second word at 0x1 and so on.

**Applying Word Addressing in Register Assistant UVM . . . . . 46**

## Applying Word Addressing in Register Assistant UVM

This section describes how you can control the addressing mode in Register Assistant UVM so it would be based on word addressing, rather than the default byte addressing.

### Procedure

1. Define your register definitions in CSV format.

2. Set the word addressing feature on the block map level by adding the following columns in your CSV input files:

**Table 4-2. Word Addressability CSV Columns**

CSV Column Name	Value	Description
BlockMap Address Mode	Byte Word	Set the value to Word. Note that the default value is “Byte”.
BlockMap Word Bytes	Decimal Number	Specify the number of bytes in a word. The default value is “4”.

For more information, refer to the CSV Columns table available on  
`<installation_folder>\registerassistant\docs\pdfdocs\RUVm_reference\CSV_columns.pdf`.

#### Note



- Byte addressing is the default addressing mode unless you explicitly set word addressing.
- You have the ability to specify the endianness using the BlockMap Endian column.

3. Run Register Assistant UVM. For more information, refer to “[Register Assistant UVM in GUI Mode](#)” on page 57 or “[Using Register Assistant in Batch/Command-Line Mode](#)” on page 61.

## Results

Having set the word addressing mode, the following aspects will be affected on running Register Assistant UVM:

- The calculation of block sizes.
- The calculation of addresses of instances within instance arrays.
- The Address Overlap check will take word addressing into account. Refer to “[Default Checks](#)” on page 27 for further information on this check.

## Examples

The following is a CSV blockmap definition file in which the addressing mode is set as “Word” in the BlockMap Address Mode column, the number of bytes in a word is set as “4” in the

BlockMap Word Bytes column, the endianness is set as “UVM\_BIG\_ENDIAN” in the BlockMap Endian column.

Block Name	BlockMap Name	BlockMap Description	BlockMap is default	BlockMap Instance Name	BlockMap Instance Address	BlockMap Instance Access	BlockMap Word Bytes	BlockMap Address Mode	BlockMap Endian
sw_top_block	TOP_MAP	SW top bloc	TRUE	sw1.SUB_MAP1	0x1000	RW	4	WORD	UVM_BIG_ENDIAN
sw_top_block	TOP_MAP			sw2.SUB_MAP1	0x2000	RW			
sw_top_block	TOP_MAP			vreg1	0x3000	RW			
sw_top_block	TOP_MAP			vreg2	0x3001	RW			
sw_top_block	TOP_MAP			vreg3	0x3002	RW			
sw_top_block	TOP_MAP			my_reg1	0x3003	RW			
sw_top_block	TOP_MAP			mem1	0x3004	RW			
sw_sub_block	SUB_MAP1	SW sub bloc	TRUE	stopwatch_value_reg	0x0	RW	4	WORD	UVM_BIG_ENDIAN
sw_sub_block	SUB_MAP1			stopwatch_reset_value_reg	0x1	RW			
sw_sub_block	SUB_MAP1			stopwatch_upper_limit_reg	0x2	RW			
sw_sub_block	SUB_MAP1			stopwatch_lower_limit_reg	0x3	RW			
sw_sub_block	SUB_MAP1			stopwatch_csr_reg	0x4	RW			
sw_sub_block	SUB_MAP1			stopwatch_memory_reg	0x5	RW			

The following are excerpts of a UVM output example generated for the above inputs. To view the full example, refer to “[UVM Word Addressable Output Example](#)” on page 73.

- In the following UVM code snippet, note the create\_map() method which is affected by the word addressability settings in the CSV input files. This method allows the creation of an address map in a block. The arguments of this method are:
  - name — The name of the address map.
  - base\_addr — The base address for the address map.
  - n\_bytes — The byte-width of the bus on which the map is used. This is related to the UVM parameter uvmgen.N\_BYTES as will be explained later.
  - endian — The endianness setting. This is based on the value of the BlockMap Endian column as will be explained later.
  - byte\_addressing — The addressing mode. The value “0” indicates word addressing, and “1” indicates byte addressing.

Note the usage of the endianness setting, which is already defined in the BlockMap Endian column in the CSV input, as an argument in the create\_map() method in the following code snippet.



```
// Function: build
//
virtual function void build();

    if (has_coverage(UVM_CVR_ADDR_MAP)) begin
        SUB_MAP1_cg =
sw_sub_block_SUB_MAP1_coverage::type_id::create("SUB_MAP1_cg");
        SUB_MAP1_cg.ra_cov.set_inst_name(this.get_full_name());
        void'(set_coverage(UVM_CVR_ADDR_MAP));
    end
    stopwatch_value_reg =
stopwatch_value::type_id::create("stopwatch_value_reg");
    stopwatch_value_reg.configure(this);
    stopwatch_value_reg.build();

    stopwatch_reset_value_reg =
stopwatch_reset_value::type_id::create("stopwatch_reset_value_reg")
;
    stopwatch_reset_value_reg.configure(this);
    stopwatch_reset_value_reg.build();

    stopwatch_upper_limit_reg =
stopwatch_upper_limit::type_id::create("stopwatch_upper_limit_reg")
;
    stopwatch_upper_limit_reg.configure(this);
    stopwatch_upper_limit_reg.build();

    stopwatch_lower_limit_reg =
stopwatch_lower_limit::type_id::create("stopwatch_lower_limit_reg")
;
    stopwatch_lower_limit_reg.configure(this);
    stopwatch_lower_limit_reg.build();

    stopwatch_csr_reg =
stopwatch_csr::type_id::create("stopwatch_csr_reg");
    stopwatch_csr_reg.configure(this);
    stopwatch_csr_reg.build();

    foreach ( stopwatch_memory_reg[i] ) begin
        stopwatch_memory_reg[i] =
stopwatch_memory::type_id::create($sprintf("stopwatch_memory_reg[%d]", i));
        stopwatch_memory_reg[i].configure(this);
        stopwatch_memory_reg[i].build();
    end

    SUB_MAP1 = create_map("SUB_MAP1", 'h0, 4, UVM_BIG_ENDIAN,
0);
    default_map = SUB_MAP1;

    SUB_MAP1.add_reg(stopwatch_value_reg, 'h0, "RW");
    SUB_MAP1.add_reg(stopwatch_reset_value_reg, 'h1, "RW");
    SUB_MAP1.add_reg(stopwatch_upper_limit_reg, 'h2, "RW");
    SUB_MAP1.add_reg(stopwatch_lower_limit_reg, 'h3, "RW");
    SUB_MAP1.add_reg(stopwatch_csr_reg, 'h4, "RW");
    foreach(stopwatch_memory_reg[i]) begin
        SUB_MAP1.add_reg(stopwatch_memory_reg[i], (i * ('h1)) +
('h5), "RW");
```

end

It is also important to note that the parameter `uvmgen.N_BYTES` can be used in the CSV input files to define the word size of the bus to which the address map is associated. The value of this parameter, which should be specified in terms of bytes, is used as an argument by the `create_map()` method. As shown in the above example, the size is “4” bytes.

If this parameter is not specified or if its value is invalid (not a positive integer), then Register Assistant UVM calculates the value automatically and a warning is raised as shown in the following example:

```
Warning: Map 'SW_MAP2' has parameter 'uvmgen.N_BYTES' set to invalid
value 'x'. Only positive integers are allowed. Using automatically
calculated value '4' instead.
```

For more information on this parameter, refer to the UVM Parameters table available on <installation\_folder>\registerassistant\docs\pdfdocs\RUVM\_reference\UVM\_Parameters.pdf.

