Incremental specification of ALF 2.0, including physical library data

Author: Wolfgang Roethig

Purpose of this document

This document contains the proposed amendments and new features of ALF2.0, with ALF 1.1 as baseline. The first part contains amendments to ALF 1.1. The second part contains new features related to function and performance modeling proposed for ALF 2.0. The third part contains new features related to physical modeling proposed for ALF 2.0.

initial version reviewed Oct. 8, 1999

updated version reviewed Nov. 4, 1999

updated version reviewed Dec. 7, 1999

updated version staged Jan. 25, 2000

updated version staged Feb. 28, 2000

updated version staged April 3, 2000

1.0 Amendments to ALF 1.1

1.1 Incremental definitions for VECTOR

Status: proposal considered acceptable Oct.8. Supplementary proposal (see end of this chapter) acceptable.

Background:

In general, it is illegal to redeclare an ALF object (see ALF1.1, capter 3.7.1, 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 redeclare the context as well. In this way, new objects can be added at the end of the library instead of being inserted somewhere in the middle within the already declared context. The classical example is the VECTOR object, which defines the context for timing and power models. In a characterization process, timing models are always there in the 1st revision of a library, whereas power models are often added later. The new rule legalizes common practice within existing ALF libraries and tools. It makes it easier to add characterization data incrementally, for instance power data. It also facilitates conversion from and to legacy library formats.

Proposal:

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 declaration. All following instances shall be interpreted as supplemental definitions of the VECTOR. The rule of illegal redeclaration 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; } }
}
```

Supplementary Proposal:

Supplemental definitions of PROPERTY, ATTRIBUTE, LIBRARY, SUBLIBRARY shall be legal as well.

1.2 Timing arcs in the context of VECTOR

Status: proposal considered acceptable Oct. 8

Background:

In ALF library practice, timing models appear always in the context of a VECTOR (see ALF1.1, chapter 4.3). However, there is no normative rule to enforce this. Furthermore, there are no normative rule for the mandatory contents of a `vector_expression` in the context of which a particular timing model is used.

Proposal:

A timing model shall always appear in the context of a VECTOR. The following detailed rules shall apply.

Rule 1 shall apply for DELAY, RETAIN, SETUP, HOLD, RECOVERY, REMOVAL, SKEW (see ALF 1.1, chapter 3.6.7.1).

These models describe timing arcs, i.e., timing measurements or timing constraints between two transitions on two pins. The pins appear as annotations in the FROM and the TO field in the respective model (see ALF 1.1, chapter 3.6.8.1). Consequently, the `vector_expression` in the context of which the model appears shall contain [exactly / at least][1] two expressions of the type `vector_single_event` (see ALF1.1, chapter 3.4.5) with the FROM and TO pin, respectively, as operand. The sense of the timing arc, i.e., 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 `vector_single_event` expressions within the `vector_expression` shall have the following implications on the measurement data:

```
from_edge_literal from_pin -> to_edge_literal to_pin
// The data are positive.

to_edge_literal to_pin -> from_edge_literal from_pin
// The data are negative.

from_edge_literal from_pin &> to_edge_literal to_pin
// The data are positive or zero.

to_edge_literal to_pin &> from_edge_literal from_pin
// The data are negative or zero.

from_edge_literal from_pin <-> to_edge_literal to_pin
// The data are positive or negative.

from_edge_literal from_pin <&> to_edge_literal to_pin
// The data are positive or negative or zero.
```

1. “exactly” applies for ALF 1.1, “at least” applies for ALF 2.0 by introduction of EDGE_NUMBER

Rule 2 shall apply for SLEWRATE (see ALF 1.1, chapter 3.6.7.1).

This model describes a measurement of transition time on one pin. The pin is defined as annotation to the model. Consequently, the `vector_expression` in the context of which the model appears shall contain [exactly / at least][1] one expression of the type `vector_single_event` (see ALF1.1, chapter 3.4.5) with the pin as operand. The direction of the transition shall be identified by the `edge_literal`, i.e., the operator of the `vector_single_event`. The sense of measurement and hence the data shall always be positive.

Rule 3 shall apply for PULSEWIDTH, NOCHANGE (see ALF 1.1, chapter 3.6.7.1).

These models describe measurements between two subsequent transitions on one pin. The pin is defined as annotation to the model. Consequently, the `vector_expression` in the context of which the model appears shall contain [exactly / at least][2] one pair of expressions of the type `vector_single_event` (see ALF1.1, chapter 3.4.5) with the same pin as operand. The direction of the transitions shall be identified by the respective `edge_literal`, i.e., the operator of the respective `vector_single_event`. They must be different from each other, with the exception of "?!" which may be used to specify an arbitrary transition. The sense of measurement and hence the data shall always be positive.

Rule 4 shall apply for PERIOD (see ALF 1.1, chapter 3.6.7.1).

This model describes a measurement of periodically occuring events on one pin. The pin is defined as annotation to the model. Consequently, the `vector_expression` in the context of which the model appears shall contain a sequence of expressions of the type `vector_single_event` (see ALF1.1, chapter 3.4.5) with the same pin as operand allowing to be repeated periodically, i.e. the final state must be equal to the initial state of the pin. The subsequent states of the pin shall be identified by the respective `edge_literal`, i.e., the operator of the respective `vector_single_event`. The sense of measurement and hence the data shall always be positive.

1.3 Normative distinction between "driver resistance" and "pin resistance"

Status: proposal considered acceptable Oct.8

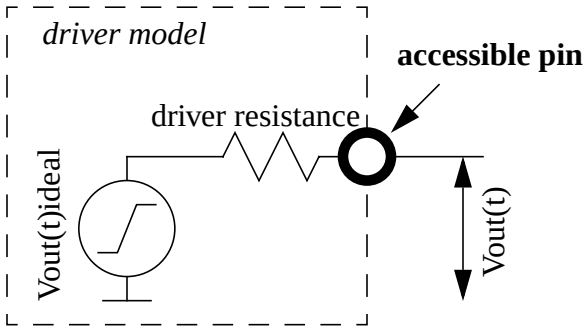
Background:

The semantic meaning of resistance is given in application notes (see ALF1.1, chapter 4.13.2, 4.14.1, 4.14.3), not in normative chapter.

Proposal 1:

1. "exactly" applies for ALF 1.1, "at least" applies for ALF 2.0 by introduction of EDGE_NUMBER
2. "exactly" applies for ALF 1.1, "at least" applies for ALF 2.0 by introduction of EDGE_NUMBER

The driver resistance of an output pin shall be a model in the context of a VECTOR. The PIN shall be an annotation to the model. In an electrically equivalent circuit, the driver resistance is in series to an ideal voltage source issuing the output signal. One terminal of the driver resistance shall be connected to the voltage source. The other terminal shall be connected to the accessible output pin itself. See following figure.

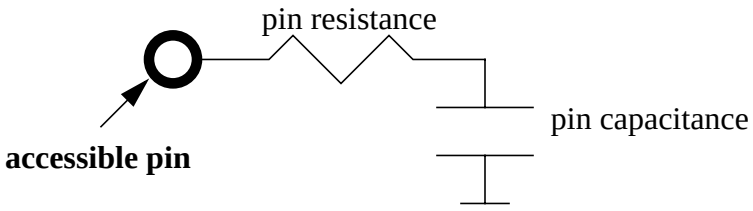


Example:

```
VECTOR ( 01 A -> 01 Z ) {
   RESISTANCE = 800 {
         UNIT = 1ohm;
         PIN = Z;
   }
}
```

Proposal 2:

A parasitic resistance on a pin shall be in the context of a PIN, in conjunction with a pin capacitance. One terminal of the resistance shall be connected to the pin capacitance. The other terminal shall be connected to the accessible pin itself. See following figure.



Example:

```
PIN A {
   DIRECTION = input;
   RESISTANCE = 200 { UNIT = 1ohm; }
   CAPACITANCE = 0.05 { UNIT = Picofarad; }
}  // the Elmor time constant is 10 picoseconds
```

1.4 Complex binary vector operators with N operands

Status: Proposal considered acceptable Nov.4

Background:

Commutative complex binary vector operators are defined (see ALF 1.1, chapter 3.5.4, table 3-19). The commutativity rules are only defined for 2 operands. Definition for N operands is necessary.

- 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 occuring”:

```
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
```

Proposal:

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 occurence 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 occurence 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
```

Note:

The following rule applies for boolean AND operation with 3 operands:

```
rule 1:
A & B & C === (A & B) & C | A & (B & C)
```

A corresponding rule also applies to the commutative followed-by operation with 3 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

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
```

1.5 Misc. amendments

Non-reserved character “#”

The character “#” is not mentioned in the lexical rules for ALF 1.1. The proposal is to declare it as non-reserved character. This will allow its legal usage in identifiers on equal foot with alphanumeric characters, dollar “$” and underscore “_”.

o.k.

Signaltype

See ALF 1.1, chapter 3.6.3.3.

Statement of purpose:

The purpose of SIGNALTYPE is to classify the functionality of a pin. The currently defined values apply for pins with PINTYPE=digital.

ALF 1.1, chapter 3.6.3.3 defines signaltypes for “flipflop or latch”, “multiplexor”, “memory or register file”. Following addition should be made: Flipflop, latch, multiplexor, memory, register file may be standalone cells or embedded in larger cells. In the former case, the celltype is flipflop, latch, multiplexor, memory, respectively. In the latter case the celltype may be “block” or “core”.

o.k.

Celltype

See ALF 1.1, chapter 3.6.5.1.

Statement of purpose:

The purpose of CELLTYPE is to classify the functionality of cells into broad categories. This is useful for information purpose, for tools which do not need the exact specification of functionality, and for tools which can interpret the exact specification of functionality only for certain categories of cells. The exact specification of the functionality is described in the FUNCTION statement.

Celltype “PAD” should be phased out, since it does not fit the purpose. PAD belongs to PLACEMENT_TYPE (see chapter 2.10).

Celltype “SPECIAL” should be defined as “Cell can only be used in certain application contexts not describable by the FUNCTION statement. Examples: Busholders, protection diodes, fillcells.”

Celltype “MEMORY” should be defined as “cell is a memory or a register file”.

o.k.

Celltype “BLOCK” should be defined as “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.”

STRUCTURE (see chapter 2) contains exclusively CELLs.

[Celltype BLOCK should migrate into PLACEMENT_TYPE]

Celltype “CORE” should be defined as “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.”

STRUCTURE (see chapter 2) contains PRIMITIVEs and eventually CELLs.

[Celltype CORE should be replaced by celltypes classifying the function, such as PLL, DSP, CPU ...]

Multiple non-scan cells

ALF 1.1, chapter 3.6.5.7 requires names for each NON_SCAN_CELL, e.g.

```
CELL scanff {
   NON_SCAN_CELL u1 = ff1 { ... }
   NON_SCAN_CELL u2 = ff2 { ... }
}
```

where `ff1`, `ff2` are the name of the non-scan equivalent cells for `u1`, and `u2`, are arbitrary names with the sole purpose of satisfying the rule of illegal redeclaration.

The proposal is either to relax this rule in the same way as for VECTOR etc. (see chapter 1.1 of this doc.) or to apply the multi_value_assignment concept, in order to get rid of the unnecessary names.

```
CELL scanff { // incremental definition of NON_SCAN_CELL
   NON_SCAN_CELL = ff1 { ... }
   NON_SCAN_CELL = ff2 { ... }
}
```

or

```
CELL scanff { // multi_value_assignment for NON_SCAN_CELL
   NON_SCAN_CELL { ff1 { ... } ff2 { ... } }
}
```

Preference needs to be decided, based on practicality (ease of implementation, compatibility with existing parser etc.)

Multi_value_assignment is the prefered concept.

Multi_value_assignment in PROPERTY

Currently, a PROPERTY statement (see ALF 1.1, chapter 3.4.7) can only contain assignments with single values.

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

The following amendment is proposed:

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

property_items ::=
   property_item { property_item }

property_item ::=
   unnamed_assignment
|  multi_value_assignment
```

Example:

```
PROPERTY {
   my_param1 = value1;
   my_param2 { val1 val2 val3 }
   my_param3 = value4;
}
```

1.6 Items to be phased out for ALF 2.0

SCAN annotation container

see ALF 1.1, chapter 3.6.1.1.

