ALF Tutorial 2000 version 0
March 15, 2000

VDSM Modeling with ALF

Wolfgang Roethig
wroethig@eda.org
Outline

• Introduction
• ALF language
• ALF applications
• Conclusion
Introduction

ALF = Advanced Library Format

- Motivation
- Background
- Status
- Contents of ALF library
- Design flow with ALF
Motivation

• Nanometer technology
  – Higher complexity: need for abstraction
  – Smaller geometry: need for accuracy

• Tools merge
  – all design steps from RTL synthesis to layout become backend
  – strong interaction between electrical analysis and optimization

• ALF provides the adequate datamodel
  – complete and comprehensive library representation
  – covers all aspects of functional, electrical, physical modeling
  – neutral format, open standard, freely available
Background

• ALF started 1996 as OVI workgroup
  – primary focus on synthesis, power, timing libraries
• ALF version 1.0 released Nov. 1997
  – contains function description, timing, power, DFT library for ASIC
• Collaboration with SI2 on OLA since Feb. 1998
  – Built on ALF and DCL technology
  – Goal: open library API for all design tools
• ALF version 1.1 released Apr. 1999
  – new items: signal integrity, more abstract macromodeling capabilities
• ALF version 2.0 release planned for June 2000
  – new items: layout, BIST, interconnect analysis
Status

• Contributors and reviewers across the industry
  – Cadence, IBM, Infineon, Logicvision, LSI Logic, Mentor Graphics, Monterey, NEC, Philips, Sente, SI2, Synopsys
  – 1.5 day face-to-face meeting per month with typical attendance > 10 people
  – 2 additional conference calls per month in average

• Emerging ALF support for new EDA tools
  – Sente, Library Technologies, Magma, Silicon Metrics, Tera Systems

• Indirect ALF support through OLA
  – Use of SI2’s OLAWorx - NDCL compiler suite
  – Synthesis & timing analysis from Cadence, Mentor Graphics

• Initiation of IEEE standard planned in Q2 2000
Contents of ALF library

• Cell data
  – Function
    • golden reference for specification, characterization, synthesis, formal verification, test, simulation
  – Electrical performance data
    • Characterization data for timing, power, signal integrity covers superset of SDF, DCL data model
  – Supplementary data for synthesis, test, layout

• Library data
  – Global technology data
    • Version 2.0 will have superset of LEF
  – Models for interconnect analysis
    • includes crosstalk and noise
Similarities with other library formats

• Readability
  – human-readable ASCII source
  – ALF language uses English keywords, no acronyms

• Data representation
  – easy to map Synopsys .lib or Cadence TLF constructs into ALF
  – However, the primary intent is not to replace existing formats for commodity, but to add value
Distinction from other library formats

• More general modeling language
• Describes the modeling concepts along with the data
  – Example: Where other libraries may contain parameters or K-factors, ALF supplies the complete equation
• Two most distinguishing concepts
  – vector_expression language provides an abstraction for describing dynamic behavior, useful for characterization, analysis, test from RTL to post-layout
  – arithmetic_model construct provides a general and concise way of expressing mathematical relationships between measurable quantities (e.g. electrical characterization data) in the library
Design flow with ALF

• Library characterization
  – Primary input: cell & technology specification in ALF
  – Primary output: cell & technology data in ALF

• Downstream library generation
  – Verilog / VHDL
  – OLA

• Model generation
  – Front end design planning: models for softmacros in ALF
  – Back end hierarchical design: models for hardmacros in ALF

• Repository for functional, electrical, physical library data
Design flow with ALF (cont.)

- ALF library spec.
- Library characterization tool
- ALF cell & technology library
- Macromodel characterization tool
- ALF macro models

Simulation model generator
- Verilog VHDL models

OLA compiler
- Long term
- Short term
- compiled ALF library

HDL design description
- Design planning tool
- Structural design description
- Physical design tool
- Layout database

For hierarchical design
- Block
- Model generation tool
- ALF block-based models
ALF language

• ALF grammar
  – Symbols
  – Expressions

• ALF statements
  – Objects in a library
  – Model data (arithmetic model)
Conventions used:

 ::= start of syntax definition
| alternative syntax definition
item normal syntax item
**item** bold item appears verbatim in ALF source
*italic_item* italic part is for semantic explanation
[ item ] optional syntax item, can appear once
{ item } optional syntax item, can be repeated multiple times

Comments in ALF source:

 /* this is an enclosed comment */
 // this is a comment until end of line
The grammar follows a common construction principle
Each object is defined by a keyword, an optional name and an optional value
Certain objects may have multiple values
Certain objects may have children objects

object ::= 
    keyword [ name ] [ = value ] ;
|    keyword [ name ] { value { value } }
|    keyword [ name ] [ = value ] { object { object } }

An object is always terminated either by a semicolon or by a values or children objects enclosed by curly braces
This construction principle allows an ALF reader to identify start and end of an object without semantic interpretation, thus allowing inclusion of customized objects
ALF grammar (cont.)

- A keyword is a symbol with optional index
- A name is either a symbol with optional index or an expression
- A value is either a symbol with optional index or an expression or a number

```
keyword ::= symbol [ index ]
name ::= symbol [ index ] | ( expression )
value ::= symbol [ index ] | expression | number
```

Examples:

- number: -3.8, 5E-9, 100
- index: [3], [1:50]
- symbol: C144, HEADER, @, <my_symbol>, “example.alf”
- expression: \(3*A + 0.5*B/C\), \(X \&\& !Y\), \((Z=='b5)\)\? A1 : A2
Symbols

symbol ::=  
  non_escaped_identifier | quoted_string  
  | placeholder_identifier | escaped_character  
  | hierarchical_identifier | based_literal | @ | :

non_escaped_identifier contains alphanumerical characters, _, $, #
quoted_string starts and ends with ", contains any character including whitespace
escaped_identifier starts with \, contains any character except whitespace
  escape character escapes the whole word, not just the following character
placeholder_identifier symbolizes a placeholder within a TEMPLATE
  starts with <, ends with >, contains alphanumerical characters, __, $, #
hierarchical_identifier contains a dot . as hierarchical delimiter
  dot overrides the escape character, must be escaped by itself, if necessary
based_literal starts with ‘b or ‘o or ‘d or ‘h
  symbolizes a binary or octal or decimal or hexadecimal value, respectively
@, : are special symbols used in a particular context
Symbols (cont.)

Symbol are primarily used for keywords or names
Symbols used as values usually refer to the name of an already declared object

Purpose of the keyword

Declare an object of a certain type, e.g. **LIBRARY, CELL, PIN**

In general, only a **non_escaped_identifier** is used as a keyword

Purpose of the name

Distinguish different objects of the same type, e.g. **PIN A, PIN Z**

Make reference to an object by name, e.g. **Z = ! A**

The allowed set of symbols or expressions depends on the type of object

Case-sensitivity

Keywords are case-insensitive

However, in this tutorial, keywords will be systematically in upper case

Names and values are case-insensitive within ALF

However, a translator or compiler targeting a case-sensitive application shall preserve the case of the name in the declaration of the object
Expressions

expression ::= arithmetic_expression | boolean_expression | vector_expression | statetable_header_expression | statetable_body_expression

arithmetic_expression
describes calculation of numbers involving arithmetic models

boolean_expression
describes calculation of states involving logical variables

vector_expression
describes a sequence of events involving logical variables

statetable_header_expression
defines logic variables in a STATETABLE

statetable_body_expression
defines logic values in a STATETABLE
Arithmetic expressions

Symbols for operators for \texttt{arithmetic\_expression}

Unary operators
\begin{itemize}
  \item \texttt{+} neutral operator
  \item \texttt{-} change of sign
\end{itemize}

Binary operators
\begin{itemize}
  \item \texttt{+} add
  \item \texttt{-} subtract
  \item \texttt{*} multiply
  \item \texttt{/} divide
  \item \texttt{%} modulo
  \item \texttt{**} power
\end{itemize}

Increasing priority

Macro operators with one argument
\begin{itemize}
  \item \texttt{ABS} absolute value
  \item \texttt{EXP} natural exponent
  \item \texttt{LOG} natural logarithm
\end{itemize}

Macro operators with multiple arguments:
\begin{itemize}
  \item \texttt{MIN} smallest value
  \item \texttt{MAX} largest value
\end{itemize}

Example for \texttt{arithmetic\_expression}

\begin{align*}
-A + 0.5^*(\texttt{EXP(C\%4)} - 3.1^*\texttt{MAX(A/4, B)})
\end{align*}
Boolean expressions

Symbols for logic values in boolean_expression

bit_literal

Pure logic values
- 0 - low
- 1 - high
- X - unknown

Logic values with weak drive strength
- L - low
- H - high
- W - unknown

Special logic values
- Z - high impedance
- U - uninitialized

based_literal

binary base
Example: ‘b1101

octal base
Example: ‘o15

decimal base
Example: ‘d13

hexadecimal base
Example: ‘hD
Boolean expressions (cont.)