- In the above UVM code snippet, the word addressing settings also impacts the “foreach” loop in the calculation of the address increments in the second parameter.
- Another item in the generated code affected by the word addressing mode is covergroups. The following code snippet shows Register Assistant UVM’s automatic calculation of instance array addresses in coverpoints.

```

/* BLOCKS */

//-----
// Class: sw_sub_block_SUB_MAP1_coverage
//
// Coverage for the 'SUB_MAP1' in 'sw_sub_block'
//-----

class sw_sub_block_SUB_MAP1_coverage extends uvm_object;
    `uvm_object_utils(sw_sub_block_SUB_MAP1_coverage)

    covergroup ra_cov(string name) with function
    sample(uvm_reg_addr_t addr, bit is_read);

        option.per_instance = 1;
        option.name = name;

        ADDR: coverpoint addr {
            bins stopwatch_value_reg = {'h0};
            bins stopwatch_reset_value_reg = {'h1};
            bins stopwatch_upper_limit_reg = {'h2};
            bins stopwatch_lower_limit_reg = {'h3};
            bins stopwatch_csr_reg = {'h4};
            bins stopwatch_memory_reg[8] = {'h5,
                                           'h6,
                                           'h7,
                                           'h8,
                                           'h9,
                                           'ha,
                                           'hb,
                                           'hc};
        }

        RW: coverpoint is_read {
            bins RD = {1};
            bins WR = {0};
        }

        ACCESS: cross ADDR, RW;

    endgroup: ra_cov

    function new(string name = "sw_sub_block_SUB_MAP1_coverage");
        ra_cov = new(name);
    endfunction: new

    function void sample(uvm_reg_addr_t offset, bit is_read);
        ra_cov.sample(offset, is_read);
    endfunction: sample

endclass: sw_sub_block_SUB_MAP1_coverage

```

## Related Topics

[Word Addressable Output](#)



# Chapter 5

## Customization

---

This chapter explains the customization methods supported by Register Assistant UVM.

<b>Parameters .....</b>	<b>53</b>
<b>Using Parameters.....</b>	<b>53</b>

## Parameters

Register Assistant UVM provides a set of predefined parameters you can use with the UVM generator. These parameters enable you to pass to the tool's generator extra information that affect the output.

For example, when generating UVM output, you can use certain parameters to specify your own user-defined packages and force certain registers to extend from classes in these packages (such registers are referred to as quirky registers).

Register Assistant UVM supports a list of fixed parameters each performing a certain function. Refer to the UVM Parameters table available on `<installation_folder>\registerassistant\docs\pdfdocs\RUVM_reference\UVM_Parameters.pdf` to view the full list of supported parameters. The UVM Parameters table contains the description of each parameter, the scope to which the parameter applies, and other information. Each parameter can be applied to objects within a certain scope such as registers, memories, and projects.

To use parameters, you have to set them in the register in the CSV input files.

## Using Parameters

You can use the parameters feature by adding the required parameters in your CSV input files.

To use parameters in CSV files, you have to add the following columns:

- `<Object>` Parameter Name
- `<Object>` Parameter Value
- `<Object>` Parameter Description

See the following example which shows the use of parameters to define quirky registers. The quirky registers feature enables you to force certain registers to extend from user-defined

packages rather than built-in packages when generating UVM output. The following columns are added in the CSV files:

**Table 5-1. Parameters — Example**

Project Parameter Name	Project Parameter Value	Project Parameter Description
uvmgen.EXTRA_IMPORTS	mypkg::* pkg2::*	Extra imports

The example above sets the parameter “uvmgen.EXTRA\_IMPORTS” which signifies that you will define your own imports in the generated UVM register package as follows:

```
// Extra imports
import mypkg::*;
import pkg2::*;
```

In addition to the above columns, you need to add the following columns and place values on the same line of the register(s) that should extend from alternative parents:

**Table 5-2. Parameters — Example (Continued)**

Register Parameter Name	Register Parameter Value	Register Parameter Description
uvmgen.ALT_PARENT	my_custom_reg	Alternative parent for this register.

In this example, the parameter “uvmgen.ALT\_PARENT” is set and given a certain value which is the actual name of the alternative parent. For example, if the above values are placed on the same line of “Register\_X”, then in the generated UVM output, this register will extend from the class my\_custom\_reg instead of extending from the built-in class uvm\_reg.

To view a list of the supported UVM parameters, refer to *<installation\_folder>\registerassistant\docs\pdfdocs\RUVm\_reference\UVM\_Parameters.pdf*. Register Assistant UVM provides specific CSV columns that can substitute the use of parameters; refer to *<installation\_folder>\registerassistant\docs\pdfdocs\RUVm\_reference\CSV\_columns.pdf* for information on the CSV columns equivalent to the UVM parameters.

In addition, you can refer to “[Supporting Simple “Quirky” Registers](#)” on page 40 for more information on quirky registers and how they can be defined using specific CSV columns rather than parameters.

---

#### Note



- The parameters common among a number of blocks can be defined only once on the project level. That is to say, if you have a set of parameters with the same set of values,

you can define them only once on the project level rather than defining them on the level of each block separately.

- Generally, you can add parameters in the CSV files containing the register definitions or the block definitions. The same also applies to project parameters, but it should be noted that if you are adding project parameters in a separate CSV file, you will need to add a dummy “Block Name” column in that file.
-





# Chapter 6

## Register Assistant UVM in GUI Mode

---

Register Assistant UVM can be used either in batch mode or in GUI (graphical user interface) mode.

For information on how to use Register Assistant UVM in batch mode, refer to “[Using Register Assistant in Batch/Command-Line Mode](#)” on page 61.

This chapters explains how you can use Register Assistant UVM in GUI mode.

**Using Register Assistant UVM in GUI Mode** ..... **57**

## Using Register Assistant UVM in GUI Mode

This procedure invokes the Register Assistant UVM in GUI mode. You can change settings in the Register Assistant UVM wizard.

### Procedure

1. To invoke the tool as a graphical interface, enter the following command:


```
vreguvm -gui [-uvmout <output file path>]
```

Optionally, you can specify the path to save the generated UVM file using the `uvmout` switch as shown above. If this path is not specified, the generated file is saved in the current working directory. See [Table A-1](#) on page 61 for information.

Upon running the command, the Register Assistant UVM wizard is invoked.