Reason: not required for DFT, since all DFT items are already identified by dedicated keywords.

Substitution: not required

o.k.

OFF_STATE annotation

see ALF 1.1, chapter 3.6.3.16.

Reason: purpose of OFF_STATE is not well-defined and basically unknown.

Substitution: not required

o.k.

SCAN_USAGE annotation

see ALF 1.1, chapter 3.6.5.6.

[Currently not in DFT requirement, to be decided.]

ENABLE_PIN annotation

see ALF 1.1, chapter 3.6.3.9.

Currently not in DFT requirement. The ENABLE_PIN annotation provides a very limited capability to describe a relationship between two pins, which is normally not described as an annotation for a pin. Relationships between pins can be described using VECTOR, supplemented by new features in ALF 2.0. (see chapter 2.5 of this document).

Also, the ENABLE_PIN can make a reference to a pin which may not yet be declared. This clashes with the general rule: an object shall not be referenced before it is declared.

Substitution:

For cells with `CELLTYPE = buffer | combinational | latch | flipflop` the following rule applies:

> For a PIN with `SIGNALTYPE = data` and `DIRECTION = output | both`, the PIN with `SIGNALTYPE = out_enable` is the enable-pin.
>
> For a PIN with `SIGNALTYPE = scan_data` and `DIRECTION = output | both`, the PIN with `SIGNALTYPE = scan_out_enable` is the enable-pin.

For cells with `CELLTYPE = memory` the following rule applies:

> For a PIN with `SIGNALTYPE = data` and `DIRECTION = output | both`, the PIN with `SIGNALTYPE = read_enable` is the enable-pin.
>
> For a PIN with `SIGNALTYPE = test_data` and `DIRECTION = output | both`, the PIN with `SIGNALTYPE = test_read_enable` is the enable-pin.

Port-specific enable-pins in multiport memories must have the same `SIGNAL_CLASS` as the related output pin (see chapter 2.6).

POLARITY for OUTPUT signal

see ALF 1.1, chapter 3.6.3.8, table 3-35.

Reason: Not required by any tool today. Applies to very few signals in a library (e.g. inverted and non-inverted output of flipflop). Different semantics than polarity for input signal, therefore potentially confusing.

Substitution:

The output polarity applies only for cells with `CELLTYPE = latch | flipflop`.
It describes the relationship between two pins.
Both pins have `SIGNALTYPE = data | scan_data`.

One PIN has `DIRECTION = output`, the other PIN has `DIRECTION = input`.
The BEHAVIOR statement contains an assignment in the following form:

```
output_pin_identifier = boolean_expression ;
```

The `boolean_expression` contains at least one sub-expression of the following form.

```
[! | ~] input_pin_identifier
```

The systematic presence or absence of the inversion-operator `[! | ~]` in each sub-expression involving *`input_pin`*`_identifier` defines the polarity as inverted or non-inverted.

Examples:

```
BEHAVIOR {
   @ ( 01 clk ) {
         Q = D ; // non-inverted
         QN = ! D ; // inverted
   }
}

BEHAVIOR {
   @ ( 01 clk ) {
         Q = (! sync_reset) & D ; // non-inverted
         Q_scan = scan_enb ? D_scan : Q_scan ; // non-inverted
   }
}
```

The polarity relationship between input and output may also be indirectly defined through the relationship between two outputs, provided a polarity relationship between one output and one input is defined as well:

```
output_pin_identifier = [! | ~] other_output_pin_identifier;
```

Examples:

```
BEHAVIOR {
   @ ( 01 clk ) {
         Q = D ; // non-inverted
   }
   QN = ! Q ; // inverted
}

BEHAVIOR {
   @ ( 01 clk ) {
         QN = ! D ; // inverted
   }
   Q = ! QN ; // non-inverted
}
```

When `primitive_instantiation` statements are used, the inverting or non-inverting output can be directly identified by the pin mapping between the model and the instance.

ATTRIBUTE READ, WRITE, TIE with POLARITY

see ALF 1.1, chapter 3.6.6.2, table 3-48.

Reason: Not a very concise modeling style . Also, this is the only case where ATTRIBUTE contains non-atomic objects. By removing this special case, ATTRIBUTE will contain only atomic objects, which simplifies the datamodel.

Substitution: see chapter 2.5 of this document.

PRIMITIVE definition in FUNCTION

see ALF 1.1, chapter 3.4.16

Current BNF:

```
function ::=
   FUNCTION [ identifier ] {
         [ all_purpose_items ]
         [ primitives ]
         function_description
   }

function_description ::=
   behavior
|  [ behavior ] statetables
```

Proposed change:

```
function ::=
   FUNCTION [ identifier ] {
         [ all_purpose_items ]
         function_description
   }

function_description ::=
   behavior
|  [ behavior ] statetables
```

Reason: PRIMITIVE definitions must contain a FUNCTION statement themselves. Therefore, the possibility of having PRIMITIVE inside FUNCTION and FUNCTION inside PRIMITIVE bears the potential risk of circular reference in the datamodel.

Substitution: use PRIMITIVE definitions inside the CELL which contains the FUNCTION.

o.k.

1.7 Use of STATETABLE for ROM initialization

The STATETABLE statement (see ALF 1.1, chapter 3.4.16) can be used to describe the contents of a ROM, as far as this content is fixed in the library.

The following example may be included in an application note.

Example:

```
CELL my_rom {
   CELLTYPE = memory;
   ATTRIBUTE { rom asynchronous }
   PIN[1:2] addr { DIRECTION = input; SIGNALTYPE = address; }
   PIN[7:0] dout { DIRECTION = output; SIGNALTYPE = data; }
   PIN[7:0] 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 may be used:

```
CELL my_rom {
   CELLTYPE = memory;
   ATTRIBUTE { rom asynchronous }
   PIN[1:3] addr { DIRECTION = input; SIGNALTYPE = address; }
   PIN[7:0] dout { DIRECTION = output; SIGNALTYPE = data; }
   PIN[7:0] mem[1:4] { DIRECTION=none; VIEW=none; SIGNALTYPE=data; }
   FUNCTION {
         BEHAVIOR { dout = mem[addr]; }
         INCLUDE "rom_initialization_file.alf"
         }
   }
}
```

Contents of the included file "rom_initialization_file.alf":

```
         STATETABLE {
               addr  :     mem ;
               'h0   :     'h5 ;
               'h1   :     'hA ;
               'h2   :     'h5 ;
               'h3   :     'hA ;
         }
```

o.k.

2.0 New features for ALF 2.0

2.1 EDGE_NUMBER for timing arc

Status: Proposal considered acceptable Oct. 8

New examples added Jan.6

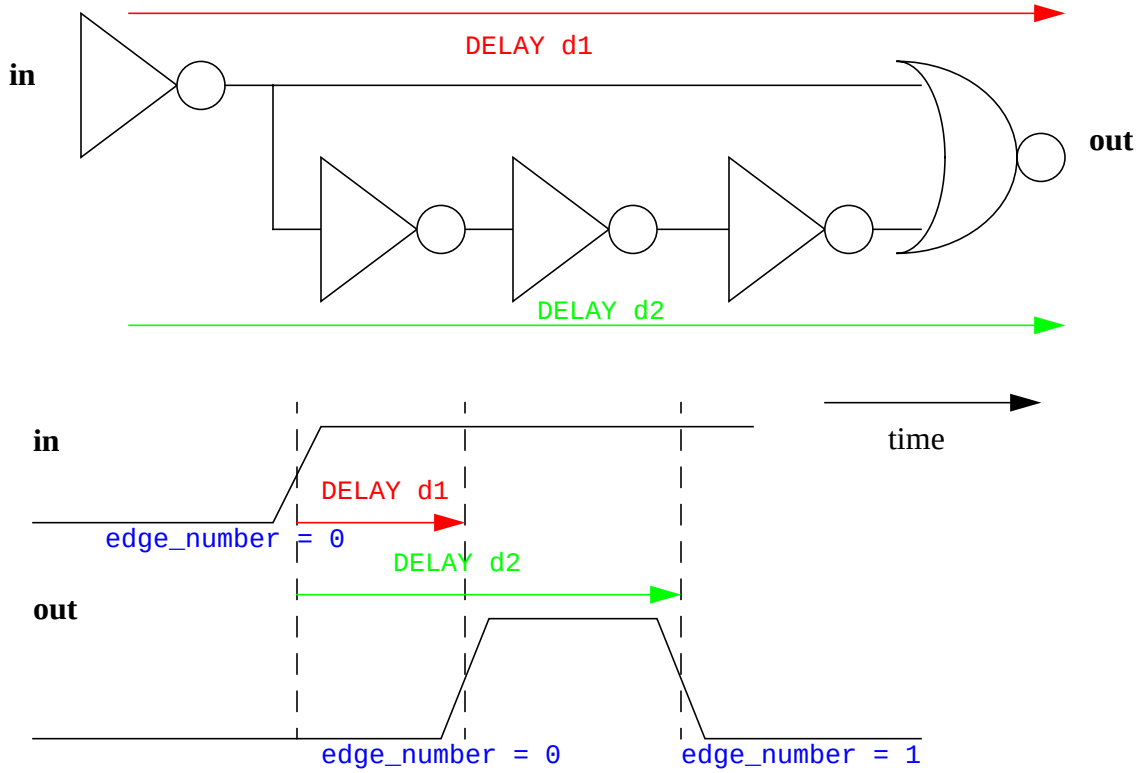
Background:

A VECTOR may contain more than one `vector_single_event` on a particular pin. In this case, a timing arc is not completely specified by the pin annotation(s) alone. An additional specification is necessary.

Proposal:

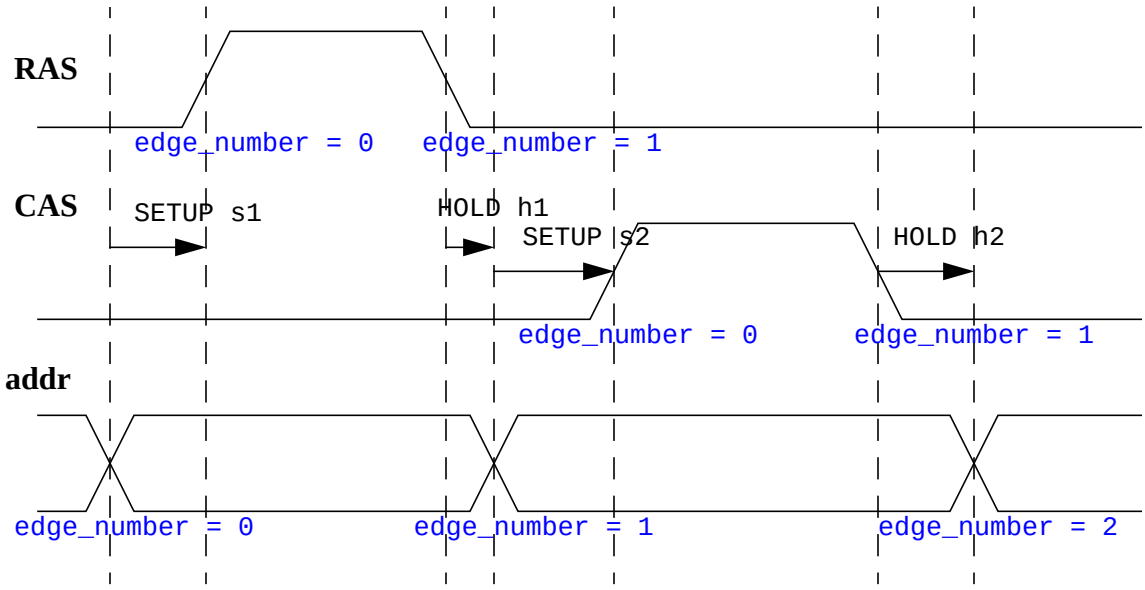
The EDGE_NUMBER annotation within the context of a timing model shall specify the edge for which the timing measurement applies. Exactly one PIN annotation shall always appear in the same context as the EDGE_NUMBER. The EDGE_NUMBER shall have an unsigned value pointing to exactly one of subsequent `vector_single_event` expressions applicable to the referenced pin. The EDGE_NUMBER shall be counted individually for each pin which appears in the VECTOR, starting with zero.

Example 1: 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; }
   }
}
```

Example 2: DRAM timing diagram




```
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; }
   }
}
```

Supplementary proposal:

Default value for EDGE_NUMBER in both the FROM and TO field of DELAY, RETAIN, SETUP, HOLD, RECOVERY, REMOVAL, SKEW shall be 0.