Symbols for logic operators in boolean_expression

Logical operations

<table>
<thead>
<tr>
<th>Logical or</th>
<th>Logical and</th>
<th>Exclusive or</th>
<th>Exclusive nor</th>
<th>Logical inversion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>&amp;&amp;</td>
<td>^</td>
</tr>
</tbody>
</table>

Increasing priority

Bitwise operations

<table>
<thead>
<tr>
<th>Bitwise or</th>
<th>Bitwise and</th>
<th>Exclusive or</th>
<th>Exclusive nor</th>
<th>Bitwise inversion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&amp;</td>
<td></td>
<td>^</td>
<td>~^</td>
</tr>
</tbody>
</table>

If-then-else operation

\[
\text{if\_then\_else\_boolean\_expression ::=}
\]

\[
\text{if\_boolean\_expression ? then\_boolean\_expression :}
\]

\[
\text{else\_if\_boolean\_expression ? then\_boolean\_expression : }
\]

\[
\text{else\_boolean\_expression}
\]
Boolean expressions (cont.)

Symbols for logic operators in `boolean_expression` (continued)

<table>
<thead>
<tr>
<th>Integer arithmetic operations</th>
<th>Logical reduction operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>&amp;, ~&amp;</td>
</tr>
<tr>
<td>-</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>^, ~^</td>
</tr>
<tr>
<td>/</td>
<td>==, !=</td>
</tr>
<tr>
<td>%</td>
<td>&gt;, &lt;</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>&gt;=, &lt;=</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td></td>
</tr>
</tbody>
</table>

Examples for `boolean_expression`

- `A && B | C`
- `C[5:3] >> 2`
- `C1 ? D1 : (C2 | C3) ? D2 : 'bX`
- `M >= N`
- `~& D[1:100]`
Vector expressions

Symbols for operators in vector_expression

edge Literal describes a transition between logic values
edge Literal ::= bit Literal bit Literal | based Literal based Literal

The simplest vector_expression describes a single event
vector_single_event ::= edge_literal boolean_expression

Examples
  01 A transition from logical 0 to logical 1 on A
  ‘05’07 B[3:1] transition from octal 5 to octal 7 on B[3:1]
Vector expressions (cont.)

Symbols for operators in `vector_expression` (continued)

Symbolic **edge_literals**

- `?!` arbitrary transition
- `?~` transition to bitwise complementary state
- `?-` non-transition

Symbolic **bit_literals** for use in **edge_literals**

- `?` arbitrary state (don’t care)
- `*` arbitrary number of transitions (don’t monitor)

Examples

```
0? A  

```

transition from logical 0 to arbitrary state on A

```
0* A

```

steady logical 0 followed by arbitrary transitions on A
Vector expressions (cont.)

Symbols for operators in vector_expression (continued)

Atomic relational operators for events

- \(\rightarrow\)  LHS immediately followed by RHS (no events in-between)
- \(\sim\rightarrow\)  LHS eventually followed by RHS (arbitrary number of events in-between)
- \&\  LHS and RHS occur simultaneously
- |\  LHS or RHS occur as alternatives

Complex relational operators for events

- \(<\rightarrow\)  \(LHS \leftrightarrow RHS \implies LHS \rightarrow RHS \mid RHS \rightarrow LHS\)
- \&\>\  \(LHS \&\> RHS \implies LHS \& RHS \mid LHS \rightarrow RHS\)
- <&>\  \(LHS <&> RHS \implies LHS \&> RHS \mid RHS \rightarrow LHS\)

Operators for conditional events

\[
\text{vector conditional event ::=} \\
\text{vector expression} \&\& \text{condition boolean expression} \\
| \text{condition boolean expression} \&\& \text{vector expression} \\
| \text{if boolean expression ? then vector expression :} \\
\{\text{else if boolean expression ? then vector expression :} \}
\text{else vector expression}
\]
ALF statements

• Statements defining objects in a library
  library_specific_objects

• Statements defining model data in a library
  arithmetic_models

• Statements for efficient library representation
  generic_objects

• Auxiliary statements
Objects in a library

Physical domain

Functional domain

LIBRARY

SUBLIBRARY

LAYER

VIA

RULE

SITE

ANTENNA

PATTERN

geometric_models

CELL

PRIMITIVE

PIN

FUNCTION

VECTOR

NODE

WIRE
Objects in a library (cont.)

- Objects of a library have names
- The ALF language naturally defines the relationship between objects
- The ALF language is modular
  - Library need not include every possible object from the functional or physical domain

```
LIBRARY my_library {
    WIRE my_wire_model_1 { /* data for wire model */}
    WIRE my_wire_model_2 { /* data for wire model */}
    CELL my_cell {
        /* data for cell */
        PIN my_pin_A { /* data for pin */ }
        PIN my_pin_B { /* data for pin */ }
        FUNCTION { /* function description */ }
        VECTOR ( vector_expression ) { /* characterization data */
        }
        VECTOR ( vector_expression ) { /* characterization data */
    }
}
```
Objects in the functional domain

- LIBRARY contains CELLs, WIREs, functional PRIMITIVEs.
- A LIBRARY may be divided into SUBLIBRARIES, each of which may contain CELLs, WIRE load models, PRIMITIVEs.
- CELLs contain PINs, a FUNCTION description and VECTORs.
- WIREs contain interconnect modeling data, eventually using NODEs and VECTORs.
- VECTOR contains characterization data, for which a specific stimulus is required. The stimulus is described by a boolean_expression for static measurements or by a vector_expression for transient measurements.
- LIBRARY, SUBLIBRARY, CELL, WIRE, PIN may contain characterization data, for which no specific stimulus is required.
- Characterization data is represented in form of arithmetic_models.
- PRIMITIVEs are technology-independent descriptions. They contain PINs and FUNCTION only, no characterization data.
- In hierarchical design, complex CELLs may also contain PRIMITIVEs and WIREs.
Objects in the physical domain

• LIBRARY or SUBLIBRARY may contain LAYER, VIA, RULE, SITE, ANTENNA.
• VIA, RULE, SITE, ANTENNA may contain PATTERN descriptions.
• CELL may contain BLOCKAGE descriptions.
• PIN may contain physical PORT descriptions.
• Each PATTERN, BLOCKAGE, PORT description is associated with a specific layer and may contain geometric_models, describing the form and shape of the object on that layer.
• LAYER, VIA, RULE, SITE, ANTENNA may also contain arithmetic_models, describing mathematical relationships and constraints related to geometrical and electrical properties associated with the objects.
Model data in a library

- Library data is described by `arithmetic_models` using context-sensitive keywords
  - Example: keyword `CAPACITANCE`
    - Wire capacitance in the context of `WIRE`
    - Pin capacitance in the context of `PIN`
    - Load capacitance as argument of a `DELAY` model
    - Load capacitance `LIMIT` for a `PIN`

- Model statements usually contain auxiliary statements
  - Purpose: complete definition of semantics within the context

- Principle of inheritance
  - Applies for unnamed model statements containing only definitions, no data
    - Definitions are inherited by all models of the same type within the same context
    - Definitions are propagated to the models within the children objects
    - Definitions can be overridden locally
  - Example: measurement units in `LIBRARY`, `SUBLIBRARY`, `CELL`
Arithmetic model

• **Data specification**
  – Trivial arithmetic models
    • Model data are single numbers
  – Equation-based models
    • Model data are fitted into an equation
  – Table-based models
    • Model data are represented in table form
  – Nested models
    • Equation is applied to raw model data, which are in table form

