SystemRDL 2.0
Register Description Language

January 2018

Abstract: Information about the registers in a circuit design is required throughout its lifetime, from initial architectural specification, through creation of an HDL description, verification of the design, post-silicon testing, to deployment of the circuit. A consistent and accurate description of the registers is necessary so the registers specified by the architects and the registers programmed by the users of the final product are the same. SystemRDL is a language for describing registers in circuit designs. SystemRDL descriptions are used as inputs to software tools that generate circuit logic, test programs, printed documentation, and other register artifacts. Generating all of these from a single source ensures their consistency and accuracy. The description of a register may correspond to a register in an preexisting circuit design, or it can serve as an input to a synthesis tool that creates the register logic and access interfaces. A description captures the behavior of the individual registers, the organization of the registers into register files, and the allocation of addresses to registers. A variety of register behaviors can be described: simple storage elements, storage elements with special read/write behavior (e.g., ‘write 1 to clear’), interrupts, and counters.
Keywords: hardware design, electronic design automation, SystemRDL, hierarchical register description, control and status registers, interrupt registers, counter registers, register synthesis, software generation, documentation generation, bus interface, memory, register addressing.

Notices

Accellera Systems Initiative (Accellera) Standards documents are developed within Accellera and the Technical Committee of Accellera. Accellera develops its standards through a consensus development process, approved by its members and board of directors, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are members of Accellera and serve without compensation. While Accellera administers the process and establishes rules to promote fairness in the consensus development process, Accellera does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an Accellera Standard is wholly voluntary. Accellera disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other Accellera Standard document.

Accellera does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or suitability for a specific purpose, or that the use of the material contained herein is free from patent infringement. Accellera Standards documents are supplied "**AS IS**."

The existence of an Accellera Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of an Accellera Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change due to developments in the state of the art and comments received from users of the standard. Every Accellera Standard is subjected to review periodically for revision and update. Users are cautioned to check to determine that they have the latest edition of any Accellera Standard.

In publishing and making this document available, Accellera is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is Accellera undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other Accellera Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of Accellera, Accellera will initiate action to prepare appropriate responses. Since Accellera Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, Accellera and the members of its Technical Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of Accellera Standards are welcome from any interested party, regardless of membership affiliation with Accellera. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Accellera Systems Initiative.
8698 Elk Grove Bldv Suite 1, #114
Elk Grove, CA 95624
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. Accellera shall not

be responsible for identifying patents for which a license may be required by an Accellera standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Accellera is the sole entity that may authorize the use of Accellera-owned certification marks and/or trademarks to indicate compliance with the materials set forth herein.

Authorization to photocopy portions of any individual standard for internal or personal use must be granted by Accellera, provided that permission is obtained from and any required fee is paid to Accellera. To arrange for authorization please contact Lynn Garibaldi, Accellera Systems Initiative, 8698 Elk Grove Bldv Suite 1, #114, Elk Grove, CA 95624, phone (916) 670-1056, e-mail lynn@accellera.org. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained from Accellera.

Suggestions for improvements to the SystemRDL 2.0 Specification are welcome. They should be sent to the SystemRDL email reflector

systemrdl@lists.accellera.org

Introduction

The SystemRDL language was specifically designed to describe and implement a wide variety of registers and memories. Using SystemRDL, developers can automatically generate and synchronize the register specification in hardware design, software development, verification, and documentation. The intent behind standardizing the language is to drastically reduce the development cycle for hardware designers, hardware verification engineers, software developers, and documentation developers.

SystemRDL is intended for

- RTL generation
- RTL verification
- SystemC generation
- Documentation
- Pass through material for other tools, e.g., debuggers
- Software development

Participants

The following members took part in the SystemRDL Working Group (RDWG):

Miles McCoo, Intel Corporation, *Chair RDWG*
Joe Daniels, *Technical Editor*

Allied Member: Michael Faust

Cisco Systems, Inc: Steve Russell, Somasundaram Arunachalam

Intel Corporation: Miles McCoo

Magillem Design Services: Guillaume Godet-Bar

NVIDIA Corporation: John Berendsen

Semifore, Inc: Richard Weber

Contents

1. Overview ... 1

1.1 Scope ... 1
1.2 Purpose ... 1
1.3 Motivation ... 1
1.4 Backward compatibility ... 2
1.5 Conventions used ... 2
1.5.1 Visual cues (meta-syntax) ... 2
1.5.2 Notational conventions ... 3
1.5.3 Examples ... 3
1.6 Use of color in this standard ... 3
1.7 Contents of this standard ... 3

2. References ... 5

3. Definitions, acronyms, and abbreviations ... 7

3.1 Definitions ... 7
3.2 Acronyms and abbreviations ... 8

4. Lexical conventions ... 9

4.1 White space ... 9
4.2 Comments ... 9
4.3 Identifiers ... 9
4.4 Keywords ... 10
4.5 Strings ... 10
4.6 Numbers ... 11

5. General concepts, rules, and properties ... 13

5.1 Key concepts and general rules ... 13
5.1.1 Defining components ... 13
5.1.2 Instantiating components ... 16
5.1.3 Specifying component properties ... 21
5.1.4 Scoping and namespaces ... 23
5.2 General component properties ... 25
5.2.1 Universal properties ... 25
5.2.2 Structural properties ... 26
5.3 Content deprecation ... 27
5.3.1 Semantics ... 27
5.3.2 Examples ... 27

6. Data types ... 29

6.1 Overview ... 29
6.2 Primary data types ... 29
6.2.1 Signed and unsigned data types ... 29
6.2.2 String data type ... 30
6.2.3 Boolean data type ... 30
6.2.4 Reserved enumeration types ... 30

6.2.5 Enumerations ... 30
6.2.6 Identifier references ... 32
6.3 Aggregate data types ... 33
6.3.1 Arrays ... 33
6.3.2 Structures ... 35
6.4 Type compatibility ... 37
6.5 Casting ... 37

7. Expressions ... 39

7.1 Overview ... 39
7.2 Operators ... 39
7.2.1 Assignment operators ... 40
7.2.2 Logical operators ... 40
7.3 Expression evaluation rules ... 40
7.3.1 Rules for determining expression types ... 40
7.3.2 Rules for evaluating expressions ... 41

8. Signals ... 43

8.1 Introduction ... 43
8.2 Signal properties ... 43
8.2.1 Semantics ... 43
8.2.2 Example ... 43
8.3 Signal definition and instantiation ... 44
8.3.1 Semantics ... 44
8.3.2 Example ... 44

9. Field component ... 45

9.1 Introduction ... 45
9.2 Defining and instantiating fields ... 45
9.3 Using field instances ... 45
9.4 Field access properties ... 46
9.4.1 Semantics ... 47
9.4.2 Example ... 48
9.5 Hardware signal properties ... 48
9.5.1 Semantics ... 48
9.5.2 Example ... 48
9.6 Software access properties ... 49
9.6.1 Semantics ... 50
9.6.2 Examples ... 51
9.7 Hardware access properties ... 51
9.7.1 Semantics ... 52
9.7.2 Example ... 53
9.8 Counter properties ... 53
9.8.1 Counter incrementing and decrementing ... 53
9.8.2 Counter saturation and threshold ... 54
9.9 Interrupt properties ... 57
9.9.1 Semantics ... 61
9.9.2 Example ... 61
9.10 Miscellaneous field properties ... 61
9.10.1 Semantics ... 62
9.10.2 Example ... 62

10. Register component 63

10.1 Defining and instantiating registers 63
10.2 Instantiating registers 63
10.3 Instantiating internal registers 64
10.4 Instantiating external registers 64
10.5 Instantiating alias registers 65
10.5.1 Semantics 65
10.5.2 Example 65
10.6 Register properties 66
10.6.1 Semantics 66
10.6.2 Example 66
10.7 Understanding field ordering in registers 67
10.7.1 Semantics 67
10.7.2 Examples 67
10.8 Understanding interrupt registers 68
10.8.1 Semantics 68
10.8.2 Example 68

11. Memory component 69

11.1 Defining and instantiating memories 69
11.2 Semantics 69
11.3 Memory properties 70
11.3.1 Semantics 70
11.3.2 Example 70

12. Register file component 71

12.1 Defining and instantiating register files 71
12.2 Semantics 72
12.3 Register file properties 72
12.3.1 Semantics 72
12.3.2 Example 73

13. Address map component 75

13.1 Introduction 75
13.2 Defining and instantiating address maps 75
13.3 Semantics 75
13.4 Address map properties 75
13.4.1 Semantics 76
13.4.2 Example 77
13.5 Defining bridges or multiple view address maps 77
13.5.1 Semantics 77
13.5.2 Example 77

14. Verification constructs 79

14.1 HDL path 79
14.1.1 Assigning HDL path 79
14.1.2 Examples 80
14.2 Constraints 81
14.2.1 Describing constraints 81

14.2.2 Constraint component 82
14.2.3 Example 83

15. User-defined properties 85

15.1 Defining user-defined properties 85
15.1.1 Semantics 86
15.1.2 Example 86
15.2 Assigning (and binding) user-defined properties 86
15.2.1 Semantics 86
15.2.2 Examples 87

16. Preprocessor directives 89

16.1 Embedded Perl preprocessing 89
16.1.1 Semantics 89
16.1.2 Example 89
16.2 Verilog-style preprocessor 89
16.2.1 Verilog-style preprocessor directives 90
16.2.2 Limitations on nested file inclusion 90

17. Advanced topics in SystemRDL 91

17.1 Application of signals for reset 91
17.2 Understanding hierarchical interrupts in SystemRDL 93
17.2.1 Example structure and perspective 94
17.2.2 Code snippet 1 95
17.2.3 Code snippet 2 95
17.2.4 Code snippet 3 96
17.2.5 Code snippet 4 96
17.2.6 Code snippet 5 97
17.2.7 Code snippet 6 98
17.2.8 Code snippet 7 98
17.2.9 Code snippet 8 99
17.2.10 Code snippet 9 100
17.2.11 Code snippet 10 101
17.2.12 Code snippet 11 102
17.3 Understanding bit ordering and byte ordering in SystemRDL 102
17.3.1 Bit ordering 103
17.3.2 Byte ordering 104

Annex A (informative) Bibliography 105

Annex B (normative) Grammar 107

Annex C (informative) Backward compatibility 113

Annex D (normative) Reserved words 117

Annex E (normative) Access modes 119

Annex F (informative) Formatting text strings 127

Annex G (informative) Component-property relationships 131

SystemRDL 2.0: A Register Description Language Specification

1. Overview

This clause explains the scope and purpose of this standard, describes the key features, details the conventions used, and summarizes its contents.

The formal syntax of SystemRDL is described using Backus-Naur Form (BNF), which is summarized in Annex B. The rest of this Standard is intended to be consistent with the SystemRDL grammar. If any discrepancies between the two occur, the grammar in Annex B shall take precedence.

1.1 Scope

SystemRDL is a language for the design and delivery of intellectual property (IP) products used in designs. SystemRDL semantics supports the entire life-cycle of registers from specification, model generation, and design verification to maintenance and documentation. Registers are not just limited to traditional configuration registers, but can also refer to register arrays and memories.

The intent of this standard is to define SystemRDL accurately. Its primary audience are implementers of tools supporting the language and users of the language. The focus is on defining the valid language constructs, their meanings and implications for the hardware and software that is specified or configured, how compliant tools are required to behave, and how to use the language.

1.2 Purpose

SystemRDL is designed to increase productivity, quality, and reuse during the design and development of complex digital systems. It can be used to share IP within and between groups, companies, and consortiums. This is accomplished by specifying a single source for the register description from which all views can be automatically generated, which ensures consistency between multiple views. A *view* is any output generated from the SystemRDL description, e.g., RTL code or documentation.

1.3 Motivation

SystemRDL was created to minimize problems encountered in describing and managing registers. Typically in a traditional environment the system architect or hardware designer creates a functional specification of the registers in a design. This functional specification is most often text and lacks any formal syntactic or

semantic rules. This specification is then used by other members of the team including software, hardware, and design verification. Each of these parties uses the specification to create representations of the data in the languages which they use in their aspect of the chip development process. These languages typically include Verilog, VHDL, C, C++, Vera, ***e***, and SystemVerilog. Once the engineering team has an implementation in a HDL and some structures for design verification, then design verification and software development can begin.

During these verification and validation processes, bugs are often encountered which require the original register specification to change. When these changes occur, all the downstream views of this data have to be updated accordingly. This process is typically repeated numerous times during chip development. In addition to the normal debug cycle, there are two additional aspects that can cause changes to the register specification. First, marketing requirements can change, which require changes to a register's specification. Second, physical aspects, such as area and timing constraints can drive changes to the register's specification. There are clearly a number of challenges with this approach:

a) The same information is being replicated in many locations by many individuals.
b) Propagating the changes to downstream customers is tedious, time-consuming, and error-prone.
c) Documentation updates are often postponed until late in the development cycle due to pressures to complete other more critical engineering items at hand.

These challenges often result in a low-quality product and wasted time due to having incompatible register views. SystemRDL was designed to eliminate these problems by defining a rich language that can formally describe register specifications. Through application of SystemRDL and a SystemRDL compiler, users can save time and eliminate errors by using a single source of specification and automatically generating any needed downstream views.

1.4 Backward compatibility

One of the main goals for this update to the SystemRDL specification was to maintain backward compatibility to SystemRDL 1.0. In some cases, however, this was not possible. Annex C shows the known areas of incompatibility in advancing the SystemRDL specification from the SystemRDL1.0 to SystemRDL 2.0 versions.

1.5 Conventions used

The conventions used throughout the document are included here.

1.5.1 Visual cues (meta-syntax)

The meta-syntax for the description of the syntax rules uses the conventions shown in Table 1.

Table 1—Document conventions

Visual cue	Represents
bold	The **bold** font is used to indicate key terms, text that shall be typed exactly as it appears. For example, in the following property definition, the keyword "default" and special character ":" (and optionally "=") shall be typed as they appear: **default** *property_name* [= *value*]**;**
italic	The *italic* font in running text represents user-defined variables. For example, a property name needs to be specified in the following line (after the "default" key term): **default** *property_name* [= *value*]**;**

Table 1—Document conventions (Continued)

Visual cue	Represents
`courier`	The `courier` font in running text indicates SystemRDL or HDL code. For example, the following line indicates SystemRDL code: `field myField {}; // defines a field type named "myField"`
plain text	The normal or plain text font in the BNF indicates syntactic categories (see Annex B).
[ ] square brackets	Square brackets indicate optional parameters. For example, the value assignment is optional in the following line: **default** *property_name* [= *value*]**;**
{ } curly braces	Curly braces ({ }) indicate items that can be repeated zero or more times. For example, the following shows one or more universal properties can be specified for this command: *mnemonic_name* = *value* [{{*universal_property***;**}***}**];**
* asterisk	An asterisk (*) signifies that parameter can be repeated. For example, the following line means multiple properties can be specified for this command: **field {**[*property***;**]***} *name* = *value***;**
| separator bar	The separator bar (|) character indicates alternative choices. For example, the following line shows the "in" or "out" key terms are possible values for the "-direction" parameter: **-direction <in** | **out>**

1.5.2 Notational conventions

The terms "required", "shall", "shall not", "should", "should not", "recommended", "may", and "optional" in this document are to be interpreted as described in the IETF Best Practices Document 14, RFC 2119.[1]

1.5.3 Examples

Any examples shown in this Standard are for information only and are only intended to illustrate the use of SystemRDL.

1.6 Use of color in this standard

This standard uses a minimal amount of color to enhance readability. The coloring is not essential and does not affect the accuracy of this standard when viewed in pure black and white. The places where color is used are the following:

— Cross references that are hyperlinked to other portions of this standard are shown in underlined-blue text (hyperlinking works when this standard is viewed interactively as a PDF file).

— Syntactic keywords and tokens in the formal language definitions are shown in **boldface-red text** when initially defined.

1.7 Contents of this standard

The organization of the remainder of this standard is as follows:

[1]Information on references can be found in Clause 2.

- Clause 2 provides references to other applicable standards that are assumed or required for this standard.
- Clause 3 defines terms and acronyms used throughout the different specifications contained in this standard.
- Clause 4 defines the lexical conventions used in SystemRDL.
- Clause 5 highlights the general concepts, rules, and properties in SystemRDL.
- Clause 6 defines the SystemRDL data types.
- Clause 7 describes how expressions are used in SystemRDL.
- Clause 8 describes how signals are used in SystemRDL.
- Clause 9 defines the field components.
- Clause 10 defines the register components.
- Clause 11 defines the memory components.
- Clause 12 defines the register file components.
- Clause 13 defines the address map components.
- Clause 14 specifies the verification constructs.
- Clause 15 specifies the user-defined properties.
- Clause 16 defines the preprocessor directives.
- Clause 17 describes advanced uses of SystemRDL.
- Annexes. Following Clause 17 are a series of annexes.

2. References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEEE Std 1364™, IEEE Standard for Verilog Hardware Description Language.[2, 3]

IEEE Std 1685™, IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows.

IEEE Std 1800™, IEEE Standard for SystemVerilog Unified Hardware Design, Specification and Verification Language.

IEEE Std 1800.2™, IEEE Standard for Universal Verification Methodology Language Reference Manual.

IETF Best Practices Document 14, RFC 2119.

The Apache ASP Embedding Syntax is available from the Apache web site:
http://www.apache-asp.org/syntax.html.

The HTML 4.01 standard syntax is available from the W3 web site:
http://www.w3.org/TR/html401/.

The MD5 Message-Digest Algorithm is available from the IETF web site:
https://tools.ietf.org/html/rfc1321.

The Perl programming language, Version 5, is available from the Perl web site:
http://www.perl.org/.

The Unicode Standard, Version 9.0.0, is available from The Unicode Consortium web site:
http://www.unicode.org/versions/Unicode9.0.0/.

[2]The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[3]IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).

3. Definitions, acronyms, and abbreviations

For the purposes of this document, the following terms and definitions apply. *The Authoritative Dictionary of IEEE Standards Terms* [B1][4] should be referenced for terms not defined in this clause.

3.1 Definitions

address map: Defines the organization of the **register**s, **register file**s, memories, and address maps into a software addressable space. Address maps can be organized hierarchically.

byte order: The ordering of the bytes from left to right or right to left or from most significant byte to least significant byte or least significant byte to most significant byte. This is often referred to as *endianness*. See also Clause 17.

bit order: The ordering of the bits from left to right or right to left or from most significant bit to least significant bit or least significant bit to most significant bit. See also Clause 17.

component: A basic building block in SystemRDL that acts as a container for information. Similar to a *struct* or *class* in programming languages.

constraint: An assertion made for verification purposes that is evaluated at the runtime of the design.

element: An instantiation of any SystemRDL **component** type.

enumeration: Used in field encodings and component property encodings. An identifier bound to some bit value or a list of values describing bit field encoding or component property encoding.

field: The most basic **component** object. Fields serve as an abstraction of hardware storage elements.

keyword: A predefined, non-escaped identifier (see 4.3) that defines a language construct; keywords cannot be used as identifiers.

memory: A contiguous array of memory data elements. A data structure within a memory can be specified with virtual registers or register files.

parameter: A generalized value of a **component** definition that can be modified for each instance of the component.

property: A characteristic, attribute, or a trait of a **component** in SystemRDL. Because they exist in their own namespace, property names do not conflict with the language and are not restricted as identifiers.

RDLFormatCode: A set of formatting tags which can be used on text strings.

register: A set of one or more **field**s which are accessible by software at a particular address.

register file: A grouping of **register**s and other register files. Register files can be organized hierarchically.

reserved words: terms which have a similar effect to **keyword**s; all reserved words are explicitly reserved for future use.

[4]The number in brackets correspond to those of the bibliography in Annex A.

signal: A wire used for interconnect or to define additional **component** inputs and/or outputs.

struct: User-defined structure for use in user-defined properties. See also Clause 15.

3.2 Acronyms and abbreviations

HDL hardware description language

HTML hypertext markup language

IP intellectual property

LSB least significant bit

MSB most significant bit

RTL register transfer level

4. Lexical conventions

This clause describes SystemRDL in terms of lexical conventions. SystemRDL source code is comprised of a stream of lexical tokens consisting of one or more characters. SystemRDL files shall use the Universal Coded Character Set, UCS, encoded using UTF-8. UCS code points beyond the ASCII code page are restricted to comments and character strings. SystemRDL is case-sensitive. SystemRDL identifiers are limited to ASCII letters, numbers, and the underscore (_). The support for UTF-8 is limited to strings to allow for non-English documentation. SystemRDL compilers shall ignore the byte-order mark.

4.1 White space

White space characters are: *space* (U+0020), *horizontal tab* (U+0009), *line feed* (U+000A), and *carriage return* (U+000D). All white space characters are syntactically insignificant, except in the following cases.

a) Strings—Any number of consecutive white space characters is treated as a single space for purposes of generating documentation. See 4.5.

b) Single-line comments—A new-line character (*line feed*, *carriage return,* or *line feed* plus *carriage return*) terminates a single-line comment. See 4.2.

c) Where more than one token is being used and spacing is required to separate the tokens.

4.2 Comments

There are two types of comments in SystemRDL: single-line comments and block comments. *Single-line comments* begin with // and are terminated by a new-line character. *Block comments* begin with /* and are terminated by the next */. Block comments may span any number of lines; they shall not be nested. Within a block comment, a single-line comment (//) has no significance.

Examples

```
// single line comment
/*
Block
comment
       // This is part of this Block comment
*/
```

4.3 Identifiers

An *identifier* assigns a name to a user-defined data type or its instance. There are two types of identifiers: simple and escaped. Identifiers are case-sensitive. *Simple identifiers* have a first character that is a letter or underscore (_) followed by zero or more letters, digits, and underscores. *Escaped identifiers* begin with \ followed by a simple identifier.

Examples

```
my_identifier
My_IdEnTiFiEr
x
_y0123
_3
\field // This is escaped because it uses a keyword
```

4.4 Keywords

Keywords are predefined, non-escaped identifiers (see 4.3) that define language constructs. Keywords cannot be used as identifiers. Escaped keywords are treated as identifiers in SystemRDL. The keywords are listed in Table 2.

Table 2—SystemRDL keywords

abstract	**accesstype**	**addressingtype**	**addrmap**	**alias**
all	**bit**	**boolean**	**bothedge**	**compact**
component	**componentwidth**	**constraint**	**default**	**encode**
enum	**external**	**false**	**field**	**fullalign**
hw	**inside**	**internal**	**level**	**longint**
mem	**na**	**negedge**	**nonsticky**	**number**
onreadtype	**onwritetype**	**posedge**	**property**	**r**
rclr	**ref**	**reg**	**regalign**	**regfile**
rset	**ruser**	**rw**	**rw1**	**signal**
string	**struct**	**sw**	**this**	**true**
type	**unsigned**	**w**	**w1**	**wclr**
woclr	**woset**	**wot**	**wr**	**wset**
wuser	**wzc**	**wzs**	**wzt**	

The following also apply.

— *Reserved words* have a similar effect as *keywords*; reserved words are explicitly reserved for future use. See also Annex D.

— The SystemRDL-detailed access modes are defined in Annex E.

— Right-hand side values defined in this standard are *keywords*. See also Annex G.

— Left-hand side values that are not keywords are *properties*. See also Annex G.

— Because they exist in their own namespace, *property names* do not conflict with the language and are not restricted as identifiers.

4.5 Strings

A *string* is a sequence of characters enclosed by double quotes. The escape sequence \" can be used to include a double quote within a string. To maintain consistency between all generated documentation formats, one or more consecutive white space characters within a string shall be converted to a single space for purposes of documentation generation. SystemRDL also has a set of formatting tags which can be used on text strings, see Annex F.

Examples

```
"This is a string"
"This is also
a string!"
"This third string contains a \"double quote\""
```

4.6 Numbers

There are several number formats in SystemRDL. All numbers in SystemRDL are unsigned.

a) **Simple decimal**: A sequence of decimal digits **0**, ..., **9**.

b) **Simple hexadecimal**: **0x** (or **0X**) followed by a sequence of decimal digits or characters **a** through **f** (upper- or lower-case).

c) **Verilog-style decimal**: Begins with a *width* specifying the number of binary bits (a positive decimal number) followed by a single quote (**'**), followed by a **d** or **D** for decimal, and then the number itself, represented as a sequence of digits **0** through **9**.

d) **Verilog-style hexadecimal**: Begins with a *width* specifying the number of binary bits (a positive decimal number) followed by a single quote (**'**), followed by an **h** or **H** for hexadecimal), and then the number itself, represented as a sequence of digits **0** through **9** or characters **a** through **f** (upper- or lower-case).

e) **Verilog-style binary**: Begins with a *width* specifying the number of binary bits (a positive decimal number) followed by a single quote (**'**), followed by a **b** or **B** for binary, and then the number itself, represented as a sequence of the digits **0** and **1**.

The numeric portion of any number may contain multiple underscores (_) at any position, except the *width* and first position, which are ignored in the computation of the associated numeric value. Additionally the *width* of a Verilog number needs to be specified. Ambiguous *width* Verilog-style numbers, e.g., `'hFF`, are not supported.

It shall be an error if the value of a Verilog-style number does not fit within the specified bit-width.

Examples

```
40         // Simple decimal example
0x45       // Simple hexadecimal example
4'd1       // Verilog style decimal example (4 bits)
3'b101     // Verilog style binary example (3 bits)
32'hDE_AD_BE_EF // Verilog style with _'s
32'hdeadbeef // Same as above
7'h7f      // Verilog style hex example (7 bits)
```

5. General concepts, rules, and properties

The concepts, rules, and properties described in this clause are common to all component types and do not determine how a component is implemented in a design.

5.1 Key concepts and general rules

This subclause describes the key concepts of SystemRDL and documents general rules about how to use the language to define hardware specifications. Subsequent clauses will refine these generic rules for each component type.

A *component* in SystemRDL is the basic building block or a container which contains properties that further describe the component's behavior. There are several structural components in SystemRDL: **field**, **reg**, **mem**, **regfile**, and **addrmap**. Additionally, there are several non-structural components: **signal**, **enum**, and **constraint**.

Components can be defined in any order, as long as each component is defined before it is instantiated. All structural components (and signals) need to be instantiated before being generated.

5.1.1 Defining components

To define components in SystemRDL, each *definition statement* shall begin with the keyword corresponding to the component object being defined (as listed in Table 3). All components need to be defined before they can be instantiated (see 5.1.2).

Table 3—Component types

Type	Keyword
Field	**field**
Register	**reg**
Register file	**regfile**
Address map	**addrmap**
Signal	**signal**
Enumeration	**enum**
Memory	**mem**
Constraint	**constraint**

SystemRDL components can be defined in two ways: definitively or anonymously.

— *Definitive* defines a named component type, which is instantiated in a separate statement. The definitive definition is suitable for reuse.

— *Anonymous* defines an unnamed component type, which is instantiated in the same statement (see also 5.1.2). The anonymous definition is suitable for components that are used once.

A *definitive definition* of a component appears as follows.

component new_component_name [**#(***parameter_definition* [**,** *parameter_definition*]***)**]
{[*component_body*]**}** [*instance_element* [**,** *instance_element*]***];**

An *anonymous definition* (and instantiation) of a component appears as follows.

component {[*component_body*]} *instance_element* [, *instance_element*]*;

a) In both cases, *component* is one of the keywords specified in Table 3.

b) For a definitively defined component, *new_component_name* is the user-specified name for the component.

c) For a definitively defined component, *parameter_definition* is the user-specified parameter as defined in 5.1.1.1.

d) For a anonymously defined component, *instance_element* is the description of the instantiation attributes, as defined in 5.1.2 a 3.

e) The *component_body* is comprised of zero or more of the following.

1) Default property assignments
2) Property assignments
3) Component instantiations
4) Nested component definitions
5) Constraint definitions
6) Struct definitions

f) The first instance name of an anonymous definition is also used as the component type name.

g) The stride (+=), alignment (**%**), and offset (**@**) of anonymous instances are the same as the definitive instances in 5.1.2.3.

The following code fragment shows a simple definitive **field** component definition for `myField`.

```
field myField {};
```

The following code fragment shows a simple anonymous **field** component definition for `myField`.

```
field {} myField;
```

5.1.1.1 Defining component parameters

All definitive component types, except enumerations and constraints, may be parametrized using Verilog-style parameters. To define Verilog-style parameters in SystemRDL, parameter definitions shall be specified after the component's name. *parameter_definition* is defined as follows.

parameter_type parameter_name [= *parameter_value*]

where

a) *parameter_type* is a type reference taken from the list of SystemRDL types (see Table 7).

b) *parameter_name* is a user-specified parameter name.

c) *parameter_value* is an expression whose resolved type should be consistent with *parameter_type*.

5.1.1.2 Semantics

a) If a parameter definition is assigned a parameter value, that value is the default value for the parameter.

b) If a parameter is not specified with a default value, every instance of the component needs to provide a value for the parameter.

c) The name of the parameter may be used elsewhere within the remainder of the component definition to represent its value.

d) Nested component definitions do not inherit from their parent's parameters.

e) Component instance references shall not be used as parameter values (either directly or as part of an aggregate type).

5.1.1.3 Inserting parameterized types in the type namespace

Each declared component's type name is added to the type namespace of the enclosing scope of component declaration. In addition, instances of parameterized components which have parameter overrides create a new type name based on the parameterized component type name rules in the type namespace of the component declaration enclosing scope. Subsequent instances of parameterized components with the same resolved parameter values matching those of a component instance existing type name in the type namespace of component declaration enclosing shall reuse the existing type name without adding a new type name.

It shall be an error if a parameterized component instance has a type name which matches an existing type name that corresponds to a parameterized component instance with different resolved parameter values or matches any other type name.

5.1.1.4 Generated type naming rules

Most generation targets for elaborated SystemRDL platforms require some means of uniquely identifying instance types. To provide a minimum level of compatibility between tool outputs, defining the type name generation process is necessary.

