A standard for an Advanced Library Format (ALF) describing Integrated Circuit (IC) technology, cells, and blocks

This is an unapproved draft for an IEEE standard and subject to change

IEEE 1596 Draft 0

August 19, 2001

put in IEEE verbage

The following individuals contributed to the creation, editing, and review of this document

Wolfgang Roethig, Ph.D.	wroethig@eda.org	Official Reporter and WG Chair
Joe Daniels	chippewea@aol.org	Technical Editor

Revision history:

IEEE 1596 Draft 0 August 19, 2001

Table of Contents

List of Figures

List of Tables

IEEE Standard for an Advanced Library Format (ALF) describing Integrated Circuit (IC) technology, cells, and blocks

1. Introduction

Add a lead-in OR change this to parallel an IEEE intro section

1.1 Motivation

Designing digital integrated circuits has become an increasingly complex process. More functions get integrated into a single chip, yet the cycle time of electronic products and technologies has become considerably shorter. It would be impossible to successfully design a chip of today's complexity within the time-to-market constraints without extensive use of EDA tools, which have become an integral part of the complex design flow. The efficiency of the tools and the reliability of the results for simulation, synthesis, timing and power analysis, layout and extraction rely significantly on the quality of available information about the cells in the technology library.

New challenges in the design flow, especially signal integrity, arise as the traditional tools and design flows hit their limits of capability in processing complex designs. As a result, new tools emerge, and libraries are needed in order to make them work properly. Library creation (generation) itself has become a very complex process and the choice or rejection of a particular application (tool) is often constrained or dictated by the availability of a library for that application. The library constraint can prevent designers from choosing an application program that is best suited for meeting specific design challenges. Similar considerations can inhibit the development and productization of such an application program altogether. As a result, competitiveness and innovation of the whole electronic industry can stagnate.

In order to remove these constraints, an industry-wide standard for library formats, the Advanced Library Format (ALF), is proposed. It enables the EDA industry to develop innovative products and ASIC designers to choose the best product without library format constraints. Since ASIC vendors have to support a multitude of libraries according to the preferences of their customers, a common standard library is expected to significantly reduce the library development cycle and facilitate the deployment of new technologies sooner.

1.2 Goals

The basic goals of the proposed library standard are

— *simplicity* - library creation process needs to be easy to understand and not become a cumbersome process only known by a few experts.
— *generality* - tools of any level of sophistication need to be able to retrieve necessary information from the library.
— *expandability* - this needs to be done for early adoption and future enhancement possibilities.
— *flexibility* - the choice of keeping information in one library or in separate libraries needs to be in the hand of the user not the standard.
— *efficiency* - the complexity of the design information requires the process of retrieving information from the library does not become a bottleneck. The right trade-off between compactness and verbosity needs to be established.
— *ease of implementation* - backward compatibility with existing libraries shall be provided and translation to the new library needs to be an easy task.
— *conciseness* - unambiguous description and accuracy of contents shall be detailed.
— *acceptance* - there needs to be a preference for the new standard library over existing libraries.

1.3 Target applications

The fundamental purpose of ALF is to serve as the primary database for all third-party applications of ASIC cells. In other words, it is an elaborate and formalized version of the *databook*.

In the early days, databooks provided all the information a designer needed for choosing a cell in a particular application: Logic symbols, schematics, and a truth table provided the functional specification for simple cells. For more complex blocks, the name of the cell (e.g., asynchronous ROM, synchronous 2-port RAM, or 4-bit synchronous up-down counters) and timing diagrams conveyed the functional information. The performance characteristics of each cell were provided by the loading characteristics, delay and timing constraints, and some information about DC and AC power consumption. The designers chose the cell type according to the functionality, estimated the performance of the design, and eventually re-implemented it in an optimized way as necessary to meet performance constraints.

Design automation enabled tremendous progress in efficiency, productivity, and the ability to deal with complexity, yet it did not change the fundamental requirements for ASIC design. Therefore, ALF needs to provide models with *functional* information and *performance* information, primarily including timing and power. Signal integrity characteristics, such as noise margin can also be included under performance category. Such information is typically found in any databook for analog cells. At deep sub-micron levels, digital cells behave similar to analog cells as electronic devices bound by physical laws and therefore are not infinitely robust against noise.

Table 1 shows a list of applications used in ASIC design flow and their relationship to ALF.

NOTE — ALF covers *library* data, whereas *design* data needs to be provided in other formats.

Table 1—Target applications and models supported by ALF

Application	Functional model	Performance model	Physical model
Simulation	Derived from ALF	N/A	N/A
Synthesis	Supported by ALF	Supported by ALF	Supported by ALF
Design for test	Supported by ALF	N/A	N/A

Table 1—Target applications and models supported by ALF (Continued)

Application	Functional model	Performance model	Physical model
Design planning	Supported by ALF	Supported by ALF	Supported by ALF
Timing analysis	N/A	Supported by ALF	N/A
Power analysis	N/A	Supported by ALF	N/A
Signal integrity	N/A	Supported by ALF	N/A
Layout	N/A	N/A	Supported by ALF

Historically, a functional model was virtually identical to a simulation model. A functional gate-level model was used by the proprietary simulator of the ASIC company and it was easy to lump it together with a rudimentary timing model. Timing analysis was done through dynamic functional simulation. However, with the advanced level of sophistication of both functional simulation and timing analysis, this is no longer the case. The capabilities of the functional simulators have evolved far beyond the gate-level and timing analysis has been decoupled from simulation.

RTL design planning is an emerging application type aiming to produce "virtual prototypes" of complex for system-on-chip (SOC) designs. RTL design planning is thought of as a combination of some or all of RTL floorplanning and global routing, timing budgeting, power estimation, and functional verification, as well as analysis of signal integrity, EMI, and thermal effects. The library components for RTL design planning range from simple logic gates to parameterizeable macro-functions, such as memories, logic building blocks, and cores.

From the point of view of library requirements, applications involved in RTL design planning need functional, performance, and physical data. The functional aspect of design planning includes RTL simulation and formal verification. The performance aspect covers timing and power as primary issues, while signal integrity, EMI, and thermal effects are emerging issues. The physical aspect is floorplanning. As stated previously, the functional and performance models of components can be described in ALF.

ALF also covers the requirements for physical data, including layout. This is important for the new generation of tools, where logical design merges with physical design. Also, all design steps involve optimization for timing, power, signal integrity, i.e. electrical correctness and physical correctness. EDA tools must be knowledgeable about an increasing number of design aspects. For example, a place and route tool must consider congestion as well as timing, crosstalk, electromigration, antenna rules etc. Therefore it is a logical step to combine the functional, electrical and physical models needed by such a tool in a unified library.

Figure 1 shows how ALF provides information to various design tools.

