

ESPRIT 8370 ESIP PROJECT

LEVEL-0 VHDL SYNTHESIS SYNTAX AND SEMANTICS

Document prepared by: E. Villar - UC

Issued by: Microelectronics Group of the University of Cantabria

Date: 14th of December, 1995

Copyright 1994 Bull S.A., International Computers Ltd, Siemens Nixdorf Informationssysteme AG, Nederlandse Philips Bedrijven B.V., Racal-Redac Systems Ltd, Tecnología y Gestion de la Informacion S.A., Thomson-CSF.

This document and the information contained herein may be copied, used or disclosed in whole or in part provided that the author and the source are referenced. The copyright and the foregoing restriction on copying, use and disclosure extend to all media in which this information may be embodied, including magnetic storage, computer print-out, visual display, etc. The document is supplied without liability for errors or omissions.

ACKNOWLEDGMENTS

The author would like to thank to all the members of the European VHDL Synthesis Working Group (EVSWG) for their contributions to this proposal. They have provided very valuable comments and remarks during the different meetings of the EVSWG and through the open discussions and ballots which have taken place by electronic mail.

The author would like to thank specially to Wolfgang Ecker, chair of the EVSWG, and Manfred Selz, both from SIEMENS, for their support.

0. CONTENTS

ABSTRACT.....	4
1. SCOPE AND MOTIVATION.....	5
2. LEVEL-0 VHDL SYNTHESIS SYNTAX AND SEMANTICS	
2.1. Introduction.....	7
2.2. Level-0 VHDL synthesis syntax.....	8
2.2.1. Design library management.....	8
2.2.2. Entity declaration.....	8
2.2.3. Architecture body.....	9
2.2.4. Configuration declaration.....	9
2.2.5. Packages.....	10
2.2.6. Configuration specification and component declarations	10
2.2.7. Subprograms.....	11
2.2.8. Processes.....	12
2.2.9. Types and subtypes.....	14
2.2.10. Attributes.....	16
2.2.11. Signals, variables and constants.....	16
2.2.12. Operators and operands.....	17
2.2.13. Other language constructs.....	18
2.3. Level-0 VHDL synthesis semantics.....	21
2.3.1. Introduction.....	21
2.3.2. Combinational logic description.....	21
2.3.3. Arithmetic representation.....	22
2.3.4. Latch, clock and register inference.....	22

ABSTRACT. Synthesis from VHDL is one of the most important applications of the language today with high user demand. If VHDL is to be used for synthesis, some degree of discrepancy has to be allowed between simulation-before-synthesis, where the intended behavior has to be validated, and simulation-after-synthesis, where the actual behavior of the resultant hardware is verified. In the general case, this discrepancy between intended and actual behavior would be extremely dangerous and could lead to incorrect implementations. The formal description of VHDL does not protect it against this possibility, and, therefore, its use for synthesis has to be based on a clear synthesis methodology fully understood by the user.

All VHDL synthesis tools are based on such an explicit or implicit synthesis methodology. No standard synthesis methodology at any level has been proposed to date. As a consequence, each synthesis tool imposes its own synthesis methodology on the user. This fact has many disadvantages the main of them is the lack of portability.

Synthesis non-portability of a VHDL description refers both to the syntax as well as the synthesis semantics of the code. Syntactical differences between tools are closely related with the different VHDL subsets and packages they support. The problem is non-trivial due to the fact that a subset will always be necessary for synthesis. In fact, there is almost general agreement in that full VHDL will never be completely supported for synthesis. Thus, a consensus is necessary in order to define a standard synthesis subset. Nevertheless, as syntactical errors are detected by the synthesis tool being used, this is probably the least dangerous aspect when trying to move a description from one tool to another. Semantical differences are harder, as they may lead to different kinds of implementations. In fact, as synthesis tools interpret the code in different ways, it is not ensured that the resulting hardware will be equivalent. So, for instance, the same code can be implemented as combinational, asynchronous sequential or synchronous sequential depending on the tool.