The following steps shall be used to construct the elaborated type names of instance with parameter arguments.

a) If the instance's defined arguments match the type's default parameter values, the instance's type name shall be used as is.

b) If the instance's type is parameterized and all the defined arguments match the type's default parameter values, the instance's type name shall be used as is.

c) In all other cases, the instance's generated type name shall be constructed by appending to the instance's type name and, for each argument its name, followed by its normalized value, separated by a single underscore (_). The sequences shall also be joined using single underscores.

type_name{(_*param_name*_*normalized_value*)}*

Normalized values shall be derived from the argument's type and from its resolved expression's value as follows.

1) Scalar values shall be rendered using their hexadecimal representation.

2) Boolean values shall be rendered using either `t` for **true** or `f` for **false**.

3) String values shall be rendered using the first eight characters of their md5 (*Message-Digest Algorithm*) `checksum`.

4) Enum values shall be rendered using their enumerator literal.

5) Arrays shall be rendered by:

i) generating the normalized values of its elements,

ii) joining these elements with single underscores (_) into a single character sequence, and

iii) using the first eight characters of the md5 `checksum` of this character sequence

… which can be semi-formalized as:

subsequence(md5(join(normalized_values, '_'), 0, 8)

6) Structs shall be rendered by:

i) generating the normalized value of each member,

ii) joining each member's name with its normalized value, separated by a single underscore (_),

iii) joining the member character sequences with single underscores,

iv) using the first eight characters of the md5 `checksum` of this character sequence

… which can be semi-formalized as:

member_normalization = concat(member_name, '_', normalized_member_value)

subsequence(md5(join(apply(struct_members, member_normalization)), 0, 8)

5.1.2 Instantiating components

In a similar fashion to defining components, SystemRDL components can be instantiated in two ways.

a) A *definitively defined component* is instantiated in a separate statement, as follows.

type_name [**#(***parameter_instance* [**,** *parameter_instance*]*)] *instance_element* [**,** *instance_element*]* **;**

where

1) *type_name* is the user-specified name for the component.

2) *parameter_instance* is specified as

.*param_name*(*param_val*)

where *param_name* is the name of the parameter defined with the component and *param_val* is an expression whose result is the value of the parameter for this instance.

3) *instance_element* is specified as follows.

instance_name [{**[***constant_expression***]**}* | **[***constant_expression* **:** *constant_expression***]**] [*addr_alloc*]

i) *instance_name* is the user-specified name for instantiation of the component.

ii) *constant_expression* is an expression that resolves to a `longint unsigned`.

iii) **[***constant_expression***]** specifies the size of the instantiated component array (optionally multidimensional) if the component is an **addrmap**, a **regfile**, a **reg**, or a **mem**; or the instantiated component's bit width if the component is a **field** or a **signal**.

iv) **[***constant_expression* **:** *constant_expression***]** specifies the bit boundaries of the instantiated component. This form of instantiation can only be used for **field** or **signal** components (see Clause 10 and Clause 8).

v) *addr_alloc* is an address allocation operator (see 5.1.2.3). These operators shall only be used when instantiating **addrmap**, **regfile**, **reg**, or **mem** components.

vi) When using multiple-dimensions, the last subscript increments the fastest.

b) An *anonymously defined component* is instantiated in the statement that defines it (see also 5.1.1).

Components need to be defined before they can be instantiated. In some cases, the order of instantiation impacts the structural implementation, e.g., for the assigning of bit positions of fields in registers (see Clause 6 — Clause 15).

The following code fragment shows a simple scalar **field** component instantiation.

```
field {} myField; // single bit field instance named "myField"
```

The following code fragment shows a simple array **field** component instantiation.

```
field {} myField[8]; // 8 bit field instance named "myField"
```

5.1.2.1 Specifying instance parameters

SystemRDL components defined with parameters (see 5.1.1.1) may have those parameters overridden or defined during non-anonymous instantiation.

Parameters are assigned by name, which explicitly links the parameter name and its new value. The name of the parameter shall be the name specified in the instantiated component. It is not necessary to assign values to all of the parameters within a component, only parameters that are assigned new values need to be specified. Parameter values are assigned using Verilog-style syntax, as defined in 5.1.2 a 2.

5.1.2.1.1 Parameter instance example

```
reg myReg #(longint unsigned SIZE = 32, boolean SHARED = true) {
    regwidth = SIZE;
    shared = SHARED;
    field {} data[SIZE - 1];
};
addrmap myAmap {
    myReg reg32;
    myReg reg32_arr[8];
    myReg #(.SIZE(16)) reg16;
    myReg #(.SIZE(8), .SHARED(false)) reg8;
};
```

5.1.2.1.2 Parameter dependence

a) A parameter (e.g., `memory_size`) can be defined with an expression containing another parameter (e.g., `word_size`).

b) Overriding a parameter effectively replaces the parameter definition with the new expression.

c) Parameters are evaluated following the order in which they are defined in the component definition. Because `memory_size` depends on the value of `word_size`, a modification of `word_size` changes the value of `memory_size`.

   For example, in the following parameter declaration, an update of `word_size` in an instantiation statement for the component that defined these parameters automatically updates `memory_size`. If `memory_size` is defined in an instantiation statement, however, it will take on that value, regardless of the value of `word_size`.

```
mem fixed_mem #(longint unsigned word_size = 32,
    longint unsigned memory_size = word_size * 4096) {
  mementries = memory_size / word_size ;
  memwidth = word_size ;
} ;
```

5.1.2.2 Instance address allocation

The offset of an instance within an object is always relative to its parent object. If an instance is not explicitly assigned an address allocation operator (see Table 4), the compiler assigns the address according to the **alignment** (see 5.1.2.2.1) and *addressing mode* (see 5.1.2.2.2). The address of an instance from the top level `addrmap` is calculated by adding the instance offset and the offset of all its parent objects.

5.1.2.2.1 Instance alignment

The **alignment** property defines the byte value of which the container's instance addresses shall be a multiple. This property can be set for **addrmap**s (see Table 26) and **regfile**s (see Table 25), and its value

shall be a power of two (1, 2, 4, etc.). Its value is inherited by all of the container's non-addrmap children. By default, instantiated objects shall be aligned to a multiple of their width (e.g., the address of a 64-bit register is aligned to the next 8-byte boundary).

5.1.2.2.2 Addressing modes

There are three *addressing modes* defined in SystemRDL: **compact**, **regalign** (the default), and **fullalign**. These addressing modes are set using the **addressing** address map property (see Table 26).

a) **compact**

Specifies the components are packed tightly together while still being aligned to the **accesswidth** parameter (see Table 23).

Example 1

Sets accesswidth=32.

```
addrmap some_map {
  default accesswidth=32;
  addressing=compact;

  reg { field {} a; } a;               // Address 0
  reg { regwidth=64; field {} a; } b; // Address 4
  reg { field {} a; } c[20];           // Address 0xC - Element 0
                                      // Address 0x10 - Element 1
                                       // Address 0x14 - Element 2
};
```

Example 2

Sets accesswidth=64.

```
addrmap some_map {
  default accesswidth=64;
  addressing=compact;

  reg { field {} a; } a;               // Address 0
  reg { regwidth=64; field {} a; } b; // Address 8
  reg { field {} a; } c[20];           // Address 0x10 - Element 0
                                      // Address 0x14 - Element 1
                                       // Address 0x18 - Element 2
};
```

b) **regalign**

Specifies the components are packed so each component's start address is a multiple of its size (in bytes). Array elements are aligned according to the individual element's size (this results in no gaps between the array elements). This generally results in simpler address decode logic.

Example 3

Uses the default accesswidth of 32.

```
addrmap some_map {
  addressing = regalign;

  reg { field {} a; } a;               // Address 0
  reg { regwidth=64; field {} a; } b; // Address 8
```

```
  reg { field {} a; } c[20];          // Address 0x10
                                     // Address 0x14 - Element 1
                                     // Address 0x18 - Element 2
};
```

c) **fullalign**

The assigning of addresses is similar **regalign**, except for arrays. The alignment value for the first element in an array is the size in bytes of the whole array (i.e., the size of an array element multiplied by the number of elements), rounded up to nearest power of two. The second and subsequent elements are aligned according to their individual size (so there are no gaps between the array elements).

Example 4

Uses the default `accesswidth` of 32.

```
addrmap some_map {
  addressing = fullalign;

  reg { field {} a; } a;              // Address 0
  reg { regwidth=64; field {} a; } b; // Address 8
  reg { field {} a; } c[20];          // Address 0x80 - Element 0
                                     // Address 0x84 - Element 1
                                     // Address 0x88 - Element 2
};
```

5.1.2.3 Address allocation operators

When instantiating **reg**s, **regfile**s, **mem**s, or **addrmap**s, the address may be assigned using one of the address allocation (`addr_alloc`) operators in Table 4.

Table 4—Address allocation operators

Property	Implementation/Application
@ *expression*	Specifies the address for the component instance. This expression resolves to a `longint unsigned`.
+= *expression*	Specifies the address stride when instantiating an array of components (controls the spacing of the components). The address stride is relative to the previous instance's address. This expression resolves to a `longint unsigned`.
%= *expression*	Specifies the alignment of the next address when instantiating a component (controls the alignment of the components). The initial address alignment is relative to the previous instance's address. This expression resolves to a `longint unsigned`.

5.1.2.4 Semantics

a) Addresses in SystemRDL are always byte addresses.

b) Addresses are assigned in incrementing order.

c) The operator **%=** is a more localized version of the **alignment** property (see Table 25).

d) The expression used for address specification shall be resolvable to a `longint unsigned`.

e) The += operator is only used when instantiating arrayed **addrmap**, **regfile**, **reg**, or **mem** components.

f) The **@** and **%=** operators are mutually exclusive per instance.

g) The alignment of an array instance specifies the alignment of the start of the array and the increment specifies the offset from one array element to the next array element.

5.1.2.5 Examples

The following set of examples demonstrate the usage of the operators defined in Table 4. The final addresses (as indicated in the comments in the example) are valid for an addressing mode called **regalign**, which is the default addressing mode (see Clause 13), with the default **regwidth**=32. The **regfile** component is defined in Clause 12.

Example 1

Using the **@** operator.

```
addrmap top {
    regfile example {
        reg some_reg { field {} a; };
        some_reg  a   @0x0;
        some_reg  b   @0x4;
        some_reg  c;  // Implies address of 8
                      // Address 0xC is not implemented or specified
        some_reg  d   @0x10;
    };
};
```

Example 2

Using the **+=** operator.

```
addrmap top {
    regfile example {
        reg some_reg { field {} a; };
        some_reg a[10]; // So these will consume 40 bytes
                        // Address 0,4,8,C....
        some_reg b[10] @0x100 += 0x10; // These consume 160-12 bytes of space
                                  // Address 0x100 to 0x103, 0x110 to 0x113,....
    };
};
```

Example 3

Using the **%=** operator.

```
addrmap top {
    regfile example {
        reg some_reg { field {} a; };
        some_reg a[10]; // So these will consume 40 bytes
                        // Address 0,4,8,C....
        some_reg b[10] @0x100 += 0x10; // These consume 160-12 bytes of space
                                  // Address 0x100 to 0x103, 0x110 to 0x113,....
        some_reg c  %=0x80; // This means ((address % 0x80) == 0))
                        // So this would imply an address of 0x200 since
                        // that is the first address satisfying address>=0x194
                        // and ((address % 0x80) == 0)
```

```
    };
};
```

5.1.3 Specifying component properties

Each property is associated with at least one data type defined in Clause 6 (and summarized in Table 7). Property types include primitive types and aggregate types.

5.1.3.1 Property assignment

Each component type has its own set of pre-defined properties. Properties may be assigned in any order. User-defined properties can also be specified to add additional properties to a component that are not pre-defined by the SystemRDL specification (see Clause 15). A specific property shall only be set once per scope (see 5.1.4). All component property assignments are optional.

A *property assignment* appears as follows.

property_name [= *expression*];

The descriptions for the data types of *expression* results that are legal for each *property_name* (and exceptions to those rules) are explained in the corresponding clause for each individual component (see Clause 8 — Clause 14).

When *expression* is not specified, it is presumed the *property_name* is of type **boolean** and the default value is set to **true**.

Example

```
field myField {
  rclr;               // Bool property assign, set implicitly to true
  woset = false;      // Bool property assign, set explicitly to false
  name = "my field"; // string property assignment
  sw = rw;           // accesstype property assignment
};
```

5.1.3.2 Assigning default values

Default values for a given property can be set within the current or any enclosing lexical scope (see 5.1.4). Any components defined in the same or enclosed lexical scope as the default property assignment shall use the default values for properties in the component not explicitly assigned in a component definition. A specific property **default** value shall only be set once per scope.

A *default property assignment* appears as follows.

default *property_name* [= *value*];

The descriptions for the types of *value*s that are legal for each *property_name* (and exceptions to those rules) are explained in the corresponding clause for each individual component (see Clause 8 — Clause 14).

When the *value* is not specified, the property shall be assigned the boolean value **true**.

Example

```
field {} outer_field ;
reg {
  default name = "default name";
```

```
  field {} f1; // assumes the name "default name" from above
  field { name = "new name"; } f2; // name assignment overrides "default name"
  outer_field f3 ; // name is undefined, since outer_field is not defined in the
                   // scope of the default name
} some_reg;
```

5.1.3.3 Dynamic assignment

Some properties may have their values assigned or overridden on a per-instance basis. When a property is assigned after the component is instantiated, the assignment itself is referred to as a *dynamic assignment*. Properties of a referenced instance shall be accessed via the arrow operator (**->**).

A *dynamic assignment* appears as follows.

instance_name **->** *property_name* [**=** *value*]**;**

where

a) *instance_name* is a previously instantiated component (see 5.1.2).

b) When *value* is not specified, it is presumed the *property_name* is of type **boolean** and the value is set to **true**.

c) The dynamically assignable properties for each component type are explained in the corresponding clause for each individual component (see Clause 8 — Clause 14).

d) In the case where *instance_name* is an array, the following possible dynamic assignment scenarios exist.

1) If the component type is **field** or **signal**, the fact the component is an array does not matter—the assignment is treated as if the component were a not an array.

2) If the component type is **reg**, **regfile**, **mem**, or **addrmap**

i) The user can dynamically assign the property for all elements of the array by eliminating the square brackets (**[]**) and the array index from the dynamic assignment.

array_instance_name **->** *property_name* [**=** *value*]**;**

ii) The user can dynamically assign the property for an individual index of the array by using square brackets (**[]**) and specifying the index to be assigned within the square brackets.

array_instance_name {**[***index***]**}* **->** *property_name* [**=** *value*]**;**

Example 1

This example assigns a simple scalar.

```
reg {
  field {} f1;
  f1->name = "New name for Field 1";
} some_reg;
```

Example 2

This example assigns an array.

```
reg {
  field {} f1;
  f1->name = "New name for Field 1";
} some_reg[8];
some_reg->name = "This value is applied to all elements in the array";
some_reg[3]->name = "Only applied to the 4th item in the array of 8";
```

5.1.3.4 Property assignment precedence

There are several ways to set values on properties. The precedence for resolving them is (from highest to lowest priority):

a) dynamic assignment (see 5.1.3.3)
b) property assignment (see 5.1.3.1)
c) default property assignment (see 5.1.3.2)
d) SystemRDL default value for property type (see Table 7)

Example

```
reg {
  default name =”def name”;
  field f_type { name = “other name”; };
  field {} f1;
  field { name = “property assigned name”; } f2;
  f_type f3;

  f3->name = “Dynamic Assignment”;
} some_reg;
```

Results

```
// Final Values of all fields
// f1 name is “def name”
// f2 name is “property assigned name”
// f3 name is “dynamic assignment”
```

5.1.4 Scoping and namespaces

A *scope* defines the conditions in which an identifier may be associated with an entity. SystemRDL is a statically (or lexically) scoped language.

The body of a component (or **struct**) definition defines a *local scope*. A valid SystemRDL description is, therefore, an aggregation of nested local scopes, ultimately nested into the outermost *global* (or *root*) *scope*.

Each local scope contains two independent namespaces, to which different scoping rules apply:

— Type names (component definitions, **enum** types, and **struct** types);
— Element (e.g., `reg` and `field` instantiations; `struct` members) names and parameter names.

Identifiers shall be unique within a namespace in a scope. *Namespaces* are differentiated implicitly by syntax. There are no namespace operators or limiters.

The root scope contains a third namespace for property names. All property references (standard and user-defined) shall be resolved by searching this namespace.

Example 1

```
property foo {
component = field ;
type = string ;
} ;
reg foo {
```

```
    field {
      foo = "abc" ;
    } foo ;
} ;
foo foo ; // instantiate reg type foo to generate instance called foo
foo.foo -> foo = "xyz" ; // property foo of field foo of reg foo gets value "xyz"
```

The root scope shall only contain component type and **struct** type definitions and `signal` instantiations. No other component instantiations shall be allowed in the root scope. The root(s) of an **addrmap** hierarchy are those **addrmap**s that are defined, but not subsequently instantiated.

By definition, a *component scope* contains component type and **struct** type definitions, as well as element references. A *struct scope* only contains member declarations. All type names shall be unique in the type namespace and all element names shall be unique within the element namespace. However, there can be a type and element with the same name in the same scope. Additionally, types shall be defined and elements declared before they are referenced in the sequence of statements.

Type references are resolved from the local scope up the enclosing lexical scope to the global scope.

a) Elements referenced in the left-hand side of an expression shall be declared in the local scope.

b) Elements referenced in the right-hand side of an expression shall be declared in the local scope or up in the enclosing lexical scope if the referenced element is a **signal**.

c) If two types (or elements) in different scopes share the same name, the type (respectively, element) name from the scope that is lexically closest to the local scope shall take precedence.

Children elements—as elements contained in the local scope of the parent scope's type—may be referenced via the dot operator (**.**).

A *element reference* appears as follows.

element_name [**.** *child_element_name*]*

where

a) *element_name* is a previously declared element in the current scope (see 5.1.2).

b) the first use of *child_element_name* shall exist in *element_name*'s local type scope.

c) for all other *child_element_name*s, any subsequent *child_element_name* shall exist in the previous *child_element_name*'s local type scope.

Element references from an assignment located in a **constraint** body are resolved from the **constraint** body's enclosing lexical scope, then up the lexical scope. Such an element reference may either be a direct **field** reference, or use the dot operator (**.**) to navigate down the referenced element's instance hierarchy to target a `field` instance.

Example 2

```
regfile foo {
   reg {
      field {} a ;
      constraint {
         a < 0xc ; // direct field reference
      } const1 ;
   } regA ;
      constraint {
         regA.a > 0x4 ; // indirect field reference
      } const2 ; } ;
```

Dynamic assignments can be layered in SystemRDL from the innermost to the outermost scope; i.e., dynamic assignments that are specified at an outer scope override those that are specified at an inner scope. No more than one assignment of a property per scope is allowed in SystemRDL.

Example 3

```
regfile foo_rf {
   reg some_reg_r {
      field {} a[2]=2'b00;// End of field: a
      a->reset = 2'b01;// Dynamic Assignment overriding reset val
      field {} b[23:16]=8'hFF; // End of field: b
   };

   some_reg_r rega;
   some_reg_r regb;

   rega.a->reset = 2'b10; // This overrides the other dynamic assign
   rega.b->reset = 8'h00;
   rega.b->reset = 8'h5C; // Error two assigns from the same scope
};                                                          // End addrmap: foo
addrmap bar {
   foo_rf foo;
   foo.rega.a->reset = 2'b11;
   // Override the reset value again at the outermost scope
};                                                          // End addrmap: bar
```

Any reference to an element in the right-hand side of an assignment shall be resolved statically, i.e., by considering the elements visible from the assignment's local scope.

Example 4

```
signal {} my_signal ;
field my_field {
  resetsignal = my_signal ; // will resolve to the signal instance
                            // declared in the global scope
} ;
addrmap top {
  signal {} my_signal ;
  reg {
    my_field a ;  // the field instance's resetsignal will
                  // still be resolved as the global scope's my_signal
  } reg_a ;
} ;
```

5.2 General component properties

This subclause details properties that generally apply to SystemRDL components.

5.2.1 Universal properties

The **name** and **desc** properties can be used to add descriptive information to the SystemRDL code. The use of these properties encourages creating descriptions that help generate rich documentation. All components have a instance name already specified in SystemRDL; **name** can provide a more descriptive name and **desc** can specify detailed documentation for that component.

Table 5 lists and describes the universal SystemRDL component properties.

Table 5—Universal component properties

Property	Implementation/Application	Type	Dynamic[a]
name	Specifies a more descriptive name (for documentation purposes).	*string*	Yes
desc	Describes the component's purpose.	*string*	Yes

[a]Indicates whether a property can be assigned dynamically.

5.2.1.1 Semantics

If **name** is undefined, it is presumed to be the instance name.

5.2.1.2 Example

This example shows usage of the **name** and **desc** properties.

```
reg {
   field {
      name="Interface Communication Control";
       // If name is not specified its implied to be ICC
      desc="This field is used [...] desired low power state.";
   } ICC[4];
} ICC_REG; // End of Reg: ICC_REG
```

5.2.2 Structural properties

Table 6 lists and describes the structural component properties.

Table 6—Structural component properties

Property	Implementation/Application	Type	Dynamic[a]
donttest	This testing property indicates the component is not included in structural testing.	*boolean* or *bit*	Yes
dontcompare	This is testing property indicates the components read data shall be discarded and not compared against expected results.	*boolean* or *bit*	Yes

[a]Indicates whether a property can be assigned dynamically.

5.2.2.1 Semantics

a) These properties can be applied as a *boolean* or a bit mask (*bit*) to a **field** component. A mask shall have the same *width* as the **field**. Masked bits (bits set to `1`) are not tested (**donttest**) or compared (**dontcompare**).

b) They can also be applied to **reg**, **regfile**, and **addrmap** components, but only as a *boolean*.

c) **donttest** and **dontcompare**

   1) cannot both be set to **true**,

   2) cannot have one **true** and the other non-zero, and

   3) the bitwise `AND` of their masks shall be zero (`0`) for a particular component (i.e., `donttest & dontcompare = 0`).

5.2.2.2 Example

This example shows usage of the **donttest** and **dontcompare** properties.

```
reg {
   field { donttest;} a;
   field {} b[8];
   field { dontcompare;} c;
   b->dontcompare = 8'hF0; // The upper four bits of this 8 bit field will
                           // not be compared.
} some_reg;
```

5.3 Content deprecation

The **ispresent** universal property can be used to configure the activation of SystemRDL component instances. Setting **ispresent** to **false** causes the given component instance to be removed from the final specification.

5.3.1 Semantics

a) **ispresent** is a universal property on all component instances (**addrmap**, **reg**, **signal**, etc.) other than **enum**s.

b) The default value of **ispresent** is **true**.

c) Instance names shall be unique within a scope even before the values of **ispresent** are resolved. This feature does not enable replacement of instances.

d) **ispresent** values may not be dependent on values contained in SystemRDL constructs. No reference values are allowed. Otherwise, the rules of expressions apply.

e) Setting **ispresent** to **false** removes the instance.

f) Setting a property on an element that is removed due to **ispresent** does not constitute an error, e.g., if an instance belong to a removed **addrmap**, modifications to the instance are acceptable.

g) Instance positions are computed presuming all instances are present. Removing an instance can introduce a hole.

h) If a component is instantiated twice, setting **ispresent** to **false** on one of them causes the hardware implementation to be removed from that instantiation.

i) If a present instance includes references (e.g., signals), the referred objects need to also be present.

j) If a present instance is an alias register (see 10.5), the primary register needs to also be present. Conversely, if a register acting as a primary register is not present, then all the alias registers that refer to it shall not be present either.

k) Component instances shall not be empty. Setting **ispresent** on all children of a parent instance to **false** shall be an error.

5.3.2 Examples

Some examples are shown highlighting simple, complex, and corner case usage.

5.3.2.1 Simple example

```
addrmap submap {
  reg { field {} a[32] ; } rega, regb, ahb_specific ;
} ;
```

```
addrmap bridge {
  bridge ;
  submap ahb ;
  submap axi ;
  axi.ahb_specific -> ispresent = false ;
} ;
```

5.3.2.2 Complex example

```
reg some_reg #(boolean RESERVED = false) {
  ispresent = !RESERVED ;
  field {} a, b, c ;
  b -> ispresent = false ;
  field { ispresent = false ; } d ;
// the default bitfield layout should be: a[0:0], c[2:2]
} ;

some_reg #(.RESERVED(true)) reserved_reg ; // entire reg not present
some_reg partially_reserved_reg ;
some_reg not_reserved_reg ; // all fields present with dynamic assigns below
not_reserved_reg.b -> ispresent = true ;
not_reserved_reg.d -> ispresent = true ;
```

5.3.2.3 Corner case

```
field {} a, b ;
b -> next = a ;
a -> ispresent = false ; // This is an error w.r.t clause (h) "If a present
                         //  instance includes references (e.g., signals), the
                          //  referred objects need to also be present."
```

6. Data types

6.1 Overview

This section presents all the data primary and aggregate data types used in SystemRDL. While some data types, such as **boolean** or **onreadtype**, are specific to SystemRDL, the data types and its associated type system are consistent with SystemVerilog semantics as specified in IEEE Std 1800-2012, unless noted otherwise.

Table 7 summarizes all the data types discussed in this document.

Table 7—Data types

Type	Parameter or struct member type name	Definition	Default
boolean	**boolean**	**true** or **false**.	**false**
string	**string**	See 4.5 and 6.2.2.	**""**
bit	**bit**	An unsigned integer with the value of `0` or a Verilog-style number, see 4.6 (c - e) and 6.2.1.	Undefined
longint unsized	**longint unsigned**	A 64-bit unsigned long integer, see 4.6 (a and b) and 6.2.1.	Undefined
accesstype	**accesstype**	One of **rw**, **wr**, **r**, **w**, **rw1**, **w1**, or **na**. See 9.4.	**rw**
addressingtype	**addressingtype**	One of **compact**, **regalign**, or **fullalign**. See 13.4.	**regalign**
onreadtype	**onreadtype**	One of **rclr**, **rset**, or **ruser**. See 9.6.	Undefined
onwritetype	**onwritetype**	One of **woset**, **woclr**, **wot**, **wzs**, **wzc**, **wzt**, **wclr**, **wset**, or **wuser**. See 9.6.	Undefined
precedencetype		One of **hw** or **sw**. Cannot be used as a parameter or struct member type. See 9.4.	**sw**
struct	*struct reference*	A reference to a struct.	Undefined
array	*array reference*	A reference to an array.	Empty array
enum	*enum reference*	A reference to a user-defined enumeration.	Undefined
instance reference	**ref**	A reference to a component instance, component instance property, parameter, or struct instance member.	Undefined

6.2 Primary data types

A subset of the SystemVerilog data types are used by the SystemRDL Expression Language, namely **bit**, **longint unsigned**, and **string** (with some changes).

Complex, user-defined, and time data types shall not be supported in SystemRDL. Unknown (`x`) and high impedance (`z`) values shall not be supported either.

6.2.1 Signed and unsigned data types

All SystemRDL number types are integral and unsigned. In order to maintain direct compatibility with the SystemRDL Expression Language, SystemRDL only supports **bit** and **longint unsigned**. Expressions

resolving into a negative value shall be cast to the two's complement of the value, e.g, the expression `1 - 2`, which occurs in a **longint unsigned** context whose bit width is `64`, is resolved as `0xFFFFFFFFFFFFFFFF`.

6.2.2 String data type

The SystemRDL Expression Language **string** data type is encoded in UTF-8.

A SystemRDL **string** can be seen as an immutable, unsized object, for which only the binary equality, concatenation, and replication operators are supported (see Table 9).

6.2.3 Boolean data type

The additional type **boolean** is introduced as a result type for logical operations, as well as for compatibility with previous SystemRDL versions. Boolean values shall be cast to the single bit values `1'b1` and `1'b0` (from `true` and `false`, respectively) for preserving sufficient compatibility with the SystemVerilog Expression Language, as defined in Clause 7.

6.2.4 Reserved enumeration types

The additional types: **accesstype**, **onreadtype**, **onwritetype**, and **addressingtype** shall be considered as reserved enumerations with no associated integral values for all purposes.

Reserved enumeration types only support binary equality operations.

6.2.5 Enumerations

An *enumerated type* encloses a set of constant named integral values into the enumeration's scope. There are no properties for the **enum** component beyond the universal properties defined in 5.2.1.

6.2.5.1 Defining enumerations

Unlike other SystemRDL components, enumerations are not instantiated and can only be defined definitively (i.e., anonymous definitions are not allowed). Enumerated types can either be assigned to a field's **encode** property (see 9.10) or their enumerators can be referenced in expressions. Enumerator references shall be prefixed with their enumerated type name and two colons (`::`), e.g., `MyEnumeration::MyValue`.

An *enum component definition* appears as follows.

enum *enum_name* **{** *encoding*; [*encoding*;]* **};**

where

a) *enum_name* is a user-defined name for the enumeration

b) *encoding* is specified as follows

mnemonic_name [= *value* [{{*universal_property*;}*}];

where

1) *mnemonic_name* is a user-defined name for a specific *value*. This name shall be unique within a given **enum**.

2) *value* shall be of an integral type.

3) All *value*s shall be unique, even if the value is automatically assigned.

4) *universal_property* is as defined in 5.2.1.

Example

This is an example of bit-field encoding.

```
enum myBitFieldEncoding {
  first_encoding_entry = 8'hab;
  second_entry = 8'hcd {
    name = "second entry";
  };
  third_entry = 8'hef {
    name = "third entry, just like others";
    desc = "this value has a special documentation";
  };
  fourth_entry = 8'b10010011;
};

field {
  encode = myBitFieldEncoding;
} a[8];
```

6.2.5.2 Automatically assigned enumerator values

When the first enumerator value is unspecified, it is assigned `0`. Other enumerator values are incremented by `1`, based on the value of the previous enumerator. Automatically assigned values cannot break the unique value constraint when automatically assigning all the values of an enumeration using **longint unsigned** values.