Default value for EDGE_NUMBER in SLEWRATE shall be 0.

Default value for EDGE_NUMBER in the FROM field of PULSEWIDTH, NOCHANGE shall be 0.

Default value for EDGE_NUMBER in the TO field of PULSEWIDTH, NOCHANGE shall be 1.

Default value for EDGE_NUMBER in PERIOD is not required.

2.2 STRUCTURE statement

Proposal reviewed Oct. 8, amended proposal considered acceptable Nov. 4

Background:

The purpose of this proposal is to describe the structure of a complex cell composed of atomic cells, for example I/O buffers, LSSD flipflops, clock trees.

Proposal:

An optional STRUCTURE statement shall be legal in the context of FUNCTION. The STRUCTURE statement shall describe a netlist of components inside the CELL. The STRUCURE statement shall not substitute the BEHAVIOR statement. If a FUNCTION contains only a STRUCTURE statement and no BEHAVIOR statement, it shall be concluded that a behavior description for that particular cell is meaningless (example: fillcells, diodes, vias, analog cells ...).

Also, 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 may use the structural information in order to define a one-to-many mapping for scan cell replacement, such as where a single flip-flop will be 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 will contain the NON_SCAN_CELL annotation to define which sequential cells it is the replacement for. No timing model will be required for this macro cell, since it should be treated as a transparent hierarchy level in the design netlist after test synthesis.

The syntax for FUNCTION statement (see ALF1.1, chapter 3.4.16) shall be augmented as follows:

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

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 }
```

Notes:

Every *instance*_identifier within a STRUCTURE statement must be different from each other.

The STRUCTURE statement provides a directive to the application (e.g. synthesis, DFT) how the CELL is implemented. A CELL referenced in named_cell_instantiation may be replaced by another CELL within the same SWAP_CLASS and RESTRICT_CLASS recognized by the application.

The *cell*_identifier within a STRUCTURE statement may refer to actual cells as well as to primitives. The usage of primitives is recommended in fault modeling for DFT.

BEHAVIOR statements also provide the possibility of instantiating primitives. However, those instantiations are for modeling purpose 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 flipflop = latch + flipflop + 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_flipflop {
   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_flipflop {
   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_flipflop 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_flipflop {
         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; }
      }
   }
}
```

2.3 Hierarchical identifier “.”

Status: new proposal Jan. 2000. Replaces and generalizes the proposal in “reference of a declared PORT inside a declared PIN” in chapter 3.1 of this document.

Proposal:

The lexical rules for identifier (see ALF spec. 1.1, chapter 3.2.12) shall be augmented as follows:

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

with no whitepace in-between.

The dot shall take precedence over the escape_character. In order to escape the dot, the escape_character must be placed directly in front of the dot.

Examples:

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

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

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

o.k.

2.4 NODE statement in the context of WIRE

Status: Proposal reviewed Nov. 4, usage of WIRE keyword to be discussed, named or unnamed CAPACITANCE, RESITANCE to be discussed, acceptable otherwise.

Jan. 2000: Separated from PORT statement in the context of PIN, which is for layout. Proposal has been modified to support interconnect delay models. PORT in the context of WIRE renamed to NODE.

Background:

For complex cells, especially in hierarchical design, modeling the electrical properties of a pin as a lumped C or RC (see this document, chapter 1.3) may not suffice. A PIN may be modeled as several electrical NODEs, and a considerable amount of routing wire may exist between the NODEs. Therefore a description capability of distributed RC or RLC components between the NODEs is needed.

Note, that SPEF (see IEEE 1481 specification) is not the most appropriate way to describe such components. The purpose of SPEF is to describe RLC components extracted from a design, not components which are part of a cell in the library. Moreover, SPEF can repre-

sent those components only as single numbers, whereas ALF may describe them as arithmetic models of temperature, process etc.

Also it is desireable that ALF can be used as a self-sufficient source to OLA (see IEEE 1481 specification). OLA has a call *getPinAdmittance*, which returns a RLC network. Therefore the equivalent description capability of an RLC network should also exist in ALF.

The most natural way for ALF is to put the arithmetic models for RLC components into a WIRE object inside a CELL object. This is in tune with the semantics of WIRE, which is a container of arithmetic models such as RESISTANCE and CAPACITANCE. Nothing precludes those models from being a description of *exact* RC values as opposed to *statistical* RC values in the context of a wireload model. The exact RC values must have connectivity information in order to be meaningful.

Proposal:

A WIRE statement may contain a NODE statement of the following form:

```
node ::=
   NODE node_identifier [ = node_purpose_identifier ] ;

node_purpose_identifier ::=
   ground
|  power
|  driver
|  receiver
```

A NODE shall be visible inside the WIRE only. The purpose of **ground** is to define an ideal ground for electrical modeling. The purpose of **power** is to define an ideal power supply for electrical modeling. The signal voltages for logic high and low, respectively, of ideal drivers shall be identical to the voltage level of **power** and **ground**, respectively. The purpose of **driver** and **receiver** is to define a driver or a receiver for electrical modeling, respectively. An internal node without *node_purpose*_identifier must only be declared, if measurements are made with respect to this node. Internal nodes for the purpose of describing a netlist of parasitics only need not be declared.

CAPACITANCE, RESISTANCE, INDUCTANCE statements inside WIRE may contain the following multi_value_assignment (see ALF1.1, chapter 3.4.1):

```
two_node_multi_value_assignment ::=
   NODE { node_identifier node_identifier }
```

where *node*_identifier is one of the following:

- a simple identifier, refering to a declared PIN of the CELL.
- a hierarchical_identifier, refering to a declared PORT of a PIN of the CELL
- a simple identifier, refering to a declared PORT of the WIRE.

a simple `identifier`, not refering to a declared object. Can be used for connectivity inside the WIRE only.

The purpose is to define the CAPACITANCE, RESISTANCE, INDUCTANCE objects to be connected components rather than statistical models.

Note:

Both PIN assignments (e.g. `PIN=A;`) and NODE assignments (e.g. `NODE { A B }`) may refer to PINs or PORTs.
The fundamental semantic difference between a PIN assignment and a NODE assignment is the following: The PIN assignment within an object defines that the object is *applied* or *measured* at the PIN or PORT. (e.g. DELAY, SLEWRATE). The NODE assignment within an object defines that the object is fundamentally *connected* with the PIN or PORT, in the same way as 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. See also the differentiation between pin resistance and driver resistance, chapter 1.3 of this document.

Description of boundary parasitics for a CELL

In the context of a WIRE inside a CELL, values or arithmetic models may be given for the connected components. If a refered PIN contains also a value or an arithmetic model for CAPACITANCE or RESISTANCE, the latter shall be considered as a reduced form of the former. A CAPACITANCE inside a PIN is not to be added or combined in any other way with the CAPACITANCE in the WIRE. Either the components for the PIN or the connected components for the WIRE shall be used in a mutually exclusive way.

Example:

```
CELL my_cell {
   PIN A {
         CAPACITANCE = 4.8;
         RESISTANCE = 37.9;
         PORT p1 { VIEW = physical; }
         PORT p2 { VIEW = none; }
   }
   PIN B {
         CAPACITANCE = 2.6;
         PORT { VIEW = physical; }
   }
   WIRE my_boundary_parasitics {
         NODE gnd = ground;
         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 } }
   }
}
```

Description of interconnect DELAY models

In the context of a VECTOR inside a WIRE, models for DELAY and SLEWRATE may be described. The PIN assignments in these models shall refer to declared NODEs inside the WIRE. Connected components with NODE assignments may be used as arguments of the models.

Example:

```
WIRE simple_interconnect_interconnect_model {
   NODE a = driver;
   NODE z = receiver;
   NODE gnd = ground;
   VECTOR ( (01 a -> 01 z) | (10 a -> 10 z) ) {
         DELAY { // analytical delay model
               FROM { PIN = a; }
               TO { PIN = z; }
               CALCULATION = absolute;
               HEADER {
                     RESISTANCE  r1 { NODE { a b} }
                     CAPACITANCE c1 { NODE { b gnd } }
                     RESISTANCE  r2 { NODE { b z} }
                     CAPACITANCE c2 { NODE { z gnd} }
               }
               EQUATION { r1*(c1+c2) + r2*c2 }
         }
         SLEWRATE { // slewrate degradation table
               PIN = z;
               CALCULATION = absolute;
               HEADER {
                     SLEWRATE { PIN = a; TABLE {/* data */} }
                     RESISTANCE { NODE { a z } TABLE {/* data */} }
                     CAPACITANCE { NODE { z gnd } TABLE {/* data */} }
               }
               TABLE { /* fill in data */ }
         }
   }
}
```

The DELAY and SLEWRATE models apply both for rise and fall. Both models represent absolute numbers (see chapter 2.10 of this document for the CALCULATION annotation).

Description of interconnect noise models

In the context of a VECTOR inside a WIRE, models for noise VOLTAGE may be described as well as models for DELAY and SLEWRATE related to noise.

```
WIRE interconnect_model_with_coupling {
   NODE aggressor_source = driver;
   NODE victim_source = driver;
   NODE aggressor_sink = receiver;
   NODE victim_sink = receiver;
   NODE vdd = power;
```

```
    NODE gnd = ground;
    VECTOR ( 01 aggressor_sink && ?- victim_sink ) {
          VOLTAGE { MEASUREMENT = peak; PIN = victim_sink;
                CALCULATION = incremental;
                HEADER {
                      SLEWRATE   tra { PIN = aggressor_sink; }
                      CAPACITANCE cc { NODE {aggressor_sink victim_sink}}
                      CAPACITANCE cv { NODE {victim_sink gnd }}
                      RESISTANCE  rv { NODE {victim_source victim_sink}}
                      VOLTAGE     va { NODE {vdd gnd} }
                      }
                EQUATION { (1-EXP(-tra/(rv*cv)))*va*rv*cc/tra }
          }
    }
    VECTOR (
          ( 01 aggressor_source <&> 01 victim_source )
           -> 01 aggressor_sink -> 01 victim_sink
    ) {
          DELAY { FROM { PIN = victim_source; } TO { PIN = victim_sink; }
                CALCULATION = incremental;
                HEADER {
                      SLEWRATE tra { PIN = aggressor_sink; }
                      SLEWRATE trv { PIN = victim_source; }
                      CAPACITANCE cc { NODE {aggressor_sink victim_sink}}
                      RESISTANCE rv  { NODE {victim_source victim_sink}}
                }
                EQUATION { (1-EXP(-tra/(rv*cv)))*rv*cc*trv/tra }
          }
    }
}
```

The VOLTAGE model applies for rising aggressor signal while the victim signal is stable. The DELAY model applies for rising victim signal simultaneous with or followed by rising aggressor signal at the coupling point. Note that the VECTOR implicitely defines the time window of interaction between aggresssor 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 (see chapter 2.10 of this document for the CALCULATION annotation).

2.5 SIGNALTYPE, OPERATION, SUPPLYTYPE

Background:

The definition of SIGNALTYPE does not follow a coherent strategy. On one hand, new signaltypes for PINs are proposed in order to enumerate all possible functionality, (examples: SIGNALTYPE=SCAN_ENABLE, SIGNALTYPE=SCAN_OUT_ENABLE), on the other hand, construction of complex functionality by assigning multiple signaltypes to a PIN is proposed (examples: SIGNALTYPE { READ CLOCK}, SIGNALTYPE { READ WRITE}).

