LIDEX Reference Manual [*]

José Eduardo Moreira
Wilson Vicente Ruggiero
Departamento de Engenharia de Eletricidade, Escola Politécnica da Universidade de São Paulo, Brasil
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign, USA

February 1990

CSRD Report No. 973

Abstract

This is the reference manual for the **LIDEX** hardware description language. The constructs of the language are syntactically defined using the YACC notation, and an explanation of the semantics is given in natural language. Current limitations of the present implementation are discussed.

[*]This work was supported in part by the Interamerican Development Bank/University of São Paulo project, and the U. S. Department of Energy under Grant No. DE–FG02–85ER25001. José E. Moreira is with Laboratório de Sistemas Integráveis, and Wilson V. Ruggiero is with Laboratório de Sistemas Digitais

Contents

1 Introduction

LIDEX is a powerful hardware description language centered on the RT level, but with capabilities for efficiently covering of the ISP and Switching levels. The currently available simulator generator implements the full language as described in this manual, allowing the simulation of complex systems.

The description of a digital systems spans over three different *design axis*: the *structural*, the *behavioral*, and the *morphological*. Some languages do allow description in all three axis, but most hardware description languages emphasize one or two of the axis [2, 5].

LIDEX is particularly strong at the behavioral axis, and very good at the structural axis. Its descriptions power in the morphological axis, however, is very limited, allowing the description of topological features, but not geometrical ones. The basic structural elements of a system described in **LIDEX** are the two type of carriers: *registers* (memory carriers) and *busses* (memoryless carriers) [1]. These basic elements are used for building up *modules*, and modules in turn can be used to build more complex modules, until we have the complete system. Besides its structure, each module also has an algorithmic description of its *behavior*. The syntax and semantics of **LIDEX** are based in a structured model of digital systems which is described in a related document [8], and the reader is suggested to read this document in order to better understand some characteristics of the language.

We will start our description of the **LIDEX** language by some very basic items used to write the descriptions of digital systems. We will the proceed in a top-down approach, describing simultaneously both the syntax and the semantics of the language.

2 Basics

We will describe the syntax of **LIDEX** using the notation of YACC [4]. This is more than reasonable, since the simulator for **LIDEX** was written with YACC, and the definitions presented here are the actual inputs to YACC, so there is no danger of disagreement between the language definition and the implementation.

A description in **LIDEX** is built up by atoms, or *tokens* as they are known in compiler terminology. The tokens for **LIDEX** are:

system	module	unit	structure
alias	connection	initialize	behavior
mod	var	onebus	zerobus
of	phase	cycle	const
for	to	do	begin
;	,	.	:
(	)	[	]
|	:=	=	− >
==	<>	>	<
>=	<=	+	−
or	xor	and	not
shr	shl	*	/
%	nop	stop	dump
end	identifier	integer	string

All the tokens, except for *identifier, integer* and *string* are represented in the description text as a string of characters, with exactly the same characters as in the tokens above; these strings are known as *keywords.*

An identifier is a string of no more than 15 characters, which may be letters and/or numbers, starting with a letter, and not equal to any keyword.

An integer is an unsigned integer number expressed either in decimal notation (only digits 0 to 9), or in hexadecimal notation (digits 0 to 9 and A to F) preceded by a '$' character.

An string is any sequence of characters enclosed by double quote marks. Example: "THIS IS A STRING".

LIDEX is a case insensitive language, which means that an upper case letter is exactly equivalent to its lower case correspondent. The effect is that of transforming all upper case letters into lower case before processing the description.

3 The Language

We will present the syntax and semantics of the language constructs in a “natural” order of definition. Literals are represented by a sequence of characters enclosed by double quote marks. In the end of the reference manual there is an alphabetic index of the language constructs, with the page numbers where the definition for each one is found. For examples of utilization of **LIDEX** the reader is recommended to read the document “LIDEX Tutorial” [7].

definition: system

syntax:

```
system : "system" identifier ";"
         systemconstclause
         unitmodulelist
         "end"
         "."
       ;

unitmodulelist : unitmodule
               | unitmodulelist unitmodule
               ;

unitmodule : unit
           | module
           ;
```

semantics: A digital system is built up of *modules*. A module is a structural member, and one or more modules can be used to build more complex modules, and so on until the last module in the description, which must be called “main” and which represents the full system. A digital system description may also contain functional *units*. A unit is not a structural member, but it can be viewed as a combinational circuit which transforms input values into an output value which is solely dependent on the input values. Units are used for describing behaviors of modules. The *systemconstclause* construct is used to define system-wise constants.

