VeriBest FPGA Synthesis VHDL Reference Manual

VB 98.0 Reprint

DLA029300

VERIBEST®
INCORPORATED

Warranties and Liabilities

All warranties given by VeriBest, Inc. (hereinafter collectively called VeriBest), are set forth in the Software License Agreement, and nothing stated in, or implied by, this document or its contents shall be considered or deemed a modification or amendment of such warranties.

The information and the software discussed in this document are subject to change without notice and should not be construed as commitments by VeriBest. VeriBest assumes no responsibility for any errors that may appear in this document.

The software discussed in this document is furnished under a license and may be used or copied only in accordance with the terms of this license.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software -- Restricted Rights at 48 CFT 52.227-19, as applicable.

Trademarks

Table of Contents

Chapter 7

Chapter 8
Register and Three-State Inference ... 8-1

Chapter 9
FPGA Express Directives ... 9-1

Chapter 10

Chapter 11

FPGA Express
VHDL Reference Manual

September 1996

Comments?
E-mail your comments about Synopsys documentation to doc@synopsys.com

SYNOPSYS®

Chapter 1
Using FPGA Express with VHDL

FPGA Express translates and optimizes a VHDL description to an internal gate-level equivalent format. This format is then compiled for a given FPGA technology.

To work with VHDL, familiarize yourself with the following concepts:

- Hardware Description Languages
- About VHDL
- About FPGA Express
- Using FPGA Express
- A Model of the Design Process

The United States Department of Defense, as part of its Very-High-Speed Integrated Circuit (VHSIC) program, developed *VHSIC HDL* (VHDL) in 1982. VHDL describes the behavior, function, inputs, and outputs of a digital circuit design. VHDL is similar in style and syntax to modern programming languages, but includes many hardware-specific constructs.

FPGA Express reads and parses the supported VHDL syntax. Chapter 11 lists all VHDL constructs and includes the level of Synopsys support provided for each construct.

Hardware Description Languages

Hardware description languages (HDLs) are used to describe the architecture and behavior of discrete electronic systems.

HDLs were developed to deal with increasingly complex designs. An analogy is often made to the history of what can be called software description languages, from machine code (transistors and solder), to assembly language (netlists), to high-level languages (HDLs).

Top-down, HDL-based system design is most useful in large projects, where several designers or teams of designers are working concurrently. HDLs provide structured development. After major architectural decisions have been made, and major components and their connections have been identified, work can proceed independently on subprojects.

Typical Uses for HDLs

HDLs typically support a mixed-level description where structural or netlist constructs can be mixed with behavioral or algorithmic descriptions. With this mixed-level capability, you can describe system architectures at a high level of abstraction; then incrementally refine a design into a particular compo-

nent-level or gate-level implementation. Alternatively, you can read an HDL design description into FPGA Express, then direct the compiler to synthesize a gate-level implementation automatically.

Advantages of HDLs

A design methodology that uses HDLs has several fundamental advantages over a traditional gate-level design methodology. Among the advantages are the following:

- You can verify design functionality early in the design process, and immediately simulate a design written as an HDL description. Design simulation at this higher level, before implementation at the gate-level, allows you to test architectural and design decisions.
- FPGA Express provides logic synthesis and optimization, so you can automatically convert a VHDL description to a gate-level implementation in a given technology. This methodology eliminates the former gate-level design bottleneck and reduces circuit design time and errors introduced when hand-translating a VHDL specification to gates. With FPGA Express *logic optimization*, you can automatically transform a synthesized design to a smaller and faster circuit. You can apply information gained from the synthesized and optimized circuits back to the VHDL description, perhaps to fine-tune architectural decisions.
- HDL descriptions provide technology-independent documentation of a design and its functionality. An HDL description is more easily read and understood than a netlist or schematic description. Since the initial HDL design description is technology-independent, you can later reuse it to generate the design in a different technology, without having to translate from the original technology.
- VHDL, like most high-level software languages, provides strong *type checking*. A component that expects a four-bit-wide signal type cannot be connected to a three- or five-bit-wide signal; this mismatch causes an error when the HDL description is compiled. If a variable's range is defined as 1 to 15, an error results from assigning it a value of 0. Incorrect use of types has been shown to be a major source of errors in descriptions. Type checking catches this kind of error in the HDL description even before a design is generated.

About VHDL

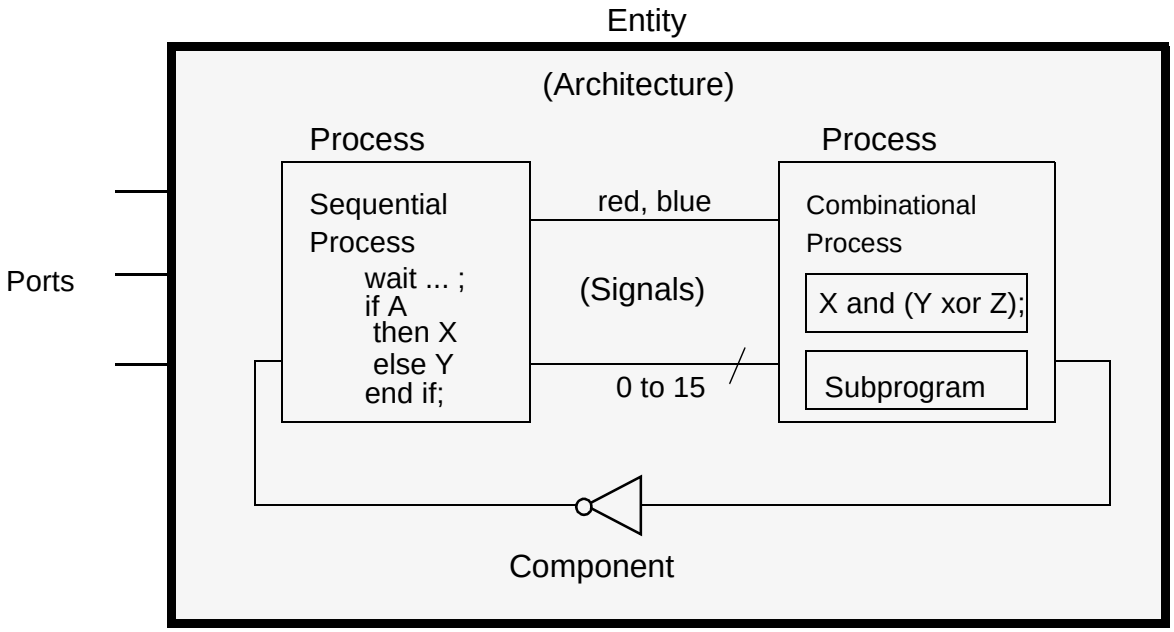
VHDL is one of just a few HDLs in widespread use today. VHDL is recognized as a standard HDL by the IEEE (IEEE Standard 1076, ratified in 1987) and by the United States Department of Defense (MIL-STD-454L).

VHDL divides *entities* (components, circuits, or systems) into an external or visible part (entity name and connections) and an internal or hidden part (entity algorithm and implementation). After you define the external interface to an entity, other entities can use that entity when they all are being developed. This concept of internal and external views is central to a VHDL view of system design. An entity is defined, with respect to other entities, by its connections and behavior. You can explore alternate implementations (*architectures*) of an entity without changing the rest of the design.

After you define an entity for one design, you can reuse it in other designs as needed. You can develop libraries of entities for use by many designs, or for a family of designs.

The VHDL model of hardware is shown in Figure 1-1.

Figure 1-1: VHDL Hardware Model



A VHDL *entity* (design) has one or more input, output, or inout *ports* that are connected (wired) to neighboring systems. An entity is itself composed of interconnected entities, *processes,* and *components*, all which operate concurrently. Each entity is defined by a particular *architecture*, which is composed of VHDL constructs such as arithmetic, signal assignment, or component instantiation statements.

In VHDL, independent *processes* model sequential (clocked) circuits, using flip-flops and latches, and combinational (unclocked) circuits, using only logic gates. Processes can define and call (*instantiate*) *subprograms* (subdesigns). Processes communicate with each other by *signals* (wires).

A signal has a source (driver*)*, one or more destinations (receivers), and a user-defined *type*, such as "color" or "number between 0 and 15."

VHDL provides a broad set of constructs. With VHDL you can describe discrete electronic systems of varying complexity (systems, boards, chips, modules) with varying levels of abstraction.

VHDL language constructs are divided into three categories by their level of abstraction: *behavioral*, *dataflow*, and *structural.* These categories are described as follows:

behavioral

The functional or algorithmic aspects of a design, expressed in a sequential VHDL process.

dataflow

The view of data as flowing through a design, from input to output. An operation is defined in terms of a collection of data transformations, expressed as concurrent statements.

structural

The view closest to hardware; a model where the components of a design are interconnected. This view is expressed by component instantiations.

FPGA Express Design Process

FPGA Express performs three functions:

- Translates VHDL to an internal format
- Optimizes the block level representation through various optimization methods
- Maps the design's logical structure for a specific FPGA technology library.

FPGA Express synthesizes VHDL descriptions according to the VHDL *synthesis policy* defined in Chapter 2, "Description Styles." The Synopsys VHDL synthesis policy has three parts: design methodology, design style, and language constructs. You use the VHDL synthesis policy to produce high quality VHDL-based designs.

Using FPGA Express to Compile a VHDL Design

When a VHDL design is read into FPGA Express, it is converted to an internal database format so FPGA Express can synthesize and optimize the design. When FPGA Express optimizes a design, it may restructure part or all the design. You control the degree of restructuring. Options include:

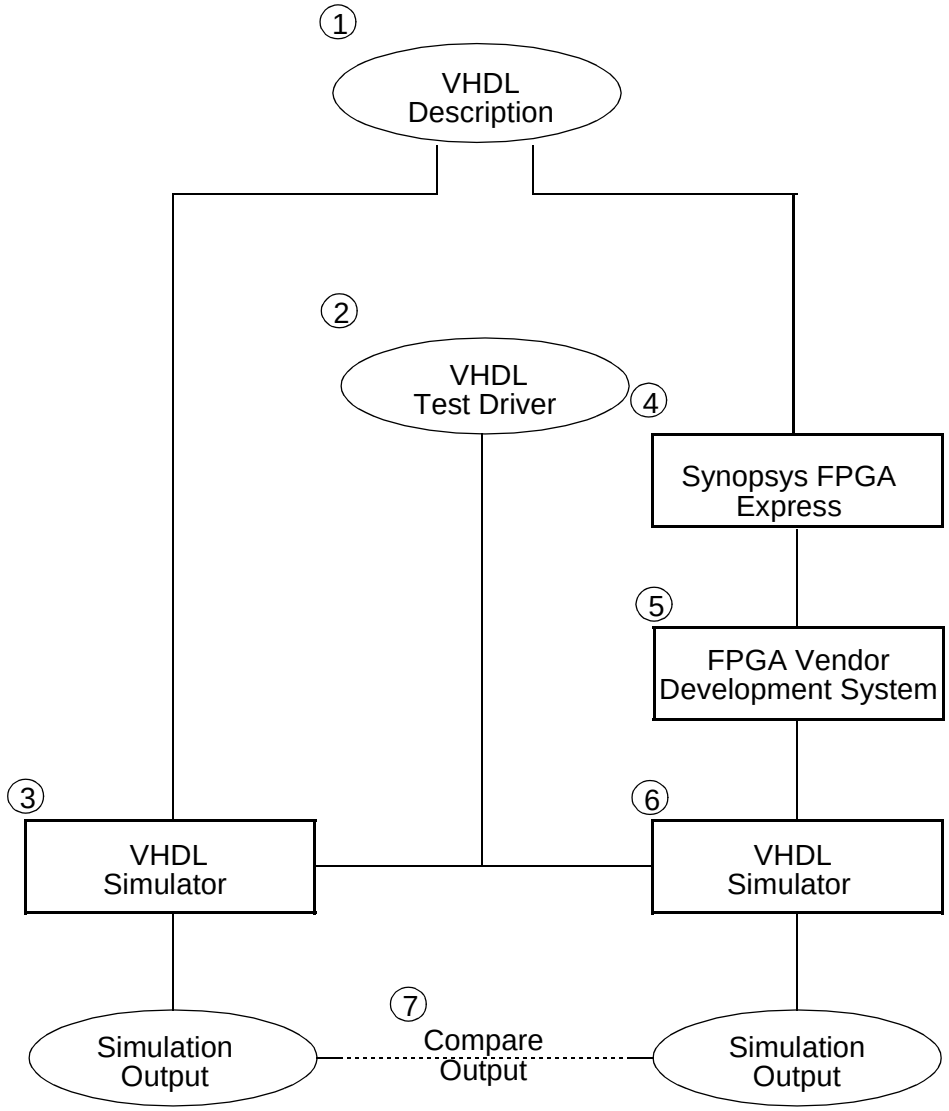
- Fully preserving a design's hierarchy
- Allowing full modules to be moved up or down in the hierarchy
- Allowing certain modules to be combined with others
- Compressing the entire design into one module (called *flattening* the design) if it is beneficial

The following section describes the design process that uses FPGA Express with a VHDL Simulator.

Design Methodology

Figure 1-2 shows a typical design process that uses FPGA Express and a VHDL Simulator. Each step of this design model is described in detail.

Figure 1-2: Design Flow



The steps in Figure 1-2 are explained below.

1. Write a design description in VHDL. This description can be a combination of structural and functional elements (as shown in Chapter 2, “Description Styles“). This description is used with both FPGA Express and the Synopsys VHDL simulator.
2. Provide VHDL-language test drivers for the simulator. For information on writing these drivers, see the appropriate simulator manual. The drivers supply test vectors for simulation and gather output data.
3. Simulate the design by using a VHDL simulator. Verify that the description is correct.
4. Use FPGA Express to synthesize and optimize the VHDL design description into a gate-level netlist. FPGA Express generates optimized netlists to satisfy timing constraints for a targeted FPGA architecture.

5. Use your FPGA development system to link the FPGA technology-specific version of the design to the VHDL simulator. The development system includes simulation models and interfaces required for the design flow.
6. Simulate the technology-specific version of the design with the VHDL simulator. You can use the original VHDL simulation drivers from Step 2 because module and port definitions are preserved through the translation and optimization processes.
7. Compare the output of the gate-level simulation (Step 6) against the output of the original VHDL description simulation (Step 3) to verify that the implementation is correct.

Chapter 2
Description Styles

The style of your initial VHDL description has a major effect on the characteristics of the resulting gate-level design synthesized by FPGA Express. The organization and style of a VHDL description determines the basic architecture of your design. Because FPGA Express automates most of the logic-level decisions required in your design, you can concentrate on architectural tradeoffs.

