Advanced Library Format
for
ASIC Cells & Blocks

containing
Power, Timing, Functional and Physical Information
for
Synthesis, Analysis and Test

Version 1.0

November 14, 1997

Open Verilog International

Additional copies of this manual may be purchased by contacting Open Verilog International at the address shown below.

Notices

The information contained in this draft manual represents the definition of the Advanced Library Format (ALF) as proposed by OVI (PS- TSC) as of November 1997. Open Verilog International makes no warranties whatsoever with respect to the completeness, accuracy, or applicability of the information in this draft manual to a user's requirements. This format is not yet fully defined and is subject to change. It is suitable for learning how to create ASIC Cell models that contain power, timing, functional and physical information for synthesis, analysis and test, and as a vehicle for providing feedback to the standards committee. ALF should not be used for production design and development.

Open Verilog International reserves the right to make changes to the ALF language and this manual at any time without notice.

Open Verilog International does not endorse any particular simulator or other CAE tool that is based on the Advanced Library Format.

Suggestions for improvements to the Advanced Library Format and/or to this manual are welcome. They should be sent to the address below.

Information about Open Verilog International and membership enrollment can be obtained by inquiring at the address below.

Published as: Advanced Library Format (ALF) Reference Manual
Version 1.0, November 1997.

Published by: Open Verilog International
15466 Los Gatos Blvd., #109071
Los Gatos, CA 95032
Phone: (408) 358-9510
Fax: (408) 358-3910

Printed in the United States of America.

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

Wolfgang Roethig, PhD	LSI Logic	Chairman
Amir Zarkesh, PhD	Viewlogic / Quad	Co-Chairman
Mike Andrews	Mentor Graphics	
Tim Ayres	Viewlogic / Sunrise	
Victor Berman	VI / IEEE	
Dennis Brophy	Mentor Graphics / IEEE DPC	
Renlin Chang	Cadence	
Sanjay Churiwala	Cadworx	
Jose De Castro	Mentor Graphics	
Timothy Ehrler	VLSI Technology	
Paul Foster	Avant! Corporation	
Vassilios Gerousis, PhD	Motorola / OVI Board	
Kevin Grotjohn	LSI Logic	
Johnson Chan Limqueco	Ambit Design Systems	
Ta-Yung Liu	Avant! Corporation	
Larry Rosenberg	Cadence/VSI	
Hamid Rahmanian	Mentor Graphics	
Ambar Sarkar, PhD	Viewlogic	
Itzhak Shapira	Cadence	
Yatin Trivedi	Seva Technologies	Technical Editor
Devadas Varma	Ambit Design Systems	
David Wallace	Mentor Graphics / Exemplar	
Frank Weiler	Avant! Corporation / OVI Board	

Revision history:

1st draft:	11/20/96
2nd draft:	12/20/96
3rd draft:	3/22/97
4th draft:	3/31/97
5th draft:	4/22/97
6th draft:	6/1/97
7th draft:	6/25/97
8th draft:	8/13/97
9th draft:	10/14/97
Version 1.0	11/14/97

Table of Contents

1 Introduction. 1-1

1.1 Motivation 1-1
1.2 Goals 1-1
1.3 Target Applications 1-2
1.4 Conventions 1-5
1.5 Organization of this manual 1-5

2 Characterization and Modeling. 2-1

2.1 Basic Concepts 2-1
2.2 Functional Modeling 2-2
2.2.1 Combinational Logic 2-2
2.2.2 Level Sensitive Sequential Logic 2-2
2.2.3 Edge Sensitive Sequential Logic 2-2
2.2.4 Vector-Sensitive Sequential Logic 2-4
2.3 Performance Modeling 2-5
2.3.1 Timing Modeling 2-5
2.3.2 Power Modeling 2-7
2.4 Physical modeling for synthesis and test 2-8
2.4.1 Cell modeling 2-8
2.4.2 Wire modeling 2-9

3 Library Format Specification 3-1

3.1 Object Model 3-1
3.1.1 Generic Objects 3-2
3.1.2 Library-specific objects 3-4
3.1.3 Arithmetic models 3-4
3.1.4 Functions 3-5
3.2 Lexical rules 3-7
3.2.1 Character set 3-7
3.2.2 Lexical tokens 3-7
3.2.3 Whitespace Characters 3-8
3.2.4 Reserved and Nonreserved Characters 3-8
3.2.5 Delimiters 3-8
3.2.6 Comments 3-9
3.2.7 Number Literals 3-9
3.2.8 Boolean Literals 3-9

3.2.9 Edge Literals 3-10
3.2.10 Quoted String 3-11
3.2.11 Identifiers 3-11
3.3 Keywords 3-12
3.3.1 Keywords for generic object types 3-12
3.3.2 Keywords for Operators 3-12
3.3.3 Context-Sensitive Keywords 3-13
3.4 Syntax Rules 3-13
3.4.1 Assignments 3-13
3.4.2 Expressions 3-14
3.4.3 Instantiations 3-15
3.4.4 Literals 3-16
3.4.5 Operators 3-17
3.4.6 Auxiliary Objects 3-18
3.4.7 Generic Objects 3-19
3.4.8 Cell Object 3-19
3.4.9 Library Object 3-20
3.4.10 Pin Object 3-20
3.4.11 Primitive Object 3-20
3.4.12 Sublibrary Object 3-20
3.4.13 Vector Object 3-21
3.4.14 Wire Object 3-21
3.4.15 Arithmetic Model 3-22
3.4.16 Function 3-23
3.5 Operators 3-23
3.5.1 Arithmetic operators 3-24
3.5.2 Boolean operators on scalars 3-24
3.5.3 Boolean operators on words 3-25
3.5.4 Vector operators 3-25
3.5.5 Operator priorities 3-27
3.6 Context-sensitive keywords 3-27
3.6.1 Annotation containers 3-28
3.6.2 Keywords for referencing objects used as annotation 3-30
3.6.3 Annotations for a PIN object 3-30
3.6.4 Annotations for a VECTOR object 3-33
3.6.5 Annotations for a CELL object 3-33
3.6.6 Attributes 3-34
3.6.7 Annotations for arithmetic models 3-35
3.6.8 Keywords for arithmetic models 3-36
3.7 Library Organization 3-38

3.7.1 Scoping Rules 3-38
3.7.2 Use of multiple files 3-39
3.8 Referenceable objects 3-39
3.8.1 Referencing PRIMITIVEs or CELLs 3-40
3.8.2 Referencing PINs in FUNCTIONs 3-41
3.8.3 Referencing PINs in VECTORs 3-43
3.8.4 Referencing arithmetic models 3-43
3.9 Functional modeling styles and rules 3-45
3.9.1 Rules for combinational functions 3-45
3.9.2 Rules for sequential functions 3-46
3.10 Predefined primitives 3-47
3.10.1 Combinational primitives 3-48
3.10.2 Tristate Primitives 3-54
3.10.3 Multiplexor 3-56
3.10.4 Flipflop 3-56
3.10.5 Latch 3-58

4 Applications 4-1

4.1 Truth Table vs Boolean Equation 4-1
4.1.1 NAND gate 4-1
4.1.2 Flipflop 4-2
4.2 Use of primitives 4-2
4.2.1 D-Flipflop with asynchronous clear 4-2
4.2.2 JK-flipflop 4-3
4.2.3 D-Flipflop with synchronous load and clear 4-3
4.2.4 D-Flipflop with input multiplexor 4-4
4.2.5 D-latch 4-5
4.2.6 SR-latch 4-5
4.2.7 JTAG BSR 4-6
4.2.8 Combinational Scan Cell 4-6
4.2.9 Scan Flipflop 4-7
4.2.10 Quad D-Flipflop 4-8
4.3 Templates and vector-specific models 4-9
4.3.1 Vector specific delay and power Tables 4-9
4.3.2 Use of TEMPLATE 4-11
4.3.3 Vector description styles for timing arcs 4-14
4.3.4 Vectors for delay, power and timing constraints 4-15
4.4 Combining tables and equations 4-16
4.4.1 Table vs equation 4-16
4.4.2 Cell with Multiple Output Pins 4-17

4.4.3 PVT Derating 4-19
4.5 Use of Annotations 4-21
4.5.1 Annotations for a PIN 4-21
4.5.2 Annotations for a timing arc 4-22
4.5.3 Creating Self-explaining Annotations 4-23
4.6 Providing fallback position for applications 4-23
4.6.1 Use of DEFAULT 4-23
4.7 Bus Modeling 4-25
4.7.1 Tristate Driver 4-25
4.7.2 Bus with multiple drivers 4-27
4.7.3 Busholder 4-28
4.8 Wire models 4-28
4.8.1 Basic Wire Model 4-28
4.8.2 Wire select model 4-29
4.9 Megacell Modeling 4-30
4.9.1 Expansion of Timing Arcs 4-30
4.9.2 Two-port memory 4-31
4.9.3 Three-port memory 4-33
4.10 Special cells 4-34
4.10.1 Pulse generator 4-34
4.10.2 VCO 4-34
4.11 Core Modeling 4-35
4.11.1 Digital Filter 4-35
4.12 Connectivity 4-36
4.12.1 External connections between pins of a cell 4-37
4.12.2 Allowed connections for classes of pins 4-37

Section 1
Introduction

1.1 Motivation

Design of 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 analysis, and power analysis relies significantly on the quality of available information about the cells in the technology library.

New challenges in the design flow, e.g. power analysis, 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 may prevent designers from choosing an application program which is best suited for meeting specific design challenges. Similar considerations may inhibit the development and productization of such an application program altogether. As a result, competitiveness and innovation of the whole electronic industry may stagnate.

In order to remove these constraints, an industry-wide standard for library format, Advanced Library Format (ALF), is proposed. It enables the EDA industry to develop innovative products and the ASIC designers to chose 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 must be easy to understand and not become a cumbersome process only known by a few experts.
- *generality* - tools of any level of sophistication must be able to retrieve necessary information from the library.
- *expandability* - for early adoption and future enhancement possibilities

- *flexibility* - the choice of keeping information in one library or in separate libraries must be in the hand of the user; it should not be dictated by the standard.
- *efficiency* - the complexity of the design information requires that the process of retrieving information from the library does not become a bottleneck. The right trade-off between compactness and verbosity must be found.
- *ease of implementation* - backward compatibility with existing libraries must be provided, and translation to the new library must be an easy task.
- *conciseness* - unambiguous description and accuracy of contents
- *acceptance* - 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 3rd 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 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, 4-bit synchronous up-down counter) 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 not infinitely robust against noise.

Table 1-1 shows a list of applications used in ASIC design flow and their relationship to ALF. The boundary between supported and not supported applications can be defined by the *physical* information provided by ALF. Information needed for area and performance estimation and

optimization, notably by synthesis tools, is provided by ALF. On the other hand, layout information is only considered for front end application, such as RTL floorplanner.

Table 1-1 Target applications and models supported by ALF

application	functional model	performance model	physical model
timing analysis	supported by ALF	supported by ALF	N/A
power analysis	supported by ALF	supported by ALF	N/A
simulation	derived from ALF	derived from ALF	N/A
synthesis	supported by ALF	supported by ALF	supported by ALF
scan insertion	supported by ALF	N/A	N/A
RTL floorplanner	N/A	N/A	planned for ALF
signal integrity	N/A	planned for ALF	planned for ALF
layout	N/A	N/A	not 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.

The figure 1-1 shows how ALF provides information to various design tools.

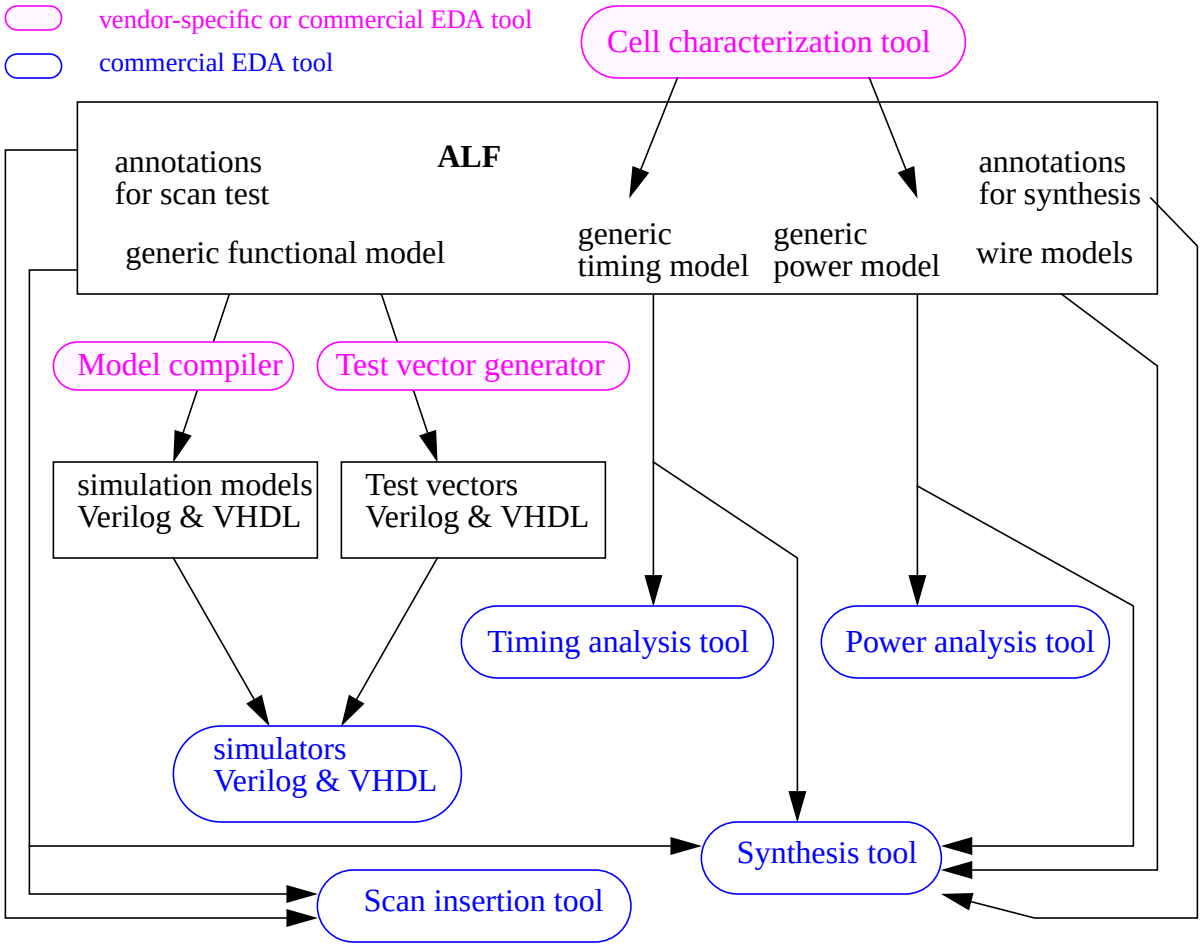

Figure 1-1: ALF and its target applications

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, 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 needs 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, not only the cost constraints but also the set of physical constraints under which the design will function or not have increased dramatically. Therefore the requirements for detailed characterization and analysis of those constraints, especially timing and power in deep submicron design, are much more sophisticated than it used to be. 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 following conventions.

`::=`	`definition of a syntax rule`
`|`	`alternative definition`
`[item]`	`an optional item`
`{item}`	`optional item which can be repeated`
`item`	`item in boldface font is taken as is`
`item`	`item in italic is for explanation purpose only`

1.5 Organization of this manual

This document presents the Advanced Library Format (ALF), a new standard library format for ASIC cells, blocks and cores, containing power, timing, functional, and physical information.

In the first chapter, motivation and goals of ALF are defined.

The second chapter describes the basics of functional modeling, cell characterization for timing and power, and additional modeling features for synthesis and test.

The third chapter is the Language Reference Manual (LRM).

The fourth chapter provides illustrative examples.

Section 2
Characterization and Modeling

This chapter elaborates on the basics of cell modeling and characterization, which is the primary source of library information.

2.1 Basic Concepts

The functional models within an ASIC library describe functions and algorithms of hardware components, as opposed to synthesizable functions or algorithms. The functional modeling language for the ASIC library is designed to make the description of existing hardware easy and efficient. The scope here is different from a hardware description language (HDL) or a programming language designed to specify functionality without other aspects of hardware implementation.

Functional description provides boolean functions or truth tables, including state variables for sequential logic. Boolean and arithmetic operators for scalars and vectors are also provided. Combinational and sequential logic cells, macrocells (e.g. adders, multipliers, comparators), and atomic megacells (e.g. memories) can be modeled with these capabilities.

Vectors describe the stimuli for characterization. This encompasses both the concept of timing arcs and logical conditions. An exhaustive set of vectors can be generated from functional information, although the complexity of the exhaustive set precludes it from practical usage. The characterizer makes a choice of the relevant subset for characterization.

Power characterization is a superset of timing characterization using the same set and range of characterization variables: load, input slew rate, skew between multiple switching inputs, voltage, temperature. Characterization measurements, such as delay, output slew rate, average current in time window, bounds of allowed skew for timing constraints, etc. can be described as functions of the characterization variables, either by equations or using lookup tables. More complicated calculation algorithms cannot be described explicitly in the library, but can be referenced using templates.

A core is not an atomic megacell, since it can be split up into smaller components. Templates provide the capability of defining and reusing blocks consisting of atomic constructs or of other blocks. Thus a hierarchical description of the complete core can be created in a simple and efficient way.

Abstraction is required for the characterization of megacells: vectors describe events on buses rather than on pins; number and range of switching pins within a bus become additional characterization variables. Characterization measurements are expandable and can be extrapolated from pin to bus.

2.2 Functional Modeling

2.2.1 Combinational Logic

Combinational logic can be described by continuous assignments of boolean values (True, False) to output variables as a function of boolean values of input variables. Such functions can be expressed in either equation format or table format[1].

Let us consider an arbitrary continuous assignment

$$z = f(a_1 .., .. a_n)$$

In a dynamic or simulation context, the left-hand size (LHS) variable *z* is evaluated whenever there is a change in one of the right-hand side (RHS) variables a_i. No storage of previous states is needed for dynamic simulation of combinational logic.

2.2.2 Level Sensitive Sequential Logic

In sequential logic, an output variable z_j can also be a function of itself, i.e. of its previous state. The sequential assignment has the form

$$z_j = f(a_1 .., .. a_n , z_1 .., .. z_m)$$

The RHS cannot be evaluated continuously, since a change in the LHS as a result of a RHS evaluation will trigger a new RHS evaluation repeatedly, unless the variables attain stable values. Modeling capabilities of sequential logic with continuous assignments would be restricted to systems with oscillating or self-stabilizing behavior.

However, if we introduce the concept of triggering conditions for the LHS, we have everything we need for modeling *normal* sequential logic. The expression of a triggered assignment can look like this:

$$@\ g(b_1 .., .. b_k)\ z_j = f(a_1 .., .. a_n , z_1 .., .. z_m)$$

The evaluation of *f* is activated whenever the *triggering function* g is true. The evaluation of g is self-triggered, i.e. at each time when an argument of g changes its value. If *g* is a boolean expression like *f*, we can model all types of *level-sensitive sequential logic*.

During the time when *g* is true, the logic cell behaves exactly like combinational logic. During the time when *g* is false, the logic cell holds its value. Hence one memory element per state bit is needed.

2.2.3 Edge Sensitive Sequential Logic

In order to model *edge-sensitive sequential logic*, we need to introduce notations for logical transitions in addition to logical states.

If the triggering function *g* is sensitive to logical transitions rather than to logical states, the function *g* evaluates to true only for an infinitely small time, exactly at the moment when the

1. Rather than defining a new syntax for boolean equations, we are just adopting existing notations people are familiar with. Those notations can already be found in the ANSI C standard, and they are widely used in popular script languages such as PERL as well as in HDLs like VERILOG.

transition happens. The sole purpose of *g* is to trigger an assignment to the output variable through evaluation of the function *f* exactly at this time.

Edge-sensitive logic requires storage of the previous output state and the input state (to detect a transition). In fact, all implementations of edge-triggered flipflops require at least two storage elements. For instance, the most popular flipflop architecture features a master latch driving a slave latch.

Using transitions in the triggering function for value assignment, the functionality of a positive edge triggered flipflop can be described as follows in ALF:

```
@ (01 CP) {Q = D;}
```

which reads “at rising edge of CP, assign Q the value of D”.

If the flipflop also has an asynchronous direct clear pin (CD), the functional description consists of two assignments:

```
@ (01 CP && CD) {Q = D;}
@ (!CD) {Q = 0;}
```

Figure 2-1: Model of a flipflop with asynchronous clear in ALF

The following two examples show an equivalent model in Verilog and VHDL:

```
reg Q;

always @(posedge CP or negedge CD)
begin
   if (!CD)
      Q = 0;
   else
      Q = D;
end
```

Figure 2-2: Model of a flipflop with asynchronous clear in Verilog

```
process (CP, CD)
begin
   if (CD = ‘0’) then
      Q <= 0;
   elsif (CP = ‘1’ and CP’event) then
      Q <= D;
   end if;
end process;
```

Figure 2-3: Model of a flipflop with asynchronous clear in Verilog

ALF provides a formal, compact and self-explaining functional description of a flipflop[2].

2.2.4 Vector-Sensitive Sequential Logic

In order to model generalized higher order sequential logic, the concept of vector expressions is introduced, an extension of the boolean expressions.

A vector expression describes sequences of logical events or transitions in addition to static logical states. A vector expression represents a description of a logical stimulus without timescale. It describes the order of occurrence of events.

Using the -> operator *(followed by* operator), we have a general capability of describing a sequence of events or a vector. For example, consider the following vector expression:

```
01 A -> 01 B
```

which reads “rising edge on A is followed by rising edge on B”.

A vector expression is evaluated by an event sequence detection function. Like a single event or a transition, this function evaluates true only at an infinitely short time when the event sequence is detected.

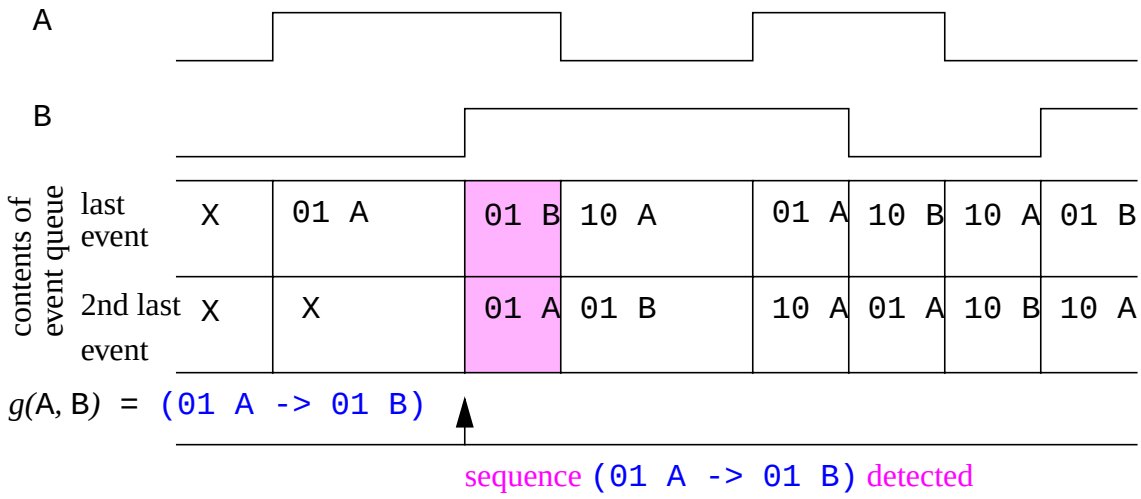

Figure 2-4: Example of event sequence detection function

The event sequence detection mechanism can be described as a queue that sorts events according to their order of arrival. The event sequence detection function evaluates true at exactly the time when a new event enters the queue and forms the required sequence with its preceding events.

---

2. The ALF description has the advantage, that one can see directly the level-sensitivity of the CD pin and the edge sensitivity of the CP pin. In Verilog or VHDL description, this is not so obvious. This makes it more difficult to see the kind of timing constraint. A complete simulation model needs to include the timing constraint checking mechanism, therefore the primary description which contains only functionality, needs to be transformed into a complete simulation model anyway. Choosing ALF as the primary functional description language avoids a bias in either VHDL or Verilog direction.