definition: system constants

syntax:

```
systemconstclause : "const"
                    systemconstlist
                    "end"
                    ";"
                  | empty
                  ;

systemconstlist : systemconst
                | systemconstlist "," systemconst
                ;

systemconst : identifier
              "."
              identifier
              "="
              integer
            | identifier
              "="
              integer
            ;
```

semantics: In the system constant declaration clause, names are associated with integer values. Whenever a constant name is referenced, the reference is substituted for the associated integer. A name associated to an integer value can appear anywhere an integer can appear, inside the scope for the name. Two types of constants can be declared in the system constant declaration clause:

1. system-valid constant – declared as just one *identifier*, which has as its scope the text that goes from the declaration of the constant until the end of the system;
2. module-valid constant – declared as two identifiers separated by a dot, the first identifier must be the name of a module inside the system, and the second identifier is the name of a constant for this module (its scope is the whole module).

definition: module

syntax:

```
module : "module" identifier
         "(" connectparamlist ")"
         ";"
         constclause
         structureclause
         aliasclause
         connectionclause
         initializeclause
         behaviorclause
         "end"
         ";"
       ;
```

semantics: Each module has a name associated to it, *identifier*, and this name is used when referencing to the module. If structural elements of a module will be connected to elements of another module, then a list of these elements must appear between the parentheses right after the module name (*connectparamlist*). Besides this list, a module description is built up of a

- *constclause:* declaration of constants internal to the module;
- *structureclause:* declaration of the structural elements that compose the module;
- *aliasclause:* declaration of aliases for the structural elements of the module;
- *connectionclause:* declaration of connections between structural elements of the module;
- *initializeclause:* declaration of initial values for the structural elements of the module;
- *behaviorclause:* algorithmic description of the behavior of the module.

definition: connection parameters

syntax:

```
connectparamlist : connectparam
                 | connectparamlist "," connectparam
                 | empty
                 ;

connectparam : identifier
             ;
```

semantics: The names of all elements of this module that will be used for connection with elements of other modules must be declared. The order in which the elements are listed is not important.

definition: constant declaration clause

syntax:

```
constclause : "const"
               constdecllist
               "end"
               ";"
            | empty
            ;

constdecllist : constdecl
              | constdecllist
                ","
                constdecl
              ;

constdecl : identifier
            "="
            integer
          ;
```

semantics: In the constant declaration clause of a module, names are associated with integer values. Whenever a constant name is referenced, the reference is substituted for the associated integer. A name associated to an integer value can appear anywhere an integer can appear, inside the scope for the name, which ranges from immediately after its declaration until the end of the module. The constant declaration clause is optional (it may be empty).

definition: structure declaration clause

syntax:

```
structureclause : "structure"
                  structure
                  "end"
                  ";"
                | empty
                ;

structure : modstructure
            varstructure
            zerobusstructure
            onebusstructure
          ;

modstructure : "mod" modstrelemlist ";"
             | empty
             ;

varstructure : "var" varstrelemlist ";"
             | empty
             ;

zerobusstructure : "zerobus" zerobusstrelemlist ";"
                 | empty
                 ;

onebusstructure : "onebus" onebusstrelemlist ";"
                | empty
                ;
```

semantics: The structural elements that compose a module can be of four different types (notice there is a proper order of declaration):

1. type *mod:* other modules, previously (in the lexicographic order of the description) declared;
2. type *var:* memory carriers, *i.e.* registers;

3. type *zerobus:* memoryless carriers, *i.e.* busses, which have a default value of zero in their lines;

4. type *onebus:* memoryless carriers, *i.e.* busses, which have a default value of one in their lines.