Examples

These are examples of automatically assigned and partially assigned enumeration definitions.

```
enum myAutoEnum { first_value ; second_value ; third_value ; } ;
// first_value = 0, second_value = 1, third_value = 2

enum myPartiallyAssignedEnum { a ; b ; c = 8'h6 ; d ; e = 8'h12 ; f ; } ;
// a = 8'h0, b = 8'h1, d = 8'h7, f = 8'h13
```

6.2.5.3 Type consistency

Enumerated types are strongly typed, therefore user-defined properties, struct members, or parameters of a given enumerated type are type-checked when used in assignments or with relational operators. In other expression contexts, enumerators are automatically cast to their integral values.

Example

The example below illustrates the use of enumerated types in operations and assignments.

```
enum FirstEnum {
  VAL1 = 3'h0 ;
  VAL2 = 3'h1 ;
  VAL3 = 3'h2 ;
} ;

enum SecondEnum {
  VAL1 = 3'h0 ;
  VAL2 = 3'h1 ;
```

```
  VAL3 = 3'h2 ;
} ;

property MyUDP { component = addrmap ; type = FirstEnum ; } ;

addrmap top {
  reg some_reg { field {} a[3] ; } ;

  addrmap {
    MyUDP = FirstEnum::VAL1 ; // Allowed
    some_reg regA ;

    regA.a -> reset = FirstEnum::VAL2 + SecondEnum::VAL3 ; // Enumerators are
   cast to their integer value and added
  } submap1 ;

  addrmap {
    reg {
      shared = longint'(FirstEnum::VAL1) == longint'(SecondEnum::VAL2) ; //
   Allowed since we're first casting the enumerators to their underlying
   integral values
      field {} b ;
    } other_shared_reg ;
  } submap2 ;
} ;
```

6.2.6 Identifier references

SystemRDL struct members, parameters, and component instances that are in the scope of a SystemRDL statement in which the expression is defined can be referenced from the expression.

In addition, the SystemRDL rules for escaped identifiers, (see 4.3) shall apply to references inside the SystemRDL Expression Language.

Hierarchical struct members and component instances are referenced using a dot delimiter (.) (see 5.1.4).

Example

```
struct inner_struct {
  string foo ;
} ;

struct my_struct {
  inner_struct inner ;
} ;

addrmap top {
  regfile some_regfile #( my_struct arg ) {
    reg {
      desc = arg.inner.foo ;
      field {} a ;
    } regA ;
  } ;

  some_regfile #( .arg( my_struct'{ inner: inner_struct'{ foo: "reg desc" } }
   ) ) regFA[2] ;
```

```
    regFA[0].regA.a -> desc = "field desc from regFA[0]" ;
    regFA[1].regA.a -> desc = "field desc from regFA[1]" ;
} ;
```

6.3 Aggregate data types

6.3.1 Arrays

A SystemRDL array describes an ordered collection of elements. Each array element shall be identified with a unique array index. Arrays may be used as **struct** members, or in **property** or parameter declarations.

a) An array shall be declared as follows:

array_type declaration **[]**

where

1) *array_type* specifies the type allowed for each array element. All the types defined in Table 7, excepting array types, may be used as array types.

   Effectively, multi-dimensional arrays are not supported. This limitation may be circumvented by defining arrays of structs containing arrays.

2) *declaration* may be a **struct** member or a parameter name.

For example:

```
reg some_reg #( string NAME_AND_DESC[] ) {
  field {} a ;
} ;
```

b) A user-defined property array shall be declared as follows:

array_type **[]**

where

array_type specifies the type allowed for each array element. All the types defined in Table 31, excepting array type (type `[]`), may be used as user-defined property array types.

For example:

```
property myUDP { component = field ; type = longint unsigned[] ; } ;
```

c) An array may be assigned a sequence of values as follows:

left_hand_side = '{ [expr [, expr]*]? }

where

1) *left_hand_side* corresponds to the **struct** member, parameter, or **property** to which the array is being assigned.

2) *expr* is an expression whose resolved type shall be assignment compatible with the type of the array (see 6.4).

For example:

```
some_reg #(.NAME_AND_DESC( '{ "hello", "world" } ) regA ;
```

d) An empty array may be declared as follows:

left_hand_side = '{}

e) Array elements may be used in expressions by referencing their position in the array, as follows:

array_reference **[***index***]**

where

1) *array_reference* is a reference to the array containing the array element.

2) *index* is an expression that shall resolve to a **longint unsigned**.

For example:

```
regA -> name = NAME_AND_DESC[0] ;
```

6.3.1.1 Semantics

a) Array indices are 0-based and strictly sequential.

b) Arrays are immutable and can only be modified by recreating an array (i.e., single values cannot be reassigned).

c) SystemRDL arrays are not constrained with respect to their sizes: a given array may be reassigned with a new array of a different size.

d) An array element cannot reference another element from the same array.

e) An out of bound array reference shall raise an error.

6.3.1.2 Examples

6.3.1.2.1 User-defined property with array type

```
property MyUDP { component = reg ;
                 type = longint unsigned[] ;
                 default = '{1, 2} ; } ;

reg some_reg {
  MyUDP = '{ 2, 34, 73 } ;
} ;
```

6.3.1.2.2 User-defined property with aggregate type array type

```
struct mystruct { string foo; longint unsigned bar ; } ;
property MyUDP { component = all ;
                 type = mystruct[] ; } ;

reg some_reg {
  MyUDP = '{ mystruct' { foo: "hello", bar: 23 },
             mystruct'{ foo: "world", bar: 42 } } ;
} ;
```

6.3.1.2.3 User-defined property with enum type array type

```
enum Location { Mem = 0, PCI = 1, DMA = 2 } ;
property MyUDP { component = reg ; type = Location[] ; } ;

reg some_reg {
  MyUDP = '{ Location::Mem, Location::Mem, Location::PCI } ;
} ;
```

6.3.1.2.4 Struct defining an array type member

```
struct mystruct { string[] foo } ;
property StructUDP { component = all ; type = mystruct ; } ;


reg other_reg {
  StructUDP = 'mystruct { foo: '{ "hello", "world"} } ;
} ;
```

6.3.1.2.5 Array element reference

```
field some_field #( string NAME_AND_DESC[] ) {
  name = NAME_AND_DESC[0] ;
  desc = NAME_AND_DESC[1] ;
} ;
```

6.3.2 Structures

S*tructs* enable the creation of structured properties for more complex extension of component types.

6.3.2.1 Defining structures

6.3.2.1.1 struct definition

A **struct** definition appears as follows.

[**abstract**] **struct** *struct_name* [**:** *base_struct_name*]
{{*member_type member_name*;}*};

where

a) **abstract** optionally defines the **struct** as an **abstract struct**.

b) *struct_name* specifies the new struct type name.

c) *base_struct_name* specifies optional inheritance or derivation.

d) *member_type* is the type of the composed value.

e) *member_name* is the name of the value. Member names shall be unique within a **struct** and its base class, recursively.

6.3.2.1.2 Semantics for defined structs

a) A **struct** can be used within user-defined property definitions, parameters, arrays, and other structs.

b) The name of the struct is added to the type name namespace. Struct type names shall be unique.

c) Structs may include all of the types defined in Table 7.

d) Structs may not include items that directly or indirectly refer to the **struct** being defined (i.e., no circular dependencies).

e) A **struct** may be declared as **abstract**, which specifies that it cannot be directly instantiated. Struct types derived from an **abstract struct** are not abstract, unless specified explicitly using the **abstract** keyword.

6.3.2.2 Deriving structures

6.3.2.2.1 struct derivation

A **struct** declaration may *derive* from another **struct** by specifying the base **struct**'s name after a colon (**:**), e.g.,

```
struct base_struct {
  bit foo ;
} ;

struct derived_struct : base_struct {
  longint unsigned bar ;
} ;
```

```
struct final_struct : derived_struct {
   // final_struct's members are foo, bar, and baz.
  string baz ;
} ;
```

6.3.2.2.2 Semantics for derived structs

a) A *derived* **struct** inherits all its base's members, recursively.

b) Any member declared in the *derived* **struct** shall be unique, relative to both the *derived* **struct** and its base, recursively.

c) Parameters and user-defined properties declaring a **struct** type may be initialized using any *derived*, non-abstract, **struct** instance in their assignment's right-hand side (i.e., derived types are considered as *assignment compatible* with all their base types, following the definition from 6.4). Derived **struct** instances passed in this way shall preserve all their member values (for code generation purposes), even though only the members from the declared **struct** type shall be visible from the SystemRDL code.

6.3.2.3 Defining struct literals

6.3.2.3.1 struct literal definition

A **struct** literal is defined as follows:

struct_name **'{** [*member_name* **:** *member_value* {**,** *member_name* **:** *member_value*}*] **}**

where

a) *struct_name* is the name of the **struct** literal that is being defined.

b) *member_name* is the name of a member as specified in the **struct**'s definition.

c) *member_value* is the value being assigned.

6.3.2.3.2 Semantics for instantiated structs

a) Struct assignments are always by value.

b) When defining struct member values, unassigned members shall receive a default value depending on their type, when available, as defined in Table 7.

c) All the members from a struct instance shall be assigned a value, either explicitly or by default. Undefined struct members shall raise an error.

6.3.2.4 Examples

Example 1

This example defines a simple **struct** and uses it in a user-defined property.

```
struct struct1 {
   bool abool;
   string astring;
};
property p1 {
   component = field;
   type = struct1;
   default = struct1'{abool:true, astring:"hello"};
};
```

Example 2

This example defines a **struct** that declares a member which is also **struct**.

```
struct struct_composed {
   struct1 s;
   string str;
};
field my_field #(struct_composed PARAM) {} ;

my_field #(.PARAM( struct_composed'{ str:"world",
                                     s: struct1'{ abool:true,
                                                  astring:"blah"}}
                  )
          ) f1 ;
```

Example 3

This example defines and derives an **abstract struct**.

```
abstract struct absstruct {
   string astring;
};
struct substruct:absstruct {
   bool abool;
};
property p3 {
   component = field;
   type = absstruct ;
   default =substruct'{abool:false, astring:"foo"};
};
```

6.4 Type compatibility

As SystemRDL uses only a subset of the data types defined in the SystemVerilog, only three levels of type compatibility shall effectively be used when resolving SystemRDL expressions: *matching*, *assignment compatible,* and *incompatible*. All three levels match their SystemVerilog equivalent. Type coercion, as happens in the context of assignments (i.e., between assignment compatible types), is detailed in 6.5.

In the context of assignments, if the left hand-side expects a given **abstract struct** type, all *derived* **struct** types shall be considered as compatible.

6.5 Casting

SystemRDL only supports *static* (i.e., type-based) and *constant expression* (i.e., bit length-based) casts from SystemVerilog. The additional types introduced in SystemRDL are bound by the casting rules in Table 8.

Supported static types are: **boolean**, **bit**, **longint unsigned**, **string**, **accesstype**, **addressingtype**, **onreadtype**, and **onwritetype**. Table 8 defines which expression types are compatible with static type casts (*x* corresponds to a conversion that is *assignment compatible* — and, thus, also *cast compatible*).

Table 8—Allowed cast operations (cast and assignment compatible types)

Type	boolean	bit	longint unsigned	string	access type	addressing type	onread type	onwrite type
boolean	*x*	*x*	*x*					
bit	*x*	*x*	*x*					
longint unsigned	*x*	*x*	*x*					
string				*x*				
accesstype					*x*			
addressing type						*x*		
onreadtype							*x*	
onwritetype								*x*

Static cast operations shall be resolved according to the following rules.

a) All types can be cast to themselves.

b) When casting `boolean` to `bit` or `longint unsigned`, `true` shall be converted to `1'b1` and `false` to `1'b0`.

c) When casting a `bit` or `longint unsigned`, if the bit width of the target type does not match, this results in the upper bit zero-extension or truncation of the most significant bits.

d) When casting `bit` or `longint unsigned` to `boolean`, zero (`0`) shall be converted to `false`, any other value shall be converted to `true`.

7. Expressions

7.1 Overview

The SystemRDL Expression Language is based on the SystemVerilog Expression Language as specified in IEEE Std 1800-2012.

The goal of the SystemRDL Expression Language is for it to be a strict subset of SystemVerilog, i.e., the expressions defined in SystemRDL should be easily ported-to or incorporated-into a SystemVerilog file and interpreted by any SystemVerilog processor.

In order to represent and manipulate types and concepts proper to SystemRDL, the SystemVerilog Expression Language has been functionally limited and changes introduced.

7.2 Operators

Table 9 gives an overview of the SystemVerilog operators and how SystemRDL supports them (or not).

Table 9—SystemVerilog operators

Operator token	Name	Operand data type
=	Binary assignment operator	Only supported for specific cases (see 7.2.1)
+= -= /= *=	Binary arithmetic assignment operators	Assignments are not supported
%=	Binary arithmetic modulus assignment operator	Assignments are not supported
&= |= ^=	Binary bit-wise assignment operator	Assignments are not supported
>>= <<=	Binary logical shift assignment operators	Assignments are not supported
>>>= <<<=	Binary arithmetic shift assignment operators	Assignments are not supported
?:	Conditional operator	First operand: boolean, other operands: any
+ -	Unary arithmetic operator	Integral
++ --	Unary decrement/increment operators	Assignments are not supported
!	Unary logical negation operator	Integral
~	Unary bitwise negation operator	Integral
& ~& | ~| ^ ~^ ^~	Unary reduction operators	Integral
+ - * / **	Binary arithmetic operators	Integral
%	Binary modulus operator	Integral
& | ^ ~^ ^~	Binary bitwise operators	Integral
>> <<	Binary logical shift operators	Integral
>>> <<<	Binary arithmetic shift operators	Not supported
&& ||	Binary logical operators	Integral

Table 9—SystemVerilog operators (Continued)

Operator token	Name	Operand data type
< <= > >=	Binary relational operators	Integral, user-defined enums
== !=	Binary logical equality operators	Any, except structural instance references
=== !==	Binary case equality operators	Unknown or high-impedance values are not supported
==? !=?	Binary wildcard equality operators	Unknown or high-impedance values are not supported
inside	Binary set membership operator	Only used within top level of constraints
dist	Binary distribution operator	Randomization is not supported
{} {{}}	Concatenation and replication operator	Integral, string, boolean, reserved enums
{<<{}} {>>{}}	Stream operators	Not supported

Additional support considerations for SystemVerilog operators are detailed below.

7.2.1 Assignment operators

Since the SystemRDL Expression Language does not allow using variables, it only supports single value assignments for which the left-hand side is a property, a parameter (in the context of a parameter declaration), or a struct member reference (in the context of a post-property assignment). All other assignment operators are not supported.

7.2.2 Logical operators

The result of the evaluation of one of the supported SystemVerilog logical operators (i.e., `AND` (`&&`) and `OR` (`||`)) shall be one of the **boolean** values `true` or `false`.

Similarly, the unary *logical negation operator* (`!`) converts a `true` value into `false` and a `false` value into `true`.

Also, the binary *logical equality operators* (== and `!=`), aggregate types may be compared for equality by comparing the values of their individual members, recursively. Primary type members are compared by applying the default type and value equality rules.

7.3 Expression evaluation rules

Due to the data types supported by SystemRDL, the rules for determining expression types and evaluating expressions are more restrictive than those defined in IEEE Std 1800-2012, subclause 11.8.

7.3.1 Rules for determining expression types

The following rules shall be applied for determining the resulting type of an expression.

— Expression type depends only on the operands. It does not depend on the left-hand side (if any).

— All numbers and expression results are unsigned.

— The size of any self-determined operand is determined by the operand itself and independent of the remainder of the expression.
— Any expression that would result in an unknown (x) value shall instead raise an error.

7.3.2 Rules for evaluating expressions

All expressions are evaluated in a *self-determined context*, as specified in IEEE Std 1800-2012, subclause 11.6.1, which implies that the left-hand side of a property assignment is never taken into consideration when evaluating expressions.

8. Signals

8.1 Introduction

A *signal* is a non-structural component used to define and instantiate wires (as additional inputs and/or outputs). Signals create named external ports on an implementation and can connect certain internal component design properties to the external world. *Signal definitions* have the same definition and instantiation as other SystemRDL components; see 5.1. To use signals to control resets in SystemRDL, see 17.1.

8.2 Signal properties

Table 10 shows the **signal** properties.

Table 10—Signal properties

Property	Implementation/Application	Type	Dynamic[a]
signalwidth	Width of the signal.	*longint unsigned*	No
sync	Signal is synchronous to the clock of the component.	*boolean*	Yes
async	Signal is asynchronous to the clock of the component.	*boolean*	Yes
cpuif_reset	Default signal to use for resetting the software interface logic. If **cpuif_reset** is not defined, this reverts to the default reset signal. This parameter only controls the CPU interface of a generated slave.	*boolean*	Yes
field_reset	Default signal to use for resetting field implementations. If **field_reset** is not defined, this reverts to the default reset signal.	*boolean*	Yes
activelow	Signal is active low (state of `0` means `ON`).	*boolean*	Yes
activehigh	Signal is active high (state of `1` means `ON`).	*boolean*	Yes

[a]Indicates whether a property can be assigned dynamically.

8.2.1 Semantics

a) **sync** and **async** shall not be set to **true** on the same **signal**.

b) A signal that does not specify **sync** or **async** is considered **sync**.

c) **activelow** and **activehigh** shall not be set to **true** on the same **signal**.

d) A **signal** that does specify **activehigh** or **activelow** has no formal specified active state.

e) **field_reset** and **cpuif_reset** follow the rules of application as defined in 17.1.

f) **cpuif_reset** property can only be set **true** for one instantiated `signal` within a lexical scope.

g) **field_reset** property can only be set to **true** for one instantiated `signal` within a lexical scope.

8.2.2 Example

See the example in 8.3.2.

8.3 Signal definition and instantiation

In addition to the general rules for component definition and instantiation (see 5.1), the following rules also apply.

8.3.1 Semantics

a) If **signalwidth** (see 8.2) is not defined, `signal` instances may be declared as single-bit or multi-bit signals, as defined in (5.1.2).

b) If **signalwidth** is not predefined in the component definition, a **signal** type may be instantiated as any width.

c) If **signalwidth** is predefined during **signal** definition, any specified `signal` width shall match the predefined width.

8.3.2 Example

This example defines an 8-bit field and connects it to a **signal** so the reset value for this field is supplied externally.

```
addrmap foo {
    reg { field {} a[8]=0; } reg1;
    signal { signalwidth=8;} mySig[8];
    reg1.a->reset = mySig; // Instead of resetting this field to a constant
                           // we connect it to a signal to provide an
                           // External reset value
};
```

9. Field component

9.1 Introduction

The **field** component is the lowest-level structural component in SystemRDL. No other structural component can be defined within a **field** component; however, **signal**, enumeration (**enum**), and **constraint** components can be defined within a **field** component. The **field** component is also the most varied component in SystemRDL because it is an abstraction representing different types of storage element structures. *Field definitions* have the same definition and instantiation as other SystemRDL components; see 5.1.

Typically, a **field** component describes a flip-flop or wire/bus, along with the logic to set and sample its value for each instantiated field in the design. Properties specified for a field serve multiple purposes, from determining the nature of the behavior that is implied for a field to naming and describing a field. Storage elements accessed by software may contain a single entity or a number of bit-fields each with its own meaning and purpose. In SystemRDL, each entity in a software read or write is termed a *field*.

9.2 Defining and instantiating fields

Since a **field** component describes the lowest-level components within SystemRDL, it cannot contain other fields. Fields are instantiated in a register (**reg**) component (see Clause 10). Fields are defined and instantiated as described in 5.1, with the following additional semantics. See also 9.3.

a) No other types of structural components shall be defined within a **field** component.

b) Fields shall be instantiated only within a **register** component.

c) Unless bit allocation is explicitly defined, fields shall be positioned sequentially in the order they are instantiated in a register, starting with the least significant bit. **lsb0** mode defines `0` as the least significant bit, which is the default, and **msb0** defines `regwidth-1` as the least significant bit.

d) In the default mode **lsb0**, unless bit allocation is explicitly defined, fields shall be positioned sequentially in the order they are instantiated in a register, starting at bit `0` with no padding between fields. (Each subsequent field's least significant bit (LSB) shall be made equal to one (`1`) greater than the most significant bit (MSB) of the previous field.)

e) In the mode **msb0**, unless bit allocation is explicitly defined, fields shall be positioned sequentially in the order they are instantiated in a register, starting at bit `regwidth-1` with no padding between fields. (Each subsequent field's least significant bit (LSB) shall be made equal to one (`1`) less than the most significant bit (MSB) of the previous field.)

f) The exact bit position of instantiated fields in a register may be determined by the SystemRDL compiler as described in d or e ,or specified explicitly by using exact indices (see Clause 10).

g) The **msb0** and **lsb0** properties shall only be applied to an address map component (see Clause 13).

h) A field instantiation which is not followed by a specific size or index contained square brackets (`[]`) defaults to size of the field definition's **fieldwidth** parameter. If the definition is anonymous, the default **fieldwidth** is `1`.

9.3 Using field instances

Fields can be instantiated as single or multiple bits. Fields shall be instantiated in a register component and the field's bit position can be derived implicitly by a compiler or specified explicitly by a user. For the **field** component only, the field's bit position can be implicitly or explicitly specified. This notation is of the form

a) for *definitive field instantiation*

field_type [**#(***field_parameter_instance* [**,** *field_parameter_instance*]*)] *field_instance_element* [**,** *field_instance_element*]***;**

where

1) *field_type* is the user-specified name for a previous definitively defined component of type field.

2) *field_parameter_instance* is specified as

.*field_param_name***(***field_param_val***)**

where *field_param_name* is the name of the parameter defined with the field and *field_param_val* is an expression whose result is the value of the parameter for this instance (see 5.1.2 a).

b) for *anonymous field instantiation*

field {*field_body***}** *field_instance_element* [**,** *field_instance_element*]***;**

where

field_body is as described in 5.1.1, subject to limitations for a *definitive field instantiation* (see a).

c) For both field instantiation types, *field_instance_element* is defined as

field_instance_name [**[***constant_expression***]** | **[***constant_expression* **:** *constant_expression***]**] [**=** *constant_expression*]

where

i) *field_instance_name* is the user-specified name for instantiation of the component.

ii) *constant_expression* is an expression that resolves to a `longint unsigned`.

[*constant_expression***]** specifies the instantiated field's bit width.

[*constant_expression* **:** *constant_expression***]** is termed a *range* and defines the `msb` and `lsb` of the **field** within the context of the register within which it is instantiated.

= *constant_expression* specifies the field instance's reset value (see 9.5).

Examples

These are examples of the anonymous form.

```
field {} singlebitfield; // 1 bit wide, not explicit about position
field {} somefield[4]; // 4 bits wide, not explicit about position
field {} somefield2[3:0]; // a 4 bit field with explicit indices
field {} somefield3[15:8]; // an 8 bit field with explicit indices
field {} somefield4[0:31]; // a 32 bit field with explicit indices
```

How the compiler resolves bit positions for implicit fields is detailed in 10.1, which describes the register component. Single element arrays may be treated by a SystemRDL compiler as a scalar or an array.

9.4 Field access properties

The combination of field access properties specified for a **field** component determines the component's behavior. Table 11 lists the available field access properties and describes how they are implemented.

Table 11—Field access properties

Property	Behavior/Application	Type	Dynamic[a]
hw	Design's ability to sample/update a **field**.	*access type*	No
sw	Programmer's ability to read/write a **field**.	*access type*	Yes

[a]Indicates whether a property can be assigned dynamically.

9.4.1 Semantics

a) All **field**s are given full **sw** and **rw** access (read and write) by default.

b) **rw** (and **wr**) signify a field is both read and write; **r** indicates read-only; **w** indicates write-only; and **na** specifies no read/write access is allowed.

c) All hardware-writable fields shall be continuously assigned unless a write enable is specified.

d) When a **field** is writable by software and write-only by hardware (but not write-enabled), all software writes shall be lost on the next clock cycle. This shall reported as an error.

e) After a reset occurs on a **field** with **rw1** or **w1** software access, that field can only be written once by software. All subsequent software writes are then ignored until the field is reset again.

f) The standard implementation behavior is based on the combination of read and write properties shown in Table 12.

Table 12—Field behavior based on properties

Software	Hardware	Code sample	Implementation
R+W	R+W	`field f { sw = rw; hw = rw; };`	Storage element
R+W	R	`field f { sw = rw; hw = r; };`	Storage element
R+W	W	`field f { sw = rw; hw = w; };`	Storage element
R+W	-	`field f { sw = rw; hw = na; };`	Storage element
R	R+W	`field f { sw = r; hw = rw; };`	Storage element
R	R	`field f { sw = r; hw = r; };`	Wire/Bus – constant value
R	W	`field f { sw = r; hw = w; };`	Wire/Bus – hardware assigns value
R	-	`field f { sw = r; hw = na; };`	Wire/Bus – constant value
W	R+W	`field f { sw = w; hw = rw; };`	Storage element
W	R	`field f { sw = w; hw = r; };`	Storage element
W	W	`field f { sw = w; hw = w; };`	Error – meaningless
W	-	`field f { sw = w; hw = na; };`	Error – meaningless
-	R+W	`field f { sw = na; hw =rw; };`	Undefined
-	R	`field f { sw = na; hw = r; };`	Undefined
-	W	`field f { sw = na; hw = w; };`	Error – unloaded net
-	-	`field f { sw = na; hw = na; };`	Error – nonexistent net

NOTE—Any hardware-writable **field** is inherently volatile, which is important for verification and test purposes.

9.4.2 Example

See Table 12.

9.5 Hardware signal properties

While all of the hardware signal properties can be set within a **field** definition, typically they are assigned after instantiation as these properties refer to items external to the field itself. By default, the reset value of fields shall be unknown, e.g., x in Verilog. A specification can use static or dynamic reset values; however, only static reset values shall be specified during field instantiation. The *reset value*, which is considered a property in SystemRDL, shall follow an equal sign (=) after the instance name and the eventual size or MSB/LSB information.

For the syntax for specifying reset values, see 9.3.

Table 13 defines the hardware signal properties.

Table 13—Hardware signal properties

Property	Behavior/Application	Type	Dynamic[a]
next	The next value of the **field**; the D-input for flip-flops.	*reference*	Yes
reset	The reset value for the **field** when **resetsignal** is asserted.	*bit* or *reference*	Yes
resetsignal	Reference to the signal used to reset the **field** (see 17.1).	*reference*	Yes

[a]Indicates whether a property can be assigned dynamically.

9.5.1 Semantics

a) Any integral value can be used to specify the reset value of a **field**.

b) When a **field** has access properties of **sw=r** and **hw=w** without having a write enable, the existence of a reset value shall implement a storage element and the reset value only holds until the **reset** is deasserted.

c) The reset value cannot be larger than can fit in the **field** or an error shall be reported.

d) When **reset** is a reference, it shall reference another field of the same size. Upon reset, the **field** is reset to the current value of the referenced **field**.

e) **next** and **reset** cannot be self-referencing.

f) **reset** always has priority over **next** when **resetsignal** is asserted.

g) If no **reset** value given, the **field** is not reset, even if it has a **resetsignal**.

9.5.2 Example

This example shows different types of hardware signal properties set during **field** instantiations.

```
signal {} some_reset;
field { reset = 1'b1; } a;
field {} b=0;
field {} c=0;
c->resetsignal = some_reset;
field {} d=0x0;
```

```
d->next = a; // d gets the value of a. D lags a by 1 clock.
field {} e[23:21]=3'b101;
b->reset = 3'b1; // Override the default reset value of e from 101 to 001
```

9.6 Software access properties

The software access field properties provide a means of specifying commonly used software modifiers on register fields. All the software properties which are defined as *boolean* values have a default value of **false**. Some of these properties perform operations that directly effect the value of a field (**rclr**, **woset**, and **woclr**), others allow the surrounding logic to effect software operations (**swwe** and **swwel**), and still others allow software operations effecting the surrounding logic (**swmod** and **swacc**). The **onread** property enables setting values equivalent to **rclr** and **rset**, while the **onwrite** property enables setting values equivalent to **woclr** and **woset**.

Table 14 defines the software access properties and uses pseudo-code snippets to define the behaviors. The pseudo-code is of Verilog style and should be interpreted as such. The exact behavior of these properties depends upon the semantics of the HDL generated by a particular SystemRDL implementation, together with the execution environment (e.g., simulator) for that HDL.

Table 14—Software access properties

Property	Behavior/Application	Type	Dynamic[a]
rclr	Clear on read (`field = 0`).	*boolean*	Yes
rset	Set on read (field = all `1`'s).	*boolean*	Yes
onread	Read side-effect. See Table 15.	*onread-type*	Yes
woset	Write one to set (`field = field | write_data`).	*boolean*	Yes
woclr	Write one to clear (`field = field & ~write_data`).	*boolean*	Yes
onwrite	Write function. See Table 16.	*onwrite-type*	Yes
swwe	Software write-enable active high (`field = swwe ? new : current`).	*boolean* or *reference*	Yes
swwel	Software write-enable active low (`field = swwel ? current : new`).	*boolean* or *reference*	Yes
swmod	Assert when field is modified by software (written or read with a `set` or `clear` side effect).	*boolean*	Yes
swacc	Assert when field is software accessed.	*boolean*	Yes
singlepulse	The field asserts for one cycle when written `1` and then clears back to `0` on the next cycle. This creates a single-cycle pulse on the hardware interface.	*boolean*	Yes

[a]Indicates whether a property can be assigned dynamically.

Table 15—Software read side-effect onread value

onread property value	Behavior/Application
rclr	All the bits of the field are cleared on read (`field = 0`).
rset	All the bits of the field are set on read (field = all `1`'s).
ruser	The read modifies the field in a way which does not match the other defined read side-effects.

Table 16—Software write function onwrite values

onwrite property value	Behavior/Application
woset	Bitwise write one to set (`field = field | write_data`).
woclr	Bitwise write one to clear (`field = field & ~write_data`).
wot	Bitwise write one to toggle (`field = field ^ write_data`).
wzs	Bitwise write zero to set (`field = field | ~write_data`).
wzc	Bitwise write zero to clear (`field = field & write_data`).
wzt	Bitwise write zero to toggle (`field = field ~^ write_data`).
wclr	All bits of the field are cleared on write (`field = 0`).
wset	All bits of the field are set on write (`field =` all `1`'s).
wuser	The write modifies the field in a way which does not match the other defined write functions and is not a write without a write function.