Therefore systematic way of defining SIGNALTYPEs is proposed, consisting of a set of fundamental signaltypes and prefixes for special signaltypes, with minimal change in existing definitions.

The OPERATION statement is introduced to define operations which are not controlled by a single pin.

The SUPPLYTYPE statement is introduced as an analogon to SIGNALTYPE, applicable to power/ground pins.

Proposal for SIGNALTYPE:

A pin with PINTYPE = DIGITAL shall have a SIGNALTYPE annotation. The values shall consist of a set of fundamental values and composite values involving prefixes and infixes.

The following fundamental values for SIGNALTYPE shall be defined:

- DATA (supported in ALF 1.1)
  no POLARITY annotation
  Definition: a signal that carries information to be transmitted, received, or subjected to logic operations within the CELL.

o.k.

- CONTROL (supported in ALF 1.1)
  no single POLARITY annotation, mode-specific polarity (see below) or VECTOR with OPERATION annotation is applicable.
  Definition: 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, however, the signal is not necessarily timing-critical.
- TIE (new for ALF 2.0, replaces the ATTRIBUTE { TIE } from ALF 1.1)
  POLARITY = HIGH | LOW or VECTOR with OPERATION annotation is applicable.
  Definition: a signal that 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. In the fixed mode, the POLARITY defines the required state of the PIN. The signal value is not allowed to change during real-time circuit operation.
- SELECT (supported in ALF 1.1)
  no POLARITY annotation, VECTOR with OPERATION annotation is applicable.
  Definition: 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.
- ADDRESS (supported in ALF 1.1)
  no POLARITY annotation
  Definition: an encoded signal, usually a bus or part of a bus, driving an address decoder within the CELL.

o.k.

- CLOCK (supported in ALF 1.1)
  POLARITY = HIGH | LOW | RISING_EDGE | FALLING_EDGE | DOUBLE_EDGE or mode-specific polarity (mandatory)
  Definition: a timing-critical signal that triggers data storage within the CELL.
- ENABLE (supported in ALF 1.1)
  POLARITY = HIGH | LOW (mandatory)
  Definition: a decoded signal which enables and disables a set of operational modes of the CELL, eventually in conjunction with other signals. The POLARITY defines the state of the PIN when the set is enabled.
  POLARITY=HIGH means: set is exclusively enabled when signal is high, set is disabled while no other set is enabled when signal is low.
  POLARITY=LOW means: set is exclusively enabled when signal is low, set is disabled while no other set is enabled when signal is high.
  The signal value is expected to change during real-time circuit operation.
- SET (supported in ALF 1.1)
  POLARITY = HIGH | LOW (mandatory)
  Definition: a signal that controls the storage of the value “1” within the CELL.

o.k.

- CLEAR (supported in ALF 1.1)
  POLARITY = HIGH | LOW (mandatory)
  Definition: a signal that controls the storage of the value “0” within the CELL.

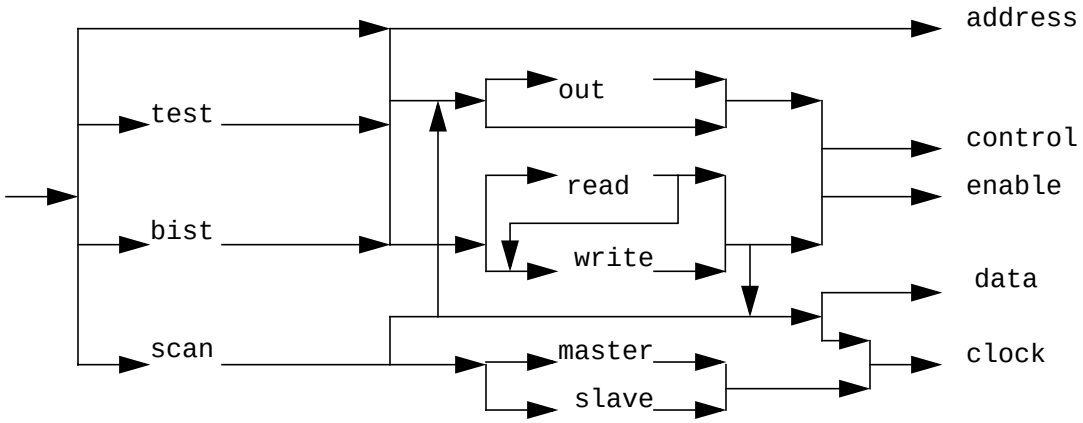
o.k.

The PINs with SIGNALTYPE=CONTROL | ENABLE | SET | CLEAR shall have ACTION annotation (supported in ALF 1.1).

ACTION = ASYNCHRONOUS, if self-triggered
ACTION = SYNCHRONOUS, if triggered by a CLOCK signal.

Composite values for SIGNALTYPE shall be constructed using one or more prefixes in combination with the fundamental values, separated by the underscore “_” character.

FIGURE 1. Construction scheme for composite SIGNALTYPE values



The following prefixes and infixes shall be defined:

- prefix or infix OUT (supported in ALF 1.1)
  The target of control for the signal is a PIN with DIRECTION=OUTPUT | BOTH within the CELL. Combinable with signaltypes CONTROL, ENABLE.

  OUT_CONTROL: controls visibility of data at an output or bidirectional pin.

  OUT_ENABLE: enables visibility of data at an output or bidirectional pin.

  Note: OUT_DATA is not supported, since this would be redundant with SIGNAL-TYPE=DATA and DIRECTION=OUTPUT.

- prefix or infix MASTER, SLAVE (supported in ALF 1.1)
  Combinable with signaltype CLOCK.

  MASTER_CLOCK: the signal is a master clock within a master-slave clocking scheme within the CELL

  SLAVE_CLOCK: the signal is a slave clock within a master-slave clocking scheme within the CELL

- prefix or infix READ, WRITE (new for ALF 2.0, replaces SIGNALTYPE = READ | WRITE from ALF 1.1
  Qualifies the signal to be relevant for a read or write operation, respectively. Combinable with each other. Combinable with signaltypes CONTROL, CLOCK, ENABLE. Not combinable with prefix OUTor SCAN.

  READ_CONTROL, WRITE_CONTROL, READ_WRITE_CONTROL:
  controls read operation, write operation or both, respectively.

  READ_ENABLE, WRITE_ENABLE, READ_WRITE_ENABLE:
  enables read operation, write operation or both, respectively.

  READ_CLOCK, WRITE_CLOCK, READ_WRITE_CLOCK:
  clock for read operation, write operation or both, respectively.

- prefix SCAN (supported in ALF 1.1)
  Qualifies the signal to be relevant for scan. Combinable with fundamental or composite signaltypes involving DATA, CONTROL, ENABLE, CLOCK. Exception: Not combin-able with READ, WRITE.

SCAN_DATA: data for scan mode

SCAN_CONTROL: signal controls the scan mode

SCAN_OUT_CONTROL: signal controls the scan output

SCAN_ENABLE: signal enables the scan mode

SCAN_OUT_ENABLE: signal enables the scan output

SCAN_CLOCK: clock for scan mode

SCAN_MASTER_CLOCK, SCAN_SLAVE_CLOCK:
master or slave clock, respectively, for scan mode.

- prefix TEST, BIST (new for ALF 2.0)
  Qualifies the signal to be relevant for test or bist, respectively. Combinable with fundamental or composite signaltypes involving DATA, CONTROL, ADDRESS, CLOCK, ENABLE. Not combinable with each other. Not combinable with SCAN.

  TEST_DATA: data signal for test mode

  TEST_CONTROL: control signal for test mode

  TEST_OUT_CONTROL: output control signal for test mode

  TEST_READ_CONTROL, TEST_WRITE_CONTROL,
  TEST_READ_WRITE_CONTROL:
  control signal for read, write, or both, respectively, in test mode

  TEST_ADDRESS: address signal for test mode

  TEST_CLOCK: clock signal for test mode

  TEST_MASTER_CLOCK, TEST_SLAVE_CLOCK:
  master or slave clock signal, respectively, for test mode

  TEST_READ_CLOCK, TEST_WRITE_CLOCK, TEST_READ_WRITE_CLOCK:
  clock signal for read, write, or both, respectively, in test mode

  TEST_ENABLE: enable signal for test mode

  TEST_OUT_ENABLE: output enable signal for test mode

  TEST_READ_ENABLE, TEST_WRITE_ENABLE,
  TEST_READ_WRITE_ENABLE:
  enable signal for read, write, or both, respectively, in test mode

- prefix BIST (new for ALF 2.0)
  Qualifies the signal to be relevant for build-in-self-test. Combinable with the same composite or fundamental signaltypes as “TEST”. Prefix “TEST”, “SCAN”, “BIST” are mutually exclusive.

A pin with PINTYPE = ANALOG may also have SIGNALTYPE annotation. However, no values are currently defined.

Proposal for mode-specific POLARITY:

Signals with composite signaltypes *mode*_CONTROL may have mode-specific polarities. Signals with composite signaltypes *mode*_CLOCK may 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; }
}
```

Proposal for OPERATION:

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 ;
```

OPERATION within the context of 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.

For a cell with CELLTYPE=memory, the following values shall be predefined:

```
operation_identifier ::=
   read
|  write
|  read_modify_write
|  write_through
|  start
|  end
|  refresh
|  load
|  iddq
```

- 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 modifing it
- load: operation for loading control registers
- iddq: operation for supply current measurements in quiescent state

The EXISTENCE_CLASS (see ALF 1.1, chapter 3.6.4.3) within the context of 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:

```
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 “read” is disabled.

Example:

```
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
}
```

Proposal for SUPPLYTYPE:

A PIN with PINTYPE = SUPPLY shall have a SUPPLYTYPE annotation.

```
supplytype_assignment ::=
   SUPPLYTYPE = supplytype_identifier ;

supplytype_identifier ::=
   power
|  ground
|  bias
```

o.k.

2.6 New usage models for CLASS

Background:

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, CHARACTERIZATION_CLASS) also carries a specific semantic meaning in the context of its usage. Note that the keyword <prefix>_CLASS is used for reference of the class, whereas the definition of the class always uses the keyword CLASS. Thus a class may 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.

Proposal for PURPOSE:

A CLASS object may contain the PURPOSE annotation, which can take one or multiple values. A VECTOR entitled to inherit the PURPOSE annotation from the CLASS may also contain the PURPOSE annotation.