Each element must have a name different from the name of any other element of this module, from any system-wise or module-wise constant, and also different from any unit of the description (this is to ensure that a name appearing in an expression can be immediately identified in its meaning).

definition: module-type elements declaration

syntax:

```
modstrelemlist : modstrelem
               | modstrelemlist "," modstrelem
               ;

modstrelem : identifier
             arrayclause
             "of" identifier
           ;
```

semantics: The list of module-type structural elements is made of individual declarations in which the name of the element (first *identifier*) is declared, then the array range, if the element is of type array, is specified (*arrayclause*) and then the module which is the basic type of this element is declared (*identifier* following "of"). This (basic type) module must be previously (in the lexicographic order of the description) declared.

definition: register-type element declaration

syntax:

```
varstrelemlist : varstrelem
               | varstrelemlist "," varstrelem
               ;

varstrelem : identifier
             arrayclause
             bitfield
           ;
```

semantics: The list of register-type structural elements is made of individual declarations in which the name of the element (*identifier*) is declared, then the array range, if the element is of type array, is specified (*arrayclause*) and then the *bitfield*, which specifies the first and last bit of the register. The first and last bits of a register must obey the following restrictions:

1. the first bit must be less than or equal to the last bit;
2. both first and last bits must be between 0 and an implementation defined constant MAXBITS.

If no bits are specified in the *bitfield*, then both first and last bits are assumed to be 0.

definition: bus-type element declaration

syntax:

```
zerobusstrelemlist : zerobusstrelem
                   | zerobusstrelemlist "," zerobusstrelem
                   ;

zerobusstrelem : identifier
                 arrayclause
                 bitfield
               ;

onebusstrelemlist : onebusstrelem
                  | onebusstrelemlist "," onebusstrelem
                  ;

onebusstrelem : identifier
                arrayclause
                bitfield
              ;
```

semantics: The lists of bus-type structural elements is made of individual declarations in which the name of the element (*identifier*) is declared, then the array range, if the element is of type array, is specified (*arrayclause*) and then the *bitfield*, which specifies the first and last bit of the bus. The first and last bits of a bus must obey the following restrictions:

1. the first bit must be less than or equal to the last bit;
2. both first and last bits must be between 0 and an implementation defined constant MAXBITS.

If no bits are specified in the *bitfield*, then both first and last bits are assumed to be 0.

definition: array specification

syntax:

```
arrayclause : "("
              integer
              ":"
              integer
              ")"
            | empty
            ;
```

semantics: The first and second *integers* of an *arrayclause*, specify the lower and upper bounds of the array being declared, respectively. The lower and upper bounds of the array must obey the following rules:

1. the lower bound must be less than or equal to the upper bound;
2. both the lower and upper bound must be between 0 and an implementation defined constant MAXARRAY.

If the *arrayclause* is empty, then the element is not of array-type.

definition: bitfield specification

syntax:

```
bitfield : "|" integer "|"
         | "|" integer ":" integer "|"
         | empty
         ;
```

semantics: In the first case, when only one integer is specified, then both the first and last bits are assumed to be this same integer. When two integers are specified (second case), the first one is taken as the first bit, and the second one is taken as the last bit. A *bitfield* may be empty.

definition: alias clause

syntax:

```
aliasclause : "alias"
              alias
              "end"
              ";"
            | empty
            ;

alias : aliasexpr
      | alias "," aliasexpr
      ;

aliasexpr : aliasident
            "="
            aliasident
          ;

aliasident : identifier
             bitfield
           ;
```

semantics: The *alias clause* declares new (aliases) names that are synonyms to carriers or part of carriers. Aliases may be declared only for *simple* (non-array) registers and busses. The aliases names must be different from any element name, and the scope of an alias goes from the *end* of the *initialize* clause to the end of the module. Each alias is declared by an expression of the form

alias name = *real carrier name*

with a possible *bitfield* appended to each name. The *bitfield* appended to the alias name specifies the first and last bit of the alias. The *bitfield* appended to the real carrier name specifies the first and last bits of a part of the carrier, which corresponds to the first and last bits of the aliases. Whenever the alias name is used inside its legal scope, the reference to the alias is substituted for a corresponding reference to the real carrier. The bit-width for the alias and the original carrier (or part of it) must be the same.

