Advanced Library Format
for
ASIC Cells & Blocks

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

Version 1.1

April 6, 1999

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

Jay Abraham | Silicon Integration Initiative
Mike Andrews | Mentor Graphics | Co-Chairman
Tim Ayres | Synopsys - Viewlogic
Arun Balakrishnan | NEC
Tim Baldwin | Cadence - Ambit
John Beatty | IBM
Victor Berman | VI / IEEE
Dennis Brophy | Mentor Graphics / OVI / IEEE
Jose De Castro | LSI Logic
Renlin Chang | Cadence
Shir-Shen Chang, PhD | Synopsys
Sanjay Churiwala | Cadworx
Timothy Ehrler | VLSI Technology
Ted Elkind | Cadence
Paul Foster | Avant!
Vassilios Gerousis, PhD | Siemens / OVI
Kevin Grotjohn | LSI Logic
Mitch Heins | Cadence - Ambit
Eric Howard | Cadence
Tim Jennings | Motorola
Timothy Jordan | Motorola
Archie Lachner | Mentor Graphics
Tai Le | Avant!
Johnson Chan Limqueco | Cadence - Ambit
Ta-Yung Liu | Avant!
Saumendra Nath Mandal | Duet Technologies
Hamid Rahmian | Mentor Graphics
Darshan Rauniyar | Mentor Graphics
Wolfgang Roethig, PhD | NEC | Chairman
Larry Rosenberg, PhD | Cadence / VSIA
Ambar Sarkar, PhD | Synopsys - Viewlogic
Itzhak Shapira | Cadence
Jin-Sheng Shyr | Toshiba
Sergei Sokolov | Sente
Peter Sauris | Mentor Graphics
Toru Toyoda | NEC
Yatin Trivedi | Seva Technologies | Technical Editor
Devadas Varma | Cadence - Ambit
David Wallace | Mentor Graphics - Exemplar
Cary Wei | Fujitsu
Frank Weiler | Avant! / OVI
Jeff Wilson | Mentor Graphics
Amir Zarkesh, PhD | TDT
Revision history:

<table>
<thead>
<tr>
<th>Draft/Version</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st draft</td>
<td>11/20/96</td>
</tr>
<tr>
<td>2nd draft</td>
<td>12/20/96</td>
</tr>
<tr>
<td>3rd draft</td>
<td>3/22/97</td>
</tr>
<tr>
<td>4th draft</td>
<td>3/31/97</td>
</tr>
<tr>
<td>5th draft</td>
<td>4/22/97</td>
</tr>
<tr>
<td>6th draft</td>
<td>6/1/97</td>
</tr>
<tr>
<td>7th draft</td>
<td>6/25/97</td>
</tr>
<tr>
<td>8th draft</td>
<td>8/13/97</td>
</tr>
<tr>
<td>9th draft</td>
<td>10/14/97</td>
</tr>
<tr>
<td>Version 1.0</td>
<td>11/14/97</td>
</tr>
<tr>
<td>Version 1.0.1</td>
<td>3/20/98</td>
</tr>
<tr>
<td>Version 1.0.2</td>
<td>4/8/98</td>
</tr>
<tr>
<td>Version 1.0.3</td>
<td>5/15/98</td>
</tr>
<tr>
<td>Version 1.0.4</td>
<td>5/31/98</td>
</tr>
<tr>
<td>Version 1.0.5</td>
<td>6/15/98</td>
</tr>
<tr>
<td>Version 1.0.6</td>
<td>9/20/98</td>
</tr>
<tr>
<td>Version 1.0.7</td>
<td>11/15/98</td>
</tr>
<tr>
<td>Version 1.0.8</td>
<td>1/12/99</td>
</tr>
<tr>
<td>Version 1.0.9</td>
<td>2/5/99</td>
</tr>
<tr>
<td>Version 1.0.10</td>
<td>2/19/99</td>
</tr>
<tr>
<td>Version 1.0.11</td>
<td>3/12/99</td>
</tr>
</tbody>
</table>
3.2.7 Numbers 38
3.2.8 Bit Literals 39
3.2.9 Based Literals 40
3.2.10 Edge Literals 41
3.2.11 Quoted Strings 41
3.2.12 Identifiers 42
3.2.13 Rules against parser ambiguity 43
3.2.14 Cross-reference of lexical tokens 43

3.3 Keywords 44
3.3.1 Keywords for Objects 44
3.3.2 Keywords for Operators 45
3.3.3 Context-Sensitive Keywords 45

3.4 Syntax Rules 45
3.4.1 Assignments 45
3.4.2 Expressions 46
3.4.3 Instantiations 47
3.4.4 Literals 48
3.4.5 Operators 49
3.4.6 Auxiliary Objects 51
3.4.7 Generic Objects 51
3.4.8 CELL 53
3.4.9 LIBRARY 53
3.4.10 PIN 53
3.4.11 PRIMITIVE 54
3.4.12 SUBLIBRARY 54
3.4.13 VECTOR 54
3.4.14 WIRE 55
3.4.15 Arithmetic Model 55
3.4.16 FUNCTION 56
3.4.17 Cross-reference of BNF items 57

3.5 Operators 61
3.5.1 Arithmetic operators 61
3.5.2 Boolean operators on scalars 62
3.5.3 Boolean operators on words 63
3.5.4 Vector operators 64
3.5.5 Operators for sequential logic 68
3.5.6 Operator priorities 68
3.5.7 Datatype mapping 69

3.6 Context-sensitive keywords 71
3.6.1 Annotation Containers 71
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6.2</td>
<td>Keywords for referencing objects used as annotation</td>
<td>73</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Annotations for a PIN object</td>
<td>73</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Annotations for a VECTOR object</td>
<td>78</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Annotations for a CELL object</td>
<td>80</td>
</tr>
<tr>
<td>3.6.6</td>
<td>Attributes</td>
<td>84</td>
</tr>
<tr>
<td>3.6.7</td>
<td>Keywords for arithmetic models</td>
<td>85</td>
</tr>
<tr>
<td>3.6.8</td>
<td>Containers for arithmetic models</td>
<td>93</td>
</tr>
<tr>
<td>3.6.9</td>
<td>Keywords for arithmetic submodels</td>
<td>95</td>
</tr>
<tr>
<td>3.6.10</td>
<td>Annotations for arithmetic models</td>
<td>100</td>
</tr>
<tr>
<td>3.7</td>
<td>Library Organization</td>
<td>108</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Scoping Rules</td>
<td>108</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Use of multiple files</td>
<td>109</td>
</tr>
<tr>
<td>3.8</td>
<td>Referenceable objects</td>
<td>109</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Referencing PRIMITIVES or CELLS</td>
<td>110</td>
</tr>
<tr>
<td>3.8.2</td>
<td>Referencing PINs in FUNCTIONs</td>
<td>111</td>
</tr>
<tr>
<td>3.8.3</td>
<td>Referencing PINs in VECTORs</td>
<td>113</td>
</tr>
<tr>
<td>3.8.4</td>
<td>Referencing multi-dimensional PINs</td>
<td>113</td>
</tr>
<tr>
<td>3.8.5</td>
<td>Referencing arithmetic models</td>
<td>115</td>
</tr>
<tr>
<td>3.9</td>
<td>Functional modeling styles and rules</td>
<td>117</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Rules for combinational functions</td>
<td>117</td>
</tr>
<tr>
<td>3.9.2</td>
<td>Basic rules for sequential functions</td>
<td>118</td>
</tr>
<tr>
<td>3.9.3</td>
<td>Concurrency in combinational and sequential functions</td>
<td>120</td>
</tr>
<tr>
<td>3.9.4</td>
<td>Initial values for logic variables</td>
<td>123</td>
</tr>
<tr>
<td>3.10</td>
<td>Primitives</td>
<td>124</td>
</tr>
<tr>
<td>3.10.1</td>
<td>Concept of user-defined and predefined primitives</td>
<td>124</td>
</tr>
<tr>
<td>3.10.2</td>
<td>Predefined combinational primitives</td>
<td>126</td>
</tr>
<tr>
<td>3.10.3</td>
<td>Predefined tristate Primitives</td>
<td>129</td>
</tr>
<tr>
<td>3.10.4</td>
<td>Predefined multiplexor</td>
<td>131</td>
</tr>
<tr>
<td>3.10.5</td>
<td>Predefined flipflop</td>
<td>132</td>
</tr>
<tr>
<td>3.10.6</td>
<td>Predefined latch</td>
<td>133</td>
</tr>
<tr>
<td>3.11</td>
<td>Parameterizeable Cells</td>
<td>135</td>
</tr>
<tr>
<td>3.12</td>
<td>Modeling with Vector Expressions</td>
<td>138</td>
</tr>
<tr>
<td>3.12.1</td>
<td>Event reports</td>
<td>139</td>
</tr>
<tr>
<td>3.12.2</td>
<td>Event Sequences</td>
<td>140</td>
</tr>
<tr>
<td>3.12.3</td>
<td>Scope and content of event sequences</td>
<td>141</td>
</tr>
<tr>
<td>3.12.4</td>
<td>Alternative event sequences</td>
<td>143</td>
</tr>
<tr>
<td>3.12.5</td>
<td>Symbolic edge operators</td>
<td>145</td>
</tr>
<tr>
<td>3.12.6</td>
<td>Non-events</td>
<td>146</td>
</tr>
<tr>
<td>3.12.7</td>
<td>Compact and verbose event sequences</td>
<td>146</td>
</tr>
<tr>
<td>3.12.8</td>
<td>Unspecified simultaneous events within scope</td>
<td>147</td>
</tr>
</tbody>
</table>
3.12.9 Simultaneous event sequences 149
3.12.10 Implicit local variables 151
3.12.11 Conditional event sequences 152
3.12.12 Alternative conditional event sequences 154
3.12.13 Change of scope within a vector expression 156
3.12.14 Sequences of conditional event sequences 159
3.12.15 Incompletely specified event sequences 161
3.12.16 Well-specified vector expressions 163

4 Applications .......................................................... 165

4.1 Truth Table vs Boolean Equation 165
   4.1.1 NAND gate 165
   4.1.2 Flipflop 166
4.2 Use of primitives 166
   4.2.1 D-Flipflop with asynchronous clear 166
   4.2.2 JK-flipflop 167
   4.2.3 D-Flipflop with synchronous load and clear 168
   4.2.4 D-Flipflop with input multiplexor 168
   4.2.5 D-latch 169
   4.2.6 SR-latch 169
   4.2.7 JTAG BSR 170
   4.2.8 Combinational Scan Cell 170
   4.2.9 Scan Flipflop 172
   4.2.10 Quad D-Flipflop 173
4.3 Templates and vector-specific models 174
   4.3.1 Vector-specific delay and power Tables 174
   4.3.2 Use of TEMPLATE 176
   4.3.3 Vector description styles for timing arcs 178
   4.3.4 Vectors for delay, power and timing constraints 179
4.4 Combining tables and equations 181
   4.4.1 Table vs equation 181
   4.4.2 Cell with Multiple Output Pins 182
   4.4.3 PVT Derating 183
4.5 Use of Annotations 185
   4.5.1 Annotations for a PIN 185
   4.5.2 Annotations for a timing arc 186
   4.5.3 Creating Self-explaining Annotations 187
4.6 Providing a fall-back position for applications 187
   4.6.1 Use of DEFAULT 187
4.7 Bus Modeling 189
4.7.1 Tristate Driver 189
4.7.2 Bus with multiple drivers 191
4.7.3 Busholder 192

4.8 Wire models 192
4.8.1 Basic Wire Model 192
4.8.2 Wire select model 193

4.9 Megacell Modeling 194
4.9.1 Expansion of Timing Arcs 194
4.9.2 Two-port memory 195
4.9.3 Three-port memory 198
4.9.4 Annotation for pins of a bus 198
4.9.5 Skew for simultaneously switching signals on a bus 199

4.10 Special cells 200
4.10.1 Pulse generator 200
4.10.2 VCO 200

4.11 Core Modeling 201
4.11.1 Digital Filter 201

4.12 Connectivity 203
4.12.1 External connections between pins of a cell 203
4.12.2 Allowed connections for classes of pins 204

4.13 Signal Integrity 206
4.13.1 I/V curves 206
4.13.2 Driver resistance 208

4.14 Resistance and Capacitance on a Pin 211
4.14.1 Self-Resistance and Capacitance on Input Pin 211
4.14.2 Pullup and Pulldown Resistance on Input Pin 211
4.14.3 Pin and Load Resistance and Capacitance on Output Pin 212

4.15 ALF/SDF cross reference 213
4.15.1 SDF delays 213
4.15.2 SDF timing constraints 218
4.15.3 SDF conditions and labels for delays and timing constraints 226
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 rely 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 that 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 choose the best product without library format constraints. Since ASIC vendors have to support a multitude of libraries according to the preferences of their customers, a common standard library is expected to significantly reduce the library development cycle and facilitate the deployment of new technologies sooner.

1.2 Goals

The basic goals of the proposed library standard are:

- \textit{simplicity} - library creation process must be easy to understand and not become a cumbersome process only known by a few experts.

- \textit{generality} - tools of any level of sophistication must be able to retrieve necessary information from the library.

- \textit{expandability} - for early adoption and future enhancement possibilities
Introduction

- **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 and design planning tools, is provided by ALF. On the other hand, layout information is considered to be available in complementary libraries such as LEF. Please note that ALF covers **library** data, whereas **design** data must be provided in other formats.
Historically, a functional model was virtually identical to a simulation model. A functional gate-level model was used by the proprietary simulator of the ASIC company, and it was easy to lump it together with a rudimentary timing model. Timing analysis was done through dynamic functional simulation. However, with the advanced level of sophistication of both functional simulation and timing analysis, this is no longer the case. The capabilities of the functional simulators have evolved far beyond the gate-level, and timing analysis has been decoupled from simulation.

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

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

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

<table>
<thead>
<tr>
<th>application</th>
<th>functional model</th>
<th>performance model</th>
<th>physical model</th>
</tr>
</thead>
<tbody>
<tr>
<td>timing analysis</td>
<td>N/A</td>
<td>supported by ALF</td>
<td>N/A</td>
</tr>
<tr>
<td>power analysis</td>
<td>N/A</td>
<td>supported by ALF</td>
<td>N/A</td>
</tr>
<tr>
<td>simulation</td>
<td>derived from ALF</td>
<td>derived from ALF</td>
<td>N/A</td>
</tr>
<tr>
<td>synthesis</td>
<td>supported by ALF</td>
<td>supported by ALF</td>
<td>supported by ALF</td>
</tr>
<tr>
<td>scan insertion</td>
<td>supported by ALF</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RTL design planning</td>
<td>derived from ALF</td>
<td>supported by ALF</td>
<td>planned for ALF</td>
</tr>
<tr>
<td>signal integrity</td>
<td>N/A</td>
<td>supported by ALF</td>
<td>N/A</td>
</tr>
<tr>
<td>layout</td>
<td>N/A</td>
<td>N/A</td>
<td>not supported by ALF</td>
</tr>
</tbody>
</table>
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.

Figure 1-1: ALF and its target applications
Hence, the use of a particular VHDL or Verilog simulation model as primary source of functional description of a cell is not very practical. Moreover, the existence of two simulation standards makes it difficult to pick one as a reference with respect to the other. The purpose of a generic functional model is to serve as an absolute reference for all applications that require functional information. Applications such as synthesis, which need functional information merely for recognizing and choosing cell types, can use the generic functional model directly. For other applications such as simulation and test, the generic functional model enables automated simulation model and test vector generation and verification, which has a tremendous benefit for the ASIC industry.

With progress of technology, 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 they 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
[item1 | item2 | ... ]
optional item with alternatives
{item} optional item that can be repeated
{item1 | item2 | ... }
optional items with alternatives which can be repeated
item item in boldface font is taken verbatim
item item in italic is for explanation purpose only
```

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

```
=== left side and right side expressions are equivalent
<int> a placeholder for an item in regular syntax
```
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 underlying concepts for 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 application notes.
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 synthesizeable 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 scalar pins; number and range of switching pins within a bus become additional characterization variables. Characterization measurements are expandable and can be extrapolated from scalar 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. Let us consider an arbitrary continuous assignment:

\[ z = f(a_1, \ldots, a_n) \]

In a dynamic or simulation context, the left-hand side (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:

\[ z_j = f(a_1, \ldots, a_n, z_1, \ldots, z_m) \]

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 level-sensitive sequential logic. The expression of a triggered assignment can look like this:

\[ @ g(b_1, \ldots, b_k) \ z_j = f(a_1, \ldots, a_n, z_1, \ldots, z_m) \]

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:

\[
Q = \begin{cases} 
D, & \text{at rising edge of CP, assign Q the value of D}. 
\end{cases}
\]

If the flipflop also has an asynchronous direct clear pin (CD), the functional description consists of either two concurrent statements or two statements ordered by priority:

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

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

The following two examples show corresponding simulation models in Verilog and VHDL:

```verilog
// full simulation model
always @(negedge CD or posedge CP) begin
  if ( ! CD ) Q <= 0;
  else if (CP && !CP_last_value) Q <= D;
  else Q <= 1'bx;
end
always @ (posedge CP or negedge CP) begin
  if (CP===0 | CP===1'bx) CP_last_value <= CP ;
end
// simplified simulation model for synthesis
always @(negedge CD or posedge CP) begin
  if ( ! CD ) Q <= 0;
  else Q <= D;
end
```

Figure 2-2: Model of a flipflop with asynchronous clear in Verilog
The following differences in modeling style can be noticed: VHDL and Verilog provide the list of sensitive signals at the beginning of the `process` or `always` block, respectively. The information of level-or edge-sensitivity must be inferred by if-then-else statements inside the block. ALF shows the level-or-edge sensitivity as well as the priority directly in the triggering expression. Verilog has another particularity: The sensitivity list indicates whether at least one of the triggering signals is edge-sensitive, by the use of `negedge` or `posedge`. However, it does not indicate which one, since either none or all signals must have `negedge` or `posedge` qualifiers. Furthermore, `posedge` is any transition with 0 as initial state or 1 as final state. A positive-edge triggered flipflop will be inferred for synthesis, yet this flipflop will only work correctly if both the initial state is 0 and the final state is 1. Therefore a simulation model for verification must be more complex than the model in the synthesizeable RTL code. In Verilog, the extra non-synthesizeable code must also reproduce the relevant previous state of the clock signal, whereas VHDL has built-in support for `last_value` of a signal.

Other aspects of simulation models include performance and trade-off between accuracy and runtime, timing annotation etc.

ALF provides a canonical, compact and highly self-explaining description of the functional specification of a cell, from which simulation models for various applications can be derived.
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:

\[ 01 \text{A} \rightarrow 01 \text{B} \]

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.

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 events can also be gated with static logical conditions. For example,

\[ (01 \text{CP} \rightarrow 10 \text{CP}) \&\& \text{CD} \]
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 for Characterization

2.3.1 Timing Modeling

The timing models of cells consist 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 that 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:

\[ 01 \text{ A} \rightarrow 10 \text{ Z} \]

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:

\[ 01 \text{ D} \rightarrow 01 \text{ CP} \]

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

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:

\[
\text{power} = \text{current} \times \text{Vdd} \\
\text{energy} = \text{current} \times \text{Vdd} \times \text{integration time}
\]

The set of vectors causing power consumption within a cell is a superset of those vectors causing the cell output to switch. While only the vectors with switching output 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:

\[
\begin{align*}
01 \, \text{CP} & \rightarrow 01 \, Q \\
01 \, \text{CP} & \rightarrow 10 \, Q
\end{align*}
\]

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

\[
\begin{align*}
01 \, \text{CP} \land Q==D \\
10 \, \text{CP} \land Q==D
\end{align*}
\]

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 with input pins \textit{A}, \textit{B} and output pin \textit{Z} a glitch is observed if the event 01 \textit{A} is detected and then the event 10 \textit{B} is detected before the input-to-output delay elapses. 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:

\[
01 \, \text{A} \rightarrow 10 \, \text{B} \rightarrow 01 \, \text{Z} \rightarrow 10 \, \text{Z}
\]

Simulation featuring transport delay mode with invalid-value-detection would exhibit the glitch as follows:\(^2\)

\[
01 \, \text{A} \rightarrow 10 \, \text{B} \rightarrow 'b0'bX \, \text{Z} \rightarrow 'bX'b0 \, \text{Z}
\]

Simulation with inertial delay mode would suppress the output transitions:

\[
(01 \, \text{A} \rightarrow 10 \, \text{B}) \land \neg \text{Z}
\]

The last expression can be used for each of the three simulation modes, since \(\neg \text{Z}\) is always true from beginning to end of the sequence 01 \textit{A} \rightarrow 10 \textit{B}, in particular at the time when the sequence 01 \textit{A} \rightarrow 10 \textit{B} is detected.

---

2. use based edge literals to avoid parser ambiguity.
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 by a simulation model in the same way as level-sensitive logic in functional modeling by a boolean expression, whereas transient power consumption is activated similar to edge-sensitive logic by a vector expression.

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

More abstract vector expressions are provided for power modeling of complex blocks, where simplification is needed in order to deal with the complexity of characterization vectors.

2.3.3 Modeling for signal integrity

The concept of vector-based cell characterization with multiple variables also accommodates the requirements for signal integrity modeling. Although signal integrity is closely related to interconnect parasitics, i.e. extracted design information, there must be data in the cell library in order to support signal integrity analysis.

- Crosstalk analysis needs characterization of driver resistance on output pins and noise margin on input pins.

- IR drop and electromigration analysis on power supply lines needs characterization of average currents as for power analysis, RMS currents and current waveforms.

- Electromigration (EM) analysis within cells needs characterization of current limits. In a direct evaluation approach, the current limits are checked against the actual currents. The latter data comes from the characterization for power and IR drop. In an indirect evaluation approach, the current limits may be expressed as frequency-dependent load limits and/or slewrate limits.

- Hot electron (HE) analysis within cells needs characterization of flux (charge density) or fluence (accumulated charge density over time) and its respective limits for performance degradation. In a direct evaluation approach, the flux or fluence limits are checked against the actual flux or fluence, respectively. In an indirect evaluation approach, the limits of performance degradation due to fluence may be expressed as frequency-dependent load limits and/or slewrate limits, in the same way as for electromigration.

The characterization vector set for driver resistance is a subset of delay characterization vectors. In buffered cells, the driving input does not matter, since the driver resistance seen at
the output is the same. However, there is always a different driver resistance for rise and fall, which is also dependent on process, voltage, temperature.

Noise margin characterization is especially important for control and data pins of sequential cells. The set of characterization vectors is complementary to the timing constraint characterization vectors. For instance, noise margin on a clock pin is complementary to the pulsewidth constraint. If pulsewidth corresponds to the smallest possible signal causing a valid functional reaction, then noise margin corresponds to the largest possible signal causing no functional reaction.

The characterization vector set for IR drop and EM on power supply lines is essentially the same as for power analysis, only the set of data per vector is richer. IR drop analysis may use average currents, peak currents, or current waveforms. EM analysis may use average, peak, RMS or a combination of the above.

The characterization vector set for EM and HE effect occurring within cells is very similar to the characterization vector set for power analysis, depending whether a direct or indirect evaluation approach is used.

In summary, modeling for crosstalk is a natural extension of modeling for timing, whereas IR drop, EM and HE modeling are natural extensions of modeling for power.

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 requirements for the modeling of test 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 remainder 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, in a similar to cells. For example,

```plaintext
// wire with fanout < 5 routed in metal 1, 2
WIRE small_wire {
    ATTRIBUTE { metal1 metal2 }
    LIMIT { FANOUT { MAX = 5; } }
    /* fill in data */
}
// wire with 10 ≤ fanout ≤ 20 routed in metal 1, 2, 3, 4, 5
WIRE big_wire {
    ATTRIBUTE { metal1 metal2 metal3 metal4 metal5 }
    LIMIT { FANOUT { MIN = 10; MAX = 20; } }
    /* fill in data */
}
```

From a modeling standpoint, no particular language 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 3.1.2 and Section 3.1.3 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. An object without children is called an atomic 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-pass parsers.

3.1.1 Syntax conventions

In order to make ALF easy to parse, we use syntax conventions that are followed by the existing syntax rules (see Section 3.4) and should also be followed for future extensions of the grammar.

The first token of the object is the object type identifier, followed by a name (mandatory or optional, depending on object type), followed by (mandatory or optional) = and value assignment, followed by (mandatory or optional) children objects enclosed by curly braces. Objects with more than one token (i.e. name and/or value) and without children are terminated with ;.

Examples:

1. unnamed object without value assignment:

    MY_OBJECT_TYPE
or

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

2. unnamed object with value assignment:

```c
MY_OBJECT_TYPE = my_object_value;
```
or

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

3. named object without value assignment:

```c
MY_OBJECT_TYPE my_object_name;
```
or

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

4. named object with value assignment:

```c
MY_OBJECT_TYPE my_object_name = my_object_value;
```
or

```c
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.

### 3.1.2 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:
### 3.1.2.1 CONSTANT

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

Example:

```plaintext
CONSTANT vdd = 3.3;
```

### 3.1.2.2 ALIAS

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

Example:

```plaintext
ALIAS RAMPTIME = SLEWRATE;
```

### 3.1.2.3 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:

```plaintext
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.

### 3.1.2.4 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:

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

### 3.1.2.5 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 that 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:

```java
CELL rr_8x128 {
    ATTRIBUTE (ROM ASYNCHRONOUS STATIC)
}
PIN read_write_select {
    ATTRIBUTE {READ{POLARITY=low;} WRITE{POLARITY=high;}}
}
```

### 3.1.2.6 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. Identifiers 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:

```java
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:

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

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

```java
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.
3.1.2.7 PROPERTY

A PROPERTY object is a named or an unnamed 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;
}
```

3.1.2.8 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; }
```

3.1.3 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.

3.1.4 Arithmetic models

An arithmetic model is an object that describes characterization data, or a more abstract, measurable relationship between physical quantities. The modeling language allows tabulated data as well as linear and non-linear equations. The equations consist of arithmetic expressions, for which the IEEE standards have been adopted.

3.1.5 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.

```
library specific object  contains  generic object
  arithmetic model       contains
                      contains
                  function                              
```

**Figure 3-2: Library-specific objects**

```
contains
arithmetic model

contains
header

contains

contains
table

contains
equation
```

**Figure 3-3: Arithmetic model**

```
vector

contains

primitive

contains

function

contains

behavior

contains

statetable

contains

vector expression
```

**Figure 3-4: Function**

Note that a function can contain a primitive and a primitive can contain a function. See figure 3-7 and syntax descriptions in Section 3.4.11 and Section 3.4.16.
Figure 3-5: Annotations

Figure 3-6: Library-specific objects
Note that a function can contain a primitive and a primitive can contain a function. See figure 3-7 and syntax descriptions in Section 3.4.11 and Section 3.4.16.

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 bit literal, a based literal, an edge literal, a number, a quoted string or an identifier.

3.2.3 Whitespace Characters
The following characters shall be considered whitespace characters:

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII code (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>20</td>
</tr>
<tr>
<td>vertical tab</td>
<td>0B</td>
</tr>
<tr>
<td>horizontal tab</td>
<td>09</td>
</tr>
<tr>
<td>line feed (new line)</td>
<td>0A</td>
</tr>
<tr>
<td>carriage return</td>
<td>0D</td>
</tr>
<tr>
<td>form feed</td>
<td>0C</td>
</tr>
</tbody>
</table>

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 | _ | $

letter ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

escape_character ::= \
any_character ::= 
  reserved_character
  | nonreserved_character
  | escape_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 a single character delimiter list shall be a single delimiter unless this character is used as a character in a compound operator or as a character in a quoted string.

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 Numbers

Constant numbers can be specified as integer or real. The integer is a decimal integer constant.

sign ::= + | -
**Lexical rules**

```
unsigned ::=  digit  { _ | digit }

integer ::=  [ sign ] unsigned

non_negative_number ::= 
    unsigned [ . unsigned ]
    | unsigned [ . unsigned ] E [ sign ] unsigned

number ::=  
    [ sign ] non_negative_number
```

**Figure 3-12: Integer and real numbers**

### 3.2.8 Bit Literals

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

```
bit_literal ::=  
    numeric_bit_literal
    | alphabetic_bit_literal
    | dont_care_literal
    | random_literal

numeric_bit_literal ::=  0 | 1

alphabetic_bit_literal ::= 
    X | Z | L | H | U | W
    | x | z | l | h | u | w

dont_care_literal ::=  ?

random_literal ::=  *
```

**Table 3-1: Single bit constants**

<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>value is logic zero</td>
</tr>
<tr>
<td>1</td>
<td>value is logic one</td>
</tr>
<tr>
<td>X or x</td>
<td>value is unknown</td>
</tr>
<tr>
<td>L or l</td>
<td>value is logic zero with weak drive strength</td>
</tr>
<tr>
<td>H or h</td>
<td>value is logic one with weak drive strength</td>
</tr>
<tr>
<td>W or w</td>
<td>value is unknown with weak drive strength</td>
</tr>
<tr>
<td>Z or z</td>
<td>value is high-impedance</td>
</tr>
<tr>
<td>U or u</td>
<td>value is uninitialized</td>
</tr>
<tr>
<td>?</td>
<td>value is any of the above, yet stable</td>
</tr>
<tr>
<td>*</td>
<td>value may randomly change</td>
</tr>
</tbody>
</table>
3.2.9 Based Literals

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

\[
\text{based\_literal ::= }
\begin{align*}
\text{binary\_base } & \quad (\_ | \text{binary\_digit }) \\
\text{octal\_base } & \quad (\_ | \text{octal\_digit }) \\
\text{decimal\_base } & \quad (\_ | \text{digit }) \\
\text{hex\_base } & \quad (\_ | \text{hex\_digit })
\end{align*}
\]

\[
\text{binary\_base ::= }
\begin{align*}
'B' | 'b'
\end{align*}
\]

\[
\text{octal\_base ::= }
\begin{align*}
'O' | 'o'
\end{align*}
\]

\[
\text{decimal\_base ::= }
\begin{align*}
'D' | 'd'
\end{align*}
\]

\[
\text{hex\_base ::= }
\begin{align*}
'H' | 'h'
\end{align*}
\]

\[
\text{binary\_digit ::= }
\begin{align*}
\text{bit\_literal}
\end{align*}
\]

\[
\text{octal\_digit ::= }
\begin{align*}
\text{binary\_digit} | 2 | 3 | 4 | 5 | 6 | 7
\end{align*}
\]

\[
\text{hex\_digit ::= }
\begin{align*}
\text{octal\_digit} | 8 | 9 | A | B | C | D | E | F | a | b | c | d | e | f
\end{align*}
\]

Figure 3-13: 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.

When an alphabetic bit literal is used as an octal digit, it shall represent 3 repeated bits with the same literal. When an alphabetic bit literal is used as a hex digit, it shall represent 4 repeated bits with the same literal.

For example,

\begin{align*}
'o2xw0u' & \quad \text{is same as} \quad 'b010\_xxx\_www\_000\_uuu' \\
'hLux' & \quad \text{is same as} \quad 'bLLL\_uuuu\_xxxx'
\end{align*}
3.2.10 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 3-14: Edge literals

3.2.11 Quoted Strings

The quoted string shall be a sequence of zero or more characters enclosed between two quotation marks (") and contained on a single line. Character escape codes are used inside the string literal to represent some common special characters. The characters that may follow the backslash (\) and their meanings are listed below in Table 3-2.

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

Figure 3-15: A quoted string

Table 3-2: Special characters in quoted strings

<table>
<thead>
<tr>
<th>Symbol</th>
<th>ASCII Code (octal)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\g</td>
<td>007</td>
<td>alert/bell</td>
</tr>
<tr>
<td>\h</td>
<td>010</td>
<td>backspace</td>
</tr>
<tr>
<td>\t</td>
<td>011</td>
<td>horizontal tab</td>
</tr>
<tr>
<td>\n</td>
<td>012</td>
<td>new line</td>
</tr>
<tr>
<td>\v</td>
<td>013</td>
<td>vertical tab</td>
</tr>
<tr>
<td>\f</td>
<td>014</td>
<td>form feed</td>
</tr>
<tr>
<td>\r</td>
<td>015</td>
<td>carriage return</td>
</tr>
<tr>
<td>&quot;</td>
<td>042</td>
<td>double quotation mark</td>
</tr>
</tbody>
</table>
Library Format Specification

Lexical rules

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

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

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

A sequence of characters starting with an escape_character is called an escaped identifier.
The purpose of the escaped identifier is to legalize the use of a digit as first character of an
identifier, the use of reserved_character anywhere in an identifier or to prevent the
misinterpretation of an identifier as a keyword. The escape character shall be followed by at
least one non-white space character to form an escaped identifier. The escaped identifier shall
contain all characters up to first white space character.

```
escaped_identifier ::= escape_character escaped_characters
escaped_characters ::= escaped_character { escaped_character }
escaped_character ::= nonreserved_character | reserved_character | escape_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 a formal parameter, which is replaced by the actual parameter in template
instantiation.

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

Identifiers are treated in a case-insensitive way. They may be used in the definition of objects
and in reference to already defined objects. A parser should preserve the case of an identifier
in the definition of an object, since a downstream application may be case-sensitive.
3.2.13 Rules against parser ambiguity

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

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

- In a context where both `bit_literal` and `number` are legal syntax items, `number` shall take priority over `numeric_bit_literal`.

- In a context where both `edge_literal` and `identifier` are legal syntax items, `identifier` shall take priority over `bit_edge_literal`.

- In a context where both `edge_literal` and `number` are legal syntax items, `number` shall take priority over `bit_edge_literal`.