---

#### Note

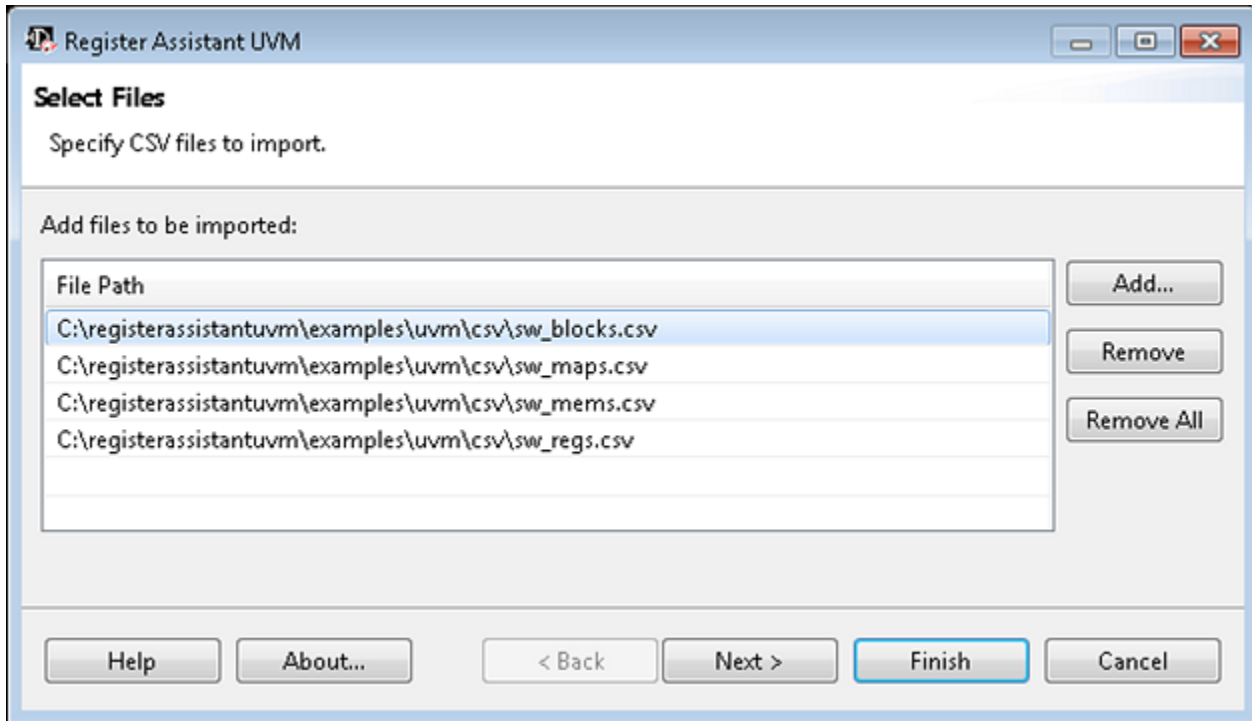
 Alternatively, you can invoke the Register Assistant UVM wizard by running the batch file named `vreguvm_gui.bat`. This file can be found on `<registerAssistantUVM_install_location>`.

---

2. On the Select Files page of the wizard, click **Add** and then browse for the CSV input files that contain your register definitions. You can add multiple definition files. Any added files are displayed in the table.

You can also select any of the added files and discard them using the **Remove** button, or you can directly click **Remove All** to remove all your selected files at once.

The **About** button opens an About dialog box showing the Register Assistant UVM version number, copyright information, and other important information.



3. Click **Next**. On the UVM Generator Settings page, set the following options:
  - a. In the Generator Options table, specify the following:
    - i. The name of the SystemVerilog file that will be generated. The default file is *vreguvm\_pkg\_uvm.sv*.
    - ii. The UVM version to use in generation. The supported UVM versions are 1.0 and 1.1; the default used by the tool is 1.1.
    - iii. The OVM Compatibility option, which can be set to “false” or “true”. This option allows you to make the generated output compatible for use within an OVM testbench. By setting this option to “true”, the tool includes the OVM macros and imports the OVM package (in addition to those of UVM).


The following code excerpt shows the generated output when this option is set to “false”:

```
import uvm_pkg::*;  
`include "uvm_macros.svh"
```

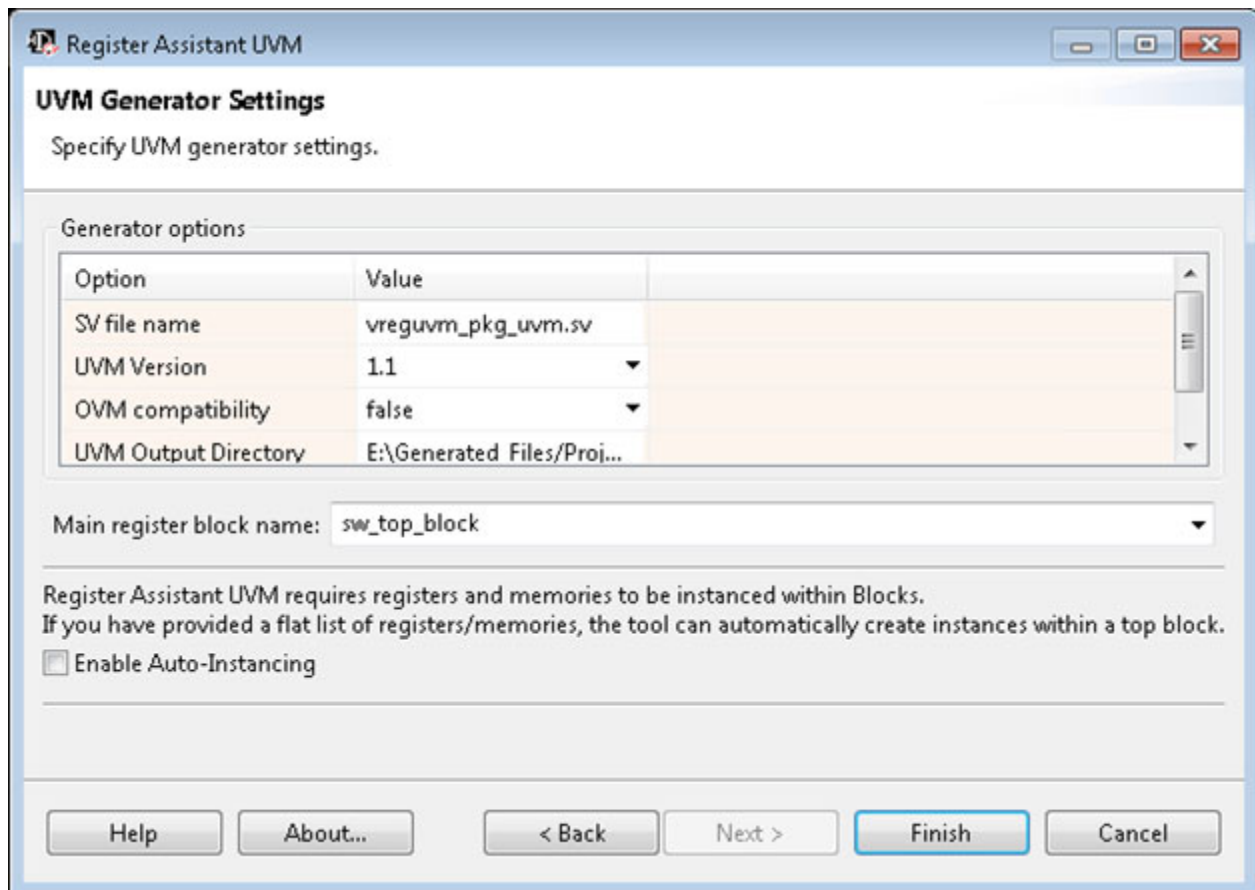
The following code excerpt shows the generated output when this option is set to “true”:

```
// Required packages for OVM backward compatibility
import ovm_pkg::*;
import uvm_reg_pkg::*;


// Required includes for OVM backward compatibility
`include "ovm_macros.svh"
`include "uvm_reg_macros.svh"
```

- iv. The output path of the generated file using the UVM Output Directory field. If you click in the field, you will be able to click on the browse button  and locate the required output directory.
- b. The Main register block name dropdown list displays all the blocks available in the input files. Select the name of your top block, or type a different name.
- c. You can set the Enable Auto-Instancing option, which allows you to automatically create instances for registers and memories in the top block within the generated output. Refer to “[Auto-Instancing](#)” on page 22.

Register Assistant UVM uses register and memory declaration names for instances.



**Note**

 If there are no blocks defined in the input files, the tool uses the name default\_top\_block in the Main register block name field and automatically sets the Enable Auto-Instancing option.