```
vector_purpose_assignment ::=
   PURPOSE { purpose_identifier { purpose_identifier } }

vector_purpose_identifier :: =
   bist
|  test
|  timing
|  power
|  integrity
```

Proposal for SIGNAL_CLASS:

The following new keyword for class reference shall be defined:

- SIGNAL_CLASS
  PIN refering to the same SIGNAL_CLASS belong to the same logic port.
  For example the ADDRESS, WRITE_ENABLE and DATA pin of a logic port of a memory have the same SIGNAL_CLASS.
  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.

Proposal for SUPPLY_CLASS:

The following new keyword for class reference shall be defined:

- SUPPLY_CLASS
  PIN refering to the same SUPPLY_CLASS belong to the same power terminal.
  For example, digital VDD and digital VSS have the same SUPPLY_CLASS.
  SIGNAL_CLASS applies to a PIN with PINTYPE=SUPPLY.
  SUPPLY_CLASS is orthogonal to SUPPLYTYPE.

Example:

```
CELL my_core {
   PIN vdd_dig { SUPPLYTYPE = power; SUPPLY_CLASS = digital; }
   PIN vss_dig { SUPPLYTYPE = ground; SUPPLY_CLASS = digital; }
   PIN vdd_ana { SUPPLYTYPE = power; SUPPLY_CLASS = analog; }
   PIN vss_ana { SUPPLYTYPE = ground; SUPPLY_CLASS = analog; }
}
```

o.k.

2.7 Scalar pins inside bus

Background:

A pin may be defined as a scalar pin or as an array in order to represent a bus. In the latter case, all pin properties apply to the bus. There is also the necessity to describe pin properties which apply only to a subrange of pins within the bus. The existing capability of annotations applicable to scalar pins within the bus (see ALF 1.1, chapter 4.9.4) does not address all requirements.

Proposal:

A PIN delared as a bus shall contain the optional pin_instantiation statement, defined as follows:

```
pin_instantiation ::=
   pin_identifier [ index ] {
         pin_items
   }
```

where `index`, `pin_items` are defined in ALF 1.1, chapter 3.4.4 and 3.4.10, respectively.

A pin_instantiation statement refering to a bus may also contain a pin_instantiation statement refering to a part of the bus.

Annotations and arithmetic models within the scope of the PIN or a higher-level pin_instantiation (see ALF 1.1, chapter 3.6.3) shall be inherited by a lower-level pin_instantiation, as long as their values are applicable for both the bus and each scalar pin within the bus. The values of VIEW, INITIAL_VALUE, CAPACITANCE shall not be inherited, since a particular value cannot apply at the same time to the bus and to its scalar pins.

Example:

```
PIN [1:4] my_address {
   DIRECTION = input;
   SIGNALTYPE = address;
   VIEW = functional;
   CAPACITANCE = 0.07;
   my_address [1:2] {
         CAPACITANCE = 0.03;
```

```
        my_address[1] { VIEW = physical; CAPACITANCE = 0.01; }
        my_address[2] { VIEW = physical; CAPACITANCE = 0.01; }
    }
    my_address [3:4] {
        CAPACITANCE = 0.04;
        my_address[3] { VIEW = physical; CAPACITANCE = 0.02; }
        my_address[4] { VIEW = physical; CAPACITANCE = 0.02; }
    }
}
```

2.8 BITMAP statement

The mapping from logical to physical address and data space of a memory shall be defined within the BITMAP statement. The BITMAP statement shall be within the context of a CELL.

The following example serves as reference for subsequent examples:

```
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 write_enb { DIRECTION=input; SIGNALTYPE=write_enable;
      POLARITY=high; ACTION=asynchronous;
   }
   PIN[3:0] bits[0:15] { DIRECTION=none; VIEW=none; SCOPE=behavior; }
   FUNCTION {
      BEHAVIOR {
         Dout = bits[addr];
         @ (write_enb) { bits[addr] = Din; }
      }
   }
   BITMAP { /* see following examples */ }
}
```

A one-dimensional array with `SIGNALTYPE=address` (here: `PIN[3:0] addr`) shall be recognized as address pin to be mapped.

A two-dimensional array (here: `PIN[3:0] bits[0:15]`) shall be recognized as a memory array to be mapped. The first index specifies the number of bits per word. The range must match with the range of data pins (here `PIN[3:0] Din` and `PIN[3:0] Dout`). The second index specifies the number of words.

The bitmap statement shall have the following form:

```
bitmap ::=
   BITMAP {
      { address_identifier index = row_column_boolean_expression ; }
      DATA { { memory_identifier 1st_index } }
   }
```

The *address*_identifier is the name of the address pin.
The *row_column*_boolean_expression involves exclusively the following logic_variables:

ROW index
COLUMN index

These variables shall contain the index of the physical row and column, respectively. The number of rows and columns is related to the number of address locations and data bits per word as follows:

number of rows * number of columns = number of address locations * number of data bits

where

number of rows <= 2**(1 + max (row index) - min (row index))
number of columns <= 2**(1 + max (column index) - min (column index))
number of address locations = number of data words
number of data words <= 2**(1 + max (address index) - min (address index))
number of data bits = 1 + max (data index) - min (data index)

The index of the physical row and column, respectively, shall represent the binary code of the actual physical row and column, respectively, in ascending order, starting with 0.

Example:

4 rows are represented in the variable ROW[1:0]
row number 3 has COLUMN[1:0] == 'b11

16 columns are represented in the variable COLUMN[3:0]
column number 5 has COLUMN[3:0] == 'b0101

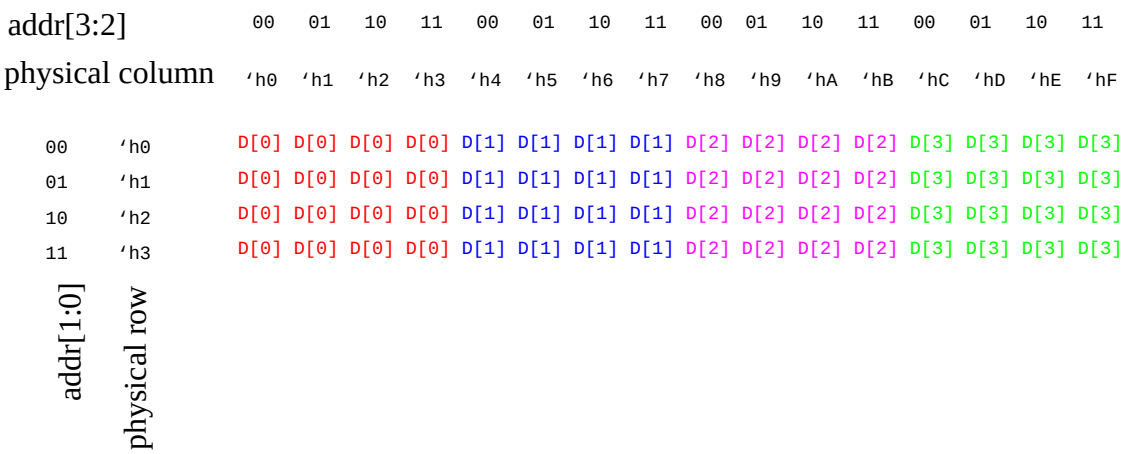
The *memory*_identifier is the name of the memory array.
The *1st*_index of *memory*_identifier indicates the number of data bits.

Example 1

addr[3:2]		00	00	00	00	01	01	01	01	10	10	10	10	11	11	11	11
physical column		'h0	'h1	'h2	'h3	'h4	'h5	'h6	'h7	'h8	'h9	'hA	'hB	'hC	'hD	'hE	'hF
00	'h0	D[0]	D[1]	D[2]	D[3]	D[0]	D[1]	D[2]	D[3]	D[0]	D[1]	D[2]	D[3]	D[0]	D[1]	D[2]	D[3]
01	'h1	D[0]	D[1]	D[2]	D[3]	D[0]	D[1]	D[2]	D[3]	D[0]	D[1]	D[2]	D[3]	D[0]	D[1]	D[2]	D[3]
10	'h2	D[0]	D[1]	D[2]	D[3]	D[0]	D[1]	D[2]	D[3]	D[0]	D[1]	D[2]	D[3]	D[0]	D[1]	D[2]	D[3]
11	'h3	D[0]	D[1]	D[2]	D[3]	D[0]	D[1]	D[2]	D[3]	D[0]	D[1]	D[2]	D[3]	D[0]	D[1]	D[2]	D[3]
addr[1:0]	physical row																

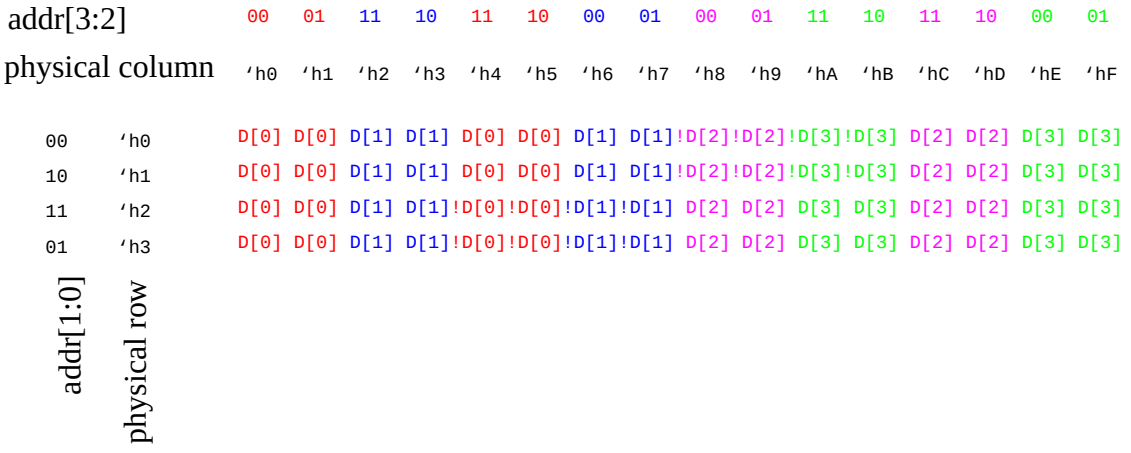
```
BITMAP {
   addr[1:0] = ROW[1:0];
   addr[3:2] = COLUMN[3:2];
   DATA { bits[0:3] bits[0:3] bits[0:3] bits[0:3] }
}
```

Example 2




```
BITMAP {
   addr[1:0] = ROW[1:0];
   addr[3:2] = COLUMN[1:0];
   DATA {
      bits[0] bits[0] bits[0] bits[0]
      bits[1] bits[1] bits[1] bits[1]
      bits[2] bits[2] bits[2] bits[2]
      bits[3] bits[3] bits[3] bits[3]
   }
}
```

Example 3



```
BITMAP {
   addr[0] = ROW[1];
   addr[1] = ROW[0] ^ ROW[1];
   addr[2] = COLUMN[0] ^ COLUMN[1] ^ COLUMN[2];
   addr[3] = COLUMN[2] ^ COLUMN[3];
   DATA {
      bits[0] bits[0] bits[1] bits[1]
      bits[0]^ ROW[1] bits[0]^ ROW[1] bits[1]^ ROW[1] bits[1]^ ROW[1]
      bits[2]^~ROW[1] bits[2]^~ROW[1] bits[3]^~ROW[1] bits[3]^~ROW[1]
      bits[2] bits[2] bits[3] bits[3]
   }
   }
```

2.9 ILLEGAL statement inside VECTOR

Background:

For complex cells, especially multi-port memories, it is useful to define the behavior as a consequence of illegal operations, for example when several ports try to access the same address.

Proposal:

A VECTOR statement shall contain the optional ILLEGAL statement, defined as follows:

```
illegal ::=
   ILLEGAL [ identifier ] { illegal_items }

illegal_items ::= illegal_item { illegal_item }

illegal_item ::=
   all_purpose_item
|  violation
```

where `all_purpose_item`, `violation` are defined in ALF 1.1, chapter 3.4.6 and 3.6.1.2, respectively.

The vector_expression within the VECTOR statement describes a state or a sequence of events which define an illegal operation. The VIOLATION statement describes the consequence of such an illegal operation.

Example:

```
VECTOR ( (addr_A == addr_B) && write_enable_A && write_enable_B ) {
   ILLEGAL write_A_write_B {
         VIOLATION {
               MESSAGE = “write conflict between port A and B“;
               MESSAGE_TYPE = error;
               BEHAVIOR { data[addrA] = ‘bxxxxxxxx; }
         }
   }
}
```

Note: An illegal operation can be legalized by using MESSAGE_TYPE=INFORMATION or MESSAGE_TYPE=WARNING.

This statement can also be used to define the behavior when an address is out of range. Note that sometimes the address space is not continguous, i.e., it may contain holes in the middle. In that case, a MIN or MAX value for leagal addresses would not be sufficient. On the other hand, a boolean_expression can always exactly describe the legal and illegal address space.

Example:

```
VECTOR ( (addr > ‘h3) && write_enb ) {
   ILLEGAL {
         VIOLATION {
               MESSAGE = “write address out of range“;
               MESSAGE_TYPE = error;
               BEHAVIOR { data[addr] = ‘bxxxxxxxx; }
         }
   }
}
```

2.10 KEYWORD statement

Background:

The ALF language allows to introduce customized context-sensitive keywords for certain purposes. While the semantics of these custom keywords can only be known by the user of such keywords, every ALF parser should have the capability to check the correct syntax of objects involving custom keywords. This cannot be achieved without a declaration of such custom keywords.

Proposal:

The category of Generic Objects (see ALF 1.1, chapter 3.4.6, 3.4.7) shall be augmented by the KEYWORD statement. The KEYWORD statement shall be defined as follows:

```
generic_object ::=
// set of current definitions in ALF 1.1, chapter 3.4.6
|  keyword_statement

keyword_statement ::=
   KEYWORD context_sensitive_keyword = syntax_item_identifier ;
```

The following syntax items use context sensitive keywords in ALF 1.1:

```
syntax_item_identifier ::=
   annotation
|  annotation_container
|  arithmetic_model
|  arithmetic_submodel
```

```
|  arithmetic_model_container
|  vector_assignment
```

Example:

```
KEYWORD my_arithmetic_model = arithmetic_model;
KEYWORD my_annotation_for_capacitance = annotation;
KEYWORD my_annotation_for_resistance = annotation;
my_arithmetic_model {
   HEADER {
         CAPACITANCE { my_annotation_for_capacitance = foo; }
         RESITANCE { my_annotation_for_resistance = bar; }
   }
   EQUATION { 10*CAPACITANCE + 0.5*RESISTANCE }
}
```

It shall be illegal to redefine an already predefined ALF keyword.

Example:

```
KEYWORD vector = arithmetic_model; // illegal
```

2.11 Misc. new statements

Some of the proposed statements in this chapter are closely related to layout. However, the statements must be also recognized by certain non-layout tools. Therefore they are in this chapter rather than in chapter 3.

New value for RESTRICT_CLASS statement

The set of predefined values for RESTRICT_CLASS (see ALF 1.1, chapter 3.6.5.9) shall be augmented by “layout”. Cells with this RESTRICT_CLASS value may be used by layout tools, i.e. place & route tools.

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 must know about “logically equivalent cells” for swapping. A layout tool must 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 my_low_power_nand2 {
   SWAP_CLASS { all_nand2 all_low_power_nand2 }
}
CELL my_high_power_nand2 {
   SWAP_CLASS { all_nand2 all_high_power_nand2 }
}
```

```
CELL another_low_power_nand2 {
   SWAP_CLASS { all_low_power_nand2 }
}
CELL another_high_power_nand2 {
   SWAP_CLASS { all_high_power_nand2 }
}
```

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

The synthesis tool may swap `my_low_power_nand2` with `my_high_power_nand2`.
The layout tool may swap `my_low_power_nand2` with `another_low_power_nand2` and `my_high_power_nand2` with `another_high_power_nand2`.

PLACEMENT_TYPE statement

A CELL may contain the following PLACEMENT_TYPE statement:

```
placement_type_assignment ::=
   PLACEMENT_TYPE = placement_type_identifier ;

placement_type_identifier ::=
   pad
|  core
|  ring
|  block
|  connector
```

- 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

ROUTING_TYPE statement

A PIN may contain the following ROUTING_TYPE statement:

```
routing_type_assignment ::=
   ROUTING_TYPE = routing_type_identifier ;

routing_type_identifier ::=
   regular
|  abutment
|  ring
|  feedthrough
```

- 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

CALCULATION statement for arithmetic model

An arithmetic model in the context of a VECTOR may have the CALCULATION annotation defined as follows:

```
calculation_annotation ::=
   CALCULATION = calculation_identifier ;

calculation_identifier ::=
   absolute
|  incremental
```

It shall specify whether the data of the model are to be used by themselves or in combination with other data. 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 must be the same. Units are allowed to be different.
  Example: slewrate measurements at the same pin, same switching direction, same threshold values.
- The model definitions for common arguments are compatible, i.e. same range of values for table-based models, measurement specifications must be the same. Units are allowed to be different.
  Example: same values for derate_case, same threshold definitions for input slewrate.
- The vector definitions are compatible, i.e. the `vector_or_boolean_expression` of the VECTOR containing **incremental** data must match 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; }
               DELAY delay_A_B { FROM { PIN = A; } TO { PIN = B; } }
         }
         EQUATION {- 0.1 + (0.05+0.002*slew_A*slew_B)*delay_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 the input B switches in a time window between A and Z.

INTERPOLATION statement for arithmetic model

An argument of a table-based arithmetic model, i.e., a model in the HEADER containing a TABLE statement, may have the INTERPOLATION annotation defined as follows:

```
interpolation_annotation ::=
   INTERPOLATION = interpolation_identifier ;

interpolation_identifier ::=
   fit
|  floor
|  ceiling
```

It shall specify, the interpolation scheme for 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, polynominal regression, may be used to fit the data points into the curbe.
- **`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.

Default is **`fit`**. Note that for multi-dimensional tables, different interpolation schemes may 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
```

DIRECTION for power and ground pins

The DIRECTION annotation for PINs with **`PINTYPE=supply`** shall have the following semantic meaning:

`DIRECTION=input` applies for a pin which is a power supply sink (default).

`DIRECTION=output` applies for a pin which is a power supply source.

`DIRECTION=both` applies for a pin that can be both source and sink.

`DIRECTION=none` applies for a pin without connection to source or sink.

Examples:

The power and ground pins of regular cells will have DIRECTION=input.

A level converter cell will have a power supply pin with DIRECTION=input and another power supply pin with DIRECTION=output.

A level converter may 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 feedthrough cell will have DIRECTION=none.

Usage of DEFAULT as qualifier for arithmetic submodel

Arithmetic submodels may have the qualifiers MIN, TYP, MAX (see ALF 1.1, chapter 3.6.9.1). For cases where the application tool cannot decide which qualifier applies, the DEFAULT annotation may point to the applicable qualifier.

Example:

```
PIN my_pin {
   CAPACITANCE {
         MIN { HEADER { ... } TABLE { ... } }
         TYP { HEADER { ... } TABLE { ... } }
         MAX { HEADER { ... } TABLE { ... } }
         DEFAULT = TYP;
   }
}
```

Arithmetic submodels may have the qualifiers RISE, FALL, HIGH, LOW (see ALF 1.1, chapter 3.6.9.2). For cases where the application tool cannot decide which qualifier applies, a supplementary arithmetic submodel with the qualifier DEFAULT may be used.

Example:

```
PIN my_pin {
   CAPACITANCE {
         RISE { HEADER { ... } TABLE { ... } }
         FALL { HEADER { ... } TABLE { ... } }
         DEFAULT { HEADER { ... } TABLE { ... } }
   }
}
```

Rename ORIENTATION into SIDE

ALF 1.1 chapter 3.6.3.11 contains the ORIENTATION annotation for a pin, specifying at which side of the rectangular cell the pin is located:

ORIENTATION = right | left | top | bottom.

We propose to rename ORIENTATION to SIDE to be consistent with PDEF and to avoid confusion with the usage of the word “orientation” in layout.

ROW and COLUMN annotation for PIN

We propose the following annotation for a pin in order to indicate the location of the pin within a placement row or column:

```
row_assignment ::=
   ROW = unsigned ;

column_assignment ::=
   COLUMN = unsigned ;
```

where *row_assignment* applies for pins with **SIDE = right | left** and *column_assignment* applies for pins with **SIDE = top | bottom**.

For bus pins, *row*_assignment and *column*_assignment shall have the form of multi_value_assignments.

```
row_multi_value_assignment ::=
   ROW { unsigned { unsigned } }

column_multi_value_assignment ::=
   COLUMN { unsigned { unsigned } }
```

Driver CELL and PIN specification for a macro

The keywords CELL and PIN used as reference to existing objects (see ALF 3.6.2) can be used to define driver cell and pin in a macro.

Example:

```
// this is a standard ASIC cell
CELL my_inv {
   PIN in { DIRECTION = input; }
   PIN out { DIRECTION = output; }
}

// this is a macro, synthesized from standard ASIC cells
CELL my_macro {
   PIN my_output {
         DIRECTION = output;
         CELL = my_inv { PIN = out; }
   }
   /* fill in other pins and stuff */
}
```

3.0 Physical modeling in ALF 2.0

Overview of new statements

The following table summarizes the proposed statements in ALF for physical modeling.

TABLE 1. New 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	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 calculatable 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
ARRAY	LIBRARY, SUBLIBRARY	Description of a regular grid for placement, global and detailed routing
geometric model	BLOCKAGE, PORT, 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

Arithmetic models in the context of layout

The following tables summarize the semantic meanings of arithmetic model keywords in the context of layout. Unless otherwise noted, the keywords are already defined in ALF 1.1, chapter 3.6.7.1. The meaning of the keywords in this context is also given.

TABLE 2. 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 3. 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 4. 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 5. 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 6. 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 routable object, measured in the x-y plane

TABLE 7. Semantic meaning of PERIMETER (new keyword)

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 routable object, measured in the x-y plane

TABLE 8. Semantic meaning of DISTANCE

context	meaning
RULE	distance between objects for which the rule applies

TABLE 9. Semantic meaning of THICKNESS (new keyword)

context	meaning
LAYER, ANTENNA	distance between top and bottom of a physical object, orthogonal to the x-y plane

TABLE 10. Semantic meaning of OVERHANG (new keyword)

context	meaning
RULE	distance *from* the outer border of an object *to* the outer border of another object inside the first one

TABLE 11. Semantic meaning of EXTENSION (new keyword)

context	meaning
LAYER, VIA, RULE, geometric model	distance between the border of the original object and the border of the same object after enlargement

3.1 PORT Statement

Status: Proposal considered acceptable Nov. 4

Jan. 2000: This chapter defines PORT statement in the context of PIN.

Background:

The reason for this proposal is the necessity of describing electrical models of cells where a logical PIN maps to one ore more physical PORTs. The electrical models for pin CAPACITANCE, RESISTANCE etc. may not suffice for such cells. Also, timing arcs may be defined from or to a PORT rather than from or to a PIN. Therefore a PORT shall be considered as connection point for electrical components and timing arcs.

Proposal:

The PORT statement shall be defined as follows:

```
port ::=
   PORT port_identifier ;
|  PORT [ port_identifier ] {
              [ all_purpose_items ]
              [ geometric_models ]
              [ geometric_transformations ]
   }
```

A numerical digit may be used as first character in *port*_identifier. In this case the number must be preceeded by the escape character (see ALF 1.1, chapter 3.2.12) in the declaration of the PORT.

See this document chapter 3.2 for the definition of geometric_models.

See this document chapter 3.3 for the definition of geometric_transformations.

Specific all_purpose_items for PORT are *port_view*_annotation, *layer*_annotation.

VIEW annotation inside a PORT

A subset of values for the VIEW annotation inside a PIN (see ALF 1.1, chapter 3.6.3.1) 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.

LAYER annotation inside a PORT

The *layer_annotation* may appear inside a PORT (see this document, chapter 3.2).

Declaration of a PORT inside PIN

The syntax for pin_item (see ALF1.1, chapter 3.4.10) shall be augmented as follows:

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

A pin may have either no PORT statement or an arbitrary number of PORT statements with *port_identifier* or exactly one PORT statement without *port_identifier*.

FEEDTHROUGH annotation inside PIN

The *pin_feedthrough_annotation* shall be used inside a PIN to indicate whether PORTs inside a PIN are electrically connected.

```
pin_feedthrough_annotation ::=
   FEEDTHROUGH [identifier] { port_identifiers }
```

Each *port_identifier* must be a declared PORT with **VIEW=physical**.

Example:

```
PIN A {
   PORT P1 { VIEW=physical; }
   PORT P2 { VIEW=physical; }
   PORT P3 { VIEW=physical; }
   FEEDTHROUGH { P1 P2 }
}
```

The router has the following choices:

connect to P1 only

connect to P2 only

connect to P3 only

connect to P1 from one point and to P2 from another point, since P1 and P2 are internally shorted.

CONNECTIVITY rules for PORTs and PINs

Connect rules may apply for ports or pins. They are constructed in the same way as in the context of RULE statements (see chapter 3.7).

Example:

```
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 must make an external connection between any port of pin B and any port of pin A.