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

3.2.14 Cross-reference of lexical tokens

<table>
<thead>
<tr>
<th>Lexical token</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphabetic_bit_literal</td>
<td>3.2.8</td>
</tr>
<tr>
<td>any_character</td>
<td>3.2.4</td>
</tr>
<tr>
<td>based_literal</td>
<td>3.2.9</td>
</tr>
<tr>
<td>binary_base</td>
<td>3.2.9</td>
</tr>
<tr>
<td>binary_digit</td>
<td>3.2.9</td>
</tr>
<tr>
<td>bit_edge_literal</td>
<td>3.2.10</td>
</tr>
<tr>
<td>bit_literal</td>
<td>3.2.8</td>
</tr>
<tr>
<td>block_comment</td>
<td>3.2.6</td>
</tr>
<tr>
<td>comment</td>
<td>3.2.6</td>
</tr>
<tr>
<td>decimal_base</td>
<td>3.2.9</td>
</tr>
<tr>
<td>delimiter</td>
<td>3.2.5</td>
</tr>
<tr>
<td>digit</td>
<td>3.2.4</td>
</tr>
<tr>
<td>dont_care_literal</td>
<td>3.2.8</td>
</tr>
<tr>
<td>edge_literal</td>
<td>3.2.10</td>
</tr>
<tr>
<td>escape_character</td>
<td>3.2.4</td>
</tr>
<tr>
<td>escaped_identifier</td>
<td>3.2.12</td>
</tr>
<tr>
<td>hex_base</td>
<td>3.2.9</td>
</tr>
<tr>
<td>hex_digit</td>
<td>3.2.9</td>
</tr>
<tr>
<td>integer</td>
<td>3.2.7</td>
</tr>
<tr>
<td>nonescaped_identifier</td>
<td>3.2.12</td>
</tr>
<tr>
<td>non_negative_number</td>
<td>3.2.7</td>
</tr>
</tbody>
</table>
Keywords are case-insensitive non-escaped identifiers. For clarity, this document uses uppercase letters for keywords and lowercase letters elsewhere, unless otherwise mentioned.

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

\begin{verbatim}
PIN \PIN {..}
\end{verbatim}

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 that contain power related information described using soft keywords.

### 3.3.1 Keywords for Objects

The following keywords are used to identify object types:

\begin{verbatim}
ALIAS CLASS GROUP PIN SUBLIBRARY WIRE
ATTRIBUTE CONSTANT HEADER PRIMITIVE SUBLIBRARY TABLE
BEHAVIOR EQUATION INCLUDE PROPERTY TEMPLATE
CELL FUNCTION LIBRARY STATETABLE VECTOR
\end{verbatim}

Figure 3-16: Keywords for objects
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 3-17: Keywords for built-in arithmetic functions](image)

3.3.3 Context-Sensitive Keywords

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

The public context-sensitive keywords and their semantic meaning are defined in Section 3.6. This set can be extended to include private context-sensitive keywords.

3.4 Syntax Rules

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

3.4.1 Assignments

```plaintext
unnamed_assignment_base ::= 
    context_sensitive_keyword = value 

unnamed_assignment ::= 
    unnamed_assignment_base ; 

unnamed_assignments ::= 
    unnamed_assignment { unnamed_assignment }

named_assignment_base ::= 
    context_sensitive_keyword identifier = value 

named_assignment ::= 
    named_assignment_base ; 

named_assignments ::= 
    named_assignment { named_assignment }

assignment_base ::= 
    named_assignment_base 
    | unnamed_assignment_base 

multi_value_assignment ::= 
    identifier { values }
```
assignment ::= 
    named_assignment  
  |  unnamed_assignment  
  |  multi_value_assignment  

pin_assignment ::= 
    pin_identifier [index] = pin_identifier [index] ;  
  |  pin_identifier [index] = logic_constant ;  
  |  logic_constant = pin_identifier [index] ;  

pin_assignments ::= 
    pin_assignment { pin_assignment }  

arithmetic_assignment ::=  
    identifier = arithmetic_expression ;  

3.4.2 Expressions  

arithmetic_expression ::=  
    ( arithmetic_expression )  
  |  number  
  |  [ arithmetic_unary ] identifier  
  |  arithmetic_expression arithmetic_binary arithmetic_expression  
  |  arithmetic_function_operator  
    ( arithmetic_expression { , arithmetic_expression } )  

boolean_expression ::=  
    ( boolean_expression )  
  |  logic_constant  
  |  logic_variable  
  |  boolean_unary boolean_expression  
  |  boolean_expression boolean_binary boolean_expression  
  |  boolean_expression  
    boolean_cond boolean_expression boolean_else  
    { boolean_expression boolean_cond boolean_else }  
    boolean_expression  

vector_single_event ::=  
    ( vector_single_event )  
  |  vector_unary boolean_expression  

vector_event ::=  
    ( vector_event )  
  |  vector_single_event  
  |  vector_event vector_and vector_event  

vector_event_sequence ::=  
    ( vector_event_sequence )  
  |  vector_event  
  |  vector_event_sequence vector_followed_by vector_event_sequence
vector_complex_event ::= 
    ( vector_complex_event )
    | vector_event_sequence
    | vector_complex_event vector_binary vector_complex_event

vector_conditional_event ::= 
    vector_expression vector_boolean_and boolean_expression
    | boolean_expression vector_boolean_and vector_expression
    | boolean_expression vector_boolean_cond vector_expression
    | vector_boolean_else
        { boolean_expression vector_boolean_cond vector_expression
          vector_boolean_else } vector_expression

vector_expression ::= 
    ( vector_expression )
    | vector_complex_event
    | vector_conditional_event
    | vector_expression vector_binary vector_expression

vector_or_boolean_expression ::= 
    vector_expression
    | boolean_expression

3.4.3 Instantiations

cell_instantiation ::= 
    cell_identifier { logic_values }
    | cell_identifier { pin_assignments }

primitive_instantiation ::= 
    primitive_identifier [ identifier ] { logic_values }
    | primitive_identifier [ identifier ] { logic_assignments }
    | primitive_identifier [ identifier ] { pin_assignments }

template_instantiation ::= 
    template_identifier ;
    | template_identifier [ = static ] { values }
    | template_identifier [ = static ] { all_purpose_items }
    | template_identifier = dynamic { values }
    | template_identifier = dynamic { dynamic_instantiation_items }

dynamic_instantiation_items ::= 
    dynamic_instantiation_item { dynamic_instantiation_item }

dynamic_instantiation_item ::= 
    all_purpose_item
    | arithmetic_model
    | arithmetic_assignment
### 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 ::= 
  [ unsigned ] 
  | [ unsigned : unsigned ] 
  | [ identifier ] 
  | [ identifier : identifier ]

logic_value ::= 
  logic_constant 
  | logic_variable

logic_values ::= 
  logic_value { logic_value }

logic_constant ::= 
  bit_literal 
  | based_literal

logic_constants ::= 
  logic_constant { logic_constant }

statetable_value ::= 
  logic_constant 
  | edge_literal 
  | ( [!] logic_variable )

statetable_values ::= 
  statetable_value { statetable_value }

logic_variable ::= 
  pin_identifier [ index ]
### Syntax Rules

```plaintext
equation

logic_variables ::= 
    logic_variable { logic_variable }

numbers ::= 
    number { number }

string ::= 
    quoted_string 
    | identifier

value ::= 
    number 
    | string 
    | logic_value

values ::= 
    value { value }

#### 3.4.5 Operators

arithmetic_unary ::= 
    + | -

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

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

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

boolean_and ::= 
    & | &&

boolean_or ::= 
    | | ||

boolean_logic_compare ::= 
    ^ | ~^ 

boolean_case_compare ::= 
    != | == | >= | <= | > | <

boolean_arithmetic ::= 
    + | - | * | / | % | >> | <<
```
boolean_binary ::= 
  boolean_and 
  |  boolean_or 
  |  boolean_logic_compare 
  |  boolean_case_compare 
  |  boolean_arithmetic 

boolean_cond ::= 
  ?

boolean_else ::= 
  :

vector_unary ::= 
  edge_literal 

vector_and ::= 
  &  |  &&

vector_or ::= 
  |  |  ||

vector_followed_by ::= 
  ->  |  ~> 

vector_binary ::= 
  vector_and 
  |  vector_or 
  |  vector_followed_by 
  |  <->
  |  &>
  |  <&>

vector_boolean_and ::= 
  &  |  &&

vector_boolean_cond ::= 
  ?

vector_boolean_else ::= 
  :

sequential_if ::= 
  @

sequential_else_if ::= 
  :

See Section 3.5 for semantics of 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
| assignment_base { all_purpose_items }

annotation_container ::= context_sensitive_keyword { all_purpose_items }

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

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

source_text ::= 
  ALF_REVISION  version_string 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 { class_items }

class_item ::= 
    all_purpose_item 
    | logic_assignment 
    | vector_assignment

class_items ::= 
    class_item { class_item }

constant ::= 
    CONSTANT identifier = number ; 
    | CONSTANT identifier = logic_constant ;

group ::= 
    GROUP group_identifier { identifiers } 
    | GROUP group_identifier { numbers } 
    | GROUP group_identifier { edge_literals } 
    | GROUP group_identifier { logic_constants } 
    | GROUP group_identifier { logic_variables } 
    | GROUP group_identifier { integer : integer }

include ::= 
    INCLUDE quoted_string ;

property ::= 
    PROPERTY [ identifier ] { unnamed_assignments }

template_item ::= 
    all_purpose_item 
    | library_specific_object 
    | arithmetic_model 
    | header 
    | table 
    | equation 
    | behavior_item

template_items ::= 
    template_item { template_item }

template ::= 
    TEMPLATE template_identifier { template_items }
3.4.8 CELL

cell ::= CELL cell_identifier { cell_items }
| CELL cell_identifier ;
| cell_template_instantiation

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

cell_items ::= cell_item {cell_item}

3.4.9 LIBRARY

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

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

pins ::= pin { pin }

pin_item ::= all_purpose_item
| arithmetic_model
pin_items ::= 
  pin_item { pin_item }

3.4.11 PRIMITIVE

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

primitives ::= 
  primitive { primitive }

primitive_item ::= 
  all_purpose_item 
  | pin 
  | function

primitive_items ::= 
  primitive_item { primitive_item }

3.4.12 SUBLIBRARY

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

sublibraries ::= 
  sublibrary { sublibrary }

3.4.13 VECTOR

vector ::= 
  VECTOR ( vector_or_boolean_expression ) { vector_items }
  | VECTOR ( vector_or_boolean_expression ) ;
  | vector_template_instantiation

vector_item ::= 
  all_purpose_item
  | arithmetic_model
  | logic_assignment
  | vector_assignment

vector_items ::= 
  vector_item { vector_item }

vector_assignment ::= 
  context_sensitive_keyword = { vector_expression } ;
3.4.14 WIRE

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

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 ] body }  
    | context_sensitive_keyword [ identifier ]  
    = value ;  
    | context_sensitive_keyword [ identifier ]  
    = value { all_purpose_items }  
    | context_sensitive_keyword [ identifier ]  
    { arithmetic_submodels }  
    | arithmetic_model_template_instantiation

arithmetic_models ::=  
arithmetic_model { arithmetic_model }

arithmetic_model_container ::=  
context_sensitive_keyword { arithmetic_models }

arithmetic_submodel ::= context_sensitive_keyword  
    { [ all_purpose_items ] [ header ] body }  
    | context_sensitive_keyword  
    = value ;  
    | context_sensitive_keyword  
    = value { all_purpose_items }  
    | context_sensitive_keyword  
    { arithmetic_submodels }  
    | arithmetic_submodel_template_instantiation

arithmetic_submodels ::=  
arithmetic_submodel { arithmetic_submodel }

header ::= HEADER { [ all_purpose_items ] arithmetic_models }  
    | header_template_instantiation
```
body ::= 
    table 
    | equation 
    | table equation 

table ::= 
    TABLE { table_items } 
    | table_template_instantiation 

table_item ::= 
    number 
    | identifier 

table_items ::= 
    table_item { table_item } 

equation ::= 
    EQUATION { arithmetic_expression } 
    | equation_template_instantiation 

3.4.16 FUNCTION 

function ::= 
    FUNCTION [ identifier ] 
    { [all_purpose_items] [primitives] behavior } 
    | { [all_purpose_items] [primitives] [behavior] statetables } 
    | function_template_instantiation 

statetable ::= 
    STATETABLE [ identifier ] { statetable_header statetable_body } 

statetables ::= 
    statetable { statetable } 

statetable_body ::= 
    statetable_values : statetable_values ; 
    { statetable_values : statetable_values ; } 

statetable_header ::= 
    logic_variables : logic_variables ; 

behavior ::= 
    BEHAVIOR [ identifier ] { behavior_items } 

behavior_item ::= 
    logic_assignment 
    | sequential_logic_statement 
    | primitive_instantiation 

behavior_items ::= 
    behavior_item { behavior_item }
logic_assignment ::=  
    identifier [index] = boolean_expression ;

logic_assignments ::=  
    logic_assignment  { logic_assignment }

sequential_logic_statement ::=  
    sequential_if ( vector_or_boolean_expression )  
    { logic_assignments }  
    { sequential_else_if ( vector_or_boolean_expression )  
      { logic_assignments } } 

3.4.17 Cross-reference of BNF items

Note: A BNF item with singular name is defined in the same section as the BNF item with the plural name. A plural item name implies one or more items with the corresponding singular name.

Table 3-4 : Cross-reference of BNF items with short semantic explanation

<table>
<thead>
<tr>
<th>BNF item</th>
<th>Section</th>
<th>Short semantic explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias</td>
<td>3.4.7</td>
<td>statement defining an alias</td>
</tr>
<tr>
<td>all_purpose_item(s)</td>
<td>3.4.6</td>
<td>item(s) that can appear inside any hierarchical object</td>
</tr>
<tr>
<td>annotation</td>
<td>3.4.6</td>
<td>parameter-value assignment inside an object, may be nested</td>
</tr>
<tr>
<td>annotation_container</td>
<td>3.4.6</td>
<td>unnamed object containing annotations</td>
</tr>
<tr>
<td>arithmetic_assignment</td>
<td>3.4.1</td>
<td>statement assigning an arithmetic expression to a variable</td>
</tr>
<tr>
<td>arithmetic_binary</td>
<td>3.4.5</td>
<td>arithmetic operator requiring two operands</td>
</tr>
<tr>
<td>arithmetic_expression</td>
<td>3.4.2</td>
<td>expression involving arithmetic operations</td>
</tr>
<tr>
<td>arithmetic_function_operator</td>
<td>3.4.5</td>
<td>arithmetic operator prefixing a list of arguments</td>
</tr>
<tr>
<td>arithmetic_model(s)</td>
<td>3.4.15</td>
<td>statement(s) for description of characterization data using single numbers, tables or equations</td>
</tr>
<tr>
<td>arithmetic_model_container</td>
<td>3.4.15</td>
<td>unnamed object containing arithmetic models</td>
</tr>
<tr>
<td>arithmetic_submodel(s)</td>
<td>3.4.15</td>
<td>statement(s) inside an arithmetic model statement for categorizing the characterization data</td>
</tr>
<tr>
<td>arithmetic_unary</td>
<td>3.4.5</td>
<td>arithmetic operator requiring one operand</td>
</tr>
<tr>
<td>assignment</td>
<td>3.4.1</td>
<td>terminated statement for single value assignment to an object</td>
</tr>
<tr>
<td>assignment_base</td>
<td>3.4.1</td>
<td>unterminated statement for single value assignment to an object</td>
</tr>
<tr>
<td>attribute</td>
<td>3.4.7</td>
<td>statement associating attributes to an object</td>
</tr>
<tr>
<td>attribute_item(s)</td>
<td>3.4.7</td>
<td>item(s) inside an attribute statement</td>
</tr>
<tr>
<td>behavior</td>
<td>3.4.16</td>
<td>statement describing the logic function of a digital circuit in a behavioral language</td>
</tr>
<tr>
<td>behavior_item(s)</td>
<td>3.4.16</td>
<td>item(s) inside a behavior statement</td>
</tr>
<tr>
<td>BNF item</td>
<td>Section</td>
<td>Short semantic explanation</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------</td>
<td>-------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>body</td>
<td>3.4.15</td>
<td>table or equation defining characterization data for an arithmetic model</td>
</tr>
<tr>
<td>boolean_and</td>
<td>3.4.5</td>
<td>boolean AND operator</td>
</tr>
<tr>
<td>boolean_arithmetic</td>
<td>3.4.5</td>
<td>operator for boolean arithmetic</td>
</tr>
<tr>
<td>boolean_binary</td>
<td>3.4.5</td>
<td>boolean operator requiring two operands</td>
</tr>
<tr>
<td>boolean_case_compare</td>
<td>3.4.5</td>
<td>binary boolean operator for magnitude comparison</td>
</tr>
<tr>
<td>boolean_cond</td>
<td>3.4.5</td>
<td>boolean postfix operator evaluating the preceding boolean expression (if-clause)</td>
</tr>
<tr>
<td>boolean_else</td>
<td>3.4.5</td>
<td>boolean infix operator separating if-and else-clauses</td>
</tr>
<tr>
<td>boolean_expression</td>
<td>3.4.2</td>
<td>expression involving boolean operations</td>
</tr>
<tr>
<td>boolean_logic_compare</td>
<td>3.4.5</td>
<td>binary boolean operator for logic comparison</td>
</tr>
<tr>
<td>boolean_or</td>
<td>3.4.5</td>
<td>boolean OR operator</td>
</tr>
<tr>
<td>boolean_unary</td>
<td>3.4.5</td>
<td>boolean operator requiring one operand</td>
</tr>
<tr>
<td>cell(s)</td>
<td>3.4.8</td>
<td>statement(s) describing the entire model of a digital or analog circuit</td>
</tr>
<tr>
<td>cell_item(s)</td>
<td>3.4.8</td>
<td>item(s) inside a cell statement</td>
</tr>
<tr>
<td>cell_instantiation</td>
<td>3.4.3</td>
<td>statement inside a cell, describing a reference to another cell with pin-to-pin correspondence</td>
</tr>
<tr>
<td>class</td>
<td>3.4.7</td>
<td>statement describing a class for the use of reference and inheritance by other objects</td>
</tr>
<tr>
<td>class_item(s)</td>
<td>3.4.6</td>
<td>item(s) inside a class statement, which will be inherited by any object referring to the class</td>
</tr>
<tr>
<td>constant</td>
<td>3.4.7</td>
<td>statement defining a numeric constant</td>
</tr>
<tr>
<td>context_sensitive_keyword</td>
<td>3.4.4</td>
<td>identifier of an object for which the semantic meaning is established by its context</td>
</tr>
<tr>
<td>dynamic_instantiation_item(s)</td>
<td>3.4.3</td>
<td>item(s) inside a dynamic instantiation of a template</td>
</tr>
<tr>
<td>edge_literal(s)</td>
<td>3.4.4</td>
<td>symbol(s) describing a transition between two states</td>
</tr>
<tr>
<td>equation</td>
<td>3.4.15</td>
<td>statement inside arithmetic model containing an arithmetic expression for the calculation of characterization data</td>
</tr>
<tr>
<td>function</td>
<td>3.4.16</td>
<td>statement describing the logic function of a circuit in a canonical way, using behavior and/or statetable statement</td>
</tr>
<tr>
<td>generic_object</td>
<td>3.4.6</td>
<td>statement with the sole purpose of being used by other objects</td>
</tr>
<tr>
<td>group</td>
<td>3.4.7</td>
<td>statement allowing expansion of one object to multiple objects</td>
</tr>
<tr>
<td>header</td>
<td>3.4.15</td>
<td>statement inside arithmetic model containing a list of parameters of the arithmetic model</td>
</tr>
<tr>
<td>identifier(s)</td>
<td>3.4.4</td>
<td>literal(s) defining a keyword or a name or a string value</td>
</tr>
<tr>
<td>include</td>
<td>3.4.7</td>
<td>statement defining the inclusion of a file</td>
</tr>
</tbody>
</table>
### Table 3-4: Cross-reference of BNF items with short semantic explanation

<table>
<thead>
<tr>
<th>BNF item</th>
<th>Section</th>
<th>Short semantic explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>3.4.4</td>
<td>symbol defining an integer or a range of integers for the use as indices</td>
</tr>
<tr>
<td>library (libraries)</td>
<td>3.4.9</td>
<td>statement(s) describing the entire contents of a library</td>
</tr>
<tr>
<td>library_item(s)</td>
<td>3.4.9</td>
<td>item(s) inside a library statement</td>
</tr>
<tr>
<td>library_specific_object</td>
<td>3.4.6</td>
<td>statement describing an object which is part of the library</td>
</tr>
<tr>
<td>logic_assignment(s)</td>
<td>3.4.1</td>
<td>statement(s) assigning a logic expression to a logic variable</td>
</tr>
<tr>
<td>logic_value(s)</td>
<td>3.4.4</td>
<td>variable(s) or constant logic value(s)</td>
</tr>
<tr>
<td>logic_constant(s)</td>
<td>3.4.4</td>
<td>constant logic value(s)</td>
</tr>
<tr>
<td>logic_variable(s)</td>
<td>3.4.4</td>
<td>variable(s) containing a logic value</td>
</tr>
<tr>
<td>multi_value_assignment</td>
<td>3.4.1</td>
<td>statement for assignment of multiple values to an object</td>
</tr>
<tr>
<td>named_assignment</td>
<td>3.4.1</td>
<td>terminated statement for single value assignment to a named object</td>
</tr>
<tr>
<td>named_assignment_base</td>
<td>3.4.1</td>
<td>unterminated statement for single value assignment to a named object</td>
</tr>
<tr>
<td>number(s)</td>
<td>3.4.4</td>
<td>integer or floating point number(s)</td>
</tr>
<tr>
<td>pin(s)</td>
<td>3.4.10</td>
<td>statement(s) describing a pin inside a cell</td>
</tr>
<tr>
<td>pin_item(s)</td>
<td>3.4.10</td>
<td>item(s) inside a pin statement</td>
</tr>
<tr>
<td>pin_assignment(s)</td>
<td>3.4.1</td>
<td>statement(s) defining a correspondence between two pins or between a pin and a constant logic value</td>
</tr>
<tr>
<td>primitive(s)</td>
<td>3.4.11</td>
<td>statement(s) describing a technology-independent cell</td>
</tr>
<tr>
<td>primitive_instantiation</td>
<td>3.4.3</td>
<td>statement inside a behavior statement for logic function description by reference to a primitive</td>
</tr>
<tr>
<td>primitive_item(s)</td>
<td>3.4.11</td>
<td>item(s) inside a primitive statement</td>
</tr>
<tr>
<td>property</td>
<td>3.4.7</td>
<td>statement describing private properties without standardized semantics</td>
</tr>
<tr>
<td>sequential_else_if</td>
<td>3.4.5</td>
<td>operator indicating a lower-priority logic state or event sequence</td>
</tr>
<tr>
<td>sequential_if</td>
<td>3.4.5</td>
<td>operator indicating a top-priority logic state or event sequence</td>
</tr>
<tr>
<td>sequential_logic_statement</td>
<td>3.4.1</td>
<td>statement inside a behavior statement for logic function description with storage elements</td>
</tr>
<tr>
<td>source_text</td>
<td>3.4.6</td>
<td>contents of a self-sufficient file in ALF</td>
</tr>
<tr>
<td>statetable(s)</td>
<td>3.4.16</td>
<td>statement(s) describing the logic function of a digital circuit in table format</td>
</tr>
<tr>
<td>statetable_body</td>
<td>3.4.16</td>
<td>list of values inside a statetable</td>
</tr>
<tr>
<td>statetable_header</td>
<td>3.4.16</td>
<td>list of variables inside a statetable</td>
</tr>
<tr>
<td>statetable_value(s)</td>
<td>3.4.4</td>
<td>literal(s) inside a statetable</td>
</tr>
<tr>
<td>string</td>
<td>3.4.4</td>
<td>identifier consisting of a restricted set of characters or quoted string containing arbitrary characters</td>
</tr>
</tbody>
</table>
Table 3-4: Cross-reference of BNF items with short semantic explanation

<table>
<thead>
<tr>
<th>BNF item</th>
<th>Section</th>
<th>Short semantic explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sublibrary (sublibraries)</td>
<td>3.4.12</td>
<td>statement(s) describing the contents of a sub-library inside a library</td>
</tr>
<tr>
<td>table</td>
<td>3.4.15</td>
<td>statement inside arithmetic model containing a list of characterization data</td>
</tr>
<tr>
<td>table_item(s)</td>
<td>3.4.15</td>
<td>item(s) inside a table statement</td>
</tr>
<tr>
<td>template</td>
<td>3.4.7</td>
<td>statement defining an object with placeholders</td>
</tr>
<tr>
<td>template_instantiation</td>
<td>3.4.3</td>
<td>statement referring to a template and filling the placeholders</td>
</tr>
<tr>
<td>template_item(s)</td>
<td>3.4.7</td>
<td>statement(s) inside a template statement</td>
</tr>
<tr>
<td>unnamed_assignment(s)</td>
<td>3.4.1</td>
<td>terminated statement(s) for single value assignment to an unnamed object</td>
</tr>
<tr>
<td>unnamed_assignment_base</td>
<td>3.4.1</td>
<td>unterminated statement for single value assignment to an unnamed object</td>
</tr>
<tr>
<td>value(s)</td>
<td>3.4.4</td>
<td>number(s) or string(s) or logic value(s)</td>
</tr>
<tr>
<td>vector(s)</td>
<td>3.4.13</td>
<td>statement(s) describing event sequence and data for characterization of a circuit</td>
</tr>
<tr>
<td>vector_and</td>
<td>3.4.5</td>
<td>operator used for description of simultaneous events or simultaneous event sequences</td>
</tr>
<tr>
<td>vector_binary</td>
<td>3.4.5</td>
<td>operator requiring two operands used for description of event sequences</td>
</tr>
<tr>
<td>vector_boolean_and</td>
<td>3.4.5</td>
<td>operator used for description of event sequences with condition, one operand is an expression describing a complex event, other operand is a boolean expression</td>
</tr>
<tr>
<td>vector_boolean_cond</td>
<td>3.4.5</td>
<td>condition operator indicating if-clause</td>
</tr>
<tr>
<td>vector_boolean_else</td>
<td>3.4.5</td>
<td>condition operator separating if-and else-clauses</td>
</tr>
<tr>
<td>vector_complex_event</td>
<td>3.4.2</td>
<td>expression describing complex event sequences without condition</td>
</tr>
<tr>
<td>vector_conditional_event</td>
<td>3.4.2</td>
<td>expression describing complex event sequences with condition</td>
</tr>
<tr>
<td>vector_event_sequence</td>
<td>3.4.2</td>
<td>expression describing one event sequence</td>
</tr>
<tr>
<td>vector_expression</td>
<td>3.4.2</td>
<td>expression describing complex event sequences</td>
</tr>
<tr>
<td>vector_followed_by</td>
<td>3.4.5</td>
<td>operator used for description of subsequent events</td>
</tr>
<tr>
<td>vector_item(s)</td>
<td>3.4.13</td>
<td>item(s) inside a vector statement</td>
</tr>
<tr>
<td>vector_event</td>
<td>3.4.2</td>
<td>expression describing one single event or multiple simultaneous events</td>
</tr>
<tr>
<td>vector_or_boolean_expression</td>
<td>3.4.2</td>
<td>a vector expression or a boolean expression</td>
</tr>
<tr>
<td>vector_expression</td>
<td>3.4.2</td>
<td>expression describing complex event sequences</td>
</tr>
<tr>
<td>vector_single_event</td>
<td>3.4.2</td>
<td>expression describing one single event</td>
</tr>
<tr>
<td>vector_unary</td>
<td>3.4.5</td>
<td>operator requiring one operand used for description of event sequences</td>
</tr>
</tbody>
</table>
3.5 Operators

The operators are divided into the following groups:

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

3.5.1 Arithmetic operators

Table 3-5, Table 3-6, and Table 3-7 list unary, binary and function arithmetic operators.

**Table 3-5 : Unary arithmetic operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>positive sign (for integer or number)</td>
</tr>
<tr>
<td>-</td>
<td>negative sign (for integer or number)</td>
</tr>
</tbody>
</table>

**Table 3-6 : Binary arithmetic operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition (integer or number)</td>
</tr>
<tr>
<td>-</td>
<td>subtraction (integer or number)</td>
</tr>
<tr>
<td>*</td>
<td>multiplication (integer or number)</td>
</tr>
<tr>
<td>/</td>
<td>division (integer or number)</td>
</tr>
<tr>
<td>**</td>
<td>exponentiation (integer or number)</td>
</tr>
<tr>
<td>%</td>
<td>modulo division (integer or number)</td>
</tr>
</tbody>
</table>

**Table 3-7 : Function arithmetic operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG</td>
<td>natural logarithm (argument is + integer or number)</td>
</tr>
<tr>
<td>EXP</td>
<td>natural exponential (argument is integer or number)</td>
</tr>
<tr>
<td>ABS</td>
<td>absolute value (argument is integer or number)</td>
</tr>
</tbody>
</table>
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

Table 3-8, Table 3-9 and Table 3-10 list unary, binary and ternary boolean operators on scalars.

**Table 3-8 : Unary boolean operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!</code>, <code>~</code></td>
<td>logical inversion</td>
</tr>
</tbody>
</table>

**Table 3-9 : Binary boolean operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;&amp;</code>, <code>&amp;</code></td>
<td>logical AND</td>
</tr>
<tr>
<td>`</td>
<td></td>
</tr>
<tr>
<td><code>~^</code></td>
<td>logic equivalence (XNOR)</td>
</tr>
<tr>
<td><code>^</code></td>
<td>logic antivalence (XOR)</td>
</tr>
</tbody>
</table>

**Table 3-10 : Ternary operator**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>?</code></td>
<td>boolean condition operator for construction of combinational if-then-else clause</td>
</tr>
<tr>
<td><code>:</code></td>
<td>boolean else operator for construction of combinational if-then-else clause</td>
</tr>
</tbody>
</table>

Combinational if-then-else clauses are constructed as follows:

```
<cond1>? <value1>: <cond2>? <value2>: <cond3>? <value3>: <default_value>
```

If `cond1` evaluates to boolean TRUE then `value1` is the result, else if `cond2` evaluates to boolean TRUE then `value2` is the result, else if `cond3` evaluates to boolean TRUE then `value3` is the result, else `default_value` is the result of this clause.
3.5.3 Boolean operators on words

Table 3-11 and Table 3-12 list unary and binary reduction operators on words (logic variables with one or more bits). The result of an expression using these operators shall be a logic value.

Table 3-11: Unary reduction operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>AND all bits</td>
</tr>
<tr>
<td>~&amp;</td>
<td>NAND all bits</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>XOR all bits</td>
</tr>
<tr>
<td>~^</td>
<td>XNOR all bits</td>
</tr>
</tbody>
</table>

Table 3-12: Binary reduction operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equality for case comparison</td>
</tr>
<tr>
<td>!=</td>
<td>non-equality for case comparison</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater</td>
</tr>
<tr>
<td>&lt;</td>
<td>smaller</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater or equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>smaller or equal</td>
</tr>
</tbody>
</table>

Table 3-13 and Table 3-14 list unary and binary bitwise operators. The result of an expression using these operators shall be an array of bits.

Table 3-13: Unary bitwise operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>bitwise inversion</td>
</tr>
</tbody>
</table>

Table 3-14: Binary bitwise operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
</tr>
<tr>
<td>~^</td>
<td>bitwise XNOR</td>
</tr>
</tbody>
</table>
The following arithmetic operators, listed in Table 3-15, are also defined for boolean operations on words. The result of an expression using these operators shall be an extended array of bits.

**Table 3-15 : Binary operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>%</td>
<td>modulo division</td>
</tr>
</tbody>
</table>

The arithmetic operations addition, subtraction, multiplication, and division shall be *unsigned* if all the operands have the datatype *unsigned*. If any of the operands have the datatype signed, the operation shall be *signed*. See Table 3.6.3.13 for DATATYPE definition.

### 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.

```plaintext
bit bit  // apply transition from bit value to bit value
```

For example,

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

No whitespace shall be allowed between the two scalar constants. The transition operators shown in Table 3-16 shall be considered legal:

**Table 3-16 : Unary vector operators on bits**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>signal toggles from 0 to 1</td>
</tr>
<tr>
<td>10</td>
<td>signal toggles from 1 to 0</td>
</tr>
<tr>
<td>00</td>
<td>signal remains 0</td>
</tr>
<tr>
<td>11</td>
<td>signal remains 1</td>
</tr>
<tr>
<td>0?</td>
<td>signal remains 0 or toggles from 0 to arbitrary value</td>
</tr>
<tr>
<td>1?</td>
<td>signal remains 1 or toggles from 1 to arbitrary value</td>
</tr>
<tr>
<td>?0</td>
<td>signal remains 0 or toggles from arbitrary value to 0</td>
</tr>
<tr>
<td>?1</td>
<td>signal remains 1 or toggles from arbitrary value to 1</td>
</tr>
<tr>
<td>??</td>
<td>signal remains constant or toggles between arbitrary values</td>
</tr>
</tbody>
</table>
Unary operators for transitions can also appear in STATETABLE. Transition operators are also defined on words (can appear in STATETABLE as well):

\[ 'base \text{ word } \text{ base word} \]

In this context, the transition operator shall apply transition from first word value to second 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 unary and binary operators for transition, listed in Table 3-17 and Table 3-18 respectively, are defined on bits and words:

**Table 3-16 : Unary vector operators on bits or words**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0*</td>
<td>a number of arbitrary signal transitions, including possibility of constant value, with the initial value 0</td>
</tr>
<tr>
<td>1*</td>
<td>a number of arbitrary signal transitions, including possibility of constant value, with the initial value 1</td>
</tr>
<tr>
<td>?*</td>
<td>a number of arbitrary signal transitions, including possibility of constant value, with arbitrary initial value</td>
</tr>
<tr>
<td>*0</td>
<td>a number of arbitrary signal transitions, including possibility of constant value, with the final value 0</td>
</tr>
<tr>
<td>*1</td>
<td>a number of arbitrary signal transitions, including possibility of constant value, with the final value 1</td>
</tr>
<tr>
<td>*?</td>
<td>a number of arbitrary signal transitions, including possibility of constant value, with arbitrary final value</td>
</tr>
</tbody>
</table>

The following canonical binary operators are necessary to define sequences of transitions:

- **vector_followed_by** for completely specified sequence of events
- **vector_and** for simultaneous events
- **vector_or** for alternative events
- **vector_followed_by** for incompletely specified sequence of events
The symbols for the boolean operators for AND, OR, are overloaded for `vector_and`, `vector_or`, respectively. New symbols are introduced for the `vector_followed_by` operators.

### Table 3-18: Canonical Binary vector operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operands</th>
<th>LHS, RHS commutative</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-&gt;</code></td>
<td>2 vector expressions</td>
<td>no</td>
<td>Left-hand side (LHS) transition is followed by Right-hand side (RHS) transition, no transition may occur in-between</td>
</tr>
<tr>
<td><code>&amp; &amp;</code>, <code>&amp;</code></td>
<td>2 vector expressions</td>
<td>yes</td>
<td>LHS and RHS transition occur simultaneously</td>
</tr>
<tr>
<td>`</td>
<td></td>
<td><code>, </code></td>
<td>`</td>
</tr>
<tr>
<td><code>~&gt;</code></td>
<td>2 vector expressions</td>
<td>no</td>
<td>Left-hand side (LHS) transition is followed by Right-hand side (RHS) transition, other transitions may occur in-between</td>
</tr>
</tbody>
</table>

Per definition, the `->`, `~>` operators shall not be commutative, whereas the `& &`, `||` operators on events shall be commutative.

