

1603™

IEEE Standard for an Advanced Library Format (ALF) Describing Integrated Circuit (IC) Technology, Cells, and Blocks

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee



Published by
The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

20 February 2004

Print: SH95179
PDF: SS95179

IEEE Standard for an Advanced Library Format (ALF) Describing Integrated Circuit (IC) Technology, Cells, and Blocks

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 11 September 2003

IEEE-SA Standards Board

Approved 29 December 2003

American National Standards Institute

Abstract: ALF is a modeling language for library elements used in IC technology. ALF enables description of electrical, functional, and physical models in a formal language suitable for electronic design automation (EDA) application tools targeted for design and analysis of an IC. This standard provides rules that describe ALF and how tool developers, integrators, library creators, and library users should use it.

Keywords: behavioral, block, cell, derate, EDA, electrical, format, functional, gate-level, integrated circuit, language, layout, library, modeling, physical, power, RTL, signal integrity, technology, timing

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Copyright © 2004 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 20 February 2004. Printed in the United States of America.

IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by the Institute of Electrical and Electronics Engineers, Incorporated.

Print: ISBN 0-7381-3832-0 SH95179
PDF: ISBN 0-7381-3833-9 SS95179

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

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

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

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

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

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE. Current interpretations can be accessed at the following URL: <http://standards.ieee.org/reading/ieee/interp/index.html>.

Errata, if any, for this and all other standards can be accessed at the following URL: <http://standards.ieee.org/reading/ieee/updates/errata/index.html>. Users are encouraged to check this URL for errata periodically.

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

Secretary, IEEE-SA Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331 USA

NOTE—Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Introduction

[This introduction is not part of IEEE Std 1603-2003, IEEE Standard for an Advanced Library Format (ALF) Describing Integrated Circuit (IC) Technology, Cells, and Blocks.]

The purpose of ALF is to provide a modeling language and semantics for the functional, physical, and electrical performance description of technology-specific libraries for cell-based and block-based design. Without a standard, EDA tools would be left to use tool-specific and fragmented library descriptions. The semantics would be defined by tool implementations only, which are subject to change and prone to misinterpretation. Therefore, ALF is proposed to create a consistent library view suitable as a reference for library creators and users, as well as for electronic design automation (EDA) tool developers and integrators.

IEEE Std 1603-2003 is based on the work of Open Verilog International (OVI) and its successor organization, Accellera.

The ALF standard began as the OVI Power & Synthesis Technical Steering Committee (PS-TSC) early in 1996, with the charter to define a standard library data format for synthesis, power analysis, and optimization. As the committee grew in membership, with the addition of experts in other fields such as design for test, it became clear that such a format could be easily extended to cover other design tools. Furthermore, the benefit to both silicon and EDA vendors of having a single, flexible format that would fully describe the functional, electrical, and physical performance of a technology library in an accurate and unambiguous fashion was widely recognized.

ALF was announced at the occasion of the OVI/VI-sponsored HDL conference in March 1997, where a trial version of the standard was released. Among the pioneers of proving the feasibility of ALF was the European CAD Standardization Initiative, sister organization of VSIA, who demonstrated an ALF-based ASIC implementation flow in 1997. In November 1997, OVI approved and released ALF version 1.0.

In 1998, the ASIC Council, under the auspices of the Silicon Integration Initiative (SI2), selected ALF as a complementary description of library elements within the open library architecture (OLA), which builds upon the IEEE 1481™-1999 standard for a delay calculation system. This endorsement triggered the initial adoption of ALF libraries by major ASIC vendors and the development of ALF version 1.1, which was approved and released by OVI in April 1999.

In June 1999, the ASIC council encouraged the ALF workgroup to include layout modeling. Consequently, deep submicron (DSM) issues, such as on-chip interconnect modeling, signal integrity, and reliability, became a major focus for ALF. The work culminated in the release of ALF version 2.0 in December 2000, under the auspices of the OVI/VI successor organization Accellera.

ALF version 2.0 became the foundation for this IEEE standard. An IEEE study group was formed in February 2001. The study group became the IEEE P1603 Working Group in June 2001. The name ALF has been retained due to already existing name recognition. By that time, the ALF had already set a standard for the industry, which can be measured by direct adoption and the influence on existing vendor-proprietary library formats. Major EDA vendors also made the specification of their existing proprietary library formats available to the industry and allowed the user community to extend those formats and strive for compatibility with ALF.

Although IEEE is now the legal owner of ALF, Accellera continues to foster and promote ALF. As a result, ALF has gained the attention of other national and international standardization bodies, such as JEITA in May 2002 and the IEC in October 2003.

From its inception, the goal for ALF has been to provide a solid foundation for library modeling within a continuously evolving application space. ALF has been designed to be more general in scope and purpose than a particular tool-oriented format. At the same time, care has been taken to make ALF easily adoptable

and to make the migration path from legacy formats as smooth as possible. Therefore, an ALF library can be very similar in appearance to a library in a conventional format, but ALF also has the expression power of a modeling language.

The construction principles for ALF can be summarized as follows:

- *Simplicity.* ALF has relatively few basic syntax construction principles. Once they are understood, reading and writing an ALF library or translating other library formats into ALF is very easy. Also, attention has been paid to the fact all ALF keywords are taken from natural language, i.e., spoken and written English, and their semantic meaning is as close to the natural language as possible. The use of artificial words or acronyms is limited to constructs, which have already become part of technical language in the industry.
- *Completeness.* Conventional library formats would support data without self-evident meaning, such as coefficients, scaling factors, etc. The interpretation of the data would be left to the application tool. On the other hand, an ALF library specifies a complete and self-contained description by providing the complete model, i.e., a calculation rule using an *arithmetic expression*. Furthermore, ALF contains information for characterization of particular measurement data, for example, delay, power, or noise. ALF introduces the original concept of a *vector expression* to describe the event pattern associated with the measurement. This concept has a far-reaching potential for creating abstract, yet accurate, modeling views for cells and larger blocks. Any timing, power, or signal integrity measurement on a digital circuit or a mixed-signal circuit can be associated with a vector expression.
- *Orthogonality.* Orthogonality allows for modeling features to be combined most efficiently with each other to yield a maximum expression capability. In ALF, orthogonality is closely related to context-sensitivity. A particular semantic meaning is created by describing a particular model in a particular context. For example, a model for capacitance can be described in the context of a wire, pin, or rule. A model for delay can be described in the context of a cell or wire. In a nonorthogonal approach, different keywords might be used for cell delay, wire delay, etc., and the fundamental semantics of delay would not be inherited by each construct.
- *Reusability and self-extensibility.* ALF supports the language constructs *template* and *group*, which allow for efficient representation of replicated statements with parameterized values. As these constructs follow the principle of orthogonality, a template can be used for parameterizing any ALF statement and not just a particular type of statement, such as a lookup table. ALF also supports language constructs for the definition of new keywords, their usage for construction of statements, and the context where they can be used.

In summary, ALF is a well-structured language that supports a true superset of virtually all existing library formats. Its conciseness and unique description features make it well-suited for innovative EDA applications.

Participants

At the time this standard was approved, the Advanced Library Format Working Group had the following membership:

Wolfgang Roethig, *Chair*
Joe Daniels, *Technical Editor*

Jay Abraham	Vassilios Gerousis	Cynthia Parrish
Tim Baldwin	Kevin Grotjohn	Steffen Rochel
Dennis Brophy	Jake Karrfalt	Sergei Sokolov
Peter Chirstensen	Srinivas Madaboosi	Vinay Srinivas
Gregory DuFour	Nancy Nettleton	Balchandra Thatte
Timothy Ehrler	Kim Nguyen	Tak Young
Uma Ekabaram		Alex Zamfirescu

The following members of the balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Shankha Basu	Sherman Fu	Robert Myers
Dennis Brophy	William Hanna	Assim Sagahyroon
Lyubomir Cekov	Masamichi Kawarabayashi	Toshiyuki Saito
Susheel Chandra	Jake Karrfalt	Marco Scorrano
Kai Moon Chow	Maddumage Karunaratne	Vinay Srinivas
Keith Chow	Gregory Luri	Mark Tillinghast
Peter Christensen	Srinivas Madaboosi	Yatin Trivedi
Maulik Dave	Eric Means	Srinivasa Vemuru
Dr. Sourav Dutta	Paul Menchini	John Williams
George Economakos	George Miao	Lance Wright
Timothy Ehrler	Yinghua Min	Tak Young
Amir El-Sheikh	Rajesh Moorkath	Li Zhang
Gerald Frenkil	Narayanan Murugesan	

When the IEEE-SA Standards Board approved this standard on 11 September 2003, it had the following membership:

Don Wright, *Chair*
Howard M. Frazier, *Vice Chair*
Judith Gorman, *Secretary*

H. Stephen Berger	Donald N. Heirman	Daleep C. Mohla
Joseph A. Bruder	Laura Hitchcock	William J. Moylan
Bob Davis	Richard H. Hulett	Paul Nikolic
Richard DeBlasio	Anant Kumar Jain	Gary S. Robinson
Julian Forster*	Lowell G. Johnson	Malcolm V. Thaden
Toshio Fukuda	Joseph L. Koepfinger*	Geoffrey O. Thompson
Arnold M. Greenspan	Tom McGean	Doug Topping
Raymond Hapeman	Steve M. Mills	Howard L. Wolfman

*Member Emeritus

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

Alan Cookson, *NIST Representative*
Satish K. Aggarwal, *NRC Representative*

Noelle D. Humenick
IEEE Standards Project Editor

Contents

1.	Overview.....	1
1.1	Scope and purpose	1
1.2	Application of this standard.....	2
1.2.1	Creation and characterization of library elements	2
1.2.2	Basic implementation and performance analysis of an IC	4
1.2.3	Hierarchical implementation and virtual prototyping of an IC.....	5
1.3	Conventions used in this standard	8
1.4	Contents of this standard.....	8
2.	References.....	9
3.	Definitions	9
4.	Acronyms	10
5.	ALF language construction principles	11
5.1	ALF metalanguage.....	12
5.2	Categories of ALF statements.....	13
5.3	Generic objects and library-specific objects.....	15
5.4	Singular statements and plural statements	17
5.5	Instantiation statement and assignment statement	18
5.6	Annotation, arithmetic model, and related statements.....	20
5.7	Statements for parser control	21
5.8	Name space and visibility of statements.....	22
6.	Lexical rules.....	22
6.1	Character set	22
6.2	Comment.....	24
6.3	Delimiter	25
6.4	Operator	25
6.4.1	Arithmetic operator.....	26
6.4.2	Boolean operator.....	26
6.4.3	Relational operator.....	27
6.4.4	Shift operator	28
6.4.5	Event operator.....	28
6.4.6	Meta operator.....	28
6.5	Number	29
6.6	Index value and index	29
6.7	Multiplier prefix symbol and multiplier prefix value	30
6.8	Bit literal	31
6.9	Based literal	32
6.10	Boolean value	32
6.11	Arithmetic value	32
6.12	Edge literal and edge value.....	33
6.13	Identifier.....	33
6.13.1	Non-escaped identifier.....	34
6.13.2	Placeholder identifier	34
6.13.3	Indexed identifier.....	34

6.13.4 Full hierarchical identifier	34
6.13.5 Partial hierarchical identifier	35
6.13.6 Escaped identifier	35
6.13.7 Keyword identifier	36
6.14 Quoted string	36
6.15 String value	37
6.16 Generic value	37
6.17 Vector expression macro	38
6.18 Rules for whitespace usage	38
6.19 Rules against parser ambiguity	38
7. Generic objects and related statements	39
7.1 Generic object	39
7.2 All purpose item	39
7.3 Annotation	40
7.4 Annotation container	40
7.5 ATTRIBUTE statement	40
7.6 PROPERTY statement	41
7.7 ALIAS declaration	41
7.8 CONSTANT declaration	42
7.9 KEYWORD declaration	42
7.10 SEMANTICS declaration	43
7.11 Annotations and rules related to a KEYWORD or a SEMANTICS declaration	44
7.11.1 VALUETYPE annotation	44
7.11.2 VALUES annotation	46
7.11.3 DEFAULT annotation	46
7.11.4 CONTEXT annotation	47
7.11.5 REFERENCETYPE annotation	48
7.11.6 SI_MODEL annotation	49
7.11.7 Rules for legal usage of KEYWORD and SEMANTICS declaration	50
7.12 CLASS declaration	51
7.13 Annotations related to a CLASS declaration	51
7.13.1 General CLASS reference annotation	51
7.13.2 USAGE annotation	52
7.14 GROUP declaration	53
7.15 TEMPLATE declaration	54
7.16 TEMPLATE instantiation	55
7.17 INCLUDE statement	58
7.18 ASSOCIATE statement and FORMAT annotation	58
7.19 REVISION statement	59
8. Library-specific objects and related statements	60
8.1 Library-specific object	60
8.2 LIBRARY and SUBLIBRARY declaration	61
8.3 Annotations related to a LIBRARY or a SUBLIBRARY declaration	61
8.3.1 LIBRARY reference annotation	61
8.3.2 INFORMATION annotation container	62
8.4 CELL declaration	63
8.5 Annotations related to a CELL declaration	64
8.5.1 CELL reference annotation	64
8.5.2 CELLTYPE annotation	64
8.5.3 RESTRICT_CLASS annotation	65

