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

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

IEEE P1603 Draft 3

January 4, 2002

put in IEEE verbage

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

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

Revision history:

IEEE P1596 Draft 0	August 19, 2001
IEEE P1603 Draft 1	September 17, 2001
IEEE P1603 Draft 2	November 12, 2001

Table of Contents

1. Introduction 1

1.1 Motivation 1
1.2 Goals 2
1.3 Target applications 2
1.4 Conventions 5
1.5 Contents of this standard 5

2. References 7

3. Definitions 9

4. Acronyms and abbreviations 11

5. ALF language construction principles 13

5.1 ALF meta-language 13

1. ALF language construction principles 13

1.1 ALF meta-language 13
5.2 Categories of ALF statements 14
5.3 Relationships between objects 14
5.4 References of objects 17
5.5 Incremental definitions 19
5.6 Scoping rules 19

6. Lexical rules 21

6.1 Cross-reference of lexical tokens 21
6.2 Characters 21
6.2.1 Character set 21
6.2.2 Whitespace characters 22
6.2.3 Other characters 22
6.3 Lexical tokens 23
6.3.1 Delimiter 23
6.3.2 Comment 24
6.3.3 Number 24
6.3.4 Bit literals 24
6.3.5 Based literals 25
6.3.6 Edge literals 26
6.3.7 Quoted strings 26
6.3.8 Identifier 27
6.4 Keywords 29
6.4.1 Keywords for objects 29
6.4.2 Keywords for operators 29
6.4.3 Context-sensitive keywords 30
6.5 Rules against parser ambiguity 30

6.6 Values ... 30
6.6.1 Arithmetic value ... 30
6.6.2 String value ... 31
6.6.3 Edge values ... 31
6.6.4 Index value ... 31

7. Auxiliary items ... 33

7.1 Index and related items ... 33
7.1.1 Index ... 33
7.1.2 Index range ... 33
7.2 Pin assignment and related items ... 33
7.2.1 Pin assignment ... 33
7.2.2 Pin variable ... 34
7.2.3 Pin value ... 34
7.3 Annotation and related items ... 35
7.3.1 Annotations ... 35
7.3.2 Annotation value ... 35
7.4 All purpose item ... 36

8. Generic objects ... 37

8.1 INCLUDE statement ... 37
8.1.1 Interpreting special symbols ... 37
8.1.2 Use of multiple files ... 37
8.2 ALIAS statement ... 38
8.3 CONSTANT statement ... 38
8.4 ATTRIBUTE statement ... 38
8.5 PROPERTY statement ... 39
8.6 CLASS statement ... 40
8.7 KEYWORD statement ... 40
8.8 GROUP statement ... 41
8.9 TEMPLATE statement ... 41
8.9.1 Referencing by placeholder ... 43
8.9.2 Parameterizeable cells ... 43

9. Library-specific objects ... 47

9.1 Library-specific objects ... 47
9.1.1 Library-specific singular objects ... 48
9.1.2 Modeling for synthesis and test ... 48
9.2 LIBRARY statement and related statements ... 48
9.2.1 LIBRARY statement ... 49
9.2.2 SUBLIBRARY statement ... 49
9.2.3 INFORMATION statement ... 49
9.2.4 INFORMATION container ... 50
9.3 CELL statement and related statements ... 50
9.3.1 CELL statement ... 50
9.3.2 NON_SCAN_CELL statement ... 51
9.3.3 Annotations and attributes for a CELL ... 52
9.4 PIN statement and related statements ... 61
9.4.1 PIN statement ... 61
9.4.2 Definitions for bus pins ... 61
9.4.3 RANGE statement ... 63

9.4.4 PIN_GROUP statement....64
9.4.5 Annotations and attributes for a PIN....65
9.5 WIRE statement and related statements....81
9.5.1 WIRE statement....81
9.5.2 NODE statement....87
9.6 VECTOR statement and related statements....90
9.6.1 VECTOR statement....90
9.6.2 ILLEGAL statement....90
9.6.3 Annotations and attributes for a VECTOR....91
9.7 LAYER statement and related statements....96
9.7.1 LAYER statement....97
9.7.2 PURPOSE annotation....98
9.7.3 PITCH annotation....98
9.7.4 PREFERENCE annotation....99
9.7.5 Example....99
9.8 VIA statement and related statements....100
9.8.1 VIA statement....100
9.8.2 USAGE annotation....101
9.8.3 Example....102
9.8.4 VIA reference statement....103
9.9 Statements related to physical design rules....104
9.9.1 RULE statement....104
9.9.2 ANTENNA statement....108
9.9.3 BLOCKAGE statement....112
9.9.4 PORT statement....113
9.10 Statements related to physical geometry....116
9.10.1 SITE statement....116
9.10.2 ARRAY statement....118
9.10.3 PATTERN statement....121
9.10.4 ARTWORK statement....123
9.10.5 Geometric model....124
9.10.6 Geometric transformation....129
9.11 Statements related to functional description....132
9.11.1 FUNCTION statement....132
9.11.2 TEST statement....133
9.11.3 Physical bitmap for memory BIST....133
9.11.4 BEHAVIOR statement....138
9.11.5 STRUCTURE statement....139
9.11.6 VIOLATION statement....144
9.11.7 STATETABLE statement....145
9.11.8 PRIMITIVE statement....147

10. Constructs for modeling of digital behavior....159

10.1 Variable declarations....159
10.2 Combinational functions....159
10.2.1 Combinational logic....159
10.2.2 Boolean operators on scalars....160
10.2.3 Boolean operators on words....160
10.2.4 Operator priorities....162
10.2.5 Datatype mapping....162
10.2.6 Rules for combinational functions....164
10.2.7 Concurrency in combinational functions....165
10.3 Sequential functions....165

10.3.1 Level-sensitive sequential logic ... 166
10.3.2 Edge-sensitive sequential logic ... 166
10.3.3 Unary operators for vector expressions ... 168
10.3.4 Basic rules for sequential functions ... 169
10.3.5 Concurrency in sequential functions ... 172
10.3.6 Initial values for logic variables ... 173
10.4 Higher-order sequential functions ... 174
10.4.1 Vector-sensitive sequential logic ... 174
10.4.2 Canonical binary operators for vector expressions ... 175
10.4.3 Complex binary operators for vector expressions ... 176
10.4.4 Extension to N operands ... 177
10.4.5 Operators for conditional vector expressions ... 179
10.4.6 Operators for sequential logic ... 180
10.4.7 Operator priorities ... 180
10.4.8 Using PINs in VECTORs ... 181
10.5 Modeling with vector expressions ... 181
10.5.1 Event reports ... 182
10.5.2 Event sequences ... 183
10.5.3 Scope and content of event sequences ... 184
10.5.4 Alternative event sequences ... 186
10.5.5 Symbolic edge operators ... 187
10.5.6 Non-events ... 188
10.5.7 Compact and verbose event sequences ... 189
10.5.8 Unspecified simultaneous events within scope ... 190
10.5.9 Simultaneous event sequences ... 191
10.5.10 Implicit local variables ... 193
10.5.11 Conditional event sequences ... 194
10.5.12 Alternative conditional event sequences ... 196
10.5.13 Change of scope within a vector expression ... 198
10.5.14 Sequences of conditional event sequences ... 201
10.5.15 Incompletely specified event sequences ... 203
10.5.16 How to determine well-specified vector expressions ... 204
10.6 Boolean expression language ... 205
10.7 Vector expression language ... 205
10.8 Control expression semantics ... 206

11. Constructs for modeling of analog behavior ... 209

11.1 Arithmetic expression language ... 209
11.1.1 Syntax of arithmetic expressions ... 209
11.1.2 Arithmetic operators ... 210
11.1.3 Operator priorities ... 211
11.2 Arithmetic model and related statements ... 211
11.2.1 Arithmetic models ... 211
11.2.2 Arithmetic model statement ... 217
11.2.3 Partial arithmetic model ... 217
11.2.4 Non-trivial arithmetic model ... 218
11.2.5 Trivial arithmetic model ... 218
11.2.6 Assignment arithmetic model ... 218
11.2.7 Items for any arithmetic model ... 219
11.3 Arithmetic submodel and related statements ... 219
11.3.1 Arithmetic submodel statement ... 219
11.3.2 Non-trivial arithmetic submodel ... 219
11.3.3 Trivial arithmetic submodel ... 219

11.3.4 Items for any arithmetic submodel 220
11.4 Arithmetic body and related statements 220
11.4.1 Arithmetic body 220
11.4.2 HEADER statement 220
11.4.3 TABLE statement 221
11.4.4 EQUATION statement 221
11.5 Arithmetic model container 221
11.5.1 LIMIT container 222
11.5.2 Containers for arithmetic models and submodels 222
11.6 Statements related to arithmetic models for general purpose 223
11.6.1 MIN and MAX statements 223
11.6.2 TYP statement 224
11.6.3 DEFAULT statement 224
11.6.4 LIMIT statement 225
11.6.5 Annotations for arithmetic models for general purpose 228
11.7 Rules for evaluation of arithmetic models 233
11.7.1 Arithmetic model with arithmetic submodels 233
11.7.2 Arithmetic model with table arithmetic body 233
11.7.3 Arithmetic model with equation arithmetic body 233
11.8 Overview of arithmetic models 233
11.8.1 Overview of modeling keywords 233
11.8.2 Arithmetic models in the context of layout 237
11.9 Arithmetic models for timing data 240
11.9.1 Specification of timing models 240
11.9.2 TIME statement 244
11.9.3 FREQUENCY statement 248
11.9.4 DELAY and RETAIN statements 250
11.9.5 SLEWRATE statement 251
11.9.6 SETUP and HOLD statement 251
11.9.7 NOCHANGE statement 252
11.9.8 RECOVERY and REMOVAL statements 252
11.9.9 SKEW statement 253
11.9.10 PULSEWIDTH statement 254
11.9.11 PERIOD statement 254
11.9.12 JITTER statement 254
11.9.13 THRESHOLD statement 255
11.10 Auxiliary statements related to timing data 257
11.10.1 FROM and TO statements 257
11.10.2 EARLY and LATE statements 258
11.10.3 Annotations for arithmetic models for timing data 258
11.11 Arithmetic models for environmental data 261
11.11.1 PROCESS and DERATE_CASE statement 261
11.11.2 TEMPERATURE statement 263
11.12 Arithmetic models for electrical data 263
11.12.1 PIN-related arithmetic models for electrical data 264
11.12.2 CAPACITANCE statement 266
11.12.3 RESISTANCE statement 266
11.12.4 INDUCTANCE statement 266
11.12.5 VOLTAGE statement 266
11.12.6 CURRENT statement 267
11.12.7 POWER and ENERGY statement 267
11.12.8 FLUX and FLUENCE statement 268
11.12.9 DRIVE_STRENGTH statement 269
11.12.10 SWITCHING_BITS statement 270

11.12.11NOISE and NOISE_MARGIN statement 271
11.12.12Annotations for arithmetic models for electrical data 278
11.13Arithmetic models for physical data 280
11.13.1 CONNECTIVITY statement 280
11.13.2 SIZE statement 283
11.13.3 AREA statement 283
11.13.4 WIDTH statement 283
11.13.5 HEIGHT statement 283
11.13.6 LENGTH statement 283
11.13.7 DISTANCE statement 283
11.13.8 OVERHANG statement 283
11.13.9 PERIMETER statement 284
11.13.10EXTENSION statement 284
11.13.11THICKNESS statement 284
11.13.12Annotations for arithmetic models for physical data 284
11.14Arithmetic submodels for timing and electrical data 287
11.14.1 RISE and FALL statement 287
11.14.2 HIGH and LOW statement 287
11.15Arithmetic submodels for physical data 288
11.15.1 HORIZONTAL and VERTICAL statement 288

(informative)Syntax rule summary 289

A.1 Lexical definitions 289

A.2 Auxiliary definitions 291

A.3 Generic definitions 293

A.4 Library definitions 294

A.5 Control definitions 301

A.6 Arithmetic definitions 302

(informative)Bibliography 305

List of Figures

Figure 1—ALF and its target applications 4
Figure 2—Objects containing annotations or annotation containers 15
Figure 3—Objects containing generic objects 15
Figure 4—Objects in a library for logical and electrical design and their relationships 16
Figure 5—Objects in a library for physical design and their relationships 17
Figure 6—Referencing rules for ALF objects 18
Figure 7—Generic objects 37
Figure 8—Library-specific objects 47
Figure 9—Library-specific singular objects 48
Figure 10—FUNCTION and TEST 48
Figure 11—Illustration of independent SWAP_CLASS and RESTRICT_CLASS 57
Figure 12—Illustration of SWAP_CLASS with inherited RESTRICT_CLASS 58
Figure 13—Construction scheme for composite SIGNALTYPE values 69
Figure 14—Example of boundary parasitic description 85
Figure 15—Example for interconnect description 89
Figure 16—Metal-poly illustration 112
Figure 17—Routing layer shapes 122
Figure 18—Illustration of VERTEX annotation 123
Figure 19—Geometric model and its context 125
Figure 20—Illustration of geometric models 126
Figure 21—Illustration of straight point-to-point connection 127
Figure 22—Illustration of rectilinear point-to-point connection 127
Figure 23—Illustration of FLIP, ROTATE, and SHIFT 132
Figure 24—Illustration of a physical memory architecture, arranged in banks, rows, columns 134
Figure 25—Illustration of the memory BIST concept 134
Figure 26—Concurrency for combinational logic 165
Figure 27—Model of a flip-flop with asynchronous clear in ALF 167
Figure 28—Model of a flip-flop with asynchronous clear in Verilog 167
Figure 29—Model of a flip-flop with asynchronous clear in VHDL 167
Figure 30—Concurrency for edge-sensitive sequential logic 172
Figure 31—Example of event sequence detection function 174
Figure 32—Arithmetic model 211
Figure 33—Illustration of extrapolation rules 232
Figure 34—General timing measurement or timing constraint 241
Figure 35—Illustration of time to peak using FROM statement 245
Figure 36—Illustration of time to peak using TO statement 246
Figure 37—Illustration of a piece-wise linear waveform 247
Figure 38—TIME and FREQUENCY in a waveform 248
Figure 39—RETAIN and DELAY 251
Figure 40—SETUP and HOLD 252
Figure 41—NOCHANGE, SETUP, and HOLD 252
Figure 42—RECOVERY and REMOVAL 253
Figure 43—THRESHOLD measurement definition 255
Figure 44—Schematic of a pulse generator 260
Figure 45—Timing diagram of a pulse generator 260
Figure 46—Timing diagram of a DRAM cycle 261
Figure 47—General representation of electrical models around a pin 264
Figure 48—Electrical models associated with input and output pins 265

Figure 49—Definition of noise margin 271
Figure 50—Timing diagram of a noisy signal 272
Figure 51—Separation between two noise pulses 273
Figure 52—Example for timing-dependent noise margin 275
Figure 53—Principle of noise propagation 276
Figure 54—Principle of signal propagation 276
Figure 55—Example of noise propagation 277
Figure 56—Example of noise rejection 278
Figure 57—Mathematical definitions for MEASUREMENT annotations 279
Figure 58—Illustration of LENGTH and DISTANCE 284
Figure 59—Illustration of REFERENCE for DISTANCE 285

List of Tables

Table 1—Target applications and models supported by ALF....2
Table 2—Categories of ALF statements....14
Table 3—Object references as annotation....19
Table 4—Cross-reference of lexical tokens....21
Table 5—List of whitespace characters....22
Table 6—Single bit constants....25
Table 7—Special characters in quoted strings....27
Table 8—Object keywords....29
Table 9—Built-in arithmetic function keywords....29
Table 10—Information annotation container....50
Table 11—CELLTYPE annotations for a CELL object....53
Table 12—Attributes within a CELL with CELLTYPE=memory....53
Table 13—Attributes within a CELL with CELLTYPE=block....54
Table 14—Attributes within a CELL with CELLTYPE=core....54
Table 15—Attributes within a CELL with CELLTYPE=special....55
Table 16—Predefined values for RESTRICT_CLASS....55
Table 17—SCAN_TYPE annotations for a CELL object....58
Table 18—SCAN_USAGE annotations for a CELL object....59
Table 19—BUFFERTYPE annotations for a CELL object....59
Table 20—DRIVERTYPE annotations for a CELL object....60
Table 21—VIEW annotations for a PIN object....65
Table 22—PINTYPE annotations for a PIN object....66
Table 23—DIRECTION annotations for a PIN object....66
Table 24—DIRECTION in combination with PINTYPE....67
Table 25—Fundamental SIGNALTYPE annotations for a PIN object....67
Table 26—Composite SIGNALTYPE annotations based on DATA....69
Table 27—Composite SIGNALTYPE annotations based on ADDRESS....69
Table 28—Composite SIGNALTYPE annotations based on CONTROL....69
Table 29—Composite SIGNALTYPE annotations based on ENABLE....70
Table 30—Composite SIGNALTYPE annotations based on CLOCK....71
Table 31—ACTION annotations for a PIN object....71
Table 32—ACTION applicable in conjunction with fundamental SIGNALTYPE values....72
Table 33—POLARITY annotations for a PIN....72
Table 34—POLARITY applicable in conjunction with fundamental SIGNALTYPE values....73
Table 35—DATATYPE annotations for a PIN object....73
Table 36—STUCK annotations for a PIN object....74
Table 37—DRIVETYPE annotations for a PIN object....76
Table 38—SCOPE annotations for a PIN object....77
Table 39—PULL annotations for a PIN object....77
Table 40—Attributes within a PIN object....78
Table 41—Attributes for pins of a memory....78
Table 42—Attributes for pins representing double-rail signals....78
Table 43—PIN attributes for memory BIST....79
Table 44—SIDE annotations for a PIN object....80

Table 45—Statements in ALF describing physical objects 96
Table 46—Items for LAYER description 97
Table 47—Items for VIA description 101
Table 48—Items for RULE description 104
Table 49—Items for ANTENNA description 109
Table 50—Annotations within VIOLATION 144
Table 51—Unary boolean operators 160
Table 52—Binary boolean operators 160
Table 53—Ternary operator 160
Table 54—Unary reduction operators 160
Table 56—Unary bitwise operators 161
Table 57—Binary bitwise operators 161
Table 55—Binary reduction operators 161
Table 58—Binary operators 162
Table 59—Case comparison operators 163
Table 60—Unary vector operators on bits 168
Table 61—Unary vector operators on bits or words 169
Table 62—Canonical binary vector operators 175
Table 63—Complex binary vector operators 176
Table 64—Operators for conditional vector expressions 179
Table 65—Operators for sequential logic 180
Table 66—Unary arithmetic operators 210
Table 67—Binary arithmetic operators 210
Table 68—Function arithmetic operators 211
Table 69—Generally applicable arithmetic submodels 216
Table 70—Submodels restricted to electrical modeling 216
Table 71—Submodels restricted to physical modeling 217
Table 72—Unnamed containers for arithmetic models 222
Table 73—UNIT annotation 228
Table 74—Timing measurements 234
Table 75—Timing constraints 234
Table 78—Analog measurements 235
Table 76—Generalized timing measurements 235
Table 77—Normalized measurements 235
Table 80—Abstract measurements 236
Table 79—Electrical components 236
Table 82—Environmental data 237
Table 83—Arithmetic models for layout data 237
Table 81—Discrete measurements 237
Table 84—Semantic meaning of SIZE 238
Table 85—Semantic meaning of WIDTH 238
Table 86—Semantic meaning of HEIGHT 239
Table 87—Semantic meaning of LENGTH 239
Table 88—Semantic meaning of AREA 239
Table 89—Semantic meaning of PERIMETER 239
Table 90—Semantic meaning of DISTANCE 240
Table 91—Semantic meaning of THICKNESS 240
Table 92—Semantic meaning of OVERHANG 240
Table 93—Semantic meaning of EXTENSION 240
Table 94—Range of time value depending on VECTOR 242

Table 95—Partially specified timing measurements and constraints 242
Table 96—Semantic interpretation of MEASUREMENT, TIME, or FREQUENCY annotation....250
Table 97—Predefined process names 262
Table 98—Predefined derating cases....262
Table 99—Direct association of models with a PIN....265
Table 100—External association of models with a PIN 266
Table 101—Relations between ENERGY and POWER 268
Table 102—Relations between FLUENCE and FLUX 269
Table 103—MEASUREMENT annotation 278
Table 104—CONNECT_RULE annotation 280
Table 105—Implications between connect rules 280
Table 106—Arguments for connectivity....282
Table 107—Boolean literals in non-interpolateable tables....282

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

1. Introduction

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

1.1 Motivation

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

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

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

1.2 Goals

The basic goals of the proposed library standard are

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

1.3 Target applications

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

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

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

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

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

Table 1—Target applications and models supported by ALF

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

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

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

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

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

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

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

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

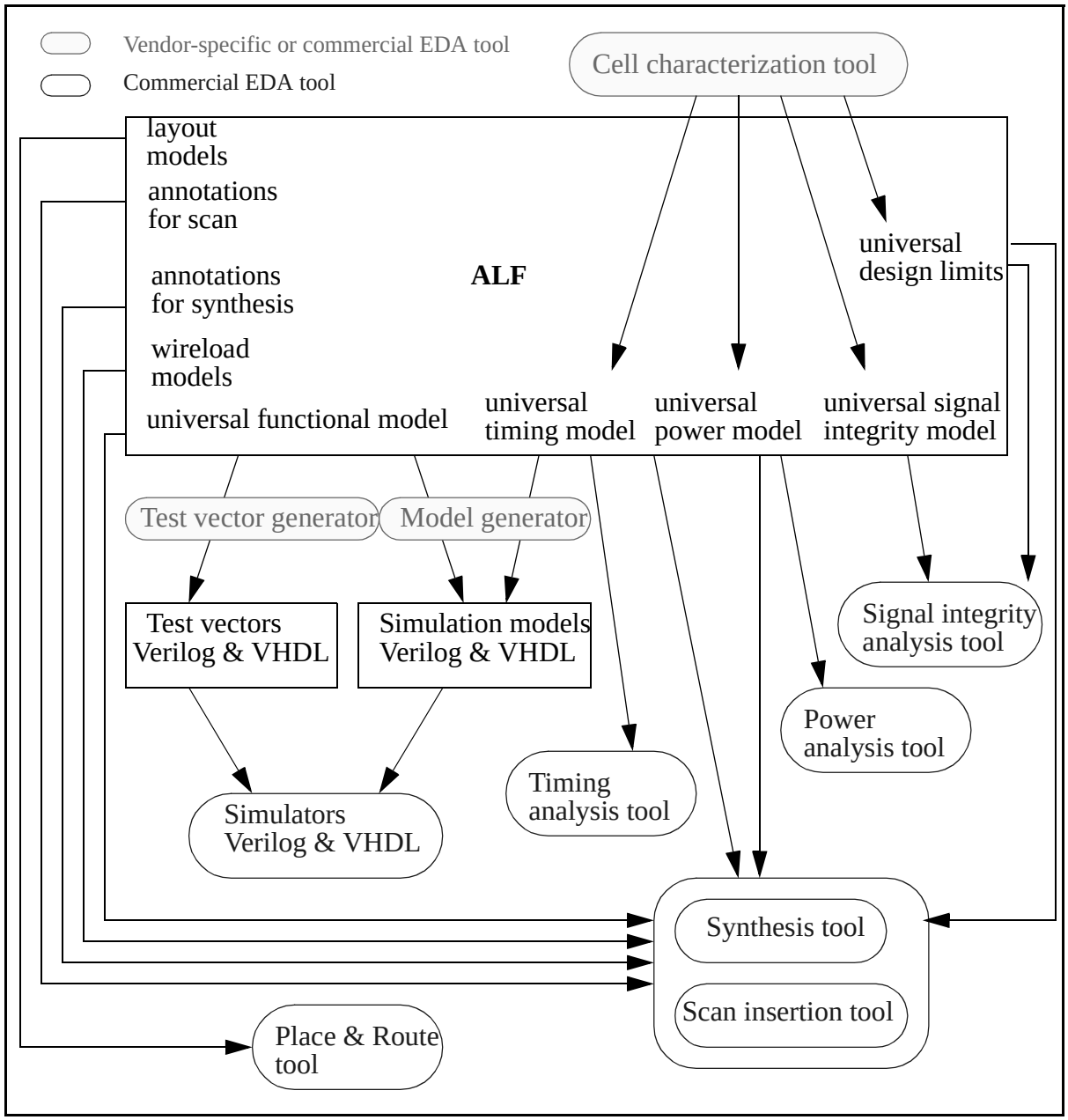


Figure 1—ALF and its target applications

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

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

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

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

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

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

1.4 Conventions

The syntax for description of lexical and syntax rules uses the following conventions.
Consider using the BNF nomenclature from IEEE 1481-1999

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

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

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

1.5 Contents of this standard

The organization of the remainder of this standard is

- Clause 2 (References) provides references to other applicable standards that are assumed or required for ALF.
- Clause 3 (Definitions) defines terms used throughout the different specifications contained in this standard.
- Clause 4 (Acronyms and abbreviations) defines the acronyms used in this standard.
- Clause 6 (Lexical rules) specifies the lexical rules.
- Clause 5 (ALF language construction principles) defines the language construction principles.
- Clause 7 (Auxiliary Syntax Rules) defines syntax and semantics of auxiliary items used in this standard.

— Clause 8 (Generic objects and related statements) defines syntax and semantics of generic objects used in this standard.
— Clause 9 (Library-specific objects and related statements) defines syntax and semantics of library-specific objects used in this standard.
— Clause 10 (Constructs for modeling of digital behavior) defines syntax and semantics of the control expression language used in this standard
— Clause 11 (Constructs for modeling of analog behavior) defines syntax and semantics of arithmetic models used in this standard.
— Annexes. Following Clause 11are a series of normative and informative annexes.

2. References

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

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

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

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

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

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

3. Definitions

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

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

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

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

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

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

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

3.6 ...

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

3.8 ...

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

3.10 ...

4. Acronyms and abbreviations

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

ALF	advanced library format, title of the herein proposed standard
ASIC	application specific integrated circuit
AWE	asymptotic waveform evaluation
BIST	built-in self test
BNF	Backus-Naur Form
CAE	computer-aided engineering [the term electronic design automation (EDA) is preferred]
CAM	content-addressable memory
CLF	Common Library Format from Avant! Corporation
CPU	central processing unit
DCL	Delay Calculation Language from IEEE 1481-1999 std
DEF	Design Exchange Format from Cadence Design Systems Inc.
DLL	delay-locked loop
DPCM	Delay and Power Calculation Module from IEEE 1481-1999 std
DPCS	Delay and Power Calculation System from IEEE 1481-1999 std
DSP	digital signal processor
DSPF	Detailed Standard Parasitic Format
EDA	electronic design automation
EDIF	Electronic Design Interchange Format
HDL	hardware description language
IC	integrated circuit
IP	intellectual property
ILM	Interface Logic Model from Synopsys Inc.
LEF	Library Exchange Format from Cadence Design Systems Inc.
LIB	Library Format from Synopsys Inc.
LSSD	level-sensitive scan design
MPU	micro processor unit
OLA	Open Library Architecture from Silicon Integration Initiative Inc.
PDEF	Physical Design Exchange Format from IEEE 1481-1999 std
PLL	Phase-locked loop
PVT	process/voltage/temperature (denoting a set of environmental conditions)
QTM	Quick Timing Model
RAM	random access memory
RC	resistance times capacitance
RICE	rapid interconnect circuit evaluator
ROM	read-only memory
RSPF	Reduced Standard Parasitic Format
RTL	Register Transfer Level
SDF	Standard Delay Format from IEEE 1497 std
SDC	Synopsys Design Constraint format from Synopsys Inc.
SPEF	Standard Parasitic Exchange Format from IEEE 1481-1999 std
SPF	Standard Parasitic Format
SPICE	Simulation Program with Integrated Circuit Emphasis
STA	Static Timing Analysis

STAMP	(STA Model Parameter ?) format from Synopsys Inc.
TCL	Tool Command Language (supported by multiple EDA vendors)
TLF	Timing Library Format from Cadence Design Systems Inc.
VCD	Value Change Dump format (from IEEE 1364 std ?)
VHDL	VHSIC Hardware Description Language
VHSIC	very-high-speed integrated circuit
VITAL	VHDL Initiative Towards ASIC Libraries from IEEE ??? std
VLSI	very-large-scale integration

5. ALF language construction principles and overview

Add lead-in text

This section presents the ALF language construction principles and gives an overview of the language features. The types of ALF statements and rules for parent/child relationships between types are presented summarily. Most of the types are associated with predefined keywords. The keywords in ALF shall be case-insensitive. However, uppercase is used for keywords throughout this section for clarity.

5.1 ALF meta-language

The following Syntax 1— establishes an ALF meta-language.

```
ALF_statement ::=
      ALF_type [ALF_name ] [ = ALF_value ] ALF_statement_termination
ALF_statement_termination ::=
      ;
    | { { ALF_value | : | ; } }
    | { { ALF_statement } }
ALF_type ::=
      non_escaped_identifier [ index ]
    | @
    | :
ALF_name ::=
      identifier [ index ]
    | control_expression
ALF_value ::=
      identifier
    | number
    | arithmetic_expression
    | boolean_expression
    | control_expression
```

Syntax 1—syntax construction for ALF meta-language

An *ALF statement* uses the delimiters "**;**", "**{**" and "**}**" to indicate its termination.

The *ALF type* is defined by a *keyword* (see Section 6.11 on page 31) eventually in conjunction with an *index* (see Section 7.7 on page 34) or by the *operator* "**@**" (Section 6.4 on page 24) or by the *delimiter* "**:**" (see Section 6.3 on page 23). The usage of keyword, index, operator, or delimiter as ALF type is defined by ALF language rules concerning the particular ALF type.

The *ALF name* is defined by an *identifier* (see Section 6.10 on page 29) eventually in conjunction with an index or by a *control expression* (see Section 10.9 on page 208). Depending on the ALF type, the ALF name is mandatory or optional or not applicable. The usage of identifier, index, or control expression as ALF name is defined by ALF language rules concerning the particular ALF type.

The *ALF value* is defined by an identifier, a *number* (see Section 6.5 on page 27), an *arithmetic expression* (see Section 11.1 on page 209), a *boolean expression* (see Section 10.7 on page 207), or a control expression. Depending on the type of the ALF statement, the ALF value is mandatory or optional or not applicable. The usage of identifier, number, arithmetic expression, boolean expression or control expression as ALF value is defined by ALF language rules concerning the particular ALF type.

An ALF statement can contain one or more other ALF statements. The former is called *parent* of the latter. Conversely, the latter is called *child* of the former. An ALF statement with child is called a *compound* ALF statement.

An ALF statement containing one or more ALF values, eventually interspersed with the delimiters "**;**" or "**:**", is called a *semi-compound* ALF statement. The items between the delimiters "**{**" and "**}**" are called *contents* of the ALF statement. The usage of the delimiters "**;**" or "**:**" within the contents of an ALF statement is defined by ALF language rules concerning the particular ALF statement.

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

Examples

a) ALF statement describing an unnamed object without value:

```
ARBITRARY_ALF_TYPE {
    // put children here
}
```

b) ALF statement describing an unnamed object with value:

```
ARBITRARY_ALF_TYPE = arbitrary_ALF_value;
```

or

```
ARBITRARY_ALF_TYPE = arbitrary_ALF_value {
    // put children here
}
```

c) ALF statement describing a named object without value:

```
ARBITRARY_ALF_TYPE arbitrary_ALF_name;
```

or

```
ARBITRARY_ALF_TYPE arbitrary_ALF_name {
    // put children here
}
```

d) ALF statement describing a named object with value:

```
ARBITRARY_ALF_TYPE arbitrary_ALF_name = arbitrary_ALF_value;
```

or

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

5.2 Categories of ALF statements

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

Statements are divided into the following categories: *generic object*, *library-specific object*, *arithmetic model*, *arithmetic submodel*, *arithmetic model container*, *geometric model*, *annotation*, *annotation container*, and *auxiliary statement*, as shown in Table 2—.

Table 2—Categories of ALF statements

category	purpose	syntax particularity
generic object	provide a definition for use within other ALF statements	Statement is atomic, semi-compound or compound. Name is mandatory. Value is either mandatory or not applicable.
library-specific object	describe the contents of a IC technology library	Statement is atomic or compound. Name is mandatory. Value does not apply. Category of parent is exclusively *library-specific object*
arithmetic model	describe an abstract mathematical quantity that can be calculated and eventually measured within the design of an IC	Statement is atomic or compound. Name is optional. Value is mandatory, if atomic.
arithmetic submodel	describe an arithmetic model under a specific measurement condition	Statement is atomic or compound. Name does not apply. Value is mandatory, if atomic. Category of parent is exclusively *arithmetic model*
arithmetic model container	provide a context for an arithmetic model	Statement is compound. Name and value do not apply. Category of child is exclusively *arithmetic model*
geometric model	describe an abstract geometrical form used in physical design of an IC	Statement is semi-compound or compound. Name is optional. Value does not apply.
annotation	provide a qualifier or a set of qualifiers for an ALF statement	Statement is atomic, semi-compound or compound. Name does not apply. Value is mandatory, if atomic or compound. Value does not apply, if semi-compound. Category of child is exclusively *annotation*
annotation container	provide a context for an annotation	Statement is compound. Name and value do not apply. Category of child is exclusively *annotation*
auxiliary statement	provide an additional description within the context of a library-specific object, an arithmetic model, an arithmetic submodel, geometric model or another auxliary statement	dependent on subcategory

The following Figure 2— illustrates the parent/child relationship between categories of statements.

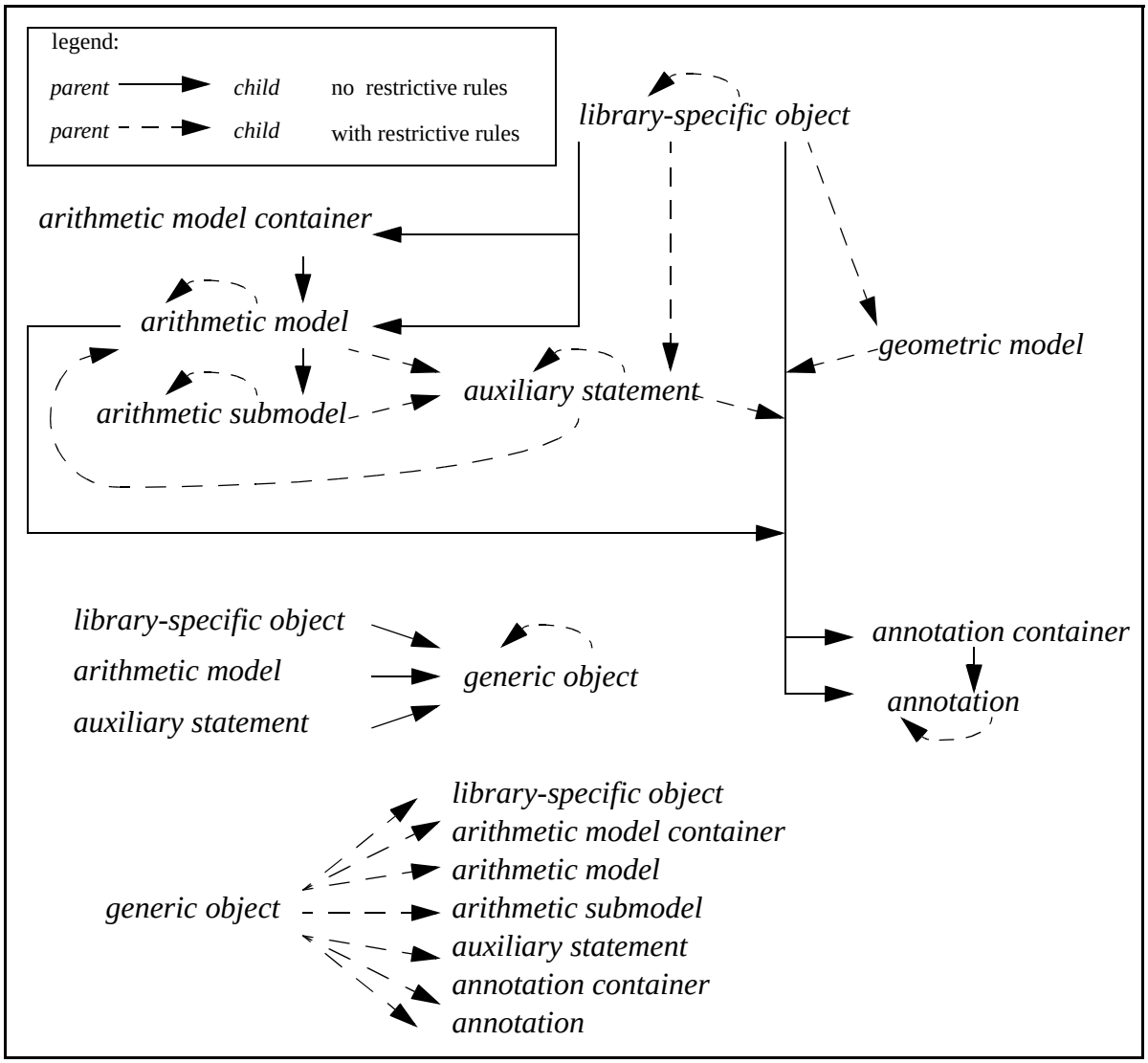


Figure 2—Parent/child relationship between ALF statements

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

5.3 Generic objects and library-specific objects

Statements with mandatory name are called *objects*, i.e., *generic object* and *library-specific object*.

The following table lists the keywords and items in the category *generic object*. The keywords used in this category are called *generic keywords.*

Table 3—Generic objects

keyword	item	section
ALIAS	alias declaration	
CONSTANT	constant declaration	
CLASS	class declaration	
GROUP	group declaration	
KEYWORD	keyword declaration	
TEMPLATE	template declaration	

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

Table 4—Library-specific objects

keyword	item	section
LIBRARY	library	
SUBLIBRARY	sublibrary	
CELL	cell	
PRIMITIVE	primitive	
WIRE	wire	
PIN	pin	
PINGROUP	pin group	
VECTOR	vector	
NODE	node	
LAYER	layer	
VIA	via	
RULE	rule	
ANTENNA	antenna	
SITE	site	

Table 4—Library-specific objects

keyword	item	section
ARRAY	array	
BLOCKAGE	blockage	
PORT	port	
PATTERN	pattern	
REGION	region	new proposal for IEEE

The following Figure 3— illustrates the parent/child relationship between statements within the category *library-specific object*.

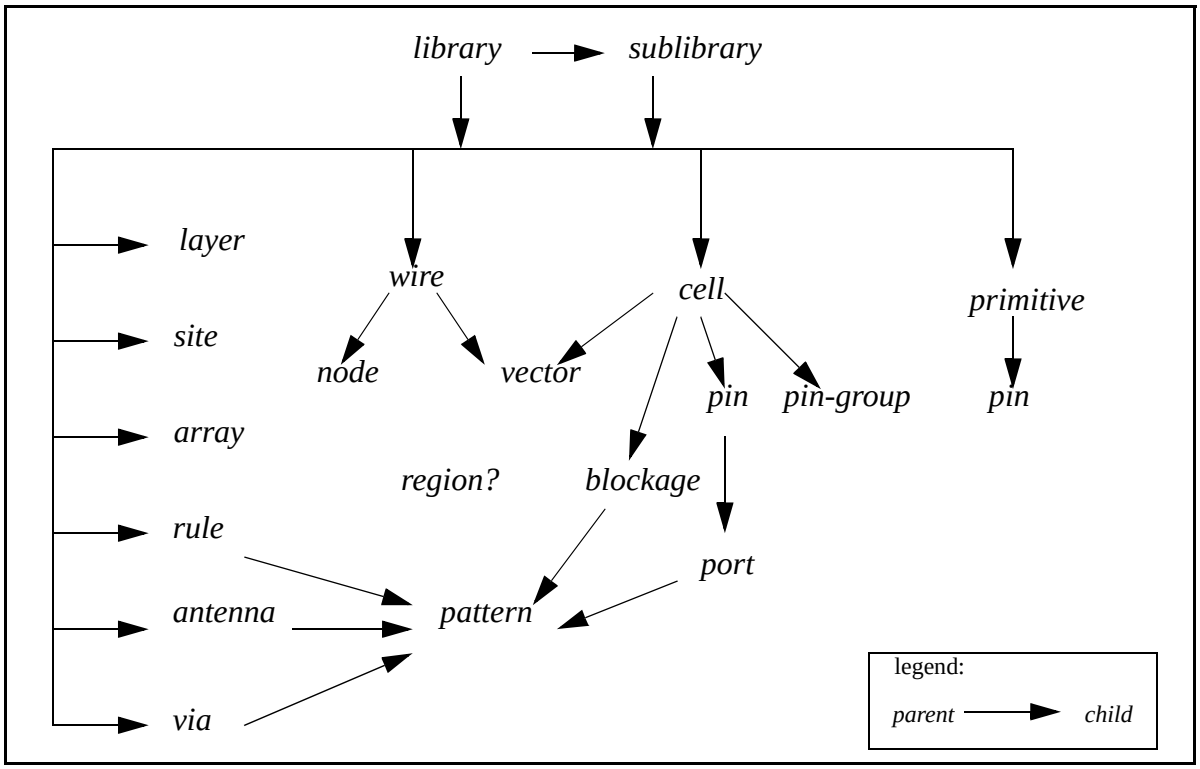


Figure 3—Parent/child relationship amongst library-specific objects

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

5.4 Singular statements and plural statements

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

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

The following Table 5— lists the singular statements.

Table 5—Singular statements

keyword	item	value	complexity	section
FUNCTION	function	N/A	compound	
TEST	test	N/A	compound	
RANGE	range	N/A	semi-compound	
FROM	from	N/A	compound	
TO	to	N/A	compound	
VIOLATION	violation	N/A	compound	
HEADER	header	N/A	compound (or semi-compound?)	
TABLE	table	N/A	semi-compound	
EQUATION	equation	N/A	semi-compound	
BEHAVIOR	behavior	N/A	compound	
STRUCTURE	structure	N/A	compound	
NON_SCAN_CELL	non-scan cell	optional	compound or semi-compound	
ARTWORK	artwork	mandatory	compound or atomic	
PULL	pull	optional	compound or atomic	

The following Table 6— lists the plural statements.

Table 6—Plural statements

keyword	item	name	complexity	section
STATETABLE	state table	optional	semi-compound	
@	control statement	mandatory	compound	
:	alternative control statement	mandatory	compound	

The following Figure 4— illustrates the parent/child relationship for singular statements and plural statements.

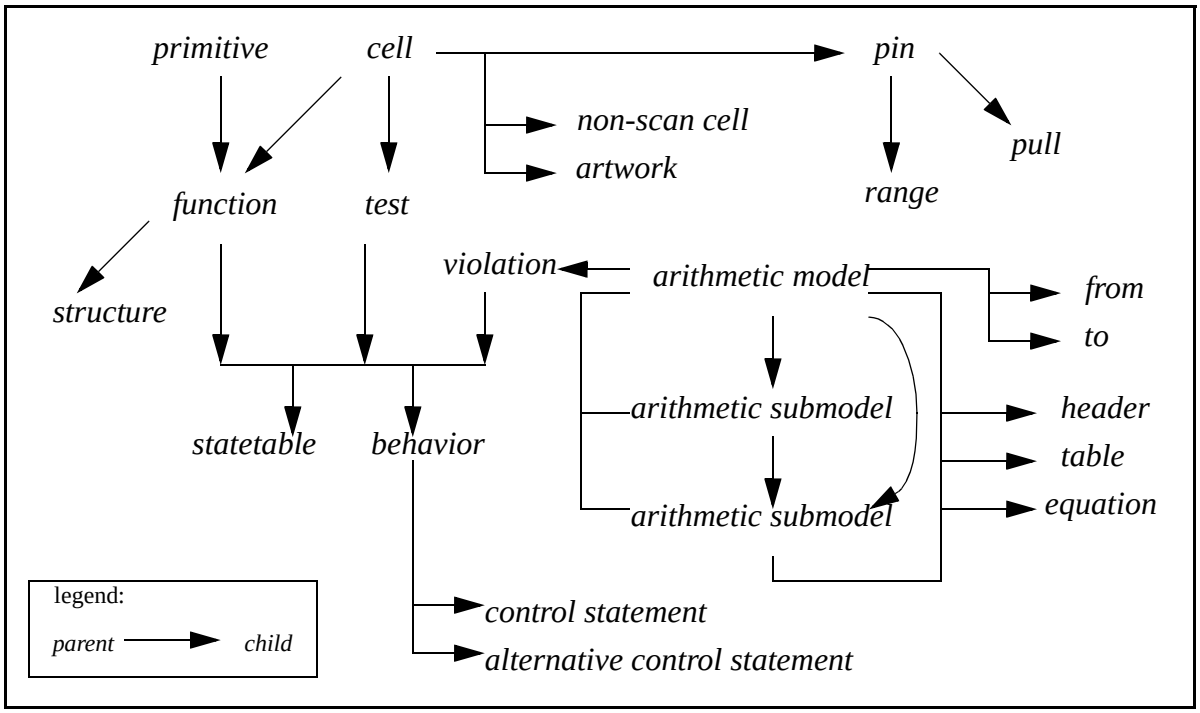


Figure 4—Parent/child relationship involving singular statements and plural statements

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

5.5 Instantiation statement and assignment statement

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

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

The following Table 7— lists the instantiation statements.

Table 7—Instantiation statements

item	name	value	section
cell instantiation	optional	N/A	
primitive instantiation	optional	N/A	
template instantiation	N/A	optional	
via instantiation	mandatory	N/A	
wire instantiation	mandatory	N/A	proposed for IEEE

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

The following Table 8— lists the assignment statements.

Table 8—Assignment statements

item	section
pin assignment	
boolean assignment	
arithmetic assignment	

The following Figure 5— illustrates the parent/child relationship involving instantiation and assigment statements.

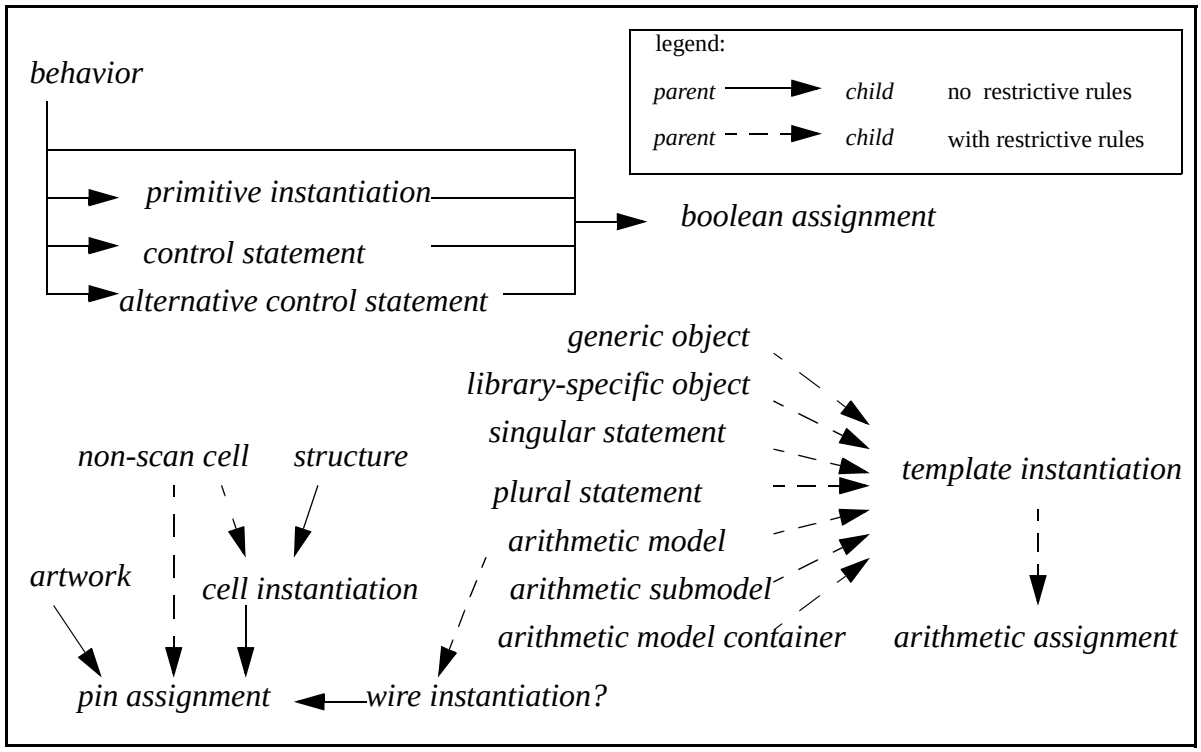


Figure 5—Parent/child relationship involving instantiation and assignment statements

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

5.6 Annotation, arithmetic model, and related statements

Multiple keywords are predefined in the categories *arithmetic model*, *arithmetic model container*, *arithmetic submodel*, *annotation*, *annotation container*, and *geometric model*. Their semantics are established within the context of their parent. Therefore they are called *context-sensitive keywords*. In addition, the ALF language allows additional definition of keywords in these categories.

The following Table 9— provides a reference to sections where more definitions about these categories can be found.

Table 9—Other categories of ALF statements

item	section
arithmetic model	
arithmetic submodel	
arithmetic model container	
annotation	
annotation container	
geometric model	

There exist predefined keywords with generic semantics in the category *annotation* and *annotation container.* They are called *generic keywords*, like the keywords for *generic objects*.

The following Table 10— lists the generic keywords in the category annotation and annotation container.

Table 10—Annotations and annotation containers with generic keyword

keyword	item / subcategory	section
PROPERTY	one_level_annotation_container	
ATTRIBUTE	multi_value_annotation	
INFORMATION	one_level_annotation_container	

The following Table 11— lists predefined keywords in categories related to arithmetic model..

Table 11—Keywords related to arithmetic model

keyword	item / category	section
LIMIT	arithmetic model container	

Table 11—Keywords related to arithmetic model

keyword	item / category	section
MIN	arithmetic submodel, operator within *arithmetic expression*	
MAX	arithmetic submodel, operator within *arithmetic expression*	
TYP	arithmetic submodel	
DEFAULT	arithmetic submodel, annotation	
ABS	operator within *arithmetic expression*	
EXP	operator within *arithmetic expression*	
LOG	operator within *arithmetic expression*	

The definitions of other predefined keywords, especially in the category arithmetic model, can be self-described in ALF using the *keyword declaration* statement (see Section 8.4 on page 38).

5.7 Statements for parser control

The following provides a reference to statements used for ALF parser control.

Table 12—Statements for ALF parser control

keyword	statement	section
INCLUDE	include statement	
ALF_REVISION	revision statement	

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

5.8 Name space and visibility of statements

The following rules for name space and visibility shall apply:

a) A statement shall be visible within its parent statement.
b) A statement visible within another statement shall also be visible within a child of that other statement.
c) All objects (i.e., generic objects and library-specific objects) shall share a common name space within their scope of visibility. No object shall use the same name as any other visible object. Conversely, an object may use the same name as any other object outside the scope of its visibility.
d) Exceptions of rule c) may be allowed for specific objects and with specific semantic implications.
e) All statements with optional names (i.e., property, arithmetic model, geometric model) shall share a common name space within their scope of visibility. No statement with optional name shall use the same name as any other visible statement with optional name. Conversely, a statement may use the same optional name as any other statement with optional name outside the scope of its visibility.

6. Lexical rules

This section discusses the lexical rules.

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

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

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

6.1 Character set

This standard shall use the ASCII character set [ISO 8859-1 : 1987(E)].

The ASCII character set shall be divided into the following categories: *whitespace*, *letter*, *digit*, and *special*, as shown in Syntax 3.

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

Syntax 3—ASCII character

The following Table 4 shows the list of *whitespace* characters and their ASCII code.

Table 4—List of whitespace characters

Name	ASCII code (octal)
space	200
horizontal tab	011
new line	012
vertical tab	013

Table 4—List of whitespace characters (Continued)

Name	ASCII code (octal)
form feed	014
carriage return	015

The following Table 5— shows the list of *special* characters and their names used in this standard

Table 5—List of special characters

Symbol	Name
&	amperesand
|	??? bar
^	??? hyphen
~	tilde
+	plus
-	minus
*	asterix
/	divider
%	percent
?	question mark
!	exclamation mark
:	colon
;	semicolon
,	comma
”	double quote
’	single quote
@	??? at
=	equal
\	escape character
.	dot
$	dollar
_	underscore
#	??? sharp
(|)	parenthesis (open | close)
< | >	angular bracket (open | close)

Table 5—List of special characters (Continued)

Symbol	Name
[|]	square bracket (open | close)
{ | }	curly brace (open | close)

6.2 Comment

A *comment* shall be divided into the subcategories *in-line comment* and *block comment*, as shown in Syntax 4.

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

Syntax 4—Comment

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

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

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

6.3 Delimiter

The special characters shown in Syntax 5 shall be considered *delimiters*.

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

Syntax 5—Delimiter

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

6.4 Operator

Operators shall be divided into the following subcategories: *arithmetic operator*, *boolean operator*, *relational operator*, *shift operator*, *event sequence operator*, and *meta operator*, as shown in Syntax 6

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

Syntax 6—Operator

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

6.4.1 Arithmetic operator

The following Table 6— shows the list of arithmetic operators and their names used in this standard.

Table 6—List arithmetic operators

Symbol	Operator name	unary / binary	section
+	plus	binary	
-	minus	both	
*	multiply	binary	
/	divide	binary	
%	modulo	binary	
**	power	binary	

Arithmetic operators shall be used to specify arithmetic operations.

6.4.2 Boolean operator

The following Table 7— shows the list of boolean operators and their names used in this standard.

Table 7—List of boolean operators

Symbol	Operator name	unary / binary	section
!	logical invert	unary	
&&	logical and	binary	
||	logical or	binary	
~	vector invert	unary	
&	vector and	both	
~&	vector nand	both	
|	vector or	both	
~|	vector nor	both	
^	exclusive or	both	
~^	exclusive nor	both	

Boolean operators shall be used to specify boolean operations.

6.4.3 Relational operator

The following Table 8— shows the list of relational operators and their names used in this standard.

Table 8—List of relational operators

Symbol	Operator name	unary / binary	section
==	equal	binary	
!=	not equal	binary	
>	greater	binary	
<	lesser	binary	
>=	greater or equal	binary	
<=	lesser or equal	binary	

Relational operators shall be used to specify mathematical relationships between numerical quantities.

6.4.4 Shift operator

The following Table 9— shows the list of shift operators and their names used in this standard.

Table 9—List of shift operators

Symbol	Operator name	unary / binary	section
<<	shift left	binary	
>>	shift right	binary	

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

6.4.5 Event sequence operator

The following Table 10— shows the list of event sequence operators and their names used in this standard.

Table 10—List of event sequence operators

Symbol	Operator name	unary / binary	section
->	immediately followed by	binary	
~>	eventually followed by	binary	
<->	immediately following each other	binary	
<~>	eventually following each other	binary	
&>	simultaneous or immediately followed by	binary	
<&>	simultaneous or immediately following each other	binary	

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

6.4.6 Meta operator

The following Table 11— shows the list of meta operators and their names used in this standard.

Table 11—List of meta operators

Symbol	Operator name	unary / binary	section
=	assignment	binary	
?	condition	binary	
@	control	unary	

Meta operators shall be used to specify transactions between variables.

6.5 Number

Numbers shall be divided into subcategories *signed number* and *unsigned number*, as shown in Syntax 7.

```
number ::=
      signed_number | unsigned_number
signed_number ::=
      signed_integer | signed_real
signed_integer ::=
      sign unsigned_integer
signed_real ::=
      sign unsigned_real
unsigned_number ::=
      unsigned_integer | unsigned_real
unsigned_integer ::=
      digit  { [  _  ] digit }
unsigned_real ::=
      unsigned  .  unsigned
    | unsigned [ . unsigned ] E [ sign ] unsigned
    | unsigned [ . unsigned ] e [ sign ] unsigned
sign ::=
      +  |  -
```

Syntax 7—Signed and unsigned numbers

Alternatively, numbers shall be divided into subcategories *integer* and *real*, as shown in Syntax 8—.

```
number ::=
      integer | real
integer ::=
      signed_integer | unsigned_integer
real ::=
      signed_real | unsigned_real
```

Syntax 8—Integer and real numbers

Numbers shall be used to represent numerical quantities.

6.6 Bit literal

Bit literals shall be divided into subcategories *numeric bit literal* and *symbolic bit literal*, as shown in Syntax 9.

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

Syntax 9—Bit literal

Bit literals shall be used to specify scalar values within a boolean system.

6.7 Based literal

Based literals shall be divided into subcategories *binary based literal*, *octal based literal*, *decimal based literal*, and *hexadecimal based literal*, as shown in Syntax 10.

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

Syntax 10—Based literal

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

6.8 Edge literal

Edge literals shall be divided into subcategories *bit edge literal*, *based edge literal*, and *symbolic edge literal*, as shown in Syntax 11—.

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

Syntax 11—Edge literal

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

6.9 Quoted string

A *quoted string* shall be a sequence of zero or more characters enclosed between two double quote characters, as shown in Syntax 12.

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

Syntax 12—Quoted string

Within a quoted string, a sequence of characters starting with an *escape character* shall represent a symbol for another character, as shown in Table 12.

Table 12—Character symbols within a quoted string

Symbol	Character	ASCII Code (octal)
\g	Alert or bell	007
\h	Backspace	010
\t	Horizontal tab	011
\n	New line	012
\v	Vertical tab	013
\f	Form feed	014
\r	Carriage return	015
\"	Double quote	042
\\	Escape character	134
\ digit digit digit	ASCII character represented by three digit octal ASCII code	digit digit digit

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

6.10 Identifier

Identifiers shall be divided into the subcategories *non-escaped identifier, escaped identifier, placeholder identifier,* and *hierarchical identifier,* as shown in Syntax 13.

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

Syntax 13—Identifier

Identifiers shall be used to specify a name of an ALF statement or a value of an ALF statement. Identifiers may also appear in an arithmetic expression, in a boolean expression, or in a vector expression, referencing an already defined statement by name.

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

6.10.1 Non-escaped identifier

A *non-escaped identifier* shall be defined as shown in Syntax 14.

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

Syntax 14—Non-escaped identifier

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

6.10.2 Escaped identifier

An *escaped identifier* shall be defined as shown in Syntax 15.

```
escaped_identifier ::=
    escape_character escapable_character { escapable_character }
escapable_character ::=
    letter | digit | special
```

Syntax 15—Escaped identifier

An escaped identifier shall be used, when there is a lexical conflict, i.e., an appearence of a character with special meaning, or a semantical conflict, i.e., the identifier is used elsewhere as a keyword.

6.10.3 Placeholder identifier

A *placeholder identifier* shall be defined as a non-escaped identifier enclosed by angular brackets without whitespace, as shown in Syntax 16.

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

Syntax 16—Placeholder identifier

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

6.10.4 Hierarchical identifier

A *hierarchical identifier* shall be defined as shown in Syntax 17.

```
hierarchical_identifier ::=
    identifier [ \ ] . identifier
```

Syntax 17—Hierarchical identifier

A hierarchical identifier shall be used to specify a hierarchical name of a statement, i.e., the name of a child preceded by the name of its parent. A dot within a hierarchical identifier shall be used to separate a parent from a child, unless the dot is directly preceded by an escape character.

Example

\id1.id2.\id3 is a hierarchical identifier, where **id2** is a child of **\id1**, and **\id3** is a child of **id2**.

id1\.id2.\id3 is a hierarchical identifier, where **\id3** is a child of "**id1.id2**".

id1\.id2\.id3 specifies the pseudo-hierarchical name "**id1.id2.id3**".

6.11 Keyword

Keywords shall be lexically equivalent to non-escaped identifiers. Predefined keywords are listed in Table 3—, Table 4—, Table 5—, Table 6—, Table 10—, and Table 11—. Additional keywords are predefined in section ...

The predefined keywords in this standard follow a more restrictive lexical rule than general non-escaped identifiers, as shown in Syntax 18—.

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

Syntax 18—Keyword

**Should this be a normative rule or a recommended practice to follow for additional keyword definitions? **

Note: This document presents keywords in all-uppercase letters for clarity.

6.12 Rules for whitespace usage

Whitespace shall be used to separate lexical tokens from each other, according to the following rules:

a) Whitespace before and after a *delimiter* shall be optional.
b) Whitespace before and after an *operator* shall be optional.
c) Whitespace before and after a *quoted string* shall be optional.
d) Whitespace before and after a *comment* shall be mandatory. This rule shall override a), b), and c).
e) Whitespace between subsequent quoted strings shall be mandatory. This rule shall override c).
f) Whitespace between subsequent lexical tokens amongst the categories *number*, *bit literal*, *based literal*, and *identifier* shall be mandatory.
g) Whitespace before and after a *placeholder identifier* shall be mandatory. This rule shall override a), b), and c).
h) Whitespace after an *escaped identifier* shall be mandatory. This rule shall override a), b), and c).
i) Either whitespace or delimiter before a *signed number* shall be mandatory. This rule shall override a), b), and c).
j) Either whitespace or delimiter before a *symbolic edge literal* shall be mandatory. This rule shall override a), b), and c).

Whitespace before the first lexical token or after the last lexical token in a file shall be optional. Hence in all rules prescribing mandatory whitespace, “before” shall not apply for the first lexical token in a file, and “after” shall not apply for the last lexical token in a file.

6.13 Rules against parser ambiguity

In a syntax rule where multiple legal interpretations of a lexical token are possible, the resulting ambiguity shall be resolved according to the following rules:

a) In a context where both *bit literal* and *identifier* are legal, a *non-escaped identifier* shall take priority over a *symbolic bit literal*.
b) In a context where both *bit literal* and *number* are legal, an *unsigned integer* shall take priority over a *numeric bit literal*.
c) In a context where both *edge literal* and *identifier* are legal, a *non-escaped identifier* shall take priority over a *bit edge literal*.
d) In a context where both *edge literal* and *number* are legal, an *unsigned integer* shall take priority over a *bit edge literal*.

If the interpretation as *bit literal* is desired in case a) or b), a *based literal* can be substituted for a *bit literal*.

If the interpretation as *edge literal* is desired in case c) or d), a *based edge literal* can be substituted for a *bit edge literal*.

7. Auxiliary Syntax Rules

This section specifies auxiliary syntax rules which are used to build other syntax rules.

7.1 All-purpose value

An *all-purpose value* shall be defined as shown in Syntax 25.

```
all_purpose_value ::=
      number
    | identifier
    | quoted_string
    | bit_literal
    | based_literal
    | edge_value
    | pin_variable
    | control_expression
```

Syntax 25—All purpose value

7.2 String

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

```
string ::=
      quoted_string | identifier
```

Syntax 26—String value

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

7.3 Arithmetic value

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

```
arithmetic_value ::=
      number | identifier | bit_literal | based_literal
```

Syntax 27—Arithmetic value

An arithmetic value shall represent data for an arithmetic model or for an arithmetic assignment. Semantic restrictions apply, depending on the particular type of arithmetic model.

7.4 Boolean value

A *boolean value* shall be defined as shown in Syntax 28.

A boolean value shall represent the contents of a pin variable (see Section 7.8 on page 34).

```
boolean_value ::=
    bit_literal | based_literal | unsigned_integer
```

Syntax 28—Boolean value

7.5 Edge value

An *edge value* shall be defined as shown in Syntax 29.

```
edge_value ::=
  ( edge_literal )
```

Syntax 29—Edge value

An edge value shall represent a standalone edge literal that is not embedded in a vector expression.

7.6 Index value

An *index value* shall be defined as shown in Syntax 30.

```
index_value ::=
    unsigned_integer | identifier
```

Syntax 30—Index value

An index value shall represent a particular position within a *vector pin* (see). The usage of identifier shall only be allowed, if that identifier represents a *constant* (see Section 8.2) with a value of the category unsigned integer.