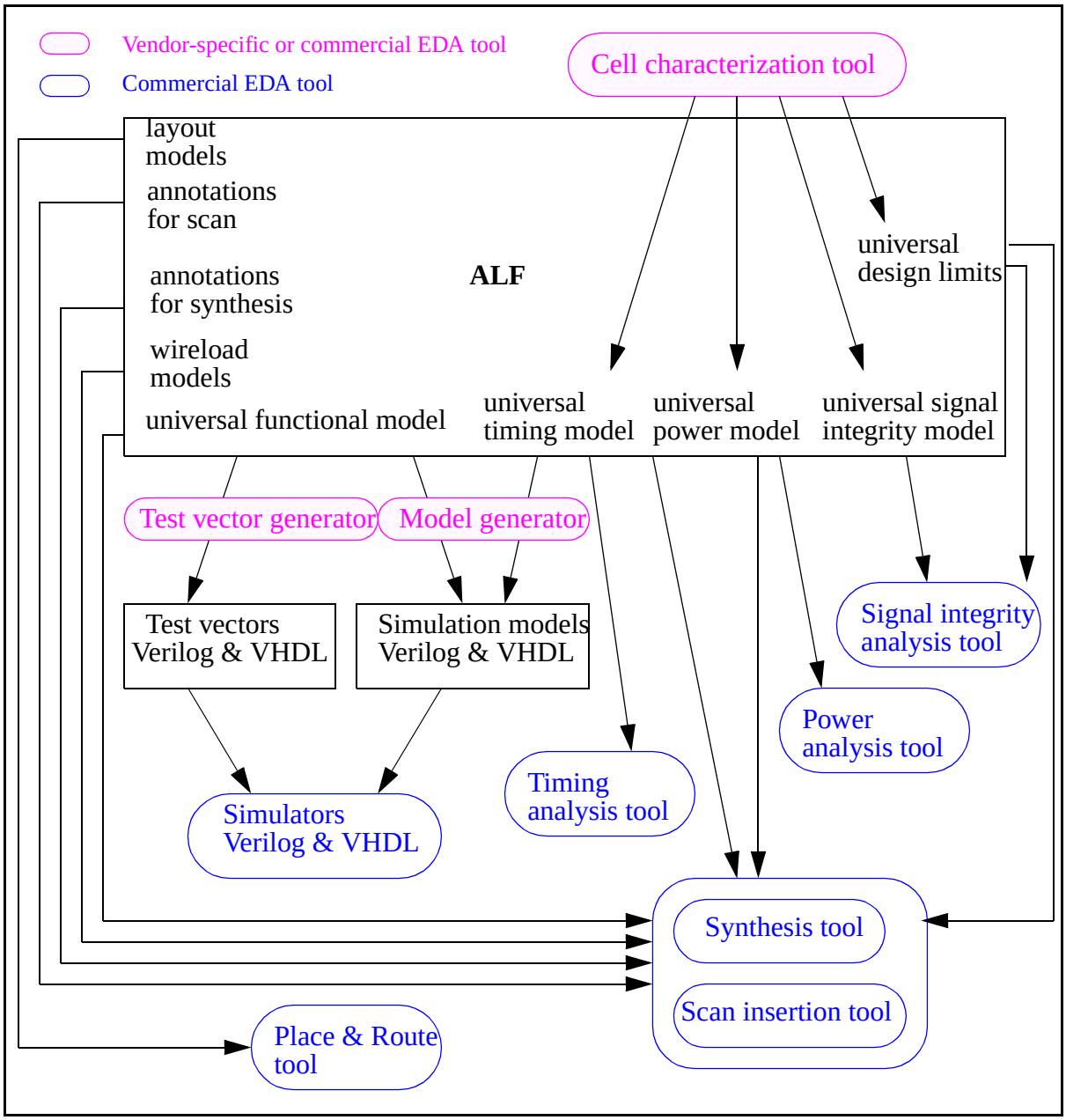

Figure 1—ALF and its target applications

The worldwide accepted standards for hardware description and simulation are VHDL and Verilog. Both languages have a wide scope of describing the design at various levels of abstraction: behavioral, functional, synthesizable RTL, and gate level. There are many ways to describe gate-level functions. The existing simulators are implemented in such a way that some constructs are more efficient for simulation run time than others. Also, how the simulation model handles timing constraints is a trade-off between efficiency and accuracy. Developing efficient simulation models which are functionally reliable (i.e., pessimistic for detecting timing constraint violation) is a major development effort for ASIC companies.

Hence, the use of a particular VHDL or Verilog simulation model as primary source of functional description of a cell is not very practical. Moreover, the existence of two simulation standards makes it difficult to pick one as a

reference with respect to the other. The purpose of a generic functional model is to serve as an absolute reference for all applications that require functional information. Applications such as synthesis, which need functional information merely for recognizing and choosing cell types, can use the generic functional model directly. For other applications, such as simulation and test, the generic functional model enables automated simulation model and test vector generation and verification, which has a tremendous benefit for the ASIC industry.

With progress of technology, the set of physical constraints under which the design functions have increased dramatically, along with the cost constraints. Therefore, the requirements for detailed characterization and analysis of those constraints, especially timing and power in deep submicron design, are now much more sophisticated. Only a subset of the increasing amount of characterization data appears in today's databooks.

ALF provides a generic format for all type of characterization data, without restriction to state-of-the art timing models. Power models are the most immediate extension and they have been the starter and primary driver for ALF.

Detailed timing and power characterization needs to take into account the *mode of operation* of the ASIC cell, which is related to the functionality. ALF introduces the concept of *vector-based modeling*, which is a generalization and a superset of today's timing and power modeling approaches. All existing timing and power analysis applications can retrieve the necessary model information from ALF.

1.4 Conventions

The syntax for description of lexical and syntax rules uses the following conventions.
Consider using the BNF nomenclature from IEEE 1481

```
::=     definition of a syntax rule
|       alternative definition
[item]  an optional item
[item1 | item2 | ... ]  optional item with alternatives
{item}  optional item that can be repeated
{item1 | item2 | ... }  optional items with alternatives
                        which can be repeated
item    item in boldface font is taken verbatim
item    item in italic is for explanation purpose only
```

The syntax for explanation of semantics of expressions uses the following conventions.

```
===     left side and right side expressions are equivalent
<item>  a placeholder for an item in regular syntax
```

1.5 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 ALF.
- Clause 3 (Definitions) defines terms used throughout the different specifications contained in this standard.
- Clause 4 (Acronyms and abbreviations) defines the acronyms used in this standard.
- Clause 6 (Lexical Rules) specifies the lexical rules.
- Clause 5 (Language construction principles) defines the language construction principles.
- Clause 7 (Auxiliary items) defines syntax and semantics of auxiliary items used in this standard.
- Clause 8 (Generic Objects) defines syntax and semantics of generic objects used in this standard.

— Clause 9 (Library-specific Objects) defines syntax and semantics of library-specific objects used in this standard.
— Clause 10 (Constructs for modeling of digital behavior) defines syntax and semantics of the control expression language used in this standard
— Clause 11 (Constructs for modeling of analog behavior) defines syntax and semantics of arithmetic models used in this standard.
— Clause 12 (Syntax Rule Summary) gives a summary of syntax rules used in this standard.
— Annexes. Following Clause 12 are a series of normative and informative annexes.

2. References

**Fill in applicable references, i.e. standards on which the herein proposed standard depends.

This standard shall be used in conjunction with the following publication. When the following standard is superseded by an approved revision, the revision shall apply.

**The following is only an example. ALF does not depend on C.

ISO/IEC 9899:1990, Programming Languages—C.[1]

[ISO 8859-1 : 1987(E)] ASCII character set

[1]ISO publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch/). IEC publications are available from the Sales Department of the International Electrotechnical Commission, Case Postale 131, 3, rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iec.ch/). ISO/IEC publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

3. Definitions

For the purposes of this standard, the following terms and definitions apply. The *IEEE Standard Dictionary of Electrical and Electronics Terms* [B4] should be consulted for terms not defined in this standard.

**Fill in definitions of terms which are used in the herein proposed standard.

3.1 **advanced library format:** The format of any file that can be parsed according to the syntax and semantics defined within this standard.

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

3.3 **arc**: *See:* **timing arc**.

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

3.5 **arithmetic model:** A representation of a library quantity that can be mathematically evaluated.

3.6 ...

3.7 **register transfer level:** A behavioral representation of a digital electronic design allowing inference of sequential and combinational logic components.

3.8 ...

3.9 **timing arc:** An abstract representation of a measurement between two points in time during operation of a library component.

3.10 ...

4. Acronyms and abbreviations

This clause lists the acronyms and abbreviations used in this standard.

ALF	advanced library format, title of the herein proposed standard
ASIC	application specific integrated circuit
AWE	asymptotic waveform evaluation
BIST	built-in self test
CAE	computer-aided engineering [the term electronic design automation (EDA) is preferred]
CAM	content-addressable memory
CLF	Common Library Format from Avant! Corporation
CPU	central processing unit
DCL	Delay Calculation Language from IEEE 1481 std
DEF	Design Exchange Format from Cadence Design Systems Inc.
DLL	delay-locked loop
DPCM	Delay and Power Calculation Module from IEEE 1481 std
DPCS	Delay and Power Calculation System from IEEE 1481 std
DSP	digital signal processor
EDA	electronic design automation
EDIF	Electronic Design Interchange Format
HDL	hardware description language
IC	integrated circuit
IP	intellectual property
ILM	Interface Logic Model from Synopsys Design Systems Inc.
LEF	Library Exchange Format from Cadence Design Systems Inc.
LIB	Library Format from Synopsys Inc.
LSSD	level-sensitive scan design
MPU	micro processor unit
OLA	Open Library Architecture from Silicon Integration Initiative Inc.
PDEF	Physical Design Exchange Format from IEEE 1481 std
PLL	Phase-locked loop
PVT	process/voltage/temperature
QTM	quick timing model
RAM	random access memory
RC	resistance times capacitance
RICE	rapid interconnect circuit evaluator
ROM	read-only memory
RSPF	Reduced Standard Parasitic Format
RTL	Register Transfer Level
SDF	Standard Delay Format from IEEE 1497 std
SLC	System Level Constraint format from Synopsys Inc.
SPEF	Standard Parasitic Exchange Format from IEEE 1481 std
SPF	Standard Parasitic Format
SPICE	Simulation Program with Integrated Circuit Emphasis
STA	Static Timing Analysis
STAMP	(STA Model Parameter ?) format from Synopsys Inc.
TCL	Tool Command Language (supported by multiple vendors)

TLF	Timing Library Format from Cadence Design Systems Inc.
VCD	Value Change Dump format (from IEEE 1364 std ?)
VHDL	VHSIC Hardware Description Language
VHSIC	very-high-speed integrated circuit
VITAL	VHDL Initiative Towards ASIC Libraries
VLSI	very-large-scale integration

5. Language construction principles

To fill in from ALF 2.0:

6. Lexical Rules

This section discusses the lexical rules.