A vector-sensitive sequential logic can be called *(N+1) order sequential logic*, where N is the number of events to be stored in the queue. The implementation of (N+1) order sequential logic requires N memory elements for the event queue and 1 memory element for the output itself.

A sequence of event can also be gated with static logical conditions. For example,

```
(01 CP -> 10 CP) && CD
```

the pin CD must have state 1 from some time before the rising edge at CP to some time after the falling edge of CP. The pin CD can not go low (state 0) after the rising edge of CP and go high again before the falling edge of CP because this would insert events into the queue, and the sequence “rising edge on CP followed by falling edge on CP” would not be detected.

The formal calculation rules for general vector expressions featuring both states and transitions will be introduced in Section 3.5.4.

The concept of vector expression supports functional modeling of devices featuring digital communication protocols with arbitrary complexity.

2.3 Performance Modeling

2.3.1 Timing Modeling

The timing models of cells consists of two types: *delay models* for combinational and sequential cells, and *timing constraint models* for sequential cells. Both types can be described by timing arcs. A timing arc is a sequence of two events which can be described by a vector expression “event *e1* is followed by event *e2*”.

For example, a particular input to output delay of an inverting logic cell is identified by the following timing arc:

```
01 A -> 10 Z
```

which reads “rising edge on input A is followed by falling edge on output Z”.

A setup constraint between data and clock input of a positive edge triggered flipflop is identified by the following timing arc:

```
01 D -> 01 CP
```

which reads “rising edge on input D is followed by rising edge on input CP”.

A crucial part in ASIC cell development is to characterize a model which describes the behavior of each timing arc with sufficient accuracy in order to guarantee correct functional behavior under all required operational conditions.

A delay model usually needs two output variables:

- *intrinsic delay*, measured between a well-defined threshold value of the input signal and a well-defined threshold value of the output signal
- *transition delay*, measured between two well-defined threshold values of the output signal. Hence the transition delay is a fraction of the total output transition time, also called *slew rate* or *edge rate*.

A timing constraint model needs just one output variable:

- A timing constraint is the *minimum or maximum allowed elapsed time* between two signals, measured between well-defined threshold values between those two signals. This definition is similar to the intrinsic delay, except there is no input-output relationship between the two signals. Both signals are usually inputs to the cell.

The actual values of transition times and load capacitances seen by each pin of a cell instance are calculated by a delay predictor. Delay prediction can be separated into two tasks:

1. Acquisition of information on pin capacitance, extracted or estimated layout parasitics for each net and fitting those into the load characterization model (lumped C, R, etc.)
2. Calculation of internal signal transition times based on the extracted internal load and on load and transition times at the boundaries of the system.

Lookup tables provide a general modeling capability without precluding any level of accuracy.

Equations may feature polynomial expressions, exponentials and logarithms, and arbitrary transcendent functions. For practical purpose, only the four basic arithmetic operations (+, -, *, /) and exponentiation and logarithm will be supported for standard models.

Some models may require transcendent functions or complicated algorithms that cannot be expressed directly in equations. Other models and algorithms may need protection from being visible. In order to address needs that go beyond standard modeling features, a template-reference scheme is proposed: Any model which is neither in table nor in equation format needs to be a pointer to a customer-defined model which may reside outside the library.

Table 2-1 Modeling choices for cell characterization library

type of model	features	purpose
table	discrete points, multidimensional	direct storage of characterization data, direct accuracy control through mesh granularity
equation	expressions with +, -, *, /, exponent, logarithm	analytical model, well-suited for optimization purpose, more compact than table, also usable for arithmetic operations on tabulated data (scale, add, subtract ..)
reference	pointer to any type of model	reuse of predefined model (which may be table or equation), protection of user-defined model

Regardless of which type of model is chosen, there is a need to specify explicitly the meaning of the variables and the units. The specification of variables and units can be made outside the model and independent of the chosen model.

Since the set of variables should not be restrictive in order to allow any enhancements (e.g. move from a lumped capacitance to an RC model), *context-sensitive keywords* are proposed (e.g. “load”, “slewrate”). The application parser need not know the meaning of the context-

sensitive keyword, except that it is used as a variable in a model and that it has some unit attached to it, e.g. picofarad, nanosecond etc.

2.3.2 Power Modeling

A power model is an extension of the delay model for each timing arc using a third variable:

- *scaled average current*, measured by integrating and scaling the total transient current through the power supply of the cell for the specific timing arc or vector.
  The current measurement can start anytime before the first event of the vector starts and can end anytime after all transients of the vector have stabilized.

Variants of this model are scaled average power and energy, which are obtained by simple scaling of average current measurements:

```
power = current * Vdd
energy = current * Vdd * integration time
```

The set of vectors causing power consumption within a cell is a superset of vectors causing the cell output to switch. While only the latter are needed for delay characterization, more vectors are needed for accurate power characterization.

For example, consider a flipflop, which consumes power at every edge of the clock, even if the output does not switch. The vectors for delay and power characterization can be described as follows:

```
01 CP -> 01 Q
01 CP -> 10 Q
```

The vectors for power characterization with only clock-switching can be described as follows:

```
01 CP && Q==D
10 CP && Q==D
```

The D input having the same value as the Q output is a necessary and sufficient condition that the output will not switch at the rising edge of CP and that the value transferred to the master latch at the falling edge of CP will be the same as already stored. Hence those two vectors capture the actual power dissipation only within the clock buffers. Additional power vectors can be defined to capture the power dissipation within the data buffers and the master latch etc.

For a 2-input AND gate, if the event `01 A` is detected and then the event `10 B` is detected before the input-to-output delay elapses, a *glitch* is observed. It is possible to describe the glitch by a higher-order vector.

In dynamic simulation with *transport delay mode*, the glitch would appear as follows:

```
01 A -> 10 B -> 01 Z -> 10 Z
```

Simulation featuring *transport delay mode with invalid-value-detection* would exhibit the glitch as follows:

```
01 A -> 10 B -> 0X Z -> X0 Z
```

Simulation with *inertial delay mode* would suppress the output transitions:

```
(01 A -> 10 B) && !Z
```

The last expression can be used for each of the three modes, since `!Z` is always true at the time when the sequence `01 A -> 10 B` is detected.

Each way of expressing vectors can be derived from the cell functionality. The different examples for delay vectors (i.e. timing arcs), power vectors, and glitch vectors emphasize the rich potential of modeling capabilities using vector expressions.

State-dependent *static power* is also within the scope of vector-based power models. Static power consumption is activated in the same way as level-sensitive logic in functional modeling by a vector expression featuring steady states, whereas *transient power* consumption is activated similar to edge-sensitive logic by a vector expression featuring transitions.

The advantages of adding power models within each delay vector and providing extra power vectors are the following:

- straightforward extension of delay characterization
- capable of yielding the most detailed and accurate model on gate-level
- each vector defines a comprehensive stimulus for power measurements

The vector-based power model is a general and reliable model. However, the pin-toggle power model can be expressed as a special case of the vector-based model. This model has its justification in applications working at a high level of abstraction, where accurate characterization is not a predominant requirement.

2.4 Physical modeling for synthesis and test

2.4.1 Cell modeling

Physical modeling of cells requires annotating cell properties (e.g. area, height, width, aspect ratio). The set of annotated properties give an application such as synthesis a choice to pick one cell from a set of functionally equivalent cells, if one property is more desirable than another one under given synthesis goals and constraints.

Cell pins can also have annotated properties, such as pin capacitance, voltage swing, switching threshold etc.

Most of the modeling for test requirements are already fulfilled by the functional model. Declaration of pins and their direction (input, output, bidirectional) is already a generic requirement for cell modeling.

Scan insertion tools require specific annotations about cell and pin properties relevant for scan test. They also require reference to equivalent non-scan cells. An equivalent non-scan cell is a scan cell, when all scan specific hardware (e.g. multiplexor, scan clock) is removed.

The variables used in the functional model must have their counterpart in the pin declaration. Only primary input pins can be primary inputs of functions, while primary output pins, internal pins, or virtual pins can be primary or intermediate outputs of functions. Furthermore, test vectors for fault coverage can be derived from the functional model in a formal way.

The reminder of the modeling for test requirements can be covered by annotations of cell properties and cell pin properties. For instance, a cell can be labeled as a scan-flipflop, a pin can be labeled as scan input or mode select pin.

2.4.2 Wire modeling

The purpose of *wire modeling* is to get good estimates of *parasitic resistance* and *capacitance* as a function of *fanout*. These estimates are technology specific, and they depend on metal layer, sheet resistance, self capacitance per unit wirelength, fringe capacitance per unit wirelength, via resistance for wires routed through multiple layers.

The wires can be characterized by types, similar to cells. For example,

```
wire with fanout < 5 in metal 1
wire with 10 < fanout < 20 routed in metal 2 and 3
```

From a modeling standpoint, nothing special is required for performance modeling of wires that would be different from performance modeling of cells. The fanout will be an input variable, and capacitance and resistance would be output variables. The values can be expressed either in tables or in equations. Usually first order equations (with slope and intercept) are used for wire modeling.

Section 3
Library Format Specification

This section discusses the object model used by ALF and provides the syntax rules for all objects. The syntax rules are provided in standard BNF form.

3.1 Object Model

A *library* consists of one or more *objects*. Each object is defined by a keyword and an optional name for the object and an optional *value* of the object.

A *keyword* defines the type of the object. Section 3.1.1 and Section 3.1.2 define various types of objects used in ALF and related keywords.

An optional *identifier* (also called *name*) following the keyword defines the *name of the object*. This name must be used while referencing an object inside other objects in the library. If an object is not referenced by name, then the object need not be named.

A *literal* defines an optional value associated with the object. An *expression* can be used when the value of the object cannot be expressed as a literal.

An object may contain one or more objects. The containing object is called a *hierarchical object*. The contained objects are called *children objects*. The children objects are defined and referenced inside curly braces ({}) in the description of the hierarchical object.

Forward referencing of objects is not allowed. Therefore, all objects must be defined before they can be instantiated. This allows library parsers to be one-pas parsers.

Examples:

1. unnamed object without value assignment:

```
MY_OBJECT_TYPE {
   //fill in children objects
}
```

2. unnamed object with value assignment:

```
MY_OBJECT_TYPE = my_object_value {
   //fill in children objects
}
```

3. named object without value assignment:

```
MY_OBJECT_TYPE my_object_name {
   //fill in children objects
}
```

4. named object with value assignment:

```
MY_OBJECT_TYPE my_object_name = my_object_value {
   //fill in children objects
}
```

The objects in ALF are divided into four categories - *generic objects*, *library-specific objects*, *arithmetic models*, and *functions*.

3.1.1 Generic Objects

A generic object can appear at every level in the library within any scope. The semantics of a generic object must be understood by any ALF compiler if the generic object is within the scope of application for that compiler.

The following objects shall be considered generic objects:

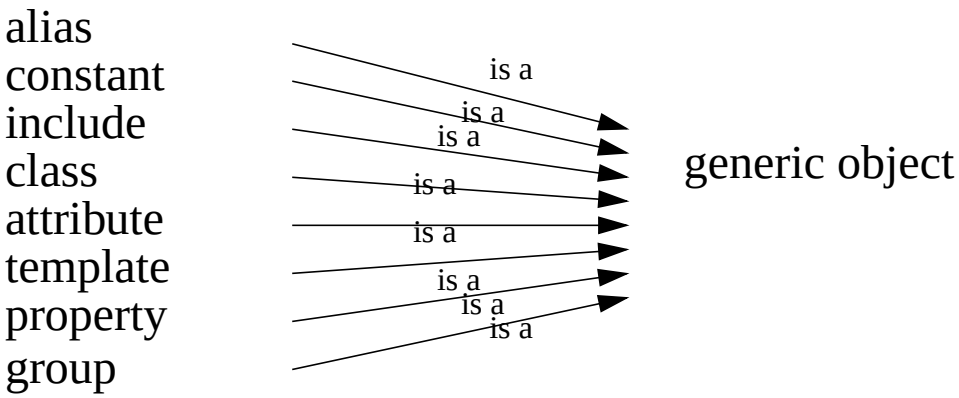

Figure 3-1: Generic objects

3.1.1.1 Constant

A *CONSTANT* object is a named object with value assignment and without children objects. Value is a number.

Example:

```
CONSTANT vdd = 3.3
```

3.1.1.2 Alias

An *ALIAS* object is a named object with value assignment and without children objects. Value is a string.

Example:

```
ALIAS RAMPTIME = SLEWRATE
```

3.1.1.3 Include

An *INCLUDE* object is a named object without value assignment and without children. The name is a quoted string containing the name of a file to be included.

Example:

```
INCLUDE 'primitives.alf'
```

Since the file name is a quoted string, any special symbols (like ~ or *) are allowed within the filename. The interpretation of those (for file search path etc.) is up to the application.

3.1.1.4 Class

A *CLASS* object is a named object with optional value assignments and children objects. The name can be used by other objects to reference the class object.

Example:

```
CLASS my_class { ... }
...
MY_OBJECT_TYPE my_object {
   CLASS = my_class
} // my_object belongs to my_class
```

3.1.1.5 Attribute

An *ATTRIBUTE* object is an unnamed object without value, but has children objects. The attribute object shall be the child object of another object. The children of the attribute object are unnamed objects which can have other unnamed objects as children objects. The purpose of an attribute object is to provide free association of objects with attributes when there is no special category available for the attributes.

Examples:

```
CELL rr_8x128 {
   ATTRIBUTE {ROM ASYNCHRONOUS STATIC}
}
PIN read_write_select {
   ATTRIBUTE {READ{POLARTITY=low} WRITE{POLARTITY=high}}
}
```

3.1.1.6 Template

A *TEMPLATE* object is a named object with one or more children objects. Any valid ALF object can be a child object of a template object. An identifier enclosed between < and > are recognized as *placeholders*. When a template object is used, each of its placeholders must be referenced by order or by explicit name association.

Example:

```
TEMPLATE std_table {
   CAPACITANCE {PIN=<pin1> UNIT=pF TABLE {0.02 0.04 0.08 0.16}}
   SLEWRATE {PIN=<pin2> UNIT=ns TABLE {0.1 0.3 0.9}}
}
```

An instantiation of the above template object with explicit reference to placeholders by name:

```
std_table{pin1=out pin2=in}
```

An instantiation of the above template object with implicit reference to placeholders by order:

```
std_table{out in}
```

If a symbol within a placeholder appears more than once in the template definition, the order for implicit reference is defined by the first appearance of the symbol. Explicit referencing improves the readability and is the recommended usage.

A template instantiation can appear at any place within a hierarchical object, as long as the template object contains the structure of valid objects inside. Hierarchical templates contain other template objects.

3.1.1.7 Property

A *PROPERTY* object is a named *annotation container*. It can be used at any level in the library. It is used for arbitrary parameter-value assignment.

Example:

```
PROPERTY items {
   parameter1=value1
   parameter2=value2
}
```

3.1.1.8 Group

A *GROUP* object is a set of elements with commonality between them. Thus the common characteristics can be defined once for the group instead of being repeated for each element.

Example:

```
GROUP time_measurements = {DELAY SLEWRATE SKEW JITTER}
```

Thus the statement

```
time_measurements { UNIT = ns }
```

replaces the following statements:

```
DELAY       { UNIT = ns }
SLEWRATE    { UNIT = ns }
SKEW        { UNIT = ns }
JITTER      { UNIT = ns }
```

3.1.2 Library-specific objects

The library-specific objects define their nature and their relationship to each other by containment rules. For example. a library may contain a cell, but a cell may not contain a library. However, both the library object and the cell object may contain any generic object. A generic object defined at the library level makes it visible inside the scope of that library, defining it on the cell level makes it visible inside the scope of that cell and its children objects.

The library-specific objects require no interpretation of their semantics.

3.1.3 Arithmetic models

An arithmetic model is an object that describes characterization data, or more abstract,

measurable relationships between physical quantities. The modeling language allows tabulated data as well as linear and non-linear equations. The equations consists of arithmetic expressions, for which the IEEE standards have been adopted.

3.1.4 Functions

A function is an object that describes the functional specification of a digital circuit (or a digital model of an analog or a mixed-signal circuit) in a canonical form. The modeling language allows behavioral models as well as statetables and structural models with primitives. The behavioral models contain boolean expressions, for which the IEEE standards have been adopted. Since boolean expressions are insufficient to describe sequential logic, ALF introduces new operators and symbols that can be used in conjunction with boolean operators and symbols. Expressions that use both the IEEE operators and the new operators, are called vector expressions.

The following figures describe the four types of objects and their relationships with each other.

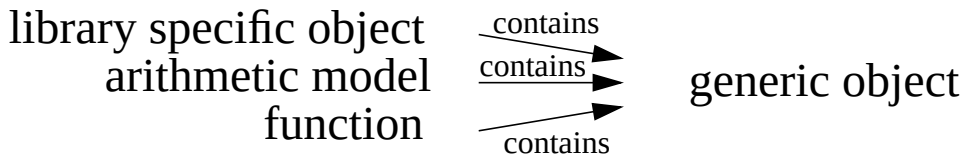

Figure 3-2: Library-specific objects

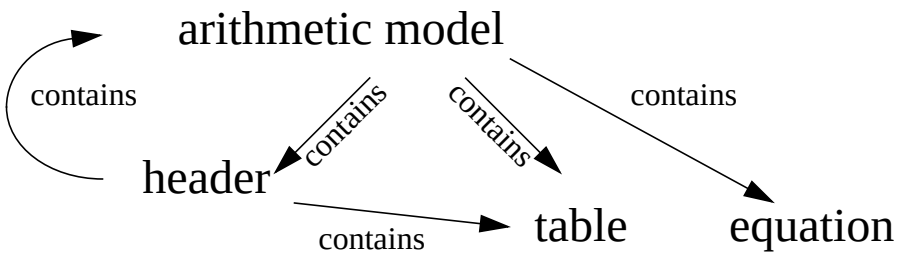

Figure 3-3: Arithmetic model

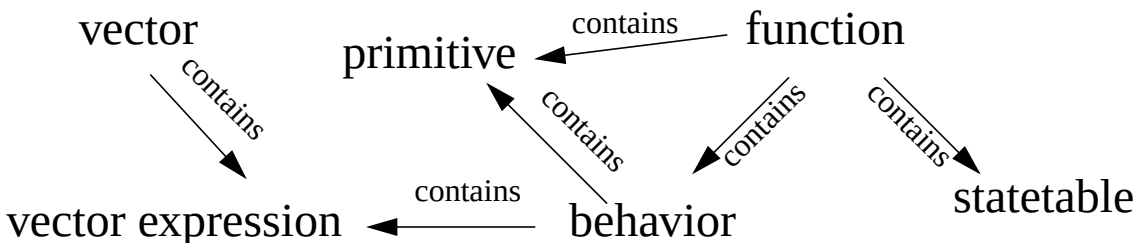

Figure 3-4: Function

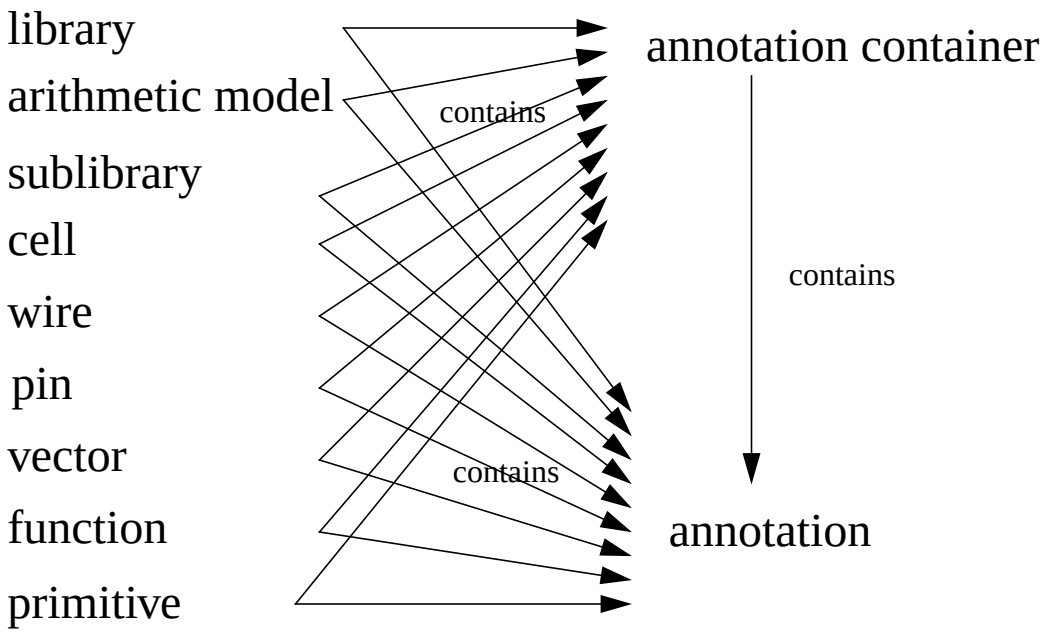

Figure 3-5: Annotations

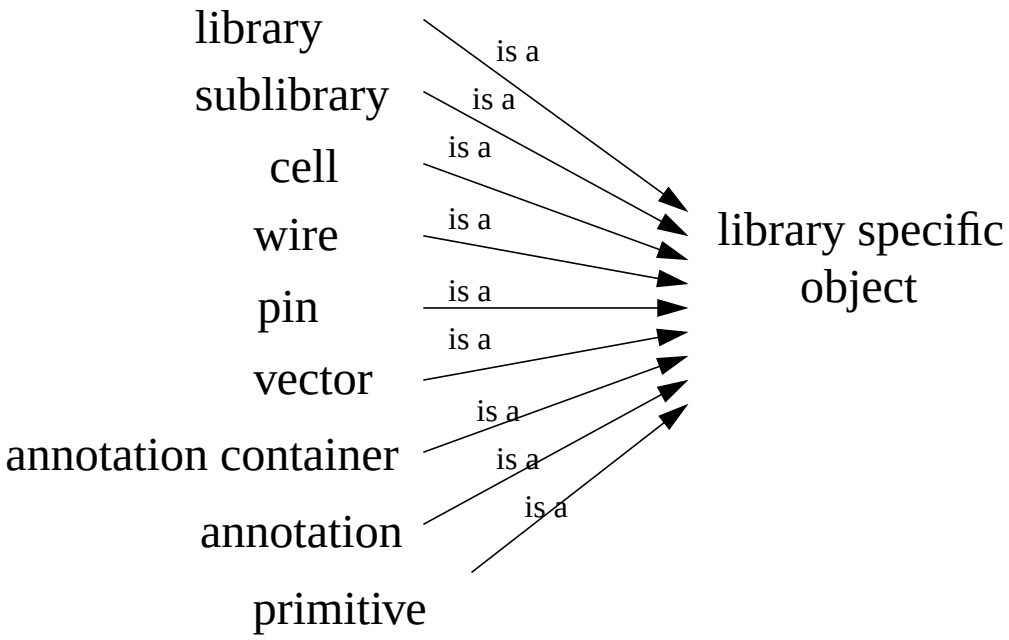

Figure 3-6: Library-specific objects

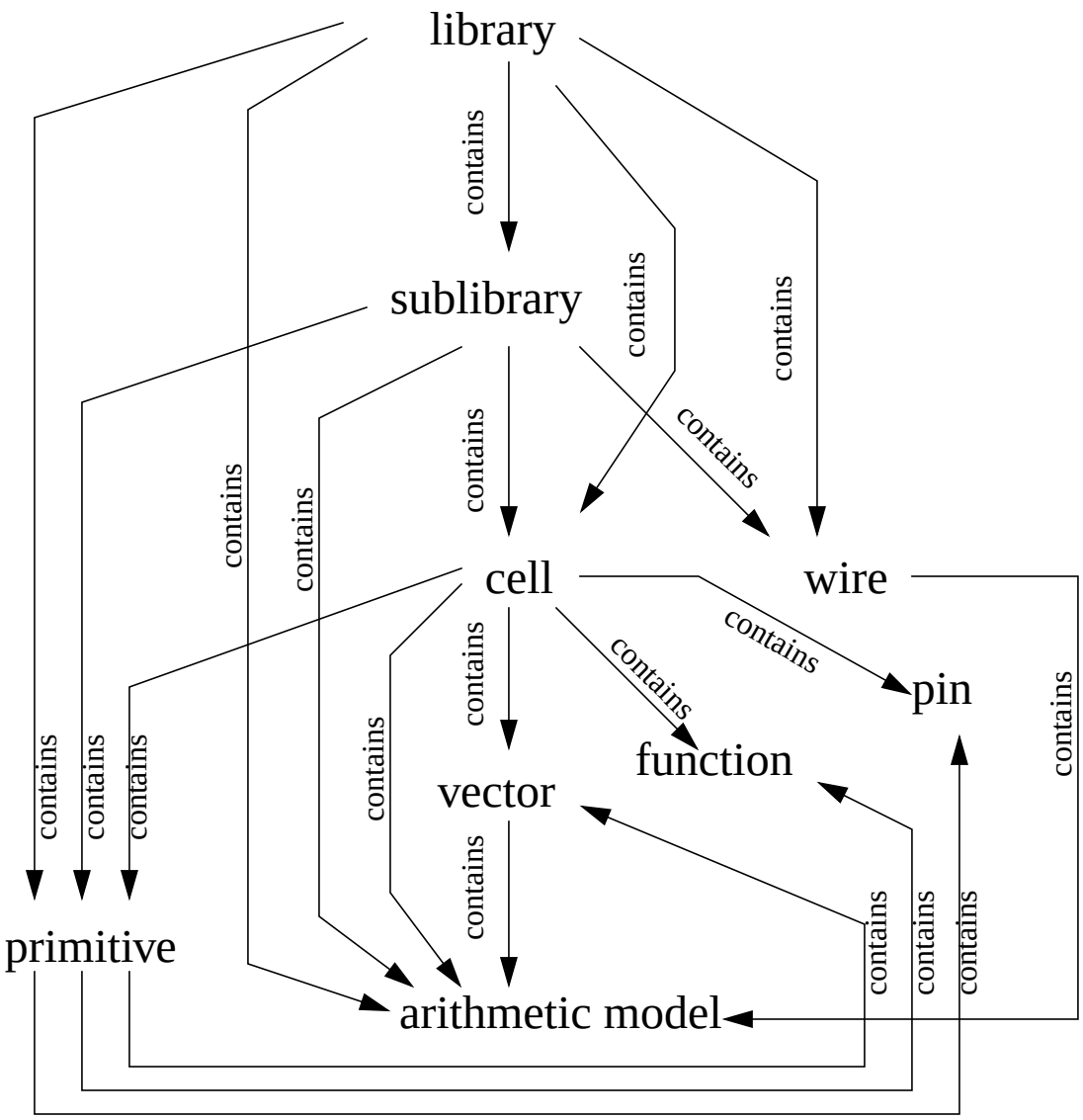

Figure 3-7: Library objects and their relationships

3.2 Lexical rules

3.2.1 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.