In this standard proposal, a Level-0 VHDL synthesis syntax and semantics is proposed. Using it, the resulting VHDL descriptions are going to be portable to almost all the commercial synthesis tools currently in the market.

1. SCOPE AND MOTIVATION

One of the main problems when VHDL is used in synthesis applications is the lack of a standard VHDL-based synthesis syntax and semantics. VHDL has an unambiguous simulation semantics. In the synthesis domain the situation is rather the opposite, each synthesis tool imposes its own synthesis methodology on the user. This fact has many disadvantages which are more relevant in a context, like synthesis, which today represents one of the most important applications of the language with a high user demand.

The most important of these disadvantages is lack of portability. Each VHDL synthesis description refers to the specific synthesis tool for which it was created and can not be used to interchange design information if different synthesis tools are used. This means that in cooperative projects in which several departments of the same company or of different companies are involved, not only the language but also the synthesis tools have to be agreed on.

Reusability of existing VHDL synthesis descriptions is also refrained. Reusability represents a key issue to increase design productivity. This is particularly important in a market continuously more competitive and with shorter time-to-market requirements. Libraries of synthesizable descriptions are becoming increasingly important opening the way to new market opportunities. Reusability does not refer exclusively to the re-use of a certain description in a new design using the same design environment. It refers also to the possibility of re-using the VHDL description independently of the synthesis tool. Many companies are expending effort in developing reusable libraries composed of synthesizable VHDL descriptions. These descriptions follow a specific coding style imposed by a specific synthesis tool. If for any reason, the synthesis tool is changed or no longer supported, all the investment would be lost.

The Level-0 synthesis syntax and semantics represents a first step towards a standard hardware semantics for VHDL allowing its use in other hardware related tasks like formal verification, fault simulation, test generation, etc. which currently can not be performed in a standard way.

Front-end tools like high-level synthesis tools, FSM graphical interfaces, etc. which provide an added value in the synthesis process, have to support different output interfaces able to generate codes for different RT-level synthesis tools. These interfaces always represent a problem for the user. Mismatches are very frequent due to the fact that the vendor of the tool is obliged to support a subset defined by another company. Any new version of the RT-level synthesis tool requires a new version of the front-end tool which usually implies some delay. Any movement to another tool, both at the front-end or at the RT-level, is constrained by the availability of the corresponding interface.

Another related aspect is the lack of VHDL synthesis experts. It is possible to master a specific synthesis tool but very few people are able to obtain the best results from any tool.

The intention behind the definition of the Level-0 syntax and semantics is to overcome all the problems described above. In fact, the Level-0 syntax and semantics will constitute a standard subset of VHDL for synthesis applications which will allow description portability between tools as well as design reusability. Moreover, interface problems between front-end tools and RT-level synthesis tools will disappear.

The Level-0 syntax and semantics would represent a stable guide of how to use VHDL for synthesis defining a clear and unambiguous design methodology, in the same way full VHDL defines a clear and unambiguous simulation methodology. It represents the best way to obtain the best results in the shortest time. As a consequence, it is particularly recommended for novel engineers at design companies.

The Level-0 syntax and semantics is not going to limit synthesis tools performances or capabilities. It could be used as an interchange format between tools in such a way that one could take advantage of some particularities of the synthesis tool he/she is using and then translate the descriptions to the standard subset. These descriptions could be read by another tool which could perform any modification necessary to improve them to the synthesis algorithms this other tool is going to apply. This will be fit possible because the synthesis semantics of the Level-0 syntax and semantics will be clear and unambiguous. Translations could be done automatically.