8.5.4	SWAP_CLASS annotation.....	67
8.5.5	SCAN_TYPE annotation.....	67
8.5.6	SCAN_USAGE annotation.....	68
8.5.7	BUFFERTYPE annotation.....	69
8.5.8	DRIVERTYPE annotation.....	69
8.5.9	PARALLEL_DRIVE annotation.....	70
8.5.10	PLACEMENT_TYPE annotation.....	70
8.5.11	SITE reference annotation for a CELL.....	71
8.5.12	ATTRIBUTE values for a CELL.....	71
8.6	PIN declaration.....	73
8.7	PINGROUP declaration.....	74
8.8	Annotations related to a PIN or a PINGROUP declaration.....	75
8.8.1	PIN reference annotation.....	75
8.8.2	MEMBERS annotation.....	75
8.8.3	VIEW annotation.....	75
8.8.4	PINTYPE annotation.....	76
8.8.5	DIRECTION annotation.....	77
8.8.6	SIGNALTYPE annotation.....	78
8.8.7	ACTION annotation.....	80
8.8.8	POLARITY annotation.....	81
8.8.9	CONTROL_POLARITY annotation container.....	82
8.8.10	DATATYPE annotation.....	83
8.8.11	INITIAL_VALUE annotation.....	84
8.8.12	SCAN_POSITION annotation.....	84
8.8.13	STUCK annotation.....	84
8.8.14	SUPPLYTYPE annotation.....	85
8.8.15	SIGNAL_CLASS annotation.....	86
8.8.16	SUPPLY_CLASS annotation.....	86
8.8.17	DRIVETYPE annotation.....	88
8.8.18	SCOPE annotation.....	89
8.8.19	CONNECT_CLASS annotation.....	89
8.8.20	SIDE annotation.....	90
8.8.21	ROW and COLUMN annotation.....	90
8.8.22	ROUTING_TYPE annotation.....	91
8.8.23	PULL annotation.....	92
8.8.24	ATTRIBUTE values for a PIN or a PINGROUP.....	93
8.9	PRIMITIVE declaration.....	95
8.10	WIRE declaration.....	95
8.11	Annotations related to a WIRE declaration.....	95
8.11.1	WIRE reference annotation.....	95
8.11.2	WIRETYPE annotation.....	96
8.11.3	SELECT_CLASS annotation.....	96
8.12	NODE declaration.....	97
8.13	Annotations related to a NODE declaration.....	98
8.13.1	NODE reference annotation.....	98
8.13.2	NODETYPE annotation.....	98
8.13.3	NODE_CLASS annotation.....	99
8.14	VECTOR declaration.....	100
8.15	Annotations related to a VECTOR declaration.....	100
8.15.1	VECTOR reference annotation.....	100
8.15.2	PURPOSE annotation.....	101
8.15.3	OPERATION annotation.....	102
8.15.4	LABEL annotation.....	102
8.15.5	EXISTENCE_CONDITION annotation.....	103

8.15.6	EXISTENCE_CLASS annotation	103
8.15.7	CHARACTERIZATION_CONDITION annotation	104
8.15.8	CHARACTERIZATION_VECTOR annotation	104
8.15.9	CHARACTERIZATION_CLASS annotation	104
8.15.10	MONITOR annotation	105
8.16	LAYER declaration	105
8.17	Annotations related to a LAYER declaration	106
8.17.1	LAYER reference annotation	106
8.17.2	LAYERTYPE annotation	106
8.17.3	PITCH annotation	107
8.17.4	PREFERENCE annotation	107
8.18	VIA declaration	107
8.19	Annotations related to a VIA declaration	108
8.19.1	VIA reference annotation	108
8.19.2	VIATYPE annotation	108
8.20	RULE declaration	109
8.21	ANTENNA declaration	109
8.22	BLOCKAGE declaration	110
8.23	PORT declaration	110
8.24	Annotations related to a PORT declaration	111
8.24.1	Reference to a PORT using PIN reference annotation	111
8.24.2	PORTTYPE annotation	111
8.25	SITE declaration	112
8.26	Annotations related to a SITE declaration	112
8.26.1	SITE reference annotation	112
8.26.2	ORIENTATION_CLASS annotation	112
8.26.3	SYMMETRY_CLASS annotation	113
8.27	ARRAY declaration	113
8.28	Annotations related to an ARRAY declaration	114
8.28.1	ARRAYTYPE annotation	114
8.28.2	LAYER reference annotation for ARRAY	114
8.28.3	SITE reference annotation for ARRAY	115
8.29	PATTERN declaration	115
8.30	Annotations related to a PATTERN declaration	115
8.30.1	PATTERN reference annotation	115
8.30.2	SHAPE annotation	115
8.30.3	VERTEX annotation	117
8.30.4	ROUTE annotation	118
8.30.5	LAYER reference annotation for PATTERN	119
8.31	REGION declaration	119
8.32	Annotations related to a REGION declaration	119
8.32.1	REGION reference annotation	119
8.32.2	BOOLEAN annotation	120
9.	Description of functional and physical implementation	120
9.1	FUNCTION statement	120
9.2	TEST statement	120
9.3	Definition and usage of a pin variable	121
9.3.1	Pin variable and pin value	121
9.3.2	Pin assignment	121
9.3.3	Usage of a pin variable in the context of a FUNCTION or a TEST statement	122
9.4	BEHAVIOR statement	123
9.5	STRUCTURE statement and CELL instantiation	124

9.6	STATETABLE statement.....	125
9.7	NON_SCAN_CELL statement.....	126
9.8	RANGE statement	127
9.9	Boolean expression	128
9.10	Boolean value system	129
9.10.1	Scalar boolean value	129
9.10.2	Vectorized boolean value.....	129
9.10.3	Non-assignable boolean value	131
9.11	Boolean operations and operators.....	131
9.11.1	Logical operation	131
9.11.2	Bitwise operation	133
9.11.3	Conditional operation	135
9.11.4	Integer arithmetic operation.....	135
9.11.5	Shift operation.....	136
9.11.6	Comparison operation.....	136
9.12	Vector expression and control expression	138
9.13	Specification of a pattern of events.....	139
9.13.1	Specification of a single event.....	139
9.13.2	Specification of a compound event.....	140
9.13.3	Specification of a compound event with alternatives	141
9.13.4	Evaluation of a specified pattern of events against a realized pattern of events	143
9.13.5	Specification of a conditional pattern of events.....	145
9.14	Predefined PRIMITIVE.....	145
9.14.1	Predefined PRIMITIVE ALF_BUF	146
9.14.2	Predefined PRIMITIVE ALF_NOT	146
9.14.3	Predefined PRIMITIVE ALF_AND	146
9.14.4	Predefined PRIMITIVE ALF_NAND.....	146
9.14.5	Predefined PRIMITIVE ALF_OR.....	147
9.14.6	Predefined PRIMITIVE ALF_NOR.....	147
9.14.7	Predefined PRIMITIVE ALF_XOR.....	147
9.14.8	Predefined PRIMITIVE ALF_XNOR.....	147
9.14.9	Predefined PRIMITIVE ALF_BUFIF1.....	147
9.14.10	Predefined PRIMITIVE ALF_BUFIF0.....	148
9.14.11	Predefined PRIMITIVE ALF_NOTIF1	148
9.14.12	Predefined PRIMITIVE ALF_NOTFIF0	148
9.14.13	Predefined PRIMITIVE ALF_MUX.....	148
9.14.14	Predefined PRIMITIVE ALF_LATCH.....	149
9.14.15	Predefined PRIMITIVE ALF_FLIPFLOP	149
9.15	WIRE instantiation.....	149
9.16	Geometric model.....	151
9.17	Predefined geometric models using TEMPLATE	153
9.17.1	Predefined TEMPLATE RECTANGLE	153
9.17.2	Predefined TEMPLATE LINE	154
9.18	Geometric transformation	154
9.19	ARTWORK statement.....	156
9.20	VIA instantiation.....	156
10.	Description of electrical and physical measurements	157
10.1	Arithmetic expression	157
10.2	Arithmetic operations and operators.....	158
10.2.1	Sign inversion	158
10.2.2	Floating point arithmetic operation.....	158
10.2.3	Macro arithmetic operator	159

10.3 Arithmetic model	159
10.4 HEADER, TABLE, and EQUATION statements	161
10.5 MIN, MAX, and TYP statements	163
10.6 Auxiliary arithmetic model.....	165
10.7 Arithmetic submodel.....	165
10.8 Arithmetic model container	166
10.8.1 General arithmetic model container.....	166
10.8.2 Arithmetic model container LIMIT	166
10.8.3 Arithmetic model container EARLY and LATE	167
10.9 Generally applicable annotations for arithmetic models	167
10.9.1 UNIT annotation	167
10.9.2 CALCULATION annotation	168
10.9.3 INTERPOLATION annotation.....	169
10.9.4 DEFAULT annotation	170
10.9.5 MODEL reference annotation	171
10.10 VIOLATION statement, MESSAGE TYPE, and MESSAGE annotation.....	172
10.11 Arithmetic models for timing, power, and signal integrity	174
10.11.1 TIME174	
10.11.2 FREQUENCY	175
10.11.3 DELAY	176
10.11.4 RETAIN.....	177
10.11.5 SLEWRATE	178
10.11.6 SETUP and HOLD	179
10.11.7 RECOVERY and REMOVAL	180
10.11.8 NOCHANGE and ILLEGAL	181
10.11.9 PULSEWIDTH.....	182
10.11.10 PERIOD.....	183
10.11.11 JITTER	184
10.11.12 SKEW	185
10.11.13 THRESHOLD	186
10.11.14 NOISE and NOISE_MARGIN.....	187
10.11.15 POWER and ENERGY	190
10.12 FROM and TO statements.....	191
10.13 Annotations related to timing, power, and signal integrity	192
10.13.1 EDGE_NUMBER annotation.....	192
10.13.2 PIN reference and EDGE_NUMBER annotation for FROM and TO	192
10.13.3 PIN reference and EDGE_NUMBER annotation for SLEWRATE	193
10.13.4 PIN reference and EDGE_NUMBER annotation for PULSEWIDTH.....	193
10.13.5 PIN reference and EDGE_NUMBER annotation for SKEW	194
10.13.6 PIN reference annotation for NOISE and NOISE_MARGIN	194
10.13.7 MEASUREMENT annotation	194
10.14 Arithmetic models for environmental conditions	196
10.14.1 PROCESS.....	196
10.14.2 DERATE_CASE	196
10.14.3 TEMPERATURE	197
10.15 Arithmetic models for electrical circuits	198
10.15.1 VOLTAGE.....	198
10.15.2 CURRENT.....	199
10.15.3 CAPACITANCE	200
10.15.4 RESISTANCE	202
10.15.5 INDUCTANCE	203
10.16 Annotations for electrical circuits	204
10.16.1 NODE reference annotation for electrical circuits	204
10.16.2 COMPONENT reference annotation.....	205

10.16.3	PIN reference annotation for electrical circuits	206
10.16.4	FLOW annotation	207
10.17	Miscellaneous arithmetic models.....	208
10.17.1	DRIVE STRENGTH.....	208
10.17.2	SWITCHING_BITS with PIN reference annotation.....	209
10.18	Arithmetic models related to structural implementation	209
10.18.1	CONNECTIVITY	209
10.18.2	DRIVER and RECEIVER	210
10.18.3	FANOUT, FANIN, and CONNECTIONS	211
10.19	Arithmetic models related to layout implementation.....	212
10.19.1	SIZE	212
10.19.2	AREA.....	213
10.19.3	PERIMETER	214
10.19.4	EXTENSION	215
10.19.5	THICKNESS	216
10.19.6	HEIGHT.....	217
10.19.7	WIDTH	217
10.19.8	LENGTH	218
10.19.9	DISTANCE.....	219
10.19.10	OVERHANG	220
10.19.11	DENSITY	220
10.20	Annotations related to arithmetic models for layout implementation	221
10.20.1	CONNECT_RULE annotation	221
10.20.2	BETWEEN annotation	222
10.20.3	BETWEEN annotation for CONNECTIVITY	222
10.20.4	BETWEEN annotation for DISTANCE, LENGTH, OVERHANG	223
10.20.5	MEASURE annotation	224
10.20.6	REFERENCE annotation container.....	225
10.20.7	ANTENNA reference annotation	226
10.20.8	TARGET annotation.....	226
10.20.9	PATTERN reference annotation	227
10.21	Arithmetic submodels for timing and electrical data.....	228
10.22	Arithmetic submodels for physical data	229
Annex A (informative) Syntax rule summary		230
Annex B (informative) Semantics rule summary		246
Annex C (informative) ALF library example		273
Annex D (informative) Bibliography		279

IEEE Standard for an Advanced Library Format (ALF) Describing Integrated Circuit (IC) Technology, Cells, and Blocks

1. Overview

This clause explains the scope and purpose of this standard, gives an overview of its applications, explains the conventions used, and summarizes its contents.

