Sequential statements like \( A := 3 \) are interpreted one after another, in the order in which they are written. VHDL sequential statements can appear only in a process or subprogram. A VHDL process is a group of sequential statements; a subprogram is a procedure or function.

To familiarize yourself with sequential statements, consider the following:

- **Assignment Statements**
- **Variable Assignment Statement**
- **Signal Assignment Statement**
- **if Statement**
- **case Statement**
- **loop Statements**
- **next Statement**
- **exit Statement**
- **Subprograms**
return Statement

wait Statement

null Statement

Processes are composed of sequential statements, but processes are themselves concurrent statements (see Chapter 7). All processes in a design execute concurrently. However, at any given time only one sequential statement is interpreted within each process.

A process communicates with the rest of a design by reading or writing values to and from signals or ports declared outside the process.

Sequential algorithms can be expressed as subprograms and called sequentially (as described in this chapter) or concurrently (as described in Chapter 7).

Sequential statements are

assignment statements that assign values to variables and signals.

flow control statements that conditionally execute statements (if and case), repeat statements (for...loop), and skip statements (next and exit).

subprograms that define sequential algorithms for repeated use in a design (procedure and function).

wait statement to pause until an event occurs (wait)

null statement to note that no action is necessary (null).
Assignment Statements

An assignment statement assigns a value to a variable or signal. The syntax is

```
target := expression;  -- Variable assignment
target <= expression;  -- Signal assignment
```

target is a variable or signal (or part of a variable or signal, such as a subarray) that receives the value of the expression. The expression must evaluate to the same type as the target. See Chapter 5 for more information on expressions.

The difference in syntax between variable assignments and signal assignments is that variables use := and signals use <=. The basic semantic difference is that variables are local to a process or subprogram, and their assignments take effect immediately.

Signals need not be local to a process or subprogram, and their assignments take effect at the end of a process. Signals are the only means of communication between processes. For more information on semantic differences, see “Signal Assignment,” later in this chapter.
Assignment Targets

Assignment statements have five kinds of targets:

- Simple names, such as `my_var`
- Indexed names, such as `my_array_var(3)`
- Slices, such as `my_array_var(3 to 6)`
- Field names, such as `my_record.a_field`
- Aggregates, such as `(my_var1, my_var2)`

A assignment target can be either a variable or a signal; the descriptions below refer to both.

Simple Name Targets

The syntax for an assignment to a simple name target is

```
identifier := expression;  -- Variable assignment
identifier <= expression;  -- Signal assignment
```

`identifier` is the name of a signal or variable. The assigned expression must have the same type as the signal or variable. For array types, all elements of the array are assigned values.

Example 6–1 shows some assignments to simple name targets.

**Example 6–1   Simple Name Targets**

```
variable A, B: BIT;
signal   C:    BIT_VECTOR(1 to 4);

-- Target    Expression
 A := '1';    -- Variable A is assigned '1'
 B := '0';    -- Variable B is assigned '0'
 C <= "1100"; -- Signal array C is assigned  -- "1100"
```
Indexed Name Targets

The syntax for an assignment to an indexed name target is

```
identifier(index_expression) := expression;
-- Variable assignment
```

```
identifier(index_expression) <= expression;
-- Signal assignment
```

*identifier* is the name of an array type signal or variable.

*index_expression* must evaluate to an index value for the

*identifier* array’s index type and bounds. It does not have to

be computable (see “Computable Operands” in Chapter 5),

but more hardware is synthesized if it is not.

The assigned *expression* must have the array’s element type.

In Example 6–2, array variable *A*’s elements are assigned

values as indexed names.

**Example 6–2   Indexed Name Targets**

```
variable A: BIT_VECTOR(1 to 4);
```

```
-- Target     Expression;
A(1)  := '1';   -- Assigns '1' to the first
               --    element of array A.
A(2)  := '1';   -- Assigns '1' to the second
               --    element of array A.
A(3)  := '0';   -- Assigns '0' to the third
               --    element of array A.
A(4)  := '0';   -- Assigns '0' to the fourth
               --    element of array A.
```

Example 6–3 shows two indexed name targets. One is computable, the other is not. Note the differences in the hardware generated for each assignment.
Example 6–3  Computable and Noncomputable Indexed Name Targets

signal A, B: BIT_VECTOR(0 to 3);
signal I: INTEGER range 0 to 3;
signal Y, Z: BIT;

A    <= "0000";
B    <= "0000";
A(I) <= Y;  -- Non-computable index expression
B(3) <= Z;  -- Computable index expression
Slice Targets

The syntax for a slice target is

\[ \text{identifier}(\text{index\_expr\_1} \ \text{direction} \ \text{index\_expr\_2}) \]

\text{identifier} is the name of an array type signal or variable. Each \text{index\_expr} expression must evaluate to an index value for the \text{identifier} array’s index type and bounds. Both \text{index\_expr} expressions must be computable (see “Computable Operands,” Chapter 5), and must lie within the bounds of the array. \text{direction} must match the \text{identifier} array type’s direction, either \text{to} or \text{downto}.

The assigned expression must have the array’s element type.

In Example 6–4, array variables \text{A} and \text{B} are assigned the same value.

