# CHAPTER 9 *Packagesand Libraries*

This chapter explains packages and how compiled design units are stored in design libraries. It explains how the contents of design units stored in different libraries may be shared by several design units.

A package provides a convenient mechanism to store and share declarations that are common across many design units. A package is represented by

**1**. a package declaration, and optionally,
**2.** a package body.

## 9.1 Package Declaration

A package declaration contains a set of declarations that may possibly be shared by many design units. It defines the interface to the package, that is, it defines items that can be made visible to other design units, for example, a function declaration. A package body, in contrast, contains the hidden details of a package, for example, a function body.

The syntax of a package declaration is

```
package package-name is
    package-item-declarations "> These may be:
        - subprogram declarations ~ type declarations
        - subtype declarations
        - constant declarations
        - signal declarations
        - file declarations
        - alias declarations
        - component declarations
        - attribute declarations
        - attribute specifications
        - disconnection specifications
        - use clauses
end [ package-name ] ;
```

An example of a package declaration is given next.

```
package SYNTH_PACK is
    constant LOW2HIGH: TIME := 20ns:
    type ALU_OP is (ADD, SUB, MUL, DIV, EQL);
    attribute PIPELINE: BOOLEAN;
    type MVL is ('U', '0', '1', 'Z');
    type MVL_VECTOR is array (NATURAL range <>) of MVL;
    subtype MY_ALU_OP is ALU_OP range ADD to DIV;
    component NAND2
        port (A, B: in MVL; C: out MVL);
    end component;
end SYNTH_PACK;
```

Items declared in a package declaration can be accessed by other design units by using the library and use context clauses. The set of common declarations may also include function and procedure declarations and deferred constant declarations. In this case, the behavior of the subprograms and the values of the deferred constants are specified in a separate design unit called the package body. Since the previous package example did not contain any subprogram declarations and deferred constant declarations, a package body was not necessary.

Consider the following package declaration.

```
use WORK.SYNTH_PACK.all:
package PROGRAM_PACK is
    constant PROP_DELAY: TIME;      -A deferred constant.
    function "and" (L, R: MVL) return MVL;
    procedure LOAD (signal ARRAY_NAME: inout MVL_VECTOR;
        START_BIT, STOP_BIT, INT_VALUE: in INTEGER);
end PROGRAM_PACK;
```

In this case, a package body is required.

## 9.2 Package Body

A package body primarily contains the behavior of the subprograms and the values of the deferred constants declared in a package declaration. It may contain other declarations as well, as shown by the following syntax of a package body.

```
package body package-name is
        package-body-item-daclarations "> These are:
                - subprogram bodies          -- complete constant declarations
                - subprogram declarations
                - type and subtype declarations
                - file and alias declarations
                - use clauses
end [ package-name ];
```

The package name must be the same as the name of its corresponding package declaration. A package body is not necessary if its associated package declaration does not have any subprogram or deferred constant declarations. The associated package body for the package declaration, PROGRAM_PACK, described in the previous section is

```
package body PROGRAM_PACK is
        constant PROP_DELAY: TIME := 15ns;
        function "and" (L, R: MVL) return MVL is
        begin
                return TABLE_AND(L, R);
                -- TABLE_AND is a 2-D constant defined elsewhere.
        end "and";
        procedure LOAD (signal ARRAY_NAME: inout MVL_VECTOR;
                START_BIT, STOP_BIT, INT_VALUE: in INTEGER) is
                -- Local declarations here.
        begin
                -- Procedure behavior here.
        end LOAD;
end PROGRAM_PACK;
```

An item declared inside a package body has its scope restricted to be within the package body and it cannot be made visible in other design units. This is in contrast to items declared in a package declaration that can be accessed by other design units. Therefore, a package body is used to store private declarations that should not be visible, while a package declaration is used to store public declarations which other design units can access. This is very similar to declarations within an architecture body which are not visible outside of its scope while items declared in an entity declaration can be made visible to other design units. An important difference between a package declaration and an entity declaration is that an entity can have multiple architecture bodies with different names, while a package declaration can have exactly one package body, the names for both being the same.

A subprogram written in any other language can be made accessible to design units by specifying a subprogram declaration in a package declaration without a subprogram body in the corresponding package body. The association of this subprogram with its declaration in the package is not defined by the language and is, therefore, tool implementation-specific.

## 9.3 Design Libraries

A compiled VHDL description is stored in a design library. A design library is an area of storage in the file system of the host environment. The format of this storage is not defined by the language. Typically, a design library is implemented on a host system as a file directory and the compiled descriptions are stored as files in this directory. The management of the design libraries is also not defined by the language and is again tool implementation-specific.

An arbitrary number of design libraries may be specified. Each design library has a logical name with which it is referenced inside a VHDL description. The association of the logical names with their physical storage names is maintained by the host environment. There is one design library with the logical name, STD, predefined in the language; this library contains the compiled descriptions for the two predefined packages, STANDARD and TEXTIO. Exactly one design library must be designated as the working library with the logical name, WORK. When a VHDL description is compiled, the compiled description is always stored in the working library. Therefore, before compilation begins, the logi'cal name WORK must point toone of the design libraries. Figure 9.1 shows a typical compilation scenario.

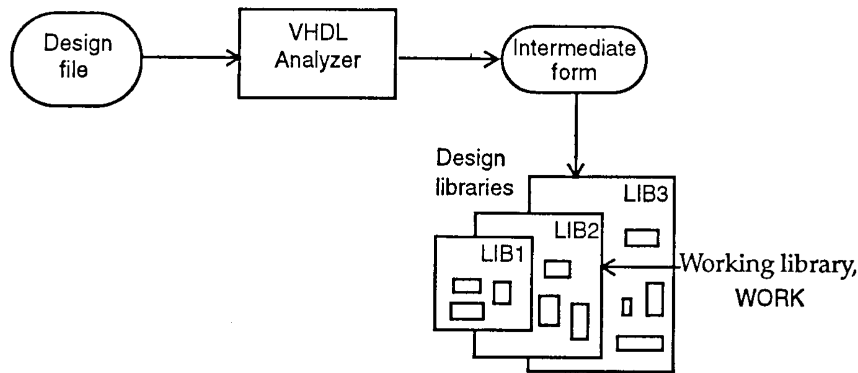



**Figure 9.1 Atypical compilation process.**

The VHDL source is present in an ASCII file called the *design file.* This is processed by the VHDL analyzer, which after verifying the syntactic and semantic correctness of the source, compiles it into an intermediate form. The intermediate form is stored in the design library that has been designated as the working library.

## 9.4 Design File

The design file is an ASCII file containing the VHDL source. It can contain one or more design units, where a design unit is one of the following:

- entity declaration,
- architecture body,
- configuration declaration,
- package declaration,
- package body.

This means that each design unit can also be compiled separately.

A design library consists of a number of compiled design units. Design units are further classified as

1. *Primary* units: These units allow items to be exported out of the design unit. They are
   a. *entity declaration:* The items declared in an entity declaration are implicitly visible within the associated architecture bodies.
   b. *package declaration:* Items declared within a package declaration can be exported to other design units using context clauses.
   c. *configuration declaration.*
2. *Secondary* units: These units do not allow items declared within them to be exported out of the design unit, that is, these items cannot be referenced in other design units. These are
   a. *architecture l)ody:* A signal declared in an architecture body, for example, cannot be referenced in other design units.
   b. *package body.*

There can be exactly one primary unit with a given name in a single design library. Secondary units associated with different primary units can have identical names in the same design library; also a secondary unit may have the same name as its associated primary unit. For example, assume there exists an entity called AND_GATE in a design library. It may have an architecture body with the same name, and another entity, MY_GATE, in the same design library may have an architecture body that also has the name, AND_GATE.

Secondary units must coexist with their associated primary units in the same design library, for example, an entity declaration and all of its architecture bodies must reside in the same library. Similarly, a package declaration and its associated package body must reside in a single library.

Even though a configuration declaration is a primary unit, it must reside in the same library as the entity declaration to which it is associated.

## 9.5 Order of Analysis

Since it is possible to export items declared in primary units to other design units, a constraint is imposed in the sequence in which design units must be analyzed. A design unit that references an item declared in another primary unit can be analyzed only after that primary unit has been analyzed. For example, if a configuration declaration references an entity, COUNTER, the entity declaration for COUNTER must be analyzed before the configuration declaration.

A primary unit must be analyzed before any of its associated secondary units. For example, an entity declaration must be analyzed before its architecture bodies are analyzed.

## 9.6 Implicit Visibility

An architecture body implicitly inherits all declarations in the entity since it is tied to that entity by virtue of the statement

**architecture** *architecture-name* **of** *entity-name* **is . . .**

Similarly, a package body implicitly inherits all items declared in the package declaration by virtue of its first statement

**package body** *package-name* **is. . .**

where the package name is the same as the one in the package declaration.

## 9.7 Explicit Visibility

Explicit visibility of items declared in other design units can be achieved using context clauses. There are two types of context clauses:

1. library clause,
2. use clause.

Context clauses are associated with design units and may be written just before the design unit to which visibility is to be made available. Figure 9.2 shows an example. Items specified in the context clause become visible only to the design unit that follows the context clause. This means that if a design file contains three design units, such as in the example shown in Fig. 9.3, then context clauses must be specified for each design unit. For example, the context clauses specified before design unit A makes them visible to design unit A only; context clauses specified before design unit B makes them visible to design unit B only.

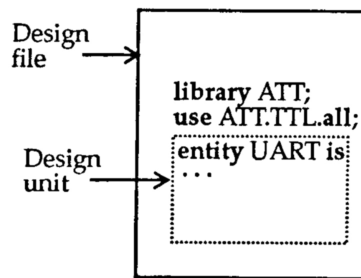



**Figure 9.2 Context clause associated w'rth following design unit.**

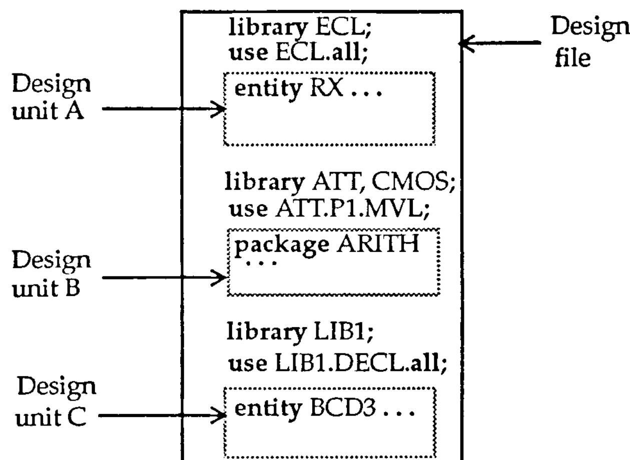



**Figure 9.3 Separate context clauses with each design unit.**

### 9.7.1 Library Clause

The library clause makes visible the logical names of design libraries that can be referenced within a design unit. The format of a library clause is

```
library list-of-logical-library-names;
```

The following example of a library clause

```
library TTL, CMOS;
```

makes the logical names, TTL and CMOS, visible in the design unit that follows. Note that the library clause does not make design units or items present in the library visible, it makes only the library name visible (it is like a declaration for a library name). For example, it would be illegal to use the expression "TTL.SYNTH_PACK.MVL" within a design unit without first declaring the library name using the "library 1TL;" clause.
The following library clause

```
library STD, WORK;
```

is implicitly declared for every design unit.

### 9.7.2 Use Clause

There are two main forms of the use clause.

```
use library-name. primafy-unit-name ;                --Form 1.
use library-name. primafy-unit-name. Item ;          --Form 2.
```

The first form of the use clause allows the specified primary unit name from the specified design library to be referenced in a design description. For example,

```
library CMOS;
use CMOS.NOR2;
configuration...
        . . . use entity NOR2( . . . );
end;
```

Note that entity NOR2 must be available in compiled form in the design library, CMOS, before attempting to compile the design unit where it is used.
The second form of the use clause makes the item declared in the pri- ' mary unit visible and the item can, therefore, be referenced within the following design unit. For example,

```
library ATTLIB;
use ATTLIB.SYNTH_PACK.MVL;
-- MVL is a type declared in SYNTH_PACK package.
-- The package, SYNTH_PACK, is stored in the ATTLIB design library.
entity NAND2 is
        port (A, B: in MVL; ...)...
```

If all items within a primary unit are to be made visible, the keyword **all** can be used. For example,

```
use ATTLIB.SYNTH_PACK.all;
```

makes all items declared in package SYNTH_PACK in design library ATTLIB visible.
Items external to a design unit can be accessed by other means as well. One way is to use a selected name. An example of using a selected name is

```
library ATTLIB;
use ATTLIB.SYNTH_PACK;
entity NOR2 is
        port (A, B: in SYNTH_PACK.MVL; ...)...
```

Since only the primary unit name was made visible by the **use** clause, the complete name of the item, that is, SYNTH_PACK.MVL must be specified. Another example is shown next. The type VALUE_9 is defined in package SIMPACK that has been compiled into the CMOS design library.

```
library CMOS;
package P1 is
        procedure LOAD (A, B: CMOS.SIMPACK.VALUE_9; ...)...
         . . .
```

```
end P1;
```

In this case, the primary unit name was specified only at the time of usage.
So far, we talked about exporting items across design libraries. What if it is necessary to export items from design units that are in the same library? In this case, there is no need to specify a library clause since every design unit has the following library clause implicitly declared.

```
library WORK;
```

The predefined design library STD contains the package STANDARD. The package STANDARD contains the declarations for the predefined types such as CHARACTER, BOOLEAN, BIT_VECTOR, and INTEGER. The following two clauses are also implicitly declared for every design unit:

```
library STD;
use STD.STANDARD.all;
```

Thus all items declared within the package STANDARD are available for use in every VHDL description.

# CHAPTER 10 *Advanced Features*

This chapter describes some of the more general features of the language that cannot be characterized as belonging to any specific modeling style. These include among others, attributes, type conversions, entity statements, aliases, generate statements, and guarded signals. The usage of block statements as a partitioning mechanism is also described.

## 10.1 Entity Statements

Certain statements that are common to all architecture bodies of an entity can be inserted into the entity declaration. Common declarations appear in the entity declarative part while other common statements appear in the entity statement part as shown.

```
entity entity-name is
        [ generic(. . . ); ]
        [port(...):]
        [ entity-item-declarations]          -- Declarative part.
[begin
        entity-statements ]                  -- Statements part.
end [ entity-name ],
```

*Entity-statements* must be *-passive* statements, that is, they must not assign values to any signals. The following statements are allowed as entity statements:

**1.** concurrent assertion statement,
**2.** concurrent procedure call (must be passive),
**3.** process statement (must be passive).

Entity statements can be used to monitor certain operating characteristics of an entity. For example, in an RS flip-flop, a check can be made to ensure that signals R and S are never high simultaneously. Here is a way of modeling this using an assertion statement.

```
entity RS_FLIPFLOP is
        port (R,S: in BIT; Q, QBAR: out BIT);
        constant FF_DELAY: TIME := 24 ns;
        type FF_STATE is (ONE, ZERO, UNKNOWN);
begin
        assert not (R = '1' and S = '1')
                report ("Not valid inputs!")
                severity ERROR;
end RS_FLIPFLOP;
```

The constant and type declarations in this example are examples of item declarations within an entity declaration. These item declarations are common to all the architecture bodies for that entity. Items declared by these declarations are also visible to all other design units associated with this entity declaration.

In the following example, the timing of a D flip-flop is checked using a concurrent procedure call that appears as an entity statement. This procedure will be called every time there is an event on input ports D and CLK, irrespective of the contents in the architecture bodies that are associated with this entity

```
use WORK.MYPACK.CHECK_SETUP;
entity DFF is
        port (D, CLK: in BIT; 0, QBAR: out BIT);
        constant SETUP: TIME := 7 ns:
begin
        CHECK_SETUP (D, CLK, SETUP);
end DFF;
```

## 10.2 Generate Statements

Concurrent statements can be conditionally selected or replicated during the elaboration phase using the generate statement. There are two forms of the generate statement.

**1.** Using the for-generaHon scheme, concurrent statements can be replicated a predetermined number of times.
**2.** With the if-generation scheme, concurrent statements can be conditionally selected for execution.

The generate statement is interpreted during elaboration, and therefore, has no simulation semantics associated with it. It resembles a macro expansion. The generate statement provides for a compact description of regular structures such as memories, registers, and counters.

The format of a generate statement using the for-generation scheme is

```
generate-label: for generale-identifierin discrete-range generate
concurrent-statements end generate [ generate-label];
```

The values in the discrete range must be globally static, that is, they must be computable at elaboration time. During elaboration, the set of concurrent statements are replicated once for each value in the discrete range. These statements can also use the generate identifier in their expressions and its value would be substituted during elaboration for each replication. There is an implicit declaration for the generate identifier within the generate statement, and therefore, no declaration for this identifier is required. The type of the identifier is defined by the discrete range.
Consider the following representation of a 4-bit full-adder, shown in Fig. 10.1, using the generate statement.

```
entity FULL_ADD4 is
        port (A, B: in BIT_VECTOR(3 downto 0); CIN: in BIT;
                SUM: out BIT_VECTOR(3 downto 0); COUT: out BIT);
end FULL_ADD4:

architecture FOR_GENERATE of FULL_ADD4 is
        component FULL_ADDER
                port (A, B, C: in BIT; COUT, SUM: out BIT);
        end component;
        signal CAR: BIT_VECTOR(4 downto 0);
begin
        CAR(0) <= CIN;
        GK: for K in 3 downto 0 generate
        FA: FULL_ADDER port map (CAR(K), A(K), B(K),
                                CAR(K+1),SUM(K));
        end generate GK;
        COUT <= CAR(4);
end FOR_GENERATE;
```

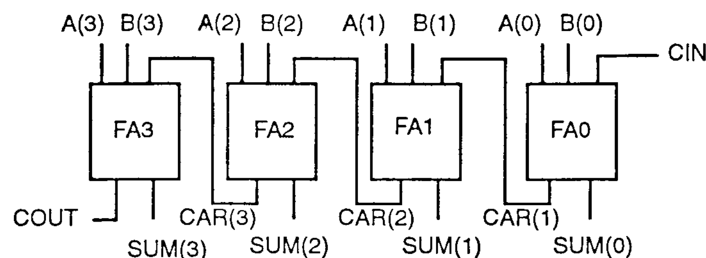



**Figure 10.1 A 4-bit full-adder.**

After elaboration, the generate statement is expanded to

```
FA(3): FULL_ADDER port map (CAR(3), A(3), B(3), CAR(4), SUM(3));
FA(2): FULL_ADDER port map (CAR(2), A(2), B(2), CAR(3), SUM(2));
FA(1): FULL_ADDER port map (CAR(1), A(1), B(1), CAR(2), SUM(1));
FA(0): FULL_ADDER port map (CAR(0), A(0), B(0), CAR(1), SUM(0));
```

Components in a generate statement can be bound to entities using a generate block configuration. A block configuration is defined for each range of generate labels. Here is an example of such a binding using a configuration declaration.

```
configuration GENERATE_BIND of FULL_ADD4 is
        use WORK.all;   -- Example of a declaration in the
                                -- configuration declarative part.
        for FOR_GENERATE        -- An architecture body block configuration.
                forGK(1)        --A generate block configuration.
                        for FA: FULL_ADDER
                                use configuration WORK.FA_HA_CON;
                        end for;
                end for;
                for GK(2 to 3)
                        for FA: FULL_ADDER - No explicit binding.
                                -- Use defaults, i.e., use entity FULL_ADDER
                                -- in working library.
                        end for;
                end for;
                for GK(0)
```

```
                    for FA: FULL_ADDER
                        use                    entity
                    WORK.FULL_ADDER(FA_DATAFLOW);
                    end for;
                end for;
            end for;
end GENERATE_BIND;
```

There are three generate block configurations, one each for GK(1), GK(2 to 3), and for GK(0). Each of these block configurations define the bindings for the components valid for that generate index.

The body of the generate statement can also have other concurrent statements. For example, in the previous architecture body, the component instantiation statement could be replaced by signal assignment statements like this

```
G2: for M in 3 downto 0 generate
        SUM(M) <= (A(M) xor B(M)) xor CAR(M);
        CAR(M+1 ) <= (A(M) and B(M)) and CAR(M);
end generate G2;
```

The second form of the generate statement uses the if-generation scheme. The format for this type of generate statement is

```
genarate-label: H expression generate
        concurrent-statements
end generate [ generete-label ] ;
```

The if-generate statement allows for conditional selection of concurrent statements based on the value of an expression. This expression must be a globally static expression, that is, the value must be computable at elaboration time.

