Section 6
Modeling for Test

6.1 Annotations and attributes for a CELL

This section defines various CELL annotations and attributes.

6.1.1 CELLTYPE annotation

CELLTYPE classifies 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.

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

which can take the values shown in Table 6-1.

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>cell is a buffer, inverting or non-inverting</td>
</tr>
<tr>
<td>combinational</td>
<td>cell is a combinational logic element</td>
</tr>
<tr>
<td>multiplexor</td>
<td>cell is a multiplexor</td>
</tr>
<tr>
<td>flipflop</td>
<td>cell is a flip-flop</td>
</tr>
<tr>
<td>latch</td>
<td>cell is a latch</td>
</tr>
<tr>
<td>memory</td>
<td>cell is a memory or a register file</td>
</tr>
<tr>
<td>block</td>
<td>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.</td>
</tr>
<tr>
<td>core</td>
<td>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.</td>
</tr>
<tr>
<td>special</td>
<td>cell is a special element, which can only be used in certain application contexts not describable by the FUNCTION statement. Examples: busholders, protection diodes, and fillcells.</td>
</tr>
</tbody>
</table>

6.1.2 ATTRIBUTE within a CELL object

An ATTRIBUTE within a CELL classifies the functionality given by CELLTYPE in more detail.
The attributes shown in Table 6-2 can be used within a CELL with CELLTYPE=memory.

**Table 6-2 : Attributes within a CELL with CELLTYPE=memory**

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

The attributes shown in Table 6-3 can be used within a CELL with CELLTYPE=block.

**Table 6-3 : Attributes within a CELL with CELLTYPE=block**

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

The attributes shown in Table 6-4 can be used within a CELL with CELLTYPE=core.

**Table 6-4 : Attributes within a CELL with CELLTYPE=core**

<table>
<thead>
<tr>
<th>Attribute item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLL</td>
<td>CELL is a phase-locked loop</td>
</tr>
<tr>
<td>DSP</td>
<td>CELL is a digital signal processor</td>
</tr>
<tr>
<td>CPU</td>
<td>CELL is a central processing unit</td>
</tr>
</tbody>
</table>

(fill in more)
The attributes shown in Table 6-5 can be used within a **CELL** with **CELLTYPE=special**.

### Table 6-5: Attributes within a **CELL** with **CELLTYPE=special**

<table>
<thead>
<tr>
<th>Attribute item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>busholder</td>
<td>CELL enables a tristate bus to hold its last value before all drivers went into high-impedance state (detail see <strong>FUNCTION</strong> statement)</td>
</tr>
<tr>
<td>clamp</td>
<td>CELL connects a net to a constant value (logic value and drive strength see <strong>FUNCTION</strong> statement)</td>
</tr>
<tr>
<td>diode</td>
<td>CELL is a diode (no <strong>FUNCTION</strong> statement)</td>
</tr>
<tr>
<td>capacitor</td>
<td>CELL is a capacitor (no <strong>FUNCTION</strong> statement)</td>
</tr>
<tr>
<td>resistor</td>
<td>CELL is a resistor (no <strong>FUNCTION</strong> statement)</td>
</tr>
<tr>
<td>inductor</td>
<td>CELL is an inductor (no <strong>FUNCTION</strong> statement)</td>
</tr>
<tr>
<td>fillcell</td>
<td>CELL is merely used to fill unused space in layout (no <strong>FUNCTION</strong> statement)</td>
</tr>
</tbody>
</table>

### 6.1.3 SWAP_CLASS annotation

**SWAP_CLASS = string ;**

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

Cell-swapping is only allowed under the following conditions:
- the **RESTRICT_CLASS** annotation (see Section 6.1.4) authorizes usage of the cell
- the cells to be swapped are compatible from an application standpoint (functional compatibility for synthesis and physical compatibility for layout)

### 6.1.4 RESTRICT_CLASS annotation

**RESTRICT_CLASS = string ;**

The value is the name of a declared **CLASS**. Multi-value annotation can be used. Cells referring to a particular class can be used in design tools identified by the value. The restricted annotations are shown in Table 6-6.

### Table 6-6: Predefined values for **RESTRICT_CLASS**

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>synthesis</td>
<td>use restricted to logic synthesis</td>
</tr>
<tr>
<td>scan</td>
<td>use restricted to scan synthesis</td>
</tr>
<tr>
<td>datapath</td>
<td>use restricted to datapath synthesis</td>
</tr>
<tr>
<td>clock</td>
<td>use restricted to clock tree synthesis</td>
</tr>
<tr>
<td>layout</td>
<td>use restricted to layout, i.e., place &amp; route</td>
</tr>
</tbody>
</table>
User-defined values are also possible. If a cell has no or only unknown values for RESTRICT_CLASS, the application tool shall not modify any instantiation of that cell in the design. However, the cell shall still be considered for analysis.

Example 1:

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

Suppose cells c1 and c2 are compatible from an application standpoint; cells c1 and c2 can be used for synthesis, where they can be swapped which each other. Cell c2 can be also used for scan insertion and for the user-defined application bar.

Example 2:

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

```plaintext
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 }
}
```

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

The synthesis tool can swap my_low_power_nand2 with my_high_power_nand2. The layout tool can swap my_low_power_nand2 with another_low_power_nand2 and my_high_power_nand2 with another_high_power_nand2.
6.1.5  SCAN_TYPE annotation

SCANTYPE = string ;
which can take the values shown in Table 6-7.

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

6.1.6  SCAN_USAGE annotation

SCAN_USAGE = string ;
which can take the values shown in Table 6-8.

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

6.2  NON_SCAN_CELL statement

non_scan_cell ::= 
  NON_SCAN_CELL { non_scan_cell_instantiations } 

non_scan_cell_instantiations ::= 
  non_scan_cell_instantiation { non_scan_cell_instantiation } 

non_scan_cell_instantiation ::= 
  cell_identifier { pin_assignments } |
  primitive_identifier { pin_assignments } 

In case of a single non-scan cell, the following syntax shall also be valid:

NON_SCAN_CELL = non_scan_cell_instantiation

Is it necessary to allow this for backward compatibility with ALF 1.1?

