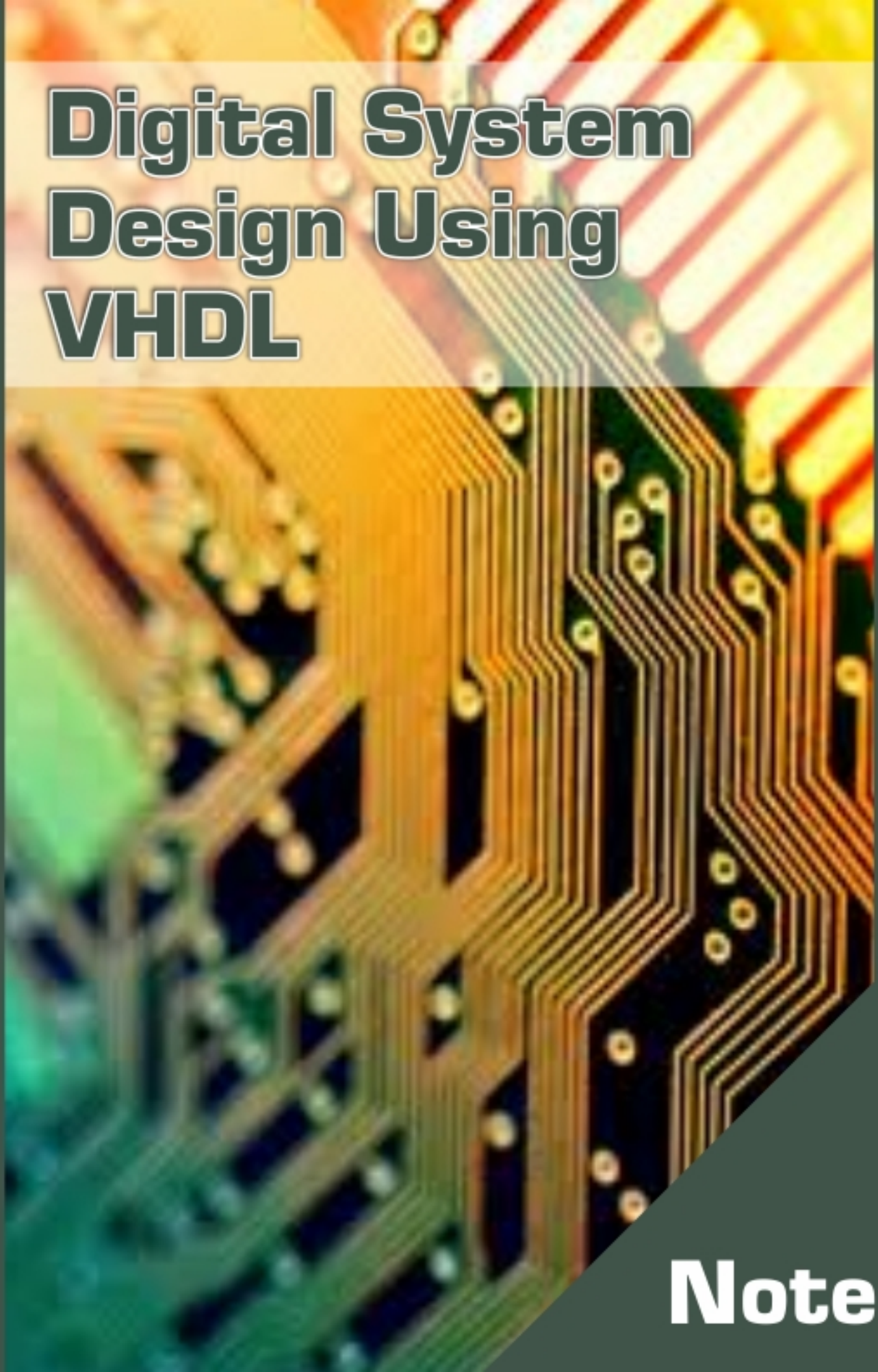
Digital System Design Using VHDL



Notes

Introduction to VHDL

VHDL is an acronym for VHSlC Hardware Description Language (VHSIC is an acronym for Very High Speed Integrated Circuits). It is a hardware description language that can be used to model a digital system at many levels of abstraction ranging from the algorithmic level to the gate level. The complexity of the digital system being modeled could vary from that of a simple gate to a complete digital electronic system, or anything in between. The digital system can also be described hierarchically. Timing can also be explicitly modeled in the same description.