Synthesis and simulation tools are expensive enough to make their acquisition and maintenance impossible for many companies, particularly small and medium enterprises, thus refraining the use of these new synthesis technologies. In many cases, these companies, when they have to deal with the design of an embedded system, have to subcontract the design of some subsystems, particularly ASICs, to design houses. Design specification becomes a problem because no formal specification is currently available. The Level-0 syntax and semantics can overcome this problem being used as a formal specification language able to describe the functionality of the circuit and with a clear design methodology which defines how this intended functionality can be modified during the synthesis process, thus serving as the contractual description of the circuit to be implemented. The level-0 syntax and semantics could be used as well as a specification language by the system design and the ASIC design departments inside the same companies or from different companies.

The Level-0 syntax and semantics represents the minimum syntactical and semantical requirements to any synthesis tool in the market. As both the language itself as well as the synthesis tools evolve in time, the standard synthesis syntax and semantics should evolve in time, leading periodically to new versions (Level-1, Level-2,..., etc.). In order to ensure stability, the new versions should appear only at fairly long intervals and should be upward compatible to previous versions.

Each new version could include as an annex language features not supported in the current version but to be supported in the next one. In this way, the Level-0 syntax and semantics would be used as an improvement guide for synthesis tool vendors reflecting the user's demands.

2. LEVEL-0 VHDL SYNTHESIS SYNTAX AND SEMANTICS

2.1. Introduction

The Level-0 synthesis syntax and semantics proposal is divided into two different parts. The first part describes the VHDL Level-0 synthesis syntax with a similar structure to the VHDL LRM. Each section firstly contains a brief list of the unsupported VHDL constructions. Then, an explicit description of the supported BNF syntax for those VHDL elements which are supported with certain restrictions is given. Next, the restrictions which can not be covered by the BNF syntax are described. Finally, the BNF syntax elements which are not supported are listed. Some constructions which are ignored for synthesis proposals are also mentioned. These constructions are allowed in the VHDL subset but their presence or absence does not affect the synthesis process (assert statements, for example). These constructions are enclosed in angle brackets (< >) in the BNF format.

The second part of the proposal corresponds to the Level-0 VHDL synthesis semantics. It presents how the different hardware elements (combinational, asynchronous sequential and synchronous sequential functions) which may contain a digital system can be described.

Of the 217 syntax elements described in Appendix A of the VHDL LRM:

- 56 syntax elements are not allowed.
- 53 are supported with restrictions (1 of them ignored).
- Another 2 syntax elements are ignored.
- 106 syntax elements are fully supported.

Despite the current limitations imposed by the state of the art in commercial synthesis technology, the Level-0 synthesis syntax and semantics allows the description of digital systems as an interconnection of combinational and sequential blocks and, therefore, allows the description of any digital system at the RT level (i.e. algorithmic finite state machines).

As a consequence, any synchronous VHDL description in any proprietary VHDL subset can be translated to the Level-0 VHDL synthesis syntax and semantics maintaining all the relevant information about the functionality of the design. This portable description will be accepted by any other synthesis tool giving functional equivalent results.

2.2. Level-0 VHDL synthesis syntax

2.2.1. Design library management

The Level-0 subset supports design library management. However, use clauses must have the following syntax:

```
use_clause ::=
    use Library_name.Package_name.all ;
```

The library IEEE and the package STD_LOGIC_1164 have to be supported. The Numeric Package contained in the IEEE 1076.3 Synthesis Package must also be supported.

2.2.2. Entity Declaration

Generics, linkage ports, and entity statement parts are not allowed. Syntax elements supported with restrictions are the following:

```
entity_declaration ::=
    entity identifier is
        entity_header
        entity_declarative_part
    end [entity_simple_name] ;
```

```
entity_header ::=
    [formal_port_clause]
```

```
formal_designator ::=
    port_name
    | parameter_name
```

```
mode ::= in | out | inout | buffer
```

```
entity_declarative_part ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | use_clause
```

The use of buffer and inout modes also has the following restrictions:

- They are not allowed if the architecture has processes.
- Buffer ports are synthesized as out ports.

Non-supported syntax elements:

entity_statement, entity_statement_part, generic_clause, generic_list,
generic_map_aspect.