This statement shall define non-scan cell equivalency to the scan cell in which this annotation is contained. A cell instantiation form is used to reference the library cell that defines the non-scan functionality of the current cell. If no such cell is available or defined, or if an explicit
reference to such a cell is not desired, then a primitive instantiation form can reference a
primitive, either ALF- or user-defined, for such use. In either case, constant values can appear
on either the left-hand side or right-hand side of the pin connectivity relationships. A constant
on the left-hand side defines the value the scan cell pins (appearing on the right-hand side) shall
have in order for the primitive to perform with the same functionality as does the instantiated
reference. A statement containing multiple non-scan cells shall indicate a choice between
alternative non-scan cells.

Example:

```plaintext
CELL my_flip-flop {
    PIN q { DIRECTION=output; }  
PIN d { DIRECTION=input; }    
PIN clk { DIRECTION=input; }   
PIN clear { DIRECTION=input; polarity=low; }
    // followed by function, vectors etc.
}
CELL my_other_flip-flop {
    // declare the pins
    // followed by function, vectors etc.
}
CELL my_scan_flip-flop {
    PIN data_out { DIRECTION=output; }
PIN data_in { DIRECTION=input; }
PIN clock { DIRECTION=input; }
PIN scan_in { DIRECTION=input; }
PIN scan_sel { DIRECTION=input; }
    NON_SCAN_CELL {
        my_flip-flop {
            q = data_out;
            d = data_in;
            clk = clock;
            clear = 'b1;  // scan cell has no clear
            'b0 = scan_in;  // non-scan cell has no scan_in
            'b0 = scan_sel;  // non-scan cell has no scan_sel
        }
        my_other_flip-flop {
            // put in the pin assignments
        }
    }
    // followed by function, vectors etc.
}
```

### 6.3 STRUCTURE statement

An optional **STRUCTURE** statement shall be legal in the context of a **FUNCTION**. The purpose of
the **STRUCTURE** statement is to describe the structure of a complex cell composed of atomic
cells, for example I/O buffers, LSSD flip-flops, or clock trees.

The syntax for the **FUNCTION** statement shall be augmented as follows:
The `STRUCTURE` statement shall describe a netlist of components inside the `CELL`. The `STRUCTURE` statement shall not be a substitute for the `BEHAVIOR` statement. If a `FUNCTION` contains only a `STRUCTURE` statement and no `BEHAVIOR` statement, a behavior description for that particular cell shall be meaningless (e.g., fillcells, diodes, vias, or analog cells).

Timing and power models shall be provided for the `CELL`, if such models are meaningful. Application tools are not expected to use function, timing, or power models from the instantiated components as a substitute of a missing function, timing, or power model at the top-level. However, tools performing characterization, construction, or verification of a top-level model shall use the models of the instantiated components for this purpose.

Test synthesis applications can use the structural information in order to define a one-to-many mapping for scan cell replacement, such as where a single flip-flop is replaced by a pair of master/slave latches. A macro cell can be defined whose structure is a netlist containing the master and slave latch and this shall contain the `NON_SCAN_CELL` annotation to define which sequential cells it is replacing. No timing model is required for this macro cell, since it should be treated as a transparent hierarchy level in the design netlist after test synthesis.

Notes:

1. Every `instance_identifier` within a `STRUCTURE` statement shall be different from each other.

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

3. The `cell_identifier` within a `STRUCTURE` statement can refer to actual cells as well as to primitives. The usage of primitives is recommended in fault modeling for DFT.

4. `BEHAVIOR` statements also provide the possibility of instantiating primitives. However, those instantiations are for modeling purposes only; they do not necessarily match a physical structure. The `STRUCTURE` statement always matches a physical structure.
Example 1:

\textbf{iobuffer} = \textbf{pre buffer} + \textbf{main buffer}

\begin{verbatim}
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
    }
  }
}
\end{verbatim}

Example 2:
lssd flip-flop = latch + flip-flop + mux

CELL my_latch {
    RESTRICT_CLASS { synthesis scan }
    PIN enable { DIRECTION = input; }
    PIN d { DIRECTION = input; }
    PIN d { DIRECTION = output; }
    FUNCTION { BEHAVIOR {
        @ ( enable ) { q = d; }
    } }
}

CELL my_flip-flop {
    RESTRICT_CLASS { synthesis scan }
    PIN clock { DIRECTION = input; }
    PIN d { DIRECTION = input; }
    PIN q { DIRECTION = output; }
    FUNCTION { BEHAVIOR {
        @ ( 01 clock ) { q = d; }
    } }
}

CELL my_mux {
    RESTRICT_CLASS { synthesis scan }
    PIN dout { DIRECTION = output; }
    PIN din0 { DIRECTION = input; }
    PIN din1 { DIRECTION = input; }
    PIN select { DIRECTION = input; }
    FUNCTION { BEHAVIOR {
        dout = select ? din1 : din0;
    } }
}

CELL my_lssd_flip-flop {
    RESTRICT_CLASS { scan }
    CELLTYPE = block;
    SCAN_TYPE = lssd;
    PIN clock         { DIRECTION = input; }
    PIN master_clock  { DIRECTION = input; }
    PIN slave_clock   { DIRECTION = input; }
    PIN scan_data     { DIRECTION = input; }
    PIN din           { DIRECTION = input; }
    PIN dout          { DIRECTION = output; }
    PIN scan_master   { VIEW = physical; }
    PIN scan_slave    { VIEW = physical; }
    PIN d_internal    { VIEW = physical; }
    FUNCTION { BEHAVIOR {
        @ ( master_clock ) {
            scan_data_master = scan_data;
        }
        @ ( slave_clock & ! clock ) {
            dout = scan_data_master;
        } : ( 01 clock ) {
            dout = din;
        }
    } }
    STRUCTURE {

my_latch U0 {
    enable = master_clock;
    din    = scan_data;
    dout   = scan_data_master;
}
my_flip-flop U1 {
    clock  = clock;
    d      = din;
    q      = d_internal;
}
my_mux U2 {
    select = slave_clock;
    din1   = scan_data_master;
    din0   = dout;
    dout   = scan_data_slave;
}
my_mux U3 {
    select = clock;
    din1   = d_internal;
    din0   = scan_data_slave;
    dout   = dout;
}
)
NON_SCAN_CELL = my_flip-flop {
    clock = clock;
    d     = din;
    q     = dout;
    'b0   = slave_clock;
}
)