7.7 Index

An *index* shall be defined as shown in Syntax 31.

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

Syntax 31—Index

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

7.8 Pin variable

A *pin variable* shall be defined as shown in Syntax 32.

```
pin_variable ::=
    pin_variable_identifier [ index ]
```

Syntax 32—Pin variable

A pin variable shall represent the name of a pin or the name of a pingroup, in conjunction with an optional index.

7.9 Pin assignment

A *pin assignment* shall be defined as shown in Syntax 33.

```
pin_assignment ::=
      pin_variable = boolean_value ;
    | pin_variable = pin_variable ;
```

Syntax 33—Pin assignment

A pin assignment shall represent an association between a pin variable and another pin variable or a boolean value.

The datatype of the left hand side (LHS) and the right hand side (RHS) of the assignment must be compatible with each other. The following rules shall apply:

a) The bitwidth of the RHS must be equal to the bitwidth of the LHS.
b) A scalar pin at the LHS may be assigned a bit literal or a based literal representing a single bit.
c) A pin group, a one-dimensional vector pin, or a one-dimensional slice of a two-dimensional vector pin at the LHS may be assigned a based literal or an unsigned integer, representing a binary number.

7.10 Annotation

An *annotation* shall be divided into the subcategories *single value annotation* and *multi value annotation*, as shown in Syntax 34

An annotation shall represent an assocation between an identifier and a set of *annotation values* (*values* for shortness). In case of a single value annotation, only one value shall be legal. In case of a multi value annotation, one or more values shall be legal. The annotation shall serve as a semantic qualifier of its parent statement. The value shall be subject to semantic restrictions, depending on the identifier.

The annotation identifier may be a keyword used for the declaration of an object (i.e., a generic object or a library-specific object). An annotation using such an annotation identifier shall be called a *reference annotation*. The annotation value of a reference annotation shall be the name of an object of matching type. A reference annotation may be a single-value annotation or a multi-value annotation. The semantic meaning of a reference annotation shall be defined in the context of its parent statement.

7.11 Annotation container

An *annotation container* shall be defined as shown in Syntax 34

```
annotation ::=
     single_value_annotation
   | multi_value_annotation
single_value_annotation ::=
   annotation_identifier = annotation_value ;
multi_value_annotation ::=
   annotation_identifier { annotation_value { annotation_value } }
annotation_value ::=
     number
   | identifier
   | quoted_string
   | bit_literal
   | based_literal
   | edge_value
   | pin_variable
   | control_expression
   | boolean_expression
   | arithmetic_expression
```

Syntax 34—Annotation

```
annotation_container ::=
     annotation_container_identifier { annotation { annotation } }
```

Syntax 35—Annotation container

An annotation container shall represent a collection of annotations. The annotation container shall serve as a semantic qualifier of its parent statement. The annotation container identifier shall be a keyword. An annotation within an annotation container shall be subject to semantic restrictions, depending on the annotation container identifier.

7.12 ATTRIBUTE statement

An *attribute* statement shall be defined as shown in Syntax 36.

```
attribute ::=
   ATTRIBUTE { identifier { identifier } }
```

Syntax 36—ATTRIBUTE statement

The attribute statement shall be used to associate arbitrary identifiers with the parent of the attribute statement. Semantics of such identifiers may be defined depending on the parent of the attribute statement. The attribute statement has a similar syntax definition as a multi-value annotation (see Section 7.10). While a multi-value annotation may have restricted semantics and a restricted set of applicable values, identifiers with and without predefined semantics may co-exist within the same attribute statement.

Example

```
CELL myRAM8x128 {
     ATTRIBUTE { rom asynchronous static }
}
```

7.13 PROPERTY statement

A *property* statement shall be defined as shown in Syntax 37.

property ::=
PROPERTY [identifier] **{** annotation { annotation } **}**

Syntax 37—PROPERTY statement

The property statement shall be used to associate arbitrary annotations with the parent of the property statement. The property statement has a similar syntax definition as an annotation container (see Section 7.11). While the keyword of an annotation container usually restricts the semantics and the set of applicable annotations, the keyword "property" does not. Annotations shall have no predefined semantics, when they appear within the property statement, even if annotation identifiers with otherwise defined semantics are used.

Example

```
PROPERTY myProperties {
     parameter1 = value1 ;
     parameter2 = value2 ;
     parameter3 { value3 value4 value5 }
}
```

7.14 INCLUDE statement

An *include* statement shall be defined as shown in Syntax 38.

include ::=
INCLUDE quoted_string **;**

Syntax 38—INCLUDE statement

The quoted string shall specify the name of a file. When the include statement is encountered during parsing of a file, the application shall parse the specified file and then continue parsing the former file. The format of the file containing the include statement and the format of the file specified by the include statement shall be the same.

Example

```
LIBRARY myLib {
     INCLUDE "templates.alf";
     INCLUDE "technology.alf";
     INCLUDE "primitives.alf";
     INCLUDE "wires.alf";
     INCLUDE "cells.alf";
}
```

The filename specified by the quoted string shall be interpreted according to the rules of the application and/or the operating system. The ALF parser itself shall make no semantic interpretation of the filename.

7.15 REVISION statement

A *revision statement* shall be defined as shown in Syntax 34

```
revision ::=
    ALF_REVISION string_value
```

Syntax 39—Revision statement

A revision statement shall be used to identify the revision or version of the file to be parsed. One, and only one, revision statement may appear at the beginning of an ALF file.

The set of legal string values within the revision statement shall be defined as shown in Table 10

Table 10—Legal string values within the REVISION statement

string value	revision or version
“1.1”	Version 1.1 by Open Verilog International, released on April 6, 1999
“2.0”	Version 2.0 by Accellera, released on December 14, 2000
“P1603.2002-04-16”	IEEE draft version as described in this document
TBD	IEEE 1603 release version

The revision statement shall be optional, as the application program parsing the ALF file may provide other means of specifying the revision or version of the file to be parsed. If a revision statement is encountered while a revision has already been specified to the parser (e.g. if an included file is parsed), the parser shall be responsible to decide whether the newly encountered revision is compatible with the originally specified revision and then either proceed assuming the original revision or abandon.

This document suggests, but does not certify, that the IEEE version of the ALF standard proposed herein be backward compatible with the Accellera version 2.0 and the OVI version 1.1.

7.16 Generic object

A *generic object* shall be defined as shown in Syntax 40.

```
generic_object ::=
      alias_declaration
    | constant_declaration
    | class_declaration
    | keyword_declaration
    | group_declaration
    | template_declaration
    | generic_object_template_instantiation
```

Syntax 40—Generic object

7.17 Library-specific object

A *library-specific object* shall be defined as shown in Syntax 41.

```
library_specific_object ::=
      library
    | sublibrary
    | cell
    | primitive
    | wire
    | pin
    | pingroup
    | vector
    | node
    | layer
    | via
    | rule
    | antenna
    | site
    | array
    | blockage
    | port
    | pattern
    | region
    | library_specific_object_template_instantiation
```

Syntax 41—Library-specific object

7.18 All purpose item

An *all purpose item* shall be defined as shown in Syntax 42.

```
all_purpose_item ::=
      generic_object
    | include_statement
    | annotation
    | annotation_container
    | arithmetic_model
    | arithmetic_model_container
    | all_purpose_item_template_instantiation
```

Syntax 42—All purpose item

8. Generic objects and related statements

Add lead-in text

8.1 ALIAS declaration

An *alias* shall be declared as shown in Syntax 33.

```
alias_declaration ::=
    ALIAS alias_identifier = original_identifier ;
```

Syntax 33—ALIAS declaration

The alias declaration shall specify an identifier which may be used instead of an original identifier to specify a name or a value of an ALF statement. The identifier shall be semantically interpreted in the same way as the original identifier.

Example

```
ALIAS reset = clear;
```

8.2 CONSTANT declaration

A *constant* shall be declared as shown in Syntax 34.

```
constant_declaration ::=
    CONSTANT constant_identifier = constant_value ;
constant_value ::=
    number | based_literal
```

Syntax 34—CONSTANT declaration

The constant declaration shall specify an identifier which can be used instead of a *constant value*, i.e., a number or a based literal. The identifier shall be semantically interpreted in the same way as the constant value.

Example

```
CONSTANT vdd = 3.3;
CONSTANT opcode = 'h0f3a;
```

8.3 CLASS declaration

A *class* shall be declared as shown in Syntax 35.

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

Syntax 35—CLASS declaration

A class declaration shall be used to establish a semantic association between ALF statements, including, but not restricted to, other class declarations. ALF statements shall be associated with each other, if they contain a reference to the same class. The semantics specified by an all purpose item within a class declaration shall be inherited by the statement containing the reference.

Example

```
CLASS \1stclass { ATTRIBUTE { everything } }
CLASS \2ndclass { ATTRIBUTE { nothing } }
CELL cell1 { CLASS = \1stclass; }
CELL cell2 { CLASS = \2ndclass; }
CELL cell3 { CLASS { \1stclass \2ndclass } }
// cell1 inherits “everything”
// cell2 inherits “nothing”
// cell3 inherits “everything” and “nothing”
```

8.4 KEYWORD declaration

A *keyword* shall be declared as shown in Syntax 36.

keyword_declaration ::=
KEYWORD *keyword*_identifier **=** *syntax_item*_identifier **;**
| **KEYWORD** *keyword*_identifier **=** *syntax_item*_identifier **{** annotation { annotation } **}**

Syntax 36—KEYWORD declaration

A keyword declaration shall be used to define a new keyword in a category or in a subcategory of ALF statements specified by a *syntax item* identifier. One or more annotations (see Section 8.5) may be used to qualify the contents of the keyword declaration.

A legal syntax item identifier shall be defined as shown in Table 10.

Table 10—Syntax item identifier

identifier	semantic meaning
annotation	The keyword shall specify an *annotation* (see Section 7.10)
single_value_annotation	The keyword shall specify a *single value annotation* (see Section 7.10)
multi_value_annotation	The keyword shall specify a *multi_value_annotation* (see Section 7.10)
annotation_container	The keyword shall specify an *annotation container* (see Section 7.11)
arithmetic_model	The keyword shall specify an *arithmetic model* (see)
arithmetic_submodel	The keyword shall specify an *arithmetic submodel* (see)
arithmetic_model_container	The keyword shall specify an *arithmetic model container* (see)

8.5 Annotations in the context of a KEYWORD declaration

This subsection defines annotations which may be used as legal children of a keyword declaration statement.

8.5.1 VALUETYPE annotation

The *valuetype* annotation shall be a *single value annotation*. The set of legal values shall depend on the syntax item identifier associated with the keyword declaration, as shown in Table 11.

Table 11—VALUETYPE annotation

syntax item identifier	set of legal values for VALUETYPE	default value for VALUETYPE	comment
annotation or single_value_annotation or multi_value_annotation	number, identifier, quoted_string, edge_value, pin_variable, control_expression, boolean_expression, arithmetic_expression	identifier	see Syntax 34, definition of *annotation value*
annotation_container	N/A	N/A	an *annotation container* (see Syntax 35) has no value
arithmetic_model	number, identifier, bit_literal, based_literal	number	see Syntax 27, definition of *arithmetic value*
arithmetic_submodel	N/A	N/A	an *arithmetic submodel* (see) shall always have the same valuetype as its parent arithmetic mdel
arithmetic_model_container	N/A	N/A	an *arithmetic model container* (see) has no value

The valuetype annotation shall specify the category of legal ALF values applicable for an ALF statement whose ALF type is given by the declared keyword.

Example:

This example shows a correct and an incorrect usage of a declared keyword with specified valuetype.

```
KEYWORD Greeting = annotation { VALUETYPE = identifier ; }
CELL cell1 { Greeting = HiThere ; } // correct
CELL cell2 { Greeting = “Hi There” ; } // incorrect
```

The first usage is correct, since `HiThere` is an identifier. The second usage is incorret, since `“Hi There”` is a quoted string and not an identifier.

8.5.2 VALUES annotation

The *values* annotation shall be a *multi value annotation* applicable in the case where the *valuetype* annotation is also applicable.

The *values* annotation shall specify a discrete set of legal values applicable for an ALF statement using the declared keyword. Compatibility between the *values* annotation and the *valuetype* annotation shall be mandatory.

Example:

This example shows a correct and an incorrect usage of a declared keyword with specified valuetype and values.

```
KEYWORD Greeting = annotation {
     VALUETYPE = identifier ;
     VALUES { HiThere Hello HowDoYouDo }
}
CELL cell3 { Greeting = Hello ; } // correct
CELL cell4 { Greeting = GoodBye ; } // incorrect
```

The first usage is correct, since `Hello` is contained within the set of values. The second usage is incorrect, since `GoodBye` is not contained within the set of values.

8.5.3 DEFAULT annotation

The *default* annotation shall be a *single value annotation* applicable in the case where the valuetype annotation is also applicable. Compatibility between the *default* annotation, the *valuetype* annotation, and the *values* annotation shall be mandatory.

The default annotation shall specify a presumed value in absence of an ALF statement specifying a value.

Example:

```
KEYWORD Greeting = annotation {
     VALUETYPE = identifier ;
     VALUES { HiThere Hello HowDoYouDo }
     DEFAULT = Hello ;
}
CELL cell5 { /* no Greeting */ }
```

In this example, the absence of a `Greeting` statement is equivalent to the following:

```
CELL cell5 { Greeting = Hello ; }
```

8.5.4 CONTEXT annotation

The *context* annotation shall specify the ALF type of a legal parent of the statement using the declared keyword. The ALF type of a legal parent may be a predefined keyword or a declared keyword.

Example:

```
KEYWORD LibraryQualifier = annotation { CONTEXT { LIBRARY SUBLIBRARY } }
KEYWORD CellQualifier = annotation { CONTEXT = CELL ; }
KEYWORD PinQualifier = annotation { CONTEXT = PIN ; }
LIBRARY library1 {
     LibraryQualifier = foo ; // correct
     CELL cell1 {
          CellQualifier = bar ; // correct
          PinQualifier = foobar ; // incorrect
```

```
    }
}
```

The following change would legalize the example above:

```
KEYWORD PinQualifier = annotation { CONTEXT { PIN CELL } }
```

8.5.5 SI_MODEL annotation

** see IEEE proposal, January 2002, chapter 27**

8.6 GROUP declaration

A *group* shall be declared as shown in Syntax 37.

group_declaration ::=
GROUP *group*_identifier **{** all_purpose_value { all_purpose_value } **}**
| **GROUP** *group*_identifier **{** *left*_index_value **:** *right*_index_value **}**

Syntax 37—GROUP declaration

A group declaration shall be used to specify the semantic equivalent of multiple similar ALF statements within a single ALF statement. An ALF statement containing a group identifier shall be semantically replicated by substituting each *group value* for the *group identifier*, or, by substituting subsequent index values bound by the left index value and by the right index value for the group identifier. The ALF parser shall verify whether each substitution results in a legal statement.

The ALF statement which has the same parent as the group declaration shall be semantically replicated, if the group identifier is found within the statement itself or within a child of the statement or within a child of a child of the statement etc. If the group identifier is found more than once within the statement or within its children, the same group value or index value per replication shall be substituted for the group identifier, but no additional replication shall occur.

The group identifier (i.e., the name associated with the goup declaration) may be re-used as name of another statement. As a consequence, the other statement shall be interpreted as multiple statements wherein the group identifier within each replication shall be replaced by the all-purpose value. On the other hand, no name of any visible statement shall be allowed to be re-used as group identifier.

Examples

The following example shows substitution involving group values.

```
// statement using GROUP:
CELL myCell {
     GROUP data { data1 data2 data3 }
     PIN data { DIRECTION = input ; }
}
// semantically equivalent statement:
CELL myCell {
     PIN data1 { DIRECTION = input ; }
     PIN data2 { DIRECTION = input ; }
```

```
        PIN data3 { DIRECTION = input ; }
    }
```

The following example shows substitution involving index values.

```
    // statement using GROUP:
    CELL myCell {
        GROUP dataIndex { 1 : 3 }
        PIN [1:3] data { DIRECTION = input ; }
        PIN clock { DIRECTION = input ; }
        SETUP = 0.5 { FROM { PIN = data[dataIndex]; } TO { PIN = clock ; } }
    }
    // semantically equivalent statement:
    CELL myCell {
        GROUP dataIndex { 1 : 3 }
        PIN [1:3] data { DIRECTION = input ; }
        PIN clock { DIRECTION = input ; }
        SETUP = 0.5 { FROM { PIN = data[1]; } TO { PIN = clock ; } }
        SETUP = 0.5 { FROM { PIN = data[2]; } TO { PIN = clock ; } }
        SETUP = 0.5 { FROM { PIN = data[3]; } TO { PIN = clock ; } }
    }
```

The following example shows multiple occurences of the same group identifier within a statement.

```
    // statement using GROUP:
    CELL myCell {
        GROUP dataIndex { 1 : 3 }
        PIN [1:3] Din { DIRECTION = input ; }
        PIN [1:3] Dout { DIRECTION = input ; }
        DELAY = 1.0 { FROM {PIN=Din[dataIndex];} TO {PIN=Dout[dataIndex];} }
    }
    // semantically equivalent statement:
    CELL myCell {
        GROUP dataIndex { 1 : 3 }
        PIN [1:3] Din { DIRECTION = input ; }
        PIN [1:3] Dout { DIRECTION = input ; }
        DELAY = 1.0 { FROM {PIN=Din[1];} TO {PIN=Dout[1];} }
        DELAY = 1.0 { FROM {PIN=Din[2];} TO {PIN=Dout[2];} }
        DELAY = 1.0 { FROM {PIN=Din[3];} TO {PIN=Dout[3];} }
    }
```

8.7 TEMPLATE declaration

A *template* shall be declared as shown in Syntax 38.

template_declaration ::=
TEMPLATE *template*_identifier **{** ALF_statement { ALF_statement } **}**

Syntax 38—TEMPLATE declaration

A template declaration shall be used to specify one or more ALF statements with variable contents that can be used many times. A template instantiation (see Section 8.8) shall specify the usage of such an ALF statement.

Within the template declaration, the variable contents shall be specified by a placeholder identifier (see Section 6.10.3).

8.8 Template instantiation

A *template* shall be instantiated in form of a *static template instantiation* or a *dynamic template instantiation*, as shown in Syntax 39

```
template_instantiation ::=
      static_template_instantiation
    | dynamic_template_instantiation

static_template_instantiation ::=
      template_identifier [ = STATIC ] ;
    | template_identifier [ = STATIC ] { { all_purpose_value } }
    | template_identifier [ = STATIC ] { { annotation } }

dynamic_template_instantiation ::=
      template_identifier = DYNAMIC { { dynamic_template_instantiation_item } }

dynamic_template_instantiation_item ::=
      annotation
    | arithmetic_model
```

Syntax 39—TEMPLATE instantiation

A template instantiation shall be semantically equivalent to the ALF statement or the ALF statements found within the template declaration, after replacing the placeholder identifiers with replacement values. A static template instantiation shall support replacement by order, using one or more all-purpose values, or alternatively, replacement by reference, using one or more annotations (see). A dynamic template instantiation shall support replacement by reference only, using one or more annotations and/or one or more arithmetic models (see).

In the case of replacement by reference, the reference shall be established by a non-escaped identifier matching the placeholder identifier when the angular brackets are removed. The matching shall be case-insensitive.

The following rules shall apply:

a) A static template instantiation shall be used when the replacement value of any placeholder identifier can be determined during compilation of the library. Only a matching identifier shall be considered a legal annotation identifier. Each occurence of the placeholder identifier shall be replaced by the annotation value associated with the annotation identifier.
b) A dynamic template instantiation shall be used when the replacement value of at least one placeholder identifier can only determined during runtime of the application. Only a matching identifier shall be considered a legal annotation identifier, or alternatively, a arithmetic model identifier, or alternatively, a legal arithmetic value.
c) Multiple replacement values within a multi-value annotation shall be legal if and only if the syntax rules for the ALF statement within the template declaration allow substitution of multiple values for one placeholder identifier.
d) In the case replacement by order, subsequently occuring placeholder identifiers in the template declaration shall be replaced by subsequently occuring all-purpose values in the template instantiation. If a placeholder identifier occurs more than once within the template declaration, all occurences of that placeholder identifier shall be immediately replaced by the same all-purpose value. The first amongst the remaining placeholder identifiers shall then be considered the next placeholder to be replaced by the next all-purpose value.

e) A static template instantiation for which a placeholder identifier is not replaced shall be legal if and only if the semantic rules for the ALF statement support a placeholder identifier outside a template declaration. However, the semantics of a placeholder identifier as an item to be substituted shall only apply within the template declaration statement.

Examples

The following example illustrates rule a).

```
// statement using TEMPLATE declaration and instantiation:
TEMPLATE someAnnotations {
     KEYWORD <oneAnnotation> = single_value_annotation ;
     KEYWORD annotation2 = single_value_annotation ;
     <oneAnnotation> = value1 ;
     annotation2 = <anotherValue> ;
}
someAnnotations {
     oneAnnotation = annotation1 ;
     anotherValue = value2 ;
}
// semantically equivalent statement:
KEYWORD annotation1 = single_value_annotation ;
KEYWORD annotation2 = single_value_annotation ;
annotation1 = value1 ;
annotation2 = value2 ;
```

The following example illustrates rule b).

```
// statement using TEMPLATE declaration and instantiation:
TEMPLATE someNumbers {
     KEYWORD N1 = single_value_annotation { VALUETYPE=number ; }
     KEYWORD N2 = single_value_annotation { VALUETYPE=number ; }
     N1 = <number1> ;
     N2 = <number2> ;
}
someNumbers = DYNAMIC {
     number2 = number1 + 1;
}
// semantically equivalent statement, assuming number1=3 at runtime:
N1 = 3 ;
N2 = 4 ;
```

The following example illustrates rule c).

```
TEMPLATE moreAnnotations {
     KEYWORD annotation3 = annotation ;
     KEYWORD annotation4 = annotation ;
     annotation3 { <someValue> }
     annotation4 = <yetAnotherValue> ;
}
moreAnnotations {
     someValue { value1 value2 }
     yetAnotherValue = value3 ;
}
```

```
// semantically equivalent statement:
KEYWORD annotation3 = annotation ;
KEYWORD annotation4 = annotation ;
annotation3 { value1 value2 }
annotation4 = value3 ;
```

The following example illustrates rule e).

```
TEMPLATE evenMoreAnnotations {
     KEYWORD <thisAnnotation> = single_value_annotation ;
     KEYWORD <thatAnnotation> = single_value_annotation ;
     <thatAnnotation> = <thisValue> ;
     <thisAnnotation> = <thatValue> ;
}
// template instantiation by reference:
evenMoreAnnotations = STATIC {
     thatAnnotation = day ;
     thisAnnotation = month;
     thatValue = April;
     thisValue = Monday;
}
// semantically equivalent template instantiation by order:
evenMoreAnnotations = STATIC { day month Monday April }

// semantically equivalent statement:
KEYWORD day = single_value_annotation ;
KEYWORD month = single_value_annotation ;
month = April;
day = Monday;
```

The following example illustrates rule d).

```
// statement using TEMPLATE declaration and instantiation:
TEMPLATE encoreAnnotation {
     KEYWORD context1 = annotation_container;
     KEYWORD context2 = annotation_container;
     KEYWORD annotation5 = single_value_annotation {
          CONTEXT { context1 context2 }
          VALUES { <something> <nothing> }
     }
     context1 { annotation5 = <nothing> ; }
     context2 { annotation5 = <something> ; }
}
encoreAnnotation {
     something = everything ;
}
// semantically equivalent statement:
KEYWORD context1 = annotation_container;
KEYWORD context2 = annotation_container;
KEYWORD annotation5 = single_value_annotation {
     CONTEXT { context1 context2 }
     VALUES { everything <nothing> }
}
context1 { annotation5 = <nothing> ; }
```

```
context2 { annotation5 = all ; }
// Both everything (without brackets) and <nothing> (with brackets)
// are legal values for annotation5.
```

9. Library-specific objects and related statements

Add lead-in text

9.1 LIBRARY and SUBLIBRARY declaration

A *library* and a *sublibrary* shall be declared as shown in Syntax 43.

```
library ::=
      LIBRARY library_identifier ;
    | LIBRARY library_identifier { { library_item } }
    | library_template_instantiation
library_item ::=
      sublibrary
    | sublibrary_item
sublibrary ::=
      SUBLIBRARY sublibrary_identifier ;
    | SUBLIBRARY sublibrary_identifier { { sublibrary_item } }
    | sublibrary_template_instantiation
sublibrary_item ::=
      all_purpose_item
    | cell
    | primitive
    | wire
    | layer
    | via
    | rule
    | antenna
    | array
    | site
    | region
```

Syntax 43—LIBRARY and SUBLIBRARY statement

A library shall serve as a repository of technology data for creation of an electronic integrated circuit. A sublibrary may optionally be used to create different scopes of visibility for particular statements describing technology data.

If any two objects of the same ALF type and the same ALF name appear in two libraries, or in two sublibraries with the same library as parents, their usage for creation of an electronic circuit shall be mutually exclusive. For example, two cells with the same name shall not be instantiated in the same integrated circuit. It shall be the responsibility of the application tool to detect and properly handle such cases, as the selection of a library or a sublibrary is controlled by the user of the application tool.

9.2 INFORMATION statement

An *information* statement shall be defined using ALF language as shown in Syntax 44.

The information statement shall be used to associate its parent statement with a product specification. While information statement complies with the syntax definition of an annotation container (see Section 7.11), the following restrictions shall apply:

a) A library, a sublibrary, or a cell shall be a legal parent of the information statement.
b) A wire, or a primitive shall be a legal parent of the information statement, provided the parent of the wire or the primitive is a library or a sublibrary.

```
KEYWORD INFORMATION = annotation_container {
  CONTEXT { LIBRARY SUBLIBRARY CELL WIRE PRIMITIVE }
}
KEYWORD PRODUCT = single_value_annotation {
  VALUETYPE = string; DEFAULT = “”; CONTEXT = INFORMATION;
}
KEYWORD TITLE = single_value_annotation {
  VALUETYPE = string; DEFAULT = “”; CONTEXT = INFORMATION;
}
KEYWORD VERSION = single_value_annotation {
  VALUETYPE = string; DEFAULT = “”; CONTEXT = INFORMATION;
}
KEYWORD AUTHOR = single_value_annotation {
  VALUETYPE = string; DEFAULT = “”; CONTEXT = INFORMATION;
}
KEYWORD DATETIME = single_value_annotation {
  VALUETYPE = string; DEFAULT = “”; CONTEXT = INFORMATION;
}
```

Syntax 44—INFORMATION statement

The semantics of the information contents are specified in the following Table 10.

Table 10—Annotations within an INFORMATION statement

annotation identifier	semantics of annotation value
PRODUCT	a code name of a product described herein
TITLE	a descriptive title of the product described herein
VERSION	a version number of the product description
AUTHOR	the name of a person or company generating this product description
DATETIME	date and time of day when this product description was created

The product developer shall be responsible for any rules concerning the format and detailed contents of the string value itself.

Example

```
LIBRARY myProduct {
    INFORMATION {
        PRODUCT = p10sc;
        TITLE = “0.10 standard cell”;
        VERSION = “v2.1.0”;
        AUTHOR = “Major Asic Vendor, Inc.”;
        DATETIME = “Mon Apr 8 18:33:12 PST 2002”;
    }
}
```

9.3 CELL declaration

A *cell* shall be declared as shown in Syntax 45.

```
cell ::=
      CELL cell_identifier ;
    | CELL cell_identifier { { cell_item } }
    | cell_template_instantiation
cell_item ::=
      all_purpose_item
    | pin
    | pingroup
    | primitive
    | function
    | non_scan_cell
    | test
    | vector
    | wire
    | blockage
    | artwork
    | pattern
    | region
```

Syntax 45—CELL statement

A cell shall represent an electronic circuit which can be used as a building block for a larger electronic circuit.

9.4 Annotations and attributes for a CELL

This section defines annotations and attribute values in the context of a cell declaration.

9.4.1 CELLTYPE annotation

A *celltype* annotation shall be defined using ALF language as shown in .

```
KEYWORD CELLTYPE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES {
    buffer combinational multiplexor flipflop latch
    memory block core special
  }
}
```

Syntax 46— annotation

The celltype shall divide cells into categories, as specified in Table 11.

Table 11—CELLTYPE annotation values

Annotation value	Description
buffer	Cell is a buffer, inverting or non-inverting.

Table 11—CELLTYPE annotation values

Annotation value	Description
`combinational`	Cell is a combinational logic element.
`multiplexor`	Cell is a multiplexor.
`flipflop`	Cell is a flip-flop.
`latch`	Cell is a latch.
`memory`	Cell is a memory or a register file.
`block`	Cell is a hierarchical block, i.e., a complex element which can be represented as a netlist. All instances of the netlist are library elements, i.e., there is a `CELL` model for each of them in the library.
`core`	Cell is a core, i.e., a complex element which can be represented as a netlist. At least one instance of the netlist is not a library element, i.e., there is no `CELL` model, but a `PRIMITIVE` model for that instance.
`special`	Cell is a special element, which can only be used in certain application contexts not describable by the `FUNCTION` statement. Examples: busholders, protection diodes, and fillcells.

9.4.2 ATTRIBUTE within a CELL

An attribute in the context of a cell declaration shall specify more specific information within the category given by the celltype annotation.

The attribute values shown in Table 12 can be used within a `CELL` with `CELLTYPE=memory`.

Table 12—Attribute values for a CELL with CELLTYPE=memory

Attribute item	Description
`RAM`	Random Access Memory
`ROM`	Read Only Memory
`CAM`	Content Addressable Memory
`static`	Static memory (e.g., static RAM)
`dynamic`	Dynamic memory (e.g., dynamic RAM)
`asynchronous`	Asynchronous memory
`synchronous`	Synchronous memory

The attributes shown in Table 13 can be used within a CELL with CELLTYPE=block.

Table 13—Attributes within a CELL with CELLTYPE=block

Attribute item	Description
counter	Cell is a complex sequential cell going through a predefined sequence of states in its normal operation mode where each state represents an encoded control value.
shift_register	Cell is a complex sequential cell going through a predefined sequence of states in its normal operation mode, where each subsequent state can be obtained from the previous one by a shift operation. Each bit represents a data value.
adder	Cell is an adder, i.e., a combinational element performing an addition of two operands.
subtractor	Cell is a subtractor, i.e., a combinational element performing a subtraction of two operands.
multiplier	Cell is a multiplier, i.e., a combinational element performing a multiplication of two operands.
comparator	Cell is a comparator, i.e., a combinational element comparing the magnitude of two operands.
ALU	Cell is an arithmetic logic unit, i.e., a combinational element combining the functionality of adder, subtractor, comparator in a selectable way.

The attributes shown in Table 14 can be used within a CELL with CELLTYPE=core.

Table 14—Attributes within a CELL with CELLTYPE=core

Attribute item	Description
PLL	CELL is a phase-locked loop.
DSP	CELL is a digital signal processor.
CPU	CELL is a central processing unit.
GPU	CELL is a graphical processing unit.

The attributes shown in Table 15 can be used within a CELL with CELLTYPE=special.

Table 15—Attributes within a CELL with CELLTYPE=special

Attribute item	Description
busholder	CELL enables a tristate bus to hold its last value before all drivers went into high-impedance state (see FUNCTION statement).

Table 15—Attributes within a CELL with CELLTYPE=special (Continued)

Attribute item	Description
clamp	CELL connects a net to a constant value (logic value and drive strength; see FUNCTION statement).
diode	CELL is a diode (no FUNCTION statement).
capacitor	CELL is a capacitor (no FUNCTION statement).
resistor	CELL is a resistor (no FUNCTION statement).
inductor	CELL is an inductor (no FUNCTION statement).
fillcell	CELL is merely used to fill unused space in layout (no FUNCTION statement).

9.4.3 SWAP_CLASS annotation

A *swap_class* annotation shall be defined using ALF language as shown in .

```
KEYWORD SWAP_CLASS = annotation {
  CONTEXT = CELL;
  VALUETYPE = identifier;
}
```

Syntax 47— annotation

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

Cell-swapping is only allowed under the following conditions:

— the RESTRICT_CLASS annotation (see 9.4.4) authorizes usage of the cell
— the cells to be swapped are compatible from an application standpoint (functional compatibility for synthesis and physical compatibility for layout)

9.4.4 RESTRICT_CLASS annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD RESTRICT_CLASS = annotation {
  CONTEXT { CELL CLASS }
  VALUETYPE = identifier;
}
```

Syntax 48— annotation

The value is the name of a declared CLASS. Multi-value annotation can be used. Cells referring to a particular class can be used in design tools identified by the value. The restricted annotations are shown in Table 16.

Table 16—Predefined values for RESTRICT_CLASS

Annotation string	Description
synthesis	Use restricted to logic synthesis.
scan	Use restricted to scan synthesis.
datapath	Use restricted to datapath synthesis.
clock	Use restricted to clock tree synthesis.
layout	Use restricted to layout, i.e., place & route.

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

9.4.4.1 Independent SWAP_CLASS and RESTRICT CLASS

SWAP_CLASS and RESTRICT_CLASS can be defined for cells, independent of each other. In this case, the set of cells that can be swapped with each other is the set of cells with a non-empty intersection of both SWAP_CLASS and RESTRICT_CLASS.

Example

```
CLASS foo;
CLASS bar;
CLASS whatever;
CLASS my_tool;
CELL cell1 {
    SWAP_CLASS { foo bar }
    RESTRICT_CLASS { synthesis datapath }
}
CELL cell2 {
    SWAP_CLASS { foo whatever }
    RESTRICT_CLASS { synthesis scan my_tool }
}
```

The cells cell1 and cell2 can be used for synthesis, where they can be swapped which each other. Cell cell1 can be also used for datapath. Cell cell2 can be also used for scan insertion and for the user-defined application my_tool. Figure 8 depicts this scenario.

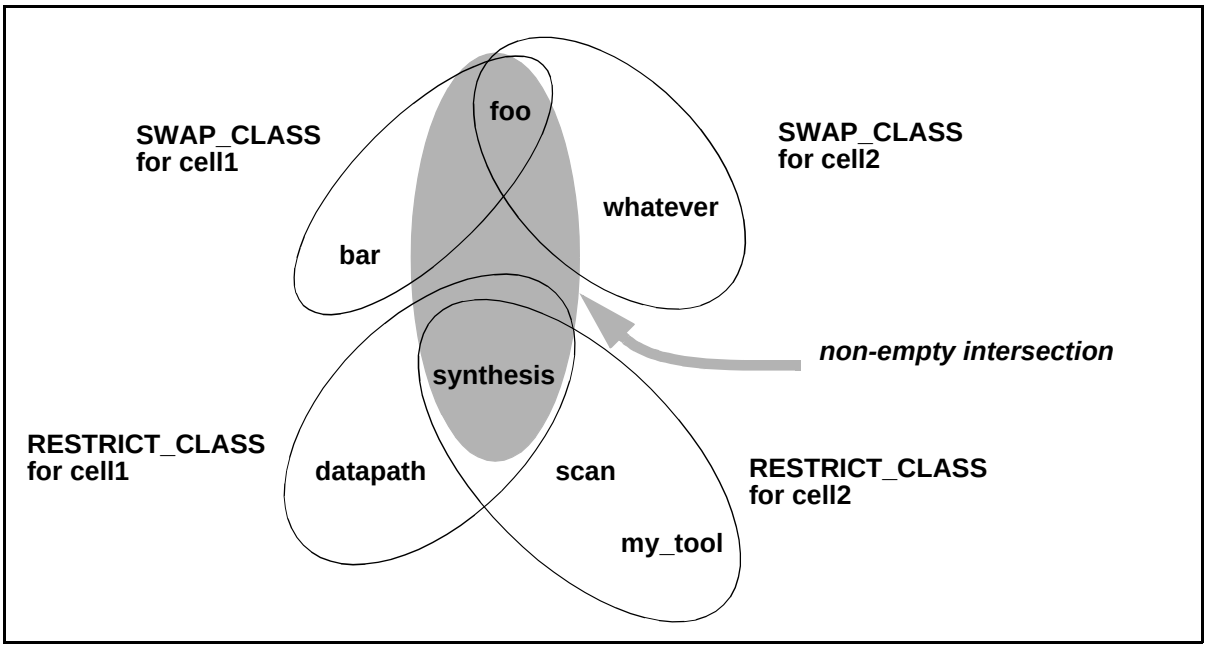


Figure 8—Illustration of independent SWAP_CLASS and RESTRICT_CLASS

9.4.4.2 SWAP_CLASS with inherited RESTRICT_CLASS

The definition of a CLASS can contain a RESTRICT_CLASS annotation. In this case, the RESTRICT_CLASS is inherited by the SWAP_CLASS. Cells can only be swapped if the intersection of their SWAP_CLASS and the inherited RESTRICT_CLASS is non-empty.

Example

A combination of SWAP_CLASS and RESTRICT_CLASS can be used to emulate the concept of “logically equivalent cells” and “electrically equivalent cells”. A synthesis tool needs to know about “logically equivalent cells” for swapping. A layout tool needs to know about “electrically equivalent cells” for swapping.

```
CLASS all_nand2 { RESTRICT_CLASS { synthesis } }
CLASS all_high_power_nand2 { RESTRICT_CLASS { layout } }
CLASS all_low_power_nand2 { RESTRICT_CLASS { layout } }

CELL cell1 {
    SWAP_CLASS { all_nand2 all_low_power_nand2 }
}
CELL cell2 {
    SWAP_CLASS { all_nand2 all_high_power_nand2 }
}
CELL cell3 {
    SWAP_CLASS { all_low_power_nand2 }
}
CELL cell4 {
    SWAP_CLASS { all_high_power_nand2 }
}
```

`all_nand2` encompasses a set of logically equivalent cells.
`all_high_power_nand2` encompasses a set of electrically equivalent cells.
`all_low_power_nand2` encompasses another set of electrically equivalent cells.

The synthesis tool can swap `cell1` with `cell2`. The layout tool can swap `cell1` with `cell3` and `cell2` with `cell4`. Figure 9 depicts this scenario.

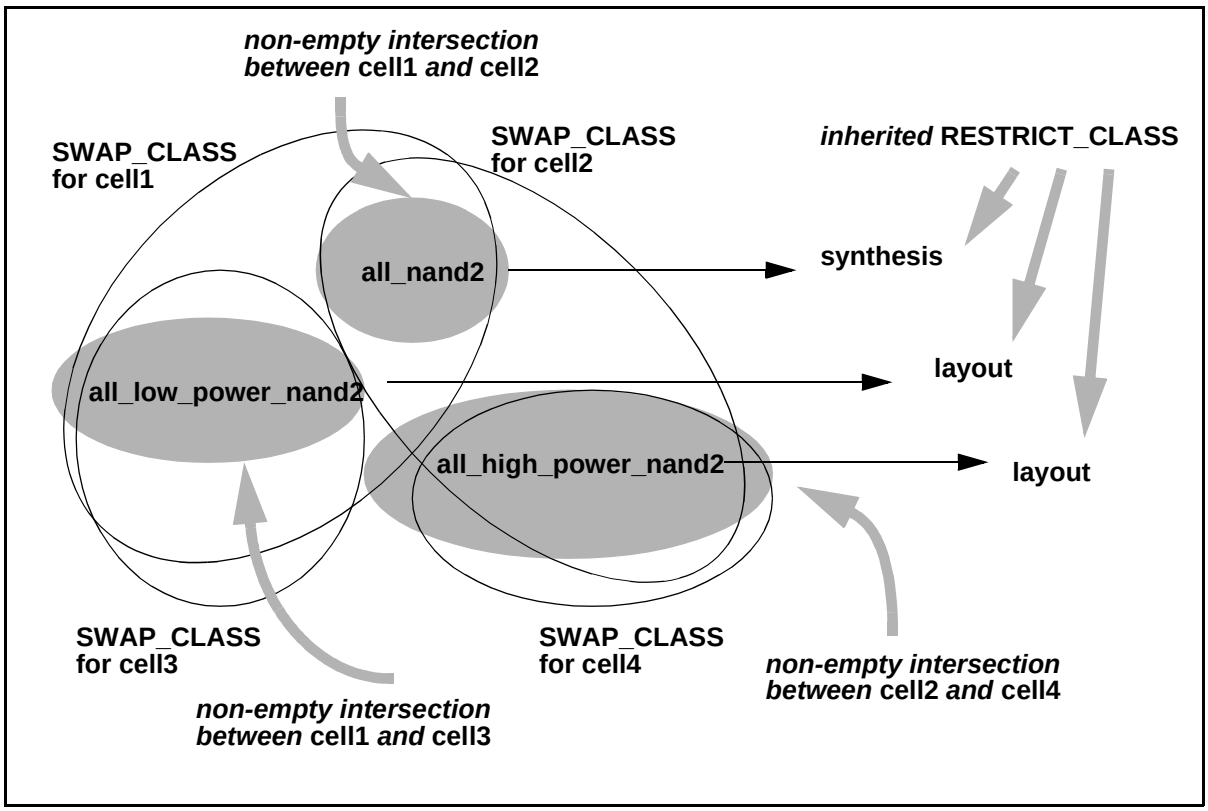


Figure 9—Illustration of SWAP_CLASS with inherited RESTRICT_CLASS

9.4.5 SCAN_TYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD SCAN_TYPE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { muxscan clocked lssd control_0 control_1 }
}
```

Syntax 49— annotation

can take the values shown in Table 17.

Table 17—SCAN_TYPE annotations for a CELL object

Annotation string	Description
`muxscan`	A multiplexor for normal data and scan data.
`clocked`	A special scan clock.
`lssd`	Combination between flip-flop and latch with special clocking (level sensitive scan design).
`control_0`	Combinational scan cell, controlling pin shall be `0` in scan mode.
`control_1`	Combinational scan cell, controlling pin shall be `1` in scan mode.

See Section A.3 for examples.

9.4.6 SCAN_USAGE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD SCAN_USAGE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { input output hold }
}
```

Syntax 50— annotation

can take the values shown in Table 18.

Table 18—SCAN_USAGE annotations for a CELL object

Annotation string	Description
`input`	Primary input in a chain of cells.
`output`	Primary output in a chain of cells.
`hold`	Holds intermediate value in the scan chain.

The `SCAN_USAGE` applies for a special cell which is designed to be the primary input, output or intermediate stage of a scan chain. It also applies for macro blocks with connected scan chains in case there are particular scan-ordering requirements.

9.4.7 BUFFERTYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD BUFFERTYPE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { input output inout internal }
  DEFAULT = internal;
}
```

Syntax 51— annotation

can take the values shown in Table 19.

Table 19—BUFFERTYPE annotations for a CELL object

Annotation string	Description
`input`	Cell has at least one external (off-chip) input pin.
`output`	Cell has at least one external (off-chip) output pin.
`inout`	Cell has at least one external (off-chip) bidirectional pin.
`internal`	Cell has only internal (on-chip) pins.

9.4.8 DRIVERTYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD DRIVERTYPE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { predriver slotdriver both }
}
```

Syntax 52— annotation

can take the values shown in Table 20.

Table 20—DRIVERTYPE annotations for a CELL object

Annotation string	Description
`predriver`	Cell is a predriver, i.e., the core part of an IO buffer.
`slotdriver`	Cell is a slotdriver, i.e., the pad of an IO buffer with off-chip connection.
`both`	Cell is both a predriver and a slot driver, i.e., a complete IO buffer.

NOTE—`DRIVERTYPE` applies only for cells with `BUFFERTYPE = input | output | inout`.

9.4.9 PARALLEL_DRIVE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD PARALLEL_DRIVE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = unsigned;
  DEFAULT = 1;
}
```

Syntax 53— annotation

specifies the number of parallel drivers. This shall be greater than zero (`0`) ; the default is `1`.

9.4.10 PLACEMENT_TYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD PLACEMENT_TYPE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { pad core ring block onnector }
  DEFAULT = core;
}
```

Syntax 54— annotation

The identifiers have the following definitions:

- *pad*: I/O pad, to be placed in the I/O rows
- *core*: regular macro, to be placed in the core rows
- *block*: hierarchical block with regular power structure
- *ring*: macro with built-in power structure
- *connector*: macro at the end of core rows connecting with power or ground

9.4.11 SITE reference annotation

A `CELL` can point to one or more legal placement `SITE`s.

Example

```
CELL my_cell {
    SITE { my_site /* fill in other sites, if applicable */ }
    /* fill in contents of cell definition */
}
```

9.5 PIN declaration

A *pin* shall be declared as a *scalar pin* or as a *vector pin* or a *matrix pin*, as shown in Syntax 55.

```
pin ::=
      scalar_pin | vector_pin | matrix_pin
scalar_pin ::=
      PIN pin_identifier ;
    | PIN pin_identifier { { scalar_pin_item } }
    | scalar_pin_template_instantiation
vector_pin ::=
      PIN multi_index pin_identifier ;
    | PIN multi_index pin_identifier { { vector_pin_item } }
    | vector_pin_template_instantiation
matrix_pin ::=
      PIN first_multi_index pin_identifier second_multi_index ;
    | PIN first_multi_index pin_identifier second_multi_index { { matrix_pin_item } }
    | matrix_pin_template_instantiation
scalar_pin_item ::=
      all_purpose_item
    | port
    | pull
vector_pin_item ::=
      all_purpose_item
    | range
matrix_pin_item ::=
      vector_pin_item
```

Syntax 55—PIN declaration

A pin shall represent a terminal of an electronic circuit for the purpose of exchanging information with the environment of the electronic circuit. A constant value of information shall be called *state*. A time-dependent value of information shall be called *signal*. A reference to a pin in general shall be established by the pin identifier.

A scalar pin may be associated with a general electrical signal. However, a vector pin or a matrix pin may only be associated with digital signals. One element of a vector pin or of a matrix pin shall be associated with one bit of information, i.e., a binary digital signal.

A vector-pin can be considered as a combination of scalar pins. A reference to a scalar or to a subvector, respectively, within the vector-pin shall be established by the pin identifier followed by a single index or by a multi index, respectively.

A matrix-pin can be considered as a combination of vector-pins. A reference to a vector or to a submatrix, respectively, within the matrix-pin shall be established by the pin identifier followed by a single index or by a multi index, respectively.

Within a matrix-pin declaration, the first multi index shall specify the range of scalars or bits, and the second multi index shall specify the range of vectors. Support for direct reference of a scalar within a vector within a matrix is not provided.

Example

```
PIN [5:8] myVectorPin ;
PIN [3:0] myMatrixPin [1:1000] ;
```

The pin variable `myVectorPin[5]` refers to the scalar associated with the MSB of `myVectorPin`.
The pin variable `myVectorPin[8]` refers to the scalar associated with the LSB of `myVectorPin`.
The pin variable `myVectorPin[6:7]` refers to a subvector within `myVectorPin`.
The pin variable `myMatrixPin[500]` refers to a vector within `myMatrixPin`.
The pin variable `myMatrixPin[500:502]` refers to 3 subsequent vectors within `myMatrixPin`.

Consider the following pin assignment:

```
myVectorPin=myMatrixPin[500];
```

This establishes the following exchange of information:

`myVectorPin[5]` receives information from element `[3]` of `myMatrixPin[500]`.
`myVectorPin[6]` receives information from element `[2]` of `myMatrixPin[500]`.
`myVectorPin[7]` receives information from element `[1]` of `myMatrixPin[500]`.
`myVectorPin[8]` receives information from element `[0]` of `myMatrixPin[500]`.

9.6 RANGE statement

A *range* statement shall be defined as shown in Syntax 56.

range ::=
RANGE **{** index_value **:** index_value **}**

Syntax 56—RANGE statement

The range statement shall be used to specify a valid address space for elements of a vector- or matrix-pin.

If no range statement is specified, the valid address space is given by the following mathematical relationship:

$$0 \le a \le 2^b - 1$$

$$b = \begin{cases} 1 + \mathrm{LSB} - \mathrm{MSB} & \text{if}(\mathrm{LSB} > \mathrm{MSB}) \\ 1 + \mathrm{MSB} - \mathrm{LSB} & \text{if}(\mathrm{LSB} \le \mathrm{MSB}) \end{cases}$$

where

a is an unsigned number representing the decimal equivalent of the bits within a vector- or matrix-pin,
b is the bitwidth of the vector- or matrix-pin,

and

MSB is the leftmost bit within the vector- or matrix-pin,
LSB is the rightmost bit within the vector or- matrix-pin,

in accordance with Section 7.7 on page 34.

The index values within a range statement shall be bound by the address space a, or else the range statement shall not be considered valid.

Example

```
PIN [5:8] myVectorPin { RANGE { 3 : 13 } }
```

bitwidth:	$b = 4$
default address space:	$0 \le a \le 15$
address space defined by range statement:	$3 \le a \le 13$

9.6.1 PINGROUP declaration

A *pingroup* shall be declared as a *simple pingroup* or as a *vector pingroup,* as shown in Syntax 57.

```
pingroup ::=
      simple_pingroup | vector_pingroup
simple_pingroup ::=
      PINGROUP pingroup_identifier { members { all_purpose_item } }
    | simple_pingroup_template_instantiation
vector_pingroup ::=
    | PINGROUP [ index_value : index_value ] pingroup_identifier
        { members { vector_pingroup_item } }
    | vector_pingroup_template_instantiation
vector_pingroup_item ::=
      all_purpose_item
    | range
members ::=
      MEMBERS { pin_identifier pin_identifier { pin_identifier } }
```

Syntax 57—PINGROUP declaration

A pingroup in general shall serve the purpose to specify items applicable to a combination of pins rather than to each pin within the combination. The combination of pins shall be specified by the *members* statement.

A *vector pingroup* may combine only scalar pins. A vector pingroup may be used as a pin variable, in the same capacity as a vector pin.

A *simple pingroup* may combine pins of any format, i.e., scalar pins, vector pins, and matrix pins. A simple pingroup may not be used as a pin variable.

9.7 Annotations and attributes for a PIN

This section defines annotations and attribute values in the context of a pin declaration or a pingroup declaration.

9.7.1 VIEW annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD VIEW = single_value_annotation {
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
  VALUES { functional physical both none }
  DEFAULT = both
}
```

Syntax 58— annotation

annotates the view where the pin appears, which can take the values shown in Table 21.

Table 21—VIEW annotations for a PIN object

Annotation string	Description
`functional`	Pin appears in functional netlist.
`physical`	Pin appears in physical netlist.
`both` (default)	Pin appears in both functional and physical netlist.
`none`	Pin does not appear in netlist.

9.7.2 PINTYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD PINTYPE = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { digital analog supply }
  DEFAULT = digital;
}
```

Syntax 59— annotation

annotates the type of the pin, which can take the values shown in Table 22.

Table 22—PINTYPE annotations for a PIN object

Annotation string	Description
`digital` (default)	Digital signal pin.
`analog`	Analog signal pin.
`supply`	Power supply or ground pin.

9.7.3 DIRECTION annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD DIRECTION = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { input output both none }
}
```

Syntax 60— annotation

annotates the direction of the pin, which can take the values shown in Table 23.

Table 23—DIRECTION annotations for a PIN object

Annotation string	Description
`input`	Input pin.
`output`	Output pin.
`both`	Bidirectional pin.
`none`	No direction can be assigned to the pin.

Table 24 gives a more detailed semantic interpretation for using `DIRECTION` in combination with `PINTYPE`.

Table 24—DIRECTION in combination with PINTYPE

DIRECTION	PINTYPE=digital	PINTYPE=analog	PINTYPE=supply
`input`	Pin receives a digital signal.	Pin receives an analog signal.	Pin is a power sink.
`output`	Pin drives a digital signal.	Pin drives an analog signal.	Pin is a power source.
`both`	Pin drives or receives a digital signal, depending on the operation mode.	Pin drives or receives an analog signal, depending on the operation mode.	Pin is both power sink and source.
`none`	Pin represents either an internal digital signal with no external connection or a feed through.	Pin represents either an internal analog signal with no external connection or a feed through.	Pin represents either an internal power pin with no external connection or a feed through.

For pins with PINTYPE=supply, the DIRECTION describes an electrical characteristic rather than a functional characteristic, since there is no functional definition for DIRECTION. For pins with PINTYPE=digital or analog, the functional definition of DIRECTION actually matches the electrical definition.

Examples

- The power and ground pins of regular cells shall have `DIRECTION=input`.
- A level converter cell shall have a power supply pin with `DIRECTION=input` and another power supply pin with `DIRECTION=output`.
- A level converter can have separate ground pins on the input and output side or a common ground pin with `DIRECTION=both`.
- The power and ground pins of a feed through cell shall have `DIRECTION=none`.

9.7.4 SIGNALTYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

`SIGNALTYPE` classifies the functionality of a pin. The currently defined values apply for pins with `PINTYPE=DIGITAL`.

```
KEYWORD SIGNALTYPE = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { to be reviewed }
  DEFAULT = data;
}
```

Syntax 61— annotation

Conceptually, a pin with `PINTYPE = ANALOG` can also have a `SIGNALTYPE` annotation. However, no values are currently defined.

annotates the type of the signal connected to the pin.

The fundamental `SIGNALTYPE` values are defined in Table 25.

Table 25—Fundamental SIGNALTYPE annotations for a PIN object

Annotation string	Description
`data` (default)	General data signal, i.e., a signal that carries information to be transmitted, received, or subjected to logic operations within the `CELL`.
`address`	Address signal of a memory, i.e., an encoded signal, usually a bus or part of a bus, driving an address decoder within the `CELL`.
`control`	General control signal, i.e., an encoded signal that controls at least two modes of operation of the `CELL`, eventually in conjunction with other signals. The signal value is allowed to change during real-time circuit operation.
`select`	Select signal of a multiplexor, i.e., a decoded or encoded signal that selects the data path of a multiplexor or de-multiplexor within the `CELL`. Each selected signal has the same `SIGNALTYPE`.
`enable`	General enable signal, i.e., a decoded signal which enables and disables a set of operational modes of the `CELL`, eventually in conjunction with other signals. The signal value is expected to change during real-time circuit operation.
`tie`	The signal needs to be tied to a fixed value statically in order to define a fixed or programmable mode of operation of the `CELL`, eventually in conjunction with other signals. The signal value is not allowed to change during real-time circuit operation.
`clear`	Clear signal of a flip-flop or latch, i.e., a signal that controls the storage of the value `0` within the `CELL`.
`set`	Set signal of a flip-flop or latch, i.e., a signal that controls the storage of the value `1` within the `CELL`.
`clock`	Clock signal of a flip-flop or latch, i.e., a timing-critical signal that triggers data storage within the `CELL`.

“Flipflop”, “latch”, “multiplexor”, and “memory” can be standalone cells or embedded in larger cells. In the former case, the celltype is `flipflop`, `latch`, `multiplexor`, and `memory`, respectively. In the latter case, the celltype is `block` or `core`.

Composite values for SIGNALTYPE shall be constructed using one or more prefixes in combination with certain fundamental values, separated by the underscore (_) character, as shown in Table 26 — Table 30.

The scheme for this is shown in Figure 10.

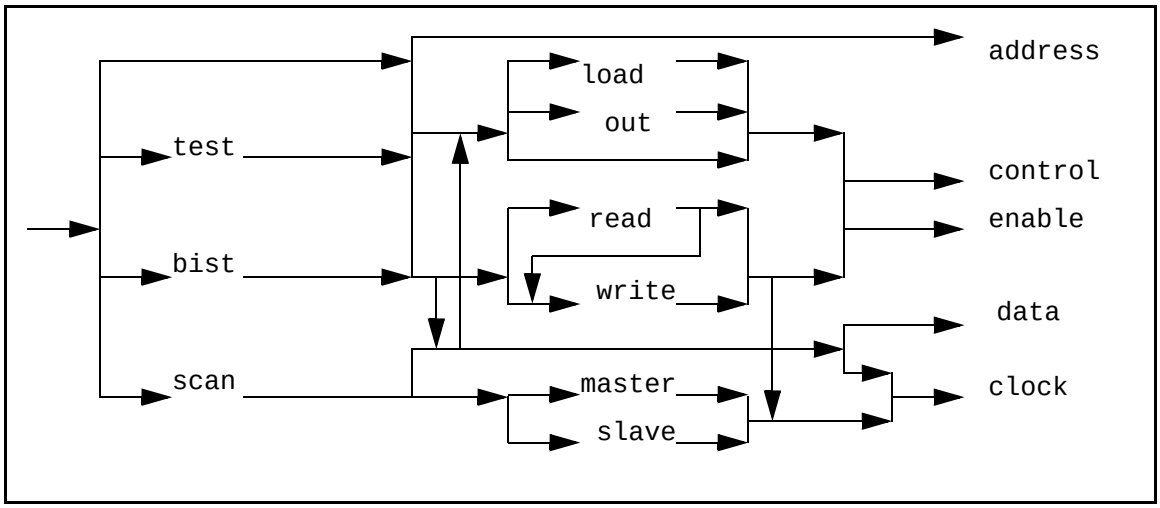


Figure 10—Construction scheme for composite SIGNALTYPE values

Table 26—Composite SIGNALTYPE annotations based on DATA

Annotation string	Description
scan_data	Data signal for scan mode.
test_data	Data signal for test mode.
bist_data	Data signal in BIST mode.

Table 27—Composite SIGNALTYPE annotations based on ADDRESS

Annotation string	Description
test_address	Address signal for test mode.
bist_address	Address signal for BIST mode.

Table 28—Composite SIGNALTYPE annotations based on CONTROL

Annotation string	Description
load_control	Control signal for switching between load mode and normal mode.
scan_control	Control signal for switching between scan mode and normal mode.
test_control	Control signal for switching between test mode and normal mode.
bist_control	Control signal for switching between BIST mode and normal mode.
read_write_control	Control signal for switching between read and write operation.
test_read_write_control	Control signal for switching between read and write operation in test mode.
bist_read_write_control	Control signal for switching between read and write operation in BIST mode.

Table 29—Composite SIGNALTYPE annotations based on ENABLE

Annotation string	Description
load_enable	Signal enables load operation in a counter or a shift register.
out_enable	Signal enables the output stage of an arbitrary cell.
scan_enable	Signal enables scan mode of a flip-flop or latch only.
scan_out_enable	Signal enables the output of a flip-flop or latch in scan mode only.
test_enable	Signal enables test mode only.
bist_enable	Signal enables BIST mode only.
test_out_enable	Signal enables the output stage in test mode only.
bist_out_enable	Signal enables the output stage in BIST mode only.
read_enable	Signal enables the read operation of a memory.
write_enable	Signal enables the write operation of a memory.
test_read_enable	Signal enables the read operation in test mode only.
test_write_enable	Signal enables the write operation in test mode only.
bist_read_enable	Signal enables the read operation in BIST mode only.
bist_write_enable	Signal enables the write operation in BIST mode only.

Table 30—Composite SIGNALTYPE annotations based on CLOCK

Annotation string	Description
scan_clock	Signal is clock of a flip-flop or latch in scan mode.
master_clock	Signal is master clock of a flip-flop or latch.
slave_clock	Signal is slave clock of a flip-flop or latch.
scan_master_clock	Signal is master clock of a flip-flop or latch in scan mode.
scan_slave_clock	Signal is slave clock of a flip-flop or latch in scan mode.
read_clock	Clock signal triggers the read operation in a synchronous memory.
write_clock	Clock signal triggers the write operation in a synchronous memory.
read_write_clock	Clock signal triggers both read and write operation in a synchronous memory.
test_clock	Signal is clock in test mode.
test_read_clock	Clock signal triggers the read operation in a synchronous memory in test mode.
test_write_clock	Clock signal triggers the write operation in a synchronous memory in test mode.
test_read_write_clock	Clock signal triggers both read and write operation in a synchronous memory in test mode.
bist_clock	Signal is clock in BIST mode.
bist_read_clock	Clock signal triggers the read operation in a synchronous memory in BIST mode.
bist_write_clock	Clock signal triggers the write operation in a synchronous memory in BIST mode.
bist_read_write_clock	Clock signal triggers both read and write operation in a synchronous memory in BIST mode.

9.7.5 ACTION annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD ACTION = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { asynchronous synchronous }
}
```

Syntax 62— annotation

annotates the action of the signal, which can take the values shown in Table 31.

Table 31—ACTION annotations for a PIN object

Annotation string	Description
`asynchronous`	Signal acts in an asynchronous way, i.e., self-triggered.
`synchronous`	Signal acts in a synchronous way, i.e., triggered by a signal with `SIGNALTYPE CLOCK` or a composite `SIGNALTYPE` with postfix `_CLOCK`.

The `ACTION` annotation applies only to pins with certain `SIGNALTYPE` values, as shown in Table 32. The rule applies also to any composite `SIGNALTYPE` values based on the fundamental values.

Table 32—ACTION applicable in conjunction with fundamental SIGNALTYPE values

Fundamental SIGNALTYPE	Applicable ACTION
`data`	N/A
`address`	N/A
`control`	Synchronous or asynchronous.
`select`	N/A
`enable`	Synchronous or asynchronous.
`tie`	N/A
`clear`	Synchronous or asynchronous.
`set`	Synchronous or asynchronous.
`clock`	N/A, but the presence of `SIGNALTYPE=clock` conditions the validity of `ACTION=synchronous` for other signals.

9.7.6 POLARITY annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD POLARITY = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { high low rising_edge falling_edge double_edge }
}
```

Syntax 63— annotation

annotates the polarity of the pin signal.

The polarity of an input pin (i.e., `DIRECTION = input;`) takes the values shown in Table 33.

Table 33—POLARITY annotations for a PIN

Annotation string	Description
`high`	Signal active high or to be driven high.
`low`	Signal active low or to be driven low.
`rising_edge`	Signal sensitive to rising edge.
`falling_edge`	Signal sensitive to falling edge.
`double_edge`	Signal sensitive to any edge.

The `POLARITY` annotation applies only to pins with certain `SIGNALTYPE` values, as shown in Table 34. The rule applies also to any composite `SIGNALTYPE` values based on the fundamental values.

Table 34—POLARITY applicable in conjunction with fundamental SIGNALTYPE values

Fundamental SIGNALTYPE	Applicable POLARITY value
`data`	N/A
`address`	N/A
`control`	Mode-specific `high` or `low` for composite signaltype.
`select`	N/A
`enable`	Mandatory `high` or `low`.
`tie`	Optional `high` or `low`.
`clear`	Mandatory `high` or `low`.
`set`	Mandatory `high` or `low`.
`clock`	Mandatory `high`, `low`, `rising_edge`, `falling_edge`, or `double_edge`, can be mode-specific for composite signaltype.

Signals with composite signaltypes *mode*`_CLOCK` can have a single polarity or mode-specific polarities.

Example

```
PIN rw {
    SIGNALTYPE = READ_WRITE_CONTROL;
    POLARITY { READ=high; WRITE=low; }
}
PIN rwc {
    SIGNALTYPE = READ_WRITE_CLOCK;
    POLARITY { READ=rising_edge; WRITE=falling_edge; }
}
```

9.7.7 DATATYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD DATATYPE = single_value_annotation {
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
  VALUES { signed unsigned }
 }
```

Syntax 64— annotation

annotates the datatype of the pin, which can take the values shown in Table 35.

Table 35—DATATYPE annotations for a PIN object

Annotation string	Description
`signed`	Result of arithmetic operation is signed 2's complement.
`unsigned`	Result of arithmetic operation is unsigned.

`DATATYPE` is only relevant for bus pins.

9.7.8 INITIAL_VALUE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD INITIAL_VALUE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = boolean_value;
}
```

Syntax 65— annotation

shall 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., a VITAL-compliant simulator) supports the notion of initial value.