1.1 Scope and purpose

The scope of this standard is to serve as the data specification language of library elements for design applications used to implement an integrated circuit (IC). The range of abstraction shall include from the register-transfer level (RTL) to the physical level. The language shall model behavior, timing, power, signal integrity, physical abstraction and physical implementation rules of library elements.

Library elements for implementation of an IC include sets of predefined components, composed of transistors and interconnect, and sets of predefined rules for the assembly of such components. The design of application-specific ICs (ASICs) in particular relies on the availability of predefined components, called cells. An IC that uses large predefined compound library elements with a standardized functionality, for example, microprocessors as building blocks, is called a system on a chip (SOC).

The design of an ASIC or an SOC involves electronic design automation (EDA) tools. These tools assist the designer in the choice and assembly of library elements for creating and implementing the IC and verifying the functionality and performance specification of the IC. In order to create an IC involving several million instances of library elements within a manageable time period counted in weeks or months, the usage of EDA tools is mandatory.

A suitable description of library elements for design applications involving EDA tools is required. A key feature is to represent a library element at a level of abstraction that does not reveal the implementation of the library element itself. This is important for the following reasons:

- The complexity of the design data itself mandates data reduction.

- The complexity of the verification process, i.e., the verification for functional, physical, and electrical correctness, mandates that a library element is already characterized and verified by itself. Only the data necessary for creation and verification of the assembled IC is represented in the library.
- A library element is considered an intellectual property (IP) of the library provider.

Therefore, the purpose of this standard is to provide a modeling language and semantics for the functional, physical, and electrical performance description of technology-specific libraries for cell-based and block-based design. Without a standard, EDA tools would use multiple proprietary and tool-specific library descriptions. The semantics would be defined by tool implementations only, which are subject to change and prone to misinterpretation. Also, there would be redundancy using multiple descriptions for similar library aspects. Therefore, this standard proposes to create a consistent library view suitable as a reference for IC designers as well as for EDA tool developers and integrators.

1.2 Application of this standard

The ALF standard can be used in many different places throughout the design flow. The major uses include creation and characterization of library elements, basic implementation and performance analysis of an IC, and hierarchical implementation and virtual prototyping of an IC.

An application, as described in 1.2.1, 1.2.2, and 1.2.3, shall be called *compliant to ALF*, if and only if it satisfies the following criteria:

- a) An application tool that uses ALF as input is capable of parsing any ALF file according to the rules specified in Clause 5, Clause 6, Clause 7, Clause 8, Clause 9, and Clause 10, even if not all data in that file is used by the application. In this way, one ALF library can be used for multiple applications with different scope.
- b) A tool, as referred to in item a), uses a well-defined set of data from the ALF file within the scope of its application and interprets this data according to the rules specified in Clause 5, Clause 6, Clause 7, Clause 8, Clause 9, and Clause 10. In this way, any two applications using the same set of ALF data interpret the ALF data in the same consistent way.
- c) An application tool that uses ALF as output is capable of generating an ALF file according to the rules specified in Clause 5, Clause 6, Clause 7, Clause 8, Clause 9, and Clause 10, and the generated file contains a well-defined set of data for an application as referred to in item a).

The following conventions are used in the flow diagrams depicted in Figure 1, Figure 2, Figure 3, and Figure 4:

- *Rectangle*: data file, format optionally indicated in parentheses
- *Oval*: application
- *Solid arrow*: existing, established function in the design flow
- *Dotted arrow*: possible design flow

1.2.1 Creation and characterization of library elements

ALF can be used to specify the desired functionality and characterization space of a library element, i.e., a cell.

The application for creation of a cell is shown in Figure 1.

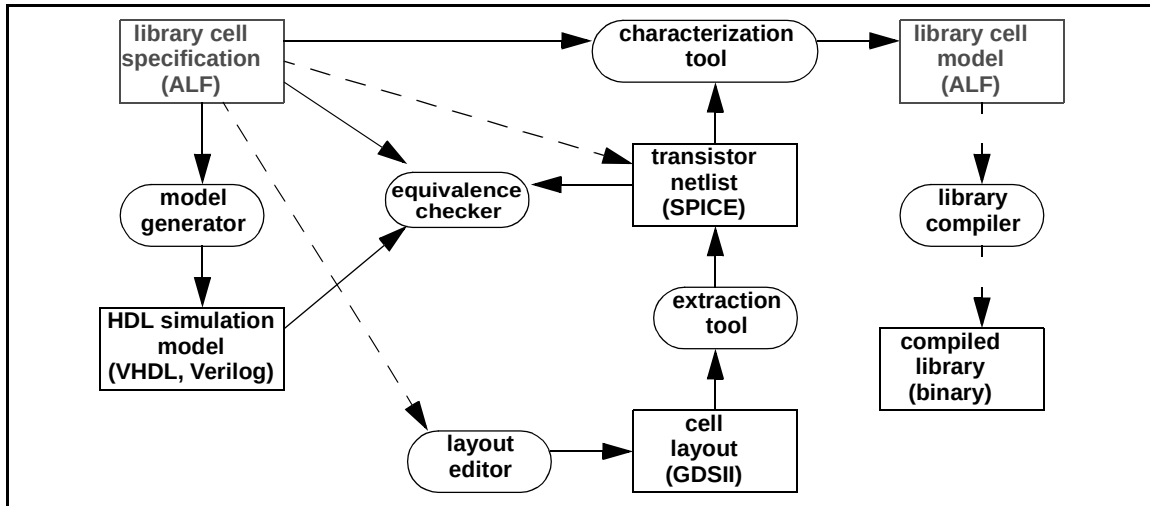


Figure 1—Cell library creation flow

A specification of a library element, i.e., a cell (see 8.2, 8.4), can be described in ALF. This specification includes the name of the cell and its terminals, i.e., pins (see 8.6), and a formal description of the function (see 9.1) performed by the cell. This formal description is sufficient for the purpose of generating hardware description language (HDL) simulation models in various languages, for example, VHDL (see IEEE Std 1076™-2002)¹ or Verilog (see IEEE Std 1364™-2001).

Multiple HDL models can be generated for different purposes, where the difference is defined by the user's preference for modeling style rather than by the functionality of the cell. For example, one model can handle unknown logic states in a crude way, resulting in fast simulation, while another model can handle unknown logic states in a case-by-case way, resulting in slow but more accurate simulation. The ALF model can serve as a common reference for all those HDL models.

A physical layout of a cell can be represented in the GDSII format. A transistor-level netlist of a cell in SPICE format (*SPICE 2G6 User's Guide* [B7]²) can be extracted from the physical layout. Such a transistor netlist includes parasitic electrical components. Alternatively, a designer can create a transistor netlist by hand or by using an EDA tool that maps a functional specification described in ALF into a transistor-level netlist. Such a transistor netlist is less accurate than one extracted from layout, but can still be useful for prototyping a library.

Both the transistor netlist and the various HDL models can be compared against the functional specification described in ALF. More importantly, the transistor netlist can be used to characterize the performance of the cell, i.e., measure timing, power, noise (see 10.11), and other electrical characteristics (see 10.15) by running a SPICE simulation. The set of necessary SPICE simulations is determined and controlled by a characterization tool. The characterization tool can infer pertinent information from the specification represented in ALF, as far as this information relates to the functionality of the cell itself. For example, the timing arcs that need to be characterized can be represented in or inferred from ALF. The output of the characterization tool is a library cell model, populated with characterization data, also represented in ALF.

Optionally, a library compiler can be used to combine all the library cell models into a binary file, as a data preparation step for an EDA application tool.

¹For information on references, see Clause 2.

²The numbers in brackets correspond to those in the bibliography in Annex D.

The ALF library can be used in an IC implementation flow which uses cells as building blocks, in particular, an ASIC implementation flow.

```

graph TD
    RTL[RTL design description  
(VHDL, Verilog)] --> Syn[RTL synthesis]
    Syn --> Netlist1[gate-level netlist]
    Netlist1 --> EC[equivalence checking]
    Netlist1 --> Perf1[performance analysis]
    EC --> Perf1
    LibCell[library cell models  
(ALF)] --> Syn
    LibCell --> EC
    LibCell --> Perf1
    Netlist1 --> Place[cell placement]
    Place --> Netlist2[gate-level netlist  
with placement]
    Netlist2 --> ParaEst[parasitic estimation]
    Netlist2 --> Perf2[performance analysis]
    ParaEst --> Perf2
    Netlist2 --> Route[interconnect routing]
    Route --> Netlist3[gate-level netlist  
with placement and routing]
    Netlist3 --> ParaFile[parasitic file  
(SPEF)]
    Netlist3 --> ParaExt[parasitic extraction]
    ParaFile --> ParaExt
    ParaExt --> Perf3[performance analysis]
    Perf3 --> Perf4[performance analysis]
    IntMod[interconnect models  
(ALF)] --> Perf4
    IntMod --> ParaExt
  
```

In this flow, an RTL design description is transformed into a netlist by an RTL synthesis tool. The netlist contains instances of cells, also called gates, rather than transistors. This application can use the ALF library to find the library elements needed to map the RTL description into a netlist containing instances of cells. The transistors inside the cells are not described in the ALF cell models.

An equivalence checking tool can be used to decide whether the RTL-to-netlist transformation has been done correctly, by comparing the RTL design description with the netlist. This application can use the same ALF library as the RTL synthesis tool. Also, an HDL simulation tool (not shown in Figure 2) can be used to decide whether both the RTL design description and the netlist behave as expected in response to a given stimulus. The simulation tool can use an ALF model or an HDL model derived from the ALF model (see 9.4, 9.6).

The flow in Figure 2 is simplified. Special netlist transformations, such as the creation of data path structures, creation structures related to design for test (DFT), and especially scan insertion, are not shown here. However, the ALF cell models also contain information pertaining to these applications (see 8.5.3, 8.5.5, 8.5.6, 8.8.12, 9.2, 9.7).

The process of cell placement and interconnect routing is summarily referred to as layout. Special layout operations, such as the layout of a power supply structure or of a clock network structure, are not explicitly shown in Figure 2. The ALF cell models contain abstract physical information, such as the size and shape of the cell, and the location, size, and shape of the cell pins and routing blockages, which are pertinent for layout (see 8.22, 8.23). Also, abstract information concerning the artwork within the cell can be represented in ALF, for example, the area, perimeter, and connectivity of artwork on specific layers (see 10.18, 10.19).

This information is pertinent for manufacturability, such as antenna rule (see 8.21, 10.19.1, 10.19.2, 10.19.3) and metal density checks (see 8.31, 10.19.11).

In addition to cell models, technology rules for routing can also be represented in ALF, such as constraints for the width and length of routing segments, the distance between routing segments, the distance between vias, etc. (see 8.16, 8.18, 8.20, 10.19.7, 10.19.8, 10.19.9).

The implemented IC needs not only be correct in terms of functionality and layout, it also has to meet electrical performance constraints, predominantly timing constraints. Other aspects of electrical performance, such as power consumption, signal integrity, and reliability, have become increasingly important. Signal integrity aspects include the cleanliness of signal waveform shapes and the immunity against noise induced by crosstalk and voltage drop (see 10.11.1, 10.11.14, 10.15.1, 10.15.2). Reliability aspects include dependable long-term operation in the presence of electromigration stress, hot electron effect, and thermal instability. The cell models in ALF support characterization data for timing, power, signal integrity, and reliability. For example, reliability data can be described as a limit for voltage, current, or operation frequency (see 10.11.2). A particular feature in ALF is the representation of these data in the context of a stimulus, described by a *vector expression* (see 8.14, 9.12, 9.13). With this feature, the data can be related to particular environmental operation conditions, and a more accurate performance analysis can be performed.

Performance analysis happens within each step of the IC implementation process. RTL synthesis, cell placement, and interconnect routing applications have embedded static timing analysis (STA) and other performance analysis capabilities. Also, after completion of each step, a stand-alone performance analysis can be applied to measure the achieved performance more accurately.

Electrical performance depends not only on the interaction between instances of cells, but also on the parasitics introduced by the interconnect wires. After netlist creation, parasitics can be statistically estimated using a wire load model (WLM). After placement, parasitics can be more accurately predicted by estimating the length of particular routing wires between pins of placed cells. After routing, actual parasitics can be extracted and represented in a file using the standard parasitic exchange format (SPEF) (IEEE Std 1481™-1999 [B4]). An interconnect model in ALF can describe a statistical WLM, a rule for parasitic estimation based on estimated routes, or an interconnect analysis model (see 8.10, 8.12). The interconnect analysis model specifies the desired level of granularity for the parasitics (see 10.15.3, 10.15.4, 10.15.5, 10.16) and the calculation of timing, noise, voltage, or current based on instances of parasitics and on an electrical model of a driver cell. The data for the electrical model of a particular driver cell can be represented in ALF as a part of the cell characterization data.