VHDL resulted from work done in the ’70s and early ’80s by the U.S. Department of Defense. Its roots are in the ADA language, as will be seen by the overall structure of VHDL as well as other VHDL statements. VHDL usage has risen rapidly since its inception and is used by literally tens of thousands of engineers around the globe to create sophisticated electronic products. VHDL is a powerful language with numerous language constructs that are capable of describing very complex behavior.

VHDL Terms

VHDL is used to describe a model for a digital hardware device. This model specifies the external view of the device and one or more internal views. The internal view of the device specifies the functionality or structure, while the external view specifies the interface of the device through which it communicates with the other models in its environment

_ ***Entity***. All designs are expressed in terms of entities. An entity is the most basic building block in a design. The uppermost level of the design is the top-level entity. If the design is hierarchical, then the top-level description will have lower-level descriptions contained in it. These lower-level descriptions will be lower-level entities contained in the top-level entity description.

- **The entity describes the external interface to the model (specifies the inputs, outputs signals). It is the Hardware abstraction of a Digital system, but it does not provide any inner details.**

_***Architecture***. All entities that can be simulated have an architecture description. The architecture describes the behavior of the entity. A single entity can have multiple architectures. One architecture might be behavioral while another might be a structural description of the design.

- **The architecture describes the function/behavior of the model. Behavior of entity is defined by the relationship of the input to the output. It specifies the inner details of the circuit**

For examples relating to architecture & entity refer to presentation_1.

_ ***Configuration***. A configuration statement is used to bind a component instance to an entity-architecture pair. A configuration can be considered like a parts list for a design. It describes which behavior to use for each entity, much like a parts list describes which part to use for each part in the design.

_***Package***. A package is a collection of commonly used data types and subprograms used in a design. Think of a package as a toolbox that contains tools used to build designs.

_***Driver***. This is a source on a signal. If a signal is driven by two sources, then when both sources are active, the signal will have two drivers.

_ ***Bus.*** The term “bus” usually brings to mind a group of signals or a particular method of communication used in the design of hardware. In VHDL, a bus is a special kind of signal that may have its drivers turned off.

_***Attribute***. An attribute is data that are attached to VHDL objects or predefined data about VHDL objects. Examples are the current drive capability of a buffer or the maximum operating temperature of the device.

Generic. A generic is VHDL's term for a parameter that passes information to an entity. For instance, if an entity is a gate level model with a rise and a fall delay, values for the rise and fall delays could be passed into the entity with generics.

Process. A process is the basic unit of execution in VHDL. All operations that are performed in a simulation of a VHDL description are broken into single or multiple processes.

Architecture Body

The internal details of an entity are specified by an architecture body using any of the following modeling styles:

1. As a set of interconnected components (to represent structure),

2. As a set of concurrent assignment statements (to represent dataflow),

3. As a set of sequential assignment statements (to represent behavior),

4. Any combination of the above three.

Structural Style of Modeling

In the structural style of modeling, an entity is described as a set of interconnected components.

Dataflow Style of Modeling

In this modeling style, the flow of data through the entity is expressed primarily using concurrent signal assignment statements. The structure of the entity is not explicitly specified in this modeling style, but it can be implicitly deduced.