9.7.9 SCAN_POSITION annotation

A *xxx* annotation shall be defined using ALF language as shown in .

annotates the position of the pin in scan chain, starting with `1`. Value 0 (default) indicates that the PIN is not on the scan chain. See A.3.1 and A.3.4 for examples.

9.7.10 STUCK annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD SCAN_POSITION = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = unsigned;
  DEFAULT = 0;
}
```

Syntax 66— annotation

```
KEYWORD STUCK = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { stuck_at_0 stuck_at_1 both none }
  DEFAULT = both;
}
```

Syntax 67— annotation

annotates the stuck-at fault model as shown in Table 36.

Table 36—STUCK annotations for a PIN object

Annotation string	Description
`stuck_at_0`	Pin can have stuck-at-0 fault.
`stuck_at_1`	Pin can have stuck-at-1 fault.
`both` (default)	Pin can have both stuck-at-0 and stuck-at-1 faults.
`none`	Pin can not have stuck-at faults.

9.7.11 SUPPLYTYPE

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD SUPPLYTYPE = annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { power ground reference }
}
```

Syntax 68— annotation

A `PIN` with `PINTYPE = SUPPLY` shall have a `SUPPLYTYPE` annotation, as shown in Syntax 69.

9.7.12 SIGNAL_CLASS

A *xxx* annotation shall be defined using ALF language as shown in .

The following new keyword for class reference shall be defined:

*supplytype*_assignment ::=
SUPPLYTYPE = *supplytype*_identifier **;**
*supplytype*_identifier ::=
power
| **ground**
| **reference**

Syntax 69—supply_type assignment

```
KEYWORD SIGNAL_CLASS = annotation {
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
}
```

Syntax 70— annotation

SIGNAL_CLASS

A PIN referring to the same SIGNAL_CLASS belong to the same set of pins related to specific data transaction operations, such as read or write operations. This set of pins is commonly called a l*ogical port*. For example, the ADDRESS, WRITE_ENABLE, and DATA pin of a logical port of a memory have the same SIGNAL_CLASS.

However, the term PORT in ALF is used to define a *physical port* (see ???) rather than a logical port.

SIGNAL_CLASS applies to a PIN with PINTYPE=DIGITAL | ANALOG.
SIGNAL_CLASS is orthogonal to SIGNALTYPE.

Example

```
CLASS portA;
CLASS portB;
CELL my_memory {
    PIN[1:4] addrA { DIRECTION = input;
       SIGNALTYPE = address;
       SIGNAL_CLASS = portA;
    }
    PIN[7:0] dataA { DIRECTION = output;
       SIGNALTYPE = data;
       SIGNAL_CLASS = portA;
    }
    PIN[1:4] addrB { DIRECTION = input;
       SIGNALTYPE = address;
       SIGNAL_CLASS = portB;
    }
    PIN[7:0] dataB { DIRECTION = input;
       SIGNALTYPE = data;
       SIGNAL_CLASS = portB;
    }
    PIN weB { DIRECTION = input;
       SIGNALTYPE = write_enable;
       SIGNAL_CLASS = portB;
    }
}
```

NOTE—The combination of SIGNAL_CLASS and SIGNALTYPE identifies the port type. CLASS portA represents a read port, since it consists of a PIN with SIGNALTYPE = address and a PIN with SIGNALTYPE = data and DIRECTION = output. CLASS portB represents a write port, since it consists of a PIN with SIGNALTYPE = address, a PIN with SIGNALTYPE = data and DIRECTION = input, and a PIN with SIGNALTYPE = write_enable.

9.7.13 SUPPLY_CLASS

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD SUPPLY_CLASS = annotation {
  CONTEXT { PIN PINGROUP CLASS }
  VALUETYPE = identifier;
}
```

Syntax 71— annotation

The following new keyword for class reference shall be defined:

SUPPLY_CLASS
a PIN referring to the same SUPPLY_CLASS belongs to the same power terminal.

For example, digital VDD and digital VSS have the same SUPPLY_CLASS.

SUPPLY_CLASS applies to not only to a PIN with PINTYPE=SUPPLY, but also to a PIN with PINTYPE=DIGITAL or PINTYPE=ANALOG in order to indicate the related set of power supply pins. For instance there can be signal pins related to digital power supply and others related to analog power supply within the same cell.

SUPPLY_CLASS is orthogonal to SUPPLYTYPE.

Example

```
CELL my_adc {
    CLASS dig;
    CLASS ana;
    PIN vdd_dig { PINTYPE=supply; SUPPLYTYPE=power; SUPPLY_CLASS=dig; }
    PIN vss_dig { PINTYPE=supply; SUPPLYTYPE=ground; SUPPLY_CLASS=dig; }
    PIN vdd_ana { PINTYPE=supply; SUPPLYTYPE=power; SUPPLY_CLASS=ana; }
    PIN vss_ana { PINTYPE=supply; SUPPLYTYPE=ground; SUPPLY_CLASS=ana; }
    PIN din { PINTYPE=analog; SUPPLY_CLASS=ana; }
    PIN[7:0] dout { PINTYPE=digital; SUPPLY_CLASS=dig; }
}
```

9.7.14 DRIVETYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD DRIVETYPE = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES {
    cmos nmos pmos cmos_pass nmos_pass pmos_pass
    ttl open_drain open_source
  }
  DEFAULT = cmos;
}
```

Syntax 72— annotation

annotates the drive type for the pin, which can take the values shown in Table 37.

Table 37—DRIVETYPE annotations for a PIN object

Annotation string	Description
`cmos` (default)	Standard cmos signal.
`nmos`	Nmos or pseudo nmos signal.
`pmos`	Pmos or pseudo pmos signal.
`nmos_pass`	Nmos passgate signal.
`pmos_pass`	Pmos passgate signal.
`cmos_pass`	Cmos passgate signal, i.e., the full transmission gate.
`ttl`	TTL signal.
`open_drain`	Open drain signal.
`open_source`	Open source signal.

9.7.15 SCOPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD SCOPE = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { behavior measure both none }
  DEFAULT = both;
}
```

Syntax 73— annotation

annotates the modeling scope of a pin, which can take the values shown in Table 38.

Table 38—SCOPE annotations for a PIN object

Annotation string	Description
`behavior`	The pin is used for modeling functional behavior and events on the pin are monitored for vector expressions in `BEHAVIOR` statements.
`measure`	Measurements related to the pin can be described, e.g., timing or power characterization, and events on the pin are monitored for vector expressions in `VECTOR` statements.
`both` (default)	The pin is used for functional behavior as well as for characterization measurements.
`none`	No model; only the pin exists.

9.7.16 ATTRIBUTE for PIN objects

The attributes shown in Table 39 can be used within a `PIN` object.

Table 39—Attributes within a PIN object

Attribute item	Description
`SCHMITT`	Schmitt trigger signal.
`TRISTATE`	Tristate signal.
`XTAL`	Crystal/oscillator signal.
`PAD`	Pad going off-chip.

The attributes shown in Table 40 are only applicable for pins within cells with `CELLTYPE=memory` and certain values of `SIGNALTYPE`.

Table 40—Attributes for pins of a memory

Attribute item	SIGNALTYPE	Description
`ROW_ADDRESS_STROBE`	`clock`	Samples the row address of the memory.
`COLUMN_ADDRESS_STROBE`	`clock`	Samples the column address of the memory.
`ROW`	`address`	Selects an addressable row of the memory.
`COLUMN`	`address`	Selects an addressable column of the memory.
`BANK`	`address`	Selects an addressable bank of the memory.

The attributes shown in Table 41 are only applicable for pins representing double-rail signals.

Table 41—Attributes for pins representing double-rail signals

Attribute item	Description
INVERTED	Represents the inverted value within a pair of signals carrying complementary values.
NON_INVERTED	Represents the non-inverted value within a pair of signals carrying complementary values.
DIFFERENTIAL	Signal is part of a differential pair, i.e., both the inverted and non-inverted values are always required for physical implementation.

The following restrictions apply for double-rail signals:

— The PINTYPE, SIGNALTYPE, and DIRECTION of both pins shall be the same.
— One PIN shall have the attribute INVERTED, the other NON_INVERTED.
— Either both pins or no pins shall have the attribute DIFFERENTIAL.
— POLARITY, if applicable, shall be complementary as follows:
  HIGH is paired with LOW
  RISING_EDGE is paired with FALLING_EDGE
  DOUBLE_EDGE is paired with DOUBLE_EDGE

9.7.17 Definitions of pin ATTRIBUTE values for memory BIST

The special pin ATTRIBUTE values shown in Table 42 shall be defined for memory BIST.

Table 42—PIN attributes for memory BIST

Attribute item	Description
ROW_INDEX	Pin is a bus with a contiguous range of values, indicating a physical row of a memory.
COLUMN_INDEX	Pin is a bus with a contiguous range of values, indicating a physical column of a memory.
BANK_INDEX	Pin is a bus with a contiguous range of values, indicating a physical bank of a memory.
DATA_INDEX	Pin is a bus with a contiguous range of values, indicating the bit position within a data bus of a memory.
DATA_VALUE	Pin represents a value stored in a physical memory location.

These attributes apply to the pins of the BIST wrapper around the memory rather than to the pins of the memory itself.

The BEHAVIOR statement within TEST shall involve the variables declared as PINs with ATTRIBUTE ROW_INDEX, COLUMN_INDEX, BANK_INDEX, DATA_INDEX, or DATA_VALUE.

9.7.18 CONNECT_CLASS annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD CONNECT_CLASS = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
}
```

Syntax 74— annotation

annotates a declared class object for connectivity determination.

Connectivity rules involving those classes shall apply for the pin.

9.7.19 SIDE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD SIDE = single_value_annotation {
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
  VALUES { left right top bottom }
}
```

Syntax 75— annotation

which can take the values shown in Table 43.

Table 43—SIDE annotations for a PIN object

Annotation string	Description
`left`	Pin is on the left side.
`right`	Pin is on the right side.
`top`	Pin is at the top.
`bottom`	Pin is at the bottom.

9.7.20 ROW and COLUMN annotation

A *xxx* annotation shall be defined using ALF language as shown in .

The following annotation shall be used for a pin in order to indicate the location of the pin within a placement row or column, as shown in Syntax 77.

where *`row`*`_assignment` applies for pins with `SIDE = right | left` and *`column`*`_assignment` applies for pins with `SIDE = top | bottom`.

```
KEYWORD ROW = annotation {
  CONTEXT { PIN PINGROUP }
  VALUETYPE = unsigned;
}
KEYWORD COLUMN = annotation {
  CONTEXT { PIN PINGROUP }
  VALUETYPE = unsigned;
}
```

Syntax 76— annotation

*row*_assignment ::=
ROW = unsigned **;**
*column*_assignment ::=
COLUMN = unsigned **;**

Syntax 77—Pin placement annotation

For bus pins, *row*_assignment and *column*_assignment shall have the form of multi_value_assignments, as shown in Syntax 78.

*row*_multi_value_assignment ::=
ROW { unsigned { unsigned } **} ;**
*column*_multi_value_assignment ::=
COLUMN { unsigned { unsigned } **} ;**

Syntax 78—Row and column multivalue assignments

9.7.21 ROUTING_TYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD ROUTING_TYPE = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { regular abutment ring feedthrough }
  DEFAULT = regular;
}
```

Syntax 79— annotation

A PIN can contain the ROUTING_TYPE statement shown in Syntax 80.

*routing_type*_assignment ::=
ROUTING_TYPE = *routing_type*_identifier **;**
*routing_type*_identifier ::=
regular
| **abutment**
| **ring**
| **feedthrough**

Syntax 80—routing_type assignment

The identifiers have the following definitions:

— *regular*: connection by regular routing
— *abutment*: connection by abutment, no routing
— *ring*: pin forms a ring around the block with connection allowed to any point of the ring
— *feedthrough*: both ends of the pin align and can be used for connection

9.8 NON_SCAN_CELL statement

A *non-scan cell* statement shall be defined as shown in Syntax 81.

```
non_scan_cell ::=
      NON_SCAN_CELL { unnamed_cell_instantiation { unnamed_cell_instantiation } }
    | NON_SCAN_CELL = unnamed_cell_instantiation
    | non_scan_cell_template_instantiation
unnamed_cell_instantiation ::=
      cell_identifier { pin_value { pin_value } }
    | cell_identifier { pin_assignment { pin_assignment } }
pin_value ::=
      pin_variable | boolean_value
```

Syntax 81—NON_SCAN_CELL statement

Example

9.9 PULL statement

A *pull* statement shall be defined as shown in .

```
pull ::=:
      PULL = pull_value ;
    | PULL = pull_value { { pull_item } }
    | pull_template_instantiation
pull_value ::=
      up | down | both | none
pull_item ::=
      voltage_arithmetic_model
    | resistance_arithmetic_model
```

Syntax 82—PULL statement

annotates the pull type for the pin, which can take the values shown in Table 44.

Table 44—PULL annotations for a PIN object

Annotation string	Description
up	Pullup device connected to pin.

Table 44—PULL annotations for a PIN object (Continued)

Annotation string	Description
`down`	Pulldown device connected to pin.
`both`	Pullup and pulldown device connected to pin.
`none` (default)	No pull device.

9.10 WIRE statement and related statements

~~Interconnect parasitics and analysis~~

This section defines interconnect parasitics and analysis.

9.10.1 WIRE statement

A `WIRE` statement XXX, as shown in Syntax 83.

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

Syntax 83—WIRE statement

9.10.1.1 Principles of the WIRE statement

Parasitic descriptions shall be in the context of a `WIRE` statement. The following fundamental modeling styles are supported.

— Statistical wireload models
— Boundary parasitics

Statistical wireload models as well as interconnect analysis calculation models can be used within the context of a `LIBRARY`, `SUBLIBRARY`, or `CELL` statement. The latter applies only for cells with `CELLTYPE=block`, i.e., hierarchical cells. Boundary parasitics apply exclusively for hierarchical cells. Statistical wireload models can be mixed with boundary parasitics within the same `WIRE` statement.

Interconnect analysis models shall also be defined within a `WIRE` statement. However, they shall not be mixed with statistical wireload models or boundary parasitic descriptions.

The purpose of interconnect analysis is to calculate electrical quantities such as `DELAY`, `SLEWRATE`, and noise `VOLTAGE` in the context of a netlist consisting of electrical components, such as `CAPACITANCE`, `RESISTANCE`, and `INDUCTANCE`.

As opposed to boundary parasitics, where the components are connected to physical nodes and pins of a cell, the components represent an abstract network targeted for analysis. The interconnect analysis model specifies a

directive for reducing the parasitic extraction/delay calculation tool to an arbitrary network. In addition, the model specifies the calculation models for delay, noise, etc. in the context of the reduced network.

9.10.1.2 Statistical wireload models

A statistical wireload model is a collection of arithmetic models for estimated the electrical quantities CAPACITANCE, RESISTANCE, and INDUCTANCE, representing the interconnect load and estimated AREA and SIZE of the interconnect nets.

These arithmetic models shall have no PIN annotation. Only environmental quantities such as PROCESS, DERATE_CASE, and TEMPERATURE shall be allowed as arguments in the HEADER.

In addition, the quantities AREA, SIZE, FANOUT, FANIN, and CONNECTIONS are allowed as arguments in the HEADER.

FANOUT and FANIN represent the number of receiver pins and driver pins, respectively, connected to the net. CONNECTIONS is the total number of pins connected to the net. CONNECTIONS equals to the sum of FANOUT and FANIN.

AREA represents a physically measurable area of an object, whereas SIZE represents an abstract symbolic quantity or cost function for area. When AREA or SIZE is used as argument within the HEADER, it shall represent the total area or size, respectively, allocated for place and route of the block for which the wireload model applies. An arithmetic model given for AREA or SIZE itself shall represent the estimated or actual area or size, respectively, of the object in the context of which the model appears. CELL and WIRE are applicable objects for AREA or SIZE models.

In order to convert SIZE to AREA (analogous to converting DRIVE_STRENGTH to RESISTANCE; see Section 8.8.1), an arithmetic model for SIZE with AREA as an argument can be used outside the WIRE statement. Arithmetic models for SIZE inside the WIRE statement shall be interpreted as a calculation model rather than a conversion model.

The total area or size of a block shall be larger or equal to the area or size, respectively, of all objects within the block, i.e., cells and wires.

NOTE—The area or size of a block is design-specific data, whereas the area or size of cells and wires is given in the library.

Example

```
LIBRARY my_library {
   WIRE my_wlm {
      CAPACITANCE {
          HEADER {
             CONNECTIONS { TABLE { 2 3 4 5 10 20 } }
             AREA { TABLE { 1000 10000 100000 } }
          }
          TABLE {
             0.03 0.06 0.08 0.10 0.15 0.25
             0.05 0.10 0.15 0.18 0.25 0.35
             0.10 0.18 0.25 0.32 0.50 0.65
          }
       }
       AREA {
          HEADER {
             CONNECTIONS { TABLE { 2 3 4 5 10 20 } }
```

```
                AREA { TABLE { 1000 10000 100000 } }
            }
            TABLE {
                0.3 0.6 0.8 1.0 1.5 2.5
                0.5 1.0 1.5 1.8 2.5 3.5
                1.0 1.8 2.5 3.2 5.0 6.5
            }
        }
    }
    CELL my_cell {
        AREA = 1.5;
        PIN my_input  { DIRECTION = input;  CAPACITANCE = 0.1; }
        PIN my_output { DIRECTION = output; CAPACITANCE = 0.0; }
    }
}
```

A net routed in a block of `AREA=10000`, driven by an instance of `my_cell` connecting to five receivers (i.e., `CONNECTIONS=5`), each of which is an instance of `my_cell`, shall have an estimated capacitance of `0.18+4*0.1 = 0.58` and wire area of `1.8`. The five cell instances together shall have an area of `7.5`.

NOTE—`CAPACITANCE`, `RESISTANCE`, and `AREA` can each be independent arithmetic models within the `WIRE` statement. No multiplication factor between area and capacitance orn between area and resistance is assumed.

9.10.1.3 Boundary parasitics

Boundary parasitics for a `CELL` can be given within a `WIRE` statement in the context of the `CELL`. The parasitics shall be identified by arithmetic models for `CAPACITANCE`, `RESISTANCE`, and `INDUCTANCE` containing a `NODE` annotation. The syntax is as shown in Syntax 84.

*two_node*_multi_value_assignment ::=
NODE { *node*_identifier *node*_identifier **}**
*four_node*_multi_value_assignment ::=
NODE { *node*_identifier *node*_identifier *node*_identifier *node*_identifier **}**

Syntax 84—Multinode multivalue assignment

where *node*_`identifier` is one of the following:

- a simple `identifier`, referring to a declared `PIN` of the `CELL`.
- a `hierarchical_identifier`, referring to a declared `PORT` of a `PIN` of the `CELL` (see 9.10.4)
- a simple `identifier`, referring to a declared `NODE` of the `WIRE` (see Section 8.15.4)
- a simple `identifier`, not referring to a declared object.
  This can be used for connectivity inside the `WIRE` only.

The *two_node*_`multi_value_assignment` applies for capacitance, resistance, and self-inductance. These components imply the following relationship between voltage and current across the nodes:

$$\text{VOLTAGE(node1, node2)} = \text{RESISTANCE(node1, node2)} \cdot \text{CURRENT(node1, node2)}$$

$$\text{CURRENT(node1, node2)} = \text{CAPACITANCE(node1, node2)} \cdot \frac{d}{dt}\text{VOLTAGE(node1, node2)}$$

$$\text{VOLTAGE(node1, node2)} = \text{INDUCTANCE(node1, node2)} \cdot \frac{d}{dt}\text{CURRENT(node1, node2)}$$

The *four_node*_multi_value_assignment applies for mutual inductance. This component implies the following relationship between voltage and current across the nodes:

$$\text{VOLTAGE(node1, node2)} = \text{INDUCTANCE(node1, node2, node3, node4)} \cdot \frac{d}{dt}\text{CURRENT(node3, node4)}$$

NOTE—Both PIN assignments (e.g., PIN=A;) and NODE assignments (e.g., NODE { A B }) can refer to PINs or PORTs. The fundamental semantic difference between a PIN assignment and a NODE assignment is the PIN assignment within an object defines the object is *applied* or *measured* at the PIN or PORT. (e.g., DELAY and SLEWRATE); the NODE assignment within an object defines the object is fundamentally *connected* with the PIN or PORT in the same way an object inside a PIN is also fundamentally connected with the PIN. Therefore, the CAPACITANCE with NODE assignment is a more detailed way of describing a CAPACITANCE of a PIN, whereas a CAPACITANCE with PIN assignment describes a load capacitance, which is applied externally to the pin.

A CELL can contain a WIRE statement describing boundary parasitics as well as PIN statements containing arithmetic models for CAPACITANCE, RESISTANCE, or INDUCTANCE. In this case the latter shall be considered as a reduced form of the former. An analysis tool shall either use the set of components inside the PIN or inside the WIRE, but not a combination of both.

Example

```
CELL my_cell {
    PIN A { PINTYPE = digital; CAPACITANCE = 4.8; RESISTANCE = 37.9;
       PORT p1 { VIEW = physical; } // see 9.10
       PORT p2 { VIEW = none; } // see 9.10
    }
    PIN B { PINTYPE = digital; CAPACITANCE = 2.6; }
    PIN gnd { PINTYPE = supply; SUPPLYTYPE = ground; }
    WIRE my_boundary_parasitics {
       CAPACITANCE = 1.3 { NODE { A.p1 gnd } }
       CAPACITANCE = 2.8 { NODE { A.p2 gnd } }
       RESISTANCE = 65 { NODE { A.p1 A.p2 } }
       CAPACITANCE = 0.7 { NODE { A.p1 B } }
       CAPACITANCE = 1.9 { NODE { B gnd } }
    }
}
```

This example corresponds to the netlist shown in Figure 11.

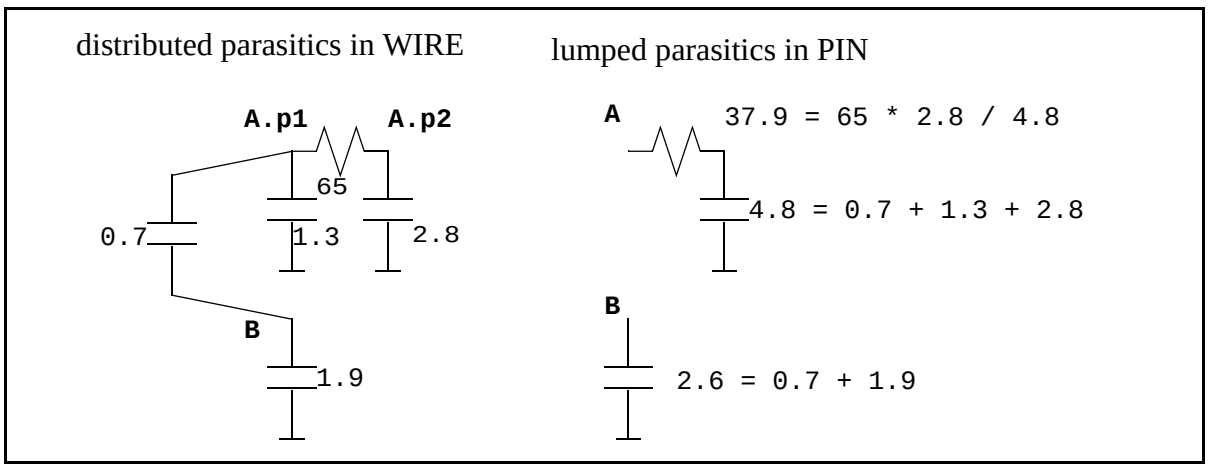


Figure 11—Example of boundary parasitic description

The distributed parasitics in the WIRE statement can be reduced to the lumped parasitics in the PIN statement.

9.10.1.4 Interconnect delay and noise calculation

Calculation models for DELAY and SLEWRATE can be described in the context of a VECTOR inside a WIRE. The PIN assignments in these models shall refer to pre-declared NODEs inside the WIRE.

Example

```
WIRE my_interconnect_model {
    /* node declarations */
    /* electrical component declarations */
    VECTOR ( (01 n0 ~> 01 n5) | (10 n0 ~> 10 n5) ) {
       /* DELAY model */
       /* SLEWRATE model */
    }
}
```

The pre-declared electrical components which are part of the network can be used within an EQUATION without being re-declared in the HEADER of the model.

Example

```
DELAY {
    FROM { PIN = n0; } TO { PIN = n5; }
    EQUATION {
       R1*(C1+C2+C3+C4+C5) + R2*(C2+C3+C4+C5)
       + R3*(C3+C4+C5) + R4*(C4+C5) + R5*C5
    }
}
```

External components or stimuli which are not part of the network shall be declared in the HEADER. Also, all arguments for TABLE-based models shall be in the HEADER. To avoid re-declaration of pre-declared components, an EQUATION shall also be used for those arguments in the HEADER which refer to pre-declared components.

Example

```
SLEWRATE {
    PIN = n5;
    HEADER {
       SLEWRATE { PIN = n0; TABLE {/* numbers */} }
       RESISTANCE  { EQUATION { R1+R2+R3+R4+R5 } TABLE {/* numbers */} }
       CAPACITANCE { EQUATION { C1+C2+C3+C4+C5 } TABLE {/* numbers */} }
    }
    TABLE { /* numbers */ }
}
```

In order to model crosstalk delay and noise, at least two driver and receiver nodes are required. The symbolic state * (see 5.4.13) shall be used to indicate the signal subjected to noise.

Example

```
WIRE interconnect_model_with_coupling {
    NODE aggressor_source { NODETYPE = driver; }
    NODE victim_source    { NODETYPE = driver; }
    NODE aggressor_sink   { NODETYPE = receiver; }
    NODE victim_sink      { NODETYPE = receiver; }
    NODE vdd { NODETYPE = power;  }
    NODE gnd { NODETYPE = ground; }
    CAPACITANCE cc { NODE {aggressor_sink victim_sink}}
    CAPACITANCE cv { NODE {victim_sink gnd }}
    RESISTANCE  rv { NODE {victim_source victim_sink}}
    VECTOR ( 01 aggressor_sink -> ?* victim_sink -> *? victim_sink ) {
       /* xtalk noise model */
    }
    VECTOR (
       ( 01 aggressor_source <&> 01 victim_source )
        -> 01 aggressor_sink -> 01 victim_sink
    ) {
    /* xtalk DELAY model */
    }
}
```

Example for noise model

```
VOLTAGE {
       PIN = victim_sink;
       MEASUREMENT = peak;
       CALCULATION = incremental;
       HEADER {
          SLEWRATE   tra { PIN = aggressor_sink; }
          VOLTAGE     va { NODE {vdd gnd} }
       }
       EQUATION { (1-EXP(-tra/(rv*cv)))*va*rv*cc/tra }
    }
}
```

Example for delay model

```
DELAY {
    FROM { PIN = victim_source; } TO { PIN = victim_sink; }
    CALCULATION = incremental;
    HEADER {
       SLEWRATE tra { PIN = aggressor_sink; }
       SLEWRATE trv { PIN = victim_source; }
    }
    EQUATION { (1-EXP(-tra/(rv*cv)))*rv*cc*trv/tra }
}
```

The VOLTAGE model applies for a rising aggressor signal while the victim signal is stable. The DELAY model applies for rising victim signal simultaneous with or followed by a rising aggressor signal at the coupling point. The VECTOR implicitly defines the time window of interaction between aggressor and victim; interaction occurs only if the aggressor signal at the coupling point intervenes during the propagation of the victim signal from its source to the coupling point. Both VOLTAGE and DELAY represent incremental numbers.

9.10.1.5 SELECT_CLASS annotation for WIRE statement

A sophisticated tool can support more than one interconnect model. Each calculation model can have its “netlist” with the appropriate validity range of the RC components. For instance, a lumped model can be used for short nets and a distributed model can be used for longer nets. Also, models with different accuracy for the same net can be defined. For instance, the lumped model can be used for estimation purpose and the distributed model for signoff.

For this purpose, classes can be defined to select a set of models. The selection shall be defined by the user, in a similar way as a user can select wireload models for pre-layout parasitic estimation. The selected class shall be indicated by the `SELECT_CLASS` annotation within the `WIRE` statement.

Example

```
LIBRARY my_library {
    CLASS estimation;
    CLASS verification;
    WIRE rough_model_for_short_nets {
       SELECT_CLASS = estimation; /* etc.*/
    }
    WIRE detailed_model_for_short_nets {
       SELECT_CLASS = verification; /* etc.*/
    }
    WIRE rough_model_for_long_nets {
       SELECT_CLASS = estimation; /* etc.*/
    }
    WIRE detailed_model_for_long_nets {
       SELECT_CLASS = verification; /* etc.*/
    }
}
```

9.10.2 NODE statement

A `NODE` statement XXX, as shown in Syntax 85.

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

Syntax 85—NODE statement

The nodes used for interconnect analysis shall be declared within the `WIRE` statement, using the following syntax.

```
node ::=
        NODE node_identifier { all_purpose_items }
```

The `NODETYPE` annotation and the `NODE_CLASS` annotation also specifically apply to a `NODE`.

```
nodetype_annotation ::=
   NODETYPE = nodetype_identifier ;

nodetype_identifier ::=
     ground
   | power
   | source
   | sink
   | driver
   | receiver
```

— A *driver node* is the interface between a cell output pin and interconnect
— A *receiver node* is the interface between interconnect and a cell input pin
— A *source node* is a virtual start point of signal propagation; it can be collapsed with a driver node
— A *sink node* is a virtual end point of signal propagation; it can be collapsed with a receiver node
— A *power node* provides the current for rising signals at the source/driver side and a reference for logic high signals at the sink/receiver side
— A *ground node* provides the current for falling signals at the source/driver side and a reference for logic low signals at the sink/receiver side

The arithmetic models for electrical components which are part of the network shall have names and NODE annotations, referring either to the pre-declared nodes or to internal nodes which need not be declared.

Example

```
WIRE my_interconnect_model {
    NODE n0  { NODETYPE = source;   }
    NODE n2  { NODETYPE = driver;   }
    NODE n4  { NODETYPE = receiver; }
    NODE n5  { NODETYPE = sink;     }
    NODE vdd { NODETYPE = power;    }
    NODE vss { NODETYPE = ground;   }
    RESISTANCE R1 { NODE { n0 n1 } }
    RESISTANCE R2 { NODE { n1 n2 } }
    RESISTANCE R3 { NODE { n2 n3 } }
    RESISTANCE R4 { NODE { n3 n4 } }
    RESISTANCE R5 { NODE { n4 n5 } }
    CAPACITANCE C1 { NODE { n1 vss } }
    CAPACITANCE C2 { NODE { n2 vss } }
    CAPACITANCE C3 { NODE { n3 vss } }
    CAPACITANCE C4 { NODE { n4 vss } }
    CAPACITANCE C5 { NODE { n5 vss } }
}
```

This example is illustrated in Figure 12.

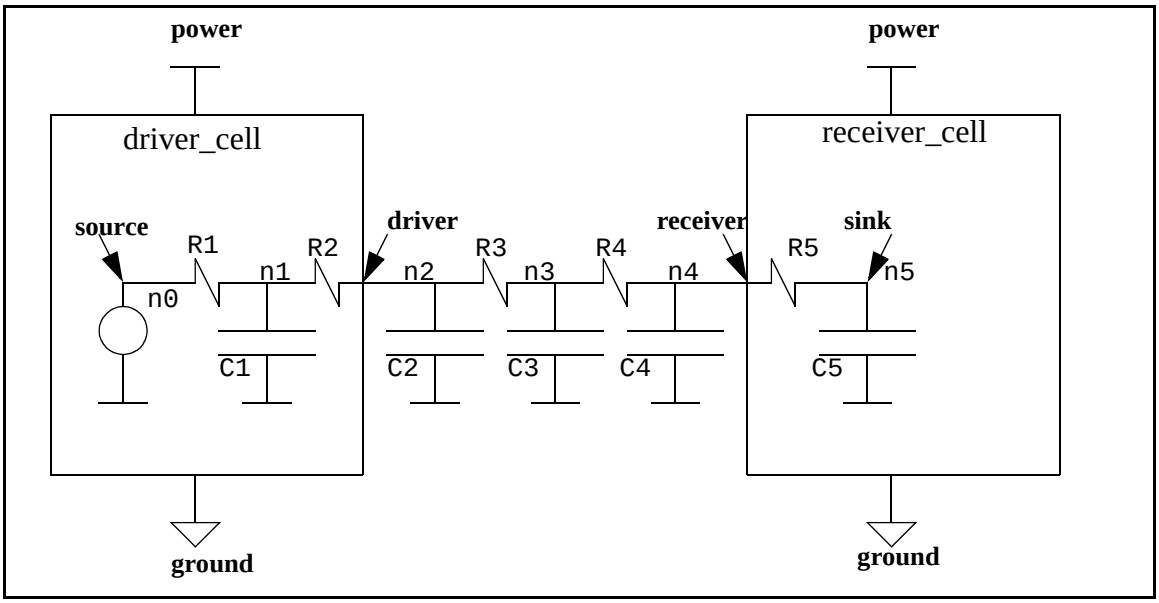


Figure 12—Example for interconnect description

The NODE_CLASS annotation is optional and orthogonal to the NODETYPE annotation.

*node_class*_annotation ::=
NODE_CLASS = *node_class*_identifier **;**

The NODE_CLASS annotation shall refer to a pre-declared CLASS within the WIRE statement to indicate which node belongs to which device in the case of separate power supplies.

Example

```
WIRE my_interconnect_model {
    CLASS driver_cell;
    CLASS receiver_cell;
    NODE n0   { NODETYPE = source; NODE_CLASS = driver_cell; }
    NODE n2   { NODETYPE = driver; NODE_CLASS = driver_cell; }
    NODE n4   { NODETYPE = receiver; NODE_CLASS = receiver_cell; }
    NODE n5   { NODETYPE = sink;     NODE_CLASS = receiver_cell; }
    NODE vdd1 { NODETYPE = power;  NODE_CLASS = driver_cell; }
    NODE vss1 { NODETYPE = ground; NODE_CLASS = driver_cell; }
    NODE vdd2 { NODETYPE = power;  NODE_CLASS = receiver_cell; }
    NODE vss2 { NODETYPE = ground; NODE_CLASS = receiver_cell; }
}
```

If NODE_CLASS is not specified, the nodes with NODETYPE=power|ground are supposed to be global. The DC-connected nodes with NODETYPE=driver|source and NODETYPE=receiver| sink are supposed to belong to the same device.

9.11 VECTOR declaration

A VECTOR statement XXX, as shown in Syntax 86.

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

Syntax 86—VECTOR statement

9.12 Annotations in context of VECTOR declaration

9.12.1 PURPOSE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD  = annotation {
  CONTEXT = VECTOR;
  VALUETYPE = ;
  VALUES {  }
  DEFAULT = ;
}
```

Syntax 87— annotation

A CLASS is a generic object which can be referenced inside another object. An object referencing a class inherits all children object of that class. In addition to this general reference, the usage of the keyword CLASS in conjunction with a predefined prefix (e.g., CONNECT_CLASS, SWAP_CLASS, RESTRICT_CLASS, EXISTENCE_CLASS, or CHARACTERIZATION_CLASS) also carries a specific semantic meaning in the context of its usage. Note the keyword *prefix*_CLASS is used for referencing a class, whereas the definition of the class always uses the keyword CLASS. Thus a class can have multiple purposes. With the growing number of usage models of the class concept, it is useful to include the purpose definition in the class itself in order to make it easier for specific tools to identify the classes of relevance for that tool.

A CLASS object can contain the PURPOSE annotation, which can take one or multiple values. A VECTOR entitled to inherit the PURPOSE annotation from the CLASS can also contain the PURPOSE annotation, as shown in Syntax 88.

```
vector_purpose_assignment ::=
   PURPOSE { purpose_identifier { purpose_identifier } }
vector_purpose_identifier :: =
    bist
  | test
  | timing
  | power
  | integrity
```

Syntax 88—PURPOSE annotation

9.12.2 OPERATION annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD  = annotation {
  CONTEXT = VECTOR;
  VALUETYPE = ;
  VALUES {  }
  DEFAULT = ;
}
```

Syntax 89— annotation

The OPERATION statement inside a VECTOR shall be used to indicate the combined definition of signal values or signal changes for certain operations which are not entirely controlled by a single signal.

*operation*_assignment ::=
OPERATION = *operation*_identifier **;**

An OPERATION within the context of a VECTOR indicates certain a function of a cell, such as a memory write, or change to some state, such as test mode. Many functions are not controlled by a single pin and are therefore not able to be defined by the use of SIGNALTYPE alone. The VECTOR shall describe the complete operation, including the sequence of events on input and expected output signals, such that one operation can be followed seamlessly by the next.

The following values shall be predefined:

*operation*_identifier ::=
read
| **write**
| **read_modify_write**
| **write_through**
| **start**
| **end**
| **refresh**
| **load**
| **iddq**

Their definitions are:

— *read*: read operation at one address
— *write*: write operation at one address
— *read_modify_write*: read followed by write of different value at same address
— *start*: first operation required in a particular mode
— *end*: last operation required in a particular mode
— *refresh*: operation required to maintain the contents of the memory without modifying it
— *load*: operation for loading control registers
— *iddq*: operation for supply current measurements in quiescent state

With exception of *iddq*, all values apply for only cells with CELLTYPE=memory.

The EXISTENCE_CLASS (see 9.12.5) within the context of a VECTOR shall be used to identify which operations can be combined in the same mode. OPERATION is orthogonal to EXISTENCE_CLASS. The EXISTENCE_CLASS statement is only necessary, if there is more than one mode of operation.

Example 1

```
CLASS normal_mode { PURPOSE = test; }
CLASS fast_page_mode { PURPOSE = test; }
VECTOR ( ! WE && (
       ?! addr -> 01 RAS -> 10 RAS ->
       ?! addr -> 01 CAS -> X? dout -> 10 CAS -> ?X dout
    ) ) {
    OPERATION = read; EXISTENCE_CLASS = normal_mode;
}
VECTOR ( WE && (
       ?! addr -> 01 RAS -> 10 RAS ->
       ?! addr -> ?? din -> 01 CAS -> 10 CAS
    ) ) {
    OPERATION = write; EXISTENCE_CLASS = normal_mode;
}
VECTOR ( ! WE && (?! addr -> 01 CAS -> X? dout -> 10 CAS -> ?X dout ) ) {
    OPERATION = read; EXISTENCE_CLASS = fast_page_mode;
}
VECTOR ( WE && ( ?! addr -> ?? din -> 01 CAS -> 10 CAS ) ) {
    OPERATION = write; EXISTENCE_CLASS = fast_page_mode;
}
VECTOR ( ?! addr -> 01 RAS -> 10 RAS ) {
    OPERATION = start; EXISTENCE_CLASS = fast_page_mode;
}
```

NOTE—The complete description of a “read” operation also contains the behavior after the “read” is disabled.

Example 2

```
VECTOR ( 01 read_enb -> X? dout -> 10 read_enb -> ?X dout) {
    OPERATION = read; // output goes to X in read-off
}
VECTOR ( 01 read_enb -> ?? dout -> 10 read_enb -> ?- dout) {
    OPERATION = read; // output holds is value in read-off
}
```

9.12.3 LABEL annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD  = annotation {
  CONTEXT = VECTOR;
  VALUETYPE = ;
  VALUES {  }
  DEFAULT = ;
}
```

Syntax 90— annotation

ensures SDF matching with conditional delays across Verilog, VITAL, etc.

See the end of B.3 for an example.

9.12.4 EXISTENCE_CONDITION annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD  = annotation {
  CONTEXT = VECTOR;
  VALUETYPE = ;
  VALUES {  }
  DEFAULT = ;
}
```

Syntax 91— annotation

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 co-factor to the vector itself. The default existence condition is *True*.

Example

```
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.

9.12.5 EXISTENCE_CLASS annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD  = annotation {
  CONTEXT = VECTOR;
  VALUETYPE = ;
  VALUES {  }
  DEFAULT = ;
}
```

Syntax 92— annotation

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. The 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

```
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.

9.12.6 CHARACTERIZATION_CONDITION annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD  = annotation {
  CONTEXT = VECTOR;
  VALUETYPE = ;
  VALUES {  }
  DEFAULT = ;
}
```

Syntax 93— annotation

For characterization tools, the characterization condition shall be treated as if the boolean expression was a co-factor to the vector itself. For all other applications, the characterization condition shall be disregarded. The 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`),
(`c & d`), or (`!c & !d`), since they all satisfy (`c | !d`) . However, the only condition chosen for delay characterization is (`c & !d`).

9.12.7 CHARACTERIZATION_VECTOR annotation

A *xxx* annotation shall be defined using ALF language as shown in .

The characterization vector is provided for the case where the vector expression cannot be constructed using the vector and a boolean co-factor. 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 can be provided, but not both. If none is provided, the vector itself shall be used by the characterization tool.

```
KEYWORD  = annotation {
  CONTEXT = VECTOR;
  VALUETYPE = ;
  VALUES {  }
  DEFAULT = ;
}
```

Syntax 94— annotation

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.

9.12.8 CHARACTERIZATION_CLASS annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD  = annotation {
  CONTEXT = VECTOR;
  VALUETYPE = ;
  VALUES {  }
  DEFAULT = ;
}
```

Syntax 95— annotation

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.

9.13 Incremental definitions for VECTOR

In general, it is illegal to re-declare an ALF object (see 4.1, *Rule 4*). However, there are objects which merely define the context for other objects. When objects are incrementally added to the library, it is natural to re-declare the context as well.

Vector-specific timing, power, signal integrity characterization can be done by different groups, each of which comes up with a set of vectors for the characterization domain. Some of the vectors can be accidentally the same. Also, timing, power, signal integrity characterization can be done in different releases of the library. In both scenarios, the “incremental vector definitions” make the merging process easier.

Multiple instances of the same `VECTOR` shall be legal for the purpose of incrementally adding children objects. The first instance of the `VECTOR` shall be interpreted as a declaration. All following instances shall be inter-

preted as supplemental definitions of the VECTOR. The rule of illegal re-declaration shall apply for the children objects within a VECTOR.

Example

```
// the following is legal
VECTOR ( 01 A -> 01 Z ) {
    DELAY = 1 { FROM { PIN = A; } TO { PIN = Z; } }
}
VECTOR ( 01 A -> 01 Z ) {
    ENERGY = 25 ;
}
// the following is illegal
VECTOR ( 01 A -> 01 Z ) {
    DELAY = 1 { FROM { PIN = A; } TO { PIN = Z; } }
}
VECTOR ( 01 A -> 01 Z ) {
    DELAY = 2 { FROM { PIN = A; } TO { PIN = Z; } }
}
```

9.14 Statements for physical modeling

~~Overview~~

Table 45 summarizes the ALF statements for physical modeling.

Table 45—Statements in ALF describing physical objects

Statement	Scope	Comment
LAYER	LIBRARY, SUBLIBRARY	Description of a plane provided for physical objects consisting of electrically conducting material.
VIA	LIBRARY, SUBLIBRARY	Description of a physical object for electrical connection between layers.
SITE	LIBRARY, SUBLIBRARY	Placement grid for a class of physically placeable objects.
BLOCKAGE	CELL	Physical object on a layer, forming an obstruction against placing or routing other objects.
PORT	PIN	Physical object on a layer, providing electrical connections to a pin.
PATTERN	VIA, RULE, BLOCKAGE, PORT	Physical object on a layer, described for the purpose of defining relationships with other physical objects.
RULE	LIBRARY, SUBLIBRARY, CELL, PIN	Set of rules defining calculable relationships between physical objects.
ANTENNA	LIBRARY, SUBLIBRARY, CELL	Set of rules defining restrictions for physical size of electrically connected objects for the purpose of manufacturing.
ARTWORK	VIA, CELL	Reference to an imported object from GDS2.

Table 45—Statements in ALF describing physical objects (Continued)

Statement	Scope	Comment
ARRAY	LIBRARY, SUBLIBRARY	Description of a regular grid for placement, global and detailed routing.
geometric model	PATTERN	Description of the geometric form of a physical object.
REPEAT	physical object	Algorithm to replicate a physical object in a regular way.
SHIFT	physical object	Specification to shift a physical object in x/y direction.
FLIP	physical object	Specification to flip a physical object around an axis.
ROTATE	physical object	Specification to rotate a physical object around an axis.
BETWEEN	CONNECTIVITY, DISTANCE	Reference to objects with a relation to each other.

9.14.1 LAYER statement

A LAYER statement is defined as shown in Syntax 96.

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

Syntax 96—LAYER statement

~~layer ::=~~
~~LAYER identifier { layer_items }~~

~~layer_items ::=~~
~~layer_item { layer_item }~~

~~layer_item ::=~~
~~all_purpose_item~~
~~| arithmetic_model~~
~~| arithmetic_model_container~~

~~The syntax and semantics of all_purpose_item, arithmetic_model_container, and arithmetic_model are defined in~~ 11.7 ~~and~~ 11.16~~.~~

Specific items applicable for LAYER are listed in Table 46.

Table 46—Items for LAYER description

Item	Applies for layer	Usable ALF statement	Comment
Purpose	all	`PURPOSE = <identifier> ;`	See 9.14.2
Property	routing, cut, master	`PROPERTY { ... }`	See 3.2.7
Current density limit	routing, cut	`LIMIT { CURRENT { ... MAX { ... } } }`	See 7.5, 8.1.2, 7.6.1, 8.9.1, and 9.14.5
Resistance	routing, cut	`RESISTANCE { ... }`	See 8.7.2 and 9.14.5
Capacitance	routing	`CAPACITANCE {... }`	See 8.7.2 and 9.14.5
Default width or minimum width	routing	`WIDTH { DEFAULT = <number>; }`	See 7.1.4., Section 9.2, and 9.14.5
Manufacturing tolerance for width	routing	`WIDTH { MIN = <number>; TYP = <number>; MAX = <number>; }`	See 7.6.1, 8.9.1, and 9.14.5
Default wire extension	routing	`EXTENSION { DEFAULT = <number>; }`	See 9.17.3.3 and 9.14.5
Height	routing, cut, master	`HEIGHT = <number>;`	See Section 9.2
Thickness	routing, cut, master	`THICKNESS = <number>;`	See Section 9.2
Preferred routing direction	routing	`PREFERENCE`	See 9.14.4

NOTE—Rules involving relationships between objects within one or several layers is described in the RULE statement (see 9.16.1).

9.14.2 PURPOSE annotation

The purpose of each layer shall be identified using the PURPOSE annotation.

*layer_purpose*_assignment ::=
  PURPOSE = *layer_purpose*_identifier **;**

*layer_purpose*_identifier ::=
    routing
  | **cut**
  | **substrate**
  | **dielectric**
  | **reserved**
  | **abstract**

The identifiers have the following definitions:

— *routing*: layer provides electrical connections within one plane
— *cut*: layer provides electrical connections between planes
— *substrate*: layer(s) at the bottom

— *dielectric*: provides electrical isolation between planes
— *reserved*: layer is for proprietary use only
— *abstract*: not a manufacturable layer, used for description of boundaries between objects

LAYER statements shall be in sequential order defined by the manufacturing process, starting bottom-up in the following sequence: one or multiple substrate layers, followed by alternating cut and routing layers, then the dielectric layer. Abstract layers can appear at the end of the sequence.

9.14.3 PITCH annotation

The PITCH annotation identifies the routing pitch for a layer with PURPOSE=routing.

```
pitch_annotation ::=
   PITCH = non_negative_number ;
```

The pitch is measured between the center of two adjacent parallel wires routed on the layer.

9.14.4 PREFERENCE annotation

The PREFERENCE annotation for LAYER shall have the following form:

```
routing_preference_annotation ::=
   PREFERENCE = routing_preference_identifier ;

routing_preference_identifier ::=
     horizontal
   | vertical
```

The purpose is to indicate the preferred routing direction.

9.14.5 Example

This example contains a default width (the syntax is all_purpose_item), resistance, capacitance, and current limits (the syntax is arithmetic_model) for arbitrary wires in a routing layer. Since width and thickness are arguments of the models, special wires and fat wires are also taken into account.

```
LAYER metal1 {
    PURPOSE = routing;
    PREFERENCE { HORIZONTAL = 0.75; VERTICAL = 0.25; }
    WIDTH { DEFAULT = 0.4; MIN = 0.39; TYP = 0.40; MAX = 0.41; }
    THICKNESS { DEFAULT = 0.2; MIN = 0.19; TYP = 0.20; MAX = 0.21; }
    EXTENSION { DEFAULT = 0; }
    RESISTANCE {
       HEADER { LENGTH WIDTH THICKNESS TEMPERATURE }
       EQUATION {

          0.5*(LENGTH/(WIDTH*THICKNESS))
          *(1.0+0.01*(TEMPERATURE-25))
       }
    }
    CAPACITANCE {
       HEADER { AREA PERIMETER }
       EQUATION { 0.48*AREA + 0.13*PERIMETER*THICKNESS }
```

```
      }
      LIMIT {
         CURRENT ac_limit_for_avg {
            UNIT = mAmp ;
            MEASUREMENT = average ;
            HEADER {
               WIDTH { UNIT = uM; TABLE { 0.4 0.8 } }
               FREQUENCY { UNIT = megHz; { 1 100 } }
               THICKNESS { UNIT = uM; TABLE { 0.2 0.4 } }
            }
            TABLE {
               2.0e-6 4.0e-6 1.5e-6 3.0e-6
               4.0e-6 8.0e-6 3.0e-6 6.0e-6
            }
         }
         CURRENT ac_limit_for_rms {
            UNIT = mAmp ;
            MEASUREMENT = rms ;
            HEADER {
               WIDTH { UNIT = uM; TABLE { 0.4 0.8 } }
               FREQUENCY { UNIT = megHz; { 1 100 } }
               THICKNESS { UNIT = uM; TABLE { 0.2 0.4 } }
            }
            TABLE {
               4.0e-6 7.0e-6 4.5e-6 7.5e-6
               8.0e-6 14.0e-6 9.0e-6 15.0e-6
            }
         }
         CURRENT ac_limit_for_peak {
            UNIT = mAmp ;
            MEASUREMENT = peak ;
            HEADER {
               WIDTH { UNIT = uM; TABLE { 0.4 0.8 } }
               FREQUENCY { UNIT = megHz; { 1 100 } }
               THICKNESS { UNIT = uM; TABLE { 0.2 0.4 } }
            }
            TABLE {
               6.0e-6 10.0e-6 5.9e-6 9.9e-6
               12.0e-6 20.0e-6 11.8e-6 19.8e-6
            }
         }
         CURRENT dc_limit {
            UNIT = mAmp ;
            MEASUREMENT = static ;
            HEADER {
               WIDTH { UNIT = uM; TABLE { 0.4 0.8 } }
               THICKNESS { UNIT = uM; TABLE { 0.2 0.4 } }
            }
            TABLE { 2.0e-6 4.0e-6 4.0e-6 8.0e-6 }
         }
      }
   }
```

9.15 VIA statement and related statements

This section defines the VIA statement and its annotations.

9.15.1 VIA statement

A VIA statement is defined as shown in Syntax 97.

```
via ::=
      VIA via_identifier { via_items }
    | VIA via_identifier ;
    | via_template_instantiation
via_items ::=
    via_item { via_item }
via_item ::=
      all_purpose_item
    | pattern
    | artwork
```

Syntax 97—VIA statement

```
via ::=
          VIA [ identifier ] { via_items }

via_items ::=
          via_item { via_item }

via_item ::=
          all_purpose_item
   |      pattern
   |      arithmetic_model
   |      arithmetic_model_container
```

The VIA statement shall contain at least three patterns, referring to the cut layer and two adjacent routing layers. Stacked vias can contain more than three patterns.

The all_purpose_items and arithmetic_models for VIA are listed in Table 47.

Table 47—Items for VIA description

Item	Usable ALF statement	Comment
Property	PROPERTY	See 3.2.7
Resistance	RESISTANCE	See 8.7.2
GDS2 reference	ARTWORK	See Section 9.4 and 9.15.3
Usage	USAGE	See 9.15.2 and 9.15.3

9.15.2 USAGE annotation

The USAGE annotation for a VIA shall have one of the following mutually exclusive values.

```
usage_annotation ::=
   USAGE = usage_identifier ;

usage_identifier ::=
     default
   | non_default
   | partial_stack
   | full_stack
```

The identifiers have the following definitions:

— *default*: via can be used per default
— *non_default*: via can only be used if authorized by a RULE
— *partial_stack*: via contains 3 patterns: lower and upper routing layer and cut layer in-between. It can only be used to build stacked vias. The bottom of a stack can be a default or a non_default via.
— *full_stack*: via contains 2N+1 patterns (N>1). It describes the full stack from bottom to top.

9.15.3 Example

```
VIA via_with_two_contacts_in_x_direction {
    ARTWORK = GDS2_name_of_my_via {
       SHIFT { HORIZONTAL = -2; VERTICAL = -3; }
       ROTATE = 180;
    }
    PATTERN via_contacts {
       LAYER = cut_1_2 ;
       RECTANGLE { 1 1 3 3 }
       REPEAT = 2 {
          SHIFT{ HORIZONTAL = 4; }
          REPEAT = 1 {
             SHIFT { VERTICAL = 4; }
    }   }   }
    PATTERN lower_metal {
       LAYER = metal_1 ;
       RECTANGLE { 0 0 8 4 }
    }
    PATTERN upper_metal {
       LAYER = metal_2 ;
       RECTANGLE { 0 0 8 4 }
    }
}
```

A TEMPLATE (see 3.2.6) can be used to define a construction rule for a via.

```
TEMPLATE my_via_rule
    VIA <via_rule_name> {
       PATTERN via_contacts {
          LAYER = cut_1_2 ;
          RECTANGLE { 1 1 3 3 }
          REPEAT = <x_repeat> {
             SHIFT{ HORIZONTAL = 4; }
             REPEAT = <y_repeat> {
                SHIFT { VERTICAL = 4; }
```

```
            }   }   }
            PATTERN lower_metal {
               LAYER = metal_1 ;
               RECTANGLE { 0 0 <x_cover> <y_cover> }
            }
            PATTERN upper_metal {
               LAYER = metal_2 ;
               RECTANGLE { 0 0 <x_cover> <y_cover> }
            }
        }
    }
```

A static instance of the TEMPLATE can be used to create the same via as in the first example (except for the reference to GDS2):

```
    my_via_rule {
        via_rule_name = via_with_two_contacts_in_x_direction;
        x_cover = 8;
        y_cover = 4;
        x_repeat = 2;
        y_repeat = 1;
    }
```

A dynamic instance of the TEMPLATE (see 5.6.8) can be used to create a via rule.

```
    my_via_rule = dynamic {
        via_rule_name = via_with_NxM_contacts;
        x_cover = 8;
        y_cover = 4;
        x_repeat {
           HEADER { x_cover { TABLE { 4 8 12 16 } } }
           TABLE { 1 2 3 4 }
        }
        y_repeat {
           HEADER { y_cover { TABLE { 4 8 12 16 } } }
           TABLE { 1 2 3 4 }
        }
    }
```

Instead of defining fixed values for the placeholders, here the mathematical relationships between the placeholders are defined, which can generate a via rule for any set of values.

9.15.4 VIA reference statement

Certain physical objects can contain a reference to one or more vias, as shown in Syntax 98.

```
via_reference ::=
          VIA { via_instantiations }

via_instantiations ::=
          via_instantiation { via_instantiation }

via_instantiation ::=
          via_identifier { geometric_transformations }
```

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

Syntax 98—VIA reference statement

The *via*_identifier shall be the name of an already defined VIA.

Example for a via reference in a PORT, see Section 9.10.

~~VIA reference~~

~~A RULE can contain a reference to one or more vias, using the via_reference statement (see).~~

9.16 Statements related to physical design rules

Add lead-in text

9.16.1 RULE statement

A RULE statement is defined as shown in Syntax 99.

```
rule ::=
      RULE rule_identifier { rule_items }
    | RULE rule_identifier ;
    | rule_template_instantiation
rule_items ::=
    rule_item { rule_item }
rule_item ::=
      all_purpose_item
    | pattern
    | via_reference
```

Syntax 99—RULE statement

~~rule ::=~~
~~RULE [identifier] { rule_items }~~

~~rule_items ::=~~
~~rule_item { rule_item }~~

~~rule_item ::=~~
~~pattern~~
~~all_purpose_item~~
~~via_reference~~
~~arithmetic_model_container~~
~~arithmetic_model~~

The all_purpose_items for RULE are listed in Table 48.

Table 48—Items for RULE description

Item	Usable ALF statement	Comment
Rule is for same net or different nets	CONNECTIVITY	See 9.16.4.2 and Section 9.15
Spacing rule	LIMIT { DISTANCE ... }	See 7.5 and 9.16.1.1
Overhang rule	LIMIT { OVERHANG ... }	See 7.5 and 9.16.1.2

The rules for spacing and overlap, respectively, shall be expressed using the LIMIT construct with DISTANCE and OVERHANG, respectively, as keywords for the arithmetic models (see 7.5 and 7.6.1). The keywords HORIZONTAL and VERTICAL shall be introduced as qualifiers for arithmetic submodels (see 7.6) to distinguish rules for different routing directions. If these qualifiers are not used, the rule shall apply in any routing direction.

9.16.1.1 Width-dependent spacing

An example of width-dependent spacing is:

```
RULE width_and_length_dependent_spacing {
    PATTERN segment1 { LAYER = metal_1; SHAPE = line; }
    PATTERN segment2 { LAYER = metal_1; SHAPE = line; }
    CONNECTIVITY {
       CONNECT_RULE = cannot_short;
       BETWEEN { segment1 segment2 }
    }
    LIMIT {
       DISTANCE { BETWEEN { segment1 segment2 }
          MIN {
             HEADER {
                WIDTH w1 {
                   PATTERN = segment1;
                   /* TABLE, if applicable */
                }
                WIDTH w2 {
                   PATTERN = segment2;
                   /* TABLE, if applicable */
                }
                LENGTH common_run {
                   BETWEEN { segment1 segment2 }
                   /* TABLE, if applicable */
                }
             }
             /* EQUATION or TABLE */
          }
          MAX { /* some technology have MAX spacing rules */ }
       }
    }
}
```

Spacing rules dependent on routing direction can be expressed as follows:

```
LIMIT {
    DISTANCE { BETWEEN { segment1 segment2 }
       HORIZONTAL {
          MIN { /* HEADER, EQUATION or TABLE */ }
       }
       VERTICAL {
          MIN { /* HEADER, EQUATION or TABLE */ }
       }
    }
}
```

9.16.1.2 End-of-line rule

End-of-line rules can be expressed as follows:

```
RULE lonely_via {
    PATTERN via_lower { LAYER = metal_1; SHAPE = line; }
    PATTERN via_cut   { LAYER = cut_1_2; }
    PATTERN via_upper { LAYER = metal_2; SHAPE = end;  }
    PATTERN adjacent  { LAYER = metal_2; SHAPE = line; }
    CONNECTIVITY {
       CONNECT_RULE = must_short;
       BETWEEN { via_lower via_cut via_upper }
    }
    CONNECTIVITY {
       CONNECT_RULE = cannot_short;
       BETWEEN { via_upper adjacent }
    }
    LIMIT {
       OVERHANG {
          BETWEEN { via_cut via_upper }
          MIN {
             HEADER {
                DISTANCE {
                   BETWEEN { via_cut adjacent }
                   /* TABLE, if applicable */
                }
             }
             /* TABLE or EQUATION */
          }
       }
    }
}
```

Overhang dependent on routing direction can be expressed as follows:

```
LIMIT {
    OVERHANG { BETWEEN { via_cut via_upper }
       HORIZONTAL {
          MIN { /* HEADER, EQUATION or TABLE */ }
       }
       VERTICAL {
```

```
            MIN { /* HEADER, EQUATION or TABLE */ }
         }
      }
   }
```

9.16.1.3 Redundant vias

Rules for redundant vias can be expressed as follows:

```
   RULE constraint_for_redundant_vias {
       PATTERN via_lower { LAYER = metal_1; }
       PATTERN via_cut   { LAYER = cut_1_2; }
       PATTERN via_upper { LAYER = metal_2; }
       CONNECTIVITY {
          CONNECT_RULE = must_short;
          BETWEEN { via_lower via_cut via_upper }
       }
       LIMIT {
          WIDTH {
             PATTERN = via_cut;
             MIN = 3; MAX = 5;
          }
          DISTANCE {
             BETWEEN { via_cut }
             MIN = 1; MAX = 2;
          }
          OVERHANG {
             BETWEEN { via_lower via_cut }
             MIN = 2; MAX = 4;
          }
          OVERHANG {
             BETWEEN { via_upper via_cut }
             MIN = 2; MAX = 4;
          }
       }
   }
```

9.16.1.4 Extraction rules

Extraction rules can be expressed as follows:

```
   RULE parallel_lines_same_layer {
       PATTERN segment1 { LAYER = metal_1; SHAPE = line; }
       PATTERN segment2 { LAYER = metal_1; SHAPE = line; }
       CAPACITANCE {
          BETWEEN { segment1 segment2 }
          HEADER {
             DISTANCE {
                BETWEEN { segment1 segment2 }
                /* TABLE, if applicable */
             }
             LENGTH {
                BETWEEN { segment1 segment2 }
                /* TABLE, if applicable */
```

```
            }
         }
         /* EQUATION or TABLE */
      }
   }
```

9.16.1.5 RULES within BLOCKAGE or PORT

General width-dependent spacing rules can not apply to blockages which are abstractions of smaller blockages collapsed together. The spacing rule between the constituents of the blockage and their neighboring objects shall be applied instead.

For example, a blockage can consist of two parallel wires in vertical direction of `width=1` and `distance=1`. They can be collapsed to form a blockage of `width=3`. Left and right of the blockage, the spacing rule shall be based on the width of the constituent wires (i.e., `1`) instead of the width of the blockage (i.e., `3`).

Therefore, it shall be legal within a `RULE` statement to appear within the context of a `BLOCKAGE` or `PORT` and reference a `PATTERN` which has been defined within the context of the `BLOCKAGE` or `PORT`.

Example

```
CELL my_cell {
    BLOCKAGE my_blockage {
       PATTERN my_pattern {
          LAYER = metal1;
          RECTANGLE { 5 0 8 10 }
       }
       RULE for_my_pattern {
          PATTERN my_metal1 { LAYER = metal1; }
          LIMIT {
             DISTANCE {
                BETWEEN { my_metal1 my_pattern }
                MIN = 1;
             }
          }
       }
    }
}
```

It shall also be legal to define the spacing rule, which normally would be inside the `RULE` statement, directly within the context of a `PATTERN` using the `LIMIT` construct and the arithmetic model for `DISTANCE`. This arithmetic model shall not contain a `BETWEEN` statement. The spacing rule shall apply between the `PATTERN` and any external object on the same layer.

Example

```
CELL my_cell {
    BLOCKAGE my_blockage {
       PATTERN p1 {
          LAYER = metal1;
          RECTANGLE { 5 0 8 10 }
          LIMIT { DISTANCE { MIN = 1; } }
       }
```

```
        }
    }
```

9.16.2 ANTENNA statement

An ANTENNA statement is defined as shown in Syntax 100.

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

Syntax 100—ANNTENA statement

~~antenna ::=~~
~~**ANTENNA** [*antenna*_identifier] { antenna_items }~~

~~antenna_items ::=~~
~~antenna_item { antenna_item }~~

~~antenna_item ::=~~
~~all_purpose_item~~
~~| arithmetic_model~~
~~| arithmetic_model_container~~

~~The syntax and semantics of all_purpose_item, arithmetic_model_container, and arithmetic_model are already defined in defined in~~ 11.7 ~~and~~ 11.16.~~.~~

The items applicable for ANTENNA are shown in Table 49.

Table 49—Items for ANTENNA description

Item	Usable ALF statement	Scope	Comment
Maximum allowed antenna size	LIMIT { SIZE { MAX { ... } } }	LIBRARY, SUBLIBRARY CELL, PIN	See 7.5, 8.1.2, 7.6.1, 8.9.1, and 9.16.2.1
Calculation method for antenna size	SIZE { HEADER { ... } TABLE { ...} or SIZE [id] { HEADER { ... } EQUATION { ...}	LIBRARY, SUBLIBRARY	See 8.1.3, and 9.16.2.1
Argument values for antenna size calculation	*argument* = *value* ; or *argument* = *value* { ... }	CELL, PIN	See 11.2 and 9.16.2.1

The use of the keyword SIZE (see 8.1.3) in the context of ANTENNA is proposed to represent an abstract, dimensionless model of the antenna size. It is related to the area of the net which forms the antenna, but it is not neces-

sary a measure of area. It can be a measure of area ratio as well. However, the arguments of the calculation function for antenna SIZE shall be measurable data, such as AREA, PERIMETER, LENGTH, THICKNESS, WIDTH, and HEIGHT of metal segments connected to the net. The argument also need an annotation defining the applicable LAYER for the metal segments.

A process technology can have more than one antenna rule calculation method. In this case, the *antenna*_identifier is mandatory for each rule.

Antenna rules apply for routing and cut layers connected to poly silicon and eventually to diffusion. The CONNECT_RULE statement in conjunction with the BETWEEN statement shall be used to specify the connected layers. Connectivity shall only be checked up to the highest layer appearing in the CONNECT_RULE statement. Connectivity through higher layers shall not be taken into account, since such connectivity does not yet exist in the state of manufacturing process when the antenna effect occurs.

9.16.2.1 Layer-specific antenna rules

Antenna rules can be checked individually for each layer. In this case, the SIZE model contains only two or three arguments: AREA of the layer or perimeter (calculated from the LENGTH and WIDTH) of the layer causing the antenna effect, the area of poly silicon, and, eventually, the area of diffusion.

Example

```
ANTENNA individual_m1 {
    LIMIT { SIZE { MAX = 1000; } }
    SIZE {
       CONNECTIVITY {
          CONNECT_RULE = must_short; BETWEEN { metal1 poly }
       }
       CONNECTIVITY {
          CONNECT_RULE = cannot_short; BETWEEN { metal1 diffusion }
       }
       HEADER {
          AREA a1 { LAYER = metal1; }
          AREA a0 { LAYER = poly; }
       }
       EQUATION { a1 / a0 }
    }
ANTENNA individual_m2 {
    LIMIT { SIZE { MAX = 1000; } }
    SIZE {
       CONNECTIVITY {
          CONNECT_RULE = must_short; BETWEEN { metal2 poly }
       }
       CONNECTIVITY {
          CONNECT_RULE = cannot_short; BETWEEN { metal2 diffusion }
       }
       HEADER {
          AREA a2 { LAYER = metal2; }
          AREA a0 { LAYER = poly; }
       }
       EQUATION { a2 / a0 }
    }
}
```

9.16.2.2 All-layer antenna rules

Antenna rules can also be checked globally for all layers. In that case, the SIZE model contains area or perimeter of all layers as additional arguments.

Example

```
ANTENNA global_m2_m1 {
    LIMIT { SIZE { MAX = 2000; } }
    SIZE {
       CONNECTIVITY {
       CONNECT_RULE = must_short;
          BETWEEN { metal2 metal1 poly }
       }
       CONNECTIVITY {
          CONNECT_RULE = cannot_short;
          BETWEEN { metal2 diffusion }
       }
       HEADER {
          AREA a2 { LAYER = metal1; }
          AREA a1 { LAYER = metal1; }
          AREA a0 { LAYER = poly; }
       }
       EQUATION { (a2 + a1) / a0 }
    }
}
```

9.16.2.3 Cumulative antenna rules

Antenna rules can also be checked by accumulating the individual effect. In that case, the SIZE model can be represented as a nested arithmetic model, each of which contain the model of the individual effect.

Example

```
ANTENNA accumulate_m2_m1 {
    LIMIT { SIZE { MAX = 3000; } }
    SIZE {
       HEADER {
          SIZE ratio1 {
             CONNECTIVITY {
                CONNECT_RULE = must_short;
                BETWEEN { metal1 poly }
             }
             CONNECTIVITY {
                CONNECT_RULE = cannot_short;
                BETWEEN { metal1 diffusion }
             }
             HEADER {
                AREA a1 { LAYER = metal1; }
                AREA a0 { LAYER = poly; }
             }
             EQUATION { a1 / a0 }
          }
          SIZE ratio2 {
```

```
            CONNECTIVITY {
               CONNECT_RULE = must_short;
               BETWEEN { metal2 poly }
            }
            CONNECTIVITY {
               CONNECT_RULE = cannot_short;
               BETWEEN { metal2 diffusion }
            }
            HEADER {
               AREA a2 { LAYER = metal2; }
               AREA a0 { LAYER = poly; }
            }
            EQUATION { a2 / a0 }
         }
      }
      EQUATION { ratio1 + ratio2 }
   }
}
```

The arguments `a0` in `ratio1` and `ratio2` can are not the same. In `ratio1`, `a0` represents the area of poly silicon connected to `metal1` in a net. In `ratio2`, `a0` represents the area of poly silicon connected to `metal2` in a net, where the connection can be established through more than one subnet in `metal1`.

9.16.2.4 Illustration

Consider the structure shown in Figure 13.

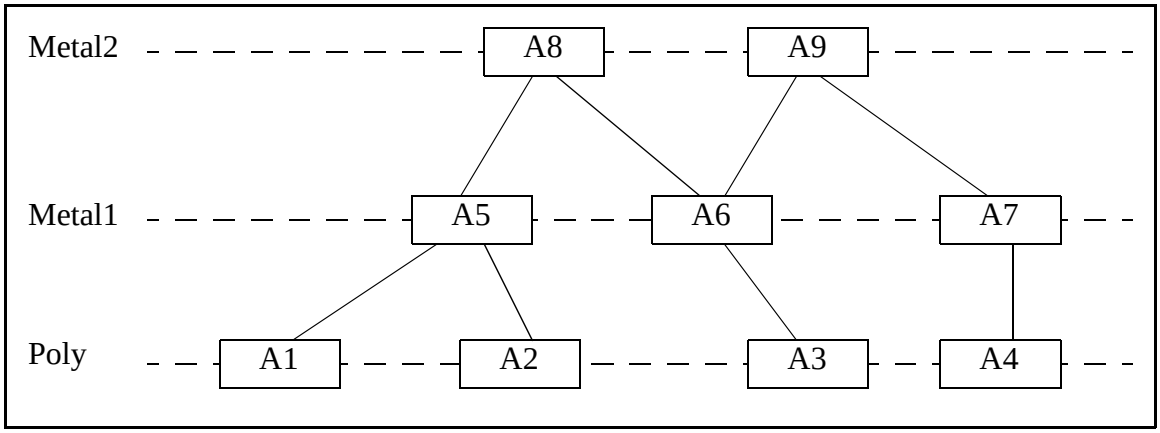


Figure 13—Metal-poly illustration

Checking this structure against the rules in the examples yields the following results:

```
individual_m1:
    1000 > A5 / (A1+A2)
    1000 > A6 / A3
    1000 > A7 / A4
individual_m2:
    1000 > (A8+A9) / (A1+A2+A3+A4)
```

```
global_m2_m1:
    2000 > (A8+A9+A5+A6+A7) / (A1+A2+A3+A4)
accumulate_m2_m1:
    3000 > (A8+A9) / (A1+A2+A3+A4) + A5 / (A1+A2)
    3000 > (A8+A9) / (A1+A2+A3+A4) + A6 / A3
    3000 > (A8+A9) / (A1+A2+A3+A4) + A7 / A4
```

9.16.3 BLOCKAGE statement

This section defines the BLOCKAGE statement and its use.

9.16.3.1 Definition

A BLOCKAGE statement is defined as shown in Syntax 101.