Here is an example of a 4-bit counter, shown in Fig. 10.2, that is modeled using the if-generate statement.

```
entity COUNTER4 is
        port (COUNT, CLOCK: in BIT; Q: buffer BIT_VECTOR(0 to 3));
end COUNTER4;

architecture IF_GENERATE of COUNTER4 is
        component D_FLIP_FLOP
                port (D, CLK: in BIT; Q: out BIT);
        end component;
begin
        GK: for K in 0 to 3 generate GKO:
                if K = 0 generate
                        DFF: D_FLIP_FLOP port map (COUNT, CLOCK, Q(K));
                end generate GK0;
                GK1_3: if K > 0 generate
                        DFF: D_FLIP_FLOP port map (Q(K-1), CLOCK, Q(K));
                end generate GK1_3;
        end generate GK;
end IF_GENERATE;
```

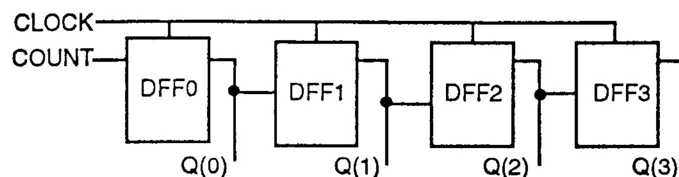



**Figure 10.2 A 4-bit counter.**

A simpler example is when a buffer is to be selected that has different delays based on the value of a constant. Here are the generate statements for such an example.

```
GA: if USER_WANTS = LOW_DELAY generate
        Z <= A after 2 ns;
end generate;
```

```
GB: if USER_WANTS = MEDIUM_DELAY generate
        Z <= A after 10 ns;
end generate;
GC: if USER_WANTS = HIGH_DEU\Y generate
        Z <= A after 25 ns;
end generate;
```

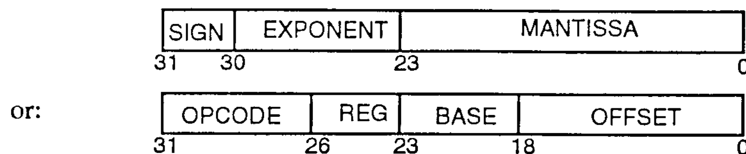
The if-generate statement is also useful in modeling repetitive structures, especially in modeling boundary conditions. An if-generate statement does not have an else or an else-if branch.

## 10.3 Aliases

An alias declares an alternate name for all or part of an existing object. It provides a convenient ghorthand notation to write names for objects that have long names. It also provides a mechanism of referring to the same object in different ways depending on the context. For example,

```
signal S: BIT_VECTOR (31 downto 0);
```

can represent:




The syntax for an alias declaration is

**alias** *identifier: identifier-type* **is** *name*;

For example,

```
variable DATA_WORD: BIT_VECTOR(15 downto 0);
alias DATA_BUS: BIT_VECTOR(7 downto 0) is
DATA_WORD(15 downto 8);
alias STATUS: BIT_VECTOR(0 to 3) is DATA_WORD(3 downto 0);
alias RESET: BIT is DATA_WORD(4);
alias RX_READY: BIT is DATA_WORD(5);
```

Given these declarations, DATA_BUS can be used wherever DATA_WORD(15 downto 8) is used, RESET can be used wherever DATA_WORD(4) is used, and so on. It is important to note that assigning a value to an alias name is the same as assigning a value to the aliased name. The alias does not declare a new object but merely provides an alternate way of referring to the original object.

The alias of an array object may change the way in which the array is in-, dexed, as shown in the alias declaration of STATUS. Only objects (signals, variables, and constants) can have aliases, not types. Alternate names can be provided for types by using subtype declarations.

## 10.4 Qualified Expressions

The type of an expression can be explicitly specified by qualifying the expression with its type. For example,

```
INTEGER'(A * B + C)
```

qualifies the expression (A * B+C) to have an integer value. The qualification is useful for type checking and it does not imply any type conversion.

Apart from its type checking usefulness, type qualification can be used to qualify types for expressions whose types cannot be determined from their context. Consider the following two overloaded procedure declarations.

```
procedure CHAR21NT (A: in CHARACTER; Z: out INTEGER);
procedure CHAR21NT (A: in BIT; Z: out INTEGER);
```

A call to a procedure such as

```
CHAR21NT ( '1', N);
```

is ambiguous since it is not clear which overloaded procedure is to be called. However, if the expression in the procedure call were modified to be

```
CHAR21NT (CHARACTER' ('1'), N);
```

it is clear that this call refers to the first procedure.

## 10.5 Type Conversions

The language does allow for a very restricted set of type casting, that is, explicitly converting values between types. For example,

```
SUM := INTEGER (POLYWIDTH * 1.5);
```

will convert the real value obtained from POLYWIDTH * 1.5 to its equivalent integer value and assign it to SUM.

Type conversions are allowed between closely related types. These include among others, between integer and real, between array types that have the same dimensions and whose index types that are the same, and element types that are the same. Any other kind of type conversion, other than those predefined by the language, can be performed by using a user-defined function.

## 10.6 Guarded Signals

A *guarded* signal is a special type of a signal that is declared to be of a register or a bus kind in its declaration. A general form of a signal declaration is

```
signal list-of-signals: resolution-function signal-type
                        signal-kind [ := expression ];
```

A guarded signal must be a resolved signal, that is, it must have a resolution function associated with it. Also, the signal can only be assigned values under the control of a guard expression, for example, using a guarded assignment (guarded option used in a concurrent signal assignment statement). This implies that guarded signals can only be assigned values within block statements.

A guarded signal behaves differently from other signals in that when the guard expression is false, the driver to the guarded signal becomes disconnected after a specific time, called the *disconnect time.* On the other hand, in an unguarded signal, if the guard expression is false, any new events on the signals appearing in the expression do not influence the value of the target signal; the driver continues to drive the target signal with the old value. To understand this difference better, consider the following guarded block BL

```
architecture GUARDED_EX of EXAMPLE is
        signal GUARD_SIG: WIRED_OR BIT register;
        signal UNGUARD_SIG: WIRED_AND BIT;
begin
        B1: block ( guard-expression )
        begin
                GUARD_SIG <= guarded expression1 ;
                UNGUARD_SIG <= guarded expression2;
        end block B1;
end GUARDED_EX;
```

Transforming the guarded signal assignment statement into its equivalent process statement, the block B1 now looks like this

```
B1: block ( guard-expression )
begin
        process
        begin
                if GUARD then
                        GUARD_SIG <= expression1;
                else
                        GUARD_SIG <= null;      -- Disconnect driver
                                                -- to GUARD_SIG.
```

```
            end if;
            wait on signals-in-expressioni1, GUARD;
        end process;
        process
        begin
            If GUARD then
                UNGUARD_SIG <= expression2;
            end if;
            wait on signals-in-expressionS, GUARD;
        end process;
    end block B1;
```

The process statement for the guarded signal, GUARD_SIG, has an explicit signal assignment statement that disconnects its driver, while there is no such statement for the unguarded signal, UNGUARD_SIG. As this example shows, a driver of a guarded signal can be explicitly disconnected by assigning a null value to the signal. Such a statement is called a *disconnection statement.*

Let us now explore the differences between a register and a bus signal. A bus signal represents a hardware bus in that when all drivers to the signal become disconnected (as might be the case on a real hardware bus), the value of the signal is determined by calling the resolution function with all the drivers off. A register signal, on the other hand, models a storage component (that is multiply driven) in which if all drivers to the signal become disconnected, the resolution function is not called and the value of the last active driver is retained. With a bus signal, the previous value is lost. Also, bus signals may either be ports of an entity or locally declared signals, whereas register signals can only be locally declared signals.

The disconnect time for a guarded signal can be specified using a *disconnection specification.* The syntax of a disconnection specification is

```
disconnect guarded-signal-name: signal-type after time-expression;
```

This is an example of a disconnection specification.

```
disconnect GUARD_SIG: BIT after 8 ns;
```

This implies that the driver of signal GUARD_SIG will get disconnected 8 ns after the corresponding GUARD goes false.

The disconnection specification is useful in modeling decay times, for example, capacitance delay on buses. An alternate way of specifying disconnect time is by assigning a value null to the signal in a disconnection statement as shown.

```
S1 <= null after 10 ns;
```

This statement specifies that the driver of SI will be disconnected after 10 ns. Thereafter, this driver does not contribute to the resolved value of the signal. However, such a statement can appear only as a sequential statement and the target signal must be a guarded signal.

Here is a more comprehensive example.

```
use WORK.RF.PACK.all;
-- Package RF_PACK contains functions WIRED_AND and WIRED_OR.
entity GUARDED_SIGNALS is
    port (CLOCK: in BIT; N: in INTEGER);
end;

architecture EXAMPLE of GUARDED_SIGNALS is
    signal REG_SIG: WIRED_AND INTEGER register;
    signal BUS_SIG: WIRED_OR INTEGER bus;
    disconnect REG_SIG: INTEGER after 50 ns;
    disconnect BUS_SIG: INTEGER after 20 ns;
begin
    BX: block (CLOCK='1' and (not CLOCK'STABLE))
    begin
        REG_SIG <= guarded N after 15 ns;
        BUS_SIG <= guarded N after 10 ns;
    end block BX;
end EXAMPLE;
```

On a rising edge on the signal CLOCK, say at time T, the current value of N is scheduled to be assigned to signal REG_SIG after 15 ns, that is, at T+15 ns, and to signal BUS_SIG after 10 ns, that is, at T+IO ns. However, because of the disconnection specifications, the drivers to signals REG_SIG and BUS_SIG are scheduled to be disconnected after 15+50 ns, that is, at T+65 ns, and after 10+20 ns, that is, at T+30 ns, respectively. At time T+30 ns, the function WIRED_OR is called to determine the value for the signal BUS_SIG, even if all its drivers are off. At time T+65 ns, the driver to signal REG_SIG disconnects, the value on signal REG_SIG is retained and the resolution function, WIRED_AND, is not called (since there are zero drivers).

## 10.7 Attributes

An attribute is a value, function, type, range, signal, or a constant that can be associated with certain names within a VHDL description. These names could be among others, an entity name, an architecture name, a label, or a signal. For example, a record that contains the X and Y coordinates of a component placement could be associated with an entity name that describes the placement for that entity in a physical layout; a capacitance value can be associated with all signals of a specific type.

A large number of attributes are predefined in the language. These are described in Sec. 10.7.2. The language also provides the facility to associate user-defined attributes to names.

### 10.7.1 User-Defined Attributes

User-defined attributes are constants of any type. They are declared using attribute declarations. An *attribute declaration* declares an attribute name and its type and has the following form:

```
attribute attribute-nama: value-type;
```

For example,

```
type COMP_LOCATION is
        record
                X, Y; INTEGER;
        end record;
type INCHES is range 0 to 1000
        units
                micron;
        end units;
type FARADS is range 0 to 5000
        units
                pf;
        end units;
attribute PLACEMENT: COMP_LOCATION;
attribute LENGTH: INCHES;
attribute CAPACITANCE: FARADS;
```

These declared attributes have not yet been associated with any name. An *attribute specification* is used to associate an attribute with a name and to assign a value to the attribute. The syntax for an attribute specification is

```
attribute attribute-name of item-names: name-class is expression;
```

The *item-names* is a list of one or more names of an entity, architecture, configuration, component, label, signal, variable, constant, type, subtype, package, procedure, or a function. The *name-class* indicates the class type, thal is, whether it is an entity, architecture, label, or others. The attribute name must have been declared earlier using an attribute declaration. The attribute specification associates an attribute with a group of names that belong to a name class that has the value as specified by the expression (the value of expression must belong to the type of attribute). Some examples of attribute specifications are

```
attribute CAPACITANCE of CLK, RESET: signal is 20 pf;
attribute LENGTH of RX_READY: signal is 3 micron;
```

In the first example, the attribute CAPACITANCE is associated with two sig nals CLK and RESET, each of which has a value of 20 pf.

The item name in the attribute specification can also be replaced witl the keyword all to indicate all names belonging to that name class. In the fol lowing example, the attribute CAPACITANCE is associated with all variable: and the attribute value is set to 0 pf.

```
attribute CAPACITANCE of all: variable is 0 pf;
```

After having created an attribute and then having associated it with a name, the value of the attribute can then be used in an expression by referring to

```
item-name ' attribute-name          -- The single quote is often read as "tick".
```

For example, RX_READY'LENGTH has the value of 3 micron. Here is a bigger example.

```
architecture NAND_PLACE of NAND_GATE is
        component NAND_COMP
                port (IN1, IN2: in BIT; OUT1: out BIT);
        end component;
        type COMP_LOCATION is
                record
                        X, Y: INTEGER;
                end record;
        attribute PLACEMENT: COMP_LOCATION;
        attribute PLACEMENT of N1: label is (50, 45);
        signal PERIMETER: INTEGER;
        signal A, B, Z: BIT;
begin
        N1: NAND_COMP port map (A, B, Z);
        PERIMETER <= 2 * (N1'PLACEMENTS +
                        N1'PLACEMENT.Y);
end NAND_PLACE;
```

User-defined attributes are useful for annotating VHDL models with tool-specific information.

### 10.7.2 Predefined Attributes

There are five classes of predefined attributes.

1. Value attributes: these return a constant value
2. Function attributes: calls a function that returns a value
3. Signal attributes: creates a new signal
4. Type attributes: returns a type name
5. Range attributes: returns a range

### Value Attributes

If T is any scalar type or subtype,

- T'LEFT: returns the left bound, that is, the leftmost value, of T.
- T'RIGHT: returns the right bound, that is, the rightmost value, of T.
- T'HIGH: returns the upper bound, that is, the value at the highest position number, of T.
- T'LOW: returns the lower bound, that is, the value at the lowest position number, of T.

For example, if

```
type ALLOWED_VALUE is range 31 downto 0;
type WEEK_DAY is (SUN, MON, TUE, WED, THU, FRI, SAT);
subtype WORK_DAY is WEEK_DAY range FRI downto MON;
```

then the following equivalence relations are true:

```
ALLOWED_VALUE'LEFT = 31
ALLOWED_VALUE'HIGH = 31

ALLOWED_VALUE'RIGHT = 0
ALLOWED_VALUE'LOW = ALLOWED_VALUE'RIGHT
WEEK_DAY'LEFT = SUN;
WEEK_DAY'LOW = WEEK_DAY'LEFT
WEEK_DAY'RIGHT = SAT;
WEEK_DAY'HIGH = WEEK_DAY'RIGHT
WORK_DAY'RIGHT = MON;
```

WORK_DAY'LOW = WORK_DAY'RIGHT
WORK_DAY'LEFT= FRI;
WORK_DAY'HIGH **=** WORK_DAY'LEFT

If A is a constrained array object, then

- A'LENGTH(N) : returns the number of elements in the Nth dimension (N=l if not specified).

For example, if

**signal** TX_BUS: MVL_VECTOR (7 **downto** 0);

then

TX_BUS'LENGTH = 8.

If BA is a block label or an architecture name, then
• BA'BEHAVIOR: is true if no structure is present, that is, there are no component instantiation statements.
• BA'STRUCTURE: is true if only structure is present, that is, it does not contain a nonpassive process statement or a concurrent statement whose equivalent process statement is nonpassive.

### Function Attributes

These attributes represent functions that are called to obtain a value. They are often used to convert values from an enumeration or physical type to an integer type. If T is a discrete type, a physical type, or a subtype,

- T'POS(V): returns the position number of the value V in the ordered list of values of T.
- T'VAUP): returns the value of the type that corresponds to position P.
- T'SUCC(V): returns the value of the parameter whose position is one larger than the position of value V in T.
- T'PRED(V): returns the value of the parameter whose position is one less than the position of value V in type T.
- T'LEFTOF(V): returns the value of th^parameter that is to the left of value V in type T.
- T'RIGHTOF(V): returns the value of the parameter that is to the right of value V in type T.

For ascending ranges,

T'SUCC(X) = T'RIGHTOF(X)
T'PRED(X) = T'LEFTOF(X)

For descending ranges,

T'SUCC(X) = T'LEFTOF(X)
T'PRED(X) = T'RIGHTOF(X)

For example, if

**type** STATUS is (SILENT, SEND, RECEIVE);
**subtype** DELAY_TIME is TIME **range 50 ns downto** 10 ns;

then the following equivalence relations are true:

STATUS'POS(SEND) = 1
STATUS'VAL(2) = RECEIVE
DELAY_TIME'SUCC(21 ns) = 22 ns
DELAY_TIME'PRED(10 ns) is an error
DELAY_TIME'LEFTOF(29 ns) = 30 ns
DELAY_TIME'SUCC(29 ns) = DELAY_TIME'LEFTOF(29 ns)
DELAY_TIME'RIGHTOF(11 ns) = 10ns
DELAY_TIME'PRED(11 ns) = DELAY_TIME'RIGHTOF(11 ns)
STATUS'SUCC(RECEIVE) will cause a run-time error
STATUS'PRED(RECEIVE) = SEND
STATUS'LEFTOF(RECEIVE) = STATUS'PRED(RECEIVE)
STATUS'SUCC(SILENT) = SEND
STATUS'RIGHTOF(SILENT) = STATUS'SUCC(SILENT)

If A is a constrained array object, then

- A'LEFT(N): returns the left bound of the Nth dimension of the array.
- A'R.IGHT(N): returns the right bound of the Nth dimension.
- A'LOW(N): returns the lower bound of the Nth dimension.
- A'HIGH(N): returns the upper bound of the Nth dimension.

For ascending ranges,

```
A'LEFT = A'LOW
A'RIGHT = A'HIGH
```

For descending ranges,

```
A'LEFT = A'HIGH
A'RIGHT = A'LOW
```

In all the previous cases, N=1 if not specified. Some more examples are shown next

If

```
type COST_TYPE is array (7 downto 0,0 to 3) of INTEGER;
variable COST_MATRIX: COST_TYPE;
```

then the following equivalence relations are true:

```
COST_MATRIX'LEFT(2) = 0
COST_MATRIX'LOW(1) = 0
COST_MATRIX'RIGHT(2) = 3
COST_MATRIX'HIGH(1) =7
```

If S is a signal object, then

- S'EVENT: returns true if an event occurred on signal S in the current delta.
- S'ACTIVE: returns true if signal S is active in the current delta.It is important to understand the difference between an event on a signal and a signal being active. A signal is said to be active when a new value is assigned to the signal, even if the new value is the same as the old value; while an event is said to have occurred only if the new value is different from the old value.
- S'LAST_EVENT: returns time elapsed since the last event on signal S.
- S'LAST_ACTIVE: returns time elapsed since the last time signal was active.
- S'LAST_VALUE: returns the value of S, before the last event.

The following examples show their usage in expressions:

```
signal CLOCK: BIT;
constant SETUP_TIME: TIME := 5 ns:
signal A: BIT;
signal COUNT: INTEGER;

(CLOCK = '1' and CLOCK'EVENT)          -- Denotes a rising edge on
                                       -- signal CLOCK.
(NOW-A'LAST.EVENT < SETUP_TIME)        -- NOW is a predefined
        -- function that returns the current simulation time.
(COUNT = 20 and COUNT'LAST_VALUE = 10)
```

### Signal Attributes

These attributes create new signals from the signals with which they are associated. These attributes, therefore, create implicit signals as compared to explicit signals that are created using signal declarations. If S is a signal object, then

- S'DELAYED (T): is a new signal that is the same type as signal S but delayed from S by time T. If T is not specified, a delay of 0ns is assumed.
- S'STABLE (T): is a boolean signal that is true when signal S has not had any event for time T. If T is not specified, it implies current delta.
- S'QUIET (T): creates a boolean signal that is true when S has not been active for time T.
- STRANSACTION: creates a bit type signal that toggles its value every time signal S becomes active.

Assuming,

**signal** CLOCK_SKEW, CTRL.A: BIT;

Figure 10.3 shows some examples of signal attributes.