Behavioral Style of Modeling

In contrast to the styles of modeling described earlier, the behavioral style of modeling specifies the behavior of an entity as a set of statements that are executed sequentially in the specified order. This set of sequential statements, that are specified inside a process statement, do not explicitly specify the structure of the entity but merely specifies its functionality. A process statement is a concurrent statement that can appear within an architecture body.

Mixed Style of Modeling

It is possible to mix the three modeling styles that we have seen so far in a single architecture body. That is, within an architecture body, we could use component

instantiation statements (that represent structure), concurrent signal assignment statements (that represent dataflow), and process statements (that represent behavior).

Basic Language Elements

Identifiers

An identifier in VHDL is composed of a sequence of one or more characters. A legal character is an upper-case letter (A... Z), or a lower-case letter (a. .. z), or a digit (0 . . . 9) or the underscore (_) character. The first character in an identifier must be a letter and the last character may not be an underscore. Lower-case and upper-case letters are considered to be identical when used in an identifier; as an example. Count, COUNT, and CouNT, all refer to the same identifier. Also,-two underscore characters cannot appear consecutively. Some more examples of identifiers are

DRIVE_BUS SelectSignal RAM_Address

SET_CK_HIGH CONST32_59 r2d2

Comments in a description must be preceded by two consecutive hyphens (-); the comment extends to the end of the line. Comments can appear anywhere within a description. Examples are

```
--This is a comment; it ends at the end of this line.
--To continue a comment onto a second line, separate 2 hyphens
```

The language defines a set of reserved words; such as integer, real, etc.These words, also called *keywords,* have a specific meaning in the language, and therefore, cannot be used as identifiers.

Data Objects

A data object holds a value of a specified type. It is created by means of an object declaration. An example is

variable COUNT: INTEGER;

This results in the creation of a data object called COUNT which can hold integer values. The object COUNT is also declared to be of *variable* class.

Every data object belongs to one of the following **three classes:**

1. ***Constant****:* An object of constant class can hold a single value of a given type. This value is assigned to the object before simulation starts and the value cannot be changed during the course of the simulation.

2***. Variable****:* An object of variable class can also hold a single value of a given type. However in this case, different values can be assigned to the object at different times using a variable assignment statement.

3. ***Signal****:* An object belonging to the signal class has a past history of values, a current value, and a set of future values. Future values can be assigned to a signal object using a signal assignment statement.

Signal objects can be regarded as wires in a circuit while variable and constant objects are analogous to their counterparts in a high-level programming language like C or Pascal. Signal objects are typically used to model wires and flip-flops while variable and constant objects are typically used to model the behavior of the circuit.

An *object declaration is* used to declare an object, its type, and its class, and optionally assign it a value. Some examples of object declarations of various types and classes follow.

Constant Declarations

Examples of constant declarations are

```
constant RISE_TIME: TIME := 10ns;
constant BUS_WIDTH: INTEGER := 8:
```

Signal Declarations

Here are some examples of signal declarations.

```
signal CLOCK: BIT;
signal DATA_BUS: BIT_VECTOR(0 to 7);
```

Other Ways to Declare Objects

Not all objects in a VHDL description are created using object declarations. These other objects are declared as

1. ports of an entity. All ports are signal objects.

2. generics of an entity (discussed in unit 6). These are constant objects.

3. formal parameters of functions and procedures (discussed later). Function parameters are constant objects or signal objects while procedure parameters can belong to any object class,

4. a file declared by a file declaration (see file types in next section).

There are two other types of objects that are implicitly declared. These are the indices of a for. . . loop statement and the generate statement (discussed in unit 6)

Data Types

Every data object in VHDL can hold a value that belongs to a set of values. This set of values is specified by using a *type declaration.* A *type is* a name that has associated with it a set of values and a set of operations. Certain types, and operations that can be performed on objects of these types, are predefined in the language. For example, INTEGER is a predefined type with the set of values being integers in a specific range provided by the VHDL system. The minimum range that must be provided is -(2^31 - 1) through +(2^31 - 1). Some of the allowable and frequently used predefined operators are +, for addition, -, for subtraction, /, for division, and *, for multiplication. BOOLEAN is another predefined type that has the values FALSE and TRUE, and some of its predefined operators are and, or, nor, nand, and not. The declarations for the predefined types of the language are contained in package STANDARD; the operators for these types are predefined in the language.

