

**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 P1603 Draft 1

September 17, 2001

Copyright© 2001, 2002, 2003 by IEEE. All rights reserved.

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.com

Technical Editor

Revision history:

IEEE P1596 Draft 0	August 19, 2001
IEEE P1603 Draft 1	September 17, 2001

Table of Contents

1.	Introduction.....	1
1.1	Motivation.....	1
1.2	Goals	2
1.3	Target applications.....	2
1.4	Conventions	5
1.5	Contents of this standard.....	5
2.	References.....	7
3.	Definitions	9
4.	Acronyms and abbreviations	11
5.	Language construction principles	13
6.	Lexical rules.....	15
6.1	Cross-reference of lexical tokens.....	15
6.2	Characters	15
6.2.1	Character set.....	15
6.2.2	Whitespace characters	16
6.2.3	Other characters.....	16
6.3	Lexical tokens	17
6.3.1	Delimiter.....	17
6.3.2	Comment	18
6.3.3	Number.....	18
6.3.4	Bit literals	18
6.3.5	Based literals	19
6.3.6	Edge literals.....	20
6.3.7	Quoted strings.....	20
6.3.8	Identifier	21
6.4	Keywords	23
6.4.1	Keywords for objects.....	23
6.4.2	Keywords for operators.....	23
6.4.3	Context-sensitive keywords	24
6.5	Rules against parser ambiguity	24
6.6	Values	24
6.6.1	Arithmetic value	24
6.6.2	String value.....	25
6.6.3	Edge values.....	25
6.6.4	Index value	25
7.	Auxiliary items	27
7.1	Index and related items	27
7.1.1	Index.....	27
7.1.2	Index range.....	27

7.2	Pin assignment and related items	27
7.2.1	Pin assignment	27
7.2.2	Pin variable	28
7.2.3	Pin value	28
7.3	Annotation and related items	29
7.3.1	Annotations.....	29
7.3.2	Annotation value.....	29
7.4	All purpose item.....	30
8.	Generic objects.....	31
8.1	INCLUDE statement.....	31
8.2	ALIAS statement.....	31
8.3	CONSTANT statement.....	31
8.4	ATTRIBUTE statement	31
8.5	PROPERTY statement.....	32
8.6	CLASS statement.....	32
8.7	KEYWORD statement.....	32
8.8	GROUP statement.....	33
8.9	TEMPLATE statement	33
9.	Library-specific objects.....	35
9.1	LIBRARY statement and related statements	35
9.1.1	LIBRARY statement	35
9.1.2	SUBLIBRARY statement	35
9.1.3	INFORMATION statement.....	35
9.2	CELL statement and related statements.....	36
9.2.1	CELL statement.....	36
9.2.2	NON_SCAN_CELL statement.....	36
9.2.3	Annotations and attributes for a CELL.....	37
9.3	PIN statement and related statements	37
9.3.1	PIN statement	37
9.3.2	RANGE statement	38
9.3.3	PIN_GROUP statement.....	38
9.3.4	Annotations and attributes for a PIN	38
9.4	WIRE statement and related statements.....	39
9.4.1	WIRE statement.....	39
9.4.2	NODE statement.....	40
9.5	VECTOR statement and related statements.....	40
9.5.1	VECTOR statement.....	40
9.5.2	ILLEGAL statement.....	40
9.5.3	Annotations and attributes for a VECTOR.....	41
9.6	LAYER statement and related statements	41
9.6.1	LAYER statement.....	41
9.6.2	Annotations for a LAYER.....	41
9.7	VIA statement and related statements.....	42
9.7.1	VIA statement.....	42
9.7.2	Annotations for a VIA	42
9.7.3	VIA reference statement.....	42
9.8	Statements related to physical design rules.....	42
9.8.1	RULE statement	43
9.8.2	ANTENNA statement.....	43
9.8.3	BLOCKAGE statement	43

9.8.4	PORT statement	44
9.9	Statements related to physical geometry	44
9.9.1	SITE statement	45
9.9.2	ARRAY statement.....	45
9.9.3	PATTERN statement.....	45
9.9.4	ARTWORK statement	46
9.9.5	Geometric model	46
9.9.6	Geometric transformation.....	47
9.10	Statements related to functional description	47
9.10.1	FUNCTION statement	47
9.10.2	TEST statement	47
9.10.3	BEHAVIOR statement	48
9.10.4	STRUCTURE statement	48
9.10.5	VIOLATION statement.....	48
9.10.6	STATETABLE statement	49
9.10.7	PRIMITIVE statement	50
10.	Constructs for modeling of digital behavior	51
10.1	Boolean expression language.....	51
10.2	Vector expression language	52
10.3	Control expression semantics	52
11.	Constructs for modeling of analog behavior.....	55
11.1	Arithmetic expression language.....	55
11.2	Arithmetic model and related statements.....	56
11.2.1	Arithmetic model statement	56
11.2.2	Partial arithmetic model	56
11.2.3	Non-trivial arithmetic model	57
11.2.4	Trivial arithmetic model	57
11.2.5	Assignment arithmetic model.....	57
11.2.6	Items for any arithmetic model	58
11.3	Arithmetic submodel and related statements	58
11.3.1	Arithmetic submodel statement.....	58
11.3.2	Non-trivial arithmetic submodel	58
11.3.3	Trivial arithmetic submodel	59
11.3.4	Items for any arithmetic submodel.....	59
11.4	Arithmetic body and related statements.....	59
11.4.1	Arithmetic body.....	59
11.4.2	HEADER statement	59
11.4.3	TABLE statement.....	60
11.4.4	EQUATION statement	60
11.5	Arithmetic model container	60
11.6	Statements related to arithmetic models for general purpose	61
11.6.1	MIN and MAX statements	61
11.6.2	TYP statement	61
11.6.3	DEFAULT statement	61
11.6.4	LIMIT statement.....	61
11.6.5	Annotations for arithmetic models for general purpose	61
11.7	Rules for evaluation of arithmetic models	61
11.7.1	Arithmetic model with arithmetic submodels	62
11.7.2	Arithmetic model with table arithmetic body.....	62
11.7.3	Arithmetic model with equation arithmetic body.....	62

11.8 Overview of arithmetic models.....	62
11.9 Arithmetic models for timing data	62
11.9.1 TIME statement	63
11.9.2 FREQUENCY statement.....	63
11.9.3 DELAY and RETAIN statements	63
11.9.4 SLEWRATE statement.....	63
11.9.5 SETUP and HOLD statement.....	63
11.9.6 NOCHANGE statement	63
11.9.7 RECOVERY and REMOVAL statements	63
11.9.8 SKEW statement.....	63
11.9.9 PULSEWIDTH statement	64
11.9.10 PERIOD statement	64
11.9.11 JITTER statement	64
11.9.12 THRESHOLD statement	64
11.10 Auxiliary statements related to timing data	64
11.10.1 FROM and TO statements	64
11.10.2 EARLY and LATE statements	65
11.10.3 Annotations for arithmetic models for timing data	65
11.11 Arithmetic models for environmental data	65
11.11.1 PROCESS and DERATE_CASE statement.....	65
11.11.2 TEMPERATURE statement.....	65
11.12 Arithmetic models for electrical data	65
11.12.1 CAPACITANCE statement.....	66
11.12.2 RESISTANCE statement.....	66
11.12.3 INDUCTANCE statement.....	66
11.12.4 VOLTAGE statement	66
11.12.5 CURRENT statement	66
11.12.6 POWER and ENERGY statement.....	66
11.12.7 FLUX and FLUENCE statement.....	66
11.12.8 DRIVE_STRENGTH statement.....	66
11.12.9 SWITCHING_BITS statement.....	66
11.12.10 NOISE and NOISE_MARGIN statement.....	67
11.12.11 Annotations for arithmetic models for electrical data	67
11.13 Arithmetic models for physical data	67
11.13.1 CONNECTIVITY statement	67
11.13.2 SIZE statement	67
11.13.3 AREA statement.....	67
11.13.4 WIDTH statement.....	67
11.13.5 HEIGHT statement.....	67
11.13.6 LENGTH statement.....	67
11.13.7 DISTANCE statement	68
11.13.8 OVERHANG statement	68
11.13.9 PERIMETER statement.....	68
11.13.10 EXTENSION statement.....	68
11.13.11 THICKNESS statement	68
11.13.12 Annotations for arithmetic models for physical data.....	68
11.14 Arithmetic submodels for timing and electrical data	68
11.14.1 RISE and FALL statement	68
11.14.2 HIGH and LOW statement.....	68
11.15 Arithmetic submodels for physical data.....	68
11.15.1 HORIZONTAL and VERTICAL statement.....	69
(normative) Syntax rule summary.....	71

A.1	Lexical definitions	71
A.2	Auxiliary definitions	73
A.3	Generic definitions.....	75
A.4	Library definitions	76
A.5	Control definitions	83
A.6	Arithmetic definitions	84
(informative)	Bibliography	87