9.6.1 Semantics

a) **swmod** indicates a generated output signal shall notify hardware when this field is modified by software. The precise name of the generated output signal is beyond the scope of this document. Additionally, this property may be used on the right-hand side of an assignment to another property.

NOTE—Since **rclr**, **rset**, and **onread** modify the field value with a software read transaction, the implementation of properties like **swmod** are asserted during software reads when **rclr** or **rset** are **true** or **onread** has a value.

b) **swacc** indicates a generated output signal shall notify hardware when this field is accessed by software. The precise name of the generated output signal is beyond the scope of this document. Additionally, this property may be used on the right-hand side of an assignment to another property.

c) Fields specified with software access properties in Table 14 need to consider how they effect the behavior defined in Table 12. For example, if a field is **rclr**, this results in a storage element regardless of whether or not the field is writable by software.

d) **swwe** and **swwel** have precedence over the software access property in determining its current access state, e.g., if a field is declared as **sw=rw**, has a **swwe** property, and the value is currently **false**, the effective software access property is **sw=r**.

e) **swwe** and **swwel** are mutually exclusive.

f) When specified, **rclr** resets a field to `0` and not its default value.

g) **singlepulse** fields shall be instantiated with a width of `1` and the reset value shall be specified as `0`.

h) **onread, rclr and rset** are mutually exclusive; only one can be set per field.

i) A **field** with an **onread** property shall have software read access.
j) A **field** with an **onread** value of **ruser** shall be external.
k) **onwrite, woclr**, and **woset** are mutually exclusive; only one can be set per field.
l) A **field** with an **onwrite** property shall have software write access.
m) A field with an **onwrite** value of **wuser** shall be external.

9.6.2 Examples

Example 1

This example applies software properties using implicit and explicit methods of setting the properties.

```
field {
   rclr; // Implicitly set the rclr property to true
   swwe = true; // Explicitly set the swwe property to true
} a;
```

Example 2

This example uses the **default** keyword with these software properties and then overrides them.

```
reg example2 {
   default woclr = true;  // Explicitly set default of woclr to true
   default swmod;         // Implicitly set default of swmod to true

   field {} a;             // Assumes defaults
   field {} b;             // Assumes defaults
   b->rclr=false; // Dynamic Assignment to false
   field {rclr = false; } c;// Overrides rclr default
   field {swmod = false; } d;// Overrides swmod default
   field {rclr = false; swmod = false; } e;// Overrides both defaults
   d->next = b->swmod;
      // next value of d will be field b's 1-bit software mod flag generated
      // by SystemRDL
};
```

9.7 Hardware access properties

Hardware access properties can be applied to fields to determine when hardware can update a hardware writable field (**we** and **wel**), generate input pins which allow designers to clear or set the field (**hwclr** and **hwset**) by asserting a single pin, or generate output pins which are useful for designers (**anded**, **ored**, and **xored**).

Write-enable is critical for certain software-writable fields. The clear on read feature (**rclr**, see Table 14) returns the **next** value (see 9.5) to software before clearing the field. In the case of counters, the write-enable is used to determine when a counter can be incremented.

The **hwenable** and **hwmask** properties can specify a bus showing which bits may be updated after any write-enables, hardware-clears/-sets or counter-increment has been performed. The **hwenable** and **hwmask** properties are similar to **we** and **wel**, but each has unique functionality. The **we** and **wel** act as write enables to an entire field for a single bit or multiple bits. The **hwmask** and **hwenable** are essentially write enables or write masks, but are applied on a bit basis. The priority of assignments a SystemRDL compiler should use is shown in Table 17, which depicts a flow of information from left to right showing the stages that happen when updating a field from its current value to determine its next state value.

A field's width is typically determined when it is instantiated; however, there are times when specifying a field's width up-front is critical. If specified, the **fieldwidth** property forces all instances of the field to be a specified width. If a field is instantiated without a specified width, the field shall be **fieldwidth** bits wide. It shall be an error if the field is instantiated with an explicitly specified width that differs from the **fieldwidth**.

Table 17—Assignment priority

Event stage ->	Hardware next stage ->	Field next stage ->	Register assign stage
we / **wel** / **intr edge** logic	**counter incr** / **counter decr**	SW/HW selection	wire / dff assign
counter load / **counter we** logic	**hwset** / **hwclr**		
	intr mask/en/sticky		

Table 18 defines the hardware access properties.

Table 18—Hardware access properties

Property	Behavior/Application	Type	Dynamic[a]
we	Write-enable (active high).	*boolean* or *reference*	Yes
wel	Write-enable (active low).	*boolean* or *reference*	Yes
anded	Logical `AND` of all bits in **field**.	*boolean*	Yes
ored	Logical `OR` of all bits in **field**.	*boolean*	Yes
xored	Logical `XOR` of all bits in **field**.	*boolean*	Yes
fieldwidth	Determines the width of all instances of the **field**. This number shall be a numeric. The default value of **fieldwidth** is undefined.	*longint unsigned*	No
hwclr	Hardware clear. This **field** need not be declared as hardware-writable.	*boolean* or *reference*	Yes
hwset	Hardware set. This **field** need not be declared as hardware-writable.	*boolean* or *reference*	Yes
hwenable	Determines which bits may be updated after any write enables, hardware clears/sets or counter increment has been performed. Bits that are set to 1 will be updated.	*reference*	Yes
hwmask	Determines which bits may be updated after any write enables, hardware clears/sets or counter increment has been performed. Bits that are set to 1 will not be updated.	*reference*	Yes

[a]Indicates whether a property can be assigned dynamically.

9.7.1 Semantics

a) **we** determines this field is hardware-writable when set, resulting in a generated input which enables hardware access.

b) **wel** determines this field is hardware-writable when not set, resulting in a generated input which enables hardware access.

c) **we** and **wel** are mutually exclusive.

d) **hwenable** and **hwmask** are mutually exclusive.

9.7.2 Example

This example shows the application of a write-enable and the *boolean* **anded**.

```
reg example {
    default sw = r;

    field { anded;} a[4]=0; // This field will update its value every clock
                  // cycle. hw=rw by default. This field will also have
                  // an output ANDing the 4 bits of the field together
    field { we; } b=0;// This field will only update on clock cycles
                // where the we is asserted. The name of the we signal is
                // a function of the SystemRDL Compiler.
};
```

9.8 Counter properties

SystemRDL defines several special purpose fields, including counters. A *counter* is a special purpose field which can be incremented or decremented by constants or dynamically specified values. Additionally, counters can have properties that allow them to be cleared, set, and indicate various status conditions like **overflow** and **underflow**.

9.8.1 Counter incrementing and decrementing

When a **field** is defined as a **counter**, the value stored by the field is the counter's current value. There is an implication of an additional input which shall increment/decrement the counter when asserted. Counter incrementing and decrementing in SystemRDL are controlled via the counter's **incrvalue/decrvalue** and **incrwidth/decrwidth** properties. The **incrvalue/decrvalue** property defaults to a value of `1`, but can be set to any constant that can be represented by the width of the counter. Additionally, the **incrvalue/decrvalue** can be assigned to any **signal** or other **field** in the current address map scope so counters can increment using dynamic or variable values. The **incrwidth**/**decrwidth** properties can be used as an alternative to **incrvalue**/**decrvalue** so an external interface can be used to control the **incrvalue**/**decrvalue** externally from SystemRDL. A SystemRDL compiler shall imply the nature of a counter as a up counter, a down counter, or an up/down counter by the properties specified for that counter field.

By default, counters are incremented/decremented by one (`1`), but another static or dynamic increment/decrement value can be specified. The increment/decrement value shall be equal to or smaller than the field's width.

Dynamic values may be another field instance in the address map of the same or smaller width, or another signal in the design. If an externally defined signal is used for dynamic incrementing, its input is inferred to have the same width as the counter.

Additionally, the properties **incr** and **decr** can be used to control the increment and decrement events of a counter. These do not control the increment or decrement values, as **incrvalue** and **decrvalue**, but the actual increment of the counter (as shown in *Example 2*). These properties can be only be assigned as references to another component.

Example 1

This shows counter incrementing and decrementing.

```
field counter_f { counter; };

counter_f count1[4]; // Define a 4 bit counter from 3 down to 0
  count1->incrvalue=4'3; // Increment the counter by 3 when incrementing
                         // count1 implies an UP counter

counter_f count2[3]; // Define a 3 bit counter from 6 down to 4
  count2->decrwidth=2; // provide 2 bit interface for a user to decide the decr
                     // value. This implies a down counter.
counter_f count3[5]=0; // Defines a 5 bit counter from 11 down to 7
  count3->incrvalue=2; // Define a an Up/Down Counter
  count3->decrvalue=4;

field {} count4_incr[8] = 8'h0f; // define a field to control the incr
                                    // value of another field.

counter_f count4[8]=0;
  count4->incrvalue = count4_incr; // Counter is incremented by the value of
                               // another field in the same register.
```

Example 2

This example uses **incr** to connect two 16-bit counters together to create a 32-bit counter.

```
field some_counter {
  counter;
  we;
};                                                  // End of Reg: some_counter

reg some_counter_reg {
  regwidth=16;
  some_counter count[16]=0;   // Create 16 bit counter POR to 0
};                                                      // End of Reg:

// Example 32 bit up counter
some_counter_reg      count1_low;
some_counter_reg      count1_high;

count1_high.count->incr = count1_low.count->overflow;
// Daisy chain the counters together to create a 32 bit counter from the 2
// 16 bit counters
```

9.8.2 Counter saturation and threshold

Counters are unsaturated by default, e.g., a 4-bit counter with a value of `0xf` that is incremented by `1` has the value `0x0`. This is referred to as *rolling over*. The value of a incrsaturate saturating counter shall never exceed the increment saturation value and the value of a decrsaturate saturating counter shall never be less than the decrement saturation value. By default, the increment saturation value is the maximum value that the counter can hold and the decrement saturation value is zero (`0`). Assigning a static or dynamic saturated value is similar to assigning increment/decrement values, see 9.8.1.

Counters in SystemRDL may have an optional (static or dynamic) threshold value. The threshold properties do not cap the value of a counter in the way saturate properties do; instead, threshold counters are inferred to contain an output which designates whether the counter's values meets or exceeds the threshold. See also 9.8.1.

saturate and **threshold** counters may be used individually and specified in any order.

Example 1

This shows counter saturation and thresholds.

```
field counter_f { counter; };
counter_f count1[4]; // Define a 4 bit counter from 3 down to 0
count1->incrsaturate=4'he; // keeps the counter from counting past 4'he

counter_f count2[3]; // Define a 3 bit counter from 6 down to 4
count2->decrthreshold=3'h2; // provide threshold indication when
                            // count reaches 2 or lower

counter_f count3[5]=0; // Defines a 5 bit counter from 11 down to 7
count3->incrsaturate;// Implies 5'h1F by default
count3->decrsaturate; // Implies 5'h00 by default
count3->decrthreshold=5'h3;

field {} count4_sat[4] = 4'ha; // define a field to control the saturate value
                               // of another field
field {} count4_thresh[4] =4'h2;

counter_f count4[4]=0; // This counters saturate and threshold are both dynamic
count4->incrthreshold = count4_thresh;
count4->incrsaturate = count4_sat;
```

Besides assigning values or references to the **saturate** or **threshold** properties on the left-hand side of an assignment in SystemRDL, these properties can also be referenced on the right-hand side of an expression to indicate the threshold has been crossed or the counter has saturated. This is often useful for generating an interrupt indicating a specific condition has occurred.

Example 2

This shows right-hand side usage of **saturate** and **threshold**.

```
field {} count4_sat[4] = 4'h2; // define a field to control the saturate value
                               // of another field
field {} count4_thresh[4] =4'ha;
field {} is_at_threshold=0;
field {} is_saturated=0;

counter_f count4[4]=0; // This counters saturate and threshold are both dynamic
count4->incrthreshold = count4_thresh;
count4->incrsaturate = count4_sat;

// Single-bit result of threshold comparison assigned to is_at_threshold field
is_at_threshold->next = count4->incrthreshold;
is_saturated->next = count4->incrsaturate;
```

Counters can also use the properties **underflow** and **overflow** to indicate the counter has wrapped (either decrementing when `0` for **underflow** or incrementing when all `1`s for **overflow**). These are useful for many applications such as generating an interrupt based on a counter overflow/underflow.

Example 3

This shows **overflow** and **underflow** counter properties.

```
field counter_f { counter; };
field {} has_overflowed;

counter_f count1[5]=0; // Defines a 5 bit counter from 6 down to 1
count1->incrthreshold=5'hF;

has_overflowed->next = count1->overflow;
```

Table 19 defines the counter field properties.

Table 19—Counter field properties

Property	Behavior/Application	Type	Dynamic[a]
counter	Field implemented as a counter.	*boolean*	Yes
threshold	This is an alias of **incrthreshold**.	*boolean*, *bit*, or *reference*	Yes
saturate	This is an alias of **incrsaturate**.	*boolean*, *bit*, or *reference*	Yes
incrthreshold	Indicates the counter has a threshold in the incrementing direction. A comparison value or the result of a comparison. See also: 9.8.2.1.	*boolean*, *bit*, or *reference*	Yes
incrsaturate	Indicates the counter saturates in the incrementing direction. A comparison value or the result of a comparison. See also: 9.8.2.1.	*boolean*, *bit*, or *reference*	Yes
overflow	Overflow signal asserted when counter overflows or wraps.	*boolean*	Yes
underflow	Underflow signal asserted when counter underflows or wraps.	*boolean*	Yes
incrvalue	Increment counter by specified value.	*bit* or *reference*	Yes
incr	References the **counter**'s increment signal. Use to actually increment the counter, i.e, the actual counter increment is controlled by another component or signal (active high).	*reference*	Yes
incrwidth	Width of the interface to hardware to control incrementing the counter externally.	*longint unsigned*	Yes
decrvalue	Decrement counter by specified value.	*bit or reference*	Yes
decr	References the **counter**'s decrement signal. Use to actually decrement the counter, i.e, the actual counter decrement is controlled by another component or signal (active high).	*reference*	Yes
decrwidth	Width of the interface to hardware to control decrementing the counter externally.	*longint unsigned*	Yes
decrsaturate	Indicates the counter saturates in the decrementing direction. A comparison value or the result of a comparison. See also: 9.8.2.1.	*boolean*, *bit*, or *reference*	Yes
decrthreshold	Indicates the counter has a threshold in the decrementing direction. A comparison value or the result of a comparison. See also: 9.8.2.1.	*boolean*, *bit*, or *reference*	Yes

[a]Indicates whether a property can be assigned dynamically.

9.8.2.1 Semantics

a) **incrwidth** and **incrvalue** are mutually exclusive (per **counter**).

b) **decrwidth** and **decrvalue** are mutually exclusive (per **counter**).

c) When **incrsaturate** has the Boolean value `true`, the incrementing saturate value is the *maximum value* (2^(number of counter bits) -1) of the counter. When **incrsaturate** has the Boolean value `false`, the counter does not saturate in the incrementing direction.

d) When **incrthreshold** has the Boolean value `true`, the incrementing threshold value is the *maximum value* (2^(number of counter bits) -1) of the counter. When **incrthreshold** has the Boolean value `false`, the counter does not have a threshold in the incrementing direction.

e) When **decrsaturate** has the Boolean value `true`, the decrementing saturate value is `0`. When **decrsaturate** has the Boolean value `false`, the counter does not saturate in the decrementing direction.

f) When **decrthreshold** has the Boolean value `true`, the decrementing threshold value is `0`. When **decrthreshold** has the Boolean value `false`, the counter does not have a threshold in the decrementing direction.

g) **incrthreshold**/**decrthreshold** used on the left-hand side of an assignment in SystemRDL assigns the counter's threshold to the number or reference specified in the right-hand side of the assignment.

h) **incrsaturate**/**decrsaturate** used on the left-hand side of an assignment in SystemRDL assigns the counter's saturation property to the number or reference specified in the right-hand side of the assignment.

i) **incrthreshold**/**decrthreshold** used on the right-hand side of an assignment in SystemRDL is referencing the counter's threshold output, which is a single bit value indicating whether the threshold has been crossed. This value shall only be asserted to `1` when the value is greater than or equal to **incrthreshold**/**threshold** or is less than or equal to **decrthreshold**.

j) **incrsaturate**/**decrsaturate** used on the right-hand side of an assignment in SystemRDL is referencing the counter's saturate output, which is a single bit value indicating whether the saturation has occurred. This value shall only be asserted to `1` when the value of the counter meets or exceeds the saturation value specified.

k) All static values used in Table 19 shall fit within the width of the field. All references need to be the same width.

9.8.2.2 Example

See *Examples 1 - 3* in 9.8.2.

9.9 Interrupt properties

Designs often have a need for interrupt signals for various reasons, e.g., so software can disable or enable various blocks of logic when errors occur. *Interrupts* are unlike most **field** properties in that they operate on both the register level and the field level. Any register which instantiates an interrupt field (a field with the **intr** property specified) is considered an interrupt register. Each *interrupt register* has an associated interrupt signal which is the logical `OR` of all interrupt fields in the register (post-masked/enabled if the fields are masked or enabled). By default, this interrupt signal is inferred as an output; however, register files and/or address maps can be used to further aggregate these interrupts (see Clause 12, Clause 13, and the hierarchical interrupt example in 17.2). Interrupts may be masked, or enabled by other **field**s or externally defined **signal**s—they have an easy way of being turned on and off by software if desired.

By default, all interrupt fields have the **stickybit** property; this can be suppressed (using **nonsticky**) or changed to **sticky**. The **stickybit** and **sticky** properties are similar as they both define a field as *sticky*, meaning once hardware or software has written a one (`1`) into any bit of the field, the value is stuck until

software clears the value (using a write or clear on read). The difference between **stickybit** and **sticky** is each bit in a **stickybit** field is handled individually, whereas **sticky** applies a sticky state to all bits in an instantiated field (which is useful when designers need to store a multi-bit value, such as an address). For single-bit fields, there is no difference between **stickybit** and **sticky**.

By default, all interrupts are level-triggered, i.e., the interrupt is triggered at the positive edge of the clock if the **next** value of the interrupt field is asserted. Since interrupts are typically **stickybit**, the value is latched and held until software clears the interrupt. The edge-interrupt triggering mechanisms (**posedge**, **negedge**, and **bothedge**), like level-triggered interrupts, are synchronous.

A **nonsticky** interrupt is typically used for hierarchical interrupts, e.g., a design has a number of interrupt registers (meaning a number of registers with one or more interrupt fields instantiated within). Rather than promoting a number of interrupt signals, the developer can specify an aggregate interrupt register (typically unmasked, though a **mask**/**enable** may be specified) containing the same number of fields as there are interrupt signals to aggregate. Each field is defined as a **nonsticky** interrupt and the **next** value of each interrupt is directly assigned an interrupt pin for each interrupt register to be aggregated. Interrupt types are defined with modifiers to the **intr** property. These modifiers are not *booleans* and are only valid in conjunction with the **intr** property. The **nonsticky** modifier can be used in conjunction with **posedge**, **negedge**, **bothedge**, and **level**.

The syntax for a interrupt property modifiers appears as follows.

[posedge | negedge | bothedge | level | nonsticky] intr;

Table 20 lists and describes the available interrupt types.

Table 20—Interrupt types

Interrupt	Description
posedge	Interrupt when `next` goes from low to high.
negedge	Interrupt when `next` goes from high to low.
bothedge	Interrupt when `next` changes value.
level	Interrupt while the `next` value is asserted and maintained (the default).
nonsticky	Defines a non-sticky (hierarchical) interrupt; the associated interrupt field shall not be locked. This modifier can be specified in conjunction with the other interrupt types.

Furthermore, there are additional interrupt properties that can be used to mask or enable an interrupt. The **enable**, **mask**, **haltenable**, and **haltmask** properties (see Table 21) are all properties of type *reference* that are used to point to other fields or signals in the SystemRDL description. The **mask** and **haltmask** properties can be assigned to **field**s and used to control the propagation of an interrupt. If an interrupt bit is set and connected to a **mask**/**enable**, the interrupt's final value is gated by the **mask**/**enable**. The logical description of this operation is

```
final interrupt value = interrupt value & enable;
final interrupt value = interrupt value & !mask;
final halt interrupt value = interrupt value & haltenable;
final halt interrupt value = interrupt value & !haltmask.
//Further information on interrupts and their behavior as well a more complete
//example can be found in 17.2.
```

Example

```
addrmap top {
  reg block_int_r {
    name = "Example Block Interrupt Register";
    desc = "This is an example of an IP Block with 3 int events. 2
            of these events are non fatal
            and the third event multi_bit_ecc_error is fatal";

    default hw=w; // HW can Set int only
    default sw=rw; // SW can clear
    default woclr; // Clear is via writing a 1

    field {
      desc = "A Packet with a CRC Error has been received";
      level intr;
    } crc_error = 0x0;
    field {
      desc = "A Packet with an invalid length has been received";
      level intr;
    } len_error = 0x0;
    field {
      desc="An uncorrectable multi-bit ECC error has been received";
      level intr;
    } multi_bit_ecc_error = 0 ;
  }; // End of Reg: block_int

  reg block_int_en_r {
    name = "Example Block Interrupt Enable Register";
    desc = "This is an example of an IP Block with 3 int events";

    default hw=na; // HW can't access the enables
    default sw=rw; // SW can control them

    field {
      desc = "Enable: A Packet with a CRC Error has been received";
    } crc_error = 0x1;

    field {
      desc = "Enable: A Packet with an invalid length has been
              received";
    } len_error = 0x1;

    field {
          desc = "Enable: A Packet with an invalid length has been received";/
   / Mask this off as it's a fatal interrupt
    } multi_bit_ecc_error = 0x0;
  }; // End of Reg: block_int_en_r

  reg block_halt_en_r {
    name = "Example Block Halt Enable Register";
    desc = "This is an example of an IP Block with 3 int events";

    default hw=na; // HW can't access the enables
    default sw=rw; // SW can control them

    field {
      desc = "Enable: A Packet with a CRC Error has been received";
```

```
    } crc_error = 0x0; // not a fatal error do not halt
    field {
      desc = "Enable: A Packet with an invalid length has been received";
    } len_error = 0x0; // not a fatal error do not halt
    field {
      desc = "Enable: A Packet with an invalid length has been received";
    } multi_bit_ecc_error = 0x1;                    // fatal error that will
    cause device to halt
  }; // End of Reg: block_halt_en_r

  // Block A Registers

  block_int_r     block_a_int; // Instance the Leaf Int Register
  block_int_en_r  block_a_int_en; // Instance the corresponding Int
                                  //Enable Register
  block_halt_en_r block_a_halt_en; // Instance the corresponding halt
                                   // enable register

  // This block connects the int bits to their corresponding int enables and
  // halt enables
  block_a_int.crc_error->enable = block_a_int_en.crc_error;
  block_a_int.len_error->enable = block_a_int_en.len_error;
  block_a_int.multi_bit_ecc_error->enable =
    block_a_int_en.multi_bit_ecc_error;
  block_a_int.crc_error->haltenable = block_a_halt_en.crc_error;
  block_a_int.len_error->haltenable = block_a_halt_en.len_error;
  block_a_int.multi_bit_ecc_error->haltenable =
    block_a_halt_en.multi_bit_ecc_error;
} ;
```

Table 21 defines the interrupt properties.

Table 21—Field access interrupt properties

Property	Behavior/Application	Type	Dynamic[a]
intr	Interrupt, part of interrupt logic for a register.	*boolean*	Yes
enable	Defines an interrupt enable (the inverse of **mask**); i.e., which bits in an interrupt field are used to assert an interrupt.	*reference*	Yes
mask	Defines an interrupt mask (the inverse of **enable**); i.e., which bits in an interrupt field are not used to assert an interrupt.	*reference*	Yes
haltenable	Defines a halt enable (the inverse of **haltmask**); i.e., which bits in an interrupt field are set to de-assert the halt out.	*reference*	Yes
haltmask	Defines a halt mask (the inverse of **haltenable**); i.e., which bits in an interrupt field are set to assert the halt out.	*reference*	Yes
sticky	Defines the entire field as sticky; i.e., the value of the associated interrupt field shall be locked until cleared by software (write or clear on read).	*boolean*	Yes
stickybit	Defines each bit in a field as sticky (the default); i.e., the value of each bit in the associated interrupt field shall be locked until the individual bits are cleared by software (write or clear on read).	*boolean*	Yes

[a]Indicates whether a property can be assigned dynamically.

9.9.1 Semantics

a) **enable** and **mask** are mutually exclusive.

b) **haltenable** and **haltmask** are mutually exclusive.

c) **nonsticky, sticky**, and **stickybit** are mutually exclusive.

d) The **sticky** and **stickybit** properties are normally used in the context of interrupts, but may be used in other contexts as well.

e) Assignments of **signal**s or **field**s to the **enable**, **mask**, **haltenable**, and **haltmask** properties shall be of the same bit width as the **field**.

f) **posedge**, **negedge**, **bothedge**, and **level** are only valid if **intr** is true and can only be specified as modifiers to the **intr** property—they cannot be specified by themselves.

g) **posedge**, **negedge**, **bothedge**, and **level** are mutually exclusive.

9.9.2 Example

This example illustrates the use of **sticky** and **stickybit** interrupts.

```
field { level intr; } some_int=0;
field {} some_mask = 1'b1;
field {} some_enable = 1'b1;

some_int->mask = some_mask;
some_int->haltenable = some_enable;

field { level intr; rclr;} a_multibut_int[4]=0;
// Individual bits being set 1 will
// Accumulate as this is stickybit by default

field { posedge intr; sticky; woclr; } some_multibit_int[4]=0;
// This field will hold the first value written to it until its cleared by
// writing ones
```

9.10 Miscellaneous field properties

There are additional properties for **field**s which do not fall into any of the previous categories. This subclause describes these additional miscellaneous properties.

a) The **encode** property enumerates a **field** definition for additional clarification purposes. **encode** can only be applied to a validly scoped component of type **enum**.

b) The **precedence** property specifies how contention issues are resolved during field updates, e.g., a field which has **hw=rw** and **sw=rw**.

   1) **precedence = sw** (the default) indicates software takes precedence over hardware on accessing registers (over the hardware updates of type **we**, **wel**, **incr**, **decr**, **hwset**, and **hwclr**). This is a field-only property and does not affect the other fields in the register.

   2) **precedence = hw** indicates hardware takes precedence over software on accessing registers (on the hardware updates of type **we**, **wel**, **incr**, **decr**, **hwset**, and **hwclr**). This is a field-only property and does not affect the other fields in the register.

   3) In some cases of collisions between hardware and software, both operations can be satisfied, but this is beyond the scope of this document and such behavior is undefined.

c) The **paritycheck** property can be applied to a **field** to indicate it should be covered and checked by parity.

1) The default is **false** (no check occurs).
2) Not all fields in a register need to have the same **paritycheck** property value.
3) Parity is calculated each cycle on the next value of every qualifying bit and the result is stored.
4) Parity is checked each cycle by comparing the generated parity on the current value of each qualifying bit with the stored parity result. A `parity_error` output for the `addrmap` is set to `1` when the generated value and stored parity do not match.

Table 22 details the miscellaneous field properties.

Table 22—Miscellaneous properties

Property	Behavior/Application	Type	Dynamic[a]
encode	Binds an enumeration to a field.	*reference* to *enum*	Yes
precedence	Controls whether precedence is granted to hardware (**hw**) or software (**sw**) when contention occurs.	*prece-dencetype*	Yes
paritycheck	Indicates whether this field is to be checked by parity.	*boolean*	No

[a]Indicates whether a property can be assigned dynamically.

9.10.1 Semantics

a) An **encode** property shall be assigned to an **enum** type.
b) The enumeration's values shall fit inside the field width.

9.10.2 Example

This example shows **paritycheck**, **precedence**, and **encode**. Here `hdrPreamble` is covered by and checked by parity, while `hdrType` is not.

```
enum cfg_header_type_enum {
    normal         = 7'h00  { desc = "Type 0 Configuration Space Header"; };
    pci_bridge     = 7'h01  { desc = "PCI to PCI Bridge"; };
    cardbus_bridge = 7'h10  { desc = "PCI to CardBus Bridge"; };
  };

field {
      hw = rw; sw = rw;
    precedence = sw;
       encode = cfg_header_type_enum;
    } hdrType [6:0]=0;

field {
      hw = rw; sw = rw;
    paritycheck;
    } hdrPreamble [15:8]=0;
```

10. Register component

In SystemRDL, a *register* is defined as a set of one or more SystemRDL field instances that are atomically accessible by software at a given address. A register definition specifies its width and the types and sizes of the fields that fit within that width (the register file and address map components determine address allocation; see Clause 12 and Clause 13).

Registers can be instantiated in three forms.

— **internal** implies all register logic is created by the SystemRDL compiler for the instantiation (the default form).

— **external** signifies the register/memory is implemented by the designer and the interface is inferred from instantiation.

— **alias** allows software to access another register with different properties (i.e., **read**, **write**, **woclr**, etc.). Alias registers are used where designers want to allow alternate software access to registers. SystemRDL allows designers to specify alias registers for internal or external registers.

10.1 Defining and instantiating registers

Register components (**reg**) have the same definition and instantiation syntax as other SystemRDL components; see 5.1. The following semantics apply for all registers.

a) Within a register, the only components that can be instantiated are **field** components, **signal**s, and **constraint**s.

b) Within a register, the only components that can be defined are **field** components, **enum**s, **constraint**s and **signal**s.

c) At least one **field** shall be instantiated within a register.

d) Two field instances shall not occupy overlapping bit positions within a register unless one field is read-only and the other field is write-only.

e) Field instances shall not occupy a bit position exceeding the MSB of the register. The default width of a register (**regwidth**) is 32 bits.

f) All registers shall have a width = 2^N, where N >=3.

g) Field instances that do not have explicit bit positions specified are automatically inferred based on the **addrmap** mode of **lsb0** (the default) or **msb0**.

h) Registers shall not overlap, unless one contains read-only fields and the other contains only write-only or write-once-only fields.