You can make some of the high-level architectural decisions that are needed by using FPGA Express. Certain VHDL constructs are well suited for synthesis. To make the decisions and use the constructs, you need to become familiar with the following concepts:

- Design Hierarchy
- Data Types
- Design Constraints
- Register Selection
- Asynchronous Designs
- Language Constructs

Design Hierarchy

FPGA Express maintains the hierarchical boundaries you define when using the structural view in VHDL. These boundaries have two major effects:

1. Each design entity specified in your VHDL description is synthesized separately and is maintained as a distinct design. The constraints for the design are maintained, and each design entity can be optimized separately in FPGA Express.
2. Component instantiations within VHDL descriptions are maintained during input. The instance name you give each user-defined entity is carried through to the gate-level implementation.

Chapter 3 discusses design entities, and Chapter 7 discusses component instantiations.

Note: FPGA Express does not automatically maintain or create a hierarchy of other nonstructural VHDL constructs such, as blocks, processes, loops, functions, and procedures. These elements of a VHDL description are translated in the context of their design. After reading in a VHDL design, you can group together the logic of a process, function, or procedure within the FPGA Express Implementation Window.

The choice of hierarchical boundaries has a significant effect on the quality of the synthesized design. Using FPGA Express, you can optimize a design while preserving these hierarchical boundaries. How-

ever, FPGA Express only partially optimizes logic across hierarchical modules. Full optimization is possible across those parts of the design hierarchy that are collapsed in FPGA Express.

Data Types

In VHDL, you must assign a data type to all ports, signals, and variables. The data type of an object defines the operations that can be applied to it. For example, the AND operator is defined for objects of type `BIT`, but not for objects of type `INTEGER`.

Data types are also important when your design is synthesized. The data type of an object determines its size (bit width) and its bit organization. The proper choice of data types greatly improves design quality and helps minimize errors.

See Chapter 4 for a discussion of VHDL data types.

Design Constraints

You can describe the performance constraints for a design module within the FPGA Express Implementation Window. Refer to the *FPGA Express User's Guide* for further information.

Register Selection

The placement of registers and the clocking scheme are important architectural decisions. There are two ways to define registers in your VHDL description. Each method has specific advantages:

- You can directly instantiate registers into a VHDL description, selecting from any element in your FPGA library. Clocking schemes can be arbitrarily complex. You can choose between a flip-flop and a latch-based architecture. The major disadvantages of this approach are
- The VHDL description is now specific to a given technology because you choose structural elements from that technology library. However, you can isolate this portion of your design as a separate entity, which you then connect to the remainder of the design.
- The description is more difficult to write.
- You can use the VHDL `if` and `wait` statements to direct FPGA Express to infer latches and flip-flops from the description. The advantages of this approach directly counter the disadvantages of the previous approach. When using register inference, the VHDL description is technology-independent and is much easier to write. This method allows FPGA Express to select the type of component inferred, on the basis of constraints. Therefore, if a specific component is necessary, instantiation should be used. Some types of registers and latches cannot be inferred.

See Chapter 8 for a discussion of register and latch inference.

Asynchronous Designs

You can use FPGA Express to construct asynchronous designs with multiple clocks and gated clocks. However, although these designs are logically (statically) correct, they might not simulate or operate correctly, because of race conditions.

Language Constructs

Another component of the VHDL synthesis policy is the set of VHDL constructs that describe your design, determine its architecture, and give consistently good results. The remainder of this manual discusses these constructs and their uses.

The concepts mentioned earlier in this chapter are described in the manual as follows:

Design Hierarchy

Chapter 3 describes the use and importance of hierarchy in VHDL designs. Chapter 7 explains how to instantiate (apply) existing components.

Data Types

Chapter 4 describes data types and their uses.

Register Selection

You can instantiate registers with the component instantiation statement discussed in Chapter 3 and Chapter 7. Chapter 6, and Chapter 8 describe register inference with the VHDL `if` and `wait` statements.

Chapter 3
Describing Designs

To describe a design in VHDL, you need to be familiar with the following concepts:

- VHDL Entities
- VHDL Constructs
- Defining Designs
- Structural Designs

VHDL Entities

Designs that are described with VHDL are composed of entities. An *entity* represents one level of the design hierarchy and can consist of a complete design, an existing hardware component, or a VHDL-defined object.

Each design has two parts: the entity specification and the architecture. The specification of an entity is its external interface. The architecture of an entity is its internal implementation. A design has only one entity specification (interface), but it can have multiple architectures (implementations). When an entity is compiled into a hardware design, a configuration specifies the architecture to use. An entity's specification and architecture can be contained in separate VHDL source files or in one VHDL source file.

Example 3-1 shows the entity specification of a simple logic gate (a 2-input NAND gate).

Example 3-1: VHDL Entity Specification

```
entity NAND2 is
  port(A, B: in BIT;    -- Two inputs, A and B
       Z: out BIT);     -- One output, Z = (A and B)'
end NAND2;
```

Note: In a VHDL description, a comment is prefixed by two hyphens (`--`). All characters from the hyphens to the end of the line are ignored by FPGA Express. The only exceptions to this rule are comments that begin with `-- pragma` or `-- synopsys`; these comments are FPGA Express directives.

After an `entity` statement declares an entity specification, that entity can be used by other entities in a design. The internal architecture of an entity determines its function.

Examples 3-2, 3-3, and 3-4 show three different architectures for the entity `NAND2`. The three examples define equivalent implementations of `NAND2`. After optimization and synthesis, each implementa-

tion produces the same circuit, probably a 2-input NAND gate in the target technology. The architecture description style you use for this entity depends on your own preferences.

Example 3-2 shows how the entity NAND2 can be implemented with two components from a technology library. The entity inputs A and B are connected to AND gate U0, producing an intermediate signal I. Signal I is then connected to inverter U1, producing the entity output Z.

Example 3-2: Structural Architecture for Entity NAND2

```
architecture STRUCTURAL of NAND2 is
  signal I:  BIT;

  component AND_2           -- From a technology library
      port(I1, I2: in BIT;
           O1: out BIT);
  end component;

  component INVERT          -- From a technology library
      port(I1: in BIT;
           O1: out BIT);
  end component;

begin
  U0: AND_2  port map (I1 => A, I2 => B, O1 => I);
  U1: INVERT port map (I1 => I, O1 => Z);
end STRUCTURAL;
```

Example 3-3 shows how you can define the entity NAND2 by its logical function.

Example 3-3: Dataflow Architecture for Entity NAND2

```
architecture DATAFLOW of NAND2 is
begin
  Z <= A nand B;
end DATAFLOW;
```

Example 3-4 shows another implementation of NAND2.

Example 3-4: RTL Architecture for Entity NAND2

```
architecture RTL of NAND2 is
begin
  process(A, B)
  begin
    if (A = '1') and (B = '1') then
      Z <= '0';
    else
      Z <= '1';
    end if;
  end process;
end RTL;
```

VHDL Constructs

The top-level VHDL constructs work together to describe a design. The description consists of

Entities

The interfaces to other designs.

Architectures

The implementations of design entities. Architectures can specify connection through instantiation to other entities.

Configurations

The bindings of entities to architectures.

Processes

Collections of sequentially executed statements. Processes are declared within architectures.

Subprograms

Algorithms that can be used by more than one architecture.

Packages

Collections of declarations used by one or more designs.

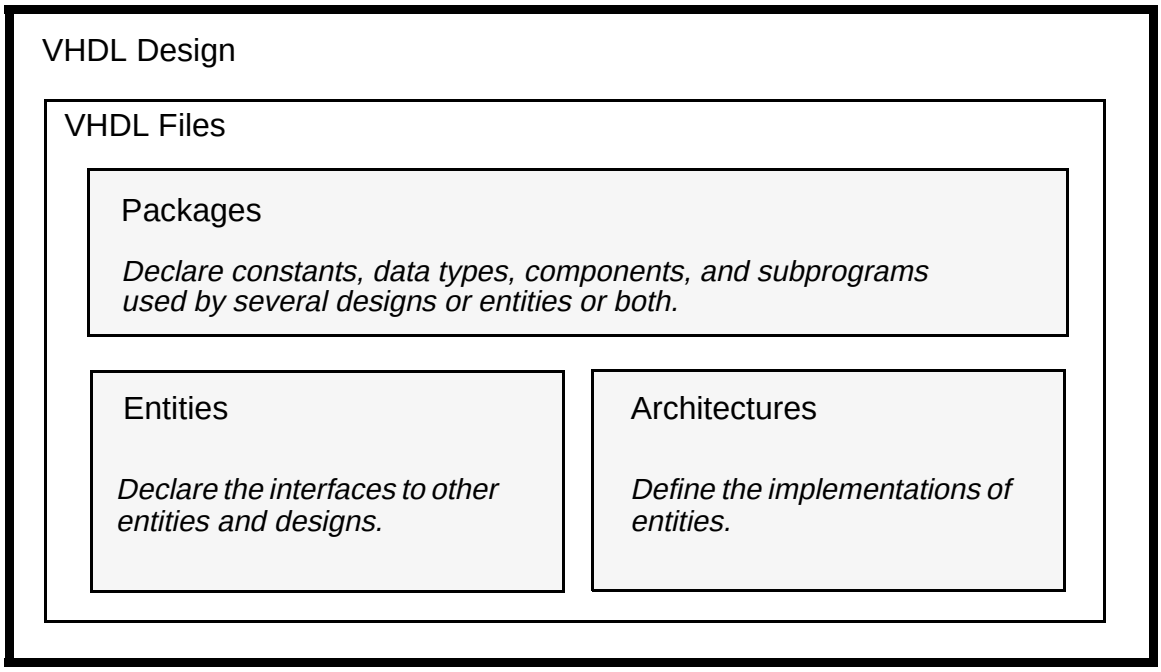
Entities

A VHDL design consists of one or more entities. Entities have defined inputs and outputs, and perform a defined function. Each design has two parts: an entity specification and an architecture. The entity specification defines the design's inputs and outputs, and the architecture determines its function.

You can describe a VHDL design in one or more files. Each file contains entities, architectures, or packages. Packages define global information that can be used by several entities. You can often reuse VHDL design files in later design projects.

Figure 3-1 shows a block diagram of a VHDL design's hierarchical organization into files.

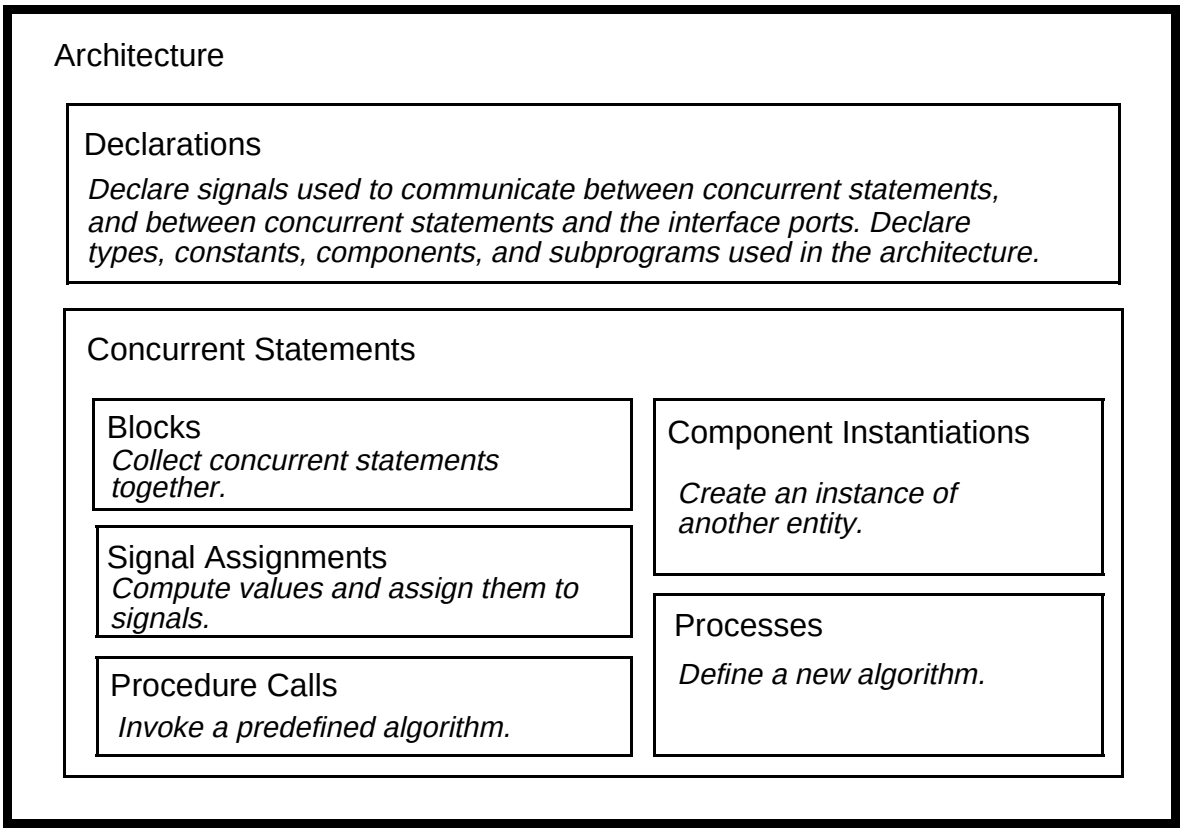
Figure 3-1: Design Organization



Architectures

An architecture determines the function of an entity. Figure 3-2 shows the organization of an architecture. Not all architectures contain every construct shown.

Figure 3-2: Architecture Organization



An architecture consists of a declaration section where you declare signals, types, constants, components, and subprograms, followed by a collection of concurrent statements.

Signals connect the separate pieces of an architecture (the concurrent statements) to each other, and to the outside world, through interface ports. You declare each signal with a type that determines the kind of data it carries. Types, constants, components, and subprograms declared in an architecture are local to that architecture. To use these declarations in more than one entity or architecture, place them in a package, as described under "Packages" later in this chapter.

Each concurrent statement in an architecture defines a unit of computation that reads signals, performs a computation that is based on the signal values, and assigns computed values to signals. Concurrent statements compute all values simultaneously. Although the order of concurrent statements has no effect on execution order, the statements often coordinate their processing by communicating with each other through signals.

The five kinds of concurrent statements are blocks, signal assignments, procedure calls, component instantiations, and processes. They are described as follows:

blocks

Group together a set of concurrent statements.

signal assignments

Assign computed values to signals or interface ports.

procedure calls

Call algorithms that compute and assign values to signals.

component instantiations

Create an instance of an entity, connecting its interface ports to signals or interface ports of the entity being defined. See "Structural Design" later in this chapter.

processes

Define sequential algorithms that read the values of signals, and compute new values to assign to other signals. Processes are discussed in the next section.

Concurrent statements are described in Chapter 7.

Configurations

A configuration specifies one combination of an entity and its associated architecture.

Note: FPGA Express supports only configurations that associate one top-level entity with an architecture.