3.2.2 Lexical tokens

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

3.2.3 Whitespace Characters

The following characters shall be considered *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

Figure 3-8: List of whitespace characters

Comments are also considered white space (see Section 3.2.6).

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

3.2.4 Reserved and Non-reserved Characters

The ASCII character set shall be divided in three categories - whitespace (Section 3.2.3), reserved characters, and non-reserved characters. The reserved characters are symbols that make up punctuation marks and operators. The non-reserved characters shall be used for creating identifiers and numbers.

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

nonreserved_character ::=
        letter | digit | _ | $

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

Figure 3-9: Reserved and non-reserved characters

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

3.2.5 Delimiters

A *delimiter* is either a reserved character or one of the following compound operators, each composed of two or three adjacent reserved characters:

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

Figure 3-10: Tokens that make up delimiters

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

3.2.6 Comments

ALF has two forms to introduce comments.

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.

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

Figure 3-11: Single-line and block comments

3.2.7 Number Literals

Constant numbers can be specified as integer literal or real literal.

The *integer* literal is a decimal integer constant.

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

sign     ::=  +  |  -

unsigned_number ::=  {_} integer_digit  { _ | integer_digit }

integer_literal ::=  [ sign ] unsigned_number

real_literal ::=
            unsigned_number .  unsigned_number
          | unsigned_number [ . unsigned_number] E [sign] unsigned_number
```

Figure 3-12: Integer and real numbers

3.2.8 Boolean Literals

The *bit* literal shall represent a single bit constant.

```
bit_literal ::= X | Z | L | H | U | ? | 0 | 1
```

where

0	logic zero
1	logic one
X or **x**	unknown value
L or **l**	weak zero
H or **h**	weak one
W or **w**	weak unknown
Z or **z**	high-impedance value
U or **u**	unknown (i.e. uninitialized)
?	don’t care

Figure 3-13: Single bit constants

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.

```
based_literal ::=
          binary_base { _ | binary_digit }
        | octal_base { _ | octal_digit }
        | decimal_base { _ | decimal_digit }
        | hex_base { _ | hex_digit }

binary_base ::=
        'B

octal_base ::=
        'O

decimal_base ::=
        'D

hex_base ::=
        'H

binary_digit ::=
        bit_literal

octal_digit ::=
        binary_digit  | 2 | 3 | 4 | 5 | 6 | 7

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

hex_digit ::=
        octal_digit | 8 | 9 | A | B | C | D | E | F | a | b | c | d | e | f
```

Figure 3-14: Based constants

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.

3.2.9 Edge Literals

An *edge literal* shall be constructed by two bit literals or two based literals. 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 shall be allowed.

```
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::=
        ?? | ?~ | ?! | ?-
```

Figure 3-15: Edge literals

3.2.10 Quoted String

The *quoted string* shall be a sequence of zero or more graphic characters enclosed between two quotation marks (" or ') and contained on a single line. Character *escape codes* are used inside the string literal to represent some common special characters. The character that may follow the backslash (\), and their meanings are listed below.

Symbol	ASCII Code (octal)	Usage
\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
\'	047	single quotation mark
\\	134	backslash
\ddd		3-digit octal value of ASCII character

Figure 3-16: Special characters in quoted strings

```
quoted string ::=
          " { any_character } "
        | ' { any_character } '
```

Figure 3-17: Quoted string

3.2.11 Identifiers

Identifiers are used in ALF as names of objects, reserved words and context sensitive keywords. An identifier shall be any sequence of letters, digits, underscore (_), and dollar sign ($) character. If an identifier is constructed from one or more non-reserved characters, it is called *non-escaped identifier*.

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

If an identifier is constructed from one or more reserved characters, it is called an *escaped identifier*. The escaped identifiers shall start with the backslash character (\) and end with a whitespace.The backslash character is considered to be part of the escaped identifier, but the whitespace is not. Therefore, an escaped identifier `\PIN` is not the same as a non-escaped identifier `PIN`.

```
escaped_identifier ::=
        ‘\’ { nonreserved_character | reserved_character }
```

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 an formal parameter, which is replaced by the actual parameter in

template instantiation.

```
placeholder_identifier ::=
         ‘<’ nonescaped_identifier ‘>’
```

All the identifiers are case-insensitive.

3.3 Keywords

Keywords are case-insensitive. For clarity, this document uses uppercase letters for keywords identifying objects, and lowercase letters for keywords identifying values of objects.

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

3.3.1 Keywords for generic object types

The following keywords are used to identify generic object types:

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

Figure 3-18: Keywords for generic object types

3.3.2 Keywords for Operators

The following keywords are used for built-in arithmetic functions:

```
ABS                 absolute value
EXP                 natural exponential function
LOG                 natural logarithm
MIN                 minimum
MAX                 maximum
```

Figure 3-19: Keywords for built-in arithmetic functions

3.3.3 Context-Sensitive Keywords

In order to address the need of extensible modeling, we introduce the concept of context-sensitive keywords. They are some kind of a free vocabulary for the library application. New context-sensitive keywords can be introduced as long as they do not clash with any existing keyword.

A predefined set of context-sensitive keywords and their semantic meaning is proposed in Section 3.6. This set can be extended.

3.4 Syntax Rules

The formal syntax of ALF language is described using Backus-Naur Form (BNF).

3.4.1 Assignments

```
unnamed_assignment ::=
           context_sensitive_keyword = number [;]
         | context_sensitive_keyword = string [;]

unnamed_assignments ::=
           unnamed_assignment { unnamed_assignment }

named_assignment ::=
           context_sensitive_keyword identifier = number [;]
         | context_sensitive_keyword identifier = string [;]

named_assignments ::=
           named_assignment { named_assignment }

assignment ::=
           named_assignment
         | unnamed_assignment

assignments ::=
           assignment { assignment }

combinational_assignment ::=
           identifier [index] = boolean_expression ;

combinational_assignments ::=
           combinational_assignment  { combinational_assignment }

pin_assignment ::=
           identifier [index] = identifier [index] [;]
         | identifier [index] = logic_literal [;]
         | logic_literal  = identifier [index] [;]

pin_assignments ::=
           pin_assignment { pin_assignment }

sequential_assignment ::=
           @ ( vector_boolean_expression ) { combinational_assignments }
         { : ( vector_boolean_expression ) { combinational_assignments } }

sequential_assignments ::=
         sequential_assignment  { sequential_assignment }
```

3.4.2 Expressions

```
arithmetic_expression ::=
          [ arithmetic_unary_operator ] arithmetic_primary
        | arithmetic_expression  arithmetic_binary_operator
          arithmetic_expression
        | arithmetic_function_operator ( arithmetic_expression
          { , arithmetic_expression } )

arithmetic_primary ::=
          number
        | identifier
        | ( arithmetic_expression )

boolean_expression ::=
          [ boolean_unary_operator ]  boolean_primary
        | boolean_expression boolean_binary_operator boolean_expression
        | boolean_expression ? boolean_expression : boolean_expression

boolean_primary ::=
          logic_literal
        | identifier [ index ]
        | ( boolean_expression )

vector_boolean_expression ::=
          vector_expression
        | boolean_expression

vector_expression ::=
        ( vector_expression )
        | vector_unary_operator boolean_expression
        | vector_expression  vector_binary_operator  vector_expression
        | vector_expression  boolean_binary_operator  vector_expression
        | vector_expression  &&  boolean_expression
        | boolean_expression  &&  vector_expression
        | vector_expression  &  boolean_expression
        | boolean_expression  &  vector_expression
        | boolean_expression ? vector_expression : vector_expression
```

3.4.3 Instantiations

```
cell_instantiation ::=
          cell_identifier { primaries }
        | cell_identifier { pin_assignments }

cell_instantiations ::=
        cell_instantiation  { cell_instantiation }

primitive_instantiation ::=
          primitive_identifier { primaries }
        | primitive_identifier { combinational_assignments }

primitive_instantiations ::=
        primitive_instantiation  { primitive_instantiation }

template_instantiation ::=
          template_identifier { primaries }
        | template_identifier { unnamed_assignments }

template_instantiations ::=
        template_instantiation  { template_instantiation }
```

3.4.4 Literals

```
context_sensitive_keyword ::=
          nonescaped_identifier

edge_literal ::=
          bit_edge_literal
        | word_edge_literal
        | symbolic_edge_literal

edge_literals::=
          edge_literal { edge_literal }

identifier ::=
          nonescaped_identifier
        | escaped_identifier
        | placeholder_identifier

identifiers ::=
         identifier { identifier }

index ::=
          [ primary ]
        | [ primary : primary ]

logic_literal ::=
          bit_literal
        | based_literal

logic_literals::=
          logic_literal { logic_literal }

logic_value ::=
          logic_literal
        | edge_literal
        | ( [!] logic_variable )

logic_values ::=
          logic_value {logic_value}

logic_variable ::=
          pin_identifier [index ]

logic_variables ::=
          logic_variable {logic_variable}

number ::=
          integer_literal
        | real_literal

numbers::=
          number { number }

primary ::=
          number
        | identifier

primaries ::=
          primary { primary }

string ::=
          quoted_string
```

```
          | identifier
```

3.4.5 Operators

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

arithmetic_function_operator ::=
           abs
         | exp
         | log
         | min
         | max

arithmetic_unary_operator ::=
           + | -

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

boolean_unary_operator ::=
           ! | ~ | & | ~& | | | ~| | ^ | ~^

vector_binary_operator ::=
            -> | <-> | &> | <&>

vector_unary_operator ::=
           edge_literal
```

See Section 3.5 for further details on operators.

3.4.6 Auxiliary Objects

```
all_purpose_item ::=
           annotation
         | annotation_container
         | generic_object
         | template_instantiation
         | cell_instantiation

all_purpose_items ::=
           all_purpose_item { all_purpose_item }

annotation ::=
           assignment [ { all_purpose_items } ]

annotation_container ::=
           context_sensitive_keyword { all_purpose_items }

generic_object ::=
           alias
         | attribute
         | constant
         | class
         | group
         | include
         | property
         | template

generic_objects ::=
           generic_object { generic_object }

library_specific_object ::=
           annotation
         | annotation_container
         | cell
         | function
         | library
         | pin
         | primitive
         | sublibrary
         | vector
         | wire

object ::=
           generic_object
         | library_specific_object
         | arithmetic_model
         | function

objects ::=
           object { object }

source_text ::=
           library
```

3.4.7 Generic Objects

```
alias ::=
          ALIAS identifier = identifier [;]

attribute ::=
          ATTRIBUTE { attribute_items }

attribute_item ::=
          identifier [ { unnamed_assignments } ]

attribute_items ::=
          attribute_item { attribute_item }

class::=
          CLASS identifier  {;}
          CLASS identifier  [ { generic_objects } ]

constant ::=
          CONSTANT identifier = number [;]
          CONSTANT identifier = logic_literal [;]

group ::=
          GROUP group_identifier { identifiers }
        | GROUP group_identifier { numbers }
        | GROUP group_identifier { edge_literals }
        | GROUP group_identifier { logic_literals }
        | GROUP group_identifier { logic_variables }
        | GROUP group_identifier { int_literal :  int_literal }

include ::=
          INCLUDE quoted_string [;]

property ::=
          PROPERTY { unnamed_assignments }

template ::=
          TEMPLATE template_identifier { objects }
```

3.4.8 Cell Object

```
cell ::=
          CELL cell_identifier  [ { cell_items } ]
        | cell_template_instantiation

cells ::=
          cell { cell }

cell_item ::=
          all_purpose_item
        | pin
        | primitive
        | function
        | arithmetic_model
        | vector

cell_items ::=
          cell_item {cell_item}
```

3.4.9 Library Object

```
library ::=
          LIBRARY library_identifier { library_items [sublibraries] }
        | library_template_instantiation

libraries ::=
          library { library }

library_item ::=
          all_purpose_item
        | arithmetic_model
        | cell
        | primitive
        | wire

library_items ::=
          library_item { library_item }
```

3.4.10 Pin Object

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

pins ::=
          pin { pin }

pin_item ::=
          all_purpose_item

pin_items ::=
          pin_item { pin_item }
```

3.4.11 Primitive Object

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

primitives ::=
          primitive { primitive }

primitive_item ::=
          all_purpose_item
        | pin
        | function

primitive_items ::=
          primitive_item { primitive_item }
```

3.4.12 Sublibrary Object

```
sublibrary ::=
          SUBLIBRARY library_identifier { library_items }
        | sublibrary_template_instantiation

sublibraries ::=
          sublibrary { sublibrary }
```

3.4.13 Vector Object

```
vector ::=
          VECTOR ( vector_boolean_expression )  { vector_items }
        | vector_template_instantiation

vectors ::=
          vector { vector }

vector_item ::=
          all_purpose_item
        | arithmetic_model

vector_items ::=
          vector_item { vector_item }
```

3.4.14 Wire Object

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

wires ::=
          wire { wire }

wire_item ::=
          all_purpose_item
        | arithmetic_model

wire_items ::=
          wire_item { wire_item }
```

3.4.15 Arithmetic Model

```
arithmetic_model ::=
          context_sensitive_keyword [ identifier ]
          { [all_purpose_items] [header] bodies }
          context_sensitive_keyword [identifier] =
          primary  [{ all_purpose_items }]
        | arithmetic_model_template_instantiation

header ::=
          HEADER { header_items [ body ] }
        | header_template_instantiation

header_items ::=
          header_item { header_item }

header_item ::=
          identifier
        | all_purpose_item
        | arithmetic_model

body ::=
          table
        | equation

bodies ::=
          body { body }

table ::=
          TABLE { primaries }
        | table_template_instantiation

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

3.4.16 Function

```
function ::=
           FUNCTION [ identifier ] { [all_purpose_items] [primitives]
           [function_bodies] }
         | function_template_instantiation

functions ::=
           function { function }

function_bodies ::=
           function_body { function_body }

function_body ::=
           STATETABLE [ identifier ]  {  statetable_body }
         | BEHAVIOR [ identifier ]  { behavior_body }

statetable_body ::=
           logic_variables : logic_variables ;
           logic_values : logic_values ;
           { logic_values : logic_values ; }

behavior_body ::=
           primitives
         | combinational_assignments
         | sequential_assignments
         | primitive_instantiations
```

3.5 Operators

The operators are divided into four groups:

- Arithmetic operators
- Boolean operators on scalars, i.e. single bits
- Boolean operators on words, i.e. arrays of bits
- Vector operators

3.5.1 Arithmetic operators

Unary operators:

```
+    // positive sign (for integer or float)
-    // negative sign (for integer or float)
```

Binary operators:

```
+    // addition (integer or float)
-    // subtraction (integer or float)
*    // multiplication (integer or float)
/    // division (integer or float)
**   // exponentiation (integer or float)
%    // modulo division (integer or float)
```

Function operators:

```
LOG  // natural logarithm (argument is + integer or float)
EXP  // natural exponential (argument is integer or float)
ABS  // absolute value (argument is integer or float)
MIN  // minimum (all arguments are integer or float)
MAX  // maximum (all arguments are integer or float)
```

Function operators with one argument (such as `log`, `exp` and `abs`) or multiple arguments (such as `min` and `max`) must have the arguments within parenthesis, e.g. `min(1.2,-4.3,0.8)`.

3.5.2 Boolean operators on scalars

Unary operators:

```
!          // logical inversion
```

Binary operators:

```
&& or &      // logical AND
|| or |      // logical OR
== or ~^     // equivalence (XNOR)
!= or ^      // antivalence (XOR)
```

Ternary operator:

```
? :          // if-then-else clause
```

3.5.3 Boolean operators on words

Unary reduction operators (result is a logic value):

```
&       // AND all bits
~&      // NAND all bits
|       // OR all bits
~|      // NOR all bits
^       // XOR all bits
~^      // XNOR all bits
```

Binary reduction operators (result is a logic value):

```
>       // greater
<       // smaller
>=      // greater or equal
<=      // smaller or equal
```

Unary bitwise operator (result is an array of bits):

```
~       // bitwise inversion
```

Binary bitwise operators (result is an array of bits):

```
&       // bitwise AND
|       // bitwise OR
^       // bitwise XOR
~^      // bitwise XNOR
```

Binary operators (result is an extended array of bits):

```
<<      // shift left
>>      // shift right
+       // addition
-       // subtraction
*       // multiplication
/       // division
%       // modulo division
```

The arithmetic operators addition, subtraction, multiplication, and division shall be *unsigned* if all the operands in an expression are unsigned variables or constants. If any of the operands are signed operands, the arithmetic operators shall be *signed* operators.

3.5.4 Vector operators

A transition operation is defined using unary operators on a scalar net. The scalar constants (see figure 3-13) shall be used to indicate the start and end states of a transition on a scalar net.

```
bit bit  apply transition from bit value to bit value
```

For example,

```
01 is a transition from 0 to 1.
```

No whitespace shall be allowed between the two scalar constants. The following transition

operators shall be considered legal:

`01`	signal toggles from 0 to 1
`10`	signal toggles from 1 to 0
`00`	signal remains 0
`11`	signal remains 1
`0?`	signal remains 0 or toggles from0 to arbitrary value
`1?`	signal remains 1 or toggles from 1 to arbitrary value
`?0`	signal remains 0 or toggles from arbitrary value to 0
`?1`	signal remains 1 or toggles from arbitrary value to 1
`??`	signal remains constant or toggles between arbitrary values

Figure 3-20: Unary vector operators on bits

Unary operators for transitions can also appear in `STATETABLE`.

Transition operators are also defined on words (can appear in `STATETABLE` as well):

```
'base word 'base word
                apply transition from word value to word value,
```

For example,

```
'hA'h5 is a transition of a 4-bit signal from 'b1010 to 'b0101.
```

No whitespace shall be allowed between *base* and *word*.

The following unary operators are defined on bits and words:

`?-`	no transition occurs
`??`	apply arbitrary transition, including possibility of constant value
`?!`	apply arbitrary transition, excluding possibility of constant value
`?~`	apply arbitrary transition with all bits toggling

Figure 3-21: Unary vector operators on bits or words

The following binary operators are defined:

`->`	Left-hand side (LHS) transition *is followed by* Right-hand side (RHS) transition
`&& or &`	LHS *and* RHS transition *occur simultaneously*
`|| or |`	LHS *or* RHS transition *occurs*
`<->`	LHS transition *follows or is followed by* RHS transition
`&>`	LHS transition *is followed by or occurs simultaneously* with RHS transition
`<&>`	LHS transition *follows or is followed by or occurs simultaneously* with RHS transition

Figure 3-22: Binary vector operators

The following expressions are considered equivalent:

```
(?? a)   ==
      (0? a)||(1? a)||(Z? a)||(X? a)
||    (H? a)||(L? a)||(W? a)
||    (?0 a)||(?1 a)||(?Z a)||(?X a)
||    (?H a)||(?L a)||(?W a)
```

```
(01 a <-> 01 b) == (01 a -> 01 b)||(01 b -> 01 a)
(01 a  &> 01 b) == (01 a -> 01 b)||(01 a && 01 b)
(01 a <&> 01 b) == (01 a -> 01 b)||(01 b -> 01 a)||(01 a && 01 b)
```

The binary AND operator is also defined between a vector expression and a boolean expression. The result is a conditional vector expression.

Example:

```
(01 a && !b)// ‘a’ rises while b==0
```

Every binary vector operator may be applied to a conditional vector expression.

3.5.5 Operator priorities

The priority of binding operators to operands shall be from strongest to weakest in the following order:

1. unary operators for vector expression
2. binary operators for vector expression
3. unary operator for arithmetic and boolean expression
4. `XNOR` (`~^`), `XOR` (`^`), relational (`>, <, >=, <=`), exponentiation (`**`) , shift (`<<, >>`)
5. `AND` (`&`), `NAND` (`~&`), multiplication (`*`), division (`/`), modulo division (`%`)
6. `OR` (`|`), `NOR` (`~|`), addition (`+`), subtraction (`-`)
7. ternary operator (`?:`)

Operators with equal priority are evaluated strictly in order of occurrence from left to right. The parenthesis `( )` shall be used for changing the priority of binding operators to operands.

3.6 Context-sensitive keywords

The context-sensitive keywords permit legal extensions to ALF syntax. An ALF parser shall either accept or ignore when an unknown keyword or annotation is encountered. The purpose of context-sensitive keywords is to have a vocabulary of keywords with already well-defined semantic meaning. That means, an ALF compiler for an application must understand those keywords needed (used) by the application. For example, a compiler that needs SLEWRATE must understand the keyword `SLEWRATE` and not expect a keyword `RAMPTIME`.

3.6.1 Annotation containers

Any complex object can be used as an *annotation container*. In addition, the following objects are defined only for the purpose of *unnamed annotation containers*.