6.1 Cross-reference of lexical tokens

Table needs update

Table 2 cross-references the lexical tokens used in ALF.

Table 2—Cross-reference of lexical tokens

Lexical token	Section	Lexical token	Section
alphabetic_bit_literal	6.3.4	integer	6.3.3
any_character	6.2.3	nonescaped_identifier	6.3.8
based_literal	6.3.5	non_negative_number	6.3.3
binary_base	6.3.5	nonreserved_character	6.2.3
binary_digit	6.3.5	number	6.3.3
bit_edge_literal	6.3.6	numeric_bit_literal	6.3.4
bit_literal	6.3.4	octal_base	6.3.5
block_comment	6.3.2	octal_digit	6.3.5
comment	6.3.2	placeholder_identifier	6.3.8
decimal_base	6.3.5	quoted_string	6.3.7
delimiter	6.3.1	reserved_character	6.2.3
digit	6.2.3	sign	6.3.3
dont_care_literal	6.3.4	single_line_comment	6.3.2
edge_literal	6.3.6	symbolic_edge_literal	6.3.6
escape_character	6.2.3	unsigned	6.3.3
escaped_identifier	6.3.8	whitespace	6.2.2
hex_base	6.3.5	word_edge_literal	6.3.6
hex_digit	6.3.5		

6.2 Characters

This section defines the use of characters in ALF.

6.2.1 Character set

Each graphic character corresponds to a unique code of the ISO eight-bit coded character set [ISO 8859-1 : 1987(E)] and is represented (visually) by a graphical symbol.

6.2.2 Whitespace characters

The characters shown in Table 3 shall be considered *whitespace characters*.

Table 3—List of whitespace characters

Character	ASCII code (hex)
space	20
vertical tab	0B
horizontal tab	09
line feed (new line)	0A
carriage return	0D
form feed	0C

Comments are also considered white space (see 6.3.2).

A whitespace character shall be ignored except when it separates other lexical tokens or when it appears in a quoted string.

6.2.3 Other characters

The ASCII character set shall be divided in four categories: reserved characters, non-reserved characters, escape character, and whitespace (see 6.2.2), as shown in Syntax 1.

```
any_character ::=
      reserved_character
    | nonreserved_character
    | escape_character
    | whitespace
```

Syntax 1—ASCII character

6.2.3.1 Reserved character

The reserved characters are symbols that make up punctuation marks and operators, as shown in Syntax 2.

```
reserved_character ::=
    & | | | ^ | ~ | + | - | * | / | % | ? | ! | = | < | > | : | ( | ) | [ | ] | { | } | @
  | ; | , | . | " | '
```

Syntax 2—Reserved character

6.2.3.2 Non-reserved character

The non-reserved characters shall be used for creating identifiers and numbers, as shown in Syntax 3 — Syntax 5.

```
nonreserved_character ::=
    letter | digit | _ | $ | #
```

Syntax 3—Non-reserved character

```
letter ::=
    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
  | 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
```

Syntax 4—Letter

```
digit ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Syntax 5—Digit

6.2.3.3 Escape character

The escape character ..., as shown in Syntax 6.

```
escape_character ::=
    \
```

Syntax 6—Escape character

ALF treats uppercase and lowercase characters as the same characters. In other words, ALF is a *case-insensitive language*.

Note — The characters $ and # can be reserved in other languages, such as VERILOG. Therefore, if translation from ALF into VERILOG is required, these characters shall not be used for items which need to be translated, e.g., the names of cells and pins. Other languages can be case-sensitive, such as VERILOG. Therefore, if translation from ALF into VERILOG is required, the case of the name used in the declaration of the object, e.g., the name of a cell or a pin, shall always be preserved as a reference. For example, if the name of a cell is declared as `MyCell`, reference to the cell can be made as `MYCELL` or `mycell`. However, it shall always be translated into VERILOG as `MyCell`.

6.3 Lexical tokens

The ALF source text files shall be a stream of lexical tokens. Each lexical token is either a *delimiter*, a *comment*, a *number*, a *bit literal*, a *based literal*, an *edge literal*, a *quoted string*, or an *identifier*.

6.3.1 Delimiter

A *delimiter* is either a reserved character or a compound operator. A compound operator is composed of two or three adjacent reserved characters, as shown in Syntax 7.

```
delimiter ::=
    reserved_character
  | && | ~& | || | ~| | ~^ | == | != | ** | >= | <=
  | ?! | ?~ | ?- | ?? | ?* | *? | -> | <-> | &> | <&> | >> | <<
```

Syntax 7—Delimiter

need to refer to operators rather than enumerating them here, otherwise we may miss some of them

Each special character in a single character delimiter list shall be a single delimiter, unless this character is used as a character in a compound operator or as a character in a quoted string.

6.3.2 Comment

ALF has two forms to introduce comments, as shown in Syntax 8.

```
comment ::=
      single_line_comment
    | block_comment
```

Syntax 8—Comment

A *single-line comment* shall start with the two characters `//` and end with a new line.

A *block comment* shall start with `/*` and end with `*/`. Comments shall not be nested. The single-line comment token `//` shall not have any special meaning in a block comment.

6.3.3 Number

make subsections for “unsigned” and “integer” and “real”

Constant *numbers* can be specified as integer or real, as shown in Syntax 9.

```
integer ::=
    [ sign ] unsigned
sign ::=
    + | -
unsigned ::=
    digit  { _ | digit }
non_negative_number ::=
      unsigned [ . unsigned ]
    | unsigned [ . unsigned ] E [ sign ] unsigned
number ::=
    [ sign ] non_negative_number
```

Syntax 9—Integer and real numbers

An *integer* is a decimal integer constant.

6.3.4 Bit literals

Bit literals can be specified as numeric or alphabetic bit, don’t care, or random, as shown in Syntax 10.

```
bit_literal ::=
      numeric_bit_literal
    | alphabetic_bit_literal
    | dont_care_literal
    | random_literal
numeric_bit_literal ::=
    0 | 1
alphabetic_bit_literal ::=
      X | Z | L | H | U | W
    | x | z | l | h | u | w
dont_care_literal ::=
    ?
random_literal ::=
    *
```

Syntax 10—Bit literal

A *bit* literal shall represent a single bit constant, as shown in Table 4.

Table 4—Single bit constants

Literal	Description
0	Value is logic zero.
1	Value is logic one.
X or x	Value is unknown.
L or l	Value is logic zero with weak drive strength.
H or h	Value is logic one with weak drive strength.
W or w	Value is unknown with weak drive strength.
Z or z	Value is high-impedance.
U or u	Value is uninitialized.
?	Value is any of the above, yet stable.
*	Value may randomly change.

6.3.5 Based literals

A *based literal* is a constant expressed in a form that specifies the base explicitly. The base can be specified in *binary, octal, decimal* or *hexadecimal* format, as shown in Syntax 11.

The underscore (_) shall be legal anywhere in the number, except as the first character and this character is ignored. This feature can be used to break up long numbers for readability purposes. No white space shall be allowed between base and digit token in a based literal.

When an alphabetic bit literal is used as an octal digit, it shall represent three repeated bits with the same literal. When an alphabetic bit literal is used as a hex digit, it shall represent four repeated bits with the same literal.

```
based_literal ::=
      binary_base { _ | binary_digit }
    | octal_base { _ | octal_digit }
    | decimal_base { _ | digit }
    | hex_base { _ | hex_digit }
binary_base ::=
    'B | 'b
binary_digit ::=
    bit_literal
octal_base ::=
    'O | 'o
octal_digit ::=
    binary_digit | 2 | 3 | 4 | 5 | 6 | 7
decimal_base ::=
    'D | 'd
hex_base ::=
    'H | 'h
hex_digit ::=
    octal_digit | 8 | 9 | A | B | C | D | E | F | a | b | c | d | e | f
```

Syntax 11—Based literal

Example

```
'o2xw0u  is the same as 'b010_xxx_www_000_uuu
'hLux    is the same as 'bLLLL_uuuu_xxxx
```

6.3.6 Edge literals

An *edge literal* shall be constructed by two bit literals or two based literals, as shown in Syntax 12. It shall describe the transition of a signal from one discrete value to another. No white space shall be allowed within (between) the two literals. An underscore can be used.

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

Syntax 12—Edge literal

6.3.7 Quoted strings

A *quoted string* shall be a sequence of zero or more characters enclosed between two quotation marks ("") and contained on a single line, as shown in Syntax 13.