\textbf{Example 6–4 \ Slice Targets}

variable \text{A}, \text{B}: \text{BIT\_VECTOR}(1 \ to \ 4);

\begin{verbatim}
-- Target \ Expression;
A(1 to 2) := "11"; \-- Assigns "11" to the first
\hspace{1cm} \hspace{1cm} \-- two elements of array \text{A}
A(3 to 4) := "00"; \-- Assigns "00" to the last
\hspace{1cm} \hspace{1cm} \-- two elements of array \text{A}
B(1 to 4) := "1100"; \-- Assigns "1100" to array \text{B}
\end{verbatim}
Field Targets

The syntax for a field target is

\[ \text{identifier.field\_name} \]

\textit{identifier} is the name of a record type signal or variable, and \textit{field\_name} is the name of a field in that record type, preceded by a period (\texttt{.}). The assigned expression must have the identified field’s type. A field can be of any type, including an array, record, or aggregate type.

Example 6-5 assigns values to the fields of record variables \texttt{A} and \texttt{B}.

\begin{verbatim}
Example 6-5 Field Targets

\footnotesize
type REC is
  record
    NUM_FIELD: INTEGER range –16 to 15;
    ARRAY_FIELD: BIT_VECTOR(3 to 0);
  end record;

variable A, B: REC;

-- Target Expression;
A.NUM_FIELD := –12; -- Assigns –12 to record A’s
                  -- field NUM\_FIELD

A.ARRAY_FIELD := “0011”; -- Assigns ”0011” to record
                      -- A’s field ARRAY\_FIELD

A.ARRAY_FIELD(3) := ‘1’; -- Assigns ‘1’ to the most-
                          -- significant bit of record
                          -- A’s field ARRAY\_FIELD

B := A; -- Assigns values of record
         -- A to corresponding fields
         -- of B

For more information, see ”Record Types” in Chapter 4.
\end{verbatim}
Aggregate Targets

The syntax for an assignment to an aggregate target is

```
function => identifier
{function => identifier}) := array_expression;
```

-- Variable assignment

```
function => identifier
{function => identifier}) <= array_expression;
```

-- Signal assignment

An aggregate assignment assigns `array_expression`'s element values to one or more variable or signal identifiers.

Each (optional) `function` is an index expression selecting an element or a slice of the assigned `array_expression`. Each `identifier` must have `array_expression`'s element type. An `identifier` can be an array type.

Example 6–6 shows some aggregate targets.

**Example 6–6   Aggregate Targets**

```vhdl
signal A, B, C, D: BIT;
signal S: BIT_VECTOR(1 to 4);
.
variable E, F: BIT;
variable G: BIT_VECTOR(1 to 2);
variable H: BIT_VECTOR(1 to 4);

-- Positional notation
S <= ('0', '1', '0', '0');
(A, B, C, D) <= S; -- Assigns '0' to A
                -- Assigns '1' to B
                -- Assigns '0' to C
                -- Assigns '0' to D

-- Named notation
(3 => E, 4 => F,
  2 => G(1), 1 => G(2)) := H;
                -- Assigns H(1) to G(2)
                -- Assigns H(2) to G(1)
                -- Assigns H(3) to E
                -- Assigns H(4) to F
```
You can assign array element values to the identifiers by position or by name. In positional notation, the `choice =>` construct is not used. Identifiers are assigned array element values in order, from the left array bound to the right array bound.

In named notation, the `choice =>` construct identifies specific elements of the assigned array. A `choice index expression` indicates a single element (such as `3`). The identifier’s type must match the assigned expression’s element type.

Positional and named notation can be mixed, but positional associations must come before named associations.

**Variable Assignment Statement**

A variable assignment changes the value of a variable. The syntax is

```vhdl
target := expression;
```

`expression` determines the assigned value; its type must be compatible with `target`. Expressions are described in Chapter 5. `target` names the variables that receive the value of `expression`. See "Assignment Targets" in the previous section for a description of variable assignment targets.

When a variable is assigned a value, the assignment takes place immediately. A variable keeps its assigned value until another assignment.
Signal Assignment Statement

A signal assignment changes the value being driven on a signal by the current process. The syntax is

\[ \text{target} \leftarrow \text{expression}; \]

expression determines the assigned value; its type must be compatible with target. Expressions are described in Chapter 5. target names the signals that receive the value of expression. See "Assignment Targets" in this chapter for a description of signal assignment targets.

Signals and variables behave differently when they are assigned values. The differences lie in the way the two kinds of assignments take effect, and how that affects the values read from either variables or signals.

Variable Assignment

When a variable is assigned a value, the assignment changes the value of the variable from that point on, and it is kept until the variable is assigned a different value.
Signal Assignment

When a signal is assigned a value, the assignment does not necessarily take effect because the value of a signal is determined by the processes (or other concurrent statements) that drive it.

- If several values are assigned to a given signal in one process, only the last assignment is effective. Even if a signal in a process is assigned, then read, then assigned again, the value read (either inside or outside the process) is the last assignment value.

- If several processes (or other concurrent statements) assign values to one signal, the drivers are wired together. The resulting circuit depends on the expressions and the target technology. It may be invalid, wired-AND, wired-OR, or a three-state bus. Refer to “Driving Signals” in Chapter 7 for more information.
Example 6-7 shows the different effects of variable and signal assignments.