• **Data qualifier specification**
  – Case 1: Model is completely specified by the keyword
    • Model statement contains the data directly
  – Case 2: Model needs qualifiers, such as LIMIT, MIN, MAX, RISE, FALL
    • Model containers contain the models which contain the data
Trivial arithmetic model

\[ \text{definitions\_for\_arithmetic\_model ::= } \]
\[ \quad \text{model\_keyword \{ auxiliary\_objects \}} \]

\[ \quad \text{CAPACITANCE \{ UNIT = 1e-15; \}} \]

\[ \text{trivial\_arithmetic\_model ::= } \]
\[ \quad \text{model\_keyword [ name ] = number ;} \]
\[ \quad | \quad \text{model\_keyword [ name ] = number \{ auxiliary\_objects \}} \]

\[ \quad \text{CAPACITANCE = 4.5 ;} \]

\[ \quad \text{CAPACITANCE = 4.5 \{ UNIT = 1e-15; \}} \]
Equation-based arithmetic model

\[
equation\_based\_arithmetic\_model ::= \\
\hspace{1em} model\_keyword \{ \{ auxiliary\_objects \} \\
\hspace{2em} \text{HEADER} \{ \text{argument\_objects} \} \\
\hspace{3em} \text{EQUATION} \{ \text{arithmetic\_expression} \} \\
\}
\]

\[
\text{argument\_object ::= } \\
\hspace{1em} \text{argument\_keyword} [ \text{name} ]; \\
\mid \hspace{0em} \text{argument\_keyword} [ \text{name} ] \{ \text{auxiliary\_objects} \}
\]

\[
\begin{align*}
\text{CAPACITANCE} \{ \text{UNIT} = 1e\text{-}15; \\
\hspace{1em} \text{HEADER} \{ \\
\hspace{2em} \text{VOLTAGE} \ V \{ \text{UNIT} = 1; \} \\
\hspace{3em} \text{TEMPERATURE} \ T \{ \text{UNIT} = 1; \} \\
\hspace{2em} \} \\
\hspace{1em} \text{EQUATION} \{ 4.5 + 0.1*(V-1.8) + 0.002*(T-25) \} \\
\}
\end{align*}
\]
Table-based arithmetic model

\[
\text{table\_based\_arithmetic\_model ::=}
\]
\[
\text{model\_keyword \{ \{ auxiliary\_objects \} \}
  \text{HEADER \{ table\_argument\_objects \}
  \text{TABLE \{ numbers \}}
\}
\]
\[
\text{table\_argument\_object ::=}
\]
\[
\text{argument\_keyword [ name ]
  \{ \{ auxiliary\_objects \} \text{TABLE \{ numbers \}} \}
\]
\]
\[
\text{CAPACITANCE \{ UNIT = 1e-15;
  \text{HEADER \{ }
    \text{VOLTAGE \{ TABLE \{ 1.6 1.8 2.0 \} \}
    \text{TEMPERATURE \{ TABLE \{ 25 125 \} \}
  \}
  \text{TABLE \{
    4.48 4.50 4.52
    4.68 4.70 4.72
  \}
  \}
\]
\]
Nested arithmetic model

\[ \text{nested\_arithmetic\_model ::= model\_keyword \{} \{ auxiliary\_objects \} \]
\[
\quad \text{HEADER} \{} table\_based\_arithmetic\_models \}
\quad \text{EQUATION} \{} \text{arithmetic\_expression} \}
\]

CAPACITANCE \{ UNIT = 1e-15;
\quad \text{HEADER} \{
\quad \quad \text{CAPACITANCE C0} \{
\quad \quad \quad \text{HEADER} \{
\quad \quad \quad \quad \text{VOLTAGE} \{} \text{TABLE} \{ 1.6 1.8 2.0 \} \}
\quad \quad \quad \}
\quad \quad \text{TABLE} \{ 4.48 4.50 4.52 \}
\quad \}
\quad \text{TEMPERATURE T0} \{
\quad \quad \text{TABLE} \{ 25 125 \} \}
\}
\quad \text{EQUATION} \{ C0 + 0.002*(T0-25) \}
\}
Arithmetic model container

\texttt{arithmetic\_model\_container ::= }
\texttt{model\_container\_keyword }
\texttt{\{ \{ auxiliary\_objects \} arithmetic\_model\_containers \}}
\texttt{model\_container\_keyword }
\texttt{\{ \{ auxiliary\_objects \} arithmetic\_models \}}

\texttt{LIMIT \{}
  \texttt{CAPACITANCE \{}
    \texttt{MAX \{}
      \texttt{HEADER \{}
        \texttt{FREQUENCY \{} \texttt{UNIT = 1e6;}
        \texttt{TABLE \{} \texttt{1 10 100} \texttt{\}}
      \texttt{\}}
      \texttt{TABLE \{} \texttt{0.5 0.4 0.04} \texttt{\}}
    \texttt{\}}
  \texttt{\}}
  \texttt{CAPACITANCE \{}$
    \texttt{RISE \{} \texttt{MIN = 4.4; TYP = 4.5; MAX = 4.6;} \texttt{\}}
    \texttt{FALL \{} \texttt{MIN = 4.3; TYP = 4.5; MAX = 4.7;} \texttt{\}}$
  \texttt{\}}$
\texttt{\}}$

March 15, 2000  ALF tutorial  37
Model data range and default

- Data range specification
  - Data in **TABLE** within *table_argument_object* must be in ascending order
  - Per default, the range of data specifies the range of validity
  - Alternatively, the range of validity can be specified using **MIN, MAX** as *auxiliary_objects* in *argument_object* or *table_argument_object*.
  - Note the context-sensitivity of the **MIN, MAX** keywords
    - Model qualifier: keywords for *arithmetic_model*.
    - Range specification: keywords for *auxiliary_objects* in the **HEADER**.

- Default specification
  - Reason: It may not always be possible for the application to calculate the value of the *argument_object* or *table_argument_object* data.

```plaintext
CAPACITANCE {
  HEADER {
    VOLTAGE V { DEFAULT = 1.8, MIN = 1.6; MAX = 2.0; } 
  }
  EQUATION { 4.5 + 0.1*(V-1.8) } 
}
```
Statements for efficient library representation

• The ALIAS, CONSTANT, INCLUDE statements can be used to make the library more readable or more maintainable.
• The TEMPLATE, GROUP statements can be used to make the library more compact.
• The CLASS statement can be used to define statements that can be inherited by multiple objects
• The KEYWORD statement can be used to extend the set of context-sensitive keywords for customized arithmetic models and other statements.
• The ATTRIBUTE, PROPERTY statements can be used to include customized attributes or properties for objects in the library.
• All of the above statements apply within the context of the object where they appear and within the context of its children objects.
alias ::= 
   ALIAS identifier = identifier ;

ALIAS foo = bar;    // definition of alias
foo = my_symbol;   // usage of alias
bar = my_symbol;   // equivalent statement without alias

constant ::= 
   CONSTANT identifier = number ;

CONSTANT c0 = 4.5;   // definition of alias
CAPACITANCE = c0;  // usage of alias
CAPACITANCE = 4.5;  // equivalent statement without alias

include ::= 
   INCLUDE quoted_string ;

LIBRARY my_library {
   INCLUDE "technology.alf" ;  // put contents of file here
   INCLUDE "cells.alf" ;       // put contents of file here
}

template ::= 
    TEMPLATE identifier { objects }

TEMPLATE \2D_LUT { // definition of the template
  CAPACITANCE { PIN = <out>; TABLE {20 40 80 160 } }
  SLEWRATE { PIN = <in>; TABLE { 0.4 0.8 } }
}

SLEWRATE { PIN = Z;
  HEADER { \2D_LUT { out=Z; in=A; } } // placeholder-replacement by name
  TABLE { 0.25 0.34 0.58 1.12 0.31 0.39 0.62 1.15 } }

SLEWRATE { PIN = Z;
  HEADER { \2D_LUT { Z A } } // placeholder-replacement by order
  TABLE { 0.25 0.34 0.58 1.12 0.31 0.39 0.62 1.15 } }