```
blockage ::=
    BLOCKAGE blockage_identifier { blockage_items }
  | BLOCKAGE blockage_identifier ;
  | blockage_template_instantiation
blockage_items ::=
    blockage_item { blockage_item }
blockage_item ::=
    all_purpose_item
  | pattern
  | rule
  | via_reference
```

Syntax 101—BLOCKAGE statement

~~blockage ::=~~
~~**BLOCKAGE** [identifier] **{**~~
~~[all_purpose_items]~~
~~[patterns]~~
~~**}**~~

~~See~~ 11.7 ~~for applicable all_purpose_items.~~

9.16.3.2 Example

```
CELL my_cell {
    BLOCKAGE my_blockage {
       PATTERN p1 {
          LAYER = metal1;
          RECTANGLE { -1 5 3 8 }
          RECTANGLE { 6 12 3 8 }
       }
       PATTERN p2 {
          LAYER = metal2;
          RECTANGLE { -1 5 3 8 }
       }
    }
}
```

The BLOCKAGE consists of two rectangles covering metal1 and one rectangle covering metal2.

9.16.4 PORT statement

A port is a collection of geometries within a pin, representing electrically equivalent points. A PORT statement is defined as shown in Syntax 102.

```
port ::=
     PORT port_identifier { port_items }
   | PORT port_identifier ;
   | port_template_instantiation
port_items ::=
    port_item { port_item }
port_item ::=
     all_purpose_item
   | pattern
   | rule
   | via_reference
```

Syntax 102—PORT statement

~~port ::=~~
~~**PORT** *port*_identifier **;**~~
~~| **PORT** [*port*_identifier] **{**~~
~~[all_purpose_items]~~
~~[patterns]~~
~~[via_reference]~~
~~**}**~~

A numerical digit can be used as the first character in *port*_identifier. In this case the number shall be proceeded by the escape character (see 10.3.8) in the declaration of the PORT.

The PORT statement is legal within the context of a PIN statement. For this purpose, the syntax for pin_item (see ~~11.11~~) shall be augmented as follows:

```
pin_item ::=
     all_purpose_item
   | arithmetic_model
   | port
```

A pin can have either no PORT statement, an arbitrary number of PORT statements with a *port*_identifier, or exactly one PORT statement without a *port*_identifier.

9.16.4.1 VIA reference

A PORT can contain a reference to one or more vias by using the via_reference statement (see xxx).

Example

```
VIA my_via { /* put via definition here */ }

// later in the same library
CELL my_cell {
    PIN my_pin {
       PORT my_port {
          VIA {
```

```
                my_via { SHIFT { HORIZONTAL = 1.0 ; VERTICAL = 2.0 ; } }
                my_via { SHIFT { HORIZONTAL = 5.0 ; VERTICAL = 8.0 ; } }
            }
        }
    }
}
```

The VIA my_via is instantiated twice in the PORT my_port within the PIN my_pin of the CELL my_cell. The origin of the instantiated vias is shifted with respect to the origin of the cell, as specified by the SHIFT statements.

9.16.4.2 CONNECTIVITY rules for PORT and PIN

By default, all connections to a pin shall be made to the same port. Different ports of a pin shall not be connected externally. Those defaults can be overridden by using connectivity rules for ports within a pin.

Pins of the same cell shall not be shorted externally by default. This default can also be overridden by using connectivity rules for pins within a cell.

Example

```
PIN A {
    PORT P1 { VIEW=physical; }
}

PIN B {
    PORT Q1 { VIEW=physical; }
    PORT Q2 { VIEW=physical; }
    PORT Q3 { VIEW=physical; }
    CONNECTIVITY {
       CONNECT_RULE = can_short;
       BETWEEN { Q1 Q3 }
    }
    CONNECTIVITY {
       CONNECT_RULE = cannot_short;
       BETWEEN { Q1 Q2 }
    }
    CONNECTIVITY {
       CONNECT_RULE = cannot_short;
       BETWEEN { Q2 Q3 }
    }
}
CONNECTIVITY {
    CONNECT_RULE = must_short;
    BETWEEN { A B }
}
```

The router can make external connections between Q1 and Q3, but not between Q1 and Q2 or between Q2 and Q3, respectively. The router shall make an external connection between A.P1 and any port of B (B.Q1, B.Q2, or B.Q3).

9.16.4.3 Reference of a declared PORT in a PIN annotation

In the context of timing modeling, a PORT can have the semantic meaning of a PIN. For examples, PORTs can be used as FROM and/or TO points of delay measurements — use a reference by a hierarchical_identifier.

Example

```
CELL my_cell {
    PIN A {
       DIRECTION = input;
       PORT p1;
       PORT p2;
    }
    PIN Z {
       DIRECTION = output;
    }
    VECTOR ( 01 A -> 01 Z ) {
       DELAY {
          FROM { PIN = A.p1; }
          TO { PIN = Z; }
       }
       DELAY {
          FROM { PIN = A.p2; }
          TO { PIN = Z; }
       }
    }
}
```

9.16.4.4 VIEW annotation

A subset of values for the VIEW annotation inside a PIN (see 6.4.1) shall be applicable for a PORT as well.

```
port_view_annotation ::=
   VIEW = port_view_identifier ;

port_view_identifier ::=
      physical
    | none
```

VIEW=physical shall qualify the PORT as a real port with the possibility to connect a routing wire to it.

VIEW=none shall qualify the PORT as a virtual port for modeling purpose only.

9.16.4.5 LAYER annotation

The *layer*_annotation can appear inside a PORT (see Section 9.10).

9.16.4.6 ROUTING_TYPE

A PORT can inherit the ROUTING_TYPE from its PIN or it can have its own ROUTING_TYPE annotation.

9.17 Statements related to physical geometry

Add lead-in text

9.17.1 SITE statement

A SITE statement is defined as shown in Syntax 103.

```
site ::=
      SITE site_identifier { site_items }
    | SITE site_identifier ;
    | site_template_instantiation
site_items ::=
    site_item { site_item }
site_item ::=
      all_purpose_item
    | ORIENTATION_CLASS_one_level_annotation
    | SYMMETRY_CLASS_one_level_annotation
```

Syntax 103—SITE statement

~~site ::=~~
~~SITE site_identifier { all_purpose_items }~~

The *width*_annotation and *height*_annotation (see Section 9.2) are mandatory.

9.17.1.1 ORIENTATION_CLASS and SYMMETRY_CLASS

A set of CLASS statements shall be used to define a set of legal orientations applicable to a SITE. Both the CLASS and the SITE statements shall be within the context of the same LIBRARY or SUBLIBRARY.

```
orientation_class ::=
   CLASS orientation_class_identifier {
      [ geometric_transformations ]
   }
```

To refer to a predefined orientation class, use the ORIENTATION_CLASS statement within a SITE and/or a CELL. ORIENTATION of a CELL means the orientation of the cell itself. ORIENTATION of a SITE means the orientation of rows that can be created using that site.

```
orientation_class_multivalue_annotation ::=
   ORIENTATION { orientation_class_identifiers }
```

The SYMMETRY_CLASS statement shall be used for a SITE to indicate symmetry between legal orientations. Multiple SYMMETRY statements shall be legal to enumerate all possible combinations in case they cannot be described within a single SYMMETRY statement.

```
symmetry_class_multivalue_annotation ::=
   SYMMETRY_CLASS { orientation_class_identifiers }
```

Legal orientation of a cell within a site shall be defined as the intersection of legal cell orientation and legal site orientation. If there is a set of common legal orientations for both cell and site without symmetry, the orientation of cell instance and site instance shall match.

If there is a set of common legal orientations for both cell and site with symmetry, the cell can be placed on the side using any orientation within that set.

Case 1: no symmetry

Site has legal orientations A and B. Cell has legal orientations A and B. When the site is instantiated in the A orientation, the cell shall be placed in the A orientation.

Case 2: symmetry

Site has legal orientations A and B and symmetry between A and B. Cell has legal orientations A and B. When the site is instantiated in the A orientation, the cell can be placed in the A or B orientation.

9.17.1.2 Example

```
LIBRARY my_library {
    CLASS north { ROTATE = 0; }
    CLASS flip_north { ROTATE = 0; FLIP = 0; }
    CLASS south { ROTATE = 180; }
    CLASS flip_south { FLIP = 90; }

    SITE Site1 {
       ORIENTATION_CLASS { north flip_north }
    }

    SITE Site2 {
       ORIENTATION_CLASS { north flip_north south flip_south}
       SYMMETRY_CLASS { north flip_north }
       SYMMETRY_CLASS { south flip_south }
    }
    CELL Cell1 {
       SITE { Site1 Site2 }
       ORIENTATION_CLASS { north flip_north }
    }
    CELL Cell2 {
       SITE { Site2 }
       ORIENTATION_CLASS { north south }
    }
}
```

`Cell1` can be placed on `site1`. The orientation of `Site1` and `Cell1` shall match because there is no symmetry between `north` and `flip_north` in `Site1`.

`Cell1` can be placed on `Site2`, provided `Site2` is instantiated in the `north` or `flip_north` orientation. The orientation of `site2` and `cell1` need not match because of the symmetry between `north` and `flip_north` in `Site2`.

`Cell2` can be placed on `Site2`, provided `Site2` is instantiated in the `north` or `south` orientation. The orientation of `Site2` and `Cell2` shall match because there is no symmetry between `north` and `south` in `Site2`.

9.17.2 ARRAY statement

An ARRAY statement is defined as shown in Syntax 104.

```
array ::=
     ARRAY array_identifier { array_items }
   | ARRAY array_identifier ;
   | array_template_instantiation
array_items ::=
    array_item { array_item }
array_item ::=
     all_purpose_item
   | PURPOSE_single_value_annotation
   | geometric_transformation
```

Syntax 104—ARRAY statement

```
array ::=
          ARRAY identifier {
                all_purpose_items
                geometric_transformations
          }
```

The geometric_transformations define the locations of the starting points within the array and the number of repetitions of the components of the array. Details are defined in the next section.

9.17.2.1 PURPOSE annotation

Each array shall have a PURPOSE assignment.

```
array_purpose_assignment ::=
   PURPOSE = array_purpose_identifier ;

array_purpose_identifier ::=
      floorplan
    | placement
    | global
    | routing
```

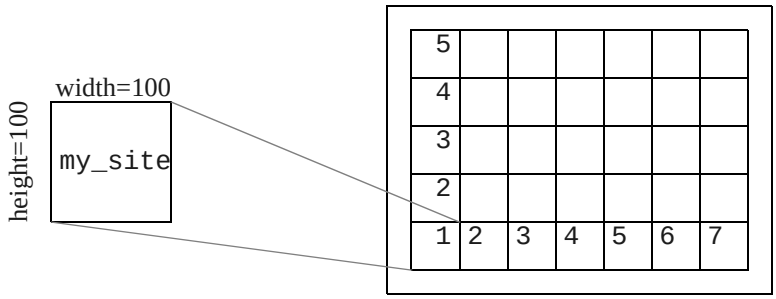
An array with purpose **floorplan** or **placement** shall have a reference to a SITE and a *shift*_annotation_container, *rotate*_annotation, and eventually a *flip*_annotation to define the location and orientation of the SITE in the context of the array.

An array with purpose **routing** shall have a reference to one or more routing LAYERs and a *shift*_annotation_container to define the location of the starting point.

An array with purpose **global** shall have a *shift*_annotation_container to define the location of the starting point.

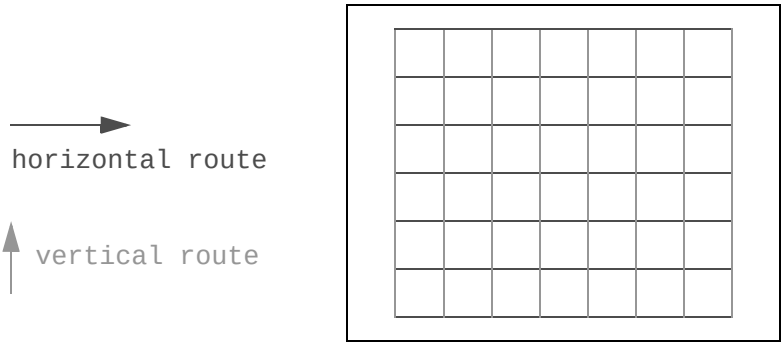
9.17.2.2 Examples

Example 1



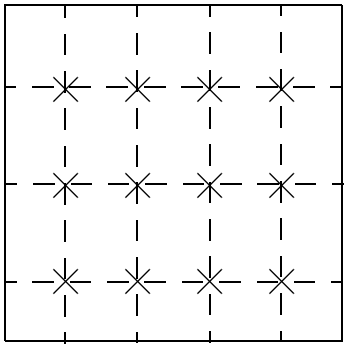

```
ARRAY grid_for_my_site {
    PURPOSE = placement;
    SITE = my_site;
    SHIFT { HORIZONTAL = 50; VERTICAL = 50; }
    REPEAT = 7 {
       SHIFT { HORIZONTAL = 100; }
       REPEAT = 5 {
          SHIFT { VERTICAL = 5; }
       }
    }
}
```

Example 2




```
ARRAY grid_for_detailed_routing {
    PURPOSE = routing;
    LAYER { metal1 metal2 metal3 }
    SHIFT { HORIZONTAL = 100; VERTICAL = 50; }
    REPEAT = 7 {
       SHIFT { VERTICAL = 100; }
       REPEAT = 8 {
          SHIFT { HORIZONTAL = 100; }
       }
    }
}
```

Example 3



```
ARRAY grid_for_global_routing {
    PURPOSE = global;
    SHIFT { HORIZONTAL = 100; VERTICAL = 100; }
    REPEAT = 3 {
       SHIFT { VERTICAL = 150; }
       REPEAT = 4 {
          SHIFT { HORIZONTAL = 100; }
       }
    }
}
```

9.17.3 PATTERN statement

A PATTERN statement is defined as shown in Syntax 105.

pattern ::=
  PATTERN *pattern*_identifier **{** pattern_items **}**
  | **PATTERN** *pattern*_identifier **;**
  | *pattern*_template_instantiation
pattern_items ::=
  pattern_item { pattern_item }
pattern_item ::=
  all_purpose_item
  | *SHAPE*_single_value_annotation
  | *LAYER*_single_value_annotation
  | *EXTENSION*_single_value_annotation
  | *VERTEX*_single_value_annotation
  | geometric_model
  | geometric_transformation

Syntax 105—PATTERN statement

~~pattern ::=~~
~~**PATTERN** [identifier] **{**~~
~~[all_purpose_items]~~
~~[geometric_models]~~
~~[geometric_transformations]~~
~~**}**~~

9.17.3.1 SHAPE annotation

The SHAPE annotation is defined as follows

```
shape_assignment ::=
   SHAPE = shape_identifier ;

shape_identifier ::=
     line
   | tee
   | cross
   | jog
   | corner
   | end
```

SHAPE applies only for a PATTERN in a routing layer, as shown in Figure 14. The default is `line`.

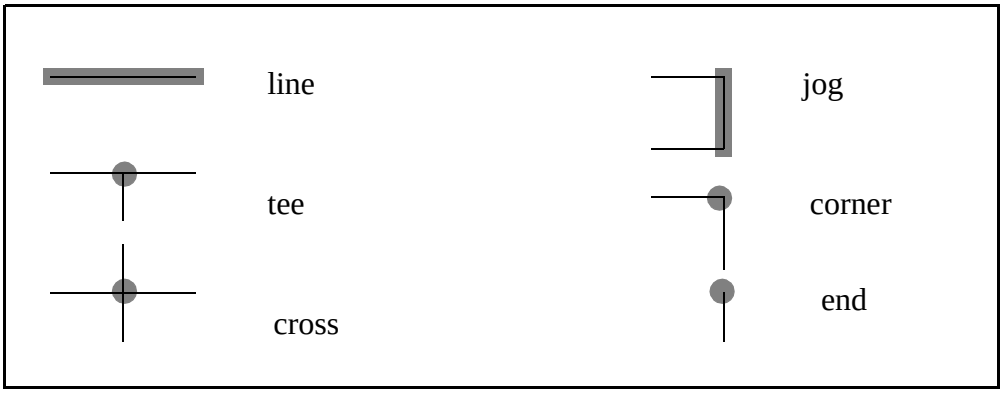


Figure 14—Routing layer shapes

`line` and `jog` represent routing segments, which can have an individual `LENGTH` and `WIDTH`. The `LENGTH` *between* routing segments is defined as the common run length. The `DISTANCE` *between* routing segments is measured orthogonal to the routing direction.

`tee`, `cross`, and `corner` represent intersections between routing segments. `end` represents the end of a routing segment. Therefore, they have points rather than lines as references. The points can have an `EXTENSION`. The `DISTANCE` between points can be measured straight or by using `HORIZONTAL` and `VERTICAL`.

9.17.3.2 LAYER annotation

The *`layer`*`_annotation` defines the layer where the object resides. The layer shall have been declared before.

```
layer_annotation ::=
   LAYER = layer_identifier ;
```

9.17.3.3 EXTENSION annotation

The *`extension`*`_annotation` specifies the value by which the drawn object is extended at all sides.

```
extension_annotation ::=
   EXTENSION = non_negative_number ;
```

The default value of *extension*_annotation is 0.

9.17.3.4 VERTEX annotation

The *vertex*_annotation shall appear only in conjunction with the *extension*_annotation. It specifies the form of the extended object, as shown in Figure 15.

```
vertex_annotation ::=
   VERTEX = vertex_identifier ;

vertex_identifier ::=
      round
   | straight
```

The default value of *vertex*_annotation is **straight**.

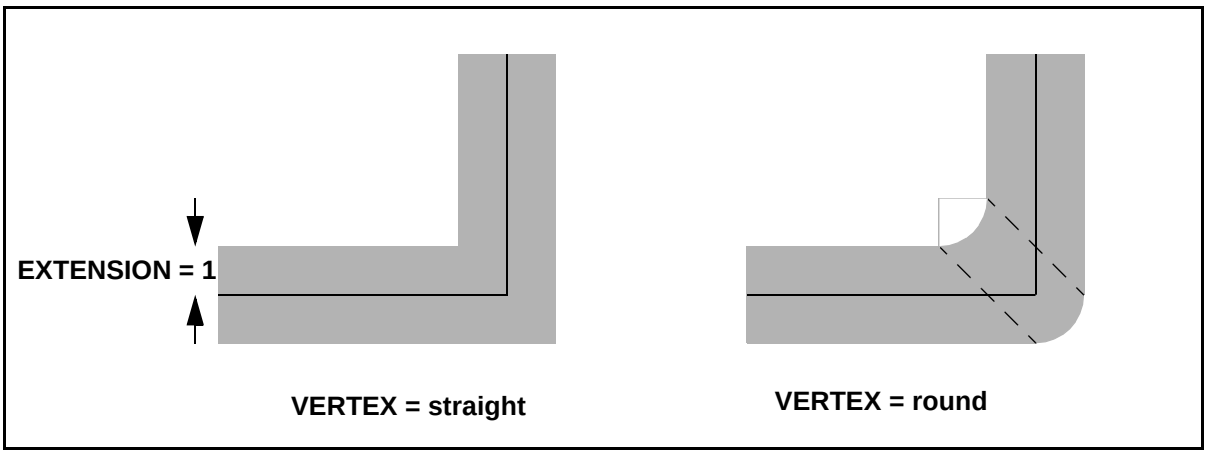


Figure 15—Illustration of VERTEX annotation

9.17.3.5 PATTERN with geometric model

A geometric_model describes the form of a physical object; it does not describe a physical object itself. The geometric_model shall be in the context of a PATTERN.

A pattern can contain geometric_model statements, geometric transformation statements (see 9.17.6.5), and all_purpose_items (see 11.7).

9.17.3.6 Example

```
PATTERN {
    LAYER = metal1;
    EXTENSION = 1;
    DOT { COORDINATES { 5 10 } }
}
```

This object is effectively a square, with a lower left corner (`x=4,y=9`) and upper right corner (`x=6,y=11`).

9.17.4 ARTWORK statement

An `ARTWORK` statement is defined as shown in Syntax 106.

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

Syntax 106—ARTWORK statement

~~artwork ::=~~
~~**ARTWORK =** *artwork*_identifier **{**~~
~~[geometric_transformations]~~
~~{ pin_assignments }~~
~~**}**~~

The `ARTWORK` statement creates a reference between the cell in the library and the original cell imported from a physical layout database (e.g., GDS2).

The `geometric_transformations` define the operations for transformation from the artwork geometry to the actual cell geometry. In other words, the artwork is considered as the original object whereas the cell is the transformed object.

The imported cell can have pins with different names. The LHS of the `pin_assignments` describes the pin names of the original cell, the RHS describes the pin names of the cell in this library. See 11.4 for the syntax of `pin_assignments` .

Example

```
CELL my_cell {
    PIN A { /* fill in pin items */ }
    PIN Z { /* fill in pin items */ }
    ARTWORK = \GDS2$!@#$ {
       SHIFT { HORIZONTAL = 0; VERTICAL = 0; }
       ROTATE = 0;
       \GDS2$!@#$A = A;
       \GDS2$!@#$B = B;
    }
}
```

9.17.5 Geometric model

This section defines the geometric model statement and how to predefine commonly used objects (using `TEMPLATE`).

A geometric model describes the form of an object in a physical library. It is in the context of a pattern, which is associated with physical objects, such as via, blockage, port, rule. Patterns and other physical objects can also be subjected to geometric transformations, as shown in Figure 16.

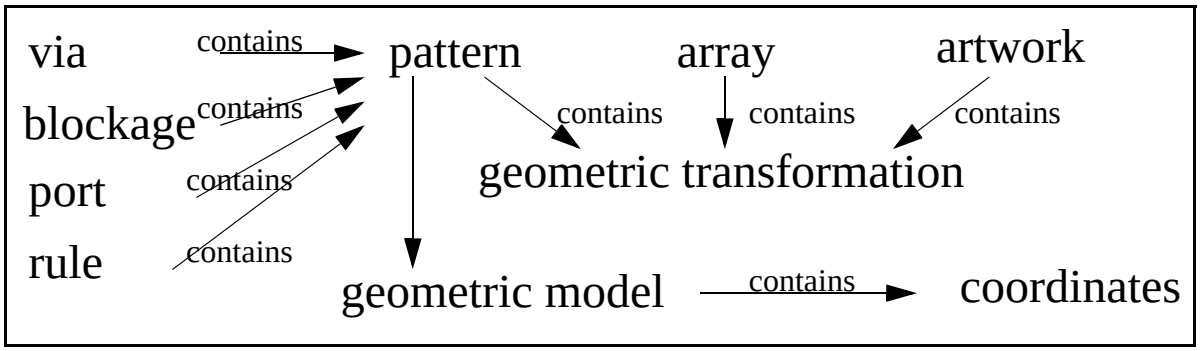


Figure 16—Geometric model and its context

9.17.5.1 Definition

A geometric model is defined as shown in Syntax 107.

```
geometric_model ::=
      nonescaped_dentifier [ geometric_model_identifier ]
        { geometric_model_items }
    | geometric_model_template_instantiation
geometric_model_items ::=
    geometric_model_item { geometric_model_item }
geometric_model_item ::=
      all_purpose_item
    | POINT_TO_POINT_one_level_annotation
    | coordinates
coordinates ::=
    COORDINATES { x_number y_number { x_number y_number } }
```

Syntax 107—Geometric model

```
geometric_model ::=
          geometric_model_identifier
            [ geometric_model_name_identifier ] {
                all_purpose_items
                coordinates
          }
    |     geometric_model_template_instantiation

geometric_models ::=
          geometric_model { geometric_model }

          geometric_model_identifier ::=
                DOT
              | POLYLINE
```

| **RING**
| **POLYGON**

coordinates ::=
COORDINATES { *x*_number *y*_number { *x*_number *y*_number } **}**

A point is a pair of *x*_number and *y*_number.

A **DOT** is 1 point.

A **POLYLINE** is defined by N>1 connected points, forming an open object.

A **RING** is defined by N>1 connected points, forming a closed object, i.e., the last point is connected with first point. The object occupies the edges of the enclosed space.

A **POLYGON** is defined by N>1 connected points, forming a closed object, i.e., the last point is connected with first point. The object occupies the entire enclosed space.

All of these are depicted in Figure 17.

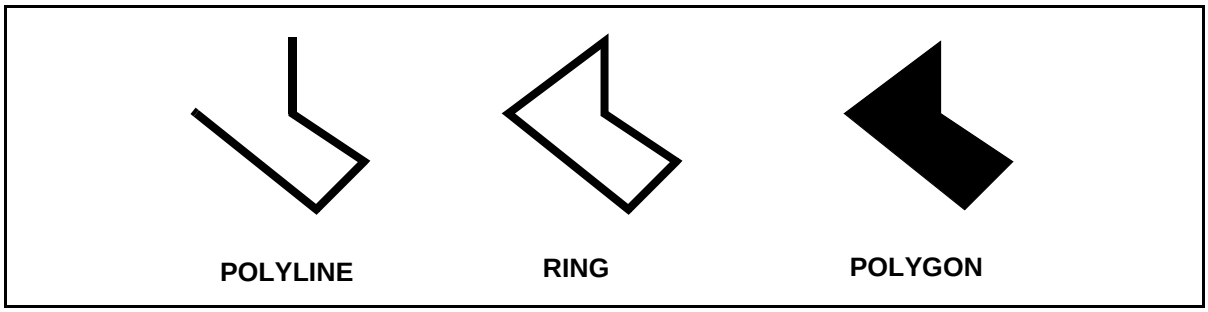


Figure 17—Illustration of geometric models

See 9.17.6.4 for the definition of the **repeat** statement.

The *point_to_point*_annotation applies for **POLYLINE**, **RING**, and **POLYGON**. It specifies how the connections between points is made. The default is straight, which defines a straight connection (see Figure 18). The value rectilinear specifies a connection by moving in the x-direction first and then moving in the y-direction (see Figure 19). This enables a non-redundant specification of rectilinear objects using N/2 points instead of N points.

```
point_to_point_annotation ::=
    POINT_TO_POINT = point_to_point_identifier ;

point_to_point_identifier ::=
      straight
    | rectilinear
```

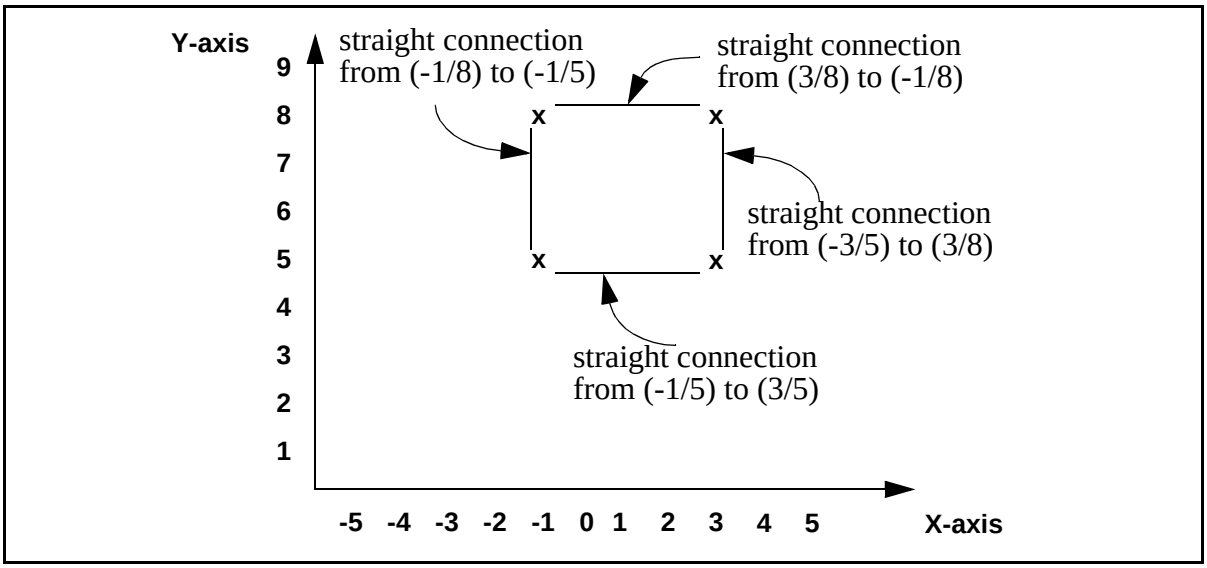


Figure 18—Illustration of straight point-to-point connection

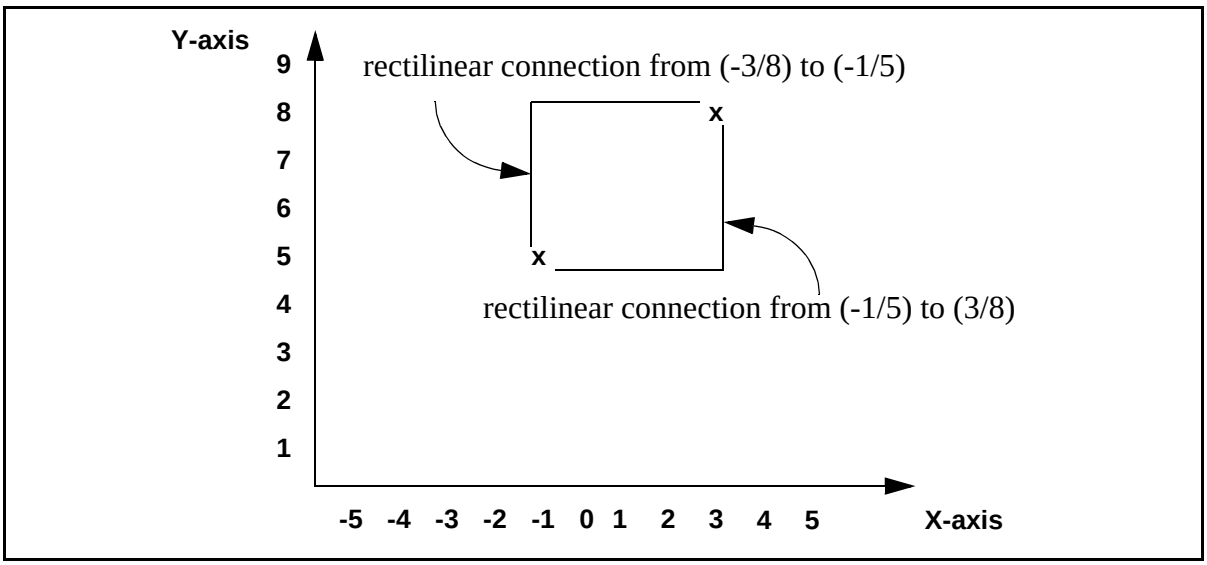


Figure 19—Illustration of rectilinear point-to-point connection

Example

```
POLYGON {
    POINT_TO_POINT = straight;
    COORDINATES { -1 5 3 5 3 8 -1 8 }
}
POLYGON {
    POINT_TO_POINT = rectilinear;
    COORDINATES { -1 5 3 8 }
}
```

Both objects describe the same rectangle.

9.17.5.2 Predefined geometric models using TEMPLATE

The TEMPLATE construct (see 3.2.6) can be used to predefine some commonly used objects.

The templates RECTANGLE and LINE shall be predefined as follows:

```
TEMPLATE RECTANGLE {
    POLYGON {
       POINT_TO_POINT = rectilinear;
       COORDINATES { <left> <bottom> <right> <top> }
    }
}
TEMPLATE LINE {
    POLYLINE {
       POINT_TO_POINT = straight;
       COORDINATES { <x_start> <y_start> <x_end> <y_end> }
    }
}
```

Example 1

The following example shows the instantiation of predefined templates.

```
// same rectangle as in previous example
RECTANGLE {left = -1; bottom = 5; right = 3; top = 8; }
//or
RECTANGLE {-1 5 3 8 }

// diagonals through the rectangle
LINE {x_start = -1; y_start = 5; x_end = 3; y_end = 8; }
LINE {x_start = 3; y_start = 5; x_end = -1; y_end = 8; }
//or
LINE { -1 5 3 8 }
LINE { 3 5 -1 8 }
```

The definitions for predefined templates are fixed. Therefore the keywords RECTANGLE and LINE are reserved. On the other hand, the definitions for user-defined templates are only known by the library supplied by the user.

Example 2

The following example shows some user-defined templates.

```
TEMPLATE HORIZONTAL_LINE {
    POLYLINE {
       POINT_TO_POINT = straight;
       COORDINATES { <left> <y> <right> <y> }
    }
}
TEMPLATE VERTICAL_LINE {
    POLYLINE {
       POINT_TO_POINT = straight;
       COORDINATES { <x> <bottom> <x> <top> }
    }
}
```

Example 3

The following example shows the instantiation of user-defined templates.

```
// lines bounding the rectangle
HORIZONTAL_LINE { y = 5; left = -1; right = 3; }
HORIZONTAL_LINE { y = 8; left = -1; right = 3; }
VERTICAL_LINE { x = -1; bottom = 5; top = 8; }
VERTICAL_LINE { x = 3; bottom = 5; top = 8; }
//or
HORIZONTAL_LINE { 5 -1 3 }
HORIZONTAL_LINE { 8 -1 3 }
VERTICAL_LINE { -1 5 8 }
VERTICAL_LINE { 3 5 8 }
```

9.17.6 Geometric transformation

A geometric transformation XXX, as shown in Syntax 108.

geometric_transformations ::=
    geometric_transformation { geometric_transformation }
geometric_transformation ::=
     *SHIFT*_two_level_annotation
   | *ROTATE*_one_level_annotation
   | *FLIP*_one_level_annotation
   | repeat
repeat ::=
   REPEAT [= unsigned] **{**
      *shift*_two_level_annotation
      [repeat]
   }

Syntax 108—Geometric transformation

~~Statements for geometric transformation~~

This section also defines SHIFT, ROTATE, FLIP, and REPEAT.

9.17.6.1 SHIFT statement

The SHIFT statement defines the horizontal and vertical offset measured between the coordinates of the geometric model and the actual placement of the object. Eventually, a layout tool only supports integer numbers. The numbers are in units of DISTANCE.

```
shift_annotation_container ::=
   SHIFT { horizontal_or_vertical_annotations }

horizontal_or_vertical_annotations ::=
     horizontal_annotation
   | vertical_annotation
   | horizontal_annotation vertical_annotation

horizontal_annotation ::=
   HORIZONTAL = number ;

vertical_annotation ::=
   VERTICAL = number ;
```

If only one annotation is given, the default value for the other one is 0. If the SHIFT statement is not given, both values default to 0.

9.17.6.2 ROTATE statement

The *rotate*_annotation statement defines the angle of rotation in degrees measured between the orientation of the object described by the coordinates of the geometric model and the actual placement of the object measured in counter-clockwise direction, specified by a number between 0 and 360. Eventually, a layout tool can only support angles which are multiple of 90 degrees. The default is 0.

```
rotate_annotation ::=
   ROTATE = number ;
```

The object shall rotate around its origin.

9.17.6.3 FLIP statement

The *flip*_annotation describes a transformation of the specified coordinates by flipping the object around an axis specified by a number between 0 and 180. The number represents the angle of the flipping direction in degrees. Eventually, a layout tool can only support angles which are multiple of 90 degrees. The axis is orthogonal to the flipping direction. The axis shall go through the origin of the object.

```
flip_annotation ::=
   FLIP = number ;
```

Example

FLIP = 0 means flip in horizontal direction, axis is vertical.
FLIP = 90 means flip in vertical direction, axis is horizontal.

9.17.6.4 REPEAT statement

The REPEAT statement shall be defined as shown in Syntax 109.

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

Syntax 109—REPEAT statement

~~repeat ::=~~
~~**REPEAT** [= unsigned] **{**~~
~~*shift*_annotation_container~~
~~[repeat]~~
~~**}**~~

The purpose of the REPEAT statement is to describe the replication of a physical object in a regular way, for example SITE (see Section 9.12). The REPEAT statement can also appear within a geometric_model.

The unsigned number defines the total number of replications. The number 1 means, the object appears just once. If this number is not given, the REPEAT statement defines a rule for an arbitrary number of replications.

REPEAT statements can also be nested.

Examples

The following example replicates an object three times along the horizontal axis in a distance of 7 units.

```
REPEAT = 3 {
    SHIFT { HORIZONTAL = 7; }
}
```

The following example replicates an object five times along a 45-degree axis.

```
REPEAT = 5 {
    SHIFT { HORIZONTAL = 4; VERTICAL = 4; }
}
```

The following example replicates an object two times along the horizontal axis and four times along the vertical axis.

```
REPEAT = 2 {
    SHIFT { HORIZONTAL = 5; }
    REPEAT = 4 {
       SHIFT { VERTICAL = 6; }
    }
}
```

NOTE—The order of nested REPEAT statements does not matter. The following example gives the same result as the previous example.

```
REPEAT = 4 {
    SHIFT { VERTICAL = 6; }
    REPEAT = 2 {
       SHIFT { HORIZONTAL = 5; }
```

```
    }
}
```

9.17.6.5 Summary of geometric transformations

```
geometric_transformations ::=
   geometric_transformation { geometric_transformation }

geometric_transformation ::=
     shift_annotation_container
   | rotate_annotation
   | flip_annotation
   | repeat
```

Rules and restrictions:

- A physical object can contain a `geometric_transformation` statement of any kind, but no more than one of a specific kind.
- The `geometric_transformation` statements shall apply to all `geometric_models` within the context of the object.
- The `geometric_transformation` statements shall refer to the origin of the object, i.e., the point with coordinates `{ 0 0 }`. Therefore, the result of a combined transformation shall be independent of the order in which each individual transformation is applied.

These are demonstrated in Figure 20.

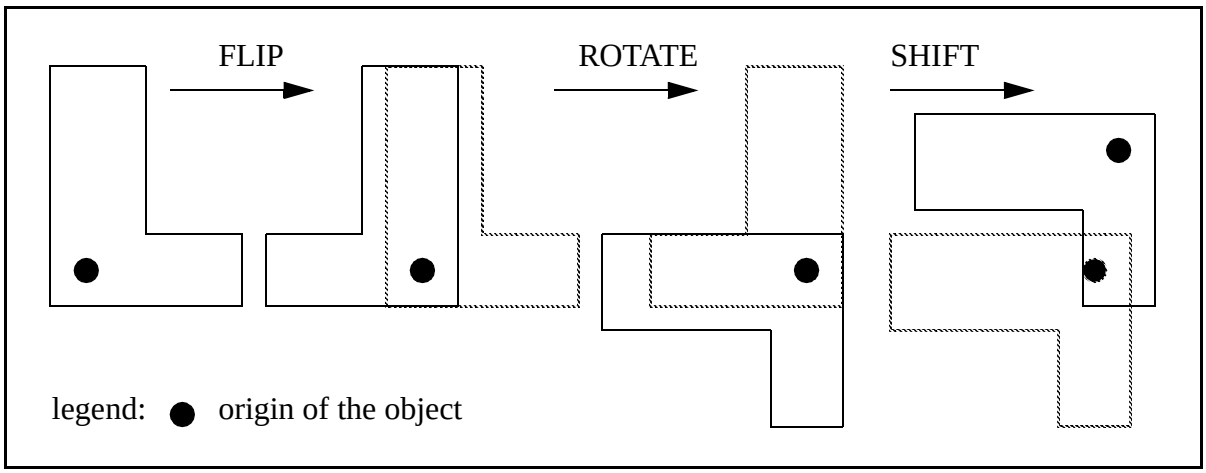


Figure 20—Illustration of FLIP, ROTATE, and SHIFT

9.18 Statements related to functional description

This section specifies the functional modeling for synthesis, formal verification, and simulation.

9.18.1 FUNCTION statement

A `FUNCTION` statement XXX, as shown in Syntax 110.

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

Syntax 110—FUNCTION statement

9.18.2 TEST statement

A CELL can contain a TEST statement, which is defined as shown in Syntax 111.

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

Syntax 111—TEST statement

~~test ::=~~
~~TEST { behavior }~~

The purpose is to describe the interface between an externally applied test algorithm and the CELL. The behavior statement within the TEST statement uses the same syntax as the behavior statement within the FUNCTION statement. However, the set of used variables is different. Both the TEST and the FUNCTION statement shall be self-contained, complete and complementary to each other.

9.18.3 Physical bitmap for memory BIST

This section defines the physical bitmap for memory BIST. This is a particular case of the usage of the TEST statement.

9.18.3.1 Definition of concepts

The physical architecture of a memory can be described by the following parameters (as depicted in Figure 21):

BANK index: A memory can be arranged in one or several banks, each of which constitutes a two-dimensional array of rows and columns
ROW index: A row of memory cells within one bank shares the same row decoder line.
COLUMN index: A column of memory cells within one bank shares the same data bit line and, if applicable, the same sense amplifier.

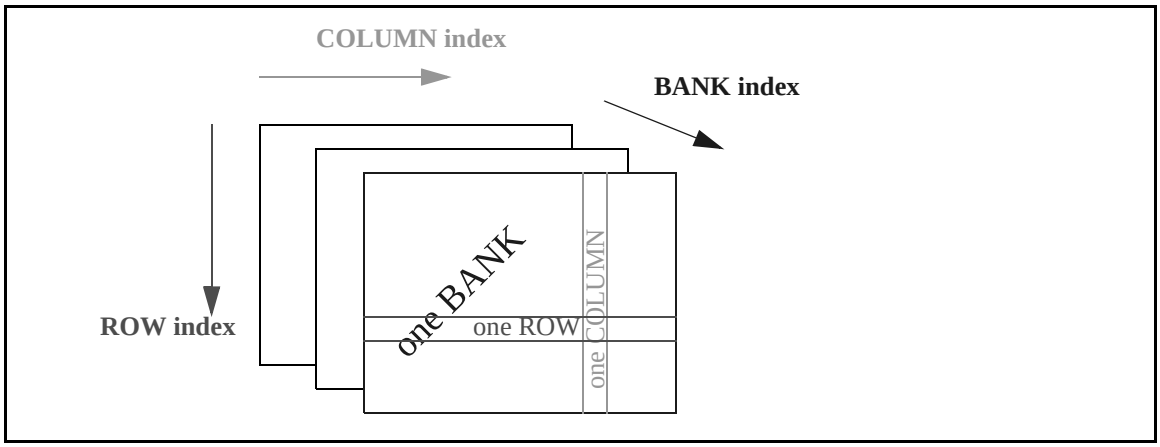


Figure 21—Illustration of a physical memory architecture, arranged in banks, rows, columns

The physical memory architecture is not evident from the functional description and the pins involved in the functional description of the memory. Those pins are called logical pins, e.g., logical address and logical data.

A memory BIST tool needs to know which logical address and data corresponds to a physical row, column, or bank in order to write certain bit patterns into the memory and read expected bit patterns from the memory. Also, the tool needs to know whether the physical data in a specific location is inverted or not with respect to the corresponding logical data (as depicted in Figure 22).

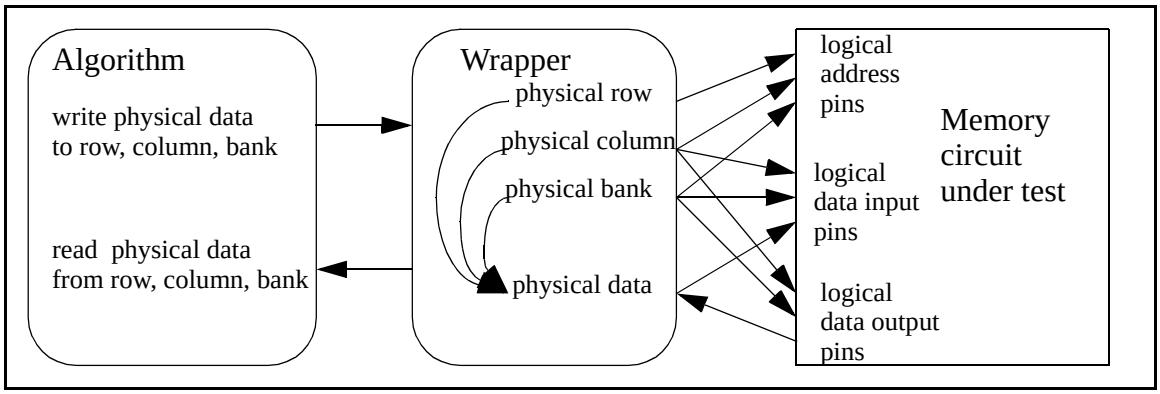


Figure 22—Illustration of the memory BIST concept

A mapper between physical rows, columns, banks, data and logical addresses, and data pins shall be part of the library description of a memory cell.

The physical row, column, and bank indices can be modeled as virtual inputs to the memory circuit. The data to be written to a physical memory location can also be modeled as a virtual input. The data to be read from a physical memory location can be modeled as a virtual output. Since every data that is written for the purpose of test also needs to be read, the data can be modeled as a virtual bidirectional pin. A virtual pin is a pin with `VIEW=none`, i.e., the pin is not visible in any netlist.

9.18.3.2 Explanatory example

One-dimensional arrays with `SIGNALTYPE=address` (here: `PIN[3:0] addr`) shall be recognized as address pins to be mapped, involving other one-dimensional arrays with `ATTRIBUTE { ROW_INDEX }` (here: `PIN[1:0] row`) and `ATTRIBUTE { COLUMN_INDEX }` (here: `PIN[3:0] col`). This memory has only one bank. Therefore, no one-dimensional array with `ATTRIBUTE { BANK_INDEX }` exists here.

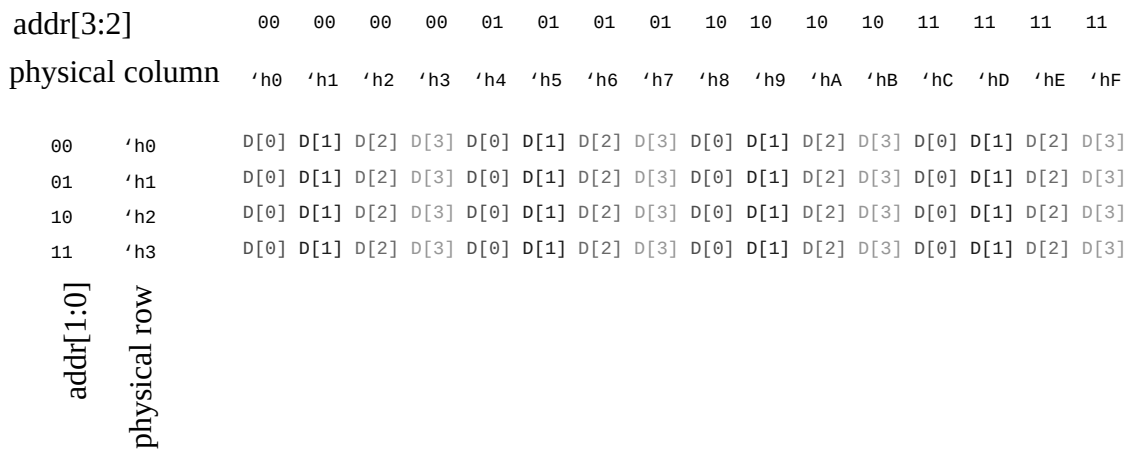
One-dimensional arrays with `SIGNALTYPE=data` (here: `PIN[3:0] Din` and `PIN[3:0] Dout`) shall be recognized as data pins to be mapped, involving other one-dimensional arrays with `ATTRIBUTE { DATA_INDEX }` (here: `PIN[1:0] dat`) and scalar pins with `ATTRIBUTE { DATA_VALUE }` (here: `PIN bit`).

NOTE—Since the data buses are 4-bits wide, the data index is 2-bits wide, since 2=log2(4).

Base Example

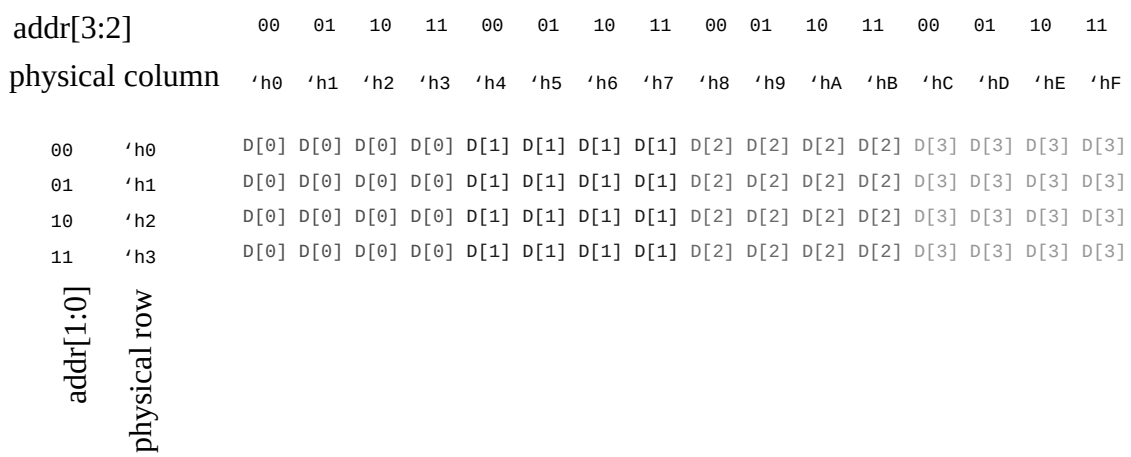
```
CELL my_memory {
    PIN[3:0] addr { DIRECTION=input;  SIGNALTYPE=address; }
    PIN[3:0] Din  { DIRECTION=input;  SIGNALTYPE=data; }
    PIN[3:0] Dout { DIRECTION=output; SIGNALTYPE=data; }
    PIN[3:0] bits[0:15] { DIRECTION=none; VIEW=none; SCOPE=behavior; }
    PIN write_enb { DIRECTION=input; SIGNALTYPE=write_enable;
       POLARITY=high; ACTION=asynchronous;
    }
    PIN[1:0] dat { ATTRIBUTE { DATA_INDEX } DIRECTION=none; VIEW=none; }
    PIN bit { ATTRIBUTE { DATA_VALUE } DIRECTION=both; VIEW=none; }
    PIN[1:0] row {
       ATTRIBUTE { ROW_INDEX }  RANGE { 0: 3 }
       DIRECTION=input; VIEW=none;
    }
    PIN[3:0] col {
       ATTRIBUTE { COLUMN_INDEX } RANGE { 0 : 15 }
       DIRECTION=input; VIEW=none;
    }
    FUNCTION {
       BEHAVIOR {
          Dout = bits[addr];
          @ (write_enb) { bits[addr] = Din; }
    }   }
/*different physical architectures are shown in the following examples*/
}
```

Example 1




```
   TEST {
      BEHAVIOR {
// map row and column index to logical address
         addr[1:0] = row[1:0];
         addr[3:2] = col[3:2];
// map column index to logical data index
         dat[1:0] = col[1:0];
// map physical data to input and output data
         Din[dat] = bit;
         bit = Dout[dat];
      }
   }
```

Example 2




```
   TEST {
      BEHAVIOR {
// map row and column index to logical address
```

```
            addr[1:0] = row[1:0];
            addr[3:2] = col[1:0];
   // map column index to logical data index
            dat[1:0] = col[3:2];
   // map physical data to input and output data
            Din[dat] = bit;
            bit = Dout[dat];
         }
      }
```

Example 3

addr[3:2] physical column

	00	01	11	10	11	10	00	01	00	01	11	10	11	10	00	01
	‘h0	‘h1	‘h2	‘h3	‘h4	‘h5	‘h6	‘h7	‘h8	‘h9	‘hA	‘hB	‘hC	‘hD	‘hE	‘hF
00	‘h0	D[0]	D[0]	D[1]	D[1]	D[0]	D[0]	D[1]	D[1]	!D[2]	!D[2]	!D[3]	!D[3]	D[2]	D[2]	D[3]	D[3]
10	‘h1	D[0]	D[0]	D[1]	D[1]	D[0]	D[0]	D[1]	D[1]	!D[2]	!D[2]	!D[3]	!D[3]	D[2]	D[2]	D[3]	D[3]
11	‘h2	D[0]	D[0]	D[1]	D[1]	!D[0]	!D[0]	!D[1]	!D[1]	D[2]	D[2]	D[3]	D[3]	D[2]	D[2]	D[3]	D[3]
01	‘h3	D[0]	D[0]	D[1]	D[1]	!D[0]	!D[0]	!D[1]	!D[1]	D[2]	D[2]	D[3]	D[3]	D[2]	D[2]	D[3]	D[3]

addr[1:0] physical row

```
      TEST {
         BEHAVIOR {
   // map row and column index to logical address
            addr[0] = row[1];
            addr[1] = row[0] ^ row[1]
            addr[2] = col[0] ^ col[1] ^ col[2];
            addr[3] = col[2] ^ col[3];
   // map column index to logical data index
            dat[0] = col[1];
            dat[1] = col[3];
   // map physical data to input and output data
           Din[dat]=bit^(row[1]&col[2]&!col[3] | !row[1]&!col[2]&col[3]);
           bit=Dout[dat]^(row[1]&col[2]&!col[3] | !row[1]&!col[2]&col[3]);
         }
      }
```

NOTES

1—This enables the description of a complete bitmap of a memory in a compact way.

2—The RANGE feature is not restricted to BIST. It can be used to describe a valid contiguous range on any bus. This alleviates the need for interpreting a VECTOR with ILLEGAL statement to get the valid range. However, the VECTOR with ILLEGAL statement is still necessary to describe the behavior of a device when illegal values are driven on a bus.

3—The TEST statement with BEHAVIOR allows for generalization from memory BIST to any test vector generation requirement, e.g., logic BIST. The only necessary additions would be other PIN ATTRIBUTEs describing particular features to be recognized by the test vector generation algorithm.

9.18.4 BEHAVIOR statement

A BEHAVIOR statement XXX, as shown in Syntax 112.

```
behavior ::=
     BEHAVIOR { behavior_items }
   | behavior_template_instantiation
behavior_items ::=
   behavior_item { behavior_item }
behavior_item ::=
     boolean_assignment
   | control_statement
   | primitive_instantiation
   | behavior_item_template_instantiation
boolean_assignments ::=
   boolean_assignment  { boolean_assignment }
boolean_assignment ::=
   pin_variable = boolean_expression ;
primitive_instantiation ::=
     primitive_identifier [ identifier ] { pin_values }
   | primitive_identifier [ identifier ] { boolean_assignments }
control_statement ::=
   @ control_expression { boolean_assignments } { : control_expression { boolean_assignments } }
```

Syntax 112—BEHAVIOR statement

~~BEHAVIOR~~

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

Examples

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

9.18.5 STRUCTURE statement

An optional STRUCTURE statement shall be legal in the context of a FUNCTION. A STRUCTURE statement describes the structure of a complex cell composed of atomic cells, for example I/O buffers, LSSD flip-flops, or clock trees. The STRUCTURE statement shall be legal inside the FUNCTION statement (see 11.17):

A STRUCTURE statement is defined as shown in Syntax 113.