---

4. Click **Finish**.

Register Assistant UVM subsequently imports the specified CSV files, runs built-in checks on the content of the files, and then generates the SystemVerilog output file.

# Appendix A

## Using Register Assistant in Batch/ Command-Line Mode

---

Register Assistant UVM can be invoked in batch mode or as a graphical interface.

**Running Register Assistant UVM** ..... 61

## Running Register Assistant UVM

To run Register Assistant UVM in batch mode, use the below command:

### Procedure

1. Run the following:

```
vreguvm [-help] [-version] [-about]
-csvin <csv file 1> <csv file 2> ...
[-uvmout <output file path>]
[-uvmversion <UVM style>]
[-block <main block>]
[-autoinstance]
[-ovm]
```

2. You can optionally invoke Register Assistant UVM wizard using the below command:

```
vreguvm -gui [-uvmout <output file path>]
```

3. You can refer to “[Register Assistant UVM in GUI Mode](#)” on page 57 for more information on invoking and using Register Assistant UVM in the graphical interface mode.
4. See the following table for more information on the command switches you can use when invoking Register Assistant UVM:

**Table A-1. Command Line Switches**

Name	Description
-help	Lists all the command line switches that can be used with Register Assistant UVM and their description.
-version	Prints the current version of Register Assistant UVM.
-about	Prints the name and version of Register Assistant UVM.

**Table A-1. Command Line Switches (cont.)**

Name	Description
-csvin	Specifies the CSV input files containing the register definitions. You can specify more than one CSV file space-separated. This argument is required (in batch mode only).
-uvmout <output file path>	<p>Specifies the path on which the generated UVM file should be saved. If this argument is not provided, then the output will be generated in the current working directory.</p> <p>Note that absolute and relative paths (relative to the working directory) are supported, in addition to environment variables.</p>
-uvmversion <UVM style>	Specifies the UVM version number to use in generation. The supported versions are 1.0 and 1.1. If not specified, Register Assistant UVM uses version 1.1 by default.
-block <main block>	Specifies the name of the top block.
-autoinstance	Enables the auto-instancing feature. Refer to “ <a href="#">Auto-Instancing</a> ” on page 22 for more information.
-ovm	Allows the generated output to be compatible for use within an OVM testbench. By using this argument, the tool includes the OVM macros and imports the OVM package in the generated output (in addition to those of UVM).
-gui	Launches the Register Assistant UVM wizard.

# Appendix B Examples

---

This appendix contains miscellaneous examples for different Register Assistant UVM output files.

<b>UVM Output Example .....</b>	<b>63</b>
<b>UVM Word Addressable Output Example .....</b>	<b>73</b>

## UVM Output Example

The following example shows the UVM register package generated by Register Assistant UVM.