Example 3:

  clock tree = chains of clock buffers

CELL my_root_buffer {
    RESTRICT_CLASS { clock }
    PIN i0 { DIRECTION = input; }
    PIN o0 { DIRECTION = output; }
    FUNCTION { BEHAVIOR { o0 = i0; } }
}
CELL my_level1_buffer {
    RESTRICT_CLASS { clock }
    PIN i1 { DIRECTION = input; }
    PIN o1 { DIRECTION = output; }
    FUNCTION { BEHAVIOR { o1 = i1; } }
}
CELL my_level2_buffer {
    RESTRICT_CLASS { clock }
    PIN i2 { DIRECTION = input; }
    PIN o2 { DIRECTION = output; }
    FUNCTION { BEHAVIOR { o2 = i2; } }
}

CELL my_level3_buffer {
    RESTRICT_CLASS { clock }
    PIN i3 { DIRECTION = input; }
    PIN o3 { DIRECTION = output; }
    FUNCTION { BEHAVIOR { o3 = i3; } }
}

CELL my_tree_from_level2 {
    RESTRICT_CLASS { clock }
    PIN in { DIRECTION = input; }
    PIN out { DIRECTION = output; }
    PIN[1:2] level3 { DIRECTION = output; }
    FUNCTION {
        BEHAVIOR { out = in; }
        STRUCTURE {
            my_level2_buffer U1 { i2=in; o2=out; }
            my_level3_buffer U2 { i3=out; o3=level3[1]; }
            my_level3_buffer U3 { i3=out; o3=level3[2]; }
        }
    }
}

CELL my_tree_from_level1 {
    RESTRICT_CLASS { clock }
    PIN in { DIRECTION = input; }
    PIN out { DIRECTION = output; }
    PIN[1:4] level2 { DIRECTION = output; }
    FUNCTION {
        BEHAVIOR { out = in; }
        STRUCTURE {
            my_level1_buffer U1 { i1=in; o1=out; }
            my_tree_from_level2 U2 { i2=out; o2=level2[1]; }
            my_tree_from_level2 U3 { i2=out; o2=level2[2]; }
            my_tree_from_level2 U4 { i2=out; o2=level2[3]; }
            my_tree_from_level2 U5 { i2=out; o2=level2[4]; }
        }
    }
}

CELL my_tree_from_root {
    RESTRICT_CLASS { clock }
    PIN in { DIRECTION = input; }
    PIN out { DIRECTION = output; }
    PIN[1:4] level1 { DIRECTION = output; }
    FUNCTION {
        BEHAVIOR { out = in; }
        STRUCTURE {
            my_root_buffer U1 { i0=in; o0=out; }
        }
    }
}
my_tree_from_level1 U2 { i1=o; o1=level1[1]; }
my_tree_from_level1 U3 { i1=o; o1=level1[2]; }
my_tree_from_level1 U4 { i1=o; o1=level1[3]; }
my_tree_from_level1 U5 { i1=o; o1=level1[4]; }

Example 4:

Multiplexor, showing the conceptional difference between BEHAVIOR and STRUCTURE.

CELL my_multiplexor {
    PIN a { DIRECTION = input; }
    PIN b { DIRECTION = input; }
    PIN s { DIRECTION = input; }
    PIN y { DIRECTION = output; }
    FUNCTION {
        BEHAVIOR {
            // s_a and s_b are virtual internal nodes
            ALF_AND { out = s_a; in[0] = !s; in[1] = a; }
            ALF_AND { out = s_b; in[0] =  s; in[1] = b; }
            ALF_OR  { out = y; in[0] = s_a; in[1] = s_b; }
        }
        STRUCTURE {
            // sbar, sel_a, sel_b are physical internal nodes
            ALF_NOT  { out = sbar; in = s; }
            ALF_NAND { out = sel_a; in[0] = sbar; in[1] = a; }
            ALF_NAND { out = sel_b; in[0] =  s; in[1] = b; }
            ALF_NAND { out = y; in[0] = sel_a; in[1] = sel_b; }
        }
    }
}

6.4 Annotations and attributes for a PIN

This section defines various PIN annotations and attributes.

6.4.1 VIEW annotation

```
VIEW = string;
```

annotates the view where the pin appears, which can take the values shown in Table 6-9.

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

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

annotates the type of the pin, which can take the values shown in Table 6-10.

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

6.4.3 DIRECTION annotation

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

annotates the direction of the pin, which can take the values shown in Table 6-11.

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

Table 6-12 gives a more detailed semantic interpretation for using DIRECTION in combination with PINTYPE.

<table>
<thead>
<tr>
<th>DIRECTION</th>
<th>PINTYPE=digital</th>
<th>PINTYPE=analog</th>
<th>PINTYPE=supply</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>pin receives a digital signal</td>
<td>pin receives an analog signal</td>
<td>pin is a power sink</td>
</tr>
<tr>
<td>output</td>
<td>pin drives a digital signal</td>
<td>pin drives an analog signal</td>
<td>pin is a power source</td>
</tr>
<tr>
<td>both</td>
<td>pin drives or receives a digital signal, depending on the operation mode</td>
<td>pin drives or receives an analog signal, depending on the operation mode</td>
<td>pin is both power sink and source</td>
</tr>
<tr>
<td>none</td>
<td>pin represents either an internal digital signal with no external connection or a feed through</td>
<td>pin represents either an internal analog signal with no external connection or a feed through</td>
<td>pin represents either an internal power pin with no external connection or a feed through</td>
</tr>
</tbody>
</table>

Examples:

- The power and ground pins of regular cells shall have DIRECTION=input.
- A level converter cell shall have a power supply pin with DIRECTION=input and another power supply pin with DIRECTION=output.
- A level converter can have separate ground pins on the input and output side or a common ground pin with DIRECTION=both.