definition: connection clause

syntax:

```
connectionclause : "connection"
                   connection
                   "end"
                   ";"
                 | empty
                 ;

connection : connectionexpr
           | connection "," connectionexpr
           ;

connectionexpr : compident "=" compident
               | "for" identifier ":=" integer "to" integer "do"
                 "begin"
                 connection
                 "end"
               ;
```

semantics: The *connectionclause* is a list of connection statements that connect two carriers (registers or busses). The carriers must be either non-array or a particular element of an array, and only carriers of the same type (var-var, onebus-onebus, zerobus-zerobus) can be connected, and only full carriers (not parts of a carrier) can be connected. Loops of the *for-type*, in which the control variable ranges from the first integer to the second integer in increments of 1, can be used to describe networks of connection.

definition: connection element

syntax:

```
compident : connidentifier
          |
            connidentifier
            "."
            connidentifier
          ;

connidentifier : identifier
               | identifier
                 "("
                 exprsim
                 ")"
               ;
```

semantics: *compident* specifies a carrier (register or bus) to be connected to another carrier. The carrier can be either an element of the module, in which case it can be specified by only one *connidentifier*, or it can be an element of a module-type element of the module, in which case two *connidentifiers* separated by a period are used, the first one to identify the module-type element and the other to identify the carrier inside this module-type element. If either the module-type element or the carrier are of array-type, then it is necessary to specify a particular element of the array through an *exprsim*, a expression that must evaluate to a valid index of the array.

definition: expression used in connections

syntax:

```
exprsim : unsgnexprsim
        | "+" unsgnexprsim
        | "-" unsgnexprsim
        ;

unsgnexprsim : term
             | unsgnexprsim "+" term
             | unsgnexprsim "-" term
             ;

term : factor
     | term "*" factor
     | term "/" factor
     | term "%" factor
     ;

factor : integer
       | identifier
       | "(" expresim ")"
       ;
```

semantics: The expression used to calculate an array index in a connection is very simple, and consists only of operators, integer and identifiers, and the identifiers must correspond to the control variables of whatever for-loop the expression is inside. The order of precedence, from highest to lowest, of the operators is the following:

1. first, what is inside parentheses is evaluated;
2. binary operators "*", "/" and "came next;
3. binary "+" and "-" are lower precedence operators;
4. unary "+" and "-" have the lowest precedence (be careful);

When there is more than one operator of equal precedence to be evaluated, the evaluation takes place from left to right.

definition: initialize clause

syntax:

```
initializeclause : "initialize" initialize "end" ";"
                 | empty
                 ;

initialize : initializationlist
           ;

initializationlist : initialization
                   | initializationlist "," initialization
                   ;

initialization : initvarident ":=" integer
               ;

initvarident : identifier
               initarrayrestr
             ;

initarrayrestr : "("
                 integer
                 ")"
               | empty
               ;
```

semantics: The *initialization clause* is used to set initial value to registers (and only registers) of a module, before the the start of the algorithm that describes the behavior of the module. If a register is of the array-type, the a particular element of the array must be specified with *initarrayrestr*.

definition: behavior clause

syntax:

```
behaviorclause : "behavior"
                 cyclespec
                 behavior
                 "end"
                 ";"
                | empty
                ;

cyclespec : "("
            "cycle"
            integer
            ")"
          | empty
          ;

behavior : commandlist
         ;

commandlist : command
            | commandlist ";" command
            ;
```

semantics: The behavior of a module is described through an algorithm composed by a list of commands. Each command corresponds to one step of the algorithm, and each step is executed either in one *machine cycle* or in one *clock cycle* (see the definition of *command*. The *cyclespec* is used to specify the number of clock cycles in one machine cycle of the module, if it is empty then the machine cycle is taken to be equal to the clock cycle (equivalent to declaring "(cycle 1)").

definition: command

syntax:

```
command : label
          ":"
          phasespec
          simplcommandlist
        ;

simplcommandlist : simplcommand
                 | simplcommandlist "," simplcommand
                 ;

label : identifier
      | empty
      ;

phasespec : "phase" integer ":" ":"
          | empty
          ;
```

semantics: Each *command*, which corresponds to one step in the algorithmic description of the behavior of a module, may be preceded by a *label*. The label is a name that identifies the command, and it is used for referencing the command in goto-type commands. Any command can have a label, so that labels may be added for clarity. No two commands can have the same label, and the scope of a label is the whole behavior clause in which it is declared.

A command is built up of one or more *simple* commands, which are all executed in parallel in the same step of the algorithm. The step may be executed in one machine cycle, if *phasespec* is empty, or in one clock cycle otherwise. The integer in *phasespec* has no use except to add clarity to the description.

