SystemRDL v1.0: A specification for a Register Description Language

Prepared by the

Register Description Working Group
of
The SPIRIT Consortium

SystemRDL™

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.

The SPIRIT Consortium.
1370 Trancas Street #184, Napa, CA 94558

Copyright © 2007-2009 by the SPIRIT Consortium.
All rights reserved. Published 24 March 2009. Printed in the United States of America.

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

The SPIRIT Consortium is a consortium of electronic system, IP provider, semiconductor, and EDA companies. Other supporting documents (with comments and examples) are available from the public area of the http://www.spiritconsortium.org web site.

Notice to users

Errata

Errata, if any, for this and all other standards of The SPIRIT Consortium can be accessed at the following URL: http://www.spiritconsortium.org/releases/errata/. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations, users guides, examples, etc. can be accessed at the following URL:
http://www.spiritconsortium.org/tech/docs/.

Patents

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.

Participants

The following members and observers took part in the Register Description Working Group (RDWG):

Will Adams, Freescale Semiconductor, *Chair RDWG*
Gary Delp, LSI, *Technical Director*
Joe Daniels, *Technical Editor*

ARM: Anthony Berent

Cadence: Jean-Michel Fernandez

Cisco Systems: Michael Faust

Denali Software: Joe Bauer, Mark Gogolewski, Gary Lippert, Sean Smith

Duolog: Edwin Dankert

Freescale Semiconductor: Will Adams, Karl Heubaum, Seth Park, Kathy Werner

LSI: Gary Delp, Dave Fechser, Gary Lippert

MIPS Technologies: Michael Uhler

NXP Semiconductors: Greg Ehmann, Jan Stuyt

Semifore: Richard Weber

ST Microelectronics: Christophe Amerijckx, Guillaume Cernier, Serge DePaoli

Special acknowledgment is given to:

Denali: Contribution of initial specification upon which the work is based

The Board of Directors of The SPIRIT Consortium active during the release of the SystemRDL Standard:

Contents

1. Overview ... 1

1.1 Scope ... 1
1.2 Purpose ... 1
1.3 Motivation ... 2
1.4 Conventions used ... 2
1.4.1 Visual cues (meta-syntax) ... 2
1.4.2 Notational conventions ... 2
1.4.3 Examples ... 2
1.5 Use of color in this standard ... 3
1.6 Contents of this standard ... 3

2. References ... 5

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

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

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 ... 14
5.1.3 Specifying component properties ... 14
5.1.4 Scoping ... 17
5.2 General component properties ... 18
5.2.1 Universal properties ... 18
5.2.2 Structural properties ... 19

6. Signals ... 21

6.1 Introduction ... 21
6.2 Signal properties ... 21
6.2.1 Semantics ... 21
6.2.2 Example ... 21
6.3 Signal definition and instantiation ... 22
6.3.1 Semantics ... 22
6.3.2 Example ... 22

7. Field component ... 23

7.1 Introduction 23
7.2 Defining and instantiating fields 23
7.3 Using scalar and array field instances 23
7.4 Field access properties 24
7.4.1 Semantics 25
7.4.2 Example 25
7.5 Hardware signal properties 25
7.5.1 Semantics 26
7.5.2 Example 26
7.6 Software access properties 27
7.6.1 Semantics 27
7.6.2 Examples 28
7.7 Hardware access properties 28
7.7.1 Semantics 29
7.7.2 Example 30
7.8 Counter properties 30
7.8.1 Counter incrementing and decrementing 30
7.8.2 Counter saturation and threshold 31
7.9 Interrupt properties 34
7.9.1 Semantics 37
7.9.2 Example 37
7.10 Miscellaneous field properties 38
7.10.1 Semantics 38
7.10.2 Example 38

8. Register component 39

8.1 Defining and instantiating registers 39
8.1.1 Semantics for all registers 39
8.2 Instantiating internal registers 39
8.2.1 Semantics 39
8.2.2 Example 40
8.3 Instantiating external registers 40
8.3.1 Semantics 40
8.3.2 Example 41
8.4 Instantiating alias registers 41
8.4.1 Semantics 41
8.4.2 Example 42
8.5 Register properties 42
8.5.1 Semantics 42
8.5.2 Example 43
8.6 Understanding field ordering in registers 43
8.6.1 Semantics 43
8.6.2 Examples 44
8.7 Understanding interrupt registers 44
8.7.1 Semantics 44
8.7.2 Example 44

9. Register file component 45

9.1 Defining and instantiating register files 45
9.1.1 Semantics 46
9.1.2 Examples 46
9.2 Register file properties 47

9.2.1 Semantics 47
9.2.2 Example 47

10. Address map component 49

10.1 Introduction 49
10.2 Defining and instantiating address maps 49
10.2.1 Semantics 49
10.2.2 Example 49
10.3 Address map properties 49
10.3.1 Semantics 51
10.3.2 Example 52
10.4 Defining bridges or multiple view address maps 52
10.4.1 Semantics 52
10.4.2 Example 52

11. User-defined properties 55

11.1 Defining user-defined properties 55
11.1.1 Semantics 55
11.1.2 Example 56
11.2 Assigning (and binding) user-defined properties 56
11.2.1 Semantics 56
11.2.2 Example 56

12. Enumeration (bit-field encoding) 57

12.1 Introduction 57
12.2 Defining enumerations 57

13. Preprocessor directives 59

13.1 Embedded Perl preprocessing 59
13.1.1 Semantics 59
13.1.2 Example 59
13.2 Verilog-style preprocessor 59
13.2.1 Verilog-style preprocessor directives 60
13.2.2 Limitations on nested file inclusion 60

14. Advanced topics in SystemRDL 61

14.1 Application of signals for reset 61
14.2 Understanding hierarchical interrupts in SystemRDL 63
14.2.1 Example structure and perspective 64
14.2.2 Code snippet 1 65
14.2.3 Code snippet 2 65
14.2.4 Code snippet 3 66
14.2.5 Code snippet 4 66
14.2.6 Code snippet 5 67
14.2.7 Code snippet 6 68
14.2.8 Code snippet 7 68
14.2.9 Code snippet 8 69
14.2.10 Code snippet 9 70
14.2.11 Code snippet 10 71

14.2.12 Code snippet 11 72
14.3 Understanding bit ordering and byte ordering in SystemRDL 72
14.3.1 Bit ordering 73
14.3.2 Byte ordering 74

Annex A (informative) Bibliography 75

Annex B (normative) Grammar 77

Annex C (informative) Code example 91

Annex D (informative) Formatting text strings 95

Annex E (informative) Component-property relationships 99

SystemRDL v1.0: A specification for a Register Description Language

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 grammar of SystemRDL is specified by the ANTLR[1] form of grammar specification (highlighted in Annex B). The rest of this Standard is intended to be consistent with the ANTLR grammar. If any discrepancies between the two occur, the ANTLR grammar 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]Information on references can be found in Clause 2.

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 Conventions used

The conventions used throughout the document are included here.

1.4.1 Visual cues (meta-syntax)

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

1.4.2 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.

1.4.3 Examples

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

Table 1—Document conventions

Visual cue	Represents
`courier`	The `courier` font indicates SystemRDL or HDL code. For example, the following line indicates SystemRDL code: `field myField {}; // defines a field type named "myField"`
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 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*]**;**
[ ] 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 a parameter list, which usually can be repeated. For example, the following shows one or more universal properties can be specified for this command: *mnemonic_name* = *value* [**{**{*universal_property*;}*}];**
{ } bold curly braces	Bold curly braces are required. For example, in the following property definition, the bold (outer) curly braces need to be typed as they appear: *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* = *unsizedNumeric*;
< > angle brackets	Angle brackets (< >) indicates a grouping, usually of alternative parameters. For example, the following line shows the “in” or “out” key terms are possible values for the “-direction” parameter: **-direction** <**in** | **out**>
| 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 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 effect 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.

1.6 Contents of this standard

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

— 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 describes how signals are used in SystemRDL.
— Clause 7 defines the field components.
— Clause 8 defines the register components.
— Clause 9 defines the register file components.
— Clause 10 defines the address map components.
— Clause 11 defines the user-defined properties.
— Clause 12 defines how to enumerate bit-field encoding.
— Clause 13 defines the preprocessor directives.
— Clause 14 describes advanced uses of SystemRDL.
— Annexes. Following Clause 14 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 1800™, IEEE Standard for SystemVerilog Unified Hardware Design, Specification and Verification Language.

The ANTLR Parser Generator Reference Manual, Version 2 is available from The SPIRIT Consortium web site: http://www.spiritconsortium.org/releases/systemRDL/1.0/antlr.txt.

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 phpBB code syntax is available from the phpBB web site:
http://www.phpbb.com/community/faq.php?mode=bbcode#f0r0.

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

The Unicode Standard, Version 5.1.0, is available from The Unicode Consortium web site:
http://www.unicode.org/versions/Unicode5.1.0/.