```
01 a & & 01 b === 01 b & & 01 a
01 a || 01 b === 01 b || 01 a
```

The `->`, `~>` operators shall be freely associative.

```
01 a -> 01 b -> 01 c === (01 a -> 01 b) -> 01 c === 01 a -> (01 b -> 01 c)
01 a -> 01 b -> 01 c === (01 a -> 01 b) -> 01 c === 01 a ~> (01 b ~> 01 c)
```

The `& &` operator is defined for single events and for event sequences with the same number of `->` operators each.

```
(01 A1 .. -> ... 01 AN) & (01 B1 .. -> ... 01 BN)
===
01 A1 & 01 B1 ... -> ... 01 AN & 01 BN
```

The `||` operator allows to reduce the set of edge operators (unary vector operators) to canonical and non-canonical operators.

```
(?? a) === (?! a) || (?- a) //a does or does not change its value
```

Hence `??` is non-canonical, since it can be defined by other operators.

If `<value1><value2>` is an edge operator consisting of two based literals `value1` and `value2` and `word` is an expression which can take the value `value1` or `value2`, then the following vector expressions are considered equivalent:

```
)value1)value2) <word>
===
10 (<word> == <value1>) & & 01 (<word> == <value2>)
===
01 (<word> ! = <value1>) & & 01 (<word> == <value2>)
===
10 (<word> == <value1>) & & 10 (<word> ! = <value2>)
===
01 (<word> ! = <value1>) & & 10 (<word> ! = <value2>)
```

// all expressions describe the same event:
// <word> makes a transition from <value1> to <value2>
Hence vector expressions with edge operators using based literals can be reduced to vector expressions using only the edge operators 01, 10.

Complex vector_binary operators are also defined. Vector expressions using those operators can be decomposed into vector expressions using only canonical operators.

### Table 3-19: Complex Binary vector operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operands</th>
<th>LHS, RHS commutative</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;-&gt;</td>
<td>2 vector expressions</td>
<td>yes</td>
<td>LHS transition follows or is followed by RHS transition</td>
</tr>
<tr>
<td>&amp;&gt;</td>
<td>2 vector expressions</td>
<td>no</td>
<td>LHS transition is followed by or occurs simultaneously with RHS transition</td>
</tr>
<tr>
<td>&lt;&amp;&gt;</td>
<td>2 vector expressions</td>
<td>yes</td>
<td>LHS transition follows or is followed by or occurs simultaneously with RHS transition</td>
</tr>
</tbody>
</table>

The following expressions shall be considered equivalent:

(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)

By their symmetric definition, the <->, <&> operators are commutative.

01 a <-> 01 b === 01 b <-> 01 a
01 a <&> 01 b === 01 b <&> 01 a

The definitions of the &&, ?, : operators are also overloaded to describe a *conditional vector expression*, involving boolean expressions and vector expressions. The clauses are boolean expressions, while vector expressions are subject to those clauses.

### Table 3-20: Operators for conditional vector expressions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operands</th>
<th>LHS, RHS commutative</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;, &amp;</td>
<td>1 vector expression, 1 boolean expression</td>
<td>yes</td>
<td>boolean expression (LHS or RHS) is true while sequence of transitions, defined by vector expression (RHS or LHS) occurs</td>
</tr>
<tr>
<td>?</td>
<td>1 vector expression, 1 boolean expression</td>
<td>no</td>
<td>boolean condition operator for construction of if-then-else clause involving vector expressions</td>
</tr>
<tr>
<td>:</td>
<td>1 vector expression, 1 boolean expression</td>
<td>no</td>
<td>boolean else operator for construction of if-then-else clause involving vector expressions</td>
</tr>
</tbody>
</table>
An example for conditional vector expression using && is given below:

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

The order of the operands in a conditional vector expression using && shall not matter.

```
<vector_exp> && <boolean_exp> === <boolean_exp> && <vector_exp>
```

The && operator is still commutative in this case, although one operand is a boolean expression defining a static state, the other operand is a vector expression defining an event or a sequence of events. However, since the operands are distinguishable per se, it is not necessary to impose a particular order of the operands.

An example for conditional vector expression using ?, : is given below.

```
!b ? 01 a : c ? 10 b : 01 d
```

```
!b & 01 a | !(b) & c & 10 b | !(b) & !c & 01 d
```

This example shows how a conditional vector expression using ternary operators can be expressed with alternative conditional vector expressions.

A conditional vector expression can be reduced to a non-conditional vector expression in some cases, as it will be explained in section 3.12.11.

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

### 3.5.5 Operators for sequential logic

#### Table 3-21 : Operators for sequential logic

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>sequential &quot;if&quot; operator, followed by a boolean logic expression (for level-sensitive assignment) or by a vector expression (for edge-sensitive assignment)</td>
</tr>
<tr>
<td>:</td>
<td>sequential &quot;else if&quot; operator, followed by a boolean logic expression (for level-sensitive assignment) or by a vector expression (for edge-sensitive assignment) with lower priority</td>
</tr>
</tbody>
</table>

Sequential assignments are constructed as follows:

```
@ ( <trigger1> ) { <action1> } : ( <trigger2> ) { <action2> } : 
( <trigger3> ) { <action3> }
```

If trigger1 event is detected then action1 is performed, else if trigger2 event is detected then action2 is performed, else if trigger3 event is detected then action3 is performed as a result of this clause.

### 3.5.6 Operator priorities

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

1. unary arithmetic operator (+, -)
2. exponentiation (**) 
3. multiplication (*), division (/), modulo division (%) 
4. addition (+), subtraction (-) 

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

1. unary boolean operator (!, ∼, &, ~&, │, ∼│, ^, ~^) 
2. XNOR (~^), XOR (^), relational (>, <, >=, <=, ==, !=), shift (<<, >>) 
3. AND ( &, &&), NAND (~&), multiply (*), divide (/), modulus (%) 
4. OR ( |, ||), NOR (~|), add (+), subtract (-) 
5. ternary operators (?:, :) 

The priority of binding operators to operands in non-conditional vector expressions shall be from strongest to weakest in the following order:

1. unary vector operators (edge literals) 
2. complex binary vector operators (<>-, &>, <&>) 
3. vector AND ( &, &&) 
4. vector_followed_by operators (->, ~>) 
5. vector OR ( |, ||) 

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.

For ternary operators and operators with hybrid operands, i.e., one operand is a non-conditional vector expression, the other operand is a boolean expression, each expression shall have a higher binding priority than the operands connecting the expressions. However, the use of parenthesis is recommended.

### 3.5.7 Datatype mapping

Logical operations can be applied to scalars and words. For that purpose, the values of the operands are reduced to a system of 3 logic values in the following way:

- H has the logic value 1 
- L has the logic value 0 
- ∀, Z, U have the logic value x 
- A word has the logic value 1, if the unary OR reduction of all bits results in 1 
- A word has the logic value 0, if the unary OR reduction of all bits results in 0 
- A word has the logic value x, if the unary OR reduction of all bits results in x
Case comparison operations can also be applied to scalars and words. For scalars, they are defined in the following way:

<table>
<thead>
<tr>
<th>Table 3-22 : Case comparison operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>H</td>
</tr>
<tr>
<td>H</td>
</tr>
<tr>
<td>H</td>
</tr>
<tr>
<td>H</td>
</tr>
<tr>
<td>H</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>L</td>
</tr>
<tr>
<td>L</td>
</tr>
<tr>
<td>L</td>
</tr>
<tr>
<td>L</td>
</tr>
<tr>
<td>L</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>W</td>
</tr>
<tr>
<td>W</td>
</tr>
<tr>
<td>W</td>
</tr>
<tr>
<td>Z</td>
</tr>
<tr>
<td>Z</td>
</tr>
<tr>
<td>Z</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

For word operands, the operations > and < are performed after reducing all bits to the 3-value system first, and then interpreting the resulting number according to the datatype of the operands. For example, if datatype is signed, 'b1111 is smaller than 'b0000; if datatype is unsigned, 'b1111 is greater than 'b0000. If two operands have the same value 'b1111 and a different datatype, the unsigned 'b1111 is greater than the signed 'b1111.
The operations \( \geq \) and \( \leq \) are defined in the following way:

\[
\begin{align*}
(a \geq b) &= (a > b) \lor (a == b) \\
(a \leq b) &= (a < b) \lor (a == b)
\end{align*}
\]

### 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 object with children objects may contain annotations. In addition, the following objects are defined only for the purpose of *unnamed annotation containers*.

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCAN</td>
<td>contains information relevant to design for test</td>
</tr>
<tr>
<td>VIOLATION</td>
<td>contains items relevant to timing violations</td>
</tr>
<tr>
<td>INFORMATION</td>
<td>contains purely informational items</td>
</tr>
</tbody>
</table>

#### 3.6.1.1 Scan container

A SCAN container may be used inside a CELL or a PIN object and may contain annotations which are allowed inside a CELL (Section 3.6.5) or a PIN object (Section 3.6.3) for limiting the scope of those annotations.

Example:

```plaintext
PIN clk1 { signaltype = master_clock; SCAN {signaltype = slave_clock;} }
PIN clk2 { SCAN {signaltype = master_clock;} }
```

In normal mode, `clk1` is master clock, `clk2` is unused. In scan mode, `clk2` is master clock, `clk1` is slave clock.

#### 3.6.1.2 VIOLATION container

A VIOLATION container may be inside a SETUP, HOLD, RECOVERY, REMOVAL, PULSEWIDTH, PERIOD, or NOCHANGE object. It may contain the BEHAVIOR object...
(Section 3.4.16), since the behavior in case of timing constraint violation cannot be described in the FUNCTION. It may also contain the following annotations:

**Table 3-24 : Violation annotation container**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESSAGE_TYPE</td>
<td>string</td>
<td>specifies the type of the message. It can be one of information, warning or error.</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>string</td>
<td>specifies the message itself.</td>
</tr>
</tbody>
</table>

Example:

```
VECTOR (01 d <&> 01 cp) {
    SETUP {
        VIOLATION {
            MESSAGE_TYPE = error;
            MESSAGE = "setup violation 01 d <&> 01 cp";
            BEHAVIOR {q = 'bx;}
        }
    }
}
```

3.6.1.3 INFORMATION container

An INFORMATION container may be inside a LIBRARY, SUBLIBRARY, CELL, or WIRE object. It may also be in PRIMITIVE objects inside a LIBRARY or SUBLIBRARY, but not in the locally defined primitives inside cells or functions. It may contain the following annotations:

**Table 3-25 : Information annotation container**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>VERSION</td>
<td>string</td>
<td>version of the object containing this INFORMATION block</td>
<td>“v1r3_2”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“1.3.2”</td>
</tr>
<tr>
<td>TITLE</td>
<td>string</td>
<td>title or comment related this object</td>
<td>“0.2u StdCell Library”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“2-input NAND, 4x drive”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“3-layer metal, best case, wireload model”</td>
</tr>
<tr>
<td>PRODUCT</td>
<td>string</td>
<td>product related to the object</td>
<td>“vsc1083”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“vsm10rs111”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“0.2u technology family”</td>
</tr>
<tr>
<td>AUTHOR</td>
<td>string</td>
<td>originator or modifier of the object</td>
<td>“<a href="mailto:user@system.com">user@system.com</a>”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“Imn N. Gineer”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“An ASIC Vendor, Inc.”</td>
</tr>
<tr>
<td>DATETIME</td>
<td>string</td>
<td>date/time stamp related to the object</td>
<td>“Wed Aug 19 08:13:01 MST 1998”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“July 4, 1998”</td>
</tr>
</tbody>
</table>
Example:

```plaintext
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";
    }
}
```

3.6.2 Keywords for referencing objects used as annotation

The following object references may be used as annotations:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELL</td>
<td>string</td>
<td>reference to a declared CELL object</td>
</tr>
<tr>
<td>PRIMITIVE</td>
<td>string</td>
<td>reference to a declared PRIMITIVE object</td>
</tr>
<tr>
<td>PIN</td>
<td>string</td>
<td>reference to a declared PIN object</td>
</tr>
<tr>
<td>CLASS</td>
<td>string</td>
<td>reference to a declared CLASS object</td>
</tr>
</tbody>
</table>

The syntax is as follows:

```plaintext
object_keyword = string ;
```

3.6.3 Annotations for a PIN object

A PIN object may contain the following annotations:

3.6.3.1 VIEW annotation

```plaintext
VIEW = string ;
```

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

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>functional</td>
<td>pin appears in functional netlist</td>
</tr>
<tr>
<td>physical</td>
<td>pin appears in physical netlist</td>
</tr>
<tr>
<td>both (default)</td>
<td>pin appears in both functional and physical netlist</td>
</tr>
<tr>
<td>none</td>
<td>pin does not appear in netlist</td>
</tr>
</tbody>
</table>

3.6.3.2 PINTYPE annotation

```plaintext
PINTYPE = string ;
```
annotes the type of the pin, which can take the following values:

Table 3-28: PINTYPE annotations for a PIN object

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>digital (default)</td>
<td>digital signal pin</td>
</tr>
<tr>
<td>analog</td>
<td>analog signal pin</td>
</tr>
<tr>
<td>supply</td>
<td>power supply or ground pin</td>
</tr>
</tbody>
</table>

3.6.3.3 SIGNALTYPE annotation

SIGNALTYPE = string;

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

Table 3-29: SIGNALTYPE annotations for a PIN object

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data (default)</td>
<td>general data signal</td>
</tr>
<tr>
<td>scan_data</td>
<td>scan data signal</td>
</tr>
<tr>
<td>control</td>
<td>general control signal</td>
</tr>
<tr>
<td>select</td>
<td>select signal of a multiplexor</td>
</tr>
<tr>
<td>enable</td>
<td>enable signal</td>
</tr>
<tr>
<td>out_enable</td>
<td>output enable signal</td>
</tr>
<tr>
<td>scan_enable</td>
<td>scan enable signal</td>
</tr>
<tr>
<td>scan_out_enable</td>
<td>scan output enable signal</td>
</tr>
<tr>
<td>clear</td>
<td>clear signal of a flipflop or latch</td>
</tr>
<tr>
<td>set</td>
<td>set signal of a flipflop or latch</td>
</tr>
<tr>
<td>write</td>
<td>write signal for memory, register file</td>
</tr>
<tr>
<td>read</td>
<td>read signal for memory, register file</td>
</tr>
<tr>
<td>clock</td>
<td>clock signal of a flipflop or latch</td>
</tr>
<tr>
<td>scan_clock</td>
<td>scan clock signal of a flipflop or latch</td>
</tr>
<tr>
<td>master_clock</td>
<td>master clock signal of a flipflop or latch</td>
</tr>
<tr>
<td>slave_clock</td>
<td>slave clock signal of a flipflop or latch</td>
</tr>
<tr>
<td>address_clock</td>
<td>address signal of a memory</td>
</tr>
</tbody>
</table>

3.6.3.4 DRIVETYPE annotation

DRIVETYPE = string ;
annotates the drive type for the pin, which can take the following values:

**Table 3-30 : DRIVETYPE annotations for a PIN object**

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmos (default)</td>
<td>standard cmos signal</td>
</tr>
<tr>
<td>nmos</td>
<td>nmos or pseudo nmos signal</td>
</tr>
<tr>
<td>pmos</td>
<td>pmos or pseudo pmos signal</td>
</tr>
<tr>
<td>nmos_pass</td>
<td>nmos passgate signal</td>
</tr>
<tr>
<td>pmos_pass</td>
<td>pmos passgate signal</td>
</tr>
<tr>
<td>cmos_pass</td>
<td>cmos passgate signal, i.e. full transmission gate</td>
</tr>
<tr>
<td>ttl</td>
<td>TTL signal</td>
</tr>
<tr>
<td>open_drain</td>
<td>open drain signal</td>
</tr>
<tr>
<td>open_source</td>
<td>open source signal</td>
</tr>
</tbody>
</table>

### 3.6.3.5 DIRECTION annotation

DIRECTION = string;

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

**Table 3-31 : DIRECTION annotations for a PIN object**

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>input pin</td>
</tr>
<tr>
<td>output</td>
<td>output pin</td>
</tr>
<tr>
<td>both</td>
<td>bidirectional pin</td>
</tr>
<tr>
<td>none</td>
<td>no direction can be assigned to the pin</td>
</tr>
</tbody>
</table>

### 3.6.3.6 SCOPE annotation

SCOPE = string;

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

**Table 3-32 : SCOPE annotations for a PIN object**

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>behavior</td>
<td>Pin is used for modeling functional behavior. Events on the pin are monitored for vector expressions in BEHAVIOR statement</td>
</tr>
<tr>
<td>measure</td>
<td>Measurements related to the pin can be described, e.g. timing or power characterization. Events on the pin are monitored for vector expressions in VECTOR statements</td>
</tr>
<tr>
<td>both (default)</td>
<td>Pin is used for functional behavior as well as for characterization measurements</td>
</tr>
<tr>
<td>none</td>
<td>no model, only pin exists</td>
</tr>
</tbody>
</table>
3.6.3.7  ACTION annotation

ACTION = string ;

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

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronous</td>
<td>signal acts in synchronous way</td>
</tr>
<tr>
<td>asynchronous</td>
<td>signal acts in asynchronous way</td>
</tr>
</tbody>
</table>

3.6.3.8  POLARITY annotation

POLARITY = string ;

annotates the polarity of the pin signal.

The polarity of an input pin (i.e. DIRECTION = input;) can take the following values:

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>high</td>
<td>signal active high or to be driven high</td>
</tr>
<tr>
<td>low</td>
<td>signal active low or to be driven low</td>
</tr>
<tr>
<td>rising_edge</td>
<td>signal sensitive to rising edge</td>
</tr>
<tr>
<td>falling_edge</td>
<td>signal sensitive to falling edge</td>
</tr>
<tr>
<td>double_edge</td>
<td>signal sensitive to any edge</td>
</tr>
</tbody>
</table>

The polarity of an output pin (i.e. DIRECTION = output;) can take the following values:

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inverted</td>
<td>polarity change between input and output</td>
</tr>
<tr>
<td>non_inverted</td>
<td>no polarity change between input and output</td>
</tr>
<tr>
<td>both</td>
<td>polarity may change or not (e.g. XOR) (default)</td>
</tr>
<tr>
<td>none</td>
<td>polarity has no meaning(e.g. analog signal)</td>
</tr>
</tbody>
</table>

3.6.3.9  ENABLE_PIN annotation

ENABLE_PIN = string ;

references an output enable pin (i.e. a pin with SIGNALTYPE = out_enable; ).

3.6.3.10  PULL annotation

PULL = string ;
which can take the following values:

3.6.3.11 ORIENTATION annotation

\[ \text{ORIENTATION} = \text{string} ; \]

which can take the following pin orientation values:

3.6.3.12 CONNECT_CLASS annotation

\[ \text{CONNECT_CLASS} = \text{identifier} ; \]

annotates a declared class object for connectivity determination.

3.6.3.13 DATATYPE annotation

\[ \text{DATATYPE} = \text{string} ; \]

is only relevant for bus pins, which can take the following values:

3.6.3.14 SCAN_POSITION annotation

\[ \text{SCAN\_POSITION} = \text{unsigned} ; \]

annotates position in scan chain.

3.6.3.15 STUCK annotation

\[ \text{STUCK} = \text{string} ; \]
which can be:

**Table 3-39 : STUCK annotations for a PIN object**

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stuck_at_0</td>
<td>pin can have stuck-at-0 fault</td>
</tr>
<tr>
<td>stuck_at_1</td>
<td>pin can have stuck-at-1 fault</td>
</tr>
<tr>
<td>both (default)</td>
<td>pin can have both stuck-at-0 and stuck-at-1 faults</td>
</tr>
<tr>
<td>none</td>
<td>pin can not have stuck-at faults</td>
</tr>
</tbody>
</table>

### 3.6.16 OFF_STATE annotation

OFF_STATE = string ;

which can be:

**Table 3-40 : OFF_STATE annotations for a PIN object**

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inverted</td>
<td>pin is inverted when in off state</td>
</tr>
<tr>
<td>non_inverted</td>
<td>pin is not inverted when in off state</td>
</tr>
</tbody>
</table>

### 3.6.17 INITIAL_VALUE annotation

INITIAL_VALUE = logic_constant ;

which must be compatible with the buswidth and DATATYPE of the signal.

INITIAL_VALUE is used for a downstream behavioral simulation model, as far as the simulator (e.g. VITAL-compliant simulator) supports the notion of initial value.

### 3.6.4 Annotations for a VECTOR object

A VECTOR object may contain the following annotations:

#### 3.6.4.1 LABEL annotation

LABEL = string ;

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

#### 3.6.4.2 EXISTENCE_CONDITION

EXISTENCE_CONDITION = boolean_expression ;

For false-path analysis tools, the existence condition shall be used to eliminate the vector from further analysis if and only if the existence condition evaluates to “false”. For applications other than false-path analysis, the existence condition shall be treated as if the boolean expression was a cofactor to the vector itself. Default existence condition is “true”.

---

78  Advanced Library Format (ALF) Reference Manual  Version 1.1
Example:

```plaintext
VECTOR (01 a -> 01 z & (c | !d) ) {
    EXISTENCE_CONDITION = !scan_select;
    DELAY { FROM { PIN=a; } TO { PIN=z; } /* data */ }
}
VECTOR (01 a -> 01 z & (!c | d) ) {
    EXISTENCE_CONDITION = !scan_select;
    DELAY { FROM { PIN=a; } TO { PIN=z; } /* data */ }
}
```

Each vector contains state-dependent delay for the same timing arc. If "!scan_select" evaluates "true", both vectors are eliminated from timing analysis.

### 3.6.4.3 EXISTENCE_CLASS

```plaintext
EXISTENCE_CLASS = string ;
```

Reference to the same existence class by multiple vectors has the following effects:

- A common mode of operation is established between those vectors, which can be used for selective analysis, for instance mode-dependent timing analysis. Name of the mode is the name of the class.

- A common existence condition is inherited from that existence class, if there is one.

Example:

```plaintext
CLASS non_scan_mode {
    EXISTENCE_CONDITION = !scan_select;
}
VECTOR (01 a -> 01 z & (c | !d) ) {
    EXISTENCE_CLASS = non_scan_mode;
    DELAY { FROM { PIN=a; } TO { PIN=z; } /* data */ }
}
VECTOR (01 a -> 01 z & (!c | d) ) {
    EXISTENCE_CLASS = non_scan_mode;
    DELAY { FROM { PIN=a; } TO { PIN=z; } /* data */ }
}
```

Each vector contains state-dependent delay for the same timing arc. If the mode "non_scan_mode" is turned off or if "!scan_select" evaluates "true", both vectors are eliminated from timing analysis.

### 3.6.4.4 CHARACTERIZATION_CONDITION

```plaintext
CHARACTERIZATION_CONDITION = boolean_expression ;
```

For characterization tools, the characterization condition shall be treated as if the boolean expression was a cofactor to the vector itself. For all other applications, the characterization condition shall be disregarded. Default characterization condition is “true”.

Example:

```
VECTOR (01 a -> 01 z & (c | !d) ) {
    CHARACTERIZATION_CONDITION = c & !d;
    DELAY { FROM { PIN=a; } TO { PIN=z; } /* data */ }
}
```

The delay value for the timing arc applies for any of the following conditions 
(c & !d) or (c & d) or (!c & !d), since they all satisfy (c | !d).
However, the only condition chosen for delay characterization is (c & !d).

### 3.6.4.5 CHARACTERIZATION_VECTOR

```
CHARACTERIZATION_VECTOR = ( vector_expression ) ;
```

The characterization vector is provided for the case that the vector expression cannot be constructed using the vector and a boolean cofactor. The use of the characterization vector is restricted to characterization tools in the same way as the use of the characterization condition. Either a characterization condition or a characterization vector may be provided, but not both. If none is provided, the vector itself will be used by the characterization tool.

Example:

```
VECTOR (01 A -> 01 Z) {
    CHARACTERIZATION_VECTOR = ( (01 A & 10 inv_A) -> (01 Z & 10 inv_Z) ) ;
}
```

Analysis tools see the signals "A" and "Z". The signals "inv_A" and "inv_Z" are visible to the characterization tool only.

### 3.6.4.6 CHARACTERIZATION_CLASS

```
CHARACTERIZATION_CLASS = string ;
```

Reference to the same characterization class by multiple vectors has the following effects:

- A commonality is established between those vectors, which can be used for selective characterization in a way defined by the library characterizer, for instance to share the characterization task between different teams or jobs or tools ...
- A common characterization condition or characterization vector is inherited from that characterization class, if there is one.

### 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:

### Table 3-41: CELLTYPE annotations for a CELL object

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>cell is a buffer</td>
</tr>
<tr>
<td>combinational</td>
<td>cell is a combinational logic element</td>
</tr>
<tr>
<td>multiplexor</td>
<td>cell is a multiplexor</td>
</tr>
<tr>
<td>flipflop</td>
<td>cell is a flip-flop</td>
</tr>
<tr>
<td>latch</td>
<td>cell is a latch</td>
</tr>
<tr>
<td>memory</td>
<td>cell is a memory element</td>
</tr>
<tr>
<td>block</td>
<td>cell is a block</td>
</tr>
<tr>
<td>core</td>
<td>cell is a core element</td>
</tr>
<tr>
<td>pad</td>
<td>cell is a pad</td>
</tr>
<tr>
<td>special</td>
<td>cell is a special element</td>
</tr>
</tbody>
</table>

### 3.6.5.2 BUFFERTYPE annotation

BUFFERTYPE = string ;

which can take the following values:

### Table 3-42: BUFFERTYPE annotations for a CELL object

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>cell is an input buffer</td>
</tr>
<tr>
<td>output</td>
<td>cell is an output buffer</td>
</tr>
<tr>
<td>inout</td>
<td>cell is an inout (bidirectional) buffer</td>
</tr>
<tr>
<td>internal</td>
<td>cell is an internal buffer</td>
</tr>
</tbody>
</table>

### 3.6.5.3 DRIVERTYPE annotation

DRIVERTYPE = string ;

which can take the following values:

### Table 3-43: DRIVERTYPE annotations for a CELL object

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>predriver</td>
<td>cell is a predriver</td>
</tr>
<tr>
<td>slotdriver</td>
<td>cell is a slotdriver</td>
</tr>
<tr>
<td>both</td>
<td>cell is both a predriver and a slot driver</td>
</tr>
</tbody>
</table>

### 3.6.5.4 PARALLEL_DRIVE annotation

PARALLEL_DRIVE = unsigned;

which specifies the number of parallel drivers.
3.6.5.5 **SCAN_TYPE annotation**

SCAN_TYPE = string ;

which can take the following values:

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>muxscan</td>
<td>There is a multiplexor for normal data and scan data</td>
</tr>
<tr>
<td>clocked</td>
<td>There is a special scan clock</td>
</tr>
<tr>
<td>lssd</td>
<td>combination between flipflop and latch with special clocking (level sensitive scan design)</td>
</tr>
<tr>
<td>control_0</td>
<td>combinational scan cell, controlling pin must be 0 in scan mode</td>
</tr>
<tr>
<td>control_1</td>
<td>combinational scan cell, controlling pin must be 1 in scan mode</td>
</tr>
</tbody>
</table>

3.6.5.6 **SCAN_USAGE annotation**

SCAN_USAGE = string ;

which can take the following values:

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>primary input in a chain of cells</td>
</tr>
<tr>
<td>output</td>
<td>primary output in a chain of cells</td>
</tr>
<tr>
<td>hold</td>
<td>holds intermediate value in the scan chain</td>
</tr>
</tbody>
</table>

3.6.5.7 **NON_SCAN_CELL annotation**

NON_SCAN_CELL [ identifier ] = cell_identifier { pin_assignments }

NON_SCAN_CELL [ identifier ] = primitive_identifier { pin_assignments }

This annotation shall define non-scan cell equivalency to the scan cell in which this annotation is contained. A cell instantiation form (Section 3.4.3) is used to reference the library cell that defines the non-scan functionality of the current cell. If no such cell is available or defined, or if an explicit reference to such a cell is not desired, then a primitive instantiation form (Section 3.4.3) may reference a primitive, either ALF- or user-defined, for such use. In either case, constant values may appear on either the left-hand side or right-hand side of the pin connectivity relationships. A constant on the left-hand side defines the value the scan cell pins (appearing on the right-hand side) must have in order for the primitive to perform with the same functionality as does the instantiated reference. Multiple non-scan cells may be referenced within the same scope by giving a name to each one.
Example:

```plaintext
CELL my_flipflop {
    PIN q { DIRECTION=output; }
    PIN d { DIRECTION=input; }
    PIN clk { DIRECTION=input; }
    PIN clear { DIRECTION=input; polarity=low; }
    // followed by function, vectors etc.
}

CELL my_other_flipflop {
    // declare the pins
    // followed by function, vectors etc.
}

CELL my_scan_flipflop {
    PIN data_out { DIRECTION=output; }
    PIN data_in { DIRECTION=input; }
    PIN clock { DIRECTION=input; }
    PIN scan_in { DIRECTION=input; }
    PIN scan_sel { DIRECTION=input; }
    NON_SCAN_CELL first_choice = my_flipflop {
        q = data_out;
        d = data_in;
        clk = clock;
        clear = 'b1;
        'b0 = scan_in; // scan cell has no clear
        'b0 = scan_sel; // non-scan cell has no scan_in
    }
    NON_SCAN_CELL second_choice = my_other_flipflop {
        // put in the pin assignments
    }
    // followed by function, vectors etc.
}
```

### 3.6.5.8 SWAP_CLASS annotation

**SWAP_CLASS** = string ;

The value is the name of a declared CLASS. Multi-value annotation may be used. Cells referring to the same CLASS may be swapped for certain applications.

Cell-swapping is only allowed under the following conditions:

- The RESTRICT_CLASS annotation (see next) authorizes usage of the cell
- The cells to be swapped are compatible from an application standpoint (functional compatibility for synthesis, physical compatibility for layout)

### 3.6.5.9 RESTRICT_CLASS annotation

**RESTRICT_CLASS** = string ;
The value is the name of a declared CLASS. Multi-value annotation may be used. Cells referring to a particular class may be used in design tools identified by the value.

Table 3-46: Predefined values for RESTRICT_CLASS

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>synthesis</td>
<td>use restricted to logic synthesis</td>
</tr>
<tr>
<td>scan</td>
<td>use restricted to scan synthesis</td>
</tr>
<tr>
<td>datapath</td>
<td>use restricted to datapath synthesis</td>
</tr>
<tr>
<td>clock</td>
<td>use restricted to clock tree synthesis</td>
</tr>
</tbody>
</table>

User-defined values are also possible. If a cell has no or only unknown values for RESTRICT_CLASS, the application tool may not modify any instantiation of that cell in the design. However, the cell must still be considered for analysis.

Example:

```plaintext
CLASS foo;
CLASS bar;
CELL c1 {
    SWAP_CLASS = foo;
    RESTRICT_CLASS = synthesis;
}
CELL c2 {
    SWAP_CLASS = foo;
    RESTRICT_CLASS { synthesis scan bar }
}
```

Supposed that the cells c1 and c2 are compatible from an application standpoint, the cells c1 and c2 can be used for synthesis, where they may be swapped which each other. The cell c2 can be also used for scan insertion and for the user-defined application “bar”.

3.6.6 Attributes

Identifiers inside ATTRIBUTE can be used to add information that does not fit into the annotation scheme. The syntax for specifying ATTRIBUTE is

```plaintext
ATTRIBUTE { attribute_items }
```

where attribute_items is a list of predefined or user-defined attributes.

3.6.6.1 ATTRIBUTE within a PIN object

The following attributes can be used within a PIN object:

Table 3-47: Attributes within a PIN object

<table>
<thead>
<tr>
<th>Attribute item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHMITT</td>
<td>Schmitt trigger signal</td>
</tr>
<tr>
<td>TRISTATE</td>
<td>tristate signal</td>
</tr>
</tbody>
</table>
The following attributes within a PIN object can also have **POLARITY** annotation:

<table>
<thead>
<tr>
<th>Attribute item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XTAL</td>
<td>crystal/oscillator signal</td>
</tr>
<tr>
<td>PAD</td>
<td>pad going off-chip</td>
</tr>
</tbody>
</table>

Example:

```plaintext
PIN rw {
   ATTRIBUTE {
      WRITE { POLARITY = high; }
      READ  { POLARITY = low ; }
   }
}
```

### 3.6.6.2 ATTRIBUTE within a CELL object

The following attributes can be used within a CELL object:

<table>
<thead>
<tr>
<th>Attribute item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>CAM</td>
<td>Content Addressable Memory</td>
</tr>
<tr>
<td>static</td>
<td>static device (e.g. static CMOS, static RAM)</td>
</tr>
<tr>
<td>dynamic</td>
<td>dynamic device (e.g. dynamic CMOS, dynamic RAM)</td>
</tr>
<tr>
<td>asynchronous</td>
<td>asynchronous operation</td>
</tr>
<tr>
<td>synchronous</td>
<td>synchronous operation</td>
</tr>
</tbody>
</table>

### 3.6.6.3 ATTRIBUTE within a LIBRARY object

There are no attributes with predefined meaning specified yet.

### 3.6.7 Keywords for arithmetic models

The following keywords shall identify arithmetic model objects inside a **LIBRARY**, a **SUBLIBRARY**, 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. The keywords for arithmetic models can also be used
• for simple annotations
• as annotation container

The annotations or annotation containers identified by keywords for arithmetic models can be
interpreted as reduced arithmetic models, since they don’t contain a header or a body, whereas
full arithmetic models always contain a header and a body (table or equation).

All the keywords for arithmetic models are considered context-sensitive keywords. In the
following sections, these arithmetic models are described along with the type of the value they
can have. If the quantity associated with the arithmetic model is a measurement, default units
and base units are also noted. The default units are applied when the unit is not specified.

3.6.7.1 Models for interpolateable tables and equations
The following tables list the keywords that identify arithmetic models that can be used as
interpolateable table indices and/or as equations.

Table 3-50: Timing measurements

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value type</th>
<th>Base Units</th>
<th>Default Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELAY</td>
<td>number</td>
<td>Second</td>
<td>n (nano)</td>
<td>time between two threshold crossings within two consecutive events on two pins</td>
</tr>
<tr>
<td>RETAIN</td>
<td>number</td>
<td>Second</td>
<td>n (nano)</td>
<td>time during which an output pin will retain its value after an event on the related input pin. RETAIN appears always in conjunction with DELAY for the same two pins.</td>
</tr>
<tr>
<td>SLEWRATE</td>
<td>non-negative number</td>
<td>Second</td>
<td>n (nano)</td>
<td>time between two threshold crossings within one event on one pin</td>
</tr>
</tbody>
</table>

Table 3-51: Timing constraints

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value type</th>
<th>Base Units</th>
<th>Default Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOLD</td>
<td>number</td>
<td>Second</td>
<td>n (nano)</td>
<td>minimum time limit for hold between two threshold crossings within two consecutive events on two pins</td>
</tr>
<tr>
<td>NOCHANGE</td>
<td>optional(^a) non-negative number</td>
<td>Second</td>
<td>n (nano)</td>
<td>minimum time limit between two threshold crossings within two arbitrary consecutive events on one pin, in conjunction with SETUP and HOLD</td>
</tr>
<tr>
<td>PERIOD</td>
<td>non-negative number</td>
<td>Second</td>
<td>n (nano)</td>
<td>minimum time limit between two identical events within a sequence of periodical events on one pin</td>
</tr>
</tbody>
</table>
Table 3-51 : Timing constraints

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value type</th>
<th>Base Units</th>
<th>Default Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PULSEWIDTH</td>
<td>number</td>
<td>Second</td>
<td>n (nano)</td>
<td>minimum time limit between two threshold crossings within two consecutive and complementary events on one pin</td>
</tr>
<tr>
<td>RECOVERY</td>
<td>number</td>
<td>Second</td>
<td>n (nano)</td>
<td>minimum time limit for recovery between two threshold crossings within two consecutive events on two pins</td>
</tr>
<tr>
<td>REMOVAL</td>
<td>number</td>
<td>Second</td>
<td>n (nano)</td>
<td>minimum time limit for removal between two threshold crossings within two consecutive events on two pins</td>
</tr>
<tr>
<td>SETUP</td>
<td>number</td>
<td>Second</td>
<td>n (nano)</td>
<td>minimum time limit for setup between two threshold crossings within two consecutive events on two pins</td>
</tr>
<tr>
<td>SKEW</td>
<td>number</td>
<td>Second</td>
<td>n (nano)</td>
<td>absolute value is maximum time limit between two threshold crossings within two consecutive events on two pins, the sign indicates positive or negative direction</td>
</tr>
</tbody>
</table>

a. The associated SETUP and HOLD measurements provide data. NOCHANGE itself need not provide data

Table 3-52 : Analog measurements

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value type</th>
<th>Base Units</th>
<th>Default Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT</td>
<td>number</td>
<td>Ampere</td>
<td>m (milli)</td>
<td>electrical current drawn by the cell. A pin may be specified as annotation.¹</td>
</tr>
<tr>
<td>ENERGY</td>
<td>number</td>
<td>Joule</td>
<td>p (pico)</td>
<td>electrical energy drawn by the cell, including charge and discharge energy, if applicable.</td>
</tr>
<tr>
<td>FREQUENCY</td>
<td>non-negative number</td>
<td>Hz</td>
<td>meg (mega)</td>
<td>frequency</td>
</tr>
<tr>
<td>JITTER</td>
<td>non-negative number</td>
<td>Second</td>
<td>n (nano)</td>
<td>uncertainty of arrival time</td>
</tr>
<tr>
<td>POWER</td>
<td>number</td>
<td>Watt</td>
<td>u (micro)</td>
<td>electrical power drawn by the cell, including charge and discharge power, if applicable.</td>
</tr>
<tr>
<td>TEMPERATURE</td>
<td>number</td>
<td>° Celsius</td>
<td>l (unit)</td>
<td>temperature</td>
</tr>
</tbody>
</table>

¹. The associated SETUP and HOLD measurements provide data. NOCHANGE itself need not provide data.
Table 3-52 : Analog measurements

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value type</th>
<th>Base Units</th>
<th>Default Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>number</td>
<td>Second</td>
<td>1 (unit)</td>
<td>time point for waveform modeling, time span for average, RMS, peak modeling</td>
</tr>
<tr>
<td>VOLTAGE</td>
<td>number</td>
<td>Volt</td>
<td>1 (unit)</td>
<td>voltage</td>
</tr>
<tr>
<td>FLUX</td>
<td>non-negative number</td>
<td>Coulomb per Square Meter</td>
<td>1 (unit)</td>
<td>amount of hot electrons in units of electrical charge per gate oxide area</td>
</tr>
<tr>
<td>FLUENCE</td>
<td>non-negative number</td>
<td>Second times Coulomb per Square Meter</td>
<td>1 (unit)</td>
<td>integral of FLUX over time</td>
</tr>
</tbody>
</table>

a. If the annotated PIN has PINTYPE=supply, the CURRENT measurement qualifies for power analysis. In this case, the current includes charge/discharge current, if applicable.

Table 3-53 : Electrical components

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value type</th>
<th>Base Units</th>
<th>Default Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAPACITANCE</td>
<td>non-negative number</td>
<td>Farad</td>
<td>p (pico)</td>
<td>pin, wire, load, or net capacitance</td>
</tr>
<tr>
<td>INDUCTANCE</td>
<td>non-negative number</td>
<td>Henry</td>
<td>n (nano)</td>
<td>pin, wire, load, or net inductance</td>
</tr>
<tr>
<td>RESISTANCE</td>
<td>non-negative number</td>
<td>Ohm</td>
<td>K (kilo)</td>
<td>pin, wire, load, or net resistance</td>
</tr>
</tbody>
</table>

Table 3-54 : Layout data

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value type</th>
<th>Base Units</th>
<th>Default Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREA</td>
<td>non-negative number</td>
<td>Square Meter</td>
<td>p (pico)</td>
<td>area in square microns (pico = micro²)</td>
</tr>
<tr>
<td>DISTANCE</td>
<td>number</td>
<td>Meter</td>
<td>u (micro)</td>
<td>distance between two points in microns</td>
</tr>
<tr>
<td>HEIGHT</td>
<td>non-negative number</td>
<td>Meter</td>
<td>u (micro)</td>
<td>x-or y- dimension of a placeable object (e.g. cell, block) x-, y-, or z- dimension of a routeable object (e.g. wire) measured in orthogonal direction to the route</td>
</tr>
</tbody>
</table>
### Table 3-54: Layout data

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value type</th>
<th>Base Units</th>
<th>Default Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENGTH</td>
<td>non-negative number</td>
<td>Meter</td>
<td>u (micro)</td>
<td>x-, y-, or z- dimension of a routeable object (e.g. wire) measured in parallel direction to the route</td>
</tr>
<tr>
<td>WIDTH</td>
<td>non-negative number</td>
<td>Meter</td>
<td>u (micro)</td>
<td>x- or y- dimension of a placeable object (e.g. cell, block) x-, y-, or z- dimension of a routeable object (e.g. wire) measured in orthogonal direction to the route</td>
</tr>
</tbody>
</table>

### Table 3-55: Abstract measurements

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value type</th>
<th>Base Units</th>
<th>Default Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRIVE_STRENGTH</td>
<td>non-negative number</td>
<td>None</td>
<td>1 (unit)</td>
<td>drive strength of a pin, abstract measure for (drive resistance)⁻¹</td>
</tr>
<tr>
<td>SIZE</td>
<td>non-negative number</td>
<td>None</td>
<td>1 (unit)</td>
<td>abstract cost function for actual or estimated area of a cell or a block</td>
</tr>
</tbody>
</table>

### Table 3-56: Normalized measurements

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value type</th>
<th>Base Units</th>
<th>Default Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>THRESHOLD</td>
<td>non-negative number</td>
<td>Normalized signal voltage swing</td>
<td>1 (unit)</td>
<td>Fraction of signal voltage swing, specifying a reference point for timing measurement data. The threshold is the voltage for which the timing measurement is taken.</td>
</tr>
<tr>
<td>NOISE_MARGIN</td>
<td>non-negative number</td>
<td>Normalized signal voltage swing</td>
<td>1 (unit)</td>
<td>Fraction of signal voltage swing, specifying the noise margin. The noise margin is a deviation of the actual voltage from the expected voltage for a specified signal level</td>
</tr>
</tbody>
</table>
Actual values for discrete measurements are always integer numbers, however, estimated values may be non-integer numbers (e.g. average fanout of a net =2.4 ).

3.6.7.2 Models for non-interpolateable tables

The following keywords identify arithmetic models that can only be used as non-interpolateable tables. The values in the table may not be used in equations.

Table 3-57: Discrete measurements

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value type</th>
<th>Base Units</th>
<th>Default Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWITCHING_BITS</td>
<td>non-negative</td>
<td>None</td>
<td>1</td>
<td>number of switching bits on a bus</td>
</tr>
<tr>
<td></td>
<td>number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FANOUT</td>
<td>non-negative</td>
<td>None</td>
<td>1</td>
<td>number of receivers connected to a net</td>
</tr>
<tr>
<td></td>
<td>number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FANIN</td>
<td>non-negative</td>
<td>None</td>
<td>1</td>
<td>number of drivers connected to a net</td>
</tr>
<tr>
<td></td>
<td>number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONNECTIONS</td>
<td>non-negative</td>
<td>None</td>
<td>1</td>
<td>number of pins connected to a net, where</td>
</tr>
<tr>
<td></td>
<td>number</td>
<td></td>
<td></td>
<td>CONNECTIONS = FANIN+FANOUT</td>
</tr>
</tbody>
</table>

The connectivity function specifies the allowed and disallowed connections amongst drivers or receivers in 1-dimensional tables, or between drivers and receivers in 2-dimensional tables. A CONNECTIVITY object requires a CONNECT_RULE annotation (3.6.7.4). The boolean literals in the table have the following meaning:

Table 3-58: Connectivity data

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Value type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONNECTIVITY</td>
<td>boolean literal</td>
<td>connectivity function</td>
</tr>
<tr>
<td>DRIVER</td>
<td>string</td>
<td>argument of connectivity function</td>
</tr>
<tr>
<td>RECEIVER</td>
<td>string</td>
<td>argument of connectivity function</td>
</tr>
</tbody>
</table>

The arguments of the connectivity functions are tables of strings, which refer to user-definable classes. Pins that are subject to a particular CONNECT_RULE refer to the relevant class via a CONNECT_CLASS annotation (see section 3.6.3.12).
Example:

```plaintext
CLASS power;
CLASS ground;
CONNECTIVITY {
    CONNECT_RULE = must_short;
    HEADER {
        RECEIVER r1 { TABLE { power ground } }
        RECEIVER r2 { TABLE { power ground } }
    }
    TABLE { 1 0 0 1 }
}
```

All pins of the `power` and `ground` class must be connected amongst themselves, but `power` and `ground` class must not be shorted together.

### 3.6.7.3 Models for non-interpolateable tables and equations

The following keywords identify arithmetic models that may be used directly as non-interpolateable tables and indirectly as equations. The use of those models as equations requires that a non-interpolateable table establishes a relationship between a symbolic identifier and a number.

The following table describes process data:

**Table 3-60: Process data**

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Value type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DERATE_CASE</td>
<td>string</td>
<td>derating case coefficient</td>
</tr>
<tr>
<td>PROCESS</td>
<td>string</td>
<td>process derating coefficient</td>
</tr>
</tbody>
</table>

The following identifiers can be used as predefined processes:

```
?n?p
```

where `?` can be

- `s` strong
- `w` weak

The possible process name combinations are

**Table 3-61: Predefined process names**

<table>
<thead>
<tr>
<th>Process name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>snsp</td>
<td>strong NMOS, strong PMOS</td>
</tr>
<tr>
<td>snwp</td>
<td>strong NMOS, weak PMOS</td>
</tr>
<tr>
<td>wnsn</td>
<td>weak NMOS, strong PMOS</td>
</tr>
<tr>
<td>wnwp</td>
<td>weak NMOS, weak PMOS</td>
</tr>
</tbody>
</table>

The following identifiers can be used as predefined derating cases:

- `nom` nominal case
- `bc?` prefix for best case
wc?  prefix for worst case
where ? can be
  com      suffix for commercial case
  ind      suffix for industrial case
  mil      suffix for military case

The possible derating case combinations are defined in Table 3-62.

<table>
<thead>
<tr>
<th>Derating case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bccom</td>
<td>best case commercial</td>
</tr>
<tr>
<td>bcind</td>
<td>best case industrial</td>
</tr>
<tr>
<td>bcmil</td>
<td>best case military</td>
</tr>
<tr>
<td>wccom</td>
<td>worst case commercial</td>
</tr>
<tr>
<td>wcind</td>
<td>worst case military</td>
</tr>
<tr>
<td>wcnil</td>
<td>worst case military</td>
</tr>
</tbody>
</table>

Example:
- Direct use of PROCESS in a non-interpolateable table:

```plaintext
DELAY {
  UNIT = ns;
  HEADER {
    PROCESS { TABLE { nom snsp wnwp } }
  }
  TABLE { 0.4 0.3 0.6 }
}
```

The delay is 0.4 ns for nominal process, 0.3 ns for snsp, 0.6 ns for wnwp.

- Indirect use of PROCESS in an equation:

```plaintext
DELAY {
  UNIT = ns;
  HEADER {
    PROCESS { HEADER { nom snsp wnwp } TABLE {0.0 -0.25 0.5} }
  }
  EQUATION { (1 + PROCESS)*0.4 }
}
```

The equation uses the derating factors 0.0 for nominal, -0.25 for snsp, 0.5 for wnwp.
3.6.8 Containers for arithmetic models

The following keywords are defined for objects that may contain arithmetic models.

<table>
<thead>
<tr>
<th>Objects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>contains start point of timing measurement or timing constraint</td>
</tr>
<tr>
<td>TO</td>
<td>contains end point of measurement or timing constraint</td>
</tr>
<tr>
<td>LIMIT</td>
<td>contains arithmetic models for limit values</td>
</tr>
<tr>
<td>EARLY</td>
<td>contains arithmetic models for timing measurements relevant for early signal</td>
</tr>
<tr>
<td>LATE</td>
<td>contains arithmetic models for timing measurements relevant for late signal</td>
</tr>
</tbody>
</table>

#### 3.6.8.1 FROM and TO container

A FROM container and a TO container shall be used inside timing measurements and timing constraints. They shall contain PIN annotations for the purpose of defining the timing arc. In addition, both containers may contain arithmetic models for THRESHOLD.

Example:

```plaintext
DELAY {
    FROM {PIN = data_in; THRESHOLD { RISE = 0.4; FALL = 0.6;}}
    TO   {PIN = data_out; THRESHOLD = 0.5;}
}
```

The delay is measured from pin `data_in` to pin `data_out`. The threshold for `data_in` is 0.4 for rising signal and 0.6 for falling signal. The threshold for `data_out` is 0.5, which applies for both rising and falling signal.

If the timing measurements or timing constraints, respectively, apply for two pins, the FROM, TO containers shall each contain the PIN annotation. These annotations shall define the sense of measurement.

```plaintext
<model_keyword> {
    FROM { PIN = <pin_name> ; }  
    TO  { PIN = <pin_name> ; }  
    /* data */
}
```

Otherwise, if the timing measurements or timing constraints, respectively, apply only for one pin, the same PIN annotation may be repeated in both containers or the PIN annotation may be outside the FROM, TO container.

```plaintext
<model_keyword> { 
    PIN = <pin_name> ;  
    /* data */
}
```
If thresholds are needed for exact definition of the model data, the FROM, TO containers shall each contain an arithmetic model for THRESHOLD.

```plaintext
<model_keyword> { 
    FROM { THRESHOLD /*data*/ } 
    TO { THRESHOLD /*data*/ } 
    /* data */
}
```

An arithmetic model for THRESHOLD outside a FROM or TO container shall only have a semantic meaning, if said annotation or arithmetic model contains a PIN annotation itself and this PIN annotation matches a PIN annotation in a FROM or TO container.

Example:

```plaintext
DELAY { 
    FROM { 
        PIN = pin1; 
        THRESHOLD /*data*/ 
    } 
    TO { 
        PIN = pin2; 
    } 
    HEADER { 
        THRESHOLD { 
            PIN = pin2; 
            TABLE { <numbers> } 
        } 
        TABLE { <numbers> } 
    } 
}
```

Note: The data of the THRESHOLD at pin1 is calculated independently of DELAY, whereas DELAY is calculated as a function of THRESHOLD at pin2.

### 3.6.8.2 LIMIT container

A LIMIT container may be used inside a library-specific object (Section 3.4.6). It shall contain arithmetic models identified by MIN and/or MAX.

Example:

```plaintext
PIN data_in { 
    LIMIT { 
        SLEWRATE { UNIT = ns; MIN = 0.05; MAX = 5.0;}}
}
```
The minimum slewrate allowed at pin `data_in` is 0.05 ns, the maximum is 5.0 ns.

```plaintext
PIN data_in {
    LIMIT {
        SLEWRATE {
            UNIT = ns;
            MAX {
                HEADER { FREQUENCY { UNIT=megahz;} }
                EQUATION { 250 / FREQUENCY }
            }
        }
    }
}
```

The maximum allowed slewrate is frequency-dependent, e.g. the value is 0.25ns for 1GHz.

### 3.6.8.3 EARLY and LATE container

The EARLY and LATE containers define the boundaries of timing measurements in one single analysis. Only applicable to DELAY and SLEWRATE. Both of them must appear in both containers.

The quadruple

```plaintext
EARLY {
    DELAY { FROM {...} TO { ...} /* data */ }
    SLEWRATE { /* data */ }
LATE {
    DELAY { FROM {...} TO { ...} /* data */ }
    SLEWRATE { /* data */ }
}
```

is used to calculate the envelope of the timing waveform at the TO point of a delay arc with respect to the timing waveform at the FROM point of a delay arc.

The EARLY DELAY is of course a smaller number (or a set of smaller numbers) than the LATE DELAY. However, the EARLY SLEWRATE is not necessarily smaller than the LATE SLEWRATE, since the SLEWRATE of the EARLY signal may be larger than the SLEWRATE of the LATE signal.

### 3.6.9 Keywords for arithmetic submodels

Arithmetic submodels are for the purpose of distinguishing different measurement conditions for the same model. The root of an arithmetic model may contain nested arithmetic submodels. The header of an arithmetic model may contain nested arithmetic models, but not arithmetic submodels.

#### 3.6.9.1 MIN/TYP/MAX

MIN, TYP, MAX provide 3 distinct sets of data

```plaintext
<model_keyword> { MIN /*data*/ TYP /*data*/ MAX /*data*/ }
```
as opposed to a single set of data

```
<model_keyword> /*data*/
```

The set of valid keywords for `<model_keyword>` is defined in section 3.6.7.1.

The MIN, TYP, MAX represent a statistical distribution of data without specifying or implying a particular cause of the distribution. If process corners or derate cases are not modeled explicitly, MIN, TYP, MAX can be used for representing the distribution of data across processes or derate cases. If process corners or delay cases are modeled explicitly, MIN, TYP, MAX can be used for representing the distribution of data within each process corner or derate case.

Note: The arithmetic model root containing MIN, TYP, MAX must not contain HEADER or TABLE or EQUATION. Instead, the MIN, TYP, MAX models may contain HEADER or TABLE or EQUATION.

```
<model_keyword> { 
  MIN { 
    HEADER{ <model_keyword> /*data*/ .. <model_keyword> /*data*/ 
    } 
    TABLE /* or equation */ { <numbers> } 
  }
  TYP { 
    HEADER{ <model_keyword> /*data*/ .. <model_keyword> /*data*/ 
    } 
    TABLE /* or equation */ { <numbers> } 
  }
  MAX { 
    HEADER{ <model_keyword> /*data*/ .. <model_keyword> /*data*/ 
    } 
    TABLE /* or equation */ { <numbers> } 
  } 
}
```

MIN, TYP, MAX can also be single numbers. In this case, they have the same syntax as annotations within the arithmetic model.

```
<model_keyword> { 
  MIN = <number> ; 
  TYP = <number> ; 
  MAX = <number> ; 
}
```

Within the scope of a LIMIT container, MIN and MAX contain the data for a lower or upper limit, respectively. There must be at least one limit, lower or upper, in each model, but not necessarily both, as shown in the example below.

```
LIMIT { 
  <model_keyword1> { MIN /*data*/ } // lower limit 
  <model_keyword2> { MAX /*data*/ } // upper limit 
  <model_keyword3> { MIN /*data*/ MAX /*data*/ } // lower and upper limit 
}
```
Note: The arithmetic model root inside LIMIT must not contain HEADER or TABLE or EQUATION. Instead, the MIN or MAX models may contain HEADER or TABLE or EQUATION.

```plaintext
LIMIT {
  <model_keyword> {
    MIN {
      HEADER( <model_keyword> /*data*/ .. )
      TABLE { <numbers> } /* or equation */
    }
    MAX {
      HEADER( <model_keyword> /*data*/ .. )
      TABLE { <numbers> } /* or equation */
    }
  }
}
```

MIN, MAX inside arithmetic model root inside LIMIT can also be single numbers.

```plaintext
LIMIT {
  <model_keyword> {
    MIN = <number> ;
    MAX = <number> ;
  }
}
```

MIN, MAX inside a model inside a HEADER define the validity limits of the data. The model inside the HEADER may contain TABLE or EQUATION. It may also contain HEADER, which represents a nested arithmetic model.

If MIN, MAX is not defined and the data is in a TABLE, the boundaries of the data in the TABLE shall be considered as validity limits.

Note: The MIN and MAX numbers qualify the data of the arithmetic model in the HEADER, they do not represent the data itself.

```plaintext
<model_keyword> {
  HEADER {
    <model_keyword> {
      MIN = <number> ; // minimum value for valid extrapolation
      MAX = <number> ; // maximum value for valid extrapolation
      TABLE { <numbers> } // data for inter- and extrapolation
    }
    TABLE { <numbers> }
  }
}
```
3.6.9.2 RISE/FALL and HIGH/LOW

RISE, FALL contain data for transient measurements. HIGH, LOW contain data for static measurements.

```
<model_keyword> { RISE /*data*/ FALL /*data*/ }  
<model_keyword> {HIGH /*data*/ LOW /*data*/ }  
```

It is generally not required that both RISE and FALL or both HIGH and LOW, respectively, appear in the arithmetic model root.

HIGH and LOW qualify states with the logic value 1 and 0, respectively. RISE and FALL qualify transitions between states with initial logic value 0 and 1, respectively and final value 1 and 0, respectively. For other states and their mapping to logic values, see Section 3.5.7. If the arithmetic model is within the scope of a vector which describes the logic values without ambiguity, the use of RISE, FALL, HIGH, LOW is not necessary.

Example:

```
VECTOR ( ?! A -> 10 B ) {  
  SLEWRATE { PIN = A; RISE = 3.1; FALL = 2.8; }  
}
```

Alternative description:

```
VECTOR ( 01 A -> 10 B ) {  
  SLEWRATE = 3.1 { PIN = A; }  
}  
VECTOR ( 10 A -> 10 B ) {  
  SLEWRATE = 2.8 { PIN = A; }  
}
```

Note: For states that cannot be mapped to logic 1 or 0, RISE, FALL, HIGH, LOW cannot be used. The use of VECTOR with unambiguous description of the relevant states is mandatory in such cases.

The arithmetic model root containing RISE, FALL or HIGH, LOW must not contain MIN, TYP, MAX, HEADER, TABLE or EQUATION. Instead, the RISE, FALL or HIGH, LOW models may contain HEADER, TABLE, EQUATION.

```
<model_keyword> {  
  <RISE or FALL or HIGH or LOW> {  
    HEADER{ <model_keyword> /*data*/ .. }  
    TABLE { <numbers> } /* or equation */  
  }  
}
```

Alternatively, the RISE, FALL or HIGH, LOW models may contain MIN, TYP, MAX, which may contain HEADER, TABLE, EQUATION themselves.

```
<model_keyword> {  
  <RISE or FALL or HIGH or LOW> {  
    MIN /*data*/  
    TYP /*data*/  
    MAX /*data*/  
  }  
}
Alternatively, the RISE, FALL or HIGH, LOW models may be single numbers.

```
<model_keyword> {  
    <RISE or FALL or HIGH or LOW> = number ;  
}
```

Semantic meaning for RISE and FALL is provided for the following measurements:
- **DELAY, RETAIN:**  
  RISE, FALL is the switching direction on the PIN specified in the TO field.  
  If the TO field does not exist (a special case for port delay), RISE, FALL is the switching  
  direction on the PIN specified in the FROM field.
- **CAPACITANCE, RESISTANCE, INDUCTANCE, CURRENT, ENERGY, POWER,  
  SLEWRATE, THRESHOLD:**  
  RISE, FALL is the switching direction on the PIN. Either the PIN is specified as annotation  
  inside the model, or the model is inside a PIN.

Semantic meaning for HIGH and LOW is provided for the following measurements:
- **CAPACITANCE, RESISTANCE, INDUCTANCE, CURRENT, ENERGY, POWER,  
  VOLTAGE, NOISE_MARGIN:**  
  HIGH, LOW is the state on the PIN. Either the PIN is specified as annotation inside the model,  
  or the model is inside a PIN.

The arithmetic model root containing RISE, FALL or HIGH, LOW may be inside a LIMIT  
container with the following rule: A model containing RISE, FALL or HIGH, LOW must not  
contain MIN or MAX. Instead, the RISE, FALL or HIGH, LOW model must contain MIN or  
MAX.

```
LIMIT {  
    <model_keyword> {  
        <RISE or FALL or HIGH or LOW> { MIN /*data*/ MAX /*data*/ }  
    }  
}
```

The arithmetic model root containing RISE, FALL may be inside EARLY, LATE containers  
with the following rules:
- If only RISE appears in one model, only RISE must appear in all models.  
- If only FALL appears in one model, only FALL must appear in all models.  
- If both RISE and FALL appear in one model, both RISE and FALL must appear in all models.

```
EARLY {  
    DELAY { RISE /*data*/ FALL /*data*/ }  
    SLEWRATE { RISE /*data*/ FALL /*data*/ }  
}  

LATE {  
    DELAY { RISE /*data*/ FALL /*data*/ }  
    SLEWRATE { RISE /*data*/ FALL /*data*/ }  
}
```

Semantic meaning for RISE and FALL is provided for the following LIMIT specifications,  
EARLY or LATE measurements:
• **DELAY, RETAIN:**
RISE, FALL is the switching direction on the PIN specified in the TO field.
Only if the TO field does not exist (a special case for port delay), RISE, FALL is the switching
direction on the PIN specified in the FROM field (since the switching direction of the
unspecified PIN in the TO field will be the same).

• **SLEWRATE:**
RISE, FALL is the switching direction on the PIN. Either the PIN is specified as annotation
inside the model, or the model is inside a PIN.
Semantic meaning for HIGH and LOW is provided for the following LIMIT specifications:
• **CURRENT, ENERGY, POWER, VOLTAGE**
HIGH, LOW is the state on the PIN. Either the PIN is specified as annotation inside the model,
or the model is inside a PIN.

### 3.6.10 Annotations for arithmetic models

Annotations and annotation containers described in this chapter are relevant for the semantic
interpretation of arithmetic models and their arguments.

Example: DELAY=f(CAPACITANCE).
DELAY is the arithmetic model, CAPACITANCE is the argument.
Arguments of arithmetic models have the form of annotation containers. They may also have
the form of arithmetic models themselves, in which case they represent nested arithmetic
models.

#### 3.6.10.1 DEFAULT annotation

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.

```
DEFAULT = number ;
```

Restrictions may apply for the allowed type of number. For instance, if the arithmetic model
allows only *non_negative_number*, then the default is restricted to *non_negative_number*.

#### 3.6.10.2 UNIT annotation

The *unit annotation* associates units with the value computed by the arithmetic model.

```
UNIT = string | non_negative_number ;
```

A unit specified by a string can take the following values (* indicates wildcard):

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f* or F*</td>
<td>equivalent to 1E-15</td>
</tr>
<tr>
<td>p* or P*</td>
<td>equivalent to 1E-12</td>
</tr>
</tbody>
</table>
Context-sensitive keywords

Arithmetic models are context-sensitive, i.e. the units for their values can be determined from the context. If `UNIT` annotation for such a context does not exist, default units are applied to the value (Section 3.6.9.2).

Example:

```
TIME { UNIT = ns; }
FREQUENCY { UNIT = gigahz; }
```

If the unit is a string, then only the first character (respectively the first 3 characters in case of `MEG`) is interpreted. The reminder of the string can be used to define base units. Metric base units are assumed, but not verified, in ALF.

There is no semantic difference between

```
unit = 1sec;
```

and

```
unit = 1volt;
```

Therefore, if the unit is specified as

```
unit = meg;
```

the interpretation is `1E+6`. However, for

```
unit = 1meg;
```

the interpretation is `1` and not `1E+6`.

Units in a non-metric system can only be specified with numbers, not with strings. For instance, if the intent is to specify inch instead of meter as base unit, the following specification will not meet the intent:

```
unit = 1inch;
```

since the interpretation is `1` and meters are assumed.

The correct way of specifying inch instead of meter is

```
unit = 25.4E-3;
```

since 1 inch is (approximately) 25.4 millimeters.
3.6.10.3 CONNECT_RULE annotation

The connect_rule annotation may be only inside a CONNECTIVITY object. It specifies connectivity requirement.

\[
\text{CONNECT\_RULE} = \text{string} \; ;
\]

which can take the following values:

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>must_short</td>
<td>short connection required</td>
</tr>
<tr>
<td>can_short</td>
<td>short connection allowed</td>
</tr>
<tr>
<td>cannot_short</td>
<td>short connection disallowed</td>
</tr>
</tbody>
</table>

3.6.10.4 PIN annotation

The use of PIN annotation in arithmetic models other than timing measurements and timing constraints is defined here.

If the PIN annotation appears inside an arithmetic model within the scope of a HEADER or a LIMIT, the physical quantity identified by the model keyword is applied to the PIN. Otherwise, if the PIN annotation appears inside an arithmetic model root that is not within the scope of a LIMIT, the physical quantity identified by the model keyword is measured at the PIN.

Example:

```c
// intrinsic capacitance of pin1
CAPACITANCE {
    PIN = pin1;
    /*data*/
}

// maximum allowed capacitance on a net connected to pin2
LIMIT {
    CAPACITANCE {
        PIN = pin2;
        MAX /*data*/
    }
}

// delay measured as function of capacitance on a net connected to pin3
DELAY {
    HEADER {
        CAPACITANCE {
            PIN = pin3;
        }
    }
    /*data*/
}
```

If the arithmetic model is within the scope of a PIN object, a PIN annotation is illegal according to the visibility rules of ALF, since a PIN cannot be visible inside another PIN, with the...
following exception: The PIN outside the arithmetic model is a bus, and the PIN annotation inside the arithmetic model refers to a bit of the bus.

Example:

```plaintext
PIN [1:2] bus_pin {
  // intrinsic capacitance of bus_pin[1]
  CAPACITANCE {
    PIN = bus_pin[1];
    /*data*/
  }
  // maximum allowed capacitance on a net connected to bus_pin[2]
  LIMIT {
    CAPACITANCE {
      PIN = bus_pin[2];
      /*data*/
    }
  }
}
```

If an arithmetic model root appears within the scope of a LIMIT inside a PIN, the physical quantity identified by the model keyword is **applied** to the PIN. Otherwise, if an arithmetic model root appears directly inside a PIN, the physical quantity identified by the model keyword is **measured** at the PIN.

Example:

```plaintext
PIN scalar_pin {
  // intrinsic capacitance of scalar_pin
  CAPACITANCE {
    /*data*/
  }
  // maximum allowed capacitance on a net connected to scalar_pin
  LIMIT {
    CAPACITANCE {
      /*data*/
    }
  }
}
```

An arithmetic model inside a bus or an arithmetic model with a PIN annotation referring to a bus shall apply to the entire bus, not to each individual scalar pin of the bus.

Example:

```plaintext
PIN [1:10] large_bus {
  CAPACITANCE = 1 { unit = pf; }
}
```

The total pin capacitance of `large_bus` is 1 pf, not 10 pf. The capacitance of individual scalar pins `large_bus[1] .. large_bus[10]` is not defined.
3.6.10.5 MEASUREMENT, TIME and FREQUENCY annotations

Arithmetic models describing analog measurements (see Table 3-52) can have a MEASUREMENT annotation. This annotation indicates the type of measurement used for the computation in arithmetic model.

\[
\text{MEASUREMENT} = \text{string} ;
\]

The string can take the following values:

Table 3-66: MEASUREMENT annotation

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transient</td>
<td>measurement is a transient value</td>
</tr>
<tr>
<td>static</td>
<td>measurement is a static value</td>
</tr>
<tr>
<td>average</td>
<td>measurement is an average value</td>
</tr>
<tr>
<td>rms</td>
<td>measurement is an root mean square value</td>
</tr>
<tr>
<td>peak</td>
<td>measurement is a peak value</td>
</tr>
</tbody>
</table>

In this context, either TIME or FREQUENCY can also be used as annotations. The semantics are defined as follows:

Table 3-67: Semantic interpretation of MEASUREMENT, TIME or FREQUENCY annotation

<table>
<thead>
<tr>
<th>MEASUREMENT annotation</th>
<th>Semantic meaning of TIME annotation</th>
<th>Semantic meaning of FREQUENCY annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>transient</td>
<td>integration of analog measurement is done during that time window</td>
<td>integration of analog measurement is repeated with that frequency</td>
</tr>
<tr>
<td>static</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>average</td>
<td>average value is measured over that time window</td>
<td>average value measurement is repeated with that frequency</td>
</tr>
<tr>
<td>rms</td>
<td>root-mean-square value is measured over that time window</td>
<td>root-mean-square measurement is repeated with that frequency</td>
</tr>
<tr>
<td>peak</td>
<td>peak value occurs during that time window</td>
<td>observation of peak value is repeated with that frequency</td>
</tr>
</tbody>
</table>

In all applicable cases, the interpretation FREQUENCY \( = 1 / \text{TIME} \) is valid.