10.2 Instantiating registers

All register instantiations follow the same syntax and semantics, with minor differences depending on the instantiated register's **internal** or **external** state. Unless specified as **external** (see 10.4), registers are, by default, **internal**.

a) A *definitive* register instantiation appears as follows.

[**external**] *reg_name* [**#(***parameter_instance* [**,** *parameter_instance*]***)**]
reg_instance_element [**,** *reg_instance_element*]* **;**

where

1) *reg_name* is the user-specified register name.

2) *parameter_instance* is specified as follows (see 5.1.2 a).

.*param_name***(***param_val***)**

3) *reg_instance_element* is defined as follows.

reg_instance_name [{**[***constant_expression***]**}* [*addr_alloc*]

where

i) *reg_instance_name* is the user-specified name for instantiation of the register.

ii) *constant_expression* is an expression that resolves to a `longint unsigned`.

iii) **[***constant_expression***]** specifies the size of the instantiated **reg** array (optionally multidimensional).

iv) *addr_alloc* is an address allocation operator (see 5.1.2.3).

v) When using multiple-dimensions, the last subscript increments the fastest.

b) An *anonymous definition* (and instantiation) of a register appears as follows.

reg {[*reg_body*]**}** [**external**] *reg_instance_element* [**,** *reg_instance_element*]***;**

where

1) *reg_body* is as described in 5.1.1, subject to the following limitations.

i) Component definitions are limited to **field**, **constraint**, **signal**, and **enum** components.

ii) Component instantiations are limited to **field**, **constraint**, and **signal** instances.

2) *reg_instance_element* is the description of the register instantiation attributes, as defined in 10.2 a 3.

10.3 Instantiating internal registers

Registers whose implementation can be built by a SystemRDL compiler are called *internal registers*.

Example

This example illustrates the definition and instantiation of **internal** registers.

```
reg myReg { field {} data[31:0]; };
myReg intReg; // single internal register
myReg intArray[32]; // internal register array of size 32
```

10.4 Instantiating external registers

SystemRDL can describe a register's implementation as external, which is applicable for large arrays of registers and provides an alternate implementation to what a SystemRDL compiler might provide. *External registers* are identical to internal registers, except the actual implementation of the register is not created by the compiler and the fields of an external register are not inferred to be implemented as wires and flip-flops.

Registers shall be instantiated as external registers by placing the keyword **external** before the register type name or by instantiating the component as described in 10.2.

Example

This example illustrates the definition and instantiation of **external** registers.

```
reg myReg { field {} data[31:0]; };
external myReg extReg; // single external register
external myReg extArray[32]; // external register array of size 32
```

10.5 Instantiating alias registers

An *alias register* is a register that appears in multiple locations of the same address map. It is physically implemented as a single register such that a modification of the register at one address location appears at all the locations within the address map. The accessibility of this register may be different in each location of the address block.

Alias registers are allocated addresses like physical registers and are decoded like physical registers, but they perform these operations on a previously instantiated register (called the *primary register*). Since alias registers are not physical, hardware access and other hardware operation properties are not used. Software access properties for the alias register can be different from the primary register.

10.5.1 Semantics

Registers shall be instantiated as alias registers by placing the keyword **alias** before the register type name.

a) An instantiation of an alias register appears as follows.

reg_name reg_primary_inst;

alias *reg_primary_inst reg_name reg_instance*;

where

1) *reg_name* is the user-specified register name.
2) *reg_instance* is the user-specified name for instantiation of the component.
3) *reg_primary_inst* is the primary register to which the alias is bound

b) Every field in the alias register needs to have the same instance name as a **field** in the primary register (though the field type may differ) and the two fields shall have the same position and size in each (corresponding) register.

c) The alias register is not required to have all the fields from the primary register.

d) The alias register shall have the same width as the primary register.

e) Only the following SystemRDL properties may be different in an alias: **desc**, **name**, **onread**, **onwrite**, **rclr**, **rset**, **sw**, **woclr**, **woset**, and any user-defined properties.

f) If the alias instance type (`internal` or `external`) is specified, it shall match the primary register instance type. If the alias instance type not specified, it uses the primary register instance type.

10.5.2 Example

This example shows the usage of register aliasing and how the primary register and its alias can have different properties.

```
reg some_intr_r { field { level intr; hw=w; sw=r; woclr; } some_event; };
addrmap foo {
    some_intr event1;
    // Create an alias for the DV team to use and modify its properties
    // so that DV can force interrupt events and allow more rigorous structural
    // testing of the interrupt.
    alias event1 some_intr event1_for_dv;
        event1_for_dv.some_event->woclr = false;
        event1_for_dv.some_event->woset = true;
};
```

The `alias` above could be done with a different register type as well, without dynamic assigns.

```
    alias event1 some_intr_rw event1_for_dv;
```

10.6 Register properties

Table 23 lists and describes the register properties.

Table 23—Register properties

Property	Implementation/Application	Type	Dynamic[a]
regwidth	Specifies the bit-width of the register (power of two).	*longint unsigned*	No
accesswidth	Specifies the minimum software access width (power of two) operation that may be performed on the register.	*longint unsigned*	Yes
errextbus	The associated register has error input.	*boolean*	No
intr	Represents the inclusive `OR` of all the interrupt bits in a register after any **field enable** and/or **field mask** logic has been applied.	N/A	No
halt	Represents the inclusive `OR` of all the interrupt bits in a register after any **field haltenable** and/or **field haltmask** logic has been applied.	N/A	No
shared	Defines a register as being shared in different address maps. This is only valid for register components and shall only be applied to shared components. See 13.5 for more information.	*boolean*	No

[a]Indicates whether a property can be assigned dynamically.

10.6.1 Semantics

a) All registers shall have a **regwidth** = 2^N, where N >=3.

b) All registers shall have a **accesswidth** = 2^N, where N >=3.

c) The value of the **accesswidth** property shall not exceed the value of the **regwidth** property.

d) The default value of the **accesswidth** property shall be identical to the width of the register.

e) Partial software reads of all fields without read side-effects are valid.

f) Any field that is software-writable or clear on read shall not span multiple software accessible subwords (e.g., a 64-bit register with a 32-bit access width may not have a writable field with bits in both the upper and lower half of the register).

g) If a register instance is not explicitly assigned an address, a compiler needs to automatically assign the address (see 13.4). Addressing is inherited from the enclosing lexical scope and applies to any direct child instances.

h) **errextbus** is only valid for external registers. It specifies an external register implementation indicating that a transaction terminated with an error. This error status is incorporated in the **addrmap** implementation transaction error indication.

10.6.2 Example

These are examples of using register properties.

```
reg my64bitReg { regwidth = 64;
   field {} a[63:0]=0;
};
reg my32bitReg {    regwidth = 32;
   accesswidth = 16;
   field {} a[16]=0;
   field {} b[16]=0;
};
```

10.7 Understanding field ordering in registers

Users can specify bit ordering implicitly and explicitly in two different ways. These approaches are called **msb0** and **lsb0** in SystemRDL (see Table 26). Users who explicitly specify bit indexes when instantiating fields in registers do not need to specify one of these attributes, as the explicit indexes imply one of these bit ordering schemes. See also 17.3.

a) The syntax:

field_type field_instance **[***high***:***low***]**
implies the use of **lsb0** ordering (the default)

b) Alternately:

field_type field_instance **[***low***:***high***]**
implies the use of **msb0** ordering

where

1) *low* and *high* are `longint unsigned`;
2) *low* == *high* implies a single bit field at the specified location;
3) for multi-bit fields, *low* < *high*.
4) The left-value is the index of the most significant bit of the field; the right-value is the index of is the least significant bit of the field.

If a form specifying only a field's size is used, then any fields are packed contiguously, end-to-end, starting at index `0` for **lsb0** registers and starting at index `regwidth-1` for **msb0** registers.

10.7.1 Semantics

a) Both the `[`*`low`*`:`*`high`*`]` and `[`*`high`*`:`*`low`*`]` bit specification forms shall not be used together in the same register.

b) As long as all the registers in an address map are consistently **msb0** or **lsb0**, no explicit **msb0** or **lsb0** property needs to be defined.

c) Setting `lsb0=true` implies `msb0=false`; setting `msb0=true` implies `lsb0=false`.

10.7.2 Examples

This example shows how fields are packed when using **lsb0** bit ordering.

```
lsb0;
reg {

       field {} A; // Single bit from 0 to 0
       field {} B[3]; // 3 bits from 3 down to 1
                      // 4 bits from 7 down to 4 are reserved and unused
       field {} C[15:8]; // 8 Bits from 15 to 8
       field {} C[5]; // 5 Bits from 20 down to 16
} regA;
```

This example shows how fields are packed when using **msb0** bit ordering.

```
msb0;
reg {
       field {} A; // Single bit from 31 to 31
       field {} B[3]; // 3 bits from 28 to 30
                      // 12 bits from 16 to 27 are reserved and unused
       field {} C[8:15]; // 8 Bits from 8 to 15
```

```
        field {} C[5]; // 5 Bits from 3 to 7
} regA;
```

10.8 Understanding interrupt registers

As discussed in 9.9, the field property **intr** also affects registers. Any register that contains an interrupt field has two implied properties: **intr** and **halt**. These properties are outputs of the register. The **intr** register property represents the inclusive `OR` of all the interrupt bits in a register after any **field enable** and/or **field mask** logic has been applied. The **halt** register property represents the inclusive `OR` of all the interrupt bits in a register after any **field haltenable** and/or **field haltmask** logic has been applied.

10.8.1 Semantics

a) The **intr** and **halt** register properties are outputs; they should only occur on the right-hand side of an assignment in SystemRDL.

b) The **intr** property shall always be present on a **intr** register even if no mask or enables are specified.

c) The **halt** property shall only be present if **haltmask** or **haltenable** is specified on at least one **field** in the register.

10.8.2 Example

This example connects an implicit **intr** output property to another field.

```
reg {
  field { intr; } some_intr;
  field { intr; } some_other_intr;
} some_intr_reg;
reg {
  field {} a;
} some_status_reg;
some_status_reg.a->next = some_intr_reg->intr;
```

11. Memory component

A *memory* is an array of storage consisting of a number of entries of a given bit width. The physical memory implementation is technology dependent and memories shall be **external**. Child instances within a memory are virtual instances. A virtual instance does not have a physical implementation, but, it is a software view of the memory data. A memory can contain instances of virtual registers and fields within a virtual register are virtual fields.

11.1 Defining and instantiating memories

Memory components have the same definition as other SystemRDL components; see 5.1.1. Memories introduce the concepts of address allocation and their supporting operators. These address allocation operators are applied after the instance name of the component. All addressing in SystemRDL is done based on byte addresses.

a) A *definitive definition* of a memory instantiation appears as follows.

external *mem_name* [**#(***parameter_instance* [**,** *parameter_instance*]*)]
mem_instance_element [**,** *mem_instance_element*]* **;**

where

1) *mem_name* is the user-specified memory name.
2) *parameter_instance* is specified as follows (see 5.1.2 a).

.*param_name***(***param_val***)**

3) *mem_instance_element* is defined as follows.

mem_instance_element [{**[***constant_expression***]**}* [*addr_alloc*]

where

i) *mem_instance_element* is the user-specified name for instantiation of the memory.
ii) *constant_expression* is an expression that resolves to a `longint unsigned`.
iii) **[***constant_expression***]** specifies the size of the instantiated **mem** array (optionally multidimensional).
iv) *addr_alloc* is an address allocation operator (see 5.1.2.3).
v) When using multiple-dimensions, the last subscript increments the fastest.

b) An *anonymous definition* (and instantiation) of a memory appears as follows.

mem {[*mem_body*]} **external** *mem_instance_element* [**,** *mem_instance_element*]* **;**

where

1) *mem_body* is as described in 5.1.1, subject to the following limitations.
i) Component definitions are limited to **field**, **reg**, **constraint**, and **enum** components.
ii) Component instantiations are limited to **reg** and **constraint** instances.
2) *mem_instance_element* is the description of the memory instantiation attributes, as defined in 11.1 a 3.

11.2 Semantics

a) All `mem` instances shall have an **external** instance type specified.
b) Addresses in SystemRDL are always byte addresses.
c) Within a memory, the only components that can be instantiated shall be virtual register components.
d) Memories can contain `reg` instances. Instances of `reg` instances within a memory are virtual registers.

e) Virtual register width is limited to the minimum power of two bytes, which can contain the memory width, and all the virtual fields shall fit within the memory width.

f) Virtual registers, register files, and fields shall have the same software access (**sw** property value) as the parent memory.

g) Hardware properties on virtual register and fields are ignored.

h) Virtual fields cannot have software properties other **sw**.

i) The address space occupied by virtual registers shall be less than or equal to the address space provided by the memory.

j) Virtual registers cannot overlap.

k) Virtual register instances are optional.

l) A **mem** cannot be prefixed by **alias**.

11.3 Memory properties

Table 24 lists and describes the memory properties.

Table 24—Memory properties

Property	Implementation/Application	Type	Dynamic[a]
mementries	The number of memory entries.	*longint unsigned*	No
memwidth	The memory entry bit width.	*longint unsigned*	No
sw	Programmer's ability to read/write a memory.	*access type*	Yes

[a]Indicates whether a property can be assigned dynamically.

11.3.1 Semantics

a) **mementries** shall be greater than `0`.

b) **mementries** defaults to `1`.

c) **memwidth** shall be greater than `0`.

d) **memwidth** defaults to **regwidth**.

11.3.2 Example

This example shows an application of memory component properties.

```
mem fifo_mem {
    mementries = 1024;
    memwidth = 32;
};
```

12. Register file component

A *register file* is as a logical grouping of one or more register and register file instances;. The register file provides address allocation support, which is useful for introducing an address gap between registers. The only difference between the register file component (**regfile**) and the **addrmap** component (see Clause 13) is an **addrmap** defines an RTL implementation boundary where the **regfile** does not. Since **addrmap**s define a implementation block boundary, there are some specific properties that are only specified for address maps (see Clause 13) and not specified for **regfile**s.

12.1 Defining and instantiating register files

Register file components have the same definition as other SystemRDL components; see 5.1.1. Register files introduce the concepts of address allocation and their supporting operators. These address allocation operators are applied after the instance name of the component. All addressing in SystemRDL is done based on byte addresses.

a) A *definitive* register file instantiation appears as follows.

[external | internal] *regfile_name* **[#(***parameter_instance* **[,** *parameter_instance***]*)]**
regfile_instance_element **[,** *regfile_instance_element***]* ;**

where

1) *regfile_name* is the user-specified **regfile** name.

2) *parameter_instance* is specified as follows (see 5.1.2 a).

.*param_name***(***param_val***)**

3) *regfile_instance_element* is defined as follows.

regfile_instance_name [{**[***constant_expression***]**}* [*addr_alloc*]

where

i) *regfile_instance_name* is the user-specified name for instantiation of the register file.

ii) *constant_expression* is an expression that resolves to a `longint unsigned`.

iii) **[***constant_expression***]** specifies the size of the instantiated **regfile** array (optionally multi-dimensional).

iv) *addr_alloc* is an address allocation operator (see 5.1.2.3).

v) When using multiple-dimensions, the last subscript increments the fastest.

b) An *anonymous definition* (and instantiation) of a register file appears as follows.

regfile {[*regfile_body*]**}** **[external | internal]**
regfile_instance_element **[,** *regfile_instance_element***]* ;**

where

1) *regfile_body* is as described in 5.1.1, subject to the following limitations.

i) Component definitions are limited to **field**, **reg**, **regfile**, **signal**, **constraint**, and **enum** components.

ii) Component instantiations are limited to **reg**, **regfile**, **constraint**, and **signal** instances.

2) *regfile_instance_element* is the description of the register file instantiation attributes, as defined in 12.1 a 3.

12.2 Semantics

a) Addresses in SystemRDL are always byte addresses.

b) Within a **regfile**, the only components that can be instantiated shall be a **regfile**, **reg**, **constraint**, and **signal**.

c) At least one **reg** or **regfile** shall be instantiated within a **regfile**.

d) A **regfile** may contain heterogeneous **internal**, **external**, and **alias** registers.

e) A **regfile** cannot be prefixed by **alias**. Only individual registers can be aliased.

f) If a **regfile** is declared **internal**, all registers in it are coerced to be **internal**, regardless of any **internal** or **external** declaration on the register instantiations. Similarly, if the **regfile** is declared **external**, all registers are coerced to be **external**; in this case, aliased registers need to be handled externally as well. If the **regfile** is not declared as either, the register instances are **internal**, **alias**, or **external** according to their individual declarations.

12.3 Register file properties

Table 25 lists and describes the register file properties.

Table 25—Register file properties

Property	Implementation/Application	Type	Dynamic[a]
alignment	Specifies alignment of all instantiated components in the associated register file.	*longint unsigned*	No
sharedextbus	Forces all external registers to share a common bus.	*boolean*	No
errextbus	For an external **regfile**, the associated **regfile** has an error input.	*boolean*	No

[a]Indicates whether a property can be assigned dynamically.

12.3.1 Semantics

a) All **alignment** values shall be a power of two (`1`, `2`, `4`, etc.) and shall be in units of bytes.

b) The default for **alignment** is the address (of the register file) aligned to the width of the component being instantiated (e.g., the address of a 64-bit register is aligned to the next 8-byte boundary).

c) The **sharedextbus** property is only relevant when dealing with multiple external components.

1) It creates a single set of control signals for entire regfile, instead of per register.

2) Write data is common to all, with max MSB and min LSB from all registers, and populated based on each register's field positions.

3) Read data is common to all, with max MSB and min LSB from all registers, and extracted based on each register's field positions.

4) For **regfile**s without an instance type, selects for each external register or **regfile** accompany the common control signals.

5) For external **regfile**s, an address bus with an offset relative to the beginning of the regfile is provided.

d) The **errextbus** property is only considered for external **regfile**s and, when nested, only the outermost external **regfile**. **errextbus** specifies an external **regfile** implementation indicating a transaction terminated with an error. This error status is incorporated in the top `addrmap` implementation transaction error indication.

e) If a register file instance is not explicitly assigned an address, an application needs to automatically assign the address.

12.3.2 Example

This example shows an application of register file component properties.

```
regfile fifo_rfile {
  alignment = 8;
  reg {field {} a;} a; // Address of 0
  reg {field {} a;} b; // Address of 8. Normally would have been 4
};
regfile {
  external fifo_rfile fifo_a;// Single regfile instance
  external fifo_rfile fifo_b[64]; // Array of regfiles
sharedextbus; // Create a common external bus for both of these instantiations
              // rather than separate external interfaces
} top_regfile;
```

13. Address map component

An address component map (**addrmap**) contains registers, register files, memories, and/or other address maps and assigns a virtual address or final addresses; this map defines the boundaries of an implementation (e.g., RTL). A *virtual address* is used on an address map that is intended to be used as a component in a higher-level, or hierarchical, address map. A *final address* is used for the top-level address map (one that is not contained in any other address maps). There is no difference in how addresses are specified. All addresses are virtual until the root of the tree is reached.

13.1 Introduction

During generation, the address map can be converted into an HDL module. All registers and fields instantiated within an address map file shall be generated within this module. Therefore, some properties are only valid for **addrmap**s and not for **regfile**s. Other than these properties and their suggested behavior, there is no difference between address maps and register files.

13.2 Defining and instantiating address maps

Address map components have the same definition and instantiation syntax as other SystemRDL components; see 5.1. The address allocation operators are shown in 5.1.2.3.

13.3 Semantics

a) The components instantiated within an address map shall be registers, register files, memories, address maps, or signals.

b) At least one register, register file, memory, or address map shall be instantiated within an address map.

13.4 Address map properties

A compiler generating an implementation based on SystemRDL has to create an external interface for each external component created. The **sharedextbus** property can be used to combine multiple external components into a single interface.

The other critical aspect to understand about address maps is how the global *addressing modes* work. There are three addressing modes defined in SystemRDL: **compact**, **regalign**, and **fullalign**. See 5.1.2.2.2.

Table 26 describes the address map properties.

Table 26—Address map properties

Property	Implementation/Application	Type	Dynamic[a]
alignment	Specifies alignment of all instantiated components in the address map.	*longint unsigned*	No
sharedext-bus	Forces all external registers to share a common bus.	*boolean*	No
errextbus	The associated `addrmap` instance has an error input.	*boolean*	No
bigendian	Uses big-endian architecture in the address map.	*boolean*	Yes
littleendian	Uses little-endian architecture in the address map.	*boolean*	Yes

Table 26—Address map properties (Continued)

Property	Implementation/Application	Type	Dynamic[a]
addressing	Controls how addresses are computed in an address map.	*addressingtype*	No
rsvdset	The read value of all fields not explicitly defined is set to `1` if **rsvdset** is *True*; otherwise, it is set to `0`.	*boolean*	No
rsvdsetX	The read value of all fields not explicitly defined is unknown if **rsvdsetX** is *True*.	*boolean*	No
msb0	Specifies register bit-fields in an address map are defined as `0:N` versus `N:0`. This property affects all fields in an address map.	*boolean*	No
lsb0	Specifies register bit-fields in an address map are defined as `N:0` versus `0:N`. This property affects all fields in an address map. This is the default.	*boolean*	No

[a]Indicates whether a property can be assigned dynamically.

13.4.1 Semantics

a) The default for the **alignment** shall be the address is aligned to the width of the component being instantiated (e.g., the address of a 64-bit register is aligned to the next 8-byte boundary).

b) All **alignment** values shall be a power of two (`1`, `2`, `4`, etc.) and shall be in units of bytes.

c) The **sharedextbus** property is only relevant when the **addrmap** contains external component instances.

   1) **sharedextbus** creates a single set of control signals for the entire **addrmap**, instead of per external `reg`, `regfile`, or `mem` instance.

   2) Any outgoing data is common to all direct external instances.

   3) Any incoming data is common to all direct external instances.

   4) `addrmap` instances are always considered external. An address with offset relative to the beginning of the `addrmap` instance is provided.

   5) If **errextbus** exists for an `addrmap` instance, an error input shall exist for that `addrmap` instance in the parent `addrmap`.

d) **errextbus** specifies an `addrmap` implementation indicating a transaction terminated with an error. This error status is incorporated in the top `addrmap` implementation transaction error indication.

e) **regalign** is identical to **compact**, except when dealing with **mem**s, **regfile**s, or **addrmap**s. If an array of components is 256 items deep and 8 bytes wide, then the next address is `(addr[2:0] == 0)` and it is only aligned to the size of the **regfile**, not the total size of the array.

f) **fullalign** is identical to **compact**, except when dealing with **mem**s, **regfile**s, or **addrmap**s. If an array of components is 256 items deep and 8 bytes wide, then the next address is aligned to `256*8` or `2048`.

g) **rsvdsetX** does not affect SystemRDL generated implementations; it can be used to verify legacy designs which do not have consistent data values for reserved fields.

h) **rsvdset** and **rsvdsetX** are mutually exclusive.

i) **msb0** and **lsb0** are mutually exclusive.

j) The **bigendian** and **littleendian** properties are used for controlling byte ordering and have no effect on bit ordering.

13.4.2 Example

See the examples shown in 5.1.2.2.2.

13.5 Defining bridges or multiple view address maps

A *bridge addrmap* is a container for address maps which can be accessed by multiple masters. Table 27 lists and describes these address map bridge properties.

Table 27—Bridge properties

Property	Implementation/Application	Type	Dynamic[a]
bridge	Defines the parent address map as being a bridge. This shall only be applied to the root address map which contains the different views of the sub address maps.	*boolean*	No

[a]Indicates whether a property can be assigned dynamically.

13.5.1 Semantics

To create a bridge, use a parent address map with a **bridge** property which contains two or more sub address maps representing the different views.

13.5.2 Example

This example below shows a bridge between two bus protocols.

```
addrmap some_bridge { // Define a Bridge Device
   desc="overlapping address maps with both shared register space and
   orthogonal register space";
   bridge; // This tells the compiler the top level map contains other maps
  reg status {// Define at least 1 register for the bridge
   shared;    // Shared property tells compiler this register
              // will be shared by multiple addrmaps
   field {
      hw=rw;
      sw=r;
    } stat1 = 1'b0;
  };

  reg some_axi_reg {
    field {
      desc="credits on the AXI interface";
    } credits[4] = 4'h7;                                  // End of field: {}
  };                                                  // End of Reg: some_axi_reg

  reg some_ahb_reg {
    field {

      desc="credits on the AHB Interface";
    } credits[8] = 8'b00000011 ;
  };

   addrmap {
      littleendian;
      some_ahb_reg ahb_credits; // Implies addr = 0
      status       ahb_stat @0x20; // explicitly at address=20
```

```
         ahb_stat.stat1->desc = "bar"; // Overload the registers property in
                         // this instance
      } ahb;

      addrmap { // Define the Map for the AXI Side of the bridge
         bigendian; // This map is big endian
         some_axi_reg axi_credits;     // Implies addr = 0
         status       axi_stat @0x40;  // explicitly at address=40
         axi_stat.stat1->desc = "foo"; // Overload the registers property
                                       // in this instance
      } axi;
   }; // Ends addrmap bridge
```

14. Verification constructs

This clause describes certain special constructs which are specifically used for creating verification models from the SystemRDLspecification.

14.1 HDL path

An *HDL path* specifies the hierarchical path of the design storage element corresponding to the address map, register, register field, or memory. By specifying an HDL path, the verification environment can have direct access to memory, register, and field implementation nets in a Design Under Test (DUT). The complete HDL path of a component is the concatenation of all HDL paths from the top-level down to the component.

An **hdl_path_slice** or **hdl_path_gate_slice** can be put on a **field** or **mem** component. It can be used when the corresponding RTL or gate-level netlist is not contiguous. The property value is an array of **hdl_path** strings, each pointing to the corresponding element in the RTL or gate-level netlist.

14.1.1 Assigning HDL path

An HDL path can be assigned using property **hdl_path** or **hdl_path_gate**. Table 28 lists and describes the HDL path properties.

Table 28—HDL path properties

Property	Implementation/Application	Type	Dynamic[a]
hdl_path	Assigns the RTL hdl_path for an **addrmap**, **reg**, or **regfile**.	*string*	Yes
hdl_path_slice	Assigns a list of RTL **hdl_path** for a **field** or **mem**.	*string []*	Yes
hdl_path_gate	Assigns the gate-level **hdl_path** for an **addrmap**, **reg**, or **regfile**.	*string*	Yes
hdl_path_gate_slice	Assigns a list of gate-level hdl_path for a **field** or **mem**.	*string []*	Yes

[a]Indicates whether a property can be assigned dynamically.

14.1.1.1 hdl_path and hdl_path_gate

hdl_path and **hdl_path_gate** are properties which contain a string-based path to the storage in an RTL or gate-level netlist.

hdl_path = "*path*";

hdl_path_gate = "*path*";

where

path is the `path` specified as an argument to the method calls in IEEE 1800.2, subclause 19.6.

The RHS value of **hdl_path** can contain a bit slice, which is specified by using `[:]`, e.g., `"reg1[31:12]"`.

14.1.1.2 hdl_path_slice and hdl_path_gate_slice

hdl_path_slice and **hdl_path_gate_slice** are properties which contain an array of string-based paths to the storage in an RTL or gate-level netlist.

hdl_path_slice = ‘{"*path*" [, "*path*"]*};
hdl_path_gate_slice = ‘{"*path*" [, "*path*"]*};

hdl_path_slice’s shall be used when the component has multiple storage elements in the RTL or gate.

14.1.2 Examples

Example 1

The corresponding (**TOP**) diagram for this example is shown in Figure 1.

```
addrmap block_def #( string ext_hdl_path = "ext_block") {
  hdl_path = "int_block" ;

  reg {
    hdl_path = { ext_hdl_path, ".externl_reg" } ;
    field {
      hdl_path_sice = '{ "field1" } ;
    } f1 ;
  } external external_reg ;

  reg {
    hdl_path = "int_reg" ;
    field {
      hdl_path_slice = '{ "field1" } ;
    } f1 ;
  } internal_reg ;
} ;

addrmap top {
  hdl_path = "TOP" ;
  block_def #( .ext_hdl_path("ext_block0")) int_block0 ;
  int_block0 -> hdl_path = "int0" ;
  block_def #( .ext_hdl_path("ext_block1")) int_block1 ;
  int_block1 -> hdl_path = "int1" ;
} ;
```

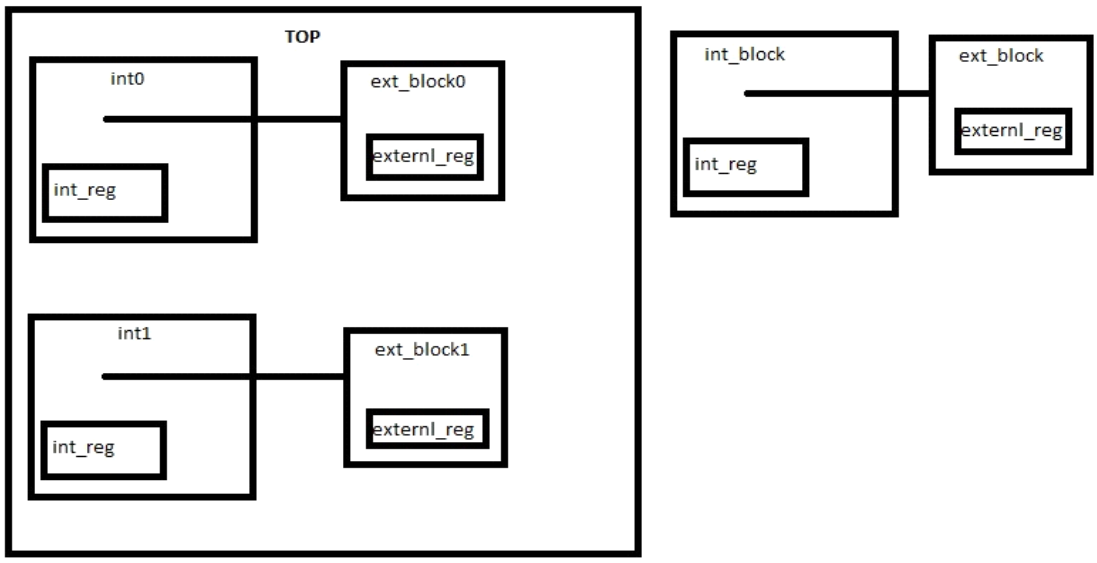

Figure 1—TOP diagram

Example 2

```
addrmap top{
    reg {
        field {
            hdl_path_slice = '{"rtl_f1_1", "rtl_f1_0"};
        } f1[1:0];
        field {
            hdl_path_slice = '{"rtl_f2"};
        } f2[5:3];
    } Reg1 ;
    Reg1.f2 -> hdl_path_slice = '{"rtl_f2_5_4", "rtl_f2_3"};
    // the hdl_path for Reg1.f2 has been overridden dynamically
};
```

14.2 Constraints

A *constraint* is a value-based condition on one or more components; e.g., constraint-driven test generation allows users to automatically generate tests for functional verification. Constraints can be instantiated within all structural components.