Processes

Processes contain *sequential statements* that define algorithms. Unlike concurrent statements, sequential statements are executed in order. The order allows you to perform step-by-step computations. Processes read and write signals and interface port values to communicate with the rest of the architecture and with the enclosing system.

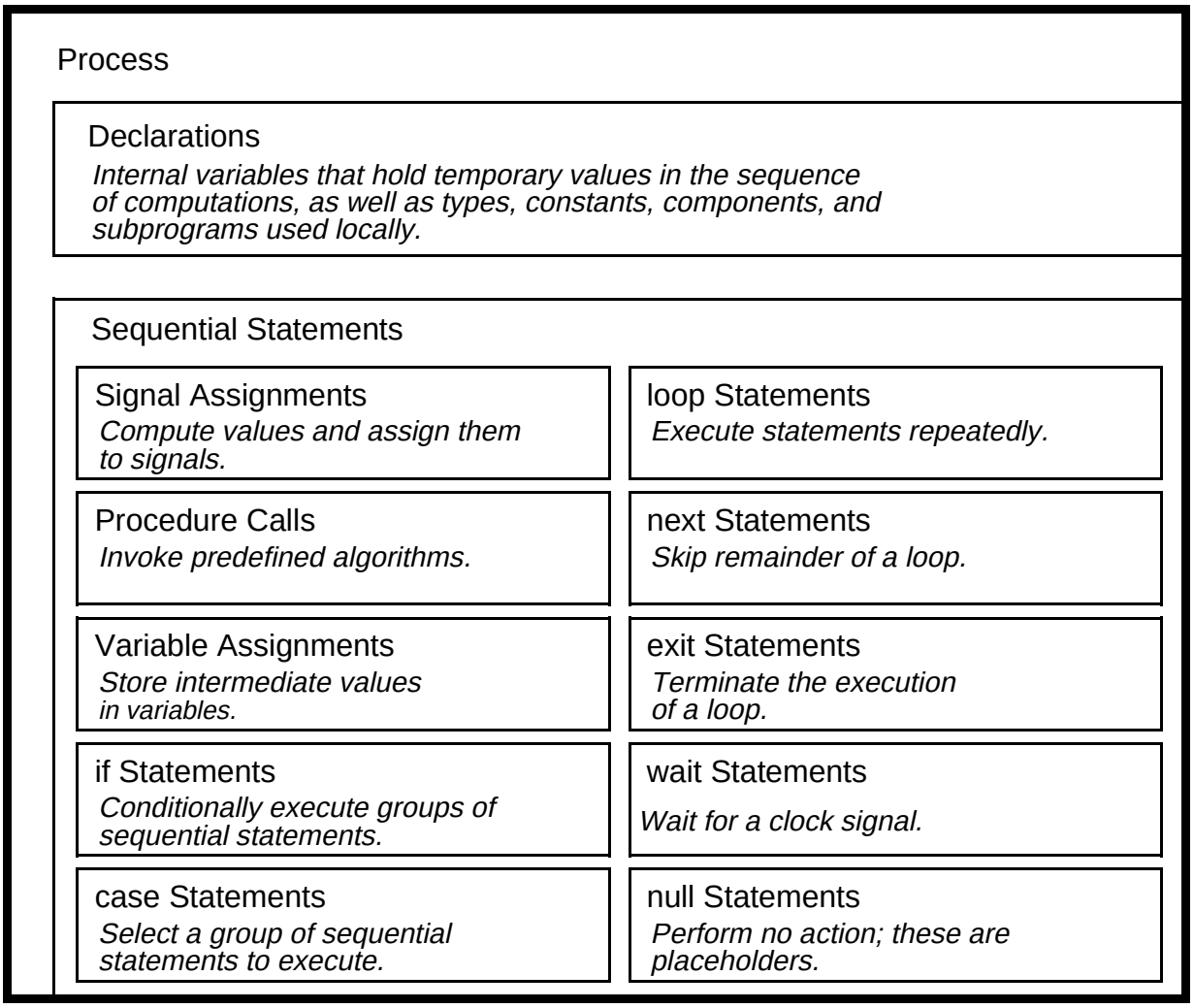
Figure 3-3 shows the organization of constructs in a process. Processes need not use all the constructs listed.

Processes are unique in that they behave like concurrent statements to the rest of the design, but they are internally sequential. In addition, only processes can define variables to hold intermediate values in a sequence of computations.

Because the statements in a process are sequentially executed, several constructs are provided to control the order of execution, such as `if` and `loop` statements.

Chapter 6 describes sequential statements.

Figure 3-3: Process Organization



Subprograms

Subprograms, like processes, use sequential statements to define algorithms that compute values. Unlike processes, however, they cannot directly read or write signals from the rest of the architecture. All communication is through the subprogram's interface; each subprogram call has its own set of interface signals.

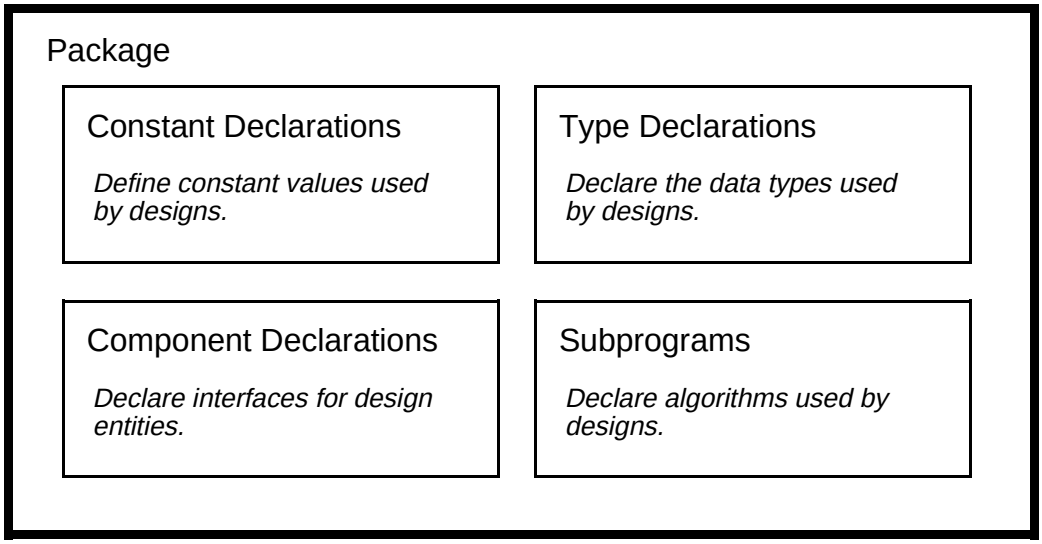
The two types of subprograms are functions and procedures. A function returns a single value directly. A procedure returns zero or more values through its interface. Subprograms are useful because you can use them to perform repeated calculations, often in different parts of an architecture.

Chapter 6 describes subprograms.

Packages

You can collect constants, data types, component declarations, and subprograms into a VHDL package that can then be used by more than one design or entity. Figure 3-4 shows the typical organization of a package.

Figure 3-4: Typical Package Organization



A package must contain at least one of the constructs listed in Figure 3-4.

- Constants in packages often declare system-wide parameters, such as data-path widths.
- VHDL data type declarations are often included in a package to define data types used throughout a design. All entities in a design must use common interface types; for example, common address bus types.
- Component declarations specify the interfaces to entities that can be instantiated in the design.
- Subprograms define algorithms that can be called anywhere in a design.

Packages are often sufficiently general so that you can use them in many different designs. For example, the `std_logic_1164` package defines data types `std_logic` and `std_logic_vector`.

Using a Package

The `use` statement allows an entity to use the declarations in a package. The supported syntax of the `use` statement is

```
use LIBRARY_NAME.PACKAGE_NAME.ALL;
```

LIBRARY_NAME is the name of a VHDL library, and *PACKAGE_NAME* is the name of the included package. A `use` statement is usually the first statement in a package or entity specification source file. Synopsys does not support different packages with the same name when they exist in different libraries. No two packages can have the same name.

Package Structure

Packages have two parts, the *declaration* and the *body*:

package declaration

Holds *public* information, including constant, type, and subprogram declarations.

package body

Holds *private* information, including local types and subprogram implementations (bodies).

Note: When a package declaration contains subprogram declarations, a corresponding package body must define the subprogram bodies.

Package Declarations

Package declarations collect information needed by one or more entities in a design. This information includes data type declarations, signal declarations, subprogram declarations, and component declarations.

Note: Signals declared in packages cannot be shared across entities. If two entities both use a signal from a given package, each entity has its own copy of that signal.

Although you can declare all this information explicitly in each design entity or architecture in a system, it is often easier to declare system information in a separate package. Each design entity in the system can then use the system's package.

The syntax of a package declaration is

```
package package_name is
  { package_declarative_item }
end [ package_name ] ;
```

where *package_name* is the name of this package.

A *package_declarative_item* is any of these:

- `use` clause (to include other packages)
- Type declaration
- Subtype declaration
- Constant declaration
- Signal declaration
- Subprogram declaration
- Component declaration

Example 3-5 shows some package declarations.

Example 3-5: Sample Package Declarations

```
package EXAMPLE is

  type BYTE is range 0 to 255;
  subtype NIBBLE is BYTE range 0 to 15;

  constant BYTE_FF: BYTE := 255;

  signal ADDEND: NIBBLE;

  component BYTE_ADDER
    port(A, B:      in BYTE;
         C:        out BYTE;
         OVERFLOW: out BOOLEAN);
  end component;

  function MY_FUNCTION (A: in BYTE) return BYTE;

end EXAMPLE;
```

To use the example declarations above, add a `use` statement at the beginning of your design description as follows:

```
use WORK.EXAMPLE.ALL;

entity . . .

architecture . . .
```

Further examples of packages and their declarations are given in the packages supplied by Synopsys. These packages are listed in Chapter 10.

Package Bodies

Package bodies contain the implementations of subprograms listed in the package declaration. However, this information is never seen by designs or entities that use the package. Package bodies can include the implementations (bodies) of subprograms declared in the package declaration and in internal support subprograms.

The syntax of a package body is

```
package body package_name is
  { package_body_declarative_item }
end [ package_name ] ;
```

where *package_name* is the name of the associated package.

A *package_body_declarative_item* is any of these:

- use clause
- Subprogram declaration
- Subprogram body
- Type declaration
- Subtype declaration
- Constant declaration

For an example of a package declaration and body, see the std_logic_arith package supplied with FPGA Express. This package is listed in Chapter 10.

Defining Designs

The high-level constructs discussed earlier in this chapter involve

- Entity specifications (interfaces)
- Entity architectures (implementations)
- Subprograms

Entity Specifications

An *entity specification* defines the characteristics of an entity that must be known before that entity can be connected to other entities and components.

For example, before you can connect a counter to other entities, you must specify the number and types of its inputs and outputs. The entity specification defines the ports (inputs and outputs) of an entity.

The syntax of an entity specification is

```
entity entity_name is
  [ generic( generic_declarations) ; ]
  [ port( port_declarations) ; ]
end [ entity_name ] ;
```

entity_name is the name of the entity, *generic_declarations* determine local constants used for sizing or timing the entity, and *port_declarations* determine the number and type of inputs and outputs. Other declarations are not supported in the entity specification.

Entity Generic Specifications

Generic specifications are entity parameters. Generics can specify the bit widths of components (such as adders) or provide internal timing values.

A generic can have a default value. A generic is assigned a nondefault value only when the entity is instantiated (see “Component Instantiation Statement” on page 3-27‘‘) or configured (see “‘‘Entity Configurations” on page 3-16). Inside an entity, a generic is a constant value.

The syntax of *generic_declarations* is

```
generic(
[ constant_name : type [ := value ]
 { ; constant_name : type [ := value ] }
);
```

constant_name is the name of a generic constant, *type* is a previously defined data type, and the optional *value* is the default value of *constant_name*.

Note: FPGA Express supports only `INTEGER` type generics.

Entity Port Specifications

The syntax of *port_declarations* is

```
port(
[ port_name :  mode port_type
 { ; port_name :  mode port_type}]
);
```

port_name is the name of a port; *mode* is either `in`, `out`, `inout`, or `buffer`; and *port_type* is a previously defined data type.

The four port modes are

in	Can only be read.
out	Can only be assigned a value.
inout	Can be read and assigned a value. The value read is that of the port’s incoming value, not the assigned value (if any).
buffer	Similar to out, but can be read. The value read is the assigned value. It can have only one driver. For more information on drivers, see "Driving Signals" in Chapter 7..

Example 3-6 shows an entity specification for a 2-input N-bit comparator, with a default bit width of 8.

Example 3-6: Interface for an N-Bit Counter

```
-- Define an entity (design) called COMP
-- that has 2 N-bit inputs and one output.

entity COMP is
  generic(N:  INTEGER := 8);      -- default is 8 bits

  port(X, Y:  in  BIT_VECTOR(0 to N-1);
       EQUAL: out BOOLEAN);
end COMP;
```

Entity Architectures

Each entity architecture defines one implementation of the entity's function. An architecture can range in abstraction from an algorithm (a set of sequential statements within a process) to a structural netlist (a set of component instantiations).

The syntax of an architecture is

```
architecture architecture_name of entity_name is
  { block_declarative_item }
begin
  { concurrent_statement }
end [ architecture_name ] ;
```

architecture_name is the name of the architecture, and *entity_name* is the name of the entity being implemented.

A *block_declarative_item* is any of these:

- `use` clause
- Subprogram declaration
- Subprogram body
- Type declaration
- Subtype declaration
- Constant declaration
- Signal declaration
- Component declaration

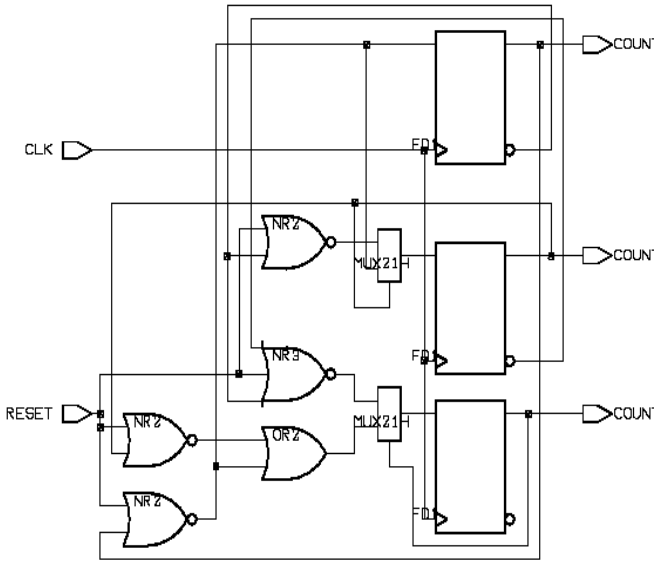
Concurrent statements are described in Chapter 7.