[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

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

3.1.2 property: A characteristic, attribute, or a trait of a component in SystemRDL.

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

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

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

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

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

3.1.8 enumeration: An alias of text bound to some bit value or a list of values to describe bit field encoding.

3.1.9 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 14.

3.1.10 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 14.

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

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
SGML	standard generalized markup language

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

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 compilers should support the UTF-8 character encoding for broad worldwide compatibility. SystemRDL is case-sensitive. Only ASCII characters are used for identifiers in SystemRDL. The support for UTF-8 is limited to strings to allow for non-Engligh documentation. SystemRDL compilers shall ignore the byte-order mark.

4.1 White space

White space characters are: *space*, *tab*, *line feed*, and *carriage return*. 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 \ and contain one or more letters, digits, or symbols and are terminated by whitespace. Escaped identifiers shall be limited to SystemRDL keywords only. See Table 2.

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 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

accesswidth	**activehigh**	**activelow**	**addressing**	**addrmap**
alias	**alignment**	**all**	**anded**	**arbiter**
async	**bigendian**	**bothedge**	**bridge**	**clock**
compact	**counter**	**cpuif_reset**	**decr**	**decrsaturate**
decrthreshold	**decrvalue**	**decrwidth**	**default**	**desc**
dontcompare	**donttest**	**enable**	**encode**	**enum**
errextbus	**external**	**false**	**field**	**field_reset**
fieldwidth	**fullalign**	**halt**	**haltenable**	**haltmask**
hw	**hwclr**	**hwenable**	**hwmask**	**hwset**
incr	**incrvalue**	**incrwidth**	**internal**	**intr**
level	**littleendian**	**lsb0**	**mask**	**msb0**
na	**name**	**negedge**	**next**	**nonsticky**
ored	**overflow**	**posedge**	**precedence**	**property**
r	**rclr**	**reg**	**regalign**	**regfile**
regwidth	**reset**	**resetsignal**	**rset**	**rsvdset**
rsvdsetX	**rw**	**saturate**	**shared**	**sharedextbus**
signal	**signalwidth**	**singlepulse**	**sticky**	**stickybit**
sw	**swacc**	**swmod**	**swwe**	**swwel**
sync	**threshold**	**true**	**underflow**	**w**
we	**wel**	**woclr**	**woset**	**wr**
xored				

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 D.

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
32'h_deadbeef // **Illegal**
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 contain details about working with each of the individual components in SystemRDL. See also Annex C.

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

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**

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 type_name {[*property*;]*};

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

component {[*property*;]*} *instance_name*;

a) In both cases, *component* is one of the keywords specified in Table 3 and each *property* is specified as a `name=value` pair, e.g., `name="foo"` (see 5.1.3.1).

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

c) For a anonymously defined component, *instance_name* is the user-specified name for instantiation of the component.

d) The *component definition body* (specified within the curly braces **{}**) is comprised of zero or more of the following.

   1) Default property assignments

   2) Property assignments

   3) Component instantiations

   4) Nested component definitions

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

```
field myField {};
```

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

```
field {} myField;
```

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 instance_name [**[***number***]** | **[***number* **:** *number***]**]**;**

   where

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

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

   3) *number* is a simple decimal or hexadecimal number.

      i) **[***number***]** specifies the size of the instantiated component array.

      ii) **[***number* **:** *number***]** specifies the specific indices of the array. This form of instantiation can only be used for **field** or **signal** components (see Clause 8 and Clause 6).

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 11).

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"
```

5.1.3 Specifying component properties

Component properties define the specific function and purpose of a component, as well as its interaction with other instantiated components. Property types include *boolean*, *string*, *numeric*, *sizedNumeric*, *unsizedNumeric*, *accessType* (enum), *addressingType* (enum), *precedenceType* (enum), *reference* to another instance, or a combination of these; see Table 4.

Table 4—Property types

Type	Definition	Default
boolean	**true** or **false**.	**false**
string	See 4.5.	""
numeric	See 4.6.	Undefined
sizedNumeric	A simple number with the value of 0 or a Verilog-style number, see 4.6 (c - e).	Undefined
unsizedNumeric	A simple number, see 4.6 (a and b).	Undefined
accessType (enum)	One of **rw**, **wr**, **r**, **w**, or **na**. See Clause 7.	**rw**
addressingType (enum)	One of **compact**, **regalign**, or **fullalign**. See Clause 10.	**regalign**
precedenceType (enum)	One of **hw** or **sw**. See Clause 7.	**sw**
reference	A pointer to a component instance or component's instance property.	Null

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 11). 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 [= *value*]**;**

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

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 6 — Clause 11).

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 parent scope (see 5.1.4). Any components defined in the same or lower 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*]**;**

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

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 6 — Clause 11).

Example

```
reg {
  default name = "default name";
  field {} f1; // assumes the name "default name" from above
  field { name = "new name"; } f2; // name assignment overrides "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 6 — Clause 11).

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 arrray.

   2) If the component type is **reg**, **regfile**, 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 = “This value is 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 4)

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

SystemRDL is a statically scoped language, where the *root scope* is the outermost scope. The body of any defined component is its own scope. All component names within a given scope shall be unique. All instance names within a given scope shall be unique. However, there can be a component and instance with the same name in the same scope.

The only component definitions visible at any scope shall be those defined in the current scope and any parent scope, up to and including the root scope. To resolve a component name, SystemRDL searches from the current scope to the outer scope until it finds the first matching component name.

The root scope shall only contain component definitions and **signal** instantiations. No other component instantiations shall be allowed in the root scope. Therefore, all component instantiations shall occur within an **addrmap** component definition (see Clause 10). The root(s) of an **addrmap** hierarchy are those **addrmap**s that are defined, but not subsequently instantiated.

Only instances instantiated in the current scope can be referenced within that scope. A child instance can be referenced via the dot operator (**.**).

A *instance reference* appears as follows.

instance_name [**.** *child_instance_name*]*

where

a) *instance_name* is a previously instantiated component in the current scope (see 5.1.2).

b) the first use of *child_instance_name* shall exist in *instance_name*'s scope.

c) for all other *child_instance_names*, any subsequent *child_instance_name* shall exist in the previous *child_instance_name*'s scope.

Dynamic assignments can also 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

```
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
```

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 *sizedNumeric*	Yes
dontcompare	This is testing property indicates the components read data shall be discarded and not compared against expected results.	*boolean* or *sizedNumeric*	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 (*sizedNumeric*) 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** can not both be set to **true** or a non-zero mask for a particular component.

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.
};
```

6. Signals

6.1 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 14.1.

6.2 Signal properties

Table 7 shows the signal properties.

Table 7—Signal properties

Property	Implementation/Application	Type	Dynamic[a]
signalwidth	Width of the signal.	*numeric*	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 **field_reset** is not defined, this reverts to the default reset signal. This parameter only controls the CPU interface of a generated slave and is provided as an advanced feature for designs when the reset strategy is different for the bus to which the registers are connected and the register implementations themselves.	*boolean*	Yes
field_reset	Default signal to use for resetting field implementations. This parameters controls on the resets used on the HW interfaces of a generated slave.	*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.

6.2.1 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 defined as neither.

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) **cpuif_reset** property can only be set **true** for one instantiated signal in an address map and that address map's non-address map instances.

f) **field_reset** property can only be set to **true** for one instantiated signal in an address map and that address map's non-address map instances.

g) If **signalwidth** is specified in a **signal** component definition, its size cannot be overridden during instantiation.

6.2.2 Example

See the example in 6.3.2.

6.3 Signal definition and instantiation

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

6.3.1 Semantics

a) If **signalwidth** (see 6.2) 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.

6.3.2 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
};
```

7. Field component

7.1 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** and enumeration (**enum**) 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*.

7.2 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 8). Fields are defined and instantiated as described in 5.1, with the following additional semantics. See also 7.3.

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 explicity defined, fields shall be positioned sequentially in the order they are instantianted 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 determined by the SystemRDL compiler as described in d or specified explicitly by using exact indices (see Clause 8).

g) The **msb0** and **lsb0** properties shall only be applied to an address map component (see Clause 10).

h) Some **field** properties can be dynamically assigned a value after the property has been defined. There is a column in the property table for each component which indicates whether dynamic assignment is supported for a particular property.

i) All *boolean* **field** properties are **false** by default.

j) 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`.

7.3 Using scalar and array field instances

Fields, like all structural components in SystemRDL, can be instanced as scalars and arrays. 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_instance_name [**[***number***]** | **[***number* **:** *number***]**]**;**

where

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

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

3) *number* is a simple decimal or hexadecimal number.

i) **[***number***]** specifies the size of the instantiated component array.

ii) **[***number* **:** *number***]** can only be used for **field** or **signal** (see Clause 6) components.

b) for *anonymous field instantiation*

field {[*property***;**]***}** *field_instance_name* [**[***number***]** | **[***number* **:** *number***]**]**;**

where

1) each *property* is specified as a `name=value` pair, e.g., `name="foo"` (see 5.1.3.1).