List of Figures

Figure 1—ALF and its target applications	4
--	---

List of Tables

Table 1—Target applications and models supported by ALF.....2

Table 2—Cross-reference of lexical tokens.....15

Table 3—List of whitespace characters16

Table 4—Single bit constants19

Table 5—Special characters in quoted strings21

Table 6—Object keywords23

Table 7—Built-in arithmetic function keywords23

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

1. Introduction

I **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
<i>Simulation</i>	Derived from ALF	N/A	N/A
<i>Synthesis</i>	Supported by ALF	Supported by ALF	Supported by ALF
<i>Design for test</i>	Supported by ALF	N/A	N/A

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

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

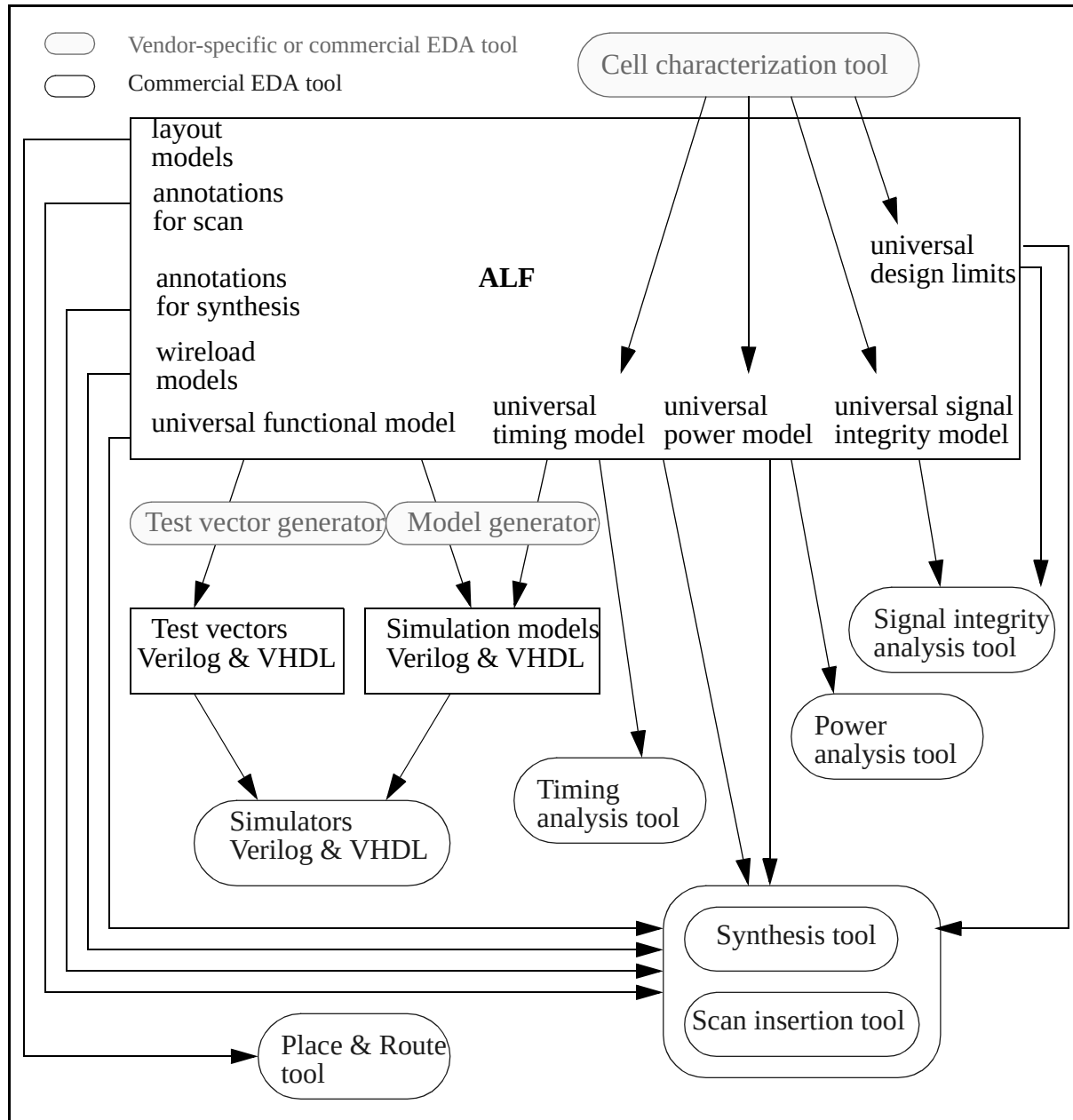


Figure 1—ALF and its target applications

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.
- Annexes. Following Clause 11 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

¹ISO 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 arithmetic 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:

3.	Object model.....	15
3.1	Syntax conventions.....	15
3.7	Relationships between objects.....	23
3.9	Relations between objects.....	27
3.9.1	Keywords for referencing objects used as annotation.....	28
3.9.3	Other incremental definitions.....	29
4.	Library organization	31
4.1	Scoping rules.....	31

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
alphanumeric_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 is shown in Syntax 6. **More??

```

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.

I

I

```
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, as shown in Syntax 14. 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.

```

identifiers ::=
  identifier { identifier }
identifier ::=
  nonescaped_identifier
  | escaped_identifier
  | placeholder_identifier
  | hierarchical_identifier

```

Syntax 14—Identifiers

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 a *non-escaped identifier*, as shown in Syntax 15.

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

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

Syntax 15—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 16.

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

Syntax 16—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 17.

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

Syntax 17—Placeholder identifier

6.3.8.4 Hierarchical identifier

A *hierarchical identifier* shall be defined as shown in Syntax 18, with no whitespace in-between the characters.

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

Syntax 18—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.
LOG	Natural logarithm.
MIN	Minimum.

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

Term	Definition
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

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

Some of these values are shown as plural (e.g., arithmetic values) others as singular (e.g., a string value)

6.6.1 Arithmetic value

Arithmetic values XXX, as shown in Syntax 19.

```

arithmetic_values ::=
    arithmetic_value { arithmetic_value }
arithmetic_value ::=
    number
    | identifier
    | pin_value

```

Syntax 19—Arithmetic values

Purpose: Data for calculation described in `arithmetic_model` or in `arithmetic_assignment` for `dynamic_template_instantiation`.

Semantic restriction: `arithmetic_value` shall resolve to a valid value for the particular `arithmetic_model`, where it is used. Some `arithmetic_models` allow only unsigned (e.g., `SWITCHING_BITS` or `FANOUT`), others allow only `non_negative_numbers` (e.g., `WIDTH` or `LENGTH`). Non-interpolatable `arithmetic_models` (e.g., `PROCESS` or `DERATE_CASE`) allow only symbolic identifiers rather than numbers.

6.6.2 String value

A string value XXX, as shown in Syntax 20.

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

Syntax 20—String value

Purpose: Textual data.

6.6.3 Edge values

Edge values XXX, as shown in Syntax 21.

```
edge_values ::=  
    edge_value { edge_value }  
edge_value ::=  
    ( edge_literal )
```

Syntax 21—Edge values

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` appears only within a `vector_expression`. In that context, the enclosing parentheses are not necessary.

6.6.4 Index value

An index value XXX, as shown in Syntax 22.

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

Syntax 22—Index value

`index_value` shall resolve to `unsigned`, i.e., `identifier` shall be the name of a `CONSTANT` with an unsigned value or a placeholder in `TEMPLATE` which gets replaced with `unsigned`.

7. Auxiliary items

****Add lead-in text****

7.1 Index and related items

****The following two syntax boxes could be combined****

7.1.1 Index

An index XXX, as shown in Syntax 23.

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

Syntax 23—Index

7.1.2 Index range

An index range XXX, as shown in Syntax 24.

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

Syntax 24—Index range

`index_range` shall define consecutive unsigned numbers, bound by the `index_value` left and right of the colon (:). In the context of a `PIN` statement, the left `index_value` shall be considered as the MSB, the right `index_value` shall be considered as the MSB. `index_value` can also be used in the `RANGE` and `GROUP` statements.

7.2 Pin assignment and related items

****Add lead-in text****

7.2.1 Pin assignment

A pin assignment XXX, as shown in Syntax 25.

```
pin_assignments ::=
    pin_assignment { pin_assignment }
pin_assignment ::=
    pin_variable = pin_value ;
```

Syntax 25—Pin assignment

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

Semantic restrictions: The `pin_value` shall 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. a `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`, another one-dimensional slice of a two-dimensional `pin_variable` of same bitwidth, a `based_literal` of the same bitwidth, or to a `unsigned` which 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 (0).