```
structure ::=
    STRUCTURE { named_cell_instantiations }
  | structure_template_instantiation
named_cell_instantiations ::=
    named_cell_instantiation { named_cell_instantiation }
named_cell_instantiation ::=
    cell_identifier instance_identifier { pin_values }
  | cell_identifier instance_identifier { pin_assignments }
```

Syntax 113—STRUCTURE statement

~~structure ::=~~
~~STRUCTURE { named_cell_instantiations }~~

~~named_cell_instantiations ::=~~
~~named_cell_instantiation { named_cell_instantiation }~~

~~named_cell_instantiation ::=~~
~~*cell*_identifier *instance*_identifier { logic_values }~~
~~*cell*_identifier *instance*_identifier { pin_instantiations }~~

The STRUCTURE statement shall describe a netlist of components inside the CELL. The STRUCURE statement shall not be a substitute for the BEHAVIOR statement. If a FUNCTION contains only a STRUCTURE statement and no BEHAVIOR statement, a behavior description for that particular cell shall be meaningless (e.g., fillcells, diodes, vias, or analog cells).

Timing and power models shall be provided for the CELL, if such models are meaningful. Application tools are not expected to use function, timing, or power models from the instantiated components as a substitute of a missing function, timing, or power model at the top-level. However, tools performing characterization, construction, or verification of a top-level model shall use the models of the instantiated components for this purpose.

Test synthesis applications can use the structural information in order to define a one-to-many mapping for scan cell replacement, such as where a single flip-flop is replaced by a pair of master/slave latches. A macro cell can be defined whose structure is a netlist containing the master and slave latch and this shall contain the NON_SCAN_CELL annotation to define which sequential cells it is replacing. No timing model is required for this macro cell, since it should be treated as a transparent hierarchy level in the design netlist after test synthesis.

NOTES

1—Every *instance*_identifier within a STRUCTURE statement shall be different from each other.

2—The STRUCTURE statement provides a directive to the application (e.g., synthesis and DFT) as to how the CELL is implemented. A CELL referenced in named_cell_instantiation can be replaced by another CELL within the same SWAP_CLASS and RESTRICT_CLASS (recognized by the application).

3—The *cell*_identifier within a STRUCTURE statement can refer to actual cells as well as to primitives. The usage of primitives is recommended in fault modeling for DFT.

4—BEHAVIOR statements also provide the possibility of instantiating primitives. However, those instantiations are for modeling purposes only; they do not necessarily match a physical structure. The STRUCTURE statement always matches a physical structure.

Example 1

iobuffer = pre buffer + main buffer

```
CELL my_main_driver {
    DRIVERTYPE = slotdriver ;
    BUFFERTYPE = output ;
    PIN i { DIRECTION = input; }
    PIN o { DIRECTION = output; }
    FUNCTION { BEHAVIOR { o = i ; } }
}
CELL my_pre_driver {
    DRIVERTYPE = predriver ;
    BUFFERTYPE = output ;
    PIN i { DIRECTION = input; }
    PIN o { DIRECTION = output; }
    FUNCTION { BEHAVIOR { o = i ; } }
}
CELL my_buffer {
    DRIVERTYPE = both ;
    BUFFERTYPE = output ;
    PIN A { DIRECTION = input; }
    PIN Z { DIRECTION = output; }
    PIN Y { VIEW = physical; }
    FUNCTION {
       BEHAVIOR { Z = A ; }
       STRUCTURE {
          my_pre_driver pre { A Y }// pin by order
          my_main_driver main { i=Y; o=Z; }// pin by name
       }
    }
}
```

Example 2

lssd flip-flop = latch + flip-flop + mux

```
CELL my_latch {
    RESTRICT_CLASS { synthesis scan }
    PIN enable { DIRECTION = input;  }
    PIN d      { DIRECTION = input;  }
    PIN d      { DIRECTION = output; }
    FUNCTION { BEHAVIOR {
       @ ( enable ) { q = d ; }
    }   }
}
CELL my_flip-flop {
    RESTRICT_CLASS { synthesis scan }
    PIN clock  { DIRECTION = input;  }
    PIN d      { DIRECTION = input;  }
    PIN q      { DIRECTION = output; }
    FUNCTION { BEHAVIOR {
       @ ( 01 clock ) { q = d ; }
    }   }
}
```

```
CELL my_mux {
    RESTRICT_CLASS { synthesis scan }
    PIN dout   { DIRECTION = output; }
    PIN din0   { DIRECTION = input;  }
    PIN din1   { DIRECTION = input;  }
    PIN select { DIRECTION = input;  }
    FUNCTION { BEHAVIOR {
       dout = select ? din1 : din0 ;
    }   }
}
CELL my_lssd_flip-flop {
    RESTRICT_CLASS { scan }
    CELLTYPE = block;
    SCAN_TYPE = lssd;
    PIN clock         { DIRECTION = input;  }
    PIN master_clock  { DIRECTION = input;  }
    PIN slave_clock   { DIRECTION = input;  }
    PIN scan_data     { DIRECTION = input;  }
    PIN din           { DIRECTION = input;  }
    PIN dout          { DIRECTION = output; }
    PIN scan_master   { VIEW = physical; }
    PIN scan_slave    { VIEW = physical; }
    PIN d_internal    { VIEW = physical; }
    FUNCTION { BEHAVIOR {
       @ ( master_clock ) {
          scan_data_master = scan_data ;
       }
       @ ( slave_clock & ! clock ) {
          dout = scan_data_master ;
       } : ( 01 clock ) {
          dout = din ;
    }   }
          STRUCTURE {
             my_latch U0 {
                enable = master_clock;
                din    = scan_data;
                dout   = scan_data_master;
             }
             my_flip-flop U1 {
                clock  = clock;
                d      = din;
                q      = d_internal;
             }
             my_mux U2 {
                select = slave_clock;
                din1   = scan_data_master;
                din0   = dout;
                dout   = scan_data_slave;
             }
             my_mux U3 {
                select = clock;
                din1   = d_internal;
                din0   = scan_data_slave;
                dout   = dout;
```

```
         }   }
      }
      NON_SCAN_CELL {
         my_flip_flop {
            clock = clock;
            d     = din;
            q     = dout;
            'b0   = slave_clock;
         }
      }
   }
```

Example 3

clock tree = chains of clock buffers

```
   CELL my_root_buffer {
       RESTRICT_CLASS { clock }
       PIN i0 { DIRECTION = input; }
       PIN o0 { DIRECTION = output; }
       FUNCTION { BEHAVIOR { o0 = i0 ; } }
   }
   CELL my_level1_buffer {
       RESTRICT_CLASS { clock }
       PIN i1 { DIRECTION = input; }
       PIN o1 { DIRECTION = output; }
       FUNCTION { BEHAVIOR { o1 = i1 ; } }
   }
   CELL my_level2_buffer {
       RESTRICT_CLASS { clock }
       PIN i2 { DIRECTION = input; }
       PIN o2 { DIRECTION = output; }
       FUNCTION { BEHAVIOR { o2 = i2 ; } }
   }
   CELL my_level3_buffer {
       RESTRICT_CLASS { clock }
       PIN i3 { DIRECTION = input; }
       PIN o3 { DIRECTION = output; }
       FUNCTION { BEHAVIOR { o3 = i3 ; } }
   }
   CELL my_tree_from_level2 {
       RESTRICT_CLASS { clock }
       PIN in  { DIRECTION = input; }
       PIN out { DIRECTION = output; }
       PIN[1:2] level3  { DIRECTION = output; }
       FUNCTION {
          BEHAVIOR { out = in ; }
          STRUCTURE {
             my_level2_buffer U1 { i2=in; o2=out; }
             my_level3_buffer U2 { i3=out; o3=level3[1]; }
             my_level3_buffer U3 { i3=out; o3=level3[2]; }
          }
       }
   }
```

```
CELL my_tree_from_level1 {
    RESTRICT_CLASS { clock }
    PIN in  { DIRECTION = input; }
    PIN out { DIRECTION = output; }
    PIN[1:4] level2  { DIRECTION = output; }
    FUNCTION {
       BEHAVIOR { out = in ; }
       STRUCTURE {
          my_level1_buffer U1 { i1=in; o1=out; }
          my_tree_from_level2 U2 { i2=out; o2=level2[1]; }
          my_tree_from_level2 U3 { i2=out; o2=level2[2]; }
          my_tree_from_level2 U4 { i2=out; o2=level2[3]; }
          my_tree_from_level2 U5 { i2=out; o2=level2[4]; }
       }
    }
}
CELL my_tree_from_root {
    RESTRICT_CLASS { clock }
    PIN in  { DIRECTION = input; }
    PIN out { DIRECTION = output; }
    PIN[1:4] level1  { DIRECTION = output; }
    FUNCTION {
       BEHAVIOR { out = in ; }
       STRUCTURE {
          my_root_buffer U1 { i0=in; o0=out; }
          my_tree_from_level1 U2 { i1=o; o1=level1[1]; }
          my_tree_from_level1 U3 { i1=o; o1=level1[2]; }
          my_tree_from_level1 U4 { i1=o; o1=level1[3]; }
          my_tree_from_level1 U5 { i1=o; o1=level1[4]; }
       }
    }
}
```

Example 4

Multiplexor, showing the conceptional difference between BEHAVIOR and STRUCTURE.

```
CELL my_multiplexor {
    PIN a { DIRECTION = input; }
    PIN b { DIRECTION = input; }
    PIN s { DIRECTION = input; }
    PIN y { DIRECTION = output; }
    FUNCTION {
       BEHAVIOR {
// s_a and s_b are virtual internal nodes
          ALF_AND { out = s_a; in[0] = !s; in[1] = a; }
          ALF_AND { out = s_b; in[0] =  s; in[1] = b; }
          ALF_OR  { out = y; in[0] = s_a; in[1] = s_b; }
       }
       STRUCTURE {
// sbar, sel_a, sel_b are physical internal nodes
          ALF_NOT  { out = sbar; in = s; }
          ALF_NAND { out = sel_a; in[0] = sbar; in[1] = a; }
          ALF_NAND { out = sel_b; in[0] =  s;   in[1] = b; }
```

```
            ALF_NAND { out = y; in[0] = sel_a; in[1] = sel_b; }
        }
    }
}
```

9.18.6 VIOLATION statement

A VIOLATION statement XXX, as shown in Syntax 114.

```
violation ::=
    VIOLATION { violation_items }
  | violation_template_instantiation
violation_items ::=
    violation_item { violation_item }
violation_item ::=
     MESSAGE_TYPE_single_value_annotation
   | MESSAGE_single_value_annotation
   | behavior
```

Syntax 114—VIOLATION statement

~~VIOLATION container~~

A VIOLATION statement can appear within an ILLEGAL statement (see 6.7) and also within a *TIMING_CONSTRAINT* or a *SAME_PIN_TIMING_CONSTRAINT*. The VIOLATION statement can contain the BEHAVIOR object (see 11.17), since the behavior in case of timing constraint violation cannot be described in the FUNCTION. The VIOLATION statement can also contain the annotations shown in Table 50.

Table 50—Annotations within VIOLATION

Keyword	Value type	Description
MESSAGE_TYPE	string	Specifies the type of the message. It can be one of information, warning, or error.
MESSAGE	string	Specifies the message itself.

Example

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

9.18.7 STATETABLE statement

A STATETABLE statement XXX, as shown in Syntax 115.

```
statetable ::=
    STATETABLE [ identifier ]
      { statetable_header statetable_row { statetable_row } }
  | statetable_template_instantiation
statetable_header ::=
    input_pin_variables : output_pin_variables ;
statetable_row ::=
    statetable_control_values : statetable_data_values ;
statetable_control_values ::=
    statetable_control_value { statetable_control_value }
statetable_control_value ::=
     bit_literal
  | based_literal
  | unsigned
  | edge_value
statetable_data_values ::=
    statetable_data_value { statetable_data_value }
statetable_data_value ::=
     bit_literal
  | based_literal
  | unsigned
  | ( [ ! ] pin_variable )
  | ( [ ~ ] pin_variable )
```

Syntax 115—STATETABLE statement

~~STATETABLE~~

9.18.7.1 Definition

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

Example

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

```
            1  1   1?  ?   0  : 0 ;
            1  1   ?0  ?   1  : 1 ;
            1  1   1?  ?   0  : 0 ;
            1  1   ?0  ?   1  : 1 ;
            1  1   01  ?   ?  :(d);
        }
    }
}
```

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

Example

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

A generic ALF parser shall 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 of 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.

9.18.7.2 ROM initialization

The STATETABLE statement can be used to describe the contents of a ROM, as far as this content is fixed in the library.

Example

```
CELL my_rom {
    CELLTYPE = memory;
    ATTRIBUTE { rom asynchronous }
    PIN[1:2] addr { DIRECTION = input; SIGNALTYPE = address; }
    PIN[3:0] dout { DIRECTION = output; SIGNALTYPE = data; }
    PIN[3:0] mem[1:4] { DIRECTION=none; VIEW=none; SIGNALTYPE=data; }
    FUNCTION {
       BEHAVIOR { dout = mem[addr]; }
       STATETABLE {
          addr : mem ;
```

```
            'h0  : 'h5 ;
            'h1  : 'hA ;
            'h2  : 'h5 ;
            'h3  : 'hA ;
        }
    }
}
```

For flexibility, a separate included file can be used:

```
CELL my_rom {
    CELLTYPE = memory;
    ATTRIBUTE { rom asynchronous }
    PIN[1:2] addr { DIRECTION = input; SIGNALTYPE = address; }
    PIN[3:0] dout { DIRECTION = output; SIGNALTYPE = data; }
    PIN[3:0] mem[1:4] { DIRECTION=none; VIEW=none; SIGNALTYPE=data; }
    FUNCTION {
       BEHAVIOR { dout = mem[addr]; }
       INCLUDE "rom_initialization_file.alf" ;
       }
    }
}
```

The contents of the included file `rom_initialization_file.alf` are:

```
STATETABLE {
    addr : mem ;
    'h0  : 'h5 ;
    'h1  : 'hA ;
    'h2  : 'h5 ;
    'h3  : 'hA ;
}
```

9.18.8 PRIMITIVE statement

A PRIMITIVE statement XXX, as shown in Syntax 116.

primitive ::=
    PRIMITIVE *primitive*_identifier **{** primitive_items **}**
  | **PRIMITIVE** *primitive*_identifier **;**
  | *primitive*_template_instantiation
primitive_items ::=
  primitive_item { primitive_item }
primitive_item ::=
    all_purpose_item
  | pin
  | pin_group
  | function
  | test

Syntax 116—PRIMITIVE statement

~~Predefined models~~

~~This section defines the use of predefined models in ALF.~~

9.18.8.1 Usage of PRIMITIVEs

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

9.18.8.2 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 using a 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:

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

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

```
    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 `PIN`s of predefined primitives, the following annotations are defined by default:

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

For predefined extensible primitives, a placeholder can be directly in the `PRIMITIVE` definition:

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

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

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

```
    max_index = <max_index>;
}
```

9.18.8.3 Predefined combinational primitives

This section defines the use of predefined combinational primitives.

9.18.8.3.1 One input, multiple output primitives

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

`ALF_BUF` and `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., `out` refers to `out[0]`.

Example — Primitive model of ALF_BUF

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

Example — Primitive model of ALF_NOT

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

9.18.8.3.2 One output, multiple input primitives

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

ALF_AND, ALF_NAND, ALF_OR, ALF_NOR, ALF_XOR, and 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].

Example — Primitive model of ALF_AND

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

Example — Primitive model of ALF_NAND

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

Example — Primitive model of ALF_OR

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

```
    }
}
```

Example — Primitive model of ALF_NOR

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

Example — Primitive model of ALF_XOR

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

Example — Primitive model of ALF_XNOR

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

9.18.8.4 Predefined tristate primitives

There are four tristate primitives:

ALF_BUFIF1, ALF_BUFIF0, ALF_NOTIF1, and ALF_NOTIF0

Example — Primitive model of ALF_BUFIF1

```
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);
       }
    }
}
```

Example — Primitive model of ALF_BUFIF0

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

```
    }
}
```

Example — Primitive model of ALF_NOTIF1

```
PRIMITIVE ALF_NOTIF1 {
    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);
       }
    }
}
```

Example — Primitive model of ALF_NOTIF0

```
PRIMITIVE ALF_NOTIF0 {
    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;
           1     ?  : Z;
           0     ?  : (!in);
       }
```

```
        }
    }
```

9.18.8.5 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.

Example — Primitive model of ALF_MUX

```
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;
       }
    }
}
```

9.18.8.6 Predefined flip-flop

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

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

Example — Primitive model of ALF_FLIPFLOP

```
PRIMITIVE ALF_FLIPFLOP {
    PIN Q     {
       DIRECTION  = output;
       SIGNALTYPE = data;
       POLARITY   = non_inverted;
    }
```

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

9.18.8.7 Predefined latch

The dual-rail D-latch with set and clear pins has the same functionality as the flip-flop, except the level-sensitive clock (ENABLE pin) is used instead of the edge-sensitive clock.

Example — Primitive model of ALF_LATCH

```
   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) { // level-sensitive behavior
            Q  = D;
            QN = !D;
         }
      }
      STATETABLE {
         D  ENABLE CLEAR SET QX  QNX :  Q    QN ;
         ?  ?      1     1   ?   ?   : (QX) (QNX);
         ?  ?      0     1   ?   ?   :  1    0 ;
         ?  ?      1     0   ?   ?   :  0    1 ;
         ?  0      0     0   ?   ?   : (Q)  (QN) ;
         ?  1      0     0   ?   ?   : (D)  (!D) ;
      }
   }
}
```

10. Constructs for modeling of digital behavior

Add lead-in text

10.1 Variable declarations

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` shall be declared as a `PIN` with `DIRECTION=input` or `both` (since `DIRECTION=both` is a bidirectional pin). However, it is not required that all declared pins are used in the function. Output variables inside a `FUNCTION` need not be declared pins, since they are implicitly declared when they appear at the left-hand side (LHS) of an assignment.

Example

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

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

10.2 Boolean value system

this paragraph needs to move into another section

A *bit* literal shall represent a single bit constant, as shown in Table 51.

Table 51—Single bit constants

Literal	Description
`0`	Value is logic zero.
`1`	Value is logic one.
`X` or `x`	Value is unknown.
`L` or `l`	Value is logic zero with weak drive strength.
`H` or `h`	Value is logic one with weak drive strength.
`W` or `w`	Value is unknown with weak drive strength.
`Z` or `z`	Value is high-impedance.
`U` or `u`	Value is uninitialized.

Table 51—Single bit constants (Continued)

Literal	Description
?	Value is any of the above, yet stable.
*	Value can randomly change.

The following symbols within an octal based literal shall represent numerical values, which can be mapped into equivalent symbols within a binary based literal, as shown in .

Table 52—Mapping between octal base and binary base

Octal	Binary (bit literal)	Numerical value
0	**000**	0
1	**001**	1
2	**010**	2
3	**011**	3
4	**100**	4
5	**101**	5
6	**110**	6
7	**111**	7

The following symbols within a hexadecimal based literal shall represent numerical values, which can be mapped into equivalent symbols within an octal based literal and a binary based literal, as shown in .

Table 53—Mapping between hexadecimal base, octal base, and binary base

Hexadecimal	Octal	Binary (bit literal)	Numerical value
0	**00**	**0000**	0
1	**01**	**0001**	1
2	**02**	**0010**	2
3	**03**	**0011**	3
4	**04**	**0100**	4
5	**05**	**0101**	5
6	**06**	**0110**	6
7	**07**	**0111**	7
8	**10**	**1000**	8
9	**11**	**1001**	9

Table 53—Mapping between hexadecimal base, octal base, and binary base (Continued)

Hexadecimal	Octal	Binary (bit literal)	Numerical value
a or **A**	**12**	**1010**	10
b or **B**	**13**	**1011**	11
c or **C**	**14**	**1100**	12
d or **D**	**15**	**1101**	13
e or **E**	**16**	**1110**	14
f or **F**	**17**	**1111**	15

Based literals involving symbolic bit literals shall not be used to represent numerical values. They shall be mapped from one base into another base according to the following rules:

a) A symbolic bit literal in a hexadecimal based literal shall be mapped into two subsequent occurences of the same symbolic bit literal in an octal based literal.
b) A symbolic bit literal in an octal based literal shall be mapped into three subsequent occurences of the same symbolic bit literal in a binary based literal.
c) A symbolic bit literal in an hexadecimal based literal shall be mapped into four subsequent occurences of the same symbolic bit literal in a binary based literal.

Example

`'o2xw0u` is equivalent to `'b010_xxx_www_000_uuu`
`'hLux` is equivalent to `'bLLLL_uuuu_xxxx`

10.3 Combinational functions

This section defines the different types of combinational functions in ALF.

10.3.1 Combinational logic

Combinational logic can be described by continuous assignments of boolean values (*True* or *False*) to output variables as a function of boolean values of input variables. Such functions can be expressed in either boolean expression format or statetable format.

Let us consider an arbitrary continuous assignment

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

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 *ai*. No storage of previous states is needed for dynamic simulation of combinational logic.

10.3.2 Boolean operators on scalars

Table 54, Table 55, and Table 56 list unary, binary, and ternary boolean operators on scalars.

Table 54—Unary boolean operators

Operator	Description
!, **~**	Logical inversion.

Table 55—Binary boolean operators

Operator	Description
&&, **&**	Logical AND.
||,		Logical OR.
~^	Logic equivalence (XNOR).
^	Logic anti valence (XOR).

Table 56—Ternary operator

Operator	Description
?	Boolean condition operator for construction of combinational if-then-else clause.
:	Boolean else operator for construction of combinational if-then-else clause.

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.

10.3.3 Boolean operators on words

Table 57 and Table 58 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 57—Unary reduction operators

Operator	Description
&	AND all bits.

Table 57—Unary reduction operators (Continued)

Operator	Description
~&	NAND all bits.
|	OR all bits.
~|	NOR all bits.
^	XOR all bits.
~^	XNOR all bits.

Table 58—Binary reduction operators

Operator	Description
==	Equality for case comparison.
!=	Non-equality for case comparison.
>	Greater.
<	Smaller.
>=	Greater or equal.
<=	Smaller or equal.

Table 59 and Table 60 list unary and binary bitwise operators. The result of an expression using these operators shall be an array of bits.

Table 59—Unary bitwise operators

Operator	Description
~	Bitwise inversion.

Table 60—Binary bitwise operators

Operator	Description
&	Bitwise AND.
|	Bitwise OR.
^	Bitwise XOR.
~^	Bitwise XNOR.

The following arithmetic operators, listed in Table 61, are also defined for boolean operations on words. The result of an expression using these operators shall be an extended array of bits.

Table 61—Binary operators

Operator	Description
<<	Shift left.
>>	Shift right.
+	Addition.
-	Subtraction.
*	Multiplication.
/	Division.
%	Modulo division.

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 6-25 for the DATATYPE definitions.

10.3.4 Operator priorities

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

a) unary boolean operator (!, ~, &, ~&, |, ~|, ^, ~^)
b) XNOR (~^), XOR (^), relational (>, <, >=, <=, ==, !=), shift (<<, >>)
c) AND (&, &&), NAND (~&), multiply (*), divide (/), modulus (%)
d) OR (|, ||), NOR (~|), add (+), subtract (-)
e) ternary operators (?, :)

10.3.5 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 three logic values in the following way:

H has the logic value 1
L has the logic value 0
W, 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 Table 62.

Table 62—Case comparison operators

A	B	A==B	A!=B	A>B	A<B
1	1	1	0	0	0
1	H	0	1	X	X
1	0	0	1	1	0
1	L	0	1	1	0
1	W, U, Z, X	0	1	X	0
H	1	0	1	X	X
H	H	1	0	0	0
H	0	0	1	1	0
H	L	0	1	1	0
H	W, U, Z, X	0	1	X	0
0	1	0	1	0	1
0	H	0	1	0	1
0	0	1	0	0	0
0	L	0	1	X	X
0	W, U, Z, X	0	1	0	X
L	1	0	1	0	1
L	H	0	1	0	1
L	0	0	1	X	X
L	L	1	0	0	0
L	W, U, Z, X	0	1	0	X
X	X	1	0	X	X
X	U	X	X	X	X
X	0, 1, H, L, W, Z	0	1	X	X
W	W	1	0	X	X
W	U	X	X	X	X
W	0, 1, H, L, X, Z	0	1	X	X
Z	Z	1	0	X	X
Z	U	X	X	X	X
Z	0, 1, H, L, X, W	0	1	X	X

Table 62—Case comparison operators (Continued)

A	B	A==B	A!=B	A>B	A<B
U	0, 1, H, L, X, W, Z, U	X	X	X	X

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 >= and <= are defined in the following way:

```
(a >= b) === (a > b) || (a == b)
(a <= b) === (a < b) || (a == b)
```

10.3.6 Rules for combinational functions

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

For evaluation of the boolean expression, input value `'bH` shall be treated as `'b1`. Input value `'bL` shall be treated as `'b0`. All other input values shall be treated as `'bX`.

Examples

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

```
BEHAVIOR {
    Z = A;
}
```

is equivalent to

```
BEHAVIOR {
    Z = A ? 'b1 : 'b0;
}
```

More explicitly, this is also equivalent to

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

In table form, this can be expressed as follows:

```
STATETABLE {
    A   :   Z;
    ?   :   (A);
}
```

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;
}
```

10.3.7 Concurrency in combinational 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 in Figure 26.

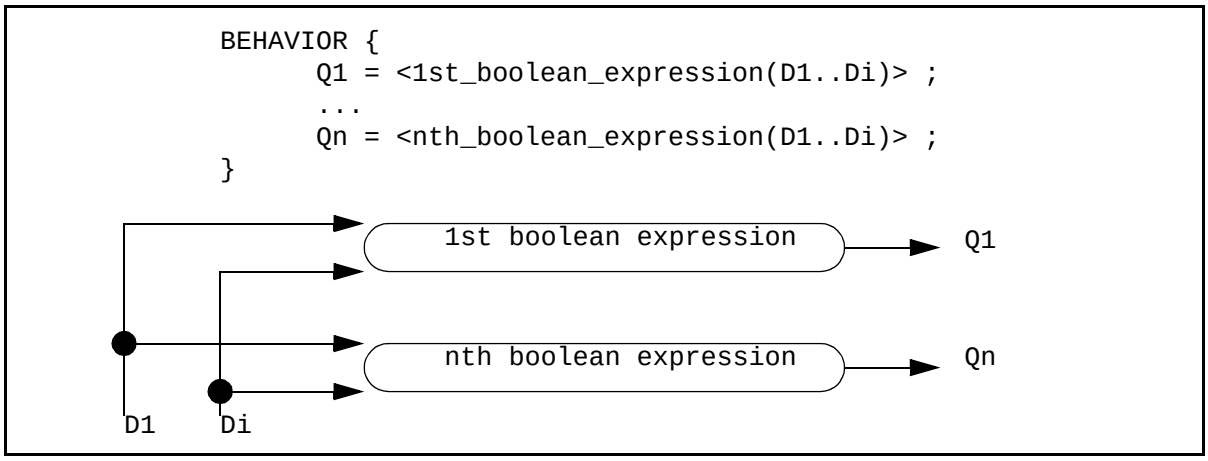


Figure 26—Concurrency for combinational logic

10.4 Sequential functions

This section defines the different types of sequential functions in ALF.

10.4.1 Level-sensitive sequential logic

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

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

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

However, using the concept of *triggering conditions* for the LHS enables everything which is necessary for modeling *level-sensitive* sequential logic. The expression of a triggered assignment can look like this:

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

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

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

10.4.2 Edge-sensitive sequential logic

In order to model *edge-sensitive sequential logic*, notations for logical transitions and logical states are needed.

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 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 flip-flops require at least two storage elements. For instance, the most popular flip-flop 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 flip-flop can be described as follows in ALF:

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

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

If the flip-flop also has an asynchronous direct clear pin (`CD`), the functional description consists of either two concurrent statements or two statements ordered by priority, as shown in Figure 27.

```
// concurrent style
@ (!CD) {Q = 0;}
@ (01 CP && CD) {Q = D;}
// priority (if-then-else) style
@ (!CD) {Q = 0;} : (01 CP) {Q = D;}
```

Figure 27—Model of a flip-flop with asynchronous clear in ALF

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

```
// 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 28—Model of a flip-flop with asynchronous clear in Verilog

```
// full simulation model

process (CP, CD) begin
   if (CD = '0') then
      Q <= '0';
   elsif (CP'last_value = '0' and CP = '1' and CP'event) then
      Q <= D;
   elsif (CP'last_value = '0' and CP = 'X' and CP'event) then
      Q <= 'X';
   elsif (CP'last_value = 'X' and CP = '1' and CP'event) then
      Q <= 'X';
   end if;
end process;

// simplified simulation model for synthesis

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

Figure 29—Model of a flip-flop with asynchronous clear in VHDL

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-sensitiv-

ity shall 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 shall have `negedge` or `posedge` qualifiers.

Furthermore, `posedge` is any transition with `0` as initial state *or* `1` as final state. A positive-edge triggered flip-flop shall be inferred for synthesis, yet this flip-flop shall only work correctly if both the initial state is `0` *and* the final state is `1`. Therefore, a simulation model for verification needs to be more complex than the model in the synthesizeable RTL code.

In Verilog, the extra non-synthesizeable code needs to also reproduce the relevant previous state of the clock signal, whereas VHDL has built-in support for `last_value` of a signal.

10.4.3 Unary operators for vector expressions

A transition operation is defined using unary operators on a scalar net. The scalar constants (see 6.7) shall be used to indicate the start and end states of a transition on a scalar net.

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

For example,

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

No whitespace shall be allowed between the two scalar constants. The transition operators shown in Table 63 shall be considered legal.

Table 63—Unary vector operators on bits

Operator	Description
01	Signal toggles from `0` to `1`.
10	Signal toggles from `1` to `0`.
00	signal remains `0`.
11	Signal remains `1`.
0?	Signal remains `0` or toggles from `0` to arbitrary value.
1?	Signal remains `1` or toggles from `1` to arbitrary value.
?0	Signal remains `0` or toggles from arbitrary value to `0`.
?1	Signal remains `1` or toggles from arbitrary value to `1`.
??	Signal remains constant or toggles between arbitrary values.
0*	A number of arbitrary signal transitions, including possibility of constant value, with the initial value `0`.
1*	A number of arbitrary signal transitions, including possibility of constant value, with the initial value `1`.
?*	A number of arbitrary signal transitions, including possibility of constant value, with arbitrary initial value.

Table 63—Unary vector operators on bits (Continued)

Operator	Description
***0**	A number of arbitrary signal transitions, including possibility of constant value, with the final value `0`.
***1**	A number of arbitrary signal transitions, including possibility of constant value, with the final value `1`.
***?**	A number of arbitrary signal transitions, including possibility of constant value, with arbitrary final value.

Unary operators for transitions can also appear in the STATETABLE.

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

```
'base word '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 64 and Table 65 respectively, are defined on bits and words.

Table 64—Unary vector operators on bits or words

Operator	Description
?-	No transition occurs.
??	Apply arbitrary transition, including possibility of constant value.
?!	Apply arbitrary transition, excluding possibility of constant value.
?~	Apply arbitrary transition with all bits toggling.

10.4.4 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 can 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 shall clearly not trigger the evaluation of a function. There are is also a set of ambiguous vectors, which can 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)
('bW'b0 CP)
('bW'bL CP)
('bZ'b0 CP)
('bZ'bL CP)
('bU'b0 CP)
('bU'bL CP)
```

and the ambiguous vectors are

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

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, and the set of ambiguous vectors is empty.

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

10.4.5 Concurrency in sequential functions

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) can 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 in Figure 30.

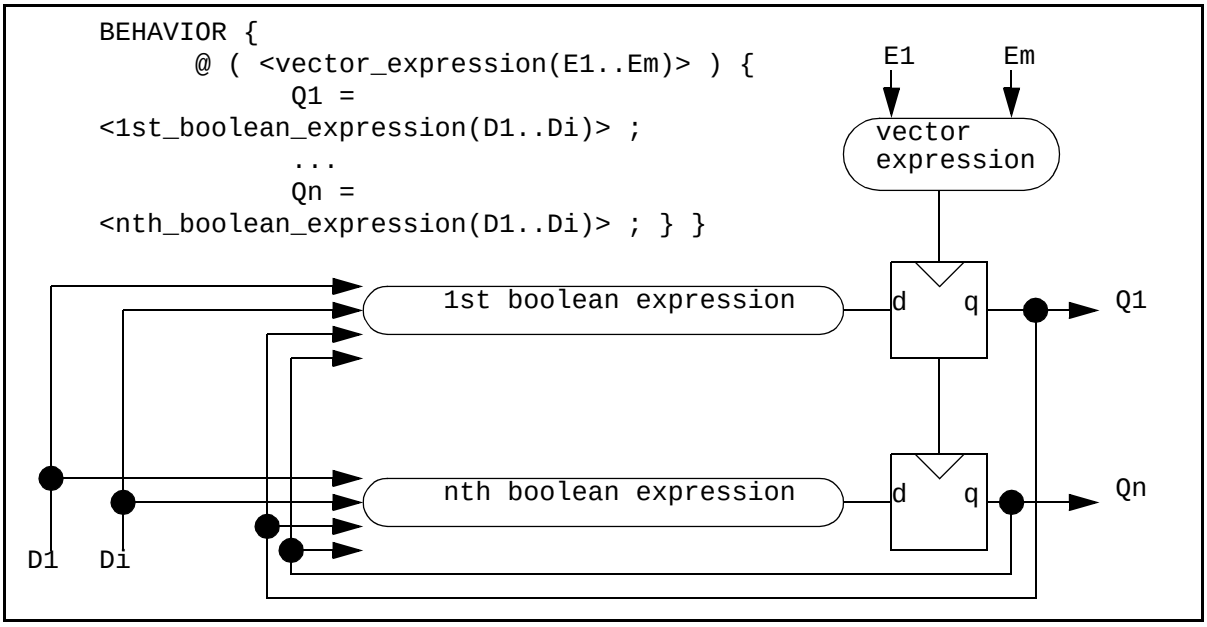


Figure 30—Concurrency for edge-sensitive sequential logic

Statements with multiple concurrent conditions for boolean assignments can also be used in sequential logic. In that case conflicting values can be assigned to the same logic variable. A default conflict resolution is not provided for the following reasons.

- Conflict resolution might 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 might be desirable, while the physical behavior of the circuit shall not change. For instance, pessimistic conflict resolution always assigns X, more accurate conflict resolution first checks whether the values are conflicting. Different choices can 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

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

Explicit pessimistic conflict resolution can be described as follows:

```
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.

10.4.6 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 shall be declared as PINs. It is possible to annotate initial values directly to a pin. Such variables shall never take the value U. Therefore vector expressions involving U for such variables (see the previous example) are meaningless.

Example

```
PIN Q1 { INITIAL_VALUE = 'b1 ; }
PIN Q2 { INITIAL_VALUE = 'b0 ; }
```

10.5 Higher-order sequential functions

This section defines the different types of higher-order sequential functions in ALF.

10.5.1 Vector-sensitive sequential logic

Vector expressions can be used to model generalized higher order sequential logic; they are 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.

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

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

which reads "rising edge on A is followed by rising edge on B".

A vector expression is evaluated by an event sequence detection function. Like a single event or a transition, this function evaluates *True* only at an infinitely short time when the event sequence is detected, as shown in Figure 31.

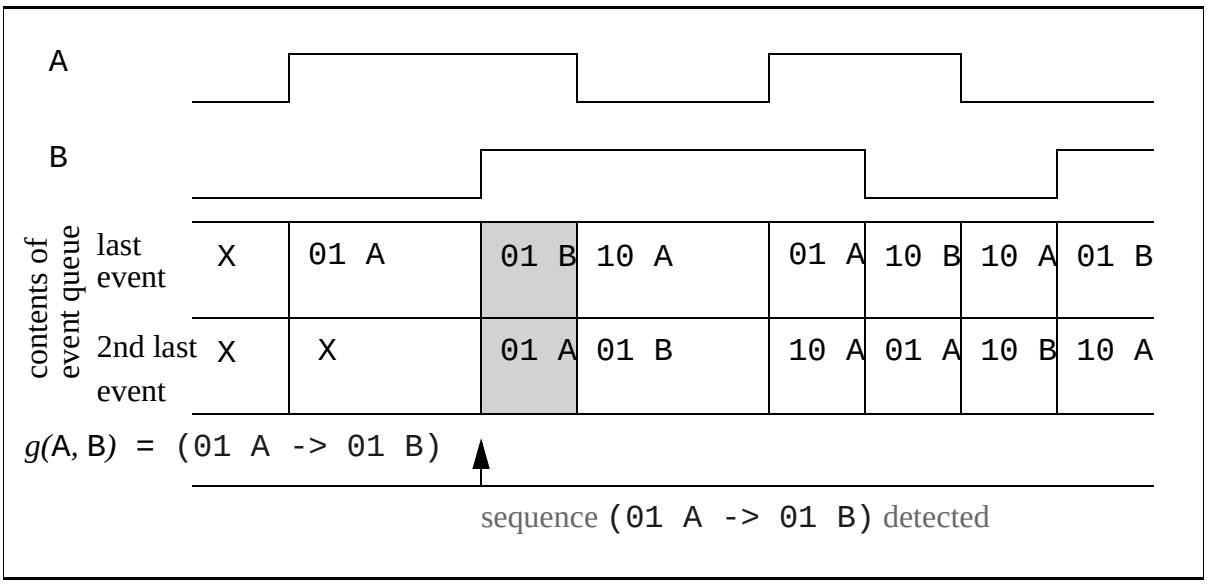


Figure 31—Example of event sequence detection function

The event sequence detection mechanism can be described as a queue that sorts events according to their order of arrival. The event sequence detection function evaluates *True* at exactly the time when a new event enters the queue and forms the required sequence, i.e., *the sequence specified by the vector expression* with its preceding events.

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 one memory element for the output itself.

A sequence of events can also be gated with static logical conditions. In the example,

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

the pin `CD` shall 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 are detailed in 10.5.2 and 10.5.3.

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

10.5.2 Canonical binary operators for vector expressions

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` and `OR` are overloaded for `vector_and` and `vector_or`, respectively. The new symbols for the `vector_followed_by` operators are shown in Table 65.

Table 65—Canonical binary vector operators

Operator	Operands	LHS, RHS commutative	Description
->	2 vector expressions	No	Left-hand side (LHS) transition *is followed by* Right-hand side (RHS) transition, no transition can occur in-between.
&&, &	2 vector expressions	Yes	LHS *and* RHS transition *occur simultaneously.*
||, |	2 vector expressions	Yes	LHS *or* RHS transition *occur alternatively.*
~>	2 vector expressions	No	Left-hand side (LHS) transition *is followed by* Right-hand side (RHS) transition, other transitions can occur in-between.

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

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

The -> and ~> 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 reduces 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 and 10.

10.5.3 Complex binary operators for vector expressions

Table 66 defines the complex binary operators for vector operators.

Table 66—Complex binary vector operators

Operator	Operands	LHS, RHS commutative	Description
<->	2 vector expressions	Yes	LHS transition follows or is followed by RHS transition.
&>	2 vector expressions	No	LHS transition *is followed by or occurs simultaneously* with RHS transition.

Table 66—Complex binary vector operators (Continued)

Operator	Operands	LHS, RHS commutative	Description
<&>	2 vector expressions	Yes	LHS transition *follows or is followed by or occurs simultaneously* with RHS transition.

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 `<->` and `<&>` operators are commutative.

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

The commutative complex binary vector operators are defined in Table 65. The commutativity rules are only defined for two operands:

— commutative “followed by”:

```
vect_expr1 <-> vect_expr2 ===
    vect_expr1 -> vect_expr2 // vect_expr1 occurs first
|   vect_expr2 -> vect_expr1 // vect_expr2 occurs first
```

— commutative “followed by or simultaneously occurring”:

```
vect_expr1 <&> vect_expr2 ===
    vect_expr1 -> vect_expr2 // vect_expr1 occurs first
|   vect_expr2 -> vect_expr1 // vect_expr2 occurs first
|   vect_expr1 && vect_expr2 // both occur simultaneously
```

10.5.4 Extension to N operands

This section defines how to use *N* operands.

A `complex_vector_expression` of the form

```
vector_expression { <-> vector_expression }
```

shall be commutative for all operands. The `complex_vector_expression` describes alternative event sequences in which the temporal order of each constituent `vector_expression` is completely permutable, excluding simultaneous occurrence of each constituent `vector_expression`.

A `complex_vector_expression` of the form

```
vector_expression { <&> vector_expression }
```

shall be commutative for all operands. The `complex_vector_expression` describes alternative event sequences in which the temporal order of each constituent `vector_expression` is completely permutable, including simultaneous occurrence of each constituent `vector_expression`.

Example

```
01 A <-> 01 B <-> 01 C ===
    01 A -> 01 B -> 01 C
|   01 B -> 01 C -> 01 A
|   01 C -> 01 A -> 01 B
|   01 C -> 01 B -> 01 A
|   01 B -> 01 A -> 01 C
|   01 A -> 01 C -> 01 B

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

10.5.4.1 Boolean rules

The following rule applies for a boolean `AND` operation with three operands:

```
rule 1:
A & B & C === (A & B) & C | A & (B & C)
```

A corresponding rule also applies to the commutative followed-by operation with three operands:

```
rule 2:
01 A <-> 01 B <-> 01 C ===
    (01 A <-> 01 B) <-> 01 C
|   01 A <-> (01 B <-> 01 C)
```

The alternative boolean expressions `(A & B) & C` and `A & (B & C)` in `rule 1` are equivalent. Therefore, `rule 1` can be reduced to the following:

```
rule 3:
A & B & C === (A & B) & C === (B & C) & A
```

A corresponding rule does *not* apply to complex vector operands, since each expression with associated operands generates only a subset of permutations:

```
(01 A <-> 01 B) <-> 01 C ===
    (01 A <-> 01 B) -> 01 C)
|   (01 C -> (01 A <-> 01 B)) ===
    01 A -> 01 B -> 01 C
|   01 B -> 01 A -> 01 C
```

```
|   01 C -> 01 A -> 01 B
|   01 C -> 01 B -> 01 A
```

The permutations

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

are missing.

```
01 A <-> (01 B <-> 01 C) ===
    (01 A -> (01 B <-> 01 C))
|   ((01 B <-> 01 C) -> 01 A) ===
    01 A -> 01 B -> 01 C
|   01 A -> 01 C -> 01 B
|   01 B -> 01 C -> 01 A
|   01 C -> 01 B -> 01 A
```

The permutations

```
|   01 B -> 01 A -> 01 C
|   01 C -> 01 A -> 01 B
```

are missing.

10.5.5 Operators for conditional vector expressions

The definitions of the &&, ?, and : operators are also overloaded to describe a *conditional vector expression* (involving boolean expressions and vector expressions), as shown in Table 67. The clauses are boolean expressions; while vector expressions are subject to those clauses.

Table 67—Operators for conditional vector expressions

Operator	Operands	LHS, RHS commutative	Description
&&, &	1 vector expression, 1 boolean expression	Yes	Boolean expression (LHS or RHS) is *True* while sequence of transitions, defined by vector expression (RHS or LHS) occurs.
?	1 vector expression, 1 boolean expression	No	Boolean condition operator for construction of if-then-else clause involving vector expressions.
:	1 vector expression, 1 boolean expression	No	Boolean else operator for construction of if-then-else clause involving vector expressions.

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 `?` and `:` 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 (see 10.6.11).

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

10.5.6 Operators for sequential logic

Table 68 defines the complex binary operators for vector operators.

Table 68—Operators for sequential logic

Operator	Description
@	Sequential `if` operator, followed by a boolean logic expression (for level-sensitive assignment) or by a vector expression (for edge-sensitive assignment).
:	Sequential `else if` operator, followed by a boolean logic expression (for level-sensitive assignment) or by a vector expression (for edge-sensitive assignment) with lower priority.

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.

10.5.7 Operator priorities

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

a) unary vector operators (edge literals)

b) complex binary vector operators (<->, &>, <&>)
c) vector AND (&, &&)
d) vector_followed_by operators (->, ~>)
e) vector OR (|, ||)

10.5.8 Using 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 A -> 01 B)
```

implies 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 A -> 01 B) && (C | D)
```

No transition on A, B occurs between 01 A and 01 B, and (C | D) needs to stay *True* in-between 01 A and 01 B as well. However, C and D can change their values as long as (C | D) is satisfied.

10.6 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, e.g., 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 the means for describing the 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 explained using terminology from simulation event reports. However, the application of vector expressions is not restricted to post-processing event reports.

Some application tools (e.g., power analysis tools) can actually evaluate vector expressions during post-processing of event reports from simulation. Other application tools, especially simulation model generators, need to 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 can 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, to an internal event queue within a simulator, or to a hypothetical event report which is merely a mathematical concept.

10.6.1 Event reports

This section describes the terminology of event reports from simulation, which is used 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 needs to be defined beforehand. Each variable also has a definition for the *set of states* it can take. For instance, there can be binary variables, 16-bit integer variables, 1-bit variables with drive-strength information, etc. Furthermore, the initial state of each variable shall be defined as well. In an ALF context, the terms *signal* and *variable* are used interchangeably. In VHDL, the corresponding term is *signal*. In Verilog, there is no single corresponding term. All `input`, `output`, `wire`, and `reg` variables in Verilog correspond to a `signal` in VHDL.

The time values `<time1>`, `<time2>`, `<time3>`, etc. shall 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 can be smaller than the resolution of the time scale or even zero (`0`). Therefore, a variable can 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 of 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.

An example of an event report in VCD format:

```
// 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
```

An example of an event report in test pattern format:

```
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.

10.6.2 Event sequences

For specification of a functional waveform (e.g., the write cycle of a memory), it is not practical to use an event report format, such as a 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, can vary from one instance to the other.

The main purpose of `vector_expressions` is waveform specification capability. The following operators can be used:

— `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. Edge operators can describe a change of state or 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`:

a) vector_single_event
A change of state in a single variable, for example:
```
01 A
```
b) vector_event
A simultaneous change of state in one or more variables, for example:
```
01 A & 10 B
```
c) 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 (`1`).

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.

10.6.3 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 shall be a subset of variables within scope.

— `PIN`s with the annotation `SCOPE = BEHAVIOR` are applicable variables for vector expressions within the context of `BEHAVIOR`.
— `PIN`s with the annotation `SCOPE = MEASURE` are applicable variables for vector expressions within the context of `VECTOR`.
— `PIN`s 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

```
time  A   B   C   D   E   // scope is 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
```

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)` is *True* at time 109, time 643, and 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)` is *True* at time 573.

```
10 A -> 01 A                                  //(3) content is A
//event pattern expressed by (3):
//   A
//   1
//   0
//   1
```

`(3)` is *True* at time 643 and time 915.

```
01 D                                          //(4) content is D
//event pattern expressed by (4):
//   D
//   0
//   1
```

`(4)` is *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)` is not be *True* at any time, since the event pattern expressed by `(5)` does not match the event report at any time.

10.6.4 Alternative event sequences

The following operator can be used 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                                          //(6)
//event pattern expressed by (6):
//   A
//   0
//   1
//alternative event pattern expressed by (6):
//   C
//   1
//   0
```

`(6)` is *True* at time 109, time 573, time 643, time 915, and 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)` is *True* at time 573, time 643, and 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) is *True* at time 573 and 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) is *True* at time 573, time 586, time 788, and time 1640.

The following operators provide a more compact description of certain alternative event sequences:

— &> events occur simultaneously or follow each other in the order RHS after LHS
— <-> a LHS event followed by a RHS event or a RHS event followed by a 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 ->.

10.6.5 Symbolic edge operators

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

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

Both edge operators include the possibility 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 also can be used:

a) `?-` no transition on the operand
b) `?!` transition from any state to any state different from the previous state
c) `??` transition from any state to any state or no transition on the operand
d) `?~` transition from any state to its bitwise complementary state

Example

Let `A` be a logic variable with the possible states `1`, `0`, and `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 and multiple bits) the explicit description of alternative events is quite verbose. Therefore the symbolic edge operators are useful for a more compact description.

This completes 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.

10.6.6 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 flip-flop 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 where the output switches from `0` to `1` after the rising edge of the clock. The vector expression `(ii)` describes the situation where 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   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
```

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 times 109, 258, 573, 586, 643, 788, and 915; in short, every time when an event happens while `E` is constant `1`.

10.6.7 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 shall 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 needs to 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 make this impossible. Rather, this `vector_event` expression needs to be padded up with unspeci-

fied 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.

10.6.8 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 by 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 of the flip-flop. For instance, in a flip-flop it is impossible for a triggering clock edge 01 CK and a switch of the data output ? Q to occur at the same time. Therefore, such events can not appear in the event report. It makes no difference whether 01 CK & ?- Q, 01 CK & ?? Q, or 01 CK is specified. The only occurring event pattern is 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 flip-flop with a 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 is less accurate anyway. In the analysis of the event report for the design without timing violation, the only occurring event pattern is `01 CK & ?- D` and this pattern can be reliably detected by specifying `01 CK`.[2]

— *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 can 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`.

— *Case 5*: The effects of simultaneous events are modeled accurately. Here it makes sense to specify all scenarios explicitly, e.g., `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 need to be allowed rather than disallowed. Consider the following vector expressions within the scope of two variables `A` and `B`.

```
01 A          // (i)
01 B          // (ii)
01 A & 01 B   // (iii)
```

The natural interpretation here is `(iii)` === `(i)` & `(ii)`. This interpretation is only possible by allowing simultaneously occurring unspecified events.

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

```
01 A & ?? B   // (i')
?? A & 01 B   // (ii')
```

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

```
01 A & ?- B   // (i'')
?- A & 01 B   // (ii'')
```

The vector expressions `(i')` and `(ii')` are compatible with `(iii)`, whereas `(i'')` and `(ii'')` are not.

10.6.9 Simultaneous event sequences

The semantic meaning of the "simultaneous event operator" can be extended to describe simultaneously occurring *event sequences*, by using the following definition:

```
(01 A#1 .. -> ... 01 A#N) & (01 B#1 .. -> ... 01 B#N)
=== 01 A#1 & 01 B#1 ... -> ... 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 `vector_event` expressions separated by `->` operators. If the number of

[2]The power analysis tool relates 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 shall report inaccurate wire capacitance for those nets. After final layout, the DRC violations shall be gone and the wire capacitance shall be accurate.

`->` 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

```
(01 A -> 01 B -> 01 C) & (01 D -> 01 E)
=== (01 A -> 01 B -> 01 C) & (?? D -> 01 D -> 01 E)
=== 01 A & ?? D -> 01 B & 01 D -> 01 C & 01 E
=== 01 A -> 01 B & 01 D -> 01 C & 01 E
```

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.

```
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
```

Example

```
01 A -> 10 B === 01 A & 11 B -> 11 A & 10 B      // (10a)
// event pattern expressed by (10a):
//   A   B
//   0   1
//   1   1
//   1   0
X0 D -> 00 D                                    // (10b)
// event pattern expressed by (10b):
//   D
//   X
//   0
//   0
(01 A -> 10 B) & (X0 D -> 00 D)                 // (10) === (10a)&(10b)
```

Both `(10a)` and `(10b)` are *True* at time 258. Therefore `(10)` is *True* at time 258.

```
10 C
=== ?? C -> ?? C -> 10 C
=== ?? C -> ?1 C -> 10 C                        // (11a)
// event pattern expressed by (11a):
//   C
//   ?
//   ?
//   1
//   0
```

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

```
01 A -> 00 D -> 11 E ===
   01 A & 00 D & ?? E
-> ?? A & 00 D & ?? E
-> ?? A & ?? D & 11 E
===
   01 A & 00 D & ?? E
-> 1? A & 00 D & ?1 E
-> ?? A & 0? D & 11 E                          // (11b)
// event pattern expressed by (11b):
//   A   D   E
//   0   0   ?
//   1   0   ?
//   ?   0   1
//   ?   ?   1
```

(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.

```
10 C & (01 A -> 00 D -> 11 E)                  // (11) === (11a)&(11b)
```

(11a) is *True* at time 573 and time 1395. (11b) is *True* at time 573 and time 915. Therefore, (11) is *True* at time 573.

10.6.10 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) is different. However, for complex cells and megacells, it is sometimes necessary to change the scope of event observation, depending on operation modes. Different modes can 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.

Suppose the local variables (A & B), (A | B) are inserted into the event report:

```
time  A   B   C   D   E   A&B   A|B
0     0   1   1   X   1    0     1
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

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

(12) is *True* at time 109 and time 915.

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

(13) is *True* at time 586 and time 1640.

```
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.

```
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.

```
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.

10.6.11 Conditional event sequences

The following definition *restricts* 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 shall be *True* during the entire vector expression. The boolean expression is called the *Existence Condition* of the vector expression.[3]

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) -> 10 (A | B)) & !D                    // (17)
// event pattern expressed by (17):
//   A&B   A|B
//    1     1
//    0     1
//    0     0
// event report without C, E:
time  A   B   D   A&B   A|B
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
```

`(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`, and `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:

```
(10 (A & B) -> 10 (A | B)) & !D
===
(10 (A & B) -> 10 (A | B)) & (11 (!D) -> 11 (!D))
===
10 (A & B) & 11 (!D) -> 10 (A | B) & 11 (!D)
```

The transformation consists of the following steps:

a) Transform the boolean condition into a non-event.
   For example, `!D` becomes `11 (!D)`.

[3]An Existence Condition can also appear as annotation to a `VECTOR` object instead of appearing in the vector expression. This enables 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 an annotation.

b) 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 `10 (A & B) -> 10 (A | B)`.

c) 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 needs to be kept in mind. An operator which can express the change of scope in the vector expression language is defined in 10.6.13. This can make the transformation more rigorous.

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

Example

```
(10 A -> 01 B -> 01 A) & A
===
10 A & 11 A -> 01 B & 11 A -> 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*.

10.6.12 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:

```
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
```

Concurrent alternative `vector_conditional_event` expressions can be paraphrased in the following way:

```
IF <boolean_expression1> THEN <vector_expression1>
OR IF <boolean_expression2> THEN <vector_expression2>
... OR IF <boolean_expressionN> THEN <vector_expressionN>
```

The conditions can be *True* within overlapping time windows and thus the vector expressions are evaluated concurrently. The `vector_boolean_and` operator and `vector_or` operator describe such vector expressions.

Example

```
C&(01 A -> 10 B) | !D&(10 B -> 10 A) | E&(10 B -> 10 C) // (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 & (01 A -> 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 & (10 B -> 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 & (10 B -> 10 C)`.

Prioritized alternative `vector_conditional_event` expressions can be paraphrased in the following way:

```
IF <boolean_expression1> THEN <vector_expression1>
ELSE IF <boolean_expression2> THEN <vector_expression2>
... ELSE IF <boolean_expressionN> THEN <vector_expressionN>
(optional) ELSE <vector_expressiondefault>
```

Only the vector expression with the highest priority *True* condition is evaluated. The `vector_boolean_cond` operator and `vector_boolean_else` operator are used in ALF to describe such vector expressions.

Example

```
C? (01 A -> 10 B): !D? (10 B -> 10 A): E? (10 B -> 10 C) // (19)
```

The prioritized alternative `vector_conditional_event` expression can be transformed into concurrent alternative `vector_conditional_event` expression as shown:

```
C ? (01 A -> 10 B) : !D ? (10 B -> 10 A) : E ? (10 B -> 10 C)
===
C & (01 A -> 10 B)
| !C & !D & (10 B -> 10 A)
| !C & !(!D) & E & (10 B -> 10 C)
```

`(19)` is *True* at time 258 because of `C & (01 A -> 10 B)`, but not at time 586 because of higher priority `C` while `!D & (10 B -> 10 A)`, nor at time 573 because of higher priority `!D` while `E & (10 B -> 10 C)`.

10.6.13 Change of scope within a vector expression

Conditions on vector expressions redefine the scope of vector expressions locally. The following definition can be used to change the scope even within a part of a vector expression. For this purpose, the symbolic state * can be used, which means “don’t care about events”. This is different from the symbolic state ? which means “don’t care about state”. When the state of a variable is *, arbitrary events occurring on that variable are disregarded.

— Edge operator with * as next state:
The variable to which the operator applies is no longer within the scope of the vector expression.
— Edge operator with * as previous state:
The variable to which the edge operator applies is now within the scope of the vector expression.

As opposed to ?, * stands for an infinite variety of possibilities.

Example

Let A be a logic variable with the possible states 1, 0, and 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. The following parts of the vector expression are separated by * *sub-sequences* of events.

— “Static” event sequence detection with *:
The event report with all variables can be maintained, but certain variables are masked for the purpose of detection of certain sub-sequences.
— “Dynamic” event sequence detection with *:
The event report shall 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 shall change until the next sub-sequence is detected, etc.

Examples

```
01 A -> 1* B -> 10 C                                  // (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
```

The event report with masking relevant for `(20)`:

```
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
```

`(20)` is *True* at time 573 and time 1395. The first sub-sequence `01 A -> 1* B` is detected at time 258, since * maps to any state. From time 258 onwards, `B` is masked. The second sub-sequence `10 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 A & 1* E -> 10 C                              // (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   *
```

The event report with masking relevant for `(21)`:

```
time  A   B   C   D   E
0     0   1   1   X   1
109   1   1   1   0   *   // detection of 1st sub-sequence
258   1   0   1   0   *   // abortion of detection process
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   *   // detection of 1st sub-sequence
1062  1   1   1   0   *   // disregard event out of scope
1395  1   0   0   0   0   // detection of 2nd sub-sequence
1640  0   0   0   1   0
```

(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
```

The event report with masking relevant for `(22)`:

```
time  A   B   C   D   E
0     0   1   1   X   1
109   1   1   1   0   1   // detection of 1st sub-sequence
258   1   0   1   0   1   // abort
573   1   *   0   0   1
586   0   *   0   0   1
643   1   *   0   0   1
788   0   *   1   0   1
915   1   *   1   0   1   // detection of 1st sub-sequence
1062  1   1   1   0   0   // continue
1395  1   0   0   0   0   // detection of 2nd sub-sequence
1640  0   0   0   1   0
```

`(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 -> 1? A & 0* B & 1* E -> 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   *
```

The event report with masking relevant for (23):

```
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   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.

10.6.14 Sequences of conditional event sequences

The symbol `*` can be used 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.

In reusing `(17)` as example:

```
(10 (A & B) -> 10 (A | B)) & !D
```

the scope of the sample event report contains contain the variables `A`, `B`, `C`, `D`, and `E`. The `vector_conditional_event` expression `(17)` contains only the variables `A`, `B`, and `D` and the implicit local variables `A&B` and `A|B`. Therefore, the global variables `C` and `E` are out of scope within `(17)`. The implicit local variables `A&B` and `A|B` are in scope within, and only within, `(17)`.

Now 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.

```
<vector_conditional_event#1> .. -> .. <vector_conditional_event#N>
```

where

```
<vector_conditional_event#i>
=== <vector_complex_event#i> & <boolean_expression#i> // 1 ≤ i ≤ 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.

```
<vector_conditional_event#i>
=== <prefix#i> -> <kernel#i> -> <postfix#i> // 1 ≤ i ≤ N
```

a) Define the prefix for each `vector_conditional_event` expression.
The *prefix* is a `vector_event` expression defining all implicit local variables.

Example

```
*? (A&B) & *? (A|B)
```

b) Define the kernel for each `vector_conditional_event` expression.
The *kernel* is the `vector_complex_event` expression equivalent to the `vector_conditional_event` expression.

```
<vector_complex_event#i> & <boolean_expression#i>
===    <vector_complex_event#i>
&    (11 <boolean_expression#i> ..->.. 11 <boolean_expression#i>)
```

The kernel can 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   // global variables out of scope
& 10 (A & B) & 11 (!D) -> 10 (A | B) & 11 (!D)
& *? C & *? E   // global variables back in scope
```

c) 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)
```

d) Join the subsequent `vector_complex_event` expressions with the `vector_and` operator between prefix#i+1and 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> ..
```

The 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 —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 the 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.

10.6.15 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 can be used 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 a sample event report:

```
time  A   B   C   D   E
0     0   1   1   X   1
109   1   1   1   0   1   // 01 A detected, set flag
258   1   0   1   0   1
573   1   0   0   0   1   // 10 C detected, clear flag
586   0   0   0   0   1
643   1   0   0   0   1   // 01 A detected, set flag
788   0   1   1   0   1
915   1   1   1   0   1   // 01 A detected again
1062  1   1   1   0   0
1395  1   0   0   0   0   // 10 C detected, clear flag
1640  0   0   0   1   0
```

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`.

10.6.16 How to determine 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.

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:
The next state of earlier event shall 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 the previous and next state of both events shall be compatible. Such events commonly occur as intermediate calculation results within vector expression transformation.

The following compatibility rules apply:

a) `?` is compatible with any other state. If the other state is `*`, the resulting state is `?`. Otherwise, the resulting state is the other state.
b) `*` is compatible with any other state. The resulting state is the other state.
c) 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`.

```
0X A ~> 01 B -> 10 A
```

Compatibility check does not apply, since intermediate events are allowed.

```
01 A & 10 A
```

Both the previous and next state of `A` are contradictory; this results in an impossible event.

```
?1 A & 1? A
```

Both previous and next state of `A` are compatible; this results in the non-event `11 A`.

10.7 Boolean expression language

The boolean expression language XXX, as shown in Syntax 84.

10.8 Vector expression language

The vector expression language XXX, as shown in Syntax 85.

```
boolean_expression ::=
      ( boolean_expression )
    | pin_value
    | boolean_unary boolean_expression
    | boolean_expression boolean_binary boolean_expression
    | boolean_expression ? boolean_expression :
        { boolean_expression ? boolean_expression : }
        boolean_expression
boolean_unary ::=
      !
    | ~
    | &
    | ~&
    | |
    | ~|
    | ^
    | ~^
boolean_binary ::=
      &
    | &&
    | |
    | ||
    | ^
    | ~^
    | !=
    | ==
    | >=
    | <=
    | >
    | <
    | +
    | -
    | *
    | /
    | %
    | >>
    | <<
```

Syntax 84—Boolean expression langauge

10.9 Control expression semantics

**Syntax 85 also shows the control expression syntax (at the bottom); is this deliberate??

```
vector_expression ::=
      ( vector_expression )
    | vector_unary boolean_expression
    | vector_expression vector_binary vector_expression
    | boolean_expression ? vector_expression :
        { boolean_expression ? vector_expression : }
        vector_expression
    | boolean_expression control_and vector_expression
    | vector_expression control_and boolean_expression
vector_unary ::=
    edge_literal
vector_binary ::=
      &
    | &&
    | |
    | ||
    | ->
    | ~>
    | <->
    | <~>
    | &>
    | <&>
control_and ::=
    & | &&
control_expression ::=
      ( vector_expression )
    | ( boolean_expression )
```

Syntax 85—Vector expression language

11. Constructs for modeling of analog behavior

Add lead-in text

11.1 Arithmetic expression language

~~Arithmetic expressions~~

Arithmetic expressions define the contents of an `EQUATION`. Variables used in the `EQUATION` are the `identifiers` of the `header_model`, if present, or else the *`model`*`_keywords` of the `header_model`.

11.1.1 Syntax of arithmetic expressions

~~The syntax of arithmetic expressions is:~~

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

An arithmetic expression XXX, as shown in Syntax 86.

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

Syntax 86—Arithemetic expression

Examples

```
1.24
- Vdd
C1 + C2
MAX ( 3.5*C , -Vdd/2 , 0.0 )
(C > 10) ? Vdd**2 : 1/2*Vdd - 0.5*C
```

An arithmetic unary XXX, as shown in Syntax 87.

An arithmetic binary XXX, as shown in Syntax 88.

```
arithmetic_unary ::=
    sign
```

Syntax 87—Arithmetic unary

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

Syntax 88—Arithmetic binary

An arithmetic macro XXX, as shown in Syntax 89.

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

Syntax 89—Arithmetic macro

11.1.2 Arithmetic operators

Table 66, Table 67, and Table 68 list unary, binary, and function arithmetic operators.

Table 66—Unary arithmetic operators

Operator	Description
+	Positive sign (for integer or number)
-	Negative sign (for integer or number)

Table 67—Binary arithmetic operators

Operator	Description
+	Addition (integer or number)
-	Subtraction (integer or number)
*	Multiplication (integer or number)
/	Division (integer or number)
**	Exponentiation (integer or number)
%	Modulo division (integer or number)

Table 68—Function arithmetic operators

Operator	Description
`LOG`	Natural logarithm (argument is + integer or number)
`EXP`	Natural exponential (argument is integer or number)
`ABS`	Absolute value (argument is integer or number)
`MIN`	Minimum (all arguments are integer or number)
`MAX`	Maximum (all arguments are integer or number)

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

11.1.3 Operator priorities

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

a) unary arithmetic operator (`+`, `-`)
b) exponentiation (`**`)
c) multiplication (`*`), division (`/`), modulo division (`%`)
d) addition (`+`), subtraction (`-`)

11.2 Arithmetic model and related statements

Add lead-in text

11.2.1 Arithmetic models

An arithmetic model is an object that describes characterization data or a more abstract, measurable relationship between physical quantities, as shown in Figure 32. The modeling language allows tabulated data as well as linear and non-linear equations. The equations consist of arithmetic expressions based on the symbols defined in *IEEE 1364-1995*.