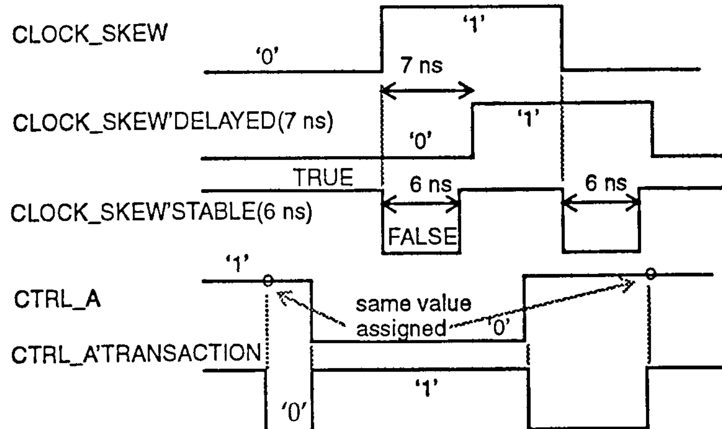



**Figure 10.3 Signal attribute examples.**

It is important to understand the difference between function attributes for signals and signal attributes. Signal attributes create new signals, and therefore, cause events when used in concurrent statements; whereas function attributes for signals do not create new signals, and therefore, do not create new events. Signal attributes can, therefore, be used wherever a signal is expected, for example, in the sensitivity list of a process statement or as a signal parameter in a procedure. For example,

```
PSTABLE: process (A, B, CLK'STABLE)
begin
 . . .
end process;
```

It would be illegal to use CLK'EVENT since only signals are allowed in the sensitivity list of a process. Therefore, function attributes of signals should not be used in expressions where the intention is to create new events.

## Type Attributes

If T is any type or subtype, T'BASE, which is the only type attribute, returns the base type of T. This attribute cannot be used in expressions as such since it returns a base type but it can be used in conjunction with other attributes. For example, if

**type** ALU_OPS is (ADDOP, SUBOP, MULOP, DIVOP, ANDOP, NANDOP, OROP, NOROP); **subtype** ARITH_OPS is ALU_OPS **range** ADDOP to DIVOP;

then

ARITH_OPS'BASE is ALU_OPS

which is the base type, and it can be used in an expression such as

ARITH_OPS'BASE'LEFT which has the value ADDOP.

## Range Attributes

If A is a constrained array object, then

- A'RANGE (N): returns the Nth index range of A (N=l, if not specified).
- A'REVERSE.RANGE(N): returns the Nth index range reversed.

For example, if

**variable** WBUS: MVL_VECTOR(7 **downto** 0);

then

WBUS'RANGE returns the range "7 **downto** 0" (quotations not included)

while

WBUS'REVERSE_RANGE returns the range "0 **to** 7".

The range attributes provide a mechanism for specifying the index constraint in for loops and for-generate statements in a parameterized way. For example, a use of this attribute in a for loop is shown next.

```
for INDEX in WBUS'REVERSE_RANGE loop
 . . .
end loop;
```

## 10.8 Aggregate Targets

It is possible for the target of a variable assignment statement or a signal assignment statement to be an aggregate target. An aggregate target represents a combination of one or more names that is of the array type or the record type.

Here is an example.

```
signal A, B, C: BIT;

(A, B, C) <= BIT_VECTOR'("100");
```

The left-hand side represents an aggregate target made up of three individual signals. The signal assignment statement indicates that signal A is assigned the value *'V,* and signals B and C are assigned the value '0'. Here is another example.

```
type COUNT_REC is
record
INPUTS, OUTPUTS: POSITIVE;
end record;
variable BOX1, BOX2: COUNT_REC;
variable TOTAL_IN, TOTAL_OUT: POSITIVE;
function "+" (A1, A2: COUNT_REC) return COUNT_REC;

(TOTAL_IN, TOTAL_OUT) := BOX1 + BOX2;
```

In this case, variable TOTALJN gets the value of the INPUTS element of the overloaded function result, and TOTAL_OUT gets the value of the OUTPUTS element of the function result.

In general, an assignment to an aggregate target causes subelement values in the right-hand-side expression to be assigned to the corresponding subelements in the left-hand-side target.

## 10.9 More on Block Statements

One other use of a block statement is in representing a portion of a design. This means that an architecture body can be made up of a number of block statements, each representing a portion of a design, along with other concurrent statements. Block statements in turn can contain one or more concurrent statements, including other block statements. Block statements, thus, provide a mechanism for creating a hierarchy within an architecture body.

When a portion of a design is represented using a block statement, it is possible to use arbitrary names within the block and then to associate these names to those outside of that block by using the generic map and port map in the block header. The next example shows a description of a full-adder that has been partitioned into three blocks (where each could have been designed by a separate designer). Two of these blocks, HAI and HA2, represent the two half-adders, and the block ORI represents an or gate. Different styles of modeling are chosen in describing the half-adders and the or gate to show that block statements can contain any form of concurrent statements.

```
entity FULL_ADDER is
    generic (HA_DELAY: TIME := 5 ns);
    port (A, B, CIN: in BIT; SUM, COUT: out BIT);
end FULL_ADDER;
```

```
architecture BLOCK_VIEW of PULL_ADDER is
        signal S1, C1, C2: BIT;
begin
        HA1: block
                generic (CARRY_DELAY: TIME);
                generic map (CARRY_DELAY=> HA_DELAY);
                port (IN1, IN2: in BIT; SUM, CARRY: out BIT);
                port map (IN1=> A, IN2=>B, SUM => S1, CARRY => C1);
        begin
                SUM <= (IN1 and (not IN2)) or (IN2 and (not IN1));
                CARRY <= IN1 and IN2 after CARRY_DELAY;
        end block HA1;

        HA2: block
                port (X, Y: in BIT; S, C: out BIT);
                port map (X => CIN, Y => S1, S => SUM, C => C2);
        begin
                process (X, Y)
                begin
                        S<=X xor Y;
                        C <= X and Y after HA_DELAY;
                end process;
        end block HA2;

        OR1: block
                port (A, B: in BIT; C: out BIT);
                port map (A => C1, C => COUT, B => C2);
                component OR_GATE
                        port (X, Y: in BIT; Z: out BIT);
                end component:
        begin
                O1: OR_GATE port map (A, B, C):
        end block OR1;
end BLOCK.VIEW;
```

Block HA1 describes a half-adder using concurrent signal assignment statements. The generic and port map associate the local names (those within the block) to external names (names outside the block). Signal A, which is the input to the entity FULL_ADDER, is associated with signal INl used inside block HAI, signal SI defined in the architecture body is associated with signal SUM declared within the HAI block, and so on. Block HA2 describes a half-adder using a process statement, while block ORI contains a component instantiation statement to represent an or gate. Port signals declared within each block are local to that block while signals declared in the architecture body and in the entity declaration are global to all the blocks. For example, signal SUM in HAI refers to the signal SUM declared in that block and not to the output signal SUM. The HA_DELAY used in block HA2 is the generic declared in the entity declaration.

When components are instantiated inside a block statement, it is necessary to specify a block configuration for the block as well when writing the configuration declaration for such a design. Here is an example of a configuration declaration for the previous FULL_ADDER entity.

```
library CMOS_LIB;
configuration BLOCK_VIEW_CON of FULL_ADDER is
        for BLOCK_VIEW
                for HA1  -- A block configuration.
                         -- Nothing to configure.
                end for;
                for HA2  -- A block configuration.
                         -- Nothing to configure.
                end for;
                for OR1   -- A block configuration.
                        for O1: OR_GATE
                                use entity CMOS_LIB.OR2(OR2);
                        end for;
                end for;
        end for;
end BLOCK_VIEW_CON;
```

Within the block configuration for the architecture body, there are three other block configurations, one for each of the block statements. Blocks HAI and HA2 do not have any component instantiations and have, therefore, nothing to bind. It is not necessary to specify the block configurations for such blocks. The block ORI has one component that is bound to an entity represented by the entity-architecture pair from library CMOS_LIB.

Since a block statement is just another concurrent statement, block statements can be nested. Here is an example.

```
B1: block
        signal A, B: BIT;
begin
        . . .
        B2: block
                signal B, C: BIT;
        begin
                A <= B and C;           -- First signal assignment.
                C <= B1 .B;             -- Second signal assignment.
        end block B2;
end block B1;
```

The signal B used in the first signal assignment in block B2 refers to the signal B declared in that block. If the signal B declared in block B1 needs to be used within block B2, the signal must be explicitly qualified by the block label as shown in the second statement in block B2.

# CHAPTER 11 *Model Simulation*

This chapter describes the environment necessary for simulating hardware models written in VHDL. These models can be tested by writing test bench models which can themselves be described in VHDL. Some approaches to writing test bench models are also described in this chapter.

## *11.1 Simulation*

Before beginning to model hardware, the first thing to decide is what type of simulation will be performed, that is, what values would be used during simulation. Would all objects (signals and variables) take values '0' or *'V* only, or would they take values '0', *'V, 'V',* or *'Z',* or use a 46-value logic? The number of values used durihg simulation will define the basic types that will be used to model hardware. This important decision has to be made prior to writing any model, since the predefined types that the language provides are not sufficient to model values such as *"U'* (undefined) and 'Z' (high-impedance). The language, however, does provide the capability of creating user-defined types that can be used to model the values required for a particular simulation. In certain cases, it may be sufficient to model hardware using the predefined types of the language, for example, when modeling hardware at an abstract level where all data may be represented strictly as integers.

For the rest of this chapter and for the chapter following this, we shall assume that we are interested in performing a four-value simulation, 'U' to represent undefined, '0' to represent logic 0, *'Y* to represent logic I, and 'Z' to represent high-impedance. The basic types are, therefore, defined as:

```
type MVL is ('U', '0', '1', Z');
type MVL_VECTOR is array (NATURAL range <>) of MVL;
```

The leftmost value of the MVL type is defined to be a 'U' so that all objects that are defined to be of this base type will have an initial value of 'U' at start of simulation. "This accurately models real hardware.

These new basic types are still insufficient to model hardware. What is lacking is a set of operations that can operate on these types. Most of the operators in the language are defined for the predefined types only. Therefore, to allow operations on these new types, we need to provide overloaded operator functions for all the language operators that are needed to operate on these types. Often, it is useful to put all such information into a package. The following is an example of such a package, called ATT_PACKAGE.

```
package ATT_PACKAGE is
        type MVL is ('U', '0', '1', 'Z');
        type MVL_VECTOR is array (NATURAL range <>) of MVL;

        type MVL_1D_TABLE is array (MVL) of MVL;
        type MVL_2D_TABLE is array (MVL, MVL) of MVL;

        -- Truth tables for logical operators:
        constant TABLE_AND: MVL_2D_TABLE :=
                ( (  'U', '0', 'U', 'U'),
                  (  '0',  '0', '0', '0'),
                  (  'U', '0', '1',  'U'),
                  (  'U', '0', 'U', 'U'));
        constant TABLE_OR: MVL_2D_TABLE :=
                ( (  'U', 'U', '1', 'U'),
                  (  'U', '0', '1',  'U'),
                  (  '1',  '1', '1', '1'),
                  (  'U', 'U', '1', 'U'));
        -- and so on.
        -- Overloaded operator declarations on MVL type:
        function "and" (L, R: MVL) return MVL;
        function "or"    (L, R: MVL) return MVL;
        -- and so on.
        -- Overloaded operators for MVL_VECTOR type:
        function "and" (L, R: MVL_VECTOR) return MVL_VECTOR;
        function "or"   (L, R: MVL_VECTOR) return MVL_VECTOR;
        function "+"    (L, R: MVL_VECTOR) return MVL_VECTOR;
        -- and so on.
end ATT_PACKAGE;

package body ATT_PACKAGE is
        -- Function definitions for overloaded operators with MVL types:
        function "and" (L, R: MVL) return MVL is
        begin
                return TABLE_AND(L, R);
        end "and";
        -- Functions for other overloaded operators can be similarly defined.
        -- Function definitions for MVL_VECTOR types:
        function "or" (L, R: MVL_VECTOR) return MVL_VECTOR is
                variable RESULT: MVL_VECTOR(L'LENGTH-1 downto 0);
        begin
                assert L'LENGTH = R'LENGTH;
                for K in RESULT'RANGE loop
                        RESULT(K) := TABLE_OR (L(K), R(K));
                end loop;
                return RESULT;
        end "or";
        -- Functions for other overloaded operators can be similarly defined.
end ATT_PACKAGE;
```

The contents of package ATT_PACKAGE can now be made visible in other design units by using the library and use clauses. But first, the package must be compiled into a design library, say ATTLIB.

If a purely behavioral model is being simulated, the previous package might be sufficient. If structural models are to be simulated that reference primitive components defined in a certain library, it is useful to provide a package containing the component declarations for all components that reside in the library. There might be a separate package provided for each class of components, for example, a package that contains the component declarations for CMOS components, a package for all TTL 7400 series components, a package for ECL components, and so on. This would eliminate the need for writing component declarations in every structural description; all that would be necessary is to reference the component declarations package by using a use clause. Here is a template for such a package.

```
library ATTLIB;
use ATTLIB.ATT_PACKAGE.all;
package CMOS_COMP is
        component AOI21
```

```
		port (A1, A2, B: in MVL; Z: out MVL);
	end component;
	component FD1S3AX
		port (D, CKA, CKB: in MVL; Q, QN: out MVL);
	end component;
	component INRB
		port (A: in MVL; Z: out MVL);
	end component;
	-- Other components would be defined similarly.
	end CMOS_COMP;
```

This package also has to be compiled into a design library, say CMOSLIB.

There is yet another piece of information that is required before a structural model can be simulated. These are the behavioral models for any primitive components used in the structural model. If the primitive components as defined in the CMOS_COMP package are used, it is necessary to provide behavioral models for these primitive components. Three such examples are shown next.

```
library ATTLIB;
use ATTUB.ATT_PACKAGE.all;
entity A0121 is
	port (A1, A2, B: in MVL; Z: out MVL);
end AOI21;

architecture BEH_MODEL of AOI21 is
begin
	process (A1, A2, B)
		variable TEMP: MVL;
	begin
		TEMP := TABLE_AND(A1, A2);
		Z <= TABLE_NOR (TEMP, B);
	end process;
end BEH_MODEL;

library ATTLIB;
use ATTUB.ATT_PACKAGE.all;
entity FD1S3AX is
	port (D, CKA, CKB: in MVL; Q, QN: out MVL);
end FD1S3AX;

architecture BEH_MODEL of FD1S3AX is
begin
	process (D, CKA, CKB)
		variable TEMP: MVL;
	begin
		if (CKA = '1' and CKA'EVENT) then
			TEMP := TABLE_BUF (D);
		end if;
		Q <= TABLE_BUF (TEMP);
		QN <= TABLE_NOT (TEMP);
	end process;
end BEH_MODEL;

library ATTLIB;
use ATTUB.ATT_PACKAGE.all;
entity INRB is
	port (A: in MVL; Z: out MVL);
end INRB;

architecture BEH_MODEL of INRB is
begin
	Z <= TABLE_NOT (A);
end BEH_MODEL;
```

The behavioral descriptions for all the primitive library components also need to be compiled into a design library, say CMOSLIB. At this point, we are ready to begin modeling and simulating hardware.

## 11.2 Writing a Test Bench

A test bench is a model that is used to exercise and verify the correctness of a hardware model. The expressive power of the VHDL language provides us with the capability of writing test bench models also in the same language. A test bench has three main purposes:

1. to *generate* stimulus for simulation (waveforms),
2. to *apply* this stimulus to the entity under test and to monitor the output responses,
3. to *compare* output responses with expected known values.

Again, the language provides a large number of ways to write a test bench. In this section, we explore only some of these. A typical format of a test bench that drives an entity under test is

```
entity TEST_BENCH is
end;

architecture TB_BEHAVIOR of TEST_BENCH is
        component ENTITY_UNDER_TEST
                port ( list-of-ports-their-types-and-modes);
        end component;
        Local-signal-declarations;
begin
        Generate-waveforms-using-behavioral-constructs;
        Apply-to-entity-under-test;
        EUT: ENTITY_UNDER_TEST port map ( port-associations );
        Monitor-values-and-compare-with-expected-values;
end TB_BEHAVIOR;
```

The application of stimulus to the entity under test is accomplished automatically by instantiating the entity in the test bench model and then specifying the appropriate interface signals. The next two subsections look at waveform generation and output response monitoring.

### 11.2.1 Waveform Generation

There are two main approaches in generating stimulus values:

1. create waveforms and apply stimulus at certain discrete time intervals,
2. generate stimulus based on the state of the entity, that is, based on the output response of the entity.

Two types of waveforms are typically needed. One is a repetitive pattern, for example in a clock, and the other is a sequential set of values.

### Repetitive Patterns

A repetitive pattern with a constant on-off delay can be created using a concurrent signal assignment statement.

```
A <= not A after 20 ns;       - Signal A is assumed to be of type BIT.
```

The waveform created is shown in Fig. 11.1.

A clock with varying on-off period can be created using a process statement. The waveform created is shown in Fig. 11.1.

```
-- Signal D_CLK is assumed to be a signal of type BIT.
process
        constant OFF_PERIOD: TIME := 30 ns;
        constant ON_FERIOD : TIME := 20 ns;
begin
        wait for OFF_PERIOD;
        D_CLK <= '1';
        wait for ON_PERIOD;
        D_CLK <= '0';
end process;
```

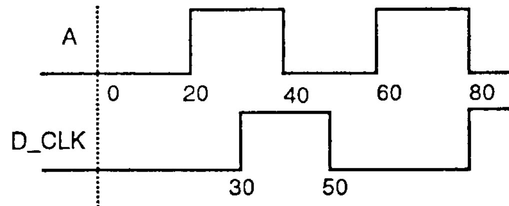



**Figure 11.1 A repetitive pattern.**

A problem with both of these approaches is that if the host environment provides no control over the amount of time simulation is to be performed, simulation would go on until time is equal to TIME'HIGH causing a lot of simulation time to be wasted. One way to avoid this is to put an assertion statement in the process that will cause the assertion to fail after a specific time. Here is such an example of an assertion statement.

```
assert (NOW <= 1000 ns)
        report "Simulation completed successfully"
        severity ERROR;
```

Assertion would fail when simulation time exceeds 1000 ns and simulation would stop (assuming that the simulator stops on getting a severity level of ERROR).

An alternate way to generate a clock with a varying on-off period is by using a conditional signal assignment statement. Here is an example.

```
D_CLK    <=    '1' after OFF_PERIOD when D_CLK = '0' else
               '0' after ON_PERIOD;
-- D_CLK is a signal of type BIT.
-- OFF_PERIOD and ON_PERIOD are constants defined elsewhere.
```

A clock that is phase delayed from another clock can be generated by using the DELAYED predefined attribute. For example,

```
DLY_D_CLK <= D_CLK'DELAYED (10 ns);
```

The waveforms for D_CLK and DLY_D_CLK are shown in Fig. 11.2.

Two monoverlapping clocks can be generated in a similar manner by making sure that the time specified for the DELAYED attribute is larger than the clock's on-period.

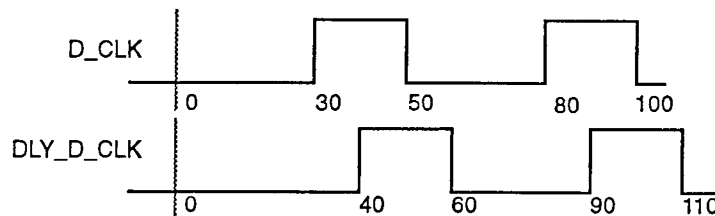



**Figure 11.2A phase-delayed clock.**

A sequential set of values can also be generated for a signal by using multiple waveforms in a concurrent signal assignment statement. For example,

```
RESET <= '0', '1' after 100 ns, '0' after 180 ns, '1' after 210 ns;
```

The waveform generated on signal RESET is shown in Fig. 11.3.