`FROM`	specifies starting point of `DELAY` or `SLEWRATE` measurement
`TO`	specifies ending point of `DELAY` or `SLEWRATE` measurement
`SCAN`	specifies information relevant to design for test
`LIMIT`	specifies limit values
`VIOLATION`	specifies items relevant to timing violations
`INFORMATION`	specifies purely informational items

Figure 3-23: Unnamed annotation containers

The following syntax notations describe the annotations:

```
unnamed_annotation_container ::=
          FROM        ( from_to_container )
        | TO          ( from_to_container )
        | SCAN        ( scan_container )
        | LIMIT       ( limit_container )
        | VIOLATION   ( violation_container )
        | INFORMATION ( information_container )

limit_container ::=
          limit_type integer_constant
        | limit_type real_constant

limit_type ::=
          MAX
        | MIN
        | TYP

from_to_container ::=
          from_to_type integer_constant
        | from_to_type real_constant

from_to_type ::=
          RISE        // applies to rising signal
        | FALL        // applies to falling signal

violation_container ::=
          MESSAGE quoted_string
        | MESSAGE_TYPE INFORMATION
        | MESSAGE_TYPE WARNING
        | MESSAGE_TYPE ERROR

information_container ::=
          VERSION string
        | TITLE quoted_string
        | PRODUCT quoted_string
        | AUTHOR quoted_string
        | DATETIME string

scan_container ::=
          SCAN_POSITION positive_integer   // position in scan chain
        | STUCK stuck_type                 // fault model type
        | OFF_STATE off_state_type         // map information

stuck_type ::=
          stuck_at_0
        | stuck_at_1
        | both
        | none

off_state_type ::=
          inverted     // polarity change between scan and non-scan cell
        | non_inverted // no polarity change
```

3.6.2 Keywords for referencing objects used as annotation

The following object references may be used as annotations:

```
CELL      [string] reference to a declared CELL object
PRIMITIVE [string] reference to a declared PRIMITIVE object
PIN       [string] reference to a declared PIN object
CLASS     [string] reference to a declared CLASS object
```

3.6.3 Annotations for a PIN object

A PIN object may contain the following annotations:

3.6.3.1 VIEW annotation

```
VIEW          [string]
```

annotates the view where the pin appears, which can take the following values:

`functional`	pin appears in functional netlist
`physical`	pin appears in physical netlist
`both`	pin appears in both functional and physical netlist
`none`	pin does not appear in netlist

3.6.3.2 PINTYPE annotation

```
PINTYPE       [string]
```

annotates the type of the pin, which can take the following values:

`digital`	digital signal pin
`analog`	analog signal pin
`supply`	power supply or ground pin

3.6.3.3 SIGNALTYPE annotation

```
SIGNALTYPE    [string]
```

annotates the type of the signal connected to the pin, which can take the following values:

`data`	general data signal {default}
`scan_data`	scan data signal
`control`	general control signal
`select`	select signal of a multiplexor
`out_enable`	output enable signal
`scan_enable`	scan enable signal
`scan_out_enable`	scan output enable signal
`clear`	clear signal of a flipflop or latch
`set`	set signal of a flipflop or latch
`clock`	clock signal of a flipflop or latch
`enable`	enable signal
`write`	write signal for memory, register file

`read`	read signal for memory, register file
`scan_clock`	scan clock signal of a flipflop or latch
`master_clock`	master clock signal of a flipflop or latch
`slave_clock`	slave clock signal of a flipflop or latch

3.6.3.4 DRIVETYPE annotation

`DRIVETYPE [string]`

annotates the drive type for the pin, which can take the following values:

`cmos`	standard cmos signal
`nmos`	nmos or pseudo nmos signal
`pmos`	pmos or pseudo pmos signal
`nmos_pass`	nmos passgate signal
`pmos_pass`	pmos passgate signal
`cmos_pass`	cmos passgate signal, i.e. full transmission gate
`ttl`	TTL signal
`open_drain`	open drain signal
`open_source`	open source signal

3.6.3.5 DIRECTION annotation

`DIRECTION [string]`

annotates the direction of the pin, which can take the following values:

`input`	input pin
`output`	output pin
`both`	bidirectional pin
`none`	no direction can be assigned to the pin

3.6.3.6 SCOPE annotation

`SCOPE [string]`

annotates modeling scope of a pin, which can take the following values:

`behavior`	can be used for modeling functional behavior
`measure`	measurements can be done related to this pin, e.g. timing or power characterization
`both`	can be used for function as well as for characterization measurements
`none`	no model, pin just exists

3.6.3.7 ACTION annotation

`ACTION [string]`

annotates action of the signal, which can take the following values:

`synchronous`	signal acts in synchronous way
`asynchronous`	signal acts in asynchronous way (default)

3.6.3.8 POLARITY annotation

```
POLARITY        [string]
```

annotates the polarity of the pin signal, which can take the following values:

`high`	signal active high or to be driven high
`low`	signal active low or to be driven low
`rising_edge`	signal sensitive to rising edge
`falling_edge`	signal sensitive to falling edge
`double_edge`	signal sensitive to any edge
`inverted`	polarity change between input and output
`non_inverted`	no polarity change between input and output
`both`	polarity may change or not (e.g. XOR)
`none`	polarity has no meaning(e.g. analog signal)

3.6.3.9 ENABLE_PIN annotation

```
ENABLE_PIN      reference to a pin with SIGNALTYPE=out_enable
```

3.6.3.10 PULL annotation

```
PULL            [string]
```

which can take the following values:

`up`	pullup device connected to pin
`down`	pulldown device connected to pin
`both`	pullup and pulldown device connected to pin
`none`	no pull device (default)

3.6.3.11 ORIENTATION annotation

`ORIENTATION` pin orientation [string]

which can take the following values:

```
left
right
top
bottom
```

3.6.3.12 CONNECT_CLASS annotation

`CONNECT_CLASS`	reference to a declared class object for connectivity determination

3.6.3.13 DATATYPE annotation

`DATATYPE`	[string], only relevant for bus pins, which can take the following values:
`signed`	result of arithmetic operation is signed 2’s complement
`unsigned`	result of arithmetic operation is unsigned

3.6.4 Annotations for a VECTOR object

A VECTOR object may contain the following annotations:

3.6.4.1 LABEL annotation

```
LABEL    [quoted string]
```

to be used to ensure SDF matching with conditional delays across Verilog, VITAL etc.

3.6.5 Annotations for a CELL object

A CELL object may contain the following annotations:

3.6.5.1 CELLTYPE annotation

```
CELLTYPE       [string]
```

which can take the following values:

```
buffer
combinational
multiplexor
flipflop
latch
memory
block
core
pad
special
```

3.6.5.2 BUFFERTYPE annotation

```
BUFFERTYPE     [string]
```

which can take the following values:

```
input
output
inout
internal
```

3.6.5.3 DRIVETYPE annotation

```
DRIVERTYPE     [string]
```

which can take the following values:

```
predriver
slotdriver
```
both

3.6.5.4 PARALLEL_DRIVE annotation

PARALLEL_DRIVE [positive integer] number of parallel drivers

3.6.5.5 SCAN_TYPE annotation

`SCAN_TYPE` [string]

which can take the following values:

```
muxscan
clocked
lssd
control_0
control_1
```

3.6.5.6 SCAN_USAGE annotation

`SCAN_USAGE` [string]

which can take the following values:

`input`	primary input in a chain of cells
`output`	primary output in a chain of cells
`hold`	

3.6.5.7 NON_SCAN_CELL annotation

`NON_SCAN_CELL` reference to `PRIMITIVE` or `CELL`, assignment of `PIN` connectivity LHS refers to pins in non-scan-cell, RHS refers to pins in scan cell. constant values (i.e. pins tied to fixed values) may appear on both sides. Multiple non-scan cells can be referenced within the same scope by giving a name to each one.

3.6.6 Attributes

Keywords inside `ATTRIBUTE` can be used for add pin information which does not fit into the annotation scheme.

3.6.6.1 ATTRIBUTE within a PIN object:

`schmitt`	Schmitt trigger signal
`tristate`	tristate signal
`xtal`	crystal/oscillator signal
`pad`	pad going off-chip

The following attributes can also have `POLARITY` annotation:

`tie`	signal that needs to be tied to a fixed value
`read`	read enable signal
`write`	write enable signal

3.6.6.2 ATTRIBUTE within a CELL object:

`RAM`	Random Access Memory
`ROM`	Read Only Memory
`CAM`	Content Addressable Memory
`static`	static device (e.g. static CMOS, static RAM)

`dynamic`	dynamic device (e.g. dynamic CMOS, dynamic RAM)
`asynchronous`	asynchronous operation
`synchronous`	synchronous operation

3.6.6.3 ATTRIBUTE within a LIBRARY object:

There are no attributes with predefined meaning specified yet.

3.6.7 Annotations for arithmetic models

The following four annotations shall be recognized within arithmetic models:

The *default annotation* allows use of the default value instead of the arithmetic model, if the arithmetic model is beyond the scope of the application tool. The default value shall be specific to the object type.

The *unit annotation* associates units with the value computed by the arithmetic model. The units can take the following values:

f* or 1E-15
p* or 1E-12
n* or 1E-9
u* or 1E-6
m* or 1E-3
1
k* or 1E+3
meg* or 1E+6
g* or 1E+9
any positive real number

The `*` indicates wildcard here, e.g. `ns, gigahz`.

The *measurement annotation* indicates the type of measurement used for the computation in arithmetic model. It can take the following values:

```
transient
static
average
rms
peak
```

The *connect_rule annotation* specifies connection requirement. It can take the following values:

`must_short`	short connection required
`can_short`	short connection allowed
`cannot_short`	short connection disallowed

```
arithmetic_model_annotations ::=
           default_annotation
         | unit_annotation
         | measurement_annotation
         | connect_annotation

default_annotation ::=
           DEFAULT constant_expression

unit_annotation ::=
           UNIT units_string
         | UNIT real_constant

measurement_annotation ::=
           MEASUREMENT measurement_string

measurement_string ::=
           TRANSIENT
         | STATIC
         | AVERAGE
         | RMS
         | PEAK

connect_annotation ::=
           CONNECT_RULE connection_type

connection_type ::=
           MUST_SHORT        // short connection required
         | CAN_SHORT         // short connection allowed
         | CANNOT_SHORT      // short connection disallowed
```

3.6.8 Keywords for arithmetic models

The following keywords shall identify arithmetic model objects inside a `CELL`, a `WIRE` or a `VECTOR` object, i.e. output variables of an arithmetic model. Inside an arithmetic model object, the same keywords identify arguments, i.e. input variables to the arithmetic model. This gives virtually unlimited choice of combination of variables for characterization. All the keywords listed below are considered context-sensitive keywords.

`DELAY`	measured between two threshold points of two signals [non-negative float]
`SLEWRATE`	measured between two threshold point of one signal [non-negative float]
`SKEW`	same definition as `DELAY`, but it may have negative values [float]
`JITTER`	uncertainty of arrival time [non-negative float]
`SETUP`	setup timing constraint [float]
`HOLD`	hold timing constraint [float]
`RECOVERY`	signal recovery timing constraint [float]
`REMOVAL`	signal removal timing constraint [float]
`PULSEWIDTH`	pulse width timing constraint [float]
`PERIOD`	period timing constraint [float]

NOCHANGE	signal stability timing constraint [float]
THRESHOLD	reference point for DELAY, SLEWRATE and timing constraint measurement [float]
PROCESS	process derating coefficient [float]
TEMPERATURE	in ^{o}C [float]
VOLTAGE	in Voilts [float]
DERATE_CASE	derating case coefficient [float]
CURRENT	in Amps [float]
POWER	in Watts [float]
ENERGY	in Joules [float]
CAPACITANCE	pin, wire, or net capacitance [non-negative float]
RESISTANCE	pin, wire, or net resistance [non-negative float]
DRIVE_STRENGTH	drive strength of a signal [non-negative float]
SWITCHING_BITS	number of switching bits on a bus [non-negative integer]
FANOUT	number of receivers connected to a net [non-negative integer]
FANIN	number of drivers connected to a net [non-negative integer]
CONNECTIONS	number of pins connected to a net: CONNECTIONS = FANIN+FANOUT
TIME	[float]
FREQUENCY	[non-negative float]
AREA	[non-negative float]
DISTANCE	[float]
LENGTH	[non-negative float]
WIDTH	[non-negative float]
HEIGHT	[non-negative float]
CONNECTIVITY	connectivity function
DRIVER	argument of connectivity function
RECEIVER	argument of connectivity function

Use the following names as predefined process identifiers:

*n*p	process definition with strength characteristic for NMOS and PMOS

* can be s (=strong) or w (=weak)

The possible combinations are snsp, snwp, wnsp, wnwp.

Use the following names as predefined derating case identifiers:

nom	identifier for nominal case
bc*	prefix for best case
wc*	prefix for worst case
*com	suffix for commercial case
*ind	suffix for industrial case
*mil	suffix for military case

The possible combinations are bccom, bcind, bcmil, wccom, wcind, wcmil.

3.7 Library Organization

3.7.1 Scoping Rules

The following scope rules shall apply to all library objects and its usage.

Rule 1: An object shall be defined before it is referenced.

Rule 2: An ALF object shall be known (referenceable) inside the parent object, inside all objects defined after that object within the same parent object, and inside all the children of those objects.

Rule 3: An object definition with only a keyword but without an object identifier implies that the content of this definition will be applied to all objects identified by this keyword at the current scope and the underlying levels of hierarchy.

Example:

```
LIBRARY my_library {
   CAPACITANCE {UNIT = pF}      // default capacitance units for all
   ...                          // cells in my_library
   CELL cell1 {
      CAPACITANCE {UNIT = fF}   // capacitance units specific to cell1
      PIN A {CAPACITANCE = 10.5}
      ...
   }
   CELL cell2 {
      PIN A {CAPACITANCE = 0.010}  // default capacitance units
      ...
   }
}
```

The capacitance of pin `A` of `cell1` is `10.5 fF`. The capacitance of pin `A` of `cell2` is `0.010 pF`.

Rule 4: An object shall not be defined again at the same level of scope A definition of an object is considered duplicate, if both keyword and object identifier are identical.

Example:

It is illegal to write the following:

```
LIBRARY my_library {
   CAPACITANCE {UNIT = fF}
   ...
   CELL cell1 {
      pin A {CAPACITANCE = 10.5}
      ...
   }
   CAPACITANCE {UNIT = pF}      // duplicate definition
   CELL cell2 {
      pin A {CAPACITANCE = 0.010}
      ...
   }
}
```

There are three possible ways capacitance units can be set to fF for some of the cells in the library and pF for other cells in the same library:

1. put each set of cells in a different sublibrary,
2. define templates for the different units and reference them appropriately, or
3. define the units locally inside each cell.

3.7.2 Use of multiple files

Sometimes it is inconvenient or impractical to include all of the data for a technology library in a single file. The *INCLUDE* keyword is used to compose a library from multiple files.

An `INCLUDE` statement may be used within any context, but any included file shall contain at least a valid object definition to be considered a legal ALF file. It shall begin with a keyword, otherwise it may be ignored by a generic parser.

In general the effect of using the `INCLUDE` statement is to be considered equivalent to inserting the contents of the included file at that point in the parent file.

For example, a top-level ALF library file may contain only the following statements, where each file contains appropriate data to make up the entire library.

```
LIBRARY ‘mylib’ {
   INCLUDE ‘libdata.alf’
   INCLUDE ‘templates.alf’
   INCLUDE ‘cells.alf’
   INCLUDE ‘wiremodels.alf’
}
```

A complete ALF library definition must begin with the `LIBRARY` keyword. A list of cell definitions shall not be considered a full, legal ALF library database.

3.8 Referenceable objects

General referenceable objects within the scope of visibility are `TEMPLATE` and `GROUP`. Library-specific referenceable objects are `PIN`, `PRIMITIVE` and arithmetic model. The figure 3-24 shows relationships between these objects and where they can be referenced.

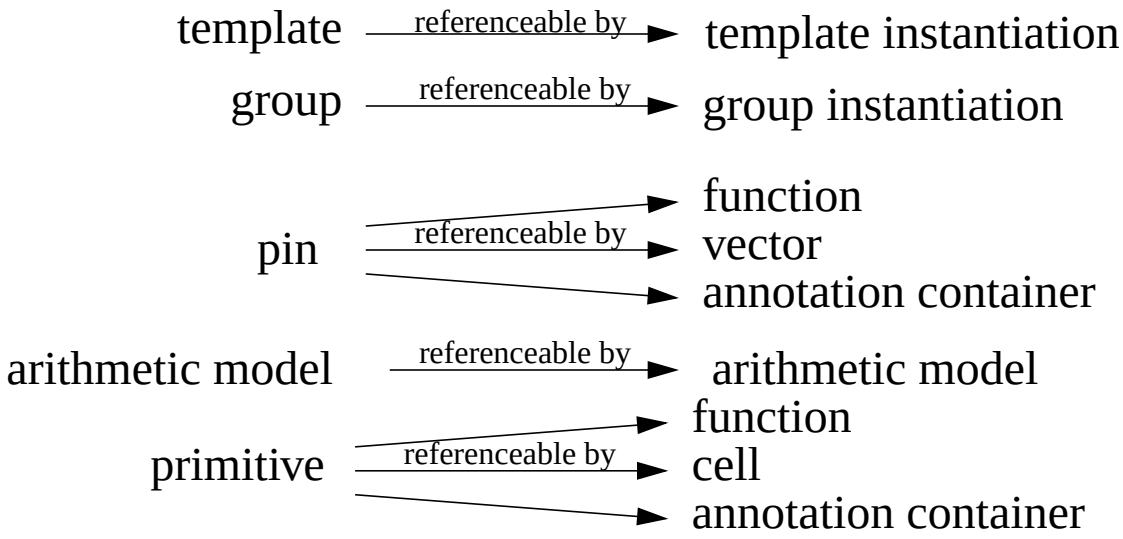

Figure 3-24: Referencing rules for ALF objects

The TEMPLATE and GROUP objects are referenceable only by their respective instantiation. The TEMPLATE definitions may contain instantiation of previously defined templates, which allows constructions of reusable objects.

The arithmetic models can be referenced by other arithmetic models, if they are contained within each other. This allows hierarchical modeling and a mix of table and equation based models.

The PIN objects are referenced within FUNCTION and VECTOR objects and within any annotation container inside the same CELL object.

The PRIMITIVEs are referenceable by a CELL in order to define pins and functionality or within a FUNCTION to define functionality only or within an annotation container, e.g. SCAN.

3.8.1 Referencing PRIMITIVEs or CELLs

A PRIMITIVE referenced in a CELL may replace the complete set of PIN and FUNCTION definition. PINs may be declared before the reference to the PRIMITIVE, in order to provide supplementary annotations that cannot be inherited from the PRIMITIVE. However, the CELL must be pin-compatible with the PRIMITIVE.

If the PRIMITIVE or a CELL is referenced in an annotation container such as SCAN, only the subset of PINs used in the non-scan cell must be compatible with the PINs of the cell.

The pin names can be referenced by order or by name. In the latter case, the LHS is the pin name of the referenced PRIMITIVE or CELL (e.g. the non-scan cell), the RHS is the pin name of the actual cell. A binary number can also appear at the LHS or RHS, indicating that a pin needs to be tied to a constant value. If this information is already specified in an annotation inside the PIN object itself, referencing between a pin name and a constant value is not necessary.

PRIMITIVEs can also be instantiated inside BEHAVIOR.

3.8.2 Referencing PINs in FUNCTIONs

Inside a CELL object, the PIN objects with the PINTYPE digital define variables for FUNCTION objects inside the same CELL. A *primary input variable* inside a FUNCTION must be declared as a PIN with DIRECTION=input or both (since DIRECTION=both is a bidirectional pin). However, it is not required that all declared pins are used in the function. Output variables inside a FUNCTION need not be declared pins, since they are implicitly declared when they appear at the left-hand side (LHS) of an assignment.

Example:

```
CELL my_cell {
   PIN A {DIRECTION = input}
   PIN B {DIRECTION = input}
   PIN C {DIRECTION = output}
   FUNCTION {
      BEHAVIOR {
      D = A && B;
      C = !D;
   }
}
```

C and D are output variables that need not be declared prior to use. After implicit declaration, D is reused as an input variable. A and B are primary input variables.

Inside BEHAVIOR, variables which appear at the LHS of an assignment conditionally controlled by a vector expression, as opposed to an unconditional continuous assignment, will hold their values, when the vector expression evaluates false. Those variables are considered to have latch-type behavior.

Examples:

```
BEHAVIOR {
   @(G){
      Q = D;  // both Q and QN have latch-type behavior
      QN = !Q;
   }
}

BEHAVIOR {
   @(G){
      Q = D;  // only Q has latch-type behavior
   }
   QN = !Q;
}
```

The functional description can be supplemented by a STATETABLE, the first row of which contains the arguments that are object IDs of declared PINs. The arguments appear in two fields, first is input, second is output. The fields are separated by colon (:). The rows are separated by (;). The arguments may appear in both fields, if the PINs have attribute direction=output or direction=both. If direction=output, then the argument has latch-type behavior. The argument on the input field is considered previous state, and the argument on the output field is considered the next state. If direction=both, then the argument on the input field applies for input direction, and the argument on the output field applies for output direction of the bidirectional PIN.

Example:

```
CELL ff_sd {
   PIN  q {DIRECTION=output}
   PIN  d {DIRECTION=input}
   PIN cp {DIRECTION=input
         ATTRIBUTE clock
         POLARITY=rising_edge}
   PIN cd {DIRECTION=input SIGNALTYPE=clear POLARITY=low}
   PIN sd {DIRECTION=input SIGNALTYPE=set POLARITY=low}
   FUNCTION {
      BEHAVIOR {
         @(!cd) {q = 0;} :(!sd) {q = 1;} :(01 cp) {q = d;}
      }
      STATETABLE {
         cd sd  cp  d   q  : q ;
         0  ?   ??  ?   ?  : 0 ;
         1  0   ??  ?   ?  : 1 ;
         1  1   1?  ?   0  : 0 ;
         1  1   ?0  ?   1  : 1 ;
         1  1   1?  ?   0  : 0 ;
         1  1   ?0  ?   1  : 1 ;
         1  1   01  ?   ?  :(d);
      }
   }
}
```

If the output variable with latch-type behavior depends only on the previous state of itself as opposed to the previous state of other output variables with latch-type behavior, it is not necessary to use that output variable in the input field. This allows a more compact form of the STATETABLE.

Example:

```
      STATETABLE {
         cd sd  cp  d  : q ;
         0  ?   ??  ?  : 0 ;
         1  0   ??  ?  : 1 ;
         1  1   1?  ?  :(q);
         1  1   ?0  ?  :(q);
         1  1   01  ?  :(d);
      }
```

A generic ALF parser must make the following semantic checks:

- Are all variables of a FUNCTION declared either by declaration as PIN names or through assignment?
- Does the STATETABLE exclusively contain declared PINs?
- Is the format of the STATETABLE, i.e. the number of elements in each field of each row, consistent?
- Are the values consistently either state or transition digits?

- Is the number of digits in each TABLE entry compatible with the signal bus width?

A more sophisticated checker for complete verification for logical consistency of a FUNCTION given in both equation and tabular representation is out of scope for a generic ALF parser, which checks only syntax and compliance to semantic rules. However, formal verification algorithms can be implemented in special-purpose ALF analyzers or model generators/ compilers.

3.8.3 Referencing PINs in VECTORs

Let us call the set of PINs of a CELL with PINTYPE=digital and SCOPE=both the *set of physical digital pins*. The *set of physical digital pins* defines the set of variables used by the VECTOR objects, since a VECTOR defines state, transition, or sequence of transitions of pins and those pins need to be physically controllable and observable for characterization.

For detection of a sequence of transitions it is necessary to observe the complete set of physical digital pins. For instance, if the set of physical digital pins consists of A, B, and C, the vector

```
(01 A -> 01 B)
```

implies, that no transition on C occurs between the transitions 01 A and 01 B.

There is a relation between the FUNCTION and the possible set of VECTORs for a CELL. Again, complete verification of the logical consistency is out of scope for the generic parser, but it will be considered in ALF analyzers or model generators/compilers.

3.8.4 Referencing arithmetic models

Input variables, also called *arguments of arithmetic models*, appear in the HEADER of the model. In the simplest case, the HEADER is just a list of arguments, each being a context-sensitive keyword. The model itself is also defined with a context-sensitive keyword.

The model can be in equation form. All arguments of the equation must be in the HEADER. The ALF parser should issue an error, if the EQUATION uses an argument not defined in the HEADER. A warning should be issued, if the HEADER contains arguments not used in the EQUATION.

Example:

```
DELAY {
   ...
   HEADER {
      CAPACITANCE {...}
      SLEWRATE {...}
   }
   EQUATION {
      0.01 + 0.3*SLEWRATE + (0.6 + 0.1*SLEWRATE)*CAPACITANCE
   }
}
```

If the model uses a TABLE, then each argument in the HEADER also needs a table in order to define the format. The order of arguments decides how the index to each entry is calculated.