14.2.1 Describing constraints

Constraints can be assigned using a **constraint** component. A **constraint** component can contain one or more constraint expressions. A *constraint expression* is a value-based condition on one or more **field**s visible from the scope in which the **constraint** component is defined. The field references may only be to a `field` instance and not a property of the **field**. Fields can also be specified using the hierarchical operator (.) (see 5.1.4). If the constraint is instantiated inside a field, the **this** keyword can be used to reference the enclosing field.

Note that the **this** keyword is legal only in the context of expressions defined in **constraint** components. Expressions defined in **constraint** components are not evaluated by SystemRDL compilers and should be transferred as written for runtime evaluation, except when translating **field** and **enum** type references.

A **field** can be limited to values in a set using the **inside** keyword and the set operator (**{}**). Values in a set can be specified separated by commas (**,**) or a range can be specified using the `[:]` syntax. The right-hand side of an **inside** can also be an **enum** type reference. When translating enum type references to SystemVerilog, the reference shall be replaced by the list of enumerators defined in the **enum** type.

Any constraint can be disabled by setting the constraint's **constraint_disable** property (see Table 29) to `true`.

14.2.2 Constraint component

a) Definitive definition

constraint *constraint_component_name* **{**[*constraint_body*]**};**

*constraint_component_name constraint_inst***;**

where

1) *constraint_component_name* is the user-specified constraint name.
2) *constraint_body* can contain one of more of the following.
   i) *LHS operator RHS*

   where

   LHS can be a *field name reference* or the **this** keyword.

   operator can be any relational operator (see Table 7).

   RHS can be any valid SystemRDL *expression*.

   ii) *LHS* **inside** **{***open_range_list***}**

   where

   LHS can be a *field name reference* or the **this** keyword.

   open_range_list is defined as follows

   constraint_value [**,** *constraint_value*]*

   and *constraint_value* is defined as either an expression or a range, as follows.

   constant_expression | **[***constant_expression* **:** *constant_expression***]**

   iii) *LHS* **inside** *enum_type_ref*

   where

   LHS can be a *field name reference* or the **this** keyword.

   enum_type_ref is an `enum` type reference.

   iv) An assignment to a universal property (see Table 5) or **constraint_disable** (see Table 29).
3) *constraint_inst* is the user-specified name for instantiation of the component.

b) Anonymous definition

constraint **{**[*constraint_body*]**}** *constraint_component_name***;**

where

1) *constraint_body* can contain one of more of the following.
   i) LHS operator RHS

   where

   LHS can be a *field name reference* or the **this** keyword.

   operator can be any relational operator (see Table 7).

RHS can be any valid SystemRDL *expression*.

ii) *LHS* **inside** {*open_range_list*}

where

LHS can be a *field name reference* or the **this** keyword.

open_range_list is defined as follows

constraint_value [**,** *constraint_value*]*

and *constraint_value* is defined as either an expression or a range, as follows.

constant_expression | [*constant_expression* **:** *constant_expression*]

iii) *LHS* **inside** *enum_type_ref*

where

LHS can be a *field name reference* or the **this** keyword.

enum_type_ref is an `enum` type reference.

iv) An assignment to a universal property (see Table 5) or **constraint_disable** (see Table 29).

2) *constraint_component_name* is the user-specified constraint name.

Table 29 lists and describes the constraint properties.

Table 29—Constraint properties

Property	Implementation/Application	Type	Dynamic[a]
constraint_ disable	Specifies whether to disable (`true`) or enable (`false`) constraints.	*boolean*	Yes

[a]Indicates whether a property can be assigned dynamically.

14.2.3 Example

```
constraint not_valid { this != 0; };              //definitive constraint
constraint  max_value { this  < 256; };

reg example_reg {
    field {
        hw=rw; sw=rw;
        not_valid not0; // instantiate not_valid constraint,
                        // where "this" resolve to current field value.
        max_value max0;
    } f3 [17:31]=1;
};

enum color {
    red   = 0 { desc = " color red ";};
    green = 1 { desc = " color green ";};
    blue  = 2 { desc = " color blue ";};
};


reg register1 {
    field { hw = rw; sw = rw; }limit[0:2]=0;
    field {
        hw = rw; sw = rw;
        max_value max1;   // another instance.
    } f1 [3:9]=3;
```

```
    constraint  { f1 > limit; } min;               //anonymous constraint

    field {
        hw = rw; sw = rw; encode=color;
        constraint { this  == red || this == green; } rg1;
        constraint { this inside {red, green}; } rg2;
        constraint { this inside color; } rgb; // full range of encode
    } f2 [10:31];
};

struct RGB {
  longint unsigned red1;
  longint unsigned green1;
  longint unsigned blue1;
} ;

reg regfoo {
    RGB pixelvalue;
};

addrmap constraint_component_example {
    example_reg reg1;
    example_reg reg2;
    register1 r1;
    register1 r2;

    constraint {
        reg1.f3 inside {5,8,[9:12],3};  // adding a new constraint by
                                        // hierarchically referring to a field
    } cont1;

r1.min->constraint_disable = true;   // disable a particular constraint
// disable all constraints on f2 individually
r2.f2.rg1->constraint_disable  = true;
r2.f2.rg2->constraint_disable  = true;
r2.f2.rgb->constraint_disable  = true;

regfoo my_struct;
    constraint {
        my_struct.pixelvalue.blue1 == my_struct.pixelvalue.red1;
    } limit;
};
```

15. User-defined properties

User-defined properties enable the creation of custom properties that extend the structural component types in a SystemRDL design. A *user-defined property* specifies one or more structural **component** types (e.g., **reg**) to which it can be bound, has a single value-type (e.g., **ref**), and (optionally) a **default** value. Unlike built-in properties, user-defined properties are not automatically bound to every instance of the specified component's type; instead they need to be bound per instance and/or component definition.

15.1 Defining user-defined properties

A *user-defined property definition* appears as follows.

property *property_name* **{***attribute***;** [*attribute***;**]*__**};**__

where

a) *property_name* specifies the new property.

b) *attribute*s are specified as `attribute=value` pairs, e.g., `type=number` (see 5.1.3.1). Component attribute values can also be combined by using the | symbol.

c) *attribute*s may be specified in any order.

Table 30 specifies which attributes can be set in a user-defined property.

Table 30—Attributes for user-defined properties

Attribute	Description	Allowable values	Required
component	The structural component type with which the property is associated. This attribute shall be one or more of the allowable values. If more than one value is specified the | operator (inclusive `OR`) is used.	**field**, **reg**, **regfile**, **addrmap**, **mem**, **signal**, **constraint**, or **all**.	Yes
type	The type of the property. This may also be arrayed by adding `[]` after the **type**'s name.	See Table 31.	Yes
default	The default value for the property.	Optional; needs to match the **type** of the property. Properties of **type ref** or any component type shall not reference an instance of a parameterized type as a **default**.	No
constraint	Additional constraints on the property's value. Currently limited to **componentwidth** for type **bit**.	**componentwidth**.	No

Table 31 details each of the possible user-defined property types.

Table 31—User-defined types

Type	Description	Example
number	A bit value (see Table 7). Used for backward compatibility.	`0x10` or `8'h8c`
string	Any valid string (see Table 7).	`"Some String"`
boolean	A two-state value (see Table 7).	`true` or `false`
ref	A reference to a component instance (see Table 7).	`chip.block.reg.field`

Table 31—User-defined types (Continued)

Type	Description	Example
bit	An unsigned integer with the value of `0` or a Verilog-style number (see Table 7).	`8'h8c, 4'b1010`
longint unsigned	A 64-bit unsigned long integer (see Table 7).	`0x10, 256`
enumerator	An enumerator from a user-defined enumeration.	`myEnum::VAL_2`
struct literal	A **struct** instance consistent with the given type (see Table 7).	`'myStruct{foo: 8'h10, bar: '{"hello", "world"} }`
type[]	An array of values whose type shall be picked from this table, excepting array.	`'{ 8'h80, 8'h8F, 8'h08, 8'h0F }`
addrmap, **regfile**, **reg**, **mem**, **field**	A specialization of the **ref** keyword to instances of given type.	`submap` or `submap.regA`

15.1.1 Semantics

a) User-defined properties are global and defined in the root scope.

b) A user-defined property definition shall include the **component** property specification.

c) A user-defined property definition shall include its *type* definition (see Table 31).

d) The **default** attribute can result in some inconsistencies relative to the behavior of built-in properties to the language, especially relating to **boolean** properties. Built-in *booleans* in SystemRDL are inherently defaulted to **false**. With user-defined **boolean** properties, the **default** can be specified to be **true** or **false**. A **default** of **true** creates an inconsistency with respect to SystemRDL property assignments.

e) The **default** value shall be *assignment compatible* with the property **type**, as defined in 6.4.

f) Field values shall not require more bits than are available in the **field**.

g) If **constraint** is set to **componentwidth**, the assigned value of the property shall not have a value of `1` for any bit beyond the width of the field.

15.1.2 Example

This example defines several user-defined properties.

```
property a_map_p { type = string; component = addrmap | regfile; };
property some_bool_p { type = boolean; component = field; default = false; };
```

15.2 Assigning (and binding) user-defined properties

User-defined properties may be assigned like general properties (see 5.1.3).

A user-defined property is bound when it is instantiated within a component definition or assigned a value.

15.2.1 Semantics

a) User-defined properties can be dynamically assigned to any component in its **component** attribute.

b) It shall be an error if there is an attempt to assign a user-defined property in a component that is not specified in its **component** attribute.

c) User-defined properties can be bound to a component without setting a value.

d) If a user-defined property has an enumeration type, any value assignment to the property shall be one of the named enumerators of that enumeration type.

15.2.2 Examples

These examples show the definition and assignment of several user-defined properties.

Example 1

```
property a_map_p { type = string; component = addrmap | regfile; };
property some_bool_p { type = boolean; component = field; default = false; };
property some_ref_p { type = ref; component = all; };
property some_num_p { type = number; default = 0x10; component = field | reg
    | regfile };

addrmap foo {
  reg{
    field { some_bool_p; } field1;  // Attach some_bool_p to the field
                                    // with a value of false;

    field { some_bool_p = true; some_num_p; } field2;
   // Attach some_bool_p to the field with a value of true;
    field1->some_ref_p = field2; // create a reference
    some_num_p = 0x20; // Assign this property to the reg and give it value
  } bar;

  a_map_p; // The property has been bound to the map but it has not been
          // assigned a value so its value is undefined
};
```

Example 2

```
enum myEncoding {
   alpha = 1'b0;
   beta = 1'b1;
};

property my_enc_prop {
   type = myEncoding;
   component = field;
   default = beta;
};

addrmap top {
   reg {
      field { my_enc_prop = alpha ; } f ;
   } regA ;
} ;
```

16. Preprocessor directives

SystemRDL provides for file inclusion and text substitution through the use of preprocessor directives. There are two phases of preprocessing in SystemRDL: embedded Perl preprocessing and a more traditional Verilog-style preprocessor. The embedded Perl preprocessing is handled first and the resulting substituted code is passed through a traditional Verilog-style preprocessor.

16.1 Embedded Perl preprocessing

The SystemRDL preprocessor provides more power than traditional macro-based preprocessing without the dangers of unexpected text substitution. Instead of macros, SystemRDL allows designers to embed snippets of Perl code into the source.

16.1.1 Semantics

a) Perl snippets shall begin with **<%** and be terminated by **%>**; between these markers any valid Perl syntax may be used.

b) Any SystemRDL code outside of the Perl snippet markers is equivalent to the Perl `print 'RDL code'` and the resulting code is printed directly to the post-processed output.

c) **<%=$VARIABLE%>** (no whitespace is allowed) is equivalent to the Perl `print $VARIABLE`.

d) The resulting Perl code is interpreted and the result is sent to the traditional Verilog-style preprocessor.

16.1.2 Example

This example shows the use of the SystemRDL preprocessor.

```
// An example of Apache's ASP standard for embedding Perl
reg myReg {

   <% for( $i = 0; $i < 6; $i += 2 ) { %>
   myField data<%=$i%> [<%=$i+1%>:<%=$i%>];
   <% } %>
};
```

When processed, this is replaced by the following.

```
// Code resulting from embedded Perl script
reg myReg {
   myField data0 [1:0];
   myField data2 [3:2];
   myField data4 [5:4];
};
```

16.2 Verilog-style preprocessor

SystemRDL also provides for file inclusion and text substitution through the use of Verilog-style preprocessor directives. A SystemRDL file containing *file inclusion directives* shall be equivalent with one containing each included file in-lined at the place of its inclusion directive. A SystemRDL file containing a *text substitution directive* shall be equivalent to one containing the text resolved according to the text substitution directive in-lined at the place of the text inclusion directive.

The Verilog-style preprocessing always takes any embedded Perl preprocessing output as its source.

16.2.1 Verilog-style preprocessor directives

These directives are a subset of those defined by the SystemVerilog (IEEE Std 1800™) and Verilog (IEEE Std 1364™) standards to allow SystemRDL source files to include other files and provide protection from definition collisions due to the multiple inclusions of a file. The text macro define directives are defined by the SystemVerilog standard and the other directives are defined by the Verilog standard.

Table 32 shows which preprocessor are included in SystemRDL.

Table 32—Verilog-style preprocessor directives

Directive	Defining standard	Description
`` `define ``	SystemVerilog	Text macro definition
`` `if ``	Verilog	Conditional compilation
`` `else ``	Verilog	Conditional compilation
`` `elsif ``	Verilog	Conditional compilation
`` `endif ``	Verilog	Conditional compilation
`` `ifdef ``	Verilog	Conditional compilation
`` `ifndef ``	Verilog	Conditional compilation
`` `include ``	Verilog	File inclusion
`` `line ``	Verilog	Source filename and number
`` `undef ``	Verilog	Undefine text macro

All other directives defined by the SystemVerilog and Verilog standards are removed during preprocessing, i.e., `` `begin_keywords ``, `` `celldefine ``, `` `default_nettype ``, `` `end_keywords ``, `` `endcelldefine ``, `` `nounconnected_drive ``, `` `pragma ``, `` `resetall ``, `` `timescale ``, and `` `unconnecteded_drive ``.

SystemRDL does not support the SystemVerilog predefined `include` files or the SystemVerilog or Verilog languages beyond the directives given in Table 32.

16.2.2 Limitations on nested file inclusion

Nested includes are allowed, although the following restrictions are placed on this.

a) The number of nesting levels for include files shall be bounded.

b) Implementations may limit the maximum number of levels to which include files can be nested, but the limit shall be at least 15 levels.

17. Advanced topics in SystemRDL

The concept of signals was introduced in Clause 8 and the signal properties were described in 8.2. Signals, in addition to providing a means of interconnecting components in SystemRDL, have a very critical role in controlling resets for generated hardware. This clause explains the advanced signal properties (see Table 10) and their application.

17.1 Application of signals for reset

There are many ways to do resets in hardware and this is where the advanced use of signals applies. Signal components have properties, such as **sync**, **async**, **activelow**, and **activehigh**, which are used to describe the use behavior of the signal, but when that signal is specified as a reference to the **resetsignal** property of a field then they affect the field's reset behavior as well. A signal does not become a reset signal until a signal instance is referenced by a field's **resetsignal** property. The following signal properties can also be used to accommodate more complex scenarios.

a) The **field_reset** property specifies all fields in the address map shall be reset by the signal to which this property is attached unless the **field** instance has a **resetsignal** property specified. This property cannot be specified on more than one signal instance in an address map and the address map's non-address map instances. This does not mean, however, that all fields then have to be reset by this signal. The user can still use the **resetsignal** property to override the default for specific fields. See c for priority and propagation rules.

b) The **cpuif_reset** property specifies the reset for the CPU interface to which the registers are connected. The designer may wish to be able to reset the CPU interface/bus while retaining the values of the registers. In the default case, the fields and the CPU interface/bus are both reset by the default signal. This property gives the designer the ability to customize such behavior. This property cannot be specified on more than one signal instance in an address map and its address maps non-address map instances.

   cpuif_reset is inherited from the enclosing lexical scope.

c) The value of a field component with a specified reset value is updated to this value or a field with write-once access is returned to a writable state when the field component reset signal is asserted. The field component reset signal is specified with the following priority (highest priority first).

   1) The dynamic assignment to the field instance **resetsignal** property.
   2) The **resetsignal** property assignment within the **field** component declaration.
   3) The default **resetsignal** property assignment inherited from the **field** component declaration's enclosing scope.
   4) The **signal** component with the **field_reset** property within the same scope as the **field** type definition.
   5) The **signal** component with the **field_reset** property following the scoping rules from the `field` instantiation.
   6) The implementation port specified by the CPU interface bus protocol as the reset signal.
   7) An implementation-dependent port or net.

The following examples highlight two different ways to customize reset behavior.

Example 1

This example shows usage of **resetsignal**.

```
addrmap top{
   signal { activelow; async; } reset_l; // Define a single bit signal
   reg {
```

```
        field {} field1=0; // This field is reset by the default IMPLIED reset
                  // signal which is named RESET and is activehigh and sync
        field {
           resetsignal = reset_l;
        } field2=0;  // This field is now reset by reset_l and the generated
                 // flops will be active low and asynchronously reset.
} some_reg_inst;
};
```

Here the **resetsignal** property is used to customize the reset behavior. Although this approach is always valid, it can be cumbersome if a user wishes to vary from the default significantly with a large number of fields. In those cases, **field_reset** and **cpuif_reset** can be used to accommodate those more complex scenarios, as shown in *Example 2*.

Example 2

This example shows usage of **cpuif_reset** and **field_reset** from the `PCI Type 0 Config Header`.

```
signal  {
     name="PCI Soft Reset";
     desc="This signal is used to issue a soft reset to the PCI(E) device";
     activelow;   // Define this signal is active low
     async;       // define this reset type is asynchronous
     field_reset; // define this signal to reset the fields by default

     // This signal will be hooked to registers PCI defines as NOT Sticky.
     // This means they will be reset by this signal.
} pci_soft_reset;

  signal  {
     name="PCI Hard Reset";
     desc="This signal the primary hard reset signal for the PCI(E) device";
     async;       // define this reset type is asynchronous
     activelow;   // Define this signal as active low
     cpuif_reset; // This signal will be or'd with the PCI Soft Reset Signal
                  // to form the master hard reset which will reset all flops.
                 // The soft reset signal above will not reset flops that PCI
                  // defines as STICKY.
  } pci_hard_reset;

reg {                                                  // PCIE_REG_BIST
    name = "BIST";
    desc = "This optional register is used for control and status of BIST.
           Devices that do not support BIST always returns a value of 0
           (i.e., treat it as a reserved register). A device whose
          BIST is invoked shall not prevent normal operation of the PCI bus.
           Figure 6-4 shows the register layout and Table 6-3 describes the
            bits in the register.";
    regwidth = 8;

    field {
      name = "cplCode";
     desc = "A value of 0 means the device has passed its test. Non-zero values
           mean the device failed. Device-specific failure codes can be encoded
              in the non-zero value.";
      hw = rw; sw = r;
      fieldwidth = 4;
   } cplCode [3:0];// since this signal has no resetsignal property it defaults
```

```
                    // to using the signal with field reset which is
                    // the pci_soft_reset signal


      field {
        name = "start";
        desc = "Write a 1 to invoke BIST. Device resets the bit when BIST is
               complete. Software should fail the device if BIST is not complete
                after 2 seconds.";
        hw = rw; sw = rw;
        fieldwidth = 1;
      } start [6:6]; // resetsignal is also pci_soft_reset

      field {
        name = "capable";

  desc = "Return 1 if device supports BIST. Return 0 if the device is not BIST
                capable.";
        hw = rw; sw = rw;
        fieldwidth = 1;
        resetsignal = pci_hard_reset;
     } capable [7:7]=0; // resetsignal is explicitly specified as pci_hard_reset

   } PCIE_REG_BIST;
```

17.2 Understanding hierarchical interrupts in SystemRDL

SystemRDL also provides the capability to create a hierarchy of interrupts. This can be useful for describing a complete interrupt tree of a design (see Figure 2).

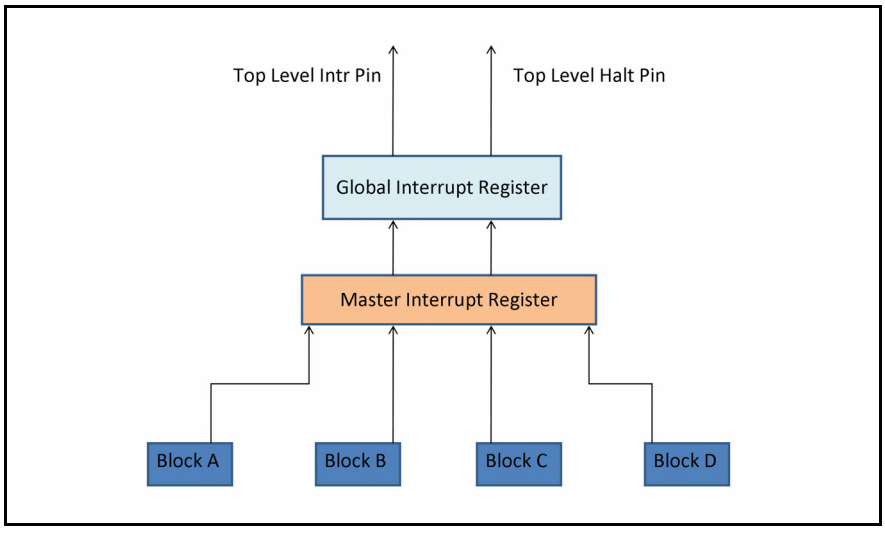

Figure 2—Hierarchical interrupt structure

Within each level of the hierarchical description, interrupt registers and enable registers can be used to gate the propagation of interrupts. The detailed diagram for a block depicted in the hierarchy shown in Figure 2 is represented by the example shown in Figure 3.

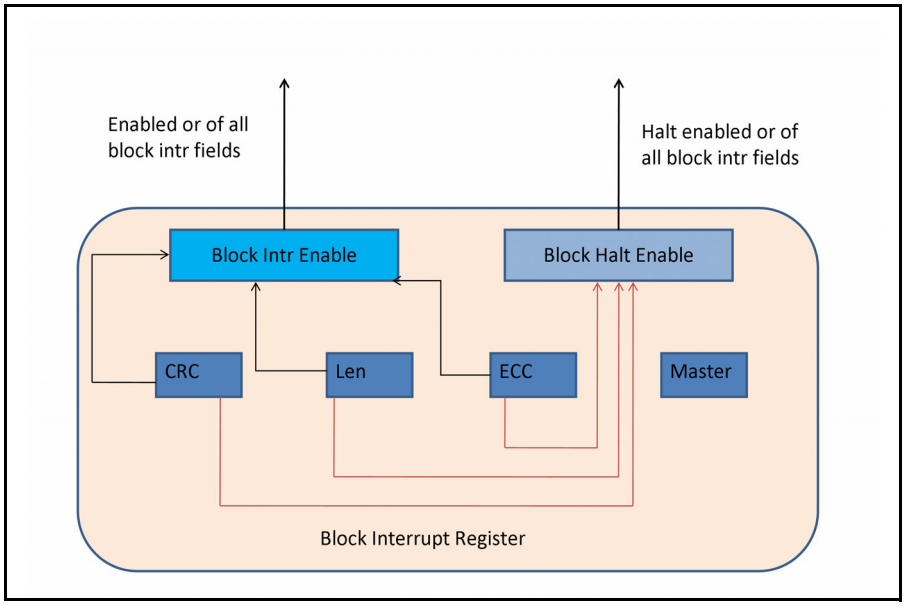

Figure 3—Block interrupt example

Multiple levels of hierarchy are needed to effectively demonstrate this interrupt tree. The example shown in the following subclauses is quite long (and broken into multiple code segments), but tries to show the use of the interrupt constructs in a practical application.

17.2.1 Example structure and perspective

The (example) SystemRDL code needed to match the hierarchical interrupt structure shown in Figure 2 needs to contain four leaf blocks. Each of these leaf blocks needs to contain three interrupt events. These lowest level events are **stickybit** and the `OR` of the three interrupts propagates that interrupt to the next level in the tree. This `OR`'d output indicates some block in the design actually has a interrupt pending. Finally, the four blocks are aggregated to create a single interrupt pin. Enables are used throughout this example, but it could just as easily be a mask instead.

This example is broken into sections to make it more readable. The previous description and example are built from a bottom-up perspective.

Considering this example from a software driver's viewpoint (from the top down), there are two top-level signals that are emitted to software: one indicates a interrupt of some priority has occurred; the other indicates an interrupt of another priority has occurred. These could map to fatal and non-fatal interrupts or anything else the user desires. For each level on the tree, there is enabling so the software can easily **disable**/**enable** these interrupts at each level of the tree.

So the software begins the process by seeing if an **intr** or **halt** is set in the top-level register. Once that has been determined, the software needs to read the master interrupt register and determine in which block(s) the interrupt has occurred. Once that has been determined, the leaf interrupt for that block can be read to determine which specific interrupt bits have been set. The software can then address the leaf interrupts and clear them when appropriate. Since the master level and global level are defined as **nonsticky** in this example, the software only needs to clear the leaf and then the next two levels of the tree will clear themselves automatically.