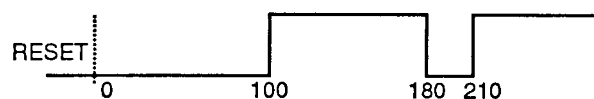



**Figure 11.3 A nonrepetitive waveform.**

This waveform can be made to repeat itself by placing the signal assignment statement inside a process statement along with a wait statement. The following example of a two-phase clock shows such a repeated waveform. Figure 11.4 shows the waveforms created.

```
signal CLK1, CLK2: MVL := '0';
. . .
TWO_PHASE: process
begin
        CLK1 <= 'U' after 5 ns, '1 ' after 10 ns, 'U' after 20 ns,
```

```
            '0' after 25 ns;
        CLK2 <= 'U' after 10 ns, '1' after 20 ns, 'U' after 25 ns,
            '0' after 30 ns;
        wait for 35 ns;
end process;
```

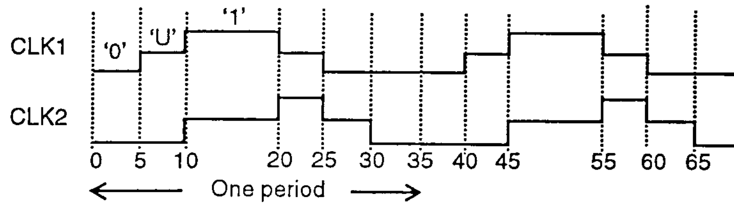



**Figure 11.4 A complex repetitive waveform.**

## Using Vectors

Another way to apply stimulus to a set of signals is to store the set of vectors in a constant table or in an ASCII file. Here is an example that stores the input vectors in a table.

```
constant NO_OF_BITS: INTEGER := 4;
constant NO_OF_VECTORS: INTEGER := 5;
type TABLE_TYPE is array (1 to NO_OF_VECTORS) of
                        MVL_VECTOR(1 to NO_OF_BITS);
constant VECTOR_PERIOD: TIME := 100 ns;
constant INPUT_VECTORS: TABLE_TYPE :=
                        ("1001", "1000", "0010", "0000", "0110");
signal INPUTS: MVL_VECTOR(1 to NO_OF_BITS);
signal A, B, C: MVL;
signal D: MVL_VECTOR(0 to 1);
```

Assume that the entity under test has four inputs. A, B, C, and D. If the vectors are to be applied at regular time intervals, a generate statement can be used as shown.

```
G1: for J in 1 to NO_OF_VECTORS generate
        INPUTS <= INPUT_VECTORS(J) after (VECTOR_PERIOD * J);
end generate G1;
A<= INPUTS(1);
B <= INPUTS(4);
C<=INPUTS(1);
D<=INPUTS(2 to 3);
```

If the vectors were to be applied at arbitrary intervals, a concurrent signal assignment statement with multiple waveforms can be used.

```
INPUTS    <=    INPUT_VECTORS(1) after 10 ns,
                INPUT_VECTORS(2) after 25 ns,
                INPUT_VECTORS(3) after 30 ns,
                INPUT_VECTORS(4) after 32 ns,
                INPUT_VECTORS(5) after 40 ns;
```

Using this approach, it is possible to assign one vector column to more than one signal.

The input vectors could also be specified in an ASCII file as a sequence of values. The following statements define a file and read the vectors from the file.

```
process
        type VEC_TYPE is file of MVL.VECTOR;
        file VEC_FILE: VEC_TYPE is in "/usr/jb/EXAMPLE.vec";
        variable LENGTH: INTEGER;
        variable IN_VECTOR: MVL_VECTOR(1 to 4);
begin
        LENGTH := 4;     - The number of bits to be read.
        while (not ENDFILE(VEC_FILE)) loop
            READ (VEC_FILE, tN_VECTOR, LENGTH);
```

```
            - It is necessary to specify the LENGTH of the vector to be read
            - since the file contains values of an unconstrained array type.
            :
            :
        end loop;
    end process;
```

A complete test bench that uses the waveform application method is shown next. The entity being simulated is called DIV. The output value is written into a file for later comparison.

```
library ATTLIB;
use ATTLIB.ATT_PACKAGE.all;
entity DIV_TB is end;

architecture DIV_TB_BEH of DIV_TB is
        component DIV
                port (CK, RESET, TESTN: in MVL: ENA: out MVL);
        end component;
        signal CLOCK, RESET, TESTN, ENABLE: MVL;
        type VEC_TYPE is file of MVL.VECTOR;
        file OUTFILE: VEC_TYPE is out "/usr1/jb/div.vec.out";
        for D1: DIV use entity WORK.DIV;
begin
        CKP: process
        begin
                CLOCK <= '0';
                wait for3 ns;
                CLOCK <= '1';
                wait for 5 ns;
                assert (NOW < 900 ns)
                        report "Simulation completed successfully.";
        end process CKP;

        RESET <= '0', '1' after 110ns;
        TESTN <= '0', '1' after 160 ns, '0' after 670 ns;
        -- Apply to entity under test:
        D1: DIV port map (CLOCK, RESET, TESTN, ENABLE);

        -- For every event on the ENABLE output signal, write to file.
        MONITOR: process (ENABLE)
        begin
                WRITE (OUTFILE, ENABLE);
        end process MONITOR;
end DIV_TB_BEH;
```

In the second approach for stimulus generation, the stimulus value generated is based on the state of the entity under test. This approach is useful in testing a finite state machine for which different input stimulus is applied based on the machine's state. Consider an entity in which the objective is to compute the factorial of an input number. The handshake mechanism between the entity under test and the test bench model is shown in Fig. 11.5.

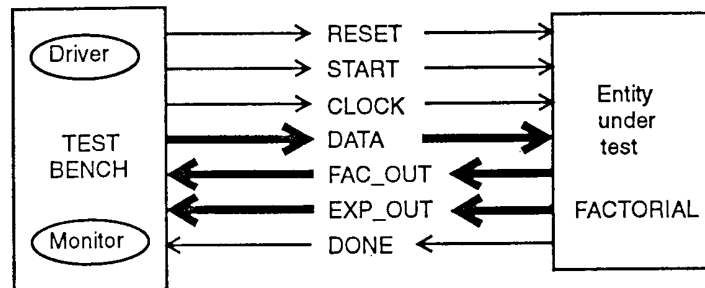



**Figure 11.5 Handshake between test bench and entity undertest.**

The RESET input to the entity resets the factorial model to an initial state. The START signal is set after the DATA input is applied. When computation is complete, the output DONE signal is set to indicate that the computed result appears on the FAC_OUT and EXP_OUT outputs. "The resulting factorial value is [ FAC.OUT * 2^EXP_OUT ]. The test bench model provides input data on signal DATA starting from values 1 to 20 in

increments of one. It applies the data, sets the START signal, waits for DONE signal, and then applies the next input data. Assertion statements are used to make sure that the values appearing at the output are correct. The test bench description follows.

```
library ATTLIB; use ATTUB.ATT_PACKAGE.all;
entity FAC_TB is
        constant IN_MAX: INTEGER := 5;
        constant OUT_MAX: INTEGER := 8;
end FAC_TB;

architecture FAC_TB_FUNC of FAC_TB is
        component FACTORIAL
                port (RESET, START, CLOCK: in MVL;
                DATA: in MVL_VECTOR(IN_MAX-1 downto 0);
                DONE: out MVL;
                FOUT, EOUT: out MVL_VECTOR(OUT_MAX-1 downto 0));
        end component;
        type FAC_STATE is (RESET_ST, START_ST,
                        APPL_DATA_ST, WAIT_RESULT_ST);
        signal CLK, RESET, START, DONE: MVL;
        signal DATA: MVL_VECTOR(7 downto 0) := (others => '0');
        signal FAC_OUT, EXP_OUT:
                        MVL_VECTOR(OUT_MAX-1 downto 0);
        signal NEXT_STATE: FAC_STATE; - Starts with RESET_ST state.
        constant MAX_APPLY: POSITIVE := 20;
begin
        CLK_P: process
        begin
                CLK <= '0';
                wait for 4 ns;
                CLK <= '1';
                wait for 6 ns;
        end process CLK_P;

        process
                variable NUM_APPLIED: POSITIVE; - Starting with 1.
        begin
                if ((CLK = '0') and CLK'EVENT) then - Falling edge transition.
                        case NEXT_STATE is
                                when RESET_ST =>
                                        RESET <= '1'; START <= '0';
                                        NEXT_STATE <= APPL_DATA_ST;
                                when APPL_DATA_ST =>
                                        DATA    <=    INT2MVL(NUM_APPLIED,
                                        IN_MAX);
                                        NEXT_STATE <= START_ST;
                                when START_ST =>
                                        START <= '1';
                                        NEXT_STATE <= WAIT_RESULT_ST;
                                when WAIT_RESULT_ST =>
                                        RESET <= '0', START <= '0';
                                        wait until (DONE = '1');
                                        assert (NUM_APPLIED =
                                        MVL21NT(FAC_OUT) *
                                                (2 ** MVL21NT(EXP_OUT)))
                                                report "Incorrect result from
                                                factorial model.";
                                        NUM_APPLIED := NUM_APPLIED +1;
                                        if (NUM_APPLIED < MAX_APPLY) then
                                                NEXT_STATE              <=
                                                APPL_DATA_ST;
                                        else
                                                assert    FALSE    -    Stop
                                        simulation.
```

```
                                        report 'Test
                                        completed
                                        successfully.";
                                end if;
                        end case;
                end if;
        end process;

        -- Apply to entity under test:
        F1: FACTORIAL port map (RESET, START, CLK, DATA,
                DONE, FAC_OUT, EXP_OUT);
end FAC_TB_FUNC;
-- Functions MVL21NT and INT2MVL are assumed to be
-- defined in package ATT_PACKAGE.
```

### 11.2.2 Monitoring Behavior

In the test bench model for the FACTORIAL entity, we saw how the test bench can monitor the behavior of the entity and apply different patterns based on the output response. Another common way to monitor the output response of an entity is to apply a vector, sample the output after a specific time and then verify to make sure that the output response matches the expected values. Here is an example of such a test bench. The input and output vectors are stored in tables. Alternately, they could have been read from an ASCII file.

```
library ATTLIB;
use ATTUB.ATT_PACKAGE.all;
entity ANOTHER_DIV_TB is end;

architecture APPLY_AND_SAMPLE of ANOTHER_DIV_TB is
        component DIV
                port (CK, RESET, TESTN: in MVL; ENA: out MVL);
        end component;
        type MVL3 is array (1 to 3) of MVL;
        type MVL2_VECTOR is array (POSITIVE range <>,
                POSITIVE range <>) of MVL;
        constant INPUT_VECTORS: MVL2_VECTOR :=
                ("100", "100", "100", "100", "110", "111", "011");
        constant OUTPUT_VECTORS: MVL2_VECTOR :=
                ("0", "0", "0", "0", "1");
        constant STROBE_DELAY: TIME := 1 ns;
        constant CYCLE_TIME: TIME := 80 ns;
        signal CLOCK, RESET, TESTN, ENABLE: MVL;
begin
        APPLY: process
        begin
                for J in 1 to INPUT_VECTORS'LENGTH(1) loop
                        P_1 (DRIVE_SIGNAL=>CLOCK,
                                SIGNAL_VALUE=>INPUT_VECTORS(J,1),
                                DRIVE_DELAY=>20 ns, DRIVE_WIDTH=>30 ns);
                        -- Procedure P_1 generates a clock of specified
                        -- delay & width.
                        RESET <= INPUT_VECTORS(J, 2);
                        TESTN <= INPUT_VECTORS(J, 3);
                        wait for (CYCLE_TIME- STROBE_DELAY);
                        assert ENABLE = OUTPUT_VECTORS(J, 1);
                        wait for STROBE_DELAY;
                end loop;
        end process APPLY;
        --Entity under test:
        D2: DIV port map (CLOCK, RESET, TESTN, ENABLE);
end APPLY_AND_SAMPLE;
```

In this test bench example, the application of test vectors is expressed as a process. After a test vector is applied, the process suspends for time (CYCLE_TIME - STROBE_DELAY), samples the output and then checks to see if the expected output is equal to the specified output vector. The process then suspends for STROBE_DELAY time before reapplying the next vector.

# CHAPTER 12 *Hardware Modeling Examples*

This chapter describes a number of examples of hardware models using VHDL. As we have seen earlier, there is more than one way to model a particular entity using this language. In this chapter, we present one such approach for each entity being modeled. We shall be using the type MVL and the ATTJPACKAGE package, discussed in the previous chapter, for all the hardware models that are described in this chapter. The complete description of the ATT_PACKAGE package appears in Appendix C.

## *12.1 Modeling Entity Interface*

The interface ports of an entity are specified using the entity declaration. The ports specify the names of signals that interact with the external environment of the entity. The semantics of each interface port is derived from the mode and type of the port. Here is an example of the external interface for an or-and-invert entity.

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity OA132 is
        port (A1, A2, A3, B1, B2: in MVL; Z: out MVL);
end OA132;
```

This entity declaration specifies that the entity OAI32, has five input ports and one output port, all of type MVL, that is, the ports can have values 'U', '0', '1' or 'Z'. An example of an entity interface for a 4-bit counter is shown next.

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity COUNTER4 is
        port (COUNT. CLOCK: in MVL;
                Q: out MVL_VECTOR(3 downto 0));
end COUNTER4;
```

This entity declaration declares two 1-bit inputs and a 4-bit output for entity COUNTER4. Its external view is shown in Fig. 12.1.

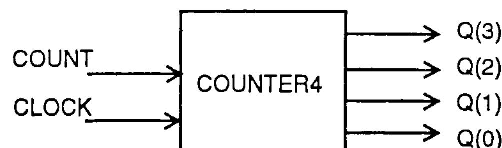



**Figure 12.1 An external view of a 4-bit counter.**

## *12.2 Modeling Simple Elements*

A basic hardware element is a wire. A wire can be modeled in VHDL as a signal. Consider a 4-bit and gate whose behavior is described next.

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
```

```
entity AND4 is end;

architecture AND4_DF of AND4 is
        signal A, B, C: MVL_VECTOR(3 downto 0);
begin
        A <= B and C after 5 ns;
end AND4_DF;
```

The gate delay for the and gate is specified to be 5 ns. An interesting point to note is that the previous architecture body is a legal VHDL description in spite of the fact that there are no inputs and outputs of the entity. The hardware represented by this behavior is shown in Fig. 12.2.

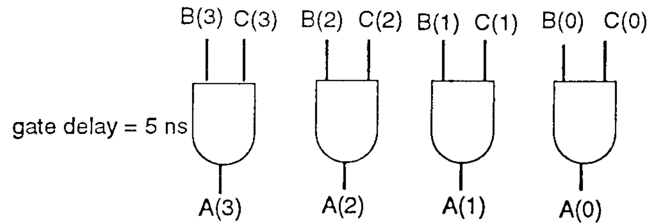



**Figure 12.2 A 4-bit and gate.**

This example and the one following show that boolean equations can be modeled as expressions in concurrent signal assignment statements. Wires can be modeled as signal objects. For example, in the following description, signal F represents a wire that connects the output of the not operator to the input of the xor operator. The circuit represented by the architecture is shown in Fig. 123.

```
architecture BOOLEAN_EX of B_EX is
        signal D, E, F, G: MVL;
begin
        D <= F xor G;
        F <= not E;
end BOOLEAN_EX;
```

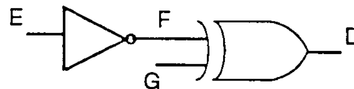



**Figure 12.3 A combinational circuit.**

Consider the following behavior and its corresponding hardware representation shown in Fig. 12.4.

```
architecture LOOP1 of ASYNCHRONOUS is
        signal A, B, C, D: MVL;
begin
        C <= A or D;
        A<= B nand C;
end LOOP1;
```

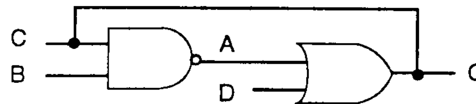



**Figure 12.4 An asynchronous loop.**

This circuit has an asynchronous loop and if the model were simulated with a certain set of signal values (B=T, C=T, D='0'), simulation time would never advance since the simulator would always be iterating between the two signal assignments. The iteration time would be two delta delays. Therefore, extra caution must be exercised when values are assigned to signals and when these same signals are used in expressions.

In certain cases, it is desirable to have such an asynchronous loop. An example of such an asynchronous loop is shown next; the statement represents a periodic waveform with a cycle of 20 ns. Its hardware representation is shown in Fig. 12.5.

```
A <= not A after 10 ns;
```

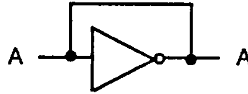



**Figure 12.5 A ciock generator.**

Elements of a vector signal can also be accessed, either as a single element or as a slice. For example,

```
signal A: MVL;
signal C: MVL_VECTOR(0 to 4);
signal B, D: MVL_VECTOR(5 downto 0);
 . . .
D(4 downto 0) <= B(5 downto 1) or C;
D(5)<=AandB(5);
```

The first signal assignment implies D(4) <= B(5) or C(0), D(3) <= B(4) or C(1), and so on.

Single bit signals and vector signals can be concatenated to form larger vector signals. For example,

```
signal C, CC: MVL_VECTOR(7 downto 0);
signal CX: MVL;
 . . .
C <= CX & CC(6 downto 0);
```

It is also possible to refer to an element of a vector signal whose index value is computable only at simulation run time. For example,

```
A<= B(K);
```

implies a decoder whose output is signal A, and K specifies the selection address. B is a vector of values of the same type as signal A. The object B models the behavior of the decoder.

Shift operations are easily modeled using the concatenation operator. For example,

```
signal A, Z: MVL_VECTOR(0 to 7):
 . . .
Z <= A(1 to 7) & A(0);                -- A left rotate operation.
Z <= A(7) & A(0 to 6):                -- A right rotate operation.
Z <= A(1 to 7) & '0';                 -- A left shift operation.
Z <= A(K to 7) & A(0 to (K-1 ));      -- A k-left rotate operation.
```

Subfields of a vector signal, called slices, can also be accessed. For example, consider a 32-bit instruction register, INSTR_REG, in which the first 16 bits denote address, next 8 bits represent opcode, and the remaining 8 bits represent the index. Given the following declarations:

```
type MEMORY_TYPE is array (0 to 1023, 31 downto 0) of MVL;
signal INSTR_REG: MVL_VECTOR(31 downto 0):
signal ADDRESS: MVL_VECTOR(15 downto 0):
signal OP_CODE, INDEX: MVL_VECTOR(7 downto 0);
signal MEMORY: MEMORY_TYPE;
signal PROG_CTR: MVL_VECTOR(0 to 9);
```

then one way to read the subfield information from the INSTR_REG is to use three concurrent signal assignment statements. The slices of the instruction register are assigned to specific signals.

```
INSTR_REG <= MEMORY (MVL21NT(PROG_CTR));
-- MVL21NT is a function that translates MVL_VECTOR value to integer.
ADDRESS <= INSTR_REG(15 downto 0);
OP_CODE <= INSTR_REG(23 downto 16);
INDEX <= INSTR_REG(31 downto 24);
 . . .
PROCEDURE_CALL(ADDRESS, OP_CODE, INDEX);
```

An alternate way is to declare the subfields of the instruction register as aliases instead of as separate signals.

```
alias ADDRESS: MVL_VECTOR(15 downto 0) is
        INSTR_REG(15 downto 0);
alias OP_CODE: MVL_VECTOR(7 downto 0) is
        INSTR_REG(23 downto 16);
```

```
alias INDEX: MVL_VECTOR(7 downto 0) is
        INSTR_REG(31 downto 24);
```

In this case, the explicit assignments to the three subfields are not necessary.

```
INSTR_REG <= MEMORY (MVL21NT(PROG_CTR));
. . .
PROCEDURE_CALL (ADDRESS, OP_CODE, INDEX);
```

## 12.3 Different Styles of Modeling

This section gives examples of the three different modeling styles provided by the language: dataflow, behavioral, and structural. Consider the circuit shown in Fig. 12.6 that saves the value of the input A into a register and then multiplies it with input C.

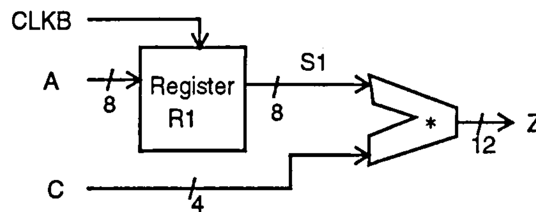



**Figure 12.6 A buffered multiplier.**

The entity declaration that declares the interface signals for the circuit is

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity SAVE_MULT is
        port (A: in MVL_VECTOR(0 to 7); C: in MVL_VECTOR(0 to 3);
                CLKB: in MVL; Z: out MVL_VECTOR(0 to 11));
end SAVE_MULT;
```

The first modeling style is the dataflow style in which concurrent signal assignment statements are used to model the circuit.

```
architecture DATA_FLOW of SAVE_MULT is
        signal S1: MVL_VECTOR(0 to 7);
begin
        Z <= S1 * C;        -- The multiplication operator is overloaded to
                            -- operate on MVL_VECTORs of different sizes.
        R1:block   (CLKB= '1' and (not CLKB'STABLE))
        begin
                S1 <= guarded A;
        end block R1;
end DATA_FLOW;
```

This representation does not ditectly imply any structure, but implicitly describes it. However, its functionality is very clear. The flip-flop has been modeled using a block statement.