- The power and ground pins of a feed through cell shall have DIRECTION=none.

### 6.4.4 SIGNALTYPE annotation

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

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

**SIGNALTYPE = string ;**

annotates the type of the signal connected to the pin.

The fundamental SIGNALTYPE values are defined in Table 6-13.

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data (default)</td>
<td>general data signal, i.e., a signal that carries information to be transmitted, received, or subjected to logic operations within the CELL.</td>
</tr>
<tr>
<td>address</td>
<td>address signal of a memory, i.e., an encoded signal, usually a bus or part of a bus, driving an address decoder within the CELL.</td>
</tr>
<tr>
<td>control</td>
<td>general control signal, i.e., an encoded signal that controls at least two modes of operation of the CELL, eventually in conjunction with other signals. The signal value is allowed to change during real-time circuit operation.</td>
</tr>
<tr>
<td>select</td>
<td>select signal of a multiplexor, i.e., a decoded or encoded signal that selects the data path of a multiplexor or de-multiplexor within the CELL. Each selected signal has the same SIGNALTYPE.</td>
</tr>
<tr>
<td>enable</td>
<td>general enable signal, i.e., a decoded signal which enables and disables a set of operational modes of the CELL, eventually in conjunction with other signals. The signal value is expected to change during real-time circuit operation.</td>
</tr>
<tr>
<td>tie</td>
<td>the signal needs to be tied to a fixed value statically in order to define a fixed or programmable mode of operation of the CELL, eventually in conjunction with other signals. The signal value is not allowed to change during real-time circuit operation.</td>
</tr>
<tr>
<td>clear</td>
<td>clear signal of a flip-flop or latch, i.e., a signal that controls the storage of the value 0 within the CELL.</td>
</tr>
</tbody>
</table>
Annotations and attributes for a PIN

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

Composite values for SIGNALTYPE shall be constructed using one or more prefixes in combination with certain fundamental values, separated by the underscore (_) character, as shown in the subsequent tables. The scheme for this is shown in Figure 6-1.

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>set signal of a flip-flop or latch, i.e., a signal that controls the storage of the value 1 within the CELL.</td>
</tr>
<tr>
<td>clock</td>
<td>clock signal of a flip-flop or latch, i.e., a timing-critical signal that triggers data storage within the CELL.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan_data</td>
<td>data signal for scan mode</td>
</tr>
<tr>
<td>test_data</td>
<td>data signal for test mode</td>
</tr>
<tr>
<td>bist_data</td>
<td>data signal in BIST mode</td>
</tr>
</tbody>
</table>

**Figure 6-1: Construction scheme for composite SIGNALTYPE values**

**Table 6-13 : Fundamental SIGNALTYPE annotations for a PIN object, continued**
### Table 6-15: Composite SIGNALTYPE annotations based on ADDRESS

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_address</td>
<td>address signal for test mode</td>
</tr>
<tr>
<td>bist_address</td>
<td>address signal for BIST mode</td>
</tr>
</tbody>
</table>

### Table 6-16: Composite SIGNALTYPE annotations based on CONTROL

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>load_control</td>
<td>control signal for switching between load mode and normal mode</td>
</tr>
<tr>
<td>scan_control</td>
<td>control signal for switching between scan mode and normal mode</td>
</tr>
<tr>
<td>test_control</td>
<td>control signal for switching between test mode and normal mode</td>
</tr>
<tr>
<td>bist_control</td>
<td>control signal for switching between BIST mode and normal mode</td>
</tr>
<tr>
<td>read_write_control</td>
<td>control signal for switching between read and write operation</td>
</tr>
<tr>
<td>test_read_write_control</td>
<td>control signal for switching between read and write operation in test mode</td>
</tr>
<tr>
<td>bist_read_write_control</td>
<td>control signal for switching between read and write operation in BIST mode</td>
</tr>
</tbody>
</table>

### Table 6-17: Composite SIGNALTYPE annotations based on ENABLE

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>load_enable</td>
<td>signal enables load operation in a counter or a shift register</td>
</tr>
<tr>
<td>out_enable</td>
<td>signal enables the output stage of an arbitrary cell</td>
</tr>
<tr>
<td>scan_enable</td>
<td>signal enables scan mode of a flip-flop or latch only</td>
</tr>
<tr>
<td>scan_out_enable</td>
<td>signal enables the output of a flip-flop or latch in scan mode only</td>
</tr>
<tr>
<td>test_enable</td>
<td>signal enables test mode only</td>
</tr>
<tr>
<td>bist_enable</td>
<td>signal enables BIST mode only</td>
</tr>
<tr>
<td>test_out_enable</td>
<td>signal enables the output stage in test mode only</td>
</tr>
<tr>
<td>bist_out_enable</td>
<td>signal enables the output stage in BIST mode only</td>
</tr>
<tr>
<td>read_enable</td>
<td>signal enables the read operation of a memory</td>
</tr>
<tr>
<td>write_enable</td>
<td>signal enables the write operation of a memory</td>
</tr>
<tr>
<td>test_read_enable</td>
<td>signal enables the read operation in test mode only</td>
</tr>
<tr>
<td>test_write_enable</td>
<td>signal enables the write operation in test mode only</td>
</tr>
</tbody>
</table>
6.4.5 ACTION annotation

**ACTION** = string ;

anno\ters action of the signal, which can take the values shown in Table 6-19.