Example 6-7  Signal and Variable Assignments

```vhdl
signal S1, S2: BIT;
signal S_OUT: BIT_VECTOR(1 to 8);
...
process( S1, S2 )
  variable V1, V2: BIT;
begin
  V1 := '1';   -- This sets the value of V1
  V2 := '1';   -- This sets the value of V2
  S1 <= '1';   -- This assignment is the driver for S1
  S2 <= '1';   -- This has no effect because of the
               -- assignment later in this process
  S_OUT(1) <= V1; -- Assigns '1', the value assigned above
  S_OUT(2) <= V2; -- Assigns '1', the value assigned above
  S_OUT(3) <= S1; -- Assigns '1', the value assigned above
  S_OUT(4) <= S2; -- Assigns '0', the value assigned
                   -- below
  V1 := '0';   -- This sets the new value of V1
  V2 := '0';   -- This sets the new value of V2
  S2 <= '0';   -- This assignment overrides the
               -- previous one since it is the last
               -- assignment to this signal in this
               -- process
  S_OUT(5) <= V1; -- Assigns '0', the value assigned above
  S_OUT(6) <= V2; -- Assigns '0', the value assigned above
  S_OUT(7) <= S1; -- Assigns '1', the value assigned above
  S_OUT(8) <= S2; -- Assigns '0', the value assigned above
end process;
```
if Statement

The if statement executes a sequence of statements. The sequence depends on the value of one or more conditions. The syntax is

```vhdl
if condition then
    { sequential_statement }
{ elsif condition then
    { sequential_statement } }
[ else
    { sequential_statement } ]
end if;
```

Each condition must be a Boolean expression. Each branch of an if statement can have one or more sequential statements.

Evaluating condition

An if statement evaluates each condition in order. The first (and only the first) TRUE condition causes the execution of its branch's statements. The remainder of the if statement is skipped.

If none of the conditions is TRUE, and the else clause is present, those statements are executed.

If none of the conditions is TRUE, and no else is present, none of the statements is executed.

Example 6-8 shows an if statement and a corresponding circuit.
Example 6-8  if Statement

signal A, B, C, P1, P2, Z: BIT;

if (P1 = '1') then
    Z <= A;
elsif (P2 = '0') then
    Z <= B;
else
    Z <= C;
end if;

Using the if Statement to Imply Registers and Latches

Some forms of the if statement can be used like the wait statement, to test for signal edges and therefore imply synchronous logic. This usage causes VHDL Compiler to infer registers or latches, as described in Chapter 8 under “Register and Three-State Inference.”
case Statement

The `case` statement executes one of several sequences of statements, depending on the value of a single expression. The syntax is

```vhdl
case expression is
  when choices =>
    { sequential_statement }
  when choices =>
    { sequential_statement }
end case;
```

`expression` must evaluate to an `INTEGER`, an enumerated type, or an array of enumerated types such as `BIT_VECTOR`. Each of the `choices` must be of the form

```vhdl
choice { | choice }
```

Each `choice` can be either a static expression (such as `3`) or a static range (such as `1 to 3`). The type of `choice_expression` determines the type of each `choice`. Each value in the range of `choice_expression`'s type must be covered by one `choice`.

The final `choice` can be `others`, which matches all remaining (unchosen) values in the range of `expression`'s type. The `others` choice, if present, matches `expression` only if no other choices match.

The `case` statement evaluates `expression` and compares that value to each `choice` value. The `when` clause with the matching `choice` value has its statements executed.

The following restrictions are placed on choices:

- No two choices can overlap.
- If no `others` choice is present, all possible values of `expression` must be covered by the set of choices.
Using Different Expression Types

Example 6–9 shows a case statement that selects one of four signal assignment statements by using an enumerated expression type.

Example 6–9    case Statement That Uses an Enumerated Type

```vhdl
type ENUM is (PICK_A, PICK_B, PICK_C, PICK_D);
signal VALUE: ENUM;

signal A, B, C, D, Z: BIT;

case VALUE is
    when PICK_A =>
        Z <= A;
    when PICK_B =>
        Z <= B;
    when PICK_C =>
        Z <= C;
    when PICK_D =>
        Z <= D;
end case;
```

![Diagram of a multiplexer with inputs A, B, C, D, and outputs VALUE[0] and VALUE[1] connected to a multiplexer, with output Z connected to the multiplexer]
Example 6-10 shows a case statement again used to select one of four signal assignment statements, this time by using an integer expression type with multiple choices.

**Example 6-10 case Statement with Integers**

```vhdl
signal VALUE is INTEGER range 0 to 15;
signal Z1, Z2, Z3, Z4: BIT;

Z1 <= '0';
Z2 <= '0';
Z3 <= '0';
Z4 <= '0';

case VALUE is
  when 0 => -- Matches 0
    Z1 <= '1';
  when 1 | 3 => -- Matches 1 or 3
    Z2 <= '1';
  when 4 to 7 | 2 => -- Matches 2, 4, 5, 6, or 7
    Z3 <= '1';
  when others => -- Matches remaining values, 8 through 15
    Z4 <= '1';
end case;
```

For further assistance, email support_center@synopsys.com or call your local support center.
Invalid case Statements

Example 6-11 shows four invalid case statements, with explanations.

Example 6-11 Invalid case Statements

signal VALUE: INTEGER range 0 to 15;
signal OUT_1: BIT;

case VALUE is
  -- Must have at least one when clause
  when 0 to 10 =>
    OUT_1 <= '1';
  when 5 to 15 =>
    OUT_1 <= '0';
end case;

case VALUE is
  -- Values 2 to 15 are not covered by choices
  when 0 =>
    OUT_1 <= '1';
  when 1 =>
    OUT_1 <= '0';
end case;

case VALUE is
  -- Choices 5 to 10 overlap
  when 0 to 10 =>
    OUT_1 <= '1';
  when 5 to 15 =>
    OUT_1 <= '0';
end case;
loop Statements

A loop statement repeatedly executes a sequence of statements. The syntax is

```
[label :] [iteration_scheme] loop
  { sequential_statement }
  { next [ label ] [ when condition ] ; }
  { exit [ label ] [ when condition ] ; }
end loop [label];
```

The optional label names the loop and is useful for building nested loops. Each type of iteration_scheme is described in this section.