Example 3-7 shows a complete circuit description for a three-bit counter, entity specification (`COUNTER3`), and an architecture (`MY_ARCH`). This example also includes a schematic of the resulting synthesized circuit.

Example 3-7: An Implementation of a Three-Bit Counter

```
entity COUNTER3 is
port ( CLK :  in bit;
       RESET: in bit;
       COUNT: out integer range 0 to 7);
end COUNTER3;

architecture MY_ARCH of COUNTER3 is
signal COUNT_tmp : integer range 0 to 7;
begin
  process
  begin
     wait until (CLK'event and CLK = '1');
                    -- wait for the clock
     if RESET = '1' or COUNT_tmp = 7 then
                    -- Ck. for RESET or max. count
          COUNT_tmp <= 0;
     else COUNT_tmp <= COUNT_tmp + 1;
                    -- Keep counting
     end if;

  end process;
  COUNT <= COUNT_tmp;
end MY_ARCH;
```

Figure 3-5: Three-Bit Counter Schematic



Note: In an architecture, you must not declare constants or signals with the same name as any of the entity's ports. If you declare a constant or signal with a port's name, the new declaration hides that port name. If the new declaration is included in the architecture declaration (as shown in Example 3-8) and not in an inner block, FPGA Express reports an error.

Example 3-8: Incorrect Use of a Port Name when Declaring Signals or Constants

```
entity X is
  port(SIG, CONST: in  BIT;
       OUT1, OUT2: out BIT);
end X;

architecture EXAMPLE of X is
  signal   SIG  : BIT;
  constant CONST: BIT := '1';
begin

...

end EXAMPLE;
```

The error messages generated for Example 3-8 are:

```
  signal   SIG  : BIT;
           ^
Error:  (VHDL-1872) line 13
    Illegal redeclaration of SIG.

  constant CONST: BIT := '1';
           ^
Error:  (VHDL-1872) line 14
    Illegal redeclaration of CONST.
```

Entity Configurations

A configuration defines one combination of an entity and architecture for a design.

Note: FPGA Express supports only configurations that associate one top-level entity with an architecture.

The supported syntax for a configuration is

```
configuration configuration_name of entity_name is
  for architecture_name
  end for;
end [ configuration_name ] ;
```

configuration_name is the name of this configuration, *entity_name* is the name of a top-level entity, and *architecture_name* is the name of the architecture to use for *entity_name*.

Example 3-9 shows a configuration for the three-bit counter in Example 3-7. This configuration associates the counter's entity specification (COUNTER3) with an architecture (MY_ARCH).

Example 3-9: Configuration of Counter in Example 3-7

```
configuration MY_CONFIG of COUNTER3 is
  for MY_ARCH
  end for;
end MY_CONFIG;
```

Note: If you do not specify a configuration for an entity with multiple architectures, IEEE VHDL specifies that the last architecture read is used. This is determined from the `.mra` (most recently analyzed) file.

Subprograms

Subprograms describe algorithms that are meant to be used more than once in a design. Unlike component instantiation statements, when a subprogram is used by an entity or another subprogram, a new level of design hierarchy is not automatically created. However, you can manually define a subprogram as a new level of design hierarchy in the FPGA Express Implementation Window.

Two types of subprograms, procedures and functions, can contain zero or more parameters:

procedures

Procedures have no return value, but can return information to their callers by changing the values of their parameters.

functions

A function has a single value that it returns to the caller, but it cannot change the value of its parameters.

Like an entity, a subprogram has two parts—its declaration and its body:

declaration

Declares the interface to a subprogram: its name, its parameters, and its return value (if any).

body

Defines an algorithm that gives the subprogram's expected results.

When you declare a subprogram in a package, the subprogram declaration must be in the package declaration, and the subprogram body must be in the package body. A subprogram defined inside an architecture has a body, but does not have a corresponding subprogram declaration.

Subprogram Declarations

A subprogram declaration lists the names and types of its parameters and, for functions, the type of its return value.

The syntax of a procedure declaration is

```
procedure proc_name [ ( parameter_declarations ) ] ;
```

proc_name is the name of the procedure.

The syntax of a function declaration is

```
function func_name [ ( parameter_declarations ) ]
    return type_name ;
```

func_name is the name of the function, and *type_name* is the type of the function's returned value.

The syntax of *parameter_declarations* is the same as the syntax of *port_declarations*:

```
[ parameter_name    :  mode  parameter_type
 { ; parameter_name :  mode  parameter_type}]
```

parameter_name is the name of a parameter; *mode* is either `in`, `out`, `inout`, or `buffer`; and *parameter_type* is a previously defined data type.

Procedure parameters can use any mode. Function parameters must use only mode `in`. Signal parameters of type range cannot be passed to a subprogram.

Example 3-10 shows sample subprogram declarations for a function and a procedure.

Example 3-10: Two Subprogram Declarations

```
type BYTE   is array (7 downto 0) of BIT;
type NIBBLE is array (3 downto 0) of BIT;

function IS_EVEN(NUM: in INTEGER) return BOOLEAN;
  -- Returns TRUE if NUM is even.

procedure BYTE_TO_NIBBLES(B:             in BYTE;
                          UPPER, LOWER: out NIBBLE);
  -- Splits a BYTE into UPPER and LOWER halves.
```

Note: When you call a subprogram, actual parameters are substituted for the declared formal parameters. Actual parameters are either constant values or signal, variable, constant, or port names. An actual parameter must support the formal parameter's type and mode. For example, an input port cannot be used as an `out` actual parameter, and a constant can be used only as an `in` actual parameter.

Example 3-11 shows some calls to the subprogram declarations from Example 3-10.

Example 3-11: Two Subprogram Calls

```
signal INT : INTEGER;
variable EVEN : BOOLEAN;
. . .
INT <= 7;
EVEN := IS_EVEN(INT);
. . .

variable TOP, BOT: NIBBLE;
. . .
BYTE_TO_NIBBLES("00101101", TOP, BOT);
```

Subprogram Bodies

A subprogram body defines an implementation of a subprogram's algorithm.

The syntax of a procedure body is

```
procedure procedure_name [ (parameter_declarations) ] is
  { subprogram_declarative_item }
begin
  { sequential_statement }
end [ procedure_name ] ;
```

The syntax of a function body is

```
function function_name [  (parameter_declarations) ]
    return type_name is
  { subprogram_declarative_item }
begin
  { sequential_statement }
end [ function_name ] ;
```

A *subprogram_declarative_item* is any of these:

- `use` clause
- Type declaration
- Subtype declaration
- Constant declaration
- Variable declaration
- Attribute declaration
- Attribute specification
- Subprogram declaration
- Subprogram body

Example 3-12 shows subprogram bodies for the sample subprogram declarations in Example 3-10.

Example 3-12: Two Subprogram Bodies

```
function IS_EVEN(NUM: in INTEGER)
    return BOOLEAN is
begin
  return ((NUM rem 2) = 0);
end IS_EVEN;

procedure BYTE_TO_NIBBLES(B: in BYTE;
                          UPPER, LOWER: out NIBBLE) is
begin
  UPPER := NIBBLE(B(7 downto 4));
  LOWER := NIBBLE(B(3 downto 0));
end BYTE_TO_NIBBLES;
```

Subprogram Overloading

You can overload subprograms; more than one subprogram can have the same name. Each subprogram that uses a given name must have a different parameter profile.

A parameter profile specifies a subprogram's number and type of parameters. This information determines which subprogram is called when more than one subprogram has the same name. Overloaded functions are also distinguished by the type of their return values.

Example 3-13 shows two subprograms with the same name, but different parameter profiles.

Example 3-13: Subprogram Overloading

```
type SMALL is range 0 to 100;
type LARGE is range 0 to 10000;

function IS_ODD(NUM: SMALL) return BOOLEAN;
function IS_ODD(NUM: LARGE) return BOOLEAN;

signal A_NUMBER: SMALL;
signal B: BOOLEAN;
. . .
B <= IS_ODD(A_NUMBER); -- Will call the first
                       -- function above
```

Operator Overloading

Predefined operators such as +, and, and mod can also be overloaded. By using overloading, you can adapt predefined operators to work with your own data types.

For example, you can declare new logic types, rather than use the predefined types BIT and INTEGER. However, you cannot use predefined operators with these new types unless you declare overloaded operators for the new logic type.

Example 3-14 shows how some predefined operators are overloaded for a new logic type.

Example 3-14: Operator Overloading

```
type NEW_BIT is ('0', '1', 'X');
  -- New logic type

function "and"(I1, I2: in NEW_BIT) return NEW_BIT;
function "or" (I1, I2: in NEW_BIT) return NEW_BIT;
  -- Declare overloaded operators for new logic type
. . .
signal A, B, C: NEW_BIT;
. . .

C <= (A and B) or C;
```

VHDL requires overloaded operator declarations to enclose the operator name or symbol in double quotation marks, because they are infix operators (they are used between operands). If you declared the overloaded operators without quotation marks, a VHDL tool considers them functions rather than operators.

Type Declarations

Type declarations define the name and characteristics of a type. Types and type declarations are fully described in Chapter 4. A type is a named set of values, such as the set of integers, or the set (red, green, blue). An object of a given type, such as a signal, can have any value of that type.

Example 3-14 shows a type declaration for type NEW_BIT, and some functions and variables of that type.

Type declarations are allowed in architectures, packages, entities, blocks, processes, and subprograms.

Subtype Declarations

Use subtype declarations to define the name and characteristics of a constrained subset of another type or subtype. A subtype is fully compatible with its parent type, but only over the subtype's range. Subtype declarations are described in Chapter 4.

The following subtype declaration (NEW_LOGIC) is a subrange of the type declaration in Example 3-14.

```
subtype NEW_LOGIC is NEW_BIT range '0' to '1';
```

Subtype declarations are allowed wherever type declarations are allowed: in architectures, packages, entities, blocks, processes, and subprograms.

Constant Declarations

Constant declarations create named values of a given type. The value of a constant can be read but not changed.

Constant declarations are allowed in architectures, packages, entities, blocks, processes, and subprograms.

Example 3-15 shows some constant declarations.

Example 3-15: Constant Declarations

```
constant WIDTH: INTEGER := 8;
constant X    : NEW_BIT := 'X';
```

You can use constants in expressions, as described in Chapter 5, and as source values in assignment statements, as described in Chapter 6.

Signal Declarations

Signal declarations create new named signals (wires) of a given type. Signals can be given default (initial) values. However, these initial values are not used for synthesis.

Signals with multiple drivers (signals driven by wired logic) can have associated resolution functions, as described in the next section.

Example 3-16 shows two signal declarations.

Example 3-16: Signal Declarations

```
signal A, B: BIT;
signal INIT: INTEGER := -1;
```

Note: Ports are also signals, with the restriction that `out` ports cannot be read, and `in` ports cannot be assigned a value. You create signals either by port declarations or by signal declarations. You create ports only by port declarations.

You can declare signals in architectures, entities, and blocks, and use them in processes and subprograms. Processes and subprograms cannot declare signals for internal use.

You can use signals in expressions, as described in Chapter 5. Signals are assigned values by signal assignment statements, as described in Chapter 6.

Resolution Functions

Resolution functions are used with signals that can be connected (wired together). For example, if two drivers are directly connected to a signal, the resolution function determines whether the signal value is the AND, OR, or three-state function of the driving values.

Use resolution functions to assign the driving value when there are multiple drivers. For simulation, you can write an arbitrary function to resolve bus conflicts.

Note: A resolution function might change the value of a resolved signal, even if all drivers have the same value.

The resolution function for a signal is part of that signal's subtype declaration. You create a resolved signal in four steps:

```
-- Step 1
type SIGNAL_TYPE is ...
-- signal's base type is SIGNAL_TYPE


-- Step 2
subtype res_type is res_function SIGNAL_TYPE;
-- name of the subtype is res_type
-- name of function is res_function
-- signal type is res_type (a subtype of SIGNAL_TYPE)
...
-- Step 3
function res_function (DATA: ARRAY_TYPE)
  return SIGNAL_TYPE is
-- declaration of the resolution function
-- ARRAY_TYPE must be an unconstrained array of SIGNAL_TYPE
...
-- Step 4
signal resolved_signal_name:res_type;
-- resolved_signal_name is a resolved signal
...
```

1. The signal's base type is declared.
2. The resolved signal's subtype is declared as a subtype of the base type and includes the name of the resolution function.
3. The resolution function itself is declared (and later defined).
4. Resolved signals are declared as resolved subtypes.

FPGA Express does not support arbitrary resolution functions. Only wired AND, wired OR, and three-state functions are allowed. FPGA Express requires that you mark all resolution functions with a special directive indicating the kind of resolution performed.

Note: FPGA Express considers the directive only when creating hardware. The body of the resolution function is parsed but ignored. Using unsupported VHDL constructs (see Appendix C) generates errors.

Do not connect signals that use different resolution functions. FPGA Express supports only one resolution function per network.

The three resolution function directives are

```
-- synopsys resolution_method wired_and

-- synopsys resolution_method wired_or

-- synopsys resolution_method three_state
```

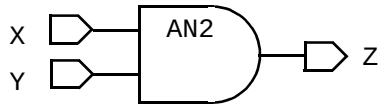
Note: Pre-synthesis and post-synthesis simulation results might not match if the body of the resolution function used by the simulator does not match the directive used by the synthesizer.

Example 3-17 shows how to create and use resolved signals, and how to use compiler directives for resolution functions. The signal's base type is the predefined type `BIT`.

Example 3-17: Resolved Signal and Its Resolution Function

```
package RES_PACK is
  function RES_FUNC(DATA: in BIT_VECTOR) return BIT;
  subtype RESOLVED_BIT is RES_FUNC BIT;
end;

package body RES_PACK is
  function RES_FUNC(DATA: in BIT_VECTOR) return BIT is
    -- pragma resolution_method wired_and
  begin
  -- The code in this function is ignored by FPGA Express
  -- but parsed for correct VHDL syntax

    for I in DATA'range loop
      if DATA(I) = '0' then
         return '0';
      end if;
    end loop;
    return '1';
  end;
end;

use work.RES_PACK.all;

entity WAND_VHDL is
  port(X, Y: in BIT; Z: out RESOLVED_BIT);
end WAND_VHDL;
```

```
architecture WAND_VHDL of WAND_VHDL is
begin
  Z <= X;
  Z <= Y;
end WAND_VHDL;
```



Variable Declarations

Variable declarations define a named value of a given type.

You can use variables in expressions, as described in Chapter 5. Variables are assigned values by variable assignment statements, as described in Chapter 6.

Example 3-18 shows some variable declarations.

Example 3-18: Variable Declarations

```
variable A, B: BIT;
variable INIT: NEW_BIT;
```

Note: Variables are declared and used only in processes and subprograms, because processes and subprograms cannot declare signals for internal use.