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

3) *number* is a simple decimal or hexadecimal number.

i) **[***number***]** specifies the size of the instantiated component array.

ii) **[***number* **:** *number***]** can only be used for **field** or **signal** (see Clause 6) components.

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 8.1, which describes the register component. Single element arrays may be treated by a SystemRDL compiler as a scalar or an array.

7.4 Field access properties

The combination of field access properties specified for a **field** component determines the component's behavior. Table 8 lists the available field access properties and describes how they are implemented.

Table 8—Field access properties

Property	Behavior/Application	Type	Dynamic[a]
hw = {rw | **wr** | **r** | **w** | **na}**	Design's ability to sample/update a **field**.	*access Type*	No
sw = {rw | **wr** | **r** | **w** | **na}**	Programmer's ability to read/write a **field**.	*access Type*	Yes

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

7.4.1 Semantics

a) All **field**s are given full 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) The standard implementation behavior is based on the combination of read and write properties shown in Table 9.

Table 9—Field behavior based on properties

Software	Hardware	Code sample	Implementation
R+W	R+W	`field f { sw = rw; hw = rw; };`	Flip-flop
R+W	R	`field f { sw = rw; hw = r; };`	Flip-flop
R+W	W	`field f { sw = rw; hw = w; };`	Flip-flop
R+W	-	`field f { sw = rw; hw = na; };`	Flip-flop
R	R+W	`field f { sw = r; hw = rw; };`	Flip-flop
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; };`	D flip-flop
W	R	`field f { sw = w; hw = r; };`	D flip-flop
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; };`	Warning – no software access
-	R	`field f { sw = na; hw = r; };`	Warning – no software access
-	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.

7.4.2 Example

See Table 9.

7.5 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.

a) An anonymous definition of a **field** component with reset specified appears as follows.

 field {[*property*; ...]} *field_instance_name* = *sizedNumeric*;

 where

 1) each *property* is specified as a `name=value` pair, e.g., `name="foo"` (see 5.1.3.1).

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

b) An instantiation of a definitive definition of a **field** component with reset specified appears as follows.

 field_type field_instance_name [**[***number***]** | **[***number* **:** *number***]**] = *sizedNumeric*;

 where

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

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

 3) *number* is a simple decimal or hexadecimal number.

  i) **[***number***]** specifies the size of the instantiated component array.

  ii) **[***number* **:** *number***]** can only be used for **field** or **signal** (see Clause 6) components.

Table 10 defines the hardware signal properties.

Table 10—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	Power on reset value for the **field**.	*numeric* or *reference*	Yes
resetsignal	Name of reset signal (the default is **RESET**).	*reference*	Yes

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

7.5.1 Semantics

a) Other than `0` or `0x0`, only Verilog-style integers can be used to specify the reset value of a **field**.

b) A reset value assigned to a **field** with access properties of **sw=r** and **hw=w** without having a write enable shall be reported as a warning.

7.5.2 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 POR value of b from 101 to 001
```

7.6 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**).

Table 11 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 11—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
woset	Write one to set (`field = field | write_data`).	*boolean*	Yes
woclr	Write one to clear (`field = field & ~write_data`).	*boolean*	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 software written or cleared.	*boolean* or *reference*	Yes
swacc	Assert when field is software accessed.	*boolean* or *reference*	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.

7.6.1 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** counts as a software write, properties like **swmod** cause the resulting output to be asserted during software reads.

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 as **rclr** (see Table 11) are treated as **rw** in all cases, even if a field is specified as read-only by software. The **rclr** operation is effectively a special form of write, as a field's value is cleared as a direct result of the software transaction.

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) When specified, **rclr** resets a field to `0` and not its power-on value.

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

7.6.2 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;
};
```

7.7 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 11) returns the **next** value (see 7.5) to software before clearing the field. When not write-enabled, the current value is used instead since the "next" value is the current value. 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 12, 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 12—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 13 defines the hardware access properties.

Table 13—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* or *reference*	Yes
ored	Logical OR of all bits in **field**.	*boolean* or *reference*	Yes
xored	Logical XOR of all bits in **field**.	*boolean* or *reference*	Yes
fieldwidth	Determines the width of all instances of the **field**. This number shall be a numeric. The default value of **fieldwidth** is 1.	*numeric*	No
hwclr	Hardware clear. This **field** need not be declared as hardware-writable.	*boolean*	Yes
hwset	Hardware set. This **field** need not be declared as hardware-writable.	*boolean*	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.	*sizedNumeric*	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.	*sizedNumeric*	Yes

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

7.7.1 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.

7.7.2 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.
};
```

7.8 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**.

7.8.1 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. By default, counters are incremented/decremented by one (`1`), but developers can specify another static or dynamic increment/decrement value. Since the validity or feasibility of an increment/decrement value is determined by the field's width (valid increment values can be of equal or smaller width than the field itself), custom increment values shall be specified after the field has been instantiated and the field's width determined. Once an increment value has been explicitly specified for an instantiated field, the value may not be changed.

Counter incrementing and decrementing in SystemRDL are controlled via the counters **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.

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 to 0
  count1->incrvalue=4'hf; // Increment the counter by 15 when incrementing
                          // count1 implies an UP counter

counter_f count2[3]; // Define a 3 bit counter from 6:4
  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 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 address map.
```

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 bitcounter 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
```

7.8.2 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 saturated counter shall never exceed the saturation value (either statically or dynamically assigned). By default, saturated counters saturate at the maximum value the counter can hold without rolling over. Assigning a static or dynamic saturated value is similar to assigning increment/decrement values, see 7.8.1.

Counters in SystemRDL may have an optional (static or dynamic) threshold value. The **threshold** property does not cap the value of a counter in the way **saturate** does; instead, threshold counters are inferred to contain an output which designates whether the counter's value exceeds the threshold. See also 7.8.1.

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 to 0
count1->incrsaturate=4'hf; // keeps the counter from counting past 4'hf

counter_f count2[3]; // Define a 3 bit counter from 6:4
count2->decrthreshold=3'b2; // provide assertion when count hits 2

counter_f count3[5]=0; // Defines a 5 bit counter from 11 to 7
count3->incrsaturate;// Implies 5'h1F by default
count3->decrsaturate; // Implies 5'h1 by default
count3->decrthreshold=5'h3;

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

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 rolled over. 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 to 1
count1->incrthreshold=5'hF;

has_overflowed = count1->overflow;
```

Table 14 defines the counter field properties.

Table 14—Counter field properties

Property	Behavior/Application	Type	Dynamic[a]
counter	Field implemented as a counter.	*boolean*	Yes
threshold	A comparison value or the result of a comparison. See also: 7.8.2.1. This is the same as **incrthreshold**.	*numeric* or *reference*	Yes
saturate	A comparison value or the result of a comparison. See also: 7.8.2.1. This is the same as **incrsaturate**.	*numeric* or *reference*	Yes
incrthresh-old	A comparison value or the result of a comparison. See also: 7.8.2.1. This is the same as **threshold**.	*numeric* or *reference*	Yes
incrsaturate	A comparison value or the result of a comparison. See also: 7.8.2.1. This is the same as **saturate**.	*numeric* or *reference*	Yes
overflow	Overflow signal asserted when counter overflows or wraps.	*reference*	Yes
underflow	Underflow signal asserted when counter underflows or wraps.	*reference*	Yes
incrvalue	Increment counter by specified value.	*numeric* 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.	*numeric*	Yes
decrvalue	Decrement counter by specified value.	*numeric 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.	*numeric*	Yes
decrsatu-rate	A comparison value or the result of a comparison. See also: 7.8.2.1.	*numeric* or *reference*	Yes
decrthresh-old	A comparison value or the result of a comparison. See also: 7.8.2.1.	*numeric* or *reference*	Yes

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

7.8.2.1 Semantics

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

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

c) The default value of **incrsaturate** is the *maximum value* (2^(number of counter bits) -1) of the counter.

d) The default value of **incrthreshold** is the *maximum value* (2^(number of counter bits) -1) of the counter.

e) The default value of **decrsaturate** is `1`.

f) The default value of **decrthreshold** is `1`.

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 exactly the threshold value specified.

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 matches exactly the saturation value specified.

7.8.2.2 Example

See *Examples 1 - 3* in 7.8.2.

7.9 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 9, Clause 10, and the hierarchical interrupt example in 14.2). 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** keyword. The **nonsticky** modifier can be used in conjunction with **posedge**, **negedge**, **bothedge**, and **level**.

The syntax for a interrupt property modifiers appears as follows.

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

Table 15 lists and describes the available interrupt types.

Table 15—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.