SLEWRATE { PIN = Z;
  HEADER { // equivalent statement without template
    CAPACITANCE { PIN = Z; TABLE {20 40 80 160 } }
  SLEWRATE { PIN = A; TABLE { 0.4 0.8 } }
    }
  TABLE { 0.25 0.34 0.58 1.12 0.31 0.39 0.62 1.15 } }
}
GROUP

\[
group ::= \quad \text{GROUP} \text{identifier} \{ \text{values} \} \\
\quad | \quad \text{GROUP} \text{identifier} \{ \text{integer} : \text{integer} \}
\]

GROUP Timing \{ DELAY SLEWRATE \} // definition of group

Timing \{ UNIT = 1e-9; \} // usage of group

// equivalent statements without group
DELAY \{ UNIT = 1e-9; \}
SLEWRATE \{ UNIT = 1e-9; \}

GROUP BitWidth \{ 1 : 3 \} // definition of group
GROUP BitWidth \{ 1 2 3 \} // equivalent definition of group

VECTOR ( 01 Clk -> 01 Q[BitWidth] ){ // usage of group
        DELAY = 1.5 \{ FROM \{ PIN=Clk; \} TO \{ PIN=Q[BitWidth]; \} \}
}

// equivalent statements without group
VECTOR ( 01 Clk -> 01 Q[1] ){ DELAY = 1.5 \{ FROM \{ PIN=Clk; \} TO \{ PIN=Q[1]; \} \} }
VECTOR ( 01 Clk -> 01 Q[2] ){ DELAY = 1.5 \{ FROM \{ PIN=Clk; \} TO \{ PIN=Q[2]; \} \} }
VECTOR ( 01 Clk -> 01 Q[3] ){ DELAY = 1.5 \{ FROM \{ PIN=Clk; \} TO \{ PIN=Q[3]; \} \} }

March 15, 2000

ALF tutorial 42
### CLASS

class ::= 

    CLASS identifier { objects }

// definition of classes
CLASS def1 { foo = 1; }
CLASS def2 { bar = 2; }
CLASS def3 { foobar = 3; }

// usage of classes
// Note: an object may refer to more than one class
// but the class definitions may not contain contradictory statements
CELL bufA { CLASS { def1 } }
CELL bufB { CLASS { def2 def3 } }
CELL bufC { CLASS { def1 def2 } }

// equivalent statements without class
CELL bufA { foo = 1; }
CELL bufB { bar = 2; foobar = 3; }
CELL bufC { foo = 1; bar = 2; }
PROPERTY, ATTRIBUTE

• use PROPERTY for customized parameter-value assignments or parameter-multivalue assignments associated with an object
• use ATTRIBUTE for customized parameters associated with an object

property ::= 

\[
\text{PROPERTY} \{ \{ \text{name} = \text{value} ; \} \{ \text{name} \{ \text{values} \} \} \}\]

// example:
PROPERTY {
    my_1st_parameter = my_1st_value ;
    my_2nd_parameter = my_2nd_value ;
    my_3rd_parameter { my_3rd_value my_4th_value my_5th_value }
}

attribute ::= 

\[
\text{ATTRIBUTE} \{ \text{symbols} \}\]

// example:
ATTRIBUTE { my_1st_parameter my_2nd_parameter my_3rd_parameter }
ALF applications

- Design creation and modification
- Functional modeling
- Characterization
  - Timing
  - Power
  - Signal Integrity
- Interconnect modeling
- Hierarchical design
- High-level design planning
Design creation and modification

• The usage restriction of each library component for design creation and modification steps is controlled by the `restrict_class` statement inside a general `CLASS` statement
  – Certain cells are usable for general synthesis, others for test synthesis, others for clock tree synthesis, others for layout

• Definitions for equivalent library components within the scope of a particular design step are provided by the `swap_class` statement inside a `CELL` statement
  – A synthesis tool may swap certain logically equivalent cells
  – A layout tool may swap certain electrically equivalent cells

```plaintext
{ CLASS class_name {
    RESTRICT_CLASS { [ synthesis ] [ datapath ] [ scan ] [ clock ] [ layout ] }
}
{ CELL cell_name {
    SWAP_CLASS { class_name { class_name } }
}
}
Design creation and modification (cont.)

// Example:
CLASS any_buffer { RESTRICT_CLASS { synthesis } }
CLASS single_height_buffer { RESTRICT_CLASS { layout } }
CLASS double_height_buffer { RESTRICT_CLASS { layout } }
CELL buf1 { SWAP_CLASS { any_buffer single_height_buffer } }
CELL buf2 { SWAP_CLASS { any_buffer double_height_buffer } }
CELL buf3 { SWAP_CLASS { single_height_buffer } }
CELL buf4 { SWAP_CLASS { double_height_buffer } }

// Synthesis tool sees the following:
CELL buf1 { SWAP_CLASS { any_buffer } }
CELL buf2 { SWAP_CLASS { any_buffer } }
CELL buf3 { /* not usable */ }
CELL buf4 { /* not usable */ }
// Therefore the synthesis tool may swap buf1 with buf2

// Layout tool sees the following:
CELL buf1 { SWAP_CLASS { single_height_buffer } }
CELL buf2 { SWAP_CLASS { double_height_buffer } }
CELL buf3 { SWAP_CLASS { single_height_buffer } }
CELL buf4 { SWAP_CLASS { double_height_buffer } }
// Therefore the layout tool may swap buf1 with buf3 and buf2 with buf4
Functional modeling

• A canonical functional model of the cell is part of the cell specification
• Useful for characterization
• Useful for generating tool-specific views of the function (Synthesis, STA, DFT … )
• Useful for generating simulation models (Verilog, VHDL … )
Functional modeling (cont.)

- PIN specification is prerequisite for FUNCTION
- Only pins with **PINTYPE=digital** may be used as variables in the FUNCTION statements
- Pins with **DIRECTION=input|output** are primary input or output variables, respectively
- Pins with **DIRECTION=both** are bi-directional, i.e., both input and output
- Pins with **DIRECTION=none** can be used as internal variables
- Pins with **VIEW=functional|physical** appear in the Verilog/VHDL or in the DEF netlist, respectively. Appearance in a netlist is orthogonal to appearance in the FUNCTION.

```verbatim
PIN pin_name {
    VIEW = functional | physical | both (default) | none ;
    PINTYPE = digital (default) | analog | supply ;
    following definitions are for pins with PINTYPE = digital:
    DIRECTION = input | output | both | none ; (mandatory)
    SIGNALTYPE = data | clock | control | etc. ; (optional)
}
```
Example:

CELL my_cell {
    PIN VDD {
        PINTYPE = supply; SUPPLYTYPE = power; VIEW = physical;
    }
    PIN A {
        DIRECTION = input; PINTYPE = digital; SIGNALTYPE = data; VIEW = both;
    }
    PIN Z {
        DIRECTION = output; PINTYPE = digital; SIGNALTYPE = data; VIEW = both;
    }
    PIN VSS {
        PINTYPE = supply; SUPPLYTYPE = ground; VIEW = physical;
    }
    // put FUNCTION statement here
}

// instance of my_cell in functional netlist will contain pins A, B
// instance of my_cell in physical netlist will contain pins VDD, A, B, VSS
// A, B are used as variables in FUNCTION statement
FUNCTION {
    BEHAVIOR { behavior_description }
    [ STRUCTURE { structure_description } ]
    [ STATETABLE [ name ] { statetable_description } ]
}

- BEHAVIOR contains a canonical description of the function. Purpose is to have a golden reference of the function.
- STRUCTURE (optional) contains a structural description of the cell in form of a netlist.
- STATETABLE (optional) contains a complementary description of the function in statetable format. One or more statetables can be used. Purpose is to facilitate generation of table-based simulation models, e.g. Verilog UDPs.
Functional modeling (cont.)

behavior_description ::=  
{ combinational_statements }  
{ sequential_statements }  
{ primitive_instance_statements }  

combinational_statement ::=  
  variable_name = boolean_expression ;  

sequential_statement ::=  
  @ ( control_expression ) { combinational_statements }  
{ : ( control_expression ) { combinational_statements } }  

control_expression ::=  
  boolean_expression  
|  vector_expression  

primitive_instance_statement ::=  
  primitive_name { combinational_statements }
Functional modeling (cont.)

- Combinational logic is modeled with `combinational_statements`.
- Level-sensitive sequential logic is modeled with `sequential_statements` containing only `boolean_expressions`.
- Edge-sensitive sequential logic is modeled with `sequential_statements` containing at least one `vector_expression`.
- The symbols `@`, `:` in `sequential_statements` mean “if”, “else-if”.
- All `sequential_statements` starting with `@` are evaluated concurrently.
- The priority of `control_expression` is in the order of occurrence in the `sequential_statement`.
- The `combinational_statements` activated by `control_expression` are evaluated concurrently.
- Any logic can be modeled with `primitive_instance_statements`, reusing predefined `FUNCTION` statements within a `PRIMITIVE`.
- The `FUNCTION` statement within the `PRIMITIVE` must contain `combinational_statements` or `sequential_statements`.
Functional modeling (cont.)

- Graphical illustration of `combinational_statements`