For more information on UVM generation, refer to “[UVM Output](#)” on page 30.

```
package sw_pkg_uvm;

import uvm_pkg::*;
import mypkg::*;

`include "uvm_macros.svh"

/* DEFINE REGISTER CLASSES */

//-----
// Class: stopwatch_lower_limit
//
// Lower limit
//-----

class stopwatch_lower_limit extends uvm_reg;
    `uvm_object_utils(stopwatch_lower_limit)

    rand uvm_reg_field F;

    // Function: new
    //
    function new(string name = "stopwatch_lower_limit");
        super.new(name, 32, UVM_NO_COVERAGE);
    endfunction

    // Function: build
    //
    virtual function void build();
        F = uvm_reg_field::type_id::create("F");
        F.configure(this, 32, 0, "RW", 1, 32'h00000000, 1, 1, 1);
    endfunction
endclass

//-----
// Class: stopwatch_upper_limit
//
// Upper limit
//-----

class stopwatch_upper_limit extends uvm_reg;
    `uvm_object_utils(stopwatch_upper_limit)

    rand uvm_reg_field F;

    // Function: new
    //
    function new(string name = "stopwatch_upper_limit");
        super.new(name, 32, UVM_NO_COVERAGE);
    endfunction

    // Function: build
    //
    virtual function void build();
        F = uvm_reg_field::type_id::create("F");
        F.configure(this, 32, 0, "RW", 1, 32'h00000000, 1, 1, 1);
    endfunction
endclass
```



```
//-----
// Class: stopwatch_reset_value
//
// Reset value
//-----

class stopwatch_reset_value extends uvm_reg;
  `uvm_object_utils(stopwatch_reset_value)

  rand uvm_reg_field F;

  // Function: new
  //
  function new(string name = "stopwatch_reset_value");
    super.new(name, 32, UVM_NO_COVERAGE);
  endfunction

  // Function: build
  //
  virtual function void build();
    F = uvm_reg_field::type_id::create("F");
    F.configure(this, 32, 0, "RW", 1, 32'h00000000, 1, 1, 1);
  endfunction
endclass

//-----
// Class: stopwatch_memory
//
// Memory register
//-----

class stopwatch_memory extends uvm_reg;
  `uvm_object_utils(stopwatch_memory)

  rand uvm_reg_field F;

  // Function: new
  //
  function new(string name = "stopwatch_memory");
    super.new(name, 32, UVM_NO_COVERAGE);
  endfunction

  // Function: build
  //
  virtual function void build();
    F = uvm_reg_field::type_id::create("F");
    F.configure(this, 32, 0, "RW", 1, 32'h00000000, 1, 1, 1);
  endfunction
endclass

//-----
// Class: stopwatch_csr
//
// Control Status Register
//-----

class stopwatch_csr extends uvm_reg;
```

```
`uvm_object_utils(stopwatch_csr)

uvm_reg_field padding; // Reserved
rand uvm_reg_field stride; // Stride length
rand uvm_reg_field updown; // Indicates whether counting up or down
uvm_reg_field upper_limit_reached; // Indicates that the upper limit
has been reached
uvm_reg_field lower_limit_reached; // Indicates that the lower limit
has been reached

// Function: coverage
//
covergroup cg_vals;
    stride : coverpoint stride.value[3:0];
    updown : coverpoint updown.value[0];
    upper_limit_reached : coverpoint upper_limit_reached.value[0];
    lower_limit_reached : coverpoint lower_limit_reached.value[0];
endgroup

// Constraints
constraint my_constraint {stride.value[3:0] < 5;}

// Function: new
//
function new(string name = "stopwatch_csr");
    super.new(name, 32, build_coverage(UVM_CVR_FIELD_VALS));
    add_coverage(build_coverage(UVM_CVR_FIELD_VALS));
    if (has_coverage(UVM_CVR_FIELD_VALS))
        cg_vals = new();
endfunction

// Function: sample_values
//
virtual function void sample_values();
    super.sample_values();
    if (get_coverage(UVM_CVR_FIELD_VALS))
        cg_vals.sample();
endfunction

// Function: build
//
virtual function void build();
    padding = uvm_reg_field::type_id::create("padding");
    stride = uvm_reg_field::type_id::create("stride");
    updown = uvm_reg_field::type_id::create("updown");
    upper_limit_reached =
uvm_reg_field::type_id::create("upper_limit_reached");
    lower_limit_reached =
uvm_reg_field::type_id::create("lower_limit_reached");

    padding.configure(this, 25, 7, "RW", 0,
25'b00000000000000000000000000000000, 1, 0, 0);
    stride.configure(this, 4, 3, "RW", 0, 4'h0, 1, 1, 0);
    updown.configure(this, 1, 2, "RW", 0, 1'b0, 1, 1, 0);
    upper_limit_reached.configure(this, 1, 1, "RO", 0, 1'b0, 1, 0, 0);
    lower_limit_reached.configure(this, 1, 0, "RO", 0, 1'b0, 1, 0, 0);
endfunction
endclass
```

```
//-----  
// Class: stopwatch_value  
//  
// Current value  
//-----  
  
class stopwatch_value extends uvm_reg;  
  `uvm_object_utils(stopwatch_value)  
  
  uvm_reg_field F;  
  
  // Function: new  
  //  
  function new(string name = "stopwatch_value");  
    super.new(name, 32, UVM_NO_COVERAGE);  
  endfunction  
  
  // Function: build  
  //  
  virtual function void build();  
    F = uvm_reg_field::type_id::create("F");  
    F.configure(this, 32, 0, "RO", 1, 32'h00000000, 1, 0, 1);  
  endfunction  
endclass  
  
//-----  
// Class: my_mem  
//  
// Memory  
//-----  
  
class my_mem extends uvm_mem;  
  `uvm_object_utils(my_mem)  
  
  // Function: new  
  //  
  function new(string name = "my_mem");  
    super.new(name, 'h400, 32, "RW", UVM_NO_COVERAGE);  
  endfunction  
endclass  
  
//-----  
// Class: my_reg  
//  
// Custom register  
//-----  
  
class my_reg extends my_reg_type;  
  `uvm_object_utils(my_reg)  
  
  rand uvm_reg_field F;  
  
  // Function: new  
  //  
  function new(string name = "my_reg");  
    super.new(name, 32, UVM_NO_COVERAGE);  
  endfunction
```

```
// Function: build
//
virtual function void build();
    F = uvm_reg_field::type_id::create("F");
    F.configure(this, 32, 0, "RW", 1, 32'h00000000, 1, 1, 1);
endfunction
endclass

//-----
// Class: stopwatch_counter
//
// Stop Watch Counter
//-----

class stopwatch_counter extends uvm_reg;
    `uvm_object_utils(stopwatch_counter)

    rand uvm_reg_field F;

    // Function: new
    //
    function new(string name = "stopwatch_counter");
        super.new(name, 32, UVM_NO_COVERAGE);
    endfunction

    // Function: build
    //
    virtual function void build();
        F = uvm_reg_field::type_id::create("F");
        F.configure(this, 32, 0, "RW", 1, 32'h00000000, 1, 1, 1);
    endfunction
endclass

/* BLOCKS */

//-----
// Class: sw_sub_block_SW_MAP_coverage
//
// Coverage for the 'SW_MAP' in 'sw_sub_block'
//-----

class sw_sub_block_SW_MAP_coverage extends uvm_object;
    `uvm_object_utils(sw_sub_block_SW_MAP_coverage)

    covergroup ra_cov(string name) with function sample(uvm_reg_addr_t
addr, bit is_read);

        option.per_instance = 1;
        option.name = name;

        ADDR: coverpoint addr {
            bins stopwatch_value_reg = {'h4};
            bins stopwatch_reset_value_reg = {'h8};
            bins stopwatch_upper_limit_reg = {'hc};
            bins stopwatch_lower_limit_reg = {'h10};
            bins stopwatch_csr_reg = {'h14};
            bins stopwatch_memory_reg[8] = {'h34,
```

```
'h38,
'h3c,
'h40,
'h44,
'h48,
'h4c,
'h50};
}

RW: coverpoint is_read {
    bins RD = {1};
    bins WR = {0};
}

ACCESS: cross ADDR, RW;

endgroup: ra_cov

function new(string name = "sw_sub_block_SW_MAP_coverage");
    ra_cov = new(name);
endfunction: new

function void sample(uvm_reg_addr_t offset, bit is_read);
    ra_cov.sample(offset, is_read);
endfunction: sample

endclass: sw_sub_block_SW_MAP_coverage

//-----
// Class: sw_sub_block
//
// Sub_block for the stopwatch design
//-----

class sw_sub_block extends uvm_reg_block;
    `uvm_object_utils(sw_sub_block)

    rand stopwatch_value stopwatch_value_reg; // Value instance
    rand stopwatch_reset_value stopwatch_reset_value_reg; // Reset Value
instance
    rand stopwatch_upper_limit stopwatch_upper_limit_reg; // Upper Limit
instance
    rand stopwatch_lower_limit stopwatch_lower_limit_reg; // Lower Limit
instance
    rand stopwatch_csr stopwatch_csr_reg; // CSR instance
    rand stopwatch_memory stopwatch_memory_reg[8]; // MEM instances

    uvm_reg_map SW_MAP; // SW sub block map
    sw_sub_block_SW_MAP_coverage SW_MAP_cg;

    // Function: new
    //
    function new(string name = "sw_sub_block");
        super.new(name, build_coverage(UVM_CVR_ALL));
    endfunction

    // Function: build
    //
```

```
virtual function void build();

    if (has_coverage(UVM_CVR_ADDR_MAP)) begin
        SW_MAP_cg =
sw_sub_block_SW_MAP_coverage::type_id::create("SW_MAP_cg");
        void'(set_coverage(UVM_CVR_ADDR_MAP));
    end
    stopwatch_value_reg =
stopwatch_value::type_id::create("stopwatch_value_reg");
    stopwatch_value_reg.configure(this);
    stopwatch_value_reg.build();

    stopwatch_reset_value_reg =
stopwatch_reset_value::type_id::create("stopwatch_reset_value_reg");
    stopwatch_reset_value_reg.configure(this);
    stopwatch_reset_value_reg.build();

    stopwatch_upper_limit_reg =
stopwatch_upper_limit::type_id::create("stopwatch_upper_limit_reg");
    stopwatch_upper_limit_reg.configure(this);
    stopwatch_upper_limit_reg.build();

    stopwatch_lower_limit_reg =
stopwatch_lower_limit::type_id::create("stopwatch_lower_limit_reg");
    stopwatch_lower_limit_reg.configure(this);
    stopwatch_lower_limit_reg.build();

    stopwatch_csr_reg =
stopwatch_csr::type_id::create("stopwatch_csr_reg");
    stopwatch_csr_reg.configure(this);
    stopwatch_csr_reg.build();

    foreach ( stopwatch_memory_reg[i] ) begin
        stopwatch_memory_reg[i] =
stopwatch_memory::type_id::create($sprintf("stopwatch_memory_reg[%0d]",
i));
        stopwatch_memory_reg[i].configure(this, null,
($sprintf("f_stopwatch_memory_reg_%0d", i)));
        stopwatch_memory_reg[i].build();
    end

    SW_MAP = create_map("SW_MAP", 'h0, 4, UVM_LITTLE_ENDIAN, 1);
    default_map = SW_MAP;

    SW_MAP.add_reg(stopwatch_value_reg, 'h4, "RW");
    SW_MAP.add_reg(stopwatch_reset_value_reg, 'h8, "RW");
    SW_MAP.add_reg(stopwatch_upper_limit_reg, 'hc, "RW");
    SW_MAP.add_reg(stopwatch_lower_limit_reg, 'h10, "RW");
    SW_MAP.add_reg(stopwatch_csr_reg, 'h14, "RW");
    foreach(stopwatch_memory_reg[i]) begin
        SW_MAP.add_reg(stopwatch_memory_reg[i], (i * ('h4)) + ('h34),
"RW");
    end

    lock_model();
endfunction

// Function: sample
```

```
//
function void sample(uvm_reg_addr_t offset, bit is_read, uvm_reg_map
map);
    if(get_coverage(UVM_CVR_ADDR_MAP)) begin
        if(map.get_name() == "SW_MAP") begin
            SW_MAP_cg.sample(offset, is_read);
        end
    end
endfunction: sample

endclass

//-----
// Class: sw_top_block_SW_MAP2_coverage
//
// Coverage for the 'SW_MAP2' in 'sw_top_block'
//-----

class sw_top_block_SW_MAP2_coverage extends uvm_object;
    `uvm_object_utils(sw_top_block_SW_MAP2_coverage)

    covergroup ra_cov(string name) with function sample(uvm_reg_addr_t
addr, bit is_read);

        option.per_instance = 1;
        option.name = name;

        ADDR: coverpoint addr {
            bins vreg1 = {'h3000};
            bins vreg2 = {'h3004};
            bins vreg3 = {'h3008};
            bins my_reg1 = {'h300c};
        }

        RW: coverpoint is_read {
            bins RD = {1};
            bins WR = {0};
        }

        ACCESS: cross ADDR, RW;

    endgroup: ra_cov

    function new(string name = "sw_top_block_SW_MAP2_coverage");
        ra_cov = new(name);
    endfunction: new

    function void sample(uvm_reg_addr_t offset, bit is_read);
        ra_cov.sample(offset, is_read);
    endfunction: sample

endclass: sw_top_block_SW_MAP2_coverage

//-----
// Class: sw_top_block
//
// Top block for the stopwatch design
//-----
```

```
class sw_top_block extends uvm_reg_block;
  `uvm_object_utils(sw_top_block)

  rand sw_sub_block sw1; // Block1 instance 1
  rand sw_sub_block sw2; // Block1 instance 2
  rand stopwatch_counter vreg1; // Counter instance 1
  rand stopwatch_counter vreg2; // Counter instance 2
  rand stopwatch_counter vreg3; // Counter instance 3
  rand my_reg my_reg1; // Custom register instance
  my_mem mem1; // Memory instance

  uvm_reg_map SW_MAP2; // SW top block map
  sw_top_block_SW_MAP2_coverage SW_MAP2_cg;

  // Function: new
  //
  function new(string name = "sw_top_block");
    super.new(name, build_coverage(UVM_CVR_ALL));
  endfunction

  // Function: build
  //
  virtual function void build();

    add_hdl_path("top.dut");

    if(has_coverage(UVM_CVR_ADDR_MAP)) begin
      SW_MAP2_cg =
sw_top_block_SW_MAP2_coverage::type_id::create("SW_MAP2_cg");
      void'(set_coverage(UVM_CVR_ADDR_MAP));
    end
    sw1 = sw_sub_block::type_id::create("sw1");
    sw1.configure(this);
    sw1.build();

    sw2 = sw_sub_block::type_id::create("sw2");
    sw2.configure(this);
    sw2.build();

    vreg1 = stopwatch_counter::type_id::create("vreg1");
    vreg1.configure(this, null, "f_vreg1");
    vreg1.build();

    vreg2 = stopwatch_counter::type_id::create("vreg2");
    vreg2.configure(this, null, "f_vreg2");
    vreg2.build();

    vreg3 = stopwatch_counter::type_id::create("vreg3");
    vreg3.configure(this, null, "f_vreg3");
    vreg3.build();

    my_reg1 = my_reg::type_id::create("my_reg1");
    my_reg1.configure(this);
    my_reg1.build();

    mem1 = my_mem::type_id::create("mem1");
    mem1.configure(this);
```



```
SW_MAP2 = create_map("SW_MAP2", 'h0, 4, UVM_LITTLE_ENDIAN, 1);
default_map = SW_MAP2;

SW_MAP2.add_submap(sw1.SW_MAP, 'h1000);
SW_MAP2.add_submap(sw2.SW_MAP, 'h2000);
SW_MAP2.add_reg(vreg1, 'h3000, "RW");
SW_MAP2.add_reg(vreg2, 'h3004, "RW");
SW_MAP2.add_reg(vreg3, 'h3008, "RW");
SW_MAP2.add_reg(my_reg1, 'h300c, "RW");
SW_MAP2.add_mem(mem1, 'h4000, "RW");

    uvm_resource_db #(bit)::set({"REG::",
this.vreg1.get_full_name()}, "NO_REG_HW_RESET_TEST", 1);
    lock_model();
endfunction

// Function: sample
//
function void sample(uvm_reg_addr_t offset, bit is_read, uvm_reg_map
map);
    if(get_coverage(UVM_CVR_ADDR_MAP)) begin
        if(map.get_name() == "SW_MAP2") begin
            SW_MAP2_cg.sample(offset, is_read);
        end
    end
endfunction: sample

endclass

endpackage
```