Further more there are additional interrupt properties that can be used to mask or enable an interrupt. The **enable**, **mask**, **haltenable**, and **haltmask** keywords (see Table 16) 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 14.2.
```

Example

```
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;
};                                                  // 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";
   // Enable
        } crc_error = 0x1;
        field {
                desc = "Enable: A Packet with an invalid length has been
   received";// Enable
        } len_error = 0x1;
};                                                // 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
};                                               // 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 16 defines the interrupt properties.

Table 16—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.

7.9.1 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.

7.9.2 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
```

7.10 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** (see 12.2).

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 **hw** accesses of type **we**, **wel**, **incrvalue**, **hwset**, and **hwclr**). This is a field-only property and does not effect the other fields in the register.

   2) **precedence** = **hw** indicates hardware takes precedence over software on accessing registers (on the **hw** accesses of type **we**, **wel**, **incrvalue**, **hwset**, and **hwclr**). This is a field-only property and does not effect 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 the SystemRDL specification and such behavior is undefined.

Table 17 details the miscellaneous field properties.

Table 17—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 when contention occurs (**sw**).	**hw** | **sw**	Yes

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

7.10.1 Semantics

a) An **encode** property shall be assigned to an **enum** type.

b) The field's width and the enumeration's width shall be identical when assigning the **encode** property to the **enum**eration.

7.10.2 Example

This example shows **precedence** and **encode**.

```
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;
```

8. 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 9 and Clause 10).

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. External registers are used for complex or proprietary memory (e.g., RAM and ROM).

— **alias** allows software to access another register with different properties (i.e., **read**, **write**, **woclr**, etc.). Alias registers are typically used for debug registers, where designers want to allow backdoor access to registers and memories that should not be touched by software, except for testing. SystemRDL allows designers to specify alias registers for internal or external registers.

8.1 Defining and instantiating registers

Register components (**reg**) have the same definition and instantiation syntax as other SystemRDL components; see 5.1.

8.1.1 Semantics for all registers

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

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

c) Field instances shall not occupy overlapping bit positions within a register.

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

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

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

8.2 Instantiating internal registers

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

8.2.1 Semantics

Registers shall be instantiated as internal registers by placing the **internal** keyword before the register type name or instantiating the component as described in 5.1.

a) A *definitive definition* of an internal register appears as follows.

reg *reg_name* **{**[*reg_body***;**]***};**

[**internal**] *reg_name reg_instance* **[***number***];**

where

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

2) *reg_body* is one or more of the following:

   i) a valid register property
   ii) a component definition for a **field**, **signal**, or **enum** component
   iii) a component instantiation for a **field** or a **signal**.
3) *reg_instance* is the user-specified name for instantiation of the component.
4) *number* is a simple decimal or hexadecimal number.
   [*number***]** specifies the size of the instantiated component array.

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

   [**internal**] **reg** **{**[*reg_body***;**]*****}** *reg_instance* **[***number***]**;

   where

   1) *reg_body* is one or more of the following:
      i) a valid register property
      ii) a component definition for a **field**, **signal**, or **enum** component
      iii) a component instantiation for a **field** or a **signal**.
   2) each *property* is specified as a `name=value` pair, e.g., `name="foo"` (see 5.1.3.1).
   3) *reg_instance* is the user-specified name for instantiation of the component.
   4) *number* is a simple decimal or hexadecimal number.
      [*number***]** specifies the size of the instantiated component array.

8.2.2 Example

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

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

8.3 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.

8.3.1 Semantics

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 5.1.

a) A *definitive definition* of an external register appears as follows.

   reg *reg_name* **{**[*reg_body***;**]*****};**

   external *reg_name reg_instance* **[***number***]**;

   where

   1) *reg_name* is the user-specified **register** name.
   2) *reg_body* is one or more of the following:
      i) a valid register property
      ii) a component definition for a **field**, **signal**, or **enum** component
      iii) a component instantiation for a **field** or a **signal**.
   3) *reg_instance* is the user-specified name for instantiation of the component.

4) *number* is a simple decimal or hexadecimal number.
   [*number***]** specifies the size of the instantiated component array.

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

   external reg {[*reg_body***;**]***}** *reg_instance* **[***number***];**

   where

   1) *reg_body* is one or more of the following:
      i) a valid register property
      ii) a component definition for a **field**, **signal**, or **enum** component
      iii) a component instantiation for a **field** or a **signal**.
   2) each *property* is specified as a `name=value` pair, e.g., `name="foo"` (see 5.1.3.1).
   3) *reg_instance* is the user-specified name for instantiation of the component.
   4) *number* is a simple decimal or hexadecimal number.
      [*number***]** specifies the size of the instantiated component array.

8.3.2 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
```

8.4 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, and typically are, different from the primary register.

8.4.1 Semantics

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

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

   reg *reg_name* **{**[*reg_body***;**]***};**

   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_body* is one or more of the following:
      i) a valid register property
      ii) a component definition for a **field**, **signal**, or **enum** component
      iii) a component instantiation for a **field** or a **signal**.

3) *reg_instance* is the user-specified name for instantiation of the component.
4) *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 name as a **field** in the primary register 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) Hardware-related properties on aliases shall not be modified.

8.4.2 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_r event1;
    // Create an alias for the DV team to use and modify its properties
    // so that DV can for interrupt events and allow more rigorous structural
    // testing of the interrupt.
    alias event1 some_intr_r event1_for_dv;
        event1_for_dv.some_event->sw=rw;
        event1_for_dv.some_event->woclr = false;
};
```

8.5 Register properties

Table 18 lists and describes the register properties.

Table 18—Register properties

Property	Implementation/Application	Type	Dynamic[a]
regwidth	Specifies the bit-width of the register (power of two).	*numeric*	No
accesswidth	Specifies the minimum software access width (power of two) operation that may be performed on the register.	*numeric*	Yes
errextbus	The associated register has error input.	*numeric*	Yes
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.	*reference*	Yes
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.	*reference*	Yes
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 10.4 for more information.	*boolean*	No

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

8.5.1 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 a non-clear on read field are valid.

f) Any field that is software-writable or clear on read shall not span multiple software accessible sub-words (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 10.3).

h) **errextbus** is only valid for external registers.

8.5.2 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;
};
```

8.6 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 21). Users who specifically specify bit indexes when instantiating fields in registers do not need to specify one of these attributes, as the explicit indices imply one of these bit ordering schemes. See also Clause 14.

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 *unsizedNumeric*s;

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, understanding how a SystemRDL compiler assigns **field**s is important. Fields are packed contiguously, end-to-end, starting at index `0` for **lsb0** registers and starting at index **regwidth**`-1` in **msb0** registers.

8.6.1 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`.

8.6.2 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 3:1
                    // 4 bits from 7 to 4 are reserved and unused
       field {} C[15:8]; // 8 Bits from 15 to 8
       field {} C[5]; // 5 Bits from 20 to 16
};
```

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
};
```

8.7 Understanding interrupt registers

As discussed in 7.9, 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.

8.7.1 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.

8.7.2 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_satus_reg.a->next = some_intr_reg->intr;
```

9. 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 10) is an **addrmap** defines an RTL implementation boundary where the **regfile** does not. Consequently, all the address allocation support described in this clause also applies to address maps. Since **addrmap**s define a implementation block boundary, there are some specific properties that are only specified for address maps (see Clause 10) and not specified for **regfile**s.

9.1 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 definition* of a register (file) instantiation appears as follows.

   regfile *regfile_name* **{**[*regfile_body***;**]***};**

   [**internal** | **external**] *regfile_name regfile_instance* [**[***number***]**] [*addr_alloc*]**;**

   where

   1) *regfile_name* is the user-specified **regfile** name.
   2) *regfile_body* is one or more of the following:
      i) a valid register file property
      ii) a component definition for a **field**, **reg**, **regfile**, **signal**, or **enum** component
      iii) a component instantiation for a **reg**, **regfile**, or **signal**.
   3) *regfile_instance* is the user-specified name for instantiation of the component.
   4) *number* is a simple decimal or hexadecimal number;
      [*number***]** specifies the size of the instantiated component array.
   5) *addr_alloc* is an address allocation operator (see Table 19).

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

   [**internal** | **external**] [**regfile**] **{**[*regfile_body***;**]***}**
   regfile_instance [**[***number***]**] [*addr_alloc*]**;**

   where

   1) *regfile_body* is one or more of the following:
      i) a valid register file property
      ii) a component definition for a **field**, **reg**, **regfile**, **signal**, or **enum** component
      iii) a component instantiation for a **reg**, **regfile**, or **signal**.
   2) *regfile_instance* is the user-specified name for instantiation of the component.
   3) *number* is a simple decimal or hexadecimal number;
      [*number***]** specifies the size of the instantiated component array.
   4) *addr_alloc* is an address allocation operator whose syntax is specified as follows (see also Table 19).

      [**@***unsizedNumeric* | **%=** *unsizedNumeric*] [**+=***unsizedNumeric*]

When instantiating **register**s or **regfile**s in a **regfile**, the address may be assigned using one of the address allocation (*addr_alloc*) operators in Table 19.

Table 19—Address allocation operators

Property	Implementation/Application
@ *unsizedNumeric*	Specifies a specific address for the component instance.
+= *unsizedNumeric*	Specifies the address increment when instantiating an array of components (controls the spacing of the components). The address increment is relative to the previous instance's address.
%= *unsizedNumeric*	Specifies the alignment of the next address when instantiating a component (controls the alignment of the components). The address increment is relative to the previous instance's address.

9.1.1 Semantics

a) Addresses in SystemRDL are always byte addresses.

b) Within a register file, the only components that can be instantiated shall be a register file, a register, and/or signal components.

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

d) Addresses are assigned in incrementing order.

e) The operator **%=** is a more localized version of **alignment** (see Table 20).

f) Only *unsizedNumeric*s may be used for address specification (see Table 4).

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

h) The @ and **%=** operators are mutually exclusive.

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

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

k) If a **regfile** is declared **internal**, all registers in it other than **alias** registers 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.

9.1.2 Examples

The following set of examples demonstrate the usage of the operators defined in Table 19. 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 10), with the default **regwidth**=32.

Example 1

Using the @ operator.