Structural Design

FPGA Express works with one or more designs. Each entity (and architecture) in a VHDL description is translated to a single design in FPGA Express. Designs can also originate from formats other than VHDL, such as equations, Programmable Logic Arrays (PLAs), state machines, other HDLs, or netlists.

A design can contain instances of lower-level designs, connected by nets (signals) to the lower-level design's ports. These lower-level designs can consist of other entities from a VHDL design, designs represented in some other Synopsys format, or cells from a technology library. By instantiating designs within designs, you create a hierarchy.

Hierarchy in VHDL is specified by using component declarations and component instantiation statements. To include a design, you must specify its interface with a component declaration. You can then create an instance of that design by using the component instantiation *statement*.

If your design consists only of VHDL entities, every component declaration statement corresponds to an entity in the design. If your design uses designs or technology library cells not described in VHDL,

create component declarations without corresponding entities. You can then use FPGA Express to associate the VHDL component with the non-VHDL design or cell.

Note: To simulate your VHDL design, you must provide entity and architecture descriptions for all component declarations.

Using Hardware Components

VHDL includes constructs to use existing hardware components. These structural constructs can be used to define a netlist of components.

The following sections describe how to use components and how FPGA Express configures these components.

Component Declaration

You must declare a component in an architecture or package before you can use (instantiate) it. A component declaration statement is similar to the entity specification statement described earlier, in that it defines the component's interface.

The syntax for a component declaration is

```
component identifier
  [ generic( generic_declarations ) ]
  [ port( port_declarations ) ]
end component ;
```

where *identifier* is the name of this type of component, and the syntax of *generic_declarations* and *port_declarations* is the same as defined previously for entity specifications.

Example 3-19 shows a simple component declaration statement.

Example 3-19: Component Declaration of a Two-Input AND Gate

```
component AND2
  port(I1, I2: in BIT;
       O1:     out BIT);
end component;
```

Example 3-20 shows a component declaration statement that uses a generic parameter.

Example 3-20: Component Declaration of an N-Bit Adder

```
component ADD
  generic(N: POSITIVE);

  port(X, Y:   in  BIT_VECTOR(N-1 downto 0);
       Z:      out BIT_VECTOR(N-1 downto 0);
       CARRY:  out BIT)
end component;
```

Although the component declaration statement is similar to the entity specification, it serves a different purpose. The component declaration is required to make the design entity AND2 or ADD usable, or visible, within an architecture. After a component is declared, it can be used in a design.

Sources of Components

A declared component can come from the same VHDL source file, from a different VHDL source file, from another format such as Electronic Data Interchange Format (EDIF) or state table, or from a technology library. If the component is not in one of the current VHDL source files, it must already be compiled by FPGA Express.

When a design that uses components is compiled by FPGA Express, previously compiled components are searched for by name in the following order:

1. In the current design.
2. In the input source file or files identified in the FPGA Express Implementation Window.
3. In the libraries of technology-specific FPGA components.

Consistency of Component Ports

FPGA Express checks for consistency among its VHDL entities. For other entities, the port names are taken from the original design description.

- For components in a technology library, the port names are the input and output pin names.
- For EDIF designs, the port names are the EDIF port names.

The bit widths of each port must also match. FPGA Express verifies matching for VHDL components, because the port types must be identical. For components from other sources, FPGA Express checks when linking the component to the VHDL description.

Component Instantiation Statement

The component instantiation statement instantiates and connects components to form a netlist (structural) description of a design. A component instantiation statement can create a new level of design hierarchy.

The syntax of the component instantiation statement is

```
instance_name : component_name
[ generic map (
   generic_name => expression
   { , generic_name => expression }
) ]
port map (
   [ port_name => ] expression
   { , [ port_name => ] expression }
);
```

instance_name is the name of this instance of component type *component_name*.

The optional `generic map` assigns nondefault values to generics. Each *generic_name* is the name of a generic, exactly as declared in the corresponding component declaration statement. Each *expression* evaluates to an appropriate value.

The `port map` assigns the component's ports to connections. Each *port_name* is the name of a port, exactly as declared in the corresponding component declaration statement. Each *expression* evaluates to a signal value.

FPGA Express uses the following two rules to decide which entity and architecture are to be associated with a component instantiation:

1. Each component declaration must have an entity with the same name: a VHDL entity, a design from another source (format), or a library component. This entity is used for each component instantiation associated with the component declaration.
2. If a VHDL entity has more than one architecture, the *last* architecture input is used for each component instantiation associated with that entity. The `.mra` file determines the last architecture analyzed.

Mapping Generic Values

When you instantiate a component with generics, you can map generics to values. A generic without a default value must be instantiated with a `generic map` value.

For example, a four-bit instantiation of the component `ADD` from Example 3-20 might use the following `generic map`.

```
U1:  ADD generic map (N => 4)
         port map (X, Y, Z, CARRY...);
```

The `port map` assigns component ports to actual signals; it is described in the next section.

Mapping Port Connections

You can specify port connections in component instantiation statements with either named or positional notation. With named notation, the *port_name* **=>** construct identifies the specific ports of the component. With positional notation, the expressions for the component ports are simply listed in the declared port order.

Example 3-21 shows named and positional notation for the U5 component instantiation statement in Example 3-22.

Example 3-21: Equivalent Named and Positional Association

```
U5: or2 port map (O => n6, I1 => n3, I2 => n1);
  -- Named association

U5: or2 port map (n3, n1, n6);
  -- Positional association
```

Note: When you use positional association, the instantiated port expressions (signals) must be in the same order as the declared ports.

Example 3-22 shows a structural (netlist) description for the COUNTER3 design entity from Example 3-7.

Example 3-22: Structural Description of a Three-Bit Counter

```
architecture STRUCTURE of COUNTER3 is
  component DFF
    port(CLK, DATA: in BIT;
         Q: out BIT);
  end component;
  component AND2
    port(I1, I2: in BIT;
         O: out BIT);
  end component;
  component OR2
    port(I1, I2: in BIT;
         O: out BIT);
  end component;
  component NAND2
    port(I1, I2: in BIT;
         O: out BIT);
  end component;
  component XNOR2
    port(I1, I2: in BIT;
         O: out BIT);
```

```
  end component;
  component INV
    port(I: in BIT;
         O: out BIT);
  end component;

  signal N1, N2, N3, N4, N5, N6, N7, N8, N9: BIT;

begin
  u1: DFF port map(CLK, N1, N2);
  u2: DFF port map(CLK, N5, N3);
  u3: DFF port map(CLK, N9, N4);
  u4: INV port map(N2, N1);
  u5: OR2 port map(N3, N1, N6);
  u6: NAND2 port map(N1, N3, N7);
  u7: NAND2 port map(N6, N7, N5);
  u8: XNOR2 port map(N8, N4, N9);
  u9: NAND2 port map(N2, N3, N8);
  COUNT(0) <= N2;
  COUNT(1) <= N3;
  COUNT(2) <= N4;
end STRUCTURE;
```

Technology-Independent Component Instantiation

When you use a structural design style, you might want to instantiate logical components. Synopsys provides generic technology library `GTECH` for this purpose. This generic technology library contains technology-independent logical components such as:

- AND, OR, and NOR gates (2, 3, 4, 5, and 8)
- one-bit adders and half adders
- 2-of-3 majority
- multiplexors
- flip-flops and latches
- multiple-level logic gates, such as AND-NOT, AND-OR, AND-OR-INVERT

You can use these simple components to create technology-independent designs. Example 3-23 shows how an N-bit ripple-carry adder can be created from N one-bit adders.

Example 3-23: Design That Uses Technology-Independent Components

```
library GTECH;
use gtech.gtech_components.all;
entity RIPPLE_CARRY is
  generic(N: NATURAL);

  port(A, B:       in BIT_VECTOR(N-1 downto 0);
       CARRY_IN:   in BIT;
       SUM:       out BIT_VECTOR(N-1 downto 0);
       CARRY_OUT: out BIT;);
end RIPPLE_CARRY;

architecture TECH_INDEP of RIPPLE_CARRY is

  signal CARRY: BIT_VECTOR(N downto 0);

begin
  CARRY(0) <= CARRY_IN;

  GEN: for I in 0 to N-1 generate
    U1: GTECH_ADD_ABC port map(
              A    => A(I),
              B    => B(I),
              C    => CARRY(I),
              S    => SUM(I),
              COUT => CARRY(I+1));

  end generate GEN;

  CARRY_OUT <= CARRY(N);
end TECH_INDEP;
```

Chapter 4
Data Types

VHDL is a strongly typed language. Every constant, signal, variable, function, and parameter is declared with a type, such as `BOOLEAN` or `INTEGER`, and can hold or return only a value of that type.

VHDL predefines abstract data types, such as `BOOLEAN`, which are part of most programming languages, and hardware-related types, such as `BIT`, found in most hardware languages. VHDL predefined types are declared in the `STANDARD` package, which is supplied with all VHDL implementations (see Example 4-12). Data types addresses information about

- Enumeration Types
- Integer Types
- Array Types
- Record Types
- Predefined VHDL Data Types
- Unsupported Data Types
- Synopsys Data Types
- Subtypes

The advantage of strong typing is that VHDL tools can catch many common design errors, such as assigning an eight-bit value to a four-bit-wide signal, or incrementing an array index out of its range.

The following code shows the definition of a new type, `BYTE`, as an array of eight bits, and a variable declaration, `ADDEND`, that uses this type.

```
type BYTE is array(7 downto 0) of BIT;


variable ADDEND: BYTE;
```

The predefined VHDL data types are built from the basic VHDL data types. Some VHDL types are not supported for synthesis, such as `REAL` and `FILE`.

The examples in this chapter show type definitions and associated object declarations. Although each constant, signal, variable, function, and parameter is declared with a type, only variable and signal declarations are shown here in the examples. Constant, function, and parameter declarations are shown in Chapter 3.

VHDL also provides *subtypes*, which are defined as subsets of other types. Anywhere a type definition can appear, a subtype definition can also appear. The difference between a type and a subtype is that

a subtype is a subset of a previously defined parent *(*or base*)* type or subtype. Overlapping subtypes of a given base type can be compared against and assigned to each other. All integer types, for example, are technically subtypes of the built-in integer base type (see "Integer Types," later in this chapter). Subtypes are described in the last section of this chapter.

Enumeration Types

An *enumeration type* is defined by listing (enumerating) all possible values of that type.

The syntax of an enumeration type definition is

```
type type_name is ( enumeration_literal
                    {, enumeration_literal} );
```

type_name is an identifier, and each *enumeration_literal* is either an identifier (enum_6) or a character literal ('A').

An identifier is a sequence of letters, underscores, and numbers. An identifier must start with a letter and cannot be a VHDL reserved word, such as TYPE. All VHDL reserved words are listed in Chapter 11.

A character literal is any value of type CHARACTER, in single quotes.

Example 4-1 shows two enumeration type definitions and corresponding variable and signal declarations.

Example 4-1: Enumeration Type Definitions

```
type COLOR is (BLUE, GREEN, YELLOW, RED);

type MY_LOGIC is ('0', '1', 'U', 'Z');

variable HUE: COLOR;

signal SIG: MY_LOGIC;
. . .
HUE := BLUE;
SIG <= 'Z';
```

Enumeration Overloading

You can overload an enumeration literal by including it in the definition of two or more enumeration types. When you use such an overloaded enumeration literal, FPGA Express can usually determine the literal's type. However, under certain circumstances determination may be impossible. In these cases, you must qualify the literal by explicitly stating its type (see "Qualified Expressions" in Chapter 5). Example 4-2 shows how you can qualify an overloaded enumeration literal.

Example 4-2: Enumeration Literal Overloading

```
type COLOR is (RED, GREEN, YELLOW, BLUE, VIOLET);
type PRIMARY_COLOR is (RED, YELLOW, BLUE);
...
A <= COLOR'(RED);
```

Enumeration Encoding

Enumeration types are ordered by enumeration *value*. By default, the first enumeration literal is assigned the value `0`, the next enumeration literal is assigned the value `1`, and so forth.

FPGA Express automatically encodes enumeration values into bit vectors that are based on each value's position. The length of the encoding bit vector is the minimum number of bits required to encode the number of enumerated values. For example, an enumeration type with five values has a three-bit encoding vector.

Example 4-3 shows the default encoding of an enumeration type with five values.

Example 4-3: Automatic Enumeration Encoding

```
type COLOR is (RED, GREEN, YELLOW, BLUE, VIOLET);
```

The enumeration values are encoded as follows:

```
RED    ⇒ "000"
GREEN  ⇒ "001"
YELLOW ⇒ "010"
BLUE   ⇒ "011"
VIOLET ⇒ "100"
```

The result is `RED` < `GREEN` < `YELLOW` < `BLUE` < `VIOLET.`

You can override the automatic enumeration encodings and specify your own enumeration encodings with the `ENUM_ENCODING` attribute. This interpretation is specific to FPGA Express.

A VHDL attribute is defined by its name and type, and is then declared with a value for the attributed type, as shown in Example 4-4 below.

Note: Several VHDL synthesis-related attributes are declared in the `ATTRIBUTES` package supplied with FPGA Express. This package is listed in Chapter 10. The section "Synthesis Attributes and Constraints" on page 1 describes how to use these VHDL attributes.

The ENUM_ENCODING attribute must be a STRING containing a series of vectors, one for each enumeration literal in the associated type. The encoding vector is specified by '0's, '1's, 'D's, 'U's, and 'Z's separated by blank spaces. The meaning of these encoding vectors is described in the next section. The first vector in the attribute string specifies the encoding for the first enumeration literal, the second vector specifies the encoding for the second enumeration literal, and so on. The ENUM_ENCODING attribute must immediately follow the type declaration.

Example 4-4 illustrates how the default encodings from Example 4-3 can be changed with the ENUM_ENCODING attribute.

Example 4-4: Using the ENUM_ENCODING Attribute

```
attribute ENUM_ENCODING: STRING;
  -- Attribute definition

type COLOR is (RED, GREEN, YELLOW, BLUE, VIOLET);
attribute ENUM_ENCODING of
  COLOR: type is "010 000 011 100 001";
  -- Attribute declaration
```

The enumeration values are encoded as follows:

```
RED    = "010"
GREEN  = "000"
YELLOW = "011"
BLUE   = "100"
VIOLET = "001"
```

The result is GREEN<VIOLET<RED<YELLOW<BLUE

Note: The interpretation of the ENUM_ENCODING attribute is specific to FPGA Express. Other VHDL tools, such as simulators, use the standard encoding (ordering).

Enumeration Encoding Values

The possible encoding values for the ENUM_ENCODING attribute are:

'0' Bit value 0

'1' Bit value 1

'D' Don't-care (can be either 0 or 1).