If the algorithm executes a step in one clock cycle, and then moves to a step that is to be executed in one machine cycle, the execution of the later step does not start until the system reaches a clock cycle that corresponds to the beginning of a machine cycle for the module.

definition: simple command

syntax:

```
simplcommand : varident
               ":="
               expression
             | busident
               "="
               expression
             | gotocommand
             | stopcommand
             | nopcommand
             | dumpcommand
             | "("
               simplcommandlist
               ")"
               condition
             ;

condition : "[" expression "]"
          ;
```

semantics: A simple command can be either one of six elementary operations, or a list of simple commands enclosed in parentheses and followed by a *condition.* If the *expression* in the condition evaluates to TRUE (a value different of 0), then all the simple commands in the list are executed. If the expression in the condition evaluates to FALSE (a value equal to 0), then the the whole list is ignored. The six elementary operations that can be performed in a simple command are:

1. assignment of the value of an expression to a register-type carrier (*varident*);
2. assignment of the value of an expression to a bus-type carrier (*busident*);
3. goto (jump) to a command, identified through its label (*gotocommand*);
4. no-operation action, *i.e.* do nothing (*nopcommand*);
5. show the contents of a carrier, an array of carriers, or all the carrier in a module-type element of the module (*dumpcommand*);

6. stop the simulation (*stopcommand*).

The last two operations are not really used to describe the system, but rather to control the simulation and the output produced during it.

definition: stop command

syntax:

```
stopcommand : "stop"
            ;
```

semantics: When a stop command is executed, the simulation for the whole system terminates. The stop command is executed at the end of the machine or clock cycle (depending on the type of command).

definition: nop command

syntax:

```
nopcommand : "nop"
           ;
```

semantics: A nop command does nothing, it can be used to specify that during a cycle (machine or clock), nothing is to be executed.

definition: dump command

syntax:

```
dumpcommand : "dump" "(" identifier arrayrange ")"
             ;

arrayrange : "(" integer ":" integer ")"
           | "(" integer ")"
           | empty
           ;
```

semantics: The dump command, when executed, generates a listing of the values of the specified carriers. If the identifier is a module-type element of the module, then all carriers of the specified element (including those in module-type elements of the latter) are listed. If the identifier specifies an array-type element (either module-type, register-type or bus-type), then an *arrayrange* can be used to specify one particular element of the array (an integer between parentheses) or a range of elements (two integer separated by colon), in which case the first integer must be less than the second one). If the *arrayrange* is empty, then all the elements of the array are listed.

definition: goto command

syntax:

```
gotocommand : "->" label;
```

semantics: A goto command specifies the next step in the algorithm to be executed. When no goto command is executed in a step, the step immediately following (in the lexicographic order of the description) the current is the next to be executed. If the last step of the algorithm is executed and no goto command changes the flow of control, then the module just sits idle, doing nothing, it *does not* cause the end of the simulation.

definition: assignment receivers

syntax:

```
varident : assignident
         ;

busident : assignident
         ;

assignident : identifier
              arrayrestr
              bitfield
            ;

arrayrestr : "(" exprsimbeh ")"
           | empty
           ;
```

semantics: In an assignment operation, a carrier receives a value an expression evaluates to. Both *varident* and *busident* must specify a single register and bus-type element (respectively) of the module. In the case of array of carriers, *arrayrestr* must be used to specify a single element. A *bitfield* can be used to specify a part of a carrier as the receiver of the value.

Assignments to bus carriers, through the "=" operator are executed at the beginning of the cycle (machine or clock). If in a particular step of the algorithm, no value is assigned to a bus, it assumes its default value of either all bits in '1', for "onebus"-type, or all bits in '0', for "zerobus"-type.