The first argument is the innermost index, the following arguments are outer indices.

```
DELAY {
   HEADER {
      CAPACITANCE {
         TABLE {0.03 0.06 0.12 0.24}
      }
      SLEWRATE {
         TABLE {0.1 0.3 0.9}
      }
   }
   TABLE {
      0.07 0.10 0.14 0.22
      0.09 0.13 0.19 0.30
      0.10 0.15 0.25 0.41
   }
}
```

The first argument `load` has 4 entries. The second argument `ramptime` has 3 entries. Hence `DELAY` has 4*3=12 entries. For readability, comments may be inserted in the table.

```
   TABLE {
   //capacitance:0.03 0.06 0.12 0.24
   //            -------------------   slewrate:
               0.07 0.10 0.14 0.22 // 0.1
               0.09 0.13 0.19 0.30 // 0.3
               0.10 0.15 0.25 0.41 // 0.9
   }
```

Comments have no significance for the ALF parser, nor has the arrangement in rows and columns. Only the order of values is important for index calculation. The table can be made more compact by removing line breaks.

```
   TABLE { 0.07 0.10 0.14 0.22 0.09 0.13 0.19 0.30 0.10 0.15 0.25 0.41 }
```

For readability, the models and arguments can also have names, i.e. object IDs. For named objects, the name is used for referencing, rather than the keyword.

```
DELAY rise_out{
   ...
   HEADER {
      CAPACITANCE c_out {...}
      SLEWRATE fall_in {...}
   }
   EQUATION {
      0.01 + 0.3 * fall_in + (0.6 + 0.1* fall_in) * c_out
   }
}
```

The arguments of an arithmetic model can be arithmetic models themselves. In this way, combinations of `TABLE`- and `EQUATION`-based models can be used, for instance, in derating.

Coherent with `FUNCTION`, both `EQUATION` and `TABLE` representation of an arithmetic model are allowed. The `EQUATION` is intended to be used when the values of the arguments fall out of range, i.e. to avoid extrapolation. This is especially used in wire models.

3.9 Functional modeling styles and rules

ALF allows the following functional modeling styles: equation based, table-based, and primitive based. Both equation- and table-based functions are canonical and specify exactly the same functionality. Each primitive must be definable in either of the canonical modeling styles.

Since ALF supports both combinational and sequential functional specification using the 8-value logic system, an exhaustive behavioral description of all scenarios, which is needed for a simulation model, would be very cumbersome and defeat the purpose of a simple, easy-to-use language. Hence the following rules shall apply for compilation of the ALF description into a full simulation model. These rules cover all cases where the functional description is not explicit. All of these rules can be overruled by explicit specification of the behavior.

3.9.1 Rules for combinational functions

If a boolean expression evaluates `True`, the assigned output value is `1`. If a boolean expression evaluates `False`, the assigned output value is `0`. If the value of a boolean expression cannot be determined, the assigned output value is `X`. Assignment of values than `1`, `0`, or `X` must be specified explicitly.

For evaluation of the boolean expression, input value 'b`H` shall be treated as 'b`1`. Input value 'b`L` shall be treated as 'b`0`. All other input values shall be treated as 'b`X`.

Examples:

In equation form, these rules can be expressed as follows.

```
Z = A;
```

is equivalent to

```
Z = A ? 'b1 : 'b0;
```

More explicitly, this is also equivalent to

```
Z = (A=='b1 || A=='bH)? 'b1 : (A=='b0 || A=='bL)? 'b0 : 'bX;
```

In table form, this can be expressed as follows:

```
A : Z;
? : (A);
```

which is equivalent to

```
A : Z;
0 : 0;
1 : 1;
```

More explicitly, this is also equivalent to

```
A : Z;
0 : 0;
L : 0;
1 : 1;
H : 1;
X : X;
W : X;
Z : X;
U : X;
```

3.9.2 Rules for sequential functions

A sequential function is described in equation form by a boolean assignment with a condition specified by a vector expression. If the vector expression evaluates to `1` (`True`), the boolean assignment is activated, and the assigned output values follows the rules for combinational functions. If the vector expression evaluates to `0` (`False`), the output variables hold their assigned value from the previous evaluation. If the vector expression evaluates to `X`, the output variables area assigned an `X`.

For evaluation of the vector expression, input value 'b`H` shall be treated as 'b`1`. Input value 'b`L` shall be treated as 'b`0`. All other input values shall be treated as 'b`X`.

Examples:

In equation form, these rules can be expressed as follows.

```
@ (E) {Z = A;}
```

which is equivalent to

```
@ (E==’b1 || E==’bH) {Z = A;}
: (E!=’b0 && E!=’bL) {Z = ’bX;}
```

For a sequential function in table form, these rules can be expressed as follows:

```
 E A :  Z;
 0 ? : (Z);
 1 ? : (A);
```

which is equivalent to

```
E A :  Z;
0 ? : (Z);
L ? : (Z);
1 ? : (A);
H ? : (A);
X ? :  X;
W ? :  X;
Z ? :  X;
U ? :  X;
```

For edge-sensitive and higher level event sensitive functions, transitions from or to 'b`L` shall be treated like transitions from or to 'b`0`, and transitions from or to 'b`H` shall be treated like transitions from or to 'b`1`.

Not every transition may trigger the evaluation of a function. The set of vectors triggering the

evaluation of a function are called *active vectors*. From the set of active vectors, a set of *inactive vectors* can be derived, which will clearly not trigger the evaluation of a function.

Example:

For the following sequential function

```
@ (01 CP) { Z = A; }
```

the active vectors are

```
('b0'b1 CP)
('b0'bH CP)
('bL'b1 CP)
('bL'bH CP)
```

and the inactive vectors are

```
('b1'b0 CP)
('b1'bL CP)
('bH'b0 CP)
('bH'bL CP)
```

The union of active and inactive vectors is the set of *legal vectors*. Any vector outside the set of legal vectors is an *illegal vector*. An illegal vector can be obtained from a legal vector by replacing arbitrary binary literals with `X` or `W` or `Z`. Let us think of the replaced digits as wildcards. Depending on the number of wildcards, the illegal vector can be matched to one or multiple legal vectors.

Example:

Given the legal vectors above, examples of illegal vectors are:

```
('bZ'b1 CP) ('bL'bX CP) ('bH'bX CP) ('bW'b0 CP) ('bX'bZ CP)
```

Representing the replaced bits with wildcards, we obtain:

```
(*'b1 CP) ('bL* CP) ('bH* CP) (*'b0 CP) (** CP)
```

Possible match for `(*'b1 CP)` is `('b0'b1 CP)` or `('bL'b1 CP)`.
Possible match for `('bL* CP)` is `('bL'b1 CP)` or `('bL'bH CP)`.
Possible match for `('bH* CP)` is `('bH'b0 CP)` or `('bH'bL CP)`.
Possible match for `(*'b0 CP)` is `('b1'b0 CP)` or `('bH'b0 CP)`.
Possible match for `(** CP)` is anything.

Illegal vector output rule: If the set of possible matches for an illegal vector contains at least one active vector, the output shall be assigned `'bX`.

In this example, the illegal vectors `('bZ'b1 CP)`, `('bL'bX CP)`, and `('bX'bZ CP)` will assign 'bX, whereas `('bH'bX CP)` and `('bW'b0 CP)` will not assign `'bX`.

Rule for resolving ambiguity in sequential functions: No particular ordering of simultaneous events is implied. It is up to the model writer to ensure correct operation when simultaneous events occur.

3.10 Predefined primitives

Primitives are described in ALF syntax. Extensible primitives, i.e. primitives with variable

number of pins use symbolic placeholders such as <N> for the pin index. This syntax is not directly supported by ALF, since only constant values are allowed as pin index. If <N> is replaced by an arbitrary positive integer, the primitive definitions become legal ALF.

However, it is possible to use variable numbers of pins in legal ALF, if they are incorporated in templates, and the templates are then instantiated for the desired range of indices.

Example:

```
TEMPLATE {
   PRIMITIVE MY_PRIMITIVE {
      GROUP index {0:<N>}
   ... }
}

MY_PRIMITIVE {N = 7} //this is legal ALF
```

The purpose here is to show the functionality of predefined primitives. The user need not reproduce them, so template definitions and instantiations for the supported primitives are not shown.

For all predefined primitives, the following annotations are assumed to be defined at library level:

```
PIN {
   VIEW = functional
   SCOPE = behavioral
}
```

Only the annotations different from the annotations shown above are described in the primitive definitions.

All predefined primitives shall have the prefix ALF_ in their name, in order to distinguish them from user-defined primitives.

3.10.1 Combinational primitives

3.10.1.1 One input, multiple output primitives

There are two combinational primitives - ALF_BUF and ALF_NOT - with one input pin and multiple output pins.

The output pins are indexed starting with 0. If 0 is the only index used, the index can be omitted when referencing the output pin, e.g. out refers to out[0].

```
PRIMITIVE ALF_BUF {
   GROUP index {0:<N>}
   PIN[index] out {
      DIRECTION = output
   }
   PIN in {
      DIRECTION = input
   }
   FUNCTION alf_buf_behavior {
      BEHAVIOR {
         out[index] = in;
```

```
      }
   }
   FUNCTION alf_buf_statetable {
      STATETABLE {
         in : out[index];
         ?  : (in);
      }
   }
}
```

Figure 3-25: Primitive model of ALF_BUF

```
PRIMITIVE ALF_NOT {
   GROUP index {0:<N>}
   PIN[index] out {
      DIRECTION = output
   }
   PIN in {
      DIRECTION = input
   }
   FUNCTION alf_not_behavior {
      BEHAVIOR {
         out[index] = !in;
      }
   }
   FUNCTION alf_not_statetable {
      STATETABLE {
         in : out[index];
         ?  : (!in);
      }
   }
}
```

Figure 3-26: Primitive model of ALF_NOT

3.10.1.2 One output, multiple input primitives

There are six combinational primitives with one output pin and multiple input pins -

```
ALF_AND, ALF_NAND, ALF_OR, ALF_NOR, ALF_XOR, ALF_XNOR
```

The input pins are indexed starting with `0`. If `0` is the only index used, the index can be omitted when referencing the input pin, e.g. `in` refers to `in[0]`.

The models are defined in a pseudo-recursive way. If `<N>` is replaced by a positive integer, they become regular ALF models.

```
PRIMITIVE ALF_AND {
   GROUP index {0:<N>}
   GROUP index2 {1:<N>}
   PIN out {
      DIRECTION = output
   }
   PIN[index] in {
      DIRECTION = input
```

```
   }
   PIN[index] in2 {
      DIRECTION = output
      VIEW = none
   }
   FUNCTION alf_and_behavior {
      BEHAVIOR {
         in2[0] = in[0];
         out    = in2[<N>];
         index3 = index2 - 1;
         in2[index2] = in[index2] && in2[index3];
      }
   }
   FUNCTION alf_and_statetable {
      STATETABLE {
         in[index2] in2[index3] : in2[index2];
         0          ?           : 0;
         1          ?           : (in2[index3]);
      }
      STATETABLE {
         in2[<N>] : out;
         ?        : (in2[<N>]);
      }
   }
}
```

Figure 3-27: Primitive model of ALF_AND

```
PRIMITIVE ALF_NAND {
   GROUP index {0:<N>}
   GROUP index2 {1:<N>}
   PIN out {
      DIRECTION = output
   }
   PIN[index] in {
      DIRECTION = input
   }
   PIN[index] in2 {
      DIRECTION = output
      VIEW = none
   }
   FUNCTION {
      BEHAVIOR alf_nand_behavior {
         in2[0] = in[0];
         out    = ! in2[<N>];
         index3 = index2 - 1;
         in2[index2] = in[index2] && in2[index3];
      }
   }
   FUNCTION alf_nand_statetable {
      STATETABLE {
         in[index2] in2[index3] : in2[index2];
         0          ?           : 0;
         1          ?           : (in2[index3]);
```

```
      }
      STATETABLE {
         in2[<N>] : out;
         ?        : (!in2[<N>]);
      }
   }
}
```

Figure 3-28: Primitive model of ALF_NAND

```
PRIMITIVE ALF_OR {
   GROUP index {0:<N>}
   GROUP index2 {1:<N>}
   PIN out {
      DIRECTION = output
   }
   PIN[index] in {
      DIRECTION = input
   }
   PIN[index] in2 {
      DIRECTION = output
      VIEW = none
   }
   FUNCTION alf_or_behavior {
      BEHAVIOR {
         in2[0] = in[0];
         out    = in2[<N>];
         index3 = index2 - 1;
         in2[index2] = in[index2] || in2[index3];
      }
   }
   FUNCTION alf_or_statetable {
      STATETABLE {
         in[index2] in2[index3] : in2[index2];
         1          ?           : 1;
         0          ?           : (in2[index3]);
      }
      STATETABLE {
         in2[<N>] : out;
         ?        : (in2[<N>]);
      }
   }
}
```

Figure 3-29: Primitive model of ALF_OR

```
PRIMITIVE ALF_NOR {
   GROUP index {0:<N>}
   GROUP index2 {1:<N>}
   PIN out {
      DIRECTION = output
   }
   PIN[index] in {
      DIRECTION = input
```

```
    }
    PIN[index] in2 {
       DIRECTION = output
       VIEW = none
    }
    FUNCTION alf_nor_behavior {
       BEHAVIOR {
          in2[0] = in[0];
          out    = ! in2[<N>];
          index3 = index2 - 1;
          in2[index2] = in[index2] || in2[index3];
       }
    }
    FUNCTION alf_nor_statetable {
       STATETABLE {
          in[index2] in2[index3] : in2[index2];
          1          ?           : 1;
          0          ?           : (in2[index3]);
       }
       STATETABLE {
          in2[<N>] : out;
          ?        : (! in2[<N>]);
       }
    }
}
```

Figure 3-30: Primitive model of ALF_NOR

```
PRIMITIVE ALF_XOR {
   GROUP index {0:<N>}
   GROUP index2 {1:<N>}
   PIN out {
      DIRECTION = output
   }
   PIN[index] in {
      DIRECTION = input
   }
   PIN[index] in2 {
      DIRECTION = output
      VIEW = none
   }
   FUNCTION alf_xor_behavior {
      BEHAVIOR {
         in2[0] = in[0];
         out    = in2[<N>];
         index3 = index2 - 1;
         in2[index2] = in[index2] != in2[index3];
      }
   }
   FUNCTION alf_xor_statetable {
      STATETABLE {
         in[index2] in2[index3] : in2[index2];
         1          ?           : (!in2[index3]);
         0          ?           : (in2[index3]);
```

```
      }
      STATETABLE {
         in2[<N>] : out;
         ?        : (in2[<N>]);
      }
   }
}
```

Figure 3-31: Primitive model of ALF_XOR

```
PRIMITIVE ALF_XNOR {
   GROUP index {0:<N>}
   GROUP index2 {1:<N>}
   PIN out {
      DIRECTION = output
   }
   PIN[index] in {
      DIRECTION = input
   }
   PIN[index] in2 {
      DIRECTION = output
      VIEW      = none
   }
   FUNCTION alf_xnor_behavior {
      BEHAVIOR {
         in2[0] = in[0];
         out    = in2[<N>];
         index3 = index2 - 1;
         in2[index2] = in[index2] == in2[index3];
      }
   }
   FUNCTION alf_xnor_statetable {
      STATETABLE {
         in[index2] in2[index3] : in2[index2];
         1          ?           : (in2[index3]);
         0          ?           : (!in2[index3]);
      }
      STATETABLE {
         in2[<N>] : out;
         ?        : (in2[<N>]);
      }
   }
}
```

Figure 3-32: Primitive model of ALF_XNOR

3.10.2 Tristate Primitives

There are four tristate primitives -

```
ALF_BUFIF1, ALF_BUFIF0, ALF_NOTIF1, ALF_NOTIF0
```

```
PRIMITIVE ALF_BUFIF1 {
   PIN out {
      DIRECTION  = output
      ENABLE_PIN = enable
      ATTRIBUTE {tristate}
   }
   PIN in  {
      DIRECTION  = input
   }
   PIN enable {
      DIRECTION  = input
      SIGNALTYPE = out_enable
   }
   FUNCTION alf_bufif1_behavior {
      BEHAVIOR {
         out = (enable)? in : 'bZ;
      }
   }
   FUNCTION alf_bufif1_statetable {
      STATETABLE {
         enable in : out;
          0     ?  : Z;
          1     ?  : (in);
      }
   }
}
```

Figure 3-33: Primitive model of ALF_BUFIF1

```
PRIMITIVE ALF_BUFIF0 {
   PIN out {
      DIRECTION  = output
      ENABLE_PIN = enable
      ATTRIBUTE {tristate}
   }
   PIN in  {
      DIRECTION  = input
   }
   PIN enable {
      DIRECTION  = input
      SIGNALTYPE = out_enable
   }
   FUNCTION alf_bufif0_behavior {
      BEHAVIOR {
         out = (!enable)? in : 'bZ;
      }
   }
   FUNCTION alf_bufif0_statetable {
```

```
      STATETABLE {
         enable in : out;
          1     ?  : Z;
          0     ?  : (in);
      }
   }
}
```

Figure 3-34: Primitive model of ALF_BUFIF0

```
PRIMITIVE ALF_NOTIF1 {
   PIN out {
      DIRECTION  = output
      ENABLE_PIN = enable
      ATTRIBUTE {tristate}
   }
   PIN in  {
      DIRECTION  = input
   }
   PIN enable {
      DIRECTION  = input
      SIGNALTYPE = out_enable
   }
   FUNCTION alf_notif1_behavior {
      BEHAVIOR {
         out = (enable)? !in : 'bZ;
      }
   }
   FUNCTION alf_notif1_statetable {
      STATETABLE {
         enable in : out;
          0     ?  : Z;
          1     ?  : (!in);
      }
   }
}
```

Figure 3-35: Primitive model of ALF_NOTIF1

```
PRIMITIVE ALF_NOTIF0 {
   PIN out {
      DIRECTION  = output
      ENABLE_PIN = enable
      ATTRIBUTE {tristate}
   }
   PIN in  {
      DIRECTION  = input
   }
   PIN enable {
      DIRECTION  = input
      SIGNALTYPE = out_enable
   }
   FUNCTION alf_notif0_behavior {
      BEHAVIOR {
```

```
            out = (!enable)? !in : 'bZ;
         }
      }
      FUNCTION alf_notif0_statetable {
         STATETABLE {
            enable in : out;
             1     ?  : Z;
             0     ?  : (!in);
         }
      }
   }
```

Figure 3-36: Primitive model of ALF_NOTIF0

3.10.3 Multiplexor

```
   PRIMITIVE ALF_MUX {
      PIN Q  {
         DIRECTION  = output
         SIGNALTYPE = data
      }
      PIN[1:0] D  {
         DIRECTION  = input
         SIGNALTYPE = data
      }
      PIN S  {
         DIRECTION  = input
         SIGNALTYPE = select
      }
      FUNCTION alf_mux_behavior {
         BEHAVIOR {
            Q = (S)? d[1] : (!S || d[0]==d[1])? d[0] : 'bX;
         }
      }
      FUNCTION alf_mux_statetable {
         STATETABLE {
            D[0] D[1] S  : Z ;
            ?    ?    0  : (D[0]);
            ?    ?    1  : (D[1]);
            0    0    ?  : 0;
            1    1    ?  : 1;
         }
      }
   }
```

Figure 3-37: Primitive model of ALF_MUX

3.10.4 Flipflop

A dual-rail output D-flipflop with asynchronous set and clear pins is a generic edge-sensitive sequential device. Simpler flipflops can be modeled using this primitive by setting input pins to appropriate constant values. More complex flipflops can be modeled by adding combinational logic around the primitive.

A particularity of this model is the use of the last two pins Q_CONFLICT and QN_CONFLICT, which are virtual pins. They specify the state of Q and QN in the event CLEAR and SET become active simultaneously.

```
PRIMITIVE ALF_FLIPFLOP {
   PIN Q     {
      DIRECTION  = output
      SIGNALTYPE = data
      POLARITY   = non_inverted
   }
   PIN QN    {
      DIRECTION  = output
      SIGNALTYPE = data
      POLARITY   = inverted
   }
   PIN D     {
      DIRECTION  = input
      SIGNALTYPE = data
   }
   PIN CLOCK {
      DIRECTION  = input
      SIGNALTYPE = clock
      POLARITY   = rising_edge
   }
   PIN CLEAR {
      DIRECTION  = input
      SIGNALTYPE = clear
      POLARITY   = high
      ACTION     = asynchronous
   }
   PIN SET   {
      DIRECTION  = input
      SIGNALTYPE = set
      POLARITY   = high
      ACTION     = asynchronous
   }
   PIN Q_CONFLICT   {
      DIRECTION  = input
      VIEW       = none
   }
   PIN QN_CONFLICT  {
      DIRECTION  = input
      VIEW       = none
   }
   FUNCTION {
      ALIAS QX  Q_CONFLICT
      ALIAS QNX QN_CONFLICT
      BEHAVIOR alf_flipflop_behavior {
         @ (CLEAR && SET) {
            Q  = QX;
            QN = QNX;
         }
         : (CLEAR) {
            Q  = 0;
```

```
            QN = 1;
         }
         : (SET) {
            Q  = 1;
            QN = 0;
         }
         : (01 CLOCK) {         // edge-sensitive behavior
            Q  = D;
            QN = !D;
         }
   }
   FUNCTION alf_flipflop_statetable {
      STATETABLE {
         D CLOCK CLEAR SET QX  QNX :  Q    QN ;
         ?  ??    1     1  ?   ?   : (QX) (QNX);
         ?  ??    0     1  ?   ?   :  1    0 ;
         ?  ??    1     0  ?   ?   :  0    1 ;
         ?  1?    0     0  ?   ?   : (Q)  (QN) ;
         ?  ?0    0     0  ?   ?   : (Q)  (QN) ;
         ?  01    0     0  ?   ?   : (D)  (!D) ;
      }
   }
}
```

Figure 3-38: Primitive model of ALF_FLIPFLOP

3.10.5 Latch

The dual-rail D-latch with set and clear pins has the same functionality as the flipflop, except the level-sensitive clock (enable pin).

```
PRIMITIVE ALF_LATCH {
   PIN Q     {
      DIRECTION  = output
      SIGNALTYPE = data
      POLARITY   = non_inverted
   }
   PIN QN    {
      DIRECTION  = output
      SIGNALTYPE = data
      POLARITY   = inverted
   }
   PIN D     {
      DIRECTION  = input
      SIGNALTYPE = data
   }
   PIN ENABLE {
      DIRECTION  = input
      SIGNALTYPE = clock
      POLARITY   = high
   }
   PIN CLEAR {
      DIRECTION  = input
      SIGNALTYPE = clear
```

```
      POLARITY   = high
      ACTION     = asynchronous
   }
   PIN SET   {
      DIRECTION  = input
      SIGNALTYPE = set
      POLARITY   = high
      ACTION     = asynchronous
   }
   PIN Q_CONFLICT   {
      DIRECTION = input
      VIEW      = none
   }
   PIN QN_CONFLICT  {
      DIRECTION = input
      VIEW      = none
   }
   FUNCTION alf_latch_behavior {
      ALIAS QX  Q_CONFLICT
      ALIAS QNX QN_CONFLICT
      BEHAVIOR {
         @ (CLEAR && SET) {
            Q  = QX;
            QN = QNX;
         }
         : (CLEAR) {
            Q  = 0;
            QN = 1;
         }
         : (SET) {
            Q  = 1;
            QN = 0;
         }
         : (ENABLE) {        // level-sensitive behavior
            Q  = D;
            QN = !D;
         }
   }
   FUNCTION alf_latch_statetable {
      STATETABLE {
         D  ENABLE CLEAR SET QX  QNX :  Q    QN ;
         ?  ?      1     1   ?   ?   : (QX) (QNX);
         ?  ?      0     1   ?   ?   :  1    0 ;
         ?  ?      1     0   ?   ?   :  0    1 ;
         ?  0      0     0   ?   ?   : (Q)  (QN) ;
         ?  1      0     0   ?   ?   : (D)  (!D) ;
      }
   }
}
```

Figure 3-39: Primitive model of ALF_LATCH

Section 4
Applications

This section shows various examples of library cells modeled using ALF.

4.1 Truth Table vs Boolean Equation

A combinational logic cell and a sequential logic cell are shown below using two different constructs - truth table and boolean equation.

4.1.1 NAND gate

A 2-input NAND gate library cell can be modeled as shown below. The behavior of the cell can be modeled either as a `STATETABLE` or as a boolean equation.