The values for average measurements and for rms measurements scale linearly with FREQUENCY and \( 1 / \text{TIME} \), respectively. For transient measurements and for peak measurements, the TIME or FREQUENCY annotations are purely informational. The values do not scale with TIME or FREQUENCY.
Mathematical definitions:

\[
\text{transient} \quad \int_{(t = 0)}^{(t = T)} dE(t) \\
\text{static} \quad E = \text{constant} \\
\text{average} \quad \int_{(t = 0)}^{(t = T)} E(t) dt \\
\text{rms} \quad \sqrt{\int_{(t = 0)}^{(t = T)} E(t)^2 dt} \\
\text{peak} \quad \max(|E(t)|) \cdot \text{sgn}E(t) \quad 0 \leq t \leq T
\]

Examples:

- transient measurement of ENERGY
- static measurement of VOLTAGE, CURRENT, POWER
- average measurement of POWER, CURRENT
- rms measurement of POWER, CURRENT
- peak measurement of VOLTAGE, CURRENT, POWER

3.6.10.6 TIME and FREQUENCY for waveform description

Both FREQUENCY and TIME can also be used in the HEADER of arithmetic models. In particular, TIME in the HEADER describes waveforms of analog measurements. The initial and final values of the measurement, respectively, apply to the time before the first measurement and after the last measurement, respectively.

The semantics are defined as follows:

<table>
<thead>
<tr>
<th>MEASUREMENT annotation</th>
<th>Semantic meaning of TIME in HEADER</th>
<th>Use of FREQUENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>transient</td>
<td>piece-wise linear waveform of instantaneous value over time</td>
<td>allowed in HEADER or as annotation, boundary restrictions apply (see below)</td>
</tr>
<tr>
<td>static</td>
<td>N/A</td>
<td>allowed in HEADER only, no restriction</td>
</tr>
<tr>
<td>average</td>
<td>incremental average value, measured from the previous time point to the actual time point</td>
<td>allowed in HEADER or as annotation, boundary restrictions apply</td>
</tr>
<tr>
<td>rms</td>
<td>incremental rms value, measured from the previous time point to the actual time point</td>
<td>allowed in HEADER or as annotation, boundary restrictions apply</td>
</tr>
<tr>
<td>peak</td>
<td>peak value encountered between the previous time point and the actual time point</td>
<td>allowed in HEADER or as annotation, boundary restrictions apply</td>
</tr>
</tbody>
</table>

In the context of analog measurement versus TIME description, FREQUENCY may still be used either as complementary argument in the HEADER or as annotation. The interpretation
FREQUENCY = 1 / TIME is not valid. Instead, the following boundary restrictions are imposed in order to make the waveforms repeatable:

- The initial value and the final value of a transient measurement must be the same.
- The initial values of average, rms, or peak measurements, i.e. the values that apply before the first time index apply also as value after the last time index.
- The overall time window between the first and the last measurement must be bound by 1 / FREQUENCY

These restrictions make sure that there is a physical interpretation of measurements as a function of TIME and FREQUENCY.

Examples:

 transient waveform, average, rms, peak of CURRENT vs. TIME, VOLTAGE vs. TIME. Resonance effects (parasitic oscillators) may influence the measurement results in a certain FREQUENCY range.

 static measurement of POWER vs. FREQUENCY. FREQUENCY of a voltage-controlled oscillator is statically controlled by a DC voltage. Measurement could also be expressed as power versus control voltage, but the control voltage may not be observable in simulation, whereas the frequency of the oscillating output signal is observable.
The following figure illustrates transient, average, rms, and peak waveforms for a repeatable analog signal.

**Figure 3-18: Illustration of Waveforms**

\[
\begin{align*}
T_0 & \quad T_1 & \quad T_2 & \quad T_3 & \quad T_4 \\
\int_{T_0}^{T_1} E(t) \, dt & \quad \frac{1}{T_2 - T_1} \int_{T_1}^{T_2} E(t) \, dt & \quad \frac{1}{T_3 - T_2} \int_{T_2}^{T_3} E(t) \, dt & \quad \frac{1}{T_4 - T_3} \int_{T_3}^{T_4} E(t) \, dt \\
\sqrt{\frac{1}{T_1 - T_0} \int_{T_0}^{T_1} [E(t)]^2 \, dt} & \quad \sqrt{\frac{1}{T_2 - T_1} \int_{T_1}^{T_2} [E(t)]^2 \, dt} & \quad \sqrt{\frac{1}{T_3 - T_2} \int_{T_2}^{T_3} [E(t)]^2 \, dt} & \quad \sqrt{\frac{1}{T_4 - T_3} \int_{T_3}^{T_4} [E(t)]^2 \, dt} \\
\max(E(t)) & \quad T_0 \leq t \leq T_1 & \quad \max(E(t)) & \quad T_2 \leq t \leq T_3 & \quad \max(E(t)) & \quad T_3 \leq t \leq T_4 \\
FREQUENCY & = \frac{1}{T_4 - T_0}
\end{align*}
\]
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.

```alf
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 3-19 shows relationships between these objects and where they can be referenced.
The `TEMPLATE` and `GROUP` objects are referenceable only by their respective instantiation. The `TEMPLATE` definitions may contain instantiation of previously defined templates, which allows construction 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 `PRIMITIVE`s 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 constant logic value 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:

```c
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 that 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:

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

```c
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:

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

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:

\[
\text{STATETABLE \{}
\quad \text{cd sd cp d q : q ;}
\quad \quad 0 \ ? \ ?? \ ? \ ? : 0 ;
\quad \quad 1 \ 0 \ ?? \ ? \ ? : 1 ;
\quad \quad 1 \ 1 \ 1\? \ ? \ 0 : 0 ;
\quad \quad 1 \ 1 \ ?0 \ ? \ 1 : 1 ;
\quad \quad 1 \ 1 \ 01 \ ? \ ? : (d);}
\]

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

A `VECTOR` defines state, transition, or sequence of transitions of pins that are controllable and observable for characterization.

Within a `CELL`, the set of `PINS` with `SCOPE=behavior` or `SCOPE=measure` or `SCOPE=both` is the default set of variables in the event queue for vector expressions relevant for `BEHAVIOR` or `VECTOR` statements or both, respectively.

For detection of a sequence of transitions it is necessary to observe the set of variables in the event queue. For instance, if the set of pins consists of A, B, C, D, the vector expression

\[(01 \text{ A } \rightarrow 01 \text{ B})\]

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

The default set of pins applies only for vector expressions without conditions. The conditional event AND operator limits the set of variables in the event queue. In this case, only the state of the condition and the variables appearing in the vector expression are observed.

Example:

\[(01 \text{ A } \rightarrow 01 \text{ B}) \&\& (C \mid D)\]

No transition on A, B occurs between 01 A and 01 B, and (C | D) must stay true in-between 01 A and 01 B as well. However, C and D may change their values as long as (C | D) is satisfied.

### 3.8.4 Referencing multi-dimensional PINs

A group of pins of a cell can be logically considered together by declaring a `PIN` with a range. A pin can be declared with one dimension or two dimensions. For example,

```plaintext
PIN A ; // declares a scalar pin A
PIN [1:8] A1 ; // declares pin A1 with bits numbered 1 through 8
```
When a pin is declared with one dimension, the left number in the range shall specify the most significant bit number and the right number shall specify the least significant bit number. If the pin is declared with two dimensions, the second dimension shall specify the index of the first and the last rows of the two-dimension pin object.

A PIN object can be referenced in one of the four forms:

1. Individual bit - pin name shall be followed by an index of the bit
2. Contiguous group of bits - pin name shall be followed by the contiguous range of bits. The most significant and least significant bit numbers shall follow the same relationship as given in the declaration.
3. Entire PIN object - Only pin name shall be used. It shall be illegal to reference entire two-dimension pin object in any operation.
4. One row of a PIN object - For a two-dimension pin object, name of the pin shall be followed by the row index of that pin. It shall be illegal to reference either individual bit or a group of bits of a two-dimension pin object directly in an operation.

When a PIN object is referenced on the left-hand side of an assignment, the result of the right-hand side expression is copied from the least significant bit towards the most significant bit. If the right-hand side value has lesser number of bits than the referenced PIN object in an assignment, the right-hand side value shall be zero-extended to fill the remaining bits of the referenced PIN object. If the right-hand side value has more bits than the referenced PIN object in an assignment, the right-hand side value shall be truncated to the size of the referenced PIN object.

Example:

```plaintext
pin [1:8] A1;

A1[8]   = 'b0 ;
A1[1:6] = 'o75 ;
A1[1:5] = 'o75 ;
A2[18]  = 'h5 ;

// is equivalent to A1[1:6] = 'b111_101
// is equivalent to A1[1:5] = 'b11_101,
// left most bit is truncated
// entire row 18 of A2 is assigned a value.
```

The two-dimension PIN objects shall be referenced with the row index. It shall be illegal to directly reference an individual bit or a contiguous group of bits of a two-dimension PIN object. It shall be illegal to reference the entire PIN object as a two-dimension PIN object.
Example:

```plaintext
pin [1:8] A2[1:32];  
pin [1:8] B1;        
pin C;             // legal references and

assignments
A2[10] = 'h45;       // assign 'h45 to row 10 of A2
\('b0100_0101\)
B1 = A2[10];        // copies whole row A2[10] to
B1
C = B1[3];          // c = 'b0

// Illegal references and assignments
3 of A2[10]
// A2 = B1;         illegal reference to
entire A2

It shall be legal to use identifiers as index, but expressions shall not be permitted as index.

Example

pin [4:1] ADDR;

ADDR = 'd 10;
A2[ADDR] = 'h45;       // assign 'h45 to row 10 of A2
\('b0100_0101\)

// A2[ADDR+1] = 'h45; illegal
```

### 3.8.5 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:

```plaintext
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.

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

```plaintext
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 new lines.

```plaintext
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.

```plaintext
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 \texttt{FUNCTION}, both \texttt{EQUATION} and \texttt{TABLE} representation of an arithmetic model are allowed. The \texttt{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 \texttt{True}, the assigned output value is \texttt{1}. If a boolean expression evaluates \texttt{False}, the assigned output value is \texttt{0}. If the value of a boolean expression cannot be determined, the assigned output value is \texttt{X}. Assignment of values other than \texttt{1}, \texttt{0}, or \texttt{X} must be specified explicitly.

For evaluation of the boolean expression, input value \texttt{‘bH} shall be treated as \texttt{‘b1}. Input value \texttt{‘bL} shall be treated as \texttt{‘b0}. All other input values shall be treated as \texttt{‘bX}.

Examples:

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

\begin{verbatim}
BEHAVIOR {
    Z = A;
}
\end{verbatim}

is equivalent to

\begin{verbatim}
BEHAVIOR {
    Z = A ? ‘b1 : ‘b0;
}
\end{verbatim}

More explicitly, this is also equivalent to

\begin{verbatim}
BEHAVIOR {
    Z = (A==‘b1 || A==‘bH)? ‘b1 : (A==‘b0 || A==‘bL)? ‘b0 : ‘bX;
}
\end{verbatim}

In table form, this can be expressed as follows:

\begin{verbatim}
STATETABLE {
    A   : Z;
    ?   : (A);
}
\end{verbatim}
which is equivalent to

```
STATETABLE {
    A :  Z;
    0 :  0;
    1 :  1;
}
```

More explicitly, this is also equivalent to

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

### 3.9.2 Basic rules for sequential functions

A sequential function is described in equation form by a boolean assignment with a condition specified by a boolean expression or a vector expression. If the condition 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.

For evaluation of a condition, the value 'bH shall be treated as true, the value 'bL shall be treated as false. All other values shall be treated as the unknown value 'bX.

Example:

The following behavior statement

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

is equivalent to

```
BEHAVIOR {
    @ (E=='b1 || E=='bH) {Z = A;}
}
```

The following statetable statement, describing the same logic function

```
STATETABLE {
    E  A  :  Z;
    0  ?  :  (Z);
    1  ?  :  (A);
}
```
is equivalent to

```
STATETABLE {
  E  A : Z;
  0 ? : (Z);
  L ? : (Z);
  1 ? : (A);
  H ? : (A);
}
```

For edge-sensitive and higher-order event sensitive functions, transitions from or to 'bL shall be treated like transitions from or to 'b0, and transitions from or to 'bH shall be treated like transitions from or to 'b1.

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. There are also a set of ambiguous vectors, which may or may not trigger the evaluation of the function.

The set of active vectors is the set of vectors for which both observed states before and after the transition are known to be logically equivalent to the corresponding states defined in the vector expression.

The set of inactive vectors is the set of vectors for which at least one of the observed states before or after the transition is known to be not logically equivalent to the corresponding states defined in the vector expression.

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)
- ('b1'bX CP)
- ('b1'bW CP)
- ('b1'bZ CP)
- ('bH'b0 CP)
- ('bH'bL CP)
- ('bH'bX CP)
- ('bH'bW CP)
- ('bH'bZ CP)
- ('bX'b0 CP)
- ('bX'bL CP)
and the ambiguous vectors are

\[
\begin{align*}
('b0'bX CP) \\
('b0'bW CP) \\
('b0'bZ CP) \\
('bL'bX CP) \\
('bL'bW CP) \\
('bL'bZ CP) \\
('bX'b1 CP) \\
('bW'b1 CP) \\
('bX'b2 CP) \\
('bX'bH CP) \\
('bW'bH CP) \\
('bX'bW CP) \\
('bX'bZ CP) \\
('bW'bX CP) \\
('bW'bZ CP) \\
('bZ'bX CP) \\
('bZ'bW CP) \\
('bU'bX CP) \\
('bU'bW CP) \\
('bU'bZ CP)
\end{align*}
\]

For vectors using exclusively based literals, the set of active vectors is the vector itself, the set of inactive vectors is any vector with at least one different literal, the set of ambiguous vectors is empty.

Therefore ALF does not provide a default behavior for ambiguous vectors, since the behavior for each vector may be explicitly defined in vectors using based literals.

### 3.9.3 Concurrency in combinational and sequential functions

Multiple boolean assignments in combinational functions are understood to be concurrent. The order in the functional description does not matter, as each boolean assignment describes a piece of a logic circuit. This is illustrated below.
In level-sensitive sequential logic, one condition may trigger more than one boolean assignment, which are also understood to be concurrent. This is illustrated below.

The principle of concurrency applies also for edge-sensitive sequential functions, where the triggering condition is described by a vector expression rather than a boolean expression. In edge-sensitive logic, the target logic variable for the boolean assignment (LHS) may also be an operand of the boolean expression defining the assigned value (RHS). Concurrency implies that the RHS expressions are evaluated immediately before the triggering edge, and the values are assigned to the LHS variables immediately after the triggering edge. This is illustrated below.
Statements with multiple concurrent conditions for boolean assignments may also be used in sequential logic. In that case conflicting values may be assigned to the same logic variable. A default conflict resolution is not provided for the following reasons:

- Conflict resolution may not be necessary, since the conflicting situation is prohibited by specification.
- For different types of analysis (e.g., logic simulation), a different conflict resolution behavior may be desirable, while the physical behavior of the circuit will not change. For instance, pessimistic conflict resolution would always assign "X", more accurate conflict resolution would first check whether the values are conflicting. Different choices may be motivated by a trade-off in analysis accuracy and runtime.
- If complete library control over analysis is desired, conflict resolution can be specified explicitly.

Example:

```plaintext
BEHAVIOR {
    @ ( <condition_1> ) { Q = <value_1>; }
    @ ( <condition_2> ) { Q = <value_2>; }
}
```

Explicit pessimistic conflict resolution can be described as follows:

```plaintext
BEHAVIOR {
    @ ( <condition_1> && <condition_2> ) { Q = 'bX; }
    @ ( <condition_1> && ! <condition_2>) { Q = <value_1>; }
    @ ( <condition_2> && ! <condition_1>) { Q = <value_2>; }
}
```
Explicit accurate conflict resolution can be described as follows:

```
BEHAVIOR {
@ ( <condition_1> && <condition_2>  ) {
    Q = (<value_1>==<value_2>)? <value_1> : 'bX;
}
@ ( <condition_1> && ! <condition_2>) { Q = <value_1>; }
@ ( <condition_2> && ! <condition_1>) { Q = <value_2>; }
}
```

Since the conditions are now rendered mutually exclusive, equivalent descriptions with priority statements can be used. They are more elegant than descriptions with concurrent statements.

```
BEHAVIOR {
@ ( <condition_1> && <condition_2>  ) {
    Q = <conflict_resolution_value>;
}
: ( <condition_1> ) { Q = <value_1>; }
: ( <condition_2> ) { Q = <value_2>; }
}
```

Given the various explicit description possibilities, the standard does not prescribe a default behavior. The model developer has the freedom of incomplete specification.

### 3.9.4 Initial values for logic variables

Per definition, all logic variables in a behavioral description have the initial value "U" which means "uninitialized". This value cannot be assigned to a logic variable, yet it can be used in a behavioral description in order to assign other values than "U" after initialization.

Example:

```
BEHAVIOR {
@ ( Q1 == 'bU ) { Q1 = 'b1 ; }
@ ( Q2 == 'bU ) { Q2 = 'b0 ; }
    // followed by the rest of the behavioral description
}
```

A template can be used to make the intent more obvious, for example:

```
TEMPLATE VALUE_AFTER_INITIALIZATION {
@ ( <logic_variable> == 'bU ) { <logic_variable> = <initial_value> ;
}
}

BEHAVIOR {
    VALUE_AFTER_INITIALIZATION ( Q1 'b1' )
    VALUE_AFTER_INITIALIZATION ( Q2 'b0' )
    // followed by the rest of the behavioral description
}
```

Logic variables in a vector expression must be declared as PINs. It is possible to annotate initial values directly to a pin. Such variables will never take the value "U". Therefore vector expressions involving "U" for such variables (see previous example) will be meaningless.
3.10 Primitives

3.10.1 Concept of user-defined and predefined primitives

Primitives are described in ALF syntax. Primitives are generic cells containing PIN and FUNCTION objects only, i.e. no characterization data. The primitives are used for structural functional modeling.

Example:

```
PRIMITIVE MY_PRIMITIVE {
    PIN x { ... }
    PIN y { ... }
    PIN z { ... }
    FUNCTION { ... }
}

CELL MY_CELL {
    PIN a { ... }
    PIN b { ... }
    PIN c { ... }
    FUNCTION {
        BEHAVIOR { MY_PRIMITIVE { x=a; y=b; z=c; } }
        ...
    }
}
```

Extensible primitives, i.e. primitives with variable number of pins can be modeled with TEMPLATE.

Example:

```
TEMPLATE EXTENSIBLE_PRIMITIVE{
    PRIMITIVE <primitive_name> { 
        PIN [0:<max_index>] pin_name { ... }
        ...
    }
}
```

```
// instantiation of the template creates a primitive
EXTENSIBLE_PRIMITIVE {
    primitive_name = MY_EXTENSIBLE_PRIMITIVE;
    max_index = 2;
}
```
The set of statements above is equivalent to the following statement:

```plaintext
PRIMITIVE MY_EXTENSIBLE_PRIMITIVE {
    PIN [0:2] pin_name {  ... }
    ...
}
```

The primitive can be used as shown in the following example:

```plaintext
CELL MY_MEGACELL {
    PIN a { ... }
    PIN b { ... }
    PIN c { ... }
    FUNCTION {
        BEHAVIOR {
            // reference to the primitive
            MY_EXTENSIBLE_PRIMITIVE {
                pin_name[0] = a;
                pin_name[1] = b;
                pin_name[2] = c;
            }
        }
    }
    ...
}
```

Primitives can be freely defined by the user. For convenience, ALF provides a set of predefined primitives with the reserved prefix `ALF_` in their name, which cannot be used by user-defined primitives.

For all PINs of predefined primitives, the following annotations are defined per default:

```plaintext
VIEW = functional;
SCOPE = behavioral;
```

For predefined extensible primitives a placeholder may be directly in the PRIMITIVE definition:

```plaintext
PRIMITIVE ALF_EXTENSIBLE_PRIMITIVE {
    PIN [0:<max_index>] pin_name {  ... }
    ...
}
```

This is equivalent to the following more verbose set of statements:

```plaintext
TEMPLATE EXTENSIBLE_PRIMITIVE{
    PRIMITIVE <primitive_name> {
        PIN [0:<max_index>] pin_name {  ... }
        ...
    }
}
```

```plaintext
EXTENSIBLE_PRIMITIVE {
    primitive_name = ALF_EXTENSIBLE_PRIMITIVE;
    max_index = <max_index>;
}
```
3.10.2 Predefined combinational primitives

3.10.2.1 One input, multiple output primitives

There are two combinational primitives with one input pin and multiple output pins:

\[ \text{ALF_BUF}, \text{ALF_NOT} \]

A GROUP statement is used to define the behavior of all output pins in one statement. 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. \text{out} refers to \text{out[0]}.

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

Figure 3-23: Primitive model of ALF_BUF

```
PRIMITIVE ALF_NOT {
   GROUP index {0:<max_index>}
   PIN[0:<max_index>] out {
      DIRECTION = output ;
   }
   PIN in {
      DIRECTION = input ;
   }
   FUNCTION {
      BEHAVIOR {
         out[index] = !in;
      }
   }
}
```

Figure 3-24: Primitive model of ALF_NOT

3.10.2.2 One output, multiple input primitives

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

\[ \text{ALF_AND}, \text{ALF_NAND}, \text{ALF_OR}, \text{ALF_NOR}, \text{ALF_XOR}, \text{ALF_XNOR} \]
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].

```plaintext
PRIMITIVE ALF_AND {
    PIN out {
        DIRECTION = output;
    }
    PIN[0:<max_index>] in {
        DIRECTION = input;
    }
    FUNCTION {
        BEHAVIOR {
            out = & in;
        }
    }
}
```

Figure 3-25: Primitive model of ALF_AND

```plaintext
PRIMITIVE ALF_NAND {
    PIN out {
        DIRECTION = output;
    }
    PIN[0:<max_index>] in {
        DIRECTION = input;
    }
    FUNCTION {
        BEHAVIOR {
            out = ~& in;
        }
    }
}
```

Figure 3-26: Primitive model of ALF_NAND

```plaintext
PRIMITIVE ALF_OR {
    PIN out {
        DIRECTION = output;
    }
    PIN[0:<max_index>] in {
        DIRECTION = input;
    }
    FUNCTION {
        BEHAVIOR {
            out = | in;
        }
    }
}
```

Figure 3-27: Primitive model of ALF_OR
PRIMITIVE ALF_NOR {
   PIN out {
         DIRECTION = output;
   }
   PIN[0:<max_index>] in {
         DIRECTION = input;
   }
   FUNCTION {
         BEHAVIOR {
               out = ~| in;
         }
   }
}

Figure 3-28: Primitive model of ALF_NOR

PRIMITIVE ALF_XOR {
   PIN out {
         DIRECTION = output;
   }
   PIN[0:<max_index>] in {
         DIRECTION = input;
   }
   FUNCTION {
         BEHAVIOR {
               out = ^in;
         }
   }
}

Figure 3-29: Primitive model of ALF_XOR

PRIMITIVE ALF_XNOR {
   PIN out {
         DIRECTION = output;
   }
   PIN[0:<max_index>] in {
         DIRECTION = input;
   }
   FUNCTION {
         BEHAVIOR {
               out = ~^in;
         }
   }
}

Figure 3-30: Primitive model of ALF_XNOR
3.10.3 Predefined tristate Primitives

There are four tristate primitives:

\[ \text{ALF_BUFIF1, ALF_BUFIF0, ALF_NOTIF1, ALF_NOTIF0} \]

**PRIMITIVE ALF_BUFIF1**

PIN out {
  DIRECTION = output;
  ENABLE_PIN = enable;
  ATTRIBUTE {TRISTATE}
}
PIN in {
  DIRECTION = input;
}
PIN enable {
  DIRECTION = input;
  SIGNALTYPE = out_enable;
}
FUNCTION {
  BEHAVIOR {
    out = (enable)? in : 'bZ;
  }
  STATETABLE {
    enable in : out;
    0 ? : Z;
    1 ? : (in);
  }
}

**Figure 3-31: 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 {
  BEHAVIOR {
    out = (!enable)? in : 'bZ;
  }
}
Figure 3-32: Primitive model of ALF_BUFIF0

Figure 3-33: Primitive model of ALF_NOTIF1

Figure 3-34: Primitive model of ALF_NOTIF0
BEHAVIOR {
    out = (!enable)? !in : 'bZ;
}
STATETABLE {
    enable in : out;
    1  ?  : Z;
    0  ?  : (!in);
}

Figure 3-34: Primitive model of ALF_NOTIF0

3.10.4 Predefined multiplexor

The predefined multiplexor has a known output value if either the select signal and the selected data inputs are known or both data inputs have the same known value while the select signal is unknown.

PRIMITIVE ALF_MUX {
    PIN Q {
        DIRECTION = output;
        SIGNALTYPE = data;
    }
    PIN[1:0] D {
        DIRECTION = input;
        SIGNALTYPE = data;
    }
    PIN S {
        DIRECTION = input;
        SIGNALTYPE = select;
    }
    FUNCTION {
        BEHAVIOR {
            Q = (S || (d[0] ^ d[1]))? d[1] : d[0];
        }
        STATETABLE {
            D[0] D[1] S  : Q ;
            ?  ?  0  : (D[0]);
            ?  ?  1  : (D[1]);
            0  0  ?  : 0;
            1  1  ?  : 1;
        }
    }
}

Figure 3-35: Primitive model of ALF_MUX
## 3.10.5 Predefined 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 \( Q\text{\_CONFLICT} \) and \( QN\text{\_CONFLICT} \), which are virtual pins. They specify the state of \( Q \) and \( QN \) in the event \texttt{CLEAR} and \texttt{SET} become active simultaneously.

```plaintext
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;
  }
}
```
Primitives Library Format Specification

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;
    }
}
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 3-36: Primitive model of ALF_FLIPFLOP

3.10.6 Predefined 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) instead of the edge-sensitive clock.

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 {
  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) {
      Q = D;
      QN = !D;
    }
  }
  STATETABLE {
    D ENABLE CLEAR SET QX QNX : Q QN ;
    ? ? 1 1 ? ? : (QX) (QNX);
  }
}
3.11 Parameterizeable Cells

The concept of describing primitives with variable bus size shall be extended to parameterize-
able cells. Dynamic template instantiations are introduced for that purpose.

Template definitions may incorporate any type of object. Placeholders in the template defini-
tion are the equivalent of parameters. Hence the definition of parameterizeable cells is already
supported within the support of general template definitions.

In a static template instantiation, which is identified by the name of the template and by the
optional value assignment static, placeholders are replaced by fixed values or by complex
objects containing fixed values. Non-referenced placeholders will stay in place and eventually
result in semantically unrecognizable objects, which cannot be processed by downstream
applications. Such unrecognizable objects shall be disregarded.

In a dynamic template instantiation, which is identified by the name of the template and by the
mandatory value assignment dynamic, some placeholders may not be replaced. Those place-
holders are application parameters. The template definition may already contain certain rela-
tionships between parameters (e.g. arithmetic model and its arguments in the header).
Therefore the template instantiation determines, which parameters need application values in
order to calculate values for other parameters.

Going one step further, even the relationship between parameters may be defined in the
dynamic template instantiation rather than in the template definition. In this case, the identifi-
ers inside the placeholders become variables for arithmetic assignments. This definition of
variables shall only be recognized within the context of the dynamic template instantiation.

Arithmetic assignments provide a shorter syntax for equation-based arithmetic models where
only placeholder-parameters are involved.

\[
\text{param1} = 1.5 + 0.4 \times \text{param2}^3 - 2.7 / \text{param3}
\]

is equivalent to

```
param1 {
  HEADER { param2 param3 }
  EQUATION { 1.5 + 0.4 * param2 ** 3 - 2.7 / param3 }
}
```
For table-based models or for models where the arguments have children objects attached to them, the verbose syntax with HEADER must be used.

Example:

TEMPLATE adder {
  CELL <cellname> {
    PIN [ <bitwidth> : 1 ] A { DIRECTION = input; }
    PIN [ <bitwidth> : 1 ] B { DIRECTION = input; }
    PIN Cin { DIRECTION = input; }
    PIN [ <bitwidth> : 1 ] S { DIRECTION = output; }
    PIN Cout { DIRECTION = output; }

    FUNCTION {
      BEHAVIOR {
        S = A + B + Cin;
        Cout = (A + B + Cin >= ('b1 << (<bitwidth> - 1)));
      }
    }
    AREA = <areavalue>;
    VECTOR (?! Cin -> ?! Cout) {
      DELAY {
        HEADER {
          CAPACITANCE {PIN = Cout; }
          SLEWRATE {PIN = Cin; }
        }
        EQUATION { </D0> + </D1>*CAPACITANCE + </D2>*SLEWRATE }
      }
    }
  }
}

The template is used for instantiation of a hardmacro:

adder { /* a hardmacro */
  cellname = ripple_carry_adder_16_bit;
  bitwidth = 16;
  areavalue = 500;
  // D0, D1, D2 are undefined. DELAY cannot be calculated.
}

The static instantiation of the hardmacro is equivalent to the following static object:

CELL ripple_carry_adder_16_bit {
  PIN [ 16 : 1 ] A { DIRECTION = input; }
  PIN [ 16 : 1 ] B { DIRECTION = input; }
  PIN Cin { DIRECTION = input; }
  PIN [ 16 : 1 ] S { DIRECTION = output; }
  PIN Cout { DIRECTION = output; }

  FUNCTION {
    BEHAVIOR {
      S = A + B + Cin;
      Cout = (A + B + Cin >= 'b1000000000000000);
    }
  }
}
AREA = 500;

VECTOR (?! Cin -> ?! Cout) {
//   DELAY {
//     HEADER {
//       CAPACITANCE {PIN = Cout; }
//       SLEWRATE {PIN = Cin; }
//     }
//   }
//   EQUATION { <D0> + <D1>*CAPACITANCE + <D2>*SLEWRATE }
// }
}

Now the template is used for instantiation of a softmacro:

adder = dynamic { /* a softmacro */
cellname = ripple_carry_adder_N_bit;
areavalue = 20 + 30 * bitwidth;
}
D0 {
   HEADER { AREA { TABLE { 10 20 30 } } }
   TABLE { 15.6 34.3 50.7 }
}
D1 = 0.29;
D2 = 0.08;
}

The dynamic instantiation of the softmacro results in an object for which certain data depend on the runtime-values of the placeholder-parameters, as indicated in italic below. The calculation method for such data, however, can be compiled statically (e.g. the equation for AREA as a function of bitwidth, the lookup table for D0 as a function of AREA).

CELL ripple_carry_adder_N_bit {
   PIN [ bitwidth : 1 ] A { DIRECTION = input; }
   PIN [ bitwidth : 1 ] B { DIRECTION = input; }
   PIN Cin { DIRECTION = input; }
   PIN [ bitwidth : 1 ] S { DIRECTION = output; }
   PIN Cout { DIRECTION = output; }

   FUNCTION {
      BEHAVIOR {
         S = A + B + Cin;
         Cout = (A + B + Cin >= ('b1 << (bitwidth - 1)));
      }
   }

   AREA = 20 + 30 * bitwidth ;

   VECTOR (?! Cin -> ?! Cout) {
      DELAY {
         HEADER {
            CAPACITANCE {PIN = Cout; }
         }
      }
   }
}
SLEWRATE \{PIN = Cin; \}
D0 {
    HEADER \{ AREA \{ TABLE \{ 10 20 30 \} \} \}
    TABLE \{ 15.6 34.3 50.7 \}
}
EQUATION \{ D0 + 0.29*CAPACITANCE + 0.08*SLEWRATE \}
}

### 3.12 Modeling with Vector Expressions

Vector expressions provide a formal language to describe digital waveforms. This capability can be used for functional specification, for timing and power characterization and for timing and power analysis.

In particular, vector expressions add value by addressing the following modeling issues:

- **Functional specification**: complex sequential functionality, for example bus protocols.
- **Timing analysis**: complex timing arcs and timing constraints involving more than two signals.
- **Power analysis**: temporal and spatial correlation between events relevant for power consumption.
- **Circuit characterization and test**: specification of characterization and/or test vectors for particular timing, power, fault or other measurements within a circuit.

Like boolean expressions, vector expressions provide means for description of functionality of digital circuits in various contexts without being self-sufficient. Vector expressions enrich this functional description capability by adding a “dynamic” dimension to the otherwise “static” boolean expressions.

The following subsections explain the semantics of vector expressions step by step. The vector expression concept is introduced using terminology from simulation event reports. This is because the ideas can be easily explained this way. However, the application of vector expressions is not restricted to post-processing event reports.

Some application tools, for example power analysis, tools may actually evaluate vector expressions during post-processing of event reports from simulation. Other application tools, especially simulation model generators, must respect the causality between the triggering events and the actions to be triggered. While it is semantically impossible to describe cause and effect in the same vector expression for the purpose of functional modeling, both cause and effect may appear in a vector expression used for a timing arc description.

ALF does not make assumption about the physical nature of the event report. Vector expressions can be applied to an actual event report written in a file, or to an internal event queue within a simulator, or to a hypothetical event report which is merely a mathematical concept.
3.12.1 Event reports

This section describes the terminology of event reports from simulation, which will be used as an instrument to explain the concept of ALF vector expressions. The intent of ALF vector expressions is not to replace existing event report formats. Non-pertinent details of event report formats are not described here.

Simulation events (e.g. from Verilog or VHDL) can be reported in a value change dump (VCD) file, which has the following general form:

```
<time1>
  <variableA> <stateU>
  <variableB> <stateV>
  ...
<time2>
  <variableC> <stateW>
  <variableD> <stateX>
  ...
<time3> ...
```

The set of variables for which simulation events are reported, i.e. the scope of the event report need to be defined beforehand. Each variable also has a definition for the set of states it can take. For instance, there may be binary variables, 16-bit integer variables, 1-bit variables with drive-strength information etc. Furthermore, the initial state of each variable must be defined as well. In an ALF context, we may use the term “signal” and “variable” interchangeably. In VHDL, the corresponding term is “signal”. In Verilog, there is no single corresponding term. All “input”, “output”, “wire”, “reg” variables in Verilog correspond to “signal” in VHDL.

The time values <time1>, <time2>, <time3> etc. must be in increasing order. The order in which simultaneous events are reported does not matter. The number of time points and the number of simultaneous events at a certain time point are unlimited.

In the physical world, each event or change of state of a variable takes a certain amount of time. A variable cannot change its state more than once at a given point in time. However, in simulation, this time may be sometimes smaller than the resolution of the time scale, or even zero. Therefore a variable may change its state more than once at a given point in simulation time. Those events are, strictly speaking, not simultaneous. They occur in a certain order, separated by an infinitely small delta-time. Multiple simultaneous events of the same variable are not reported in the VCD. Only the final state of each variable is reported.

A VCD file is the most compact format that allows reconstruction of entire waveforms for a given set of variables. A more verbose form is the test pattern format.