'U' Unknown. If U appears in the encoding vector for an enumeration, you cannot use that enumeration literal except as an operand to the = and /= operators. You can read an enumeration literal encoded with a U from a variable or signal, but you cannot assign it.

For synthesis, the = operator returns FALSE and the /= operator returns TRUE when either of the operands is an enumeration literal whose encoding contains U.

'Z' High impedance. See "Three-State Inference" in Chapter 8 for more information.

Integer Types

The maximum range of a VHDL integer type is –(¡¬¿–¿) to ¡¬¿–¿ (`-2_147_483_647 .. 2_147_483_647`). Integer types are defined as subranges of this anonymous built-in type. Multidigit numbers in VHDL can include underscores (_) to make them easier to read.

FPGA Express encodes an integer value as a bit vector whose length is the minimum necessary to hold the defined range and encodes integer ranges that include negative numbers as 2's-complement bit vectors.

The syntax of an integer type definition is

`type` *`type_name`* **`is range`** *`integer_range`* **`;`**

`type_name` is the name of the new integer type, and *`integer_range`* is a subrange of the anonymous integer type.

Example 4-5 shows some integer type definitions.

Example 4-5: Integer Type Definitions

```
type PERCENT is range -100 to 100;
  -- Represented as an 8-bit vector
  --   (1 sign bit, 7 value bits)

type INTEGER is range -2147483647 to 2147483647;
  -- Represented as a 32-bit vector
  --   This is the definition of the INTEGER type
```

Note: You cannot directly access the bits of an `INTEGER` or explicitly state the bit width of the type. For these reasons, Synopsys provides overloaded functions for arithmetic. These functions are defined in the `std_logic` package, listed in Chapter 10.

Array Types

An array is an object that is a collection of elements of the same type. VHDL supports N-dimensional arrays, but FPGA Express supports only one-dimensional arrays. Array elements can be of any type. An array has an index whose value selects each element. The index range determines how many elements are in the array and their ordering (low `to` high, or high `downto` low). An index can be of any integer type.

You can declare multidimensional arrays by building one-dimensional arrays where the element type is another one-dimensional array, as shown in Example 4-6.

Example 4-6: Declaration of Array of Arrays

```
type BYTE   is array (7 downto 0) of BIT;
type VECTOR is array (3 downto 0) of BYTE;
```

VHDL provides both constrained arrays and unconstrained arrays. The difference between these two arrays comes from the index range in the array type definition.

Constrained Array

A constrained array's index range is explicitly defined; for example, an integer range `(1 to 4)`. When you declare a variable or signal of this type, it has the same index range.

The syntax of a constrained array type definition is

```
type array_type_name is
    array ( integer_range ) of type_name ;
```

array_type_name is the name of the new constrained array type, *integer_range* is a subrange of another integer type, and *type_name* is the type of each array element.

Example 4-7 shows a constrained array definition.

Example 4-7: Constrained Array Type Definition

```
type BYTE is array (7 downto 0) of BIT;
  -- A constrained array whose index range is
  -- (7, 6, 5, 4, 3, 2, 1, 0)
```

Unconstrained Array

You define an unconstrained array's index range as a *type*, for example, `INTEGER`. This definition implies that the index range can consist of any contiguous subset of that type's values. When you declare an array variable or signal of this type, you also define its actual index range. Different declarations can have different index ranges.

The syntax of an unconstrained array type definition is

```
type array_type_name is
    array (range_type_name range <>)
        of element_type_name ;
```

array_type_name is the name of the new unconstrained array type,
range_type_name is the name of an integer type or subtype, and
element_type_name is the type of each array element.

Example 4-8 shows an unconstrained array type definition and a declaration that uses it.

Example 4-8: Unconstrained Array Type Definition

```
type BIT_VECTOR is array(INTEGER range <>) of BIT;
  -- An unconstrained array definition
. . .
variable MY_VECTOR : BIT_VECTOR(5 downto -5);
```

The advantage of using unconstrained arrays is that a VHDL tool remembers the index range of each declaration. You can use *array attributes* to determine the range (bounds) of a signal or variable of an unconstrained array type. With this information, you can write routines that use variables or signals of an unconstrained array type, independently of any one array variable's or signal's bounds. The next section describes array attributes and how they are used.

Array Attributes

FPGA Express supports the following predefined VHDL attributes for use with arrays:

- left
- right
- high
- low
- length
- range
- `reverse_range`

These attributes return a value corresponding to part of an array's range. Table 4-1 shows the values of the array attributes for the variable `MY_VECTOR` in Example 4-8.

Table 4-1: Array Index Attributes

MY_VECTOR'left	5
MY_VECTOR'right	-5
MY_VECTOR'high	5
MY_VECTOR'low	5
MY_VECTOR'length	11
MY_VECTOR'range	(5 down to -5)
MY_VECTOR' reverse_range	(-5 to 5)

Example 4-9 shows the use of array attributes in a function that ORs together all elements of a given `BIT_VECTOR` (declared in Example 4-8) and returns that value.

Example 4-9: Use of Array Attributes

```
function OR_ALL (X: in BIT_VECTOR) return BIT is
  variable OR_BIT: BIT;
  begin
    OR_BIT := '0';
    for I in X'range loop
      OR_BIT := OR_BIT or X(I);
    end loop;

    return OR_BIT;
  end;
```

Note that this function works for a `BIT_VECTOR` of any size.

Record Types

A record is a set of named fields of various types, unlike an array, which is composed of identical anonymous entries. A record's field can be of any previously defined type, including another record type.

Note: Constants in VHDL of type `record` are not supported for synthesis (the initialization of records is not supported).

Example 4-11 shows a record type declaration (`BYTE_AND_IX`), three signals of that type, and some assignments.

Example 4-11: Record Type Declaration and Use

```
constant LEN:  INTEGER := 8;

subtype BYTE_VEC is BIT_VECTOR(LEN-1 downto 0);

type BYTE_AND_IX is
  record
    BYTE: BYTE_VEC;
    IX:   INTEGER range 0 to LEN;
  end record;

signal X, Y, Z: BYTE_AND_IX;

signal DATA: BYTE_VEC;
signal NUM:  INTEGER;
. . .
```

```
X.BYTE <= "11110000";
X.IX   <= 2;

DATA <= Y.BYTE;
NUM  <= Y.IX;

Z <= X;
```

As shown in Example 4-11, you can read values from or assign values to records in two ways:

- By individual field name

```
X.BYTE <= DATA;
X.IX   <= LEN;
```

- From another record object of the same type

```
Z <= X;
```

Note: A record type object's individual fields are accessed by the object name, a period, and a field name: `X.BYTE` or `X.IX`. To access an element of the `BYTE` field's array, use the array notation `X.BYTE(2)`.

Predefined VHDL Data Types

IEEE VHDL describes two site-specific packages, each containing a standard set of types and operations: the `STANDARD` package and the `TEXTIO` package.

The `STANDARD` package of data types is included in all VHDL source files by an implicit `use` clause. The `TEXTIO` package defines types and operations for communication with a standard programming environment (terminal and file I/O). This package is not needed for synthesis, and therefore FPGA Express does not support it.

The FPGA Express implementation of the `STANDARD` package is listed in Example 4-12. This `STANDARD` package is a subset of the IEEE VHDL `STANDARD` package. Differences are described in "Unsupported Data Types" later in this chapter.

Example 4-12: FPGA Express STANDARD Package

```
package STANDARD is

  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, DC1, 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 INTEGER is range -2147483647 to 2147483647;

  subtype NATURAL is INTEGER range 0 to 2147483647;

  subtype POSITIVE is INTEGER range 1 to 2147483647;

  type STRING is array (POSITIVE range <>)
       of CHARACTER;

  type BIT_VECTOR is array (NATURAL range <>)
       of BIT;

end STANDARD;
```

Data Type *BOOLEAN*

The BOOLEAN data type is actually an enumerated type with two values, FALSE and TRUE, where FALSE < TRUE. Logical functions such as equality (=) and comparison (<) functions return a BOOLEAN value.

Convert a BIT value to a BOOLEAN value as follows:

```
BOOLEAN_VAR := (BIT_VAR = '1');
```

Data Type *BIT*

The BIT data type represents a binary value as one of two characters, '0' or '1'. Logical operations such as and can take and return BIT values.

Convert a BOOLEAN value to a BIT value as follows:

```
if (BOOLEAN_VAR) then
  BIT_VAR := '1';
else
  BIT_VAR := '0';
end if;
```

Data Type *CHARACTER*

The CHARACTER data type enumerates the ASCII character set. Nonprinting characters are represented by a three-letter name, such as NUL for the null character. Printable characters are represented by themselves, in single quotation marks, as follows:

```
variable CHARACTER_VAR: CHARACTER;
. . .
CHARACTER_VAR := 'A';
```

Data Type *INTEGER*

The INTEGER data type represents positive and negative whole numbers and zero.

Data Type *NATURAL*

The NATURAL data type is a subtype of INTEGER that is used to represent natural (nonnegative) numbers.

Data Type *POSITIVE*

The POSITIVE data type is a subtype of INTEGER that is used to represent positive (nonzero and nonnegative) numbers.

Data Type *STRING*

The STRING data type is an unconstrained array of CHARACTER data types. A STRING value is enclosed in double quotation marks, as follows:

```
variable STRING_VAR: STRING(1 to 7);
. . .
STRING_VAR := "Rosebud";
```

Data Type *BIT_VECTOR*

The BIT_VECTOR data type represents an array of BIT values.

Unsupported Data Types

Some data types are either not useful for synthesis or are not supported. Unsupported types are parsed but ignored by FPGA Express. These types are listed and described below.

Chapter 11 describes the level of FPGA Express support for each VHDL construct.

Physical Types

FPGA Express does not support physical types, such as units of measure (for example, nS). Because physical types are relevant to the simulation process, FPGA Express allows but ignores physical type declarations.

Floating Point Types

FPGA Express does not support floating point types, such as REAL. Floating point *literals*, such as 1.34, are allowed in the definitions of FPGA Express-recognized attributes.

Access Types

FPGA Express does not support access (pointer) types because no equivalent hardware construct exists.

File Types

FPGA Express does not support file (disk file) types. A hardware file is a RAM or ROM.

SYNOPSYS Data Types

The std_logic_arith package provides arithmetic operations and numeric comparisons on array data types. The package also defines two major data types: UNSIGNED and SIGNED. These data types, unlike the predefined INTEGER type, provide access to the individual bits (wires) of a numeric value. For more information, see Chapter 10.

Subtypes

A *subtype* is defined as a subset of a previously defined type or subtype. A subtype definition can appear wherever a type definition is allowed.

Subtypes are a powerful way to use VHDL type checking to ensure valid assignments and meaningful data handling. Subtypes inherit all operators and subprograms defined for their parent (base) types.

Subtypes are also used for resolved signals to associate a resolution function with the signal type. (See "Signal Declarations" in Chapter 3 for more information.)

For example, in Example 4-12 NATURAL and POSITIVE are subtypes of INTEGER and they can be used with any INTEGER function. These subtypes can be added, multiplied, compared, and assigned to each other, as long as the values are within the appropriate subtype's range. All INTEGER types and subtypes are actually subtypes of an anonymous predefined numeric type.

Example 4-13 shows some valid and invalid assignments between NATURAL and POSITIVE values.

Example 4-13: Valid and Invalid Assignments between INTEGER Subtypes

```
variable NAT:  NATURAL;
variable POS:  POSITIVE;
. . .
POS := 5;
NAT := POS + 2;
. . .
NAT := 0;
POS := NAT;      -- Invalid; out of range
For example, the type BIT_VECTOR is defined as
type BIT_VECTOR is array(NATURAL range <>) of BIT;
```

If your design uses only 16-bit vectors, you can define a subtype MY_VECTOR as

```
subtype MY_VECTOR is BIT_VECTOR(0 to 15);
```

Example 4-14 shows that all functions and attributes that operate on BIT_VECTOR also operate on MY_VECTOR.

Example 4-14: Attributes and Functions Operating on a Subtype

```
type BIT_VECTOR is array(NATURAL range <>) of BIT;
subtype MY_VECTOR is BIT_VECTOR(0 to 15);
. . .
signal   VEC1, VEC2:  MY_VECTOR;
signal   S_BIT:  BIT;
variable UPPER_BOUND: INTEGER;
. . .
if (VEC1 = VEC2)
. . .
VEC1(4) <= S_BIT;
VEC2 <= "0000111100001111";
. . .
RIGHT_INDEX := VEC1'high;
```

Chapter 5
Expressions

Expressions perform arithmetic or logical computations by applying an operator to one or more operands. Operators specify the computation to be performed. Operands are the data for the computation.

Expressions are discussed as

- Operators
- Operands

In the following VHDL fragment, `A` and `B` are operands, `+` is an operator, and `A + B` is an expression.

```
C := A + B;  -- Computes the sum of two values
```

You can use expressions in many places in a design description. Expressions can be:

- Assign to variables or signals or used as the initial values of constants.
- Used as operands to other operators.
- Used for the return value of functions.
- Used for the `IN` parameters in a subprogram call.
- Assigned to the `OUT` parameters in a procedure body.
- Used to control the actions of statements like `if`, `loop`, and `case`.

To understand expressions for VHDL, consider the individual components of operators and operands.

Operators

- Logical operators
- Relational operators
- Adding operators
- Unary (sign) operators
- Multiplying operators
- Miscellaneous arithmetic operators

Operands

- Computable operands

- Literals
- Identifiers
- Indexed names
- Slice names
- Aggregates
- Attributes
- Function calls
- Qualified expressions
- Type conversions

Operators

A VHDL operator is characterized by

- Name
- Computation (function)
- Number of operands
- Type of operands (such as `Boolean` or `Character`)
- Type of result value

You can define new operators, like functions, for any type of operand and result value. The predefined VHDL operators are listed in Table 5-1.

Table 5-1: Table 5-1Predefined VHDL Operators

Type	**Operators**						**Precedence**
Logical	`and`	`or`	`nand`	`nor`	`xor`		Lowest
Relational	`=`	`/=`	`<`	`<=`	`>`	`>=`	
Adding	`+`	`-`	`&`				
Unary (sign)	`+`	`-`					
Multiplying	`*`	`/`	`mod`	`rem`			
Miscellaneous	`**`	`abs`	`not`				Highest

Each row in the table lists operators with the same precedence. Each row's operators have greater precedence than those in the row above. An operator's precedence determines whether it is applied before or after adjoining operators.

Example 5-1 shows several expressions and their interpretations.

Example 5-1: Operator Precedence

```
A + B * C                =  A + (B * C)
not BOOL and (NUM = 4)   =  (not BOOL) and (NUM = 4)
```

VHDL allows existing operators to be overloaded (applied to new types of operands). For example, the `and` operator can be overloaded to work with a new logic type. For more information, see ‘‘Operator Overloading" in Chapter 3.