The next and exit statements are sequential statements used only within loops. The next statement skips the remainder of the current loop and continues with the next loop iteration. The exit statement skips the remainder of the current loop and continues with the next statement after the exited loop.

VHDL provides three types of loop statements, each with a different iteration scheme:

- **loop**
  The basic loop statement has no iteration scheme. Enclosed statements are executed repeatedly forever until an exit or next statement is encountered.

- **while .. loop**
  The while .. loop statement has a Boolean iteration scheme. If the iteration condition evaluates true, enclosed statements are executed once. The iteration condition is then reevaluated. Although the iteration condition remains true, the loop is repeatedly executed. When the iteration condition evaluates false, the loop is skipped, and execution continues with the next statement after the loop.
for .. loop
The for .. loop statement has an integer iteration scheme, where the number of repetitions is determined by an integer range. The loop is executed once for each value in the range. After the last value in the iteration range is reached, the loop is skipped, and execution continues with the next statement after the loop.

Note:
Noncomputable loops (loop and while..loop statements) must have at least one wait statement in each enclosed logic branch. Otherwise, a combinational feedback loop is created. See “wait Statement,” later in this chapter, for more information.

Conversely, computable loops (for..loop statements) must not contain wait statements. Otherwise, a race condition may result.
**loop Statement**

The `loop` statement, with no iteration scheme, repeats enclosed statements forever. The syntax is

```vhdl
[label : ] loop
  { sequential_statement }
end loop [label];
```

The optional `<label>` names this loop.

`sequential_statement` can be any statement described in this chapter. Two sequential statements are used only with loops: the `next` statement, which skips the remainder of the current loop iteration, and the `exit` statement, which terminates the loop. These statements are described in the next two sections.

*Note:*

A `loop` statement must have at least one `wait` statement in each enclosed logic branch. See `wait Statement`, later in this chapter, for an example.

**while .. loop Statement**

The `while .. loop` statement repeats enclosed statements as long as its iteration condition evaluates true. The syntax is

```vhdl
[label : ] while condition loop
  { sequential_statement }
end loop [label];
```

The optional `<label>` names this loop. `condition` is any Boolean expression, such as `((A = '1') or (X < Y))`.

`sequential_statement` can be any statement described in this chapter. Two sequential statements are used only with loops: the `next` statement, which skips the remainder of the current loop iteration, and the `exit` statement, which terminates the loop. These statements are described in the next two sections.
Note:
A while..loop statement must have at least one wait statement in each enclosed logic branch. See “wait Statement,” later in this chapter, for an example.

for .. loop Statement

The for .. loop statement repeats enclosed statements once for each value in an integer range. The syntax is

[label : ] for identifier in range loop
  { sequential_statement }
end loop [label];

The optional label names this loop.

The use of identifier is specific to the for .. loop statement:

- identifier is not declared elsewhere. It is automatically declared by the loop itself, and is local to the loop. A loop identifier overrides any other identifier with the same name, but only within the loop.

- identifier’s value can be read only inside its loop (identifier does not exist outside the loop). You cannot assign a value to a loop identifier.

VHDL Compiler currently requires that range must be a computable (see “Computable Operands” in Chapter 5) integer range, in either of two forms:

integer_expression to integer_expression

integer_expression downto integer_expression

Each integer_expression evaluates to an integer.
sequential_statement can be any statement described in this chapter. Two sequential statements are used only with loops: the next statement, which skips the remainder of the current loop iteration, and the exit statement, which terminates the loop. These statements are described in the next two sections.

**Note:**

A for..loop statement must not contain any wait statements.

A for .. loop statement executes as follows:

1. A new, local, integer variable is declared with the name identifier.
2. identifier is assigned the first value of range, and the sequence of statements is executed once.
3. identifier is assigned the next value in range, and the sequence of statements is executed once more.
4. Step 3 is repeated until identifier is assigned to the last value in range. The sequence of statements is then executed for the last time, and execution continues with the statement following end loop. The loop is then inaccessible.
Example 6-12 shows two equivalent code fragments.

**Example 6–12  for..loop Statement with Equivalent Fragment**

variable A, B: BIT_VECTOR(1 to 3);

-- First fragment is a loop statement
for I in 1 to 3 loop
    A(I) <= B(I);
end loop;

-- Second fragment is three equivalent statements
A(1) <= B(1);
A(2) <= B(2);
A(3) <= B(3);

You can use a loop statement to operate on all elements of an array, without explicitly depending on the size of the array. Example 6–13 shows how the VHDL array attribute ’range can be used, in this case to invert each element of bit vector A.
Example 6-13  for..loop Statement Operating on an Entire Array

variable A, B: BIT_VECTOR(1 to 10);

for I in A'range loop
  A(I) := not B(I);
end loop;

Unconstrained arrays and array attributes are described under "Array Types" in Chapter 4.
**next Statement**

The next statement terminates the current iteration of a loop, then continues with the first statement in the loop. The syntax is

```vhdl
next [ label ] [ when condition ] ;
```

A next statement with no label terminates the current iteration of the innermost enclosing loop. When you specify a loop label, the current iteration of that named loop is terminated.

The optional when clause executes its next statement when its condition (a BOOLEAN expression) evaluates TRUE.