```
<TIME>  <variableA> <variableB> <variableC> <variableD>
<time1> <stateU>    <stateV>    ...         ...
<time2> <stateU>    <stateV>    <stateW>    <stateX>
<time3> ...         ...         ...         ...
```

The test pattern format reports the state of each variable at every point in time, regardless whether the state has changed or not. Previous and following states are immediately available in the previous and next row, respectively. This makes the test pattern format more readable than the VCD and well-suited for taking a snapshot of events in a time window.
Example of an event report in VCD format:

```plaintext
// initial values
A 0 B 1 C 1 D X E 1
// event dump
109  A 1  D 0
258  B 0
573  C 0
586  A 0
643  A 1
788  A 0 B 1 C 1
915  A 1
1062 E 0
1395 B 0 C 0
1640 A 0 D 1
// end of event dump
```

Example of an event report in test pattern format:

```plaintext
time    A    B    C    D    E
0       0    1    1    X    1
109     1    1    1    0    1
258     1    0    1    0    1
573     1    0    0    0    1
586     0    0    0    0    1
643     1    0    0    0    1
788     0    1    1    0    1
915     1    1    1    0    1
1062    1    1    1    0    0
1395    1    0    0    0    0
1640    0    0    0    1    0
```

Both VCD and test pattern formats represent the same amount of information and can be translated into each other.

### 3.12.2 Event Sequences

For specification of a functional waveform (for example the write cycle of a memory), it is not practical to use an event report format, such as VCD or test pattern format. In such waveforms, there is no absolute time. And the relative time, for example the setup time between address change and write enable change, may vary from one instance to the other.

The main purpose of `vector_expressions` is waveform specification capability. The following operators are introduced:

- **vector_unary** (also called “edge operator” or “unary vector operator”)
  The edge operator is a prefix to a variable in a vector expression. It contains a pair of states, the first being the previous state, the second being the new state.
  In the following presentation the set of edge operators with different previous and new state, where there are (N-1)*N possibilities in a system of N possible states, are treated first. Later we shall also explain edge operators that have no change of state.

- **vector_and** (also called “simultaneous event operator”)
  This operator uses the overloaded symbol “&” or “&&” interchangeably.
  The “&” operator is the separator between simultaneously occurring events
• vector_followed_by (also called “followed-by operator”)
  The “immediately followed-by operator” using the symbol “->” is treated first.
  The “->” operator is the separator between consecutively occurring events.

These operators are necessary and sufficient to describe the following subset of vector_expressions:

• vector_single_event
  A change of state in a single variable, for example:
  01 A

• vector_event
  A simultaneous change of state in one or more variables, for example:
  01 A & 10 B

• vector_event_sequence
  Subsequently occurring changes of state in one or more variables, for example:
  01 A & 10 B -> 10 A

The vector_and operator has a higher binding priority than the vector_followed_by operator.

We can now express the pattern of the sample event report in a vector_event_sequence expression:

```
01 A & X0 D -> 10 B -> 10 C -> 10 A -> 01 A
  -> 10 A & 01 B & 01 C -> 01 A -> 10 E -> 10 B & 10 C -> 10 A & 01 D
```

We can define the length of a vector_event_sequence expression as the number of subsequent events described in the vector_event_sequence expression. The length is equal to the number of "->" operators plus one.

Although the vector expression format contains an inherent redundancy, since the old state of each variable is always the same as the new state of the same variable in a previous event, it is more human-readable, especially for waveform description. On the other hand, it is more compact than the test pattern format. For short event sequences, it is even more compact than the VCD, since it eliminates the declaration of initial values. To be accurate, for variables with exactly one event the vector expression is more compact than the VCD. For variables with more than one event the VCD is more compact than the vector expression. In summary, the vector expression format offers readability similar to the test pattern format and compactness close to the VCD format.

Again, the intent is not to propose another event report format but to specify a pattern of events that may be detected within an event report.

### 3.12.3 Scope and content of event sequences

The scope applicable to a vector expression defines the set of variables in the event report. The content of a vector expression is the set of variables that appear in the vector expression itself. The content of a vector expression must be a subset of variables within scope.

• PINs with the annotation SCOPE = BEHAVIOR are applicable variables for vector expressions within the context of BEHAVIOR.
• PINs with the annotation SCOPE = MEASURE are applicable variables
for vector expressions within the context of VECTOR.

- PINs with the annotation `SCOPE = BOTH` are applicable variables for all vector expressions.

A `vector_event_sequence` expression is an event pattern without time, containing only the variables within its own content. This event pattern is evaluated against the event report containing all variables within scope. The vector expression is true when the event pattern matches the event report.

Example:

<table>
<thead>
<tr>
<th>time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>// scope is A, B, C, D, E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>258</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>573</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>586</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>643</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>788</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>915</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1062</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1395</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1640</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Consider the following vector expressions in the context of the sample event report:

- `01 A // (1) content is A`

  // event pattern expressed by (1):
  //   A
  //   0
  //   1

  (1) will be true at time 109, at time 643 and at time 915.

- `10 B -> 10 C // (2) content is B, C`

  // event pattern expressed by (2):
  //   B C
  //   1 1
  //   0 1
  //   0 0

  (2) will be true at time 573.

- `10 A -> 01 A // (3) content is A`

  // event pattern expressed by (3):
  //   A
  //   1
  //   0
  //   1

  (3) will be true at time 643 and at time 915.
01 D
  // (4) content is D

  // event pattern expressed by (4):
  //  D
  //   0
  //   1

(4) will be true at time 1640.

01 A -> 10 C
  // (5) content is A, C

  // event pattern expressed by (5):
  //    A   C
  //   0   1
  //   1   1
  //   1   0

(5) will not be true at any time, since the event pattern expressed by (5) does not match the event report at any time.

3.12.4 Alternative event sequences

The following operator is introduced to describe alternative events:

- `vector_or`, also called “event-or operator” or “alternative-event operator”, using the overloaded symbol “|” or “||” interchangeably.

  The “|” operator is the separator between alternative events or alternative event sequences.

In analogy to boolean operators, “|” has a lower binding priority than “&” and “->”. Parentheses can be used to change the binding priority.

Example:

(01 A -> 01 B) | 10 C === 01 A -> 01 B | 10 C
01 A -> (01 B | 10 C) === 01 A -> 01 B | 01 A -> 10 C

Consider the following vector expressions in the context of the sample event report:

01 A | 10 C
  // (6)

  // event pattern expressed by (6):
  //   A
  //   0
  //   1

  // alternative event pattern expressed by (6):
  //   C
  //   1
  //   0

(6) will be true at time 109, at time 573, at time 643, at time 915 and at time 1395.
10 B -> 10 C | 10 A -> 01 A  // (7)
  //event pattern expressed by (7):
  //  B  C
  //   1  1
  //   0  1
  //   0  0
  //alternative event pattern expressed by (7):
  //    A
  //     1
  //     0
  //     1

(7) will be true at time 573, at time 643 and at time 915.

01 D | 10 B -> 10 C  // (8)
  //event pattern expressed by (8):
  //    D
  //     0
  //     1
  //alternative event pattern expressed by (8):
  //    B  C
  //     1  1
  //     0  1
  //     0  0

(8) will be true at time 573 and at time 1640.

10 B -> 10 C | 10 A  // (9)
  //event pattern expressed by (9):
  //    B  C
  //     1  1
  //     0  1
  //     0  0
  //alternative event pattern expressed by (9):
  //     A
  //      1
  //      0

(9) will be true at time 573, at time 586, at time 788 and at time 1640.

The following operators are introduced for a more compact description of certain alternative event sequences:

- “&>” events occur simultaneously or follow each other in the order RHS after LHS
- “<->” LHS event followed by RHS event or RHS event followed by LHS event
- “<&>” events occur simultaneously or follow each other in arbitrary order

Example:

01 A &> 01 C === 01 A & 01 C | 01 A -> 01 C
01 A <-> 01 C === 01 A -> 01 C | 01 C -> 01 A
01 A <&> 01 C === 01 A <-> 01 C | 01 A & 01 C

The binding priority of these operators is higher than of “&” and “->”.
3.12.5 Symbolic edge operators

Alternative events of the same variable can be described in a even more compact way through use of edge operators with symbolic states. The symbol “?” stands for “any state”.

- edge operator with “?” as previous state:
  transition from any state to the defined new state
- edge operator with “?” as next state:
  transition from the defined previous state to any state.

Both edge operators include the possibility that no transition occurred at all, i.e., the previous and the next state are the same. This situation can be explicitly described with the following operator:

- edge operator with next state = previous state, also called “non-event operator”
  The operand stays in the state defined by the operator.

The following symbolic edge operators are also introduced:

- “?~” no transition on the operand
- “?!” transition from any state to any state different from the previous state
- “??” transition from any state to any state or no transition on the operand
- “?-” transition from any state to its bitwise complementary state

Example: Let “A” be a logic variable with the possible states “1”, “0”, “X”.

```
?0 A === 00 A | 10 A | X0 A
?1 A === 01 A | 11 A | X1 A
?X A === 0X A | 1X A | XX A
0? A === 00 A | 01 A | 0X A
1? A === 10 A | 11 A | 1X A
X? A === X0 A | X1 A | XX A
?! A === 01 A | 0X A | 10 A | 1X A | X0 A | X1 A
?~ A === 01 A | 10 A | XX A
?? A === 00 A | 01 A | 0X A | 10 A | 11 A | 1X A | X0 A | X1 A | XX A
?- A === 00 A | 11 A | XX A
```

For variables with more possible states (e.g. logic states with different drive strength, multiple bits) the explicit description of alternative events would be quite verbose. Therefore the symbolic edge operators are useful for a more compact description.

So far we have introduced the set of vector_binary operators necessary for the description of a subset of vector_expressions called vector_complex_event expressions. All vector_binary operators have two vector_complex_event expressions as operands. The set of vector_event_sequence expressions is a subset of vector_complex_event expressions. Every vector_complex_event expression can be expressed in terms of alternative vector_event_sequence expressions. The latter could be called “minterms”, in analogy to boolean algebra.
3.12.6 Non-events

A vector_single_event expression involving a non-event operator is called a non-event. A rigorous definition is required for vector_complex_event expressions containing non-events. Consider the following example of a flipflop with clock input CLK and data output Q.

```
01 CLK -> 01 Q  // (i)
01 CLK -> 00 Q  // (ii)
```

The vector expression (i) describes the situation that the output switches from 0 to 1 after the rising edge of the clock. The vector expression (ii) describes the situation that the output remains at 0 after the rising edge of the clock.

How is it possible to decide whether (i) or (ii) is true, without knowing the delay between CLK and Q? The only way is to wait until any event occurs after the rising edge of CLK. If the event is not on Q and the state of Q is 0 during that event, then (ii) is true.

Hence a non-event is true every time when another event happens and the state of the variable involved in the non-event satisfies the edge operator of the non-event.

Example:

```
time  A  B  C  D  E
0    0  0  1  1  X  1
109  1  1  1  0  1
258  1  0  1  0  1
573  1  0  0  0  1
586  0  0  0  0  0
643  1  0  0  0  1
788  0  1  1  0  1
915  1  1  1  0  1
1062 1  1  1  0  0
1395 1  0  0  0  0
1640 0  0  0  1  0
```

The test pattern format represents an event, for example “01 A “, in no different way than a non-event, for example “11 E “. This non-event is true at time 109, 258, 573, 586, 643, 788, 915, in short every time when an event happens while E is constant 1.

3.12.7 Compact and verbose event sequences

A vector_event_sequence expression in a compact form can be transformed into a verbose form by padding up every vector_event expression with non-events. The next state of each variable within a vector_event expression must be equal to the previous state of the same variable in the subsequent vector_event expression.

Example:

```
01 A -> 10B === 01 A & 11 B -> 11 A & 10 B
```
A vector expression for a complete event report in compact form resembles the VCD, whereas the verbose form looks like the test pattern.

```
// compact form
01 A & X0 D -> 10 B -> 10 C -> 10 A -> 01 A
-> 10 A & 01 B & 01 C -> 01 A -> 10 E
-> 10 B & 10 C -> 10 A & 01 D
===
// verbose form
?0 A & ?1 B & ?1 C & ?X D & ?1 E->
01 A & 11 B & 11 C & X0 D & 11 E->
11 A & 10 B & 11 C & 00 D & 11 E->
11 A & 00 B & 10 C & 00 D & 11 E->
10 A & 00 B & 00 C & 00 D & 11 E->
01 A & 00 B & 00 C & 00 D & 11 E->
10 A & 01 B & 01 C & 00 D & 11 E->
01 A & 11 B & 11 C & 00 D & 11 E->
11 A & 11 B & 11 C & 00 D & 10 E->
11 A & 10 B & 10 C & 00 D & 00 E->
10 A & 00 B & 00 C & 01 D & 00 E
```

The transformation rule must be slightly modified in case the compact form contains a vector_event expression consisting only of non-events. By definition, the non-event is true only if a real event happens simultaneously with the non-event. Padding up a vector_event expression consisting of non-events with other non-events would make this impossible. Rather, this vector_event expression should be padded up with unspecified events, using the “??” operator. Eventually, unspecified events can be further transformed into partly specified events, if a former or future state of the involved variable is known.

Example:

```
01 A -> 00 B
=== 01 A & 00 B -> ?? A & 00 B
```

In the first transformation step, the unspecified event “?? A” is introduced.

```
01 A & 00 B -> ?? A & 00 B
=== 01 A & 00 B -> 1? A & 00 B
```

In the second step, this event becomes partly specified. “?? A” is bound to be “1? A” due to the previous event on A.

### 3.12.8 Unspecified simultaneous events within scope

Variables which are within the scope of the vector expression yet do not appear in the vector expression can be used to pad up the vector expression with unspecified events as well. This is equivalent to omitting them from the vector expression.

Example:

```
01 A -> 10 B // let us assume a scope containing A, B, C, D, E
===
01 A & 10 B & ?? C & ?? D & ?? E -> 11 A & 10 B & ?? C & ?? D & ?? E
```
This definition allows unspecified events to occur *simultaneously* with specified events or specified non-events. However, it disallows unspecified events to occur *in-between* specified events or specified non-events.

At first sight, this distinction seems to be arbitrary. Why not disallow unspecified events altogether? Yet there are several reasons why this definition is practical:

If a vector expression disallows simultaneously occurring unspecified events, the application tool has the burden not only to match the pattern of specified events with the event report but also to check whether the other variables remain constant. Therefore it is better to specify this extra pattern matching constraint explicitly in the vector expression, using the “?-” operator.

There are many cases where it actually does not matter whether simultaneously occurring unspecified events are allowed or disallowed:

- **Case 1:** Simultaneous events are impossible by design. For instance, in a flipflop it is impossible that a triggering clock edge “01 CK” and a switch of the data output “- Q” happen at the same time. Therefore such events will not be in the event report. It makes no difference whether to specify “01 CK & - Q” or “01 CK & ?? Q” or “01 CK”. The only occurring event pattern will be “01 CK & - Q”, and this pattern can be reliably detected by specifying “01 CK”.

- **Case 2:** Simultaneous events are prohibited by design. For instance, in a flipflop with positive setup time and positive hold time, the triggering clock edge “01 CK” and a switch of the data input “?! D” is a timing violation. A timing checker tool needs the violating pattern specified explicitly, i.e. “01 CK &?! D”. In this context it makes sense to specify the non-violating pattern also explicitly, i.e. “01 CK &?- D”. The pattern “01 CK” by itself is not applicable.

- **Case 3:** Simultaneous events do not occur in correct design. For instance, power analysis of the event “01 CK” needs no specification of “?! D” or “?- D”. In the analysis of an event report with timing violations, the power analysis will be less accurate anyway. In the analysis of the event report for the design without timing violation, the only occurring event pattern will be “01 CK &?- D”, and this pattern can be reliably detected by specifying “01 CK”. 1

- **Case 4:** The effects of simultaneous events are not modeled accurately. This is the case in static timing analysis and also to some degree in dynamic timing simulation. For instance, a NAND gate may have the inputs A and B and the output Z. The event sequence exercising the timing arc “01 A -> 10 Z” can only happen if B is constant 1. No event on B can happen in-between “01 A” and “10 Z”.
  Likewise, the timing arc “01 B -> 10 Z” can only happen if A is constant 1 and no event happens in-between “01 B” and “10 Z”.
  The timing arc with simultaneously switching inputs is commonly ignored. A tool encountering the scenario “01 A & 01 B -> 10 Z” has no choice other than treating it arbitrarily as “01 A -> 10 Z” or as “01 B -> 10 Z”.

---

1. The power analysis tool related to a timing constraint checker in a similar way as a parasitic extraction tool relates to a DRC tool. If the layout has DRC violations, for instance shorts between nets, the parasitic extraction tool will report inaccurate wire capacitance for those nets. After final layout, the DRC violations will be gone and the wire capacitance will be accurate.
• Case 5: The effects of simultaneous events are modeled accurately. Here it makes sense to specify all scenarios explicitly, i.e.,
“01 A & ?- B -> 10 Z”, “01 A &?! B -> 10 Z”, “?- A & 01 B -> 10 Z” etc., whereas the patterns “01 A -> 10 Z” and “01 B -> 10 Z” by themselves apply only for less accurate analysis (see case 4).

There is also a formal argument why unspecified events on a vector expression should be allowed rather than disallowed. Let us consider the following vector expressions within the scope of two variables A and B.

\[
\begin{align*}
01 A & \quad \text{\textit{(i)}} \\
01 B & \quad \text{\textit{(ii)}} \\
01 A & \& 01 B & \quad \text{\textit{(iii)}} \\
\end{align*}
\]

One would naturally interpret (iii) === (i) & (ii). This interpretation is only possible by allowing simultaneously occurring unspecified events.

\textbf{Allowing} simultaneously occurring unspecified events, the vector expressions (i) and (ii), respectively, are interpreted as follows:

\[
\begin{align*}
01 A & \& ?? B & \quad \text{\textit{(i')}} \\
?? A & \& 01 B & \quad \text{\textit{(ii')}} \\
\end{align*}
\]

\textbf{Disallowing} simultaneously occurring unspecified events, the vector expressions (i) and (ii), respectively, are interpreted as follows:

\[
\begin{align*}
01 A & \& ?- B & \quad \text{\textit{(i'')}} \\
?- A & \& 01 B & \quad \text{\textit{(ii'')}} \\
\end{align*}
\]

The vector expressions (i’) and (ii’) are compatible with (iii) whereas (i’’) and (ii’’) are not.

\section*{3.12.9 Simultaneous event sequences}

The semantic meaning of the “simultaneous event operator” can be extended to describe simultaneously occurring \textit{event sequences}, by introducing the following definition:

\[
(01 A^1 \ldots \rightarrow \ldots 01 A^N) \& (01 B^1 \ldots \rightarrow \ldots 01 B^N) \\
==\quad 01 A^1 & 01 B^1 \ldots \rightarrow \ldots 01 A^N & 01 B^N
\]

This definition is analogous to scalar multiplication of vectors with the same number of indices. The number of indices corresponds to the number of \textit{vector_event} expressions separated by “->” operators. If the number of “->” in both vector expressions is not the same, the shorter vector expression can be left-extended with unspecified events, using the “??” operator, in order to align both vector expressions.

Example:

\[
\begin{align*}
(01 A \rightarrow 01 B \rightarrow 01 C) & \& (01 D \rightarrow 01 E) \\
==\quad (01 A \rightarrow 01 B \rightarrow 01 C) & \& (?? D \rightarrow 01 D \rightarrow 01 E) \\
==\quad 01 A & ?? D \rightarrow 01 B & 01 D \rightarrow 01 C & 01 E \\
==\quad 01 A \rightarrow 01 B & 01 D \rightarrow 01 C & 01 E
\end{align*}
\]
The easiest way to understand the meaning of “simultaneous event sequences” is to consider the event report in test pattern format. If each vector_event_sequence expression matches the event report in the same time window, then the event sequences happen simultaneously.

<table>
<thead>
<tr>
<th>time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>109</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>258</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>573</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>586</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>643</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>788</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>915</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1062</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1395</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1640</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Example:

01 A \rightarrow 10 B \implies 01 A \& 11 B \rightarrow 11 A \& 10 B \quad // (10a)

// event pattern expressed by (10a):
// A   B
// 0   1
// 1   1
// 1   0

X0 D \rightarrow 00 D \quad // (10b)

// event pattern expressed by (10b):
// D
// X
// 0
// 0

(01 A \rightarrow 10 B) \& (X0 D \rightarrow 00 D) \quad // (10) \implies (10a)\&(10b)

Both (10a) and (10b) are true at time 258. Therefore (10) is true at time 258.

10 C
\implies ?? C \rightarrow ?? C \rightarrow 10 C
\implies ?? C \rightarrow ?1 C \rightarrow 10 C \quad // (11a)

// event pattern expressed by (11a):
// C
// ?
// ?
// 1
// 0

(11a) is left-extended to match the length of the following (11b).

01 A \rightarrow 00 D \rightarrow 11 E \implies
01 A \& 00 D \& ?? E
\rightarrow ?? A \& 00 D \& ?? E
\rightarrow ?? A \& ?? D \& 11 E
\implies
01 A \& 00 D \& ?? E
(11b) contains explicitly specified non-events. The non-event “00 D” calls for the unspecified events “?? A” and “?? E”. The non-event “00 E” calls for the unspecified events “?? A” and “?? D”. By propagating well-specified previous and next states to subsequent events, some unspecified events become partly specified.

(11a) is true at time 573 and at time 1395. (11b) is true at time 573 and at time 915. Therefore (11) is true at time 573.

### 3.12.10 Implicit local variables

Until now, vector expressions are evaluated against an event report containing all variables within the scope of a cell. It is practical for the application to work with only one event report per cell or at most two event reports, if the set of variables for BEHAVIOR (scope=behavior) and VECTOR (scope=measure) was different. However, for complex cells and megacells, it is sometimes necessary to change the scope of event observation, dependent on operation modes. Different modes may require a different set of variables to be observed in different event reports.

The following definition allows to extend the scope of a vector expression locally:

- Edge operators apply not only to variables but also to boolean expressions involving those variables. Those boolean expressions represent implicit local variables that are visible only within the vector expression where they appear.

Let us insert the local variables \((A \& B), (A | B)\) into the event report:

| time | A  | B  | C  | D  | E  | A&B | A|B |
|------|----|----|----|----|----|-----|---|
| 0    | 0  | 0  | 1  | 1  | X  | 1   | 0 |
| 109  | 1  | 1  | 1  | 0  | 1  | 1   | 1 |
| 258  | 1  | 0  | 1  | 0  | 1  | 0   | 1 |
| 573  | 1  | 0  | 0  | 0  | 1  | 0   | 1 |
| 586  | 0  | 0  | 0  | 0  | 1  | 0   | 0 |
| 643  | 1  | 0  | 0  | 0  | 1  | 0   | 1 |
| 788  | 0  | 1  | 1  | 0  | 1  | 0   | 1 |
| 915  | 1  | 1  | 1  | 0  | 1  | 1   | 1 |
| 1062 | 1  | 1  | 1  | 0  | 0  | 1   | 1 |
| 1395 | 1  | 0  | 0  | 0  | 0  | 0   | 1 |
| 1640 | 0  | 0  | 0  | 1  | 0  | 0   | 0 |
Example:

```plaintext
01 (A & B) // (12)
// event pattern expressed by (12):
//   A&B
//   0
//   1
```

(12) is true at time 109 and at time 915.

```plaintext
10 (A | B) // (13)
// event pattern expressed by (13):
//   A|B
//   1
//   0
```

(13) is true at time 586 and at time 1640.

```plaintext
01 (A & B) -> 10 B // (14)
// event pattern expressed by (14):
//   B   A&B
//   1   0
//   1   1
//   0   1
```

(14) is true at time 258.

```plaintext
10 (A & B) & 10 B -> 10 C // (15)
// event pattern expressed by (15):
//   B   C   A&B
//   1   1   1
//   0   1   0
//   0   0   0
```

(15) is true at time 573.

```plaintext
10 (A & B) -> 10 (A | B) // (16)
// event pattern expressed by (16):
//   A&B   A|B
//   1   1
//   0   1
//   0   0
```

(16) is true at time 1640.

### 3.12.11 Conditional event sequences

The following definition allows to restrict the scope of a vector expression locally:

- **vector_boolean_and**, also called “conditional event operator”
  This operator is defined between a vector expression and a boolean expression, using the overloaded symbol “&” or “&&”. The scope of the vector expression is restricted to the variables and eventual implicit local variables appearing within that vector expression. The boolean expression must be true during the entire vector expression. The boolean expression is called *Existence Condition* of the vector expression."
Vector expressions using the `vector_boolean_and` operator are called `vector_conditional_event` expressions. Scope and contents of such expressions are identical, as opposed to non-conditional `vector_complex_event` expressions, where the content is a subset of the scope.

Example:

\[(10 (A \& B) \rightarrow 10 (A \mid B)) \& !D) \quad // \quad (17)\]

\[// \text{ event pattern expressed by (17):} \]

\[// \quad A\&B \quad A|B \]

\[// \quad 1 \quad 1 \]

\[// \quad 0 \quad 1 \]

\[// \quad 0 \quad 0 \]

\[// \text{ event report without C, E:} \]

time \quad A \quad B \quad D \quad A\&B \quad A|B
\begin{array}{cccc}
0 & 0 & 1 & X & 0 & 1 \\
109 & 1 & 1 & 0 & 1 & 1 \\
258 & 1 & 0 & 0 & 0 & 1 \\
586 & 0 & 0 & 0 & 0 & 0 \\
643 & 1 & 0 & 0 & 0 & 1 \\
788 & 0 & 1 & 0 & 0 & 1 \\
915 & 1 & 1 & 0 & 1 & 1 \\
1062 & 1 & 1 & 0 & 1 & 1 \\
1395 & 1 & 0 & 0 & 0 & 1 \\
1640 & 0 & 0 & 1 & 0 & 0 \\
\end{array}

(17) contains the same `vector_complex_event` expression as (16). However, although (16) is not true at time 586, (17) is true at time 586, since the scope of observation is narrowed to “A“, “B“, “A\&B“, “A|B“ by the existence condition “!D“, which is statically true while the specified event sequence is observed.

Within and only within the narrowed scope of the `vector_conditional_event` expression, (17) can be considered equivalent to the following:

\[\begin{align*}
(10 (A & B) \rightarrow 10 (A | B)) & \& !D \\
\Rightarrow & (10 (A & B) \rightarrow 10 (A | B)) & (11 (!D) \rightarrow 11 (!D)) \\
\Rightarrow & 10 (A & B) & 11 (!D) \rightarrow 10 (A | B) & 11 (!D)
\end{align*}\]

The transformation consists of the following steps:

- **Step 1**: transform the boolean condition into a non-event. For example, “!D” becomes “11 (!D)”
- **Step 2**: left-extend the `vector_single_event` expression containing the non-event in order to match the length of the `vector_complex_event` expression. For example, “11 (!D)” becomes “11 (!D) -> 11 (!D)” to match the length of

---

1. An Existence Condition may also appear as annotation to a VECTOR object instead of appearing in the vector expression. The purpose is to enable recognition of existence conditions by application tools which can not evaluate vector expressions (e.g. static timing analysis tools). However, for tools that can evaluate vector expressions, there is no difference between existence condition as a co-factor in the vector expression or as annotation.
“10 (A & B) -> 10 (A | B)”

- Step 3: apply scalar multiplication rule for simultaneously occurring event sequences.

Thus a `vector_conditional_event` expression can be transformed into an equivalent `vector_complex_event` expression, but the change of scope must be kept in mind. In section 3.12.13 an operator will be introduced which will allow to express the change of scope in the vector expression language. This will make the transformation more rigorous.

Regardless of scope, the transformation from `vector_conditional_event` expression to `vector_complex_event` expression also provides means of detecting ill-specified `vector_conditional_event` expressions.

Example:

\[
(10 A \rightarrow 01 B \rightarrow 01 A) \& A
\]

\[
\equiv
\]

\[
10 A \& 11 A \rightarrow 01 B \& 11 A \rightarrow 01 A \& 11 A
\]

The first expression "10 A & 11 A" and the third expression "01 A & 11 A" within the `vector_complex_event` expression are contradictory. Hence the `vector_conditional_event` expression can never be true.

### 3.12.12 Alternative conditional event sequences

All `vector_binary` operators, in particular the `vector_or` operator, can be applied to `vector_conditional_event` expressions as well as to `vector_complex_event` expressions.

Consider again the event report:

<table>
<thead>
<tr>
<th>time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>109</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>258</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>573</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>586</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>643</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>788</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>915</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1062</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1395</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1640</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Concurrent alternative `vector_conditional_event` expressions can be paraphrased in the following way:

IF `<boolean_expression_1>` THEN `<vector_expression_1>`

OR IF `<boolean_expression_2>` THEN `<vector_expression_2>`

... OR IF `<boolean_expression_N>` THEN `<vector_expression_N>`

The conditions may be true within overlapping time windows and hence the vector expressions are evaluated concurrently. The `vector_boolean_and` operator and `vector_or` operator are used in ALF to describe such vector expressions.
Example:

\[
C \land (01 \ A \rightarrow 10 \ B) \mid !D \land (10 \ B \rightarrow 10 \ A) \mid E \land (10 \ B \rightarrow 10 \ C) \quad // \ (18)
\]

// Event pattern expressed by (18):
//  A   B   C
//  0   1   1
//  1   1   1
//  1   0   1

(18) is true at time 258 because of “\(C \land (01 \ A \rightarrow 10 \ B)\)“.

// Alternative event pattern expressed by (18):
//  A   B   D
//  1   1   0
//  1   0   0
//  0   0   0

(18) is also true at time 586 because of “\(!D \land (10 \ B \rightarrow 10 \ A)\)“.

// Alternative event pattern expressed by (18):
//  B   C   E
//  1   1   1
//  0   1   1
//  0   0   1

(18) is also true at time 573 because of “\(E \land (10 \ B \rightarrow 10 \ C)\)“.

Prioritized alternative \texttt{vector\_conditional\_event} expressions can be paraphrased in the following way:

\[
\text{IF } \text{<boolean\_expression\_1> THEN <vector\_expression\_1> ELSE IF } \text{<boolean\_expression\_2> THEN <vector\_expression\_2> ... ELSE IF } \text{<boolean\_expression\_N> THEN <vector\_expression\_N> (optional) ELSE <vector\_expression\_default>}
\]

Only the vector expression with the highest priority true condition is evaluated. The \texttt{vector\_boolean\_cond} operator and \texttt{vector\_boolean\_else} operator are used in ALF to describe such vector expressions.

Example:

\[
C \ ? \ (01 \ A \rightarrow 10 \ B) \ : \ !D \ ? \ (10 \ B \rightarrow 10 \ A) \ : \ E \ ? \ (10 \ B \rightarrow 10 \ C) \quad // \ (19)
\]

The prioritized alternative \texttt{vector\_conditional\_event} expression can be transformed into concurrent alternative \texttt{vector\_conditional\_event} expression as shown:

\[
C \ ? \ (01 \ A \rightarrow 10 \ B) \ : \ !D \ ? \ (10 \ B \rightarrow 10 \ A) \ : \ E \ ? \ (10 \ B \rightarrow 10 \ C) \\
===
C \land (01 \ A \rightarrow 10 \ B) \\
| !C \land !D \land (10 \ B \rightarrow 10 \ A) \\
| !C \land !(!D) \land E \land (10 \ B \rightarrow 10 \ C)
\]

(19) is true at time 258 because of “\(C \land (01 \ A \rightarrow 10 \ B)\)“, but not at time 586 because of higher priority “\(C\)“ while “\(!D \land (10 \ B \rightarrow 10 \ A)\)“, nor at time 573 because of higher priority “\(!D\)“ while “\(E \land (10 \ B \rightarrow 10 \ C)\)“. 
3.12.13 Change of scope within a vector expression

Conditions on vector expressions redefine the scope of vector expressions locally. The following definition allows to change the scope even within a part of a vector expression. For this purpose, the symbolic state “*” is introduced, which means “don’t care about events”. This is different from the symbolic state “?” which means “don’t care about state”. When state of a variable is “*”, arbitrary events may occur on that variable which are all disregarded.

- edge operator with “*” as next state.
  The variable to which the operator applies is no longer within the scope of the vector expression from now on.
- edge operator with “*” as previous state.
  The variable to which the edge operator applies is within the scope of the vector expression from now on.

As opposed to “?”,”*” stand for an infinite variety of possibilities.

Example:
Let “A” be a logic variable with the possible states “1”, “0”, “X”.

* 0 A ===
    00 A | 10 A | X0 A
    | 00 A -> 00 A | 10 A -> 00 A | X0 A -> 00 A
    | 01 A -> 10 A | 11 A -> 10 A | X1 A -> 10 A
    | 0X A -> X0 A | 1X A -> X0 A | XX A -> X0 A
    | 00 A -> 00 A -> 00 A | ...

0* A ===
    00 A | 01 A | 0X A
    | 00 A -> 00 A | 00 A -> 01 A | 00 A -> 0X A
    | 01 A -> 10 A | 01 A -> 11 A | 01 A -> 1X A
    | 0X A -> X0 A | 0X A -> X1 A | 0X A -> XX A
    | 00 A -> 00 A -> 00 A | ...

A vector expression with an infinite variety of possible event sequences cannot be directly matched with an event report. However, there are feasible ways to implement event sequence detection involving “*”. In principle there is a “static” and “dynamic” way. Let us name the parts of the vector expression separated by “*” sub-sequences of events.

- “Static” event sequence detection with “*”: The event report with all variables may be maintained, but certain variables will be masked for the purpose of detection of certain sub-sequences.
- “Dynamic” event sequence detection with “*”: The event report will contain the set of variables necessary for detection of a relevant sub-sequence. When such a sub-sequence is detected, the set of variables in the event report will change until the next sub-sequence is detected etc.

Examples:
\[01 \text{A} \rightarrow 1^* \text{B} \rightarrow 10 \text{C} \quad // (20)\]

// Event pattern expressed by (20):
// A  B  C
// 0  1  1
// 1  1  1
// 1  *  1
// 1  *  0

// pattern for 1st sub-sequence:
// A  B  C
// 0  1  1
// 1  1  1
// 1  *  1

// pattern for 2nd sub-sequence:
// A  B  C
// 1  *  1
// 1  *  0

Event report with masking relevant for (20):

\begin{verbatim}
time  A  B  C  D  E
  0   0  1  1  X  1
  109 1  1  1  0  1
  258 1  *  1  0  1 // detection of 1st sub-sequence
  573 1  *  0  0  1 // detection of 2nd sub-sequence
  586 0  0  0  0  1
  643 1  0  0  0  1
  788 0  1  1  0  1
  915 1  1  1  0  1
 1062 1  *  1  0  0 // detection of 1st sub-sequence
 1395 1  *  0  0  0 // detection of 2nd sub-sequence
 1640 0  0  0  1  0
\end{verbatim}

(20) is true at time 573 and at time 1395. The first sub-sequence "01 \text{A} \rightarrow 1^* \text{B}" is detected at time 258, since * maps to any state. From time 258 onwards, B is masked. The second sub-sequence "10 \text{C}" is detected at time 573. From time 573 onwards, B is unmasked. The first sub-sequence is detected again at time 1062. The second sub-sequence is detected again at time 1395.

\[01 \text{A} \& 1^* \text{E} \rightarrow 10 \text{C} \quad // (21)\]

// Event pattern expressed by (21):
// A  C  E
// 0  1  1
// 1  1  *
// 1  0  *

// pattern for 1st sub-sequence:
// A  C  E
// 0  1  1
// 1  1  *

// pattern for 2nd sub-sequence:
// A  C  E
// 1  1  *
// 1  0  *
Event report with masking relevant for (21):

<table>
<thead>
<tr>
<th>time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>109</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>258</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>573</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>586</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>643</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>788</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>915</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1062</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1395</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1640</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(21) is true at time 1395. The first sub-sequence “01 A & 1* E“ is detected at time 109. From time 109 onwards, E is masked. The event on B at time 258 aborts continuation of the detection process and triggers restart from the beginning. The first sub-sequence is detected again at time 915. From time 915 onwards, E is masked. The event at time 1062 is therefore out of scope. The second sub-sequence “10 C“ is detected at time 1395.

01 A -> *1 B -> 10 B & 10 C // (22)

// Event pattern expressed by (22):
// A  B  C
// 0  *  1
// 1  *  1
// 1  1  1
// 1  0  0

// pattern for 1st sub-sequence:
// A  B  C
// 0  *  1
// 1  *  1

// pattern for 2nd sub-sequence:
// A  B  C
// 1  *  1
// 1  1  1
// 1  0  0

Event report with masking relevant for (22):

<table>
<thead>
<tr>
<th>time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>109</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>258</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>573</td>
<td>1</td>
<td>*</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>586</td>
<td>0</td>
<td>*</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>643</td>
<td>1</td>
<td>*</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>788</td>
<td>0</td>
<td>*</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>915</td>
<td>1</td>
<td>*</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1062</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1395</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1640</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Modeling with Vector Expressions Library Format Specification

(22) is true at time 1395. The first sub-sequence “01 A” is detected at time 109. Therefore B is unmasked. Since B=0 at time 258, the attempt to detect the second sub-sequence is aborted, and the detection process restarts from the beginning. The first sub-sequence “01 A” is detected again at time 109. The second sub-sequence “*1 B -> 10 B & 10 C” is detected at time 1395.

\[
01 A \rightarrow 1? A \& 0* B \& 1* E \rightarrow 10 C
\] // (23)

// Event pattern expressed by (23):
// A B C E
// 0 0 1 1
// 1 0 1 1
// 1 * 1 *
// 1 * 0 *

// pattern for 1st sub-sequence:
// A B C E
// 0 0 1 1
// 1 0 1 1
// ? * 1 *

// pattern for 2nd sub-sequence:
// A B C E
// ? * 1 *
// ? * 0 *

Event report with masking relevant for (23):

<table>
<thead>
<tr>
<th>time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>109</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>258</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>573</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>586</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>643</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
| 788  | 0 | * | 1 | 0 | * // detection of 1st sub-sequence
| 915  | 1 | * | 1 | 0 | * // abort
| 1062 | 1 | 1 | 1 | 0 | 0 |
| 1395 | 1 | 0 | 0 | 0 | 0 |
| 1640 | 0 | 0 | 0 | 1 | 0 |

(23) is not true at any time. The first sub-sequence is detected at time 788. The event at time 915 does not match the expected second sub-sequence.

### 3.12.14 Sequences of conditional event sequences

The introduction of the symbol “*” allows to describe the scope of a vector expression directly in the vector expression language. This is particularly useful for sequences of vector_conditional_event expressions.

Let us reuse (17) as example:

\[
(10 (A & B) \rightarrow 10 (A | B)) & !D
\]

The scope the sample event report contains contain the variables A, B, C, D, E. The vector_conditional_event expression (17) contains only the variables A, B, D and the implicit local variables A&B, A|B. Therefore the global variables C, E are out of scope within (17). The implicit local variables A&B, A|B are in scope within and only within (17).
Now let us consider a sequence of vector_conditional_event expressions, where variables move in and out of scope. With the following formalism it is possible to transform such a sequence into an equivalent vector_complex_event expression, allowing for a change of scope within each vector_conditional_event expression.

\[
\langle \text{vector\_conditional\_event}\#i \rangle \rightarrow \ldots \rightarrow \langle \text{vector\_conditional\_event}\#N \rangle
\]

where

\[
\langle \text{vector\_conditional\_event}\#i \rangle = \langle \text{vector\_complex\_event}\#i \rangle \& \langle \text{boolean\_expression}\#i \rangle \quad /1 \leq i \leq N
\]

The principle is to decompose each vector_conditional_event expression into a sequence of three vector expressions prefix, kernel, and postfix and then to reassemble the decomposed expressions.

\[
\langle \text{vector\_conditional\_event}\#i \rangle = \langle \text{prefix}\#i \rangle \rightarrow \langle \text{kernel}\#i \rangle \rightarrow \langle \text{postfix}\#i \rangle \quad /1 \leq i \leq N
\]

- **Step 1:** Define the prefix for each vector_conditional_event expression.
  The prefix is a vector_event expression introducing all implicit local variables.

  Example:
  
  \[
  \,* (A&B) \& \,* (A|B)
  \]

- **Step 2:** Define the kernel for each vector_conditional_event expression.
  The kernel is the vector_complex_event expression equivalent to the vector_conditional_event expression.

  \[
  \langle \text{vector\_complex\_event}\#i \rangle \& \langle \text{boolean\_expression}\#i \rangle = \langle \text{vector\_complex\_event}\#i \rangle \& (11 \langle \text{boolean\_expression}\#i \rangle \rightarrow \ldots \rightarrow 11 \langle \text{boolean\_expression}\#i \rangle)
  \]

  The kernel may consist of one or several alternative vector_event_sequence expressions. Within each vector_event_sequence expression, the same set of global variables are pulled out of scope at the first vector_event expression and pushed back in scope at the last vector_event expression.

  Example:

  \[
  \,* C \& \,* E \quad // \text{global variables out of scope} \\
  \& 10 (A \& B) \& 11 (!D) \rightarrow 10 (A \mid B) \& 11 (!D) \\
  \& \,* C \& \,* E \quad // \text{global variables back in scope}
  \]

- **Step 3:** Define the postfix for each vector_conditional_event expression.
  The postfix is a vector_event expression removing all implicit local variables.

  Example:

  \[
  \,* (A&B) \& \,* (A|B)
  \]

- **Step 4:** join the subsequent vector_complex_event expressions with the vector_and
operator between prefix#i+1 and kernel#i and also between postfix#i and kernel#i+1.

.. <vector_conditional_event#i> -> <vector_conditional_event#i+1> ..

=== .. <prefix#i>
> <postfix#i-1> & <kernel#i> & <prefix#i+1>
> <postfix#i> & <kernel#i+1> & <prefix#i+2>
> <postfix#i+1> ..

Complete example:

(10 (A & B) -> 10 (A | B)) & !D

===

*? (A&B) & *? (A|B)

-> ?* C & ?* E

& 10 (A & B) & 11 (!D) -> 10 (A | B) & 11 (!D)

& *? C & *? E

-> ?* (A&B) & ?* (A|B)

Note that the in-and-out-of-scope definitions for global variables are within the kernel, whereas the in-and-out-of-scope definitions for global variables are within prefix and postfix. In this way, the resulting vector_complex_event expression contains the same uninterrupted sequence of events as the original sequence of vector_conditional_event expressions.

3.12.15 Incompletely specified event sequences

So far the vector expression language has provided support for completely specified event sequences and also the capability to put variables temporarily in and out of scope for event observation. As opposed to changing the scope of event observation, incompletely specified event sequences require continuous observation of all variables while allowing the occurrence of intermediate events between the specified events. The following operator is introduced for that purpose:

- vector_followed_by, also called “followed-by operator” using the symbol “~”.

The “~” operator is the separator between consecutively occurring events with possible unspecified events in-between.

Detection of event sequences involving “~” requires detection of the sub-sequence before “~”, setting a flag, detection of the sub-sequence after “~” and clearing the flag.

This can be illustrated with our sample event report:

<table>
<thead>
<tr>
<th>time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>025</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>057</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>058</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>064</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>078</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>091</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>106</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>139</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>164</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Example:

```
01 A -> 10 C  // (24)
// as opposed to previous example (5): 01 A -> 10 C
```

(24) is true at time 573 because of “01 A” at time 109 and “10 C” at time 573. It is true again at time 1395 because of “01 A” at time 643 and “10 C” at 1395. On the other hand, (5) is never true because there are always events in-between “01 A” and “10 C”.

Vector expressions consisting of `vector_event` expressions separated by “->” or by “->” are called `vector_event_sequence` expressions, using the same syntax rules for the two different `vector_followed_by` operators. Consequently, all vector expressions involving `vector_event_sequence` expressions and `vector_binary` operators are called `vector_complex_event` expressions.

However, only a subset of the semantic transformation rules can be applied to vector expressions containing “->”.

**Associative rule applies for both “->” and “->”:**

```
(01 A ~> 01 B) ~> 01 C === 01 A ~> (01 C ~> 01 B ~> 01 C)
(01 A -> 01 B) -> 01 C === 01 A -> (01 C -> 01 B -> 01 C)
(01 A ~> 01 B) -> 01 C === 01 A ~> (01 C ~> 01 B -> 01 C)
(01 A -> 01 B) ~> 01 C === 01 A -> (01 C -> 01 B ~> 01 C)
```

**Distributive rule applies for both “->” and “->”:**

```
(01 A | 01 B) -> 01 C === 01 A ~> 01 C | 01 B -> 01 C
(01 A | 01 B) -> 01 C === 01 A ~> 01 C | 01 B -> 01 C
(01 A | 01 B) -> 01 C === 01 A ~> 01 C | 01 B -> 01 C
```

**Scalar multiplication rule applies only for “->”. The transformation involving “~>” is more complicated.**

```
(01 A -> 01 B) & (01 C -> 01 D)
=== (01 A & 01 C) -> (01 B & 01 D)
(01 A ~> 01 B) & (01 C -> 01 D)
=== (01 A & 01 C) ~> (01 B & 01 D)
| 01 A ~> 01 C -> (01 B & 01 D)
(01 A ~> 01 B) & (01 C ~> 01 D)
=== (01 A & 01 C) ~> (01 B & 01 D)
| 01 A ~> 01 C ~> (01 B & 01 D)
| 01 C ~> 01 A ~> (01 B & 01 D)
```

Transformation of `vector_conditional_event` expressions into `vector_complex_event` expressions applies only for “->”.

```
(01 A -> 01 B) & C
=== 01 A & 11 C -> 01 B & 11 C
(01 A ~> 01 B) & C
--- 01 A & 11 C ~> 01 B & 11 C
```

Since the “~>” operator allows intermediate events, there is no way to express the continuously true condition “C”.
3.12.16  Well-specified vector expressions

By defining semantics for

- alternative vector_event_sequence expressions

and establishing calculation rules for

- transforming vector_complex_event expressions into alternative vector_event_sequence expressions

and for

- transforming alternative vector_conditional_event expressions into alternative vector_complex_event expressions,

semantics are now defined for all vector expressions.

As we have seen for vector_conditional_event expressions, the calculation rules also provide means to determine whether a vector expression is well-specified or ill-specified. An ill-specified vector expression is contradictory in itself and can therefore never be true.

Once a vector expression is reduced to a set of alternative vector_event_sequence expressions, two criteria define whether a vector expression is well-defined or not.

- Compatibility between subsequent events on the same variable:
  Next state of earlier event must be compatible with previous state of later event. This check applies only if no “~>” operator is found between the events.

- Compatibility between simultaneous events on the same variable:
  Both previous and next state of both events must be compatible. Such events commonly occur as intermediate calculation results within vector expression transformation.

The following compatibility rules apply:

- “?” is compatible with any other state. If the other state is “*”, the resulting state is “?”. Otherwise, the resulting state is the other state.
- “*” is compatible with any other state. Resulting state is the other state.
- Any other state is only compatible with itself.

Examples:

- 01 A -> 01 B -> 10 A
  The next state of “01 A” is compatible with the previous state of “10 A”

- 0X A -> 01 B -> 10 A
  The next state of “0X A” is not compatible with the previous state of “10 A”

Compatibility check does not apply, since intermediate events are allowed.

- 01 A & 10 A
  Both previous and next state of “A” are contradictory and result in an impossible event.

- ?1 A & 1? A
  Both previous and next state of “A” are compatible and result in the non-event “11 A”.

Section 4
Applications

This section shows various examples of library elements 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 function of the cell can be modeled either as a statetable or as behavior using a boolean equation.

Modeling a NAND gate using truth table:

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

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

Modeling a NAND gate using boolean expression:

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

    FUNCTION {
        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;}
   FUNCTION {
      .../* One of the descriptions 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 {
   @(01 cp && cd) {q = d;}
   @(!cd) {q = 0;}
}

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;}
   FUNCTION {
      .../* One of the descriptions 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 the derivation of new cells from other cells. Below, a D-Flipflop with asynchronous clear is derived from a predefined ALF_FLIPFLOP with asynchronous set and clear (see Section 4.1.2):

\[
\text{BEHAVIOR} \{ \\
\quad \text{ALF_FLIPFLOP} \{ \text{CLOCK}=cp; \text{D}=d; \text{Q}=q; \text{SET}=\text{\texttt{b0}}; \text{CLEAR}=!cd; \} \\
\}
\]

### 4.2.2 JK-flipflop

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

CELL jk_flipflop {
    PIN cp {DIRECTION=input; SIGNALTYPE=\texttt{clock}; POLARITY=rising\_edge;} 
    PIN j {DIRECTION=input;} 
    PIN k {DIRECTION=input;} 
    PIN q {DIRECTION=output;} 
    FUNCTION {
        .../* One of the descriptions below go here */
    }
}

**Explicit description:**

\[
\text{BEHAVIOR} \{ \\
\quad d = \\
\quad \begin{cases} 
        \{!j \&\& k \} & ? 0 : \\
        \{ j \&\& !k \} & ? 1 : \\
        \{ j \&\& k \} & ? !(q) : \\
        \{!j \&\& !k \} & ? (q) : \\
        \text{\texttt{bx \_}} \\
    \end{cases} \\
\quad @(01 \ cp) \{ q = d; \} \\
\}
\]

**Use of primitives (using predefined ALF_MUX and ALF_FLIPFLOP):**

\[
\text{BEHAVIOR} \{ \\
\quad \text{ALF_MUX} \{ \text{Q}=d; \text{D}[0]=j; \text{D}[1]=!k; \text{S}=q; \} \\
\quad \text{ALF_FLIPFLOP} \{ \text{CLOCK}=cp; \text{D}=d; \text{Q}=q; \text{SET}=\text{\texttt{b0}}; \text{CLEAR}=\text{\texttt{b0}}; \} \\
\}
\]

**Use of a hybrid form (boolean expressions within primitive instantiation):**

\[
\text{BEHAVIOR} \{ \\
\quad \text{ALF_FLIPFLOP} \{ \text{CLOCK}=cp; \text{D}=(q \ ? \ !k \ : \ j); \text{Q}=q; \text{SET}=\text{\texttt{b0}}; \text{CLEAR}=\text{\texttt{b0}}; \} \\
\}
\]

**Use of truth table:**

\[
\text{STATETABLE} \{ \\
\quad cp \ j \ k \ q : (q) ; \\
\quad 01 \ 0 \ 0 \ ? : (q) ; \\
\quad 01 \ 0 \ 1 \ ? : 0 ; \\
\quad 01 \ 1 \ 0 \ ? : 1 ; \\
\quad 01 \ 1 \ 1 \ ? : (!q); \\
\quad 1? \ ? \ ? \ : (q) ; \\
\quad 0? \ ? \ ? \ : (q) ; \\
\}
\]
4.2.3  D-Flipflop with synchronous load and clear

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

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

Explicit description:

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

Use of primitives:

```plaintext
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:

```plaintext
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;}
    FUNCTION { /* fill in BEHAVIOR */ }
}

Explicit description:

```plaintext
BEHAVIOR {
    @(!cd) {q = 0;}
    @(!sd && cd) {q = 1;}
    @(01 cp && cd && sd) {q = sel? d1 : d2;}
}
```
More efficient description can be created using if-then-else style:

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

Use of primitive:

```alphanum
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.

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

Explicit description:

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

Use of primitive:

```alphanum
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 a don’t care situation.

```alphanum
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;}
    FUNCTION { ... }
}
```

Explicit description:

```alphanum
BEHAVIOR {
    @(sn) {q = 'b1; qn = !rn;}
    @ (!rn) {qn = 'b1; q = !sn;}
}
Use of primitive:

```plaintext
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.

```plaintext
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.

```plaintext
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);
        }
    }
    /* fill in timing and power data for SCAN_ND4 cell */
}
4.2.9 Scan Flipflop

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

```plaintext
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; OFFSETSTATE = 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 { // flipflop_d is an implicitly defined internal pin
            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;}
} // H03 and H01 have no corresponding pin in ALF_FLIPFLOP
...
}
```
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;
      }
    }
  } 
}
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 slewrate, 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}
                }
            }
        }
    }
}
```
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 = <input_pin>;
      out_pin = <output_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 = <input_pin>;
      out_pin = <output_pin>;
    }
    TABLE <slewrate_data>
  }
  ENERGY {
    UNIT = pJ;
    std_header_2d {
      in_pin = <input_pin>;
      out_pin = <output_pin>;
    }
    TABLE <energy_data>
  }
}
Now only the delay, slewrate 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:

```c
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
    }
    slewrate_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.

```c
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 ramp-time as variables, the vectors for power model require 1-dimensional tables with input ramp-time as variable, and the vectors for time constraints require 2-dimensional tables with ramp-time 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";
        }
    }

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

    HOLD {
        /* fill in arithmetic model for hold time constraint */
        VIOLATION {
            BEHAVIOR (q = 'bx;)
            MESSAGE_TYPE = error;
            MESSAGE = "hold violation 01 d <-> 10 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:

```plaintext
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:

```plaintext
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.

```plaintext
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 - 25)*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 - 25)*<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 express voltage, temperature and delay with the derating case as an independent variable:

```
VOLTAGE {
    HEADER { DERATE_CASE { TABLE {nom bccom wcmil} } }
    TABLE {3.3 3.5 2.8}
}
TEMPERATURE {
    HEADER { DERATE_CASE { TABLE {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)
    }
}
```

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:

```plaintext
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 that was characterized using an average measurement (as opposed to RMS or peak measurements). 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.

```plaintext
PIN data_out {
    DIRECTION = output; CAPACITANCE = 0.002;
    LIMIT {CAPACITANCE {MAX = 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:

```plaintext
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 a fall-back 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;
```
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 =
    }
  }
}
```
A different model can be used for transmission-gate type of buffers, which also passes the high impedance state from input to output.

```plaintext
(e & a) ? 'b1:
(e & !a) ? 'b0:
(!e) ? 'bz:
'bx;

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.

```plaintext
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 `BEHAVIOR {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.

```plaintext
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 Bushholder

A *bushholder* 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.

```plaintext
CELL bushholder {
    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.

```plaintext
CELL bushholder_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.

```plaintext
WIRE small_wire {
    CAPACITANCE {
        UNIT = pF;
        HEADER {
            CONNECTIONS {
                TABLE (2 3 4 5)
            }
        }
    }
}
```
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 {UNIT=1e-6; MAX=25;}}
    ...
}

WIRE large_wire {
    LIMIT {AREA {UNIT=1e-6; MIN=25; MAX=100;}}
    ...
}
If the area covering the routing space is smaller than $25\text{mm}^2$, the small\_wire model will be chosen. If the area covering the routing space is between $25\text{mm}^2$ and $100\text{mm}^2$, the large\_wire model is chosen. The unit for area is $1\text{mm}^2$.

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,

```plaintext
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:

```plaintext
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:

```plaintext
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}
  GROUP /* {0:31}
  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) {
  SETUP {
    VIOLATION {
      BEHAVIOR { data_store[*] = 'bxxxxxxxx'; }
      MESSAGE_TYPE = error;
      MESSAGE = "setup violation: changing 'adr_write' -> rising 'enable_write', memory -> 'X';"
    }
    FROM { pin = adr_write; }
    TO { pin = enable_write; }
    /* fill in header, table or equation */
  }
}

VECTOR (10 enable_write <-> ! adr_write[all]) {
  HOLD {
    VIOLATION {
      BEHAVIOR { data_store[*] = 'bxxxxxxxx'; }
      MESSAGE_TYPE = error;
      MESSAGE = "hold violation: falling 'enable_write' -> changing 'adr_write', memory -> 'X';"
    }
    FROM { pin = enable_write; }
    TO { pin = adr_write; }
    /* fill in header, table or equation */
  }
}

VECTOR (! data_write[byte] <-> 10 enable_write) {
  SETUP {
    VIOLATION {
      BEHAVIOR { data_store[adr_write] = 'bxxxxxxxx'; }
      MESSAGE_TYPE = error;
      MESSAGE = "setup violation: changing 'data_write' -> falling 'enable_write', memory[adr_write] -> 'X';"
    }
    FROM { pin = data_write; }
    TO { pin = enable_write; }
    /* fill in header, table or equation */
  }
}

HOLD {
  VIOLATION {

```
BEHAVIOR ( data_store[adr_write] = 'bxxxxxxxx; )
MESSAGE_TYPE = error;
MESSAGE = "hold violation: falling 'enable_write' -> changing 'data_write',
memory[adr_write] -> 'X';"
}
FROM ( pin = enable_write; )
TO ( pin = data_write; )
/* fill in header, table or equation */
}
VECTOR (01 enable_write -> 10 enable_write) (
PULSEWIDTH {
VIOLATION {
MESSAGE_TYPE = error;
MESSAGE = "pulsewidth violation: high 'enable_write';"
}
PIN = enable_write;
/* fill in header, table or equation */
}
VECTOR (10 enable_write -> 01 enable_write) (
PULSEWIDTH {
VIOLATION {
MESSAGE_TYPE = error;
MESSAGE = "pulsewidth violation: low 'enable_write';"
}
PIN = enable_write;
/* fill in header, table or equation */
}
)

The energy consumption for each operation depends on the number of switching bits of the bus.
Therefore, the model for power inside a particular vector may look like this:
VECTOR (?! data_write && 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;}
@(?! ? addr_write && enable_write)
{data_store[\*] = 'bxxxxxx;}
}
### 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;
            }
        }
    }
}
```

### 4.9.4 Annotation for pins of a bus

Annotations of numeric values to a bus apply to the total bus, not to each individual pin.

Example:

```
PIN [1:4] my_bus_pin {
    CAPACITANCE = 0.04 ;
}
```

The total bus pin capacitance is 0.4, the capacitance values on each individual pin are not defined.
The individual pin capacitance can be defined as follows:

```plaintext
PIN [1:4] my_bus_pin {
    CAPACITANCE c1 = 0.01 { PIN = my_bus_pin[1]; }
    CAPACITANCE c2 = 0.01 { PIN = my_bus_pin[2]; }
    CAPACITANCE c3 = 0.01 { PIN = my_bus_pin[3]; }
    CAPACITANCE c4 = 0.01 { PIN = my_bus_pin[4]; }
}
```

### 4.9.5 Skew for simultaneously switching signals on a bus

Vectors with simultaneously switching bits on a bus may contain a specification of the allowed skew in order to be still considered as simultaneously switching bits.

Example:

```plaintext
PIN [1:3] address;
VECTOR (?! address )
    SKEW {
        PIN = address;
        /* fill in data */
    }
}
```

SKEW applied to a bus pin is the maximal allowed time window between the earliest and latest edge within simultaneously switching signals of a bus.

The multiple value annotation feature allows the definition of a group of pins equivalent to a bus for SKEW modeling in the following way:

```plaintext
PIN A;
PIN [1:4] B;
VECTOR (?! A && ?! B)
    SKEW { PIN { A B[2:3] } }
}
```

SKEW applies to the group of pins A, B[2], B[3]. Note that the following is semantically different, since this would result in expansion of each object where the group is instantiated:

```plaintext
PIN A;
PIN [1:4] B;
VECTOR (?! my_group)
    SKEW { PIN = my_group; }
}```
The expansion yields the following:

```
PIN A;
PIN [1:4] B;
VECTOR (?! A)
   SKEW { PIN = A ; }
} VECTOR (?! B[2])
   SKEW { PIN = B[2] ; }
} VECTOR (?! B[3])
   SKEW { PIN = B[3] ; }
}
```

See Section 4.15.2.7 for definition of SKEW for scalar pins.

## 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

\[
\begin{align*}
  dout(t) &= state(t) + b1 \times state(t-1) + b2 \times state(t-2) \\
  state(t) &= din(t) - a1 \times state(t-1) - a2 \times state(t-2)
\end{align*}
\]

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)
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.

```quiz
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}}
        }
    }
```
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`.

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.

```plaintext
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}
    }
    ...
}
```

### 4.13 Signal Integrity

#### 4.13.1 I/V curves

I/V curves describe the driven or drawn current at a pin as a function of the voltage at one or several pins. The following example describes the output current of a buffer as a function of the input and output voltage with a 2-dimensional lookup table.

```plaintext
CELL simple_buffer {
    PIN z { DIRECTION = output; }
    PIN a { DIRECTION = input; }
    // current @ z dependent on voltage @ z and @ a
    CURRENT {
        PIN = z;
        UNIT = ma;
        HEADER {
            VOLTAGE vout {
                PIN = z;
                TABLE { 0.0 0.5 1.0 1.5 2.0 2.5 3.0 }
            }
            VOLTAGE vin {
                PIN = a;
                TABLE { 0.0 1.0 2.0 3.0 }
            }
        }
        TABLE {
            5.0 5.0 4.8 4.2 3.2 1.6 0.0
            2.5 1.5 0.2 -0.4 -1.8 -2.7 -3.5
            1.2 0.1 -1.3 -1.9 -2.5 -3.8 -4.6
            0.0 -2.0 -3.8 -4.7 -5.5 -6.2 -6.3
        }
    }
    // fill in function, vector and other stuff
}
```
An equation can also be used instead of a lookup table, for example:

```
CURRENT {
    PIN = z;
    UNIT = ma;
    HEADER {
        VOLTAGE vout {
            PIN = z;
        }
        VOLTAGE vin {
            PIN = a;
        }
    }
    EQUATION {
        (1 - exp(6.3 - 2.4*vout))*exp(0.9 - 0.3*vin) \\
        - (1 - exp(3.2*vout))*exp(0.3*vin)
    }
}
```

A buffer may have programmable drive strength controlled by the state of additional input pins. State-dependent I/V curves can be described by vector-specific CURRENT models.

```
CELL programmable_drive_strength_buffer {
    PIN z { DIRECTION = output; }
    PIN a { DIRECTION = input; }
    // control pins for drive strength
    PIN p1 { DIRECTION = input; }
    PIN p2 { DIRECTION = input; }
    VECTOR (!p1 && !p2) {
        CURRENT {
            // fill in the model
        }
    }
    VECTOR (!p1 &&  p2) {
        CURRENT {
            // fill in the model
        }
    }
    VECTOR ( p1 && !p2) {
        CURRENT {
            // fill in the model
        }
    }
    VECTOR ( p1 &&  p2) {
        CURRENT {
            // fill in the model
        }
    }
}
```

Note that it is also possible to describe other analog cell characteristics, state-dependent or state-independent, for instance voltage versus voltage, frequency versus voltage, current versus temperature etc. in the same way.
4.13.2 Driver resistance

Driver resistance is used to model the transient behavior of signals especially for crosstalk. The drivers are modeled by voltage sources and driver resistances, as illustrated below:

![Figure 4-1: Modeling driver resistance](image)

The purpose is to use linear circuit theory for the analysis of multiple drivers interacting with coupled RC-interconnect networks. In reality, the drivers have non-linear resistance. The linear resistance is a model of the non-linear resistance with the best-fitting linear resistance. Therefore the driver resistance is state-dependent and eventually also load-and slewrate dependent, since for different states and different ranges of load and slewrates the best-fitting value for driver resistance is different.

The following example shows a buffer featuring different driver resistance values for static low and high states, and tables of slewrate and load-dependent transient driver resistance values for rise and fall transitions.

```plaintext
cell simple_buffer {
    PIN z { DIRECTION = output; }
    PIN a { DIRECTION = input; }
    // state-dependent static driver resistance
    VECTOR (!z) {
        RESISTANCE = 0.7k { PIN = z; }
    }
    VECTOR (z) {
        RESISTANCE = 1.2k { PIN = z; }
    }
    // slew & load dependent transient driver resistance
    VECTOR (01 a -> 01 z) {
        RESISTANCE {
            PIN = z;
            UNIT = kohm;
            HEADER {
                CAPACITANCE {
                    PIN = z;
                    UNIT = pfarad;
                    TABLE { 0.1  0.4  1.6 }
                }
            }
        }
    }
}
```
The transient driver resistance can also be state-dependent, for example in the case of a buffer with programmable drive-strength.

CELL programmable_drive_strength_buffer {
    PIN z { DIRECTION = output; }
    PIN a { DIRECTION = input; }
    // control pins for drive strength
    PIN p1 { DIRECTION = input; }
    PIN p2 { DIRECTION = input; }
    // state-dependent static driver resistance
    VECTOR (!z && !p1 && !p2) {
        RESISTANCE = 0.7k { PIN = z; }
    }
    VECTOR (!z && !p1 &&  p2) {
        RESISTANCE = 0.6k { PIN = z; }
    }
    VECTOR (!z &&  p1 && !p2) {
        RESISTANCE = 0.5k { PIN = z; }
    }
    VECTOR (!z &&  p1 && !p2) {
        RESISTANCE = 0.4k { PIN = z; }
    }
    VECTOR (z && !p1 && !p2) {
        RESISTANCE = 1.2k { PIN = z; }
    }
    VECTOR (z && !p1 &&  p2) {
        RESISTANCE = 0.8k { PIN = z; }
    }
    VECTOR (z &&  p1 && !p2) {
        RESISTANCE = 1.1k { PIN = z; }
    }
    VECTOR (z &&  p1 && !p2) {
        RESISTANCE = 0.8k { PIN = z; }
    }
}
RESISTANCE = 1.0k { PIN = z; }

VECTOR (z && p1 && !p2) {
    RESISTANCE = 0.8k { PIN = z; }
}

VECTOR (z && p1 && p2) {
    RESISTANCE = 0.6k { PIN = z; }
}

// slew & load and state dependent transient driver resistance
VECTOR (01 a -> 01 z && !p1 && !p2) {
    RESISTANCE {
        // fill in the model
    }
}

VECTOR (01 a -> 01 z && !p1 && p2) {
    RESISTANCE {
        // fill in the model
    }
}

VECTOR (01 a -> 01 z && p1 && !p2) {
    RESISTANCE {
        // fill in the model
    }
}

VECTOR (01 a -> 01 z && p1 && p2) {
    RESISTANCE {
        // fill in the model
    }
}

VECTOR (10 a -> 10 z && !p1 && !p2) {
    RESISTANCE {
        // fill in the model
    }
}

VECTOR (10 a -> 10 z && !p1 && p2) {
    RESISTANCE {
        // fill in the model
    }
}

VECTOR (10 a -> 10 z && p1 && !p2) {
    RESISTANCE {
        // fill in the model
    }
}

VECTOR (10 a -> 10 z && p1 && p2) {
    RESISTANCE {
        // fill in the model
    }
}

}

The model for transient driver resistance has the same form as a slewrate and load dependent model for delay. Voltage, process, and temperature dependent driver resistance can also be modeled in the same way as voltage, process, and temperature-dependent delay.
4.14 Resistance and Capacitance on a Pin

4.14.1 Self-Resistance and Capacitance on Input Pin

A pin resistance is a resistance inside a PIN object.

```plaintext
PIN <pin_identifier> {
    DIRECTION = input;
    RESISTANCE = <resistance_number>;
    CAPACITANCE = <capacitance_number>;
}
```

The pin resistance is in series with the pin capacitance, as shown in figure 4-2:

![Figure 4-2: Resistance and capacitance on a pin](image1)

4.14.2 Pullup and Pulldown Resistance on Input Pin

A pullup or pulldown resistance or a combination of both on an input pin can be described as follows:

```plaintext
PIN <pin_identifier> {
    DIRECTION = input;
    PULL = < up | down | both > {
        VOLTAGE = <voltage_number>;
        RESISTANCE = <resistance_number>;
    }
}
```

The pullup/pulldown resistance is in series with a clamp voltage, as shown in figure 4-3:

![Figure 4-3: Pullup or pulldown resistance](image2)
In the case of a pullup/pulldown combination, the resistance and voltage represent the Thevenin equivalent resistance and voltage, respectively, as shown in figure 4-4:

$$R_{\text{Thevenin}} = \frac{R_1 \cdot R_0}{R_1 + R_0}$$

$$(V_1 \cdot R_0 + V_0 \cdot R_1) / (R_1 + R_0)$$

**Figure 4-4: Thevenin equivalent resistance**

### 4.14.3 Pin and Load Resistance and Capacitance on Output Pin

The driver resistance (see 4.13.2) can also be represented as pin capacitance of an output pin, in case there is no state dependency.