```
Primary inputs

variable
variable
variable
variable

boolean_expression

Internal variable

variable

Primary outputs

variable

variable
```

- Primary inputs and outputs must be declared as PINs
- Internal scalar variables need not be declared as PINs
- All 1-or 2-dimensional variables must be declared as PINs
Functional modeling (cont.)

- Graphical illustration of `sequential_statements`

- Feedback from `storage outputs` to `data inputs` is only allowed for edge-sensitivity, i.e., when the `control_expression` is a `vector_expression`

- Feedback from `storage outputs` to `control inputs` is only valid for modeling special functionality, e.g. oscillators
Functional modeling example: NAND gate

CELL my_nand {
    PIN A { DIRECTION = input; }
    PIN B { DIRECTION = input; }
    PIN Z { DIRECTION = output; }
    FUNCTION {
        BEHAVIOR { Z = ! ( A & B ); }
        /* alternative description using a primitive_instance_statement
        BEHAVIOR { predefined_nand { out = Z; in[0] = A; in[1] = B; } } */
        STATETABLE { // optional
            A B : Z ; // statetable_header_expression
            0 0 : 1 ; // statetable_body_expression
            0 1 : 1 ; // statetable_body_expression
            1 0 : 1 ; // statetable_body_expression
            1 1 : 0 ; // statetable_body_expression
        }
    }
}

PRIMITIVE predefined_nand {
    PIN out { DIRECTION = output; }
    PIN[0:<num_bits>] in { DIRECTION = input; }
    FUNCTION {
        BEHAVIOR { out = ~ & in; }
    }
}
Functional modeling example: Flipflop

CELL my_ff {
    PIN D { DIRECTION = input; }
    PIN CLK { DIRECTION = input; }
    PIN RST { DIRECTION = input; }
    PIN Q { DIRECTION = output; }
    PIN QN { DIRECTION = output; }
    FUNCTION {
        BEHAVIOR {
            @ ( !RST ) { Q = 'b0; QN = 'b1; }
            : ( 01 CLK ) { Q = D; QN = !D; }
        }
        /* alternative description with concurrent statements
        @ ( (01 CLK) & RST ) { Q = D; QN = !D; }
        @ ( !RST ) { Q = 'b0; QN = 'b1; }
        */
    }
    STATETABLE { // optional
        RST CLK D Q : Q QN ;
        0 ? ? ? : 0 1 ;
        1 01 0 ? : 0 1 ;
        1 01 1 ? : 1 0 ;
        1 ?0 ? 0 : 0 1 ;
        1 ?0 ? 1 : 1 0 ;
        1 1? ? 0 : 0 1 ;
        1 1? ? 1 : 1 0 ;
    }
}
Functional modeling example: 2 port memory

CELL my_2port_memory {
    CLASS port_A; // read port
    CLASS port_B; // write port
    PIN[1:8] Dout { DIRECTION = output; SIGNALTYPE = data; SIGNAL_CLASS = port_A; }
    PIN Renb { DIRECTION = input; SIGNALTYPE = read_enable; SIGNAL_CLASS = port_A; }
    PIN[3:0] Raddr {
        DIRECTION = input; SIGNALTYPE = address; SIGNAL_CLASS = port_A;
    }
    PIN[1:8] Din { DIRECTION = input; SIGNALTYPE = data; SIGNAL_CLASS = port_B; }
    PIN Wenb { DIRECTION = input; SIGNALTYPE = write_enable; SIGNAL_CLASS = port_B; }
    PIN[3:0] Waddr {
        DIRECTION = input; SIGNALTYPE = address; SIGNAL_CLASS = port_B;
    }
    PIN[1:8] core[0:15] { DIRECTION = none; VIEW = none; }
    FUNCTION {
        BEHAVIOR {
            @(Wenb) { core[Waddr] = Din; }
            @(Renb) { Dout = core[Raddr]; }
        }
    }
}
Characterization

- **Input**
  - Specification of CELL, PINs, FUNCTION
  - Specification of characterization models and range
  - Specification of characterization VECTORS

- **Output**
  - Models with characterization data
Timing

- Timing characterization data
  - DELAY, RETAIN
  - SLEWRATE
  - SETUP, HOLD, RECOVERY, REMOVAL, SKEW
  - PULSEWIDTH
  - PERIOD, NOCHANGE

- Timing violations
Timing (cont.)

• Timing characterization data are in the context of a VECTOR
  – Characterization waveform is described by vector_expression
• Sense of measurement is defined in FROM, TO statements
• Definitions for THRESHOLD
  – represent a voltage reference point normalized to the signal voltage swing
  – may be included in local FROM, TO statements for the model or in global FROM, TO statements at CELL, SUBLIBRARY, or LIBRARY level
• Reference to PIN and EDGE_NUMBER
  – indicate the measurement points related to the vector_expression
  – appear in the context of the model
  – appear in the FROM, TO statements for measurements between different pins
  – must contain EDGE_NUMBER, if PIN appears more than once in vector_expression
Timing (cont.)

// timing measurement between two consecutive events on two pins
DELAY | RETAIN {
  FROM { PIN = pin_name ; [ THRESHOLD = number ; ] [ EDGE_NUMBER = number ; ] } 
  TO { PIN = pin_name ; [ THRESHOLD = number ; ] [ EDGE_NUMBER = number ; ] } 
}

// timing measurement for one event on one pin
SLEWRATE { PIN = pin_name ; [ EDGE_NUMBER = number ; ] 
  [ FROM { THRESHOLD = number ; } ] [ TO { THRESHOLD = number ; } ] }

// early and late timing measurements
EARLY {
  DELAY { /* fill in */ } 
  SLEWRATE { /* fill in */ } 
}
LATE {
  DELAY { /* fill in */ } 
  SLEWRATE { /* fill in */ } 
}
Timing (cont.)

// timing check between two consecutive events on two pins
SETUP | HOLD | RECOVERY | REMOVAL | SKEW { [ violation_statement ]
  FROM { PIN = pin_name ; [ THRESHOLD = number ; ] [ EDGE_NUMBER = number ; ] } 
  TO { PIN = pin_name ; [ THRESHOLD = number ; ] [ EDGE_NUMBER = number ; ] } 
}

// timing check between two consecutive events on one pin
PULSEWIDTH { [ violation_statement ] PIN = pin_name ;
  FROM { [ THRESHOLD = number ; ] [ EDGE_NUMBER = number ; ] } 
  TO { [ THRESHOLD = number ; ] [ EDGE_NUMBER = number ; ] } 
}

// timing check for entire vector
PERIOD | NOCHANGE { [ violation_statement ] [ PIN = pin_name ; ] }

violation_statement ::= 
  VIOLATION {
    MESSAGE_TYPE = information | warning | error ;
    MESSAGE = quoted_string ;
    BEHAVIOR { behavior_description }
  }
Timing example: delay

VECTOR (01 A -> 10 Z) {
  DELAY {
    FROM { PIN = A; THRESHOLD = 0.5; } TO { PIN = Z; THRESHOLD = 0.5; }
    HEADER {
      CAPACITANCE {
        PIN = Z; TABLE { 5 10 20 40 80 160 320 }
      }
      SLEWRATE {
        FROM { THRESHOLD = 0.1; } TO { THRESHOLD = 0.9; }
        PIN = A; TABLE { 0.2 0.4 0.8 }
      }
    }
    TABLE {
      0.5 0.7 1.2 2.3 4.5 9.0
      0.4 0.6 1.2 2.3 4.5 9.0
      0.3 0.6 1.2 2.3 4.5 9.0
    }
  }
}

March 15, 2000

ALF tutorial
VECTOR (10 D -> 01 CLK) {

  SETUP {
    FROM { PIN = D; THRESHOLD = 0.5; } TO { PIN = CLK; THRESHOLD = 0.5; }
    HEADER {
      SLEWRATE slew1 {
        FROM { THRESHOLD = 0.9; } TO { THRESHOLD = 0.1; }
        PIN = D; TABLE { 0.1 0.3 0.9 2.7 }
      }
      SLEWRATE slew2 {
        FROM { THRESHOLD = 0.1; } TO { THRESHOLD = 0.9; }
        PIN = CLK; TABLE { 0.2 0.4 0.8 }
      }
    }
    TABLE {
      0.4 0.4 0.6 1.1
      0.4 0.5 0.8 1.4
      0.5 0.6 1.2 1.9
    }
  }
}
VECTOR (01 CLK -> 10 D) {
    HOLD {
        FROM { PIN = CLK; THRESHOLD = 0.5; } TO { PIN = D; THRESHOLD = 0.5; }
        HEADER {
            SLEWRATE slew1 {
                FROM { THRESHOLD = 0.1; } TO { THRESHOLD = 0.9; }
                PIN = CLK; TABLE { 0.2 0.4 0.8 }
            }
            SLEWRATE slew2 {
                FROM { THRESHOLD = 0.9; } TO { THRESHOLD = 0.1; }
                PIN = D; TABLE { 0.1 0.3 0.9 2.7 }
            }
        }
        TABLE {
            0.4 0.4 0.5
            0.4 0.5 0.6
            0.6 0.8 1.2
            1.1 1.4 1.9
        }
    }
}
Timing example: combined setup & hold

VECTOR (01 D -> 01 CLK -> 10 D) {
  SETUP {
    FROM { PIN = D; THRESHOLD = 0.5; } TO { PIN = CLK; THRESHOLD = 0.5; }
    HEADER {
      SLEWRATE slew1 { FROM { THRESHOLD = 0.1; } TO { THRESHOLD = 0.9; }
        PIN = D; EDGE_NUMBER = 0; TABLE { /* data */ }
      }
      SLEWRATE slew2 { FROM { THRESHOLD = 0.1; } TO { THRESHOLD = 0.9; }
        PIN = CLK; EDGE_NUMBER = 0; TABLE { /* data */ }
      }
    }
  }
  TABLE { /* data */ }

  HOLD {
    FROM { PIN = CLK; THRESHOLD = 0.5; } TO { PIN = D; THRESHOLD = 0.5; }
    HEADER {
      SLEWRATE slew2 { FROM { THRESHOLD = 0.1; } TO { THRESHOLD = 0.9; }
        PIN = CLK; EDGE_NUMBER = 0; TABLE { /* data */ }
      }
      SLEWRATE slew3 { FROM { THRESHOLD = 0.9; } TO { THRESHOLD = 0.1; }
        PIN = D; EDGE_NUMBER = 1; TABLE { /* data */ }
      }
    }
    TABLE { /* data */ }
  }
}
Process-dependent timing modeling

Process can be used as index for table-based timing model