The second way to describe the circuit is to model it as a sequential process.

```
architecture SEQUENTIAL of SAVE_MULT is
        signal S1:MVL_VECTOR(0 to 7);
begin
        process (A,C,CLKB)
        begin
                if (CLKB='1' and (not CLKB'STABLE)) then
                        S1<= A;
                end if;
                Z <= S1*C;
        end process;
end SEQUENTIAL;
```

This model also describes the behavior but does not imply any structure , explicitly or implicitly. In this case, the flip-flop has been modeled using an if statement.
The third way to describe the SAVE_MULT circuit is to model it as a netlist assuming the existence of an 8-bit register and an 8-bit multiplier.

```
architecture NET_LIST of SAVE_MULT is
        component REG8
                port (DIN: in MVL_VECTOR( 0 to 7); CLK: in MVL;
                        DOUT: out MVL_VECTOR(0 to 7));
        end component;
        component MULT8
                port (A,B: in MVL_VECTOR(0 to 7);
                        Z: out MVL_VECTOR(0 to 15));
        end component;
        signal S1,S3: MVL_VECTOR(0 to 7);
        signal S2: MVL_VECTOR(0 to 15);
begin
        R1: REG8 port map (A,CLKB,S1);
        M1: MULT8 port map (S1,S3,S2);
        Z<= S2(4 to 15);
        S3 <= "0000" & C;   -- This assignment is necessary since
                -- an expression cannot be specified directly in a port map.
end NET_LIST;
```

This description explicitly describes the structure but the behavior is unknown. This is because the REGS and MULT8 component names are arbitrary and they could have any behavior associated with them. The signal assignment to Z is necessary since the multiplier was assumed to produce a 16-bit output

Of these three different modeling styles, the behavioral style of modeling is generally the fastest to simulate, if simulation time is critical.

## 12.4 Modeling Regular Structures

Let us assume that we want to construct a IO-deep 8-bit stack using 8-bit registers. The stack has an input port and an output port and a control signal that specifies whether to pop or to push data. Overflow and underflow conditions are ignored. Ten 8-bit registers model the stack. The input to each register is connected from either the top register or from the bottom register depending on whether to push or pop. Since this basically involves a replication of the register 10 times, a generate statement can be used to model the stack. Here is the model.

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity STACK is
        port (DATAIN: in MVL_VECTOR(7 downto 0);
                DATAOUT: out MVL_VECTOR(7 downto 0);
                CLK, STCK_CTRL: in MVL);
end STACK;

architecture GENERATE_EX of STACK is
        type STACK_TYPE is array (1 to 10) of
                        MVL_VECTOR(7 downto 0);
        signal REGIN: MVL_VECTOR(7 downto 0);
        signal DATA: STACK_TYPE;
        component REGS
                port (DIN: in MVL_VECTOR(0 to 7);
                        DOUT: out MVL_VECTOR(0 to 7); CLK: in MVL);
        end component;
begin
        G1: for K in 1 to 10 generate - 10 is top of stack, 1 is bottom.
                G2: if (K = 10) generate
                        REGIN<= DATAIN when STCK_CTRL= '1'
                                else DATA(K-1);
                end generate G2;
                G3: if K > 1 and K < 10 generate
                        REGIN <= DATA(K+1) when STCK_CTRL = '1' - PUSH
                                else DATA(K-1);             - POP
                end generate,G3;
```

```
            G4: if K = 1 generate REGIN <= DATA(K+1) when STCK_CTRL=
    '1'
                    else (others=>'U');
            end generate G4;
            FF8: REGS port map (REGIN, DATA(K), CLK);
    end generate G1;
    DATAOUT <= DATA(10);            --Output is connected to
                                    -- top of stack.
end GENERATE_EX;
```

The generate statement is expanded by unrolling the loop during elaboration time. When variable K has a value of I, the input to the bottom-most register is 'U', if the stack is being popped; when K is 10, the input to the topmost register is the input to the stack, DATAIN. The output of the top-most register is the output of the stack.

## *12.5 Modeling Delays*

Consider a 3-input nor gate. Its behavior can be modeled using a concurrent signal assignment statement such as

```
-A, B,C and Z are signals of type MVL.
Z <= not (A or B or C) after 12 ns:
```

This statement models the nor gate with an inertial delay of 12 ns. This delay represents the time from when an event occurs on signal A, B, or C until the result value appears on signal Z. An event could be any value change, for example, 'U'->'Z', 'U'->'0', or '1'->'0'.

If the rise time, that is, the transition time for '0'->'l', and the fall time, that is, the transition time for '1'->'0', were to be explicitly modeled, a different mechanism is needed. Consider the following conditional signal assignment.

```
signal Z, A: BIT;
 . . .
Z   <=   not A after 12 ns when A = '1 ' else
         not A after 14ns;        - when A = '0';
```

If an event occurs on signal A (it can only be a '0'->'1' or a'1'-> '0' transition since the signal type is BIT), the inverted value of A will appear on signal Z after 12 ns if signal A had the transition '0'->'Y (12 ns is then the fall time), else Z will get its value after 14 ns if the transition on signal A is from '1'->'0' (14 ns is then the rise time). Such a model is sufficient for BIT types but is insufficient when a signal type has more than two values, for example, when using an MVL type. In such a case, the following example illustrates an approach to model rise and fall times, independently.

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity NOR3 is
        port (A, B, C: in MVL; Z: out MVL);
end NOR3;

architecture TIMES of NOR3 is
begin
        process (A, B, C)
                variable OLD_Z, NEW_Z: MVL;
                constant RISE_TIME: TIME := 8 ns;
                constant FALL_TIME: TIME := 10 ns;
        begin
                NEW_Z := not (A or B or C);
                if (OLD_Z = '0') and (NEW_Z = '1 ') then
                        Z <= NEW_Z after RISE_TIME;
                elsif (OLD_Z = '1') and (NEW_Z = '0') then
                        Z <= NEW_Z after FALL_TIME;
                else
                        Z <= NEW_Z after MAX(RISE_TIME, PALL_TIME);
                        -- Function MAX returns the maximum of the two time
                        -- values. It is declared in the package ATT_PACKAGE.
                end if;
                OLD_Z := NEW_Z;
```

```
        end process;
end TIMES;
```

There is nothing special in the previous example. The old value of signal Z is saved. When a new value for Z is computed, a check is made to see what type of transition occurred. Based on whether a rising or a falling edge occurred, the computed value is assigned to the signal Z after the appropriate delay. The rise and fall times are specified using constant declarations in this example. They could also have been declared elsewhere, for example, in an entity declaration or in a package declaration or as generics for the NOR3 entity such as

```
entity NOR3 is
        generic (RISE_TIME: TIME := 8 ns; FALL_TIME: TIME := 10ns);
        port (A, B, C; in MVL; Z: out MVL);
end NOR3;
```

## *12.6 Modeling Conditional Operations .*

Operations that occur under certain conditions can be modeled using either a selected signal assignment statement, a conditional signal assignment statement, an if statement, or a case statement. Let us consider a 3-bit decoder circuit. Its behavior can be modeled using a conditional signal assignment as shown below.

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity DECODER is
        port (CTRL: in MVL_VECTOR(0 to 2);
                Z: out MVL_VECTOR(0 to 7));
        constant DECODER_DELAY: TIME := 24 ns;
end DECODER;

architecture CONDITIONAL_EX of DECODER is
begin
        G1: for K in 0 to 7 generate
                Z(K) <= '0' after DECODER_DELAY
                                when MVL21NT(CTRL) = K else
                        '1 ' after DECODER_DELAY;
                -- MVL21NT is a function declared in the package
                -- ATT_PACKAGE; it converts an MVL_VECTOR value into
                -- an integer value.
        end generate G1;
end CONDITIONAL_EX;
```

A multiplexer can be modeled using a process statement. The value of the select lines are first determined and based on this value, a case statement selects the appropriate input that is to be assigned to the output.

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity MULTIPLEXER is
        generic (MUX_DELAY: TIME := 15ns);
        port (SEL: in MVL_VECTOR(0 to 1); A, B, C, D: in MVL;
                MUX_OUT: out MVL);
end MULTIPLEXER;

architecture PROCESS_MUX of MULTIPLEXER is
begin
        P1: process (SEL, A, B, C, D)
                variable TEMP: MVL;
        begin
                case SEL is
                        when "00" => TEMP := A;
                        when "01" => TEMP := B;
                        when "10" => TEMP := C;
                        when "11" => TEMP := D;
                end case;
                MUX_OUT <= TEMP after MUX_DELAY;
        end process P1;
end PROCESS_MUX;
```

## 12.7 Modeling Synchronous Logic

So far in this chapter, most of the examples that we have seen are that of combinational logic. As far as synchronous logic is concerned, there is no explicit object in the language that declares registers or memories. The semantics of the language, however, provides for signals to be interpreted as registers or memories depending on how the values to these signals are assigned. There are again a large number of ways in which synchronous logic can be modeled. Some of these are

- by controlling the signal assignment,
- by using a guarded assignment,
- by using a register kind signal.

Consider the following example that shows how controlling a signal assignment can model a synchronous, edge-triggered D-type flip-flop.

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity D_FLIP_FLOP is
        port (D, CLOCK: in MVL; Q: out MVL);
end D_PLIP_PLOP:

architecture SYNCHRONOUS of D_PLIP_PLOP is
begin
        process (CLOCK)
        begin
                if ((CLOCK = '1') and CLOCK'EVENT) then - Rising edge.
                - Alternately "if ((CLOCK = '1') and
                - (not CLOCK'STABLE)) then".
                        Q <= D after 5 ns;
                end if;
        end process;
end SYNCHRONOUS:
```

The semantics of the process statement indicates that when there is a rising edge on signal CLOCK, Q will get the value of D after 5 ns, else the value of signal Q does not change (a signal retains its value until it is assigned a new value). Even though Q and D are signals, the behavior in the process statement expresses the semantics of a D-type flip-flop. Given this entity, an 8-bit register can be modeled as follows:

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity REGISTERS is
        generic (START: INTEGER := 0; STOP: INTEGER := 7);
        port (D: in MVL_VECTOR(START to STOP);
                Q: out MVL_VECTOR(START to STOP); CLOCK: in MVL);
end REGISTERS;

architecture COMPONENT_BEH of REGISTERS is
        component D_PLIP_PLOP
                port (D, CLOCK: in MVL; Q: out MVL);
        end component;
begin
        G_LABEL: for K in START to STOP generate
                DPP: D_PLIP_FLOP port map (D(K), CLOCK, Q(K));
                -- This component instantiation statement could very well have
                -- been replaced by the process statement that appears
                -- in the architecture body of the D_PLIP_PLOP entity.
        end generate G_LABEL;
end COMPONENT_BEH;
```

Consider a gated cross-coupled latch circuit shown in Fig. 12.7 and its dataflow model.

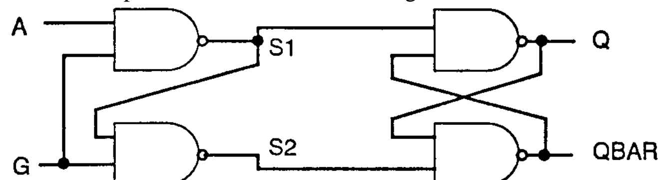



**Figure 12.7 A gated latch.**

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity GATED_PP is
        port (A, G: in MVL; Q, QBAR: buffer MVL);
end GATED_FP;

architecture IMPLICIT_LATCH of GATED_PF is
        signal S1, S2: MVL;
begin
        S1 <= A nand G;
        S2 <= S1 nand G;
        Q <=QBAR nand S1;
        QBAR <= Q nand S2;
end IMPLICIT_LATCH;
```

In this example, the concurrent signal assignment statements do not have any conditions associated with them but the semantics of the behavior implies a latch. Note that the outputs of the gated flip-flop are of mode buffer since they are being assigned to and being read within the architecture body.

A memory can be modeled as a 2-dimensional array of flip-flops. Consider a generic RAM model in which each memory element is a GATED_FF component that was described earlier. ASIZE is the number of bits on the address port and DSIZE is the number of bits on the data port of the RAM.

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity RAM_GENERIC is
        generic (ASIZE, DSIZE: INTEGER);
        port (ADDRESS: in MVL_VECTOR(ASIZE-1 downto 0);
                DATA_IN: in MVL_VECTOR(DSIZE-1 downto 0);
                DATA_OUT: buffer MVL_VECTOR(DSIZE-1 downto 0));
end RAM_GENERIC;

architecture COMPONENT_BEH of RAM_GENERIC is
        component GATED_FF
                port (A, G: in MVL; Q, QBAR: buffer MVL := '0');
        end component;
        component DECODER
                generic (SELECT_SIZE: INTEGER);
                port (ADDR: out
                        MVL_VECTOR(2**SELECT_SIZE-1 downto 0);
                SEL_CTRL: in
                        MVL_VECTOR(SELECT_SIZE-1 downto 0));
        end component;
        signal ADDR_SELECT: MVL_VECTOR(2**ASIZE-1 downto 0);
begin
        D1: DECODER generic map (ASIZE)
                port map (ADDR_SELECT, ADDRESS);
        L1: for J in 0 to 2**ASIZE-1 generate
                L2: for K in 0 to DSIZE-1 generate
                        GFF: GATED_FF port map (DATA_IN(K),
                                ADDR_SELECT(J), DATA_OUT(K), open);
                end generate L2;
        end generate L1;
end COMPONENT_BEH;
```

The keyword **open** in the instantiation of the gated latch indicates that the QBAR port of the gated latch is not connected to any other signal.

Synchronous logic can also be modeled using guarded assignments. For example, a level-sensitive D flip-flop can be modeled as

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity LEVEL_SENS_DFF is
        port (STROBE, D: in MVL; Q, QBAR: out MVL);
end LEVEL_SENS_DFF;

architecture GUARDED_BEH of LEVEL_SENS_DFF is
        signal TEMP; MVL;
begin
```

```
B1: block (STROBE = '1') - STROBE is the name of the clock input.
begin
        TEMP <= guarded D;
        Q <= TEMP;
        QBAR <= not TEMP;
end block B1;
end GUARDED_BEH;
```

When STROBE is '1', any events on signal D are transferred to TEMP and eventually to Q, but when STROBE becomes '0', the value in TEMP (also in Q and QBAR) is retained and any change in input D no longer affects the value of signal TEMP.

It is important to understand the semantics of a signal assignment to determine the inference of synchronous logic. Consider the difference between the following two architecture bodies, BODY1 and BODY2.

```
architecture BODY1 of NO_NAME is
        signal A: MVL;
begin
        A <= not A;
end BODY1;

architecture BODY2 of NO_NAME is
        signal A, CLOCK: MVL;
begin
        process (A, CLOCK)
        begin
                if (CLOCK = '0' and CLOCK'EVENT) then
                        A <= not A;
                end if;
        end process;
end BODY2;
```

Architecture BODYI implies the circuit shovm in Fig. 12.8 while architecture BODY2 implies the circuit shown in Fig. 12.9.

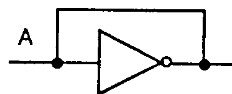



**Figure 12.8 No latch implied.**

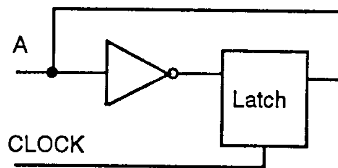



**Figure 12.9 A latch implied.**

If architecture BODYI were simulated as is, simulation will go into an endless loop due to the zero delay asynchronous loop (simulation time does not advance) and each execution of the concurrent signal assignment statement will keep triggering itself after a delta delay. Whereas in architecture BODY2, the value of A is latched only on the falling edge of the CLOCK signal and thereafter, any changes on A (input of latch) do not affect the output of latch.

A signal declared to be of the register kind also models a latch. If all drivers to such a signal are disconnected, the signal retains its old value, thereby implying the semantics of a latch. Consider the circuit shown in Fig. 12.10 in which a capacitive latch is loaded with different values based on the two mutually exclusive clock signals. The disconnect time models the decay time for the capacitive latch.

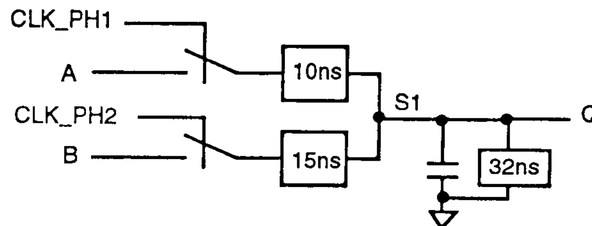



**Figure 12.10 A capacitive latch.**

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
```

```
entity REGISTER_SIGNAL is
        port (A, B, CLK_PH1, CLK_PH2: in MVL; Q: out MVL);
end REGISTER_SIGNAL;

architecture EXAMPLE of REGISTER_SIGNAL is
        signal S1: WIRED_OR MVL register;
        disconnect SI: MVL after 32 ns;
begin
        B1: block (CLK_PH1 ='1')
        begin
                S1 <= guarded A after 10 ns;
        end block B1;
        B2: block (CLK_PH2 = '1')
        begin
                S1 <= guarded B after 15 ns;
        end block B2;
        Q<=S1;
end EXAMPLE;
```

## 12.8 State Machine Modeling

State machines can usually be modeled using a case statement in a process. The state information is stored in a signal. The multiple branches of the case statement contain the behavior for each state. Here is an example of a simple multiplication algorithm represented as a state machine. When RESET signal is high, the accumulator ACC and the counter COUNT are initialized. When RESET goes low, multiplication starts. If the bit of the multiplier MPLR in position COUNT is 1', the multiplicand MCND is added to the accumulator. Next, the multiplicand is left-shifted by one bit and the counter is incremented. If COUNT is 16, multiplication is complete and the DONE signal is set high. If not, the COUNT bit of the multiplier MPLR is checked and the process repeated. The state diagram is shown in Fig. 12.11 and the corresponding state machine model is shown next.

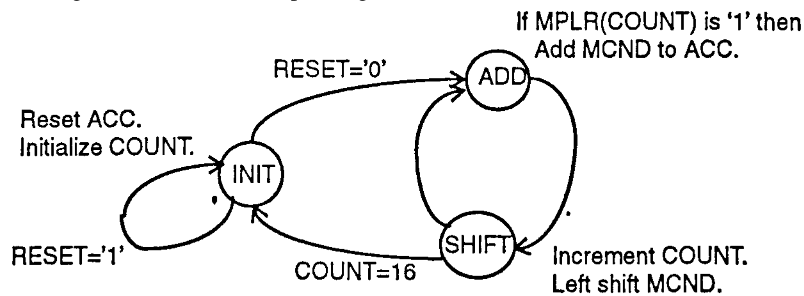



**Figure 12.11** State diagram for multiplier.

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity MULTIPLY is
        port (MPLR, MCND: in MVL_VECTOR(15 downto 0);
                -- MPLR is multiplier, MCND is multiplicand.
                CLOCK, RESET: in MVL;
                DONE: out MVL; ACC: buffer MVL_VECTOR(31 downto 0));
end MULTIPLY;

architecture STATE_MACHINE of MULTIPLY is
        type STATE_TYPE is (INIT, ADD, SHIFT);
        signal MPY_STATE: STATE_TYPE; -- Initial state is INIT.
begin
        process
                variable COUNT: NATURAL;
                variable MCND_TEMP: MVL_VECTOR(31 downto 0);
        begin
                wait until (CLOCK = '0' and CLOCK'EVENT);
                -- Alternately, "wait until (CLOCK = '0' and
                -- not CLOCK'STABLE);";
                --Both are equivalent in this context.
                case MPY_STATE is
                        when INIT=>
                                if RESET = '1' then
```