1.2.3 Hierarchical implementation and virtual prototyping of an IC

An IC implementation flow with cells as building blocks has its limits imposed by the number of objects, i.e., the instances of cells and nets that can be reasonably handled by designers and by application flows.

For ICs exceeding the limits of objects that can be reasonably handled, the following approaches are used, possibly in combination with each other:

- *Bottom-up design*: Create larger building blocks from cells first, then use these blocks for IC implementation.
- *Top-down design*: Divide a design into subdesigns first, implement each subdesign as a block, then assemble the blocks.
- *Virtual prototyping*: Do a simplified so-called virtual implementation of the entire design first, then partition the virtually implemented design into blocks, use the results of the virtual implementation as constraints for actual implementation of each block, and implement and assemble the blocks.

The common denominator for all these methods is creation of blocks, in order to reduce the number of objects seen by the application.

The application for creation of a block is shown in Figure 3.

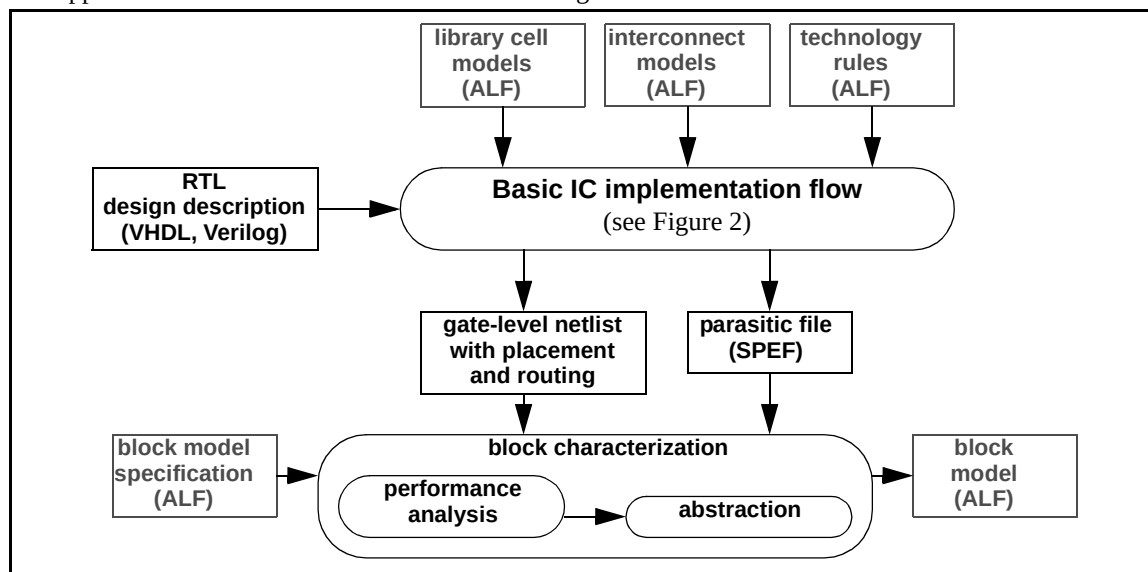


Figure 3—Block creation flow

A block can be created by using the basic IC implementation flow (see Figure 2). A block with a functionality that can be used and reused is commonly referred to as IP of the designer. In case of a “hard” block, the primary output of the implementation flow, i.e., a gate-level netlist with placement and routing, is preserved and eventually transformed into a physical artwork. In case of a “soft” block, only the primary input of the implementation flow, i.e., the RTL design description, is preserved. The output of the implementation flow serves only for the purpose of block characterization, i.e., creation of an abstract model for the block. The block characterization consists of a repeated application of performance analysis within the range of desired characterization followed by abstraction. Abstraction includes reduction of the physical implementation data and association of the performance analysis data with a specified model. Both the specification of the model and the model itself can be represented in ALF.

Variants to this flow include partial IC implementation, for example, only RTL synthesis and placement without routing, especially in the case of a soft block, where the implementation data is not preserved. The rationale for not preserving the implementation data of a block is the possibility of achieving a better overall IC implementation result by implementing the block later in the context of other blocks, instead of implementing the block stand-alone up front.

Depending on whether a block is used as a hard block or a soft block, the ALF model can represent a different level of abstraction. An ALF model for a hard block can have features similar to an ALF model for a cell (see 1.2.1 and 1.2.2). In addition, the netlist and the parasitics representing the output of the implementation flow can be partially preserved in the ALF model, especially at the boundary of the block (see 9.5). This enables accurate analysis of the electrical interaction of a block with adjacent blocks in the context of an IC implementation. On the other hand, an ALF model for a soft block can represent a statistical range or upper and lower bounds (see 10.5) for characterization data rather than “hard” characterization data, since there is a degree of variability in the implementation of actual instances of the block. Also, a statistical WLM can be encapsulated within the model of the block.

ALF supports specific modeling features for parameterizeable blocks, i.e., blocks that can be implemented in various physical shapes or sizes and with variable bitwidth and performance characteristics. The ALF

constructs *group* (see 7.14), *template* (see 7.15), *static*, and *dynamic template instantiation* (see 7.16) can be used for this purpose.

Independent of whether a block is a hard block or a soft block, the application for creating the IC can now use the abstract model of the block as a library element rather than using a cell. In a similar way, as an ALF model of a cell does not reveal transistor-level implementation details, an ALF model of a block does not reveal gate-level implementation details. However, the ALF model of a block still provides enough information for an application to implement or explore the implementation of an IC and analyze the performance and the compliance to logical and physical design constraints.

An IC is designed in the context of a specific environment with specific constraints. Environmental constraints include for the characteristics of the package, the printed board, the range of process, voltage, and temperature (PVT) conditions (see 10.14). Other constraints are given by globally applicable physical design rules, for example, the available routing layers, the amount of routing resources reserved for the power distribution, and the available locations for IO pins at the boundary and in the center of a chip. The virtual prototyping approach can be used to evaluate whether a design can be implemented within these constraints. The electrical characterization data in ALF, i.e., timing, power, noise, physical and electrical rules, estimation models for parasitics, etc., can be represented as mathematical functions of environmental conditions and constraints (see 10.3, 10.4).

A conceptual flow for the virtual prototyping and hierarchical implementation of an IC involving ALF models at different levels of abstraction is shown in Figure 4.

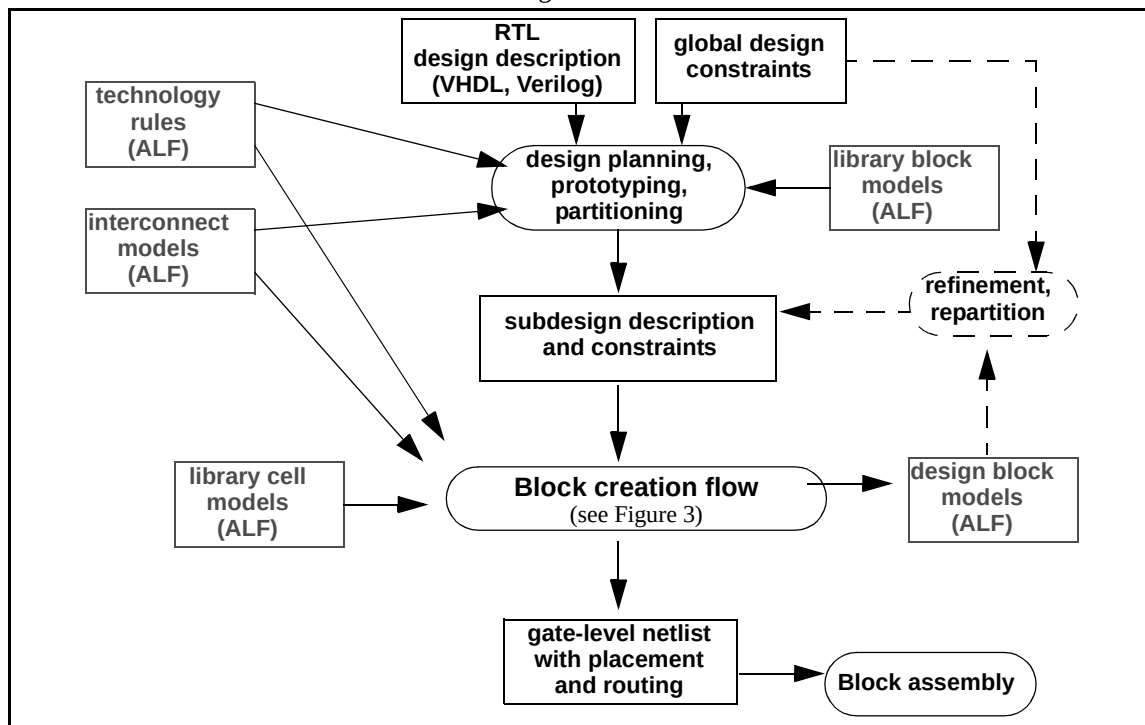


Figure 4—IC prototyping and hierarchical implementation flow

The design planning and prototyping application uses predefined models of blocks as library elements, referred to as “library block models.” The design is partitioned into subdesigns. The block creation flow (see Figure 3), i.e., a combination of block implementation and block characterization, is applied to each subdesign. The applicable library elements for each block are cells. The outputs of the block creation flow are the characterized models of the subdesigns, referred to as “design block models.” The design block models can be used to iterate on the design planning application, resulting in a possible refinement and repartitioning of the design. Once the evaluation of each block against the subdesign constraints and the

evaluation of the virtually assembled blocks against the global design constraints are satisfactory, the block implementation results, i.e., the netlist with placement and routing for each block, can actually be assembled to form the IC.

The design of an IC can use a combination of cells, hard blocks and soft blocks, blocks with fixed specification, and parameterizeable blocks as library elements. Some of the library elements are available independent of the design, others are created during and only for the purpose of that particular design. An abstract model for a soft block can be used in conjunction with a more detailed model for a hard block. The abstract model can be replaced with a more detailed model during implementation of the block. Technology rules and interconnect models are used throughout the flow.

In summary, the ALF standard provides a common modeling language for library elements, technology rules, and interconnect models. ALF models at different levels of abstraction can be used concurrently by EDA applications for planning, prototyping, implementation, analysis, optimization, and verification of complex ICs.

1.3 Conventions used in this standard

The syntax for description of lexical and syntax rules uses the following conventions:

<code>::=</code>	definition of a syntax rule
<code> </code>	alternative definition
<code>[item]</code>	an optional item
<code>[item1 item2 ...]</code>	optional item with alternatives
<code>{item}</code>	optional item that can be repeated
<code>{item1 item2 ... }</code>	optional items with alternatives which can be repeated
item	boldface specifies verbatim usage of a string of characters.
ITEM	uppercase boldface specifies verbatim usage of a keyword.
<i>prefix_item</i>	prefix in italic is for explanation purpose only
<i>PREFIX_item</i>	prefix in uppercase italic indicates that a keyword is used

NOTE—These conventions do not prescribe usage of uppercase or lowercase characters, as ALF is case-insensitive.

1.4 Contents of this standard

The organization of the remainder of this standard is

- Clause 2 (References) provides references to other applicable standards that are assumed or required for this standard.
- Clause 3 (Definitions) defines terms used throughout the different specifications contained in this standard.
- Clause 4 (Acronyms) defines the acronyms used in this standard.
- Clause 5 (ALF language construction principles) defines the language construction principles used in this standard.
- Clause 6 (Lexical rules) specifies the lexical rules.
- Clause 7 (Generic objects and related statements) defines syntax and semantics of generic objects used in this standard.
- Clause 8 (Library-specific objects and related statements) defines syntax and semantics of library-specific objects used in this standard.

- Clause 9 (Description of functional and physical implementation) defines syntax and semantics of statements related to functional and physical implementation of library elements used in this standard
- Clause 10 (Description of electrical and physical measurements) defines syntax and semantics of statements describing electrical and physical measurements related to library elements used in this standard.
- Annex A, Annex B, Annex C, and Annex D. Following Clause 10 are a series of informative annexes.

2. References

This standard shall be used in conjunction with the following publications. When the following standards are superseded by an approved revision, the revision shall apply.

IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual.^{3,4}

IEEE Std 1364-2001, IEEE Standard for Verilog Hardware Description Language.

IEEE Std 1497TM-2001, IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process.

IEEE/ASTM SI 10-2002, American National Standard for Use of the International System of Units (SI): The Modern Metric System.

ISO/IEC 8859-1:1998(E), Information technology—8-bit single-byte coded graphic character sets—Part 1: Latin Alphabet.⁵

ISO/IEC 9899:1999, Programming languages—C.

ISO/IEC 14882:1998, Programming language—C++.

3. Definitions

For the purposes of this standard, the following terms and definitions apply. *The Authoritative Dictionary of IEEE Standards and Terms*, Seventh Edition [B3], should be consulted for terms not defined in this standard.