Modeling a NAND gate using truth table:

```
CELL ND2 { /* 2 input NAND gate */
   PIN a {DIRECTION=input}
   PIN b {DIRECTION=input}
   PIN z {DIRECTION=output}

   STATETABLE {
      a  b : z ;
      0  ? : 1 ;
      1  ? : (!b);
   }
)
```

Modeling a NAND gate using boolean expression:

```
CELL ND2 { /* 2 input NAND gate */
   PIN a {DIRECTION=input}
   PIN b {DIRECTION=input}
   PIN z {DIRECTION=output}

   BEHAVIOR {
      z = !(a && b);
   }
)
```

4.1.2 Flipflop

A flipflop with asynchronous set and clear signals is shown below using truth table.

```
CELL FLIPFLOP {
   PIN CLEAR {DIRECTION=input SIGNALTYPE=clear POLARITY=low}
   PIN SET   {DIRECTION=input SIGNALTYPE=set   POLARITY=low}
   PIN CLOCK {DIRECTION=input SIGNALTYPE=clock POLARITY=rising_edge}
   PIN D     {DIRECTION=input}
   PIN Q     {DIRECTION=output}
   .../* One of the two behaviors below go here */
}
```

```
   STATETABLE {
      CLEAR SET CLOCK  D Q : Q;
      0     ?     ??   ? ? : 0;
      1     0     ??   ? ? : 1;
      1     1     01   ? ? : (d);
      1     1     1?   ? ? : (q);
      1     1     ?0   ? ? : (q);
   }
```

Modeling a flipflop with asynchronous set and clear using boolean expression:

```
   BEHAVIOR {
      @(!CLEAR) {Q = 0;} : (!SET) {Q = 1;} : (01 CLOCK) {Q = D;}
   }
```

4.2 Use of primitives

The functionality of a cell can be described using instances of other cells.

4.2.1 D-Flipflop with asynchronous clear

```
CELL d_flipflop_clr {
   PIN cd {DIRECTION=input SIGNALTYPE=clear POLARITY=low}
   PIN cp {DIRECTION=input SIGNALTYPE=clock POLARITY=rising_edge}
   PIN d  {DIRECTION=input}
   PIN q  {DIRECTION=output}
   .../* One of the two behaviors below go here */
}
```

Explicit description does not use instances of other cells defined in the library:

```
   BEHAVIOR {
      @(01 cp && cd) {q = d;}
      @(!cd) {q = 0;}
   }
```

Use of primitives permit derivation of new cells from other cells. Below, a D-Flipflop with asynchronous clear is derived from a D-Flipflop with asynchronous set and clear (see Section 4.1.2):

```
   BEHAVIOR {
      ALF_FLIPFLOP {CLOCK=cp D=d Q=q SET='b0 CLEAR=!cd}
   }
```

4.2.2 JK-flipflop

This example shows three ways of modeling a JK-Flipflop.

```
CELL jk_flipflop {
   PIN cp {DIRECTION=input SIGNALTYPE=clock POLARITY=rising_edge}
   PIN j  {DIRECTION=input}
   PIN k  {DIRECTION=input}
   PIN q  {DIRECTION=output}
   ...
}
```

Explicit description:

```
   BEHAVIOR {
      d =
         (!j &&  k) ?   0  :
         ( j && !k) ?   1  :
         ( j &&  k) ? !(q) :
         (!j && !k) ?  (q) :
                      'bx  ;
      @(01 cp) {q = d;}
   }
```

Use of primitives (assumes `ALF_MUX` cell is described in the library):

```
   BEHAVIOR {
      ALF_MUX {Q=d D0=j D1=!k SELECT=q}
      ALF_FLIPFLOP {CLOCK=cp D=d Q=q SET='b0 CLEAR='b0}
   }
```

Use of truth table:

```
   STATETABLE {
      cp j k q : (q) ;
      01 0 0 ? : (q) ;
      01 0 1 ? :  0  ;
      01 1 0 ? :  1  ;
      01 1 1 ? : (!q);
      1? ? ? ? : (q) ;
      ?0 ? ? ? : (q) ;
   }
```

4.2.3 D-Flipflop with synchronous load and clear

This example shows two different models of a synchronous D-Flipflop.

```
CELL d_flipflop_ld_clr {
   PIN cs {DIRECTION=input SIGNALTYPE=clear
           POLARITY=high   ACTION=synchronous}
   PIN ls {DIRECTION=input}
   PIN cp {DIRECTION=input SIGNALTYPE=clock POLARITY=rising_edge}
   PIN d  {DIRECTION=input}
   PIN q  {DIRECTION=output}
   ...
}
```

Explicit description:

```
BEHAVIOR {
   d1 = (ls)? d : q;
   d2 = d1 && !cs;
   @(01 cp) {q = d2;}
}
```

Use of primitives:

```
BEHAVIOR {
   ALF_MUX {Q=d1 D0=q D1=d SELECT=ls} /* Connection by pin name */
   ALF_AND {d2 d1 !cs}                /* Connection by pin order */
   ALF_FLIPFLOP {CLOCK=cp D=d2 Q=q SET='b0 CLEAR='b0 }
}
```

4.2.4 D-Flipflop with input multiplexor

This example shows three different modeling styles for a D-flipflop with input multiplexor, asynchronous set and asynchronous clear:

```
CELL d_flipflop_mux_set_clr {
   PIN sel {DIRECTION=input}
   PIN sd  {DIRECTION=input SIGNALTYPE=set   POLARITY=low}
   PIN cd  {DIRECTION=input SIGNALTYPE=clear POLARITY=low}
   PIN cp  {DIRECTION=input SIGNALTYPE=clock POLARITY=rising_edge}
   PIN d1  {DIRECTION=input}
   PIN d2  {DIRECTION=input}
   PIN q   {DIRECTION=output}
   ...
}
```

Explicit description:

```
BEHAVIOR {
   @(!cd) {q = 0;}
   @(!sd && cd) {q = 1;}
   @(01 cp && cd && sd) {q = (sel)? d1: d2;}
}
```

More efficient description can be created using priority assignments:

```
BEHAVIOR {
   @(!cd)  {q = 0;}
   :(!sd)  {q = 1;}
   :(01 cp){q = (sel)? d1: d2;}
}
```

Use of primitive:

```
BEHAVIOR {
   ALF_FLIPFLOP {CLOCK=cp D=((sel)? d1: d2) Q=q SET=!sd CLEAR=!cd}
}
```

Note that the use of `ALF_MUX` primitive is eliminated by using an assignment expression to D input in `ALF_FLIPFLOP` instance.

4.2.5 D-latch

This example shows a level-sensitive cell in two different styles.

```
CELL d_latch {
   PIN g {DIRECTION=input SIGNALTYPE=clock POLARITY=high}
   PIN d {DIRECTION=input}
   PIN q {DIRECTION=output}
   ...
}
```

Explicit description:

```
   BEHAVIOR {
      @(g) {q = d;}
   }
```

Use of primitive:

```
   BEHAVIOR {
      ALF_LATCH {ENABLE=g D=d Q=q SET='b0 CLEAR='b0}
   }
```

4.2.6 SR-latch

The example below shows how some of the input pins can be left unconnected if they represent don’t care situation.

```
CELL sr_latch {
   PIN sn {DIRECTION=input SIGNALTYPE=set   POLARITY=low}
   PIN rn {DIRECTION=input SIGNALTYPE=clear POLARITY=low}
   PIN q  {DIRECTION = output}
   PIN qn {DIRECTION = output}
   ...
}
```

Explicit description:

```
   BEHAVIOR {
      @ (!sn) {q  = 'b1; qn = !rn;}
      @ (!rn) {qn = 'b1; q  = !sn;}
   }
```

Use of primitive:

```
   BEHAVIOR {
      ALF_LATCH {ENABLE='b0 Q=q SET=!sn CLEAR=!rn}
   }
```

Since `ENABLE` pin is always set to `0`, the connection of `D` pin is irrelevant. Even if `D` is considered `'bX` or `'bZ`, the behavior will not change.

4.2.7 JTAG BSR

The following example shows a JTAG BSR cell with built-in scan chain.

```
CELL F10_18 {
   PIN SysOut {DIRECTION  = output}
   PIN TDO    {DIRECTION  = output SIGNALTYPE = scan_data}
   PIN SysIn  {DIRECTION  = input}
   PIN TDI    {DIRECTION  = input SIGNALTYPE = scan_data}
   PIN Shift  {DIRECTION  = input SIGNALTYPE = scan_enable}
   PIN Clk    {DIRECTION  = input POLARITY = rising_edge
               SIGNALTYPE = master_clock}
   PIN Update {DIRECTION  = input POLARITY = rising_edge
               SIGNALTYPE = slave_clock}
   PIN Mode   {DIRECTION  = input SIGNALTYPE = select}
   PIN STATE0 {  // This state is on the scan chain
               SCAN_POSITION = 1 DIRECTION = output VIEW = none}
   PIN STATE1 {  // NOT on scan chain (just update latch)
               DIRECTION = output VIEW = none}
   FUNCTION {
      BEHAVIOR {
         @(01 Clk) {STATE0 = Shift ? TDI : SysIn;}
         @(01 Update) {STATE1 = STATE0;}
         TDO = STATE0;
         SysOut = Mode ? STATE1 : SysIn;
      }
   }
}
```

4.2.8 Combinational Scan Cell

The following example shows a combinational scan cell with a reused primitive.

```
LIBRARY major_ASIC_vendor {
   INFORMATION {
      version = v2.1.0
      title = “0.35 standard cell”
      product = p35sc
      author = “Major Asic Vendor, Inc.”
      datetime = “Wed Jul 23 13:50:12 MST 1997”
   }
   ..
   CELL ND3A {
      INFORMATION {
         version = v6.0
         title = “3 input nand”
         product = p35sc_lib
         author = “Joe Cell Designer”
         datetime = “Tue Apr 1 01:39:47 PST 1997”
      }
      PIN Z {DIRECTION=output}
      PIN A {DIRECTION=input}
      PIN B {DIRECTION=input}
      PIN C {DIRECTION=input}
      FUNCTION {
```

```
            BEHAVIOR {
                      ALF_NAND {Z A B C}
            }
         }
         /* fill in timing and power data for ND3A cell */
      }
      ..
      CELL ND3B {
         PIN Z {DIRECTION=output}
         PIN A {DIRECTION=input}
         PIN B {DIRECTION=input}
         PIN C {DIRECTION=input}
         FUNCTION {
            BEHAVIOR {
                      ALF_NAND {Z A B C}
            }
         }
         /* fill in timing and power data for ND3B cell */
      }
      ..
      CELL SCAN_ND4 {
         PIN Z {DIRECTION=output}
         PIN A {DIRECTION=input}
         PIN B {DIRECTION=input}
         PIN C {DIRECTION=input}
         PIN D {DIRECTION=input SIGNALTYPE=scan_enable}

         SCAN_TYPE = control_0
         NON_SCAN_CELL = ALF_NAND {Z A B C}
         FUNCTION {
            BEHAVIOR { Z = !(A && B && C && D);}
         }
      }
      ..
   }
```

4.2.9 Scan Flipflop

The following example shows a scan flipflop using the generic `ALF_FLIPFLOP` primitive.

```
   LIBRARY major_ASIC_vendor {
      ...
      CELL F614 {
         PIN H01 {DIRECTION = input SIGNALTYPE = data}
         PIN H02 {DIRECTION = input SIGNALTYPE = clock}
         PIN H03 {DIRECTION = input SIGNALTYPE = clear POLARITY = high}
         PIN H04 {DIRECTION = input SIGNALTYPE = set POLARITY = high}
         PIN N01 {DIRECTION = output
                  SCAN {SIGNALTYPE = data POLARITY = non_inverted}}
         PIN N02 {DIRECTION = output POLARITY = inverted}
         FUNCTION {
            BEHAVIOR {
               ALF_FLIPFLOP {D=H01 CLOCK=H02 CLEAR=H03 SET=H04
                             Q=N01 QN=N02 Q_CONFLICT='bX QN_CONFLICT='bX}
```

```
            }
         }
      }
      ...
      CELL S000 {
         PIN H01 {DIRECTION = input  SIGNALTYPE = scan_data}
         PIN H02 {DIRECTION = input  SIGNALTYPE = clock
                  OFFSTATE  = non_inverted}
         PIN H03 {DIRECTION = input  SIGNALTYPE = scan_enable
                  POLARITY  = low}
         PIN H04 (DIRECTION = input  SIGNALTYPE = set   POLARITY = high}
         PIN H05 {DIRECTION = input  SIGNALTYPE = clear POLARITY = high}
         PIN H06 {DIRECTION = input  SIGNALTYPE = data}
         PIN N01 {DIRECTION = output SIGNALTYPE = data
                  POLARITY  = non_inverted}
         PIN N02 {DIRECTION = output POLARITY = inverted}
         FUNCTION{
            BEHAVIOR {
               ALF_MUX {Q=flipflop_d D0=H06 D1=H01 SELECT=H03}
               ALF_FLIPFLOP {D=flipflop_d CLOCK=H02 CLEAR=H05 SET=H04
                             Q=N01 QN=N02 Q_CONFLICT='bX QN_CONFLICT='bX}
            }
         }
         SCAN_TYPE = muxscan
         NON_SCAN_CELL = ALF_FLIPFLOP {D=H06 CLOCK=H02 CLEAR=H05 SET=H04
                                       Q=N01 QN=N02 Q_CONFLICT='bX
                                       QN_CONFLICT='bX 'b0=H03 'b0=H01}
      }
      ...
   }
```

4.2.10 Quad D-Flipflop

The following example shows a quad D-Flipflop with and without built-in scan chain.

```
   LIBRARY major_ASIC_vendor {
      PRIMITIVE FFX4 {
         PIN CK { DIRECTION = input }
         PIN D0 { DIRECTION = input }
         PIN D1 { DIRECTION = input }
         PIN D2 { DIRECTION = input }
         PIN D3 { DIRECTION = input }
         PIN Q0 { DIRECTION = output }
         PIN Q1 { DIRECTION = output }
         PIN Q2 { DIRECTION = output }
         PIN Q3 { DIRECTION = output }
         FUNCTION {
            BEHAVIOR {
               ALF_FLIPFLOP {Q=Q0 D=D0 CLOCK=CK SET='b0 CLEAR='b0}
               ALF_FLIPFLOP {Q=Q1 D=D1 CLOCK=CK SET='b0 CLEAR='b0}
               ALF_FLIPFLOP {Q=Q2 D=D2 CLOCK=CK SET='b0 CLEAR='b0}
               ALF_FLIPFLOP {Q=Q3 D=D3 CLOCK=CK SET='b0 CLEAR='b0}
            }
         }
```

```
    }
    CELL SCAN_FFX4 {
       PIN OUT0  {DIRECTION = output}
       PIN OUT1  {DIRECTION = output}
       PIN OUT2  {DIRECTION = output}
       PIN OUT3  {DIRECTION = output}
       PIN SO  {DIRECTION = output SIGNALTYPE = scan_data}
       PIN IN0  {DIRECTION = input SIGNALTYPE = data}
       PIN IN1  {DIRECTION = input SIGNALTYPE = data}
       PIN IN2  {DIRECTION = input SIGNALTYPE = data}
       PIN IN3  {DIRECTION = input SIGNALTYPE = data}
       PIN CLK {DIRECTION = input SIGNALTYPE = clock}
       PIN SI  {DIRECTION = input SIGNALTYPE = scan_data}
       PIN SE  {DIRECTION = input SIGNALTYPE = scan_enable}
       PIN STATE0 {SCAN_POSITION = 1 DIRECTION = output VIEW = none}
       PIN STATE1 {SCAN_POSITION = 2 DIRECTION = output VIEW = none}
       PIN STATE2 {SCAN_POSITION = 3 DIRECTION = output VIEW = none}
       PIN STATE3 {SCAN_POSITION = 4 DIRECTION = output VIEW = none}
       FUNCTION {
          BEHAVIOR {
             OUT0 = STATE0; OUT1 = STATE1; OUT2 = STATE2; OUT3 = STATE3;
             SO = !STATE3;
             @(01 CLK) {
                STATE0 = SE ? !SI     : IN0;
                STATE1 = SE ? !STATE0 : IN1;
                STATE2 = SE ? !STATE1 : IN2;
                STATE3 = SE ? !STATE2 : IN3;
             }
          }
       }
       SCAN_TYPE = muxscan
       NON_SCAN_CELL = FFX4 {CLK IN0 IN1 IN2 IN3 OUT0 OUT1 OUT2 OUT3}
       } // this example shows referencing by order
    }
}
```

4.3 Templates and vector-specific models

4.3.1 Vector specific delay and power Tables

In this example, the use of vector specific models for input-to-output delay, output ramptime, and switching energy is shown.

```
CELL nand2 {
   PIN a {DIRECTION = input CAPACITANCE = 0.02 {UNIT = pF}}
   PIN b {DIRECTION = input CAPACITANCE = 0.02 {UNIT = pF}}
   PIN z {DIRECTION = output}
   FUNCTION {
      BEHAVIOR {z = !(a && b); }
   }
   VECTOR (10 a -> 01 z){    /* Vector specific characterization */
      DELAY {
         UNIT = ns
```

```
      FROM {PIN = a  THRESHOLD = 0.4}
      TO   {PIN = z  THRESHOLD = 0.6}
      HEADER {
         CAPACITANCE {
            PIN = z UNIT = pF
            TABLE {0.01 0.02 0.04 0.08 0.16}
         }
         SLEWRATE {
            PIN = a UNIT = ns
            FROM {THRESHOLD = 0.5}
            TO   {THRESHOLD = 0.3}
            TABLE {0.1 0.3 0.9}
         }
      }
      TABLE {
            0.1 0.2 0.4 0.8 1.6
            0.2 0.3 0.5 0.9 1.7
            0.4 0.5 0.7 1.1 1.9
      }
   }
   SLEWRATE {
      PIN = z UNIT = ns
      FROM {THRESHOLD = 0.3}
      TO   {THRESHOLD = 0.5}
      HEADER {
         CAPACITANCE {
            PIN = z UNIT = pF
            TABLE {0.01 0.02 0.04 0.08 0.16}
         }
         SLEWRATE {
            PIN = a UNIT = ns
            FROM {THRESHOLD = 0.5}
            TO   {THRESHOLD = 0.3}
            TABLE {0.1 0.3 0.9}
         }
      }
      TABLE {
            0.1 0.2 0.4 0.8 1.6
            0.1 0.2 0.4 0.8 1.6
            0.2 0.4 0.6 1.0 1.8
      }
   }
   ENERGY {
      UNIT = pJ
      HEADER {
         CAPACITANCE {
            PIN = z UNIT = pF
            TABLE {0.01 0.02 0.04 0.08 0.16}
         }
         SLEWRATE {
            PIN = a UNIT = ns
            FROM {THRESHOLD = 0.5}
            TO   {THRESHOLD = 0.3}
            TABLE {0.1 0.3 0.9}
```

```
            }
         }
         TABLE {
               0.21 0.32 0.64 0.98 1.96
               0.22 0.33 0.65 0.99 1.97
               0.31 0.42 0.74 1.08 2.06
         }
      }
   }
   VECTOR (01 a -> 10 z){
      DELAY    { ... }
      SLEWRATE { ... }
      ENERGY   { ... }
   }
   VECTOR (10 b -> 01 z){
      DELAY    { ... }
      SLEWRATE { ... }
      ENERGY   { ... }
   }
   VECTOR (01 b -> 10 z){
      DELAY    { ... }
      SLEWRATE { ... }
      ENERGY   { ... }
   }
}
```

4.3.2 Use of TEMPLATE

Notice that the header for the delay, ramptime, and energy models was the same in the example above. Therefore creating a template definition can eliminate duplicate information, reduce the possibility of inadvertent errors, and make the models compact. For example, a header template can be created as shown below:

```
TEMPLATE std_header_2d {
   HEADER {
      CAPACITANCE {
         PIN = <out_pin> UNIT = pF
         TABLE {0.01 0.02 0.04 0.08 0.16}
      }
      SLEWRATE {
         PIN = <in_pin> UNIT = ns
         FROM {THRESHOLD {RISE = 0.3  FALL = 0.5} }
         TO   {THRESHOLD {RISE = 0.5  FALL = 0.3} }
         TABLE {0.1 0.3 0.9}
      }
   }
```

The use of TEMPLATE eliminates the repetition of header information by rewriting the previous example (only the first vector) as shown below.

```
        DELAY {
           UNIT = ns
           THRESHOLD {RISE=0.4 FALL=0.6}
           FROM {PIN = a}
           TO   {PIN = z}
           std_header_2d {     /* Template is used */
              in_pin = a
              out_pin = z
           }
           TABLE {
                 0.1 0.2 0.4 0.8 1.6
                 0.2 0.3 0.5 0.9 1.7
                 0.4 0.5 0.7 1.1 1.9
           }
        }
        SLEWRATE {
           PIN = z UNIT = ns
           FROM {THRESHOLD {RISE = 0.3  FALL = 0.5} }
           TO   {THRESHOLD {RISE = 0.5  FALL = 0.3} }
           std_header_2d {     /* Template is used */
              in_pin = a
              out_pin = z
           }
           TABLE {
                 0.1 0.2 0.4 0.8 1.6
                 0.1 0.2 0.4 0.8 1.6
                 0.2 0.4 0.6 1.0 1.8
           }
        }
        ENERGY {
           UNIT = pJ
           std_header_2d {     /* Template is used */
              in_pin = a
              out_pin = z
           }
           TABLE {
                 0.21 0.32 0.64 0.98 1.96
                 0.22 0.33 0.65 0.99 1.97
                 0.31 0.42 0.74 1.08 2.06
           }
        }
     }
```

Note that the entire characterization model for CELL nand2 is the same for each vector (i.e. pair of input and output pins), so further efficiency can be achieved by defining the

characterization model itself as a template. This template definition uses the instantiation of the previously defined header template.

```
TEMPLATE std_char_2d {
   DELAY {
      UNIT = ns
      THRESHOLD {RISE=0.4 FALL=0.6}
      FROM {PIN = <in_pin>  }
      TO   {PIN = <out_pin> }
      std_header_2d {
         in_pin = <in_pin>
         out_pin = <out_pin>
      }
      TABLE <delay_data>
   }
   SLEWRATE {
      PIN = <out_pin> UNIT = ns
      FROM {THRESHOLD {RISE = 0.3  FALL = 0.5} }
      TO   {THRESHOLD {RISE = 0.5  FALL = 0.3} }
      std_header_2d {
         in_pin = <in_pin>
         out_pin = <out_pin>
      }
      TABLE <ramptime_data>
   }
   ENERGY {
      UNIT = pJ
      std_header_2d {
         in_pin = <in_pin>
         out_pin = <out_pin>
      }
      TABLE <energy_data>
   }
}
```

Now only the delay, ramptime and energy models contain specific data that is different for each vector. All repetitive information is in the template definition. The characterization model can be rewritten compactly as shown below:

```
      std_char_2d {
         in_pin = a
         out_pin = z
         delay_data = {
               0.1 0.2 0.4 0.8 1.6
               0.2 0.3 0.5 0.9 1.7
               0.4 0.5 0.7 1.1 1.9
         }
         ramptime_data = {
               0.1 0.2 0.4 0.8 1.6
               0.1 0.2 0.4 0.8 1.6
               0.2 0.4 0.6 1.0 1.8
```

```
      }
      energy_data = {
            0.21 0.32 0.64 0.98 1.96
            0.22 0.33 0.65 0.99 1.97
            0.31 0.42 0.74 1.08 2.06
      }
   }
```

4.3.3 Vector description styles for timing arcs

In previous examples, the vectors were specified as timing arcs. This is not ambiguous, since the sequence of transitions can only happen under one test condition.

```
VECTOR (10 a -> 01 z){
   std_char_2d { ... }
}
VECTOR (01 a -> 10 z){
   std_char_2d { ... }
}
VECTOR (10 b -> 01 z){
   std_char_2d { ... }
}
VECTOR (01 b -> 10 z){
   std_char_2d { ... }
}
```

An alternate way of describing the above vectors is to specify the input transition and the state of the other input(s) which control the output transition.

```
VECTOR (10 a && b){
   std_char_2d { ... }
}
VECTOR (01 a && b){
   std_char_2d { ... }
}
VECTOR (10 b && a){
   std_char_2d { ... }
}
VECTOR (01 b && a){
   std_char_2d { ... }
}
```

A redundant yet safe way of vector description is to specify both output transition and input state(s) together with the input transition.

```
VECTOR (10 a -> 01 z && b){
   std_char_2d { ... }
}
VECTOR (01 a -> 10 z && b){
   std_char_2d { ... }
}
VECTOR (10 b -> 01 z && a){
   std_char_2d { ... }
}
VECTOR (01 b -> 10 z && a){
   std_char_2d { ... }
}
```

In the non-redundant specification, either the input state or the output transition can be derived from the functional description.