2.2.3. Architecture body

The following are syntax elements supported with restrictions:

```

block_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | component_declaration
    | configuration_specification
    | use_clause

concurrent_statement ::=
    process_statement
    | concurrent_procedure_call
    | <concurrent_assertion_statement>
    | concurrent_signal_assignment_statement
    | component_instantiation_statement
    | generate_statement

```

2.2.4. Configuration Declaration

Configuration design units are not allowed. The following are syntax elements supported with restrictions:

```

primary_unit ::=
    entity_declaration
    | package_declaration

declaration ::=
    type_declaration
    | subtype_declaration
    | object_declaration
    | interface_declaration
    | component_declaration
    | entity_declaration
    | subprogram_declaration
    | package_declaration

```

Non-supported syntax elements:

block_configuration, block_specification, component_configuration,
configuration_declaration, configuration_declarative_item,
configuration_declarative_part, configuration_item, index_specification.

2.2.5. Packages

Declaring global signals in packages is not allowed. The following are syntax elements supported with restrictions:

```
package_body_declarative_item ::=  
    subprogram_declaration  
    | subprogram_body  
    | type_declaration  
    | subtype_declaration  
    | constant_declaration  
    | use_clause
```

```
package_declarative_item ::=  
    subprogram_declaration  
    | type_declaration  
    | subtype_declaration  
    | constant_declaration  
    | component_declaration  
    | use_clause
```

2.2.6. Configuration Specification and Component Declarations

Configuration specifications are supported with the following restrictions:

- Binding indication must be an entity aspect.
- Instantiation list must be the reserved word all.
- Entity aspect must be:

entity *Library_name*.*Entity_name* (*Architecture_name*).

The following are syntax elements supported with restriction:

```
binding_indication ::=  
    entity_aspect
```

```
instantiation_list ::=  
    all
```

```
entity_aspect ::=
    entity Library_name.Entity_name (Architecture_name)
```

The *Component_name* of the configuration specification must be the same as the *Entity_name*. Thus, the component name of the component declaration must coincide with an existing entity. Moreover, as generics are not supported, the supported syntax of the component declaration is the following:

```
component_declaration ::=
    component entity_name
        [local_port_clause]
    end component ;
```

2.2.7. Subprograms

2.2.7.1. Subprogram Declarations

Both procedures and functions are allowed. Nevertheless, user-defined operators or overloading of predefined operators are not supported. Thus, *operator_symbol* is not allowed as a subprogram name. The designator primitive must, therefore, have the following syntax:

```
designator ::= identifier
```

Specification of actual parameters by default is not supported. The interface object declarations must have the following syntax:

```
interface_constant_declaration ::=
    [constant] identifier_list : [in] subtype_indication

interface_signal_declaration ::=
    [signal] identifier_list : [mode] subtype_indication

interface_variable_declaration ::=
    [variable] identifier_list : [mode] subtype_indication
```

The following items are not allowed within subprograms:

- Specification of actual parameters by name for functions defined in the STD_LOGIC_1164 package.
- Unconstrained and variable-width arrays as subprogram parameters.

2.2.7.2. Subprogram Bodies

The declarative part is not supported. Thus, the syntax of the subprogram_body is:

```
subprogram_body ::=  
    subprogram_specification is  
    begin  
        subprogram_statement_part  
    end [designator] ;
```

The following items are not allowed:

- Read signals unless declared in the subprogram or in the parent architecture or subprogram.
- Recursive calls.
- Event attributes.
- Wait statements.

2.2.7.3. Resolution functions

No user-defined resolution functions are allowed. However, the "resolved" resolution function has to be supported.

Non-supported syntax elements:

subprogram_declarative_item, subprogram_declarative_part.

2.2.8. Processes

Processes must have a set of special characteristics, which can be summarized in four different kinds of processes.

a) Processes which contain a sensitivity list including all the signals which are read into the process and in which all signals and variables are assigned in all the conditional branches.