ROUTING_TYPE inside PORT

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

Reference of a declared PORT in a PIN annotation

In the context of timing modeling, a PORT may have the semantic meaning of a PIN. For examples, PORTs may be used as FROM and/or TO points of delay measurements. A reference by a `hierarchical_identifier` (see chapter 2.3 of this document) may be used:

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; }
         }
   }
}
```

3.2 Geometric Model Statement

Status: reviewed Dec. 7, modified February 2000

Proposal:

The geometric model statement shall be defined as follows:

```
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. 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. last point is connected with first point. The object occupies the entire enclosed space.

See this document chapter 3.3 for the definition of the `repeat` statement.

The *point_to_point*_annotation applies for **POLYLINE**, **RING**, **POLYGON**.
It specifies how the connections between points is made. Default is straight. The value straight defines a straight connection. The value rectilinear specifies a connection by moving in x-direction first and then moving in y-direction. 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
```

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.

Use of TEMPLATE to construct special geometric models

The TEMPLATE contruct (see ALF 1.1, chapter 3.1.2.6) can be used to predefine some commonly used objects.

```
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> }
   }
}

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> }
   }
}

// 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 }

// 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 }
```

Context of geometric model statements

A `geometric_model` describes the form of a physical object, it does not describe a physical object itself. The `geometric_model` must be in the context of a physical object.

The following keywords for physical objects shall be introduced:

- PORT (see chapter 3.1)
- BLOCKAGE (see chapter 3.5)
- PATTERN (see chapter 3.6)

Physical objects may contain `geometric_model` statements, geometric transformation statements (see chapter 3.3) as well as `all_purpose_items` (see ALF 1.1, chapter 3.4.6).

New keywords for `all_purpose_items` are defined as follows:

The *`layer`*`_annotation` defines the layer where the object resides. The layer must have been declared before.

```
layer_annotation ::=
   LAYER = layer_identifier ;
```

The *`extension`*`_annotation` specifies the value by which the drawn object is extended at all sides.

```
extension_annotation ::=
   EXTENSION = non_negative_number ;
```

Default value of *`extension`*`_annotation` is 0.

Example:

```
PATTERN {
   LAYER = metal1;
   EXTENSION = 1;
   DOT { COORDINATES { 5 10 } }
}
```

This object is effectively a square with lower left corner (x=4,y=9) and upper right corner (x=6,y=11).

3.3 Statements for geometric transformation

Status: statements individually reviewed Dec. 7. SHIFT, ROTATE, FLIP, REPEAT are now regrouped in one chapter.

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 may only support integer numbers.

```
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.

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 in mathematical positive sense. Eventually, a layout tool may only support angles which are multiple of 90 degrees. Default is 0.

```
rotate_annotation ::=
   ROTATE = number ;
```

The object shall rotate around its origin.

FLIP statement

The *flip*_annotation specifies a transformation of the specified coordinates by flipping the object around an axis specified by a number between 0 and 90. The number indicates the flipping direction. 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.

REPEAT statement

The REPEAT statement shall be defined as follows:

```
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 chapter 3.8). The REPEAT statement may 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 3 times along the horizontal axis in a distance of 7 units.

```
REPEAT = 3 {
   SHIFT { HORIZONTAL = 7; }
}
```

The following example replicates an object 5 times along a 45-degree axis.

```
REPEAT = 5 {
   SHIFT { HORIZONTAL = 4; VERTICAL = 4; }
}
```

The following example replicates an object 2 times along the horizontal axis and 4 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; }
   }
}
```

Summary of geometric transformations

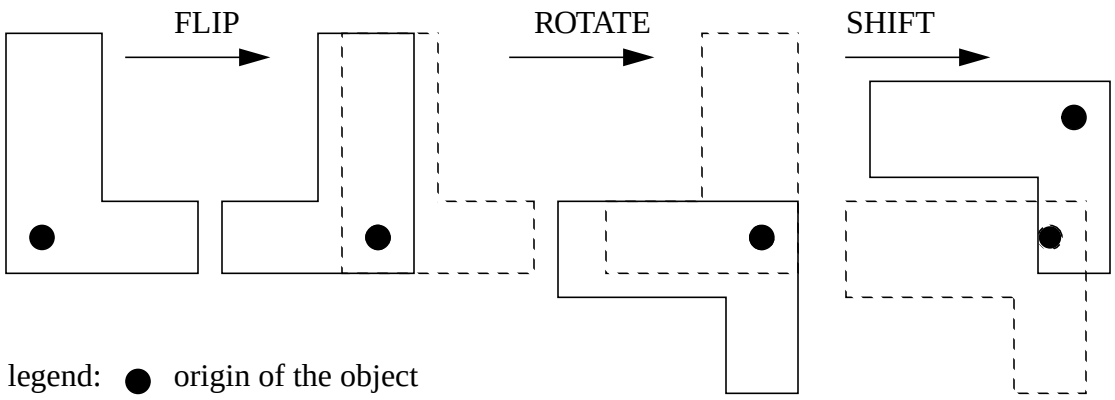
```
geometric_transformations ::=
   geometric_transformation { geometric_transformation }

geometric_transformation ::=
   shift_annotation_container
|  rotate_annotation
|  flip_annotation
|  repeat
```

Rules and restrictions:

- A physical object may 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 will be independent of the order in which each individual transformation is applied.

FIGURE 2. Illustration of FLIP, ROTATE, SHIFT



3.4 LAYER Statement

Status: proposal modified February 2000

Proposal:

The LAYER statement shall be defined as follows:

```
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 already defined in ALF1.1.

Specific items applicable for LAYER are listed in the following table.

TABLE 12. Items for LAYER description

item	applies for layer	usable ALF statement	comment
purpose	all	PURPOSE = <identifier> ;	see this doc., chapter 3.4
property	routing, cut, master	PROPERTY { ... }	see ALF 1.1, chapter 3.1.2.7

TABLE 12. Items for LAYER description

item	applies for layer	usable ALF statement	comment
current density limit	routing, cut	LIMIT { CURRENT { ... MAX { ... } } }	see ALF 1.1, chapters 3.6.7.1, 3.6.8.2, 3.6.9.1, 3.6.10.5 and example
resistance	routing, cut	RESISTANCE { ... }	see ALF 1.1, chapter 3.6.7.1 and example
capacitance	routing	CAPACITANCE {... }	see ALF 1.1, chapter 3.6.7.1 and example
default width or minimum width	routing	WIDTH { DEFAULT = <number>; }	see ALF 1.1, chapters 3.6.7.1, 3.6.10.1 and example
default wire extension	routing	EXTENSION { DEFAULT = <number>; }	see this doc., chapter 3.0 and example
height	routing, cut, master	HEIGHT = <number>;	see this doc., chapter 3.0
thickness	routing, cut, master	THICKNESS = <number>;	see this doc., chapter 3.0
prefered routing direction	routing	PREFERENCE	see this doc., chapter 3.4

Note: Rules involving relationships between objects within one or several layers will be described in the RULE statement (chapter 3.6).

The purpose of the LAYER shall be defined by the following statement:

```
layer_purpose_assignment ::=
   PURPOSE = layer_purpose_identifier ;

layer_purpose_identifier ::=
   routing
|  cut
|  master
```

LAYER statements must be in sequential order defined by the manufacturing process, starting with the substrate in the following sequence: One or multiple master layers, followed by alternating cut and routing layers.

ThePREFERENCE statement for LAYER shall have the following form:

```
routing_preference_annotation_container ::=
   PREFERENCE { routing_preference_annotations }

routing_preference_annotations ::=
   routing_preference_annotation { routing_preference_annotation }

routing_preference_annotation ::=
   routing_preference_identifier = non_negative_number ;

routing_preference_identifier ::=
   HORIZONTAL
|  VERTICAL
```

The purpose is to give a weighting factor for the preferered routing direction. The weighting factors on each routing layer shall add up to 1.

Example:

This example contains default width (syntax is `all_purpose_item`), resistance, capacitance, current limits (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 }
        }
   }
}
```

3.5 BLOCKAGE Statement

Status: proposal modified February 2000

Proposal:

The BLOCKAGE statement shall be defined as follows:

```
blockage ::=
BLOCKAGE [ identifier ] {
              [ all_purpose_items ]
              [ geometric_models ]
              [ geometric_transformations ]
   }
```

See chapter 3.2 for applicable `all_purpose_items`.

Example:

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

```
        RECTANGLE { -1 5 3 8 }
   }
}
```

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

3.6 PATTERN and VIA Statement

Status: proposal modified February 2000

Proposal for PATTERN statement

The PATTERN statement shall be defined as follows:

```
pattern ::=
PATTERN [ identifier ] {
   [ all purpose_items ]
   [ geometric_models ]
   [ geometric_transformations ]
}
```

A special `all_purpose_item` for PATTERN is the following:

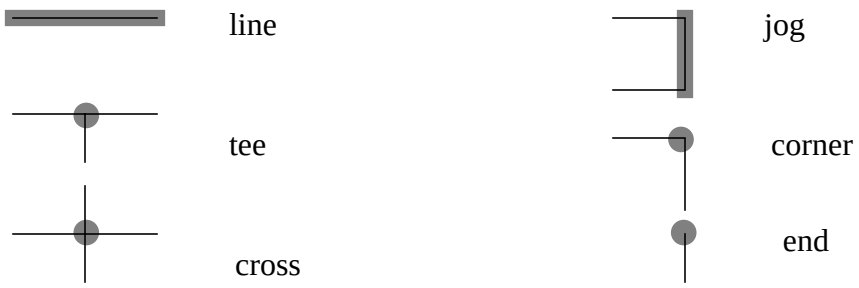
```
shape_assignment ::=
   SHAPE = shape_identifier ;

shape_identifier ::=
   line
|  tee
|  cross
|  jog
|  corner
|  end
```

SHAPE applies only for PATTERN in a routing layer. Default is **`line`**.

Line and jog represent routing segments, which can have an individual LENGTH and WIDTH. LENGTH *between* routing segments is defined as common run length. DISTANCE *between* routing segments is measured orthogonal to routing direction.

Tee, cross, corner represent intersections between routing segments. End represents the end of a routing segment. Therefore they have points rather than lines as reference. The points can have an EXTENSION. DISTANCE between points can be measured either straight or HORIZONTAL and VERTICAL.

See the following illustration:



See chapter 3.2 for other applicable `all_purpose_items`.

Proposal for VIA statement

The VIA statement shall be defined as follows:

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

via_items ::= via_item { via_item }

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

The VIA statement must contain at least 3 patterns, refering to the cut layer and two adjacent routing layers. Stacked vias may contain more than 3 patterns.

Specific `all_purpose_items` and `arithmetic_models` for VIA are listed in the following table.

TABLE 13. Items for VIA description

item	usable ALF statement	comment
property	PROPERTY	see ALF 1.1, chapter 3.1.2.7
resistance	RESISTANCE	see ALF 1.1, chapter 3.6.7.1
GDS2 reference	ARTWORK	see this document, chapter 3.10 and example
usage	USAGE	see this document, chapter 3.6 and example

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

```
usage_annotation ::=
   USAGE = usage_identifier ;
```

```
usage_identifier ::=
   default
|  non_default
|  top_of_stack
```

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 ALF 1.1, chapter 3.1.2.6) 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 ALF 1.1, chapter 3.11) 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, mathematical relationships between the placeholders are defined which allow to generate a via rule for any set of values.

3.7 RULE Statement

Status: proposal modified February 2000

Proposal:

The RULE statement shall be defined as follows:

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

rule_items ::= rule_item { rule_item }

rule_item ::=
   pattern
|  all_purpose_item
|  arithmetic_model
```

Specific `all_purpose_items` for RULE are listed in the following table.

TABLE 14. Items for RULE description

item	usable ALF statement	comment
rule is for same net or different nets	CONNECTIVITY	see ALF 1.1, chapter 3.6.10.3 and this chapter
spacing rule	LIMIT { DISTANCE ... }	see this document, chapter 3.0 and example
overhang rule	LIMIT { OVERHANG ... }	see this document, chapter 3.0 and example

The CONNECTIVITY statement shall contain the CONNECT_RULE statement in conjunction with the BETWEEN statement.