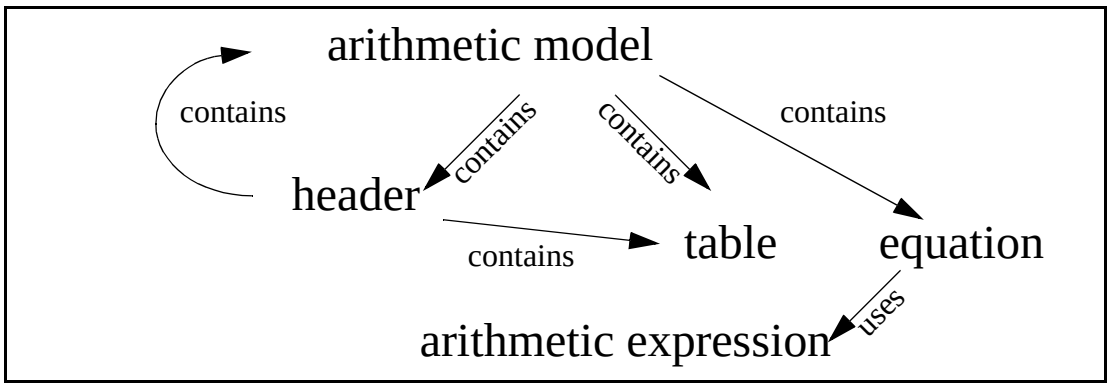


Figure 32—Arithmetic model

~~General Rules for Arithmetic Models~~

~~This chapter defines the general rules for arithmetic models.~~

11.2.1.1 Principles of arithmetic models

The purpose of arithmetic models is to specify calculable mathematical relationships between objects representing physical quantities in the library. Arithmetic models are identified by context-sensitive keywords, because how these quantities are measured, extracted, or interpreted depends on the context in which the objects are placed.

The quantity identified by the keyword `CAPACITANCE` can serve as example. In the context of a `PIN`, it represents pin capacitance. In the context of a `WIRE`, it represents wire capacitance. In the context of a `RULE`, it represents the calculation method for a capacitance formed by a layout pattern described within the rule. The context-specific semantics of each arithmetic model are specified in 8 for electrical models and 9 for physical models.

In certain cases, the context alone does not completely specify the semantics of an arithmetic model. Auxiliary definitions within the arithmetic model are needed; these are represented by using annotations or annotation containers.

A simple example is the `UNIT` annotation, which is applicable for most arithmetic models. It specifies the unit in terms of which the arithmetic model data is represented. The applicable auxiliary objects for each arithmetic model are specified in 8 for electrical models and 9 for physical models.

11.2.1.1.1 Global definitions for arithmetic models

In many cases, auxiliary definitions apply globally to all arithmetic models within a certain context, for instance, the `UNIT` can apply for all `CAPACITANCE` objects within a library. In order to specify such global definitions, the arithmetic model construct can be used without data.

A model definition XXX, as shown in Syntax 90.

```
model_definition ::=
    model_keyword [ identifier ] { all_purpose_items }
```

Syntax 90—model_definition

This construct has the syntactical form of an `annotation_container` (see 11.7).

11.2.1.1.2 Trivial arithmetic model

The simplest form of an arithmetic model contains just constant data, as shown in Syntax 91.

```
trivial_model ::=
      model_keyword [ identifier ] = number ;
    | model_keyword [ identifier ] = number { all_purpose_items }
```

Syntax 91—trival_model

This construct has the syntactical form of an `annotation` (see 11.7).

11.2.1.1.3 Arithmetic model using EQUATION

The arithmetic model data can be represented as an `EQUATION`. In this case, a `HEADER` defines the arguments of the equation. It is also possible to use other arithmetic models, which are visible within the context of this arithmetic model, as arguments. Those arguments need not appear in the `HEADER`, as shown in Syntax 92.

```
equation_based_model ::=
    model_keyword [ identifier ] {
      [ all_purpose_items ] [ equation_based_header ] equation }
equation_based_header ::=
      HEADER { model_keyword { model_keyword } }
    | HEADER { model_definition { model_definition } }
equation ::=
    EQUATION { arithmetic_expression }
```

Syntax 92—equation_based_model

The syntax of `arithmetic_expression` is explained in xxx.

11.2.1.1.4 Arithmetic model using TABLE

The arithmetic model data can be represented as a lookup table. In this case, a `TABLE` is necessary for the data itself and for each argument, as shown in Syntax 93.

```
table_based_model ::=
    model_keyword [ identifier ] {
      [ all_purpose_items ] table_based_header table [ equation ] }
table_based_header ::=
    HEADER { table_model_definition { table_model_definition } }
table_model_definition ::=
    model_keyword [ identifier ] { all_purpose_items table }
table ::=
      TABLE { symbol { symbol } }
    | TABLE { number { number } }
```

Syntax 93—table_based_model

Tables containing symbols are only meant for lookup of discrete datapoints. Tables containing numbers are for calculation and, eventually, interpolation of datapoints. The *`model`*`_keyword` (see 8 and 9) defines whether symbols or numbers are legal for a particular table.

The size of the table inside the `table_based_model` shall be the product of the size of the tables inside the `table_header`. In order to support interpolation, the numbers in each table inside the `table_header` shall be in strictly monotonic ascending order. See 11.2.1.2 for more details.

The `table_model_definition` can also be used outside the context of a `table_header`, very much like a `model_definition`. In this case, the `model_definition` supplies the same information as the `table_model_definition`, plus the additional information of a discrete set of valid numbers applicable for the model.

For example, the `WIDTH` of a physical layout object can contain only a discrete set of legal values. Those can be specified using a `table_model_definition`.

However, the table in a `table_model_definition` *outside* a `table_header` shall not substitute the table *inside* the `table_header`. The former defines a legal set of values, the latter defines the table-lookup indices.

If all table data are numbers, the `table_based_model` can also have an optional equation. This equation is to be used when the argument data are out of interpolation range. Without the equation, extrapolation shall be applied for data which are out of range.

11.2.1.1.5 Complex arithmetic model

A complex arithmetic model can be constructed by defining a nested arithmetic model within another arithmetic model, as shown in Syntax 94.

```
complex_model ::=
      model_keyword [ identifier ] {
       [ all_purpose_items ] HEADER { model { model } }
       equation }
    | model_keyword {
       all_purpose_items HEADER { header_model { header_model } }
       table [ equation ] }
header_model ::=
      model_definition
    | table_model_definition
    | equation_based_model
    | table_based_model
    | header_table_model
header_table_model ::=
    model_keyword [ identifier ] {
       all_purpose_items HEADER { symbol { symbol } }
       TABLE { number { number } } }
```

Syntax 94—complex_model

The data of the inner arithmetic model is calculated first. Then the result is applied for calculation of the data of the outer arithmetic model.

If any `header_model` is either `model_definition` or `table_model_definition`, then the `complex_model` reduces to the previously defined `equation_based_model` and `table_based_model`, respectively. In order to support a table in the `general_model`, any `header_model` shall be either a `table_model_definition` or `table_based_model`, and the numbers in each table inside each `header_model` shall be strictly monotonically increasing.

The `header_table_model` construct can be used to associate symbols with numbers. For example, process corners can be defined as discrete symbols and associated with process derating factors. The numbers can be used in equations and for interpolation, whereas the symbols cannot.

11.2.1.2 Construction of 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 shall be in the `HEADER`. The ALF parser shall issue an error if the `EQUATION` uses an argument not defined in the `HEADER`. A warning shall be issued if the `HEADER` contains arguments not used in the `EQUATION`.

Example

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

If the model uses a TABLE, then each argument in the HEADER also needs a table defining the format. The order of arguments decides how the index to each entry is calculated. The first argument is the innermost index, the following arguments are outer indices.

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

The first argument CAPACITANCE has four entries. The second argument SLEWRATE has three entries. Thus, DELAY has 4*3=12 entries. For readability, comments can be inserted in the table.

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

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

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

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

```
DELAY rise_out{
    ...
```

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

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

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

11.2.1.3 Arithmetic submodels

Arithmetic submodels can be used to distinguish different measurement conditions for the same model. The root of an arithmetic model can contain nested arithmetic submodels. The header of an arithmetic model can contain nested arithmetic models, but not arithmetic submodels.

The arithmetic submodels shown in Table 69 are generally applicable.

Table 69—Generally applicable arithmetic submodels

Object	Description
MIN	For measured or calculated data: the data represents the minimal value / set of values within a statistical distribution. For data within LIMIT container: the data represents the lower limi.t value / set of values
TYP	For measured or calculated data: the data represents the typical value / set of values within a statistical distribution.
MAX	For measured or calculated data: the data represents the maximal value / set of values within a statistical distribution. For data within LIMIT container: the data represents the lower limit value / set of values.
DEFAULT	For measured or calculated data: the data represents the default value / set of values to be used per default.

The arithmetic submodels shown in Table 70 are only applicable in the context of electrical modeling.

Table 70—Submodels restricted to electrical modeling

Object	Description
HIGH	Applicable for electrical data measured at a logic high state of a pin.
LOW	Applicable for electrical data measured at a logic low state of a pin.

Table 70—Submodels restricted to electrical modeling (Continued)

Object	Description
RISE	Applicable for electrical data measured during a logic low to high transition of a pin.
FALL	Applicable for electrical data measured during a logic high to low transition of a pin.

The arithmetic submodels shown in Table 71 are only applicable in the context of physical modeling.

Table 71—Submodels restricted to physical modeling

Object	Description
HORIZONTAL	Applicable for layout measurements in horizontal direction.
VERTICAL	Applicable for layout measurements in vertical direction.

The semantics of the restricted submodels are explained in 8 and 9.

11.2.2 Arithmetic model statement

An arithmetic model statement XXX, as shown in Syntax 95.

```
arithmetic_models ::=
    arithmetic_model { arithmetic_model }
arithmetic_model ::=
      partial_arithmetic_model
    | non_trivial_arithmetic_model
    | trivial_arithmetic_model
    | assignment_arithmetic_model
    | arithmetic_model_template_instantiation
```

Syntax 95—Arithmetic model statement

11.2.3 Partial arithmetic model

A partial arithmetic model XXX, as shown in Syntax 96.

```
partial_arithmetic_model ::=
    nonescaped_identifier [ arithmetic_model_identifier ] { partial_arithmetic_model_items }
partial_arithmetic_model_items ::=
    partial_arithmetic_model_item { partial_arithmetic_model_item }
partial_arithmetic_model_item ::=
      any_arithmetic_model_item
    | table
```

Syntax 96—Partial arithmetic model

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

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

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

— If a partial arithmetic model outside a HEADER contains a TABLE, the arithmetic values in the TABLE shall define a discrete set of valid values for the model.
— If a partial arithmetic model within a HEADER contains a TABLE, the arithmetic values in the TABLE shall define the entries for table-lookup.

11.2.4 Non-trivial arithmetic model

A non-trivial arithmetic model XXX, as shown in Syntax 97.

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

Syntax 97—Non-trivial arithmetic model

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

11.2.5 Trivial arithmetic model

A trivial arithmetic model XXX, as shown in Syntax 98.

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

Syntax 98—Trivial arithmetic model

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

11.2.6 Assignment arithmetic model

An assignment arithmetic model XXX, as shown in Syntax 99.

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

Syntax 99—Assignment arithmetic model

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

— A partial arithmetic model has been defined using the arithmetic model identifier AND arithmetic models for all arguments contained in the arithmetic expression have been defined.
— This construct is used in a dynamic template instantiation.

11.2.7 Items for any arithmetic model

Arithmetic model items XXX, as shown in Syntax 100.

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

Syntax 100—Arithmetic model items

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

11.3 Arithmetic submodel and related statements

Add lead-in text

11.3.1 Arithmetic submodel statement

An arithmetic submodel statement XXX, as shown in Syntax 101.

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

Syntax 101—Arithmetic submodel statement

11.3.2 Non-trivial arithmetic submodel

A non-trivial arithmetic submodel XXX, as shown in Syntax 102.

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

Syntax 102—Non-trivial arithmetic submodel

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

11.3.3 Trivial arithmetic submodel

A trivial arithmetic submodel XXX, as shown in Syntax 103.

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

Syntax 103—Trivial arithmetic submodel

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

11.3.4 Items for any arithmetic submodel

Arithmetic submodel items XXX, as shown in Syntax 104.

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

Syntax 104—Arithmetic submodel items

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

11.4 Arithmetic body and related statements

Add lead-in text

11.4.1 Arithmetic body

An arithmetic body XXX, as shown in Syntax 105.

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

Syntax 105—Arithmetic body

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

11.4.2 HEADER statement

A HEADER statement XXX, as shown in Syntax 106.

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

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

```
header ::=
      HEADER { identifiers }
    | HEADER { header_arithmetic_models }
    | header_template_instantiation
header_arithmetic_models ::=
    header_arithmetic_model { header_arithmetic_model }
header_arithmetic_model ::=
      non_trivial_arithmetic_model
    | partial_arithmetic_model
```

Syntax 106—HEADER statement

11.4.3 TABLE statement

A TABLE statement XXX, as shown in Syntax 107.

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

Syntax 107—TABLE statement

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

11.4.4 EQUATION statement

An EQUATION statement XXX, as shown in Syntax 108.

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

Syntax 108—EQUATION statement

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

11.5 Arithmetic model container

An arithmetic model container XXX, as shown in Syntax 109.

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

Syntax 109—Arithmetic model container

~~Containers for arithmetic models~~

The keywords shown in Table 72 are defined for objects that can contain arithmetic models.

Table 72—Unnamed containers for arithmetic models

Object	Description
FROM	Contains start point of timing measurement or timing constraint.
TO	Contains end point of measurement or timing constraint.
LIMIT	Contains arithmetic models for limit values.
EARLY	Contains arithmetic models for timing measurements relevant for early signal arrival time.
LATE	Contains arithmetic models for timing measurements relevant for late signal arrival time.

The LIMIT container is for general use. The FROM, TO, EARLY, and LATE containers are only for use within the context of timing models.

11.5.1 LIMIT container

A LIMIT container shall contain arithmetic models. The arithmetic models shall contain submodels identified by MIN and/or MAX.

Example

```
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.

```
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.

11.5.2 Containers for arithmetic models and submodels

Containers for arithmetic models can supplement the context-specific semantics of the arithmetic model. Therefore, arithmetic models can be placed in the context of arithmetic model containers, as shown in Syntax 110.

```
model_container ::=
    model_container_keyword {
      [ all_purpose_items ] model_container_contents { model_container_contents } }
model_container_contents ::=
      model_container
    | trivial_model
    | complex_model
```

Syntax 110—model_container

There is a dedicated set of *model_container*_keywords. In addition, *model*_keywords can also be used as *model_container*_keywords and dedicated *submodel*_keywords can be used as *model*_keywords. The number of levels in nested arithmetic model containers is restricted by the set of allowed combinations between *model_container*_keywords, *model*_keywords and *submodel*_keywords (see 11.2.1.3).

11.6 Statements related to arithmetic models for general purpose

Add lead-in text

11.6.1 MIN and MAX statements

~~Semantics of MIN / TYP / MAX~~

MIN, TYP, and MAX indicate the data of the arithmetic model represent minimal, typical, or maximal values within a statistical distribution. No correlation is assumed or implied between MIN data, TYP data, or MAX data across different arithmetic models.

Example

```
DELAY {
    FROM { PIN=A; } TO { PIN=Z; }
    MIN = 0.34; TYP = 0.38; MAX = 0.45;
}
POWER {
    MEASUREMENT = average; FREQUENCY = 1e6;
    MIN = 1.2; TYP = 1.4; MAX = 1.5;
}
```

The MIN value for DELAY could simultaneously apply with the MIN value for POWER. Typically, the case with smaller delay is also the case with larger power consumption.

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

Example

```
LIMIT {
    SLEWRATE { PIN=A; MAX=5.0; }
    VOLTAGE { PIN=VDD; MIN=1.6; MAX=2.0; }
}
```

MIN, MAX as an annotation inside a model or inside a model argument within the HEADER define the validity range of the data. 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.

Example

```
POWER {
    HEADER {
       SLEWRATE { PIN=A; MIN=0.01; MAX=5.0; TABLE { 0.1 0.5 1.0 } }
       CAPACITANCE { PIN=Z; TABLE { 0.0 0.4 0.8 1.6 } }
    }
    TABLE { 0.2 0.3 0.6 0.4 0.5 0.7 0.8 0.8 1.0 1.5 1.5 1.6 }
}
```

The data for POWER is valid for SLEWRATE in the range between 0.01 and 5.0 (via extrapolation) and for CAPACITANCE in the range between 0.0 and 1.6.

11.6.2 TYP statement

Add lead-in text

11.6.3 DEFAULT statement

Add lead-in text

11.6.3.1 DEFAULT annotation

Default annotation promotes 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 can 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`.

11.6.3.2 Semantics of DEFAULT

Arithmetic submodels can be identified by `MIN`, `TYP`, and `MAX` or context-restricted keywords. For cases where the application tool cannot decide which qualifier applies, a supplementary arithmetic submodel with the qualifier `DEFAULT` can be used.

Example

```
PIN my_pin {
    CAPACITANCE {
       MIN { HEADER { ... } TABLE { ... } }
       TYP { HEADER { ... } TABLE { ... } }
       MAX { HEADER { ... } TABLE { ... } }
       DEFAULT { HEADER { ... } TABLE { ... } }
    }
}
```

NOTE—The `DEFAULT` model can also degenerate to a single value; it represents a trivial arithmetic model.

In certain cases, there is no supplementary submodel. Instead, one of the already defined submodels is used by default. For this case, the DEFAULT annotation can be used to point to the applicable keyword.

Example

```
PIN my_pin {
    CAPACITANCE {
       MIN { HEADER { ... } TABLE { ... } }
       TYP { HEADER { ... } TABLE { ... } }
       MAX { HEADER { ... } TABLE { ... } }
       DEFAULT = TYP;
    }
}
```

The trivial arithmetic model construct with DEFAULT can also be used for an argument in the context of the HEADER of an arithmetic model. This enables evaluation of the arithmetic model in case the data of the argument can not be supplied by the application tool.

Example

```
PIN my_pin {
    CAPACITANCE {
       HEADER { TEMPERATURE { DEFAULT=50; TABLE { 0 50 100 } } }
       TABLE { 0.05 0.07 0.10 } }
    }
}
```

The DEFAULT value of the CAPACITANCE here is 0.07.

11.6.4 LIMIT statement

~~Reliability calculation~~

In general, reliability is modeled by arithmetic models using the LIMIT construct.

11.6.4.1 Global LIMIT specifications

Global limits can be specified for electrical quantities, even if they are related to CELLs, PINs, or VECTORs. Such global limits apply, unless local limits are specified within the context of CELLs, PINs, or VECTORs. The priorities are given below.

a) LIMIT within the context of the VECTOR
b) LIMIT within the context of a PIN (if the LIMIT in the VECTOR has PIN annotation)
c) LIMIT within the context of the CELL
d) LIMIT within the context of the SUBLIBRARY
e) LIMIT within the context of the LIBRARY
f) LIMIT outside LIBRARY

The arguments in the HEADER of the LIMIT model can only be items that are visible within the scope of the LIMIT model. In particular, arguments with PIN annotations are only legal for LIMIT models in the context of a CELL or a VECTOR within the CELL.

11.6.4.2 LIMIT and model specification in the same context

An arithmetic model for a physical quantity and a limit specification for the same physical quantity can appear within the same context, for example, an arithmetic model for FLUENCE calculation and a LIMIT for FLUENCE within the context of a VECTOR. In such a case, the calculated quantity shall be checked against the limit of the quantity within that context.

On the other hand, if multiple arithmetic models are given within the context for which the limit applies, the limit shall be checked against the combination of all arithmetic models in the case of cumulative quantities, or against the minimum or maximum calculated value in the case of non-cumulative or mutually exclusive quantities.

For example, a LIMIT for FLUENCE can be given in the context of a CELL. Calculation models for FLUENCE can be given for multiple VECTORs within the context of the CELL. The LIMIT for FLUENCE shall be checked against the accumulated FLUENCE calculated for all VECTORs.

Example

```
CELL my_cell {
    PIN A { DIRECTION = input; }
    PIN B { DIRECTION = input; }
    PIN C { DIRECTION = input; }
    PIN Z { DIRECTION = output; }
    LIMIT { FLUENCE { MAX = 1e20; } } }
    VECTOR ( 01 A -> 10 Z ) {
       FLUENCE = 1e-5;
    }
    VECTOR ( 01 B -> 10 Z ) {
       FLUENCE = 1e-5;
    }
    VECTOR ( 01 C -> 10 Z ) {
       FLUENCE = 1e-6;
       LIMIT { FLUENCE { MAX = 1e18; } }
    }
}
```

The fluence limit for the cell is reached after 10^{25} occurrences of `VECTOR ( 01 A -> 10 Z )` or `VECTOR ( 01 B -> 10 Z )` counted together. The fluence limit for the `VECTOR ( 01 C -> 10 Z )` is reached after 10^{24} occurrences of that vector.

An example for a non-cumulative quantity is SLEWRATE. The VECTORs in the context of which SLEWRATE is modeled describe timing arcs with mutually exclusive conditions. Therefore, if a minimum or maximum LIMIT for SLEWRATE is given for a PIN in the context of a CELL, this SLEWRATE shall be checked against the minimum or maximum value of any calculated SLEWRATE applicable to that PIN.

Example

```
CELL my_cell {
    PIN A { DIRECTION = input; }
    PIN B { DIRECTION = input; }
    PIN C { DIRECTION = input; }
    PIN Z { DIRECTION = output; LIMIT { SLEWRATE { MAX = 5; } } }
    VECTOR ( 01 A -> 10 Z ) {
       SLEWRATE { PIN = Z; /* fill in HEADER, TABLE */ }
    }
```

```
    VECTOR ( 01 B -> 10 Z ) {
       SLEWRATE { PIN = Z; /* fill in HEADER, TABLE */ }
    }
    VECTOR ( 01 C -> 10 Z ) {
       SLEWRATE { PIN = Z; /* fill in HEADER, TABLE */ }
    }
}
```

Here the slewrate on pin Z calculated in the context of any vector is checked against the same maximum limit.

11.6.4.3 Model and argument specification in the same context

An cumulative quantity can also be an argument in the HEADER of an arithmetic model. If the model for calculation of that quantity is within the same context as the argument of the other model, then the value of the calculated quantity shall be used. Otherwise, the value of the accumulated quantity shall be used.

For example, SLEWRATE can be modeled as a function of FLUENCE in the context of a VECTOR. If a calculation model for FLUENCE appears in the context of the same VECTOR, the value for FLUENCE shall be used for the SLEWRATE calculation. On the other hand, if there is no calculation model for FLUENCE in the context of the same VECTOR, but there is one in the context of other VECTORs, then the accumulated value of FLUENCE from the other calculation models shall be used for SLEWRATE calculation.

Example

```
CELL my_cell {
    PIN A { DIRECTION = input; }
    PIN B { DIRECTION = input; }
    PIN C { DIRECTION = input; }
    PIN Z { DIRECTION = output; }
    VECTOR ( (01 A | 01 B) -> 10 Z ) { FLUENCE = 1e-5; }
    VECTOR ( 01 A -> 10 Z ) {
       SLEWRATE { CALCULATION=incremental; PIN = Z;
          HEADER { FLUENCE } EQUATION { 1e-8 * FLUENCE }
       }
    }
    VECTOR ( 01 B -> 10 Z ) {
       SLEWRATE { CALCULATION=incremental; PIN = Z;
          HEADER { FLUENCE } EQUATION { 1e-8 * FLUENCE }
    }
    }
    VECTOR ( 01 C -> 10 Z ) {
       FLUENCE = 1e-6;
       SLEWRATE { CALCULATION=incremental; PIN = Z;
          HEADER { FLUENCE } EQUATION { 1e-9 * FLUENCE }
       }
    }
}
```

After $10^{13} = 10^5 * 10^8$ occurrences of `VECTOR ( (01 A | 01 B) -> 10 Z )`, the slewrate at pin Z for `VECTOR ( 01 A -> 10 Z )` and `VECTOR ( 01 B -> 10 Z )` is increased by 1 unit.
After $10^{15} = 10^6 * 10^9$ occurrences of `VECTOR ( 01 C -> 10 Z )`, the slewrate at pin Z for `VECTOR ( 01 C -> 10 Z )` is increased by 1 unit.

11.6.5 Annotations for arithmetic models for general purpose

~~Annotations for arithmetic models~~

Annotations and annotation containers described in this section 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 can also have the form of arithmetic models themselves, in which case they represent nested arithmetic models.

11.6.5.1 UNIT annotation

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 values (`*` indicates a wild card) shown in Table 73.

Table 73—UNIT annotation

Annotation string	Description
`f*` or `F*`	Equivalent to `1E-15`.
`p*` or `P*`	Equivalent to `1E-12`.
`n*` or `N*`	Equivalent to `1E-9`.
`u*` or `U*`	Equivalent to `1E-6`.
`m*` or `M*`	Equivalent to `1E-3`.
`1*`	Equivalent to `1E+0`.
`k*` or `K*`	Equivalent to `1E+3`.
`meg*` or `MEG*`[a]	Equivalent to `1E+6`.
`g*` or `G*`	Equivalent to `1E+9`.

[a]or any uppercase/lowercase combination of these three characters

Arithmetic models are context-sensitive, i.e., the units for their values can be determined from the context. If the `UNIT` annotation for such a context does not exist, default units are applied to the value (see 11.2.1.3).

Example

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

If the unit is a string, then only the first character (the first three 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 an inch instead of a meter as the base unit, the following specification does 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.

11.6.5.2 CALCULATION annotation

An arithmetic model in the context of a `VECTOR` can have the `CALCULATION` annotation defined as shown in Syntax 111.

*calculation*_annotation ::=
CALCULATION = *calculation*_identifier **;**
*calculation*_identifier ::=
absolute
| **incremental**

Syntax 111—calculation_annotation

It shall specify whether the data of the model are to be used by themselves or in combination with other data. The default is **absolute**.

The **incremental** data from one `VECTOR` shall be added to **absolute** data from another `VECTOR` under the following conditions:

— The model definitions are compatible, i.e., measurement specifications shall be the same. Units are allowed to be different.
Example: slewrate measurements at the same pin, same switching direction, and same threshold values.
— The model definitions for common arguments are compatible, i.e., the same range of values for table-based models and measurement specifications are the same. Units can be different.
Example: same values for `derate_case` and same threshold definitions for input slewrate.
— The vector definitions are compatible, i.e, the `vector_or_boolean_expression` of the `VECTOR` containing **incremental** data matches the `vector_or_boolean_expression` of the `VECTOR` containing **absolute** data by removing all variables appearing exclusively in the former expression.

Example

```
VECTOR ( 01 A -> 01 Z ) {
    DELAY {
       CALCULATION = absolute;
       FROM { PIN = A; } TO { PIN = Z; }
       HEADER {
          CAPACITANCE load { PIN = Z; }
          SLEWRATE slew { PIN = A; }
       }
       EQUATION { 0.5 + 0.3*slew + 1.2*load }
    }
}
VECTOR ( 01 A &> 01 B &> 01 Z ) {
    DELAY {
       CALCULATION = incremental;
       FROM { PIN = A; } TO { PIN = Z; }
       HEADER {
          SLEWRATE slew_A { PIN = A; }
          SLEWRATE slew_B { PIN = B; }
          TIME time_A_B { FROM { PIN = A; } TO { PIN = B; } }
       }
       EQUATION {- 0.1 + (0.05+0.002*slew_A*slew_B)*time_A_B) }
    }
}
```

Both models describe the rise-to-rise delay from A to Z. The second delay model describes the incremental delay (here negative), when input B switches in a time window between A and Z.

11.6.5.3 INTERPOLATION annotation

An argument of a table-based arithmetic model, i.e., a model in the `HEADER` containing a `TABLE` statement, can have the `INTERPOLATION` annotation defined as shown in Syntax 112.

*interpolation*_annotation ::=
INTERPOLATION = *interpolation*_identifier **;**
*interpolation*_identifier ::=
fit
| **linear**
| **floor**
| **ceiling**

Syntax 112—interpolation_annotation

This also needs to specify the interpolation scheme for the values in-between the values of the TABLE.

- **fit**
  the data points in the table are supposed to be part of a smooth curve. Linear interpolation or other algorithms, e.g., cubic spline or polynomial regression can be used to fit the data points into the curve.
- **linear**
  the data points in the table are supposed to be part of a piece wise linear curve. Linear interpolation shall be used.
- **floor**
  the value to the left in the table, i.e., the smaller value is used.
- **ceiling**
  the value to the right in the table, i.e., the larger value is used.

The default is **fit**. For multi-dimensional tables, different interpolation schemes can be used for each dimension.

Example

```
my_model {
    HEADER {
       dimension1 { INTERPOLATION = fit; TABLE { 1 2 4 8 }
       dimension2 { INTERPOLATION = floor; TABLE { 10 100 }
       dimension3 { INTERPOLATION = ceiling; TABLE { 10 100 }
    }
    TABLE {
       1 7 3 5
       10 20 60 40
       50 30 20 100
       0.8 0.4 0.2 0.9
    }
}
```

Consider the following values:

```
    dimension1 = 6
=> following subtable is chosen:
       3   5    // interpolation between 3 and 5
       60  40   // or between 60 and 40
       20  100  // or between 20 and 100
       0.2 0.9  // or between 0.2 and 0.9
    dimension2 = 50
=> following subtable is picked:
       3   5    // interpolation between 3 and 5
       20  100  // or between 20 and 100
    dimension3 = 50
=> following subtable is picked:
       20  100  // interpolation between 20 and 100
```

The following rules shall apply for each dimension of a table-based model:

For values outside the range of the table, extrapolation shall apply, using the table data points at the leftmost or rightmost side, respectively, as reference.

If the value is smaller than the smallest, i.e. leftmost, data point in the table, the extrapolation shall be calculated as if the value would fall in-between the leftmost and second leftmost value.

If the value is greater than the greatest, i.e. rightmost, data point in the table, the extrapolation shall be calculated as if the value would fall in-between the rightmost and second rightmost value.

Example

```
my_model Y {
    HEADER {
       my_argument X {
          TABLE { 0    2    4    8   }
          //      X[0] X[1] X[2] X[3]
       }
    }
    TABLE { 0.5  0.6  1.0  1.5 }
    //      Y[0] Y[1] Y[2] Y[3]
}
```

For linear interpolation, the following equation is used:

$$Y = Y[N] + \frac{Y[N+1] - Y[N]}{X[N+1] - X[N]} \cdot X \qquad X[N] \le X \le X[N+1]$$

If X < X[0], the values X[0], X[1], Y[0], Y[1] are plugged into the equation.

If X > X[3], the values X[2], X[3], Y[2], Y[3] are plugged into the equation.

Figure 33 illustrates a non-linear interpolation scheme with the goal of fitting three neighboring points into a smooth curve.

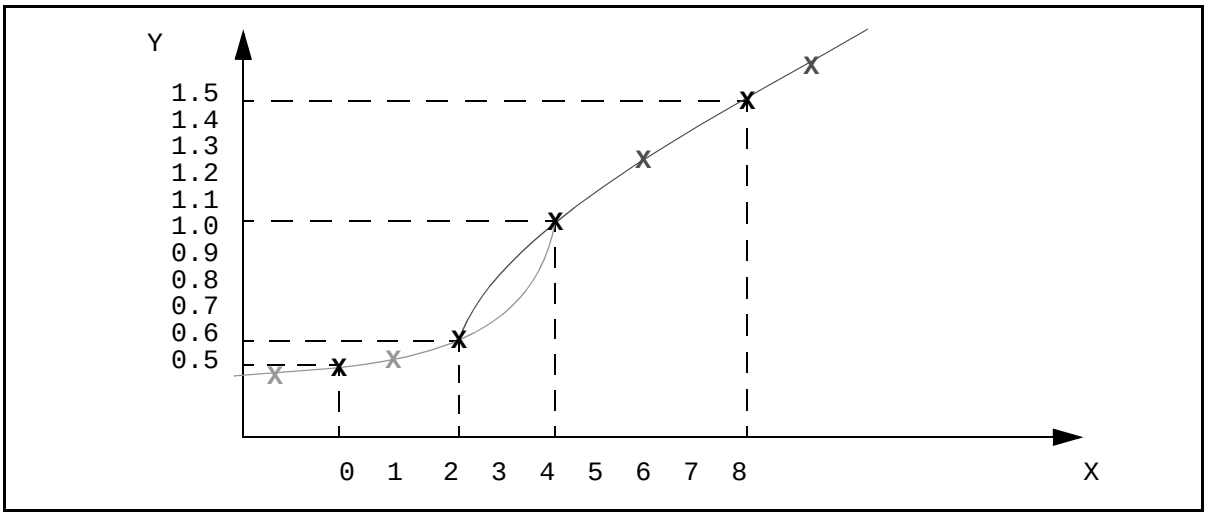


Figure 33—Illustration of extrapolation rules

The curve based on the 3 rightmost or the 3 leftmost points, respectively, is used for extrapolation to the right side or the left side, respectively.

11.7 Rules for evaluation of arithmetic models

Add lead-in text

11.7.1 Arithmetic model with arithmetic submodels

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

11.7.2 Arithmetic model with table arithmetic body

All arithmetic models in the HEADER shall contain a TABLE.

- Describe algorithm to identify correct table entry.
- Refer to INTERPOLATION annotation.

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

11.7.3 Arithmetic model with equation arithmetic body

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

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

A supplementary TABLE is legal; this shall be used asa lookup entry for downstream arithmetic models, when the arithmetic model itself is within HEADER.

11.8 Overview of arithmetic models

Add lead-in text

~~Electrical Performance Modeling~~

11.8.1 Overview of modeling keywords

This section details the keywords used for performance modeling.

11.8.1.1 Timing models

Table 74 — Table 77 show the set of keywords used for timing measurements and constraints. All keywords have implied semantics that restrict their capability to describe general temporal relations between arbitrary signals. For unrestricted purposes, the keyword `TIME` shall be used.

Table 74—Timing measurements

Keyword	Value type	Base units	Default units	Description
`DELAY`	number	Second	n (nano)	Time between two threshold crossings within two consecutive events on two pins. A causal relationship between the two events is implied.
`RETAIN`	number	Second	n (nano)	Time when an output pin shall retain its value after an event on the related input pin. `RETAIN` appears always in conjunction with `DELAY` for the same two pins.
`SLEWRATE`	non-negative number	Second	n (nano)	Time between two threshold crossings within one event on one pin.

Table 75—Timing constraints

Keyword	Value type	Base units	Default units	Description
`HOLD`	number	Second	n (nano)	Minimum time limit for hold between two threshold crossings within two consecutive events on two pins.
`NOCHANGE`	optional[a] non-negative number	Second	n (nano)	Minimum time limit between two threshold crossings within two arbitrary consecutive events on one pin, in conjunction with `SETUP` and `HOLD`.
`PERIOD`	non-negative number	Second	n (nano)	Minimum time limit between two identical events within a sequence of periodical events.
`PULSEWIDTH`	number	Second	n (nano)	Minimum time limit between two threshold crossings within two consecutive and complementary events on one pin.
`RECOVERY`	number	Second	n (nano)	Minimum time limit for recovery between two threshold crossings within two consecutive events on two pins.
`REMOVAL`	number	Second	n (nano)	Minimum time limit for removal between two threshold crossings within two consecutive events on two pins.
`SETUP`	number	Second	n (nano)	Minimum time limit for setup between two threshold crossings within two consecutive events on two pins.
`SKEW`	number	Second	n (nano)	Absolute value is maximum time limit between two threshold crossings within two consecutive events on two pins; the sign indicates positive or negative direction.

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

Table 76—Generalized timing measurements

Keyword	Value type	Base units	Default units	Description
TIME	number	Second	1 (unit)	Time point for waveform modeling, time span for average, RMS, and peak modeling .
FREQUENCY	non-negative number	Hz	meg (mega)	Frequency.
JITTER	non-negative number	Second	n (nano)	Uncertainty of arrival time.

Table 77—Normalized measurements

Keyword	Value type	Base units	Default units	Description
THRESHOLD	non-negative number between 0 and 1	Normalized signal voltage swing	1 (unit)	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.
NOISE_MARGIN	non-negative number between 0 and 1	Normalized signal voltage swing	1 (unit)	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.

11.8.1.2 Analog models

Table 78 and Table 79 define the keywords for analog modeling.

Table 78—Analog measurements

Keyword	Value type	Base units	Default units	Description
CURRENT	number	Ampere	m (milli)	Electrical current drawn by the cell. A pin can be specified as annotation.[a]
ENERGY	number	Joule	p (pico)	Electrical energy drawn by the cell, including charge and discharge energy, if applicable.
POWER	number	Watt	u (micro)	Electrical power drawn by the cell, including charge and discharge power, if applicable.

Table 78—Analog measurements (Continued)

Keyword	Value type	Base units	Default units	Description
TEMPERATURE	number	^{o} Celsius	1 (unit)	Temperature.
VOLTAGE	number	Volt	1 (unit)	Voltage.
FLUX	non-negative number	Coulomb per Square Meter	1 (unit)	Amount of hot electrons in units of electrical charge per gate oxide area.
FLUENCE	non-negative number	Second times Coulomb per Square Meter	1 (unit)	Integral of FLUX over time.

[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 79—Electrical components

Keyword	Value type	Base units	Default units	Description
CAPACITANCE	non-negative number	Farad	p (pico)	Pin, wire, load, or net capacitance.
INDUCTANCE	non-negative number	Henry	n (nano)	Pin, wire, load, or net inductance.
RESISTANCE	non-negative number	Ohm	K (kilo)	Pin, wire, load, or net resistance.

11.8.1.3 Supplementary models

Table 80 and Table 81 define the keywords for supplementary models.

Table 80—Abstract measurements

Keyword	Value type	Base units	Default units	Description
DRIVE_STRENGTH	non-negative number	None	1 (unit)	Drive strength of a pin, abstract measure for (drive resistance)$^{-1}$.
SIZE	non-negative number	None	1 (unit)	Abstract cost function for actual or estimated area of a cell or a block.

Table 81—Discrete measurements

Keyword	Value type	Base units	Default units	Description
SWITCHING_BITS	non-negative number	None	1	Number of switching bits on a bus.
FANOUT	non-negative number	None	1	Number of receivers connected to a net.
FANIN	non-negative number	None	1	Number of drivers connected to a net.
CONNECTIONS	non-negative number	None	1	Number of pins connected to a net, where CONNECTIONS = FANIN+FANOUT.

The actual values for discrete measurements are always integer numbers, however, estimated values can be non-integer numbers (e.g., the average fanout of a net is 2.4).

Table 82 describes the arguments for arithmetic models to describe environmental dependency.

Table 82—Environmental data

Annotation string	Value type	Description
DERATE_CASE	string	Derating case, i.e., the combination of process, supply voltage, and temperature.
PROCESS	string	Process corner.
TEMPERATURE	number	Environmental temperature.

11.8.2 Arithmetic models in the context of layout

Table 83 shows keywords for arithmetic models in the context of layout.

Table 83—Arithmetic models for layout data

Keyword	Value type	Base units	Default units	Description
SIZE	Non-negative number	N/A	1	Abstract, unitless measurement for the size of a physical object.
AREA	Non-negative number	Square Meter	p (pico)	Area in square microns (pico = micro^2).
DISTANCE	Non-negative number	Meter	u (micro)	Distance between two points in microns.
HEIGHT	Positive number	Meter	u (micro)	y- dimension of a placeable object (e.g., cell or block). z- dimension of a routeable object (e.g., pattern on routing layer), representing the absolute height above substrate.

Table 83—Arithmetic models for layout data (Continued)

Keyword	Value type	Base units	Default units	Description
LENGTH	Positive number	Meter	u (micro)	x-, or y- dimension of a routeable object (e.g., pattern on routing layer) measured in routing direction.
WIDTH	Positive number	Meter	u (micro)	x-dimension of a placeable object (e.g., cell or block). x- or y- dimension of a routeable object (e.g., pattern on routing layer) measured in orthogonal direction to the route.
PERIMETER	Positive number	Meter	u (micro)	Circumference of a physical object.
THICKNESS	Positive number	Meter	u (micro)	z- dimension of a manufacturable physical object, representing the distance between the bottom of the object above and the top of the object below.
OVERHANG	Non-negative number	Meter	u (micro)	Distance between the edges of two overlapping physical objects.
EXTENSION	Non-negative number	Meter	u (micro)	Distance between the center and the outer edge of a physical object.

Table 84 — Table 93 summarize the semantic meanings of arithmetic model keywords in the context of layout.

Table 84—Semantic meaning of SIZE

Context	Meaning
CELL	Abstract measure for size of the cell, cost function for design implementation.
WIRE	- As a model (TABLE or EQUATION): abstract measure for the size of the wire itself. - As argument of a model (HEADER): abstract measure for size of the block for which the wireload model applies, can be calculated by combining the size of all cells and all wires in the block.
ANTENNA	Abstract measure for size of the antenna for which the antenna rule applies.

Table 85—Semantic meaning of WIDTH

Context	Meaning
CELL, SITE	Horizontal distance between cell or site boundaries, respectively.
WIRE	As argument of a model (HEADER): horizontal distance between block boundaries for which wireload model applies.
LAYER, ANTENNA	Width of a wire, orthogonal to routing direction.

Table 86—Semantic meaning of HEIGHT

Context	Meaning
CELL, SITE	Vertical distance between cell or site boundaries, respectively.
WIRE	As argument of a model (HEADER): vertical distance between block boundaries for which wireload model applies.
LAYER	Distance from top of ground plane to bottom of wire.

Table 87—Semantic meaning of LENGTH

Context	Meaning
WIRE	Estimated routing length of a wire in a wireload model.
LAYER, ANTENNA	Actual routing length of a wire in layout.

Table 88—Semantic meaning of AREA

Context	Meaning
CELL	Physical area of the cell, product of width and height of a rectangular cell.
WIRE	- As a model (TABLE or EQUATION): physical area of the wire itself. - As argument of a model (HEADER): physical area of the block for which wireload model applies, product of width and height of rectangular block.
LAYER, VIA, ANTENNA	Physical area of a placeable or routeable object, measured in the x-y plane.

Table 89—Semantic meaning of PERIMETER

Context	Meaning
CELL	Perimeter of the cell, twice the sum of height and width for rectangular cell.
WIRE	- As a model (TABLE or EQUATION): perimeter the wire itself. - As argument of a model (HEADER): perimeter of the block for which wireload model applies, twice the sum of height and width for rectangular block.
LAYER, VIA, ANTENNA	Perimeter of a placeable or routeable object, measured in the x-y plane.

Table 90—Semantic meaning of DISTANCE

Context	Meaning
RULE	Distance between objects for which the rule applies.

Table 91—Semantic meaning of THICKNESS

Context	Meaning
LAYER, ANTENNA	Distance between top and bottom of a physical object, orthogonal to the x-y plane.

Table 92—Semantic meaning of OVERHANG

Context	Meaning
RULE	Distance between the outer border of an object and the outer border of another object inside the first one.

Table 93—Semantic meaning of EXTENSION

Context	Meaning
LAYER, VIA, RULE, geometric model	Distance between the border of the original object and the border of the same object after enlargement.

11.9 Arithmetic models for timing data

Add lead-in text

11.9.1 Specification of timing models

Timing models shall be specified in the context of a VECTOR statement.

11.9.1.1 Template for timing measurements / constraints

The following templates show a general timing measurement and a general timing constraint description, respectively, applicable for two pins.

```
TEMPLATE TIMING_MEASUREMENT {
    <timeKeyword> = <timeValue> {
       FROM {
          PIN=<fromPin>;
          THRESHOLD=<fromThreshold>;
          EDGE_NUMBER=<fromEdge>;
       }
       TO {
```

```
            PIN=<toPin>;
            THRESHOLD=<toThreshold>;
            EDGE_NUMBER=<toEdge>;
         }
      }
   }
   TEMPLATE TIMING_CONSTRAINT {
      LIMIT {
         <timeKeyword> {
            FROM {
               PIN=<fromPin>;
               THRESHOLD=<fromThreshold>;
               EDGE_NUMBER=<fromEdge>;
            }
            TO {
               PIN=<toPin>;
               THRESHOLD=<toThreshold>;
               EDGE_NUMBER=<toEdge>;
            }
            MIN = <timeValueMin>;
            MAX = <timeValueMax>;
         }
      }
   }
```

For simplicity, trivial arithmetic models shown here. In general, a `HEADER`, `TABLE`, or `EQUATION` construct can be used for calculation of `<timeValue>`, `<timeValueMin>`, or `<timeValueMax>`.

A particular timing constraint does not necessarily contain both `<timeValueMin>` and `<timeValueMax>`.

The `<fromThreshold>` and `<toThreshold>` can be globally predefined as explained in 11.10.3.2.

The `vector_expression` in the context where the `<timeKeyword>` appears shall contain at least two expressions of the type `vector_single_event` with the `<fromPin>` and `<toPin>`, respectively, as operands. The `<fromEdge>` and `<toEdge>` point to their respective `vector_single_event`, as shown in Figure 34.

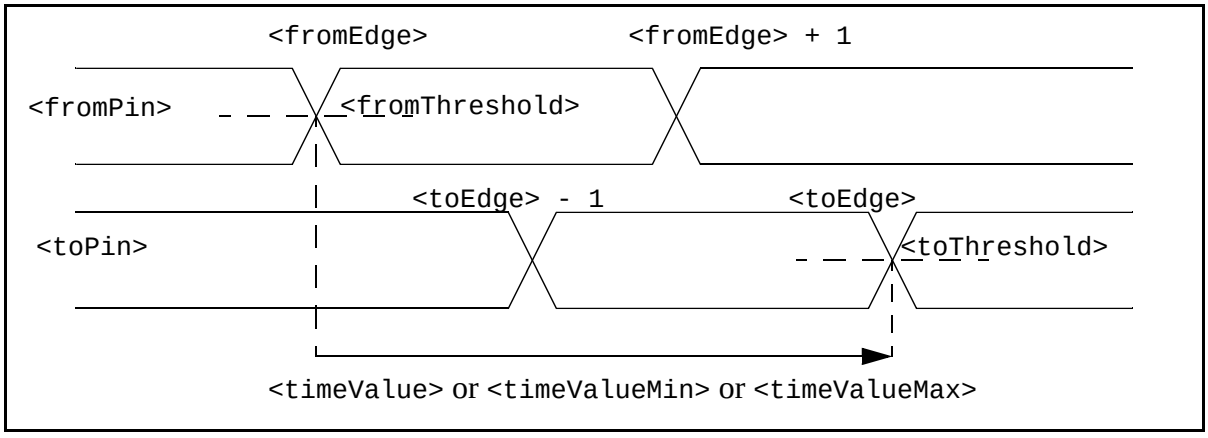


Figure 34—General timing measurement or timing constraint

The direction of the respective transition shall be identified by the respective edge_literal, i.e., the operator of the respective vector_single_event.

The temporal order of the LHS and RHS vector_single_event expressions within the vector_expression is indicated by a vector_binary operator.

The implications on the range of <timeValue> or <refPin> or <timeValueMax> are shown in Table 94.

Table 94—Range of time value depending on VECTOR

LHS	operand	RHS	range of <timeValue> or <timeValueMin> or <timeValueMax>
<fromPin>	-> or ~>	<toPin>	Positive
<toPin>	-> or ~>	<fromPin>	Negative
<fromPin>	&>	<toPin>	Positive or zero
<toPin>	&>	<fromPin>	Negative or zero
<fromPin>	<->	<toPin>	Positive or negative
<toPin>	<->	<fromPin>	Positive or negative
<fromPin>	<&>	<toPin>	Positive or negative or zero
<toPin>	<&>	<fromPin>	Positive or negative or zero

NOTE—This table does not apply for models with CALCULATION=incremental. Incremental values can always be positive, negative, or zero.

11.9.1.2 Partially defined timing measurements and constraints

A partially defined timing measurement or timing constraint contains only a FROM statement or a TO statement, but not both. This construct can be used to specify measurements from any point to a specific point (only TO is specified) or from a specific point to any point (only FROM is specified).

This is summarized in Table 95.

Table 95—Partially specified timing measurements and constraints

DIRECTION of PIN	FROM or TO specified	Specified model applicablity
input	FROM only	Cell timing arcs starting at this pin.
input	TO only	Interconnect timing arcs ending at this pin.
output	FROM only	Interconnect timing arcs starting at this pin.
output	TO only	Cell timing arcs ending at this pin.

It is recommended to use the constructs for interconnect timing arcs only in conjunction with CALCULATION=incremental. The <timeValue>, <timeValueMin>, or <timeValueMax> from this model is added to the <timeValue>, <timeValueMin>, or <timeValueMax> from timing arcs starting or end-

ing at this pin, respectively. If the construct is used with CALCULATION=absolute, the timing model can only be used if completely specified interconnect timing models are not available and the result is not be accurate in general.

11.9.1.3 Template for same-pin timing measurements / constraints

The following templates show a timing measurement and a timing constraint description, respectively, applicable for the same pin.

```
TEMPLATE SAME_PIN_TIMING_MEASUREMENT {
    <timeKeyword> = <timeValue> {
       PIN=<refPin>;
       EDGE_NUMBER=<refEdge>;
       FROM { THRESHOLD=<fromThreshold>; }
       TO { THRESHOLD=<toThreshold>; }
    }
}
TEMPLATE SAME_PIN_TIMING_CONSTRAINT {
    LIMIT {
       <timeKeyword> {
          PIN=<refPin>;
          EDGE_NUMBER=<refEdge>;
          FROM { THRESHOLD=<fromThreshold>; }
          TO { THRESHOLD=<toThreshold>; }
          MIN = <timeValueMin>;
          MAX = <timeValueMax>;
       }
    }
}
```

Depending on the <timeKeyword>, the <timeValue>, <timeValueMin>, or <timeValueMax> is measured on the same <refEdge> or between <refEdge> and <refEdge> *plus 1*. Only the -> or ~> operators are applicable between subsequent edges. Therefore, the <timeValue>, <timeValueMin>, or <timeValueMax> are positive by definition.

NOTE—The <fromThreshold> and <toThreshold> can be globally predefined as explained in 11.10.3.2. However, the THRESHOLD in the context of a PIN does not apply for *SAME_PIN_TIMING_MEASUREMENT* or *SAME_PIN_TIMING_CONSTRAINT*, since the <refPin> is not within a FROM or TO statement.

11.9.1.4 Absolute and incremental evaluation of timing models

As mentioned in the previous sections, the calculation models for *TIMING_MEASUREMENT*, *TIMING_CONSTRAINT*, *SAME_PIN_TIMING_MEASUREMENT*, and *SAME_PIN_TIMING_CONSTRAINT* can have the annotation CALCULATION=absolute (the default) or CALCULATION=incremental. These annotations are only relevant more than one calculation model for the same timing arc exists.

Calculation models for the same timing arc with CALCULATION=absolute shall be within the context of mutually exclusive VECTORs. The vector_expression specifies which model to use under which condition.

Example

```
VECTOR ( (01 A -> 01 Z) && B & !C ) {
    DELAY { CALCULATION=absolute; FROM { PIN=A; } TO { PIN=Z; }
```

```
        /* fill in HEADER, TABLE */ }
    }
    VECTOR ( (01 A -> 01 Z) && !B & C ) {
        DELAY { CALCULATION=absolute; FROM { PIN=A; } TO { PIN=Z; }
        /* fill in HEADER, TABLE */ }
    }
```

The vectors `( (01 A -> 01 Z) && B & !C )` and `( (01 A -> 01 Z) && !B & C )` are mutually exclusive. They describe the same timing arc with two mutually exclusive conditions.

In the case of a `VECTOR` containing a calculation model for a timing arc with `CALCULATION=incremental`, there shall be another `VECTOR` with a calculation model for the same timing arc with `CALCULATION=absolute` and both vectors shall be compatible. The `vector_expression` of the latter shall necessarily be true when the `vector_expression` of the former is true.

Example

```
VECTOR (01 A -> 01 Z) {
    DELAY { CALCULATION=absolute; FROM { PIN=A; } TO { PIN=Z; }
    /* fill in HEADER, TABLE */ }
}
VECTOR ( (01 A -> 01 Z) && B & !C ) {
    DELAY { CALCULATION=incremental; FROM { PIN=A; } TO { PIN=Z; }
    /* fill in HEADER, TABLE */ }
}
VECTOR ( (01 A -> 01 Z) && !B & C ) {
    DELAY { CALCULATION=incremental; FROM { PIN=A; } TO { PIN=Z; }
    /* fill in HEADER, TABLE */ }
}
```

The vectors `( (01 A -> 01 Z) && B & !C )` and `( (01 A -> 01 Z) && !B & C )` are both compatible with the vector `(01 A -> 01 Z)` and mutually exclusive with each other. The latter describe the same timing arc with two mutually exclusive conditions. The former describes the same timing arc without conditions. This modeling style is useful for timing analysis tools with or without support for conditions. The vectors with conditions, if supported, add accuracy to the calculation. However, the vector without conditions is always available for basic calculation.

11.9.1.5 PIN-related timing models

SAME_PIN_TIMING_MEASUREMENT and *SAME_PIN_TIMING_CONSTRAINT* (see 11.9.1 and 11.12.1.4) are pin-related timing models. They are defined with reference to the externally accessible node.

11.9.2 TIME statement

Add lead-in text

11.9.2.1 TIME

The `<timeKeyword>` `TIME` describes a general *TIMING_MEASUREMENT* or *TIMING_CONSTRAINT* without implying any particular relationship between `<fromEdge>` and `<toEdge>`.

In general, `<fromPin>` and `<toPin>` refer to two different pins. However, it is legal for `<fromPin>` and `<toPin>` to refer to the same pin.

The default value for <fromEdge> and <toEdge> shall be 0.

11.9.2.2 TIME within the LIMIT construct

Within a LIMIT construct, TIME can be used in the following ways:

— TIME itself is subjected to a LIMIT (see 11.12.11.2)
— TIME is the argument of a model subjected to a LIMIT

When TIME is used as argument of a model within the LIMIT construct, it shall mean the amount of time during which the device is exposed to the quantity modeled within the LIMIT construct. This amount of time is also called a *lifetime*.

Example

```
LIMIT {
    CURRENT {
       PIN = my_pin;
       MEASUREMENT = static;
       MAX {
          HEADER { TIME TEMPERATURE }
          EQUATION { 6.5*EXP(-10/(TEMPERATURE+273))*TIME**(-0.3) }
       }
    }
}
```

The limit for maximum current depends on the temperature and the expected lifetime of the device.

11.9.2.3 TIME to peak measurement

For a model in the context of a VECTOR, with a peak measurement, the TIME annotation shall define the time between a reference event within the vector_expression and the instant when the peak value occurs.

For that purpose, either the FROM or the TO statement shall be used in the context of the TIME annotation, containing a PIN annotation and, if necessary, a THRESHOLD and/or an EDGE_NUMBER annotation.

If the FROM statement is used, the start point shall be the reference event and the end point shall be the occurrence time of the peak, as shown in Figure 35.

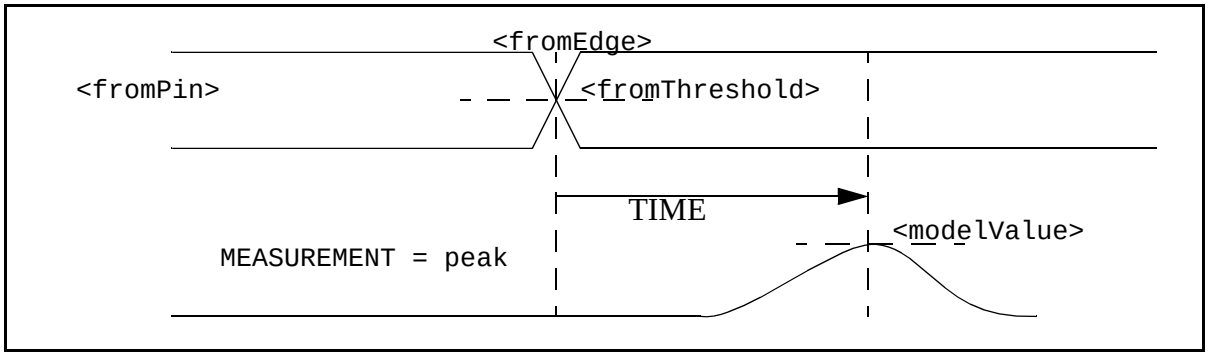


Figure 35—Illustration of time to peak using FROM statement

If the TO statement is used, the start point shall be the occurrence time of the peak and the end point shall be the reference event, as shown in Figure 36.

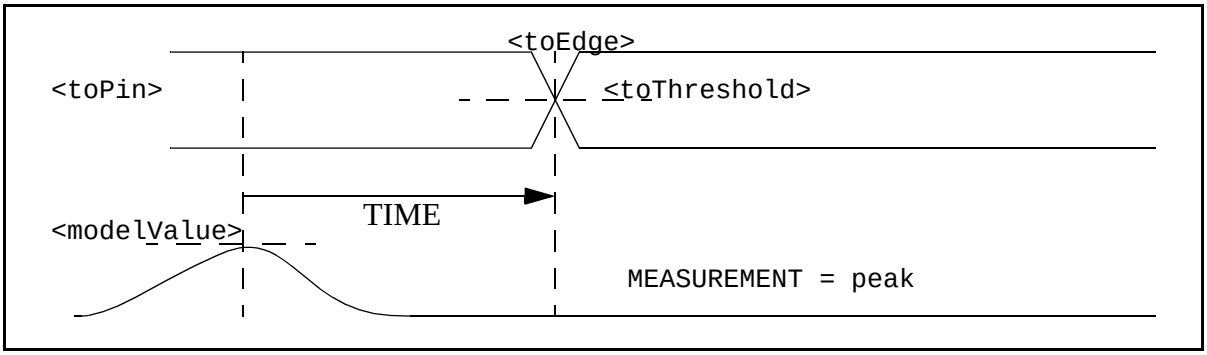


Figure 36—Illustration of time to peak using TO statement

Example

```
VECTOR (01 A -> 01 B -> 10 B) {
    CURRENT peak1 = 10.8 {
       PIN = Vdd;
       MEASUREMENT = peak;
       TIME = 3.0 { UNIT=ns; FROM { PIN=A; EDGE_NUMBER=0; } }
    }
    CURRENT peak2 = 12.3 {
       PIN = Vdd;
       MEASUREMENT = peak;
       TIME = 2.0 { UNIT=ns; TO { PIN=B; EDGE_NUMBER=1; } }
    }
}
```

Here, the peak with magnitude 10.8 occurs 3 nanoseconds after the event 01 A.

The peak with magnitude 12.3 occurs 2 nanoseconds before the event 10 B.

11.9.2.4 Waveform description

This section specifies waveform descriptions.

11.9.2.4.1 Principles

In order to describe an arithmetic model representing a waveform, TIME shall be an argument in the HEADER. Other arguments can appear in the HEADER as well. The model can be described as a TABLE or EQUATION.

Example for TABLE

```
VOLTAGE {
    HEADER {
       TIME {
          UNIT = ns;
          INTERPOLATION=linear;
          TABLE { 0.0  1.0  1.5  2.0  3.0 }
```

```
        }
    }
    TABLE { 0.0  0.0  5.0  0.0  0.0 }
}
```

Example for EQUATION

```
VOLTAGE {
    HEADER {
       TIME { UNIT = ns; }
    }
    EQUATION {
       (TIME < 1.0) ? 0 :
       (TIME < 1.5) ? 5.0*(TIME - 1.0) :
       (TIME < 2.0) ? 5.0*(2.0 - TIME) :
       0.0
    }
}
```

Both models describe the same piece-wise linear waveform, as shown in Figure 37.

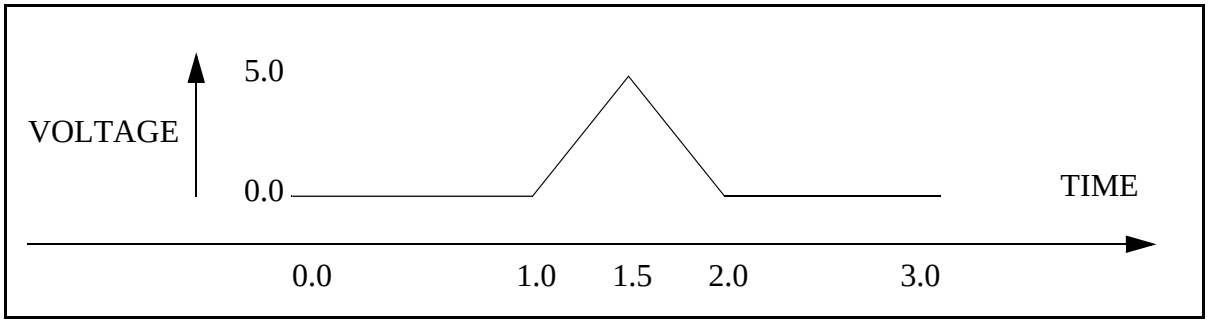


Figure 37—Illustration of a piece-wise linear waveform

If the model is within the context of a VECTOR, either the FROM or the TO statement can be used in the context of TIME, pointing to a reference event which occurs at TIME = 0 relative to the waveform description. See xxx for the definition of start and end points of measurements.

Example

```
VECTOR (01 A -> 01 B -> 10 B) {
    VOLTAGE {
       HEADER {
          TIME {
             FROM { PIN = B; EDGE_NUMBER = 1; }
             TABLE { 0.0  1.0  1.5  2.0  3.0 }
// alternative description:
//           TO { PIN = B; EDGE_NUMBER = 1; }
//           TABLE { -3.0  -2.0  -1.5  -1.0  0.0 }
          }
       }
       TABLE { 0.0  0.0  5.0  0.0  0.0 }
```

```
        }
    }
```

NOTE—Use the `FROM` statement. If the `TO` statement is used, `TIME` is measured backwards, which is counter-intuitive. For dynamic analysis, use the last event in the `vector_expression` as the reference. Otherwise, the analysis tool remembers the occurrence time of previous events in order to place the waveform into the context of absolute time.

11.9.2.4.2 Annotations within a waveform

The `MEASUREMENT` annotation `transient` shall apply as a default for waveforms.

The `FREQUENCY` annotation can be used to specify a repetition frequency of the waveform. The following boundary restrictions are imposed in order to make the waveform repeatable:

- The initial value and the final value of waveform shall be the same.
- The extrapolation beyond the initial and the final value of the waveform shall yield the same result. Thus, the first, second, last, and second-to-last point of the waveform shall be the same.
- The time window between the first and the last measurement shall be smaller or equal to `1 / FREQUENCY`.

This is illustrated in Figure 38.