```
DELAY { /* FROM, TO */
  HEADER {
    CAPACITANCE { /* PIN, TABLE */ }
    SLEWRATE { /* PIN, TABLE */ }
    PROCESS { TABLE { bccom nom wccom } }
  }
  TABLE { bccom_numbers nom_numbers wccom_numbers }
}
```

Process index can be converted into coefficients for equation-based timing model

```
DELAY { /* FROM, TO */
  HEADER {
    DELAY nominal {
      HEADER {
        CAPACITANCE { /* PIN, TABLE */ }
        SLEWRATE { /* PIN, TABLE */ }
      } TABLE { numbers }
      PROCESS Kp {
        HEADER { bccom nom wccom } TABLE { -0.15 0.0 0.27 }
      }
    }
  }
  EQUATION { nominal * (1 + Kp) }
}
```
Power

• Power consumption data
  – ENERGY
  – POWER

• Power supply data
  – CURRENT
  – VOLTAGE

• Waveform descriptions
Power calculation

- Power characterization data is expressed as `POWER` or `ENERGY`.
- Measurement method is defined by `MEASUREMENT` statement.

```plaintext
// static power measurement
POWER {
    MEASUREMENT = static ;
}

// transient power measurement: $\text{TIME} = 1 / \text{FREQUENCY}$
POWER {
    MEASUREMENT = average | rms | peak ;
    TIME | FREQUENCY = number ;
}