The language also provides the facility to define new types by using type declarations and also to define a set of operations on these types by writing functions that return values of this new type. All the possible types that can exist in the language can be categorized into the following four major categories:

1. *Scalar* types: Values belonging to these types appear in a sequential order.

2. *Composite* types: These are composed of elements of a single type (an array type) or elements of different types (a record type).

3. *Access* types: These provide access to objects of a given type (via pointers).

4. *File* types: These provide access to objects that contain a sequence of values of a given type.

It is possible to derive restricted types, called subtypes, from other predefined or user-defined types.

Subtypes

A *subtype* is a type with a constraint. The constraint specifies the subset of values for the type. The type is called the *base type* of the subtype. An object is said to belong to a subtype if it is of the base type and if it satisfies the constraint. *Subtype declarations* are used to declare subtypes. An object can be declared to either belong to a type or to a subtype.

The set of operations belonging to a subtype is the same as that associated with its base type. Subtypes are useful for range checking and for imposing additional constraints on types.

Examples of subtypes are

subtype MY_INTEGER is INTEGER **range** 48 **to** 156 ;

type DIGIT is ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9') ;

subtype MIDDLE **is** DIGIT **range** '3' to '7' ;

Scalar Types

The values belonging to this type are ordered, that is, relational operators can be used on these values. For example, BIT is a scalar type and the expression '0' < '1' is valid and has *the* value TRUE. There are four different kinds of scalar types. These types are

1. enumeration,

2. integer,

3. physical,

4. floating point.

Integer types, floating point types, and physical types are classified as *numeric* types since the values associated with these types are numeric. Further, enumeration and integer types are called *discrete* types since these types have discrete values associated with them. Every value belonging to an enumeration type, integer type, or a physical type has a *position number* associated with it. This number is the position of the value in the ordered list of values belonging to that type.

Enumeration Types

An enumeration type declaration defines a type that has a set of user-defined values consisting of identifiers and character literals. Examples are

Type MVL **is** ('U','0','1','Z);

type MICRO_OP **is** (LOAD, STORE, ADD, SUB, MUL, DIV);

subtype ARITH_OP **is** MICRO_OP **range** ADD **to** DIV;

MVL is an enumeration type that has the set of ordered values, 'U', '0', '1', and 'Z'. ARITH_OP is a subtype of the base type MICRO_OP and has a range constraint specified to be from ADD to DIV, that is, the values ADD, SUB, MUL, and DIV belong to the subtype ARITH_OP.

Integer Types

An integer type defines a type whose set of values fall within a specified integer range. Examples of integer type declarations are

type INDEX **is range** 0 **to** 15;

type WORD_LENGTH **is range** 31 **downto** 0;

Values belonging to an integer type are called *integer literals.* Examples of integer literals are

56349 6E2 0 98_71_28

Literal 6E2 refers to the decimal value 6 * (10^2) = 600. The underscore (_) character can be used freely in writing integer literals and has no impact on the value of the literal;

98_71_28 is same as 987128.

INTEGER is the only predefined integer type of the language. The range of the INTEGER type is implementation dependent but must at least cover the range -(2^ 31 - 1) to +(2^31 - 1).