```
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.

```
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 will consume 160-12 bytes of space
                              // Address 0x100 to 0x103, 0x110 to 0x113,....
};
```

Example 3

Using the **%=** operator.

```
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[4] @0x100 += 0x10; // These will consume 64-12 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 0x180 since
                       // that is the first address satisfying address>=0x134
                       // and ((address % 0x80) == 0)
};
```

9.2 Register file properties

Table 20 lists and describes the register file properties.

Table 20—Register file properties

Property	Implementation/Application	Type	Dynamic[a]
alignment	Specifies alignment of all instantiated components in the associated register file.	*unsized-Numeric*	No
sharedext-bus	Forces all external registers to share a common bus.	*boolean*	No

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

9.2.1 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.

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

9.2.2 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;
```

10. Address map component

An address component map (**addrmap**) contains registers, register files, 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. Address allocation support in **addrmap**s is identical to **regfile**s (see Clause 9), except for the exceptions noted in this clause.

10.1 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.

10.2 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 Table 19.

10.2.1 Semantics

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

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

c) If a register, register file, or address map instance is not explicitly assigned an address, a compiler implementing SystemRDL needs to automatically assign the address. This address is determined by the **alignment** and **addressing** properties (see Table 19), and the **+=** and **%=** operators (see Table 21).

10.2.2 Example

See the examples in 9.1.2.

10.3 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**.

a) **compact**

Specifies the components are packed tightly together while still being aligned to the **accesswidth** parameter.

Example 1

Sets accesswidth=32

```
addrmap some_map {
  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 {
  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 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
};
```

Table 21 describes the address map properties.

Table 21—Address map properties

Property	Implementation/Application	Type	Dynamic[a]
alignment	Specifies alignment of all instantiated components in the address map.	*unsized-Numeric*	No
sharedext-bus	Forces all external registers to share a common bus.	*boolean*	No
bigendian	Uses big-endian architecture in the address map.	*boolean*	Yes
littleendian	Uses little-endian architecture in the address map.	*boolean*	Yes
addressing	Controls how addresses are computed in an address map.	*address-ingType*	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 **rsvd-setX** is *True*. **rsvdsetX** takes precedence over **rsvdset**.	*boolean*	No
msb0	Specifies register bit-fields in an address map are defined as `0:N` versus `N:0`. This property effects 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 effects all fields in an address map. This is the default.	*boolean*	No

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

10.3.1 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 dealing with multiple external components.

d) **regalign** is identical to **compact**, except when dealing with **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.

e) **fullalign** is identical to **compact**, except when dealing with **regfiles** 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`.

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

g) **msb0** and **lsb0** are mutually exclusive.

h) **msb0** and **lsb0** are required for address maps that use relative size versus explicit indexes for fields in registers.

i) **msb0** and **lsb0** are also required on address maps that instance registers defined in the global scope.

j) The **bigendian** and **littleendian** properties are used for controlling byte ordering in generated code and have no effect on bit ordering or the SystemRDL code itself. These properties are purely information for a back-end generator.

10.3.2 Example

See the examples shown in 10.3.

10.4 Defining bridges or multiple view address maps

There are often scenarios in modern designs where a register needs to be connected to two or more different buses and accessed differently; Table 22 lists and describes these address map bridge properties.

Table 22—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
arbiter	Specifies the arbiter to be used between the different software interfaces to the register. This shall only be applied to an address map.	*boolean*	No

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

10.4.1 Semantics

a) 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.

b) **arbiter** specifies the HDL module used to arbitrate between control interfaces in a bridge.

10.4.2 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
    arbiter = "round_robin"; // This will instance external verilog module
                            // round_robin as the arbiter
  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
```

11. 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.

11.1 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) *attributes* 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.

Table 23 specifies which attributes can be set in a user-defined property. Table 24 details each of the possible user-defined property types.

Table 23—Attributes for user-defined properties

Attribute	Description	Allowable values
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**, or **all**.
type	The type of the property. This attribute shall be one of the allowable values. See Table 24.	**string**, **number**, **boolean**, or **ref**.
default	The default value for the property.	Optional; needs to match the **type** of the property.

Table 24—User-defined types

Type	Description	Example
number	A numeric value (see Table 4).	`0x10` or `8h'8C`
string	Any valid string (see Table 4).	`"Some String"`
boolean	A two-state value (see Table 4).	`true` or `false`
ref	A reference to a component instance or an instance's property (see Table 4).	`chip.block.reg.field` or `-> outback_steakhouse`

11.1.1 Semantics

a) User-defined properties are global and defined in the root scope.

b) A user-defined property definition shall also define to which **component**s the property can bind.

c) A user-defined property definition shall include its **type** definition (see Table 24).

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 **false** creates an inconsistency with respect to SystemRDL property assignments.

11.1.2 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; };
```

11.2 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.

11.2.1 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.

11.2.2 Example

This example shows the definition and assignment of several user-defined properties.

```
property a_map_p { type = string; component = addrmap | regfile; };
property some_bool_p { type = boolean; component = field; default = false; };
property some_ref_p { type = reference; 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
};
```

12. Enumeration (bit-field encoding)

12.1 Introduction

An *enumeration* is a set of named values that provides mnemonic names for field values. There are no properties for the **enum** component beyond the universal properties described in 5.2.1.

12.2 Defining enumerations

Unlike other SystemRDL components, bit-field encodings are not instantiated, rather they are assigned to a field's **encode** property (see 7.10). Bit-field encodings can only be defined definitively; anonymous definitions are not allowed.

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 a *sizedNumeric*.

3) All *value*s shall be the same size.

4) All *value*s shall be unique.

5) *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;
};
```

13. 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.

13.1 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.

13.1.1 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.

13.1.2 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];
};
```

13.2 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.

13.2.1 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 25 shows which preprocessor are included in SystemRDL.

Table 25—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
`` `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 25.

13.2.2 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.

14. Advanced topics in SystemRDL

The concept of signals was introduced in Clause 6 and the signal properties were described in 6.2. 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 7) and their application.

14.1 Application of signals for reset

A SystemRDL compiler, by default, creates a single reset signal (**resetsignal**) for a generated RTL block called **RESET**. This *reset signal* shall be positive active and is used to synchronously reset flip-flops. Clearly, there are many other 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 effect 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.

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.

The following examples highlight two different ways to customize reset behavior.

Example 1

This example shows usage of **resetsignal**.