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

A pin variable XXX, as shown in Syntax 26.

```
pin_variables ::=
    pin_variable { pin_variable }
pin_variable ::=
    pin_variable_identifier [ index ]
```

Syntax 26—Pin variable

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

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

7.2.3 Pin value

A pin value XXX, as shown in Syntax 27.

```
pin_values ::=
    pin_value { pin_value }
pin_value ::=
    pin_variable
    | bit_literal
    | based_literal
    | unsigned
```

Syntax 27—Pin values

Purpose: `pin_value` defines the set of values which can be assigned to a `pin_variable`. Assigning a `pin_variable` to another `pin_variable` shall be legal. It can also be used in the context of a NON_SCAN_CELL, STRUCTURE, or 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

****Add lead-in text****

7.3.1 Annotations

An annotation is an auxiliary statement within the context of a `library_specific_object`, a `library_specific_singular_object`, or an `arithmetic_model`, as shown in Syntax 28. 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 }
```

Syntax 28—Annotations

7.3.2 Annotation value

An annotation value XXX, as shown in Syntax 29.

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

Syntax 29—Annotation values

NOTE—There is lexical overlap, but semantic distinction between the possible annotation values???

7.4 All purpose item

An all purpose item XXX, as shown in Syntax 30.

```
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
```

Syntax 30—All purpose items

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:

3.2 Generic objects	16
---------------------------	----

8.1 INCLUDE statement

| An INCLUDE statement XXX, as shown in Syntax 31.

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

Syntax 31—INCLUDE statement

To fill in from ALF 2.0:

3.2.3 INCLUDE statement	17
4.2 Use of multiple files.....	32

8.2 ALIAS statement

| An ALIAS statement XXX, as shown in Syntax 32.

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

Syntax 32—ALIAS statement

To fill in from ALF 2.0:

3.2.2 ALIAS statement.....	17
----------------------------	----

8.3 CONSTANT statement

| A CONSTANT statement XXX, as shown in Syntax 33.

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

Syntax 33—CONSTANT statement

To fill in from ALF 2.0:

3.2.1 CONSTANT statement	17
--------------------------------	----

8.4 ATTRIBUTE statement

| An ATTRIBUTE statement XXX, as shown in Syntax 34.

```

attribute ::=
ATTRIBUTE { identifiers }

```

Syntax 34—ATTRIBUTE statement

To fill in from ALF 2.0:

3.2.5	ATTRIBUTE statement.....	18
-------	--------------------------	----

8.5 PROPERTY statement

A PROPERTY statement XXX, as shown in Syntax 35.

```

property ::=
PROPERTY [ identifier ] { one_level_annotations }

```

Syntax 35—PROPERTY statement

To fill in from ALF 2.0:

3.2.7	PROPERTY statement	19
-------	--------------------------	----

8.6 CLASS statement

A CLASS statement XXX, as shown in Syntax 36.

```

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

```

Syntax 36—CLASS statement

To fill in from ALF 2.0:

3.2.4	CLASS statement	17
-------	-----------------------	----

8.7 KEYWORD statement

A KEYWORD statement XXX, as shown in Syntax 37.

```

keyword_declaration ::=
KEYWORD context_sensitive_keyword = syntax_item_identifier ;

```

Syntax 37—KEYWORD statement

To fill in from ALF 2.0.2:

3.2.9	KEYWORD statement	19
-------	-------------------------	----

8.8 GROUP statement

| A GROUP statement XXX, as shown in Syntax 38.

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

Syntax 38—GROUP statement

To fill in from ALF 2.0:

3.2.8	GROUP statement	19
-------	-----------------------	----

Semantics: When the group identifier is used in 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

| A TEMPLATE statement XXX, as shown in Syntax 39.

To fill in from ALF 2.0:

3.2.6	TEMPLATE statement.....	18
5.6.8	Parameterizeable cells	97

```

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

```

Syntax 39—*TEMPLATE* statement

9. Library-specific objects

To fill in from ALF 2.0:

3.3 Library-specific objects	20
3.6 Library-specific singular objects	22
6. Modeling for synthesis and test	101

9.1 LIBRARY statement and related statements

****Add lead-in text****

9.1.1 LIBRARY statement

A LIBRARY statement XXX, as shown in Syntax 40.

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

Syntax 40—LIBRARY statement

9.1.2 SUBLIBRARY statement

A SUBLIBRARY statement XXX, as shown in Syntax 41.

```
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
```

Syntax 41—SUBLIBRARY statement

9.1.3 INFORMATION statement

An INFORMATION statement XXX, as shown in Syntax 42.

```

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

```

Syntax 42—*INFORMATION statement*

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

To fill in from ALF 2.0:

3.8 INFORMATION container..... 26

9.2 CELL statement and related statements

****Add lead-in text****

9.2.1 CELL statement

A CELL statement XXX, as shown in Syntax 43.

```

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

```

Syntax 43—*CELL statement*

9.2.2 NON_SCAN_CELL statement

A NON_SCAN_CELL statement XXX, as shown in Syntax 44.

To fill in from ALF 2.0:

```

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 }

```

Syntax 44—NON_SCAN_CELL statement

6.2 NON_SCAN_CELL statement.....	108
----------------------------------	-----

9.2.3 Annotations and attributes for a CELL

****Add lead-in text****

Syntax snippet:

To fill in from ALF 2.0:

6.1 Annotations and attributes for a CELL.....	101
6.1.1 CELLTYPE annotation	101
6.1.2 ATTRIBUTE within a CELL object	101
6.1.3 SWAP_CLASS annotation.....	103
6.1.3 RESTRICT_CLASS annotation.....	103
6.1.3 Independent SWAP_CLASS and RESTRICT_CLASS.....	104
6.1.3 SWAP_CLASS with inherited RESTRICT_CLASS.....	105
6.1.4 SCAN_TYPE annotation	106
6.1.5 SCAN_USAGE annotation	106
6.1.6 BUFFERTYPE annotation.....	107
6.1.7 DRIVERTYPE annotation	107
6.1.8 PARALLEL_DRIVE annotation	107
9.16 Physical annotations for CELL.....	253
9.16.1 PLACEMENT_TYPE annotation	262
9.16.2 Reference of a SITE by a CELL	262

9.3 PIN statement and related statements

****Add lead-in text****

9.3.1 PIN statement

A PIN statement XXX, as shown in Syntax 45.

To fill in from ALF 2.0:

6.5 Definitions for bus pins.....	127
5.5.3 Multi-dimensional variables.....	83
6.5.2 Scalar pins inside a bus	128

```

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 }

```

Syntax 45—PIN statement

9.3.2 RANGE statement

A RANGE statement XXX, as shown in Syntax 46.

```

range ::=
    RANGE { index_range }

```

Syntax 46—RANGE statement

To fill in from ALF 2.0:

6.5.1 RANGE for bus pins..... 127

9.3.3 PIN_GROUP statement

A PIN_GROUP statement XXX, as shown in Syntax 47.

```

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

```

Syntax 47—PIN_GROUP statement

To fill in from ALF 2.0:

6.5.3 PIN_GROUP statement..... 129

9.3.4 Annotations and attributes for a PIN

To fill in from ALF 2.0:

6.4 Annotations and attributes for a PIN..... 114
 6.4.1 VIEW annotation..... 114
 6.4.2 PINTYPE annotation..... 115

6.4.3	DIRECTION annotation.....	115
6.4.4	SIGNALTYPE annotation	116
6.4.5	ACTION annotation	120
6.4.6	POLARITY annotation	121
6.4.7	DATATYPE annotation	122
6.4.8	INITIAL_VALUE annotation.....	122
6.4.9	SCAN_POSITION annotation	122
6.4.10	STUCK annotation.....	122
6.4.11	SUPPLYTYPE	123
6.4.12	SIGNAL_CLASS.....	123
6.4.13	SUPPLY_CLASS.....	124
6.4.14	Driver CELL and PIN specification.....	125
6.4.15	DRIVETYPE annotation.....	125
6.4.16	SCOPE annotation.....	126
6.4.17	PULL annotation.....	126
6.4.18	ATTRIBUTE for PIN objects	126
6.9.2	Definitions of pin ATTRIBUTE values for memory BIST	138
9.17	Physical annotations for PIN	263
9.17.1	CONNECT_CLASS annotation.....	263
9.17.2	SIDE annotation	263
9.17.3	ROW and COLUMN annotation.....	263
9.17.4	ROUTING_TYPE annotation	264

9.4 WIRE statement and related statements

****Add lead-in text****

9.4.1 WIRE statement

A WIRE statement XXX, as shown in Syntax 48.

```

wire ::=
    WIRE wire_identifier { wire_items }
    | WIRE wire_identifier ;
    | wire_template_instantiation
wire_items ::=
    wire_item { wire_item }
wire_item ::=
    all_purpose_item
    | node

```

Syntax 48—WIRE statement

To fill in from ALF 2.0:

8.15	Interconnect parasitics and analysis.....	209
8.15.1	Principles of the WIRE statement	209
8.15.2	Statistical wireload models.....	210
8.15.3	Boundary parasitics	211
8.15.5	Interconnect delay and noise calculation.....	216
8.15.6	SELECT_CLASS annotation for WIRE statement.....	217

9.4.2 NODE statement

A NODE statement XXX, as shown in Syntax 49.

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

Syntax 49—NODE statement

To fill in from ALF 2.0:

8.15.4 NODE declaration 213

9.5 VECTOR statement and related statements

Add lead-in text

9.5.1 VECTOR statement

A VECTOR statement XXX, as shown in Syntax 50.

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

Syntax 50—VECTOR statement

9.5.2 ILLEGAL statement

A ILLEGAL statement XXX, as shown in Syntax 51.