17.2.2 Code snippet 1

The first code snippet section defines a basic block's interrupt register, which contains three single-bit interrupts. It also has a single multi-bit **sticky** field used for capturing the cause of the multi-bit error correcting code interrupt. These interrupt events are created by hardware and cleared by software. The software then needs to do a write one to clear. Notice how the **default** keyword is used to reduce the size of the code.

```
//-----------------------------------------------------------
// Block Level Interrupt Register
//-----------------------------------------------------------

reg block_int_r {
  name = "Example Block Interrupt Register";
  desc = "This is an example of an IP Block with 3 int events. 2 of these
          are non-fatal and the third event multi_bit_ecc_error is fatal";

  default hw=w;   // HW can Set int only
  default sw=rw;  // SW can clear
  default woclr;  // Clear is via writing a 1

  field {
    desc = "A Packet with a CRC Error has been received";
    level intr;
  } crc_error = 0x0;

  field {
    desc = "A Packet with an invalid length has been received";
           level intr;
  } len_error = 0x0;

  field {
    desc="An uncorrectable multi-bit ECC error has been received";
    level intr;
  } multi_bit_ecc_error = 0 ;

  field {
    desc="Master who was active when ECC Error Occurred";
    sticky;
  } active_ecc_master[7:4] = 0; // Example of multi-bit sticky field
                                // This field is not an intr
};  // End of Reg: block_int_r
```

17.2.3 Code snippet 2

This next code snippet only defines the enable register associated with the interrupt register from 17.2.2—it does not instantiate the register or connect it up at this point.

```
reg block_int_en_r {
  name = "Example Block Interrupt Enable Register";
  desc = "This is an example of an IP Block with 3 int events";

  default hw=na;  // HW can't access the enables
  default sw=rw;  // SW can control them

  field {
    desc = "Enable: A Packet with a CRC Error has been received";
  } crc_error = 0x1;
```

```
  field {
    desc = "Enable: A Packet with an invalid length has been received";
  } len_error = 0x1;

  field {
    desc = "Enable: A multi-bit error has been detected";
  } multi_bit_ecc_error = 0x0;
};  // End of Reg: block_int_en_r
```

17.2.4 Code snippet 3

This next code snippet only defines a second-priority enable register associated with the interrupt register from 17.2.2—it does not instantiate the register or connect it up at this point.

```
reg block_halt_en_r {
  name = "Example Block Halt Enable Register";
  desc = "This is an example of an IP Block with 3 int events";

  default hw=na; // HW can't access the enables
  default sw=rw; // SW can control them

  field {
    desc = "Enable: A Packet with a CRC Error has been received";
  } crc_error = 0x0; // not a fatal error do not halt

  field {
    desc = "Enable: A Packet with an invalid length has been received";
  } len_error = 0x0; // not a fatal error do not halt

  field {
    desc = "Enable: A Packet with an invalid length has been received";
  } multi_bit_ecc_error = 0x1; // fatal error that will
                               // cause device to halt
}; // End of Reg: block_halt_en_r
```

17.2.5 Code snippet 4

This next code snippet defines the next level up interrupt register (called the *master interrupt register*). Each of the outputs of the leaf block's interrupt registers will connect into this block later. This section is made **nonsticky**, so the leaf interrupts are automatically cleared by this register.

```
//-------------------------------------------------------------
// Master Interrupt Status Register
//-------------------------------------------------------------

reg master_int_r {
  name = "Master Interrupt Status Register";
  desc = "This register contains the status of the 4 lower Module interrupts.
          Also an interrupt signal (myMasterInt) is generated which is the 'OR'
          of the four Module interrupts. A Halt signal is also generated which
          represents the bitwise or the masked/enabled halt bits";

  default nonsticky intr; // Unless we want to have to clear this separately
                          // from the leaf intr this should be non sticky
  default hw=w; // HW normally won't want to access this but it could
  default sw=r; // Software can just read this. It clears the leaf intr's
                // to clear this
```

```
  field {
    desc = "An interrupt has occurred with ModuleD.
            Software must read the ModuleD Master Interrupt Register
            in order to determine the source of the interrupt.";
  } module_d_int[3:3] = 0x0;

  field {
    desc = "An interrupt has occurred with ModuleC.
            Software must read the ModuleC Master Interrupt Register
            in order to determine the source of the interrupt.";
  } module_c_int[2:2] = 0x0;

  field {
    desc = "An interrupt has occurred with ModuleB.
            Software must read the ModuleB Interrupt Register
            in order to determine the source of the interrupt.";
} module_b_int[1:1] = 0x0;

field {
  desc = "An interrupt has occurred with ModuleA.
          Software must read the ModuleA Master Interrupt Register
          in order to determine the source of the interrupt.";
  } module_a_int[0:0] = 0x0;
};
```

17.2.6 Code snippet 5

This next code snippet defines the enable register for the master interrupt register set in 17.2.5.

```
//
// The following is the accompanying enable register. Since the combinatorial
// logic for processing the interrupt is internal to the generated verilog,
// there's no need for an external port - which is realized by assigning "na"
// to the hw attribute of the specific field.  This could have been defined as
// a mask register just as easily...
//

//------------------------------------------------------------
// Interrupt Enable Register
//------------------------------------------------------------

reg master_int_en_r {
  name = "Master Interrupt Enable Register";
  desc = "Configurable register used in order to enable the corresponding
          interrupts found in myMasterInt register.";

  default hw = na;
  default sw = rw;

  field {
    desc = "Interrupt enable for ModuleD Interrupts. 1 = enable, 0 = disable";
  } module_d_int_en[3:3] = 0x0;

  field {
    desc = "Interrupt enable for ModuleC Interrupts. 1 = enable, 0 = disable";
  } module_c_int_en[2:2] = 0x0;
```

```
  field {
    desc = "Interrupt enable for ModuleB Interrupts. 1 = enable, 0 = disable";
  } module_b_int_en[1:1] = 0x0;

  field {
    desc = "Interrupt enable for ModuleA Interrupts. 1 = enable, 0 = disable";
  } module_a_int_en[0:0] = 0x0;
};
```

17.2.7 Code snippet 6

This next code snippet defines an alternate enable register for the master interrupt register set in 17.2.5.

```
//-----------------------------------------------------------
// Halt Enable Register
//-----------------------------------------------------------

// The halt en is another enable or mask that could be used to generate an
// alternate signal like a halt that represents a fatal error in the system or
// some other event NOTE: It does not have to mean fatal as the name implies
// its just another priority level for interrupts...

reg master_halt_en_r {
  name = "Master Halt Enable Register";
  desc = "Configurable register used in order to enable the corresponding
          interrupts found in myMasterInt register.";

  default hw = na;
  default sw = rw;

  field {
    desc = "Halt enable for ModuleD Interrupts. 1 = enable, 0 = disable";
  } module_d_halt_en[3:3] = 0x0;

  field {
    desc = "Halt enable for ModuleC Interrupts. 1 = enable, 0 = disable";
  } module_c_halt_en[2:2] = 0x0;

  field {
    desc = "Halt enable for ModuleB Interrupts. 1 = enable, 0 = disable";
  } module_b_halt_en[1:1] = 0x0;

  field {
    desc = "Halt enable for ModuelA Interrupts. 1 = enable, 0 = disable";
  } module_a_halt_en[0:0] = 0x0;
};
```

17.2.8 Code snippet 7

Now, the third level up from the leaf in the interrupt tree needs to be addressed (called the *global interrupt register*). This register distills down the fact there is an interrupt present in at least one of the four blocks into a single **intr** signal and a single **halt** signal.

```
//-----------------------------------------------------------
// Global Interrupt Status Register
//-----------------------------------------------------------
```

```
// This takes the block int which feeds the master int and then distills it
// down one more level so we end up with a single bit intr and single bit halt...

//-------------------------------------------------------------
// Global Interrupt/Halt  Enable Register
//-------------------------------------------------------------

reg final_en_r {
  name = "My Final Enable Register";
  desc = "This enable allows all interrupts/halts to be suppressed
          with a single bit";

  default hw = na;
  default sw = rw;

  field {
    desc = "Global Interrupt Enable. 1 = enable, 0 = disable";
  } global_int_en = 0x0;

  field {
    desc = "Global Halt Enable. 1 = enable, 0 = disable";
  } global_halt_en = 0x0;

};

reg final_int_r {
  name = "My Final Int/Halt Register";
  desc = "This distills a lower level interrupts into a final bit than can be
          masked";
  default sw = r; // sw does not need to clear global_int
                  // (global_int is of type final_int_r)
                  // instead it clears itself when all master_int intr
                  // bits get serviced

  default nonsticky intr;
  default hw = w; // w needed since dyn assign below implies interconnect to hw
                  //   global_int.global_int->next = master_int->intr;

  field {
    desc = "Global Interrupt";
  } global_int = 0x0;

  field {
    desc = "Global Halt";
  } global_halt = 0x0;
};
```

17.2.9 Code snippet 8

Once all the components for the three-level interrupt tree have been defined, an address map needs to be defined and any previously defined components need to be instantiated and interconnected. This section does all this—it is the most critical part of the example to understand.

```
addrmap int_map_m {

  name = "Sample ASIC Interrupt Registers";
  desc = "This register map is designed how one can use interrupt concepts
   effectively in SystemRDL";
```

```
// Leaf Interrupts

  // Block A Registers

  block_int_r      block_a_int;     // Instance the Leaf Int Register
  block_int_en_r   block_a_int_en;  // Instance the corresponding Int Enable
                                    // Register
  block_halt_en_r  block_a_halt_en; // Instance the corresponding halt enable
                                    // register

  // This block connects the int bits to their corresponding
  // int enables and halt enables
  //
  block_a_int.crc_error->enable = block_a_int_en.crc_error;
  block_a_int.len_error->enable = block_a_int_en.len_error;
  block_a_int.multi_bit_ecc_error->enable =
   block_a_int_en.multi_bit_ecc_error;

  block_a_int.crc_error->haltenable = block_a_halt_en.crc_error;
  block_a_int.len_error->haltenable = block_a_halt_en.len_error;
  block_a_int.multi_bit_ecc_error->haltenable =
   block_a_halt_en.multi_bit_ecc_error;
```

17.2.10 Code snippet 9

17.2.9 instances the leaf interrupt, instances its enable and halt enable, and assigns **enable** and **haltenable** properties to reference the respective enable registers. This code snippet repeats this process three more times: one each for blocks `b`, `c`, and `d`.

```
  // Block B Registers

  block_int_r      block_b_int  @0x100;
  block_int_en_r   block_b_int_en;
  block_halt_en_r  block_b_halt_en;

  block_b_int.crc_error->enable = block_b_int_en.crc_error;
  block_b_int.len_error->enable = block_b_int_en.len_error;
  block_b_int.multi_bit_ecc_error->enable =
   block_b_int_en.multi_bit_ecc_error;

  block_b_int.crc_error->haltenable = block_b_halt_en.crc_error;
  block_b_int.len_error->haltenable = block_b_halt_en.len_error;
  block_b_int.multi_bit_ecc_error->haltenable =
   block_b_halt_en.multi_bit_ecc_error;


  // Block C Registers

  block_int_r      block_c_int @0x200;
  block_int_en_r   block_c_int_en;
  block_halt_en_r  block_c_halt_en;

  block_c_int.crc_error->enable = block_c_int_en.crc_error;
  block_c_int.len_error->enable = block_c_int_en.len_error;
  block_c_int.multi_bit_ecc_error->enable =
   block_c_int_en.multi_bit_ecc_error;
```

```
block_c_int.crc_error->haltenable = block_c_halt_en.crc_error;
block_c_int.len_error->haltenable = block_c_halt_en.len_error;
block_c_int.multi_bit_ecc_error->haltenable =
 block_c_halt_en.multi_bit_ecc_error;

// Block D Registers

block_int_r     block_d_int @0x300;
block_int_en_r  block_d_int_en;
block_halt_en_r block_d_halt_en;

block_d_int.crc_error->enable = block_d_int_en.crc_error;
block_d_int.len_error->enable = block_d_int_en.len_error;
block_d_int.multi_bit_ecc_error->enable =
 block_d_int_en.multi_bit_ecc_error;

block_d_int.crc_error->haltenable = block_d_halt_en.crc_error;
block_d_int.len_error->haltenable = block_d_halt_en.len_error;
block_d_int.multi_bit_ecc_error->haltenable =
 block_d_halt_en.multi_bit_ecc_error;
```

17.2.11 Code snippet 10

This code snippet instances the master interrupt register and its associated enables. The interesting part of this section is how the leaf register's **intr** property (which represents the `OR` of all the interrupts in the leaf register) are connected together.

```
//
// Master Interrupts
//

master_int_r              master_int              @0x01000;
master_int_r              master_halt                     ;
master_int_en_r           master_int_en                   ;
master_halt_en_r          master_halt_en                  ;

// Associate the INT's with the EN's
master_int.module_d_int->enable = master_int_en.module_d_int_en;
master_int.module_c_int->enable = master_int_en.module_c_int_en;
master_int.module_b_int->enable = master_int_en.module_b_int_en;
master_int.module_a_int->enable = master_int_en.module_a_int_en;
// Associate the HALT's with the EN's
master_halt.module_d_int->haltenable = master_halt_en.module_d_halt_en;
master_halt.module_c_int->haltenable = master_halt_en.module_c_halt_en;
master_halt.module_b_int->haltenable = master_halt_en.module_b_halt_en;
master_halt.module_a_int->haltenable = master_halt_en.module_a_halt_en;

// Now hook the lower level leaf interrupts to the higher level interrupts

// This connects the Implicit Or from Block A's INT reg after
// masking/enable to the next level up (master)
master_int.module_a_int->next = block_a_int->intr;


// This connects the Implicit Or from Block B's INT reg after
// masking/enable to the next level up (master)
master_int.module_b_int->next = block_b_int->intr;
```

```
    // This connects the Implicit Or from Block C's INT reg after
    // masking/enable to the next level up (master)
    master_int.module_c_int->next = block_c_int->intr;

    // This connects the Implicit Or from Block D's INT reg after
    // masking/enable to the next level up (master)
    master_int.module_d_int->next = block_d_int->intr;

    // This connects the Implicit Or from Block A's HALT reg after
    // masking/enable to the next level up (master)
    master_halt.module_a_int->next = block_a_int->halt;

    // This connects the Implicit Or from Block B's HALT reg after
    // masking/enable to the next level up (master)
    master_halt.module_b_int->next = block_b_int->halt;

    // This connects the Implicit Or from Block C's HALT reg after
    // masking/enable to the next level up (master)
    master_halt.module_c_int->next = block_c_int->halt;

    // This connects the Implicit Or from Block D's HALT reg after
    // masking/enable to the next level up (master)
    master_halt.module_d_int->next = block_d_int->halt;
```

17.2.12 Code snippet 11

This final section of the example instantiates a single top-level interrupt register containing a single **intr** and a single **halt** signal. This constitutes the final resolved interrupt that has been fully masked/enabled throughout the tree.

```
    final_int_r     global_int    @0x1010;
    // Inst the global int/halt register

    final_en_r      global_int_en @0x1014;
    // Inst the global int/halt enable register

    global_int.global_int->enable = global_int_en.global_int_en;
    // Associate the INT with the EN

    global_int.global_halt->haltenable = global_int_en.global_halt_en;
    // Associate the HALT with the EN

    global_int.global_int->next = master_int->intr;
    // Take the or of the 4 blocks in the master
    // Int and create one final interrupt

    global_int.global_halt->next = master_halt->halt;
    // Take the or of the 4 blocks in the master
    // Int and create one final halt

};
```

17.3 Understanding bit ordering and byte ordering in SystemRDL

Bit ordering and byte ordering are common source of confusion for many engineers. This subclause discusses the bit ordering and byte ordering rules in SystemRDL and also illustrates their use with some examples.

17.3.1 Bit ordering

The most common bit ordering is called **lsb0**. This is demonstrated in the scheme below, where the least significant bit is `0`.

Bit: `76543210`

Value: `10010110` (decimal 150)

The alternative scheme is called **msb0**. This is demonstrated in the scheme below, where the least significant bit is `7`.

Bit: `01234567`

Value: `10010110` (decimal 150)

In SystemRDL, a user can define address maps using both conventions, but a single **addrmap** needs to have homogeneous **lsb0** or **msb0** descriptions. The compiler shall determine **lsb0** and **msb0** when explicit indices for a register are defined, e.g., `[0:7]`, but it is not possible to determine the bit order when the first field uses implicit indices and leaves the choice of assigning final indexes to the compiler.

Example 1

In this case, the first field is explicit and defines the map as **msb0**, therefore no **explicit** keyword is needed.

```
addrmap some_map {
  reg {
    field {}f1[12:19] = 8'b10010110;
       // In this example its clear the compiler should use
       // msb0 mode as the first field infers this by its
       // use of explicit indices. Register bit 12 is reset to a 1.

    field {}f2[4] = 4'b1010;
       // f2 is from register bits 8 to 11
       // reset value of bit 8 is 1, bit 9 is 0,
       // bit 10 is 1, and bit 11 is 0
  } reg1;
};
```

Example 2

In this case, the first field is implicit and the compiler needs to see a keyword to decide bit ordering.

```
addrmap some_map {
  lsb0;
  reg {
    field {}f1[8] = 8'b10010110;
     // In this example the compiler can't tell if it's [7:0] or [0:7]
     // without the lsb0 keyword above.
     // It could be either bit order.
     // Here register bit 0 is reset to a 0.
    field {}f2[4] = 4'b1010;
  } reg1;
       // f2 is from register bits 11 to 8
       // reset value of bit 8 is 0, bit 9 is 1,
       // bit 10 is 0, and bit 11 is 1
};
```

Example 3

In this case, the first field is implicit and the compiler needs to see a keyword to decide bit ordering.

```
addrmap some_map {
  msb0;
  reg {
    field {}f1[8] = 8'b10010110;
       // In this example the compiler can't tell if it's [7:0] or [24:31]
       // without the msb0 keyword above.
       // The msb0 keyword implies it's from 24 to 31.
       // Here register bit 24 is reset to a 1.
    field {}f2[4] = 4'b1010;
       // f2 is from register bits 20 to 23
       // reset value of bit 20 is 1, bit 21 is 0,
       // bit 22 is 1, and bit 23 is 0
  } reg1;
};
```

17.3.2 Byte ordering

Byte ordering is another common source of confusion. Byte order is often called *endianness* (see [B2]). In SystemRDL, two properties are defined for dealing with this: **bigendian** and **littleendian**. These properties do nothing related to the structure of SystemRDL, but they provide information to back-end generators which are generating bus interfaces. Therefore, these properties are only attached to **addrmap** blocks since they define the boundary of a generatable RTL module. SystemRDL's smallest endian or atomic unit is a byte and the data unit on which the endianness is performed is controlled by the **accesswidth** property. The following example uses a 64-bit register with a 32-bit accesswidth, where the words are ordered in a big endian fashion (per convention) and the bytes are ordered as shown.

Example

If `0x0123456789ABCDEF` is assigned a base address of `0x800`,

a **bigendian** bus would address the bytes as:

```
800 801 802 803 804 805 806 807
01  23  45  67  89  AB  CD  EF
```

a **littleendian** bus would address the bytes as:

```
800 801 802 803 804 805 806 807
67  45  23  01  EF  CD  AB  89
```

Thus, these two properties do not affect bit ordering in a SystemRDL file; instead, they correspond to byte ordering on output generators.

Annex A

(informative)

Bibliography

[B1] IEEE 100, *The Authoritative Dictionary of IEEE Standards Terms,* Seventh Edition. New York: Institute of Electrical and Electronics Engineers, Inc.

[B2] *Endianness References*, see http://en.wikipedia.org/wiki/Endianness and http://3bc.bertrand-blanc.com/endianness05.pdf.

Annex B

(normative)

Grammar

The formal syntax of SystemRDL is described using Backus-Naur Form (BNF). The syntax of SystemRDL source is derived from the starting symbol `root`. If there is a conflict between a grammar element shown anywhere in this Standard and the material in this annex, the material shown in this annex shall take precedence.

The full syntax and semantics of SystemRDL are not described solely using BNF. The normative text description contained within the clauses and annexes of this standard provide additional details on the syntax and semantics described in this BNF.

B.1 SystemRDL source text

```
root ::= { description }
description ::=
      component_def
    | enum_def
    | property_definition
    | struct_def
    | constraint_def
    | explicit_component_inst
    | property_assignment
```

B.2 User-defined properties

```
property_definition ::= property id { property_body } ;
property_body ::= property_attribute { property_attribute }
property_attribute ::=
    property_type
  | property_usage
  | property_default
  | property_constraint
property_type ::= type = property_data_type [ array_type ] ;
property_data_type ::=
    component_primary_type
  | ref
  | number
  | basic_data_type
property_usage ::= component = property_comp_types ;
property_comp_types ::= property_comp_type { | property_comp_type }
property_comp_type ::=
    component_type
  | constraint
  | all
property_default ::= default = constant_expression ;
property_constraint::= constraint = property_constraint_type ;
property_constraint_type::= componentwidth
```

B.3 Component definition

```
component_def ::=
     component_named_def component_inst_type component_insts ;
   | component_anon_def component_inst_type component_insts ;
   | component_named_def [ component_insts ] ;
   | component_anon_def component_insts ;
   | component_inst_type component_named_def component_insts ;
   | component_inst_type component_anon_def component_insts ;
component_named_def::= component_type id [ param_def ] component_body
component_anon_def::= component_type component_body
component_body ::= { { component_body_elem } }
component_body_elem ::=
    component_def
  | enum_def
  | struct_def
  | constraint_def
  | explicit_component_inst
  | property_assignment
component_type ::=
    component_primary_type
  | signal
component_primary_type ::= addrmap | regfile | reg | field | mem
explicit_component_inst ::= [ component_inst_type ] [ component_inst_alias ]
   id component_insts ;
component_insts ::= [ param_inst ] component_inst { , component_inst }
component_inst ::=
    id [ component_inst_array_or_range ]
    [ = constant_expression ]
    [ @ constant_expression ]
    [ += constant_expression ]
    [ %= constant_expression ]
component_inst_alias ::= alias id
component_inst_type ::= external | internal
component_inst_array_or_range ::=
    array { array }
  | range
```

B.4 Struct definitions

```
struct_def ::= [ abstract ] struct id [ : id ] struct_body ;
struct_body ::= { { struct_elem } }
struct_elem ::= struct_type id [ array_type ] ;
struct_type ::=
    data_type
  | component_type
```

B.5 Constraints

```
constraint_def ::=
    constraint constraint_def_exp ;
  | constraint constraint_def_anon ;
constraint_def_exp ::= id constraint_body [ constraint_insts ]
```

```
constraint_def_anon ::= constraint_body constraint_insts
constraint_insts ::= id { , id }
constraint_body ::= { { constraint_elem ; } }
constraint_prop_assignment ::= id = constant_expression
constraint_elem ::=
    constant_expression
  | constraint_prop_assignment
  | constraint_lhs inside { constraint_values }
  | constraint_lhs inside id
constraint_values ::= constraint_value { , constraint_value }
constraint_value ::=
    constant_expression
  | [ constant_expression : constant_expression ]
constraint_lhs ::=
    this
  | instance_ref
```

B.6 Parameters

```
param_def ::= # ( param_def_elem { , param_def_elem } )
param_def_elem ::= data_type id [ array_type ] [ = constant_expression ]
param_inst ::= # ( param_elem { , param_elem } )
param_elem ::= . id ( param_value )
param_value ::= constant_expression
```

B.7 Enums

```
enum_def ::= enum id enum_body ;
enum_body ::= { enum_entry { enum_entry } }
enum_entry ::= id [ = constant_expression ] [ enum_property_assignment ] ;
enum_property_assignment ::= { { explicit_prop_assignment ; } }
```

B.8 Property assignment

```
property_assignment ::=
    explicit_or_default_prop_assignment
  | post_prop_assignment
explicit_or_default_prop_assignment ::=
    [ default ] explicit_prop_modifier ;
  | [ default ] explicit_prop_assignment ;
explicit_prop_modifier ::= prop_mod id
explicit_encode_assignment ::= encode = id
explicit_prop_assignment ::=
    prop_assignment_lhs [ = prop_assignment_rhs ]
  | explicit_encode_assignment
post_encode_assignment ::= instance_ref -> encode = id
post_prop_assignment ::=
    prop_ref [ = prop_assignment_rhs ] ;
  | post_encode_assignment ;
prop_mod ::= posedge | negedge | bothedge | level | nonsticky
```

```
prop_assignment_lhs ::=
    prop_keyword
  | id
prop_keyword ::= sw | hw | rclr | rset | woclr | woset
prop_assignment_rhs ::=
    constant_expression
  | precedencetype_literal
```

B.9 Struct literal

```
struct_literal ::= id '{ struct_literal_body }
struct_literal_body ::= [ struct_literal_elem { , struct_literal_elem } ]
struct_literal_elem ::= id : constant_expression
```

B.10 Array literal

```
array_literal ::= '{ array_literal_body }
array_literal_body ::= constant_expression { , constant_expression }
```

B.11 Reference

```
instance_ref ::= instance_ref_element { . instance_ref_element }
prop_ref ::=
    instance_ref -> prop_keyword
  | instance_ref -> id
instance_or_prop_ref ::=
    instance_ref -> prop_keyword
  | instance_ref -> id
  | instance_ref
instance_ref_element ::= id { array }
```

B.12 Array and range

```
range ::= [ constant_expression : constant_expression ]
array ::= [ constant_expression ]
array_type ::= [ ]
```

B.13 Concatenation

```
constant_concatenation ::= { constant_expression { , constant_expression } }
constant_multiple_concatenation ::=
   { constant_expression constant_concatenation }
```

B.14 Data types

```
integer_type ::=
    integer_vector_type
  | integer_atom_type
```

```
integer_atom_type ::= longint
integer_vector_type ::= bit
simple_type ::= integer_type
signing ::= unsigned
basic_data_type ::=
    simple_type [ signing ]
  | string
  | boolean
  | id
data_type ::=
    basic_data_type
  | accesstype
  | addressingtype
  | onreadtype
  | onwritetype
```

B.15 Literals

```
boolean_literal ::= true | false
number ::= number as specified in 4.6
string_literal ::= string as specified in 4.5
enumerator_literal ::= id :: id
accesstype_literal ::= na | rw | wr | r | w | rw1 | w1
onreadtype_literal ::= rclr | rset | ruser
onwritetype_literal ::= woset | woclr | wot | wzs | wzc | wzt | wclr | wset | wuser
addressingtype_literal ::= compact | regalign | fullalign
precedencetype_literal ::= hw | sw
```

B.16 Expressions

```
constant_expression ::=
    constant_primary
  | unary_operator constant_primary
  | constant_expression binary_operator constant_expression
  | constant_expression ? constant_expression : constant_expression
constant_primary ::=
    primary_literal
  | constant_concatenation
  | constant_multiple_concatenation
  | ( constant_expression )
  | constant_cast
  | instance_or_prop_ref
  | struct_literal
  | array_literal
primary_literal ::=
    number
  | string_literal
  | boolean_literal
  | accesstype_literal
  | onreadtype_literal
  | onwritetype_literal
  | addressingtype_literal
  | enumerator_literal
  | this
```

```
binary_operator ::=
    &&  | || | < | > | <= | >= | == | != | >> | <<
  | &   | |  | ^ | ~^ | ^~ | * | / | % | + | - | **
unary_operator :
    ! | + | - | ~ | & | ~& | | | ~| | ^ | ~^ | ^~
constant_cast ::= casting_type '  ( constant_expression )
casting_type ::= simple_type | constant_primary | boolean
```

B.17 Identifiers

```
id ::= identifier as specified in 4.3
```

Annex C

(informative)

Backward compatibility

One of the main goals for this update to the SystemRDL specification was to maintain backward compatibility. However, in some cases, this was just not possible to achieve what needed to be done or when there were mistakes in the original SystemRDL 1.0 specification. Where the SystemRDL 2.0 grammar and the SystemRDL 1.0 specification differ, it is unclear with which to maintain compatibility. Below are known areas of incompatibility with an explanation of why and possible workarounds.

C.1 Keywords

Some of the new features require additional keywords (see Table C1). If these keywords are used in legacy code as instance names, there will now be a conflict. Refer to Table 2 for the current keyword list and Annex D for additional reserved words. Where instances in legacy code use one of these keywords or reserved words, the name either needs to be changed or escaped (by using a \, see also 4.3).

Table C1—New keywords added in SystemRDL 2.0

Feature	Keywords added
Structs	**abstract**, **struct**
Casting	**accesstype**, **addressingtype**, **onreadtype**, **onwritetype**
Data types	**bit**, **boolean**, **longint**, **ruser**, **rw1**, **string**, **unsigned**, **w1**, **wclr**, **wot**, **wr**, **wset**, **wuser**, **wzc**, **wzs**, **wzt**
User-defined properties	**component**, **componentwidth**, **number**, **ref**, **type**
Constraints	**constraint**, **inside**, **this**
Memories	**mem**

Many of the previously defined keywords are now properties (see Table C2) or obsolete (see Table C3).

Table C2—Keywords in SystemRDL 1.0 now changed to properties

accesswidth	**activehigh**	**activelow**	**addressing**	**alignment**	**anded**
async	**bigendian**	**bridge**	**counter**	**cpuif_reset**	**decr**
decrsaturate	**decrthreshold**	**decrvalue**	**decrwidth**	**desc**	**dontcompare**
donttest	**enable**	**errextbus**	**field_reset**	**fieldwidth**	**halt**
haltenable	**haltmask**	**hwclr**	**hwenable**	**hwmask**	**hwset**
incr	**incrsaturate**	**incrthreshold**	**incrvalue**	**incrwidth**	**intr**

Table C2—Keywords in SystemRDL 1.0 now changed to properties

littleendian	**lsb0**	**mask**	**msb0**	**name**	**next**
ored	**overflow**	**precedence**	**regwidth**	**reset**	**resetsignal**
rsvdset	**rsvdsetX**	**saturate**	**shared**	**sharedextbus**	**signalwidth**
singlepulse	**sticky**	**stickybit**	**swacc**	**swmod**	**swwe**
swwel	**sync**	**threshold**	**underflow**	**we**	**wel**
xored					

Table C3—Keywords in SystemRDL 1.0 now obsolete

arbiter	**clock**				

C.2 next

The SystemRDL 1.0 specification showed examples with invalid syntax for the **next** property, where a field without **->** implied `->next`, but did not specify it. This was also allowed by the grammar, but is no longer supportable. In section 7.8.2 of the SystemRDL 1.0 specification, within Example 3:

```
field counter_f { counter; };
field {} has_overflowed;
counter_f count1[5]=0; // Defines a 5 bit counter from 6 to 1
count1->incrthreshold=5'hF;
has_overflowed = count1->overflow;
```

the last line should instead be:

```
has_overflowed->next = count1->overflow;
```

SystemRDL 2.0 does not support implied `->next`.

C.3 Use of 0 size

The SystemRDL 1.0 specification was silent on the meaning of a `0` **size** field or register array. This is no longer a valid syntax. A single positive integer **size** (or for **field**s a legal range) shall be specified.

C.4 Range for register arrays

The SystemRDL 1.0 specification was silent on allowing a **range** for a register array. This is no longer a valid syntax. A single positive integer size shall be specified.

C.5 decrsaturate

The SystemRDL 1.0 specification specified the default value for **decrsaturate** as `1`. This is incorrect. The SystemRDL 2.0 specification correctly lists the default as `0`.

C.6 enum

The SystemRDL 1.0 grammar allowed for enumeration types to have no enumerators. For SystemRDL 2.0, enumeration types are required to specify at least one enumerator.

C.7 alias

The SystemRDL 1.0 specification was unclear regarding the **alias** register type. SystemRDL 2.0 clarifies that an **alias** shall be of the same type (**internal** or **external**) as the primary register. Since this is done by default, alias registers do not need to specify a register type.

C.8 hwenable and hwmask

These two properties are mentioned in SystemRDL 1.0 as `sizedNumeric` or *boolean*. In reality, these are references, to allow another element to specify the enable or mask value. This has been corrected in SystemRDL 2.0.

C.9 threshold, incrthreshold, and decrtheshold

The *threshold* properties are not clearly defined in SystemRDL 1.0. In one place, a threshold occurs when the field value exceeds the threshold value. In another place, it says the field "exactly matches" the value. In SystemRDL 2.0, this has been clarified: Threshold is set for **threshold** and **incrthreshold** when the field value is greater than or equal to the property value and for **decrthreshold** when the field value is less than or equal to the property value.

Annex D

(normative)

Reserved words

Reserved words have a similar effect as keywords; reserved words are explicitly reserved for future use. The reserved words are listed in Table D1.

Table D1—SystemRDL reserved words

alternate	**byte**	**int**	**precedencetype**	**real**
shortint	**shortreal**	**signed**	**with**	**within**

See also 4.4.

Annex E

(normative)

Access modes

IEEE 1685-2014 IP-XACT inherited the access modes from SystemRDL, but, several more were added in addition to those from SystemRDL. UVM also inherited a subset of the IP-XACT access modes. Supporting all the IP-XACT access modes supports all of the UVM access modes; however, many of the SystemRDL and IP-XACT access mode combinations still map to the UVM User-defined access mode.

Table E1 shows the access combinations between SystemRDL and the UVM and IP-XACT Standards.