```plaintext
PIN <pin_identifier> {
    DIRECTION = output;
    CAPACITANCE = <capacitance_number>;
    RESISTANCE {
        RISE = <rise_resistance_number>;
        FALL = <rise_resistance_number>;
    }
}
```

Please note the distinction of capacitance and resistance of the pin itself and capacitance and resistance applied as load to the pin in the following schematic. The load capacitance and resistance would be specified in a characterization vector (see Section 4.3).

See the following schematic for driver signal, pin and load resistance and capacitance:

**Figure 4-5: Resistance and capacitance on output pin**
4.15 ALF/SDF cross reference

This section provides a cross reference between the representation of timing data in ALF and SDF. In general, ALF is used as a characterization library, which is the input to a delay calculator, whereas SDF is the output from a delay calculator. Therefore ALF typically contains tables or equations (i.e. arithmetic models) for timing data whereas SDF contains a discrete set of data in fixed format. However, in an ALF representation of timing shells for cores, which are typically represented in SDF today, the ALF library would contain the same data as the SDF.

The specification of the stimulus for a particular timing measurement (i.e. the timing diagram) is pertinent to both ALF and SDF. In ALF, timing diagrams are directly described in the vector expression language, and the timing measurements are always specified in relation to a particular timing vector. In SDF, timing diagrams are partly described in the language and partly implied by the keyword for timing measurements. Therefore SDF needs a larger set of keywords than ALF for the same description capability.

4.15.1 SDF delays

4.15.1.1 SDF DELAY for IOPATH and INTERCONNECT

DELAY is a measurement of the time needed for a signal to travel from one port to another port. In ALF, delay measurements are described in a uniform language, independent of whether A and Z are the input and output port of the same cell, respectively, or A and Z are the driver and receiver connected to the same net, or A and Z are both outputs of a cell. Therefore the SDF keywords IOPATH and INTERCONNECT have no counterpart in ALF.

VECTOR (01 A -> 01 Z) {
    DELAY {
        FROM {PIN = A;}
        TO {PIN = Z;}
        /* fill in data */
    }
}

![Figure 4-6: Measurement of SDF IOPATH or INTERCONNECT delay](image)

The ALF VECTOR describes the sequence of events shown in figure 4-6

*rising edge at A followed by rising edge at Z.*
The FROM and TO pin annotations define the sense of measurement for DELAY.
As opposed to SDF where input ports of an IOPATH may have an edge specification and output ports may not, the vector expression language in ALF always contains the specification of the edge:

\[
\text{rising edge = "01", falling edge = "10", any edge = "?!".}
\]

### 4.15.1.2 SDF PATHPULSE

PATHPULSE in SDF defines the smallest pulse that may appear at a port in form of

1. a full-swing pulse
2. a pulse to X.

The equivalent model in ALF uses two vectors in conjunction with the keyword PULSEWIDTH. ¹

The ALF keywords are of more general use than the SDF PATHPULSE keyword, which is just for one specific use.

```
VECTOR (01 Z -> 10 Z) {
    PULSEWIDTH {
        PIN = Z;
        /* fill in data */
    }
}
```

Figure 4-7: Measurement of SDF PATHPULSE full-swing

The ALF VECTOR above describes the sequence of events

\[
\text{rising edge at Z followed by falling edge at Z.}
\]

The smallest possible full-swing pulse applies at pin Z.

```
VECTOR ('b0' 'bX' Z -> 'bX' 'b0' Z) {
    PULSEWIDTH {
        PIN = Z;
        /* fill in data */
    }
}
```

1. The same keyword PULSEWIDTH is also used for a timing constraint in ALF. The semantic meaning in both usage cases is consistent: PULSEWIDTH = smallest possible pulse at output or smallest allowed pulse at input. Therefore the usage of the same keyword is justified.
This ALF VECTOR describes the sequence of events

\textit{rising edge at Z from 0 to X followed by falling edge at Z from X to 0.}

The smallest possible pulse to “X” applies at pin Z.

\begin{verbatim}
VECTOR (01 A -> 10 B -> 01 Z -> 10 Z) {
PULSEWIDTH {
    PIN = Z;
    /* fill in data */
}
}
\end{verbatim}

This ALF VECTOR describes the sequence of events as shown in figure 4-9

\begin{itemize}
    \item \textit{rising edge at A followed by falling edge at B followed by rising edge at Z followed by falling edge at Z.}
\end{itemize}

This is a detailed specification of the pulse itself at pin Z as well as of the triggering input signals A and B.

\subsection*{4.15.1.3 SDF RETAIN delays}

RETAIN delay in SDF is a measurement for the time for which an output signal will retain its value after a change at a related input signal occurs. It appears always in conjunction with a IOPATH delay, which is the time for which an output will stabilize after changing its value.
RETAIN is mainly used for asynchronous memories, where decoder glitches may appear at the data output port.

```
VECTOR (01 A -> ?! Z) {
    RETAIN {
        FROM {PIN = A;}
        TO {PIN = Z;}
        /* fill in data */
    }
    DELAY {
        FROM {PIN = A;}
        TO {PIN = Z;}
        /* fill in data */
    }
}
```

![Diagram of RETAIN and IOPATH delay](image)

**Figure 4-10: RETAIN and IOPATH delay**

The ALF VECTOR describes the sequence of events shown in figure 4-10

*rising edge at A followed by any edge at Z.*

The intermediate events at Z, occurring eventually between retain and delay time, are not specified.

### 4.15.1.4 SDF PORT delays

PORT delay in SDF is a delay measurement with unspecified start point, since the start point is going to be established by a connection to a driver in the design and not in the library.

```
VECTOR (01 A) {
    DELAY {
        TO {PIN = A;}
        /* fill in data */
    }
}
```
This ALF VECTOR describes the event figure 4-11

\textit{rising edge at A}.

The absence of a FROM pin defines the absence of a start point, which corresponds to the exact meaning of PORT in SDF.

ALF also has the capability of describing a delay measurement with unspecified end point.

\begin{verbatim}
VECTOR (01 Z) {
   DELAY {
      FROM {PIN = Z;}
      /* fill in data */
   }
}
\end{verbatim}

Hence ALF provides the description capability for both a delay from unspecified driver to specified receiver and a delay from specified driver to unspecified receiver.

\subsection{SDF DEVICE delays}

DEVICE delay in SDF is a delay that applies from all input ports of a device to one specific output port or to all output ports by default.

The ALF vector expression language has no notion of “all input ports of a device”. ALF has a more general capability of declaring groups of pins and define delays from group to group or from group to pin or from pin to group.

\begin{verbatim}
GROUP any_input { A B }
GROUP any_output { Y Z }
VECTOR (01 any_input -> 01 any_output) {
   DELAY {
      FROM {PIN = any_input;}
      TO {PIN = any_output;}
      /* fill in data */
   }
}
\end{verbatim}

The ALF VECTOR above describes the event

\textit{rising edge at any_input (i.e. A or B) followed by rising edge at any_output (i.e. Y or Z).}
This construct is equivalent to the following four vectors:

```plaintext
VECTOR (01 A -> 01 Y) {
   DELAY {
      FROM {PIN = A;}
      TO {PIN = Y;}
      /* fill in data */
   }
}
VECTOR (01 B -> 01 Y) {
   DELAY {
      FROM {PIN = B;}
      TO {PIN = Y;}
      /* same data */
   }
}
VECTOR (01 A -> 01 Z) {
   DELAY {
      FROM {PIN = A;}
      TO {PIN = Z;}
      /* same data */
   }
}
VECTOR (01 B -> 01 Z) {
   DELAY {
      FROM {PIN = B;}
      TO {PIN = Z;}
      /* same data */
   }
}
```

### 4.15.2 SDF timing constraints

#### 4.15.2.1 SDF SETUP

SETUP in SDF is the minimal time required for a data signal to arrive before the sampling edge of a clock signal in order to be sampled correctly.

```plaintext
VECTOR (?! din -> 01 clk) {
   SETUP {
      FROM {PIN = din;}
      TO {PIN = clk;}
      /* fill in data */
   }
}
```
The ALF VECTOR describes the sequence of events as shown in figure 4-12

\[ \text{any edge at din followed by rising edge at clk.} \]

The FROM and TO pin annotations define the sense of measurement for SETUP. Since setup time is measured in positive sense from data to clock, din is the data pin, and clk is the clock pin.

### 4.15.2.2 SDF HOLD

HOLD in SDF is the minimal non-negative time required for a data signal to stay at its value after the sampling edge of a clock signal in order to be sampled correctly.

```verbatim
VECTOR (01 clk -> ?! din) {
  HOLD {
    FROM {PIN = clk;}
    TO {PIN = din;}
    /* fill in data */
  }
}
```

The ALF VECTOR describes the sequence of events as shown in figure 4-13

\[ \text{rising edge at clk followed by any edge at din.} \]

The FROM and TO pin annotations define the sense of measurement for HOLD. Since hold time is measured in positive sense from clock to data, clk is the clock pin, and din is the data pin.
4.15.2.3 SDF SETUPHOLD

SETUPHOLD in SDF is a combination of SETUP and HOLD. In this combination, either SETUP or HOLD may be a negative value, but the sum of both values, which represents the minimal pulselength of the data in order to be sampled correctly, must be non-negative. The time from the leading data edge to the sampling clock edge is SETUP. The time from the sampling clock edge to the trailing data edge is HOLD.

VECTOR // for SETUPHOLD
{
   ! din -> 01 clk -> ! din //setup & hold both positive
   | 01 clk -> ! din -> ! din //negative setup, positive hold
   | ! din -> ?! din -> 01 clk //positive setup, negative hold
} { 
   SETUP {
      FROM {PIN = din;
      TO {PIN = clk;}
      /* fill in data */
   }
   HOLD {
      FROM {PIN = clk;}
      TO {PIN = din;}
      /* fill in data */
   }
}

Figure 4-14: Measurement of SDF SETUPHOLD

The ALF VECTOR describes the alternative sequences of events as shown in figure 4-14

any edge at din followed by rising edge at clk followed by any edge at din
or rising edge at clk followed by any edge at din followed by any edge at din
or any edge at din followed by any edge at din followed by rising edge at clk.

The FROM and TO pin annotations define the sense of measurement for SETUP and HOLD, respectively, in the same way as if they were specified in separate vectors.
4.15.2.4 SDF RECOVERY

RECOVERY in SDF is the minimal time required for a higher priority asynchronous control signal to be released before a lower priority clock signal in order to allow the clock to be in control.

VECTOR (01 clearbar -> 01 clk) {
  RECOVERY {
    FROM {PIN = clearbar;}
    TO {PIN = clk;}
  }
}

Figure 4-15: Measurement of SDF RECOVERY

The ALF VECTOR describes the sequence of events as shown in figure 4-15
  rising edge at clearbar followed by rising edge at clk.

The FROM and TO pin annotations define the sense of measurement for RECOVERY. Since recovery time is measured in positive sense from the higher priority asynchronous control signal to the lower priority clock, clearbar is the asynchronous control pin, and clk is the clock pin.

4.15.2.5 SDF REMOVAL

REMOVAL in SDF is the minimal time required for a higher priority asynchronous control signal to stay active after a lower priority clock signal in order to keep overriding the clock.

VECTOR (01 clk -> 01 clearbar) {
  REMOVAL {
    FROM {PIN = clk;}
    TO {PIN = clearbar;}
  }
}
The ALF VECTOR describes the sequence of events as shown in figure 4-16

*rising edge at clk followed by rising edge at clearbar.*

The FROM and TO pin annotations define the sense of measurement for REMOVAL. Since removal time is measured in positive sense from the lower priority clock to the higher priority asynchronous control signal, clk is the clock pin, and clearbar is the asynchronous control pin.

### 4.15.2.6 SDF RECREM

RECREM in SDF is a combination of RECOVERY and REMOVAL. In this combination either RECOVERY or REMOVAL may be negative, but the sum of both must be non-negative. The sum of RECOVERY and REMOVAL represents the width of the “forbidden zone” for the phase between the higher priority and the lower priority signal. The boundary to the left is RECOVERY, the boundary to the right is REMOVAL.

In a characterization vector for RECREM, either the RECOVERY or the REMOVAL effect can be observed, depending on the phase relationship between the signals. This is different from SETUPHOLD where the effects of both SETUP and HOLD can be observed in the same characterization vector.

```plaintext
VECTOR // for RECREM
  ( 01 clearbar -> 01 clk // pos. recovery or neg. removal
    | 01 clk -> 01 clearbar // neg. recovery or pos. removal
  ) {
    RECOVERY {
      FROM {PIN = clearbar;}
      TO {PIN = clk;}
      /* fill in data */
    }
    REMOVAL {
      FROM {PIN = clk;}
      TO {PIN = clearbar;}
      /* fill in data */
    }
  }
```

---

Figure 4-16: Measurement of SDF REMOVAL
4.15.2.7 SDF SKEW

SKEW in SDF is maximum allowed difference in arrival time between signals. The allowed region for the phase between signals is bound by zero to the left and SKEW to the right for positive SKEW or by SKEW to the left and zero to the right for negative SKEW.

VECTOR (01 clk1 & 01 clk2) { // pos. or neg. or zero skew
    SKEW {
        FROM {PIN = clk1;}
        TO {PIN = clk2;}
        /* fill in data */
    }
}

Figure 4-18: Measurement of SDF SKEW
The ALF VECTOR describes the alternative sequences of events as shown in figure 4-18

- rising edge at clk1 followed by rising edge at clk2
- or rising edge at clk2 followed by rising edge at clk1
- or rising edge at clk2 simultaneously with rising edge at clk1

This is the most general case, where the skew may be positive, negative or zero across the characterization space. The FROM and TO pin annotations define the sense of measurement for SKEW.

4.15.2.8 SDF WIDTH

```
VECTOR (01 clk -> 10 clk) {// high pulse
    PULSEWIDTH {
        PIN = clk;
        /* fill in data */
    }
}
```

This ALF vector describe the sequence of events as shown in figure 4-19

- rising edge at clk followed by falling edge at clk.

The pulsewidth applies to the positive phase of the signal clk.

```
VECTOR (10 clk -> 01 clk) {// low pulse
    PULSEWIDTH {
        PIN = clk;
        /* fill in data */
    }
}
```

This ALF vector describe the sequence of events

- falling edge at clk followed by rising edge at clk.

The pulsewidth applies to the negative phase of the signal clk.

```
VECTOR (01 clk -> 10 clk | 10 clk -> 01 clk) {// high or low pulse
    PULSEWIDTH {
        PIN = clk;
        /* fill in data */
    }
}
```

This ALF vectors describes the alternative sequences of events as shown in figure 4-20
rising edge at clk followed by falling edge at clk
or falling edge at clk followed by rising edge at clk.

The pulsewidth applies to both phases of the signal clk.

4.15.2.9 SDF PERIOD

VECTOR (01 clk -> 10 clk -> 01 clk) {
    PERIOD {
        PIN = clk;
        /* fill in data */
    }
}

Figure 4-20: Measurement of SDF PERIOD

This ALF vectors describes the sequence of events as shown in figure 4-21
rising edge at clk followed by falling edge at clk followed by rising edge at clk.
Thus the period is measured between two consecutive rising edges at the signal clk.

4.15.2.10 SDF NOCHANGE

VECTOR (!? addr -> 10 write -> 01 write -> ?! addr) {
    SETUP {
        FROM {PIN = addr;}
        TO {PIN = write;}
        /* fill in data */
    }
    HOLD {
        FROM {PIN = write;}
        TO {PIN = addr;}
        /* fill in data */
    }
    NOCHANGE {
        PIN = addr;
        /* fill in optional data */
    }
}
This ALF vector describes the sequence of events as shown in figure 4-21:

\[
\text{any edge at addr followed by falling edge at write followed by rising edge at write followed by any edge at addr.}
\]

The SETUP time is measured from the first edge at addr to the first edge at write. The HOLD time is measured from the second edge at write to the second edge at addr. The signal addr may not change between the start time of the setup measurement until the end time of the hold measurement. ALF allows to specify an additional measurement between the first and second edge of the signal subject to NOCHANGE. However, this additional measurement could not be directly translated into SDF and would be for characterization and future purpose only.

### 4.15.3 SDF conditions and labels for delays and timing constraints

Conditions for IOPATH timing arcs in SDF apply to the entire timing arc. The condition is evaluated during the event on the “from” port (i.e. an input pin), and the event on the “to” port (i.e. an output pin) is scheduled consequently.

Conditions for timing constraints in SDF can be defined individually for each port. The condition associated with the start point of the timing constraint (i.e. data for SETUP, clock for HOLD etc.) is called stamp condition. The condition associated with the end point of the timing constraint (i.e. clock for SETUP, data for HOLD) is called check condition.

The use of SETUPHOLD instead of individual SETUP and HOLD or RECREM instead of individual RECOVERY and REMOVAL in SDF imposes restrictions in the definition of conditions. Whereas the use of 2 individual timing constraints allows the definition of 4 conditions (2 stamp, 2 check), the use of 1 combined timing constraint allows only the definition of 2 conditions (1 stamp, 1 check).

The ALF vector expression language allows to specify conditions during the sequence of events in a more general way than SDF.

Some more examples in ALF:
ALF/SDF cross reference

**Figure 4-22: Condition during sequence of two events**

VECTOR ( C & ( 01 A -> 01 B ) )

alternative specification options:

VECTOR ( ?1 C -> 01 A -> 01 B -> 1? C ) // verbose
VECTOR ( ?1 C -> 01 A -> 01 B ) // C must be true before start
VECTOR ( 01 A -> 01 B -> 1? C ) // C must be true until the end

This ALF vector describes the sequence of events as shown in figure 4-22

*rising edge at A is followed by rising edge at B, C is true before rising edge of A until after rising edge of B.*

Either of the pseudo-events (?1 C, 1? C) at the boundary can be omitted, since either one of them is sufficient to specify that the condition C must be true during the entire event sequence.

**Figure 4-23: Condition during leading event**

VECTOR ( ( C & 01 A ) -> 01 B )

alternative specification options:

VECTOR ( ?1 C -> 01 A -> 1? C -> 01 B )
VECTOR ( 01 A -> 1? C -> 01 B )

This ALF vector describes the sequence of events as shown in figure 4-23
rising edge at A is followed by rising edge at B, C is true before rising edge of A until after rising edge of A.

VECTOR ( 01 A -> (C & 01 B) )

alternative syntax:

VECTOR ( 01 A -> ?1 C -> 01 B -> 1? C )

This ALF vector describes the sequence of events as shown in figure 4-24

rising edge at A is followed by rising edge at B, C is true before rising edge of B until after rising edge of B.

SETUPHOLD with SCOND (stamp condition) and CCOND (check condition) in SDF can be described in ALF in the following way:
VECTOR ( ?! din -> ?1 ccond -> 01 clk -> 1? scond -> ?! din ) {
    SETUP {
        FROM {PIN = din;}
        TO {PIN = clk;}
        /* fill in data */
    }
    HOLD {
        FROM {PIN = clk;}
        TO {PIN = din;}
        /* fill in data */
    }
}

A more verbose specification of the vector looks as follows:

VECTOR {
    ?1 scond // scond must be true at the beginning
    -> ?! din // din toggles
    -> ?1 ccond // last chance for ccond to become true
    -> 01 clk // rising edge at clk
    -> 1? scond // scond gets a break
    -> ?! din // din toggles
    -> 1? ccond // ccond gets a break at last
}

The optional condition label in SDF has its counterpart in ALF (see 3.6.4.1). As in SDF, the use and interpretation of this label is defined by the application tool and not by the standard.
Index

Symbols
(N+1) order sequential logic 21
-> operator 21
?- 41
?! 41
?? 41
?- 41
@ 19

Numerics
2-dimensional tables 179
3-dimensional table 182, 187

A
ABS 61
abs 49
abstract transition operators 195
active vectors 119
ALF_AND 126, 127, 168
ALF_BUF 126
ALF_BUFIF0 129
ALF_BUFIF1 129
ALF_FLIPFLOP 132, 166
ALF_LATCH 133
ALF_MUX 131, 167
ALF_NAND 126, 127
ALF_NAND2 165
ALF_NOR 126, 128
ALF_NOT 126
ALF_NOTIF0 129, 130
ALF_NOTIF1 129, 130
ALF_OR 126, 127
ALF_XNOR 126, 128
ALF_XOR 126, 128
ALIAS 31
alias 51
all_purpose_items 51
alphabetic_bit_literal 39
annotated properties 26
annotation 51
  arithmetic model tables
    AREA 88
    CAPACITANCE 88
    CONNECTIONS 90
    CONNECTIVITY 90
    CURRENT 87
    DELAY 86
    DERATE_CASE 91
    DISTANCE 88
    DRIVE_STRENGTH 89
    DRIVER 90
    ENERGY 87
    FANIN 90
    FANOUT 90
    FREQUENCY 87
    HEIGHT 88
    HOLD 86
    JITTER 87
    LENGTH 89
    NOCHANGE 86
    POWER 87
    PROCESS 91
    PULSEWIDTH 87
    RECEIVER 90
    RECOVERY 87
    REMOVAL 87
    RESISTANCE 88
    SETUP 87
    SKEW 87
    SLEWRATE 86
    SWITCHING_BITS 90
    TEMPERATURE 87
    THRESHOLD 89
    TIME 88
    VOLTAGE 88
    WIDTH 89

arithmetic models 100
  average 104
  can_short 102
  cannot_short 102
  CONNECT_RULE 102
  DEFAULT 100
  FALL 99
  MAX 96
  MEASUREMENT 104
MIN 96
must_short 102
peak 104
RISE 99
rms 104
static 104
transient 104
TYP 96
UNIT 100
CELL 80
BUFFERTYPE 81
CELLTYPE 80
DRIVERTYPE 81
NON_SCAN_CELL 82
PARALLEL_DRIVE 81
SCAN_TYPE 82
SCAN_USAGE 82
cell buffertype
inout 81
input 81
internal 81
output 81
cell celltype
block 81
buffer 81
combinational 81
core 81
flipflop 81
latch 81
memory 81
multiplexor 81
pad 81
special 81
cell drivertype
both 81
predriver 81
slotdriver 81
cell scan_type
clocked 82
control_0 82
control_1 82
lssd 82
muxscan 82
cell scan_usage
hold 82
input 82
output 82
default 100
from 93
information 71
AUTHOR 72
DATETIME 72
PRODUCT 72
TITLE 72
VERSION 72
limit 93
object reference
cell 73
pin 73
primitive 73
PIN
ACTION 76
CONNECT_CLASS 77
DATATYPE 77
DIRECTION 75
DRIVERTYPE 74
ENABLE_PIN 76
OFF_STATE 78
ORIENTATION 77
POLARITY 76
PULL 76
SCAN_POSITION 77
SCOPE 75
SIGNALTYPE 74
STUCK 77
VIEW 73
pin
PINTYPE 73
pin action
asynchronous 76
synchronous 76
pin datatype
signed 77
unsigned 77
pin direction
both 75
input 75
none 75
output 75
pin drivertype
cmos 75
cmos_pass 75
nmos 75
nmos_pass 75
open_drain 75
open_source 75
pmos 75
pmos_pass 75
ttl 75
pin off_state
  inverted 78
  non_inverted 78
pin orientation
  bottom 77
  left 77
  right 77
  top 77
pin pintype
  analog 74
  digital 74
  supply 74
pin polarity
  both 76
  double_edge 76
  falling_edge 76
  high 76
  inverted 76
  low 76
  non_inverted 76
  none 76
  rising_edge 76
pin pull
  both 77
  down 77
  none 77
  up 77
pin scope
  behavior 75
  both 75
  measure 75
  none 75
pin signaltype
  clear 74
  clock 74
  control 74
  data 74
  enable 74
  master_clock 74
  out_enable 74
  read 74
  scan_clock 74
  scan_data 74
  scan_enable 74
  scan_out_enable 74
  select 74
  set 74
  slave_clock 74
  write 74
pin stuck
  both 78
  none 78
  stuck_at_0 78
  stuck_at_1 78
pin view
  both 73
  functional 73
  none 73
  physical 73
  scan 71
  to 93
  unnamed 71
  VECTOR 78
  LABEL 78, 79, 80
  violation 71
  MESSAGE 72
  MESSAGE_TYPE 72
annotation container 33, 71, 93
annotation_container 51
annotations 185
PIN 185
pin 204
self-explaining 187
timing arc 186
annotation
  object reference
class 73
any_character 38
arithmetic models 30
arithmetic operations 23
arithmetic operators
  binary 61
  function 61
  unary 61
arithmetic_binary_operator 49
arithmetic_expression 46
arithmetic_function_operator 49
arithmetic_model 55
arithmetic_model_template_instantiation 55
arithmetic_unary_operator 49
assignment_base 45
async_2write_1read_ram 198
atomic megacell 17
atomic object 29
ATTRIBUTE 32, 84
attribute 51
   CELL 85
   cell
      asynchronous 85
      CAM 85
      dynamic 85
      RAM 85
      ROM 85
      static 85
      synchronous 85
LIBRARY 85
PIN 84
pin
   PAD 85
   SCHMITT 84
   TRISTATE 84
   XTAL 85
pin polarity
   READ 85
   TIE 85
   WRITE 85
attribute_items 52
average 181

B
based literal 40
based_literal 40
BEHAVIOR 165
behavior 56
behavior_body 56
bidirectional pin 192
binary 40
Binary operators
   arithmetic 61
   bitwise 63
   boolean, scalars 62
   reduction 63
   vector 66, 67
   binary_base 40
   binary_digit 40
   bit 39
   bit_edge_literal 41
   bit_literal 39
   Bitwise operators
      binary 63
      unary 63
   block comment 38
   bodies 56
   Boolean Equation 165
   boolean functions 17
   boolean operators
      binary 62
      unary 62
      boolean_and_operator 49
      boolean_arithmetic_operator 49
      boolean_binary_operator 50
      boolean_case_compare_operator 49
      boolean_condition_operator 50
      boolean_else_operator 50
      boolean_expression 46
      boolean_logic_compare_operator 49
      boolean_or_operator 49
      boolean_unary_operator 49
      both 192
      bus contention 190
      bus modeling 189
      bus with multiple drivers 191
      busholder 191

C
can_float 187
CAPACITANCE 174, 192
case-insensitive language 38
cell 53
cell modeling 26
cell_identifier 47, 53
cell_instantiation 47
cell_items 53
cell_template_instantiation 53
characterization 15, 17
   power 17, 24
   timing 17
characterization model 178
Characterization Modeling 22
characterization variables 17
children object 29
CLASS 31, 204
class 52
  connectivity 204
combinational logic 18
combinational primitives 126
combinational scan cell 170
combinational_assignments 57
comment 37
  block 38
  long 38
  short 38
  single-line 38
comments
  nested 38
compound operators 38
CONNECT_RULE 204
CONNECTION 187
connections
  allowed 204
  disallowed 203
  external 203
CONNECTIVITY 204
connectivity 203
  class 204
  net-specific 205
  pin-specific 205
connectivity class 204
CONSTANT 31
constant 52
constant numbers 38
constraints
  delay 179
  power 179
  timing 179
context_sensitive_keyword 48
context-sensitive keyword 44, 190
context-sensitive keywords 23
core 17
core cell 190
core modeling 201
D
  d_flipflop_clr 166
d_flipflop ld_clr 168
d_flipflop mux_set_clr 168
dLatch 169
decimal 40
decimal_base 40
deep submicron 15
DEFAULT 187, 188
default annotation 100
delay mode
  inertial 24
  invalid-value-detection 24
  transport 24
delay models 22
delay predictor 23
delimiter 37, 38
derating 183
derating equation 184
digit 40
digital filter 201
digital_filter 202
DRIVE_STRENGTH 190
DRIVER 204
E
  edge literal 41
  edge rate 22
  edge_literal 41
eX 49
drive_strength 190
energy 24
equation 56
equation_template_instantiation 56
escape codes 41
escape_character 37
escaped identifier 42
escaped_identifier 42
event sequence detection 21
EXP 61
exp 49
expansion
  bit-wise 194
  bytewise 194
expansion of vectors 194
exponentiation 23
extensible primitives 124
external connections 203

F
fanout 27
Flipflop 132
flipflop 166
forward referencing 29
fringe capacitance 27
FUNCTION 165
function 56
  exponentiation 23
  logarithm 23
Function operators
  arithmetic 61
function_template_instantiation 56
functional model 15
functional modeling 18
functional models 17
functions 30

G
generic objects 30
generic_object 51
glitch 24
GROUP 33, 194
group 52
group_identifier 52

H
hard keyword 44
hardware description language 17
HDL 17
header 55
header_template_instantiation 55
hex_base 40
hex_digit 40
hexadecimal 40
hierarchical object 29

I
identifier 29, 37
Identifiers 42
identifiers 48
inactive vectors 119
INCLUDE 31, 109
include 52
index 48
inertial delay mode 24
infinite impulse response filter 201
INFORMATION 170
integer 38, 39
internal load 23
intrinsic delay 22

J
JK-flipflop 167
JTAG BSR cell 170

K
keyword 29
Keywords
  context-sensitive 45
  generic objects 44
  operators 45
keywords
  context-sensitive 23

L
Latch 133
layout parasitics 23
level-sensitive cell 169
level-sensitive sequential logic 18
libraries 53
LIBRARY 170
library 29
Library creation 11
library_identifier 54
library_items 53
library_specific_object 51
library_template_instantiation 53
library-specific objects 30
LIMIT 187
literal 29, 37
load characterization model 23
LOG 61
log 49
logarithm 23
logic_literals 48
logic_values 48
logic_variables 49

M
macrocells 17
MAX 62
max 49
MEASUREMENT 181
megacell modeling 194
megacells 17
metal layer 27
MIN 62
min 49
mode of operation 15
modeling
  bus 189
  cell 26
  characterization 22
  cores 201
  functional 18
  megacell 194
  physical 26
  power 23
  synthesis 26
  test 26
  timing 22
  wire 27
  wireload 192
multiplexor 131
must_connect 187
muxscan 172

N
named_assignment 45
named_assignment_base 45
NAND gate 165
nested comments 38
no_connection 187
non_negative_number 39
NON_SCAN_CELL 171
non-escaped identifier 42
nonescaped_identifier 42
nonreserved_character 37
non-scan cells 26
Number 38
number 39
numbers 49
numeric_bit_literal 39

O
objects 29, 52
octal 40
octal_base 40
octal_digit 40
one_shot 200
one-pass parser 29
operation mode 15
operator
  -> 21
  followed by 21
operators
  arithmetic 61
  boolean, scalars 62
  boolean, words 63
  signed 64
  unsigned 64
output ramptime 174

P
parasitic capacitance 27
parasitic resistance 27
physical modeling 26
pin_assignments 46
pin_identifier 53
pin_items 54
pin_template_instantiation 53
pins 53
placeholder identifier 42
placeholder_identifier 42
placeholders 32
POLARITY 172
power 24
Power characterization 17
power characterization 24
power constraint 15
power dissipation 24
Power model 15
power modeling 23
predefined derating cases 92, 104, 105
  bccom 92
  bcind 92
  bcmil 92
  wcocom 92
wcind 92
wcnil 92
predefined process names 91
  snsp 91
  snwp 91
  wns 91
  wnwp 91
primitive 166
primitive_identifier 47, 54
primitive_instantiation 47
primitive_items 54
primitive_template_instantiation 54
primitives 54
private keywords 45
PROCESS 183
PROPERTY 33
property 52
public keywords 45
pulse generator 200
PVT Derating 183
Q
Q_CONFLICT 132
QN_CONFLICT 132
quad D-Flipflop 173
quoted string 37, 41
quoted_string 41
R
RAM16X4 203
real 38
Reduction operators
  binary 63
  unary 63
reserved keyword 44
reserved_character 37
RESISTANCE 193
RTL 14
S
scaled average current 24
scaled average power 24
scan cell
  combinational 170
scan chai 170
Scan Flipflop 172
Scan insertion 26
scan test 26
scan_data 172
scan_enable 172
SCAN_FFX4 173
SCAN_ND4 171
SCAN_TYPE 171
self capacitance 27
self-explaining annotations 187
sequential logic
  edge-sensitive 18
  level-sensitive 18
  N+1 order 21
  vector-sensitive 21
sequential_assignment 57
sheet resistance 27
sign 38
signed operators 64
simulation model 15
single-line comment 38
slew rate 22
SLEWRATE 174, 189
soft keyword 44
source_text 51
sr_latch 169
state-dependent drive strength 190
STATETABLE 165
statetable 56
statetable_body 56
static power 25
std_derating 184
std_header_2d 176
string 49
sublibraries 54
sublibrary_template_instantiation 54
switching energy 174
symbolic_edge_literal 41
T
TABLE 174
table 56
table_items 56
table_template_instantiation 56
TEMPERATURE 183
TEMPLATE 32, 176
template 52, 174
template definition 176
template_identifier 52
template_instantiation 47
template-reference scheme 23
Ternary operator 62
Three-port Memory 198
timing arc 186
timing characterization model 17
timing constraint model 22
timing constraint models 22
timing constraints 15, 179
timing modeling 22
timing models 15
transcendent functions 23
transient power 25
transition delay 22
transmission-gate 190
transport delay mode 24
  invalid-value-detection 24
triggering conditions 18
triggering function 18
tristate driver 189
tristate primitives 129
tristate_buffer 189
Truth Table 165
truth table 17
Two-port memory 195
U
Unary operator
  bitwise 63
Unary operators
  arithmetic 61
  boolean, scalar 62
  reduction 63
Unary vector operators 64
unnamed annotation containers 71
unnamed_assignment 45
unnamed_assignment_base 45
unnamed_assignments 45
unsigned 39
unsigned operators 64
V
VCO 200
VECTOR 174
vector 54
vector expression 21
Vector operators
  binary 66, 67
  unary, bits 64
  unary, words 65
vector_binary_operator 50
vector_elsif_operator 50
vector_expression 47, 54
vector_if_operator 50
vector_items 54
vector_template_instantiation 54
vector_unary_operator 50
vector-based modeling 15
Vector-Sensitive Sequential Logic 21
vector-specific model 174
Verilog 14, 19
VHDL 14, 19
via resistance 27
VIOLATION 180
virtual pins 26, 132
VOLTAGE 183, 188
voltage_controlled_delay 201
W
whitespace 38
whitespace characters 37
wildcard_literal 39
wire 55
wire modeling 27
wire select model 193
wire_identifier 55
wire_items 55
wire_template_instantiation 55
word_edge_literal 41