Floating Point Types

A floating point type has a set of values in a given range of real numbers. Examples of floating point type declarations are

type TTL_VOLTAGE **is range** -5.5 **to** -1.4;

type REAL_DATA **is range** 0.0 **to** 31.9;

Physical Types

A physical type contains values that represent measurement of some physical quantity, like time, length, voltage, and current. Values of this type are expressed as integer multiples of a base unit. An example of a physical type declaration is

```
type CURRENT is range 0 to 1 E9
  units
  nA; -- (base unit) nano-ampere
  uA = 1000 nA; -- micro-ampere
  mA = 1000 μA; --milli-ampere
  Amp = 1000 mA; -- ampere
  end units;
subtype FILTER_CURRENT is CURRENT range 10 μA to 5 mA;
```

CURRENT is defined to be a physical type that contains values from 0 nA to 10^9 nA. The base unit is a nano-ampere while all others are derived units.

Composite Types

A composite type represents a collection of values. There are two composite types: an array type and a record type. An array type represents a collection of values all belonging to a single type; on the other hand, a record type represents a collection of values that may belong to same or different types. An object belonging to a composite type, therefore, represents a collection of subobjects, one for each element of the composite type. An element of a composite type could have a value belonging to either a scalar type, a composite type, or an access type. For example, a composite type may be defined to represent an array of an array of records. This provides the capability of defining arbitrarily complex composite types.

Array Types

An object of an array type consists of elements that have the same type. Examples of array type declarations are

```
type ADDRESS_WORD is array (0 to 63) of BIT;
```

type DATA_WORD **is array** (7 **downto** 0) **of** MVL;

Elements of an array can be accessed by specifying the index values into the array. For example, ADDRESS_BUS(26) refers to the 27th element of ADDRESS_BUS array object.

Record Types

An object of a record type is composed of elements of same or different types. It is analogous to the record data type in Pascal and the struct declaration in C. An example of a record type declaration is

type PIN_TYPE **is range** 0 **to** 10;
type MODULE **is**
record
SIZE: INTEGER **range** 20 **to** 200;
CRITICAL_DLY: TIME;
NO_INPUTS: PIN_TYPE:
NO_OUTPUTS: PIN_TYPE;
end record;

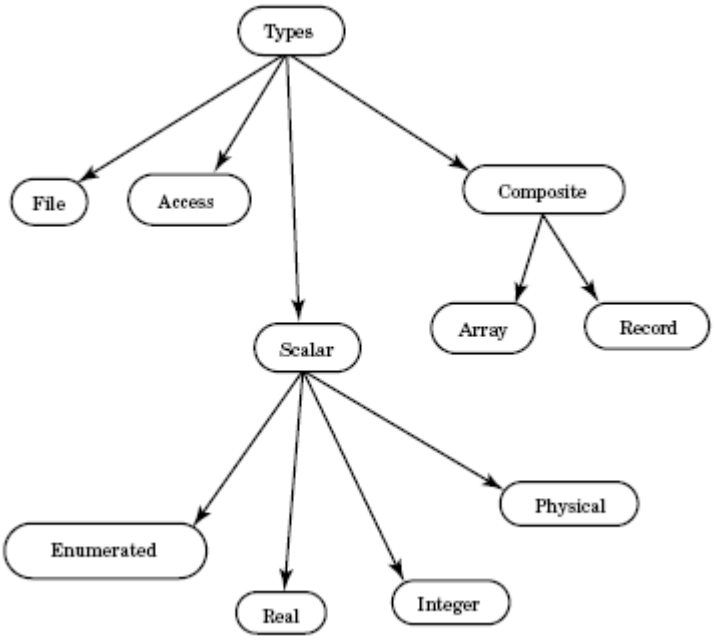
File Types

Objects of file types represent files in the host environment. They provide a mechanism by which a VHDL design communicates with the host environment. The syntax of a file type declaration is

type *file-type-name* **Is file of** *type-name,*

The *type-name* is the type of values contained in the file. Here are two examples.

type VECTORS **is file of** BIT_VECTOR;
type NAMES **is file of** STRING;



VHDL Data Types Diagram.

OPERATORS

The predefined operators in the language are classified into the following five categories:

Classification:

1. Logical operators
2. Relational operators
3. Shift operators
4. Addition operators
5. Multiplying op.
6. Miscellaneous op.