```
illegal ::=
    ILLEGAL { illegal_items }
    | illegal_template_instantiation
illegal_items ::=
    illegal_item { illegal_item }
illegal_item ::=
    all_purpose_item
    | violation
```

Syntax 51—ILLEGAL statement

To fill in from ALF 2.0:

6.7	ILLEGAL statement for VECTOR.....	135
-----	-----------------------------------	-----

9.5.3 Annotations and attributes for a VECTOR

To fill in from ALF 2.0:

6.6	Annotations for CLASS and VECTOR.....	130
6.6.1	PURPOSE annotation.....	130
6.6.2	OPERATION annotation	131
6.6.3	LABEL annotation	133
6.6.4	EXISTENCE_CONDITION annotation	133
6.6.5	EXISTENCE_CLASS annotation.....	133
6.6.6	CHARACTERIZATION_CONDITION annotation	134
6.6.7	CHARACTERIZATION_VECTOR annotation.....	134
6.6.8	CHARACTERIZATION_CLASS annotation	135
3.9.2	Incremental definitions for VECTOR	29

9.6 LAYER statement and related statements

To fill in from ALF 2.0:

9.	Physical modeling.....	219
9.1	Overview.....	219

9.6.1 LAYER statement

A LAYER statement XXX, as shown in Syntax 52.

```

layer ::=
    LAYER layer_identifier { layer_items }
    | LAYER layer_identifier ;
    | layer_template_instantiation
layer_items ::=
    layer_item { layer_item }
layer_item ::=
    all_purpose_item

```

Syntax 52—LAYER statement

9.6.2 Annotations for a LAYER

To fill in from ALF 2.0:

9.5	LAYER statement.....	227
9.5.1	Definition.....	227
9.5.2	PURPOSE annotation.....	228
9.5.3	PITCH annotation.....	229
9.5.4	PREFERENCE annotation.....	229
9.5.5	Example.....	229

9.7 VIA statement and related statements

Add lead-in text

9.7.1 VIA statement

A VIA statement XXX, as shown in Syntax 53.

```
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
```

Syntax 53—VIA statement

9.7.2 Annotations for a VIA

To fill in from ALF 2.0:

9.8	VIA statement	237
9.8.1	Definition.....	237
9.8.2	USAGE annotation.....	238
9.8.3	Example.....	239

9.7.3 VIA reference statement

A VIA reference statement XXX, as shown in Syntax 54.

```
via_reference ::=
    VIA { via_instantiations }
    | VIA { via_identifiers }
via_instantiations ::=
    via_instantiation { via_instantiation }
via_instantiation ::=
    via_identifier { geometric_transformations }
```

Syntax 54—VIA reference statement

To fill in from ALF 2.0:

9.8.4	VIA reference	240
9.11.7	VIA reference	249

9.8 Statements related to physical design rules

Add lead-in text

9.8.1 RULE statement

A RULE statement XXX, as shown in Syntax 55.

```
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
```

Syntax 55—RULE statement

To fill in from ALF 2.0:

9.11 RULE statement.....	244
9.11.1 Definition.....	244
9.11.2 Width-dependent spacing	245
9.11.3 End-of-line rule	246
9.11.4 Redundant vias	247
9.11.5 Extraction rules.....	248
9.11.6 RULES within BLOCKAGE or PORT	248

9.8.2 ANTENNA statement

An ANTENNA statement XXX, as shown in Syntax 56.

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

Syntax 56—ANTENNA statement

To fill in from ALF 2.0:

9.13 ANTENNA statement.....	251
9.13.1 Definition.....	251
9.13.2 Layer-specific antenna rules.....	252
9.13.3 All-layer antenna rules	253
9.13.4 Cumulative antenna rules	254
9.13.5 Illustration	255

9.8.3 BLOCKAGE statement

A BLOCKAGE statement XXX, as shown in Syntax 57.

```

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

```

Syntax 57—BLOCKAGE statement

To fill in from ALF 2.0:

9.9 BLOCKAGE statement.....	240
9.9.1 Definition.....	240
9.9.2 Example.....	241

9.8.4 PORT statement

A PORT statement XXX, as shown in Syntax 58.

```

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

```

Syntax 58—PORT statement

To fill in from ALF 2.0:

9.10 PORT statement.....	241
9.10.1 Definition.....	241
9.10.2 VIA reference	242
9.10.3 CONNECTIVITY rules for PORT and PIN.....	242
9.10.4 Reference of a declared PORT in a PIN annotation.....	243
9.10.5 VIEW annotation.....	244
9.10.6 LAYER annotation.....	244
9.10.7 ROUTING_TYPE	244

9.9 Statements related to physical geometry

Add lead-in text

9.9.1 SITE statement

A SITE statement XXX, as shown in Syntax 59.

```

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

```

Syntax 59—SITE statement

To fill in from ALF 2.0.2:

9.12 SITE statement.....	249
9.12.1 Definition.....	249
9.12.2 ORIENTATION_CLASS and SYMMETRY_CLASS.....	249
9.12.3 Example.....	250

9.9.2 ARRAY statement

An ARRAY statement XXX, as shown in Syntax 60.

```

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

```

Syntax 60—ARRAY statement

To fill in from ALF 2.0:

9.14 ARRAY Statement	256
9.14.1 Definition.....	256
9.14.2 PURPOSE annotation.....	256
9.14.3 Examples	257

9.9.3 PATTERN statement

A PATTERN statement XXX, as shown in Syntax 61.

To fill in from ALF 2.0:

9.7 PATTERN statement	235
-----------------------------	-----

```

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

```

Syntax 61—*PATTERN* statement

9.7.1	Definition.....	235
9.7.2	SHAPE annotation.....	235
9.7.3	LAYER annotation.....	236
9.7.4	EXTENSION annotation.....	236
9.7.5	VERTEX annotation.....	237
9.7.6	PATTERN with geometric model	237
9.7.7	Example.....	237

9.9.4 ARTWORK statement

An ARTWORK statement XXX, as shown in Syntax 62.

```

artwork ::=
    ARTWORK = artwork_identifier { artwork_items }
    | ARTWORK = artwork_identifier ;
    | artwork_template_instantiation
artwork_items ::=
    artwork_item { artwork_item }
artwork_item ::=
    geometric_transformation
    | pin_assignment

```

Syntax 62—*ARTWORK* statement

To fill in from ALF 2.0:

9.4	ARTWORK statement	226
-----	-------------------------	-----

9.9.5 Geometric model

A geometric model XXX, as shown in Syntax 63.

To fill in from ALF 2.0:

3.5	Geometric models	22
9.6	Geometric model statement	230
9.6.1	Definition.....	231
9.6.2	Predefined geometric models using TEMPLATE.....	233

```

geometric_model ::=
    nonescaped_identifier [ 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 } }

```

Syntax 63—Geometric model

9.9.6 Geometric transformation

A geometric transformation XXX, as shown in Syntax 64.

```

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 ]
}

```

Syntax 64—Geometric transformation

To fill in from ALF 2.0.2:

9.3	Statements for geometric transformation.....	223
9.3.1	SHIFT statement.....	223
9.3.2	ROTATE statement.....	223
9.3.3	FLIP statement	224
9.3.4	REPEAT statement.....	224
9.3.5	Summary of geometric transformations	225

9.10 Statements related to functional description

To fill in from ALF 2.0.2:

5.	Functional modeling	33
----	---------------------------	----

9.10.1 FUNCTION statement

A FUNCTION statement XXX, as shown in Syntax 65.

9.10.2 TEST statement

A TEST statement XXX, as shown in Syntax 66.