4.3.4 Vectors for delay, power and timing constraints

A D-Flipflop model without the set and clear signals is shown below. This model has vectors for specific purpose - some for delay and power, some for power only (output is not switching), and some for timing constraints. However, each vector has the same structure, although the input variables change. The vectors for delay and power model require 2-dimensional tables with load capacitance and input ramptime as variables, the vectors for power model require 1-dimensional tables with input ramptime as variable, and the vectors for time constraints require 2-dimensional tables with ramptime on two inputs as variables.

```
CELL d_flipflop {
   PIN cp {DIRECTION = input}
   PIN d  {DIRECTION = input}
   PIN q  {DIRECTION = output}
   FUNCTION {
      BEHAVIOR { @(01 cp) {q = d; } }
   }
   VECTOR (01 cp -> 01 q) {
      /* fill in arithmetic models for delay and power */
   }
   VECTOR (01 cp -> 10 q) {
      /* fill in arithmetic models for delay and power */
   }
   VECTOR (01 cp && d == q) {
      /* fill in arithmetic model for power */
   }
   VECTOR (10 cp && d == q) {
      /* fill in arithmetic model for power */
   }
   VECTOR (10 cp && d != q) {
      /* fill in arithmetic model for power */
   }
   VECTOR (01 d && !cp) {
      /* fill in arithmetic model for power */
   }
```

```
      VECTOR (10 d && !cp) {
         /* fill in arithmetic model for power */
      }
      VECTOR (01 d && cp) {
         /* fill in arithmetic model for power */
      }
      VECTOR (10 d && cp) {
         /* fill in arithmetic model for power */
      }
      VECTOR (01 d <&> 01 cp)
         SETUP {
            /* fill in arithmetic model for setup time constraint */
            VIOLATION {
               BEHAVIOR {q = 'bx;}
               MESSAGE_TYPE = error
               MESSAGE = “setup violation 01 d <-> 01 cp“
            }
         }
         HOLD {
            /* fill in arithmetic model for hold time constraint */
            VIOLATION {
               BEHAVIOR {q = 'bx;}
               MESSAGE_TYPE = error
               MESSAGE = “hold violation 01 d <-> 01 cp“
            }
         }
      VECTOR (10 d <&> 01 cp)
         SETUP {
            /* fill in arithmetic model for setup time constraint */
            VIOLATION {
               BEHAVIOR {q = 'bx;}
               MESSAGE_TYPE = error
               MESSAGE = “setup violation 10 d <-> 01 cp“
            }
         }
         HOLD {
            /* fill in arithmetic model for hold time constraint */
            VIOLATION {
               BEHAVIOR {q = 'bx;}
               MESSAGE_TYPE = error
               MESSAGE = “hold violation 10 d <-> 01 cp“
            }
         }
      }
   }
```

4.4 Combining tables and equations

4.4.1 Table vs equation

The following examples show the usage of `TABLE` and `EQUATION` in the model.

Example with table:

```
CURRENT {
   PIN = VDD
   UNIT = mA
   TIME = 30 {UNIT = ns}
   MEASUREMENT = average
   HEADER {
      CAPACITANCE {
         PIN = z UNIT = pF
         TABLE {0.02 0.04 0.08 0.16}
      }
      SLEWRATE {
         PIN = a UNIT = ns
         TABLE {0.1 0.3 0.9}
      }
   }
   TABLE {
      0.0011 0.0021 0.0041 0.0081
      0.0013 0.0023 0.0043 0.0083
      0.0019 0.0029 0.0049 0.0089
   }
}
```

Equivalent example with equation:

```
CURRENT {
   PIN = VDD UNIT = mA
   TIME = 30 {UNIT = ns}
   MEASUREMENT = average
   HEADER {
      CAPACITANCE {PIN = z UNIT = pF}
      SLEWRATE    {PIN = a UNIT = ns}
   }
   EQUATION {0.05*CAPACITANCE + 0.001*SLEWRATE}
}
```

If the model uses an `EQUATION`, then each argument must appear in the `HEADER`. If the model uses a `TABLE`, then the `HEADER` must contain a `TABLE` for each argument. The number of values in the main table and the indexing scheme is defined by the order and the number of values in each table inside the header.

4.4.2 Cell with Multiple Output Pins

The following example shows how to use combinations of tables and equations for efficient modeling of energy consumption of a cell with two (buffered) outputs. When two outputs are switching, triggered by the same input, the dynamic energy consumption depends on ramptime of the input signal and load capacitance on each output.

Instead of creating a 3-dimensional table, two 2-dimensional tables are used, varying the load capacitance at one output and keeping zero load at the other output. The equation calculates the energy for both outputs switching by adding the values from each table together for the applicable load capacitance and by subtracting a corresponding correction term. The result is exact for cells with buffered outputs.

As shown in the example below, an arithmetic model must be a named object, if several objects of the same type occur within the same scope (e.g. `ENERGY`). For named objects, the equation uses the object name instead of the object type.

```
VECTOR (01 ci -> (01 co <-> 10 s) & a) {
   ENERGY {
      UNIT = pJ
      HEADER {
         ENERGY energy_co {        // named object
            UNIT = pJ
            HEADER {
               CAPACITANCE {
                  PIN = co UNIT = pF
                  TABLE { ... }
               }
               SLEWRATE {
                  PIN = ci UNIT = ns
                  TABLE { ... }
               }
            }
            TABLE { ... }
         }
         ENERGY energy_s {         // named object
            UNIT = pJ
            HEADER {
               CAPACITANCE {
                  PIN = s UNIT = pF
                  TABLE { ... }
               }
               SLEWRATE {
                  PIN = ci UNIT = ns
                  TABLE { ... }
               }
            }
            TABLE { ... }
         }
         ENERGY energy_noload {    // named object
            UNIT = pJ
            HEADER {
               SLEWRATE {
                  PIN = ci UNIT = ns
                  TABLE { ... }
               }
            }
            TABLE { ... }
         }
      }
      EQUATION {energy_co + energy_s - energy_noload}
   }
}
```

4.4.3 PVT Derating

Combinations of tables and equations can also be used for derating with respect to voltage and temperature, since those variables would add more dimensions to a purely table-based model.

In this example, the DELAY objects must be named, since there is both a nominal and a derated DELAY.

```
DELAY rise_out{
   HEADER {
      PROCESS {
         HEADER {nom snsp snwp wnsp wnwp}
         TABLE  {0.0 -0.1 -0.2 +0.3 +0.2}
      }
      VOLTAGE {//fill in any annotations
      }
      TEMPERATURE {//fill in any annotations
      }
      DELAY nom_rise_out {
         HEADER {
            CAPACITANCE {
               TABLE {0.03 0.06 0.12 0.24}
            }
            SLEWRATE {
               TABLE {0.1 0.3 0.9}
            }
         }
         TABLE {
            0.07 0.10 0.14 0.22
            0.09 0.13 0.19 0.30
            0.10 0.15 0.25 0.41
         }
      }
   }
   EQUATION {
      nom_rise_out
      * (1 + process)
      * (1 + (temperature - 273)*0.001)
      * (1 + (voltage - 3.3)*(-0.3))
   }
}
```

The HEADER in the process object contains exclusively named variables (nom, snsp...), similar to the truth table of a FUNCTION that contains only pin names. Therefore the TABLE is expected to have as many entries as the HEADER. The TABLE inside nom_rise_out must follow the format defined by each TABLE inside the declarations of load and ramptime. Other declared object in the HEADER would be ignored for the TABLE format, if they do not have a TABLE inside themselves.

For convenience, the derating equation can be defined as a template for future reuse.

```
TEMPLATE std_derating {
   EQUATION {
      <variable>
      * (1 + <Kp>)
      * (1 + (TEMPERATURE - 273)*<Kt>)
      * (1 + (VOLTAGE - 3.3)*<Kv>)
   }
}
```

Instantiation of the template in the model:

```
DELAY rise_out{
   HEADER {
      PROCESS {
         HEADER {nom snsp snwp wnsp wnwp}
         TABLE  {0.0 -0.1 -0.2 +0.3 +0.2}
      }
      VOLTAGE     { ... }
      TEMPERATURE { ... }
      DELAY nom_rise_out {
         HEADER {
            CAPACITANCE {TABLE { ... }}
            SLEWRATE    {TABLE { ... }}
         }
         TABLE { ... }
   }
   std_derating {
      variable = nom_rise_out
      Kp = PROCESS
      Kt = 0.001
      Kv = -0.3
   }
}
```

It is possible to assign explicit values to the predefined process and derating case identifiers.

Example:

```
PROCESS snsp = 0.9
PROCESS wnwp = 1.1

TEMPERATURE nom = 25
VOLTAGE nom = 3.3

TEMPERATURE bccom = 0
VOLTAGE bccom = 3.5

TEMPERATURE wcmil = 125
VOLTAGE wcmil = 2.8
```

It is also possible to express voltage, temperature and delay with the derating case as an independent variable:

```
VOLTAGE {
   HEADER {nom bccom wcmil}
   TABLE {3.3 3.5 2.8}
}
TEMPERATURE {
   HEADER {nom bccom wcmil}
   TABLE  {25 0 125}
}
DELAY {
   HEADER {
      DERATE_CASE {
         HEADER {nom bccom  wcmil}
         TABLE  {0 -0.0835 0.265}
      }
      PROCESS
         HEADER {nom snsp snwp wnsp wnwp}
         TABLE  {0.0 -0.1 -0.2 +0.3 +0.2}
      }
      DELAY nom_rise_out { ... }
   }
   EQUATION {
      nom_rise_out
      * (1 + PROCESS)
      * (1 + DERATE_CASE)
   }
```

Yet another possibility is a completely tabulated model, where the process and derating identifiers can be directly used as table items.

```
DELAY {
   HEADER {
      DERATE_CASE {
         TABLE {nom bccom wcmil}
      }
      PROCESS
         TABLE {nom snsp snwp wnsp wnwp}
      }
   TABLE {
      // 3*5 = 15 values
   }
```

4.5 Use of Annotations

4.5.1 Annotations for a PIN

Direct annotation:

```
PIN data_in {DIRECTION = input THRESHOLD = 0.35 CAPACITANCE = 0.010}
```

Using annotation containers:

```
PIN data_in {DIRECTION = input
             THRESHOLD = 0.35
             CAPACITANCE = 0.010 {
                  UNIT = pF   MEASUREMENT = average
                  MIN = 0.009 TYP = 0.010 MAX = 0.012
             }
             LIMIT {SLEWRATE {MAX=3.0 UNIT=ns}
                    VOLTAGE  {MAX=3.5 MIN=-0.2}
             }
}
```

The input pin `data_in` has a non-linear capacitance which was characterized using an average type of measurement. Different measurements yield average capacitances between 0.009 pF and 0.012 pF, typical average capacitance is 0.010 pF. The slewrate applied to the pin must not exceed 3.0 ns. The voltage swing must not exceed the lower bound of -0.2 V and the upper bound of 3.5 volt.

```
CAPACITANCE {UNIT = pF}
PIN data_out {
   DIRECTION = output CAPACITANCE = 0.002
   LIMIT {CAPACITANCE = 0.96}
}
```

The output pin data_out has a capacitance of 0.002 pF. The maximum load capacitance that may be applied to the pin is 0.96 pF.

4.5.2 Annotations for a timing arc

Specifications for a particular timing arc references specific pins:

```
DELAY {
   UNIT = ns
   FROM {PIN = data_in  THRESHOLD = 0.4}
   TO   {PIN = data_out THRESHOLD = 0.6}
}

SLEWRATE {
   PIN = data_out  UNIT = ns
   FROM {THRESHOLD = 0.3}
   TO   {THRESHOLD = 0.5}
}
```

Specifications for a generic timing arc does not reference specific pins, but values for both switching directions must be defined):

```
DELAY {
   UNIT = ns
   THRESHOLD {RISE=0.4 FALL=0.6}
}

SLEWRATE {
   UNIT = ns
   FROM {THRESHOLD {RISE=0.3 FALL=0.5}}
   TO   {THRESHOLD {RISE=0.5 FALL=0.3}}
}
```

4.5.3 Creating Self-explaining Annotations

The self-explaining annotations can be created using TEMPLATE.

Example: number of connections allowed for each pin

```
TEMPLATE must_connect {
      LIMIT {CONNECTION {MIN = 1}}
}
TEMPLATE can_float {
      LIMIT {CONNECTION {MIN = 0}}
}
TEMPLATE no_connection {
      LIMIT {CONNECTION {MAX = 0}}
}
CELL a_flipflop {
   PIN q  {must_connect  DIRECTION=output}
   PIN qn {can_float     DIRECTION=output}
   PIN qi {no_connection DIRECTION=output}
   ...
}
```

4.6 Providing fallback position for applications

4.6.1 Use of DEFAULT

ALF's modeling capabilities address the needs for all types of applications. However, ALF should also work for applications that use only a subset of information. In order to make the subset of information controllable, modeling capability with DEFAULT is provided. The information provided by DEFAULT can be strictly ignored by applications that understand the full information.

A particular application may not be able to use 3-dimensional tables, or it may not understand certain models. DEFAULT values can be provided for each model.

Example:

```
DELAY {
   HEADER {
      SLEWRATE {
         PIN = a UNIT = 1e-9
         TABLE {0.5  1.0  1.5}
         DEFAULT = 1.0
      }
      CAPACITANCE {
         PIN = z UNIT = 1e-12
         TABLE {0.1  0.2  0.3  0.4}
         DEFAULT = 0.1
      }
      VOLTAGE {
         PIN = vdd UNIT = 1
         TABLE {3.0  3.3  3.6}
```

```
            DEFAULT = 3.3
         }
      }
      TABLE {
         // arrangement of whitespaces and comments
         // is only for readability
         // parser sees just a sequence of 3x4x3=36 numbers

//slewrate  0.5  1.0  1.5  capacitance   voltage
//          --------------+--------------+-------
            0.2  0.8  1.1 // 0.1           3.0
            0.4  1.0  1.2 // 0.2
            0.7  1.2  1.4 // 0.3
            0.9  1.5  1.8 // 0.4

            0.1  0.7  1.2 // 0.1           3.3
            0.3  0.9  1.3 // 0.2
            0.6  1.1  1.5 // 0.3
            0.8  1.3  1.7 // 0.4

            0.1  0.6  1.0 // 0.1           3.6
            0.2  0.8  1.1 // 0.2
            0.4  1.0  1.3 // 0.3
            0.7  1.2  1.6 // 0.4
      }
   }
```

An application that does not understand `VOLTAGE`, will extract the following information from this example:

```
   DELAY {
      HEADER {
         SLEWRATE {
            PIN = a UNIT = 1e-9
            TABLE {0.5  1.0  1.5}
         }
         CAPACITANCE {
            PIN = z UNIT = 1e-12
            TABLE {0.1  0.2  0.3  0.4}
         }
      }
      TABLE {

//slewrate  0.5  1.0  1.5     capacitance   voltage
//             --------------+--------------+-------
            0.1  0.7  1.2    // 0.1            3.3
            0.3  0.9  1.3    // 0.2
            0.6  1.1  1.5    // 0.3
            0.8  1.3  1.7    // 0.4
      }
   }
```

An application that does not understand SLEWRATE, will extract only the following information:

```
      DELAY {
         HEADER {
            CAPACITANCE {
               UNIT = 1e-12
               PIN = z
               TABLE {0.1 0.2 0.3 0.4}
            }
         }
         TABLE {

   //slewrate  1.0  capacitance    voltage
   //          ----+--------------+-------
               0.7   // 0.1           3.3
               0.9   // 0.2
               1.1   // 0.3
               1.3   // 0.4
         }
      }
```

4.7 Bus Modeling

4.7.1 Tristate Driver

Bus drivers are usually tristate buffers, which have straightforward functional models. If both input signal and enable signal have well-defined logic states, the output is driven to 'b1, 'b0, or 'bz, otherwise it is driven to 'bx.

```
CELL tristate_buffer {
   PIN a {DIRECTION   = input    SIGNALTYPE = data}
   PIN e {DIRECTION   = input    SIGNALTYPE = out_enable}
   PIN z {DIRECTION   = output   SIGNALTYPE = data
          SIGNALDRIVE = tristate ENABLE_PIN = e}
   FUNCTION {
      BEHAVIOR {
         z =
          (e & a)  ? 'b1:
          (e & !a) ? 'b0:
          (!e)     ? 'bz:
                     'bx;
      }
   }
}
```

A different model can be used for transmission-gate type of buffers, which also passes the high impedance state from input to output.

```
      BEHAVIOR {
         z =
          ( e) ? a  :
          (!e) ? 'bz:
                 'bx;
      }
   }
```

In order to model bus contention, the drive strength information of tristate buffers is needed. This is easily achieved by annotation of a pin property, using a context-sensitive keyword.

```
   CELL tristate_buffer {
      ...
      PIN z {DIRECTION = output DRIVE_STRENGTH = 4}
      ...
   }
```

The pin-property `DRIVE_STRENGTH` can take an arbitrary positive integer or a real number. In general, greater values override smaller values, and that `DRIVE_STRENGTH=0` is equivalent to

```
   FUNCTION {z='bz}.
```

ALF does not assume a particular set of legal drive strengths. The scale and granularity is left to the discretion of the ASIC vendor (user).

Modeling of state-dependent drive strength is achieved by annotating drive strength within a vector rather than within a pin. The following example shows a buffer with `strong-0` and `weak-1` drive.

```
   CELL tristate_buffer {
      ...
      PIN z {DIRECTION = output}
      ...
      VECTOR (z==0) {
         DRIVE_STRENGTH = 4 {PIN = z}
      }
      VECTOR (z==1) {
         DRIVE_STRENGTH = 2 {PIN = z}
      }
   }
```

The bus itself is not described by an ALF model, since the bus is a design construct rather than a library cell. A simulation model (Verilog or VHDL) would handle the bus contention. However, since buses can also be embedded within a core cell, the functional model of the core would need a functional model of that bus as well.

4.7.2 Bus with multiple drivers

The following example shows a bus with 3 drivers of equal strength. The output is the resolved value of the bus.

```
CELL bus3 {
   PIN z1 {DIRECTION = input}
   PIN z2 {DIRECTION = input}
   PIN z3 {DIRECTION = input}
   PIN z  {DIRECTION = output}
   FUNCTION {
      BEHAVIOR {
         z =
          ((z2=='bz || z2==z1) && z3=='bz)?  z1:
          ((z3=='bz || z3==z2) && z1=='bz)?  z2:
          ((z1=='bz || z1==z3) && z2=='bz)?  z3:
           (z1=='b1 && z2=='b1 && z3=='b1)? 'b1:
           (z1=='b0 && z2=='b0 && z3=='b0)? 'b0:
                                            'bx;
      }
   }
}
```

The following example shows a bus with two drivers of equal strength and one driver with weaker strength (e.g. a busholder).

```
CELL bus2s1w {
   PIN z_strong1 {DIRECTION = input}
   PIN z_strong2 {DIRECTION = input}
   PIN z_weak    {DIRECTION = input}
   PIN z         {DIRECTION = output}
   FUNCTION {
      BEHAVIOR {
         z =
          (z_strong1=='b1 && z_strong2=='b1)? 'b1:
          (z_strong1=='b0 && z_strong2=='b0)? 'b0:
          (z_strong1=='bz && z_strong2=='bz)? z_weak:
                                              'bx;
      }
   }
}
```

4.7.3 Busholder

A *busholder* is a cell that retains the previous value of a tristate bus, when all drivers go to high impedance. This device has only one external pin, which is bidirectional. The input to this bidirectional pin is the resolved value of the bus.

```
CELL busholder {
   PIN a {DIRECTION = both}
   PIN z {DIRECTION = output VIEW = none}
   FUNCTION {
      BEHAVIOR {
         a = !z;
         @(a==0)   {z = 1;}
         @(a==1)   {z = 0;}
         @(a=='bx) {z = 'bx;}
      }
   }
}
```

In order to understand the functionality of a bidirectional pin, we split the pin conceptually into an input pin and an output pin as shown below.

```
CELL busholder_explicit {
   PIN a_in  {DIRECTION = input}
   PIN a_out {DIRECTION = output}
   PIN z     {DIRECTION = output VIEW = none}
   FUNCTION {
      BEHAVIOR {
         a_out = !z;
         @(a_in==0)   {z = 1;}
         @(a_in==1)   {z = 0;}
         @(a_in=='bx) {z = 'bx;}
      }
   }
}
```

The function of this device is well defined, if `a_out==a_in` for all cases where `a_in!='bz`. In the case of `a_in=='bz, a_out` can take any value. This is a general modeling rule for functions with bidirectional pins.

4.8 Wire models

4.8.1 Basic Wire Model

This example shows two wire models, using tables and equations. The equation is used outside the defined table range. If no equation was defined, the table would be extrapolated.

```
WIRE small_wire {
   CAPACITANCE {
      UNIT = pF
      HEADER {
         CONNECTIONS {
            TABLE {2 3 4 5}
         }
```

```
      }
      TABLE {0.05 0.09 0.13 0.17}
      EQUATION {CONNECTIONS * 0.04 - 0.03}
   }
   RESISTANCE {
      UNIT = mOHM
      HEADER {
         CONNECTIONS {
            TABLE {2 3 4 5}
         }
      }
      TABLE {7.5 10.0 12.5 15.0}
      EQUATION {CONNECTIONS * 2.5 + 2.5}
   }
}

WIRE large_wire {
   CAPACITANCE {
      UNIT = pF
      HEADER {
         CONNECTIONS {
            TABLE {2 3 4}
         }
      }
      TABLE {0.10 0.16 0.22}
      EQUATION {CONNECTIONS * 0.06 - 0.2}
   }
   RESISTANCE {
      UNIT = mOhm
      HEADER {
         CONNECTIONS {
            TABLE {2 3 4}
         }
      }
      TABLE {10.0 12.5 15.0}
      EQUATION {CONNECTIONS * 2.5 + 5.0}
   }
}
```

4.8.2 Wire select model

Since a library may contain multiple wire models, it is necessary to specify which model should be selected for an application. The annotations inside each wire model can be used for this purpose.

```
WIRE small_wire {
   LIMIT {AREA=25 {UNIT=1e-6}}
   ...
}

WIRE large_wire {
   LIMIT {AREA {UNIT=1e-6 MIN=25 MAX=100}}
   ...
}
```

If the area covering the routing space is smaller than 25mm^2, the `small_wire` model will be chosen. If the area covering the routing space is between 25mm^2 and 100mm^2, the `large_wire` model is chosen. The unit for area is 1mm^2.

More annotations using the `USAGE` keyword can be introduced in order to enable customized wire model selection.