3.1 advanced library format (ALF): The format of any file that can be parsed according to the syntax and semantics defined within this standard, IEEE Std 1603-2003.

3.2 ALF: *See:* **advanced library format.**

3.3 ALF name: The name of an ALF object.

3.4 ALF object: An element described in ALF.

³The IEEE standards or products referred to in Clause 2 are trademarks owned by the Institute of Electrical and Electronics Engineers, Incorporated.

⁴IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (<http://standards.ieee.org/>).

⁵ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembe, CH-1211, Genève 20, Switzerland/Suisse (<http://www.iso.ch/>). ISO/IEC publications are also available in the United States from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112, USA (<http://global.ihs.com/>). Electronic copies are available in the United States from the American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (<http://www.ansi.org/>).

3.5 ALF type: The type of an ALF object.

3.6 ALF value: A value associated with an ALF object.

3.7 application, electric design automation (EDA) application: Any software program that uses data represented in ALF. Examples include RTL (register transfer level) synthesis tools, static timing analyzers, etc. *See also:* **advanced library format; register transfer level.**

3.8 arc: *See:* **timing arc.**

3.9 argument: A data item required for the mathematical evaluation of an arithmetic model. *See also:* **arithmetic model.**

3.10 arithmetic model: A description of a mathematical model for an electrical or physical measurement in ALF.

3.11 cell, library cell: An electronic circuit that is a component of a library described in ALF.

3.12 geometric model: A description of a layout geometry in ALF.

3.13 register transfer level: A technology-independent description of a digital electronic design allowing inference of sequential and combinatorial logic components.

3.14 timing arc: An abstract representation of a measurement of an interval between two points in time during operation of a library cell.

4. Acronyms

This clause lists the acronyms used in this standard.

ALF	advanced library format
ASIC	application specific integrated circuit
BIST	built-in self test
BNF	Backus-Naur form
CAE	computer-aided engineering [the term electronic design automation (EDA) is preferred]
CAM	content-addressable memory
CPU	central processing unit
DFT	design for test
DSP	digital signal processor
EDA	electronic design automation
EDIF	electronic design interchange format
GPU	graphical processing unit

HDL	hardware description language
IC	integrated circuit
IP	intellectual property
LSB	least significant bit
LSSD	level-sensitive scan design
MPU	micro processor unit
MSB	most significant bit
PLL	phase-locked loop
PVT	process/voltage/temperature (denoting a set of environmental conditions)
RAM	random access memory
RC	resistance (times) capacitance
ROM	read-only memory
RTL	register transfer level
SDF	standard delay format (see IEEE Std 1497-2001)
SOC	system on a chip
SPEF	standard parasitic exchange format (see IEEE Std 1481-1999 [B4])
SPICE	simulation program with integrated circuit emphasis (see <i>SPICE 2G6 User's Guide</i> [B7])
STA	static timing analysis
VHDL	VHSIC hardware description language (see IEEE Std 1076-2002)
VHSIC	very high-speed integrated circuit
VLSI	very large-scale integration
WLM	wire load model

5. ALF language construction principles

This clause presents the ALF language construction principles and gives an overview of the language features. The ALF statements and the rules for relationships between ALF statements are presented summarily. Keywords are involved in the declaration of ALF statements. The keywords in ALF shall be case-insensitive. However, uppercase is used for keywords throughout this clause for clarity.

5.1 ALF metalanguage

Syntax 1 establishes an *ALF metalanguage*.

```

ALF_statement ::=
    ALF_type [ [ index ] ALF_name [ index ] ] [ = ALF_value ] ;
    | ALF_type [ [ index ] ALF_name [ index ] ] { = ALF_value } { { ALF_value | : | ; } }
    | ALF_type [ [ index ] ALF_name [ index ] ] [ = ALF_value ] { { ALF_statement } }
ALF_type ::=
    identifier
    | @
    | :
ALF_name ::=
    identifier
    | control_expression
ALF_value ::=
    number
    | multiplier_prefix_symbol
    | identifier
    | quoted_string
    | bit_literal
    | based_literal
    | edge_value
    | arithmetic_expression
    | boolean_expression
    | control_expression

```

Syntax 1—Syntax construction for ALF metalanguage

The *ALF type* is defined by an *identifier* (see 6.13) or by the *operator* “@” (see 6.4) or by the *delimiter* “:” (see 6.3). The usage of an identifier, an operator, or a delimiter as ALF type is defined by ALF language rules concerning the particular ALF type. The identifier can be a predefined *keyword* (see 6.13.7).

The *ALF name* is defined by an *identifier* (see 6.13) or by a *control expression* (see 9.4). Depending on the ALF type, the ALF name is mandatory or optional or not applicable. The usage of an identifier or a control expression as ALF name is defined by ALF language rules concerning the particular ALF type. The ALF name is optionally preceded by an *index* (see 6.6) to specify a *vectorized object*. Another index can optionally succeed the ALF name to specify a two-dimensional vectorized object. A two-dimensional vectorized object shall be called *matrix object*. An object without index shall be called *scalar object*. The usage of an index in conjunction with an ALF name is defined by ALF language rules concerning the particular ALF type.

The *ALF value* is defined by a *number* (see 6.5), a *multiplier prefix symbol* (see 6.7), an *identifier* (see 6.13), a *quoted string* (see 6.14), a *bit literal* (see 6.8), a *based literal* (see 6.9), an *edge value* (see 6.12), an *arithmetic expression* (see 10.1), a *boolean expression* (see 9.9), or a *control expression* (see 9.4). Depending on the type of the ALF statement, the ALF value is mandatory or optional or not applicable. The usage of a particular kind of ALF value is defined by ALF language rules concerning the particular ALF type.

An ALF statement can contain one or more other ALF statements. The former is called *parent* of the latter. Conversely, the latter is called *child* of the former. An ALF statement with a child is called a *compound* ALF statement. An ALF statement that is related to another ALF statement by ancestry in the parent/child relationship is called an *ancestor* of the other ALF statement. Conversely, the latter is called a *descendant* of the former.

An ALF statement containing one or more ALF values, possibly interspersed with the delimiters “;” or “:”, is called a *semi compound* ALF statement. The items between the delimiters “{“ and “}” are called *contents*.

of the ALF statement. The usage of the delimiters “;” or “:” within the contents of an ALF statement is defined by ALF language rules concerning the particular ALF statement.

An ALF statement without child is called an *atomic* ALF statement. An ALF statement which is either compound or semi compound is called a *non-atomic* ALF statement.

Example

- a) ALF statement describing an unnamed object without value:

```
ARBITRARY_ALF_TYPE {
    // put children here
}
```
- b) ALF statement describing an unnamed object with value:

```
ARBITRARY_ALF_TYPE = arbitrary_ALF_value;
or
ARBITRARY_ALF_TYPE = arbitrary_ALF_value {
    // put children here
}
```
- c) ALF statement describing a named object without value:

```
ARBITRARY_ALF_TYPE arbitrary_ALF_name;
or
ARBITRARY_ALF_TYPE arbitrary_ALF_name {
    // put children here
}
```
- d) ALF statement describing a named object with value:

```
ARBITRARY_ALF_TYPE arbitrary_ALF_name = arbitrary_ALF_value;
or
ARBITRARY_ALF_TYPE arbitrary_ALF_name = arbitrary_ALF_value {
    // put children here
}
```

End of example

5.2 Categories of ALF statements

In this subclause, the terms *statement*, *type*, *name*, *value* are used for shortness in lieu of *ALF statement*, *ALF type*, *ALF name*, *ALF value*, respectively.

Statements are divided into the following categories: *generic object*, *library-specific object*, *arithmetic model*, *arithmetic submodel*, *arithmetic model container*, *geometric model*, *annotation*, *annotation container*, and *auxiliary statement*, as shown in Table 1.

Table 1—Categories of ALF statements

Category	Purpose	Syntax particularity
Generic object	Provide a definition for use within other ALF statements.	Statement is atomic, semi compound, or compound. Name is mandatory. Value is either mandatory or not applicable.
Library-specific object	Describe the contents of a IC technology library.	Statement is atomic or compound. Name is mandatory. Value does not apply. Category of parent is <i>library-specific object</i> .
Arithmetic model	Describe an abstract mathematical quantity that can be calculated and possibly measured within the design of an IC.	Statement is atomic or compound. Name is optional. Value is mandatory, if atomic.
Arithmetic submodel	Describe an arithmetic model under a specific measurement condition.	Statement is atomic or compound. Name does not apply. Value is mandatory, if atomic. Category of parent is <i>arithmetic model</i> .
Arithmetic model container	Provide a context for an arithmetic model.	Statement is compound. Name and value do not apply. Category of child is <i>arithmetic model</i> .
Geometric model	Describe an abstract geometry used in physical design of an IC.	Statement is semi compound or compound. Name is optional. Value does not apply.
Annotation	Provide a qualifier or a set of qualifiers for an ALF statement.	Statement is atomic or semi compound. Name does not apply. Value is mandatory, if atomic. Value does not apply, if semi compound.
Annotation container	Provide a context for an annotation.	Statement is compound. Name and value do not apply. Category of child is <i>annotation</i> .
Auxiliary statement	Provide an additional description within the context of a library-specific object, an arithmetic model, an arithmetic submodel, geometric model, or another auxiliary statement.	Dependent on subcategory.

Figure 5 illustrates the parent/child relationship between categories of statements.

More detailed rules for parent/child relationships for particular types of statements apply.

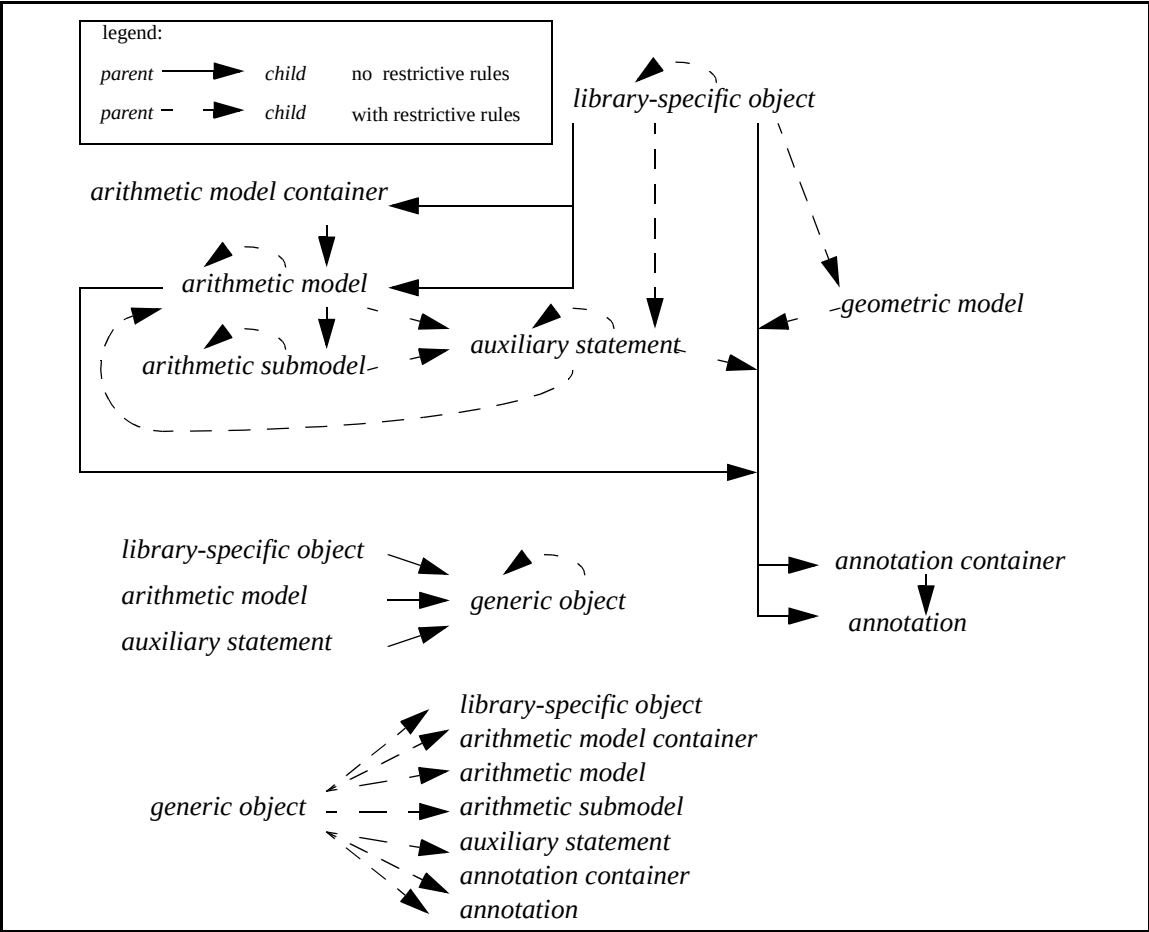


Figure 5—Parent/child relationship between ALF statements

5.3 Generic objects and library-specific objects

Statements with mandatory names are called *objects*, i.e., *generic object* and *library-specific object*. Table 2 lists the keywords and items in the category *generic object*. The keywords used in this category are called *generic keywords*.