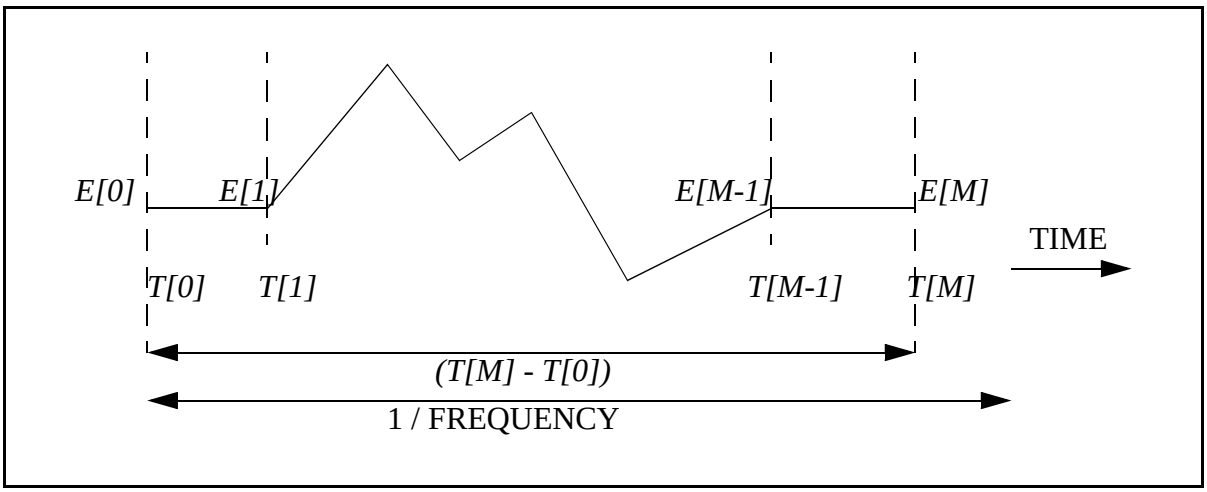


Figure 38—TIME and FREQUENCY in a waveform

11.9.3 FREQUENCY statement

Add lead-in text

11.9.3.1 FREQUENCY within a LIMIT construct

Within a `LIMIT` construct, `FREQUENCY` can be used in the following ways:

- `FREQUENCY` itself is subjected to a `LIMIT`
- `FREQUENCY` is the argument of a model subjected to a `LIMIT`

FREQUENCY can be subjected to a LIMIT within the context of a VECTOR. The LIMIT construct specifies an upper and/or lower limit for the repetition frequency of the event sequence described by the vector_expression.

Example

```
VECTOR ( 01 A -> 01 Z ) {
    LIMIT {
       FREQUENCY {
          MAX {
             HEADER {
                SLEWRATE { PIN = A; TABLE { 0.1 0.5 1.0 5.0 } }
                CAPACITANCE { PIN = Z; TABLE { 0.1 0.4 1.6 } }
             }
             TABLE {
                200 190 180 120
                150 150 145 130
                 80  80  80  70
             }
          }
       }
    }
}
```

The maximum allowed switching frequency for a rising edge on A, followed by a rising edge on Z, depends on the slewrate on A and the load capacitance on Z.

A LIMIT for a quantity with MEASUREMENT annotation average, rms, or peak can be frequency-dependent. The FREQUENCY specifies the repetition frequency for the measurement.

Example

```
LIMIT {
   CURRENT {
      PIN = Vdd;
      MEASUREMENT = average;
       MAX {
          HEADER { FREQUENCY TIME TEMPERATURE }
          EQUATION {
             (FREQUENCY<1)? 6.5*EXP(-10/(TEMPERATURE+273))*TIME**(-0.3) :
              7.8*EXP(-9/(TEMPERATURE+273))*TIME**(-0.2) :
          }
       }
    }
}
```

The limit for average current is specified for low frequencies (< 1MHz) and for higher frequencies. In both cases, the limit depends on temperature and lifetime.

11.9.3.2 TIME and FREQUENCY annotation

Arithmetic models with certain values of MEASUREMENT annotation can also have *either* TIME *or* FREQUENCY as annotations.

The semantics are defined in Table 96.

Table 96—Semantic interpretation of MEASUREMENT, TIME, or FREQUENCY annotation

MEASUREMENT annotation	Semantic meaning of TIME annotation	Semantic meaning of FREQUENCY annotation
`transient`	Integration of analog measurement is done during that time window.	Integration of analog measurement is repeated with that frequency.
`static`	N/A	N/A
`average`	Average value is measured over that time window.	Average value measurement is repeated with that frequency.
`rms`	Root-mean-square value is measured over that time window.	Root-mean-square measurement is repeated with that frequency.
`peak`	Peak value occurs at that time (only within context of `VECTOR`).	Observation of peak value is repeated with that frequency.

In the case of `average` and `rms`, the interpretation `FREQUENCY = 1 / TIME` is valid. Either one of these annotations shall be mandatory. The values for `average` measurements and for `rms` measurements scale linearly with `FREQUENCY` and `1 / TIME`, respectively.

In the case of `transient` and `peak`, the interpretation `FREQUENCY = 1 / TIME` is not valid. Either one of these annotations shall be optional. The values do not necessarily scale with `TIME` or `FREQUENCY`. The `TIME` or `FREQUENCY` annotations for `transient` measurements are purely informational.

11.9.4 DELAY and RETAIN statements

Add lead-in text

11.9.4.1 DELAY

The `<timeKeyword> DELAY` describes a *`TIMING_MEASUREMENT`* implying a causal relationship between `<fromEdge>` and `<toEdge>`.

Usually, `<fromPin>` refers to an input pin and `<toPin>` refers to an output pin. However, it is legal for `<fromPin>` and `<toPin>` to refer to an output pin.

The default value for `<fromEdge>` and `<toEdge>` shall be `0`, unless the `DELAY` statement appears in conjunction with a `RETAIN` statement within the context of the same `VECTOR`.

11.9.4.2 RETAIN

The `<timeKeyword> RETAIN` describes a *`TIMING_MEASUREMENT`* implying a causal relationship between `<fromEdge>` and `<toEdge>` in the same way as `DELAY`.

`RETAIN` is used to describe the elapsed time until the output changes its old value, whereas `DELAY` is used to describe the elapsed time until the output settles to a stable new value, as shown in Figure 39.

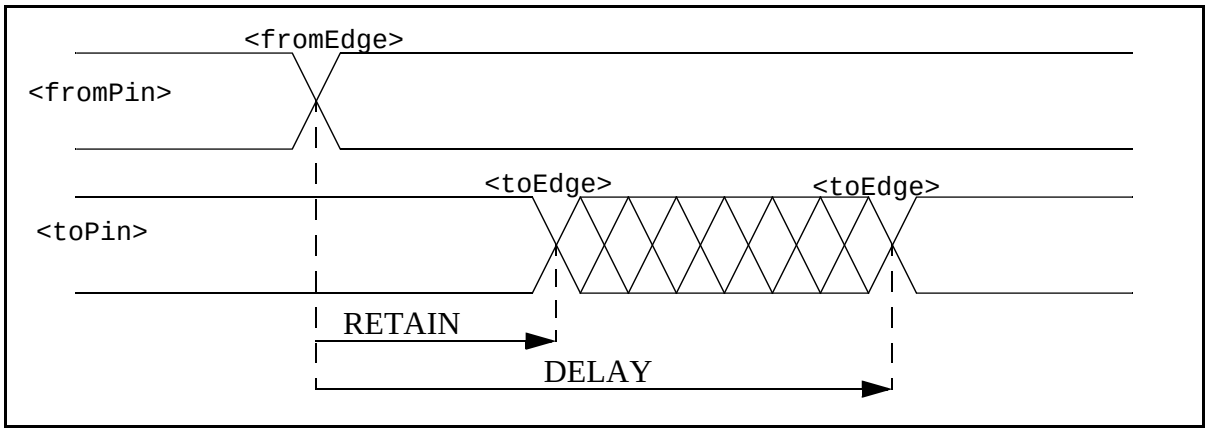


Figure 39—RETAIN and DELAY

When DELAY appears in conjunction with RETAIN, the <fromEdge> for both measurements shall be the same. The <toEdge> for DELAY shall be the <toEdge> for RETAIN *plus 1*.

The default value for <fromEdge> and <toEdge> for RETAIN shall be 0. The default value for <toEdge> for DELAY shall be 1.

11.9.5 SLEWRATE statement

The <timeKeyword> SLEWRATE describes a *SAME_PIN_TIMING_MEASUREMENT* for <timeValue> defining the duration of a signal transition or a fraction thereof.

The SLEWRATE applies for the <refEdge> on the <refPin>. The default value for <refEdge> shall be 0.

11.9.6 SETUP and HOLD statement

Add lead-in text

11.9.6.1 SETUP

The <timeKeyword> SETUP describes a *TIMING_CONSTRAINT* for <timeValueMin> defining the minimum stable time required for the data signal on the <fromPin> before it is sampled by the strobe signal on the <toPin>.

The <fromPin> usually is an input pin with SIGNALTYPE=data. The <toPin> is an input pin with SIGNALTYPE=clock.

The default value for <fromEdge> and <toEdge> for SETUP shall be 0.

11.9.6.2 HOLD

The <timeKeyword> HOLD describes a *TIMING_CONSTRAINT* for <timeValueMin> defining the minimum stable time required for the data signal on the <toPin> after it is sampled by the strobe signal on the <fromPin>.

The <toPin> usually is an input pin with SIGNALTYPE=data. The <fromPin> is an input pin with SIGNALTYPE=clock.

The default value for <fromEdge> shall be 0. The default value for <toEdge> shall be 0, unless HOLD appears in conjunction with SETUP in the context of the same VECTOR. In that case, the default value for <toEdge> shall be 1. All of this is depicted in Figure 40.

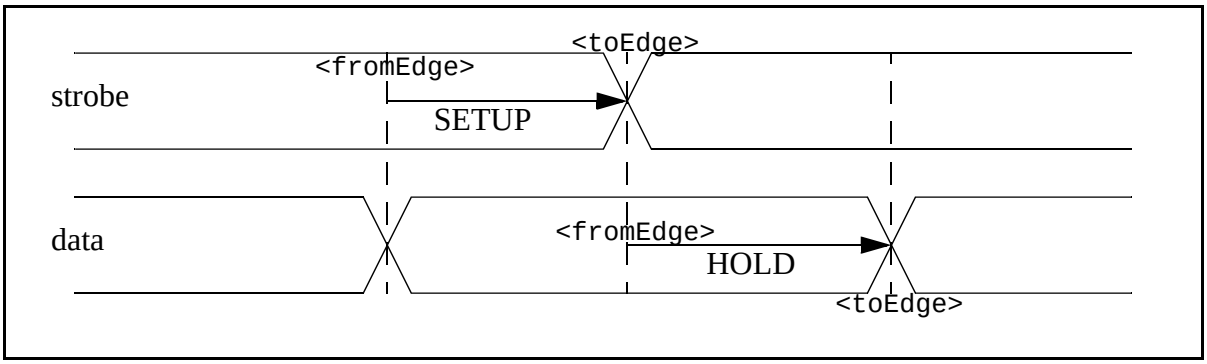


Figure 40—SETUP and HOLD

The <timeValueMin> for SETUP or the <timeValueMin> for HOLD with respect to the same strobe can be negative. However, the sum of both values shall be positive. The sum represents the minimum duration of a valid data signal around a strobe signal.

11.9.7 NOCHANGE statement

The <timeKeyword> NOCHANGE describes a *SAME_PIN_TIMING_CONSTRAINT* defining the requirement for a stable signal on a pin subjected to SETUP and HOLD on subsequent edges of a strobe signal., as shown in Figure 41.

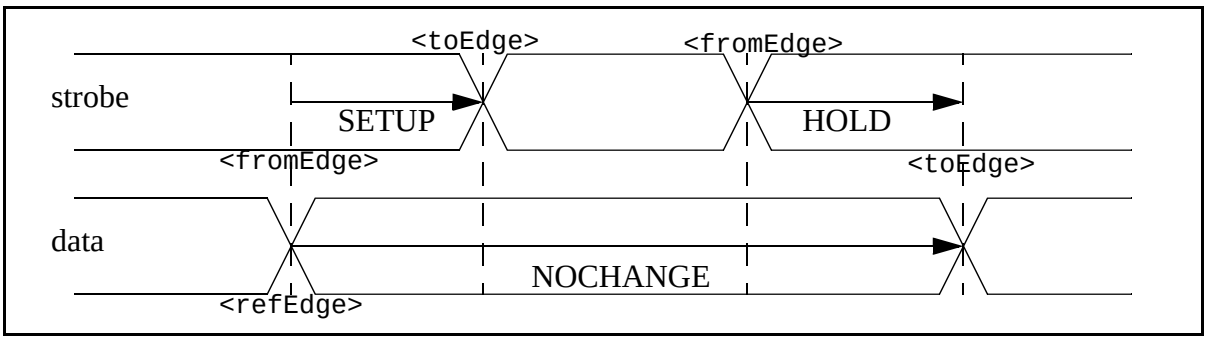


Figure 41—NOCHANGE, SETUP, and HOLD

The NOCHANGE applies between the <refEdge> and the subsequent edge, i.e., <refEdge> *plus 1* on the <refPin>. The default value for <refEdge> shall be 0.

When NOCHANGE appears in conjunction with SETUP and HOLD within the context of the same VECTOR, the default value for <fromEdge> and <toEdge> of SETUP shall be 0 and the default value for <fromEdge> and <toEdge> of HOLD shall be 1.

11.9.8 RECOVERY and REMOVAL statements

Add lead-in text

11.9.8.1 RECOVERY

The <timeKeyword> RECOVERY describes a *TIMING_CONSTRAINT* for <timeValueMin> defining the minimum stable time required for an asynchronous control signal on the <fromPin> to be inactive before a strobe signal on the <toPin> can be active.

The <fromPin> usually is an input pin with SIGNALTYPE=set|clear. The <toPin> is an input pin with SIGNALTYPE=clock.

The default value for <fromEdge> and <toEdge> for RECOVERY shall be 0.

11.9.8.2 REMOVAL

The <timeKeyword> REMOVAL describes a *TIMING_CONSTRAINT* for <timeValueMin> defining the minimum stable time required for an asynchronous control signal on the <toPin> to remain active after overriding a strobe signal on the <fromPin>.

The <toPin> usually is an input pin with SIGNALTYPE=set|clear. The <fromPin> is an input pin with SIGNALTYPE=clock.

The default value for <fromEdge> and <toEdge> for REMOVAL shall be 0.

REMOVAL can appear in conjunction with RECOVERY within the context of the same VECTOR, as shown in Figure 42.

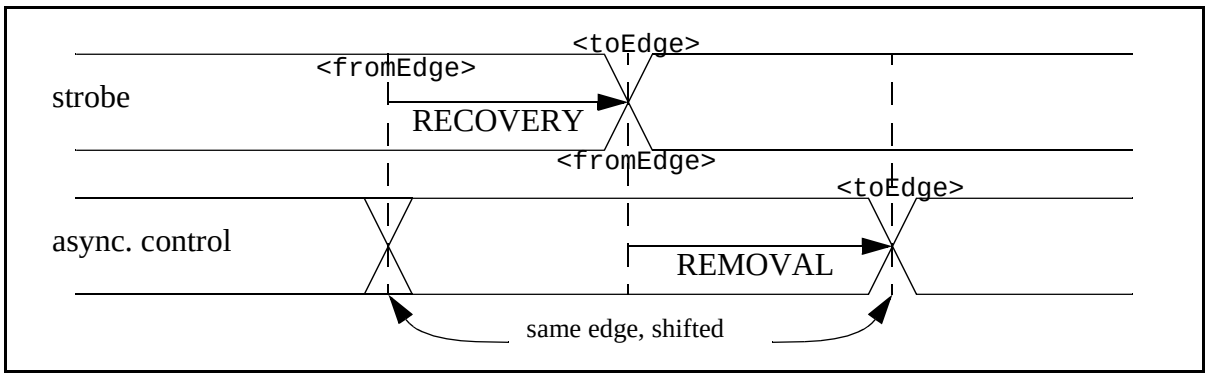


Figure 42—RECOVERY and REMOVAL

The <timeValueMin> for RECOVERY or the <timeValueMin> for REMOVAL with respect to the same strobe can be negative. However, the sum of both values shall be positive. The sum represents the time window around the clock signal when the asynchronous control signal shall not switch.

11.9.9 SKEW statement

Add lead-in text

11.9.9.1 SKEW between two signals

The <timeKeyword> SKEW describes a *TIMING_CONSTRAINT* for <timeValueMax> defining the maximum allowed time separation between <fromEdge> on <fromPin> and <toEdge> on <toPin>.

The default value for <fromEdge> and <toEdge> for SKEW shall be 0.

11.9.9.2 SKEW between multiple signals

SKEW can also describe the maximum time distortion between signals on multiple pins. In this case, a list of pins appears in form of a multi-value annotation. No FROM or TO containers can be used here.

Example

```
SKEW {
    PIN { <pinList> }
    EDGE_NUMBER { <edgeList> }
    <skewData>
}
```

The default for EDGE_NUMBER in SKEW for multiple signals shall be a list of 0s.

A special case of multiple pins is a single bus. In this case, the unnamed_assignment syntax is also valid as alternative to the multi_value_assignment syntax (see Section 8.15.3).

Example

```
SKEW { PIN = my_bus_pin[8:1]; }
```

or

```
SKEW { PIN { my_bus_pin[8:1] } }
```

11.9.10 PULSEWIDTH statement

The <timeKeyword> PULSEWIDTH describes a *SAME_PIN_TIMING_CONSTRAINT* for <timeValueMin> defining the minimum duration of the signal before changing state.

The PULSEWIDTH statement is applicable for both input and output pins. In the case of an input pin, it represents a timing check against the minimum duration. In case of an output pin, it represents the minimum possible duration of the signal.

The PULSEWIDTH applies between the <refEdge> and the subsequent edge, i.e., <refEdge> *plus 1* on the <refPin>. The default value for <refEdge> shall be 0.

11.9.11 PERIOD statement

The <timeKeyword> PERIOD describes a *SAME_PIN_TIMING_CONSTRAINT* for <timeValueMin> defining the minimum time between subsequent repetitions of a signal. Because of periodicity, <fromThreshold> and <toThreshold> are not required. Therefore, FROM and TO statements do not appear.

If the VECTOR describes a completely specified event sequence, <refPin> and <refEdge> are not required. PERIOD applies for the complete event sequence. If the VECTOR describes a partially specified event sequence, involving the ~> operator, <refPin> and <refEdge> are required.

11.9.12 JITTER statement

The <timeKeyword> JITTER describes a *SAME_PIN_TIMING_MEASUREMENT* for <timeValue> defining the actual uncertainty of arrival time for a periodical signal at a pin.

The JITTER applies for the <refEdge> on the <refPin>. The default value for <refEdge> shall be 0. Threshold definitions, i.e., <fromThreshold> or <toThreshold> do not apply.

A limit for tolerable jitter at a pin can be expressed using the LIMIT construct, as shown in the template for *SAME_PIN_TIMING_CONSTRAINT*.

11.9.13 THRESHOLD statement

Add lead-in text

11.9.13.1 THRESHOLD definition

The THRESHOLD represents a reference voltage level for timing measurements, normalized to the signal voltage swing and measured with respect to the logic 0 voltage level, as shown in Figure 43.

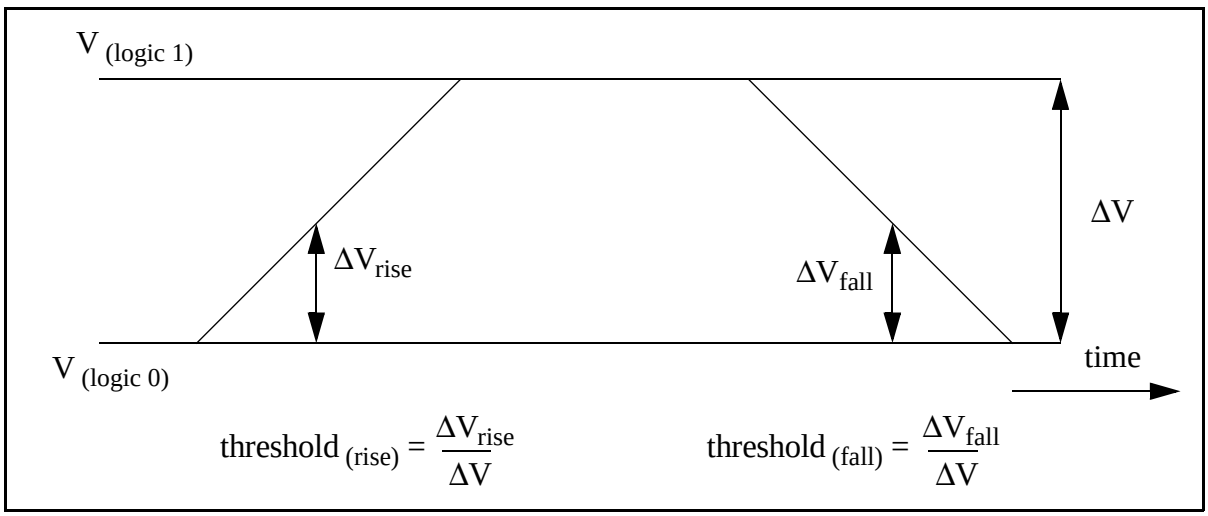


Figure 43—THRESHOLD measurement definition

The voltage levels for logic 1 and 0 represent a full voltage swing.

Different threshold data for RISE and FALL can be specified or else the data shall apply for both rising and falling transitions.

The THRESHOLD statement has the form of an arithmetic model. If the submodel keywords RISE and FALL are used, it has the form of an arithmetic model container.

Examples

```
THRESHOLD = 0.4;
THRESHOLD { RISE = 0.3;  FALL = 0.5; }
THRESHOLD { HEADER { TEMPERATURE {TABLE{ 0 50 100 }}}
            TABLE { 0.5 0.4 0.3}}
```

11.9.13.2 Context of THRESHOLD definitions

The THRESHOLD statement can appear in the context of a FROM or TO container. In this case, it specifies the applicable reference for the start and end point of the timing measurement, respectively.

Example

```
SLEWRATE {
    FROM { THRESHOLD = 0.2; }
    TO { THRESHOLD = 0.8; }
}
```

The THRESHOLD statement can also appear in the context of a PIN. In this case, it specifies the applicable reference for the start or end point of timing measurements indicated by the PIN annotation inside a FROM or TO container, unless a THRESHOLD is specified explicitly inside the FROM or TO container.

If both the RISE and FALL thresholds are specified and the switching direction of the applicable pin is clearly indicated in the context of a VECTOR, the RISE or FALL data shall be applied accordingly.

Example

```
PIN A { THRESHOLD { RISE = 0.3;  FALL = 0.5; } }
PIN Z { THRESHOLD = 0.4; }
// other statements ...
VECTOR ( 01 A -> 10 Z ) {
    DELAY { FROM { PIN=A; } TO { PIN=Z; } }
// the applicable threshold for A is 0.3
// the applicable threshold for Z is 0.4
```

If thresholds are needed for exact definition of the model data, the FROM and TO containers shall each contain an arithmetic model for THRESHOLD.

A THRESHOLD statement can also appear as argument of an arithmetic model for timing measurements. In this case, it shall contain a PIN annotation matching another PIN annotation in the FROM or TO container.

Example

```
DELAY {
    FROM { PIN = A; THRESHOLD = 0.5; }
    TO { PIN = Z; }
    HEADER { THRESHOLD { PIN = Z; TABLE { 0.3 0.4 0.5 } }
    TABLE { 1.23 1.45 1.78 }
}
/* The measurement reference for pin A is always 0.5. The delay from A to
Z is expressed as a function of the measurement reference for pin Z. */
```

FROM and TO containers with THRESHOLD definitions, yet without PIN annotations, can appear within unnamed timing model definitions in the context of a VECTOR, CELL, WIRE, SUBLIBRARY, or LIBRARY object for the purpose of specifying global threshold definitions for all timing models within scope of the definition. The following priorities apply:

a) THRESHOLD in the HEADER of the timing model
b) THRESHOLD in the FROM or TO statement within the timing model
c) THRESHOLD for timing model definition in the context of the same VECTOR
d) THRESHOLD within the PIN definition
e) THRESHOLD for timing model definition in the context of the same CELL or WIRE
f) THRESHOLD for timing model definition in the context of the same SUBLIBRARY
g) THRESHOLD for timing model definition in the context of the same LIBRARY
h) THRESHOLD for timing model definition outside LIBRARY

Example

```
LIBRARY my_library {
    DELAY {
       FROM { THRESHOLD = 0.4;  }
       TO   { THRESHOLD = 0.4;  }
    }
    SLEWRATE {
       FROM { THRESHOLD { RISE = 0.2; FALL = 0.8; } }
       TO   { THRESHOLD { RISE = 0.8; FALL = 0.2; } }
    }
    CELL my_cell {
       PIN A { DIRECTION=input; THRESHOLD { RISE = 0.3; FALL = 0.5; } }
       PIN Z { DIRECTION=output; }
       VECTOR (01 A -> 10 Z) {
          DELAY { FROM { PIN=A; } TO { PIN=Z; } }
          SLEWRATE { PIN = Z; }
       }
    }
}
// delay is measured from A (threshold=0.3) to Z (threshold=0.4)
// slewrate on Z is measured from threshold=0.8 to threshold=0.2.
```

11.10 Auxiliary statements related to timing data

Add lead-in text

11.10.1 FROM and TO statements

A FROM container and a TO container shall be used inside timing measurements and timing constraints. Depending on the semantics of the timing model (see 11.9.1), they can contain a THRESHOLD statement, PIN annotation, and/or EDGE_NUMBER annotation, as shown in Syntax 113.

from ::=
FROM { from_to_items **}**
to ::=
TO { from_to_items **}**
from_to_items ::=
from_to_item { from_to_item }
from_to_item ::=
*PIN*_single_value_annotation
| *EDGE*_single_value_annotation
| *THRESHOLD*_arithmetic_model

Syntax 113—FROM and TO statements

The data in the FROM and TO containers define the measurement start and end point, respectively.

Example

```
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 the rising signal and `0.6` for the falling signal. The threshold for `data_out` is `0.5`, which applies for both the rising and falling signals.

11.10.2 EARLY and LATE statements

The `EARLY` and `LATE` containers define the boundaries of timing measurements in one single analysis, as shown in Syntax 114. They only apply to `DELAY` and `SLEWRATE`. Both of them need to appear in both containers.

*EARLY*_arithmetic_model_container ::=
EARLY { *early_late*_arithmetic_models **}**
*LATE*_arithmetic_model_container ::=
LATE { *early_late*_arithmetic_models **}**
*early_late*_arithmetic_models ::=
*early_late*_arithmetic_model { *early_late*_arithmetic_model }
*early_late*_arithmetic_model ::=
*DELAY*_arithmetic_model
| *RETAIN*_arithmetic_model
| *SLEWRATE*_arithmetic_model

Syntax 114—EARLY and LATE statements

The quadruple

```
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 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 can be larger than the `SLEWRATE` of the `LATE` signal.

11.10.3 Annotations for arithmetic models for timing data

~~Auxiliary statements for timing models~~

This section details the auxiliary statements used for timing modeling.

11.10.3.1 PIN annotation

If the timing measurements or timing constraints, respectively, apply semantically for two pins (see 11.9.1.1), the `FROM` and `TO` containers shall each contain the `PIN` annotation.

Example

```
DELAY {
    FROM { PIN = A ; }
    TO { PIN = Z ; }
}
```

Otherwise, if the timing measurements or timing constraints apply semantically only to one pin (see 11.9.1.3), the PIN annotation shall be outside the FROM or TO container.

Example

```
SLEWRATE {
    PIN = A ;
}
```

11.10.3.2 EDGE_NUMBER annotation

The EDGE_NUMBER annotation within the context of a timing model shall specify the edge where the timing measurement applies. The timing model shall be in the context of a VECTOR. The EDGE_NUMBER shall have an unsigned value pointing to exactly one of subsequent vector_single_event expressions applicable to the referenced pin. The EDGE_NUMBER shall be counted individually for each pin which appears in the VECTOR, starting with zero (0).

If the timing measurements or timing constraints, apply semantically to two pins (see 11.9.1.1), the EDGE_NUMBER annotation shall be legal inside the FROM or TO container in conjunction with the PIN annotation.

Example

```
DELAY {
    FROM { PIN = A ; EDGE_NUMBER = 0; }
    TO { PIN = Z ; EDGE_NUMBER = 0; }
}
```

Otherwise, if the timing measurements or timing constraints apply semantically only to one pin (see 11.9.1.3), the EDGE_NUMBER annotation shall be legal outside the FROM or TO container in conjunction with the PIN annotation.

Example

```
SLEWRATE {
    PIN = A ;  EDGE_NUMBER = 0;
}
```

The default values for EDGE_NUMBER are specific for each timing model keyword (see 11.9.1).

The EDGE_NUMBER annotation is necessary for complex timing models involving multiple transitions on the same pin, as illustrated by the Figure 44 — Figure 46 and their examples.

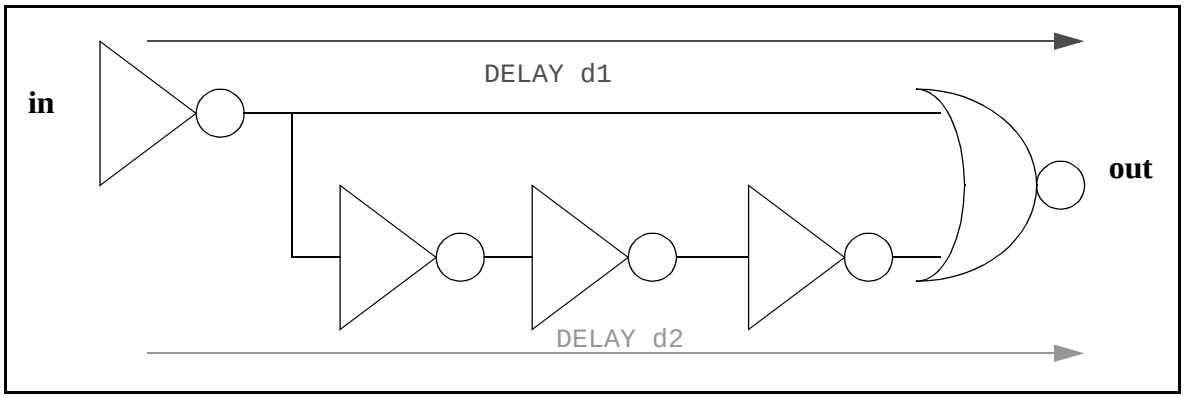


Figure 44—Schematic of a pulse generator

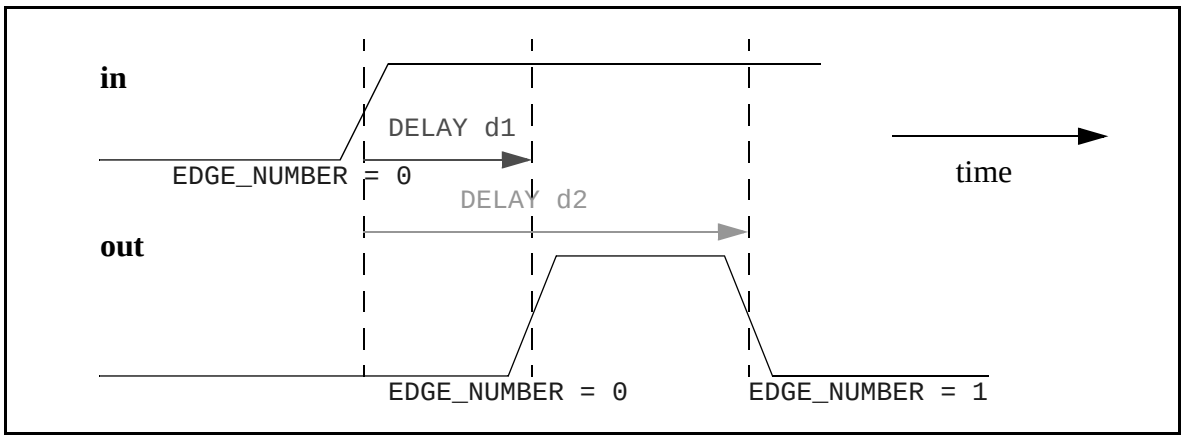


Figure 45—Timing diagram of a pulse generator

```
VECTOR ( 01 in -> 01 out -> 10 out ) {
    DELAY d1 {
       FROM { PIN = in; }
       TO { PIN = out; EDGE_NUMBER = 0; }
    }
    DELAY d2 {
       FROM { PIN = in; }
       TO { PIN = out; EDGE_NUMBER = 1; }
    }
}
```

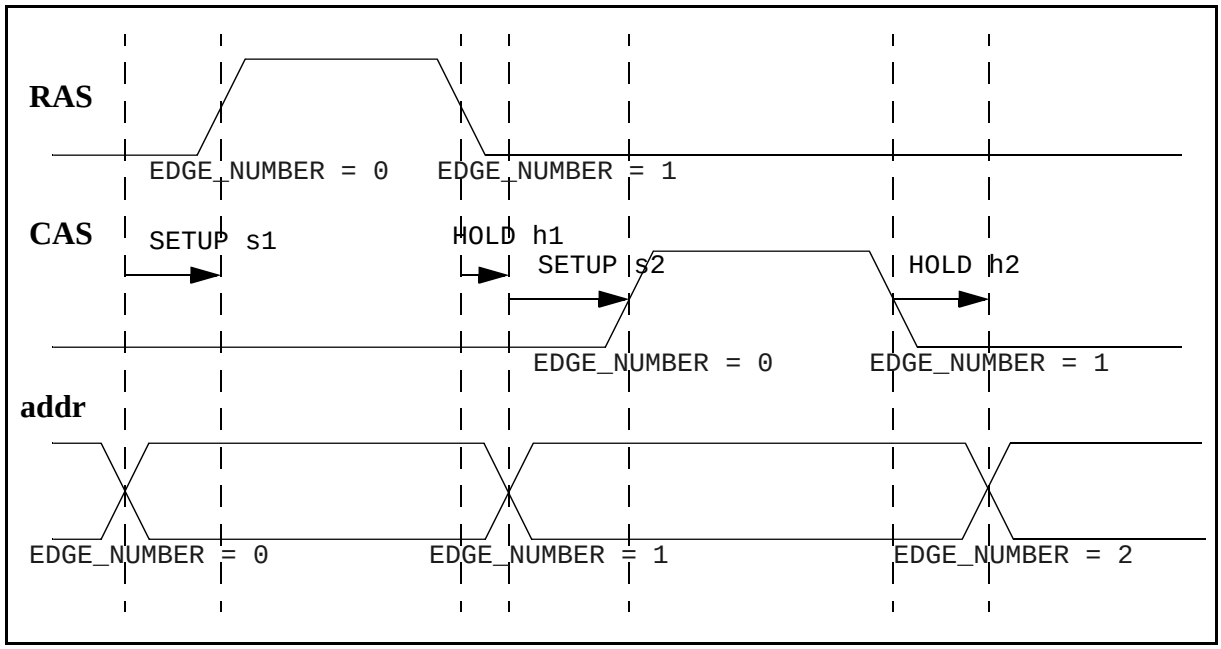


Figure 46—Timing diagram of a DRAM cycle

```
VECTOR(?! addr ->01 RAS ->10 RAS ->?! addr ->01 CAS ->10 CAS ->?! addr){
    SETUP s1 {
       FROM { PIN = addr; EDGE_NUMBER = 0; }
       TO { PIN = RAS; EDGE_NUMBER = 0; }
    }
    HOLD h1 {
       FROM { PIN = RAS; EDGE_NUMBER = 1; }
       TO { PIN = addr; EDGE_NUMBER = 1; }
    }
    SETUP s2 {
       FROM { PIN = addr; EDGE_NUMBER = 1; }
       TO { PIN = CAS; EDGE_NUMBER = 0; }
    }
    HOLD h2 {
       FROM { PIN = CAS; EDGE_NUMBER = 1; }
       TO { PIN = addr; EDGE_NUMBER = 2; }
    }
}
```

11.11 Arithmetic models for environmental data

~~Environmental dependency for electrical data~~

This section defines the environmental dependencies for electrical data.

11.11.1 PROCESS and DERATE_CASE statement

Add lead-in text

11.11.1.1 PROCESS

The following identifiers can be used as predefined process corners:

`?n?p` process definition with transistor strength

where `?` can be

`s` strong
`w` weak

The possible process name combinations are shown in Table 97.

Table 97—Predefined process names

Process name	Description
`snsp`	Strong NMOS, strong PMOS.
`snwp`	Strong NMOS, weak PMOS.
`wnsp`	Weak NMOS, strong PMOS.
`wnwp`	Weak NMOS, weak PMOS.

11.11.1.2 DERATE_CASE

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 98.

Table 98—Predefined derating cases

Derating case	Description
`bccom`	Best case commercial.
`bcind`	Best case industrial.
`bcmil`	Best case military.
`wccom`	Worst case commercial.
`wcind`	Worst case military.

Table 98—Predefined derating cases (Continued)

Derating case	Description
wcmil	Worst case military.

11.11.1.3 Lookup table without interpolation

The PROCESS or DERATE_CASE can be used in a TABLE within the HEADER of an arithmetic model for electrical data, e.g., DELAY. Data can not be interpolated in the dimension of this table.

Example

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

Here , the DELAY is 0.4 ns for nominal process, 0.3 ns for snsp, and 0.6 ns for wnwp. A delay "in-between" snsp and wnwp can not be interpolated.

11.11.1.4 Lookup table for process- or derating-case coefficients

A nested arithmetic model construct can be used to describe lookup tables for coefficients, based on PROCESS or DERATE_CASE. These coefficients can be used in an EQUATION to calculate electrical data, e.g., DELAY.

Example

```
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 PROCESS coefficient 0.0 for nominal, -0.25 for snsp, and 0.5 for wnwp. Therefore the DELAY is 0.4 ns for the nominal process, 0.3 ns for snsp, and 0.6 ns for wnwp. Conceivably, the DELAY can be calculated for any value of the coefficient.

11.11.2 TEMPERATURE statement

TEMPERATURE can be used as argument in the HEADER of an arithmetic model for timing or electrical data. It can also be used as an arithmetic model with DERATE_CASE as argument, in order to describe what temperature applies for the specified derating case.

11.12 Arithmetic models for electrical data

Add lead-in text

11.12.1 PIN-related arithmetic models for electrical data

This section details the PIN arithmetic models for electrical data.

11.12.1.1 Principles

Arithmetic models for electrical data can be associated with a pin of a cell. Their meaning is illustrated in Figure 47.

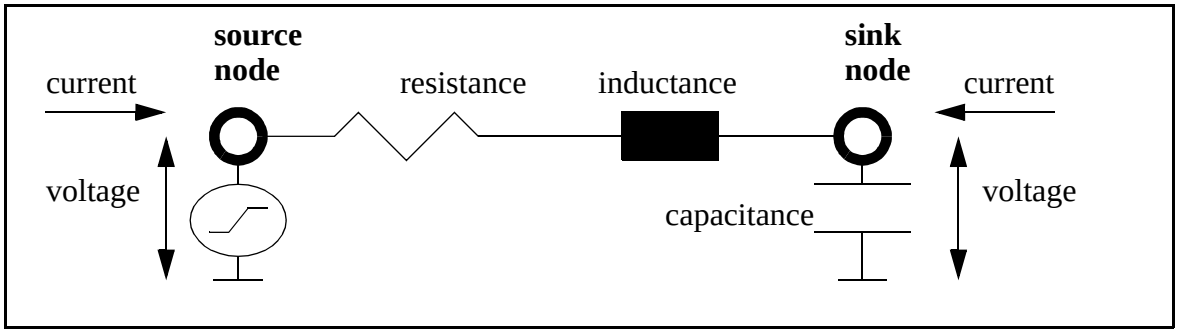


Figure 47—General representation of electrical models around a pin

A pin is represented as a source node and a sink node. For pins with DIRECTION=input, the source node is externally accessible. For pins with DIRECTION=output, the sink node is externally accessible.

11.12.1.2 CAPACITANCE, RESISTANCE, and INDUCTANCE

RESISTANCE and INDUCTANCE apply between the source and sink node. CAPACITANCE applies between the sink node and ground. By default, the values for resistance, inductance and capacitance shall be zero (0).

11.12.1.3 VOLTAGE and CURRENT

VOLTAGE and CURRENT can be measured at either source or sink node, depending on which node is externally accessible. However, a voltage source can only be connected to a source node. The sense of measurement for voltage shall be from the node to ground. The sense of measurement for current shall be *into* the node.

11.12.1.4 Context-specific semantics

An arithmetic model for VOLTAGE, CURRENT, SLEWRATE, RESISTANCE, INDUCTANCE, and CAPACITANCE can be associated with a PIN in one of the following ways.

a) A model in the context of a PIN

Example

```
PIN my_pin {
    CAPACITANCE = 0.025;
```

b) A model in the context of a CELL, WIRE, or VECTOR with PIN annotation

Example

```
VOLTAGE = 1.8 { PIN = my_pin; }
```

The model in the context of a PIN shall be used if the data is completely confined to the pin. That means, no argument of the model shall make reference to any pin, since such reference implies an external dependency. A model with dependency only on environmental data not associated with a pin (e.g., TEMPERATURE, PROCESS, and DERATE_CASE) can be described within the context of the PIN.

A model with dependency on external data applied to a pin (e.g., load capacitance) shall be described outside the context of the PIN, using a PIN annotation. In particular, if the model involves a dependency on logic state or logic transition of other PINs, the model shall be described within the context of a VECTOR.

Figure 48 illustrates electrical models associated with input and output pins.

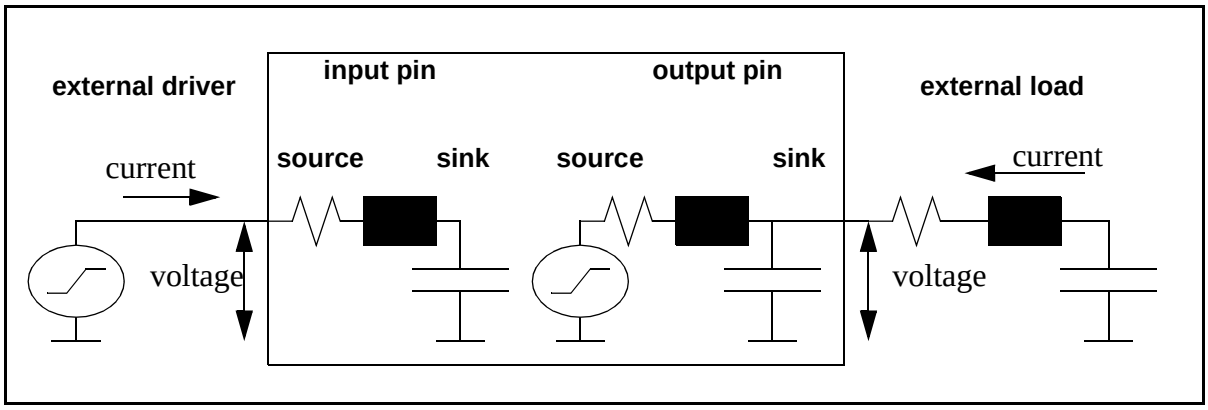


Figure 48—Electrical models associated with input and output pins

Table 99 and Table 100 define how models are associated with the pin, depending on the context.

Table 99—Direct association of models with a PIN

Model	Model in context of PIN	Model in context of CELL, WIRE, and VECTOR with PIN annotation
CAPACITANCE	Pin self-capacitance.	Externally controlled capacitance at the pin, e.g., voltage-dependent.
INDUCTANCE	Pin self-inductance.	Externally controlled inductance at the pin, e.g., voltage-dependent.
RESISTANCE	Pin self-resistance.	Externally controlled resistance at the pin, e.g., voltage-dependent, in the context of a VECTOR for timing-arc specific driver resistance.
VOLTAGE	Operational voltage measured at pin.	Externally controlled voltage at the pin.
CURRENT	Operational current measured into pin.	Externally controlled current into pin.
SAME_PIN_TIMING_MEASUREMENT	For model definition, default, etc.; not for the timing arc.	In context of VECTOR for timing arc, other context for definition, default, etc.
SAME_PIN_TIMING_CONSTRAINT	For model definition, default, etc.; not for the timing arc.	In context of VECTOR for timing arc, other context for definition, default, etc.

Table 100—External association of models with a PIN

Model / Context	LIMIT within PIN or with PIN annotation	Model argument with PIN annotation
CAPACITANCE	Min or max limit for applicable load.	Load for model characterization.
INDUCTANCE	Min or max limit for applicable load.	Load for model characterization.
RESISTANCE	Min or max limit for applicable load.	Load for model characterization.
VOLTAGE	Min or max limit for applicable voltage.	Voltage for model characterization.
CURRENT	Min or max limit for applicable current.	Current for model characterization.
SAME_PIN_TIMING_MEASUREMENT	Currently applicable for min or max limit for SLEWRATE.	Stimulus with SLEWRATE for model characterization.
SAME_PIN_TIMING_CONSTRAINT	N/A, since the keyword means a min or max limit by itself.	N/A

Example

```
CELL my_cell {
    PIN pin1 { DIRECTION=input; CAPACITANCE = 0.05; }
    PIN pin2 { DIRECTION=output; LIMIT { CAPACITANCE { MAX=1.2; } } }
    PIN pin3 { DIRECTION=input; }
    PIN pin4 { DIRECTION=input; }
    CAPACITANCE {
       PIN=pin3;
       HEADER { VOLTAGE { PIN=pin4; } }
       EQUATION { 0.25 + 0.34*VOLTAGE }
    }
}
```

The capacitance on `pin1` is `0.05`. The maximum allowed load capacitance on `pin2` is `1.2`. The capacitance on `pin3` depends on the voltage on pin4.

11.12.2 CAPACITANCE statement

Add lead-in text

11.12.3 RESISTANCE statement

Add lead-in text

11.12.4 INDUCTANCE statement

Add lead-in text

11.12.5 VOLTAGE statement

Add lead-in text

11.12.6 CURRENT statement

Add lead-in text

11.12.7 POWER and ENERGY statement

~~Arithmetic models for power calculation~~

This section defines the arithmetic models used for power calculation.

11.12.7.1 Principles

The purpose of power calculation is to evaluate the electrical power supply demand and electrical power dissipation of an electronic circuit. In general, both power supply demand and power dissipation are the same, due to the energy conservation law. However, there are scenarios where power is supplied and dissipated locally in different places. The power models in ALF shall be specified in such a way that the total power supply and dissipation of a circuit adds up correctly to the same number.

Example

> A capacitor `C` is charged from `0` volt to `V` volt by a switched DC source. The energy supplied by the source is $C*V^2$. The energy stored in the capacitor is $1/2*C*V^2$. Hence the dissipated energy is also $1/2*C*V^2$. Later the capacitor is discharged from `V` volt to `0` volt. The supplied energy is `0`. The dissipated energy is $1/2*C*V^2$. A supply-oriented power model can associate the energy $E_1=C*V^2$ with the charging event and $E_2=0$ with the discharging event. The total energy is $E=E_1+E_2=C*V^2$. A dissipation-oriented power model can associate the energy $E_3=1/2*C*V^2$ with both the charging and discharging event. The total energy is also $E=2*E_3=C*V^2$.

In many cases, it is not so easy to decide when and where the power is supplied and where it is dissipated. The choice between a supply-oriented and dissipation-oriented model or a mixture of both is subjective. Hence the ALF language provides no means to specify, which modeling approach is used. The choice is up to the model developer, as long as the energy conservation law is respected.

11.12.7.2 POWER and ENERGY

`POWER` and/or `ENERGY` models shall be in the context of a `CELL` or within a `VECTOR`. The total energy and/or power of a cell shall be calculated by combining the data of all models within the scope of the `CELL` or the `VECTOR`s within the cell.

The data for `POWER` and/or `ENERGY` shall be positive when energy is actually supplied to the `CELL` and/or dissipated within the `CELL`. The data shall be negative when energy is actually supplied or restored by the `CELL`.

Table 101 shows the mathematical relationship between ENERGY and POWER and the applicable MEASUREMENT annotations.

Table 101—Relations between ENERGY and POWER

MEASUREMENT for ENERGY	MEASUREMENT for POWER	Formula to calculate POWER from ENERGY	Formula to calculate ENERGY from POWER
`transient`	`transient`	$\frac{d}{dt}\text{ENERGY}$	$\int \text{POWER} dt$
`transient`	`average`	$\frac{\text{ENERGY}}{\text{TIME}}$	POWER · TIME
`transient`	`peak`	$\max\left(\left\|\frac{d}{dt}\text{ENERGY}\right\|\right)$	N/A
`transient`	`rms`	$\frac{1}{\text{TIME}} \cdot \int\left(\frac{d}{dt}\text{ENERGY}\right)^2 dt$	N/A
N/A	`static`	N/A	POWER · TIME
`static`	N/A	`0`	N/A

To establish a meaningful relationship between energy and power, the measurement for energy shall be `transient`. A `static` measurement for energy is conceivable, modeling a state with constant energy, but no power is dissipated during such a state. A `static` measurement for power models a state during which constant power dissipation occurs. Although it is not meaningful to describe an energy model for such a state, it is conceivable to calculate the energy by multiplying the power with the duration of the state. A 1-to-1 correspondence between power and energy can be established for `transient` and `average` power measurements, modeling instantaneous and average power, respectively. Therefore, it is redundant to specify both energy and power in such case. Also, `peak` and `rms` power can be conceivably calculated from a transient energy or power waveform, but transient energy can not be calculated from a `peak` or `rms` power measurement.

11.12.8 FLUX and FLUENCE statement

~~Arithmetic models for hot electron calculation~~

This section defines arithmetic models for hot electron calculation.

11.12.8.1 Principles

The purpose of hot electron calculation is to evaluate the damage done to the performance of an electronic device due to the hot electron effect. The hot electron effect consists in accumulation of electrons trapped in the gate oxide of a transistor. The more electrons are trapped, the more the device slows down. At a certain point, the performance specification no longer is met and the device is considered to be damaged.

11.12.8.2 FLUX and FLUENCE

FLUX and/or FLUENCE models shall be in the context of a CELL or within a VECTOR. Total fluence and/or flux of a cell shall be calculated by combining the data of all models within the scope of the CELL or the VECTORs within the cell.

Both FLUX and FLUENCE are measures for hot electron damage. FLUX relates to FLUENCE in the same way as POWER relates to ENERGY.

Table 102 shows the mathematical relationship between FLUENCE and FLUX and the applicable MEASUREMENT annotations.

Table 102—Relations between FLUENCE and FLUX

MEASUREMENT for FLUENCE	MEASUREMENT for FLUX	Formula to calculate FLUX from FLUENCE	Formula to calculate FLUENCE from FLUX
transient	transient	$\frac{d}{dt}\text{FLUENCE}$	$\int \text{FLUX}\,dt$
transient	average	$\frac{\text{FLUENCE}}{\text{TIME}}$	FLUX · TIME
N/A	static	N/A	FLUX · TIME
static	N/A	0	N/A

Since hot electron damage is purely cumulative, the only meaningful MEASUREMENT annotations are transient, average, and static.

11.12.9 DRIVE_STRENGTH statement

~~Other PIN-related arithmetic models~~

~~This section details some other PIN-related arithmetic models.~~

~~DRIVE_STRENGTH~~

DRIVE_STRENGTH is a unit-less, abstract measure for the drivability of a PIN. It can be used as a substitute of driver RESISTANCE. The higher the DRIVE_STRENGTH, the lower the driver RESISTANCE. However, DRIVE_STRENGTH can only be used within a coherent system of calculation models, since it does not represent an absolute quantity, as opposed to RESISTANCE. For example, the weakest driver of a library can have drive strength 1, the next stronger driver can have drive strength 2 and so forth. This does not necessarily mean the resistance of the stronger driver is exactly half of the resistance of the weaker driver.

An arithmetic model for conversion from DRIVE_STRENGTH to RESISTANCE can be given to relate the quantity DRIVE_STRENGTH across technology libraries.

Example

```
SUBLIBRARY high_speed_library {
    RESISTANCE {
       HEADER { DRIVE_STRENGTH } EQUATION { 800 / DRIVE_STRENGTH }
    }
    CELL high_speed_std_driver {
       PIN Z { DIRECTION = output; DRIVE_STRENGTH = 1; }
    }
}
SUBLIBRARY low_power_library {
    RESISTANCE {
       HEADER { DRIVE_STRENGTH } EQUATION { 1600 / DRIVE_STRENGTH }
    }
    CELL low_power_std_driver {
       PIN Z { DIRECTION = output; DRIVE_STRENGTH = 1; }
    }
}
```

Drive strength 1 in the high speed library corresponds to 800 ohm. Drive strength 1 in the low power library corresponds to 1600 ohm.

NOTE—Any particular arithmetic model for RESISTANCE in either library shall locally override the conversion formula from drive strength to resistance.

11.12.10 SWITCHING_BITS statement

The quantity SWITCHING_BITS applies only for bus pins. The range is from 0 to the width of the bus. Usually, the quantity SWITCHING_BITS is not calculated by an arithmetic model, since the number of switching bits on a bus depends on the functional specification rather than the electrical specification. However, SWITCHING_BITS can be used as argument in the HEADER of an arithmetic model to calculate electrical quantities, for instance, energy consumption.

Example

```
CELL my_rom {
    PIN [3:0] addr { DIRECTION=input;  SIGNALTYPE=address; }
    PIN [7:0] dout { DIRECTION=output; SIGNALTYPE=data; }
    VECTOR ( ?! addr -> ?! dout ) {
       ENERGY {
          HEADER {
             SWITCHING_BITS addr_bits { PIN = addr; }
             SWITCHING_BITS dout_bits { PIN = dout; }
          }
```

```
            EQUATION { 0.45*LOG(addr_bits) + 2.6*dout_bits }
        }
    }
}
```

The energy consumption of `my_rom` depends on the number of switching data bits and on the logarithm of the number of switching address bits.

11.12.11 NOISE and NOISE_MARGIN statement

~~Noise calculation~~

This section details the noise calculation definitions.

11.12.11.1 NOISE_MARGIN definition

Noise margin is defined as the maximal allowed difference between the ideal signal voltage under a well-specified operation condition and the actual signal voltage normalized to the ideal voltage swing. This is illustrated in Figure 49.

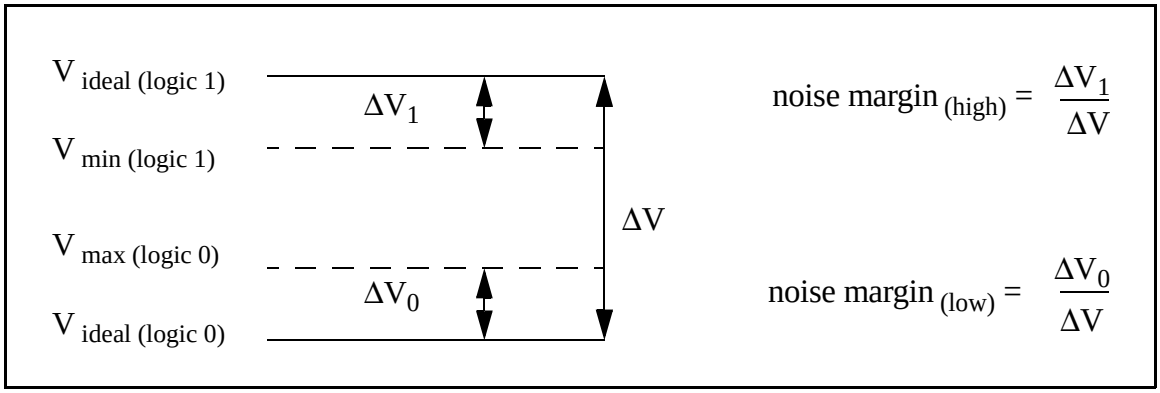


Figure 49—Definition of noise margin

Noise margin is measured at a signal input pin of a digital cell. The terms *ideal signal voltage* and *actual signal voltage* apply from the standpoint of that particular pin. In CMOS technology, the ideal signal voltage at a pin is the actual supply voltage of the cell, which is not necessarily identical to the nominal supply voltage of the chip.

The `NOISE_MARGIN` statement has the form of an arithmetic model. If the submodel keywords `HIGH` and `LOW` are used, it has the form of an arithmetic model container.

Examples

```
NOISE_MARGIN = 0.3;
NOISE_MARGIN { HIGH = 0.2;  LOW = 0.4; }
NOISE_MARGIN {
    HEADER { TEMPERATURE { TABLE { 0 50 100 } } }
    TABLE { 0.4 0.3 0.2 }
}
```

`NOISE_MARGIN` can be related to signal `VOLTAGE` by using the following statement:

```
        VOLTAGE {
           LOW = 0;
           HIGH = 2.5;
        }
        NOISE_MARGIN {
           LOW = 0.4;
           HIGH = 0.3;
        }
    }
```

In this example, the valid signal voltage levels are bound by 1 volt = `2.5 volt * 0.4` for logic `0` and 1.75 volt = `2.5 volt * (1 - 0.3)` for logic `1`.

11.12.11.2 Representation of noise in a VECTOR

In order to describe timing diagrams involving noisy signals, the symbolic state `*` (see 5.4.13) shall be used. This state represents arbitrary transitions between arbitrary states, which corresponds to the nature of noise, as shown in Figure 50.

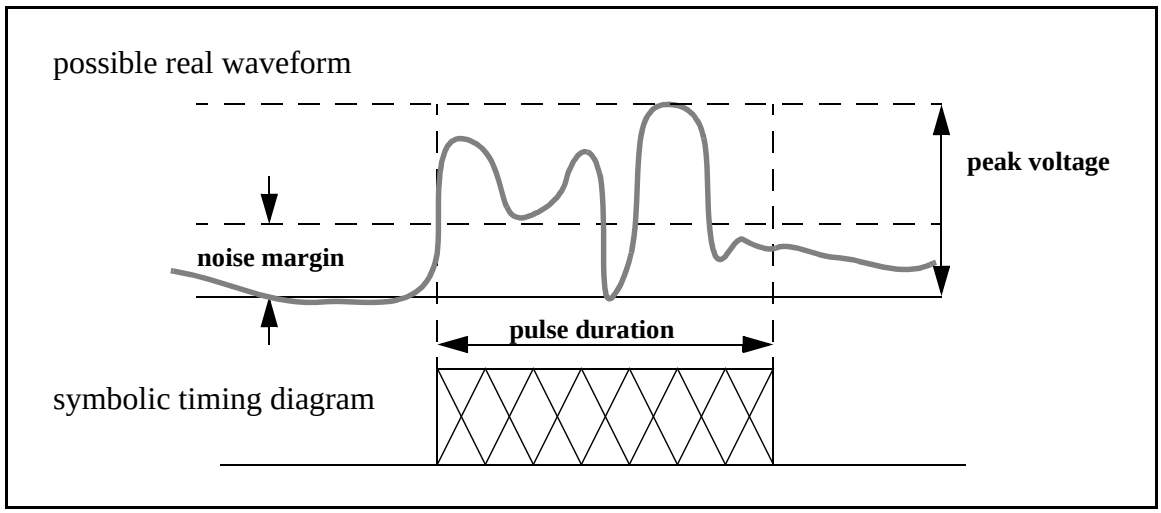


Figure 50—Timing diagram of a noisy signal

The signal can be above or below noise margin during the state `*`, but it shall be within noise margin during the state `0` or `1`. During the state `*`, the signal is bound by an envelope defined by the pulse duration and the peak voltage.

A description of the noisy signal is given in the following template:

```
VECTOR ( 0* my_pin -> *0 my_pin ) {
    TIME = <pulse_duration> {
       FROM { PIN=my_pin; EDGE_NUMBER=0; }
       TO   { PIN=my_pin; EDGE_NUMBER=1; }
    }
    VOLTAGE = <peak_voltage> {
       CALCULATION = incremental;
       MEASUREMENT = peak;
       PIN = my_pin;
```

```
      }
   }
```

The VECTOR describes the symbolic timing diagram. The TIME statement specifies the duration of the pulse. The VOLTAGE statement specifies the peak voltage. The annotation CALCULATION=incremental specifies that the voltage is measured from the nominal signal voltage level rather than from an absolute reference level and that noise voltage can add up.

It is also necessary to specify whether a noisy signal (which can oscillate above and below the noise margin) is considered as one symbolic noise pulse or separated into multiple symbolic noise pulses.

The LIMIT statement for TIME shall be used for that purpose, as shown in the following example and illustrated by the timing diagram shown in Figure 51.

Example

```
VECTOR ( *0 my_pin -> 0* my_pin ) {
   LIMIT {
      TIME {
         FROM { PIN = my_pin; EDGE_NUMBER = 0; }
         TO   { PIN = my_pin; EDGE_NUMBER = 1; }
         MIN = <minimum_pulse_separation> ;
      }
   }
}
```

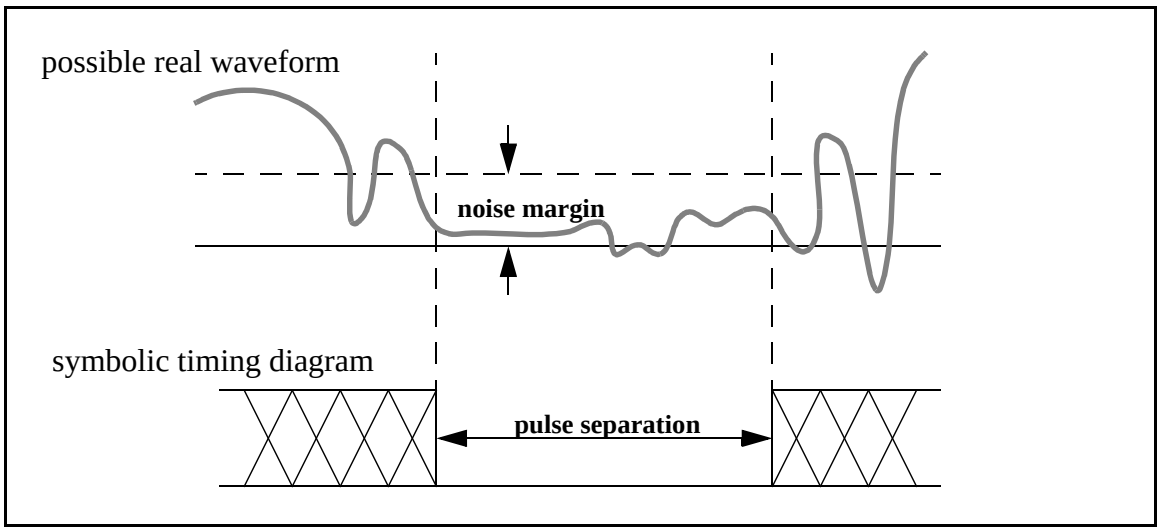


Figure 51—Separation between two noise pulses

When the minimum pulse separation is not met, consecutive noise pulses shall be symbolically merged into one pulse.

11.12.11.3 Context of NOISE_MARGIN

NOISE_MARGIN is a pin-related quantity. It can appear either in the context of a PIN statement or in the context of a VECTOR statement with PIN annotation. It can also appear in the global context of a CELL, SUBLIBRARY, or LIBRARY statement.

If a NOISE_MARGIN statement appears in multiple contexts, the following priorities apply:

a) NOISE_MARGIN with PIN annotation in the context of the VECTOR, NOISE_MARGIN with PIN annotation in the context of the CELL, or NOISE_MARGIN in the context of the PIN
b) NOISE_MARGIN without PIN annotation in the context of the CELL
c) NOISE_MARGIN in the context of the SUBLIBRARY
d) NOISE_MARGIN in the context of the LIBRARY
e) NOISE_MARGIN outside the LIBRARY

If the noise margin is constant or depends only on environmental quantities, the NOISE_MARGIN statement shall appear within the context of the PIN. The noise margin shall relate to the signal VOLTAGE levels applicable for that pin.

Example

```
PIN my_signal_pin {
    PINTYPE = digital;
    DIRECTION = input;
    VOLTAGE { LOW = 0; HIGH = 2.5; }
    NOISE_MARGIN { LOW = 0.4; HIGH = 0.3; }
}
```

If the noise margin depends on electrical quantities related to other pins, e.g., the supply voltage, the NOISE_MARGIN statement shall have a PIN annotation and appear in the context of the CELL.

Example