```
                    MPY_STATE <= INIT;
                    -- The above statement is not really
necessary
                    --since MPY_STATE will retain its old value.
                else
                    ACC <= (others => '0');
                    COUNT := 0;
                    MPY_STATE <= ADD;
                    DONE <= '0';
                    MCND_TEMP(15 downto 0) := MCND;
                    MCND_TEMP(31 downto 16) := (others =>
'0');
                end if;
            when ADD =>
                if (MPLR(COUNT) = '1') then
                    ACC <= ACC + MCND_TEMP;
                end if;
                MPY_STATE <= SHIFT;
            when SHIFT =>
                - Left-shift MCND:
                MCND_TEMP := MCND_TEMP(30 downto 0) &
'0';
                COUNT := COUNT+1;
                if (COUNT = 16) then
                    MPY_STATE <= INIT;
                    DONE <= '1';
                else
                    MPY_STATE <= ADD;
                end if;
        end case;
    end process;
end STATE_MACHINE;
```

The signal MPY_STATE holds the state of the model. Initially the model is in state INIT and stays in this state as long as signal RESET is '1'. When RESET gets the value '0', the accumulator ACC is cleared, the counter COUNT is reset and the multiplicand MCND is loaded into a temporary variable MCND_TEMP, and model advances to state ADD. When model is in ADD state, the multiplicand in MCND_TEMP is added to ACC only if the bit at the COUNT position of the multiplier is a '1', and then the model advances to state SHIFT. In this state, the multiplier is left shifted once, counter is incremented and if the counter value is 16, signal DONE is set to '1' and model returns to state INIT. At this time, ACC contains the result of the multiplication. If the counter value was less than 16, the mode? repeats itself going through states ADD and SHIFT until the counter value becomes 16.

State transitions occur at every falling clock edge; this is specified by the wait statement. The mode of signal ACC is set to buffer since the value is read and updated within the model.

## *12.9 Interacting State Machines*

Interacting state machines can be described as separate processes communicating via common signals. Consider the state transition diagram shown in Fig. 12.12 for two interacting processes, TX, a transmitter, and MP, a microprocessor. If process TX is not busy, process MP sets the data to be transmitted on a data bus and sends a signal, LOAD_TX, to process TX to load the data and to begin transmitting. A signal, TX_BUSY, is set by process TX during transmission to indicate that it is busy and that it cannot receive any further data from process MP.

A skeleton model for these two interacting processes is shown next. Only the control signals and state transitions are shown. Data manipulation code is not described.

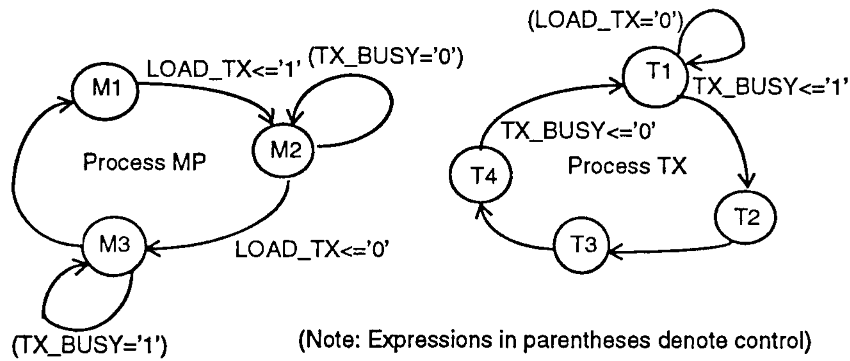



**Figure 12.12 State diagram of two interacting processes.**

```
architecture INTERACTING of FSM1 is
        type MP_STATE_TYPE is (M1, M2, M3);
        type TX_STATE_TYPE is (T1, T2, T3, T4);
        signal MP_STATE: MP_STATE_TYPE;
        signal TX_STATE: TX_STATE_TYPE;
        signal LOAD_TX,TX_BUSY:BIT;
        -- CLOCK is an input signal.
begin
        MP: process
        begin
                wait until (CLOCK = '0' and CLOCK'EVENT);
                case MP_STATE is
                        when M1 =>      -- Load data on data bus.
                                LOAD_TX <= '1';
                                MP_STATE <= M2;
                        when M2 =>      -- Wait for acknowledge.
                                if TX_BUSY = '1'  then
                                        MP_STATE <= M3;
                                        LOAD_TX <= '0';
                                end if;
                        when M3 =>      -- Wait for TX to finish.
                                if TX_BUSY = '0' then
                                        MP_STATE<=M1;
                                end if;
                end case;
        end process MP;
        TX: process (CLOCK)
        begin
                if (CLOCK = '0' and CLOCK'EVENT) then
                        case TX_STATE is
                                when T1 =>   -- Wait for data to load.
                                    if LOAD_TX = '1' then
                                        TX_STATE <= T2;
                                        TX_BUSY <= '1'; - Read data from data
                                        bus.
                                    end if;
                                when T2 =>   - Transmitting data.
                                        TX_STATE <= T3;
                                when T3 =>
                                        TX_STATE <= T4;
                                when T4 =>   - Transmission completed.
                                        TX_BUSY <= '0';
                                        TX_STATE <= T1;
                        end case;
                end if;
        end process TX;
end INTERACTING;
```

Two different ways of modeling a falling clock edge are shown in the two processes, MP and TX. Both are equivalent in this case since all statements in the processes are synchronized to the clock edge. However, if there

were some asynchronous statements, it would be necessary to use the if statement style. The wait statement style could also be used, but: then two separate processes would have to be used, one for the synchronous part and one for the asynchronous part. The sequence of actions for the previous entity is shown in Fig. 12.13.

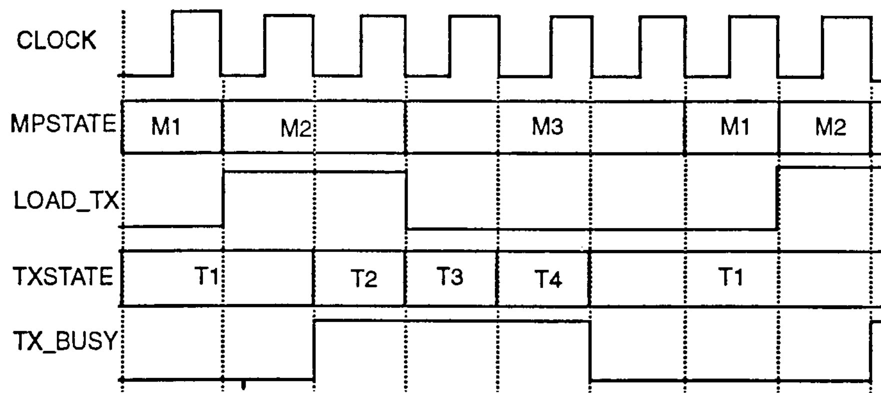



**Figure 12.13 Sequence of actions for the two interacting processes.**

Consider another example of two interacting processes, DIV, a clock divider, and RX, a receiver. In this case, process DIV generates a new clock, and process RX goes through its sequence of states synchronized to this new clock. The state diagram is shown in Fig. 12.14.

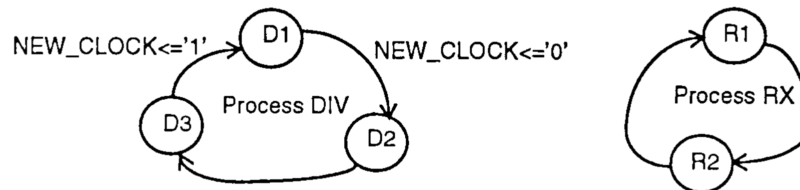



**Figure 12.14 DIV generates clock for RX.**

```
architecture ANOTHER_EXAMPLE of FSM2 is
        type DIV_STATE_TYPE is (D1, D2, D3);
        type RX_STATE_TYPE is (R1, R2);
        signal DIV_STATE: DIV_STATE_TYPE;
        signal RX_STATE; RX_STATE_TYPE;
        signal NEW_CLOCK: BIT;
        -- Signal CLOCK is an input port.
begin
        DIV: process
        begin
                wait until (CLOCK = '1 ' and CLOCK'EVENT);
                case DIV_STATE is
                        when D1 => DIV_STATE <= D2; NEW_CLOCK <= '0';
                        when D2 => DIV_STATE <= D3;
                        when D3 => NEW_CLOCK <= '1'; DIV_STATE <= D1;
                end case;
        end process DIV;
        RX: process
        begin
                wait until (NEW_CLOCK = '0' and NEW_CLOCK'EVENT);
                case RX_STATE is
                        when R1 => RX_STATE <= R2;
                        when R2 => RX_STATE <= R1;
                end case;
        end process RX;
end ANOTHER_EXAMPLE;
```

Process DIV generates a new clock, NEW_CLOCK, as it goes through its sequence of states. The state transitions in this process occur on the rising edge of the signal CLOCK. Process RX is executed every time a falling edge on the NEW_CLOCK signal occurs. The sequence of waveforms for these interacting state machines is shown in Fig. 12.15.

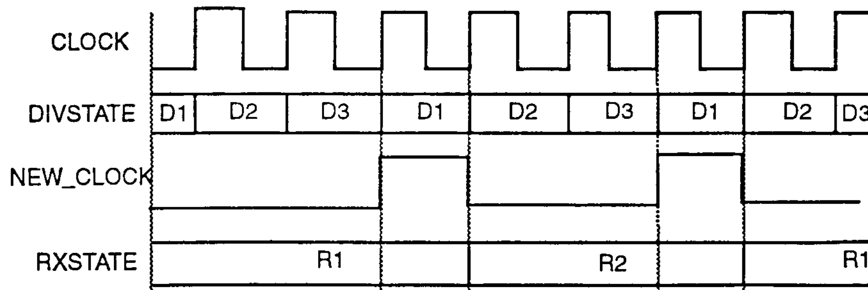



**Figure 12.15 Interaction between processes, RX and DIV.**

## 12.10 Modeling Moore FSM

The output of a Moore finite state machine (FSM) depends only on the state and not on its inputs. This type of behavior can be modeled using a single process with a case statement that switches on the state value. An example of a state transition diagram for a Moore finite state machine is shown in Fig. 12.16 and its corresponding behavior model appears next.

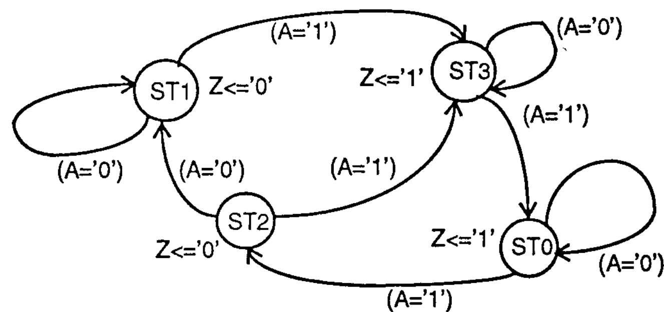



**Figure 12.16 State diagram of a Moore machine.**

```
library ATTLIB; use ATTLIB.ATT_PACKAGE.all;
entity MOORE_FSM is
        port (A, CLOCK: in BIT; Z: out MVL);
end MOORE_FSM;

architecture FSM_EXAMPLE of MOORE_FSM is
        type STATE_TYPE is (ST0, ST1, ST2, ST3);
        signal MOORE_STATE: STATE_TYPE;
begin
        process (CLOCK)
        begin
                if (CLOCK = '0' and CLOCK'EVENT) then
                        case MOORE_STATE is
                                when ST0 =>
                                        Z<='1';
                                        if (A = '1') then
                                                MOORE_STATE <= ST2;
                                        end if;
                                when ST1 =>
                                        Z<= '0';
                                        if (A = '1') then
                                                MOORE_STATE <= ST3;
                                        end if;
                                when ST2 =>
                                        Z <= '0';
                                        if (A = '0') then
                                                MOORE_STATE <= ST1:
                                        else
                                                MOORE_STATE <= ST3;
                                        end if;
                                when ST3 =>
                                        Z<='1';
                                        if (A = '1') then
                                                MOORE_STATE <= ST0;
```

```
                        end if;
                end case;
            end if;
        end process;
end FSM_EXAMPLE;
```

## 12.11 Modeling Mealy FSM

In a Mealy type finite state machine, the outputs not only depend on the state of the machine but also on its inputs. This type of finite state machine can also be modeled in a similar style as the Moore example case, that is, using a single process. To show the variety of the language, a different style is used to model a Mealy machine. In this case, we use two processes, one process that models the synchronous aspect of the finite state machine and the second process models the combinational part of the finite state machine. Here is an example of a state transition table, shown in Fig. 12.17, and its corresponding behavior model.

| Present state | Input A: 0 | Input A: 1 |
|---|---|---|
| ST0 | ST0 / 0 | ST3 / 1 |
| ST1 | ST1 / 1 | ST0 / 0 |
| ST2 | ST2 / 0 | ST1 / 1 |
| ST3 | ST2 / 0 | ST1 / 0 |

(Entries in table are next state and output Z)

**Figure 12.17 State transition table for a Mealy machine.**

```
library ATTLIB; use ATTUB.ATT_PACKAGE.all;
entity MEALY_FSM is
        port (A, CLOCK: in BIT; Z: out MVL);
end MEALY_FSM;

architecture YET_ANOTHER_EXAMPLE of MEALY_FSM is
        type MEALY_TYPE is (ST0, ST1, ST2, ST3);
        signal P_STATE, N_STATE: MEALY_TYPE;
begin
        SEQ_PART: process (CLOCK)
        begin
                -- Synchronous section:
                if (CLOCK = '0' and CLOCK'EVENT) then
                        P_STATE <= N_STATE;
                end if;
        end process SEQ_PART;
        COMB_PART: process (P_STATE, A)
        begin
                case P_STATE is
                        when ST0 =>
                                if (A = '1') then
                                        Z <= '1 ': N_STATE <= ST3:
                                else
                                        Z<='0';
                                end if;
                        when ST1 =>
                                if (A = '1') then
                                        Z <= '0';
                                        N_STATE <= ST0;
                                else
                                        Z<='1' ;
                                end if;
                        when ST2 =>
                                if (A = '0') then
                                        Z <= '0';
                                else
```

```
                        Z <=' 1'; N_STATE <= ST1;
                    end if;
                when ST3 => Z<='0';
                    if (A = '0') then
                        N_STATE <= ST2;
                    else
                        N_STATE <= ST1;
                    end if;
            end case;
        end process COMB_PART;
end YET_ANOTHER_EXAMPLE;
```

In this type of finite state machine, it is important to put the input signals in the sensitivity list for the combinational part process, since the outputs may directly depend on the inputs independent of the clock. Such a condition does not occur in a Moore finite state machine since outputs depend only on states and state changes occur synchronously.

## 12.12 A Simplified Blackjack Program

This section presents a state machine description of a simplified blackjack program. The blackjack program is played with a deck of cards. Cards 2 to 10 have values equal to their face value, and an ace has a value of either I or II. The object of the game is to accept a number of random cards such that the total score (sum of values of all cards) is as close to 21 without exceeding 21.

When a new card is inserted, signal CARD_RDY is true and signal CARD_VALUE has the value of the card. The signal REQUEST_CARD indicates when the program is ready to accept a new card. If a sequence of cards is accepted such that total exceeds 21, signal LOST is set to true indicating that it has lost, else the signal WON is set to true indicating that the game has been won. The state sequencing is controlled by the signal CLOCK. This external interface of the blackjack program, shown in Fig. 12.18, is expressed using the following entity declaration.

```
entity BLACKJACK is
    port (CARD_RDY: in BOOLEAN; CARD_VALUE: in INTEGER;
        REQUEST_CARD, WON, LOST: out BOOLEAN;
        CLOCK: in BIT);
end BLACKJACK;
```

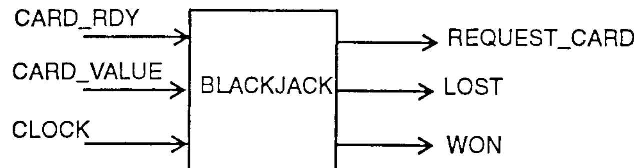



**Figure 12.18 External view of blackjack program.**

The behavior of the program is described in the following architecture body. The program accepts cards until its score is at least 17. The first ace is counted as a II unless the score exceeds 21 in which case 10 is subtracted so that the value for an ace of I is used. Three registers are used to store the state of the program: TOTAL to hold the sum, CURRENT_CARD_VALUE to hold the value of the card read (which could be I through 10), and ACE_AS_11 to remember whether an ace was counted as a II instead of a 1. The state of the blackjack program is stored in signal BJ_STATE.

```
architecture STATE_MACHINE of BLACKJACK is
    type STATE_TYPE is (INITIAL_ST, GETCARD_ST, REMCARD_ST,
        ADD_ST, CHECK_ST, WIN_ST, BACKUP_ST, LOSE_ST);
    signal BJ_STATE: STATE_TYPE;
begin
    process
        variable CURRENT_CARD_VALUE, TOTAL: INTEGER;
        variable ACE_AS_11: BOOLEAN;
    begin
        wait until (CLOCK = '0' and CLOCK'EVENT);
        case BJ_STATE is
            when INITIAL_ST=>
              TOTAL := 0; ACE_AS_11 := FALSE; WON <= FALSE;
              LOST <= FALSE; BJ_STATE <= GETCARD_ST;
```

```
                    when GETCARD_ST =>
                        REQUEST_CARD <= TRUE;
                        if CARD_RDY then
                                CURRENT_CARD_VALUE := CARD_VALUE;
                                BJ_STATE <= REMCARD_ST;
                        end if;        -- Else stay in GETCARD_ST state.
                     when REMCARD_ST => -- Wait for card to be removed.
                        if CARD_RDY then
                                REQUEST_CARD <= FALSE;
                        else
                                BJ_STATE <= ADD_ST;
                        end if;
                    when ADD_ST =>
                        if ( not ACE_AS_11) and
                                (CURRENT_CARD_VALUE = 1) then
                                CURRENT_CARD_VALUE := 11;
                                ACE_AS_H := TRUE;
                        end if;
                        TOTAL := TOTAL + CURRENT_CARD_VALUE;
                        BJ_STATE <= CHECK_ST;
                    when CHECK_ST =>
                        if (TOTAL < 17) then
                                BJ_STATE <= GETCARD_ST;
                        elsif (TOTAL < 22) then
                                BJ_STATE <= WIN_ST;
                        else
                                BJ_STATE <= BACKUP_ST;
                        end if;
                    when BACKUP_ST =>
                        if ACE_AS_11 then
                                TOTAL := TOTAL - 10; ACE_AS_11 := FALSE;
                                BJ_STATE <= CHECK_ST:
                        else
                                BJ_STATE <= LOSE_ST;
                        end if;
                    when LOSE_ST =>
                        LOST <= TRUE; REQUEST_CARD <= TRUE;
                        If CARD_RDY then
                                BJ_STATE <= INITIAL_ST;
                        end if;        -- Else stay in this state.
                    when WIN_ST =>
                        WON <= TRUE; REQUEST_CARD <= TRUE;
                        if CARD_RDY then
                                BJ_STATE <= INITIAL_ST;
                        end if;        -- Else stay in this state.
            end case;
        end process;
    end STATE_MACHINE;
```

## *12.13 Hierarchy in Design*

A hierarchy in a design is managed by using component instantiation and component declaration statements. Given a large design, one could decompose a design into a number of smaller subdesigns. The components (same as a subdesign) at each level would instantiate components at lower levels in the hierarchy. The description of components at any level could be in any design style, that is, dataflow, behavioral, structural, or mixed. For example, all components could be described as sequential processes initially, and as each com-
ponent at each level is synthesized, it may be described using the structural style. Consider the hierarchicaJ design shown in Fig. 12.19.

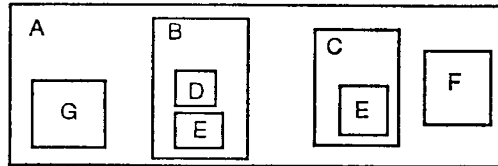



**Figure 12.19 A hierarchical design.**

Design A is composed of four components, B, C, F, and G. Subdesign B is composed of components D, E, and some glue behavioral logic. Subdesign C is composed of component E and some other behavioral logic. Component E is used by components B and C. Initially, there would be a separate entity declaration plus architecture body for each component A, B, C, D, E, F, and G. As each component is synthesized to its primitive gate level, different architecture bodies for each component are developed. A skeleton model for this large design example is shown next.

```
--Library and use clauses, if any.
entity A is port (... ) end A;

architecture A_INTERNALS of A is
        component B port (... ) end component;
        component C port (... ) end component;
        component F port (... ) end component;
        component G port (... ) end component;
begin
        B_COMP1 : B port map ( . . . );
        C_COMP1 : C port map ( . . . );
        F_COMP1 : F port map ( . . . );
        G_COMP1: G port map ( . . . );
        -- There might be other glue logic here, for example, high
        -- power drivers for the I/O for A, etc.
end A _INTERNALS;

entity F is port ( . . . ) end F;
architecture F_INTERNALS of F is
begin
        -- Behavior of F.
end F_INTERNALS;

entity G is port ( . . . ) end;
architecture G_INTERNALS of G is
begin
        -- Behavior of Q.
end G_INTERNALS;

entity B is port ( . . . ) end B;
architecture B_INTERNALS of B is
        component D port ( . . . ) end component;
        component E port ( . . . ) end component;
begin
        D_COMP1: D port map ( . . . );
        E_COMP1: E port map ( . . . );
        -- Remaining behavior of B.
end B_INTERNALS;

entity C is port ( . . . ) end C;
architecture C_INTERNALS of C is
component E port ( . . .) end component;
begin
        E_COMP2:E port map (...);
        -- Other behavior of C.
end C_INTERNALS;

entity D is port (...) end D;
architecture D_INTERNALS of D is
begin
        -- Behavior of D.
```

```
end D_INTERNALS;

entity E is port (...) end E;
architecture E_INTERNALS of E is
begin
        -- Behavior of E.
end E_INTERNALS;
```

Each entity (entity declaration plus an architecture body) could be placed in a separate text file and compiled into a working library. Having described the entire system behavior for design A, it could then be simulated and verified. There is no need to specify a configuration if all the entities are compiled into the working library, since the default bindings will be used (see Sec. 7.5).

Next the synthesis of each component could either proceed top-down, that is, translate behavior of design A to structure, then for B, C, D, E, F, G, or it can proceed bottom-up, that is, synthesize subdesign E and D before B or C. Say component E is synthesized into a structure either using available synthesis tools or by a manual design process, this structure could be saved in a new architecture body that is still associated with the same entity declaration E. For example,

```
architecture E_2 of E is
begin
        -- Structure of E.
end E_2;
```

To simulate this new architecture, a configuration declaration would have to be written that would bind entity E to architecture E_2. For example,

```
configuration E_CON of E is
        for E_2
        end for;
end E_CON;
```

Configuration E_CON would now be verified on a stand-alone basis. This would verify the entity E using architecture E_2. To simulate the entire design A again, references to component E in design B and C must bind its instantiations to the new architecture for E. This could be done using a configuration specification that is added to the architecture body for both B and C, such as

```
for camp-label: E use configuration WORK.E_CON;
```

If the architecture bodies for B and C cannot be changed, a configuration declaration can be written for design A as shown next.