## UVM Word Addressable Output Example

The following example shows the UVM code generated by Register Assistant UVM with the word addressing mode set.

In the register definition files, the BlockMap Address Mode is set as “Word” and the BlockMap Word Bytes is set as “4”.

For more information on word addressing, refer to “[Word Addressable Output](#)” on page 46.

```
package output_pkg_uvm;

import uvm_pkg::*;
import mypkg::*;

`include "uvm_macros.svh"

/* DEFINE REGISTER CLASSES */

//-----
// Class: stopwatch_lower_limit
//
// Lower limit
//-----

class stopwatch_lower_limit extends uvm_reg;
  `uvm_object_utils(stopwatch_lower_limit)

  rand uvm_reg_field F;

  // Function: new
  //
  function new(string name = "stopwatch_lower_limit");
    super.new(name, 32, UVM_NO_COVERAGE);
  endfunction

  // Function: build
  //
  virtual function void build();
    F = uvm_reg_field::type_id::create("F");
    F.configure(this, 32, 0, "RW", 1, 32'h00000000, 1, 1, 1);
  endfunction
endclass

//-----
// Class: stopwatch_upper_limit
//
// Upper limit
//-----

class stopwatch_upper_limit extends uvm_reg;
  `uvm_object_utils(stopwatch_upper_limit)

  rand uvm_reg_field F;

  // Function: new
  //
  function new(string name = "stopwatch_upper_limit");
    super.new(name, 32, UVM_NO_COVERAGE);
  endfunction

  // Function: build
  //
  virtual function void build();
    F = uvm_reg_field::type_id::create("F");
    F.configure(this, 32, 0, "RW", 1, 32'h00000000, 1, 1, 1);
  endfunction
endclass
```

```
//-----  
// Class: stopwatch_reset_value  
//  
// Reset value  
//-----  
  
class stopwatch_reset_value extends uvm_reg;  
  `uvm_object_utils(stopwatch_reset_value)  
  
  rand uvm_reg_field F;  
  
  // Function: new  
  //  
  function new(string name = "stopwatch_reset_value");  
    super.new(name, 32, UVM_NO_COVERAGE);  
  endfunction  
  
  // Function: build  
  //  
  virtual function void build();  
    F = uvm_reg_field::type_id::create("F");  
    F.configure(this, 32, 0, "RW", 1, 32'h00000000, 1, 1, 1);  
  endfunction  
endclass  
  
//-----  
// Class: stopwatch_memory  
//  
// Memory register  
//-----  
  
class stopwatch_memory extends uvm_reg;  
  `uvm_object_utils(stopwatch_memory)  
  
  rand uvm_reg_field F;  
  
  // Function: new  
  //  
  function new(string name = "stopwatch_memory");  
    super.new(name, 32, UVM_NO_COVERAGE);  
  endfunction  
  
  // Function: build  
  //  
  virtual function void build();  
    F = uvm_reg_field::type_id::create("F");  
    F.configure(this, 32, 0, "RW", 1, 32'h00000000, 1, 1, 1);  
  endfunction  
endclass  
  
//-----  
// Class: stopwatch_csr  
//  
// Control Status Register  
//-----  
  
class stopwatch_csr extends uvm_reg;
```

```

    `uvm_object_utils(stopwatch_csr)

    uvm_reg_field padding; // Reserved
    rand uvm_reg_field stride; // Stride length
    rand uvm_reg_field updown; // Indicates whether counting up or down
    uvm_reg_field upper_limit_reached; // Indicates that the upper limit
has been reached
    uvm_reg_field lower_limit_reached; // Indicates that the lower limit
has been reached

    // Function: coverage
    //
    covergroup cg_vals;
        stride : coverpoint stride.value[3:0];
        updown : coverpoint updown.value[0];
        upper_limit_reached : coverpoint upper_limit_reached.value[0];
        lower_limit_reached : coverpoint lower_limit_reached.value[0];
    endgroup

    // Constraints
    constraint my_constraint {x > 5;}

    // Function: new
    //
    function new(string name = "stopwatch_csr");
        super.new(name, 32, build_coverage(UVM_CVR_FIELD_VALS));
        add_coverage(build_coverage(UVM_CVR_FIELD_VALS));
        if (has_coverage(UVM_CVR_FIELD_VALS))
            cg_vals = new();
    endfunction

    // Function: sample_values
    //
    virtual function void sample_values();
        super.sample_values();
        if (get_coverage(UVM_CVR_FIELD_VALS))
            cg_vals.sample();
    endfunction

    // Function: build
    //
    virtual function void build();
        padding = uvm_reg_field::type_id::create("padding");
        stride = uvm_reg_field::type_id::create("stride");
        updown = uvm_reg_field::type_id::create("updown");
        upper_limit_reached =
uvm_reg_field::type_id::create("upper_limit_reached");
        lower_limit_reached =
uvm_reg_field::type_id::create("lower_limit_reached");

        padding.configure(this, 25, 7, "RW", 0,
25'b00000000000000000000000000000000, 1, 0, 0);
        stride.configure(this, 4, 3, "RW", 0, 4'h0, 1, 1, 0);
        updown.configure(this, 1, 2, "RW", 0, 1'b0, 1, 1, 0);
        upper_limit_reached.configure(this, 1, 1, "RO", 0, 1'b0, 1, 0, 0);
        lower_limit_reached.configure(this, 1, 0, "RO", 0, 1'b0, 1, 0, 0);
    endfunction
endclass