### Table 6-17: Composite SIGNALTYPE annotations based on ENABLE, continued

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bist_read_enable</td>
<td>signal enables the read operation in BIST mode only</td>
</tr>
<tr>
<td>bist_write_enable</td>
<td>signal enables the write operation in BIST mode only</td>
</tr>
</tbody>
</table>

### Table 6-18: Composite SIGNALTYPE annotations based on CLOCK

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan_clock</td>
<td>signal is clock of a flip-flop or latch in scan mode</td>
</tr>
<tr>
<td>master_clock</td>
<td>signal is master clock of a flip-flop or latch</td>
</tr>
<tr>
<td>slave_clock</td>
<td>signal is slave clock of a flip-flop or latch</td>
</tr>
<tr>
<td>scan_master_clock</td>
<td>signal is master clock of a flip-flop or latch in scan mode</td>
</tr>
<tr>
<td>scan_slave_clock</td>
<td>signal is slave clock of a flip-flop or latch in scan mode</td>
</tr>
<tr>
<td>read_clock</td>
<td>clock signal triggers the read operation in a synchronous memory</td>
</tr>
<tr>
<td>write_clock</td>
<td>clock signal triggers the write operation in a synchronous memory</td>
</tr>
<tr>
<td>read_write_clock</td>
<td>clock signal triggers both read and write operation in a synchronous memory</td>
</tr>
<tr>
<td>test_clock</td>
<td>signal is clock in test mode</td>
</tr>
<tr>
<td>test_read_clock</td>
<td>clock signal triggers the read operation in a synchronous memory in test mode</td>
</tr>
<tr>
<td>test_write_clock</td>
<td>clock signal triggers the write operation in a synchronous memory in test mode</td>
</tr>
<tr>
<td>test_read_write_clock</td>
<td>clock signal triggers both read and write operation in a synchronous memory in test mode</td>
</tr>
<tr>
<td>bist_clock</td>
<td>signal is clock in BIST mode</td>
</tr>
<tr>
<td>bist_read_clock</td>
<td>clock signal triggers the read operation in a synchronous memory in BIST mode</td>
</tr>
<tr>
<td>bist_write_clock</td>
<td>clock signal triggers the write operation in a synchronous memory in BIST mode</td>
</tr>
<tr>
<td>bist_read_write_clock</td>
<td>clock signal triggers both read and write operation in a synchronous memory in BIST mode</td>
</tr>
</tbody>
</table>
The **ACTION** annotation applies only to pins with certain **SIGNALTYPE** values, as shown in Table 6-20. The rule applies also to any composite **SIGNALTYPE** values based on the fundamental values.

<table>
<thead>
<tr>
<th><strong>Annotation string</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronous</td>
<td>signal acts in synchronous way, i.e., self-triggered</td>
</tr>
<tr>
<td>asynchronous</td>
<td>signal acts in asynchronous way, i.e., triggered by a signal with <strong>SIGNALTYPE</strong> CLOCK or a composite <strong>SIGNALTYPE</strong> with postfix _CLOCK.</td>
</tr>
</tbody>
</table>

**Table 6-21 : POLARITY annotations for a PIN**

<table>
<thead>
<tr>
<th><strong>Annotation string</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>high</td>
<td>signal active high or to be driven high</td>
</tr>
<tr>
<td>low</td>
<td>signal active low or to be driven low</td>
</tr>
<tr>
<td>rising_edge</td>
<td>signal sensitive to rising edge</td>
</tr>
<tr>
<td>falling_edge</td>
<td>signal sensitive to falling edge</td>
</tr>
<tr>
<td>double_edge</td>
<td>signal sensitive to any edge</td>
</tr>
</tbody>
</table>
The **POLARITY** annotation applies only to pins with certain **SIGNALTYPE** values, as shown in Table 6-22. The rule applies also to any composite **SIGNALTYPE** values based on the fundamental values.

### Table 6-22: POLARITY applicable in conjunction with fundamental SIGNALTYPE values

<table>
<thead>
<tr>
<th>fundamental SIGNALTYPE</th>
<th>applicable POLARITY value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>N/A</td>
</tr>
<tr>
<td>address</td>
<td>N/A</td>
</tr>
<tr>
<td>control</td>
<td>mode-specific high or low for composite signaltype</td>
</tr>
<tr>
<td>select</td>
<td>N/A</td>
</tr>
<tr>
<td>enable</td>
<td>Mandatory high or low</td>
</tr>
<tr>
<td>tie</td>
<td>Optional high or low</td>
</tr>
<tr>
<td>clear</td>
<td>Mandatory high or low</td>
</tr>
<tr>
<td>set</td>
<td>Mandatory high or low</td>
</tr>
<tr>
<td>clock</td>
<td>Mandatory high, low, rising_edge, falling_edge, or double_edge, can be mode-specific for composite signaltype.</td>
</tr>
</tbody>
</table>

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

Example:

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

PIN rwc {
  SIGNALTYPE = READ_WRITE_CLOCK;
  POLARITY { READ=rising_edge; WRITE=falling_edge; }
}
```

### 6.4.7 DATATYPE annotation

**DATATYPE = string ;**

Annotates the datatype of the pin, which can take the values shown in Table 6-23.

### Table 6-23: DATATYPE annotations for a PIN object

<table>
<thead>
<tr>
<th>Annotation string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed</td>
<td>result of arithmetic operation is signed 2’s complement</td>
</tr>
<tr>
<td>unsigned</td>
<td>result of arithmetic operation is unsigned</td>
</tr>
</tbody>
</table>

**DATATYPE** is only relevant for bus pins.

### 6.4.8 INITIAL_VALUE annotation

**INITIAL_VALUE = logic_constant ;**
shall be compatible with the buswidth and \texttt{DATATYPE} of the signal. 
\texttt{INITIAL\_VALUE} is used for a downstream behavioral simulation model, as far as the simulator (e.g., a VITAL-compliant simulator) supports the notion of initial value.

### 6.4.9 \texttt{BUFFERTYPE} annotation

\begin{verbatim}
BUFFERTYPE = string ;
\end{verbatim}

can take the values shown in Table 6-24.

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

### 6.4.10 \texttt{DRIVERTYPE} annotation

\begin{verbatim}
DRIVERTYPE = string ;
\end{verbatim}

can take the values shown in Table 6-25.

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

### 6.4.11 \texttt{PARALLEL\_DRIVE} annotation

\begin{verbatim}
PARALLEL\_DRIVE = unsigned ;
\end{verbatim}

specifies the number of parallel drivers.

### 6.4.12 \texttt{SCAN\_POSITION} annotation

\begin{verbatim}
SCAN\_POSITION = unsigned ;
\end{verbatim}

annotates the position of the pin in scan chain, starting with 0.

### 6.4.13 \texttt{STUCK} annotation

\begin{verbatim}
STUCK = string ;
\end{verbatim}
annotes the stuck-at fault model as shown in Table 6-26.

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

### 6.4.14 SUPPLYTYPE

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