```
signal { activelow; async; } reset_l; // Define a single bit signal

reg {
  field {} field=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;
```

14.2 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 1).

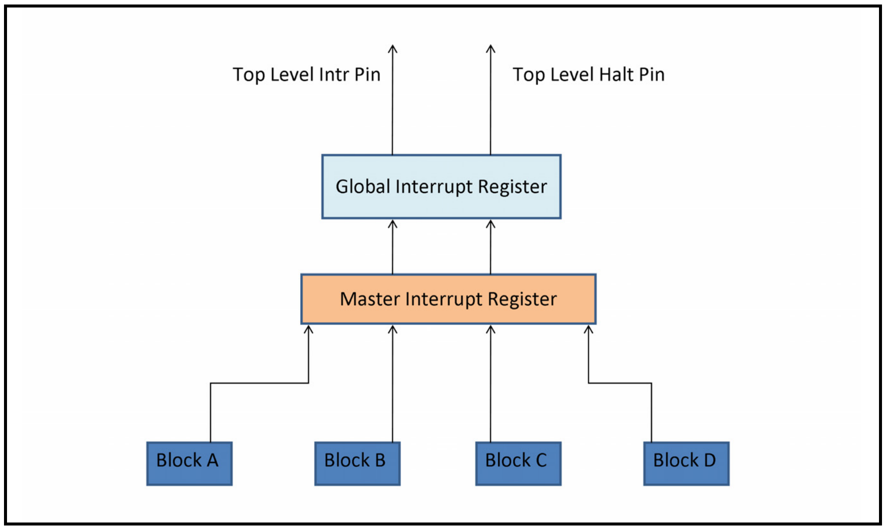

Figure 1—Hierarchical interrupt structure

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 1 is represented by the example shown in Figure 2.

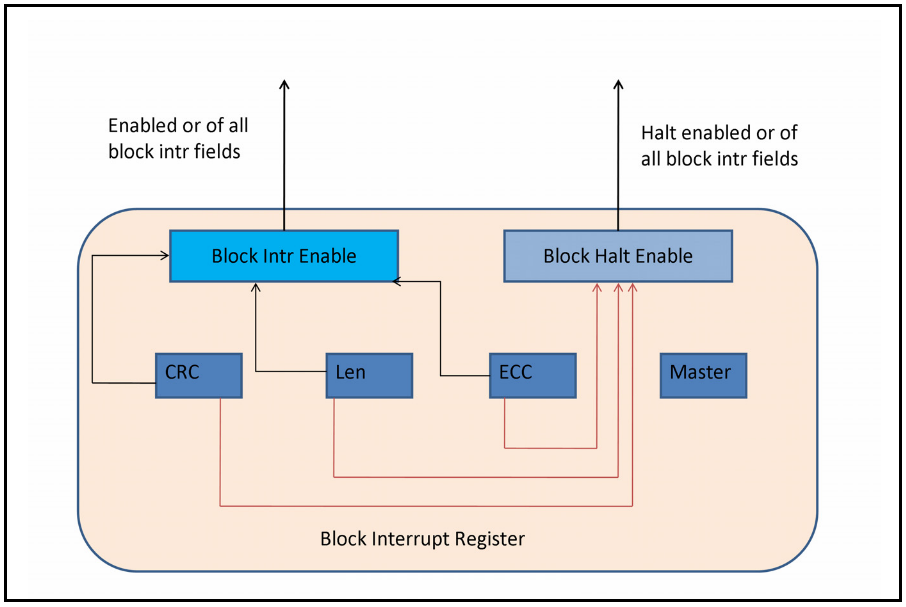
Enabled or of all block intr fields
Halt enabled or of all block intr fields
Block Intr Enable
Block Halt Enable
CRC
Len
ECC
Master
Block Interrupt Register

Figure 2—Block interrupt example

Multiple levels of hierarchy are needed to effectively to 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.

14.2.1 Example structure and perspective

The (example) SystemRDL code needed to match the hierarchical interrupt structure shown in Figure 1 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.

14.2.2 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
```

14.2.3 Code snippet 2

This next code snippet only defines the enable register associated with the interrupt register from 14.2.2—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
```

14.2.4 Code snippet 3

This next code snippet only defines a second-priority enable register associated with the interrupt register from 14.2.2—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
```

14.2.5 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;
  };
```

14.2.6 Code snippet 5

This next code snippet defines the enable register for the master interrupt register set in 14.2.5.

```
  //
  // 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;
  };
```

14.2.7 Code snippet 6

This next code snippet defines an alternate enable register for the master interrupt register set in 14.2.5.

```
//------------------------------------------------------------
// 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;
};
```

14.2.8 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 **interrupt** 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;
};
```

14.2.9 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;
```

14.2.10 Code snippet 9

14.2.9 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;
```

14.2.11 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;
```

14.2.12 Code snippet 11

This final section of the example instantiates a single top-level interrupt register containing a single **interrupt** 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

  };
```

14.3 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.

14.3.1 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 homogenous **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;
}
```

14.3.2 Byte ordering

Byte ordering is another common source of confusion. Byte order is often called *endianness*. 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** parameter. 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 effect 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] *IP-XACT release*, v1.4, see http://www.spiritconsortium.org/releases/1.4.

[B3] *SystemRDL Examples*, v1.0, see http://www.spiritconsortium.org/doc_downloads/.

[B4] *Endianess References*, see http://en.wikipedia.org/wiki/Endianess and http://3bc.bertrand-blanc.com/endianness05.pdf.

Annex B

(normative)

Grammar

The following shows lexer grammar based on the ANTLR Parser Generator Reference Manual, Version 2. 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.

Parser grammar:

```
root
    :   (   component_def
        |   enum_def
        |   explicit_component_inst
        |   property_assign
        |   property_definition
        )*
        EOF
    ;


component_def
    :   (   "addrmap"
        |   "regfile"
        |   "reg"
        |   "field"
        |   "signal"
        )
        (   id
        |
        )
        LBRACE
        (   component_def
        |   explicit_component_inst
        |   property_assign
        |   enum_def
        )*
        RBRACE
        (   anonymous_component_inst_elems
        |
        )
        SEMI
    ;


enum_def
    :   "enum" id enum_body SEMI
    ;


explicit_component_inst
    :   (   "external"
        |
        )
        (   "internal"
        |
        )
        (   "alias" id
        |
        )
        id component_inst_elem ( COMMA component_inst_elem )* SEMI
    ;


property_assign
    :   default_property_assign SEMI
    |   explicit_property_assign SEMI
    |   post_property_assign SEMI
    ;


property_definition
    :   "property" id LBRACE property_body RBRACE SEMI
    ;


id
    :   ID
    |   INST_ID
    ;


property_body
    :   property_type
        (   property_usage
            (   property_default
            |
            )
        |   property_default property_usage
        )
    |   property_usage
        (   property_type
            (   property_default
            |
            )
        |   property_default property_type
        )
    |   property_default
        (   property_type property_usage
        |   property_usage property_type
        )
    ;


property_type
    :   "type" EQ
        (   property_string_type
        |   property_number_type
        |   property_boolean_type
        |   property_ref_type
        )
        SEMI
    ;
```

```
property_usage
    :   "component" EQ property_component ( OR property_component )* SEMI
    ;


property_default
    :   ( "default" EQ
            (   str
            |   num
            |   "true"
            |   "false"
            )
            SEMI )
    ;


property_string_type
    :   "string"
    ;


property_number_type
    :   "number"
    ;


property_boolean_type
    :   "boolean"
    ;


property_ref_type
    :   (   "addrmap"
        |   "reg"
        |   "regfile"
        |   "field"
        |   "ref"
        )
    ;


str
    :   STR
    ;


num
    :   NUM
    ;


property_component
    :   (   "signal"
        |   "addrmap"
        |   "reg"
        |   "regfile"
        |   "field"
        |   "all"
        )
    ;


anonymous_component_inst_elems
    :   (   "external"
        |
        )
        component_inst_elem ( COMMA component_inst_elem )*
    ;


component_inst_elem
    :   id
        (   array
        |
        )
        (   EQ num
        |
        )
        (   AT num
        |
        )
        (   INC num
        |
        )
        (   MOD num
        |
        )
    ;


array
    :   LSQ num
        (   COLON num
        |
        )
        RSQ
    ;


instance_ref
    :   instance_ref_elem ( DOT instance_ref_elem )*
        (   DREF property
        |
        )
    ;


instance_ref_elem
    :   id
        (   LSQ num RSQ
        |
        )
    ;


property
    :   "name"
```

```
|    "desc"
|    "arbiter"
|    "rset"
|    "rclr"
|    "woclr"
|    "woset"
|    "we"
|    "wel"
|    "swwe"
|    "swwel"
|    "hwset"
|    "hwclr"
|    "swmod"
|    "swacc"
|    "sticky"
|    "stickybit"
|    "intr"
|    "anded"
|    "ored"
|    "xored"
|    "counter"
|    "overflow"
|    "sharedextbus"
|    "errextbus"
|    "reset"
|    "littleendian"
|    "bigendian"
|    "rsvdset"
|    "rsvdsetX"
|    "bridge"
|    "shared"
|    "msb0"
|    "lsb0"
|    "sync"
|    "async"
|    "cpuif_reset"
|    "field_reset"
|    "activehigh"
|    "activelow"
|    "singlepulse"
|    "underflow"
|    "incr"
|    "decr"
|    "incrwidth"
|    "decrwidth"
|    "incrvalue"
|    "decrvalue"
|    "saturate"
|    "decrsaturate"
|    "threshold"
|    "decrthreshold"
|    "dontcompare"
|    "donttest"
|    "internal"
|    "alignment"
|    "regwidth"
|    "fieldwidth"
|    "signalwidth"
|    "accesswidth"
```