```

function ::=
    FUNCTION { function_items }
    | function_template_instantiation
function_items ::=
    function_item { function_item }
function_item ::=
    all_purpose_item
    | behavior
    | structure
    | statetable

```

Syntax 65—FUNCTION statement

```

test ::=
    TEST { test_items }
    | test_template_instantiation
test_items ::=
    test_item { test_item }
test_item ::=
    all_purpose_item
    | behavior
    | statetable

```

Syntax 66—TEST statement

To fill in from ALF 2.0:

6.8	TEST statement.....	136
6.9	Physical bitmap for memory BIST	136
6.9.1	Definition of concepts	136
6.9.3	Explanatory example	138

9.10.3 BEHAVIOR statement

A BEHAVIOR statement XXX, as shown in Syntax 67.

To fill in from ALF 2.0:

5.5.1	BEHAVIOR.....	81
-------	---------------	----

9.10.4 STRUCTURE statement

A STRUCTURE statement XXX, as shown in Syntax 68.

To fill in from ALF 2.0:

6.3	STRUCTURE statement.....	109
-----	--------------------------	-----

9.10.5 VIOLATION statement

A VIOLATION statement XXX, as shown in Syntax 69.

To fill in from ALF 2.0.2:

8.4	VIOLATION container.....	180
-----	--------------------------	-----

```

behavior ::=
    BEHAVIOR { behavior_items }
    | behavior_template_instantiation
behavior_items ::=
    behavior_item { behavior_item }
behavior_item ::=
    boolean_assignments
    | 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 } }

```

Syntax 67—*BEHAVIOR* statement

```

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 }

```

Syntax 68—*STRUCTURE* statement

```

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

```

Syntax 69—*VIOLATION* statement

9.10.6 STATETABLE statement

A STATETABLE statement XXX, as shown in Syntax 70.

To fill in from ALF 2.0:

5.5.2	STATETABLE.....	81
5.5.4	ROM initialization.....	84

```

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 )

```

Syntax 70—STATETABLE statement

9.10.7 PRIMITIVE statement

A PRIMITIVE statement XXX, as shown in Syntax 71.

```

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

```

Syntax 71—PRIMITIVE statement

To fill in from ALF 2.0:

5.6	Predefined models	85
5.6.1	Usage of PRIMITIVES	85
5.6.2	Concept of user-defined and predefined primitives	85
5.6.3	Predefined combinational primitives	87
5.6.4	Predefined tristate primitives	91
5.6.5	Predefined multiplexor	93
8.6.6	Predefined flip-flop	94
8.6.7	Predefined latch	95

10. Constructs for modeling of digital behavior

To fill in from ALF 2.0:

5.5	Variable declarations	80
5.1	Combinational functions	33
5.1.1	Combinational logic	33
5.1.2	Boolean operators on scalars	33
5.1.3	Boolean operators on words	34
5.1.4	Operator priorities	35
5.1.5	Datatype mapping	36
5.1.6	Rules for combinational functions	37
5.1.7	Concurrency in combinational functions	38
5.2	Sequential functions	39
5.2.1	Level-sensitive sequential logic	39
5.2.2	Edge-sensitive sequential logic	39
5.2.3	Unary operators for vector expressions	41
5.2.4	Basic rules for sequential functions	43
5.2.5	Concurrency in sequential functions	45
5.2.6	Initial values for logic variables	47
5.3	Higher-order sequential functions	48
5.3.1	Vector-sensitive sequential logic	48
5.3.2	Canonical binary operators for vector expressions	49
5.3.3	Complex binary operators for vector expressions	50
5.3.4	Operators for conditional vector expressions	53
5.3.5	Operators for sequential logic	54
5.3.6	Operator priorities	54
5.3.7	Using PINs in VECTORS	54
5.4	Modeling with vector expressions	55
5.4.1	Event reports	56
5.4.2	Event sequences	57
5.4.3	Scope and content of event sequences	58
5.4.4	Alternative event sequences	60
5.4.5	Symbolic edge operators	61
5.4.6	Non-events	62
5.4.7	Compact and verbose event sequences	63
5.4.8	Unspecified simultaneous events within scope	64
5.4.9	Simultaneous event sequences	66
5.4.10	Implicit local variables	68
5.4.11	Conditional event sequences	69
5.4.12	Alternative conditional event sequences	71
5.4.13	Change of scope within a vector expression	72
5.4.14	Sequences of conditional event sequences	76
5.4.15	Incompletely specified event sequences	78
5.4.16	How to determine well-specified vector expressions	79

10.1 Boolean expression language

The boolean expression language XXX, as shown in Syntax 72.

```

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

```

Syntax 72—Boolean expression language

10.2 Vector expression language

The vector expression language XXX, as shown in Syntax 73.

10.3 Control expression semantics

**Syntax 73 also shows the control expression syntax (at the bottom); is this deliberate??

```

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 )

```

Syntax 73—Vector expression language

11. Constructs for modeling of analog behavior

****Add lead-in text****

11.1 Arithmetic expression language

An arithmetic expression XXX, as shown in Syntax 74.

```
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 } )
```

Syntax 74—Arithmetic expression

An arithmetic unary XXX, as shown in Syntax 75.

```
arithmetic_unary ::=  
    sign
```

Syntax 75—Arithmetic unary

An arithmetic binary XXX, as shown in Syntax 76.

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

Syntax 76—Arithmetic binary

An arithmetic macro XXX, as shown in Syntax 77.

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

Syntax 77—Arithmetic macro

To fill in from ALF 2.0:

7.2 Arithmetic expressions.....	146
7.2.1 Syntax of arithmetic expressions.....	146
7.2.2 Arithmetic operators.....	147
7.2.3 Operator priorities.....	148

11.2 Arithmetic model and related statements

To fill in from ALF 2.0:

3.4 Arithmetic models.....	22
7. General rules for arithmetic models.....	143
7.1 Principles of arithmetic models.....	143
7.1.1 Global definitions for arithmetic models.....	143
7.1.2 Trivial arithmetic model	143
7.1.3 Arithmetic model using EQUATION.....	144
7.1.4 Arithmetic model using TABLE	144
7.1.5 Complex arithmetic model	145
7.3 Construction of arithmetic models.....	148
7.6 Arithmetic submodels	156

11.2.1 Arithmetic model statement

An arithmetic model statement XXX, as shown in Syntax 78.

```

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

```

Syntax 78—Arithmetic model statement

11.2.2 Partial arithmetic model

A partial arithmetic model XXX, as shown in Syntax 79.

```

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

```

Syntax 79—Partial arithmetic model

A 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., a partial arithmetic model without an 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.

A named partial arithmetic model (i.e., a partial arithmetic model without an 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

A non-trivial arithmetic model XXX, as shown in Syntax 80.

```

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

```

Syntax 80—Non-trivial arithmetic model

A non-trivial arithmetic model contains sufficient data to evaluate the model.

11.2.4 Trivial arithmetic model

A trivial arithmetic model XXX, as shown in Syntax 81.

```

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

```

Syntax 81—Trivial arithmetic model

A 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

An assignment arithmetic model XXX, as shown in Syntax 82.

```

assignment_arithmetic_model ::=
  arithmetic_model_identifier = arithmetic_expression ;

```

Syntax 82—Assignment arithmetic model

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.
- This construct is used in a dynamic template instantiation.

11.2.6 Items for any arithmetic model

Arithmetic model items XXX, as shown in Syntax 83.

```
any_arithmetic_model_items ::=
    any_arithmetic_model_item { any_arithmetic_model_item }
any_arithmetic_model_item ::=
    all_purpose_item
    | from
    | to
    | violation
```

Syntax 83—Arithmetic model items

Semantic restrictions apply, depending on the type and context of the arithmetic model. **Define these**

11.3 Arithmetic submodel and related statements

Add lead-in text

11.3.1 Arithmetic submodel statement

An arithmetic submodel statement XXX, as shown in Syntax 84.

```
arithmetic_submodels ::=
    arithmetic_submodel { arithmetic_submodel }
arithmetic_submodel ::=
    non_trivial_arithmetic_submodel
    | trivial_arithmetic_submodel
    | arithmetic_submodel_template_instantiation
```

Syntax 84—Arithmetic submodel statement

11.3.2 Non-trivial arithmetic submodel

A non-trivial arithmetic submodel XXX, as shown in Syntax 85.

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

Syntax 85—Non-trivial arithmetic submodel

A non-trivial arithmetic submodel contains sufficient data to evaluate the arithmetic submodel.

11.3.3 Trivial arithmetic submodel

A trivial arithmetic submodel XXX, as shown in Syntax 86.

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

Syntax 86—Trivial arithmetic submodel

A 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

Arithmetic submodel items XXX, as shown in Syntax 87.

```
any_arithmetic_submodel_items ::=  
    any_arithmetic_submodel_item { any_arithmetic_submodel_item }  
any_arithmetic_submodel_item ::=  
    all_purpose_item  
    | violation
```

Syntax 87—Arithmetic submodel items

Semantic restrictions apply, depending on the type and context of the arithmetic model. **Define these**

11.4 Arithmetic body and related statements

Add lead-in text

11.4.1 Arithmetic body

An arithmetic body XXX, as shown in Syntax 88.

```
arithmetic_body ::=  
    arithmetic_submodels  
    | table_arithmetic_body  
    | equation_arithmetic_body  
table_arithmetic_body ::=  
    header table [ equation ]  
equation_arithmetic_body ::=  
    [ header ] equation [ table ]
```

Syntax 88—Arithmetic body

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

11.4.2 HEADER statement

A HEADER statement XXX, as shown in Syntax 89.

```

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

```

Syntax 89—HEADER statement

The **HEADER** shall contain arguments for evaluating the arithmetic model. The arithmetic values of those arguments shall be supplied by application program.

Semantic restriction: No arithmetic submodel is allowed within an arithmetic model body.

11.4.3 TABLE statement

A **TABLE** statement XXX, as shown in Syntax 90.