Logical Operators

Operands of a logical operator must be of the same type. The logical operators `and, or, nand, nor, xor,` and `not` accept operands of type `BIT,` type `BOOLEAN`, and one-dimensional arrays of `BIT` or `BOOLEAN`. Array operands must be the same size. A logical operator applied to two array operands is applied to pairs of the two arrays’ elements.

Example 5-2 shows some logical signal declarations and logical operations on them.

Example 5-2: Logical Operators

```
signal A, B, C:       BIT_VECTOR(3 downto 0);
signal D, E, F, G:    BIT_VECTOR(1 downto 0);
signal H, I, J, K:    BIT;
signal L, M, N, O, P: BOOLEAN;


A <= B and C;
D <= E or F or G;
H <= (I nand J) nand K;
L <= (M xor N) and (O xor P);
```

Normally, to use more than two operands in an expression, you must use parentheses to group the operands. Alternately you can combine a sequence of `and`, `or`, or `xor` operators without parentheses, such as

```
A and B and C and D
```

However, sequences with different operators, such as

```
A or B xor C
```

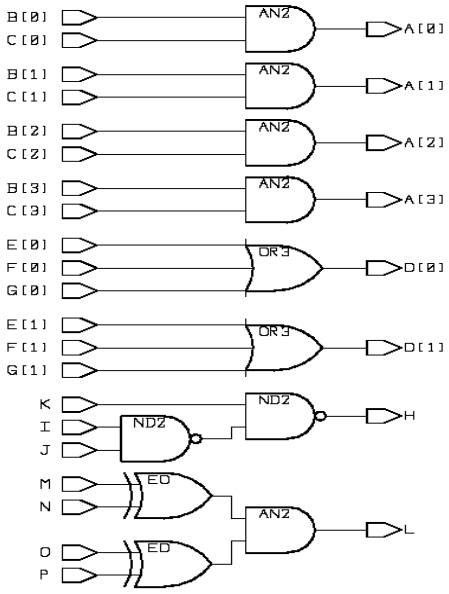
do require parentheses.

Example 5-3 uses the declarations from Example 5-2 to show some common errors.

Example 5-3: Errors in Using Logical Operators

```
H <= I and J or K;            -- Parenthesis required;
L <= M nand N nand O nand P;  -- Parenthesis required;
A <= B and E;       -- Operands must be the same size;
H <= I or L;        -- Operands must be the same type;
```

Figure 5-1: Common Errors Using Logical Operators



Relational Operators

Relational operators, such as = or >, compare two operands of the same base type and return a BOOLEAN value.

IEEE VHDL defines the equality (=) and inequality (/=) operators for all types. Two operands are equal if they represent the same value. For array and record types, IEEE VHDL compares corresponding elements of the operands.

IEEE VHDL defines the ordering operators (<, <=, "" (relational operator)">>, and ="" (relational operator)">>=) for all enumerated types, integer types, and one-dimensional arrays of enumeration or integer types.

The internal order of a type's values determines the result of the ordering operators. Integer values are ordered from negative infinity to positive infinity. Enumerated values are in the same order as they were declared, unless you have changed the encoding.

Note: If you set the encoding of your enumerated types (see ‘‘Enumeration Encoding" in Chapter 4), the ordering operators compare your encoded value ordering, not the declaration ordering. Because this interpretation is specific to FPGA Express, a VHDL simulator continues to use the declaration's order of enumerated types.

Arrays are ordered like words in a dictionary. The relative order of two array values is determined by comparing each pair of elements in turn, beginning from the left bound of each array's index range. If a pair of array elements is not equal, the order of the different elements determines the order of the arrays. For example, bit vector 101011 is less than 1011 because the fourth bit of each vector is different, and 0 is less than 1.

If the two arrays have different lengths and the shorter array matches the first part of the longer array, the shorter one is ordered before the longer. Thus, the bit vector `101` is less than `101000`. Arrays are compared from left to right, regardless of their index ranges (`to` or `downto`).

Example 5-4 shows several expressions that evaluate to `TRUE`.

Example 5-4: TRUE Relational Expressions

```
 '1'  =  '1'
"101" = "101"
 "1"  > "011"   -- Array comparison
"101" < "110"
```

To interpret bit vectors such as `011` as signed or unsigned binary numbers, use the relational operators defined in the FPGA Express `std_logic_arith` package (listed in Appendix B). The third line in Example 5-4 evaluates to `FALSE` if the operands are of type `UNSIGNED`.

```
UNSIGNED'"1"  < UNSIGNED'"011"   -- Numeric comparison
```

Example 5-5 shows some relational expressions and the resulting synthesized circuits.

Example 5-5: Relational Operators

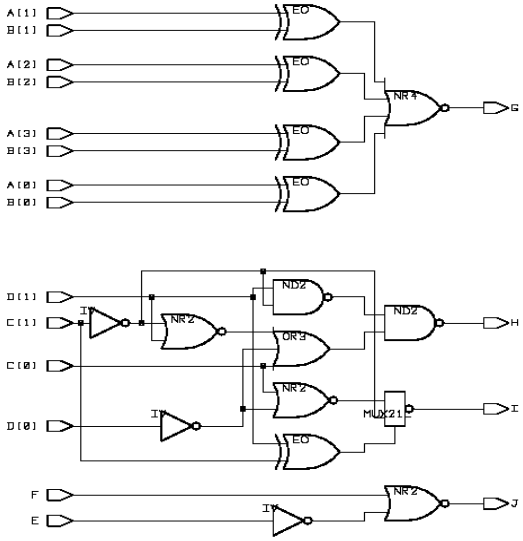
```
signal A, B: BIT_VECTOR(3 downto 0);
signal C, D: BIT_VECTOR(1 downto 0);
signal E, F, G, H, I, J: BOOLEAN;


G <= (A = B);
H <= (C < D);
I <= (C >= D);
J <= (E > F);
```

Adding Operators

Adding operators include *arithmetic* and *concatenation* operators.

The arithmetic operators `+` and `-` are predefined by FPGA Express for all integer operands. These addition and subtraction operators perform conventional arithmetic, as shown in Example 5-6. For adders and subtracters more than four bits wide, a synthetic library component is used (see Chapter 9).

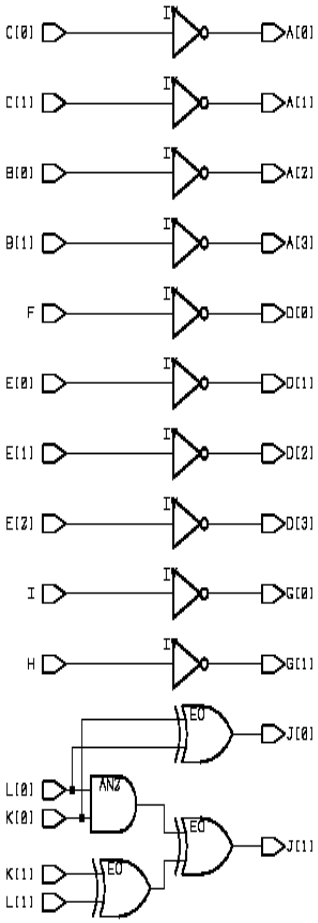
The concatenation (`&`) operator is predefined for all one-dimensional array operands. The concatenation operator builds arrays by combining the operands. Each operand of `&` can be an array or an element of an array. Use `&` to add a single element to the beginning or end of an array, to combine two arrays, or to build an array from elements, as shown in Example 5-6.


A[1]
B[1]
A[2]
B[2]
A[3]
B[3]
A[0]
B[0]
EO
EO
EO
EO
NR4
G
D[1]
C[1]
C[0]
D[0]
IV
NR2
ND2
OR3
NR2
IV
EO
ND2
MUX21
H
I
F
E
IV
NR2
J

Example 5-6: Adding Operators

```
signal A, D:    BIT_VECTOR(3 downto 0);
signal B, C, G: BIT_VECTOR(1 downto 0);
signal E:       BIT_VECTOR(2 downto 0);
signal F, H, I: BIT;

signal J, K, L: INTEGER range 0 to 3;

A <= not B & not C;  -- Array & array
D <= not E & not F;  -- Array & element
G <= not H & not I;  -- Element & element
J <= K + L;          -- Simple addition
```

Figure 5-2: Adding Operators



Unary (Sign) Operators

A unary operator has only one operand. FPGA Express predefines unary operators + and - for all integer types. The + operator has no effect. The - operator negates its operand. For example,

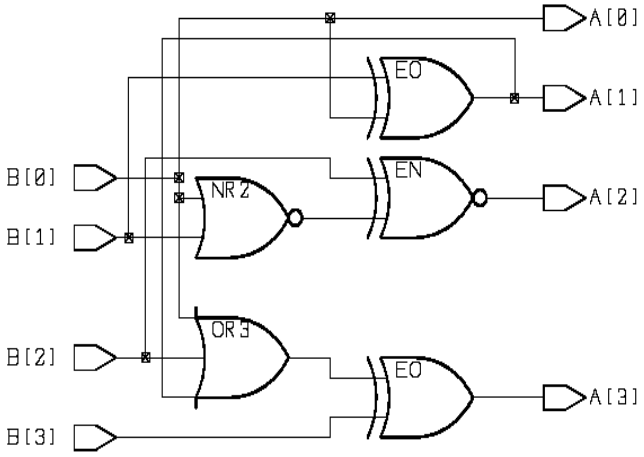
```
5 = +5
5 = -(-5)
```

Example 5-7 shows how unary negation is synthesized.

Example 5-7: Unary (Signed) Operators

```
signal A, B: INTEGER range -8 to 7;


A <= -B;
```

Figure 5-3: Unary (Signed) Operators



Multiplying Operators

FPGA Express predefines the multiplying operators (*, /, mod, and rem) for all integer types.

FPGA Express places some restrictions on the supported values for the right operands of the multiplying operators, as follows:

- * Integer multiplication: no restrictions.

  A multiplication operator is implemented as a synthetic library cell.
- / Integer division: The right operand must be a *computable* power of 2 (see "Computable Operands," later in this chapter). Neither operand can be negative.

  This operator is implemented as a bit shift.

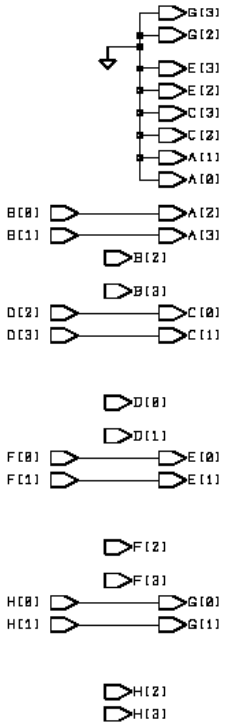
  `mod` Modulus: Same as `/`.

  `rem` Remainder: Same as `/`.

Example 5-8 shows some uses of the multiplying operators whose right operands are all powers of 2. The resulting synthesized circuit is also shown.

Example 5-8: Multiplying Operators with Powers of 2

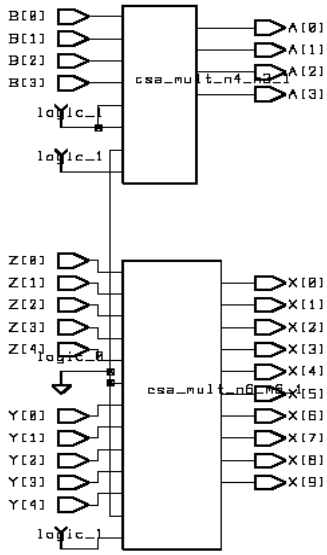
```
signal A, B, C, D, E, F, G, H: INTEGER range 0 to 15;

  A <= B * 4;
  C <= D / 4;
  E <= F mod 4;
  G <= H rem 4;
```



Example 5-9 shows two multiplication operations, one with a four-bit operand times a two-bit constant (`B * 3`), and one with two five-bit operands (`X * Y`). Because the synthetic library is enabled by default, these multiplications are implemented as synthetic library cells.

Example 5-9: Multiply Operator (*) Using Synthetic Cells

```
signal A, B: INTEGER range 0 to 15;
signal Y, Z: INTEGER range 0 to 31;
signal X:    INTEGER range 0 to 1023;
. . .
  A <= B * 3;
  X <= Y * Z;
```



Miscellaneous Arithmetic Operators

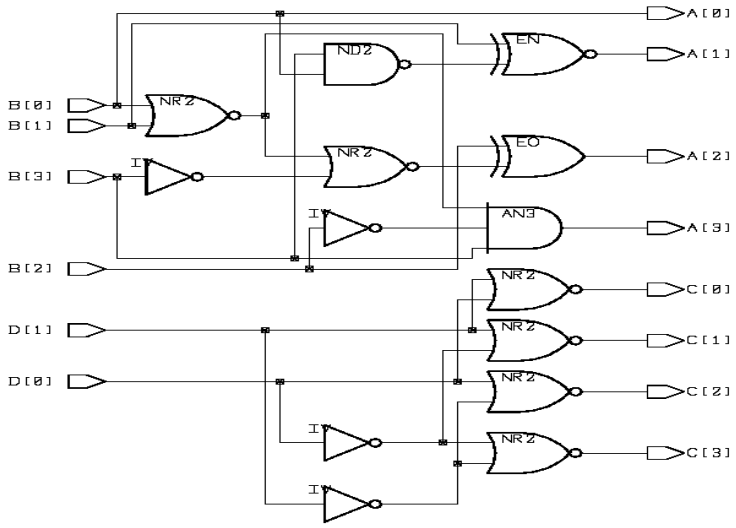
FPGA Express predefines the absolute value (`abs`) and exponentiation (`**`) operators for all integer types. One FPGA Express restriction placed on `**`, as follows:

`**` Exponentiation: Left operand must have a computable value of 2 (see ‘‘Computable Operands," later in this chapter).

Example 5-10 shows how these operators are used and synthesized.

Example 5-10: Miscellaneous Arithmetic Operators

```
signal A, B: INTEGER range -8 to 7;
signal C:    INTEGER range  0 to 15;
signal D:    INTEGER range  0 to 3;

A <= abs(B);
C <= 2 ** D;
```



Operands

Operands determine the data used by the operator to compute its value. An operand is said to return its value to the operator.

There are many categories of operands. The simplest operand is a literal, such as the number 7, or an identifier, such as a variable or signal name. An operand itself can be an expression. You create expression operands by surrounding an expression with parentheses.

The operand categories are

Expressions: `(A nand B)`

Literals: `'0', "101", 435, 16#FF3E#`

Identifiers: `my_var, my_sig`

Indexed names: `my_array(7)`

Slice names: `my_array(7 to 11)`

Fields: `my_record.a_field`

Aggregates:`my_array_type'(others => 1)`

Attributes: `my_array'range`

Function calls: `LOOKUP_VAL(my_var_1, my_var_2)`

Qualified expressions:`BIT_VECTOR'('1' & '0')`

Type conversions: `THREE_STATE('0')`

The next two sections discuss operand bit widths and explain computable operands. Subsequent sections describe the operand types listed above.