Example 6–14 uses the next statement to copy bits conditionally from bit vector `B` to bit vector `A` only when the next condition evaluates as true.

**Example 6–14  next Statement**

```vhdl
signal A, B, COPY_ENABLE: BIT_VECTOR (1 to 8);

. .
A <= "00000000";
. .
-- B is assigned a value, such as "01011011"
-- COPY_ENABLE is assigned a value, such as "11010011"
. .
for I in 1 to 8 loop
  next when COPY_ENABLE(I) = ’0’;
  A(I) <= B(I);
end loop;
```
Example 6–15 shows the use of nested `next` statements in named loops. This example processes:

- The first element of vector $x$ against the first element of vector $y$,
- The second element of $x$ against each of the first two elements of $y$, 

![Diagram of the VHDL code example](image-url)
The third element of \( x \) against each of the first three elements of \( y \),

The processing continues in this fashion until it is completed.

**Example 6–15  Named next Statement**

```vhdl
signal X, Y: BIT_VECTOR(0 to 7);

A_LOOP: for I in X'range loop
  ...
  B_LOOP: for J in Y'range loop
    ...
    next A_LOOP when I < J;
    ...
  end loop B_LOOP;
  ...
end loop A_LOOP;
```

**exit Statement**

The `exit` statement terminates a loop. Execution continues with the statement following `end loop`. The syntax is

```vhdl
exit [ label ] [ when condition ];
```

An `exit` statement with no `label` terminates the innermost enclosing loop. When you identify a loop `label`, that named loop is terminated, as shown previously in Example 6–15.

The optional `when` clause executes its `exit` statement when its `condition` (a `BOOLEAN` expression) evaluates `TRUE`. 
The `exit` and `next` statements are equivalent constructs; they have an identical syntax, and they both skip the remainder of the enclosing (or named) loop. The difference between them is that `exit` terminates its loop, and `next` continues with the next loop iteration (if any).

Example 6-16 compares two bit vectors. An `exit` statement exits the comparison loop when a difference is found.

Example 6–16 Comparator that Uses the `exit` Statement

```vhdl
signal A, B: BIT_VECTOR(1 downto 0);
signal A_LESS_THAN_B: BOOLEAN;

--
A_LESS_THAN_B <= FALSE;

for I in 1 downto 0 loop
  if (A(I) = '1' and B(I) = '0') then
    A_LESS_THAN_B <= FALSE;
    exit;
  elsif (A(I) = '0' and B(I) = '1') then
    A_LESS_THAN_B <= TRUE;
    exit;
  else
    null;  -- Continue comparing
  end if;
end loop;
```

![Diagram](image)
Subprograms

Subprograms are independent, named algorithms. A subprogram is either a procedure (zero or more in, inout, or out parameters) or a function (zero or more in parameters, one return value). Subprograms are called by name from anywhere within a VHDL architecture or a package body. Subprograms can be called sequentially (as described later in this chapter) or concurrently (as described in Chapter 7).

In hardware terms, a subprogram call is similar to module instantiation, except that a subprogram call becomes part of the current circuit. A module instantiation adds a level of hierarchy to the design. A synthesized subprogram is always a combinational circuit (use a process to create a sequential circuit).

Subprograms, like packages, have subprogram declarations and subprogram bodies. A subprogram declaration specifies its name, parameters, and return value (for functions). A subprogram body then implements the operation you want.

Often, a package contains only type and subprogram declarations for use by other packages. The bodies of the declared subprograms are then implemented in the bodies of the declaring packages.

The advantage of the separation between declarations and bodies is that subprogram interfaces can be declared in public packages during system development. One group of developers can use the public subprograms as another group develops the corresponding bodies. You can modify package bodies, including subprogram bodies, without affecting existing users of that package’s declarations.

You can also define subprograms locally inside an entity, block, or process.
VHDL Compiler implements procedure and function calls with combinational logic, unless you use the `map_to_entity` compiler directive (see “Mapping Subprograms to Components (Entities),” later in this chapter). VHDL Compiler does not allow inference of sequential devices, such as latches or flip-flops, in subprograms.

Example 6–17 shows a package containing some procedure and function declarations and bodies. The example itself is not synthesizable; it just creates a template. Designs that instantiate procedure P, however, compile normally.

**Example 6–17  Subprogram Declarations and Bodies**

```vhdl
package EXAMPLE is
  procedure P (A: in INTEGER; B: inout INTEGER);
    -- Declaration of procedure P
  function INVERT (A: BIT) return BIT;
    -- Declaration of function INVERT
end EXAMPLE;

package body EXAMPLE is
  procedure P (A: in INTEGER; B: inout INTEGER) is
    -- Body of procedure P
  begin
    B := A + B;
  end;

  function INVERT (A: BIT) return BIT is
    -- Body of function INVERT
  begin
    return (not A);
  end;
end EXAMPLE;

For more information about subprograms, see the “Subprograms” section in Chapter 3.
```
Subprogram Calls

Subprograms can have zero or more parameters. A subprogram declaration defines each parameter's name, mode, and type. These are a subprogram's formal parameters. When the subprogram is called, each formal parameter is given a value, termed the actual parameter. Each actual parameter's value (of an appropriate type) may come from an expression, a variable, or a signal.

The mode of a parameter specifies whether the actual parameter can be read from (mode in), written to (mode out), or both read from and written to (mode inout). Actual parameters that use modes out and inout must be variables or signals, including indexed names (A(1)) and slices (A(1 to 3)), but cannot be constants or expressions.