```
        |   "sw"
        |   "hw"
        |   "addressing"
        |   "precedence"
        |   "encode"
        |   "resetsignal"
        |   "clock"
        |   "mask"
        |   "enable"
        |   "hwenable"
        |   "hwmask"
        |   "haltmask"
        |   "haltenable"
        |   "halt"
        |   "next"
        |   PROPERTY
        ;


    default_property_assign
        :   "default" explicit_property_assign
        ;


    explicit_property_assign
        :   property_modifier property
        |   property ( EQ property_assign_rhs )
        ;


    post_property_assign
        :   instance_ref ( EQ property_assign_rhs )
        ;


    property_modifier
        :   "posedge"
        |   "negedge"
        |   "bothedge"
        |   "level"
        |   "nonsticky"
        ;


    property_assign_rhs
        :   property_rvalue_constant
        |   "enum" enum_body
        |   instance_ref
        |   concat
        ;


    property_rvalue_constant
        :   "true"
        |   "false"
        |   "rw"
        |   "wr"
        |   "r"
        |   "w"
        |    "na"
        |    "compact"
        |    "regalign"
        |    "fullalign"
        |    "hw"
        |    "sw"
        |    num
        |    str
        ;


    enum_body
        :    LBRACE ( enum_entry )* RBRACE
        ;


    concat
        :    LBRACE concat_elem ( COMMA concat_elem )* RBRACE
        ;


    concat_elem
        :    instance_ref
        |    num
        ;


    enum_entry
        :    id EQ num
             (    LBRACE ( enum_property_assign )* RBRACE
             |
             )
             SEMI
        ;


    enum_property_assign
        :    (    "name"
             |    "desc"
             )
             EQ str SEMI
        ;
```

Lexer grammar:

```
    mWS
    |    mSL_COMMENT
    |    mML_COMMENT
    |    mID
    |    mNUM
    |    mSTR
    |    mLBRACE
    |    mRBRAC
    |    mLSQ
    |    mRSQ
    |    mLPAREN
    |    mRPAREN
```

```
|    mAT
|    mOR
|    mSEMI
|    mCOLON
|    mCOMMA
|    mDOT
|    mDREF
|    mEQ
|    mINC
|    mMOD
protected mLETTER
     :('a'..'z'
     |'A'..'Z'
     )
     ;

mWS
     :(' '
     |'\t'
     |('\n'
     |'\r'
     |"\r\n"
     )
     )
     ;

mSL_COMMENT
     :"//" ( ('\n'
     |'\r'
     ) )*
     ('\n'
     |'\r'
     |"\r\n"
     )
     ;

mML_COMMENT
     :"/*"
     ('*'
     |'\n'
     |'\r'
     |"\r\n"
     |('*'
     |'\n'
     |'\r'
     )
     )*
     "*/"
     ;

mID
     :('\\'
     |
     )
     (mLETTER
     |'_'
     )
     (mLETTER
     |'_'
```

```
          |'0'..'9'
          )*
          ;

   protected mVNUM
          :'\''
          ('b'
          ('0'
          |'1'
          |'_'
          )+
          |'d'
          ('0'..'9'
          |'_'
          )+
          |'o'
          ('0'..'7'
          |'_'
          )+
          |'h'
          ('0'..'9'
          |'a'..'f'
          |'A'..'F'
          |'_'
          )+
          )
          ;

   mNUM
          :( '0'..'9' )*
          (mVNUM
          |( '0'..'9' )
          )
          |"0x"
          ('0'..'9'
          |'a'..'f'
          |'A'..'F'
          )+
          ;

   protected mESC_DQUOTE
          :"\\\""
          ;

   mSTR
          :'"'
          (('"'
          |'\n'
          |'\\'
          )
          |mESC_DQUOTE
          |'\n'
          )*
          '"'
          ;

   mLBRACE
          :'{'
          ;
```

```
mRBRAC
     :'}'
     ;

mLSQ
     :'['
     ;

mRSQ
     :']'
     ;

mLPAREN
     :'('
     ;

mRPAREN
     :')'
     ;

mAT
     :'@'
     ;

mOR
     :'|'
     ;

mSEMI
     :';'
     ;

mCOLON
     :':'
     ;

mCOMMA
     :','
     ;

mDOT
     :'.'
     ;

mDREF
     :"->"
     ;

mEQ
     :'='
     ;

mINC
     :"+="
     ;

mMOD
     :"%="
```

;

Annex C

(informative)

Code example

The following code sample provides an example of a specification written in SystemRDL.

```
//============================================================================
//
// Program  : generic_example.rdl
// Language : Register Description Language (RDL)
// Purpose  : This is a generic example designed to show a number of the
//            RDL Language Features...
//
//============================================================================

signal gen_reset_signal_type { // Define a generic reset signal type
   name="Generic Reset Signal";
   desc="This is a generic reset signal used to reset";
};

gen_reset_signal_type generic_reset; // Instance the Generic Reset Signal

//
// This example shows the concept of a register file
// A register file is a group of registers that belong together...
// Now we can easily instance multiple fifo status registers very easily...
//

regfile fifoRfile {
  reg pointerReg { field { we; hwmask;} data[31:0]; };

  reg fifoStatusReg {
    field {} full;
    field {} empty;
    field {} almost_empty[4:4];
    field {} almost_full[5:5];

    full->reset  = 1'b0;
    full->resetsignal = generic_reset;
    // Just the full signal uses generic reset. Others use reset...
    empty->reset = 1'b1;
    almost_empty->reset = 1'b1;
    almost_full ->reset = 1'b0;
  };

  pointerReg head; // Define a register pointing to the head of the fifo

  head->resetsignal = generic_reset;
  // Assign an alternate reset to register head

  pointerReg tail;      // Define a register pointing to the tail of a fifo
  fifoStatusReg status; // Define a register for the Fifo's Status
};
```

```
// This example shows using perl to do anything you desire

field myField {
  desc = "My example 2bit status field";
  rclr;                                                   // Read to Clear
};

// An example of Apache's ASP standard for embedding Perl
reg myReg {
  <% $num_fields = 16;
     for( $i = 0; $i < $num_fields*2; $i += 2 ) { %>
    myField data<%=$i/2%> [<%=$i+1%>:<%=$i%>];
    data<%=$i/2%>->reset = 2'd<%=$i/2%4%>;
  <% } %>
};

//
// Enumeration Example
//
enum link_status_enum {
  not_present = 4'd0  { desc = "No link peer is currently detected"; };
  training    = 4'd1  { desc = "Link is currently training"; };
  snooze      = 4'd5  { desc = "Link is in a partial low power state"; };
  sleep       = 4'd6  { desc = "Link is a Full low power state"; };
  wake        = 4'd7  { desc = "Link is waking up from snooze or sleep state"; };
  active      = 4'd10 { desc = "Link is operating normally"; };
};

field link_status_field {
  hw     = rw;
  sw     = r;
  desc   = "Status of a Serdes Link";
  encode = link_status_enum;
  fieldwidth = 4;
};

reg serdes_link_status_reg {
  link_status_field port0; // Instance 4 ports of Link Status
  link_status_field port1;
  link_status_field port2;
  link_status_field port3;
};

//
// Counter Example
//

field count_field {                                    // Anonymous Generic
   Counter definition.
     hw   = r; sw = rw; rclr; counter;
     desc = "Number of certain packet type seen";
};

reg gige_pkt_count_reg {
  count_field port0[31:24];
  count_field port1[23:16];
  count_field port2[15:8];
  count_field port3[7:0];
};
```

```
reg spi4_pkt_count_reg {
  count_field port0[31:16];
  count_field port1[15:0];
  port0->threshold = 16'hCFFF;
  port1->threshold = 16'hCFFF;
};

reg vc_pkt_count_reg {
  count_field vc_count[30:0];
  field { desc="VC is Active"; stickybit; } active;
  active->reset = 1'b1;
  vc_count->reset = 31'h0;
};


addrmap some_register_map {

  name = "RDL Example Registers";
  desc = "This address map contains some example registers to show
          how RDL can be utilized in various situations.";

  //
  // This register is a inline register definition.
  // It defines a simple ID register.  No flip-flop is implemented
  //
  reg chip_id {
     name = "This chip part number and revision #";
     desc = "This register contains the part # and revision # for XYZ ASIC";

     field {
       hw   = w; // This combination of attributes creates an input port for
       sw   = r; // hardware to set the part num external to the reg block
       desc = "This field represents the chips part number";
     } part_num[31:4] = 28'h12_34_56_7; // Verilog Style number with _'s

     field {
      hw   = na; // This combination creates the ID num as a constant internal
       sw   = r;  // to the reg block
       desc = "This field represents the chips revision number";
     } rev_num[3:0] = 4'b00_01; // Verilog Style number with _'s
  }; // End chip_id register definition

  // Create an Instance of CHIP_ID type called chip_id_reg at Addr=0;
  external chip_id chip_id_reg @0x0000;

  serdes_link_status_reg link_status; // Instance a reg. Auto Address

 myReg            myRegInst          @0x0010; // This instance starts at 0x10

  spi4_pkt_count_reg spi4_pkt_count      @0x0020;
  gige_pkt_count_reg gige_pkt_count_reg;

  // Create 8 Instances of Fifo Reg File Starting at Address=0x100
  fifoRfile          fifo_port[8]        @0x100 += 0x10;
  external vc_pkt_count_reg   vc_pkt_count[256]   @0x1000 +=0x10;
};                                                     // End some_register_map
```

Annex D

(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 the rules and semantics associated with these tags) see Clause 2.

D.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 outmost one is.