This kind of processes models pure combinational logic.

b) Processes which contain a sensitivity list including all the signals that are read into the process and whose variables are assigned in all the conditional branches of the processes.

This kind of processes can model a mixture of pure combinational logic and

asynchronous latches. Latches are inferred when signals are not assigned in a conditional branch.

c) Processes which have a wait statement of the form:

```
wait until clock = value and clock'event ;
```

This statement has to be the first statement of the process. In this kind of processes, only sequential signal assignments are allowed. Thus, the process must have the following syntax:

```
process
    process_declarative_part
begin
    wait until clock_name = value and clock_name'event ;
    { signal_assignment_statement }
end process ;
```

This kind of processes model a Moore synchronous sequential machine.

d) Processes which have a sensitivity list including the clock signal and optionally an asynchronous reset signal, and an "if" statement controlled by the event and edge of the clock signal. As before, only sequential signal assignments are allowed.

Thus, this kind of process has the following syntax:

```
process (clock_name [, reset_name])
    process_declarative_part
begin
    [ if (reset_name = value) then
        { signal_assignment_statement }
        elsif (clock_name = value and clock_name'event) then
            { signal_assignment_statement }
        end if ;
    end process ;
```

The syntax element supported with restrictions is the following:

```
process_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | use_clause
```

2.2.9. Types and subtypes

The types supported are Scalar and Composite types. Of the four classes of scalar types, Enumerated and Integer types are supported. The Composite types which are supported are one-dimensional Array types. Below, a detailed analysis of the restrictions on the declaration and use of the different VHDL types is given.

2.2.9.1. Type declaration

Incomplete type declaration is not allowed. The syntax is:

```
type_declaration ::= full_type_declaration
```

Neither access nor file types are allowed. For this reason, the primitive type_definition has the following syntax:

```
type_definition ::=  
    scalar_type_definition  
    | composite_type_definition
```

Scalar_type_definition has the following syntax:

```
scalar_type_definition ::=  
    enumeration_type_definition  
    | integer_type_definition
```

This is because neither floating point nor physical types are allowed. Record type definition is not allowed, so the syntax of composite type definition is the following:

```
composite_type_definition ::=  
    array_type_definition
```

2.2.9.2. Subtype declaration

User-defined resolution functions are not allowed. The supported syntax of subtype indication is:

```
subtype_indication ::=  
    type_mark [constraint]
```

Subtype declaration of enumerated types is not allowed.

2.2.9.3. IEEE 1164

The STD_LOGIC_1164 types are supported. The synthesis interpretation of the literal values of the STD_ULOGIC type has to follow the IEEE 1056.3 Synthesis Package.

2.2.9.4. Enumerated types

Enumerated types which contain character literals are not allowed, so the syntax of an enumeration_literal is the following:

enumeration_literal ::= identifier

The predefined types CHARACTER and SEVERITY_LEVEL are not supported.

2.2.9.5. Integers

Allowed integer ranges are the following:

0 to (2**n) -1
Or
- ((2**n) - 1) to (2**n) - 1

Or the reverse ranges (with downto direction).

2.2.9.6. Arrays

Constrained array type definition is not supported, so the syntax of array type definition is the following:

array_type_definition ::=
unconstrained_array_type_definition

Moreover, array types must be one-dimensional arrays, so the syntax of unconstrained_array_type_definition has to be:

unconstrained_array_type_definition ::=
array (index_subtype_definition)
of element_subtype_indication

Thus, the index_constraint and indexed_name primitives are supported in the following forms:

index_constraint ::= (discrete_range)

indexed_name ::= prefix (expression)

Arrays of integer elements are not allowed. The predefined type STRING is not supported either.

Non-supported syntax elements:

access_type_definition, base_unit_declaration, constrained_array_definition,
element_declaration, element_subtype_definition, file_declaration, file_logical_name,
file_type_definition, floating_type_definition, incomplete_type_declaration,
physical_type_definition, record_type_definition, secondary_unit_declaration.