```
supplytype_assignment ::= 
    SUPPLYTYPE = supplytype_identifier ;

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

### 6.4.15 SIGNAL_CLASS

The following new keyword for class reference shall be defined:

```
SIGNAL_CLASS
```

A PIN referring 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;
    }
```
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.

6.4.16 SUPPLY_CLASS

The following new keyword for class reference shall be defined:

**SUPPLY_CLASS**

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

Example:

```plaintext
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; }
}
```

6.4.17 Driver CELL and PIN specification

The keywords CELL and PIN can be used as references to existing objects to define a driver cell and pin in a macro, i.e., a cell with CELLTYPE=block.

Example:

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

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

6.4.18 ATTRIBUTE for PIN objects

The attributes shown in Table 6-27 can be used within a PIN object.

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

The attributes shown in Table 6-28 are only applicable for pins within cells with CELLTYPE=memory and certain values of SIGNALTYPE.

<table>
<thead>
<tr>
<th>Attribute item</th>
<th>SIGNALTYPE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROW_ADDRESS_STROBE</td>
<td>clock</td>
<td>samples the row address of the memory</td>
</tr>
<tr>
<td>COLUMN_ADDRESS_STROBE</td>
<td>clock</td>
<td>samples the column address of the memory</td>
</tr>
<tr>
<td>ROW</td>
<td>address</td>
<td>selects an addressable row of the memory</td>
</tr>
<tr>
<td>COLUMN</td>
<td>address</td>
<td>selects an addressable column of the memory</td>
</tr>
<tr>
<td>BANK</td>
<td>address</td>
<td>selects an addressable bank of the memory</td>
</tr>
</tbody>
</table>

The attributes shown in Table 6-29 are only applicable for pins representing double-rail signals.

<table>
<thead>
<tr>
<th>Attribute item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVERTED</td>
<td>represents the inverted value within a pair of signals carrying complementary values</td>
</tr>
<tr>
<td>NON_INVERTED</td>
<td>represents the non-inverted value within a pair of signals carrying complementary values</td>
</tr>
<tr>
<td>DIFFERENTIAL</td>
<td>signal is part of a differential pair, i.e., both the inverted and non-inverted values are always required for physical implementation</td>
</tr>
</tbody>
</table>

The following restrictions apply for double-rail signals:

- The PINTYPE, SIGNALTYPE, and DIRECTION of both pins shall be the same.
• One PIN shall have the attribute INVERTED, the other NON_INVERTED.

• Either both pins or no pins shall have the attribute DIFFERENTIAL.

• POLARITY, if applicable, shall be complementary as follows:
  HIGH is paired with LOW
  RISING_EDGE is paired with FALLING_EDGE
  DOUBLE_EDGE is paired with DOUBLE_EDGE

6.5 Definitions for bus pins

This section defines how to specify bus pins and group pins.

6.5.1 RANGE for bus pins

A one-dimensional bus pin can contain a RANGE statement, defined as follows:

\[
\text{range ::= RANGE \{ unsigned : unsigned \}}
\]

The RANGE statement applies only if the range of valid indices is contiguous. The range is limited by the width of the bus. The possible range for a N-bit wide bus is between 0 and \(2^N\). The possible range of values shall also be the default range.

Example:

A 4-bit wide bus has the following possible range of indices: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15.

\[
\text{RANGE \{ 3 : 13 \}}\text{ specifies the indices } 0, 1, 2, 14, \text{ and } 15 \text{ are invalid.}
\]

In the case where non-contiguous indices are valid, for example 1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 14, 15, the RANGE statement does not apply.

6.5.2 Scalar pins inside a bus

A PIN declared as a bus shall contain the optional \text{pin_instantiation} statement, defined as follows:

\[
\text{pin_instantiation ::= pin_identifier \[ index \] \{ pin_items \}}
\]

where \text{index} and \text{pin_items} are defined in Section 13.5 and Section 13.11, respectively.

A \text{pin_instantiation} statement can also refer to a part of the bus.
Annotations within the scope of the PIN or a higher-level pin_instantiation shall be inherited by a lower-level pin_instantiation (see Section 6.4), as long as their values are applicable for both the bus and each scalar pin within the bus. Values of VIEW, INITIAL_VALUE, and arithmetic models such as 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] { ATTRIBUTE { ROW } CAPACITANCE = 0.03; }
      my_address [1] { VIEW = physical; CAPACITANCE = 0.01; }
      my_address [2] { VIEW = physical; CAPACITANCE = 0.02; }
      my_address [3:4] { ATTRIBUTE { COLUMN } CAPACITANCE = 0.04; }
      my_address [3] { VIEW = physical; CAPACITANCE = 0.02; }
      my_address [4] { VIEW = physical; CAPACITANCE = 0.02; }
   }

6.5.3 PIN_GROUP statement

A pin group shall be defined as follows:

   pin_group ::= 
      PIN_GROUP [ index ] pin_group_identifier {
         pin_items
            MEMBERS { pins }
      }

where pin_items is defined in Section 13.11.

The pins in the MEMBERS field shall refer to previously defined pins. The range of the index, if defined, shall match the number and range of pins in the MEMBERS field.

Annotations within the scope of the PIN contained in the MEMBERS field shall be inherited by the PIN_GROUP, as long as their values are applicable for both the pin and the pin group. Values of VIEW, INITIAL_VALUE, and arithmetic models such as CAPACITANCE shall not be inherited, since a particular value cannot apply at the same time to the pin and the pin group.

A pin group with VIEW=functional shall be treated like a bus pin in the functional netlist. It shall appear in the netlist in place of the first defined pin within the MEMBERS field.
Example 1:

```plaintext
PIN my_address_1 {DIRECTION = input; VIEW = physical; CAPACITANCE = 0.01;}
PIN my_address_2 {DIRECTION = input; VIEW = physical; CAPACITANCE = 0.02;}
PIN my_address_3 {DIRECTION = input; VIEW = physical; CAPACITANCE = 0.02;}
PIN my_address_4 {DIRECTION = input; VIEW = physical; CAPACITANCE = 0.02;}
PIN_GROUP [1:2] my_address_1_2 {
    ATTRIBUTE { ROW }
    CAPACITANCE = 0.03;
    MEMBERS { my_address_1 my_address_2 }
}
PIN_GROUP [1:2] my_address_3_4 {
    ATTRIBUTE { COLUMN }
    CAPACITANCE = 0.03;
    MEMBERS { my_address_3 my_address_4 }
}
PIN_GROUP [1:4] my_address {
    VIEW = functional;
    CAPACITANCE = 0.07;
    MEMBERS { my_address_1 my_address_2 my_address_3 my_address_4 }
}
```

Pairs of complementary pins, differential pins in particular, are special cases of pin groups.

Example 2:

```plaintext
CELL my_flip-flop {
    PIN CLK { DIRECTION=input; SIGNALTYPE=clock; POLARITY=rising_edge; }
    PIN D { DIRECTION=input; SIGNALTYPE=data; }
    PIN Q { DIRECTION=output; SIGNALTYPE=data; ATTRIBUTE { NON_INVERTED } }
    PIN Qbar { DIRECTION=output; SIGNALTYPE=data; ATTRIBUTE { INVERTED } }
    PIN_GROUP [0:1] Q_double_rail { RANGE { 1 : 2 } MEMBERS { Q Qbar } }
}
```

The pins Q and Qbar are complementary. Their valid set of data comprises ‘b01==='d1 and ‘b10==='d2. The values ‘b00==='d0 and ‘b11==='d3 are invalid.

```plaintext
CELL my_differential_buffer {
    PIN DIN { DIRECTION=input; ATTRIBUTE { DIFFERENTIAL NON_INVERTED } }
    PIN DINN { DIRECTION=input; ATTRIBUTE { DIFFERENTIAL INVERTED } }
    PIN DOUT { DIRECTION=output; ATTRIBUTE { DIFFERENTIAL NON_INVERTED } }
    PIN DOUTN { DIRECTION=output; ATTRIBUTE { DIFFERENTIAL INVERTED } }
    PIN_GROUP [0:1] DI { RANGE { 1 : 2 } MEMBERS { DIN DINN } }
    PIN_GROUP [0:1] DO { RANGE { 1 : 2 } MEMBERS { DOUT DOUTN } }
}
```

The pins DIN and DINN represent a pair of differential input pins. The pins DOUT and DOUTN represent a pair of differential output pins.

### 6.6 Annotations for other objects

This section defines the annotations for CLASS and VECTOR.
6.6.1 PURPOSE for CLASS

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

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

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

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

6.6.2 OPERATION for VECTOR

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