// transient energy measurement: transient ENERGY = average POWER * TIME
ENERGY {
    [ MEASUREMENT = transient ; ]
    [ TIME | FREQUENCY = number ; ]
}
```
Power calculation (cont.)

• Power characterization data are in the context of a **VECTOR**
  – Characterization waveform is described by `vector_expression`
  – Usually the vectors for power are a superset of vectors for timing

• Dynamic interpretation of `vector_expression` in the context of simulation-based power analysis
  – All events at the PINs of the CELL are observed
  – A `boolean_expression` describes a detectable state of the pins
  – A `vector_expression` describes a detectable sequence of events
  – Static power consumption occurs while a particular `boolean_expression` matches the actual recorded state
  – Transient power consumption occurs while a particular `vector_expression` matches the actual recorded sequence of events

• Power consumption for all vectors adds up
  – Vectors need not be mutually exclusive
Power calculation (cont.)

Particularities for dynamic interpretation of vector_expression

- An edge_literal indicating “no event on operand” defines another event by exclusion
  
  (00 Z)  // event on another pin occurs while Z == 0
  
  (?- Z)  // event on another pin occurs while Z is constant

- A vector_expression without condition implies that all pins must be taken into account, not only those appearing in the vector_expression
  
  (01 A -> 10 Z)  // if there is a pin B, no event must occur on pin B

- A vector_expression with condition limits the scope of observation to the pins appearing in the vector_expression
  
  (01 A -> 01 Z) & B  // if there is a pin C, any event may occur on pin C

- An edge_literal containing * puts the operand in or out of scope, respectively
  
  (?* B)  // pin B is not observed from now on
  
  (*? C)  // pin C is observed from now on
  
  (1* B -> 10 A -> *1 C -> 10 Z)
  // B must be 1 before (01 A) is detected, C must be 1 after (01 A) is detected
CELL my_cell {
    /* my_cell has pins A, B, C, Z */
    VECTOR (01 A -> 01 C) { /* power data */ }
    VECTOR (01 B -> 00 Z) { /* power data */ }
    VECTOR ((01 A -> 01 Z) & B) { /* power data */ }
    VECTOR (1* B -> 10 A -> *1 C -> 10 Z) { /* power data */ }
}
/* simulation event report

<table>
<thead>
<tr>
<th>time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>60</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>70</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>80</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>90</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

// (01 A -> 01 C) detected at time 30
// (01 B -> 00 Z) detected at time 50
// ((01 A -> 01 Z) & B) detected at time 80
// (1* B -> 10 A -> *1 C -> 10 Z) detected at time 100
*/
Power supply data and waveforms

// static current, depending on supply voltage
CURRENT { PIN = supply_pin_name ;
  MEASUREMENT = static ;
  HEADER { VOLTAGE { PIN = supply_pin_name ; TABLE { numbers } } }
  TABLE { numbers }
}

// transient current, dependent on supply voltage, slewrate, load capacitance
CURRENT { PIN = supply_pin_name ;
  MEASUREMENT = average | rms | peak ; TIME | FREQUENCY = number ;
  HEADER {
    VOLTAGE { PIN = supply_pin_name ; TABLE { numbers } } }
    SLEWRATE { PIN = input_pin_name ; TABLE { numbers } } }
    CAPACITANCE { PIN = output_pin_name ; TABLE { numbers } } }
  TABLE { numbers }
}

// transient current waveform
CURRENT { PIN = supply_pin_name ;
  MEASUREMENT = transient | average | rms | peak ;
  HEADER { TIME { TABLE { numbers } } }
  TABLE { numbers }
}
Signal Integrity

• Crosstalk
  – NOISE_MARGIN
  – driver RESISTANCE

• Electromigration, Hot electron
  – limits for CURRENT
  – limits for FREQUENCY
  – FLUENCE

• Output buffer characteristics
  – I/V characteristics
  – parasitic INDUCTANCE
Crosstalk

• Noise margin is a measure of signal voltage tolerance
  – globally for the library or locally on cell input pins
  – also possible on vector for dynamic noise analysis
  – simple number or dependent on process, temperature etc.

• Mathematical definition for ALF:

\[
\text{noise} = \frac{|\text{actual voltage} - \text{nominal voltage}|}{\text{voltage swing}}
\]

- normalized nominal signal voltage (high) = 1
- normalized nominal signal voltage (low) = 0

March 15, 2000

ALF tutorial
LIBRARY my_library {
    NOISE_MARGIN { HIGH = number; LOW = number; }
    /* other data */
    CELL my_flipflop {
        PIN D { DIRECTION = input; SIGNALTYPE = data; }
        PIN CLK { DIRECTION = input; SIGNALTYPE = clock;
        NOISE_MARGIN { HIGH = number; LOW = number; }
        }
        PIN RST { DIRECTION = input; SIGNALTYPE = clear;
        NOISE_MARGIN { HIGH = number; LOW = number; }
        }
        PIN Q { DIRECTION = output; SIGNALTYPE = data; }
        /* other data */
        VECTOR ( 01 CLK && !D && !Q) {
        NOISE_MARGIN = number { PIN = D; }
        }
        VECTOR ( 01 CLK && D && Q) {
        NOISE_MARGIN = number { PIN = D; }
        }
    }
}
// CLK and RST are always sensitive to noise, since it can trigger a malfunction
// D is only sensitive to noise during rising edge of CLK
// global noise margin applies per default
• Driver resistance is a model of Voltage/Current characteristics
  – used for interconnect delay and noise analysis on-chip and off-chip
  – measured on cell output pins, path-dependent and state-dependent
  – always in context of vector, to be distinguished from parasitic resistance
  – simple number or dependent on input slewrate, load capacitance etc.
CELL my_inv {
    PIN A { DIRECTION = input; }
    PIN Z { DIRECTION = output; }
    // transient driver resistance
    VECTOR ( 01 A -> 10 Z) {
        RESISTANCE { PIN = Z; }  
        HEADER { SLEWRATE { PIN = A; TABLE { 0.4 0.8 1.6 } } TABLE { 812.6 815.8 820.7 } }
    }  
    VECTOR ( 10 A -> 01 Z) {
        RESISTANCE { PIN = Z; }  
        HEADER { SLEWRATE { PIN = A; TABLE { 0.4 0.8 1.6 } } TABLE { 1601.5 1610.2 1633.6 } }
    }  
    // static driver resistance
    VECTOR ( !Z ) {
        RESISTANCE = 825.3 { PIN = Z; }
    }
    // static driver resistance
    VECTOR ( Z ) {
        RESISTANCE = 1601.9 { PIN = Z; }
    }
}
Electromigration

- Electromigration shortens the lifetime of a circuit by inflicting permanent damage due to excessive current density
  - Power supply wires: DC and AC currents
  - Signal wires: AC currents only, wire self heat
  - Damage can also occur on wires and contacts inside cells
- Currents through interconnect wires or external cell pins can be observed by measurement, simulation, calculation
  - Limits for observable currents can be defined as arithmetic_models
- Internal cell currents can not be observed directly
- However, they depend on observable quantities
  - Current depends on slewrate, load capacitance, switching frequency
  - Limits for observable quantities can be defined as arithmetic_models
Electromigration (cont.)

Example:
- Current limits for a wire segment on a particular metal layer
- Limits depend on wire width, characterization frequency and lifetime
- AC limits (average, rms, peak) and DC limits (static) are provided

LIBRARY my_library {
  LAYER metal1 {
    LIMIT {
      CURRENT max_avg { measurement = average; 
        MAX {
          HEADER {
            WIDTH { TABLE { numbers } } 
            FREQUENCY { TABLE { numbers } } 
            TIME { TABLE { numbers } } 
          }
          TABLE {numbers }
        }
      }
      CURRENT max_rms { measurement = rms; 
        MAX { // similar model as for max_avg 
      }
      CURRENT max_rms { measurement = peak; 
        MAX { // similar model as for max_avg 
      }
      CURRENT max_static { measurement = static; 
        MAX { // similar model as for max_avg 
      }
    }
  }
}
}
Electromigration (cont.)

Example:
- max. current limit for a pin of a cell
- max. frequency limit for a vector, exercising a particular current path inside the cell

```plaintext
CELL my_cell {
    PIN VDD {
        LIMIT { CURRENT { measurement = average | rms | peak | static ;
            MAX { /* HEADER, EQUATION or TABLE */ }
        }
    }
    PIN VSS { /* put another current limit, if necessary */ }
    PIN A  { /* put another current limit, if necessary */ }
    PIN B  { /* put another current limit, if necessary */ }
    PIN Z  { /* put another current limit, if necessary */ }
    VECTOR ( 01 A -> 10 Z ) {
        LIMIT { FREQUENCY { MAX {
            HEADER {
                CAPACITANCE { PIN = Z; TABLE { numbers } }
                SLEWRATE { PIN = A; TABLE { numbers } }
            }
            TABLE { numbers }
        }
    }
    VECTOR ( 01 B -> 10 Z ) { /* put another frequency limit, if necessary */ }
    VECTOR ( 10 A -> 01 Z ) { /* put another frequency limit, if necessary */ }
    VECTOR ( 10 B -> 01 Z ) { /* put another frequency limit, if necessary */ }
}
```
Hot electron effect

- Hot electron effect degrades performance of a circuit by trapping electrons in gate oxide due to excessive electrical field.
- A direct measure of hot electron effect is fluence, i.e., amount of accumulated charge per gate oxide area.
  - Similarity between fluence and energy:
    - Both are accumulative, path-and state-dependent.
    - Both can be characterized for vectors as arithmetic_models dependent on input slewrate and load capacitance etc.
  - Limits for fluence can be given for a cell, similar to limits for current:
    - Can be characterized as single number or as arithmetic_models dependent on lifetime.
- An indirect measure is frequency, in the same way as for electromigration:
  - Frequency limits for hot electron and electromigration can be combined.
Hot electron effect (cont.)

- Particularities of electromigration and hot electron effect
  - High electric field (hot electron damage) for fast input, slow output
  - High internal current (electromigration damage) for slow input, fast output
  - Hot electron effect occurs only on NMOS transistors
  - Electromigration occurs on any device, e.g. diffusion to metal contact of transistor
Interconnect modeling

- Statistical wireload models
- Physical parasitic models
- Interconnect delay models
- Interconnect crosstalk models
Statistical wireload model

WIRE my_wireload_model {
    CAPACITANCE { // estimated capacitance of the wire
        HEADER {
            // number of connections
            CONNECTIONS { TABLE { numbers } }
            // area of the block enclosing the wire
            AREA { TABLE { numbers } }
        } TABLE { numbers }
        }
    RESISTANCE { // estimated resistance of the wire
        /* HEADER, TABLE */
    }
    AREA { // estimated area of the wire itself
        /* HEADER, TABLE */
    }
}
CELL my_cell {
    AREA = number ; // area of the cell
}
// utilization = (total cell area + total wire area) / (area of the block)
Physical parasitic model

```
LAYER metal1 {
    // estimated grounded capacitance for a wire on a layer
    CAPACITANCE {
        HEADER {
            WIDTH { UNIT = 1e-6; }
            LENGTH { UNIT = 1e-6; }
        } EQUATION { 1.08*WIDTH*LENGTH }
    }
}