```
quoted_string ::=
      " { any_character } "
```

Syntax 13—Quoted string

Character *escape codes* are used inside the string literal to represent some common special characters. The characters which can follow the backslash (\) and their meanings are listed in Table 5.

Table 5—Special characters in quoted strings

Symbol	ASCII Code (octal)	Meaning
\g	007	Alert/bell.
\h	010	Backspace.
\t	011	Horizontal tab.
\n	012	New line.
\v	013	Vertical tab.
\f	014	Form feed.
\r	015	Carriage return.
\"	042	Double quotation mark.
\\	134	Backslash.
\ddd		Octal value of ASCII character (three digits).

A non-quoted string can not contain any reserved character. Therefore, use of a quoted string is necessary when referencing file names (which typically contain a dot (.) character).

6.3.8 Identifier

Identifiers are used in ALF as names of objects, reserved words, and context-sensitive keywords. An identifier shall be any sequence of letters, digits, underscore (_), and dollar sign ($) character. Identifiers are treated in a case-insensitive way. They can be used in the definition of objects and in reference to already defined objects. A parser should preserve the case of an identifier in the definition of an object, since a downstream application could be case-sensitive.

Syntax:

```
identifiers ::= identifier { identifier }

identifier ::=
     nonescaped_identifier
   | escaped_identifier
   | placeholder_identifier
   | hierarchical_identifier
```

Purpose: Create a name for an object, create a predefined value for an object

6.3.8.1 Non-escaped identifier

If an identifier is constructed from one or more non-reserved characters, it is called an *non-escaped identifier*, as shown in Syntax 14.

```
nonescaped_identifier ::=
    nonreserved_character { nonreserved_character }
```

Syntax 14—Non-escaped identifier

A `digit` shall not be allowed as first character of a non-escaped identifier.

6.3.8.2 Escaped identifier

A sequence of characters starting with an `escape_character` is called an *escaped identifier*. The escaped identifier legalizes the use of a `digit` as first character of an identifier and the use of `reserved_character` anywhere in an identifier. Or it can be used to prevent the misinterpretation of an identifier as a keyword. The escape character shall be followed by at least one non-white space character to form an escaped identifier. The escaped identifier shall contain all characters up to first white space character, as shown in Syntax 15.

```
escaped_identifier ::=
    escape_character escaped_characters
escaped_characters ::=
    escaped_character { escaped_character }
escaped_character ::=
      nonreserved_character
    | reserved_character
    | escape_character
```

Syntax 15—Escaped identifier

6.3.8.3 Placeholder identifier

A *placeholder identifier* shall be a non-escaped identifier between the less-than character (<) and the greater-than character (>). No whitespace or delimiters are allowed between the non-escaped identifier and the placeholder characters (< and >). The placeholder identifier is used in template objects as a formal parameter, which is replaced by the actual parameter in template instantiation, as shown in Syntax 16.

```
placeholder_identifier ::=
    < nonescaped_identifier >
```

Syntax 16—Placeholder identifier

6.3.8.4 Hierarchical identifier

A *hierarchical identifier* shall be defined as shown in Syntax 17, with no whitespace in-between the characters.

```
hierarchical_identifier ::=
    identifier . { identifier . } identifier
```

Syntax 17—Hierarchical identifier

A dot (.) shall take precedence over an `escape_character`. To escape a dot, the `escape_character` shall be placed directly in front of it.

Examples

```
\id1.id2 //Only id1 is escaped.
```

```
id1\.id2 //Only the dot is escaped.
id1.\id2 //Only id2 is escaped.
```

6.4 Keywords

Keywords are case-insensitive non-escaped identifiers. For clarity, this document uses uppercase letters for keywords and lowercase letters elsewhere, unless otherwise mentioned.

Keywords are reserved for use as object identifiers, not for general symbols. To use an identifier that conflicts with the list of keywords, use the escape character, e.g., to declare a pin that is called `PIN`, use the form

```
PIN \PIN {..}
```

A keyword can either be a *reserved keyword* (also called a *hard keyword*) or a *context-sensitive keyword* (also called a *soft keyword*). The hard keywords have fixed meanings and shall be understood by any parser of ALF. The soft keywords might be understood only by specific applications. For example, a parser for a timing analysis application can ignore objects that contain power related information described using soft keywords.

6.4.1 Keywords for objects

table not up to date, maybe should be omitted

The keywords shown in Table 6 are used to identify object types.

Table 6—Object keywords

ALIAS	**ATTRIBUTE**	**BEHAVIOR**	**CELL**
CLASS	**CONSTANT**	**EQUATION**	**FUNCTION**
GROUP	**HEADER**	**INCLUDE**	**LIBRARY**
PIN	**PRIMITIVE**	**PROPERTY**	**STATETABLE**
SUBLIBRARY	**TABLE**	**TEMPLATE**	**VECTOR**
WIRE			

6.4.2 Keywords for operators

**table not up to date, refer to “arithmetic expression language” **

The keywords shown in Table 7 are used for built-in arithmetic functions.

Table 7—Built-in arithmetic function keywords

Term	Definition
ABS	Absolute value.
EXP	Natural exponential function.

Table 7—Built-in arithmetic function keywords (Continued)

Term	Definition
LOG	Natural logarithm.
MIN	Minimum.
MAX	Maximum.

6.4.3 Context-sensitive keywords

In order to address the need of extensible modeling, ALF provides a predefined set of *public* context-sensitive keywords. Additional private context-sensitive keywords can be introduced as long as they do not have the same name as any existing public keyword.

6.5 Rules against parser ambiguity

The following rules shall apply when resolving ambiguity in parsing ALF source

- In a context where both `bit_literal` and `identifier` are legal syntax items, a `nonescaped_identifier` shall take priority over an `alphabetic_bit_literal`.
- In a context where both `bit_literal` and `number` are legal syntax items, a `number` shall take priority over a `numeric_bit_literal`.
- In a context where both `edge_literal` and `identifier` are legal syntax items, an `identifier` shall take priority over a `bit_edge_literal`.
- In a context where both `edge_literal` and `number` are legal syntax items, a `number` shall take priority over a `bit_edge_literal`.

In such contexts, a `based_literal` shall be used instead of a `bit_literal`.

6.6 Values

Note: A lexical token is semantically interpreted as a value, once its lower-level lexical components (i.e., literals) have been identified.

6.6.1 Arithmetic value

Syntax:

```
arithmetic_values ::= arithmetic_value { arithmetic_value }

arithmetic_value ::=
      number
   |  identifier
   |  pin_value
```

Purpose: data for calculation described in arithmetic_model or in arithmetic_assignment for dynamic_template_instantiation

Semantic restriction: arithmetic_value must resolve to a valid value for the particular arithmetic_model, where it is used. Some arithmetic_models allow only unsiged (e.g. SWITCHING_BITS, FANOUT), others allow only

non_negative_number (e.g. WIDTH, LENGTH). Non-interpolatable arithmetic_models (e.g. PROCESS, DERATE_CASE) allow only symbolic identifiers rather than numbers.

6.6.2 String value

Syntax:

```
string_value ::=
     quoted_string
   | identifier
```

Purpose: textual data

6.6.3 Edge value

Syntax:

```
edge_values ::=
   edge_value { edge_value }

edge_value ::= ( edge_literal )
```

Purpose: use edge_literal as a standalone value. For that purpose the edge_literal is enclosed by parentheses, to avoid parser ambiguity. Normally, an edge_literal would appear only within a vector_expression. In that context, the enclosing parentheses are not necessary.

6.6.4 Index value

Syntax:

```
index_value ::=
     unsigned
   | identifier
```

Index_value must resolve to unsigned, i.e., identifier must be the name of a CONSTANT with unsigned value or a placeholder in TEMPLATE that gets replaced with unsigned.

7. Auxiliary items

7.1 Index and related items

7.1.1 Index

Syntax:

```
index ::=
     [ index_range ]
   | [ index_value ]
```

7.1.2 Index range

Syntax:

```
index_range ::= index_value : index_value
```

Index_range shall define consecutive unsigned numbers, bound by index_value left and right of ":". In the context of a PIN statement, left index_value shall be considered as MSB, right index_value shall be considered as MSB. Also used in RANGE statement and in GROUP statement.

7.2 Pin assignment and related items

7.2.1 Pin assignment

Syntax:

```
pin_assignments ::= pin_assignment { pin_assignment }

pin_assignment ::=
     pin_variable = pin_value ;
```

Purpose: Associates a pin_value with a pin_variable for the purpose of pin mapping. Used in NON_SCAN_CELL statement, in ARTWORK statement, and in STRUCTURE statement.

Semantic restrictions:

The pin_value must be compatible with the pin_variable. A scalar pin_variable can be assigned to another scalar pin_variable, or to a scalar pin_value, i.e. bit_literal or one-bit binary based_literal. A one-dimensional pin_variable or a one-dimensional slice of a two-dimensional pin_variable can be assigned to another one-dimensional pin_variable, or to another one-dimensional slice of a two-dimensional pin_variable of same bit-width, or to a based_literal of the same bitwidth, or to a unsigned that can be converted into a binary number of the same bitwidth.

If the bitwidth of the pin_value is smaller than the bitwidth of the pin_variable, the LSBs shall be aligned. Excessive leading bits of the pin_variable shall be filled with zeros.

To be discussed: If the bitwidth of the pin_value is greater than the bitwidth of the pin_variable, the LSBs shall be aligned. Excessive leading bits of the pin_value shall be cut off.

7.2.2 Pin variable

Syntax:

```
pin_variables ::= pin_variable { pin_variable }

pin_variable ::= pin_variable_identifier [ index ]
```

Purpose: A pin_variable shall be used to represent the information accessible through a PIN. A PIN (see section xxx) is the interface between a library component (i.e., a CELL or a PRIMITIVE) and its environment.

Semantics: A legal pin_variable_identifier shall make reference to a previously declared PIN, or to a previously declared PIN_GROUP, or to a previously declared NODE, or to a previously declared PORT (i.e., pin_identifier.port_identifier). A legal index shall be bound by the MSB and by the LSB of the index_range in the refered PIN.

7.2.3 Pin value

Syntax:

```
pin_values ::= pin_value { pin_value }

pin_value ::=
     pin_variable
   | bit_literal
   | based_literal
   | unsigned
```

Purpose: pin_value defines the set of values that can be assigned to a pin_variable. Assigning a pin_variable to another pin_variable shall be legal.

Can be used context of NON_SCAN_CELL, STRUCTURE, primitive_instantiation as a short form of pin_assignment, i.e., pin mapping by order instead of pin mapping by name.

7.3 Annotation and related items

7.3.1 Annotation

Note: An annotation is an auxiliary statement within the context of a library_specific_object, a library_specific_singular_object, or an arithmetic_model. It serves as a qualifier of its context.

```
annotation ::=
     one_level_annotation
```

```
   | two_level_annotation
   | multi_level_annotation

one_level_annotations ::=
   one_level_annotation { one_level_annotation }

one_level_annotation ::=
     single_value_annotation
   | multi_value_annotation

single_value_annotation ::=
   identifier = annotation_value ;

multi_value_annotation ::=
   identifier { annotation_values }

two_level_annotations ::=
   two_level_annotation { two_level_annotation }

two_level_annotation ::=
     one_level_annotation
   | identifier [ = annotation_value ]
      { one_level_annotations }

multi_level_annotations ::=
   multi_level_annotation { multi_level_annotation }

multi_level_annotation ::=
     one_level_annotation
   | identifier [ = annotation_value ]
      { multi_level_annotations }
```

7.3.2 Annotation value

Syntax:

```
annotation_values ::= annotation_value { annotation_value }

annotation_value ::=
     index_value
   | string_value
   | edge_value
   | pin_value
   | arithmetic_value
   | boolean_expression
   | control_expression
```

Note: there is lexical overlap, but semantic distinction between the possible annotation values.

7.4 All purpose item

Syntax:

```
all_purpose_items ::=
   all_purpose_item { all_purpose_item }

all_purpose_item ::=
     include
   | alias
   | constant
   | attribute
   | property
   | class_declaration
   | keyword_declaration
   | group_declaration
   | template_declaration
   | template_instantiation
   | annotation
   | arithmetic_model
   | arithmetic_model_container
```

Purpose: Provide flexibility and generality of the ALF syntax. The ALF semantics shall define whether a particular all_purpose_item is legal within a specific context.

8. Generic Objects

To fill in from ALF 2.0:

8.1 INCLUDE statement

Syntax sniplet:

```
include ::=
   INCLUDE quoted_string ;
```

To fill in from ALF 2.0:

8.2 ALIAS statement

Syntax sniplet:

```
alias ::=
   ALIAS identifier = identifier ;
```

To fill in from ALF 2.0:

8.3 CONSTANT statement

Syntax sniplet:

```
constant ::=
     CONSTANT identifier = arithmetic_value ;
```

To fill in from ALF 2.0:

8.4 ATTRIBUTE statement

Syntax sniplet:

```
attribute ::=
   ATTRIBUTE { identifiers }
```

To fill in from ALF 2.0:

8.5 PROPERTY statement

Syntax sniplet:

```
property ::=
   PROPERTY [ identifier ] { one_level_annotations }
```

To fill in from ALF 2.0:

8.6 CLASS statement

Syntax sniplet:

```
class_declaration ::=
     CLASS identifier  ;
   | CLASS identifier  { all_purpose_items }
```

To fill in from ALF 2.0:

8.7 KEYWORD statement

Syntax sniplet:

```
keyword_declaration ::=
   KEYWORD context_sensitive_keyword =
      syntax_item_identifier ;
```

To fill in from ALF 2.0.2:

8.8 GROUP statement

Syntax sniplet:

```
group_declaration ::=
     GROUP group_identifier { annotation_values }
   | GROUP group_identifier { index_value : index_value }
```

To fill in from ALF 2.0:

Semantics: When the group identifier is used within an ALF statement within the scope of the GROUP declaration, that ALF statement shall be replaced by several statements, substituting the annotation values or the index values, respectively, for the group identifier. The replacing statements shall appear at the same scope as the GROUP declaration.

8.9 TEMPLATE statement

Syntax sniplet:

```
template_declaration ::=
   TEMPLATE template_identifier { template_items }

template_items ::=
   template_item { template_item }

template_item ::=
     all_purpose_item
   | cell
   | library
   | node
   | pin
   | pin_group
   | primitive
   | sublibrary
   | vector
   | wire
   | antenna
   | array
   | blockage
   | layer
   | pattern
   | port
   | rule
   | site
   | via
   | function
   | non_scan_cell
   | test
   | range
   | artwork
   | from
   | to
   | illegal
   | violation
   | header
   | table
   | equation
   | arithmetic_submodel
```

```
   | behavior_item
   | geometric_model

template_instantiation ::=
      static_template_instantiation
   |  dynamic_template_instantiation

static_template_instantiation ::=
     template_identifier [ = static ] ;
   | template_identifier [ = static ] { annotation_values }
   | template_identifier [ = static ]{ one_level_annotations }

dynamic_template_instantiation ::=
     template_identifier = dynamic
      { dynamic_template_instantiation_items }

dynamic_template_instantiation_items ::=
   dynamic_template_instantiation_item
      { dynamic_template_instantiation_item }

dynamic_template_instantiation_item ::=
      one_level_annotation
   |  arithmetic_model
```

To fill in from ALF 2.0:

9. Library-specific Objects

To fill in from ALF 2.0:

9.1 LIBRARY statement and related statements

9.1.1 LIBRARY statement

Syntax sniplet:

```
library ::=
     LIBRARY library_identifier { library_items }
   | LIBRARY library_identifier ;
   | library_template_instantiation

library_items ::=
   library_item { library_item }

library_item ::=
     sublibrary
   | sublibrary_item
```

9.1.2 SUBLIBRARY statement

Syntax sniplet:

```
library ::=
     SUBLIBRARY sublibrary_identifier { sublibrary_items }
   | SUBLIBRARY sublibrary_identifier ;
   | sublibrary_template_instantiation

sublibrary_items ::=
   sublibrary_item { sublibrary_item }

sublibrary_item ::=
     all_purpose_item
   | cell
```

```
   | primitive
   | wire
   | layer
   | via
   | rule
   | antenna
   | array
   | site
```

9.1.3 INFORMATION statement

Syntax sniplet:

```
INFORMATION_two_level_annotation ::=
   INFORMATION { information_one_level_annotations }

information_one_level_annotations ::=
   information_one_level_annotation
      { information_one_level_annotation }

information_one_level_annotation ::=
     AUTHOR_one_level_annotation
   | VERSION_one_level_annotation
   | DATETIME_one_level_annotation
   | PROJECT_one_level_annotation
```

INFORMATION shall be used within LIBRARY, SUBLIBRARY, CELL, WIRE, PRIMITIVE, since these objects can be considered as standalone deliverables. Other objects, for example PIN, PORT, LAYER, VIA etc., cannot be considered as standalone deliverables.

To fill in from ALF 2.0:

9.2 CELL statement and related statements

9.2.1 CELL statement

Syntax sniplet:

```
cell ::=
     CELL cell_identifier { cell_items }
```

```
   | CELL cell_identifier ;
   | cell_template_instantiation

cell_items ::=
   cell_item { cell_item }

cell_item ::=
     all_purpose_item
   | pin
   | pin_group
   | primitive
   | function
   | non_scan_cell
   | test
   | vector
   | wire
   | blockage
   | artwork
```

9.2.2 NON_SCAN_CELL statement

Syntax sniplet:

```
non_scan_cell ::=
     NON_SCAN_CELL { unnamed_cell_instantiations }
   | NON_SCAN_CELL = unnamed_cell_instantiation
   | non_scan_cell_template_instantiation

unnamed_cell_instantiations ::=
   unnamed_cell_instantiation { unnamed_cell_instantiation }

unnamed_cell_instantiation ::=
     cell_identifier { pin_values }
   | cell_identifier { pin_assignments }
```

To fill in from ALF 2.0:

9.2.3 Annotations and attributes for a CELL

Syntax sniplet:

To fill in from ALF 2.0:

9.3 PIN statement and related statements

9.3.1 PIN statement

Syntax sniplet:

```
pin ::=
     PIN [ [ index_range ] ] pin_identifier [ [ index_range ] ]
      { pin_items }
   | PIN [ [ index_range ] ] pin_identifier [ [ index_range ] ] ;
   | pin_template_instantiation

pin_item ::=
     all_purpose_item
   | range
```

```
   | port
   | pin_instantiation

pin_items ::=
   pin_item { pin_item }

pin_instantiation ::=
   pin_variable { pin_items }
```

To fill in from ALF 2.0:

9.3.2 RANGE statement

Syntax sniplet:

```
range ::=
   RANGE { index_range }
```

To fill in from ALF 2.0:

9.3.3 PIN_GROUP statement

Syntax sniplet:

```
pin_group ::=
   PIN_GROUP [ [ index_range ] ] pin_group_identifier
```

```
      { pin_group_items }
   | pin_group_template_instantiation

pin_group_items ::=
   pin_group_item { pin_group_item }

pin_group_item ::=
     all_purpose_item
   | range
```

To fill in from ALF 2.0:

9.3.4 Annotations and attributes for a PIN

To fill in from ALF 2.0:

9.4 WIRE statement and related statements

9.4.1 WIRE statement

Syntax sniplet:

```
wire ::=
     WIRE wire_identifier { wire_items }
   | WIRE wire_identifier ;
   | wire_template_instantiation

wire_items ::=
   wire_item { wire_item }

wire_item ::=
     all_purpose_item
   | node
```

To fill in from ALF 2.0:

9.4.2 NODE statement

Syntax sniplet:

```
node ::=
     NODE node_identifier { node_items }
   | NODE node_identifier ;
   | node_template_instantiation

node_items ::=
   node_item { node_item }

node_item ::=
     all_purpose_item
```

To fill in from ALF 2.0:

9.5 VECTOR statement and related statements

9.5.1 VECTOR statement

Syntax sniplet:

```
vector ::=
     VECTOR control_expression { vector_items }
   | VECTOR control_expression ;
   | vector_template_instantiation

vector_items ::=
   vector_item { vector_item }

vector_item ::=
     all_purpose_item
   | illegal
```

9.5.2 ILLEGAL statement

Syntax sniplet:

```
illegal ::=
     ILLEGAL { illegal_items }
   | illegal_template_instantiation

illegal_items ::=
   illegal_item { illegal_item }

illegal_item ::=
     all_purpose_item
   | violation
```

To fill in from ALF 2.0:

9.5.3 Annotations and attributes for a VECTOR

To fill in from ALF 2.0:

9.6 LAYER statement and related statements

To fill in from ALF 2.0:

9.6.1 LAYER statement

Syntax sniplet:

```
layer ::=
     LAYER layer_identifier { layer_items }
   | LAYER layer_identifier ;
   | layer_template_instantiation

layer_items ::=
   layer_item { layer_item }

layer_item ::=
     all_purpose_item
```

9.6.2 Annotations for a LAYER

To fill in from ALF 2.0:

9.7 VIA statement and related statements

9.7.1 VIA statement

Syntax sniplet:

```
via ::=
     VIA via_identifier { via_items }
   | VIA via_identifier ;
   | via_template_instantiation

via_items ::=
   via_item { via_item }

via_item ::=
     all_purpose_item
   | pattern
   | artwork
```

9.7.2 Annotations for a VIA

To fill in from ALF 2.0:

9.7.3 VIA reference statement

```
via_reference ::=
     VIA { via_instantiations }
   | VIA { via_identifiers }

via_instantiations ::=
   via_instantiation { via_instantiation }

via_instantiation ::=
   via_identifier { geometric_transformations }
```

To fill in from ALF 2.0:

9.8 Statements related to physical design rules

9.8.1 RULE statement

Syntax sniplet:

```
rule ::=
     RULE rule_identifier { rule_items }
   | RULE rule_identifier ;
   | rule_template_instantiation

rule_items ::=
   rule_item { rule_item }

rule_item ::=
     all_purpose_item
   | pattern
   | via_reference
```

To fill in from ALF 2.0:

9.8.2 ANTENNA statement

Syntax sniplet:

```
antenna ::=
     ANTENNA antenna_identifier { antenna_items }
   | ANTENNA antenna_identifier ;
   | antenna_template_instantiation

antenna_items ::=
   antenna_item { antenna_item }

antenna_item ::=
     all_purpose_item
```

To fill in from ALF 2.0:

9.8.3 BLOCKAGE statement

Syntax sniplet:

```
blockage ::=
     BLOCKAGE blockage_identifier { blockage_items }
   | BLOCKAGE blockage_identifier ;
   | blockage_template_instantiation

blockage_items ::=
   blockage_item { blockage_item }

blockage_item ::=
     all_purpose_item
   | pattern
```

```
   | rule
   | via_reference
```

To fill in from ALF 2.0:

9.8.4 PORT statement

Syntax sniplet:

```
port ::=
     PORT port_identifier { port_items }
   | PORT port_identifier ;
   | port_template_instantiation

port_items ::=
   port_item { port_item }

port_item ::=
     all_purpose_item
   | pattern
   | rule
   | via_reference
```

To fill in from ALF 2.0:

9.9 Statements related to physical geometry

9.9.1 SITE statement

Syntax sniplet:

```
site ::=
     SITE site_identifier { site_items }
   | SITE site_identifier ;
   | site_template_instantiation

site_items ::=
   site_item { site_item }

site_item ::=
     all_purpose_item
   | ORIENTATION_CLASS_one_level_annotation
   | SYMMETRY_CLASS_one_level_annotation
```

To fill in from ALF 2.0.2:

9.9.2 ARRAY statement

Syntax sniplet:

```
array ::=
     ARRAY array_identifier { array_items }
   | ARRAY array_identifier ;
   | array_template_instantiation

array_items ::=
   array_item { array_item }

array_item ::=
     all_purpose_item
   | PURPOSE_single_value_annotation
   | geometric_transformation
```

To fill in from ALF 2.0:

9.9.3 PATTERN statement

Syntax sniplet:

```
pattern ::=
     PATTERN pattern_identifier { pattern_items }
   | PATTERN pattern_identifier ;
   | pattern_template_instantiation

pattern_items ::=
   pattern_item { pattern_item }

pattern_item ::=
     all_purpose_item
   | SHAPE_single_value_annotation
   | LAYER_single_value_annotation
   | EXTENSION_single_value_annotation
   | VERTEX_single_value_annotation
   | geometric_model
   | geometric_transformation
```

To fill in from ALF 2.0:

9.9.4 ARTWORK statement

Syntax sniplet:

```
artwork ::=
     ARTWORK = artwork_identifier { artwork_items }
```

```
   | ARTWORK = artwork_identifier ;
   | artwork_template_instantiation

artwork_items ::=
   artwork_item { artwork_item }

artwork_item ::=
     geometric_transformation
   | pin_assignment
```

To fill in from ALF 2.0:

9.9.5 Geometric model

Syntax sniplet:

```
geometric_model ::=
     nonescaped_dentifier [ geometric_model_identifier ]
         { geometric_model_items }
   | geometric_model_template_instantiation

geometric_model_items ::=
   geometric_model_item { geometric_model_item }

geometric_model_item ::=
     all_purpose_item
   | POINT_TO_POINT_one_level_annotation
   | coordinates

coordinates ::=
   COORDINATES { x_number y_number { x_number y_number } }
```

To fill in from ALF 2.0:

9.9.6 Geometric transformation

```
geometric_transformations ::=
   geometric_transformation { geometric_transformation }

geometric_transformation ::=
     SHIFT_two_level_annotation
   | ROTATE_one_level_annotation
   | FLIP_one_level_annotation
   | repeat

repeat ::=
   REPEAT [ = unsigned ] {
      shift_two_level_annotation
      [ repeat ]
   }
```

To fill in from ALF 2.0.2:

9.10 Statements related to functional description

To fill in from ALF 2.0.2:

9.10.1 FUNCTION statement

Syntax sniplet:

```
function ::=
     FUNCTION { function_items }
   | function_template_instantiation

function_items ::=
   function_item { function_item }

function_item ::=
   all_purpose_item
```

```
   | behavior
   | structure
   | statetable
```

9.10.2 TEST statement

Syntax sniplet:

```
test ::=
     TEST { test_items }
   | test_template_instantiation

test_items ::=
   test_item { test_item }

test_item ::=
     all_purpose_item
   | behavior
   | statetable
```

To fill in from ALF 2.0:

9.10.3 BEHAVIOR statement

Syntax sniplet:

```
behavior ::=
     BEHAVIOR { behavior_items }
   | behavior_template_instantiation

behavior_items ::=
   behavior_item { behavior_item }

behavior_item ::=
     boolean_assignment
   | control_statement
```

```
   | primitive_instantiation
   | behavior_item_template_instantiation

boolean_assignments ::=
   boolean_assignment  { boolean_assignment }

boolean_assignment ::=
   pin_variable = boolean_expression ;

primitive_instantiation ::=
     primitive_identifier [ identifier ] { pin_values }
   | primitive_identifier [ identifier ]
      { boolean_assignments }

control_statement ::=
   @ control_expression { boolean_assignments }
      { : control_expression { boolean_assignments } }
```

To fill in from ALF 2.0:

9.10.4 STRUCTURE statement

Syntax sniplet:

```
structure ::=
     STRUCTURE { named_cell_instantiations }
   | structure_template_instantiation

named_cell_instantiations ::=
   named_cell_instantiation { named_cell_instantiation }

named_cell_instantiation ::=
     cell_identifier instance_identifier { pin_values }
   | cell_identifier instance_identifier { pin_assignments }
```

To fill in from ALF 2.0:

9.10.5 VIOLATION statement

Syntax sniplet:

```
violation ::=
     VIOLATION { violation_items }
   | violation_template_instantiation

violation_items ::=
   violation_item { violation_item }

violation_item ::=
     MESSAGE_TYPE_single_value_annotation
   | MESSAGE_single_value_annotation
   | behavior
```

To fill in from ALF 2.0.2:

9.10.6 STATETABLE statement

Syntax sniplet:

```
statetable ::=
     STATETABLE [ identifier ]
      { statetable_header statetable_row { statetable_row } }
   | statetable_template_instantiation

statetable_header ::=
   input_pin_variables : output_pin_variables ;

statetable_row ::=
   statetable_control_values : statetable_data_values ;

statetable_control_values ::=
     statetable_control_value { statetable_control_value }

statetable_control_value ::=
     bit_literal
   | based_literal
```

```
   | unsigned
   | edge_value

statetable_data_values ::=
     statetable_data_value { statetable_data_value }

statetable_data_value ::=
     bit_literal
   | based_literal
   | unsigned
   | ( [ ! ] pin_variable )
   | ( [ ~ ] pin_variable )
```

To fill in from ALF 2.0:

9.10.7 PRIMITIVE statement

Syntax sniplet:

```
primitive ::=
     PRIMITIVE primitive_identifier { primitive_items }
   | PRIMITIVE primitive_identifier ;
   | primitive_template_instantiation

primitive_items ::=
   primitive_item { primitive_item }

primitive_item ::=
     all_purpose_item
   | pin
   | pin_group
   | function
   | test
```

To fill in from ALF 2.0:

10. Constructs for modeling of digital behavior

To fill in from ALF 2.0:

10.1 Boolean expression language

```
boolean_expression ::=
     ( boolean_expression )
   | pin_value
   | boolean_unary boolean_expression
   | boolean_expression boolean_binary boolean_expression
   | boolean_expression ? boolean_expression :
      { boolean_expression ? boolean_expression : }
      boolean_expression

boolean_unary ::=
     !
   | ~
   | &
   | ~&
   | |
   | ~|
   | ^
   | ~^

boolean_binary ::=
     &
   | &&
   | |
   | ||
   | ^
   | ~^
   | !=
   | ==
   | >=
   | <=
   | >
   | <
   | +
   | -
   | *
   | /
   | %
   | >>
   | <<
```

10.2 Vector expression language

```
vector_expression ::=
     ( vector_expression )
```

```
   | vector_unary boolean_expression
   | vector_expression vector_binary vector_expression
   | boolean_expression ? vector_expression :
      { boolean_expression ? vector_expression : }
      vector_expression
   | boolean_expression control_and vector_expression
   | vector_expression control_and boolean_expression

vector_unary ::=
   edge_literal

vector_binary ::=
     &
   | &&
   | |
   | ||
   | ->
   | ~>
   | <->
   | <~>
   | &>
   | <&>

control_and ::=
   & | &&

control_expression ::=
     ( vector_expression )
   | ( boolean_expression )
```

10.3 Control expression semantics

11. Constructs for modeling of analog behavior

11.1 Arithmetic expression language

Syntax sniplet:

```
arithmetic_expression ::=
     ( arithmetic_expression )
   | arithmetic_value
   | [ arithmetic_unary ] arithmetic_expression
   | arithmetic_expression arithmetic_binary
      arithmetic_expression
   | boolean_expression ? arithmetic_expression :
      { boolean_expression ? arithmetic_expression : }
      arithmetic_expression
   | arithmetic_macro
      ( arithmetic_expression { , arithmetic_expression } )


arithmetic_unary ::= sign


arithmetic_binary ::=
     +
   | -
   | *
   | /
   | **
   | %


arithmetic_macro ::=
     abs
   | exp
   | log
   | min
   | max
```

To fill in from ALF 2.0:

11.2 Arithmetic model and related statements

To fill in from ALF 2.0:

11.2.1 Arithmetic model statement

Syntax sniplet:

```
arithmetic_models ::=
   arithmetic_model { arithmetic_model }

arithmetic_model ::=
     partial_arithmetic_model
   | non_trivial_arithmetic_model
   | trivial_arithmetic_model
   | assignment_arithmetic_model
   | arithmetic_model_template_instantiation
```

11.2.2 Partial arithmetic model

```
partial_arithmetic_model ::=
   nonescaped_identifier [ arithmetic_model_identifier ]
      { partial_arithmetic_model_items }

partial_arithmetic_model_items ::=
   partial_arithmetic_model_item
      { partial_arithmetic_model_item }

partial_arithmetic_model_item ::=
     any_arithmetic_model_item
   | table
```

Partial arithmetic model contains only definitions relevant for the model, but not sufficient data to evaluate the model.

Definitions within unnamed partial arithmetic model (i.e. partial arithmetic model without arithmetic model identifier) shall be inherited by all arithmetic models of the same type (i.e., using the same nonescaped identifier) within scope. However, these definitions can be locally overwritten.

Named partial arithmetic model (i.e. partial arithmetic model without arithmetic model identifier) can be used as argument of an EQUATION within another arithmetic model within scope, without appearing in the HEADER.

If a partial arithmetic model outside a HEADER contains a TABLE, the arithmetic values in the TABLE shall define a discrete set of valid values for the model.

If a partial arithmetic model within a HEADER contains a TABLE, the arithmetic values in the TABLE shall define the entries for table-lookup.

11.2.3 Non-trivial arithmetic model

```
non_trivial_arithmetic_model ::=
   nonescaped_identifier [ arithmetic_model_identifier ] {
      [ any_arithmetic_model_items ]
      arithmetic_body
      [ any_arithmetic_model_items ]
   }
```

Non-trivial arithmetic model contains sufficient data to evaluate the model.

11.2.4 Trivial arithmetic model

```
trivial_arithmetic_model ::=
     nonescaped_identifier [ arithmetic_model_identifier ]
      = arithmetic_value ;
   | nonescaped_identifier [ arithmetic_model_identifier ]
      = arithmetic_value { any_arithmetic_model_items }
```

Trivial arithmetic model is associated with a constant arithmetic value. Therefore the evaluation of the arithmetic model is trivial.

11.2.5 Assignment arithmetic model

```
assignment_arithmetic_model ::=
     arithmetic_model_identifier = arithmetic_expression ;
```

This form of arithmetic model is valid only in the following cases:

A partial arithmetic model has been defined using the arithmetic model identifier
AND
arithmetic models for all arguments contained in the arithmetic expression have been defined.

OR

Thisconstruct is used in a dynamic template instantiation.

11.2.6 Items for any arithmetic model

```
any_arithmetic_model_items ::=
   any_arithmetic_model_item { any_arithmetic_model_item }

any_arithmetic_model_item ::=
     all_purpose_item
   | from
   | to
   | violation
```

Semantic restrictions apply, dependent on type and context of the arithmetic model.

11.3 Arithmetic submodel and related statements

11.3.1 Arithmetic submodel statement

```
arithmetic_submodels ::=
   arithmetic_submodel { arithmetic_submodel }

arithmetic_submodel ::=
     non_trivial_arithmetic_submodel
   | trivial_arithmetic_submodel
   | arithmetic_submodel_template_instantiation
```

11.3.2 Non-trivial arithmetic submodel