4.9 Megacell Modeling

4.9.1 Expansion of Timing Arcs

`GROUP` can be used for sets of numbers or for a continuous range of numbers. This can be useful for defining timing arcs between all bits of two vectors. For example,

```
GROUP adr_bits  {1 2 3}
GROUP data_bits {1 2}
VECTOR (01 adr[adr_bits] -> 01 dout[data_bits]) { ... }
```

replaces the following statements:

```
VECTOR (01 adr[1] -> 01 dout[1]) { ... }
VECTOR (01 adr[2] -> 01 dout[1]) { ... }
VECTOR (01 adr[3] -> 01 dout[1]) { ... }
VECTOR (01 adr[1] -> 01 dout[2]) { ... }
VECTOR (01 adr[2] -> 01 dout[2]) { ... }
VECTOR (01 adr[3] -> 01 dout[2]) { ... }
```

The following example shows bit-wise expansion of two vectors:

```
GROUP data_bits {1 2}
VECTOR (01 din[data_bits] -> 01 dout[data_bits]) { ... }
```

This replaces the following statements:

```
VECTOR (01 din[1] -> 01 dout[1]) { ... }
VECTOR (01 din[2] -> 01 dout[2]) { ... }
```

Example for bytewise (or sub-word wise) expansion:

```
GROUP low_byte  {1 2}
GROUP high_byte {3 4}
VECTOR (01 we[0] -> 01 din[low_byte])  { ... }
VECTOR (01 we[1] -> 01 din[high_byte]) { ... }
```

This replaces the following statements:

```
VECTOR (01 we[0] -> 01 din[1]) { ... }
VECTOR (01 we[0] -> 01 din[2]) { ... }
VECTOR (01 we[1] -> 01 din[3]) { ... }
VECTOR (01 we[1] -> 01 din[4]) { ... }
```

4.9.2 Two-port memory

The memory model example below shows the use of abstract transition operators on words in various vectors. Note the simplicity of the functional description of this two-port asynchronous memory. This example also contains some vectors with distinction between events on row and column address lines.

```
CELL async_1write_1read_ram {
   GROUP col {1:0}
   GROUP row {4:2}
   GROUP all {row col}
   GROUP byte{7:0}
   PIN       enable_write {DIRECTION = input}
   PIN [4:0] adr_write    {DIRECTION = input}
   PIN [4:0] adr_read     {DIRECTION = input}
   PIN [7:0] data_write   {DIRECTION = input}
   PIN [7:0] data_read    {DIRECTION = output}
   PIN [7:0] data_store [0:31] {DIRECTION = output VIEW = none}
   FUNCTION {
      BEHAVIOR {
         data_read = data_store[adr_read];
         @(enable_write) {data_store[adr_write] = data_write;}
      }
   }
   VECTOR
   (?! adr_read[col] -> ?? data_read[byte]) {
      /* fill in arithmetic models for delay and power */
   }
   VECTOR
   (?! adr_read[row] -> ?? data_read[byte]) {
      /* fill in arithmetic models for delay and power */
   }
   VECTOR
   ((?!adr_read[col] && ?!adr_read[row]) -> ??data_read[byte]){
      /* fill in arithmetic models for delay and power */
   }
   VECTOR (01 enable_write -> ?? data_read[byte]) {
      /* fill in arithmetic models for delay and power */
   }
   VECTOR (?! data_write[byte] -> ?? data_read[byte]) {
      /* fill in arithmetic models for delay and power */
   }
   VECTOR (?! adr_write[col]) {
      /* fill in arithmetic models for power */
   }
   VECTOR (?! adr_write[row]) {
      /* fill in arithmetic models for power */
   }
   VECTOR (?! adr_write[row] && ?! adr_write[col]) {
      /* fill in arithmetic models for power */
   }
   VECTOR (01 enable_write) {
      /* fill in arithmetic models for power */
   }
```

```
      VECTOR (10 enable_write) {
         /* fill in arithmetic models for power */
      }
      VECTOR (?! data_write[byte] && !enable_write) {
         /* fill in arithmetic models for power */
      }
      VECTOR (?! data_write[byte] &&  enable_write) {
         /* fill in arithmetic models for power */
      }
      VECTOR (?! adr_write[all] <-> 01 enable_write) {
         /* fill in arithmetic models for setup time  constraint */
      }
      VECTOR (10 enable_write <-> ?! adr_write[all]) {
         /* fill in arithmetic models for hold time constraint */
      }
      VECTOR (?! data_write[byte] <-> 10 enable_write) {
         /* fill in arithmetic models for setup time constraint */
      }
      VECTOR (10 enable_write -> ?! data_write[byte]) {
         /* fill in arithmetic models for hold time constraint */
      }
      VECTOR (01 enable_write -> 10 enable_write) {
         /* fill in arithmetic models for pulsewidth constraint */
      }
      VECTOR (10 enable_write -> 01 enable_write) {
         /* fill in arithmetic models for pulsewidth constraint */
      }
   }
```

The energy consumption for each operation depends on the number of switching bits of the bus. Therefore, the model inside a particular vector may look like this:

```
      VECTOR (?! data_write[byte] && 1 enable_write) {
         ENERGY {
            UNIT = pJ
            HEADER {switching_bits {PIN = data_write}}
            EQUATION {1.3 * switching_bits}
         }
      }
```

The rule that the address on a write port must not change during write enable high can be incorporated easily in the functional model. A pessimistic model assumes that the whole memory content will become unknown, if such an illegal address change occurs.

```
      BEHAVIOR {
         data_read = data_store[adr_read];
         @(enable_write) {data_store[adr_write] = data_write;}
         @(!?adr_write && enable_write)
            {data_store[0:31] = 8'bx;}
      }
```

4.9.3 Three-port memory

Functional models of more complex memories are also straightforward. The conflicts of writing to one memory location simultaneously from different ports can be modeled in a pessimistic way as follows:

```
CELL async_2write_1read_ram {
   PIN       enb_write1  {DIRECTION = input}
   PIN       enb_write2  {DIRECTION = input}
   PIN [4:0] adr_write1  {DIRECTION = input}
   PIN [4:0] adr_write2  {DIRECTION = input}
   PIN [4:0] adr_read    {DIRECTION = input}
   PIN [7:0] data_write1 {DIRECTION = input}
   PIN [7:0] data_write2 {DIRECTION = input}
   PIN [7:0] data_read   {DIRECTION = output}
   PIN [7:0] data_store [0:31] {DIRECTION = output VIEW = none}
   FUNCTION {
      BEHAVIOR {
         data_read = data_store[adr_read]
         @(enb_write1 && !enb_write2)
            {data_store[adr_write1] = data_write1;}
         @(enb_write2 && !enb_write1)
            {data_store[adr_write2] = data_write2;}
         @(enb_write1 && enb_write2 && adr_write1!=adr_write2) {
            data_store[adr_write1] = data_write1;
            data_store[adr_write2] = data_write2;
         }
         @(enb_write1 && enb_write2 && adr_write1==adr_write2) {
            data_store[adr_write1] =
               (data_write1==data_write2)? data_write1:8'bx;
            data_store[adr_write2]
               (data_write2==data_write1)? data_write2:8'bx;
         }
      }
   }
}
```

4.10 Special cells

4.10.1 Pulse generator

The following cell generates a one-shot pulse of 1 ns duration when enable goes high.

```
CELL one_shot {
   PIN enable {DIRECTION = input}
   PIN q      {DIRECTION = output}
   FUNCTION {
      BEHAVIOR {
         @(01 enable) {q = 1;}
         @(q)         {q = 0;}
      }
   }
   VECTOR (01 q -> 10 q) {
      DELAY = 1.0 {UNIT = ns}
   }
}
```

4.10.2 VCO

The following cell is a voltage controlled oscillator with 50% duty cycle and enable.

```
CELL vco {
   PIN enable {DIRECTION = input  PINTYPE = digital}
   PIN v_in   {DIRECTION = input  PINTYPE = analog}
   PIN q      {DIRECTION = output PINTYPE = digital}
   FUNCTION {
      BEHAVIOR {
         @(!enable)      {q = 0;}
         @(!q && enable) {q = 1;}
         @( q && enable) {q = 0;}
      }
   }
   TEMPLATE voltage_controlled_delay {
      DELAY {
         UNIT = ns
         HEADER {
            voltage {
               PIN = v_in
               TABLE {0.5 1.0 1.5 2.0 2.5 3.0}
            }
         }
         TABLE {10.00 5.00 3.33 2.50 2.00 1.67}
      }
   }
   VECTOR (01 q -> 10 q)
      voltage_controlled_delay
   }
   VECTOR (10 q -> 01 q)
      voltage_controlled_delay
   }
}
```

The template shown above can also be written as an equation to map voltage to frequency:

```
TEMPLATE voltage_controlled_delay {
   DELAY {
      UNIT = ns
      HEADER {voltage {PIN = v_in}}
      EQUATION {5.0 / voltage}
   }
}
```

4.11 Core Modeling

4.11.1 Digital Filter

This example illustrates the potential of ALF for modeling complex blocks. It shows a digital filter performing the following operation

```
dout(t) = state(t) + b1 * state(t-1) + b2 * state(t-2)
state(t) = din(t) - a1 * state(t-1) - a2 * state(t-2)
```

This second order infinite impulse response (IIR) filter is implemented with a single multiplier and a single adder/subtractor in a way that a new `dout` is produced every 4 clock cycles. The variable coefficients `a1, a2, b1,` and `b2` are stored in a dual port RAM.

The model uses templates for the functional blocks of a 2-bit counter used as controller for memory access and I/O operation, a RAM for coefficient storage, and the filter itself. In the top module they are instantiated as a structural netlist.

The use of templates is more general than the use of primitives, since not all basic blocks of the core may be supported as primitives.

```
LIBRARY core_lib {
   TEMPLATE CNT2 {
      BEHAVIOR {
         @ (!<cd>)   {<cnt> = 2'b0;}
         : (01 <cp>) {<cnt> = <start> ? 2'b0 : <cnt> + 1;}
      }
   }

   TEMPLATE RAM16X4 {
      BEHAVIOR {
         <dout> = <dmem>[<r_adr>];
         @ (<we>)  {<dmem>[<w_adr>] = <din>;}
      }
   }

   TEMPLATE IIR2 {
      BEHAVIOR {
         sum =
            (<cntrl>=='d0)? <din> - product :
            (<cntrl>=='d1)? accu  - product :
            (<cntrl>=='d2)? accu  + product :
            (<cntrl>=='d3)? accu  + product;
         @ (!<cd>)  {
```

```
            product = 16'b0;
            accu    = 16'b0;
         }
         : (01 <cp>){
            product =
               (<cntrl>=='d0)? coeff * state2 :
               (<cntrl>=='d1)? coeff * state1 :
               (<cntrl>=='d2)? coeff * state2 :
               (<cntrl>=='d3)? coeff * state1 :
               16'bX;
            accu  = sum;
         }

         @ (!<cd>)  {
            <dout> = 16'b0;
            state1 = 16'b0;
            state2 = 16'b0;
         }
         : (01 <cp> && <cntrl>=='d0){
            state2 = state1;
            state1 = accu;
            <dout> = accu;
         }
      }
   }

   CELL digital_filter {
      PIN [15:0] data_out    {DIRECTION = output}
      PIN [15:0] data_in     {DIRECTION = input}
      PIN [1:0]  index_coeff {DIRECTION = input}
      PIN        write_coeff {DIRECTION = input}
      PIN [15:0] coeff_in    {DIRECTION = input}
      PIN [15:0] coeff_out   {DIRECTION = output VIEW = none}
      PIN [15:0] coeff_array [1:4] {DIRECTION = output VIEW = none}
      PIN        data_strobe {DIRECTION = input}
      PIN [1:0]  count       {DIRECTION = output VIEW = none}
      PIN        clock       {DIRECTION = input}
      PIN        reset       {DIRECTION = input}
      FUNCTION {
         IIR2 U1 {din=data_in dout=data_out coeff=coeff_out
                  cp=clock cd=reset cntrl = count}
         CNT2 U2 {start=data_strobe cnt=count ck=clock cd=reset}
         RAM16X4 U3 {we=write_coeff din=coeff_in dout=coeff_out
                     dmem=coeff_array r_adr=count w_adr=index_coeff}
      }
   }
}
```

4.12 Connectivity

Connectivity information may be specified within the definition of the ALF language format as described below. A connectivity object always contains a rule specifying the type of

connections (e.g. must short, can short, cannot short) and a table. If no header is given, then the table contains the pins or pin classes subject to the connectivity rule. If a header is given, then the table contains the values of the connectivity function between arguments in the header. There must be a table inside each connectivity argument, containing the pins or pin classes subject to the connectivity rule. Valid arguments are `DRIVER` and/or `RECEIVER`. Valid values are the boolean digits `0`, `1`, and `?`. The value `1` implies the connection rule is true, the value `0` implies the connection rule is false, the value `?` implies don't care situation with the connection rule.

4.12.1 External connections between pins of a cell

The following example shows how to specify required and disallowed interconnections external to a cell.

```
CELL pll {
   PIN vdd_ana {PINTYPE=supply}
   PIN vdd_dig {PINTYPE=supply}
   PIN vss_ana {PINTYPE=supply}
   PIN vss_dig {PINTYPE=supply}
   CONNECTIVITY common_ground {
      CONNECT_RULE = must_short
      TABLE {vss_ana vss_dig}
   CONNECTIVITY separate_supply {
      CONNECT_RULE = cannot_short
      TABLE {vdd_ana vdd_dig}
   }
}
```

4.12.2 Allowed connections for classes of pins

The following example defines allowable pin interconnections. The constants for the desired connectivity classes, the grouping of these classes, and the allowable class connectivity table are first defined at the library level. The non-zero values within the matrix specify allowable connectivity of indexed classes. The connectivity classes for pins are then specified with the pin annotation sections.

```
LIBRARY example_library {
   ...
   CLASS default_class
   CLASS clock_class
   CLASS enable_class
   CLASS reset_class
   CLASS tristate_class
   ...
   TEMPLATE drivers {
      default_class
      clock_class
      enable_class
      reset_class
      tristate_class
   }
   TEMPLATE receivers {
```

```
      default_class
      clock_class
      enable_class
      reset_class
   }
   CONNECTIVITY driver_to_driver {
      CONNECT_RULE = can_short
      HEADER {
         DRIVER {TABLE {drivers}}
      }
      TABLE {// def clk enb rst tri
             0 0 0 0 1
      }
   }
   CONNECTIVITY receiver_to_receiver {
      CONNECT_RULE = can_short
      HEADER {
         RECEIVER {TABLE {receivers}}
      }
      TABLE {// def clk enb rst
             1 1 1 1
      }
   }
   CONNECTIVITY driver_to_receiver {
      CONNECT_RULE = can_short
      HEADER {
         DRIVER {TABLE {drivers}}
         RECEIVER {TABLE {receivers}}
      }
      TABLE {//  def  clk enb  rst   tri // driver/receiver
                  1    1   1    1     0  // def
                  0    1   0    0     0  // clk
                  0    0   1    0     0  // enb
                  0    0   0    1     0  // rst
      }
   }
```

The above table specifies allowed connectivity from each class to itself, as well as from each class to `default_class` except for the `tristate_class` class which may only connect to itself. Note also that while any class may connect to `default_class`, the `default_class` may only connect to itself.

Once the library level connectivity is defined, connection class specifications are defined for each pin within cells. The default integer value for the CLASS annotation is 0, which corresponds to the constant declaration value for default_class.

```
CELL d_flipflop_clr {
   PIN cd {PINTYPE  = input       SIGNALTYPE    = clear
           POLARITY = low         CONNECT_CLASS = reset_class}
   PIN cp {PINTYPE  = input       SIGNALTYPE = clock
           POLARITY = rising_edge CONNECT_CLASS = clock_class}
   PIN d  {PINTYPE  = input}
   PIN q  {PINTYPE  = output      CONNECT_CLASS = default_class}
}

CELL d_latch {
   PIN g {PINTYPE  = input  SIGNALTYPE    = enable
          POLARITY = high   CONNECT_CLASS = enable_class}
   PIN d {PINTYPE  = input  CONNECT_CLASS = default_class}
   PIN q {PINTYPE  = output CONNECT_CLASS = default_class}
}

CELL tristate_buffer {
   PIN a      {PINTYPE = input}
   PIN enable {PINTYPE = input  CONNECT_CLASS = enable_class}
   PIN z      {PINTYPE = output CONNECT_CLASS = tristate_class}
   ...
}
```

Net-specific connectivity, as opposed to the pin-specific connectivity as shown above, is also possible within the syntax of the language, since a CLASS is not restricted to pins. Specific applications may assign all pins of a specific type as well as nets like power and ground rails to a defined class. This class may be used within the connectivity tables to allow or disallow certain connectivity.

For example, if vddrail_class was defined as a net-specific connectivity class, then a specific pin may be disallowed from connecting to any net in the vddrail_class connectivity class.

```
CLASS vddrail_class
...
CELL inverter {
   PIN in_pin {PINTYPE = input SIGNALTYPE = clear
               POLARITY = low CONNECT_CLASS = reset_class}
   CONNECTIVITY dont_tie {
      CONNECT_RULE = cannot_short
      TABLE {in_pin vddrail_class}
   }
   ...
}
```

Index

Symbols

(N+1) order sequential logic 2-5
-> operator 2-4
?- 3-10
?! 3-10
?? 3-10
?~ 3-10
@ 2-3

Numerics

2-dimensional tables 4-15
3-dimensional table 4-17, 4-23

A

abstract transition operators 4-31
active vectors 3-47
ALF_AND 3-49, 4-4
ALF_BUF 3-48
ALF_BUFIF0 3-54
ALF_BUFIF1 3-54
ALF_FLIPFLOP 3-57, 4-2
ALF_LATCH 3-58
ALF_MUX 3-56, 4-3
ALF_NAND 3-49, 3-50
ALF_NAND2 4-1
ALF_NOR 3-49, 3-51
ALF_NOT 3-48, 3-49
ALF_NOTIF0 3-54, 3-55
ALF_NOTIF1 3-54, 3-55
ALF_OR 3-49, 3-51
ALF_XNOR 3-49, 3-53
ALF_XOR 3-49, 3-52
ALIAS 3-2
annotated properties 2-8
annotation
    author 3-29
    cell 3-30
    connect_rule 3-35
    datetime 3-29
    default 3-35
    error 3-29
    fall 3-29
    from 3-28
    information 3-28, 3-29
    limit 3-28
    max 3-29
    measurement 3-35
    message 3-29
    message_type 3-29
    min 3-29
    pin 3-30
    primitive 3-30
    product 3-29
    rise 3-29
    scan 3-28
    title 3-29
    to 3-28
    typ 3-29
    unit 3-35
    unnamed 3-28
    version 3-29
    violation 3-28
    warning 3-29
annotation container 3-4, 3-28
annotations 4-21
    PIN 4-21
    pin 4-37
    self-explaining 4-23
    timing arc 4-22
anotation
    class 3-30
arithmetic models 3-2
arithmetic operations 2-6
async_2write_1read_ram 4-33
atomic megacell 2-1
ATTRIBUTE 3-3
average 4-17

B

based literal 3-10
bidirectional pin 4-28
binary 3-10

bit 3-9
block comment 3-9
Boolean Equatio 4-1
boolean functions 2-1
both 4-28
bus contention 4-26
bus modeling 4-25
bus with multiple drivers 4-27
busholder 4-27

C

can_float 4-23
CAPACITANCE 4-10, 4-28
case-insensitive langauge 3-8
cell modeling 2-8
characterization 1-5, 2-1
    power 2-1, 2-7
    timing 2-1
characterization model 4-13
characterization variables 2-1
children object 3-1
CLASS 3-3, 4-37
class
    connectivity 4-37
combinational logic 2-2
combinational primitives 3-48
combinational scan cell 4-6
comment 3-7
    block 3-9
    long 3-9
    short 3-9
    single-line 3-9
comments
    nested 3-9
compound operators 3-8
CONNECT_RULE 4-38
connect_rule annotation 3-35
CONNECTION 4-23
connections
    allowed 4-37
    disallowed 4-37
    external 4-37
CONNECTIVITY 4-38
connectivity 4-36
    class 4-37
    net-specific 4-39
    pin-specific 4-39
connectivity class 4-37
CONSTANT 3-2
constant numbers 3-9
constraints
    delay 4-15
    power 4-15
    timing 4-15
context-sensitive keyword 3-12, 4-26
context-sensitive keywords 2-6
core 2-1
core cell 4-26
core modeling 4-35

D

d_flipflop_clr 4-2
d_flipflop_ld_clr 4-3
d_flipflop_mux_set_clr 4-4
d_latch 4-5
decimal 3-10
deep submicron 1-5
DEFAULT 4-23
default annotation 3-35
delay mode
    inertial 2-7
    invalid-value-detection 2-7
    transport 2-7
delay models 2-5
delay predictor 2-6
delimiter 3-7, 3-8
derating 4-19
derating equation 4-20
digital filter 4-35
digital_filter 4-36
DRIVE_STRENGTH 4-26
DRIVER 4-38

E

edge literal 3-10
edge rate 2-5
edge-sensitive sequential logic 2-2
elapsed time 2-6
ENERGY 4-18
energy 2-7
escape codes 3-11
escaped identifier 3-11

event sequence detection 2-4
expansion
    bit-wise 4-30
    bytewise 4-30
expansion of vectors 4-30
exponentiation 2-6
extensible primitives 3-47
external connections 4-37

F

fanout 2-9
Flipflop 3-56
flipflop 4-2
forward referencing 3-1
fringe capacitance 2-9
function
    exponentiation 2-6
    logarithm 2-6
functional model 1-4
functional modeling 2-2
functional models 2-1
functions 3-2

G

generic objects 3-2
glitch 2-7
GROUP 3-4, 4-30

H

hard keywor 3-12
hardware description language 2-1
HDL 2-1
hexadecimal 3-10
hierarchical object 3-1

I

identifier 3-1, 3-7
Identifiers 3-11
illegal vector 3-47
inactive vectors 3-47
INCLUDE 3-2, 3-39
inertial delay mode 2-7
infinite impulse response filter 4-35
INFORMATION 4-6
integer 3-9
internal load 2-6
intrinsic delay 2-5

J

JK-flipflop 4-3
JTAG BSR cell 4-6

K

keyword 3-1
keywords
    context-sensitive 2-6

L

Latch 3-58
layout parasitics 2-6
legal vectors 3-47
level-sensitive cell 4-5
level-sensitive sequential logic 2-2
LIBRARY 4-6
library 3-1
Library creation 1-1
library-specific objects 3-2
LIMIT 4-23
literal 3-1, 3-7
load characterization model 2-6
logarithm 2-6

M

macrocells 2-1
MEASUREMENT 4-17
measurement annotation 3-35
megacell modeling 4-30
megacells 2-1
metal layer 2-9
mode of operation 1-5
modeling
    bus 4-25
    cell 2-8
    cores 4-35
    functional 2-2
    megacell 4-30
    performance 2-5
    physical 2-8
    power 2-7
    synthesis 2-8
    test 2-8
    timing 2-5

wire 2-9
wireload 4-28
multiplexor 3-56
must_connect 4-23
muxscan 4-8

N

NAND gate 4-1
nested comments 3-9
no_connection 4-23
NON_SCAN_CELL 4-7
non-escaped identifier 3-11
non-scan cells 2-8

O

objects 3-1
octal 3-10
one_shot 4-34
one-pass parser 3-1
operation mode 1-5
operator
-> 2-4
followed by 2-4
operators
signed 3-25
unsigned 3-25
output ramptime 4-9

P

parasitic capacitance 2-9
parasitic resistance 2-9
performance modeling 2-5
physical modeling 2-8
pin-toggle power model 2-8
placeholder identifier 3-11
placeholders 3-3
POLARITY 4-7
power 2-7
Power characterization 2-1
power characterization 2-7
power constraint 1-5
power dissipation 2-7
Power model 1-5
power modeling 2-7
primitive 4-2
PROCESS 4-19
PROPERTY 3-4
pulse generator 4-34
PVT Derating 4-19

Q

Q_CONFLICT 3-57
QN_CONFLICT 3-57
quad D-Flipflop 4-8
quoted string 3-8, 3-11

R

RAM16X4 4-36
real 3-9
reserved keyword 3-12
RESISTANCE 4-29
RTL 1-4

S

scaled average current 2-7
scaled average power 2-7
scan cell
combinational 4-6
scan chai 4-6
Scan Flipflop 4-7
Scan insertion 2-8
scan test 2-8
scan_data 4-8
scan_enable 4-8
SCAN_FFX4 4-9
SCAN_ND4 4-7
SCAN_TYPE 4-7
self capacitanc 2-9
self-explaining annotations 4-23
sequential logic
edge-sensitive 2-2
level-sensitive 2-2
N+1 order 2-5
vector-sensitive 2-4
sheet resistance 2-9
signed operators 3-25
simulation model 1-5
single-line comment 3-9
slew rate 2-5
SLEWRATE 4-10, 4-25
soft keyword 3-12
sr_latch 4-5

state-dependent drive strength 4-26
STATETABLE 4-1
static power 2-8
std_derating 4-20
std_header_2d 4-11
switching energy 4-9

T

TABLE 4-10
TEMPERATURE 4-19
TEMPLATE 3-3, 4-12
template 4-9
template definition 4-11
template-reference scheme 2-6
Three-port Memory 4-33
timing arc 4-22
timing characterization 2-1
timing constraint model 2-6
timing constraint models 2-5
timing constraints 1-5, 4-15
timing modeling 2-5
timing models 1-5
transcendent functions 2-6
transient power 2-8
transition delay 2-5
transmission-gate 4-26
transport delay mode 2-7
    invalid-value-detection 2-7
triggering conditions 2-2
triggering function 2-2
tristate driver 4-25
tristate primitives 3-54
tristate_buffer 4-25
Truth Table 4-1
truth table 2-1
Two-port memory 4-31

U

unit annotation 3-35
unnamed annotation containers 3-28
unsigned operators 3-25

V

VCO 4-34
VECTOR 4-9
vector expression 2-4
vector-based modeling 1-5
Vector-Sensitive Sequential Logic 2-4
vector-specific model 4-9
Verilog 1-4, 2-3
VHDL 1-4, 2-3
via resistance 2-9
VIOLATION 4-16
virtual pins 2-8, 3-57
VOLTAGE 4-19, 4-24
voltage_controlled_delay 4-35

W

whitespace characters 3-8
wire modeling 2-9
wire select model 4-29