```
[b]Text[/b]                   -- Bold

[i]Text[/i]                   -- Italic

[u]Text[/u]                   -- Underline

[color=colorValue]Text[/color]-- Color See D.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.spiritconsortium.com[/url] -- which places the target link
   the generated code.
2. [url=http://www.spiritconsortium.com]Spirit Consortium[/url] -- Which
   displays the text Spirit Consortium 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.
```

D.2 Single-tag RDLFormatCode constructs

```
[br]             -- Line break

[lb]             -- Left bracket ([)

[rb]             -- Right bracket (])

[p]              -- Paragraph begin

[sp]             -- White Space (equivalent to an HTML &nbsp;)

[index]          -- Replaced by the index # of the individual component
                    instance when instantiated as an array. When representing
   an individual array element this is 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 is 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).
```

D.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%	🟦

D.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];
};
```

The complete SystemRDL source and sample output from this example can be found in the SystemRDL release in the `examples/rdl_code` directory.

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 E

(informative)

Component-property relationships

Table E1 lists all properties defined in SystemRDL. For each property, Table E1 specifies which component types allow the property and gives references to the tables (or section) where the property is defined (e.g., Table 18 for the property **accesswidth** (within the *Register component description*)). The **Mutual exclusion** 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 E1 also shows the type for each property, what side of an assignment is may appear on, and if it can be dynamically assigned. The two *Constant* columns indicate if a property appears on the left-hand side (*LH assign*), the type shown can be assigned on the right-hand side (*RH assign*), e.g., a *numeric* type for the property **accesswidth**. The left-hand side in the **Signal** columns indicate if a property may be assigned a signal. The right-hand side in the **Signal** columns indicate if a property may be assigned to a signal property.

Table E1—Property cross-reference

Property	Mutual exclusion	SystemRDL components					Constant		Signal		Dynamic assignment	Notes
		Field	Register	Register file	Address map	Signal	LH assign	RH assign	LH assign	RH[a] assign		
accesswidth			Table 18				x	numeric			x	
activehigh	`A`					Table 7	x	boolean			x	
activelow	`A`					Table 7	x	boolean			x	
addressing					Table 21		x	enumera-tion value				**compact**, **regalign**, or **fullalign**
alignment				Table 20	Table 21		x	unsized numeric				
anded		Table 13					x	boolean		x	x	
arbiter					Table 22		x	boolean				

Table E1—Property cross-reference (Continued)

Property	Mutual exclusion	SystemRDL components					Constant		Signal		Dynamic assignment	Notes
		Field	Register	Register file	Address map	Signal	LH assign	RH assign	LH assign	RH[a] assign		
async	N					Table 7	x	boolean			x	
bigendian	L				Table 21		x	boolean			x	
bothedge	H	Table 15						N/A			x	**intr** modifier
bridge					Table 22		x	boolean				
counter	E	Table 14					x	boolean			x	
cpuif_reset						Table 7	x	boolean			x	
decr		Table 14						N/A	x	y	x	
decrsaturate		Table 14					x	numeric	x		x	Decrementing counter saturate value
decrsaturate		Table 14						N/A		x	x	Decrementing counter saturate reached
decrthreshold		Table 14					x	numeric	x		x	Decrementing counter threshold value
decrthreshold		Table 14						N/A		x	x	Decrementing counter threshold reached
decrvalue	G	Table 14					x	numeric	x	y	x	
decrwidth	G	Table 14					x	numeric			x	
desc		Table 5	Table 5	Table 5	Table 5	Table 5	x	string	x			Also used in **enum**eration (Table 5)
dontcompare	O	Table 6					x	boolean sized numeric			x	

Table E1—Property cross-reference (Continued)

Property	Mutual exclusion	SystemRDL components					Constant		Signal		Dynamic assignment	Notes
		Field	Register	Register file	Address map	Signal	LH assign	RH assign	LH assign	RH[a] assign		
dontcompare	O		Table 6, Table 18	Table 6	Table 6, Table 21		x	boolean			x	
donttest	O	Table 6					x	boolean sized numeric			x	
donttest	O		Table 6, Table 18	Table 6	Table 6, Table 21		x	boolean			x	
enable	J	Table 16						N/A	x	y	x	
encode		Table 17					x	N/A			x	**enum**eration object reference
errextbus			Table 18				x	numeric			x	
field_reset						Table 7	x	boolean			x	
fieldwidth		Table 13					x	numeric				
halt			See 8.7					N/A		x	x	
haltenable	K	Table 16						N/A	x	y	x	
haltmask	K	Table 16						N/A	x	y		
hw		Table 8					x	enumeration value				**r**, **w**, **rw**, **wr**, or **na**
hwclr		Table 13					x	boolean	x	y	x	
hwenable	D	Table 13					x	boolean	x	y	x	
hwmask	D	Table 13					x	boolean	x	y	x	
hwset		Table 13					x	boolean	x	y	x	

Table E1—Property cross-reference (Continued)

Property	Mutual exclusion	SystemRDL components					Constant		Signal		Dynamic assignment	Notes
		Field	Register	Register file	Address map	Signal	LH assign	RH assign	LH assign	RH[a] assign		
incr		Table 14						N/A	x	y	x	
incrsaturate		Table 14					x	numeric	x		x	Incrementing counter saturate value
incrsaturate		Table 14						N/A		x	x	Incrementing counter saturate reached
incrthreshold		Table 14					x	numeric	x		x	Incrementing counter threshold value
incrthreshold		Table 14						N/A		x	x	Incrementing counter threshold reached
incrvalue	F	Table 14					x	numeric	x	y	x	
incrwidth	F	Table 14					x	numeric			x	
intr	E	Table 16					x	boolean			x	
intr			See 8.7					N/A		x	x	
level	H	Table 15						N/A			x	**intr** modifier
littleendian	L				Table 21		x	boolean			x	
lsb0	M				Table 21		x	boolean				
mask	J	Table 16						N/A	x	y	x	
msb0	M				Table 21		x	boolean				
name		Table 5	Table 5	Table 5	Table 5	Table 5	x	string			x	Also used in **enum**eration (Table 5)
negedge	H	Table 15						N/A			x	**intr** modifier
next		Table 10						N/A	x	y	x	

Table E1—Property cross-reference (Continued)

Property	Mutual exclusion	SystemRDL components					Constant		Signal		Dynamic assignment	Notes
		Field	Register	Register file	Address map	Signal	LH assign	RH assign	LH assign	RH[a] assign		
nonsticky	`I`	Table 15						N/A			x	**intr** modifier
ored		Table 13					x	boolean		x	x	
overflow		Table 14						N/A		x	x	
posedge	`H`	Table 15						N/A			x	**intr** modifier
precedence		Table 17					x	enumeration value			x	**hw** or **sw**
rclr		Table 11					x	boolean			x	
regwidth			Table 18				x	numeric				
reset		Table 10					x	numeric	x	y	x	
resetsignal		Table 10						N/A	x	y	x	
rset		Table 11					x	boolean			x	
rsvdset					Table 21		x	boolean				
rsvdsetX					Table 21		x	boolean				
saturate		Table 14					x	numeric	x		x	Incrementing counter saturate value
saturate		Table 14						N/A		x	x	Incrementing counter saturate reached
shared			Table 18				x	boolean				
sharedextbus				Table 20	Table 21		x	boolean				
signalwidth						Table 7	x	numeric				
singlepulse		Table 11					x	boolean			x	

Table E1—Property cross-reference (Continued)

Property	Mutual exclusion	SystemRDL components					Constant		Signal		Dynamic assignment	Notes
		Field	Register	Register file	Address map	Signal	LH assign	RH assign	LH assign	RH[a] assign		
sticky	I	Table 16					x	boolean			x	
stickybit	I	Table 16					x	boolean			x	
sw		Table 8					x	enumeration value			x	**r**, **w**, **rw**, **wr**, or **na**
swacc		Table 11					x	boolean		x	x	
swmod		Table 11					x	boolean		x	x	
swwe		Table 11					x	boolean	x	y	x	
swwel		Table 11					x	boolean	x	y	x	
sync	N					Table 7	x	boolean			x	
threshold		Table 14					x	numeric	x		x	Incrementing counter threshold value
threshold		Table 14						N/A		x	x	Incrementing counter threshold reached
underflow		Table 14						N/A		x	x	
we	C	Table 13					x	boolean	x	y	x	
wel	C	Table 13					x	boolean	x	y	x	
woclr	B	Table 11					x	boolean			x	
woset	B	Table 11					x	boolean			x	
xored		Table 13					x	boolean		x	x	

[a]y indicates a RH assignment can only take place if a LH assignment also occurs; otherwise this is an error.