```

```
//-----  
// Class: stopwatch_value  
//  
// Current value  
//-----  
  
class stopwatch_value extends uvm_reg;  
  `uvm_object_utils(stopwatch_value)  
  
  uvm_reg_field F;  
  
  // Function: new  
  //  
  function new(string name = "stopwatch_value");  
    super.new(name, 32, UVM_NO_COVERAGE);  
  endfunction  
  
  // Function: build  
  //  
  virtual function void build();  
    F = uvm_reg_field::type_id::create("F");  
    F.configure(this, 32, 0, "RO", 1, 32'h00000000, 1, 0, 1);  
  endfunction  
endclass  
  
//-----  
// Class: my_mem  
//  
// Memory  
//-----  
  
class my_mem extends uvm_mem;  
  `uvm_object_utils(my_mem)  
  
  // Function: new  
  //  
  function new(string name = "my_mem");  
    super.new(name, 'h400, 32, "RW", UVM_NO_COVERAGE);  
  endfunction  
endclass  
  
//-----  
// Class: my_reg  
//  
// Custom register  
//-----  
  
class my_reg extends my_reg_type;  
  `uvm_object_utils(my_reg)  
  
  rand uvm_reg_field F;  
  
  // Function: new  
  //  
  function new(string name = "my_reg");  
    super.new(name, 32, UVM_NO_COVERAGE);  
  endfunction
```

```

        // Function: build
        //
        virtual function void build();
            F = uvm_reg_field::type_id::create("F");
            F.configure(this, 32, 0, "RW", 1, 32'h00000000, 1, 1, 1);
        endfunction
    endclass

//-----
// Class: stopwatch_counter
//
// Stop Watch Counter
//-----

class stopwatch_counter extends uvm_reg;
    `uvm_object_utils(stopwatch_counter)

    rand uvm_reg_field F;

    // Function: new
    //
    function new(string name = "stopwatch_counter");
        super.new(name, 32, UVM_NO_COVERAGE);
    endfunction

    // Function: build
    //
    virtual function void build();
        F = uvm_reg_field::type_id::create("F");
        F.configure(this, 32, 0, "RW", 1, 32'h00000000, 1, 1, 1);
    endfunction
endclass

/* BLOCKS */

//-----
// Class: sw_sub_block_SUB_MAP1_coverage
//
// Coverage for the 'SUB_MAP1' in 'sw_sub_block'
//-----

class sw_sub_block_SUB_MAP1_coverage extends uvm_object;
    `uvm_object_utils(sw_sub_block_SUB_MAP1_coverage)

    covergroup ra_cov(string name) with function sample(uvm_reg_addr_t
addr, bit is_read);

        option.per_instance = 1;
        option.name = name;

        ADDR: coverpoint addr {
            bins stopwatch_value_reg = {'h0};
            bins stopwatch_reset_value_reg = {'h1};
            bins stopwatch_upper_limit_reg = {'h2};
            bins stopwatch_lower_limit_reg = {'h3};
            bins stopwatch_csr_reg = {'h4};
            bins stopwatch_memory_reg[8] = {'h5,