The operators have increasing precedence going from category (1) to (5). Operators in the same category have the same precedence and evaluation is done left to right. Parentheses may be used to override the left to right evaluation.

Logical Operators

The six logical operators are

and or nand nor xor not

These operators are defined for the predefined types BIT and BOOLEAN. They are also defined for one-dimensional arrays of BIT and BOOLEAN. During evaluation, bit values '0' and 1' are treated as FALSE and TRUE values of the BOOLEAN type, respectively. The result of a logical operation has the same type as its operands. The not operator is a unaiy logical operator and has the same precedence as that of miscellaneous operators.

Relational Operators

These are

= /= < <= > >=

The result types for all relational operations is always BOOLEAN. The = (equality) and the /= (inequality) operators are permitted on any type except file types. The remaining four relational operators are permitted on any scalar type (e.g., integer or enumerated types) or discrete array type (i.e., arrays in which element values belong to a discrete type). When operands are discrete array types, comparison is performed one element at a time from left to right. For example,

BIT_VECTOR('0', '1', '1') < BIT_VECTOR('1', '0', '1')

is true, since the first element in the first array aggregate is less than the first element in the second aggregate.

Adding Operators

These are

+ - &

The operands for the + (addition) and - (subtraction) operators must be of the same numeric type with the result being of the same numeric type. The addition and subtraction operators may also be used as unary operators, in which case, the operand and the result type must be the same. The operands for the & (concatenation) operator can be either a I-dirnensional array type or an element type. The result is always an array type. For example,

'0' & '1'

results in an array of characters "01".

'C' & 'A' & 'T'

results in the value "CAT".

"BA" & "LL"

creates an array of characters "BALL".

Multiplying Operators

These are

*** / mod rem**

The * (multiplication) and / (division) operators are predefined for both operands being of the same integer or floating point type. The result is also of the same type. The multiplication operator is also defined for the case when one of the operands is of a physical type and the second operand is of integer or real type. The result is of physical type.

For the division operator, division of an object of a physical type by either an integer or a real type object is allowed and the result type is of the physical type. Division of an object of a physical type by another object of the same physical type is also defined and it yields an integer value as a result.

The rem (remainder) and mod (modulus) operators operate on operands of integer types and the result is also of the same type. The result of a rem operation has the sign of its first operand and is defined as

A **rem** B = A - (A / B) * B

The result of a mod operator has the sign of the second operand and is defined as

A **mod** B = A – B * N -For some integer N.

Miscellaneous Operators

The miscellaneous operators are

abs **

The **abs** (absolute) operator is defined for any numeric type.

The ****** (exponentiation) operator is defined for the left operand to be of integer or floating point type and the right operand (i.e., the exponent) to be of integer type only.

The not logical operator has the same precedence as the above two operators.

Sequential statements

- Statements that may only be used in the body of a process.
- *Sequential statements* are executed one after the other as they appear in the design from the top of the process body to the bottom sequentially.
- A process is constantly switching between the two states:

(1) the execution phase in which the process is active (statements within this process are executed),

(2) the suspended state.

Syntax:

- **process** (*sensitivity_list*)
  [*proc_declarativ_part*]
- **begin**
  [*sequential_statement_part*]
  end process [*proc_label*];

- The *sensitivity_list* is a list of signal names within round brackets, for example (A, B, C).

A **process, becomes active by an event on one or more signal belonging to the *sensitivity list*.**

All statements between the keywords begin and end process are then executed sequentially.

Variable assignment

- Declared and used inside a process statement
- Assigned a value using statement of form
  variable object:=expression;

Variables retain value during entire simulation run

Because the process is never exited – either suspended or under execution

Digital System Design Using VHDL Notes eBook

Publisher : VTU eLearning

Author :

Type the URL : http://www.kopykitab.com/product/1830