2.2.10. Attributes

Neither attribute declaration nor attribute specification are allowed. The only predefined attributes supported are 'EVENT, 'LEFT, 'RIGHT, 'LOW, 'HIGH and 'LENGTH. Attribute names cannot, therefore, be used to specify ranges and the supported syntax of the range primitive is:

range ::=
simple_expression direction simple_expression

Non-supported syntax elements:

attribute_declaration, attribute_designator, attribute_name, attribute_specification,
entity_class, entity_designator, entity_name_list, entity_specification.

2.2.11. Signals, variables, and constants

2.2.11.1. Signal, Variable, and constant Declaration

Neither **register** nor **bus** signals are allowed. Initial values on signals and variables are allowed:

signal_declaration ::=
signal identifier_list : subtype_indication [< := expression>] ;

The syntax of variable declaration is:

variable_declaration ::=
variable identifier_list : subtype_indication [< := expression>] ;

Deferred constant declarations are not allowed. So the syntax of constant declaration is the following:

```
constant declaration ::=
    constant identifier_list : subtype_indication := expression ;
```

2.2.11.2. Signal assignment statements

Signal assignment statements must have a single waveform element. **Transport** and **after** clauses are ignored. The **Null** waveform element and **guarded** assignments are not allowed. Syntax elements supported with restrictions are the following:

```
signal_assignment_statement ::=
    target <= [transport]> waveform ;

waveform ::= waveform_element

waveform_element ::=
    value_expression < [after expression unit_name] >

options ::= < [transport] >
```

Disconnection specification is not allowed.

Non-supported syntax elements:

disconnection_specification, guarded_signal_specification, signal_kind, signal_list.

2.2.12. Operators and operands

2.2.12.1. Operators

The operators: "abs", "**", "/", "mod", and "rem" are not supported. Syntax elements supported with restrictions are the following:

```
factor ::=
    primary
    | not primary

miscellaneous_operator ::= not

multiplying_operator ::= *
```

The "*" operator is supported only if both operands are constants, or the second operand is a power of two.

2.2.12.2. Operands

Physical, Real, Based, String, and Null literals are not allowed as operands. Selected names are not supported, and the only attribute name allowed is "*signal_name*'EVENT". Thus, the modified syntax elements are:

```
abstract_literal ::= decimal_literal
```

```
decimal_literal ::= integer
```

```
literal ::=
    numeric_literal
    | enumeration_literal
    | string_literal
    | bit_string_literal
```

```
name ::=
    simple_name
    | operator_symbol
    | indexed_name
    | slice_name
```

```
numeric_literal ::=
    abstract_literal
```

```
range ::=
    simple_expression direction simple_expression
```

The range bounds of Slice Names must be constant.

Non-supported syntax elements:

base, based_integer, based_literal, exponent, physical_literal, selected_name, suffix.

2.2.13. Other language constructs

2.2.13.1. Wait statement

The **until** clause is the only supported clause. The syntax of the wait statement is:

```

wait_statement ::=
    wait condition_clause ;

condition_clause ::=
    until signal_name'event and signal_name = value

```

This statement must be the first statement of the process.

Non-supported syntax elements: sensitivity_clause, timeout_clause.

2.2.13.2. Loop statement

The only loops allowed are the **for** loops. The supported syntax of this statement is the following:

```

loop_statement ::=
    [ loop_label : ]
    iteration_scheme loop
        sequence_of_statements
    end loop [ loop_label ] ;

iteration_scheme ::=
    for loop_parameter_specification

```

2.2.13.3. Block statement

Blocks are not allowed.

Non-supported syntax elements:

block_declarative_part, block_header, block_statement.

2.2.13.4. Generate statements

Generate statements are allowed:

```

generate_statement ::=
    generate_label :
        generation_scheme generate
            { concurrent_statement }
        end generate [ generate_label ] ;

```

```
generation_scheme ::=
    for generate_parameter_specification
    | if condition
```

For a generate statement with an if generation scheme, the condition must be static.