```

table ::=
    TABLE { arithmetic_values }
    | table_template_instantiation

```

Syntax 90—TABLE statement

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

11.4.4 EQUATION statement

An **EQUATION** statement XXX, as shown in Syntax 91.

```

equation ::=
    EQUATION { arithmetic_expression }
    | equation_template_instantiation

```

Syntax 91—EQUATION statement

An **EQUATION** shall provide the means for evaluation using an analytical method.

11.5 Arithmetic model container

To fill in from ALF 2.0:

7.5 Containers for arithmetic models.....	155
7.1.6 Containers for arithmetic models and submodels.....	146

An arithmetic model container XXX, as shown in Syntax 92.

```

arithmetic_model_container ::=
    arithmetic_model_container_identifier { arithmetic_models }

```

Syntax 92—Arithmetic model container

11.6 Statements related to arithmetic models for general purpose

****Add lead-in text****

11.6.1 MIN and MAX statements

To fill in from ALF 2.0:

7.6.1	Semantics of MIN / TYP / MAX	157
-------	------------------------------------	-----

11.6.2 TYP statement

****Add lead-in text****

11.6.3 DEFAULT statement

To fill in from ALF 2.0:

7.4.1	DEFAULT annotation.....	150
7.6.2	Semantics of DEFAULT	158

11.6.4 LIMIT statement

To fill in from ALF 2.0:

8.13	Reliability calculation	197
8.13.3	Global LIMIT specifications	199
8.13.4	LIMIT and model specification in the same context.....	199
8.13.5	Model and argument specification in the same context	201

11.6.5 Annotations for arithmetic models for general purpose

To fill in from ALF 2.0:

7.4	Annotations for arithmetic models	150
7.4.2	UNIT annotation.....	150
7.4.3	CALCULATION annotation.....	151
7.4.4	INTERPOLATION annotation	152

11.7 Rules for evaluation of arithmetic models

****Add lead-in text****

11.7.1 Arithmetic model with arithmetic submodels

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

11.7.2 Arithmetic model with table arithmetic body

All arithmetic models in the HEADER shall contain a TABLE.

Describe algorithm to identify correct table entry.

Refer to INTERPOLATION annotation.

Supplementary EQUATION is legal; this shall be used for interpolation or extrapolation of values out-of-range.

11.7.3 Arithmetic model with equation arithmetic body

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

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

A supplementary TABLE is legal; this shall be used as a 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:

8.	Electrical performance modeling	161
8.1	Overview of modeling keywords	161
8.1.1	Timing models	161
8.1.2	Analog models	163
8.1.3	Supplementary models	164
9.2	Arithmetic models in the context of layout	220

11.9 Arithmetic models for timing data

To fill in from ALF 2.0:

8.3	Specification of timing models	171
8.3.1	Template for timing measurements / constraints	171
8.3.2	Partially defined timing measurements and constraints	173
8.3.3	Template for same-pin timing measurements / constraints	173
8.3.4	Absolute and incremental evaluation of timing models	174
8.7.4	PIN-related timing models	184

11.9.1 TIME statement

To fill in from ALF 2.0:

8.3.6	TIME	176
8.13.1	TIME within the LIMIT construct	197
8.9.3	TIME to peak measurement	190
8.10	Waveform description.....	192
8.10.1	Principles	192
8.10.2	Annotations within a waveform	194

11.9.2 FREQUENCY statement

To fill in from ALF 2.0:

8.13.2	FREQUENCY within a LIMIT construct	198
8.9.2	TIME and FREQUENCY annotation.....	189

11.9.3 DELAY and RETAIN statements

To fill in from ALF 2.0:

8.3.7	DELAY	176
8.3.8	RETAIN	176

11.9.4 SLEWRATE statement

To fill in from ALF 2.0:

8.3.9	SLEWRATE.....	177
-------	---------------	-----

11.9.5 SETUP and HOLD statement

To fill in from ALF 2.0:

8.3.10	SETUP.....	177
8.3.11	HOLD	177

11.9.6 NOCHANGE statement

To fill in from ALF 2.0:

8.3.12	NOCHANGE.....	178
--------	---------------	-----

11.9.7 RECOVERY and REMOVAL statements

To fill in from ALF 2.0:

8.3.13	RECOVERY	178
8.3.14	REMOVAL	178

11.9.8 SKEW statement

To fill in from ALF 2.0:

8.3.15	SKEW between two signals	179
8.3.16	SKEW between multiple signals	179

11.9.9 PULSEWIDTH statement

To fill in from ALF 2.0:

8.3.17	PULSEWIDTH.....	180
--------	-----------------	-----

11.9.10 PERIOD statement

To fill in from ALF 2.0:

8.3.18	PERIOD.....	180
--------	-------------	-----

11.9.11 JITTER statement

To fill in from ALF 2.0:

8.3.19	JITTER	180
--------	--------------	-----

11.9.12 THRESHOLD statement

To fill in from ALF 2.0:

8.2.1	THRESHOLD definition.....	165
8.2.5	Context of THRESHOLD definitions	169

11.10 Auxiliary statements related to timing data

****Add lead-in text****

11.10.1 FROM and TO statements

To fill in from ALF 2.0:

8.2.2	FROM and TO container.....	166
-------	----------------------------	-----

A FROM or TO statement XXX, as shown in Syntax 93.

```

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

```

Syntax 93—FROM and TO statements

11.10.2 EARLY and LATE statements

To fill in from ALF 2.0:

8.5 EARLY and LATE container	181
------------------------------------	-----

An EARLY or LATE statement XXX, as shown in Syntax 94.

```
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
```

Syntax 94—EARLY and LATE statements

11.10.3 Annotations for arithmetic models for timing data

To fill in from ALF 2.0:

8.2 Auxiliary statements for timing models.....	165
8.2.3 PIN annotation.....	166
8.2.4 EDGE_NUMBER annotation	166

11.11 Arithmetic models for environmental data

To fill in from ALF 2.0:

8.6 Environmental dependency for electrical data.....	181
---	-----

11.11.1 PROCESS and DERATE_CASE statement

To fill in from ALF 2.0:

8.6.1 PROCESS.....	182
8.6.2 DERATE_CASE	182
8.6.3 Lookup table without interpolation	182
8.6.4 Lookup table for process- or derating-case coefficients	183

11.11.2 TEMPERATURE statement

To fill in from ALF 2.0:

8.6.5 TEMPERATURE	183
-------------------------	-----

11.12 Arithmetic models for electrical data

To fill in from ALF 2.0:

8.7	PIN-related arithmetic models for electrical data	183
8.7.1	Principles	183
8.7.2	CAPACITANCE, RESISTANCE, and INDUCTANCE	184
8.7.3	VOLTAGE and CURRENT	184
8.7.6	Context-specific semantics	185

11.12.1 CAPACITANCE statement

| **Add lead-in text**

11.12.2 RESISTANCE statement

| **Add lead-in text**

11.12.3 INDUCTANCE statement

| **Add lead-in text**

11.12.4 VOLTAGE statement

| **Add lead-in text**

11.12.5 CURRENT statement

| **Add lead-in text**

11.12.6 POWER and ENERGY statement

To fill in from ALF 2.0:

8.11	Arithmetic models for power calculation.....	194
8.11.1	Principles	194
8.11.2	POWER and ENERGY	195

11.12.7 FLUX and FLUENCE statement

To fill in from ALF 2.0:

8.12	Arithmetic models for hot electron calculation.....	196
8.12.1	Principles	196
8.12.2	FLUX and FLUENCE	196

11.12.8 DRIVE_STRENGTH statement

To fill in from ALF 2.0.2:

8.8	Other PIN-related arithmetic models	187
8.8.1	DRIVE_STRENGTH	187

11.12.9 SWITCHING_BITS statement

To fill in from ALF 2.0:

8.8.2	SWITCHING_BITS	188
-------	----------------------	-----

11.12.10 NOISE and NOISE_MARGIN statement

To fill in from ALF 2.0:

8.14 Noise calculation.....	201
8.14.1 NOISE_MARGIN definition	202
8.14.2 Representation of noise in a VECTOR	203
8.14.3 Context of NOISE_MARGIN	204
8.14.4 Noise propagation.....	206
8.14.5 Noise rejection.....	208

11.12.11 Annotations for arithmetic models for electrical data

To fill in from ALF 2.0:

8.9 Annotations for arithmetic models	188
8.9.1 MEASUREMENT annotation.....	189
8.9.4 Rules for combinations of annotations.....	192

11.13 Arithmetic models for physical data

| **Add lead-in text**

11.13.1 CONNECTIVITY statement

To fill in from ALF 2.0:

9.15 CONNECTIVITY statement	258
9.15.1 Definition.....	258
9.15.2 CONNECT_RULE annotation.....	259
9.15.3 CONNECTIVITY modeled with BETWEEN statement.....	259
9.15.4 CONNECTIVITY modeled as lookup TABLE	260

11.13.2 SIZE statement

| **Add lead-in text**

11.13.3 AREA statement

| **Add lead-in text**

11.13.4 WIDTH statement

| **Add lead-in text**

11.13.5 HEIGHT statement

| **Add lead-in text**

11.13.6 LENGTH statement

| **Add lead-in text**

11.13.7 DISTANCE statement

****Add lead-in text****

11.13.8 OVERHANG statement

****Add lead-in text****

11.13.9 PERIMETER statement

****Add lead-in text****

11.13.10 EXTENSION statement

****Add lead-in text****

11.13.11 THICKNESS statement

****Add lead-in text****

11.13.12 Annotations for arithmetic models for physical data

To fill in from ALF 2.0:

9.18 Physical annotations for arithmetic models	264
9.18.1 BETWEEN statement within DISTANCE, LENGTH	264
9.18.2 MEASUREMENT annotation for DISTANCE	265
9.18.3 REFERENCE annotation for DISTANCE	265
9.18.4 Reference to ANTENNA	266
9.18.5 Reference to PATTERN	267

11.14 Arithmetic submodels for timing and electrical data

****Add lead-in text****

11.14.1 RISE and FALL statement

To fill in from ALF 2.0:

8.3.5 RISE and FALL submodels	175
-------------------------------------	-----

11.14.2 HIGH and LOW statement

To fill in from ALF 2.0:

8.7.5 Submodels for RISE, FALL, HIGH, and LOW	184
---	-----

11.15 Arithmetic submodels for physical data

****Add lead-in text****

11.15.1 HORIZONTAL and VERTICAL statement

Add lead-in text

This is a single subheader

Annex A

(normative)

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. ****The current ordering is as each item appears in its subchapter****

A.1 Lexical definitions

any_character ::= (see 6.2.3)

reserved_character
| nonreserved_character
| escape_character
| whitespace

reserved_character ::= (see 6.2.3.1)

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

nonreserved_character ::= (see 6.2.3.2)

letter | digit | _ | \$ | #

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**

digit ::=

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

escape_character ::= (see 6.2.3.3)

delimiter ::= (see 6.3.1)

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

comment ::= (see 6.3.2)

single_line_comment
| block_comment

integer ::= (see 6.3.3)

[sign] unsigned

sign ::=

+ | -

unsigned ::=

digit { _ | digit }

non_negative_number ::=

unsigned [. unsigned]
| unsigned [. unsigned] **E** [sign] unsigned

number ::=

[sign] non_negative_number

bit_literal ::= (see 6.3.4)

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 ::=
    *
based_literal ::=                                     (see 6.3.5)
    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
edge_literal ::=                                     (see 6.3.6)
    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 ::=
    ?? | ?~ | ?! | ?-
quoted_string ::=                                     (see 6.3.7)
    " { any_character } "
identifiers ::=                                     (see 6.3.8)
    identifier { identifier }
identifier ::=
    nonescaped_identifier
    | escaped_identifier
    | placeholder_identifier
    | hierarchical_identifier
nonescaped_identifier ::=                             (see 6.3.8.1)
    nonreserved_character { nonreserved_character }