```
operation_assignment ::= OPERATION = operation_identifier ;
```

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

\begin{verbatim}
operation_identifier ::= 
  read 
  | write 
  | read_modify_write 
  | write_through 
  | start 
  | end 
  | refresh 
  | load 
  | iddq
\end{verbatim}

Their definitions are:

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

The \texttt{EXISTENCE_CLASS} (see Section 9.2.3) within the context of a \texttt{VECTOR} shall be used to identify which operations can be combined in the same mode. \texttt{OPERATION} is orthogonal to \texttt{EXISTENCE_CLASS}. The \texttt{EXISTENCE_CLASS} statement is only necessary, if there is more than one mode of operation.

Example 1:

\begin{verbatim}
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 -> 01 CAS -> 10 CAS 
) ) { 
  OPERATION = write; EXISTENCE_CLASS = normal_mode; }
VECTOR ( ! WE & & (?! addr -> 01 CAS -> X? dout -> 10 CAS -> ?X dout ) ) {
\end{verbatim}
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 2:

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

6.7 ILLEGAL statement for VECTOR

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.

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 and violation are defined in Section 13.7 and Section 8.1, 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 1:

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. Sometimes the address space is not continuous, i.e., it can contain holes in the middle. In this case, a MIN or MAX value for legal addresses would not be sufficient. On the other hand, a boolean_expression can always exactly describe the legal and illegal address space.

Example 2:

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

6.8 TEST statement

A CELL can contain a TEST statement, which is defined as follows:

test ::= 
  TEST { behavior }

The purpose is to describe the interface between an externally applied test algorithm and the CELL. The behavior statement within the TEST statement uses the same syntax as the behavior statement within the FUNCTION statement. However, the set of used variables is different. Both the TEST and the FUNCTION statement shall be self-contained, complete and complementary to each other.

6.9 Physical bitmap for memory BIST

This section defines the physical bitmap for memory BIST. This is a particular case of the usage of the TEST statement.
6.9.1 Definition of concepts

The physical architecture of a memory can be described by the following parameters:

**BANK index**: A memory can be arranged in one or several banks, each of which constitutes a two-dimensional array of rows and columns.

**ROW index**: A row of memory cells within one bank shares the same row decoder line.

**COLUMN index**: A column of memory cells within one bank shares the same data bit line and, if applicable, the same sense amplifier.

![Illustration of a physical memory architecture, arranged in banks, rows, columns](image)

---

The physical memory architecture is not evident from the functional description and the pins involved in the functional description of the memory. Those pins are called logical pins, e.g., logical address and logical data.

A memory BIST tool needs to know which logical address and data corresponds to a physical row, column, or bank in order to write certain bit patterns into the memory and read expected bit patterns from the memory. Also, the tool needs to know whether the physical data in a specific location is inverted or not with respect to the corresponding logical data.
A mapper between physical rows, columns, banks, data and logical addresses, and data pins shall be part of the library description of a memory cell.

The physical row, column, and bank indices can be modeled as virtual inputs to the memory circuit. The data to be written to a physical memory location can also be modeled as a virtual input. The data to be read from a physical memory location can be modeled as a virtual output. Since every data that is written for the purpose of test also needs to be read, the data can be modeled as a virtual bidirectional pin. A virtual pin is a pin with `VIEW=none`, i.e., the pin is not visible in any netlist.

### 6.9.2 Definitions of pin ATTRIBUTE values for memory BIST

The special pin `ATTRIBUTE` values shown in Table 6-30 shall be defined for memory BIST.

<table>
<thead>
<tr>
<th>Attribute item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROW_INDEX</td>
<td>pin is a bus with a contiguous range of values, indicating a physical row of a memory</td>
</tr>
<tr>
<td>COLUMN_INDEX</td>
<td>pin is a bus with a contiguous range of values, indicating a physical column of a memory</td>
</tr>
<tr>
<td>BANK_INDEX</td>
<td>pin is a bus with a contiguous range of values, indicating a physical bank of a memory</td>
</tr>
<tr>
<td>DATA_INDEX</td>
<td>pin is a bus with a contiguous range of values, indicating the bit position within a data bus of a memory</td>
</tr>
<tr>
<td>DATA_VALUE</td>
<td>pin represents a value stored in a physical memory location</td>
</tr>
</tbody>
</table>
These attributes apply to the pins of the BIST wrapper around the memory rather than to the pins of the memory itself.

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

### 6.9.3 Explanatory example

One-dimensional arrays with **SIGNALTYPE=address** (here: **PIN[3:0] addr**) shall be recognized as address pins to be mapped, involving other one-dimensional arrays with **ATTRIBUTE \{ ROW_INDEX \}** (here: **PIN[1:0] row**) and **ATTRIBUTE \{ COLUMN_INDEX \}** (here: **PIN[3:0] col**). This memory has only one bank. Therefore, no one-dimensional array with **ATTRIBUTE \{ BANK_INDEX \}** exists here.

One-dimensional arrays with **SIGNALTYPE=data** (here: **PIN[3:0] Din** and **PIN[3:0] Dout**) shall be recognized as data pins to be mapped, involving other one-dimensional arrays with **ATTRIBUTE \{ DATA_INDEX \}** (here: **PIN[1:0] dat**) and scalar pins with **ATTRIBUTE \{ DATA_VALUE \}** (here: **PIN bit**).

Note: Since the data buses are 4-bits wide, the data index is 2-bits wide, since \(2 = \log_2(4)\).

Base Example:

```verbatim
CELL my_memory {
    PIN[3:0] addr { DIRECTION=input; SIGNALTYPE=address; }
    PIN[3:0] Din  { DIRECTION=input; SIGNALTYPE=data; }
    PIN[3:0] Dout { DIRECTION=output; SIGNALTYPE=data; }
    PIN[3:0] bits[0:15] { DIRECTION=none; VIEW=none; SCOPE=behavior; }
    PIN write_enb { DIRECTION=input; SIGNALTYPE=write_enable;
        POLARITY=high; ACTION=asynchronous; }
    PIN[1:0] dat { ATTRIBUTE \{ DATA_INDEX \} DIRECTION=none; VIEW=none; }
    PIN bit { ATTRIBUTE \{ DATA_VALUE \} DIRECTION=both; VIEW=none; }
    PIN[1:0] row {
        ATTRIBUTE \{ ROW_INDEX \} RANGE { 0: 3 }
        DIRECTION=input; VIEW=none; }
    PIN[3:0] col {
        ATTRIBUTE \{ COLUMN_INDEX \} RANGE { 0 : 15 }
        DIRECTION=input; VIEW=none; }
    FUNCTION {
        BEHAVIOR {
            Dout = bits[addr];
            @ (write_enb) { bits[addr] = Din; }
        }
    } /*different physical architectures are shown in the following examples*/
}
```

Example 1
Modeling for Test

Physical bitmap for memory BIST

TEST {
    BEHAVIOR {
        // map row and column index to logical address
        addr[1:0] = row[1:0];
        addr[3:2] = col[3:2];
        // map column index to logical data index
        dat[1:0] = col[1:0];
        // map physical data to input and output data
        Din[dat] = bit;
        bit = Dout[dat];
    }
}

Example 2
TEST {
  BEHAVIOR {
    // map row and column index to logical address
    addr[1:0] = row[1:0];
    addr[3:2] = col[1:0];
    // map column index to logical data index
    dat[1:0] = col[3:2];
    // map physical data to input and output data
    Din[dat] = bit;
    bit = Dout[dat];
  }
}
Example 3

```
TEST {
    BEHAVIOR {
        // map row and column index to logical address
        addr[0] = row[1];
        addr[1] = row[0] ^ row[1]
        // map column index to logical data index
        dat[0] = col[1];
        dat[1] = col[3];
        // map physical data to input and output data
    }
}
```

Notes:

1. This enables the description of a complete bitmap of a memory in a compact way.
2. The `RANGE` feature is not restricted to BIST. It can be used to describe a valid contiguous range on any bus. This alleviates the need for interpreting a `VECTOR` with `ILLEGAL` statement to get the valid range. However, the `VECTOR` with `ILLEGAL` statement is still necessary to describe the behavior of a device when illegal values are driven on a bus.
3. The `TEST` statement with `BEHAVIOR` allows for generalization from memory BIST to any test vector generation requirement, e.g., logic BIST. The only necessary additions would be other `PIN ATTRIBUTE` describing particular features to be recognized by the test vector generation algorithm for the target test algorithm.