RULE parallel_lines {
    PATTERN line1 { LAYER = metal1; SHAPE = line; }
    PATTERN line2 { LAYER = metal1; SHAPE = line; }
    // estimated coupling capacitance between parallel lines on the same layer
    CAPACITANCE {
        BETWEEN { line1 line2 }
        HEADER {
            DISTANCE D { BETWEEN { line1 line2 } }
            LENGTH L1 { PATTERN = line1; }
            LENGTH L2 { PATTERN = line2; }
        } EQUATION { 0.27*(L1+L2)/D }
    }
}
```
Interconnect delay model

WIRE simple_interconnect_delay_model {
    NODE N1 = driver;
    NODE N2;
    NODE N3 = receiver;
    NODE N0 = ground;
    VECTOR ( ?! N1 -> ?! N2 ) { // models apply for both rise and fall
        DELAY { FROM { PIN = N1; } TO { PIN = N3; } }
        HEADER {
            RESISTANCE  R1 { NODE { N1 N2 } }
            CAPACITANCE C1 { NODE { N2 N0 } }
            RESISTANCE  R2 { NODE { N2 N3 } }
            CAPACITANCE C2 { NODE { N3 N0 } }
            } EQUATION { R1*(C1+C2) + R2*C2 } // Elmore delay
        }
    SLEWRATE { PIN = N3; 
        HEADER {
            DELAY { FROM { PIN = N1; } TO { PIN = N3; } TABLE { numbers } }
            SLEWRATE { PIN = N1; } 
            } TABLE { numbers } // slewrate degradation
        }
    }
}
Interconnect crosstalk model

WIRE interconnect_xtalk_delay_model {
    NODE N1 = driver; NODE N2 = receiver;   // aggressor
    NODE N3 = driver; NODE N4 = receiver;   // victim
    NODE N0 = ground;
    // aggressor is rising, victim is stable low
    VECTOR ( (01 N1 -> 01 N2) && !N3 && !N4 ) {
        // xtalk-induced noise voltage
        VOLTAGE { PIN = N4; MEASUREMENT = peak; CALCULATION = incremental;
                  HEADER {
                      SLEWRATE SA { PIN = N2; }
                      CAPACITANCE CC { NODE { N2 N4 } }
                      CAPACITANCE CV { NODE { N4 N0 } }
                      RESISTANCE  RV { NODE { N3 N4 } }
                  } EQUATION { 1.35*(1 - EXP(-SA/(RV*CV)))*RV*CC/SA }
        }
    }
    // aggressor is rising, victim is falling
    VECTOR ( 01 N1 -> 10 N3 -> 01 N2 -> 10 N4 ) {
        // xtalk-induced delay
        DELAY { FROM { PIN = N3; } TO { PIN = N4; } CALCULATION = incremental;
               HEADER {
                   SLEWRATE SA { PIN = N2; }
                   SLEWRATE SV { PIN = N3; }
                   CAPACITANCE CC { NODE { N2 N4 } }
                   CAPACITANCE CV { NODE { N4 N0 } }
                   RESISTANCE  RV { NODE { N3 N4 } }
               } EQUATION { 0.442*(1 - EXP(-SA/(RV*CV)))*RV*CC*SV/SA }
        }
    }
}
Hierarchical design

- Pins with multiple ports
- Boundary parasitics
- Structural models
- Timing models
Hierarchical design (cont.)

CELL my_cell {
    PIN X { /* pin data */ } PIN Y { /* pin data */ } PIN Z { /* pin data */ }
    /* cell data */
}

CELL my_hierarchical_block {
    PIN A1 { DIRECTION = input; 
        PORT P1 {VIEW=physical;} PORT P2 {VIEW=none;} PORT P3 {VIEW=none;}
    } 
    PIN A2 { DIRECTION = input; }
    PIN N1 { DIRECTION = none; PORT Q1 {VIEW=none;} PORT Q2 {VIEW=none;}}
    PIN Z1 { DIRECTION = input; 
        PORT R1 {VIEW=none;} PORT R2 {VIEW=physical;} PORT R3 {VIEW=physical;}
    }
    /* boundary parasitics */
    /* structural description */
    /* timing description */
}
Hierarchical design (cont.)

CELL my_hierarchical_block {
   /* pin description */
   WIRE boundary_parasitics {
      NODE GND = ground;
      RESISTANCE r1 = number { NODE { A1.P1 A1.P2 } }
      RESISTANCE r2 = number { NODE { A1.P1 A1.P3 } }
      RESISTANCE r3 = number { NODE { N1.Q1 N1.Q2 } }
      RESISTANCE r4 = number { NODE { Z1.R1 nn1 } }
      RESISTANCE r5 = number { NODE { nn1 Z1.R2 } }
      RESISTANCE r6 = number { NODE { nn1 Z1.R3 } }
      CAPACITANCE c1 = number { NODE { A1.P2 GND } }
      CAPACITANCE c2 = number { NODE { A1.P3 GND } }
      CAPACITANCE c3 = number { NODE { A2 GND } }
      CAPACITANCE c4 = number { NODE { nn1 GND } }
   }
   /* structural description */
   /* timing description */
}
Hierarchical design (cont.)

CELL my_hierarchical_block {
    /* pin description */
    /* boundary parasitics */
    FUNCTION { STRUCTURE {
        my_cell inst1 { X=A1.P2; Y=A2; Z=N1.Q1; }
        my_cell inst2 { X=A1.P3; Y=N1.Q1; Z=Z1.R1; }
    } }

    VECTOR (?! A1.P2 -> ?! N1.Q1) {
        DELAY { FROM { PIN=A1.P2; } TO { PIN=N1.Q1; } /* HEADER, TABLE */ }
    }

    VECTOR (?! A2 -> ?! N1.Q1) {
        DELAY { FROM { PIN=A2; } TO { PIN=N1.Q1; } /* HEADER, TABLE */ }
    }

    VECTOR (?! N1.Q1 -> N1.Q2 -> Z1.R1) {
        DELAY { FROM { PIN=N1.Q1; } TO { PIN=Z1.R1; } /* HEADER, TABLE */ }
    }

    VECTOR (?! A1.P3 -> ?! Z1.R1) {
        DELAY { FROM { PIN=A1.P3; } TO { PIN=Z1.R1; } /* HEADER, TABLE */ }
    }
}
High-level design planning

• Tool makes architectural trade-offs
  – area vs timing vs power

• Library supports abstract models
  – parameterized models for macrocells and logic building blocks
  – TEMPLATE construct is used
High-level design planning (cont.)

// Example: adder with fixed bitwidth
CELL my_8_bit_adder { AREA = 36.4;
    PIN [8:1] A { DIRECTION = input; }
    PIN [8:1] B { DIRECTION = input; }
    PIN [8:1] S { DIRECTION = output; }
        DELAY = 2.5 { FROM { PIN = A[1]; } TO { PIN = S[8]; }
        ENERGY = 139.7;
    }
}

// Template for adder with variable bitwidth
TEMPLATE my_N_bit_adder {
    CELL <cellname> { AREA = <cellarea> ;
        PIN [<N>:1] A { DIRECTION = input; }
        PIN [<N>:1] B { DIRECTION = input; }
        PIN [<N>:1] S { DIRECTION = output; }
            DELAY = <celldelay> { FROM { PIN = A[1]; } TO { PIN = S[<N>]; }
            ENERGY = <cellenergy> ;
        }
    }
}
High-level design planning (cont.)

// Static template instance creates adder with fixed bitwidth
// Every placeholder is replaced with a value

my_N_bit_adder {
    N = 8;
    cellname = my_8_bit_adder;
    cellarea = 36.4;
    celldelay = 2.5;
    cellenergy = 139.7;
}

// Dynamic template instance creates parameterized adder model
// Mathematical relationships between certain placeholders are defined

my_N_bit_adder = dynamic {
    cellname = N_bit_ripple_carry_adder;
    cellarea = N * 4.55;
    celldelay = N * 0.3125;
    cellenergy = N * 12.3 + N**2 * 5.1625;
}

// Tool can make tradeoff between N_bit_ripple_carry_adder and
// other dynamic template instances of my_N_bit_adder for a given N
Conclusion

• ALF covers the complete ASIC/SOC modeling space from RTL to Silicon
• Modeling concepts of vector_expression and arithmetic_model go a long way
• ALF is one of the most rapidly evolving OVI standards
• Please share the information from this tutorial freely with your colleagues
• You are very welcome to join the ALF and the related OLA workgroups
Further information

Please visit the ALF webpage at

www.eda.org/alf

Subscribe to the email reflector by sending email to

majordomo@eda.org

Contents:

subscribe ALF <your_email_address>