```
CELL my_cell {
    PIN my_signal_pin { PINTYPE = digital; DIRECTION = input; }
    PIN my_power_pin  { PINTYPE = supply; SUPPLYTYPE = power; }
    PIN my_ground_pin { PINTYPE = supply; SUPPLYTYPE = ground; }
    NOISE_MARGIN {
       PIN = my_signal_pin;
       HEADER {
          VOLTAGE vdd { PIN = my_power_pin;  }
          VOLTAGE vss { PIN = my_ground_pin; }
       }
       EQUATION { 0.16 * (vdd - vss ) }
    }
}
```

If the noise margin depends on the logical states and/or the timing of other pins, the NOISE_MARGIN statement shall have a PIN annotation and appear in the context of a VECTOR, describing the state-and/or timing dependency.

Example for state-dependent noise margin

```
CELL my_latch {
    PIN Q { DIRECTION = output; SIGNALTYPE = data; }
    PIN D { DIRECTION = input; SIGNALTYPE = data; }
    PIN CLK { DIRECTION = input; SIGNALTYPE = clock; POLARITY = high; }
    VECTOR ( CLK && ! D ) { NOISE_MARGIN = 0.4 { PIN = D; } }
    VECTOR ( CLK && D ) { NOISE_MARGIN = 0.3 { PIN = D; } }
}
```

Here, the pin `D` is only noise-sensitive when `CLK` is high. No noise margin is given for the case when `CLK` is low.

In the case of timing-dependency, the `vector_expression` shall indicate the time window where noise is allowed and not allowed for the applicable pin. The symbolic state `*` (see 5.4.13) shall be used to indicate a noisy signal.

Example for timing-dependent noise margin

```
VECTOR ( *? D -> 10 CLK -> ?* D ) {
    TIME T1 = 0.35 {
       FROM { PIN = D;   EDGE_NUMBER = 0; }
       TO   { PIN = CLK; EDGE_NUMBER = 0; }
    }
    TIME T2 = 0.28 {
       FROM { PIN = CLK; EDGE_NUMBER = 0; }
       TO   { PIN = D;   EDGE_NUMBER = 1; }
    }
    NOISE_MARGIN = 0.44 { PIN = D; }
}
```

This example corresponds to the timing diagram shown in Figure 52.

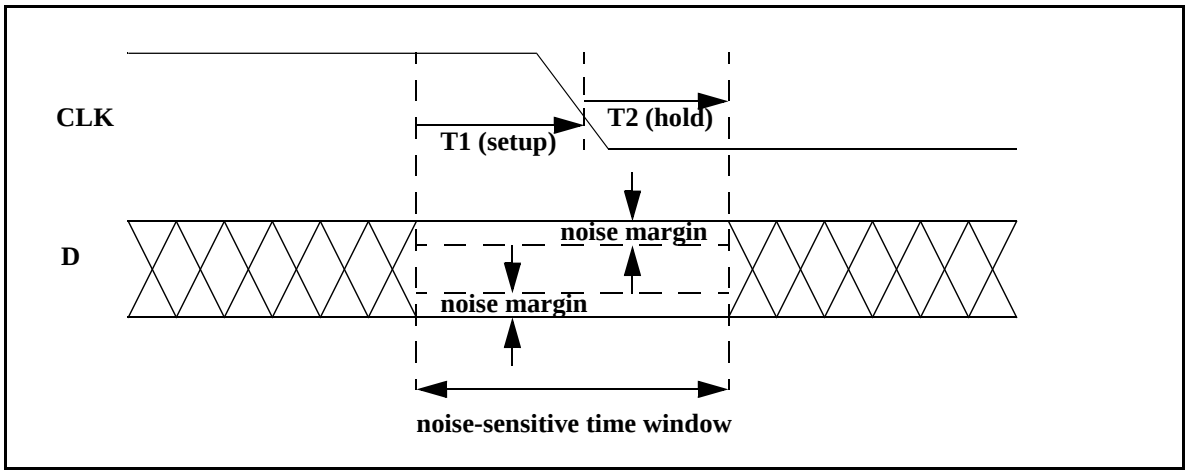


Figure 52—Example for timing-dependent noise margin

Noise on pin `D` is allowed `0.35` time-units before and `0.28` time-units after the falling edge of `CLK`. During the time window in-between, the noise margin is `0.44`.

11.12.11.4 Noise propagation

Noise propagation from input to output can be modeled in a similar way as signal propagation, using the concept of timing arcs. This is illustrated in Figure 53.

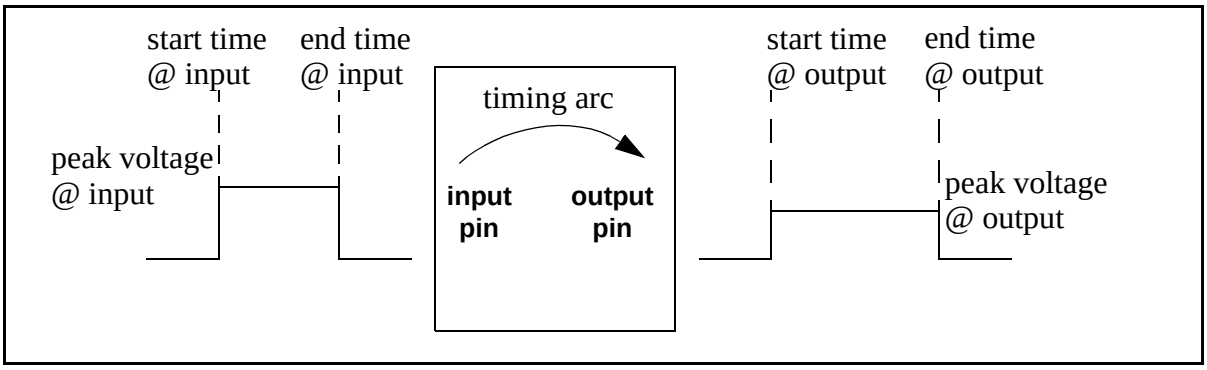


Figure 53—Principle of noise propagation

The principle of *signal propagation* is to calculate the output arrival time and slewrate from the input arrival time and slewrate. In a more abstract way, two points in time propagate from input to output. The same principle applies for noise propagation. Two points in time, start and end time of the noise waveform, propagate from input to output. In addition, the noise peak voltage also propagates from input to output. This is illustrated in Figure 54.

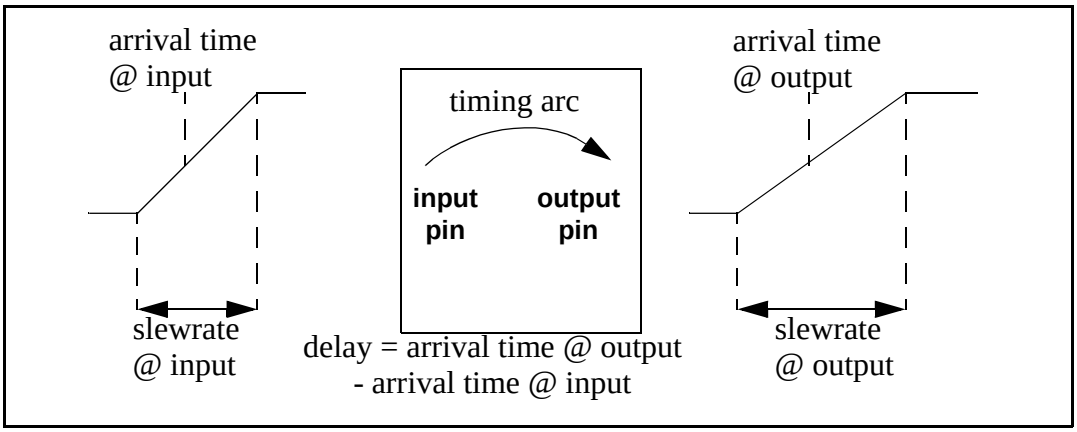


Figure 54—Principle of signal propagation

A `VECTOR` shall be used to describe the timing of the noise waveform. Again, the symbolic state `*` (see 5.4.13) shall be used to indicate a noisy signal.

Example

```
CELL my_cell {
    PIN A { DIRECTION = input; }
    PIN Z { DIRECTION = output; }
    VECTOR ( 0* A -> *0 A <&> 0* Z -> *0 Z ) {
       DELAY T1 {
          FROM { PIN = A; EDGE_NUMBER = 0; }
```

```
            TO   { PIN = Z; EDGE_NUMBER = 0; }
            /* fill in HEADER, TABLE or EQUATION */
         }
         DELAY T2 {
            FROM { PIN = A; EDGE_NUMBER = 1; }
            TO   { PIN = Z; EDGE_NUMBER = 1; }
            /* fill in HEADER, TABLE or EQUATION */
         }
         VOLTAGE { PIN = Z; MEASUREMENT = peak;
            /* fill in HEADER, TABLE or EQUATION */
         }
      }
```

This example corresponds to the timing diagram shown in Figure 55.

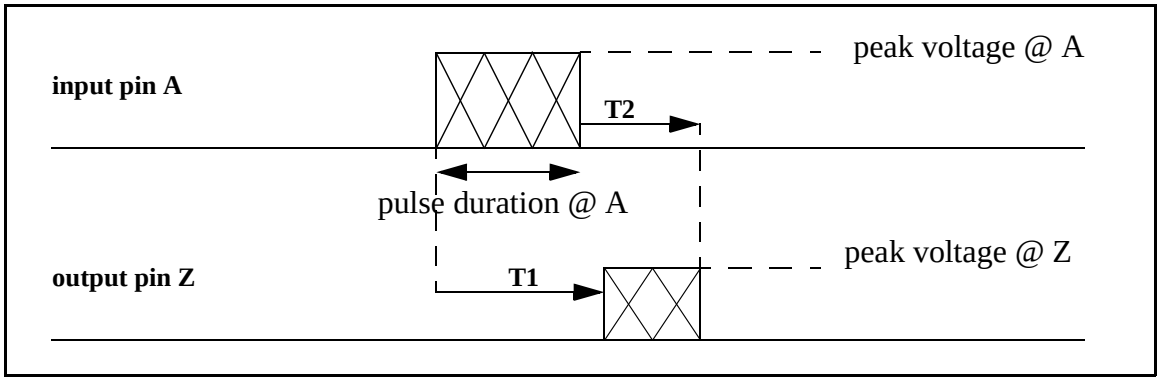


Figure 55—Example of noise propagation

The input to output delay of the leading edge of the noise pulse can depend on the peak voltage at pin A, the load capacitance at pin Z and other electrical quantities. In addition, the input to output delay of the trailing edge of the noise pulse as well as the peak voltage at pin Z can also depend on the duration of the pulse at pin A.

NOTE—The time measurement from start to end of the noise pulse shall be represented by the keyword `TIME` (no causality between start and end time), whereas the time measurement from input to output shall be represented by the keyword `DELAY` (causality between input and output arrival time).

11.12.11.5 Noise rejection

Noise rejection is a limit case for noise propagation, when the output peak voltage is so low the noise is considered rejected. In this case, the input peak voltage can still be above noise margin, whereas the output peak voltage is way below noise margin.

Example

```
CELL my_cell {
    PIN A { DIRECTION = input; }
    PIN Z { DIRECTION = output; }
    VECTOR ( 0* A -> *0 A -> 00 Z ) {
       LIMIT {
          VOLTAGE {
             PIN = A; MEASUREMENT = peak;
             MAX { /* fill in HEADER, TABLE or EQUATION */ }
```

```
            }
        }
    }
}
```

NOTE—The `vector_expression 00 Z` says explicitly a transition at pin `Z` does *not* happen.

This example corresponds to the timing diagram shown in Figure 56.

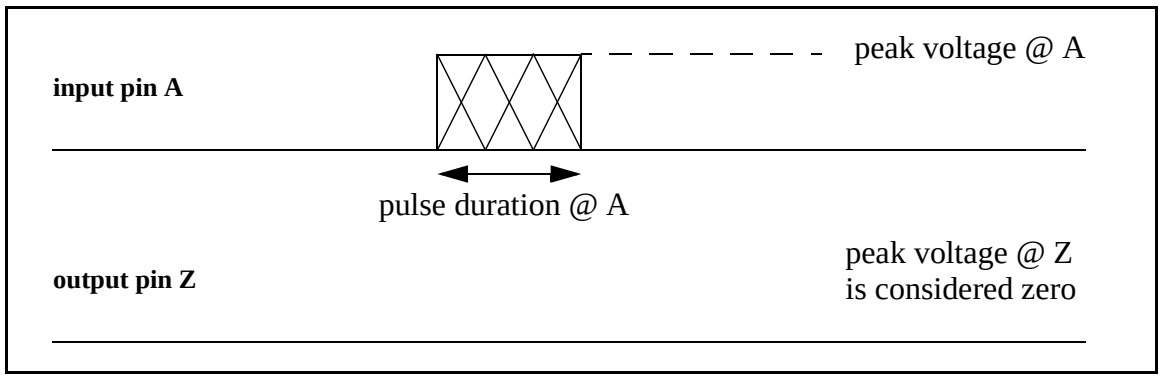


Figure 56—Example of noise rejection

The peak voltage limit for noise rejection can depend on the duration of the noise pulse at pin A and other electrical quantities, e.g., the load capacitance at pin `Z`. If the peak voltage limit does not depend on the duration of the noise pulse, the `NOISE_MARGIN` statement shall be used rather than the vector-specific `LIMIT` construct for noise rejection.

11.12.12 Annotations for arithmetic models for electrical data

~~Annotations for arithmetic models~~

This section defines the annotations for arithmetic models.

11.12.12.1 MEASUREMENT annotation

Arithmetic models describing analog measurements (see Table 78) can have a `MEASUREMENT` annotation. This annotation indicates the type of measurement used for the computation in arithmetic model.

MEASUREMENT = `string ;`

The string can take the values shown in Table 103.

Table 103—MEASUREMENT annotation

Annotation string	Description
`transient`	Measurement is a transient value.
`static`	Measurement is a static value.
`average`	Measurement is an average value.

Table 103—MEASUREMENT annotation (Continued)

Annotation string	Description
rms	Measurement is an root mean square value.
peak	Measurement is a peak value.

Their mathematical definitions are shown in Figure 57.

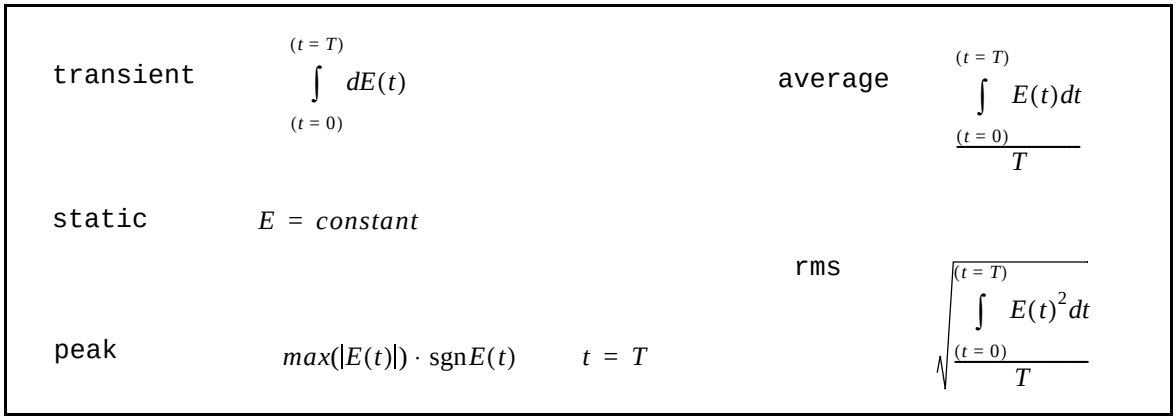


Figure 57—Mathematical definitions for MEASUREMENT annotations

Examples

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

11.12.12.2 Rules for combinations of annotations

Cumulative values of arithmetic models can be calculated for models which are cumulative in nature (e.g., ENERGY or POWER) or by the usage of CALCULATION=incremental (e.g., CURRENT or VOLTAGE). The MEASUREMENT annotation can be used in conjunction with the calculation of cumulative values under the following restrictions:

- Data with MEASUREMENT=average for each model can be combined, provided the TIME annotation value is the same.
- Data with MEASUREMENT=peak for each model can be combined, provided the TIME annotation or a complementary TIME model within the same context specify that the peak values can occur at the same time.
- Data with MEASUREMENT=rms for each model can not be combined.
- Data with different MEASUREMENT annotations can not be combined.
- Data with MEASUREMENT=transient|static can be combined with each other.

All data that can be combined under the above mentioned restrictions, shall be in a compatible context, e.g., mutually non-exclusive VECTORs within a CELL.

11.13 Arithmetic models for physical data

Add lead-in text

11.13.1 CONNECTIVITY statement

This section defines the `CONNECTIVITY` statement and its use.

11.13.1.1 Definition

A `CONNECTIVITY` statement is defined as shown in Syntax 115.

```
connectivity ::=
    CONNECTIVITY [ identifier ] {
     connect_rule_annotation between_multi_value_assignment }
   |CONNECTIVITY [ identifier ] {
    connect_rule_annotation table_based_model }
```

Syntax 115—CONNECTIVITY statements

11.13.1.2 CONNECT_RULE annotation

The *connect_rule annotation* can be only inside a `CONNECTIVITY` object. It specifies the connectivity requirement.

CONNECT_RULE = `string ;`

which can take the values shown in Table 104.

Table 104—CONNECT_RULE annotation

Annotation string	Description
`must_short`	Electrical connection required.
`can_short`	Electrical connection allowed.
`cannot_short`	Electrical connection disallowed.

It is not necessary to specify more than one rule between a given set of objects. If one rule is specified to be *True*, the logical value of the other rules can be implied shown in Table 105.

Table 105—Implications between connect rules

must_short	cannot_short	can_short
False	*False*	*True*
False	*True*	*False*
True	*False*	N/A

11.13.1.3 CONNECTIVITY modeled with BETWEEN statement

The BETWEEN statement specifies the objects for which the connectivity applies, as shown in Syntax 116.

*between*_multi_value_assignment ::=
BETWEEN { identifiers **}**

Syntax 116—BETWEEN statements

If the BETWEEN statement contains only one identifier, than the CONNECTIVITY shall apply between multiple instances of the same object.

Example

```
CLASS analog_power;
CLASS analog_ground;
CLASS digital_power;
CLASS digital_ground;
CONNECTIVITY Aground { // connect all members of CLASS analog_ground
    CONNECT_RULE = must_short;
    BETWEEN { analog_ground }
}
CONNECTIVITY Dground { // connect all members of CLASS digital_ground
    CONNECT_RULE = must_short;
    BETWEEN { digital_ground }
}
CONNECTIVITY Apower { // connect all members of CLASS analog_power
    CONNECT_RULE = must_short;
    BETWEEN { analog_power }
}
CONNECTIVITY Dpower { // connect all members of CLASS digital_power
    CONNECT_RULE = must_short;
    BETWEEN { digital_power }
}
CONNECTIVITY Aground2Dground {
    CONNECT_RULE = must_short;
    BETWEEN { analog_ground digital_ground }
}
CONNECTIVITY Apower2Dpower {
    CONNECT_RULE = can_short;
    BETWEEN { analog_power digital_power }
}
CONNECTIVITY Apower2Aground {
    CONNECT_RULE = cannot_short;
    BETWEEN { analog_power analog_ground }
}
CONNECTIVITY Apower2Dground {
    CONNECT_RULE = cannot_short;
    BETWEEN { analog_power digital_ground }
}
CONNECTIVITY Dpower2Aground {
    CONNECT_RULE = cannot_short;
    BETWEEN { digital_power analog_ground }
```

```
}
CONNECTIVITY Dpower2Dground {
    CONNECT_RULE = cannot_short;
    BETWEEN { digital_power digital_ground }
}
```

11.13.1.4 CONNECTIVITY modeled as lookup TABLE

Connectivity can also be described as a lookup table model. This description is usually more compact than the description using the BETWEEN statements.

The connectivity model can have the arguments shown in Table 106 in the HEADER.

Table 106—Arguments for connectivity

Argument	Value type	Description
DRIVER	string	Argument of connectivity function.
RECEIVER	string	Argument of connectivity function.

Each argument shall contain a TABLE.

The connectivity model specifies the allowed and disallowed connections amongst drivers or receivers in one-dimensional tables or between drivers and receivers in two-dimensional tables.The boolean literals in the table refer to the CONNECT_RULE as shown in Table 107.

Table 107—Boolean literals in non-interpolateable tables

Boolean literal	Description
1	CONNECT_RULE is *True*.
0	CONNECT_RULE is *False*.
?	CONNECT_RULE does not apply.

Example

```
CLASS analog_power;
CLASS analog_ground;
CLASS digital_power;
CLASS digital_ground;
CONNECTIVITY all_must_short {
    CONNECT_RULE = must_short;
    HEADER {
       RECEIVER r1 {
         TABLE {analog_ground analog_power digital_ground digital_power}
       }
       RECEIVER r2 {
          TABLE {analog_ground analog_power digital_ground digital_power}
```

```
        }
    }
    TABLE {
        1 0 1 0
        0 1 0 0
        1 0 1 0
        0 0 0 1
    }
/*
The following table would apply, if the CONNECT_RULE was "cannot_short":
    TABLE {
        0 1 0 1
        1 0 1 0
        0 1 0 1
        1 0 1 0
    }
The following table would apply, if the CONNECT_RULE was "can_short":
    TABLE {
        ? 0 ? 0
        0 ? 0 ?
        ? 0 ? 0
        0 ? 0 ?
    }
*/
}
```

11.13.2 SIZE statement

Add lead-in text

11.13.3 AREA statement

Add lead-in text

11.13.4 WIDTH statement

Add lead-in text

11.13.5 HEIGHT statement

Add lead-in text

11.13.6 LENGTH statement

Add lead-in text

11.13.7 DISTANCE statement

Add lead-in text

11.13.8 OVERHANG statement

Add lead-in text

11.13.9 PERIMETER statement

Add lead-in text

11.13.10 EXTENSION statement

Add lead-in text

11.13.11 THICKNESS statement

Add lead-in text

11.13.12 Annotations for arithmetic models for physical data

~~Physical annotations for arithmetic models~~

This section defines the physical annotations for arithmetic models.

11.13.12.1 BETWEEN statement within DISTANCE, LENGTH

The `BETWEEN` statement within `DISTANCE` or `LENGTH` (see 11.8.2 and the example in Section 9.11.5) shall identify the objects for which the measurement applies. The syntax is shown in Syntax 116.

If the `BETWEEN` statement contains only one identifier, than the `DISTANCE` or `LENGTH`, respectively, shall apply between multiple instances of the same object, as shown in the following example and Figure 58.

Example

```
DISTANCE = 4 { BETWEEN { object1 object2 } }
LENGTH = 2 { BETWEEN { object1 object2 } }
```

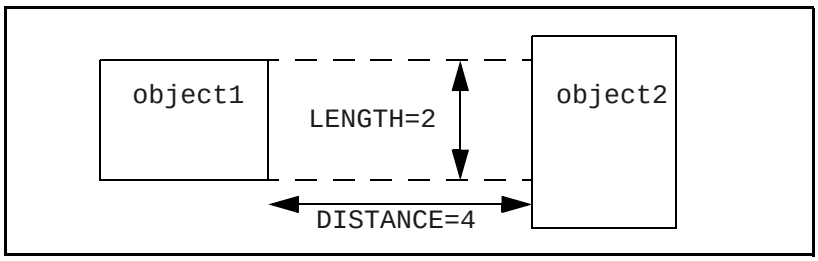


Figure 58—Illustration of LENGTH and DISTANCE

11.13.12.2 MEASUREMENT annotation for DISTANCE

The `MEASUREMENT` statement specifies the objects for which the connectivity applies, as shown in Syntax 117.

The default for measuring the distance between objects is **straight**.

The mathematical definitions for distance measurements between two points with differential coordinates Δx and Δy are:

— *straight* distance = $(\Delta x^2 + \Delta y^2)^{1/2}$
— *horizontal* distance = Δx

*distance_measurement*_assignment ::=
MEASUREMENT = *distance_measurement*_identifier **;**
*distance_measurement*_identifier ::=
straight
| **horizontal**
| **vertical**
| **manhattan**

Syntax 117—MEASUREMENT statements

— *vertical* distance = Δy
— *manhattan* distance = Δx + Δy

11.13.12.3 REFERENCE annotation for DISTANCE

The *reference*_annotation shall specify the reference for distance measurements between objects, as shown in Syntax 118.

*reference*_annotation ::=
REFERENCE = *reference*_identifier **;**
*reference*_identifier ::=
center
| **origin**
| **edge**

Syntax 118—REFERENCE annotation

The default shall be **edge**. The value center is only applicable for objects with EXTENSION, whereas the value edge is applicable for any physical object. The value origin is only applicable for objects with specified coordinates. This is depicted in Figure 59.

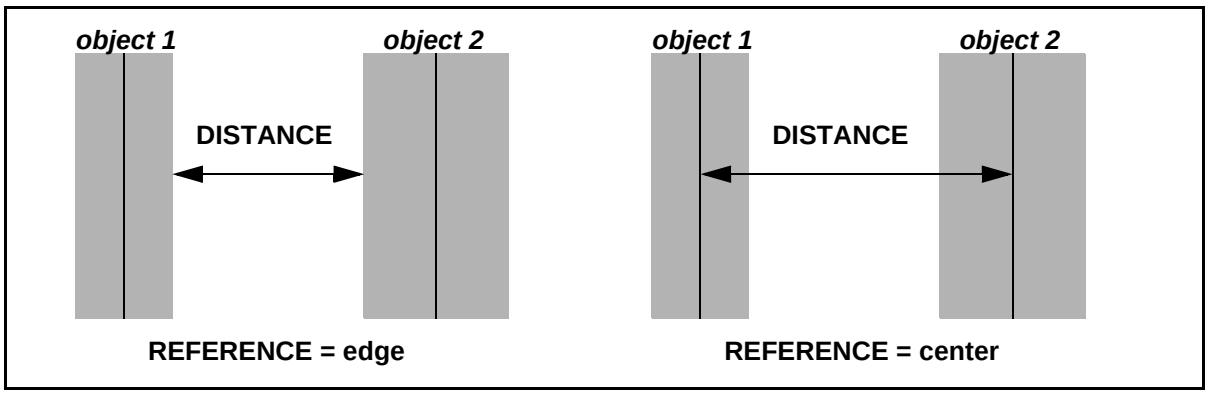


Figure 59—Illustration of REFERENCE for DISTANCE

11.13.12.4 Reference to ANTENNA

In hierarchical design, a PIN with physical PORTs can be abstracted. Therefore, an arithmetic model for SIZE, AREA, PERIMETER, etc. **relevant?? for certain antenna rules can be precalculated. An ANTENNA statement within the arithmetic model enables references to the set of antenna rules for which the arithmetic model applies, as shown in Syntax 119.

*antenna_reference*_multi_value_assignment ::=
ANTENNA { *antenna*_identifiers **}**

Syntax 119—ANTENNA statement

Example

```
CELL cell1 {
    PIN pin1 {
       AREA poly_area = 1.5 {
          LAYER = poly;
          ANTENNA { individual_m1 individual_via1 }
       }
       AREA m1_area = 1.0 {
          LAYER = metal1;
          ANTENNA { individual_m1 }
       }
       AREA via1_area = 0.5 {
          LAYER = via1;
          ANTENNA { individual_via1 }
       }
    }
}
```

The area `poly_area` is used in the rules `individual_m1` and `individual_via1`.
The area `m1_area` is used in the rule `individual_m1` only.
The area `via1_area` is used in the rule `individual_via1` only.

The case with diffusion is illustrated in the following example:

```
CELL my_diode {
    CELLTYPE = special; ATTRIBUTE { DIODE }
    PIN my_diode_pin {
       AREA = 3.75 {
          LAYER = diffusion;
          ANTENNA { rule1_for_diffusion rule2_for_diffusion }
       }
    }
}
```

11.13.12.5 Reference to PATTERN

Reference to a PATTERN shall be legal within arithmetic models, if the pattern and the model are within the scope of the same parent object, as shown in Syntax 120.

*pattern_reference*_assignment ::=
PATTERN = *pattern*_identifier **;**

Syntax 120—PATTERN reference

The pattern reference shall be applicable for LENGTH, WIDTH, HEIGHT, SIZE, AREA, THICKNESS, PERIMETER, EXTENSION (see 11.8.2 and the example in Section 9.11.2).

11.14 Arithmetic submodels for timing and electrical data

Add lead-in text

11.14.1 RISE and FALL statement

~~RISE and FALL submodels~~

For timing models in the context of a VECTOR, submodels for RISE and FALL are only applicable if the vector_expression does not specify the switching direction of the referenced PIN and EDGE_NUMBER. This is the case, when symbolic vector_unary operators are used, i.e., ?!, ??, ?*, or *? instead of 01, 10, etc.

For *SAME_PIN_TIMING_MEASUREMENT* or *SAME_PIN_TIMING_CONSTRAINT*, the RISE and FALL submodels apply for the <refEdge>.

For a partially specified *TIMING_MEASUREMENT* or *TIMING_CONSTRAINT*, the RISE and FALL submodels apply for the <fromEdge> or <toEdge>, whichever is specified.

For a completely specified *TIMING_MEASUREMENT* or *TIMING_CONSTRAINT*, it is not possible to apply a RISE and FALL submodel for both <fromEdge> and <toEdge>. The vector_unary operator shall specify the switching direction for at least one edge. If the switching direction for both edges is unspecified, the RISE and FALL submodel shall apply for the <toEdge>.

Example

```
VECTOR ( 01 CLK -> ?! Q ) {
    DELAY { FROM { PIN = CLK; } TO { PIN = Q; }
       RISE = 0.76; FALL = 0.58;
    }
}
// If Q is a scalar pin, the following construct is equivalent:
VECTOR ( 01 CLK -> 01 Q ) {
    DELAY = 0.76 { FROM { PIN = CLK; } TO { PIN = Q; } }
}
VECTOR ( 01 CLK -> 10 Q ) {
    DELAY = 0.58 { FROM { PIN = CLK; } TO { PIN = Q; } }
}
```

11.14.2 HIGH and LOW statement

~~Submodels for RISE, FALL, HIGH, and LOW~~

RISE and FALL contain data characterized in transient measurements. HIGH and LOW contain data characterized in static measurements.

```
<modelKeyword> { RISE=<modelValueRise>; FALL=<modelValueFall>; }
<modelKeyword> { HIGH=<modelValueHigh>;  LOW=<modelValueLow>; }
```

It is generally not required that both RISE and FALL or both HIGH and LOW, respectively, appear as an arithmetic submodel.

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 values 1 and 0, respectively. For other

states and their mapping to logic values, see 5.1.5. If the arithmetic model is within the scope of a vector which describes the logic values without ambiguity, the use of RISE and FALL or HIGH and LOW does not apply.

HIGH, LOW, RISE, and FALL apply for all pin-related arithmetic models with the following exceptions:

— RISE and FALL do not apply for VOLTAGE.
— HIGH and LOW do not apply for *SAME_PIN_TIMING_MEASUREMENT* and *SAME_PIN_TIMING_CONSTRAINT*.

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

11.15 Arithmetic submodels for physical data

Add lead-in text

11.15.1 HORIZONTAL and VERTICAL statement

Add lead-in text

This is a single subheader

Annex A

(informative)

Syntax rule summary

This summary replicates the syntax detailed in the preceding clauses. If there is any conflict, in detail or completeness, the syntax presented in the clauses shall considered as the normative definition.

The current ordering is as each item appears in its subchapter; this needs to be updated to be complete.

A.1 Lexical definitions

any_character ::= (see 6.2.3)
  reserved_character
| nonreserved_character
| escape_character
| whitespace

reserved_character ::= (see 6.2.3)
  & | **|** | **^** | **~** | **+** | **-** | ***** | **/** | **%** | **?** | **!** | **=** | **<** | **>** | **:** | **(** | **)** | **[** | **]** | **{** | **}** | **@** | **;** | **,** | **.** | **”** | **’**

nonreserved_character ::= (see 6.2.4)
  letter | digit | **_** | **$** | **#**

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

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

escape_character ::= (see 6.2.5)
  ****

delimiter ::= (see 6.3)
  reserved_character
| **&&** | **~&** | **||** | **~|** | **~^** | **==** | **!=** | ****** | **>=** | **<=** | **?!** | **?~** | **?-** | **??** | **?*** | ***?**
| **->** | **<->** | **&>** | **<&>** | **>>** | **<<**

comment ::= (see 6.2)
  single_line_comment
| block_comment

integer ::= (see 6.5)
  [sign] unsigned

sign ::=
  + | **-**

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

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

number ::=
  [sign] non_negative_number

bit_literal ::= (see 6.6)
  numeric_bit_literal

| alphabetic_bit_literal
| dont_care_literal
| random_literal

numeric_bit_literal ::=
0 | **1**

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

dont_care_literal ::=
?

random_literal ::=
*****

based_literal ::= (see 6.7)
binary_base { **_** | binary_digit }
| octal_base { **_** | octal_digit }
| decimal_base { **_** | digit }
| hex_base { **_** | hex_digit }

binary_base ::=
'B | **'b**

binary_digit ::=
bit_literal

octal_base ::=
'O | **'o**

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

decimal_base ::=
'D | **'d**

hex_base ::=
'H | **'h**

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

edge_literal ::= (see 6.8)
bit_edge_literal
| word_edge_literal
| symbolic_edge_literal

bit_edge_literal ::=
bit_literal bit_literal

word_edge_literal ::=
based_literal based_literal

symbolic_edge_literal ::=
?? | **?~** | **?!** | **?-**

quoted_string ::= (see 6.9)
" { any_character } **"**

identifiers ::= (see 6.10)
identifier { identifier }

identifier ::=
nonescaped_identifier
| escaped_identifier
| placeholder_identifier
| hierarchical_identifier

nonescaped_identifier ::= (see 6.10.1)
nonreserved_character { nonreserved_character }

escaped_identifier ::= (see 6.10.2)
    escape_character escaped_characters
escaped_characters ::=
    escaped_character { escaped_character }
escaped_character ::=
      nonreserved_character
    | reserved_character
    | escape_character
placeholder_identifier ::= (see 6.10.3)
    < nonescaped_identifier >
hierarchical_identifier ::= (see 6.10.4)
    identifier . { identifier . } identifier
arithmetic_values ::= (see 6.6.1)
    arithmetic_value { arithmetic_value }
arithmetic_value ::=
      number
    | identifier
    | pin_value
string_value ::= (see 6.6.2)
      quoted_string
    | identifier
edge_values ::= (see 6.6.3)
    edge_value { edge_value }
edge_value ::=
    (edge_literal)
index_value ::= (see 6.6.4)
      unsigned
    | identifier

A.2 Auxiliary definitions

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

```
annotation ::=                                                    (see 7.3.1)
        one_level_annotation
      | two_level_annotation
      | multi_level_annotation
one_level_annotations ::=
        one_level_annotation { one_level_annotation }
one_level_annotation ::=
        single_value_annotation
      | multi_value_annotation
single_value_annotation ::=
        identifier = annotation_value ;
multi_value_annotation ::=
        identifier { annotation_values }
two_level_annotations ::=
        two_level_annotation { two_level_annotation }
two_level_annotation ::=
        one_level_annotation
      | identifier [ = annotation_value ]
          { one_level_annotations }
multi_level_annotations ::=
        multi_level_annotation { multi_level_annotation }
multi_level_annotation ::=
        one_level_annotation
      | identifier [ = annotation_value ]
          { multi_level_annotations }
annotation_values ::=                                             (see 7.3.2)
        annotation_value { annotation_value }
annotation_value ::=
        index_value
      | string_value
      | edge_value
      | pin_value
      | arithmetic_value
      | boolean_expression
      | control_expression
all_purpose_items ::=                                             (see 7.18)
        all_purpose_item { all_purpose_item }
all_purpose_item ::=
        include
      | alias
      | constant
      | attribute
      | property
      | class_declaration
      | keyword_declaration
      | group_declaration
      | template_declaration
      | template_instantiation
      | annotation
      | arithmetic_model
      | arithmetic_model_container
```

A.3 Generic definitions

include ::= (see 8.1)
INCLUDE quoted_string **;**

alias ::= (see 8.1)
ALIAS identifier **=** identifier **;**

constant ::= (see 8.2)
CONSTANT identifier **=** arithmetic_value **;**

attribute ::= (see 8.4)
ATTRIBUTE { identifiers **}**

property ::= (see 8.5)
PROPERTY [identifier] **{** one_level_annotations **}**

class_declaration ::= (see 8.3)
CLASS identifier **;**
| **CLASS** identifier **{** all_purpose_items **}**

keyword_declaration ::= (see 8.4)
KEYWORD context_sensitive_keyword **=** *syntax_item*_identifier **;**

group_declaration ::= (see 8.6)
GROUP *group*_identifier **{** annotation_values **}**
| **GROUP** *group*_identifier **{** index_value **:** index_value **}**

template_declaration ::= (see 8.7)
TEMPLATE *template*_identifier **{** template_items **}**

template_items ::=
template_item { template_item }

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

| table
| equation
| arithmetic_submodel
| behavior_item
| geometric_model

template_instantiation ::=
static_template_instantiation
| dynamic_template_instantiation

static_template_instantiation ::=
*template*_identifier [**= static**] **;**
| *template*_identifier [**= static**] **{** annotation_values **}**
| *template*_identifier [**= static**]**{** one_level_annotations **}**

dynamic_template_instantiation ::=
*template*_identifier **= dynamic**
{ dynamic_template_instantiation_items **}**

dynamic_template_instantiation_items ::=
dynamic_template_instantiation_item
{ dynamic_template_instantiation_item }

dynamic_template_instantiation_item ::=
one_level_annotation
| arithmetic_model

A.4 Library definitions

library ::= (see 9.1)
LIBRARY *library*_identifier **{** library_items **}**
| **LIBRARY** *library*_identifier **;**
| *library*_template_instantiation

library_items ::=
library_item { library_item }

library_item ::=
sublibrary
| sublibrary_item

library ::=
SUBLIBRARY *sublibrary*_identifier **{** sublibrary_items **}**
| **SUBLIBRARY** *sublibrary*_identifier **;**
| *sublibrary*_template_instantiation

sublibrary_items ::= (see 9.2.2)
sublibrary_item { sublibrary_item }

sublibrary_item ::=
all_purpose_item
| cell
| primitive
| wire
| layer
| via
| rule
| antenna

| array
| site

*INFORMATION*_two_level_annotation ::= (see 9.2.3)
INFORMATION { *information*_one_level_annotations **}**

*information*_one_level_annotations ::=
*information*_one_level_annotation
{ *information*_one_level_annotation }

*information*_one_level_annotation ::=
*AUTHOR*_one_level_annotation
| *VERSION*_one_level_annotation
| *DATETIME*_one_level_annotation
| *PROJECT*_one_level_annotation

cell ::= (see 9.3.1)
CELL *cell*_identifier **{** cell_items **}**
| **CELL** *cell*_identifier **;**
| *cell*_template_instantiation

cell_items ::=
cell_item { cell_item }

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

non_scan_cell ::= (see 9.8)
NON_SCAN_CELL { unnamed_cell_instantiations **}**
| **NON_SCAN_CELL =** unnamed_cell_instantiation
| *non_scan_cell*_template_instantiation

unnamed_cell_instantiations ::=
unnamed_cell_instantiation { unnamed_cell_instantiation }

unnamed_cell_instantiation ::=
*cell*_identifier **{** pin_values **}**
| *cell*_identifier **{** pin_assignments **}**

pin ::= (see 9.4.1)
PIN [**[** index_range **]**] *pin*_identifier [**[** index_range **]**] **{** pin_items **}**
| **PIN** [**[** index_range **]**] *pin*_identifier [**[** index_range **]**] **;**
| *pin*_template_instantiation

pin_item ::=
all_purpose_item
| range
| port
| pin_instantiation

pin_items ::=
pin_item { pin_item }

pin_instantiation ::=
pin_variable **{** pin_items **}**

range ::= (see 9.6)
RANGE { index_range **}**

pin_group ::= (see 9.6.1)
PIN_GROUP [**[** index_range **]**] *pin_group*_identifier **{** pin_group_items **}**
| *pin_group*_template_instantiation

```
pin_group_items ::=
        pin_group_item { pin_group_item }
pin_group_item ::=
         all_purpose_item
        | range
wire ::=                                                        (see 9.10.1)
         WIRE wire_identifier { wire_items }
        | WIRE wire_identifier ;
        | wire_template_instantiation
wire_items ::=
        wire_item { wire_item }
wire_item ::=
         all_purpose_item
        | node
node ::=                                                        (see 9.10.2)
         NODE node_identifier { node_items }
        | NODE node_identifier ;
        | node_template_instantiation
node_items ::=
        node_item { node_item }
node_item ::=
        all_purpose_item
vector ::=                                                      (see 9.11)
         VECTOR control_expression { vector_items }
        | VECTOR control_expression ;
        | vector_template_instantiation
vector_items ::=
        vector_item { vector_item }
vector_item ::=
         all_purpose_item
        | illegal
illegal ::=                                                     (see 9.6.2)
         ILLEGAL { illegal_items }
        | illegal_template_instantiation
illegal_items ::=
        illegal_item { illegal_item }
illegal_item ::=
         all_purpose_item
        | violation
layer ::=                                                       (see 9.14.1)
         LAYER layer_identifier { layer_items }
        | LAYER layer_identifier ;
        | layer_template_instantiation
layer_items ::=
        layer_item { layer_item }
layer_item ::=
        all_purpose_item
via ::=                                                         (see 9.15.1)
         VIA via_identifier { via_items }
        | VIA via_identifier ;
        | via_template_instantiation
via_items ::=
        via_item { via_item }
```

```
via_item ::=
        all_purpose_item
      | pattern
      | artwork
via_reference ::=                                              (see 9.15.4)
        VIA { via_instantiations }
      | VIA { via_identifiers }
via_instantiations ::=
      via_instantiation { via_instantiation }
via_instantiation ::=
      via_identifier { geometric_transformations }
rule ::=                                                       (see 9.16.1)
        RULE rule_identifier { rule_items }
      | RULE rule_identifier ;
      | rule_template_instantiation
rule_items ::=
      rule_item { rule_item }
rule_item ::=
        all_purpose_item
      | pattern
      | via_reference
antenna ::=                                                    (see 9.16.2)
        ANTENNA antenna_identifier { antenna_items }
      | ANTENNA antenna_identifier ;
      | antenna_template_instantiation
antenna_items ::=
      antenna_item { antenna_item }
antenna_item ::=
      all_purpose_item
blockage ::=                                                   (see 9.16.3)
        BLOCKAGE blockage_identifier { blockage_items }
      | BLOCKAGE blockage_identifier ;
      | blockage_template_instantiation
blockage_items ::=
      blockage_item { blockage_item }
blockage_item ::=
        all_purpose_item
      | pattern
      | rule
      | via_reference
port ::=                                                       (see 9.16.4)
        PORT port_identifier { port_items }
      | PORT port_identifier ;
      | port_template_instantiation
port_items ::=
      port_item { port_item }
port_item ::=
        all_purpose_item
      | pattern
      | rule
      | via_reference
```

site ::= (see 9.17.1)
SITE *site*_identifier **{** site_items **}**
| **SITE** *site*_identifier **;**
| *site*_template_instantiation

site_items ::=
site_item { site_item }

site_item ::=
all_purpose_item
| *ORIENTATION_CLASS*_one_level_annotation
| *SYMMETRY_CLASS*_one_level_annotation

array ::= (see 9.17.2)
ARRAY *array*_identifier **{** array_items **}**
| **ARRAY** *array*_identifier **;**
| *array*_template_instantiation

array_items ::=
array_item { array_item }

array_item ::=
all_purpose_item
| *PURPOSE*_single_value_annotation
| geometric_transformation

pattern ::= (see 9.17.3)
PATTERN *pattern*_identifier **{** pattern_items **}**
| **PATTERN** *pattern*_identifier **;**
| *pattern*_template_instantiation

pattern_items ::=
pattern_item { pattern_item }

pattern_item ::=
all_purpose_item
| *SHAPE*_single_value_annotation
| *LAYER*_single_value_annotation
| *EXTENSION*_single_value_annotation
| *VERTEX*_single_value_annotation
| geometric_model
| geometric_transformation

artwork ::= (see 9.17.4)
ARTWORK = *artwork*_identifier **{** artwork_items **}**
| **ARTWORK =** *artwork*_identifier **;**
| *artwork*_template_instantiation

artwork_items ::=
artwork_item { artwork_item }

artwork_item ::=
geometric_transformation
| pin_assignment

geometric_model ::= (see 9.17.5)
nonescaped_dentifier [*geometric_model*_identifier]
{ geometric_model_items **}**
| *geometric_model_t*emplate_instantiation

geometric_model_items ::=
geometric_model_item { geometric_model_item }

geometric_model_item ::=
all_purpose_item
| *POINT_TO_POINT*_one_level_annotation
| coordinates

coordinates ::=
    COORDINATES { *x*_number *y*_number { *x*_number *y*_number } **}**

geometric_transformations ::= (see 9.17.6)
    geometric_transformation { geometric_transformation }

geometric_transformation ::=
     *SHIFT*_two_level_annotation
    | *ROTATE*_one_level_annotation
    | *FLIP*_one_level_annotation
    | repeat

repeat ::=
    REPEAT [**=** unsigned] **{**
       *shift_t*wo_level_annotation
       [repeat]
    }

function ::= (see 9.18.1)
     FUNCTION { function_items **}**
    | *function*_template_instantiation

function_items ::=
    function_item { function_item }

function_item ::=
    all_purpose_item
    | behavior
    | structure
    | statetable

test ::= (see 9.18.2)
     TEST { test_items **}**
    | *test*_template_instantiation

test_items ::=
    test_item { test_item }

test_item ::=
     all_purpose_item
    | behavior
    | statetable

behavior ::= (see 9.18.4)
     BEHAVIOR { behavior_items **}**
    | *behavior*_template_instantiation

behavior_items ::=
    behavior_item { behavior_item }

behavior_item ::=
     boolean_assignments
    | control_statement
    | primitive_instantiation
    | *behavior_item*_template_instantiation

boolean_assignments ::=
    boolean_assignment { boolean_assignment }

boolean_assignment ::=
    pin_variable **=** boolean_expression **;**

primitive_instantiation ::=
     *primitive*_identifier [identifier] **{** pin_values **}**
    | *primitive*_identifier [identifier]
        { boolean_assignments **}**

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

structure ::= (see 9.18.5)
STRUCTURE { named_cell_instantiations **}**
| *structure*_template_instantiation

named_cell_instantiations ::=
named_cell_instantiation { named_cell_instantiation }

named_cell_instantiation ::=
*cell*_identifier *instance*_identifier **{** pin_values **}**
| *cell*_identifier *instance*_identifier **{** pin_assignments **}**

violation ::= (see 9.18.6)
VIOLATION { violation_items **}**
| *violation*_template_instantiation

violation_items ::=
violation_item { violation_item }

violation_item ::=
*MESSAGE_TYPE*_single_value_annotation
| *MESSAGE*_single_value_annotation
| behavior

statetable ::= (see 9.18.7)
STATETABLE [identifier]
{ statetable_header statetable_row { statetable_row } **}**
| *statetable*_template_instantiation

statetable_header ::=
*input*_pin_variables **:** *output*_pin_variables **;**

statetable_row ::=
statetable_control_values **:** statetable_data_values **;**

statetable_control_values ::=
statetable_control_value { statetable_control_value }

statetable_control_value ::=
bit_literal
| based_literal
| unsigned
| edge_value

statetable_data_values ::=
statetable_data_value { statetable_data_value }

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

primitive ::= (see 9.18.8)
PRIMITIVE *primitive*_identifier **{** primitive_items **}**
| **PRIMITIVE** *primitive*_identifier **;**
| *primitive*_template_instantiation

primitive_items ::=
primitive_item { primitive_item }

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

A.5 Control definitions

boolean_expression ::= (see 10.7)
(boolean_expression **)**
| pin_value
| boolean_unary boolean_expression
| boolean_expression boolean_binary boolean_expression
| boolean_expression **?** boolean_expression **:**
{ boolean_expression **?** boolean_expression **:** }
boolean_expression

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

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

vector_expression ::= (see 10.8)
(vector_expression **)**
| vector_unary boolean_expression
| vector_expression vector_binary vector_expression
| boolean_expression **?** vector_expression **:**
{ boolean_expression **?** vector_expression **:** }
vector_expression
| boolean_expression control_and vector_expression
| vector_expression control_and boolean_expression

vector_unary ::=
edge_literal

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

control_and ::=
& | **&&**

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

A.6 Arithmetic definitions

arithmetic_expression ::= (see 11.1)
(arithmetic_expression **)**
| arithmetic_value
| [arithmetic_unary] arithmetic_expression
| arithmetic_expression arithmetic_binary
arithmetic_expression
| boolean_expression **?** arithmetic_expression **:**
{ boolean_expression **?** arithmetic_expression **:** }
arithmetic_expression
| arithmetic_macro
(arithmetic_expression { **,** arithmetic_expression } **)**

arithmetic_unary ::=
sign

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

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

arithmetic_models ::= (see 11.2.2)
arithmetic_model { arithmetic_model }

arithmetic_model ::=
partial_arithmetic_model
| non_trivial_arithmetic_model
| trivial_arithmetic_model

| assignment_arithmetic_model
| *arithmetic_model*_template_instantiation

partial_arithmetic_model ::= (see 11.2.3)
nonescaped_identifier [*arithmetic_model*_identifier] **{** partial_arithmetic_model_items **}**

partial_arithmetic_model_items ::=
partial_arithmetic_model_item { partial_arithmetic_model_item }

partial_arithmetic_model_item ::=
any_arithmetic_model_item
| table

non_trivial_arithmetic_model ::= (see 11.2.4)
nonescaped_identifier [*arithmetic_model*_identifier] **{**
[any_arithmetic_model_items]
arithmetic_body
[any_arithmetic_model_items]
}

trivial_arithmetic_model ::= (see 11.2.5)
nonescaped_identifier [*arithmetic_model*_identifier] **=** arithmetic_value **;**
| nonescaped_identifier [*arithmetic_model*_identifier] **=** arithmetic_value
{ any_arithmetic_model_items **}**

assignment_arithmetic_model ::= (see 11.2.6)
*arithmetic_model*_identifier **=** arithmetic_expression **;**

any_arithmetic_model_items ::= (see 11.2.7)
any_arithmetic_model_item { any_arithmetic_model_item }

any_arithmetic_model_item ::=
all_purpose_item
| from
| to
| violation

arithmetic_submodels ::= (see 11.3.1)
arithmetic_submodel { arithmetic_submodel }

arithmetic_submodel ::=
non_trivial_arithmetic_submodel
| trivial_arithmetic_submodel
| *arithmetic_submodel*_template_instantiation

non_trivial_arithmetic_submodel ::= (see 11.3.2)
nonescaped_identifier **{**
[any_arithmetic_submodel_items]
arithmetic_body
[any_arithmetic_submodel_items]
}

trivial_arithmetic_submodel ::= (see 11.3.3)
nonescaped_identifier **=** arithmetic_value **;**
| nonescaped_identifier **=** arithmetic_value **{** any_arithmetic_submodel_items **}**

any_arithmetic_submodel_items ::= (see 11.3.4)
any_arithmetic_submodel_item { any_arithmetic_submodel_item }

any_arithmetic_submodel_item ::=
all_purpose_item
| violation

arithmetic_body ::= (see 11.4.1)
arithmetic_submodels
| table_arithmetic_body
| equation_arithmetic_body

table_arithmetic_body ::=
header table [equation]

equation_arithmetic_body ::=
[header] equation [table]

header ::= (see 11.4.2)
HEADER { identifiers **}**
| **HEADER {** header_arithmetic_models **}**
| *header*_template_instantiation

header_arithmetic_models ::=
header_arithmetic_model { header_arithmetic_model }

header_arithmetic_model ::=
non_trivial_arithmetic_model
| partial_arithmetic_model

table ::= (see 11.4.3)
TABLE { arithmetic_values **}**
| *table*_template_instantiation

equation ::= (see 11.4.4)
EQUATION { arithmetic_expression **}**
| *equation*_template_instantiation

arithmetic_model_container ::= (see 11.5)
*arithmetic_model_container*_identifier **{** arithmetic_models **}**

from ::= (see 11.10.1)
FROM { from_to_items **}**

to ::=
TO { from_to_items **}**

from_to_items ::=
from_to_item { from_to_item }

from_to_item ::=
*PIN*_single_value_annotation
| *EDGE*_single_value_annotation
| *THRESHOLD*_arithmetic_model

*EARLY*_arithmetic_model_container ::= (see 11.10.2)
EARLY { *early_late*_arithmetic_models **}**

*LATE*_arithmetic_model_container ::=
LATE { *early_late*_arithmetic_models **}**

*early_late*_arithmetic_models ::=
*early_late*_arithmetic_model { *early_late*_arithmetic_model }

*early_late*_arithmetic_model ::=
*DELAY*_arithmetic_model
| *RETAIN*_arithmetic_model
| *SLEWRATE*_arithmetic_model

Annex B

(informative)

Bibliography

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

[B2] SPICE 2G6 User’s Guide.

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

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

Index

Symbols

(N+1) order sequential logic 175
-> operator 174
?- 26, 290
?! 26, 290
?? 26, 290
?~ 26, 290
@ 166

A

ABS 211
abs 210, 302
active vectors 170
ALF_AND 151
ALF_BUF 150
ALF_BUFIF0 153
ALF_BUFIF1 153
ALF_FLIPFLOP 155
ALF_LATCH 157
ALF_MUX 155
ALF_NAND 151
ALF_NOR 151, 152
ALF_NOT 150
ALF_NOTIF0 153, 154
ALF_NOTIF1 153, 154
ALF_OR 151
ALF_XNOR 151, 152
ALF_XOR 151, 152
ALIAS 38
alias 38, 293
all_purpose_items 36, 292
alphabetic_bit_literal 25, 290
annotation
    arithmetic model tables
        AREA 237
        CAPACITANCE 236
        CONNECTIONS 237
        CURRENT 235
        DELAY 234
        DERATE_CASE 237
        DISTANCE 237
        DRIVE_STRENGTH 235, 236
        DRIVER 282
        ENERGY 235
        FANIN 237
        FANOUT 237
        FREQUENCY 235
        HEIGHT 237
        HOLD 234
        JITTER 235
        LENGTH 238
        NOCHANGE 234
        PERIOD 234
        POWER 235
        PROCESS 237
        PULSEWIDTH 234
        RECEIVER 282
        RECOVERY 234
        REMOVAL 234
        RESISTANCE 236
        SETUP 234
        SKEW 234
        SLEWRATE 234
        SWITCHING_BITS 237
        TEMPERATURE 236
        THRESHOLD 235
        TIME 235
        VOLTAGE 236
        WIDTH 238
    arithmetic models 228
        average 278
        can_short 280
        cannot_short 280
        CONNECT_RULE 280
        DEFAULT 224
        MEASUREMENT 278
        must_short 280
        peak 279
        rms 279
        static 278
        transient 278
        UNIT 228
    CELL
        BUFFERTYPE 59
        CELLTYPE 53
        DRIVERTYPE 60

NON_SCAN_CELL 51, 295
PARALLEL_DRIVE 60
SCAN_TYPE 58
SCAN_USAGE 59
cell buffertype
inout 59
input 59
internal 59
output 59
cell celltype
block 53
buffer 53
combinational 53
core 53
flipflop 53
latch 53
memory 53
multiplexor 53
special 53
cell drivertype
both 60
predriver 60
slotdriver 60
cell scan_type
clocked 58
control_0 58
control_1 59
lssd 58
muxscan 58
cell scan_usage
hold 59
input 59
output 59
default 224
from 222
information
AUTHOR 50
DATETIME 50
PRODUCT 50
TITLE 50
VERSION 50
limit 222
object reference
cell 19
pin 19
primitive 19
PIN
ACTION 71
CONNECT_CLASS 80
DATATYPE 73
DIRECTION 66
DRIVETYPE 76
ORIENTATION 80
POLARITY 72
PULL 77
SCAN_POSITION 74
SCOPE 77
SIGNALTYPE 67
STUCK 74
VIEW 65
pin
PINTYPE 66
pin action
asynchronous 71
synchronous 71
pin datatype
signed 73
unsigned 73
pin direction
both 66, 67
input 66, 67
none 66, 67
output 66, 67
pin drivetype
cmos 76
cmos_pass 77
nmos 77
nmos_pass 77
open_drain 77
open_source 77
pmos 77
pmos_pass 77
ttl 77
pin orientation
bottom 80
left 80
right 80
top 80
pin pintype
analog 66
digital 66
supply 66

pin polarity
double_edge 72
falling_edge 72
high 72
low 72
rising_edge 72
pin pull
both 78
down 77
none 78
up 77
pin scope
behavior 77
both 77
measure 77
none 77
pin signaltype
clear 68, 72, 73
clock 68, 72, 73
control 68, 70, 72, 73
data 67, 72, 73
enable 68, 72, 73
master_clock 71
out_enable 69, 70
scan_clock 71
scan_data 69
scan_enable 70
scan_out_enable 70
select 68, 72, 73
set 68, 72, 73
slave_clock 71
pin stuck
both 74
none 74
stuck_at_0 74
stuck_at_1 74
pin view
both 66
functional 65
none 66
physical 66
to 222
VECTOR
LABEL 93, 94, 95
violation
MESSAGE 144
MESSAGE_TYPE 144
annotation container 39
anotation
object reference
class 19
any_character 22, 289
arithmetic models 14
arithmetic operators
binary 210
function 211
unary 210
arithmetic_binary_operator 210, 302
arithmetic_expression 209, 302
arithmetic_function_operator 210, 302
arithmetic_unary_operator 210, 302
atomic object 13
ATTRIBUTE 38
attribute 39, 293
CELL 53, 54, 55
cell
asynchronous 54
CAM 53
dynamic 54
RAM 53
ROM 53
static 53
synchronous 54
PIN 78
pin
PAD 78
SCHMITT 78
TRISTATE 78
XTAL 78

B

based literal 25
based_literal 26, 290
behavior 138, 299
behavior_body 138, 299
binary 25
Binary operators
arithmetic 210
bitwise 161
boolean, scalars 160
reduction 161
vector 175, 176, 179

binary_base 26, 290
binary_digit 26, 290
bit 25
bit_edge_literal 26, 290
bit_literal 25, 289
Bitwise operators
    binary 161
    unary 161
block comment 24
boolean operators
    binary 160
    unary 160
boolean_binary_operator 206, 301
boolean_expression 206, 301
boolean_unary_operator 206, 301

C

case-insensitive langauge 23
cell 51, 295
cell_identifier 51, 295
cell_items 51, 295
cell_template_instantiation 51, 295
characterization 5
children object 13
CLASS 40
class 40, 293
combinational logic 159
combinational primitives 150
combinational_assignments 138, 299
comment 23
    block 24
    long 24
    short 24
    single-line 24
comments
    nested 24
compound operators 23
CONSTANT 38
constant 38, 293
constant numbers 24
context-sensitive keyword 29

D

decimal 25
decimal_base 26, 290
deep submicron 5
default annotation 224, 228
delimiter 23, 289
digit 26, 290

E

edge literal 26
edge_literal 26, 290
edge_literals 31, 291
edge-sensitive sequential logic 166
equation 221, 304
equation_template_instantiation 221, 304
escape codes 27
escape_character 23, 289
escaped identifier 28
escaped_identifier 28, 291
event sequence detection 175
EXP 211
exp 210, 302
extensible primitives 148

F

Flipflop 155
function 133, 299
Function operators
    arithmetic 211
function_template_instantiation 133, 299
functional model 5

G

generic objects 14
GROUP 41
group 41, 293
group_identifier 41, 293

H

hard keyword 29
header 221, 304
header_template_instantiation 221, 304
hex_base 26, 290
hex_digit 26, 290
hexadecimal 25

I

identifier 13, 23
Identifiers 27
identifiers 27, 290

inactive vectors 170
INCLUDE 37
include 37, 293
index 33, 291
integer 24, 289

K

keyword 13
Keywords
    context-sensitive 30
    generic objects 29
    operators 29

L

Latch 157
level-sensitive sequential logic 166
Library creation 1
library_items 49, 294
library_template_instantiation 49, 294
library-specific objects 14
literal 13, 23
LOG 211
log 210, 302
logic_values 145, 300
logic_variables 34, 291

M

MAX 211
max 210, 302
MIN 211
min 210, 302
mode of operation 5
multiplexor 155

N

nested comments 24
non_negative_number 24, 289
non-escaped identifier 27
nonescaped_identifier 28, 290
nonreserved_character 23, 289
Number 24
number 24, 289
numeric_bit_literal 25, 290

O

objects 42, 293
octal 25
octal_base 26, 290
octal_digit 26, 290
operation mode 5
operator
    -> 174
    followed by 174
operators
    arithmetic 210
    boolean, scalars 160
    boolean, words 160
    signed 162
    unsigned 162

P

pin_assignments 33, 291
pin_identifier 61, 295
pin_items 61, 295
pin_template_instantiation 61, 295
placeholder identifier 28
placeholder_identifier 27
placeholders 43
power constraint 5
Power model 5
predefined derating cases 250, 262
    bccom 262
    bcind 262
    bcmil 262
    wccom 262
    wcind 262
    wcmil 263
predefined process names 262
    snsp 262
    snwp 262
    wnsp 262
    wnwp 262
primitive_identifier 138, 147, 299, 300
primitive_instantiation 138, 299
primitive_items 147, 300
primitive_template_instantiation 147, 300
private keywords 30
PROPERTY 39
property 39, 293
public keywords 30

Q

Q_CONFLICT 155
QN_CONFLICT 155
quoted string 22, 26
quoted_string 26, 290

R

real 24
Reduction operators
    binary 161
    unary 160
reserved keyword 29
reserved_character 22, 289
RTL 4

S

sequential logic
    edge-sensitive 166
    level-sensitive 166
    N+1 order 175
    vector-sensitive 174
sequential_assignment 138, 299
sign 24, 289
signed operators 162
simulation model 5
single-line comment 24
soft keyword 29
statetable 145, 300
statetable_body 145, 300
string 31, 291
symbolic_edge_literal 26, 290

T

table 221, 304
table_template_instantiation 221, 304
TEMPLATE 41
template 42, 293
template_identifier 42, 293
template_instantiation 42, 294
Ternary operator 160
timing constraints 5
timing models 5
triggering conditions 166
triggering function 166
tristate primitives 152

U

Unary operator
    bitwise 161
Unary operators
    arithmetic 210
    boolean, scalar 160
    reduction 160
Unary vector operators 168
unnamed_assignment 35, 292
unsigned 24, 289
unsigned operators 162

V

vector 90, 296
vector expression 174
Vector operators
    binary 175, 176
    unary, bits 168
    unary, words 169
vector_expression 90, 207, 296, 301
vector_items 90, 296
vector_template_instantiation 90, 296
vector_unary_operator 207, 301
vector-based modeling 5
Vector-Sensitive Sequential Logic 174
Verilog 4, 167
VHDL 4, 167
virtual pins 155

W

whitespace 22, 289
whitespace characters 22
wildcard_literal 25, 290
wire 81, 88, 97, 101, 104, 109, 113, 114, 117, 118, 121, 124, 296, 297, 298
wire_identifier 81, 88, 97, 101, 104, 109, 117, 296, 297, 298
wire_items 81, 88, 296
wire_template_instantiation 81, 88, 97, 101, 104, 109, 113, 114, 117, 118, 121, 124, 296, 297, 298
word_edge_literal 26, 290