```
configuration A_CON of A is
        for B
                for E_COMP1 :E
                        use configuration WORK.E_CON;
                end for;
        end for;
        for C
                for E_COMP2: E
                        use configuration WORK.E_CON;
                end for;
        end for;
        -- For unbound components, use default rules (see Sec. 7.5).
end A_CON;
```

The basic idea is that every time a component is synthesized into its structure, a new architecture body is created that is associated with the same entity declaration. A configuration declaration or a configuration specification can be written to bind the new architecture with the component one level above in the hierarchy.

It is not necessary to keep the subdesigns of an entire design in a single design library. However, all architecture bodies of a component must reside in the same library. This is because they must reside with their corresponding entity declaration. If components reside in different libraries, configuration specifications would have to be written (the default rules work for only components in one library, that is, working library) and these will be made visible to their next higher level component by using context clauses. For example, if component E were to reside in library JOHNS_LIB, then the configuration specification appearing in the architecture for subdesign C might be

```
library JOHNS_LIB;
architecture C_2 of C is
        for E_COMP2: E use configuration JOHNS_LIB.E_CON;
begin
        . . .
end C_2;
```

# APPENDIX A *Predefined Environment*

This appendix describes the predefined environment of the language. The set of reserved words for the language is listed in the first section. The next two sections give the source code listing for the predefined packages STANDARD and TEXT10. The predefined attributes have already been described in Chap. 10. All information in this appendix has been reprinted from the IEEE Std 1076-1987 IEEE Standard VHDL Language Reference Manual. (Reprinted here from the IEEE Std VHDL LRM 1076-1987 by permission from IEEE).

## *A.1 Reserved Words*

The following identifiers are reserved words in the language (also called keywords), and therefore, cannot be used as identifiers in a VHDL description.

| | | | |
|---|---|---|---|
| **Abs** | **access** | **after** | **alias** |
| **all** | **and** | **architecture** | **array** |
| **assert** | **attribute** | | |
| **begin** | **block** | **body** | **buffer** |
| **bus** | | | |
| **case** | **component** | **configuration** | **constant** |
| **disconnect** | **downto** | | |
| **else** | **elsif** | **end** | **entity** |
| **exit** | | | |
| **file** | **for** | **function** | |
| **generate** | **generic** | **guarded** | |
| **if** | **in** | **inout** | **is** |
| **label** | **library** | **linkage** | **loop** |
| **map** | **mod** | | |
| **nand** | **new** | **next** | **nor** |
| **not** | **null** | | |
| **of** | **on** | **open** | **or** |
| **others** | **out** | | |
| **package** | **port** | **procedure** | **process** |
| **range** | **record** | **register** **rem** | |
| **report** | **return** | | |
| **select** | **severity** | **signal** | **subtype** |
| **then** | **to** | **transport** | **type** |
| **units** | **until** | **use** | |
| **variable** | | | |
| **wait** | **when** | **while** | **with** |
| **xor** | | | |

## *A.2 Package STANDARD*

The package STANDARD is a predefined package that contains the definitions for the predefined types and functions of the language. A source code listing of this package follows.

```
package STANDARD is
        -- Predefined enumeration types:
        type BOOLEAN is (FALSE, TRUE);
        type BIT is ('0', '1');
        type CHARACTER is (

                NUL,    SOH,    STX,    ETX,    EOT,    ENQ,    ACK,
```

```
BEL,    BS,     HT,     LF,     VT,     FF,     CR,
SO,     SI,     DLE,    DCI,    DC2,    DC3,    DC4,
NAK,    SYN,    ETB,    CAN,    EM,     SUB,    ESC,
FSP,    GSP,    RSP,    USP,

' ',    '!',    '”',    '#',    '$',    '%',    '&',
''',    '(',    ')',    '*',    '+',    ',',    '-',
'.',    '/',    '0',    '1',    '2',    '3',    '4',
'5',    '6',    '7',    '8',    '9',    ':',    ';',
'<',    '=',    '>',    '?',

'@',    'A',    'B',    'C',    'D',    'E',    'F',
'G',    'H',    'I',    'J',    'K',    'L',    'M',
'N',    'O',    'P',    'Q',    'R',    'S',    'T',
'U',    'V',    'W',    'X',    'Y',    'Z',    '[',
'\',    ']',    '^',    '_',

'`',    'a',    'b',    'c',    'd',    'e',    'f',
'g',    'h',    'i',    'j',    'k',    'l',    'm',
'n',    'o',    'p',    'q',    'r',    's',    't',
'u',    'v',    'w',    'x',    'y',    'z',    '{',
'|',    '}',    '~',    DEL
);

type SEVERITY_LEVEL is (NOTE, WARNING,
                        ERROR, FAILURE);

-- Predefined numeric types:
type INTEGER is range implementation_defined;
type REAL is range implementation_defined;

- Predefined type TIME:
type TIME is range implementation_defined
        units
                fs;                     -- femtosecond
                ps   = 1000 fs;         -- picosecond
                ns   = 1000 ps;         -- nanosecond
                us   = 1000 ns;         -- microsecond
                ms   = 1000 us;         -- microsecond
                sec  = 1000 ms;         -- seconds
                min  = 60 sees;         -- minutes
                hr   = 60 min;          -- hours
        end units;

        -- Function that returns the current simulation time:
        function NOW return TIME;

        -- Predefined numeric subtypes:
        subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
        subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;

        - Predefined array types:
        type STRING is array (POSITIVE range <>) of CHARACTER;
        type BIT_VECTOR is array (NATURAL range <>) of BIT;
end STANDARD;
```

## *A.3 Package TEXTIO*

Package TEXTIO contains declarations of types and subprograms that support formatted ASCII I/O operations. A source code listing of the package follows.

```
package TEXTIO is
        -- Type definitions for text I/O:
        type LINE is access STRING;     -- A line is a pointer to a
                                        -- STRING value.
        type TEXT is file of STRING;    -- A file of variable-length
```

```
                                    --ASCII records.
type SIDE is (RIGHT, LEFT);         -- For justifying output data
                                    -- within fields.
subtype WIDTH is NATURAL;           -- For specifying widths of
                                    -- output fields.

-- Standard text files:
file INPUT: TEXT is in "STD_INPUT";
file OUTPUT: TEXT is out "STD_OUTPUT":

-- Input routines for standard types:
procedure READLINE (F: in TEXT; L: out LINE);

procedure READ (L: inout LINE; VALUE: out BIT;
                GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out BIT);

procedure READ (L: inout LINE; VALUE: out BIT_VECTOR;
                GOOD; out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out BIT_VECTOR);

procedure READ (L; inout LINE; VALUE: out BOOLEAN;
                GOOD: out BOOLEAN);
procedure READ (L; inout LINE; VALUE: out BOOLEAN):

procedure READ (L: inout LINE; VALUE: out CHARACTER;
                GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out CHARACTER);

procedure READ (L: inout LINE; VALUE: out INTEGER;
                GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out INTEGER);

procedure READ (L: inout LINE; VALUE: out REAL;
                GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out REAL);

procedure READ (L: inout LINE; VALUE: out STRING;
                GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out STRING);

procedure READ (L: inout LINE; VALUE: out TIME;
                GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out TIME);

-- Output routines for standard types;
procedure WRITELINE (F: out TEXT; L: in LINE);

procedure WRITE (L: inout LINE; VALUE: in BIT;
        JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in BIT_VECTOR;
        JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in BOOLEAN:
        JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in CHARACTER;
        JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in INTEGER;
        JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in REAL;
        JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0;
        DIGITS: in NATURAL := 0);
procedure WRITE (L: inout LINE; VALUE: in STRING;
        JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in TIME;
        JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0;
```

```
                    UNIT; in TIME := ns);

        -- File position predicates:
        function ENDLINE (L: in LINE) return BOOLEAN;
        -- function ENDFILE (F: in TEXT) return BOOLEAN;
end TEXTIO;
```

# APPENDIX B *Syntax Reference*

This appendix presents the complete syntax of the VHDL language. ( Reprinted here from the IEEE Std VHDL LRM 1076-1987 by permission from IEEE ).

## *B.1 Conventions*

The following conventions are used in describing this syntax.

**1.** The syntax rules are organized in an alphabetical order by their left-hand nonterminal name.
**2.** Reserved words are written in **boldface.**
**3.** A name in *italics* prefixed to a nonterminal name represents the semantic meaning associated with that nonterminal name.
**4.** The vertical bar symbol ( | ) separates alternative items unless it appears immediately after an opening brace in which case it stands for itself.

**5.** Square brackets ([. . . ]) denote optional items.
**6.** Curly braces identify ({ . . . }) an item that is repeated zero or more times.
**7.** "The starting nonterminal name is "design_file".
**8.** The terminal names used in this grammar appear in upper case.

## *B.2 The Syntax*

Abstract_literal ::= decimal_literal | based_literal

access_type_definition ::= **access** subtype_indication

actual_designator ::=
    expression
    | *signal*_name
    | *variable*_name
    | **open**

actual_parameter_part := parameter_association_list

actual_part ::= actual_designator I function_name ( actual_designator )

adding_operator ::= + | - | &

aggregate *::*= ( element_association { , element_association } )

alias_declaration ::= **alias** identifier: subtype_indication **is** name;

allocator ::= **new** subtype_indication | **new** qualified_expression

architecture_body **::=**
    **architecture** identifier **of** *entity*_nama **is**
        architecture_declarative_part
    **begin**
        architecture_statement_part
    **end** [ *architecture*_simple_name ] ;

architecture_declarative_part ::= { block_declarative_item}

architecture_statement_part ::= { concurrent_statement}

array_type_definition ::=
    unconstrained_array_definition | constrained_array_definition

assertion_statement ::=
    **assert** condition
        [ **report** expression ]
        [ **severity** expression ] ;

association_element ::= [ format_part => ] actual_part

association_list ::= association_element { , association_element}

attribute_declaration ::= **attribute** identifier: type_mark;

attribute_designator ::= *attribute*_simple_name

attribute_name ::= prefix ' attribute_designator [ ( static_expression ) ]

attribute_specification ::=
    **attribute** attribute_designator **of** entity_specification **is** expression ;

base ::= integer

base_specifier ::= B | O | X

base_unit_declaration ::= identifier;

based_integer ::= extended_digit { [ UNDERLINE ] extended_digit )

based_literal ::= base # based_integer [. Based_integer ] # [ exponent ]

basic_character ::= basic_graphic_character | FORMAT_EFFECTOR

basic_graphic_character :=
    UPPER_CASE_LETTER
    | DIGIT
    | SPECIAL_CHARACTER
    | SPACE_CHARACTER

binding_indication ::=
    entity_aspect [ generic_map_aspect ] [ port_map_aspect ]

bit_string_literal ::= base_specifier " bit_value "

bit_value ::= extended_digit { [ UNDERLINE ] extended_digit }

block_configuration ::=
    **for** block_specification
        { use_clause}
        { configuration_item}
    **end for;**

block_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | configuration_specification
    | disconnection_specification
    | use_clause

block_declarative_part ::= { block_declarative_item}

block_header ::=
    [ generic_clause [ generic_map_aspect ; ] ]
    [ port_clause [ port_map_aspect ; ] ]

block_specification ::=
    *architecture*_name
    | *block*_staten1ent_label
    | *generate_statement*_label [ ( index_specification ) ]

block_statement ::=
    *block*_label:
        **block** [ ( *guard*_expression ) ]
            block_header
            block_declarative_part
        **begin**
            block_statement_part
        **end block** [ *block*_label ] ;

block_statement_part ::= { concurrent_statement}
case_statement ::=
    **case** expression **is**
        case_statement_alternative
        { case_statement_alternative}

**end case;**

case_statement_alternative ::= **when** choices => sequence_of_statements

character_literal ::= ' graphic_character '

choice ::=
simple_expression
| discrete_range
| *element*_simple_name
I **others**

choices ::= choice { | choice )

component_configuration ::=
**for** component_specification
[ **use** binding_indication ; ]
[ block_configuration ]
**end for;**

component_declaration ::=
**component** identifier
[ *local*_generic_clause ]
[ *local*_port_clause ]
**end component;**

component_instantiation_statement ::=
instantiation_label:
component_name [ generic_map_aspect ] [ port_map_aspect ];

component_specification ::= instantiation_list: *component_name*

composite_type_definition ::= array_type_definition | record_type_definition

concurrent_assertion_statement ::= [ label : ] assertion_statement

concurrent_procedure_call ::= [ label : ] procedureLcall_statement

concurrent_signal_assignment_statement ::=
[ label : ] conditional_signaLassignment
| [ label : ] selected_signal_assignment

concurrent_statement ::=
block_statement
| process_statement
| concurrent_procedure_call
| concurrent_assertion_statement
| concurrent_signal_assignment_statement
| component_instantiation_statement
| generate_statement

condition ::= boolean_expression

condition_clause ::= **until** condition

conditional_signal_assignment ::= target <= options conditional_waveforms ;

conditional_waveforms ::= { waveform when condition else} waveform

configuration_declaration ::=
**configuration** identifier **of** *entity*_name **is**
configuration_declarative_part
block_configuration
**end** [ *configuration*_simple_name ];

configuration_declarative_item ::= use_clause | attribute_specification

configuration_declarative_part ::= { configuration_declarative_item}

configuration_item ::= block_configuration | component_configuration

configuration_specification ::=
    **for** component_specification **use** binding_indication ;

constant_declaration ::=
    **constant** identifier_list: subtype_indication [ := expression ] ;

constrained_array_definition ::=
    **array** index_constraint **of** *element*_subtype_indication

constraint ::= range_constraint | index_constraint

context_clause ::= { context_Item }

context_item ::= library_clause | use_clause

decimal_literal ::= integer [. integer ] [ exponent ]

declaration ::=
    type_declaration
    | subtype_declaration
    | object_declaration
    | file_declaration
    | interface_declaration
    | alias_declaration
    | attribute_declaration
    **|** component_declaration
    | entity_declaration
    | configuration_declaration
    | subprogram_declaration
    | package_declaration

design_file ::= design_unit { design_unit}

design_unit ::= context_clause library_unit

designator ::= identifier | operator_symbol

direction ::= **to** | **downto**

disconnection_specification ::=
    **disconnect** guarded_signal_specification **after** *time*_expression ;

discrete_range ::= discrete_subtype_indication | range

element_association ::= [ choices => ] expression

element_declaration ::= identifier_list: element_subtype_definition ;

element_subtype_definition ::= subtype_indication

entity_aspect ::=
    **entity** *entity*_name [ ( *arcitecture*_identifier ) **]**
    **| configuration** *configuration*_name
    | **open**

entity_class ::=
    **entity | architecture | configuration | procedure**
    **| function | package | type | subtype | constant | signal**
    **| variable | component | label**

entity_declaration ::=
    **entity** identifier **is**

entity_header
entity_declarative_part
[ **begin**
entity_statement_part ]
**end** [ *entify*_simple_name];

entity_declarative_item ::=
subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| signal_declaration
| file_declaration
| alias_declaration
| attribute_declaration
| attribLrte_specification
| disconnection_specification
| use_clause

entity_declarative_part ::= { entity_declarative_item}

entity_designator ::= simple_name | operator_symbol

entity_header ::= [ *fomal*_generic_clause ] [ *formal*_port_clause ]

entity_name_list ::=
entity_designator { , entity_designator}
| **others**
| **all**

entity_specification ::= entity_name_list: entity_class

entity_statement ::=
concurrent_assertion_statement
| *passive*_concurrent_procedure_call
| *passive*_process_statement

entity_statement_part ::= { entity_statement}

enumeration_literal ::= identifier | character_literal

enumeration_type_definition ::= ( enumeration_literal { , enumeration_literal} )

exit_statement ::= **exit** [ *loop*_label ] [ **when** condition ] ;

exponent *::=* E [ + ] integer | E - integer

expression ::=
relation { **and** relation }
| relation { **or** relation }
| relation { **xor** relation }
| relation [ **nand** relation ]
| relation [ **nor** relation ]

extended_digit ::= DIGIT | letter

factor ::=
primary [ ** primary ]
| **abs** primary
| **not** primary

file_declaration ::=
**file** identifier: subtype_indication **is** [ mode ] file_logical_name;

file_logical_name ::= *string*_expression

file_type_definition ::= **file of** type_mark

floating_type_definition ::= range_constraint

formal_designator ::= *generic*_name | *port*_name | *parameter*_name

formal_parameter_list ::= *parameter*_interface_list

formal_part ::= formal_designator | *function*_name ( formal_designator )

full_type_declaration ::= **type** identifier **is** type_definition ;

function_call ::= *function*_name [ ( actual_parameter_part ) ]

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

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

generic_clause ::= **generic** ( generic_list ) ;

generic_list ::= *generic*_interface_list

generic_map_aspect ::= **generic map** ( genenc_association_list )

graphic_character ::=
    basic_graphic_character
    | LOWER_CASE_LETTER
    | OTHER_SPECIAL_CHARACTER

guarded_signal_specification ::= guarde_signal_list : type_mark

identifier ::= letter { [ UNDERLINE ] letter_or_digit}

identifier_list ::= identifier { , identifier}

if_statement ::=
    **if** condition **then**
        sequence_of_statements
    { **elsif** condition **then**
        sequence_of_statements}
    **[ else**
        sequence_of_statements ]
    **end if**;

incomplete_type_declaration ::= **type** identifier;

index_constraint ::= ( discrete_range {, discrete_range } )

index_specification ::= discrete_range | *static*_expression

index_subtype_definition ::= type_rnark **range <>**

indexed_name ::= prefix ( expression { , expression } )

instantiation_list ::=
instantiation_*label* {, *instantiation*_label}
| **others**
| **all**

integer ::= DIGIT { [ UNDERLINE ] DIGIT}

integer_type_definition ::= range_constraint

interface_constant_declaration ::=
    [ **constant** ] identifier_list: [ **in** ] subtype_indication
    [ := *static*_expression ]

interface_declaration ::=
    interface_constant_declaration
    | interface_signal_dsclaration
    | interface_variable_declaration

interface_element ::= interface_declaration

interface_list ::= interface_element {, interface_element}

interface_signal_declaration ::=
    [ **signal** ] identifier_list : [ mode ] subtype_indication [ **bus** ]
    [ := static_expression ]

interface_variable_declaration ::=
    [ **variable** ] identifier_list: [ mode ] subtype_indication
    [ := static_expression ]

iteration_scheme ::=
    **while** condition
    | **for** *loop*_parameter_specification

label ::= identifier

letter *::=* UPPER_CASE_LETTER | LOWER_CASE_LETTER

letter_or_digit ::= letter | digit

library_clause ::= **library** logical_name_list ;

library_unit ::= primary_unit | secondary_unit

literal ::=
    numeric_literal
    | enumeration_literal
    | string_literal
    | bit_string_literal
    | **null**

logical_name ::= identifier

logical_name_list ::= logical_name { , logical_name }

logical_operator ::= **and** | **or** | **nand** | **nor** | **xor**

loop_statement ::=
    [ toop_label: ]
    [ iteration_scheme ] **loop**
        sequence_of_statements
    **end loop** [ *loop*_label ] ;

miscellaneous_operator ::= ** | **abs** | **not**

mode ::= **in | out** | **inout** | **buffer** | **linkage**

multiplying_operator ::= * | / | **mod** | **rem**

name ::=
simple_name
| operator_symbol
| selected_name
| indexed_name
| slice_name
| attribute_name

next_statement ::= **next** [ *loop*_label ] [ **when** condition ];

null_statement ::= **null;**

numeric_literal ::= abstract_literal | physical_literal

object_declaration ::=
constant_declaration
| signal_declaration
| variable_declaration

operator_symbol ::= string_literal

options ::= [ **guarded** ] [ **transport** ]

package_body ::=
**package body** *pactege*_simple_name **is**
package_body_declarative_part
**end** [ *package*_simple_name ];

package_body_declarative_item ::=
subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| file_declaration
| alias_declaration
| use_clause

package_body_declarative_part ::= { package_body_declarative_item}

package_declaration ::=
**package** identifier **is**
package_declarative_part
**end** [ *package*_simple_name ];

package_declarative_item ::=
subprogram_declaration
| type_declaration
| subtype_declaration
| constant_declaration
| signal_declaration
| file_declaration
| alias_declaration
| component_declaration
| attribute_declaration
| attribLite_specification
| disconnection_specification
| use_clause

package_declarative_part ::= { package_declarative_item )

parameter_specification ::= identifier **in** discrete_range

physical_literal ::= [ abstract_literal ] *unit*_name

physical_type_definition ::=
    range_constraint
        **units**
            base_unit_declaration
            { secondary_unit_declaration}
        **end units**

port_clause ::= **port** ( port_list ) ;

port_list ::= port_interface_list

port_map_aspect ::= **port map** ( *port*_association_list )

prefix ::= name | function_call

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

primary_unit ::=
    entity_declaration
    | configuration_declaration
    | package_declaration

procedure_call_statement ::= *procedure*_name [ ( actual_parameter_part ) ] ;

process_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause

process_declarative_part ::= { process_declarative_item}

process_statement ::=
    [ *process*_label: ]
        **process** [ ( sensitivity_list ) ]
            process_declarative_part
        **begin**
            process_statement_part
        **end process** [ *process*_label ];

process_statement_part ::= { sequential_statement}

qualified_expression ::=
    type_mark ' ( expression )
    | type_mark ' aggregate

range ::=
    range_attribute_name
    | simple_expression direction simple_expression

range_constraint ::= **range** range

record_type_definition ::=
**record**
element_declaration
{ element_declaration}
**end record**

relation ::= simple_expression [ relational_operator simple_expression ]

relational_operator ::= = | /= | < | <= | > | >=

return_statement ::= **return** [ expression ] ;

scalar_type_definition ::=
enumeration_type_definition
| integer_type_definition
| floating_type_definition
| physical_type_definition

secondary_unit ::= architecture_body | package_body

secondary_unit_declaration ::= identifier = physical_literal;

selected_name ::= prefix. suffix

selected_signal_assignment ::=
**with** expression **select**
target <= options selected_waveforms ;

selected_waveforms ::= { waveform **when** choices ,} waveform **when** choices

sensitivity_clause ::= **on** sensitivity_list

sensitivity_list ::= *signal*_name {, *signal*_name }

sequence_of_statements ::= { sequential_statement}

sequential_statement ::=
wait_statement
| assertion_statement
| signal_assignment_statement
| variable_assignment_statement
| procedure_call_statement
| if_statement
| case_statement
| loop_statement
| next_statement
| exit_statement
| retum_statement
| null_statement

sign ::=+ | -
signal_assignment_statement ::= target **<=** [ **transport** ] waveform ;

signal_declaration ::=
**signal** identifier_list : subtype_indication [ signal_kind ] [ := expression ]**;**

signal_kind ::= **register** | **bus**

signal_list ::=
*signal*_narne {, *signal*_name }
| **others**
| **all**

simple_expression ::= [sign] term { adding_operator term}

simple_name ::= identifier

slice_name ::= prefix ( discrete_range )

string_literal ::= " { graphic_character} "

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

subprogram_declaration ::= subprogram_specification;

subprogram_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause

subprogram_declarative_part ::= { subprogram_declarative_item}

subprogram_specification ::=
    **procedure** designator [ ( formal_parameter_list ) ]
    | **function** designator [ ( formal_parameter_list ) ] **return** type_mark

subprogram_statement_part ::= { sequential_statement}

subtype_declaration ::= **subtype** identifier **is** subtype_indication ;

subtype_Indication ::= [ *resolution_function*_name ] type_mark [ constraint ]

suffix ::=
    simple_name
    | character_literal
    | operator_syrnbol
    | **all**

target ::= name | aggregate

term ::= factor { multiplying_operator factor}

timeout_clause := **for** *time*_expression

type_conversion ::= type_mark ( expression )

type_declaration ::= full_type_declaration | incomplete_type_declaration

type_definition ::=
    scalar_type_definition
    | composite_type_definition
    | access_type_definition
    | file_type_definition

type_mark ::= *type*_name | *subtype*_name

unconstrained_array_definition ::=

**array** ( index_subtype_definition {, index_subtype_definition } )
**of** *element*_subtype_indication

use_clause ::= **use** selected_name {, selected_name } ;

variable_assignment_statement ::= target := expression ;

variable_declaration ::=
**variable** identifier_list : subtype_indication [ := expression ] ;

wait_statement ::=
**wait** [ sensitivity_clause ] [ condition_clause ] [ timeout_clause ] ;

waveform ::= waveform_element {, waveform_element}

waveform_element ::=
*value*_expression [ **after** *time*_expression ]
| **null** [ **after** *time*_expression ]

# APPENDIX C *Apackage Example*

This appendix describes the complete ATT_PACKAGE package that has been referred to in Chaps. II and 12. The package is provided more as an illustration of what goes into a package rather than trying to present a single comprehensive package.

## *C.1 The Package ATT_PACKAGE*

The ATT_PACKAGE contains the definition of a new 4-value type called MVL and its associated overloaded logical operator definitions. Here is the VHDL source code listing for this package.