Assignments to register carriers, through the ":=" operator are performed at the end of the cycle (machine or clock). Since registers are memory carriers, they keep their last value while a new one is not assigned.

definition: expression in the behavior clause

syntax:

```
expression : exprsimbeh
           | exprsimbeh "==" exprsimbeh
           | exprsimbeh "<>" exprsimbeh
           | exprsimbeh ">" exprsimbeh
           | exprsimbeh ">=" exprsimbeh
           | exprsimbeh "<" exprsimbeh
           | exprsimbeh "<=" exprsimbeh
           ;

exprsimbeh : unsgnexprsimbeh
           | "+" unsgnexprsimbeh
           | "-" unsgnexprsimbeh
           ;

unsgnexprsimbeh : termbeh
                | unsgnexprsimbeh "+" termbeh
                | unsgnexprsimbeh "-" termbeh
                | unsgnexprsimbeh "or" termbeh
                | unsgnexprsimbeh "xor" termbeh
                ;

termbeh : factorbeh
        | termbeh "*" factorbeh
        | termbeh "/" factorbeh
        | termbeh "%" factorbeh
        | termbeh "and" factorbeh
        | termbeh "shr" factorbeh
        | termbeh "shl" factorbeh
        ;

factorbeh : integer
          | ident
          | unitcall
          | "(" expression ")"
          | "not" factorbeh
          ;
```

semantics: Expression are evaluated to values according to precedence rules for the operators. As usual, what is inside parentheses is evaluated first. The order of precedence, from highest to lowest, of the operators is the following:

1. bitwise "not" operator;
2. arithmetic operators "*", "/" and "%" (remainder), bitwise operator "and", shift right operator "shr" and shift left operator "shl";
3. arithmetic operators "+" and "–", bitwise operators "or" and "xor";
4. unary "+" and "–" operators (be careful);
5. relational operators "==" (equal), "<>" (different), ">" (greater than), ">=" (greater than or equal), "<" (less than) and "<=" (less than or equal).

When there is more than one operator of equal precedence to be evaluated, the evaluation takes place from left to right.

An expression which is assigned to a bus-type carrier, or which conditions the assignment of a value to a bus-type carrier, must not involve values of bus-type carriers. The values of register-type carriers used in the evaluation of an expression are those values the registers have *during* the present cycle, before the assignments of new values are performed (these assignments occur all instantaneously at the end of the cycle).

definition: evaluation of units

syntax:

```
unitcall : unitident "(" expressionlist ")"
         ;

expressionlist : expression
               | expressionlist "," expression
               ;

unitident : identifier
          ;
```

semantics: When a unit evaluation is requested in an expression through the use of the unit name (specified by *unitident*), the output of the unit for the specified inputs (*expressionlist*), is calculated, and this value is used in the expression. The inputs passed to a unit are a list of expression that are evaluated to their corresponding values. The number of inputs must be equal to the number of input parameters specified in the unit declaration. A unit must be declared before (in the lexicographic order) any module that uses it.

definition: carrier values

syntax:

```
ident : identifier arrayrestr bitfield
      ;
```

semantics: Values of carriers are specified in an expression through the names of carriers (*identifier*), a particular element identification (*(arrayrestr*), for the case of arrays of carriers, and a *bitfield*, that specifies that only the value of a part of the carrier is to be used. If the *bitfield* is empty, then the whole carrier value is used. The identifier must be a name of a register or bus-type carrier of the module.

The model of digital systems on which **LIDEX** is based imposes some restrictions on the use of carrier values in expressions. A register-type carrier value can be used in any expression, but a bus-type carrier value must *not* be used in an expression that is assigned to a bus-type carrier, or in an expression in a condition that controls the execution of an assignment to a bus-type carrier.

When used in expression in a unit definition, *ident* must refer to a unit variable.

definition: functional unit

syntax:

```
unit : "unit" identifier "(" unitparamlist ")" bitfield ";"
       constclause
       unitvariables
       unitbehavior
       "end"
       ";"
     ;

unitparamlist : unitparam
              | unitparamlist "," unitparam
              ;

unitparam : identifier
            bitfield
          ;
```

semantics: The *units* of **LIDEX** are equivalent to the functions of Pascal, with the restriction that all parameters are passed as values. From the hardware point of view, it can be interpreted as a combinational circuit that has an output solely dependent on its inputs. A **LIDEX** unit is an entity that instantly operates in its inputs, generating as its output a new value.

Each *unit* has a name (the *identifier* following "unit"), through which is referenced in other modules and units. Only modules and units that follow (in the lexicographic order of description) the declared unit can use it. No form of recursion is possible.

The list of parameters *(unitparamlist)*, specifies the input parameters of the unit. They are declared just like the variables in *unitvariables*, and they can be considered just like variables that are initialized to the input values of the unit each time it is referenced. Each parameter and variable must have a different name.