```
non_trivial_arithmetic_submodel ::=
   nonescaped_identifier {
      [ any_arithmetic_submodel_items ]
      arithmetic_body
      [ any_arithmetic_submodel_items ]
   }
```

Non-trivial arithmetic submodel contains sufficient data to evaluate the arithmetic submodel.

11.3.3 Trivial arithmetic submodel

```
trivial_arithmetic_submodel ::=
   nonescaped_identifier = arithmetic_value ;
   | nonescaped_identifier
      = arithmetic_value { any_arithmetic_submodel_items }
```

Trivial arithmetic submodel is associated with a constant arithmetic value. Therefore the evaluation of the arithmetic submodel is trivial.

11.3.4 Items for any arithmetic submodel

```
any_arithmetic_submodel_items ::=
   any_arithmetic_submodel_item
      { any_arithmetic_submodel_item }

any_arithmetic_submodel_item ::=
     all_purpose_item
   | violation
```

Semantic restrictions apply, dependent on type and context of the arithmetic submodel.

11.4 Arithmetic body and related statements

11.4.1 Arithmetic body

```
arithmetic_body ::=
     arithmetic_submodels
```

```
   | table_arithmetic_body
   | equation_arithmetic_body

table_arithmetic_body ::=
     header table [ equation ]

equation_arithmetic_body ::=
     [ header ] equation [ table ]
```

Arithmetic model body shall supply the data necessary for evaluation of the arithmetic model.

11.4.2 HEADER statement

```
header ::=
     HEADER { identifiers }
   | HEADER { header_arithmetic_models }
   | header_template_instantiation

header_arithmetic_models ::=
   header_arithmetic_model { header_arithmetic_model }

header_arithmetic_model ::=
     non_trivial_arithmetic_model
   | partial_arithmetic_model
```

HEADER shall contain arguments for evaluation of arithmetic model. The arithmetic values of those arguments must be supplied by application program.

Semantic restriction: No arithmetic submodel within arithmetic model body.

11.4.3 TABLE statement

```
table ::=
     TABLE { arithmetic_values }
   | table_template_instantiation
```

TABLE shall provide means for evaluation using a look-up method. All arithmetic_values within TABLE must all be of the same type and compatible with the type of the arithmetic model under evaluation.

11.4.4 EQUATION statement

```
equation ::=
     EQUATION { arithmetic_expression }
   | equation_template_instantiation
```

EQUATION shall provide means for evaluation using an analytical method.

11.5 Arithmetic model container

To fill in from ALF 2.0:


```
arithmetic_model_container ::=
   arithmetic_model_container_identifier
      { arithmetic_models }
```

11.6 Statements related to arithmetic models for general purpose

11.6.1 MIN and MAX statement

To fill in from ALF 2.0:

11.6.2 TYP statement

11.6.3 DEFAULT statement

To fill in from ALF 2.0:

11.6.4 LIMIT statement

To fill in from ALF 2.0:

11.6.5 Annotations for arithmetic models for general purpose

To fill in from ALF 2.0:

11.7 Rules for evaluation of arithmetic models

11.7.1 Arithmetic model with arithmetic submodels

The application program shall decide which arithmetic submodel applies for evaluation in a particular situation. Per default, the arithmetic submodel identified by the DEFAULT keyword or the arithmetic submodel referenced by the DEFAULT annotation shall apply.

11.7.2 Arithmetic model with table arithmetic body

All arithmetic models in HEADER must contain TABLE.

Describe algorithm to identify correct table entry.

Refer to INTERPOLATION annotation.

Supplementary EQUATION is legal, shall be used for interpolation or extrapolation values outside range.

11.7.3 Arithmetic model with equation arithmetic body

Operands in arithmetic expression shall be defined as arithmetic models in HEADER or as partial arithmetic models outside HEADER, but within scope. It shall be legal to some arguments defined in the HEADER and some others outside HEADER.

For named arithmetic model, the name shall be used as operand. For unnamed arithmetic model, the keyword shall be used as operand.

Supplementary TABLE is legal, shall be used as lookup entry for downstream arithmetic models, when the arithmetic model itself is within HEADER.

11.8 Overview of arithmetic models

To fill in from ALF 2.0:

11.9 Arithmetic models for timing data

To fill in from ALF 2.0:

11.9.1 TIME statement

To fill in from ALF 2.0:

11.9.2 FREQUENCY statement

To fill in from ALF 2.0:

11.9.3 DELAY and RETAIN statement

To fill in from ALF 2.0:

11.9.4 SLEWRATE statement

To fill in from ALF 2.0:

11.9.5 SETUP and HOLD statement

To fill in from ALF 2.0:

11.9.6 NOCHANGE statement

To fill in from ALF 2.0:

11.9.7 RECOVERY and REMOVAL statement

To fill in from ALF 2.0:

11.9.8 SKEW statement

To fill in from ALF 2.0:

11.9.9 PULSEWIDTH statement

To fill in from ALF 2.0:

11.9.10 PERIOD statement

To fill in from ALF 2.0:

11.9.11 JITTER statement

To fill in from ALF 2.0:

11.9.12 THRESHOLD statement

To fill in from ALF 2.0:

11.10 Auxiliary statements related to timing data

11.10.1 FROM and TO statement

To fill in from ALF 2.0:


```
from ::=
   FROM { from_to_items }

to ::=
   TO { from_to_items }

from_to_items ::= from_to_item { from_to_item }

from_to_item ::=
      PIN_single_value_annotation
   |  EDGE_single_value_annotation
   |  THRESHOLD_arithmetic_model
```

11.10.2 EARLY and LATE statement

To fill in from ALF 2.0:

```
EARLY_arithmetic_model_container ::=
   EARLY { early_late_arithmetic_models }

LATE_arithmetic_model_container ::=
   LATE { early_late_arithmetic_models }

early_late_arithmetic_models ::=
   early_late_arithmetic_model
      { early_late_arithmetic_model }

early_late_arithmetic_model ::=
      DELAY_arithmetic_model
   |  RETAIN_arithmetic_model
   |  SLEWRATE_arithmetic_model
```

11.10.3 Annotations for arithmetic models for timing data

To fill in from ALF 2.0:

11.11 Arithmetic models for environmental data

To fill in from ALF 2.0:

11.11.1 PROCESS and DERATE_CASE statement

To fill in from ALF 2.0:

11.11.2 TEMPERATURE statement

To fill in from ALF 2.0:

11.12 Arithmetic models for electrical data

To fill in from ALF 2.0:

11.12.1 CAPACITANCE statement

11.12.2 RESISTANCE statement

11.12.3 INDUCTANCE statement

11.12.4 VOLTAGE statement

11.12.5 CURRENT statement

11.12.6 POWER and ENERGY statement

To fill in from ALF 2.0:

11.12.7 FLUX and FLUENCE statement

To fill in from ALF 2.0:

11.12.8 DRIVE_STRENGTH statement

To fill in from ALF 2.0.2:

11.12.9 SWITCHING_BITS statement

To fill in from ALF 2.0:

11.12.10 NOISE and NOISE_MARGIN statement

To fill in from ALF 2.0:

11.12.11 Annotations for arithmetic models for electrical data

To fill in from ALF 2.0:

11.13 Arithmetic models for physical data

11.13.1 CONNECTIVITY statement

To fill in from ALF 2.0:

11.13.2 SIZE statement

11.13.3 AREA statement

11.13.4 WIDTH statement

11.13.5 HEIGHT statement

11.13.6 LENGTH statement

11.13.7 DISTANCE statement

11.13.8 OVERHANG statement

11.13.9 PERIMETER statement

11.13.10 EXTENSION statement

11.13.11 THICKNESS statement

11.13.12 Annotations for arithmetic models for physical data

To fill in from ALF 2.0:

11.14 Arithmetic submodels for timing and electrical data

11.14.1 RISE and FALL statement

To fill in from ALF 2.0:

11.14.2 HIGH and LOW statement

To fill in from ALF 2.0:

11.15 Arithmetic submodels for physical data

11.15.1 HORIZONTAL and VERTICAL statement

12. Syntax Rule Summary

Suggestion: put all syntax items in alphabetical order, no subsections, since the syntax is already introduced for each item in a dedicated subchapter.

Annex A

(informative)

Bibliography

[B1] Ratzlaff, C. L., Gopal, N., and Pillage, L. T., “RICE: Rapid Interconnect Circuit Evaluator,” *Proceedings of 28th Design Automation Conference*, pp. 555–560, 1991.

[B2] SPICE 2G6 User’s Guide.

[B3] Standard Delay Format Specification, Version 3.0, Open Verilog International, May 1995.

[B4] The IEEE Standard Dictionary of Electrical and Electronics Terms, Sixth Edition.

Index

T

U

V

W