```

escaped_identifier ::= (see 6.3.8.2)
 escape_character escaped_characters
 escaped_characters ::=
 escaped_character { escaped_character }
 escaped_character ::=
 nonreserved_character
 | reserved_character
 | escape_character
 placeholder_identifier ::= (see 6.3.8.3)
 < nonescaped_identifier >
 hierarchical_identifier ::= (see 6.3.8.4)
 identifier . { identifier . } identifier
 arithmetic_values ::= (see 6.6.1)
 arithmetic_value { arithmetic_value }
 arithmetic_value ::=
 number
 | identifier
 | pin_value
 string_value ::= (see 6.6.2)
 quoted_string
 | identifier
 edge_values ::= (see 6.6.3)
 edge_value { edge_value }
 edge_value ::=
 (edge_literal)
 index_value ::= (see 6.6.4)
 unsigned
 | identifier

A.2 Auxiliary definitions

index ::= (see 7.1.1)
 [index_range]
 | [index_value]
 index_range ::= (see 7.1.2)
 index_value : index_value
 pin_assignments ::= (see 7.2.1)
 pin_assignment { pin_assignment }
 pin_assignment ::=
 pin_variable = pin_value ;
 pin_variables ::= (see 7.2.2)
 pin_variable { pin_variable }
 pin_variable ::=
 pin_variable_identifier [index]
 pin_values ::= (see 7.2.3)
 pin_value { pin_value }
 pin_value ::=
 pin_variable
 | bit_literal
 | based_literal
 | unsigned

```

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 }
annotation_values ::=
    annotation_value { annotation_value }
annotation_value ::=
    index_value
    | string_value
    | edge_value
    | pin_value
    | arithmetic_value
    | boolean_expression
    | control_expression
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

```

(see 7.3.1)

(see 7.3.2)

(see 7.4)

A.3 Generic definitions

include ::=	(see 8.1)
INCLUDE quoted_string ;	
alias ::=	(see 8.2)
ALIAS identifier = identifier ;	
constant ::=	(see 8.3)
CONSTANT identifier = arithmetic_value ;	
attribute ::=	(see 8.4)
ATTRIBUTE { identifiers }	
property ::=	(see 8.5)
PROPERTY [identifier] { one_level_annotations }	
class_declaration ::=	(see 8.6)
CLASS identifier ;	
CLASS identifier { all_purpose_items }	
keyword_declaration ::=	(see 8.7)
KEYWORD context_sensitive_keyword = syntax_item_identifier ;	
group_declaration ::=	(see 8.8)
GROUP group_identifier { annotation_values }	
GROUP group_identifier { index_value : index_value }	
template_declaration ::=	(see 8.9)
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

```

A.4 Library definitions

```

library ::= (see 9.1.1)
    LIBRARY library_identifier { library_items }
    | LIBRARY library_identifier ;
    | library_template_instantiation
library_items ::=
    library_item { library_item }
library_item ::=
    sublibrary
    | sublibrary_item
library ::=
    SUBLIBRARY sublibrary_identifier { sublibrary_items }
    | SUBLIBRARY sublibrary_identifier ;
    | sublibrary_template_instantiation
sublibrary_items ::= (see 9.1.2)
    sublibrary_item { sublibrary_item }
sublibrary_item ::=
    all_purpose_item
    | cell
    | primitive
    | wire
    | layer
    | via
    | rule
    | antenna

    | array
    | site
INFORMATION_two_level_annotation ::= (see 9.1.3)
    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

cell ::= (see 9.2.1)
    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

non_scan_cell ::= (see 9.2.2)
    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 }

pin ::= (see 9.3.1)
    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 }

range ::= (see 9.3.2)
    RANGE { index_range }

pin_group ::= (see 9.3.3)
    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
wire ::= (see 9.4.1)
    WIRE wire_identifier { wire_items }
    | WIRE wire_identifier ;
    | wire_template_instantiation
wire_items ::=
    wire_item { wire_item }
wire_item ::=
    all_purpose_item
    | node
node ::= (see 9.4.2)
    NODE node_identifier { node_items }
    | NODE node_identifier ;
    | node_template_instantiation
node_items ::=
    node_item { node_item }
node_item ::=
    all_purpose_item
vector ::= (see 9.5.1)
    VECTOR control_expression { vector_items }
    | VECTOR control_expression ;
    | vector_template_instantiation
vector_items ::=
    vector_item { vector_item }
vector_item ::=
    all_purpose_item
    | illegal
illegal ::= (see 9.5.2)
    ILLEGAL { illegal_items }
    | illegal_template_instantiation
illegal_items ::=
    illegal_item { illegal_item }
illegal_item ::=
    all_purpose_item
    | violation
layer ::= (see 9.6.1)
    LAYER layer_identifier { layer_items }
    | LAYER layer_identifier ;
    | layer_template_instantiation
layer_items ::=
    layer_item { layer_item }
layer_item ::=
    all_purpose_item
via ::= (see 9.7.1)
    VIA via_identifier { via_items }
    | VIA via_identifier ;
    | via_template_instantiation
via_items ::=
    via_item { via_item }

```