A unit can have its own constant declaration clause *constclause*, in which constants local to the unit are declared.

The *unitbehavior* is an algorithm that computes the value of the output of the *unit* as a function of its inputs.

definition: unit variables declaration

syntax:

```
unitvariables : "var"
                unitvarlist
                "end"
                ";"
              | empty
              ;

unitvarlist : unitvar
            | unitvarlist "," unitvar
            ;

unitvar : identifier
          arrayclause
          bitfield
        ;
```

semantics: A *unit* can have local variables just like Pascal functions. The variables are activated each time the unit is referenced and destroyed when the unit is left. The variables can be of array-type (when a non-empty *arrayclause* is present) and can have *bitfield* specifiers just like the carriers of a module. Besides the explicit declared variables in *unitvar*, the input parameters can be considered variables, and a variable of the same name as the *unit* also exists. The output of the unit is the value of this later variable, when the unit algorithm is terminated.

definition: unit behavior

syntax:

```
unitbehavior : "behavior"
                unitcommandlist
                "end"
                ";"
             ;

unitcommandlist : unitcommand
                | unitcommandlist "," unitcommand
                ;

unitcommand : label
              ":"
              simplunitcommandlist
            ;

simplunitcommandlist : simplunitcommand
                     | simplunitcommandlist "," simplunitcommand
                     ;
```

semantics: The behavior, or algorithm, of a unit is described in a way much similar to the behavior of a module. It is an algorithm, formed by a sequence of steps that are executed in the lexicographic order, unless a jump (goto command) occurs. Each step (*unitcommand*) can have an associated label that identifies it, and it is composed by a list of simple commands, which are all executed concurrently and instantaneously. All these simple values operate on the *present* state of the variables, and compute the *future* values of them, that is, the values that will be used in the next step of the algorithm. A *unit* behavior may be thought as a module behavior in which the variables are register-type carriers and the machine cycle is infinitely fast.

definition: simple command of a unit

syntax:

```
simplunitcommand : unitvarident ":=" expression
                 | gotocommand
                 | nopcommand
                 | "("
                   simplunitcommandlist
                   ")"
                   condition
                 ;

unitvarident : assignident
             ;
```

semantics: The valid elementary operations for units are a subset of the elementary operations for modules. Again, a list of simple commands can be conditioned to execution by an expression. The execution takes place only if the expression evaluates to TRUE (a value different of 0).

The 3 simple operations for units are:

1. assignment of the value of an expression to a variable (*unitvar*);
2. goto (jump) to a command, identified through a label (*gotocommand*);
3. no-operation, *i. e.* do nothing, (*nopcommand*);

The expressions in a unit have the same form as in a module, but of course, they may refer only to unit variables.

4 Current Limitations

The currently implemented **LIDEX** environment has the following value for the implementation dependent constants:

- MAXBITS = 63;
- MAXARRAY = 32767.

The current implementation also restricts the lower bound of an array to be 0.

References

[1] Mario R. Barbacci. A Comparison of Register Transfer Languages for Describing Computers and Digital Systems. *IEEE Transactions on Computers*, C-24(2):137–150, February 1975.

[2] Reiner W. Hartenstein. *Hardware Description Languages*, volume 7 of *Advances in CAD for VLSI*, chapter 2. North-Holland, Amsterdam, 1987.

[3] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report.* Springer-Verlag, New York, 1975.

[4] Stephen C. Johnson. *UNIX Programmer's Manual Suplementary Documents*, chapter Yacc: Yet Another Compiler-Compiler. Regents of the University of California, Berkeley, CA, 1978.

[5] José Eduardo Moreira and Wilson Vicente Ruggiero. A Review of HDLs. Technical Report 971, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Urbana, IL, 1990.

[6] José Eduardo Moreira and Wilson Vicente Ruggiero. LIDEX Simulation Environment User's Manual. Technical Report 974, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Urbana, IL, 1990.

[7] José Eduardo Moreira and Wilson Vicente Ruggiero. LIDEX Tutorial. Technical Report 972, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Urbana, IL, 1990.

[8] José Eduardo Moreira and Wilson Vicente Ruggiero. The LIDEX Approach. Technical Report 970, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Urbana, IL, 1990.

Index