Table 2—Generic objects

Keyword	Item	Subclause
ALIAS	Alias declaration	See 7.7.
CONSTANT	Constant declaration	See 7.8.
CLASS	Class declaration	See 7.12.
GROUP	Group declaration	See 7.14.
KEYWORD	Keyword declaration	See 7.9.

Table 2—Generic objects (*continued*)

Keyword	Item	Subclause
SEMANTICS	Semantics declaration	See 7.10.
TEMPLATE	Template declaration	See 7.15.

Table 3 lists the keywords and items in the category *library-specific object*. The keywords used in this category are called *library-specific keywords*.

Table 3—Library-specific objects

Keyword	Item	Subclause
ANTENNA	Antenna declaration	See 8.21.
ARRAY	Array declaration	See 8.27.
BLOCKAGE	Blockage declaration	See 8.22.
CELL	Cell declaration	See 8.4.
LAYER	Layer declaration	See 8.16.
LIBRARY	Library declaration	See 8.2.
NODE	Node declaration	See 8.12.
PATTERN	Pattern declaration	See 8.29.
PIN	Pin declaration	See 8.6.
PINGROUP	Pin group declaration	See 8.7.
PORT	Port declaration	See 8.23.
PRIMITIVE	Primitive declaration	See 8.9.
REGION	Region declaration	See 8.31.
RULE	Rule declaration	See 8.20.
SITE	Site declaration	See 8.25.
SUBLIBRARY	Sublibrary declaration	See 8.2.
VECTOR	Vector declaration	See 8.14.
VIA	Via declaration	See 8.18.
WIRE	Wire declaration	See 8.10.

Figure 6 illustrates the parent/child relationship between statements within the category *library-specific object*.

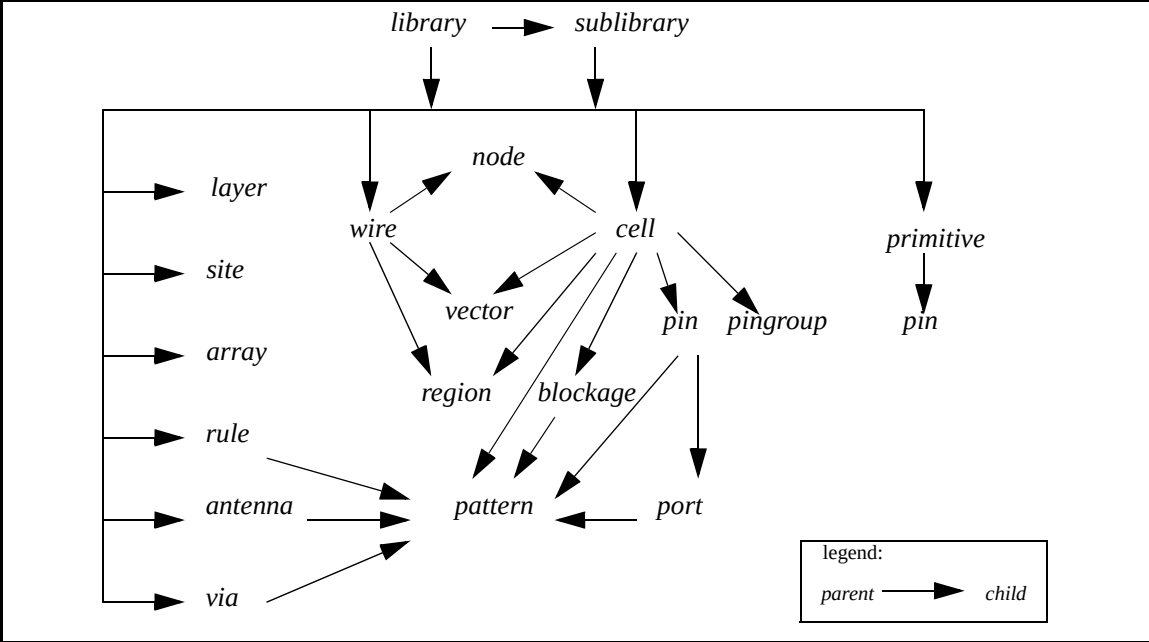


Figure 6—Parent/child relationship among library-specific objects

A parent can have multiple library-specific objects of the same type as children. Each child is distinguished by name.

5.4 Singular statements and plural statements

Auxiliary statements with predefined keywords are divided in the following subcategories: *singular statement* and *plural statement*.

Auxiliary statements with predefined keywords and without name are called *singular statements*. Auxiliary statements with predefined keywords and with name, yet without value, are called *plural statements*.

Table 4 lists the singular statements.

Table 4—Singular statements

Keyword	Item	Value	Complexity	Subclause
FUNCTION	Function statement	N/A	Compound	See 9.1.
TEST	Test statement	N/A	Compound	See 9.2.
RANGE	Range statement	N/A	Semi compound	See 9.8.
FROM	From statement	N/A	Compound	See 10.12.
TO	To statement	N/A	Compound	See 10.12.
VIOLATION	Violation statement	N/A	Compound	See 10.10.
HEADER	Header statement	N/A	Compound	See 10.4.
TABLE	Table statement	N/A	Semi compound	See 10.4.

Table 4—Singular statements (*continued*)

Keyword	Item	Value	Complexity	Subclause
EQUATION	Equation statement	N/A	Semi compound	See 10.4.
BEHAVIOR	Behavior statement	N/A	Compound	See 9.4.
STRUCTURE	Structure statement	N/A	Compound	See 9.5.
NON_SCAN_CELL	Non-scan cell statement	Optional	Compound or semi compound	See 9.7.
ARTWORK	Artwork statement	Mandatory	Compound or atomic	See 9.19.

Table 5 lists the plural statements.

Table 5—Plural statements

Keyword	Item	Name	Complexity	Subclause
STATETABLE	State table statement	Optional	Semi compound	See 9.6.
@	Control statement	Mandatory	Compound	See 9.4.
:	Alternative control statement	Mandatory	Compound	See 9.4.

Figure 7 illustrates the parent/child relationship for singular statements and plural statements.

A parent can have at most one child of a particular type in the category singular statements, but multiple children of a particular type in the category plural statements.

5.5 Instantiation statement and assignment statement

Auxiliary statements without predefined keywords use the name of an object as keyword. Such statements are divided in the following subcategories: *instantiation statement* and *assignment statement*.

Compound or semi compound statements using the name of an object as keyword are called *instantiation statements*. Their purpose is to specify an instance of the object.

Table 6 lists the instantiation statements.

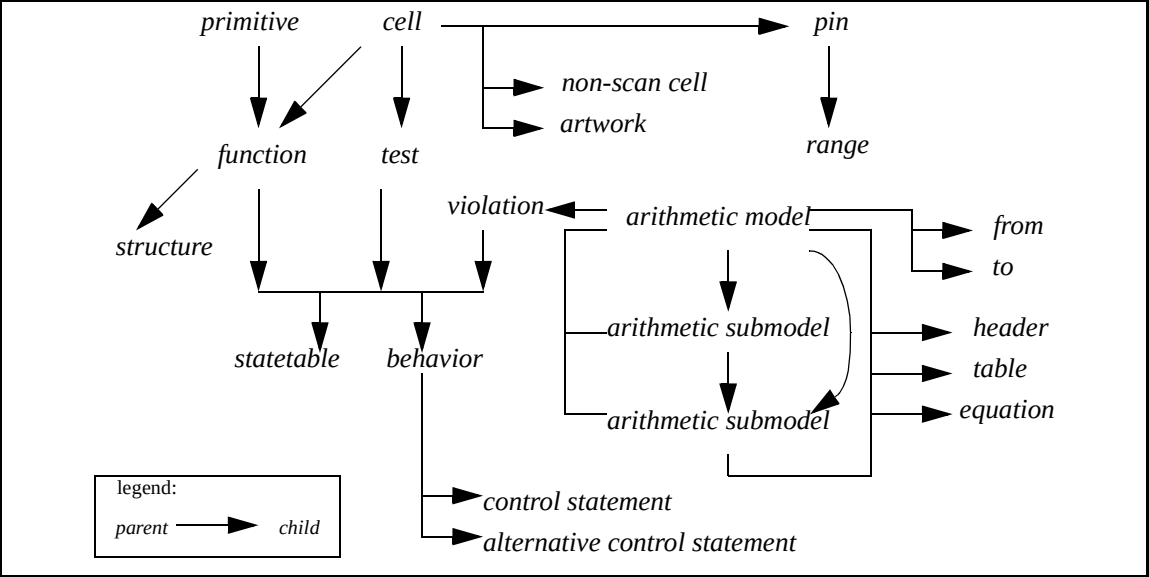


Table 6—Instantiation statements

Item	Subclause
Cell instantiation	See 9.5.
Primitive instantiation	See 9.4.
Template instantiation	See 7.16.
Via instantiation	See 9.20.
Wire instantiation	See 9.15.

Atomic statements without name using an identifier as keyword that has been defined within the context of another object are called assignment statements. A value is mandatory for assignment statements, as their purpose is to assign a value to the identifier. Such an identifier is called a *variable*.

Table 7 lists the assignment statements.

Table 7—Assignment statements

Item	Subclause
Pin assignment	See 9.3.2, Syntax 68.
Arithmetic assignment	See 7.16, Syntax 42.
Boolean assignment	See 9.4, Syntax 69.

Figure 8 illustrates the parent/child relationship involving instantiation and assignment statements.

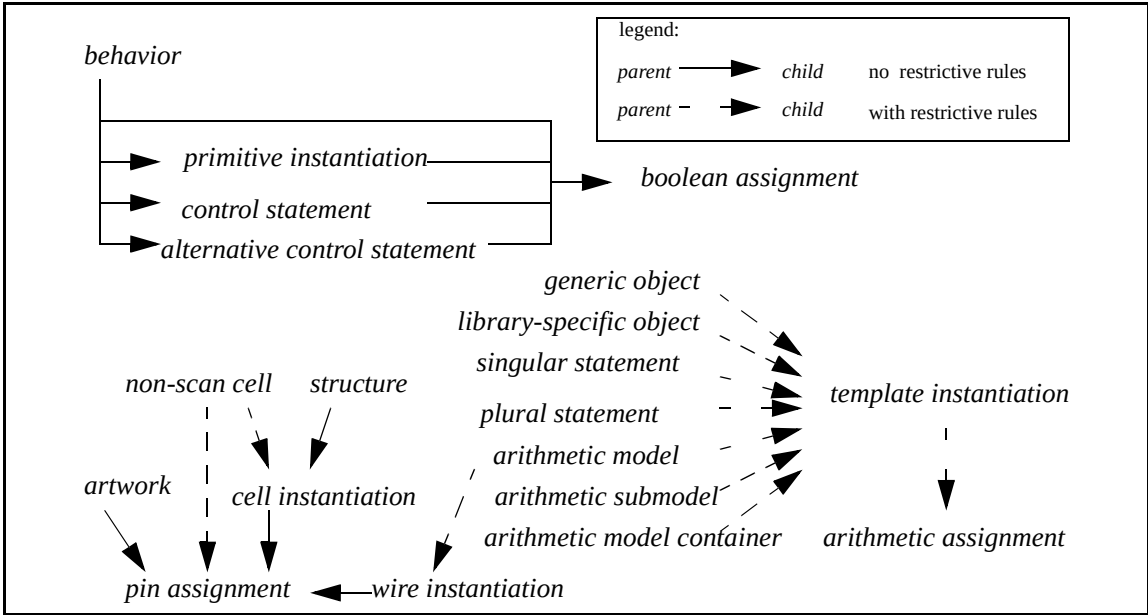


Figure 8—Parent/child relationship involving instantiation and assignment statements

A parent can have multiple children using the same keyword in the category instantiation statement, but at most one child using the same variable in the category assignment statement.

5.6 Annotation, arithmetic model, and related statements

Multiple keywords are predefined in the categories *arithmetic model*, *arithmetic model container*, *arithmetic submodel*, *annotation*, *annotation container*, and *geometric model*. Their semantics are established within the context of their parent. Therefore, they are called *context-sensitive keywords*. In addition, the ALF language allows additional definition of keywords in these categories. Table 8 provides a reference to subclauses where more definitions about these categories can be found.

Table 8—Other categories of ALF statements

Item	Subclause
Arithmetic model	See 10.3, Syntax 82.
Arithmetic submodel	See 10.7, Syntax 96.
Arithmetic model container	See 10.8, Syntax 97.
Annotation	See 7.3, Syntax 31.
Annotation container	See 7.4, Syntax 32.
Geometric model	See 9.16, Syntax 77.