Two kinds of subprograms are procedure and function:

procedure
Can have multiple parameters that use modes in, inout, and out. Does not itself return a value.

Procedures are used when you want to update some parameters (modes out and inout), or when you do not need a return value. An example would be a procedure with one inout bit vector parameter that inverted each bit in place.

function
Can have multiple parameters, but only parameters that use mode in. Returns its own function value. Part of a function definition specifies its return value type (also called the function type).

Functions are used when you do not need to update the parameters and you want a single return value. For example, the arithmetic function ABS returns the absolute value of its parameter.
Procedure Calls

A procedure call executes the named procedure with the given parameters. The syntax is

```
procedure_name [ ( [ name => ] expression 
{ , [ name => ] expression } ) ] ;
```

Each `expression` is called an actual parameter; `expression` is often just an identifier. If a `name` is present (positional notation), it is a formal parameter name associated with the actual parameter's expression.

Formal parameters are matched to actual parameters by positional or named notation. Named and positional notation can be mixed, but positional parameters must come before named parameters.

Conceptually, a procedure call is performed in three steps. First, the values of the `in` and `inout` actual parameters are assigned to their associated formal parameters. Second, the procedure is executed. Third, the values of the `inout` and `out` formal parameters are assigned to the actual parameters.

In the synthesized hardware, the procedure’s actual inputs and outputs are wired to the procedure’s internal logic.

Example 6–18 shows a local procedure named `SWAP` that compares two elements of an array and exchanges them if they are out of order. `SWAP` is called repeatedly to sort an array of three numbers.
Example 6–18  Procedure Call to Sort an Array

package DATA_TYPES is
  type DATA_ELEMENT is range 0 to 3;
  type DATA_ARRAY is array (1 to 3) of DATA_ELEMENT;
end DATA_TYPES;

use WORK.DATA_TYPES.ALL;

entity SORT is
  port(IN_ARRAY:   in DATA_ARRAY;
       OUT_ARRAY: out DATA_ARRAY);
end SORT;

architecture EXAMPLE of SORT is
  begin

    process(IN_ARRAY)
      procedure SWAP(DATA:   inout DATA_ARRAY;
                     LOW, HIGH: in INTEGER) is
        variable TEMP: DATA_ELEMENT;
      begin
        if(DATA(LOW) > DATA(HIGH)) then  -- Check data
          TEMP := DATA(LOW);
          DATA(LOW) := DATA(HIGH);     -- Swap data
          DATA(HIGH) := TEMP;
        end if;
      end SWAP;

      variable MY_ARRAY: DATA_ARRAY;

      begin
        MY_ARRAY := IN_ARRAY;   -- Read input to variable

        -- Pair-wise sort
        SWAP(MY_ARRAY, 1, 2);   -- Swap first and second
        SWAP(MY_ARRAY, 2, 3);   -- Swap second and third
        SWAP(MY_ARRAY, 1, 2);   -- Swap 1st and 2nd again
        OUT_ARRAY <= MY_ARRAY;  -- Write result to output
      end process;
  end EXAMPLE;
**Function Calls**

A function call is similar to a procedure call, except that a function call is a kind of expression because it returns a value.

Example 6–19 shows a simple function definition and two calls to that function.

*Example 6–19   Function Call*

```vhdl
function INVERT (A : BIT) return BIT is
  begin
    return (not A);
  end;
...
process
  variable V1, V2, V3: BIT;
begin
  V1 := '1';
  V2 := INVERT(V1) xor 1;
  V3 := INVERT('0');
end process;
```

For more information, see “Function Calls,” under “Operands” in Chapter 5.
return Statement

The return statement terminates a subprogram. It is required in function definitions, and optional in procedure definitions. The syntax is

```
return expression ;        -- Functions
return ;                   -- Procedures
```

The required expression provides the function’s return value. Every function must have at least one return statement. The expression’s type must match the declared function type. A function can have more than one return statement. Only one return statement is reached by a given function call.

A procedure can have one or more return statements, but no expression is allowed. A return statement, if present, is the last statement executed in a procedure.

In Example 6–20, the function OPERATE returns either the and or the or of its parameters A and B. The return depends on the value of its parameter OPERATION.

**Example 6–20  Use of Multiple return Statements**

```
function OPERATE(A, B, OPERATION: BIT) return BIT is
begin
  if (OPERATION = ‘1’) then
    return (A and B);
  else
    return (A or B);
  end if;
end OPERATE;
```
Mapping Subprograms to Components (Entities)

In VHDL, entities cannot be invoked from within behavioral code. Procedures and functions cannot exist as entities (components), but must be represented by gates. You can overcome this limitation with the compiler directive `map_to_entity`, which causes VHDL Compiler to implement a function or procedure as a component instantiation. Procedures and functions that use `map_to_entity` are represented as components in designs where they are called.

You can also use the Design Compiler command `group -hdl_block` to create a new level of hierarchy from a VHDL subprogram, as described in Chapter 12.

When you add a `map_to_entity` directive to a subprogram definition, VHDL Compiler assumes the existence of an entity with the identified name and the same interface. Design Compiler does not check this assumption until it links the parent design. The matching entity must have the same input and output port names. If the subprogram is a function, you must also provide a `return_port_name` directive, where the matching entity has an output port of the same name.