```

```
        'h6,  
        'h7,  
        'h8,  
        'h9,  
        'ha,  
        'hb,  
        'hc};  
    }  
  
    RW: coverpoint is_read {  
        bins RD = {1};  
        bins WR = {0};  
    }  
  
    ACCESS: cross ADDR, RW;  
  
endgroup: ra_cov  
  
function new(string name = "sw_sub_block_SUB_MAP1_coverage");  
    ra_cov = new(name);  
endfunction: new  
  
function void sample(uvm_reg_addr_t offset, bit is_read);  
    ra_cov.sample(offset, is_read);  
endfunction: sample  
  
endclass: sw_sub_block_SUB_MAP1_coverage  
  
//-----  
// Class: sw_sub_block  
//  
// Sub_block for the stopwatch design  
//-----  
  
class sw_sub_block extends uvm_reg_block;  
    `uvm_object_utils(sw_sub_block)  
  
    rand stopwatch_value stopwatch_value_reg; // Value instance  
    rand stopwatch_reset_value stopwatch_reset_value_reg; // Reset Value  
instance  
    rand stopwatch_upper_limit stopwatch_upper_limit_reg; // Upper Limit  
instance  
    rand stopwatch_lower_limit stopwatch_lower_limit_reg; // Lower Limit  
instance  
    rand stopwatch_csr stopwatch_csr_reg; // CSR instance  
    rand stopwatch_memory stopwatch_memory_reg[8]; // MEM instances  
  
    uvm_reg_map SUB_MAP1; // SW sub block map 1  
    sw_sub_block_SUB_MAP1_coverage SUB_MAP1_cg;  
  
    // Function: new  
    //  
    function new(string name = "sw_sub_block");  
        super.new(name, build_coverage(UVM_CVR_ALL));  
    endfunction  
  
    // Function: build  
    //
```

```

        virtual function void build();

            if(has_coverage(UVM_CVR_ADDR_MAP)) begin
                SUB_MAP1_cg =
sw_sub_block SUB_MAP1_coverage::type_id::create("SUB_MAP1_cg");
                SUB_MAP1_cg.ra_cov.set_inst_name(this.get_full_name());
                void'(set_coverage(UVM_CVR_ADDR_MAP));
            end
            stopwatch_value_reg =
stopwatch_value::type_id::create("stopwatch_value_reg");
            stopwatch_value_reg.configure(this);
            stopwatch_value_reg.build();

            stopwatch_reset_value_reg =
stopwatch_reset_value::type_id::create("stopwatch_reset_value_reg");
            stopwatch_reset_value_reg.configure(this);
            stopwatch_reset_value_reg.build();

            stopwatch_upper_limit_reg =
stopwatch_upper_limit::type_id::create("stopwatch_upper_limit_reg");
            stopwatch_upper_limit_reg.configure(this);
            stopwatch_upper_limit_reg.build();

            stopwatch_lower_limit_reg =
stopwatch_lower_limit::type_id::create("stopwatch_lower_limit_reg");
            stopwatch_lower_limit_reg.configure(this);
            stopwatch_lower_limit_reg.build();

            stopwatch_csr_reg =
stopwatch_csr::type_id::create("stopwatch_csr_reg");
            stopwatch_csr_reg.configure(this);
            stopwatch_csr_reg.build();

            foreach ( stopwatch_memory_reg[i] ) begin
                stopwatch_memory_reg[i] =
stopwatch_memory::type_id::create($sprintf("stopwatch_memory_reg[%0d]",
i));
                stopwatch_memory_reg[i].configure(this);
                stopwatch_memory_reg[i].build();
            end

            SUB_MAP1 = create_map("SUB_MAP1", 'h0, 4, UVM_BIG_ENDIAN, 0);
            default_map = SUB_MAP1;

            SUB_MAP1.add_reg(stopwatch_value_reg, 'h0, "RW");
            SUB_MAP1.add_reg(stopwatch_reset_value_reg, 'h1, "RW");
            SUB_MAP1.add_reg(stopwatch_upper_limit_reg, 'h2, "RW");
            SUB_MAP1.add_reg(stopwatch_lower_limit_reg, 'h3, "RW");
            SUB_MAP1.add_reg(stopwatch_csr_reg, 'h4, "RW");
            foreach(stopwatch_memory_reg[i]) begin
                SUB_MAP1.add_reg(stopwatch_memory_reg[i], (i * ('h1)) + ('h5),
"RW");
            end

            lock_model();
        endfunction

        // Function: sample

```



```

        //
        function void sample(uvm_reg_addr_t offset, bit is_read, uvm_reg_map
map);
            if(get_coverage(UVM_CVR_ADDR_MAP)) begin
                if(map.get_name() == "SUB_MAP1") begin
                    SUB_MAP1_cg.sample(offset, is_read);
                end
            end
        endfunction: sample

    endclass

//-----
// Class: sw_top_block_TOP_MAP_coverage
//
// Coverage for the 'TOP_MAP' in 'sw_top_block'
//-----

class sw_top_block_TOP_MAP_coverage extends uvm_object;
    `uvm_object_utils(sw_top_block_TOP_MAP_coverage)

    covergroup ra_cov(string name) with function sample(uvm_reg_addr_t
addr, bit is_read);

        option.per_instance = 1;
        option.name = name;

        ADDR: coverpoint addr {
            bins vreg1 = {'h3000};
            bins vreg2 = {'h3001};
            bins vreg3 = {'h3002};
            bins my_reg1 = {'h3003};
        }

        RW: coverpoint is_read {
            bins RD = {1};
            bins WR = {0};
        }

        ACCESS: cross ADDR, RW;

    endgroup: ra_cov

    function new(string name = "sw_top_block_TOP_MAP_coverage");
        ra_cov = new(name);
    endfunction: new

    function void sample(uvm_reg_addr_t offset, bit is_read);
        ra_cov.sample(offset, is_read);
    endfunction: sample

endclass: sw_top_block_TOP_MAP_coverage

//-----
// Class: sw_top_block
//
// Top block for the stopwatch design
//-----

```

```
class sw_top_block extends uvm_reg_block;
  `uvm_object_utils(sw_top_block)

  rand sw_sub_block sw1[5]; // Block instance 1
  rand sw_sub_block sw2; // Block instance 2
  rand stopwatch_counter vreg1; // Counter instance 1
  rand stopwatch_counter vreg2; // Counter instance 2
  rand stopwatch_counter vreg3; // Counter instance 3
  rand my_reg my_reg1; // Custom register instance
  my_mem mem1; // Memory instance

  uvm_reg_map TOP_MAP; // SW top block map
  sw_top_block_TOP_MAP_coverage TOP_MAP_cg;

  // Function: new
  //
  function new(string name = "sw_top_block");
    super.new(name, build_coverage(UVM_CVR_ALL));
  endfunction

  // Function: build
  //
  virtual function void build();

    add_hdl_path("top.dut");

    if (has_coverage(UVM_CVR_ADDR_MAP)) begin
      TOP_MAP_cg =
sw_top_block TOP_MAP_coverage::type_id::create("TOP_MAP_cg");
      TOP_MAP_cg.ra_cov.set_inst_name(this.get_full_name());
      void'(set_coverage(UVM_CVR_ADDR_MAP));
    end
    foreach ( sw1[i] ) begin
      sw1[i] = sw_sub_block::type_id::create($psprintf("sw1[%0d]",
i));
      sw1[i].configure(this);
      sw1[i].build();
    end

    sw2 = sw_sub_block::type_id::create("sw2");
    sw2.configure(this);
    sw2.build();

    vreg1 = stopwatch_counter::type_id::create("vreg1");
    vreg1.configure(this, null, "count_FF1");
    vreg1.build();

    vreg2 = stopwatch_counter::type_id::create("vreg2");
    vreg2.configure(this, null, "count_FF2");
    vreg2.build();

    vreg3 = stopwatch_counter::type_id::create("vreg3");
    vreg3.configure(this, null, "count_FF3");
    vreg3.build();
```

```
my_reg1 = my_reg::type_id::create("my_reg1");
my_reg1.configure(this);
my_reg1.build();

mem1 = my_mem::type_id::create("mem1");
mem1.configure(this);

TOP_MAP = create_map("TOP_MAP", 'h0, 4, UVM_BIG_ENDIAN, 0);
default_map = TOP_MAP;

foreach(sw1[i]) begin
    TOP_MAP.add_submap(sw1[i].SUB_MAP1, (i * ('hd)) + ('h1000));
end
TOP_MAP.add_submap(sw2.SUB_MAP1, 'h2000);
TOP_MAP.add_reg(vreg1, 'h3000, "RW");
TOP_MAP.add_reg(vreg2, 'h3001, "RW");
TOP_MAP.add_reg(vreg3, 'h3002, "RW");
TOP_MAP.add_reg(my_reg1, 'h3003, "RW");
TOP_MAP.add_mem(mem1, 'h3004, "RW");

    lock_model();
endfunction

// Function: sample
//
function void sample(uvm_reg_addr_t offset, bit is_read, uvm_reg_map
map);
    if(get_coverage(UVM_CVR_ADDR_MAP)) begin
        if(map.get_name() == "TOP_MAP") begin
            TOP_MAP_cg.sample(offset, is_read);
        end
    end
endfunction: sample

endclass

endpackage
```



# Third-Party Information

Details on open source and third-party software that may be included with this product are available in the *<your\_software\_installation\_location>/legal* directory.



# **End-User License Agreement with EDA Software Supplemental Terms**

Use of software (including any updates) and/or hardware is subject to the End-User License Agreement together with the Mentor Graphics EDA Software Supplement Terms. You can view and print a copy of this agreement at:

[mentor.com/eula](http://mentor.com/eula)