There exist predefined keywords with generic semantics in the category *annotation* and *annotation container*. They are called *generic keywords*, comparable to keywords for *generic objects*. Table 9 lists the generic keywords in the category annotation and annotation container.

Table 9—Annotations and annotation containers with generic keyword

Keyword	Item/category/subcategory	Subclause
PROPERTY	Annotation container	See 7.6.
ATTRIBUTE	Multi-value annotation	See 7.5.
INFORMATION	Annotation container	See 8.3.2.

Table 10 lists predefined keywords in categories related to arithmetic model.

Table 10—Keywords related to arithmetic model

Keyword	Item/category/subcategory	Subclause
LIMIT	Arithmetic model container	See 10.8.2.
MIN	Arithmetic submodel, also operator within <i>arithmetic expression</i>	See 10.2.3, 10.5.
MAX	Arithmetic submodel, also operator within <i>arithmetic expression</i>	See 10.2.3, 10.5.
TYP	Arithmetic submodel	See 10.5.
DEFAULT	Annotation	See 10.9.4.
ABS	Operator within <i>arithmetic expression</i>	See 10.2.3.
EXP	Operator within <i>arithmetic expression</i>	See 10.2.3.
LOG	Operator within <i>arithmetic expression</i>	See 10.2.3.

The definitions of other predefined keywords, especially in the category arithmetic model, can be self-described in ALF using the *keyword declaration* statement (see 7.9).

5.7 Statements for parser control

Table 11 provides a reference to statements used for ALF parser control.

Table 11—Statements for ALF parser control

Keyword	Statement	Subclause
INCLUDE	Include statement	See 7.17.
ASSOCIATE	Associate statement	See 7.18.
ALF_REVISION	Revision statement	See 7.19.

The statements for parser control do not necessarily follow the ALF metalanguage shown in Syntax 1.

5.8 Name space and visibility of statements

The following rules for name space and visibility shall apply.

- a) A statement shall be visible within its parent statement, but not outside its parent statement.
- b) A statement visible within another statement shall also be visible within a child of that other statement.
- c) All objects (i.e., generic objects and library-specific objects) shall share a common name space within their scope of visibility. No object shall use the same name as any other visible object. Conversely, an object can use the same name as any other object outside the scope of its visibility.
- d) The following exception of rule c) is allowed for specific objects and with specific semantic implications. An object of the same type and the same name can be redeclared, if semantic support for this redeclaration is provided. The purpose of such a redeclaration is to supplement the original declaration with new children statements which augment the original declaration without contradicting it.
- e) All statements with optional names (i.e., property, arithmetic model, geometric model) shall share a common name space within their scope of visibility. No statement with optional name shall use the same name as any other visible statement with optional name. Conversely, a statement can use the same optional name as any other statement with optional name outside the scope of its visibility.

6. Lexical rules

This clause discusses the lexical rules.

The ALF source text files shall be a stream of *lexical tokens* and *whitespace*. Lexical tokens shall be divided into the categories *delimiter*, *operator*, *comment*, *number*, *bit literal*, *based literal*, *edge*, *quoted string*, and *identifier*.

Each lexical token shall be composed of one or more characters. Whitespace shall be used to separate lexical tokens from each other. Whitespace shall not be allowed within a lexical token with the exception of *comment* and *quoted string*.

The specific rules for construction of lexical tokens and for usage of whitespace are defined in this clause.

6.1 Character set

This standard shall use the ASCII character set [see ISO/IEC 8859-1:1998(E)].

The *ASCII character set* shall be divided into the following categories: *whitespace*, *letter*, *digit*, and *special*, as shown in Syntax 2.

```

character ::=
    whitespace
    | letter
    | digit
    | special
whitespace ::=
    space | horizontal_tab | new_line | vertical_tab | form_feed | carriage_return
letter ::=
    uppercase | lowercase
uppercase ::=
    A | B | C | D | E | F | G | H | I | J | K | L | M
    | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
lowercase ::=
    a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
digit ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
special ::=
    & | | ^ | ~ | + | - | * | / | % | ? | ! | : | ; | , | " | ' | @ | = | \ | . | $ | _ | #
    | ( | ) | < | > | [ | ] | { | }

```

Syntax 2—ASCII character set divided into categories

Table 12 shows the list of *whitespace* characters and their ASCII code.

Table 12—List of whitespace characters

Name	ASCII code (octal)
Space	040
Horizontal tab	011
New line	012
Vertical tab	013
Form feed	014
Carriage return	015

Table 13 shows the list of *special* characters and their names used in this standard.

Table 13—List of special characters

Symbol	Name
&	Ampersand
	Vertical bar
^	Caret
~	Tilde
+	Plus

Table 13—List of special characters (*continued*)

Symbol	Name
-	Dash
*	Asterisk
/	Slash
%	Percent
?	Question mark
!	Exclamation mark
:	Colon
;	Semicolon
,	Comma
"	Double quote
'	Single quote
@	At sign
=	Equal sign
\	Backslash
.	Dot
\$	Dollar
_	Underscore
#	Pound
()	Parenthesis (open, close)
< >	Angular bracket (open, close)
[]	Square bracket (open, close)
{ }	Curly bracket (open, close)

6.2 Comment

A *comment* shall be divided into the subcategories *in-line comment* and *block comment*, as shown in Syntax 3.

The start of an in-line comment shall be determined by the occurrence of two subsequent *slash* characters without whitespace in-between. The end of an in-line comment shall be determined by the occurrence of a *new line* or of a *carriage return* character.

```

comment ::=
    in_line_comment
  | block_comment
in_line_comment ::=
    //{character} new_line
  | //{character} carriage_return
block_comment ::=
    /*{character}*/

```

Syntax 3—Comment

The start of a block comment shall be determined by the occurrence of a *slash* character followed by an *asterix* without whitespace in-between. The end of a block comment shall be determined by the occurrence of an *asterix* character followed by a *slash* character.

A comment shall have the same semantic meaning as a whitespace. Therefore, no syntax rule shall involve a comment.

6.3 Delimiter

The special characters shown in Syntax 4 shall be considered *delimiters*.

```

delimiter ::=
    ( ) [ ] { } | : ; | ,

```

Syntax 4—Delimiter

When appearing in a syntax rule, a delimiter shall be used to indicate the end of a statement or of a partial statement, or the beginning and end of an expression or of a partial expression.

6.4 Operator

Operators shall be divided into the following subcategories: *arithmetic operator*, *boolean operator*, *relational operator*, *shift operator*, *event operator*, and *meta operator*, as shown in Syntax 5.

When appearing in a syntax rule, an operator shall be used within a statement or within an expression. An operator with one operand shall be called *unary operator*. An unary operator shall precede the operand. An operator with two operands shall be called a *binary operator*. A binary operator shall succeed the first operand and precede the second operand.

operator ::=
arithmetic_operator
boolean_operator
relational_operator
shift_operator
event_operator
meta_operator
arithmetic_operator ::=
+ - * / % **
boolean_operator ::=
&& ~& ~ ^ ~^ ~ ! &
relational_operator ::=
== != >= <= > <
shift_operator ::=
<< >>
event_operator ::=
-> ~> <-> <~> &> <&>
meta_operator ::=
= ? @

Syntax 5—Operator

6.4.1 Arithmetic operator

Table 14 shows the list of arithmetic operators and their names used in this standard.

Table 14—List of arithmetic operators

Symbol	Operator name	Unary / binary	Subclause
+	Plus	Binary	See 9.11.4.
—	Minus	Both	See 9.11.4.
*	Multiply	Binary	See 9.11.4.
/	Divide	Binary	See 9.11.4.
%	Modulus	Binary	See 9.11.4.
**	Power	Binary	See 10.2.2.

Arithmetic operators shall be used to specify arithmetic operations.

6.4.2 Boolean operator

Table 15 shows the list of boolean operators and their names used in this standard.

Table 15—List of boolean operators

Symbol	Operator name	Unary / binary	Subclause
!	Logical inversion	Unary	See 9.11.1.
&&	Logical and	Binary	See 9.11.1.
	Logical or	Binary	See 9.11.1.
~	Bit-wise inversion	Unary	See 9.11.2.
&	Bit-wise and	Both	See 9.11.2.
~&	Bit-wise nand	Both	See 9.11.2.
	Bit-wise or	Both	See 9.11.2.
~	Bit-wise nor	Both	See 9.11.2.
^	Exclusive or	Both	See 9.11.2.
~^	Exclusive nor	Both	See 9.11.2.

Boolean operators shall be used to specify boolean operations.

6.4.3 Relational operator

Table 16 shows the list of relational operators and their names used in this standard.

Table 16—List of relational operators

Symbol	Operator name	Unary / binary	Subclause
==	Equal	Binary	See 9.11.6.
!=	Not equal	Binary	See 9.11.6.
>	Greater	Binary	See 9.11.6.
<	Lesser	Binary	See 9.11.6.
>=	Greater or equal	Binary	See 9.11.6.
<=	Lesser or equal	Binary	See 9.11.6.

Relational operators shall be used to specify mathematical relationships between numbers.

6.4.4 Shift operator

Table 17 shows the list of shift operators and their names used in this standard.

Table 17—List of shift operators

Symbol	Operator name	Unary / binary	Subclause
<<	Shift left	Binary	See 9.11.5.
>>	Shift right	Binary	See 9.11.5.

Shift operators shall be used to specify manipulations of discrete mathematical values.

6.4.5 Event operator

Table 18 shows the list of event operators and their names used in this standard.

Table 18—List of event operators

Symbol	Operator name	Unary / binary	Subclause
->	Immediately followed by	Binary	See 9.13.2.
~>	Eventually followed by	Binary	See 9.13.2.
<->	Immediately following each other	Binary	See 9.13.3.
<~>	Eventually following each other	Binary	See 9.13.3.
&>	Simultaneous or immediately followed by	Binary	See 9.13.3.
<&>	Simultaneous or immediately following each other	Binary	See 9.13.3.

Event operators shall be used to express temporal relationships between discrete events.

6.4.6 Meta operator

Table 19 shows the list of meta operators and their names used in this standard.

Table 19—List of meta operators

Symbol	Operator name	Unary / binary	Subclause
=	Assignment	Binary	See 7.16, 9.3.2, 9.4.
?	Condition	Binary	See 9.13.5.
@	Control	Unary	See 9.4.

Meta operators shall be used to specify transactions between variables.

6.5 Number

Numbers shall be divided into subcategories *signed integer*, *signed real*, *unsigned integer*, and *unsigned real*. Furthermore, the categories *signed number*, *unsigned number*, *integer*, and *real* shall be defined as shown in Syntax 6.

```

number ::=
    signed_integer | signed_real | unsigned_integer | unsigned_real
signed_number ::=
    signed_integer | signed_real
unsigned_number ::=
    unsigned_integer | unsigned_real
integer ::=
    signed_integer | unsigned_integer
signed_integer ::=
    sign unsigned_integer
unsigned_integer ::=
    digit { [ _ ] digit }
real ::=
    signed_real | unsigned_real
signed_real ::=
    sign unsigned_real
unsigned_real ::=
    mantissa [ exponent ]
    | unsigned_integer exponent
sign ::=
    + | -
mantissa ::=
    . unsigned_integer
    | unsigned_integer . [ unsigned_integer ]
exponent ::=
    E [ sign ] unsigned_integer
    | e [ sign ] unsigned_integer

```

Syntax 6—Number

A number shall be used to represent a numerical quantity.

6.6 Index value and index

An *index value* shall be defined as shown in Syntax 7.

```

index_value ::=
    unsigned_integer | atomic_identifier

```

Syntax 7—Index value

The purpose of an *index value* is to represent a position within a range of discrete, countable values. A discrete, countable value shall be represented by an *unsigned integer* (see 6.5). The usage of *atomic identifier* (see 6.13) as an index value shall only be allowed, if the semantic interpretation of the atomic identifier resolves to a value of the category *unsigned integer*.

An index value can represent a particular position within a *pin* of the category *vector pin*, a *matrix pin* (see 8.6), or a *pingroup* (see 8.7).

An index value can also be used in the context of a *group* declaration (see 7.14) and in the context of a *range* statement (see 9.8).

An *index* shall be defined as shown in Syntax 8.

```
index ::=
    single_index | multi_index
single_index ::=
    [ index_value ]
multi_index ::=
    [ index_value : index_value ]
```

Syntax 8—Index

An *index* shall be used in conjunction with the name of a *pingroup*, a *vector pin*, or a *matrix pin*. A *single index* shall represent a particular scalar within a one-dimensional vector or a particular one-dimensional vector within a two-dimensional matrix. A *multi index* shall represent a range of scalars or a range of vectors, wherein the position of the most significant bit (MSB) is specified by the left index value and the position of the least significant bit (LSB) is specified by the right index value.