These two directives are called component implication directives:
-- pragma map_to_entity  entity_name
-- pragma return_port_name  port_name

Insert these directives after the function or procedure definition. For example:

```vhdl
function MUX_FUNC(A,B: in TWO_BIT; C: in BIT) return TWO_BIT is
  -- pragma map_to_entity MUX_ENTITY
  -- pragma return_port_name Z
...
```

When VHDL Compiler sees the `map_to_entity` directive, it parses but ignores the contents of the subprogram definition. Use `-- pragma translate_off` and `-- pragma translate_on` to hide simulation–specific constructs in a `map_to_entity` subprogram.

**Note:**

The matching entity (`entity_name`) does not need to be written in VHDL. It can be in any format the Synopsys Design Compiler supports.

**Note:**

Be aware that the behavioral description of the subprogram is not checked against the functionality of the entity overloading it. Presynthesis and postsynthesis simulation results may not match if differences in functionality exist between the VHDL subprogram and the overloaded entity.

Example 6–21 shows a function that uses the component implication directives.
Example 6–21 Using Component Implication Directives on a Function

package MY_PACK is
  subtype TWO_BIT is BIT_VECTOR(1 to 2);
  function MUX_FUNC(A, B: in TWO_BIT; C: in BIT) return TWO_BIT;
end;

package body MY_PACK is
  function MUX_FUNC(A, B: in TWO_BIT; C: in BIT) return TWO_BIT is
    -- pragma map_to_entity MUX_ENTITY
    -- pragma return_port_name Z
    -- contents of this function are ignored but should
    -- match the functionality of the module MUX_ENTITY
    -- so pre- and post simulation will match
    begin
      if (C = '1') then
        return(A);
      else
        return(B);
      end if;
    end;
  end;

  use WORK.MY_PACK.ALL;

entity TEST is
  port(A: in TWO_BIT; C: in BIT; TEST_OUT: out TWO_BIT);
end;

architecture ARCH of TEST is
begin
  process
  begin
    TEST_OUT <= MUX_FUNC(not A, A, C);
    -- Component implication call
  end process;
end;
use WORK.MY_PACK.ALL;

-- the following entity ‘overloads’ the function
Example 6-22 shows the same design as Example 6-21, but without the creation of an entity for the function. The compiler directives have been removed.
Example 6-22  Using Gates to Implement a Function

package MY_PACK is
    subtype TWO_BIT is BIT_VECTOR(1 to 2);
    function MUX_FUNC(A,B: in TWO_BIT; C: in BIT)
        return TWO_BIT;
end;

package body MY_PACK is
    function MUX_FUNC(A,B: in TWO_BIT; C: in BIT)
        return TWO_BIT is
    begin
        if(C = '1') then
            return(A);
        else
            return(B);
        end if;
    end;
end;

use WORK.MY_PACK.ALL;

entity TEST is
    port(A: in TWO_BIT; C: in BIT; Z: out TWO_BIT);
end;

architecture ARCH of TEST is
begin
    process
    begin
        Z <= MUX_FUNC(not A, A, C);
    end process;
end;

A[1]
  C
A[2]
**wait Statement**

A `wait` statement suspends a process until a positive-going or negative-going edge is detected on a signal. The syntax is

```vhdl
wait until signal = value;
wait until signal'event and signal = value;
wait until not signal'stable and signal = value;
```

`signal` is the name of a single-bit signal—a signal of an enumerated type encoded with one bit (see "Enumeration Encoding" in Chapter 4). `value` must be one of the literals of the enumerated type. If the signal type is `BIT`, the awaited `value` is either `'1'` for a positive-going edge or `'0'` for a negative-going edge.

**Note:**

The three forms of the `wait` statement, a subset of IEEE VHDL, are specific to the current implementation of VHDL Compiler.
Inferring Synchronous Logic

A `wait` statement implies synchronous logic, where `signal` is usually a clock signal. The next section describes how VHDL Compiler infers and Design Compiler implements this logic.

Example 6–23 shows three equivalent `wait` statements (all positive-edge triggered).

**Example 6–23  Equivalent `wait` Statements**

```vhdl
wait until CLK = '1';
wait until CLK'event and CLK = '1';
wait until not CLK'stable and CLK = '1';
```

When a circuit is synthesized, the hardware in the three forms of `wait` statements does not differ.

Example 6–24 shows a `wait` statement used to suspend a process until the next positive edge (a 0-to-1 transition) on signal `CLK`.

**Example 6–24  `wait` for a Positive Edge**

```vhdl
signal CLK: BIT;
...
process
begin
  wait until CLK'event and CLK = '1';
  -- Wait for positive transition (edge)
  ...
end process;
```

**Note:**

IEEE VHDL specifies that a `process` containing a `wait` statement must not have a sensitivity list. See “Process Statements” in Chapter 7 for more information.

Example 6-25 shows how a `wait` statement is used to describe a circuit where a value is incremented on each positive clock edge.

**Example 6-25  Loop that Uses a `wait` Statement**

```vhdl
process
begin
  y <= 0;
  wait until (clk'event and clk = '1');
  while (y < MAX) loop
    wait until (clk'event and clk = '1');
    x <= y;
    y <= y + 1;
  end loop;
end process;
```

Example 6-26 shows how multiple `wait` statements describe a multicycle circuit. The circuit provides an average value of its input over four clock cycles.

**Example 6-26  Using Multiple `wait` Statements**

```vhdl
process
begin
  wait until CLK'event and CLK = '1';
  AVE <= A;
  wait until CLK'event and CLK = '1';
  AVE <= AVE + A;
  wait until CLK'event and CLK = '1';
  AVE <= AVE + A;
  wait until CLK'event and CLK = '1';
  AVE <= (AVE + A)/4;
end process;
```
Example 6-27 shows two equivalent descriptions, the first with implicit state logic, and the second with explicit state logic.