Table E1—Access combinations

Access	Read effect	Write function	IP-XACT IEEE 1685-2014	UVM (1.2)	System RDL 1.0	SystemRDL 2.0
No Access			-	-	sw=na	sw=na
Read-only			access=read-only	RO	sw=r	sw=r onread=r
Read-only	clear		access=read-only readAction=clear	RC	sw=r rclr	sw=r onread =rclr
Read-only	set		access=read-only readAction=set	RS	sw=r rset	sw=r onread =rset
Read-only	other		access=read-only readAction=modify	-	-	sw=r onread =ruser
Write-only			access=write-only	WO	sw=w	sw=w onwrite =w
Write-only		One-clear	access=write-only modifiedWriteValue=oneToClear	-	sw=w woclr	sw=w onwrite =woclr
Write-only		One-set	access=write-only modifiedWriteValue=oneToSet	-	sw=w woset	sw=w onwrite =woset
Write-only		One-toggle	access=write-only modifiedWriteValue=oneToTog-gle	-	-	sw=w onwrite =wot
Write-only		Zero-clear	access=write-only modifiedWriteValue=zeroToClear	-	-	sw=w onwrite =wzc
Write-only		Zero-set	access=write-only modifiedWriteValue=zeroToSet	-	-	sw=w onwrite =wzs
Write-only		Zero-tog-gle	access=write-only modifiedWriteValue=zeroToTog-gle	-	-	sw=w onwrite =wzt
Write-only		Clear	access=write-only modifiedWriteValue=clear	WOC	-	sw=w onwrite =wclr

Table E1—Access combinations (Continued)

Access	Read effect	Write function	IP-XACT IEEE 1685-2014	UVM (1.2)	System RDL 1.0	SystemRDL 2.0
Write-only		Set	access=write-only modifiedWriteValue=set	WOS	-	sw=w onwrite =wset
Write-only		Other	access=write-only modifiedWriteValue=modify	-	-	sw=w onwrite =wuser
Write-only-once			access=writeOnce	WO1	-	sw=w1 onwrite =w
Write-only-once		One-clear	access=writeOnce modifiedWriteValue=oneToClear	-	-	sw=w1 onwrite =woclr
Write-only-once		One-set	access= writeOnce modifiedWriteValue=oneToSet	-	-	sw=w1 onwrite =woset
Write-only-once		One-toggle	access= writeOnce modifiedWriteValue=oneToTog-gle	-	-	sw=w1 onwrite =wot
Write-only-once		Zero-clear	access= writeOnce modifiedWriteValue=zeroToClear	-	-	sw=w1 onwrite =wzc
Write-only-once		Zero-set	access= writeOnce modifiedWriteValue=zeroToSet	-	-	sw=w1 onwrite =wzs
Write-only-one		Zero-tog-gle	access= writeOnce modifiedWriteValue=zeroToTog-gle	-	-	sw=w1 onwrite =wzt
Write-only-once		Clear	access= writeOnce modifiedWriteValue=clear	-	-	sw=w1 onwrite =wclr
Write-only-once		Set	access= writeOnce modifiedWriteValue=set	-	-	sw=w1 onwrite =wset
Write-only-once		Other	access= writeOnce modifiedWriteValue=modify	-	-	sw=w1 onwrite =wuser
Read-write			access=read-write	RW	sw=rw	sw=rw onread =r onwrite =w
Read-write		One-clear	access= read-write modifiedWriteValue=oneToClear	W1C	sw=rw woclr	sw=rw onread =r onwrite =woclr
Read-write		One-set	access= read-write modifiedWriteValue=oneToSet	W1S	sw=rw woset	sw=rw onread =r onwrite =woset
Read-write		One-toggle	access= read-write modifiedWriteValue=oneToTog-gle	W1T	-	sw=rw onread =r onwrite =wot
Read-write		Zero-clear	access= read-write modifiedWriteValue=zeroToClear	W0C	-	sw=rw onread =r onwrite =wzc

Table E1—Access combinations (Continued)

Access	Read effect	Write function	IP-XACT IEEE 1685-2014	UVM (1.2)	System RDL 1.0	SystemRDL 2.0
Read-write		Zero-set	access= read-write modifiedWriteValue=zeroToSet	W0S	-	sw=rw onread =r onwrite =wzs
Read-write		Zero-toggle	access= read-write modifiedWriteValue=zeroToToggle	W0T	-	sw=rw onread =r onwrite =wzt
Read-write		Clear	access= read-write modifiedWriteValue=clear	WC	-	sw=rw onread =r onwrite =wclr
Read-write		Set	access= read-write modifiedWriteValue=set	WS	-	sw=rw onread =r onwrite =wset
Read-write		Other	access= read-write modifiedWriteValue=modify	-	-	sw=rw onread =r onwrite =wuser
Read-write	clear		access=read-write readAction=clear	WRC	sw=rw rclr	sw=rw onread =rclr onwrite =w
Read-write	clear	One-clear	access= read-write readAction=clear modifiedWriteValue=oneToClear	-	sw=rw rclr woclr	sw=rw onread =rclr onwrite =woclr
Read-write	clear	One-set	access= read-write readAction=clear modifiedWriteValue=oneToSet	W1SRC	sw=rw rclr woset	sw=rw onread =rclr onwrite =woset
Read-write	clear	One-toggle	access= read-write readAction=clear modifiedWriteValue=oneToToggle	-	-	sw=rw onread =rclr onwrite =wot
Read-write	clear	Zero-clear	access= read-write readAction=clear modifiedWriteValue=zeroToClear	-	-	sw=rw onread =rclr onwrite =wzc
Read-write	clear	Zero-set	access= read-write readAction=clear modifiedWriteValue=zeroToSet	W0SRC	-	sw=rw onread =rclr onwrite =wzs
Read-write	clear	Zero-toggle	access= read-write readAction=clear modifiedWriteValue=zeroToToggle	-	-	sw=rw onread =rclr onwrite =wzt
Read-write	clear	Clear	access= read-write readAction=clear modifiedWriteValue=clear	-	-	sw=rw onread =rclr onwrite =wclr
Read-write	clear	Set	access= read-write readAction=clear modifiedWriteValue=set	WSRC	-	sw=rw onread =rclr onwrite =wset

Table E1—Access combinations (Continued)

Access	Read effect	Write function	IP-XACT IEEE 1685-2014	UVM (1.2)	System RDL 1.0	SystemRDL 2.0
Read-write	clear	Other	access= read-write readAction=clear modifiedWriteValue=modify	-	-	sw=rw onread =rclr onwrite =wuser
Read-write	set		access=read-write readAction=set	WRS	sw=rw rset	sw=rw onread = rset onwrite =w
Read-write	set	One-clear	access= read-write readAction=set modifiedWriteValue=oneToClear	W1CRS	sw=rw rset woclr	sw=rw onread = rset onwrite =woclr
Read-write	set	One-set	access= read-write readAction=set modifiedWriteValue=oneToSet	-	sw=rw rset woset	sw=rw onread = rset onwrite =woset
Read-write	set	One-toggle	access= read-write readAction= set modifiedWriteValue=oneToTog-gle	-	-	sw=rw onread = rset onwrite =wot
Read-write	set	Zero-clear	access= read-write readAction= set modifiedWriteValue=zeroToClear	W0CRS	-	sw=rw onread = rset onwrite =wzc
Read-write	set	Zero-set	access= read-write readAction= set modifiedWriteValue=zeroToSet	-	-	sw=rw onread = rset onwrite =wzs
Read-write	set	Zero-tog-gle	access= read-write readAction= set modifiedWriteValue=zeroToTog-gle	-	-	sw=rw onread = rset onwrite =wzt
Read-write	set	Clear	access= read-write readAction= set modifiedWriteValue=clear	WCRS	-	sw=rw onread = rset onwrite =wclr
Read-write	set	Set	access= read-write readAction= set modifiedWriteValue=set	-	-	sw=rw onread = rset onwrite =wset
Read-write	set	Other	access= read-write readAction= set modifiedWriteValue=modify	-	-	sw=rw onread =rset onwrite =wuser
Read-write	other		access=read-write readAction= modify	-	-	sw=rw onread = ruser onwrite =w
Read-write	other	One-clear	access= read-write readAction=modify modifiedWriteValue=oneToClear	-	-	sw=rw onread = ruser onwrite =woclr
Read-write	other	One-set	access= read-write readAction=modify modifiedWriteValue=oneToSet	-	-	sw=rw onread = ruser onwrite =woset

Table E1—Access combinations (Continued)

Access	Read effect	Write function	IP-XACT IEEE 1685-2014	UVM (1.2)	System RDL 1.0	SystemRDL 2.0
Read-write	other	One-toggle	access= read-write readAction= modify modifiedWriteValue=oneToToggle	-	-	sw=rw onread = ruser onwrite =wot
Read-write	other	Zero-clear	access= read-write readAction= modify modifiedWriteValue=zeroToClear	-	-	sw=rw onread = ruser onwrite =wzc
Read-write	other	Zero-set	access= read-write readAction= modify modifiedWriteValue=zeroToSet	-	-	sw=rw onread = ruser onwrite =wzs
Read-write	other	Zero-toggle	access= read-write readAction= modify modifiedWriteValue=zeroToToggle	-	-	sw=rw onread = ruser onwrite =wzt
Read-write	other	Clear	access= read-write readAction= modify modifiedWriteValue=clear	-	-	sw=rw onread = ruser onwrite =wclr
Read-write	other	Set	access= read-write readAction= modify modifiedWriteValue=set	-	-	sw=rw onread = ruser onwrite =wset
Read-write	other	Other	access= read-write readAction= modify modifiedWriteValue=modify	-	-	sw=rw onread = ruser onwrite =wuser
Read-write-once			access=read-writeOnce	W1	-	sw=rw1 onread =r onwrite =w
Read-write-once		One-clear	access= read-writeOnce modifiedWriteValue=oneToClear	-	-	sw=rw1 onread =r onwrite =woclr
Read-write-once		One-set	access= read-writeOnce modifiedWriteValue=oneToSet	-	-	sw=rw1 onread =r onwrite =woset
Read-write-once		One-toggle	access= read-writeOnce modifiedWriteValue=oneToToggle	-	-	sw=rw1 onread =r onwrite =wot
Read-write-once		Zero-clear	access= read-writeOnce modifiedWriteValue=zeroToClear	-	-	sw=rw1 onread =r onwrite =wzc
Read-write-once		Zero-set	access= read-writeOnce modifiedWriteValue=zeroToSet	-	-	sw=rw1 onread =r onwrite =wzs
Read-write-once		Zero-toggle	access= read-writeOnce modifiedWriteValue=zeroToToggle	-	-	sw=rw1 onread =r onwrite =wzt

Table E1—Access combinations (Continued)

Access	Read effect	Write function	IP-XACT IEEE 1685-2014	UVM (1.2)	System RDL 1.0	SystemRDL 2.0
Read-write-once		Clear	access= read-writeOnce modifiedWriteValue=clear	-	-	sw=rw1 onread =r onwrite =wclr
Read-write-once		Set	access= read-writeOnce modifiedWriteValue=set	-	-	sw=rw1 onread =r onwrite =wset
Read-write-once		Other	access= read-writeOnce modifiedWriteValue=modify	-	-	sw=rw1 onread =r onwrite =wuser
Read-write-once	clear		access=read-writeOnce readAction=clear	-	-	sw=rw1 readeffect=rclr onwrite =w
Read-write-once	clear	One-clear	access= read-writeOnce readAction=clear modifiedWriteValue=oneToClear	-	-	sw=rw1 onread =rclr onwrite =woclr
Read-write-once	clear	One-set	access= read-writeOnce readAction=clear modifiedWriteValue=oneToSet	-	-	sw=rw1 onread =rclr onwrite =woset
Read-write-once	clear	One-toggle	access= read-writeOnce readAction=clear modifiedWriteValue=oneToTog-gle	-	-	sw=rw1 onread =rclr onwrite =wot
Read-write-once	clear	Zero-clear	access= read-writeOnce readAction=clear modifiedWriteValue=zeroToClear	-	-	sw=rw1 onread =rclr onwrite =wzc
Read-write-once	clear	Zero-set	access= read-writeOnce readAction=clear modifiedWriteValue=zeroToSet	-	-	sw=rw1 onread =rclr onwrite =wzs
Read-write-once	clear	Zero-tog-gle	access= read-writeOnce readAction=clear modifiedWriteValue=zeroToTog-gle	-	-	sw=rw1 onread =rclr onwrite =wzt
Read-write-once	clear	Clear	access= read-writeOnce readAction=clear modifiedWriteValue=clear	-	-	sw=rw onread =rclr onwrite =wclr
Read-write-once	clear	Set	access= read-writeOnce readAction=clear modifiedWriteValue=set	-	-	sw=rw1 onread =rclr onwrite =wset
Read-write-once	clear	Other	access= read-writeOnce readAction=clear modifiedWriteValue=modify	-	-	sw=rw1 onread =rclr onwrite =wuser
Read-write-once	set		access=read-writeOnce readAction=set	-	-	sw=rw1 onread = rset onwrite =w

Table E1—Access combinations (Continued)

Access	Read effect	Write function	IP-XACT IEEE 1685-2014	UVM (1.2)	System RDL 1.0	SystemRDL 2.0
Read-write-once	set	One-clear	access= read-writeOnce readAction=set modifiedWriteValue=oneToClear	-	-	sw=rw1 onread = rset onwrite =woclr
Read-write-once	set	One-set	access= read-writeOnce readAction=set modifiedWriteValue=oneToSet	-	-	sw=rw1 onread = rset onwrite =woset
Read-write-once	set	One-toggle	access= read-writeOnce readAction= set modifiedWriteValue=oneToToggle	-	-	sw=rw1 readeffect= rset onwrite =wot
Read-write-once	set	Zero-clear	access= read-writeOnce readAction= set modifiedWriteValue=zeroToClear	-	-	sw=rw1 onread = rset onwrite =wzc
Read-write-once	set	Zero-set	access= read-writeOnce readAction= set modifiedWriteValue=zeroToSet	-	-	sw=rw1 onread = rset onwrite =wzs
Read-write-once	set	Zero-toggle	access= read-writeOnce readAction= set modifiedWriteValue=zeroToToggle	-	-	sw=rw1 onread = rset writefunction=wzt
Read-write-once	set	Clear	access= read-writeOnce readAction= set modifiedWriteValue=clear	-	-	sw=rw1 onread = rset onwrite =wclr
Read-write-once	set	Set	access= read-writeOnce readAction= set modifiedWriteValue=set	-	-	sw=rw1 onread = rset onwrite =wset
Read-write-once	set	Other	access= read-writeOnce readAction= set modifiedWriteValue=modify	-	-	sw=rw1 onread =rset onwrite =wuser
Read-write-once	other		access=read-writeOnce readAction= modify	-	-	sw=rw1 onread = ruser onwrite =w
Read-write-once	other	One-clear	access= read-writeOnce readAction=modify modifiedWriteValue=oneToClear	-	-	sw=rw1 onread = ruser onwrite =woclr
Read-write-once	other	One-set	access= read-writeOnce readAction=clear modifiedWriteValue=oneToSet	-	-	sw=rw1 onread = ruser onwrite =woset
Read-write-once	other	One-toggle	access= read-writeOnce readAction= modify modifiedWriteValue=oneToToggle	-	-	sw=rw1 onread = ruser onwrite =wot
Read-write-once	other	Zero-clear	access= read-writeOnce readAction= modify modifiedWriteValue=zeroToClear	-	-	sw=rw1 onread = ruser onwrite =wzc

Table E1—Access combinations (Continued)

Access	Read effect	Write function	IP-XACT IEEE 1685-2014	UVM (1.2)	System RDL 1.0	SystemRDL 2.0
Read-write-once	other	Zero-set	access= read-writeOnce readAction= modify modifiedWriteValue=zeroToSet	-	-	sw=rw1 onread = ruser onwrite =wzs
Read-write-once	other	Zero-toggle	access= read-writeOnce readAction= modify modifiedWriteValue=zeroToToggle	-	-	sw=rw1 onread = ruser onwrite =wzt
Read-write-once	other	Clear	access= read-writeOnce readAction= modify modifiedWriteValue=clear	-	-	sw=rw1 onread = ruser onwrite =wclr
Read-write-once	other	Set	access= read-writeOnce readAction= modify modifiedWriteValue=set	-	-	sw=rw1 onread = ruser onwrite =wset
Read-write-once	other	Other	access= read-writeOnce readAction= modify modifiedWriteValue=modify	-	-	sw=rw1 onread = ruser onwrite= wuser

Annex F

(informative)

Formatting text strings

SystemRDL has a set of tags which can be used to format text strings. These tags are based on the phpBB code formatting tags, which are extended for use with SystemRDL and referred to as *RDLFormatCode*. The RDLFormatCode tags shall be interpreted by the SystemRDL compiler and rendered in the generated output. The set of tags specified below is the complete set and is not extensible like phpBB code. These tags are only interpreted within the **name** and **desc** properties in SystemRDL (see Table 5). If a SystemRDL compiler encounters an unknown tag, this tag shall be ignored by the compiler and passed through as is.

The concept of phpBB code takes its origin from the HTML 4.01 standard; for additional information on using phpBB tags, see https://www.phpbb.com/community/faq.php?mode=bbcode (which suggests a formatted section illustrating the results of such usage).

F.1 Well-formed RDLFormatCode constructs

A well-formed tag also has an end-tag. For nesting well-formed tags, the innermost shall be closed before the outermost one is.

```
[b]Text[/b]                    -- Bold

[i]Text[/i]                    -- Italic

[u]Text[/u]                    -- Underline

[color=colorValue]Text[/color]-- Color See F.3 for colorValues

[size=size]Text[/size]         -- Font size where size is a valid HTML size

[url]Text[/url]                -- URL reference

URL references can specified in two forms.

1. [url]http://www.accellera.org[/url] -- which places the target link the
   generated code.
2. [url=http://www.accellera.org]Accellera[/url] -- Which displays the text
   Accellera but links the URL provided.

[email]Text[/email]            -- Email address in the form of user@domain

[img]image reference[/img]     -- Insert image reference here. Image reference
   can be relative pathname or absolute path name. Its up to the user to follow
   valid path rules for the target system that they are generating code for.

[code]Text[/code]          -- Anything that requires a fixed width
                              with a Courier-type font
[list] , [list=1]
 or [list=a]                   -- Listing directives, un-ordered or
   [*] list element               ordered (numbered: list=1,
   [*] list element                          alpha: list=a)
   [*] list element
```

```
[/list]

[quote]text[/quote]           -- Replaces with ". useful for putting
                                 "'s inside a name or desc field.
```

F.2 Single-tag RDLFormatCode constructs

```
[br]             -- Line break

[lb]             -- Left bracket ([)

[rb]             -- Right bracket (])

[p]              -- Paragraph begin

[sp]             -- White Space (equivalent to an HTML  )

[index]          -- Replaced by the index # of the individual component
                    instance when instantiated as an array. When representing
                    an individual array element this substitutes the index and
                    for an entire array it substitutes the range.

[index_parent]   -- Replaced by the index # of the individual component
                    parent instance when the parent is instantiated as an
                    array (extends phpBB).When representing an individual
                    array element this substitutes the index and for an entire
                    array it substitutes the range.


[name]           -- Replaced by the descriptive name of the component
                    (extends phpBB). This tag is undefined when used inside
                    the value of the name property.

[desc]           -- Replaced by the component's description (extends phpBB).

[instname]       -- Replaced by the instance name (extends phpBB).
```

F.3 colorValues for the color tag

The RDLFormatCode color can accept two forms of arguments for color: enumerated values specified by the HTML 4.01 or CSS specifications and RGB #'s.

Example

```
Who is afraid of [color=red]red[/color],  [color=#eeaa00]yellow[/color]
   and [color=#30f]blue[/color]?
```

HTML 4.01 & CSS2 Colors

Color Name	Hex 6	RGB	RGB%	Sample
black	#000000	0,0,0	0%,0%,0%	
silver	#C0C0C0	#######	75%,75%,75%	
gray	#808080	#######	50%,50%,50%	
white	#FFFFFF	#######	100%,100%,100%	
maroon	#800000	128,0,0	50%,0%,0%	
red	#FF0000	255,0,0	100%,0%,0%	
purple	#800080	128,0,128	50%,0%,50%	
fuchsia	#FF00FF	255,0,255	100%,0%,100%	
green	#008000	0,128,0	0%,50%,0%	
lime	#00FF00	0,255,0	0%,100%,0%	
olive	#808000	128,128,0	50%,50%,0%	
yellow	#FFFF00	255,255,0	100%,100%,0%	
navy	#000080	0,0,128	0%,0%,50%	
blue	#0000FF	0,0,255	0%,0%,100%	
teal	#008080	0,128,128	0%,50%,50%	
aqua	#00FFFF	0,255,255	0%,100%,100%	

F.4 Example

The following code sample demonstrates some simple uses of RDLFormatCode.

```
addrmap top {
  name = "RDLCode Example";
  // desc  = "Please refer to [quote]the[/quote] specification [url=http://
    www.yahoo.com]here[/url] for details.";
  reg {
    name = "Register my index = [index] my [b]parents index = [index_parent]
   my instname = [instname] [index][/b]";
    desc  = "Please [b][u]refer[index] to the [index] specification[/u][/b]
   [url=http://www.yahoo.com]here[/url] for details.";
    field {
```

```
        name = "START [test] [br] [b]Some bold text for
      [instname][lb][index][rb][/b],
               [i]italic[/i], [u]underline[/u], [email]tcook@denali.com[/email],
                 [img]some_image.gif[/img]
                 [p][color=#ff3366]Some Color[/color][/p]
                 [code]echo This is some code;[/code]
                 [size=18][color=red][b]LOOK AT ME![/b][/color][/size]
                 [list]
                 [*][color=red]Red[/color]
                 [*][color=blue]Blue[/color]
                 [*][color=green]Green[/color]
                 [/list]
                 [list=1]
                 [*]Red
                 [*]Blue
                 [*]Yellow
                 [/list]
                 [list=a]
                 [*]Red
                 [*]Blue
                 [*]Yellow
                 [/list]
                 ";

        desc  = "Please [some unknown tag] refer to [list=1] [*]Red [*]Green [/
      list] the specification [url=http://www.google.com]here[/url] for
      details.";
       } f1;
     } r1 [10];
   };
```

NOTE—Some details of the sample output are the result of factors outside the control of RDLCode and are functions of the compiler, its arguments, or supporting style sheets.

Annex G

(informative)

Component-property relationships

Table G1 lists all properties defined in SystemRDL. For each property, Table G1 specifies which component types allow the property and gives references to the tables (or section) where the property is defined (e.g., Table 23 for the property **accesswidth** (within the **reg** component description)). The **Mutual exclude** column designates groups of properties which are mutually exclusive (e.g, group A shows **activehigh** and **activelow** are mutually exclusive). Each mutual exclusion group is given a letter (e.g., A), which is shown next to all members of that group. Table G1 also shows the type for each property and if it can be dynamically assigned (**Dyn assign**). The **Ref target** column indicates if a property may be a reference target if the column value is 'x' or 'y'. When the **Ref target** column contains a 'y', the implementation of the target needs to have the referenced net available due to an inherited or an assigned property value.

Table G1—Property cross-reference

Property	Mutual exclude	Components	See also	Type	Ref target	Dyn assign	Notes
accesswidth		**reg**	Table 23	longint unsigned		x	
activehigh	A	**signal**	Table 10	boolean		x	
activelow	A	**signal**	Table 10	boolean		x	
addressing		**addrmap**	Table 26	addressingtype			**compact**, **regalign**, or **fullalign**
alignment		**addrmap**, **regfile**	Table 26 Table 25	longint unsigned			
anded		**field**	Table 18	boolean		x	
anded		**field**	Table 18	N/A	x		Reduction AND of field value
async	N	**signal**	Table 10	boolean		x	
bigendian	L	**addrmap**	Table 26	boolean		x	
bridge		**addrmap**	Table 27	boolean			
constraint_disable		**constraint**	Table 29	boolean		x	
counter	E	**field**	Table 19	boolean		x	
cpuif_reset		**signal**	Table 10	boolean		x	
decr		**field**	Table 19	instance reference	y	x	
decrsaturate		**field**	Table 19	boolean, bit, instance reference		x	Decrementing counter saturate value
decrsaturate		**field**	Table 19	N/A	y		Decrementing counter saturate reached
decrthreshold		**field**	Table 19	boolean, bit, instance reference		x	Decrementing counter threshold value

Table G1—Property cross-reference (Continued)

Property	Mutual exclude	Components	See also	Type	Ref target	Dyn assign	Notes
decrthreshold		**field**	Table 19	N/A	y		Decrementing counter threshold reached
decrvalue	G	**field**	Table 19	bit, instance reference	y	x	
decrwidth	G	**field**	Table 19	longint unsigned		x	
desc		**addrmap**, **constraint**, **field**, **mem**, **reg**, **regfile**, **signal**	Table 5	string		x	Also used in **enum**eration (Table 5)
dontcompare	O	**field**	Table 6	boolean, bit		x	
dontcompare	O	**addrmap**, **reg**, **regfile**	Table 6	boolean		x	
donttest	O	**field**	Table 6	boolean, bit		x	
donttest	O	**addrmap**, **reg**, **regfile**	Table 6	boolean		x	
enable	J	**field**	Table 21	instance reference	y	x	
encode		**field**	Table 22	enum type reference		x	**enum**eration object reference
errextbus		**addrmap**, **reg**, **regfile**	Table 26 Table 23 Table 25	boolean			
field_reset		**signal**	Table 10	boolean		x	
fieldwidth		**field**	Table 18	longint unsigned			
halt		**reg**	Table 23	N/A	y		Reduction OR of halt Reference target needs to contain fields with halt
haltenable	K	**field**	Table 21	instance reference	y	x	
haltmask	K	**field**	Table 21	instance reference	y	x	
hdl_path		**addrmap**, **reg**, **regfile**	Table 28	string		x	
hdl_path_gate		**addrmap**, **reg**, **regfile**	Table 28	string		x	
hdl_path_gate_slice		**field**, **mem**	Table 28	string[]		x	
hdl_path_slice		**field**, **mem**	Table 28	string[]		x	
hw		**field**	Table 11	accesstype			**r**, **w**, **rw**, **wr**, **w1**, **rw1**, or **na**
hwclr		**field**	Table 18	boolean, instance reference	y	x	
hwenable	D	**field**	Table 18	instance reference	y	x	

Table G1—Property cross-reference (Continued)

Property	Mutual exclude	Components	See also	Type	Ref target	Dyn assign	Notes
hwmask	D	**field**	Table 18	instance reference	y	x	
hwset		**field**	Table 18	boolean, instance reference	y	x	
incr		**field**	Table 19	instance reference	y	x	
incrsaturate		**field**	Table 19	boolean, bit, instance reference		x	Incrementing counter saturate value
incrsaturate		**field**	Table 19	N/A	y		Incrementing counter saturate reached
incrthreshold		**field**	Table 19	boolean, bit, instance reference		x	Incrementing counter threshold value
incrthreshold		**field**	Table 19	N/A	y		Incrementing counter threshold reached
incrvalue	F	**field**	Table 19	bit, instance reference	y	x	
incrwidth	F	**field**	Table 19	longint unsigned		x	
intr	E	**field**	Table 21	boolean		x	
intr		**reg**	Table 23	N/A	y		Reference target needs to contain interrupt fields
ispresent		**addrmap**, **constraint**, **field**, **mem**, **reg**, **regfile**, **signal**	See 5.3	boolean		x	
littleendian	L	**addrmap**	Table 26	boolean		x	
lsb0	M	**addrmap**	Table 26	boolean			
mask	J	**field**	Table 21	instance reference	y	x	
mementries		**mem**	Table 24	longint unsigned			
memwidth		**mem**	Table 24	longint unsigned			
msb0	M	**addrmap**	Table 26	boolean			
name		**addrmap**, **constraint**, **field**, **mem**, **reg**, **regfile**, **signal**	Table 5	string		x	Also used in **enum**eration (Table 5)
next		**field**	Table 13	instance reference	y	x	
onread	P	**field**	Table 14	onreadtype		x	
onwrite	B	**field**	Table 14	onwritetype		x	
ored		**field**	Table 18	boolean		x	

Table G1—Property cross-reference (Continued)

Property	Mutual exclude	Components	See also	Type	Ref target	Dyn assign	Notes
ored		**field**	Table 18	N/A	x		Reduction OR of field value
overflow		**field**	Table 19	boolean		x	
overflow		**field**	Table 19	N/A	y		Counter overflow
paritycheck		**field**	Table 22	boolean			
precedence		**field**	Table 22	precedencetype		x	**hw** or **sw**
rclr	P	**field**	Table 14	boolean		x	
regwidth		**reg**	Table 23	longint unsigned			
reset		**field**	Table 13	bit, instance reference	y	x	
resetsignal		**field**	Table 13	instance reference	y	x	
rset	P	**field**	Table 14	boolean		x	
rsvdset	Q	**addrmap**	Table 26	boolean			
rsvdsetX	Q	**addrmap**	Table 26	boolean			
saturate		**field**	Table 19	boolean, bit, instance reference		x	Incrementing counter saturate value
saturate		**field**	Table 19	N/A	y		Incrementing counter saturate reached
shared		**reg**	Table 23	boolean			
sharedextbus		**addrmap**, **regfile**	Table 26 Table 25	boolean			
signalwidth		**signal**	Table 10	longint unsigned			
singlepulse		**field**	Table 14	boolean		x	
sticky	I	**field**	Table 21	boolean		x	
stickybit	I	**field**	Table 21	boolean		x	
sw		**field**, **mem**	Table 11 Table 24	accesstype		x	**r**, **w**, **rw**, **wr**, **w1**, **rw1**, or **na**
swacc		**field**	Table 14	boolean		x	
swacc		**field**	Table 14	N/A	x		Accessed by software
swmod		**field**	Table 14	boolean		x	
swmod		**field**	Table 14	N/A	x		Modified by software
swwe	R	**field**	Table 14	boolean, instance reference	y	x	
swwel	R	**field**	Table 14	boolean, instance reference	y	x	
sync	N	**signal**	Table 10	boolean		x	

Table G1—Property cross-reference (Continued)

Property	Mutual exclude	Components	See also	Type	Ref target	Dyn assign	Notes
threshold		**field**	Table 19	boolean, bit, instance reference		x	Incrementing counter threshold value
threshold		**field**	Table 19	N/A	y		Incrementing counter threshold reached
underflow		**field**	Table 19	boolean		x	
underflow		**field**	Table 19	N/A	y		
we	C	**field**	Table 18	boolean, instance reference	y	x	
wel	C	**field**	Table 18	boolean, instance reference	y	x	
woclr	B	**field**	Table 14	boolean		x	
woset	B	**field**	Table 14	boolean		x	
xored		**field**	Table 18	boolean		x	
xored		**field**	Table 18	N/A	x		Reduction XOR of field value