6.7 Multiplier prefix symbol and multiplier prefix value

A *multiplier prefix symbol* shall be defined as shown in Syntax 9.

```
multiplier_prefix_symbol ::=
    unity { letter } | K { letter } | M E G { letter } | G { letter }
    | M { letter } | U { letter } | N { letter } | P { letter } | F { letter }
unity ::=
    1
K ::=
    K | k
M ::=
    M | m
E ::=
    E | e
G ::=
    G | g
U ::=
    U | u
N ::=
    N | n
P ::=
    P | p
F ::=
    F | f
```

Syntax 9—Multiplier prefix symbol

The purpose of a multiplier prefix symbol is the specification of a multiplier for the base unit associated with an *arithmetic model* (see 10.3). Only the leading characters of the multiplier prefix symbol shall be used for identification of the corresponding number. Optional subsequent letters can be used to indicate the base unit. For example, “pF” can be used to denote “picofarad,” “MHz” can be used to denote “megahertz,” etc.

A multiplier prefix symbol shall relate to the *International System of Units* [B9], as shown in Table 20.

Table 20—Multiplier prefix symbol and corresponding SI-prefix

Lexical token	SI-prefix (symbol)	SI-prefix (word)	Numerical value
F	f	femto	1e-15
P	p	pico	1e-12
N	n	nano	1e-9
U	μ	micro	1e-6
M	m	milli	1e-3
unity	1	one	1e0
K	k	kilo	1e+3
MEG	M	mega	1e+6
G	G	giga	1e+9

A multiplier prefix value shall be defined as shown in Syntax 10.

multiplier_prefix_value ::= unsigned_number multiplier_prefix_symbol

Syntax 10—Multiplier prefix value

The *multiplier prefix value* shall be represented either as an *unsigned number* (see 6.5) or a *multiplier prefix symbol* (see 6.7). An application shall interpret a multiplier prefix value semantically as an unsigned number.

6.8 Bit literal

Bit literals shall be divided into the subcategories *alphanumeric bit literal* and *symbolic bit literal*, as shown in Syntax 11.

bit_literal ::= alphanumeric_bit_literal symbolic_bit_literal alphanumeric_bit_literal ::= numeric_bit_literal alphabetic_bit_literal numeric_bit_literal ::= 0 1 alphabetic_bit_literal ::= X Z L H U W x z l h u w symbolic_bit_literal ::= ? *
--

Syntax 11—Bit literal

Bit literals shall be used to specify scalar values within a boolean value system (see 9.10).

6.9 Based literal

Based literals shall be divided into subcategories *binary based literal*, *octal based literal*, *decimal based literal*, and *hexadecimal based literal*, as shown in Syntax 12.

```
based_literal ::=
    binary_based_literal | octal_based_literal | decimal_based_literal | hexadecimal_based_literal
binary_based_literal ::=
    binary_base bit_literal { [ _ ] bit_literal }
binary_base ::=
    'B' | 'b'
octal_based_literal ::=
    octal_base octal_digit { [ _ ] octal_digit }
octal_base ::=
    'O' | 'o'
octal_digit ::=
    bit_literal | 2 | 3 | 4 | 5 | 6 | 7
decimal_based_literal ::=
    decimal_base digit { [ _ ] digit }
decimal_base ::=
    'D' | 'd'
hexadecimal_based_literal ::=
    hexadecimal_base hexadecimal_digit { [ _ ] hexadecimal_digit }
hexadecimal_base ::=
    'H' | 'h'
hexadecimal_digit ::=
    octal_digit | 8 | 9
    | A | B | C | D | E | F
    | a | b | c | d | e | f
```

Syntax 12—Based literal

Based literals shall be used to specify vectorized values within a boolean value system.

6.10 Boolean value

A *boolean value* shall be defined as shown in Syntax 13.

```
boolean_value ::=
    alphanumeric_bit_literal | based_literal | integer
```

Syntax 13—Boolean value

The semantics of a boolean value are explained in 9.10.

6.11 Arithmetic value

An *arithmetic value* shall be defined as shown in Syntax 14.

An arithmetic value shall represent data for an *arithmetic model* (see 10.3) or for an *arithmetic assignment* (see 7.16). Semantic restrictions apply, depending on the particular type of arithmetic model.

```

arithmetic_value ::=
    number | identifier | bit_literal | based_literal

```

Syntax 14—Arithmetic value

6.12 Edge literal and edge value

Edge literals shall be divided into subcategories *bit edge literal*, *based edge literal*, and *symbolic edge literal*, as shown in Syntax 15.

```

edge_literal ::=
    bit_edge_literal
    | based_edge_literal
    | symbolic_edge_literal
bit_edge_literal ::=
    bit_literal bit_literal
based_edge_literal ::=
    based_literal based_literal
symbolic_edge_literal ::=
    ?~ | ?! | ?-

```

Syntax 15—Edge literal

Edge literals shall be used to specify a change of value within a boolean system. In general, bit edge literals shall specify a change of a scalar value, based edge literals shall specify a change of a vectorized value, and symbolic edge literals shall specify a change of a scalar or a vectorized value.

An *edge value* shall be defined as shown in Syntax 16.

```

edge_value ::=
    ( edge_literal )

```

Syntax 16—Edge value

An edge value shall be used to represent a stand-alone edge literal that is not embedded in a vector expression.

6.13 Identifier

Identifiers shall be divided into the subcategories *atomic identifier*, *indexed identifier*, *hierarchical identifier*, and *escaped identifier*, as shown in Syntax 17. The subcategory *atomic identifier* shall be further divided into *non-escaped identifier* and *placeholder identifier*. The subcategory *hierarchical identifier* shall be further divided into *full hierarchical identifier* and *partial hierarchical identifier*.

```

identifier ::=
    atomic_identifier | indexed_identifier | hierarchical_identifier | escaped_identifier
atomic_identifier ::=
    non_escaped_identifier | placeholder_identifier
hierarchical_identifier ::=
    full_hierarchical_identifier | partial_hierarchical_identifier

```

Syntax 17—Identifier

An identifier shall be used to specify an *ALF name* or an *ALF value*. An identifier can also appear as a *variable* in an *arithmetic expression* (see 10.1), in a *boolean expression* (see 9.9), or in a *vector expression* (see 9.12).

A lowercase character used within a keyword or within an identifier shall be considered equivalent to the corresponding uppercase character, i.e., ALF shall be case-insensitive. However, whenever an identifier is used to specify an ALF name, the usage of the exact uppercase or lowercase letters shall be preserved by the parser to enable usage of the same name by a case-sensitive application.

6.13.1 Non-escaped identifier

A *non-escaped identifier* shall be defined as shown in Syntax 18.

```
non_escaped_identifier ::=  
  letter { letter | digit | _ | $ | # }
```

Syntax 18—Non-escaped identifier

A non-escaped identifier shall be used when there is no lexical conflict, i.e., no appearance of a character with special meaning, and no semantic conflict, i.e., the identifier is not used elsewhere as a keyword.

6.13.2 Placeholder identifier

A *placeholder identifier* shall be defined as a *non-escaped identifier* enclosed by angular brackets without whitespace, as shown in Syntax 19.

```
placeholder_identifier ::=  
  < non_escaped_identifier >
```

Syntax 19—Placeholder identifier

A placeholder identifier shall be used to represent a formal parameter in a *template* statement (see 7.15), which is to be replaced by an actual parameter in a *template instantiation* statement (see 7.16).

6.13.3 Indexed identifier

An *indexed identifier* shall be defined as an *atomic identifier* followed by an *index* (see 6.6) without whitespace, as shown in Syntax 20.

```
indexed_identifier ::=  
  atomic_identifier index
```

Syntax 20—Indexed identifier

The *atomic identifier* shall be interpreted as the *ALF name* of a one- or two-dimensional object, i.e., a *vector pin* or a *matrix pin* (see 8.6). The *index* shall be interpreted as the position of a scalar element within a one-dimensional object or a one-dimensional slice within a two-dimensional object.

6.13.4 Full hierarchical identifier

A *full hierarchical identifier* shall be defined as shown in Syntax 21.

```

full_hierarchical_identifier ::=
    atomic_identifier [ index ] . atomic_identifier [ index ] { . atomic_identifier [ index ] }

```

Syntax 21—Hierarchical identifier

A *full hierarchical identifier* shall be used to specify a hierarchical name, i.e., the name of a child preceded by the name of its parent. A *dot* within a hierarchical identifier shall be used to separate a parent from a child.

6.13.5 Partial hierarchical identifier

A *partial hierarchical identifier* shall be defined as shown in Syntax 22.

```

partial_hierarchical_identifier ::=
    atomic_identifier [ index ] { . atomic_identifier [ index ] } . .
    { atomic_identifier [ index ] { . atomic_identifier [ index ] } . . }
    [ atomic_identifier [ index ] { . atomic_identifier [ index ] } ]

```

Syntax 22—Partial hierarchical identifier

A *partial hierarchical identifier* shall be used to specify an incomplete hierarchical name. The two dots shall indicate that the preceding atomic identifier is an *ancestor* of the subsequent atomic identifier. A partial hierarchical identifier terminated by two dots shall be interpreted as a reference to any possible *descendant* of the preceding ancestor.

NOTE—A restriction as to which descendant is applicable can be given by a particular syntax or semantic rule.

6.13.6 Escaped identifier

An *escaped identifier* shall be defined as shown in Syntax 23.

```

escaped_identifier ::=
    \ escapable_character { escapable_character }
    escapable_character ::=
        letter | digit | special

```

Syntax 23—Escaped identifier

An *escaped identifier* shall be used to legalize the usage of a special character or the usage of an identifier otherwise reserved as a keyword.

A *dot* within an escaped identifier shall be semantically interpreted in the same way as a dot within a *full hierarchical identifier* (see 6.13.4), unless the dot is immediately preceded by a *backslash*.

A lexical sequence of characters according to Syntax 8 (see 6.6) at the end of the escaped identifier or preceding a dot within the escaped identifier shall be interpreted as an *index* (see 6.6) in the same way as within a *full hierarchical identifier* or within an *indexed identifier* (see 6.13.3), unless the lexical sequence of characters is immediately preceded by a *backslash*.

A backslash within an escaped identifier shall semantically be considered part of an ALF name or of an ALF value designated by the escaped identifier, with exception of the leading backslash and a backslash immediately preceding a dot or an index.

Example

`\id1[0].id2[1].id3.id4` represents 3 levels of hierarchy.

The ancestor is the element at position **0** of the one-dimensional object **"id1"**. The child of **"id1[0]"** is the scalar object **"id2[1]"**. The child of **"id2[1]"** is the scalar object **"id3.id4"**.

NOTE—The scalar object **"id2[1]"** by itself has to be declared as **"\id2[1]"**. The scalar object **"id3.id4"** by itself has to be declared as **"\id3.id4"**.

End of example

6.13.7 Keyword identifier

Keywords shall be lexically equivalent to non-escaped identifiers. Predefined keywords are listed in Table 2, Table 3, Table 4, Table 5, Table 9, Table 10, and Table 11. Additional keywords are predefined in 7.9.

The predefined keywords in this standard shall follow a more restrictive lexical rule than general non-escaped identifiers, as shown in Syntax 24.

```
keyword_identifier ::=
    letter { [ _ ] letter }
```

Syntax 24—Keyword identifier

The reason for the more restrictive lexical rule is to encourage the use of words taken from a natural language as keywords. Words in a natural language are constructed from lexical characters only, not from numbers. The underscore can be used to indicate that there would be a whitespace or a dash in the word from the natural language.

NOTE—This standard presents keywords in all-uppercase letters for clarity.

6.14 Quoted string

A *quoted string* shall be a sequence of zero or more characters enclosed between two double-quote characters, as shown in Syntax 25.

```
quoted_string ::=
    " { character } "
```

Syntax 25—Quoted string

Within a quoted string, a sequence of characters starting with an *escape character* shall represent a symbol for another character, as shown in Table 21.

Table 21—Character symbols within a quoted string

Symbol	Character	ASCII code (octal)
<code>\g</code>	Alert or bell	007
<code>\h</code>	Backspace	010

Table 21—Character symbols within a quoted string (continued)

\t	Horizontal tab	011
\n	New line	012
\v	Vertical tab	013
\f	Form feed	014
\r	Carriage return	015
\"	Double quote	042
\\	Backslash	134
\ digit digit digit	ASCII character represented by three digit octal ASCII code	digit digit digit

The start of a quoted string shall be determined by a double-quote character. The end of a quoted string shall be determined by a double-quote character preceded by an even number of escape characters or by any other character than escape character.

6.15 String value

A *string value* shall be defined as shown in Syntax 26.

string_value ::= quoted_string identifier
--

Syntax 26—String value

A string value shall represent textual data in general and the name of a referenced object in particular.

6.16 Generic value

An *generic value* shall be defined as shown in Syntax 27.

generic_value ::= number multiplier_prefix_symbol identifier quoted_string bit_literal based_literal edge_value
--

Syntax 27—Generic value

A *generic value* shall be used as an *ALF value* for an *annotation* (see 7.3), for a *group* declaration (see 7.14), or for a *template* instantiation (see 7.16). Restrictions for applicable values in a particular context shall be defined by semantic rules.