```

via_item ::=
    all_purpose_item
    | pattern
    | artwork
via_reference ::= (see 9.7.3)
    VIA { via_instantiations }
    | VIA { via_identifiers }
via_instantiations ::=
    via_instantiation { via_instantiation }
via_instantiation ::=
    via_identifier { geometric_transformations }
rule ::= (see 9.8.1)
    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
antenna ::= (see 9.8.2)
    ANTENNA antenna_identifier { antenna_items }
    | ANTENNA antenna_identifier ;
    | antenna_template_instantiation
antenna_items ::=
    antenna_item { antenna_item }
antenna_item ::=
    all_purpose_item
blockage ::= (see 9.8.3)
    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
port ::= (see 9.8.4)
    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

```

`site ::=` (see 9.9.1)
 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*
`array ::=` (see 9.9.2)
 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*
`pattern ::=` (see 9.9.3)
 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*
`artwork ::=` (see 9.9.4)
 ARTWORK = *artwork_identifier* { *artwork_items* }
 | **ARTWORK** = *artwork_identifier* ;
 | *artwork_template_instantiation*
`artwork_items ::=`
 artwork_item { *artwork_item* }
`artwork_item ::=`
 geometric_transformation
 | *pin_assignment*
`geometric_model ::=` (see 9.9.5)
 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 } }

geometric_transformations ::=
    geometric_transformation { geometric_transformation }
    (see 9.9.6)

geometric_transformation ::=
    SHIFT_two_level_annotation
    | ROTATE_one_level_annotation
    | FLIP_one_level_annotation
    | repeat

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

function ::=
    FUNCTION { function_items }
    | function_template_instantiation
    (see 9.10.1)

function_items ::=
    function_item { function_item }

function_item ::=
    all_purpose_item
    | behavior
    | structure
    | statetable

test ::=
    TEST { test_items }
    | test_template_instantiation
    (see 9.10.2)

test_items ::=
    test_item { test_item }

test_item ::=
    all_purpose_item
    | behavior
    | statetable

behavior ::=
    BEHAVIOR { behavior_items }
    | behavior_template_instantiation
    (see 9.10.3)

behavior_items ::=
    behavior_item { behavior_item }

behavior_item ::=
    boolean_assignments
    | 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 } }

```

```

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 }
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
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 )
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

```

(see 9.10.4)

(see 9.10.5)

(see 9.10.6)

(see 9.10.7)

A.5 Control definitions

boolean_expression ::= (see 10.1)

```
( 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 ::=

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

vector_expression ::= (see 10.2)

```
( 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
```

$$\text{control_and} ::= \& \mid \&\&$$

```

    | assignment_arithmetic_model
    | arithmetic_model_template_instantiation
partial_arithmetic_model ::= (see 11.2.2)
    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
non_trivial_arithmetic_model ::= (see 11.2.3)
    nonescaped_identifier [ arithmetic_model_identifier ] {
        [ any_arithmetic_model_items ]
        arithmetic_body
        [ any_arithmetic_model_items ]
    }
trivial_arithmetic_model ::= (see 11.2.4)
    nonescaped_identifier [ arithmetic_model_identifier ] = arithmetic_value ;
    | nonescaped_identifier [ arithmetic_model_identifier ] = arithmetic_value
      { any_arithmetic_model_items }
assignment_arithmetic_model ::= (see 11.2.5)
    arithmetic_model_identifier = arithmetic_expression ;
any_arithmetic_model_items ::= (see 11.2.6)
    any_arithmetic_model_item { any_arithmetic_model_item }
any_arithmetic_model_item ::=
    all_purpose_item
    | from
    | to
    | violation
arithmetic_submodels ::= (see 11.3.1)
    arithmetic_submodel { arithmetic_submodel }
arithmetic_submodel ::=
    non_trivial_arithmetic_submodel
    | trivial_arithmetic_submodel
    | arithmetic_submodel_template_instantiation
non_trivial_arithmetic_submodel ::= (see 11.3.2)
    nonescaped_identifier {
        [ any_arithmetic_submodel_items ]
        arithmetic_body
        [ any_arithmetic_submodel_items ]
    }
trivial_arithmetic_submodel ::= (see 11.3.3)
    nonescaped_identifier = arithmetic_value ;
    | nonescaped_identifier = arithmetic_value { any_arithmetic_submodel_items }
any_arithmetic_submodel_items ::= (see 11.3.4)
    any_arithmetic_submodel_item { any_arithmetic_submodel_item }
any_arithmetic_submodel_item ::=
    all_purpose_item
    | violation
arithmetic_body ::= (see 11.4.1)
    arithmetic_submodels
    | table_arithmetic_body
    | equation_arithmetic_body
table_arithmetic_body ::=
    header table [ equation ]

```

```

equation_arithmetic_body ::=
    [ header ] equation [ table ]
header ::= (see 11.4.2)
    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
table ::= (see 11.4.3)
    TABLE { arithmetic_values }
    | table_template_instantiation
equation ::= (see 11.4.4)
    EQUATION { arithmetic_expression }
    | equation_template_instantiation
arithmetic_model_container ::= (see 11.5)
    arithmetic_model_container_identifier { arithmetic_models }
from ::= (see 11.10.1)
    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
EARLY_arithmetic_model_container ::= (see 11.10.2)
    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

```

Annex B

(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

Symbols

?- 20, 72
?! 20, 72
?? 20, 72
?~ 20, 72

A

abs 55, 84
alias 31, 75
all_purpose_items 30, 74
alphabetic_bit_literal 19, 72
annotation
 CELL
 NON_SCAN_CELL 37, 77
any_character 16, 71
arithmetic_binary_operator 55, 84
arithmetic_expression 55, 84
arithmetic_function_operator 55, 84
arithmetic_unary_operator 55, 84
attribute 32, 75

B

based literal 19
based_literal 20, 72
behavior 49, 81
behavior_body 49, 81
binary 19
binary_base 20, 72
binary_digit 20, 72
bit 19
bit_edge_literal 20, 72
bit_literal 19, 71
block comment 18
boolean_binary_operator 52, 83
boolean_expression 52, 83
boolean_unary_operator 52, 83

C

case-insensitive language 17
cell 36, 77
cell_identifier 36, 37, 77
cell_items 36, 77
cell_template_instantiation 36, 77

characterization 5
class 32, 75
combinational_assignments 49, 81
comment 17
 block 18
 long 18
 short 18
 single-line 18
comments
 nested 18
compound operators 17
constant 31, 75
constant numbers 18
context-sensitive keyword 23

D

decimal 19
decimal_base 20, 72
deep submicron 5
delimiter 17, 71
digit 20, 72

E

edge literal 20
edge_literal 20, 72
edge_literals 25, 73
equation 60, 86
equation_template_instantiation 60, 86
escape codes 21
escape_character 17, 71
escaped identifier 22
escaped_identifier 22, 73
exp 55, 84

F

function 48, 81
function_template_instantiation 48, 81
functional model 5

G

group 33, 75
group_identifier 33, 75

H

hard keyword 23
header 60, 86
header_template_instantiation 60, 86
hex_base 20, 72
hex_digit 20, 72
hexadecimal 19

I

identifier 17
Identifiers 21
identifiers 21, 72
include 31, 75
index 27, 73
integer 18, 71

K

Keywords
 context-sensitive 24
 generic objects 23
 operators 23

L

Library creation 1
library_items 35, 76
library_template_instantiation 35, 76
literal 17
log 55, 84
logic_values 50, 82
logic_variables 28, 73

M

max 55, 84
min 55, 84
mode of operation 5

N

nested comments 18
non_negative_number 18, 71
non-escaped identifier 21
nonescaped_identifier 22, 72
nonreserved_character 17, 71
Number 18
number 18, 71
numeric_bit_literal 19, 72

O

objects 34, 75
octal 19
octal_base 20, 72
octal_digit 20, 72
operation mode 5

P

pin_assignments 27, 73
pin_identifier 38, 77
pin_items 38, 77
pin_template_instantiation 38, 77
placeholder identifier 22
placeholder_identifier 21
power constraint 5
Power model 5
primitive_identifier 49, 50, 81, 82
primitive_instantiation 49, 81
primitive_items 50, 82
primitive_template_instantiation 50, 82
private keywords 24
property 32, 75
public keywords 24

Q

quoted string 16, 20
quoted_string 20, 72

R

real 18
reserved keyword 23
reserved_character 16, 71
RTL 4

S

sequential_assignment 49, 81
sign 18, 71
simulation model 5
single-line comment 18
soft keyword 23
statetable 50, 82
statetable_body 50, 82
string 25, 73
symbolic_edge_literal 20, 72

T

- table 60, 86
- table_template_instantiation 60, 86
- template 34, 75
- template_identifier 34, 75
- template_instantiation 34, 76
- timing constraints 5
- timing models 5

U

- unnamed_assignment 29, 74
- unsigned 18, 71

V

- vector 40, 78
- vector_expression 40, 53, 78, 83
- vector_items 40, 78
- vector_template_instantiation 40, 78
- vector_unary_operator 53, 83
- vector-based modeling 5
- Verilog 4
- VHDL 4

W

- whitespace 16, 71
- whitespace characters 16
- wildcard_literal 19, 72
- wire 39, 40, 41, 42, 43, 44, 45, 46, 78, 79, 80
- wire_identifier 39, 40, 41, 42, 43, 45, 78, 79, 80
- wire_items 39, 40, 78
- wire_template_instantiation 39, 40, 41, 42, 43, 44, 45, 46, 78, 79, 80
- word_edge_literal 20, 72