2.2.13.5. Assertion Statements

Assertion statements are ignored.

2.2.13.6. Aggregates and Allocators

Aggregates and allocators are not supported. Syntax elements supported with restrictions are the following:

```
primary ::=
    name
    | literal
    | function_call
    | qualified_expression
    | type_conversion
    | (expression)

qualified_expression ::=
    type_mark'(expression)

target ::= name
```

Non-supported syntax elements:

aggregate, allocator, element_association.

2.2.13.7. Alias declarations

Alias declarations are not supported.

Non-supported syntax elements:

alias_declaration.

2.2.13.8. Textio Package

The "textio" package is not supported.

2.3. Level-0 VHDL synthesis semantics

2.3.1. Introduction

The Level-0 subset allows the description of digital systems as an interconnection of combinational and sequential blocks. In this section, the description of both types of blocks is given.

2.3.2. Combinational logic description

The Level-0 subset allows the description of combinational blocks by means of data-flow or behavioral descriptions.

2.3.2.1.- Data flow descriptions

Data flow descriptions are composed of concurrent signal assignment statements. The concurrent signal assignment statements produce combinational logic, unless the waveform depends on the target signal or the signal assignments originate a combinational loop. Both cases correspond to asynchronous circuits.

2.3.2.2. Behavioral descriptions

Behavioral descriptions are modeled by means of the statements contained in one or more processes. A process which models combinational logic has two main characteristics:

- a) A sensitivity list containing all the signals which are read into the process.
- b) All signals and variables are assigned in all the conditional branches of the process.

A mixture of pure combinational logic and asynchronous latches can be modeled. When not all the signals are assigned in all the conditional branches of the processes, in this case, the signals which are not assigned in all the conditional branches have to induce sequential behaviour. However, modeling such sequential behavior with variables is not allowed. Thus, variables must always be assigned in all the conditional branches. Moreover, the sensitivity list containing all the signals which are read in the process is required if there is no explicit EVENT condition within the process.

2.3.2.3. Modeling buses

Buses can be modeled by assigning the character literal 'Z'.

2.3.3. Arithmetic representation

As defined in the IEEE 1076.3 Synthesis Packages.

2.3.4. Latch, clock and register inference

2.3.4.1. Latch inference

Latches are inferred when a signal is not assigned in every conditional branch of a process, if the process has a sensitivity list containing all the signals which are read. However, it is not possible to model latches using variables.

Concurrent assignments of a signal with itself have to be implemented as an asynchronous circuit using a latch or by a combinational circuit with a feedback loop.

2.3.4.2. Clock inference

A clock is inferred through the following condition:

$$\text{clock} = \text{value and clock'event}$$

The clock signal must be of one of the following types: BIT, BOOLEAN, STD_LOGIC or STD_ULOGIC.

Use of the above condition as an operand in expressions is not allowed, nor the use of more than one of these conditions in a process.

When the synchronous behavior is modeled in the "THEN" branch of an "IF" statement whose control condition is of the above type, neither "ELSE" branches nor assignments outside the "IF" statement are allowed.

2.3.4.3. Register inference

The only way to model sequential synchronous behavior is by using processes. The statements which model the synchronous behavior must be executed when the event and edge conditions of the clock signal occur. As we have seen, the condition accepted in the Level-0 subset is the following:

clock = value **and** clock'event

There are two ways to describe synchronous behavior in a process with this condition:

1) Using a wait statement of the form:

wait until clock = value **and** clock'event ;

This statement must be the first statement of the process.

2) Using a sensitivity list and an if statement.

With this kind of process the synchronous behavior must be inside the "then" branch of an "if" statement whose control condition is the event and edge of the clock signal. In contrast, asynchronous behavior must be within the "then" branch of an "if" statement, whose control condition is the active level of the reset signal, and the synchronous behavior must be within the "elsif" branch.

The sensitivity list must contain, at least, the clock and optionally an asynchronous reset signal.