```
package ATT_PACKAGE is
        type MVL is ('U', '0', '1', 'Z');
        type MVL_VECTOR is array (NATURAL range <>) of MVL;

        type MVL_1D_TABLE is array (MVL) of MVL;
        type MVL_2D_TABLE is array (MVL, MVL) of MVL;

        -- Truth tables for logical operators;
        constant TABLE_AND: MVL_2D_TABLE :=
        --      U       0       1       Z
             (( 'U',    '0',    'U',    'U'),    -- U
              ( '0',    '0',    '0',    '0'),    -- 0
              ( 'U',    '0',    '1',    'U'),    -- 1
              ( 'U',    '0',    'U',    'U'));   -- Z
        constant TABLE_OR: MVL_2D_TABLE :=
        --      U       0       1       Z
             (( 'U',    'U',    '1',    'U'),    -- U
              ( 'U',    '0',    '1',    'U'),    -- 0
              ( '1',    '1',    '1',    '1'),    -- 1
              ( 'U',    'U',    '1',    'U'));   -- Z
```

```
constant TABLE_NAND: MVL_2D_TABLE :=
--        U       0       1       Z
      (( 'U',    '1',    'U',    'U'),    -- U
       ( '1',    '1',    '1',    '1'),    -- 0
       ( 'U',    '1',    '0',    'U'),    -- 1
       ( 'U',    '1',    'U',    'U'));   -- Z
constant TABLE_NOR: MVL_2D_TABLE :=
--        U       0       1       Z
      (( 'U',    'U',    '0',    'U'),    -- U
       ( 'U',    '1',    '0',    'U'),    -- 0
       ( '0',    '0',    '0',    '0'),    -- 1
       ( 'U',    'U',    '0',    'U'));   -- Z
constant TABLE_XOR: MVL_2D_TABLE :=
--        U       0       1       Z
      (( 'U',    'U',    'U',    'U'),    -- U
       ( 'U',    '0',    '1',    'U'),    -- 0
       ( 'U',    '1',    '0',    'U'),    -- 1
       ( 'U',    'U',    'U',    'U'));   -- Z
constant TABLE_NOT: MVL_1D_TABLE :=
--        U       0       1       Z
       ( 'U',    '1',    '0',    'U');
constant TABLE_BUF: MVL_1D_TABLE :=
--        U       0       1       Z
       ( 'U',    '0',    '1',    'U');

-- Truth tables for resolution functions:
constant TABLE_TAND: MVL_2D_TABLE :=
--        U       0       1       Z
      (( 'U',    '0',    'U',    'U'),    -- U
       ( '0',    '0',    '0',    '0'),    -- 0
       ( 'U',    '0',    '1',    '1'),    -- 1
       ( 'U',    '0',    '1',    'Z'));   -- Z
constant TABLE_TOR: MVL_2D_TABLE :=
--        U       0       1       Z
      (( 'U',    'U',    '1',    'U'),    -- U
       ( 'U',    '0',    '1',    '0'),    -- 0
       ( '1',    '1',    '1',    '1'),    -- 1
       ( 'U',    '0',    '1',    'Z'));   -- Z

-- Overloaded logical operator declarations on MVL type:
function "and"  (L, R: MVL) return MVL;
function "or"   (L, R: MVL) return MVL;
function "nand" (L, R: MVL) return MVL;
function "nor"  (L, R: MVL) return MVL;
function "xor"  (L, R: MVL) return MVL;
function "not"  (L: MVL) return MVL;

-- Overloaded logical operator declarations on MVL_VECTOR type:
function "and"  (L, R: MVL_VECTOR) return MVL_VECTOR;
function "or"   (L, R: MVL_VECTOR) return MVL_VECTOR;
function "nand" (L, R: MVL_VECTOR) return MVL_VECTOR;
function "nor"  (L, R: MVL_VECTOR) return MVL_VECTOR;
function "xor"  (L, R: MVL_VECTOR) return MVL_VECTOR;
function "not"  (L: MVL_VECTOR) return MVL_VECTOR;

-- Common utilities:
function MAX   (T1, T2: TIME) return TIME;
function INT2MVL (OPD, NO_BITS: INTEGER)
         return MVL_VECTOR;
function MVL21NT (OPD: MVL_VECTOR) return INTEGER;

-- Overloaded arithmetic operators on MVL_VECTOR:
function "+"    (L, R: MVL_VECTOR) return MVL_VECTOR;
function "*"    (L, R: MVL_VECTOR) return MVL_VECTOR;

-- Clock functions:
```

```
        function ES_RISING    (signal CLOCK_NAME: MVL)
                return BOOLEAN;
        function ES_FALLING   (signal CLOCK_NAME: MVL)
                return BOOLEAN;

        -- Bus resolution functions:
        function WIRED_AND (SIG_DRIVERS: MVL_VECTOR)
                return MVL;
        function WIRED_OR (SIG_DRIVERS: MVL_VECTOR)
                return MVL;
end ATT_PACKAGE;

package body ATT_PACKAGE is
        -- Function definitions for overloaded operators with MVL types:
        function "and" (L, R: MVL) return MVL Is
        begin
                return TABLE_AND(L, R);
        end "and";
        function "or" (L, R; MVL) return MVL is
        begin
                return TABLE_OR(L, R);
        end "or";
        function "nand" (L, R; MVL) return MVL is
        begin
                return TABLE_NAND(L, R);
        end "nand";
        function "nor" (L, R: MVL) return MVL is
        begin
                return TABLE_NOR(L, R);
        end "nor";
        function "xor" (L, R: MVL) return MVL is
        begin
                return TABLE_XOR(L, R);
        end "xor";
        function "not" (L: MVL) return MVL is
        begin
                return TABLE_NOT(L);
        end "not";

        -- Function definitions for MVL_VECTOR types:
        function "and" (L, R: MVL_VECTOR) return MVL_VECTOR is
                variable RESULT: MVL_VECTOR(L'LENGTK-1 downto 0);
        begin
                assert L'LENGTH = R'LENGTH;
                for K in RESULT'RANGE loop
                        RESULT(K) :=TABLE_AND (L(K), R(K));
                end loop;
                return RESULT;
        end "and";
        function "or" (L, R: MVL_VECTOR) return MVL_VECTOR is
                variable RESULT; MVL_VECTOR(L'LENGTH-1 downto 0);
        begin
                assert L'LENGTH = R'LENGTH;
                for K in RESULT'RANGE loop
                        RESULT(K) := TABLE_OR (L(K), R(K));
                end loop;
                return RESULT;
        end "or";
        function "nand" (L, R: MVL_VECTOR) return MVL_VECTOR is
                variable RESULT: MVL_VECTOR(L'LENGTH-1 downto 0);
        begin
                assert L'LENGTH = R'LENGTH;
                for K in RESULT'RANGE loop
                        RESULT(K) :=TABLE_NAND (L(K), R(K));
                end loop;
```

```
          return RESULT;
end "nand";
function "nor" (L, R: MVL_VECTOR) return MVL_VECTOR is
          variable RESULT: MVL_VECTOR(L'LENGTH-1 downto 0);
begin
          assert L'LENGTH = R'LENGTH;
          for K in RESULT'RANGE loop
                    RESULT(K) := TABLE_NOR (L(K), R(K));
          end loop;
          return RESULT;
end "nor";
function "xor" (L, R: MVL_VECTOR) return MVL_VECTOR is
          variable RESULT: MVL_VECTOR(L'LENGTH-1 downto 0);
begin
          assert L'LENGTH = R'LENGTH;
          for K in RESULT'RANGE loop
                    RESULT(K) := TABLE_XOR (L(K), R(K));
          end loop;
          return RESULT;
end "xor";
function "not" (L: MVL_VECTOR) return MVL_VECTOR is
          variable RESULT: MVL_VECTOR(L'LENGTH-1 downto 0);
begin
          for K in RESULT'RANGE loop
                    RESULT(K) := TABLE_NOT (L(K));
          end loop;
          return RESULT;
end "not";

- Common utilities:
function MAX   (T1, T2: TIME) return TIME is
begin
          if (T1 > T2) then
                    return T1;
          else
                    return T2;
          end if;
end MAX;
function INT2MVL (OPD, NO_BITS: INTEGER)
                    return MVL_VECTOR is
          variable M1: INTEGER;
          variable RET: MVL_VECTOR (NO_BITS-1 downto 0)
                              := (others => '0');
begin
          assert OPD >= 0;
          M1 :=OPD;
          forJ in RET'REVERSE_RANGE loop
                    if (M1 mod 2) = 1 then
                              RET(J) := '1';
                    else
                              RET(J) := '0';
                    end if;
                    M1 :=M1/2;
          end loop;
          return RET;
end INT2MVL;
function MVL21NT (OPD: MVL_VECTOR) return INTEGER is
          -- Leftmost is LSB.
          variable TEMP, J: INTEGER;
 begin
          assert OPD'LENGTH< 32;
          TEMP := 0;
          J:=OPD'LENGTH - 1;
          for M3 in OPD'RANGE loop
                    if OPD(M3) = '1' then
```

```
                    TEMP:=TEMP+2**J;
               end if;
               J:=J - 1;
          end loop;
          return TEMP;
end MVL21NT;

-- Overloaded arithmetic operators on MVL_VECTOR:
function "+"    (L, R: MVL_VECTOR) return MVL_VECTOR is
          -- Assume 0 is LSB; unsigned numbers.
          variable SUM: MVL_VECTOR (L-LENGTH-1 downto 0);
          variable CARRY: MVL := '0';
begin
          assert L'LENGTH = R'LENGTH;
          for J in SUM'REVERSE_RANGE loop
               SUM(J) := L(J) xor R(J) xor CARRY;
               CARRY := (L(J) and R(J)) or (L(J) and CARRY) or
                    (R(J) and CARRY);
          end loop;
          return SUM;
end "+";
function "*'    (L, R: MVL_VECTOR) return MVL_VECTOR is
          -- Unsigned numbers being multiplied; result < 32 bits.
          variable T1, T2, R_SIZE: INTEGER;
begin
          assert L'LENGTH < 32 and R'LENGTH < 32;
          T1 := MVL21NT (L);
          T2 := MVL21NT (R);
          R_SIZE = L'LENGTH + R'LENGTH;
          if (R_SIZE > 31) then
               R_SIZE := 31;
          end if;
          return INT2MVL (T1 * T2, R_SIZE);
end "*";

-- Clock functions:
function ES_RISING    (signal CLOCK_NAME: MVL)
               return BOOLEAN is
begin
          return (CLOCK_NAME = '1') and (CLOCK_NAME'EVENT);
end ES_RISING;
function ES_FALLINQ   (signal CLOCK_NAME: MVL)
               return BOOLEAN is
begin
          return (CLOCK_NAME = -0') and (CLOCK_NAME'EVENT);
end ES_FALLINQ;

-- Bus resolution functions:
function WIRED_AND (SIQ_DRIVERS: MVL_VECTOR)
               return MVL is
          constant MEMORY: MVL := 'Z';
          variable RESOLVE_VALUE: MVL;
          variable FIRST: BOOLEAN := TRUE;
begin
          if SIG_DRIVERS'LENGTH = 0 then
               return MEMORY;
          else
               for I in SIG_DRIVERS'RANGE loop
                    if (FIRST = TRUE) then
                         RESOLVE_VALUE := SIG_DRIVERS(I);
                         FIRST := FALSE;
                    else
                         RESOLVE_VALUE := TABLE_TAND
                              (RESOLVE_VALUE,
                         SIG_DRIVERS(I));
```

```
                end if;
            end loop;
            return RESOLVE_VALUE;
        end if;
    end WIRED_AND;
    function WIRED_OR (SIG_DR[VERS: MVL_VECTOR)
            return MVL is
        constant MEMORY: MVL := 'Z;
        variable RESOLVE_VALUE: MVL;
        variable FIRST: BOOLEAN := TRUE;
    begin
        if SIGJRIVERS'LENGTH =0 then
            return MEMORY;
        else
            for K in SIGJRIVERS'RANGE loop
                if (FIRST = TRUE) then
                    RESOLVE_VALUE                          :=
                SIG_DRIVERS(K);
                    FIRST := FALSE:
                else
                    RESOLVE_VALUE := TABLE_TOR
                        (RESOLVE_VALUE,
                    SIG_DRIVERS(K));
                end if;
            end loop;
            return RESOLVE_VALUE;
        end if;
    end WIRED_OR;
end ATT_PACKAGE;
```

# *Bibliograpliy*

Following is a list of suggested readings and books on the language. The list is not intended to be comprehensive.


**1.** Armstrong, J. R,, *Chip-level Modeling with VHDL,* Englewood Cliffs, NJ: Prentice Hall, 1988.

**2.** Armstrong, J.R. et al.. *The VHDL validation suite,* Proc. 27th Design Automation Conference, June 1990, pp. 2-7.

**3.** Barton, D., *Afirst course in VHDL,* VLSI Systems Design, January 1988.

**4.** Bhasker, J., *Process-Graph Analyzer: Afront-end tool for VHDL behavioral synthesis,* Software Practice and Experience, vol. 18, no. 5, May 1988.

**5.** Bhasker, J., *An algorithm for microcode compaction of VHDL behavioral descriptions,* Proc. 20th Microprogramming Workshop, December 1987.

**6.** Coelho, D., *The VHDL handbook,* Boston: Kluwer Academic, 1988.

**7.** Coelho, D., *VHDL: A call for standards,* Proc. 25th Design Automation Conference, June 1988.

**8.** Farrow, R., and A. Stanculescu, *A VHDL compiler based on attribute grammar methodology,* SIGPLAN 1989.

**9.** Gilman, A.S., *Logic Modeling in WAVES,* IEEE Design & Test of Computers, June 1990, pp. 49-55.

**10.** Hands, J.P, *What is VHDL?* Computer-Aided Design, vol. 22, no. 4, May 1990.

**11.** Hines, J., *Where VHDL fits within the CAD environment,* Proc. 24th Design Automation Conference, 1987.

**12.** *IEEE Standard VHDL Language Reference Manual, Std 1076-1987,* IEEE, NY, 1988.

**13.** *IEEE Standard 1076 VHDL Tutorial,* CLSI, Maryland, March 1989.

**14.** Kirn, K., and J. Trout, *Automatic insertion of BIST hardware using VHDL,* Proc. 25th Design Automation Conference, 1988.

**15.** Leung, *ASIC system design with VHDL,* Boston: Kluwer Academic, 1989.

**16.** Lipsett, R., et. Al., *VHDL: Hardware description and design,* Boston: Kluwer Academic, 1989.

**17.** Moughzail, M,, et. al,, *Experience with the VHDL environment,* Proc. 25th Design Automation Conference, 1988.

**16.** Perry, D., *VHDL,* New York: McGraw Hill, 1991.

**19.** *Military Standard 454,* 1988 US Government Printing Office.

**20.** Saunders, L., *The IBM VHDL design system,* Proc. 24th Design Automation Conference, 1987.

**21.** Schoen, J.M., *Performance and fault modeling with VHDL,* Englewood Cliffs, NJ: Prentice Hall, 1992.

**22.** Ward, P.C., and J. Armstrong, *Behavioralfault simulation in VHDL,* Proc. 27th Design Automation Conference, June 1990, pp. 587-593.