```
between_multi_value_assignment ::=
   BETWEEN { pattern_identifiers }
```

Rules for spacing and overlap, respectively, shall be expressed using the LIMIT construct with DISTANCE and OVERHANG, respectively, as keyword for arithmetic models (see ALF spec. 1.1, chapter 3.6.8.2 and 3.6.9.1). The keywords HORIZONTAL and VERTICAL shall be introduced as qualifiers for arithmetic submodels (see ALF spec. 1.1, chapter 3.6.9) in order to distinguish rules for different routing directions. If these qualifiers are not used, the rule shall apply in any routing direction.

Example:

```
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 */ }
            }
        }
    }
```

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 rules 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 */ }
                }
          }
   }
```

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;
         }
   }
}
```

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 */
    }
}
```

3.8 SITE Statement

Status: proposal reviewed December 1999, modified March 2000

Proposal:

The SITE statement shall be defined as follows:

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

Specific `all_purpose_items` for SITE are *`symmetry`*`_annotation_container`, *`width`*`_annotation`, *`height`*`_annotation`.

The *`symetry`*`_annotation_container` is optional. If not specified, the SITE is considered asymetric. The *`symetry`*`_annotation_container` contains *`multi_flip`*`_annotation` or *`multi_rotate`*`_annotation` or both.

```
symmetry_annotation_container ::=
    SYMMETRY {
        [ multi_flip_annotation ]
        [ multi_rotate_annotation ]
    }
```

The *`multi_flip`*`_annotation` specifies whether the object preserves its symmetry when flipped around an axis specified by one or multiple numbers modulo 180.

```
multi_flip_annotation ::=
    FLIP { number { number } }
```

The *`multi_rotate`*`_annotation` specifies whether the object preserves its symetry when rotate around an axis specified by one or several numbers modulo 360.

```
multi_rotate_annotation ::=
    ROTATE { number { number } }
```

A SITE may have more than one SYMMETRY statement, each of which describing a legal combination of FLIP and ROTATE operations.

The *`width`*`_annotation` and *`height`*`_annotation` (see this document, chapter 3.0) are mandatory.

Example:

```
SITE my_site {
   WIDTH = 100 ;
   HEIGHT = 100 ;
   SYMMETRY { FLIP { 0 90 } ROTATE { 90 } }
}
```

The following site orientations are legal in this example:

- normal orientation (no flip, no rotate), also called “north”
- horizontal flip, also called “flip north”
- vertical flip, also called “flip south”
- 90 degrees rotation, also called “west”
- horizontal flip combined with 90 degrees rotation, resulting in so-called “flip east”
- vertical flip combined with 90 degrees rotation, resulting in so-called “flip west”

Reference of a SITE by a CELL

A CELL may point to one or more legal placement SITEs and also contain one or more SYMMETRY statements. The intersection between the SYMMETRY of the CELL and the SYMMETRY of the refered SITE shall define the set of valid orientations of the CELL.

Example:

```
CELL my_cell {
   SITE { my_site /* fill in other sites, if applicable */ }
   SYMMETRY { ROTATE { 90 180 270 } }
}
```

The following cell orientations are legal in this example:

- normal orientation (no flip, no rotate), also called “north”
- rotate 90 degrees, also called “west”
- rotate 180 degrees, also called “south”
- rotate 270 degrees, also called “east”

Given the legal orientations of the site in the previous example, `my_cell` can be placed at `my_site` with orientation “north” or “west”.

3.9 ANTENNA Statement

Status: proposal modified March 2000

Proposal:

The ANTENNA statement shall be defined as follows:

```
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 ALF1.1.

Specific items applicable for ANTENNA are in the following table.

TABLE 15. Items for ANTENNA description

item	usable ALF statement	scope	comment
maximum allowed antenna size	LIMIT { SIZE { MAX { ... } } }	LIBRARY, SUBLIBRARY CELL, PIN	see ALF 1.1, chapters 3.6.7.1, 3.6.8.2, 3.6.9.1, 3.6.10.5 and example
calculation method for antenna size	SIZE { HEADER { ... } TABLE { ...} or SIZE [id] { HEADER { ... } EQUATION { ...}	LIBRARY, SUBLIBRARY	see ALF 1.1, chapter 3.6.7.1 and example
argument values for antenna size calculation	*argument* = *value* ; or *argument* = *value* { ... }	CELL, PIN	see ALF 1.1, chapter 3.4.1 and example

The use of the keyword SIZE (see ALF 1.1, chapter 3.6.7.1) 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 necessary a measure of area. It can be a measure of area ratio as well. However, the arguments of the calculation function for antenna SIZE must be measureable data, such as AREA, PERIMETER, LENGTH, THICKNESS, WIDTH, 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 may 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 polysilicon 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.

Layer-specific antenna rules

Antenna rules may be checked individually for each layer. In this case, the SIZE model contains only 2 or 3 arguments: AREA of the layer or perimeter (calculated from LENGTH and WIDTH) of the layer causing the antenna effect, area of polysilicon, eventually 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 }
   }
}
```

All-layer antenna rules

Antenna rules may 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 }
   }
}
```

Accumulative antenna rules

Antenna rules may 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:

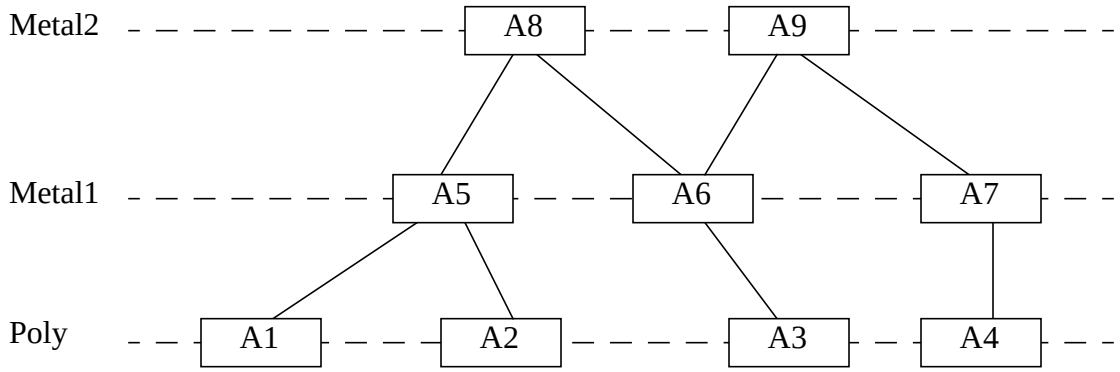
```
ANTENNA accumulate_m2_m1 {
   LIMIT { SIZE { MAX = 3000; } }
   SIZE {
        HEADER {
              SIZE ratio1 {
                    CONNECTIVITY {
                          CONNECT_RULE = must_short;
                          BETWEEN { metal1 poly }
                    }
                    CONNECTIVITY {
                          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 }
    }
}
```

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

Illustration

Consider the following structure:



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
```

A PIN of a CELL must contain information supplying values for the arguments of the antenna rule calculation function, e.g. AREA. These arguments shall refer to one or more ANTENNA rules using a `multi_value_assignment`.

```
antenna_multi_value_assignment ::=
   ANTENNA { antenna_identifier { antenna_identifier } }
```

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.

3.10 ARTWORK Statement

Status: proposal reviewed December 1999

Proposal:

The ARTWORK statement shall be defined as follows:

```
artwork ::=
   ARTWORK = artwork_identifier {
         [ shift_annotation_container ]
         [ rotate_annotation ]
         { 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 *shift*`_annotation_container` statement (see this document, chapter 3.3) defines the (x,y) offset measured between the origin of the original cell and the origin of the cell in this library. Eventually, a layout tool may only support integer numbers.

The *rotate*`_annotation` statement (see this document, chapter 3.3) defines the angle of rotation in degrees measured between the orientation of the original cell and the orienta-

tion of the cell in this library in mathematical positive sense. Eventually, a layout tool may only support angles which are multiple of 90 degrees.

The imported cell may have pins with different names. The LHS of the `pin_assignments` describes the pinnames of the original cell, the RHS describes the pinnames of the cell in this library. Syntax for `pin_assignments` see ALF1.1, chapter 3.4.3.

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;
   }
}
```

3.11 ARRAY Statement

Status: proposal reviewed December 1999

Proposal modified March 2000

Proposal:

The ARRAY statement shall be defined as follows:

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

Each array shall have a PURPOSE assignment.

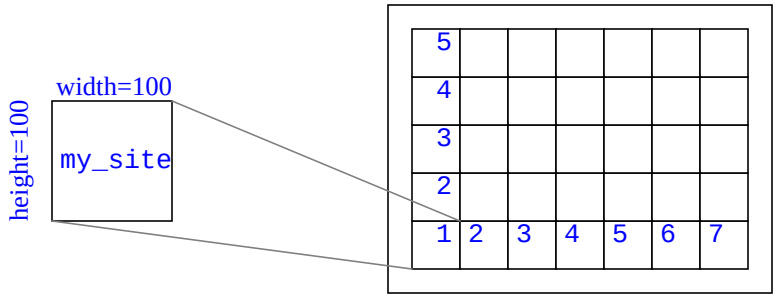
```
array_purpose_assignment ::=
   PURPOSE = array_purpose_identifier ;

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

An array with purpose **`floorplan`** or **`placement`** shall have a reference to a SITE, a *`shift`*`_annotation_container` , *`rotate`*`_annotation`, *`flip`*`_annotation` to define the location and oritentation of the SITE in the context of the array.

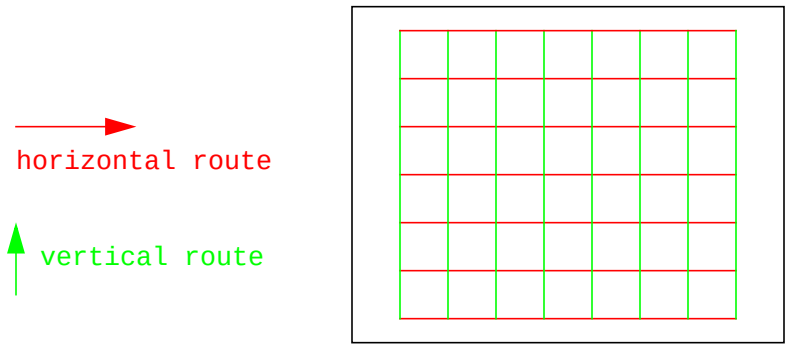
An array with purpose **`routing`** shall have a reference to one or more routing LAYERs and a *`shift`*`_annotation_container` to define the location of the starting point.

An array with purpose **`global`** shall have a *`shift`*`_annotation_container` to define the location of the starting point.

Examples:

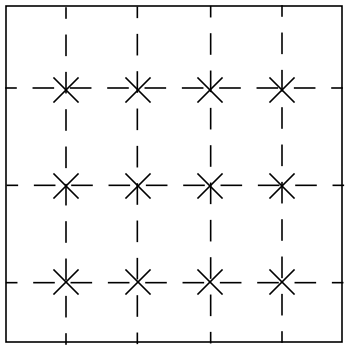



```
ARRAY grid_for_my_site {
   PURPOSE = placement;
   SITE = my_site;
   SHIFT { HORIZONTAL = 50; VERTICAL = 50; }
   REPEAT = 7 {
         SHIFT { HORIZONTAL = 100; }
         REPEAT = 5 {
               SHIFT { VERTICAL = 5; }
         }
   }
}
```




```
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; }
          }
   }
}
```



```
ARRAY grid_for_global_routing {
   PURPOSE = global;
   SHIFT { HORIZONTAL = 100; VERTICAL = 100; }
   REPEAT = 3 {
          SHIFT { VERTICAL = 150; }
          REPEAT = 4 {
                SHIFT { HORIZONTAL = 100; }
          }
   }
}
```