Example 6-27  wait Statements and State Logic

-- Implicit State Logic
process
begin
  wait until CLOCK'event and CLOCK = '1';
  if (CONDITION) then
    X <= A;
  else
    wait until CLOCK'event and CLOCK = '1';
  end if;
end process;

-- Explicit State Logic
...
type STATE_TYPE is (SO, S1);
variable STATE : STATE_TYPE;
...
process
begin
  wait until CLOCK'event and CLOCK = '1';
  case STATE is
    when S0 =>
      if (CONDITION) then
        X <= A;
        STATE := S0;  -- Set STATE here to avoid an
        -- extra feedback loop in the
        -- synthesized logic.
      else
        STATE := S1;
      end if;
    when S1 =>
      STATE := S0;
  end case;
end process;
Note:
wait statements can be used anywhere in a process except in for..loop statements or subprograms. However, if any path through the logic has one or more wait statements, all paths must have at least one wait statement.

Example 6–28 shows how a circuit with synchronous reset can be described by using wait statements in an infinite loop. The reset signal must be checked immediately after each wait statement. The assignment statements in Example 6–28 (X <= A; and Y <= B;) simply represent the sequential statements used to implement your circuit.

Example 6–28  Synchronous Reset That Uses wait Statements

process
begin
    RESET_LOOP: loop
        wait until CLOCK’event and CLOCK = ’1’;
        next RESET_LOOP when (RESET = ’1’);
        X <= A;
        wait until CLOCK’event and CLOCK = ’1’;
        next RESET_LOOP when (RESET = ’1’);
        Y <= B;
    end loop RESET_LOOP;
end process;
Example 6-29 shows two invalid uses of \texttt{wait} statements. These limitations are specific to VHDL Compiler.

\textit{Example 6-29  Invalid Uses of the \texttt{wait} Statement}

\begin{verbatim}
... 
  type COLOR is (RED, GREEN, BLUE); 
  attribute ENUM_ENCODING : STRING; 
  attribute ENUM_ENCODING of COLOR : type is "100 010 001"; 
  signal CLK : COLOR; 
...
  process 
    begin 
      wait until CLK'event and CLK = RED; 
      -- Illegal: clock type is not encoded with one bit 
    ... 
  end; 
...
  process 
    begin 
      if (X = Y) then 
        wait until CLK'event and CLK = '1'; 
        ... 
      end if; 
      -- Illegal: not all paths contain \texttt{wait} statements 
    ... 
  end; 
\end{verbatim}
**Combinational vs. Sequential Processes**

When a process has no `wait` statements, the process is synthesized with combinational logic. The computations performed by the process react immediately to changes in input signals.

When a process uses one or more `wait` statements, it is synthesized with sequential logic. The process computations are performed only once for each specified clock edge (positive or negative edge). The results of these computations are saved until the next edge by storing them in flip-flops.

The following values are stored in flip-flops:

- Signals driven by the process; see “Signal Assignment Statements” at the beginning of this chapter.
- State vector values, where the state vector may be implicit or explicit (as in Example 6-27).
- Variables that *may* be read before they are set.

**Note:**

Like the `wait` statement, some uses of the `if` statement can also imply synchronous logic, causing VHDL Compiler to infer registers or latches. These methods are described in Chapter 8, under “Register and Three-State Inference.”

Example 6-30 uses a `wait` statement to store values across clock cycles. The example code compares the parity of a data value with a stored value. The stored value (called `CORRECT_PARITY`) is set from the `NEW_CORRECT_PARITY` signal if the `SET_PARITY` signal is TRUE.
Example 6–30 Parity Tester That Uses the wait Statement

signal CLOCK: BIT;
signal SET_PARITY, PARITY_OK: BOOLEAN;
signal NEW_CORRECT_PARITY: BIT;
signal DATA: BIT_VECTOR(0 to 3);
...
process
    variable CORRECT_PARITY, TEMP: BIT;
begin
    wait until CLOCK’event and CLOCK = ’1’;

    -- Set new correct parity value if requested
    if (SET_PARITY) then
        CORRECT_PARITY := NEW_CORRECT_PARITY;
    end if;

    -- Compute parity of DATA
    TEMP := ’0’;
    for I in DATA’range loop
        TEMP := TEMP xor DATA(I);
    end loop;

    -- Compare computed parity with the correct value
    PARITY_OK <= (TEMP = CORRECT_PARITY);
end process;
Note that two flip-flops are in the synthesized schematic for Example 6-30. The first (input) flip-flop holds the value of `CORRECT_PARITY`. A flip-flop is needed here because `CORRECT_PARITY` is read (when it is compared to `TEMP`) before it is set (if `SET_PARITY` is `FALSE`). The second (output) flip-flop holds the value of `PARITY_OK` between clock cycles. The variable `TEMP` is not given a flip-flop because it is always set before it is read.
null Statement

The null statement explicitly states that no action is required. null is often used in case statements because all choices must be covered, even if some of the choices are ignored. The syntax is

null;

Example 6-31 shows a typical usage.

Example 6-31 null Statement

signal CONTROL: INTEGER range 0 to 7;
signal A, Z: BIT;
...
Z <= A;

case CONTROL is
  when 0 | 7 =>      -- If 0 or 7, then invert A
    Z <= not A;
  when others =>      -- If not 0 or 7, then do nothing
    null;
end case;