Operand Bit Width

FPGA Express uses the bit width of the largest operand to determine the bit width needed to implement an operator in hardware. For example, an `INTEGER` operand is 32 bits wide by default. An addition of two `INTEGER` operands causes FPGA Express to build a 32-bit adder.

To use hardware resources efficiently, always indicate the bit width of numeric operands. For example, use a subrange of `INTEGER` when declaring types, variables, or signals.

```
type     ENOUGH:  INTEGER range 0 to 255;
variable WIDE:    INTEGER range -1024 to 1023;
signal   NARROW:  INTEGER range 0 to 7;
```

Note: During optimization, FPGA Express removes hardware for unused bits.

Computable Operands

Some operators, such as the division operator, restrict their operands to be *computable*. A computable operand is one whose value can be determined by FPGA Express. Computability is important because noncomputable expressions can require logic gates to determine their value.

Following are examples of computable operands:

- Literal values
- `for ... loop` parameters, when the loop's range is computable
- Variables assigned a computable expression
- Aggregates that contain only computable expressions
- Function calls with a computable return value
- Expressions with computable operand
- Qualified expressions, where the expression is computable
- Type conversions, when the expression is computable
- Value of the `and` or `nand` operators when one of the operands is a computable `0`
- Value of the `or` or `nor` operators when one of the operands is a computable `1`

Additionally, a variable is given a computable value if it is an `OUT` or `INOUT` parameter of a procedure that assigns it a computable value.

Following are examples of noncomputable operands:

- Signals
- Ports
- Variables that are assigned different computable values that depend on a noncomputable condition
- Variables assigned noncomputable values

Example 5-11 shows some definitions and declarations, followed by several computable and noncomputable expressions.

Example 5-11: Computable and Noncomputable Expressions

```
signal S: BIT;
. . .
function MUX(A, B, C: BIT) return BIT is
begin
  if (C = '1') then
    return(A);
  else
    return(B);
  end if;
end;

procedure COMP(A: BIT; B: out BIT) is
begin
  B := not A;
end;

process(S)
  variable V0, V1, V2: BIT;
  variable V_INT:      INTEGER;

  subtype MY_ARRAY is BIT_VECTOR(0 to 3);
  variable V_ARRAY:    MY_ARRAY;
begin
  V0 := '1';             -- Computable (value is '1')
  V1 := V0;              -- Computable (value is '1')
  V2 := not V1;          -- Computable (value is '0')

  for I in 0 to 3 loop
```

```
    V_INT := I;          -- Computable (value depends
  end loop;              --   on iteration)

  V_ARRAY := MY_ARRAY'(V1, V2, '0', '0');
                         -- Computable ("1000")
  V1 := MUX(V0, V1, V2); -- Computable (value is '1')
  COMP(V1, V2);
  V1 := V2;              -- Computable (value is '0')
  V0 := S and '0';       -- Computable (value is '0')
  V1 := MUX(S, '1', '0');-- Computable (value is '1')
  V1 := MUX('1', '1', S);-- Computable (value is '1')

  if (S = '1') then
    V2 := '0';           -- Computable (value is '0')
  else
    V2 := '1';           -- Computable (value is '1')
  end if;
  V0 := V2;            -- Noncomputable; V2 depends
                       --   on S
  V1 := S;             -- Noncomputable; S is signal
  V2 := V1;            -- Noncomputable; V1 is no
                       --   longer computable
end process;
```

Literals

A literal (constant) operand can be a numeric literal, a character literal, an enumeration literal, or a string literal. The following sections describe these four kinds of literals.

Numeric Literals

Numeric literals are constant integer values. The two kinds of numeric literals are decimal and based. A decimal literal is written in base 10. A based literal can be written in a base from 2 to 16 and is composed of the base number, an octothorpe (#), the value in the given base, and another octothorpe (#); for example, `2#101#` is decimal 5.

The digits in either kind of numeric literal can be separated by an underscore (_) character. Example 5-12 shows several different numeric literals, all representing the same value.

Example 5-12: Numeric Literals

```
170
1_7_0
10#170#
2#1010_1010#
16#AA#
```

Character Literals

Character literals are single characters enclosed in single quotation marks, for example, `A`. Character literals can be used as values for operators and to define enumerated types, such as `CHARACTER` and `BIT`. See Chapter 4 for more information about the legal character types.

Enumeration Literals

Enumeration literals are values of enumerated types. The two kinds of enumeration literals are character literals and identifiers. Character literals were described previously. Enumeration identifiers are those literals listed in an enumeration type definition. For example:

```
type SOME_ENUM is ( ENUM_ID_1, ENUM_ID_2, ENUM_ID_3);
```

If two enumerated types use the same literals, those literals are said to be *overloaded*. You must qualify overloaded enumeration literals (see "Qualified Expressions," later in this chapter) when you use them in an expression unless their type can be determined from context. See Chapter 4 for more information.

Example 5-13 defines two enumerated types and shows some enumeration literal values.

Example 5-13: Enumeration Literals

```
type ENUM_1 is (AAA, BBB, 'A', 'B', ZZZ);
type ENUM_2 is (CCC, DDD, 'C', 'D', ZZZ);

AAA           -- Enumeration identifier of type ENUM_1
'B'           -- Character literal of type ENUM_1
CCC           -- Enumeration identifier of type ENUM_2
'D'           -- Character literal of type ENUM_2
ENUM_1'(ZZZ)  -- Qualified because overloaded
```

String Literals

String literals are one-dimensional arrays of characters, enclosed in double quotes (" "). The two kinds of string literals are character strings and bit strings. *Character strings* are sequences of characters in double quotes; for example, `"ABCD"`. *Bit strings* are similar to character strings, but represent binary, octal, or hexadecimal values; for example, `B"1101"`, `O"15"`, and `X"D"` all represent decimal value 13.

A string value's type is a one-dimensional array of an enumerated type. Each of the characters in the string represents one element of the array.

Example 5-14 shows some character-string literals.

Example 5-14: Character-String Literals

```
"10101"
"ABCDEF"
```

Note: Null string literals ("") are not supported.

Bit strings, like based numeric literals, are composed of a *base specifier character*, a double quotation mark, a sequence of numbers in the given base, and another double quotation mark. For example, B"0101" represents the bit vector 0101. A bit-string literal consists of the base specifier B, O, or X, followed by a string literal. The bit-string literal is interpreted as a *bit vector*, a one-dimensional array of the predefined type BIT. The base specifier determines the interpretation of the bit string as follows:

B (binary)
The value is in binary digits (*bits*, 0 or 1). Each bit in the string represents one BIT in the generated bit vector (array).

O *(octal)*
The value is in octal digits (0 to 7). Each octal digit in the string represents three BITs in the generated bit vector (array).

X *(hexadecimal)*
The value is in hexadecimal digits (0 to 9 and A to F). Each hexadecimal digit in the string represents four BITs in the generated bit vector (array).

You can separate the digits in a bit-string literal value with underscores (_) for readability. Example 5-15 shows several bit-string literals that represent the same value.

Example 5-15: Bit-String Literals

```
X"AAA"
B"1010_1010_1010"

O"5252"
B"101_010_101_010"
```

Identifiers

Identifiers are probably the most common operand. An identifier is the name of a constant, variable, signal, entity, port, subprogram, or parameter and returns the object's value to an operand.

Example 5-16 shows several kinds of identifiers and their usage. All identifiers are shown in boldface.

Example 5-16: Identifiers

```
entity EXAMPLE is
  port (INT_PORT:   in INTEGER;
        BIT_PORT:  out BIT);
end;
. . .
signal   BIT_SIG: BIT;
signal   INT_SIG: INTEGER;
```

```
. . .
INT_SIG  <= INT_PORT;   -- Signal assignment from port
BIT_PORT <= BIT_SIG;    -- Signal assignment to port

function FUNC(INT_PARAM:  INTEGER)
    return INTEGER;
end function;
. . .
constant CONST:   INTEGER := 2;
variable VAR:     INTEGER;
. . .
VAR := FUNC(INT_PARAM => CONST);  -- Function call
```

Indexed Names

An *indexed name* identifies one element of an array variable or signal. *Slice names* identify a sequence of elements in an array variable or signal; *aggregates* create array literals by giving a value to each element of an instance of an array type. Slice names and aggregates are described in the next two sections.

The syntax of an indexed name is

identifier **(** *expression* **)**

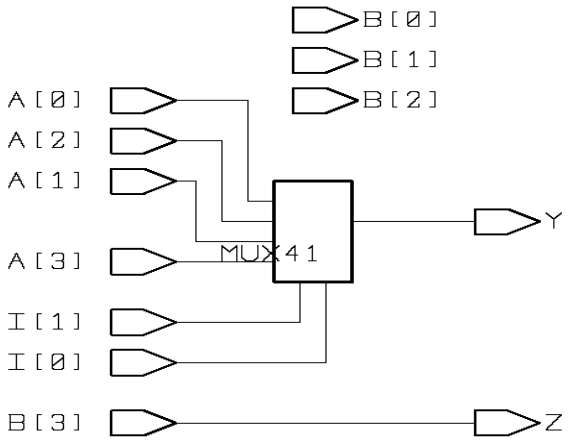
identifier must name a signal or variable of an array type. The *expression* must return a value within the array's index range. The value returned to an operator is the specified array element.

If *expression* is computable (see ‘‘Computable Operands," earlier in this chapter), the operand is synthesized directly. If the expression is not computable, hardware that extracts the specified element from the arrayis synthesized.

Example 5-17 shows two indexed names—one computable and one not computable.

Example 5-17: Indexed Name Operands

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

Y <= A(I);  -- Noncomputable index expression
Z <= B(3);  -- Computable index expression
```



You can also use indexed names as assignment targets; see "Indexed Name Targets" in Chapter 6.

Slice Names

Slice names return a sequence of elements in an array. The syntax is

identifier (*expression direction expression*)

identifier must name a signal or variable of an array type. Each *expression* must return a value within the array's index range, and must be computable. See ‘‘Computable Operands," earlier in this chapter.

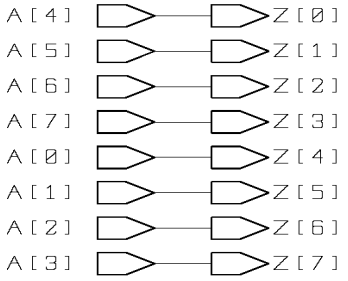
The `direction` must be either `to` or `downto`. The direction of a slice must be the same as the direction of `identifier` array type. If the left and right expressions are equal, define a single element.

The value returned to an operator is a subarray containing the specified array elements.

Example 5-18 uses slices to assign an eight-bit input to an eight-bit output, exchanging the lower and upper four bits.

Example 5-18: Slice Name Operands

```
signal A, Z: BIT_VECTOR(0 to 7);

Z(0 to 3) <= A(4 to 7);
Z(4 to 7) <= A(0 to 3);
```



In Example 5-18, slices are also used as assignment targets. This usage is described in Chapter 6, under ‘‘Slice Targets."

Limitations on Null Slices

FPGA Express does not support null slices. A null slice is indicated by a null range, such as `(4 to 3)`, or a range with the wrong direction, such as `UP_VAR(3 downto 2)` when the declared range of `UP_VAR` is ascending (Example 5-19).

Example 5-19 shows three null slices and one noncomputable slice.

Example 5-19: Null and Noncomputable Slices

```
subtype DOWN is BIT_VECTOR(4 downto 0);
subtype UP   is BIT_VECTOR(0 to 7);
. . .
variable UP_VAR:   UP;
variable DOWN_VAR: DOWN;
. . .
UP_VAR(4 to 3)       -- Null slice (null range)

UP_VAR(4 downto 0)   -- Null slice (wrong direction)
DOWN_VAR(0 to 1)     -- Null slice (wrong direction)
. . .

variable I: INTEGER range 0 to 7;
. . .
UP_VAR(I to I+1)     -- Noncomputable slice
```

Limitations on Noncomputable Slices

IEEE VHDL does not allow noncomputable slices—slices whose range contains a noncomputable expression.

Records and Fields

Records are composed of named fields of any type. For more information, see ‘‘Record Types" in Chapter 4.

In an expression, you can refer to a record as a whole, or you can refer to a single field. The syntax of field names is

```
record_name.field_name
```

record_name is the name of the record variable or signal, and *field_name* is the name of a field in that record type. A field_name is separated from the record name by a period (.). Note that a *record_name* is different for each variable or signal of that record type. A *field_name* is the field name defined for that record type.

Example 5-20 shows a record type definition, and record and field access.

Example 5-20: Record and Field Access

```
type BYTE_AND_IX is
  record
    BYTE: BIT_VECTOR(7 downto 0);
    IX:   INTEGER range 0 to 7;
  end record;

signal X: BYTE_AND_IX;
. . .
X           -- record
X.BYTE      -- field: 8-bit array
X.IX        -- field: integer
```

A field can be of any type—including an array, record, or aggregate type. Refer to an element of a field with that type’s notation, for example:

```
X.BYTE(2)           -- one element from array field BYTE
X.BYTE(3 downto 0)  -- 4-element slice of array field BYTE
```

Aggregates

Aggregates can be considered array literals, because they specify an array type and the value of each array element. The syntax is

type_name'**(***[choice* **=>***] expression*
{, [choice **=>***] expression}***)**

Note that the syntax is more restrictive than the syntax in the Library Reference Manual (LRM). *type_name* must be a constrained array type. The optional *choice* specifies an element index, a sequence of indexes, or `others`. Each *expression* provides a value for the chosen elements, and must evaluate to a value of the element's type.

Example 5-21 shows an array type definition and an aggregate representing a literal of that array type. The two sets of assignments have the same result.

Example 5-21: Simple Aggregate

```
subtype MY_VECTOR is BIT_VECTOR(1 to 4);
signal X:      MY_VECTOR;
variable A, B: BIT;

X <= MY_VECTOR'('1', A nand B, '1', A or B)  -- Aggregate
                                             -- assignment
...
X(1) <= '1';                                 -- Element
X(2) <= A nand B;                            -- assignment
X(3) <= '1';
X(4) <= A or B;
```

You can specify an element's index with either positional or named notation. With positional notation, each element is given the value of its expression in order, as shown in Example 5-21.

By using named notation, the `choice =>` construct specifies one or more elements of the array. The choice can contain an expression (such as `(I mod 2) =>`) to indicate a single element index, or a range (such as `3 to 5 =>` or `7 downto 0 =>`) to indicate a sequence of element indexes.

An aggregate can use both positional and named notation, but positional expressions must appear before named (*choice*) expressions.

It is not necessary to specify all element indexes in an aggregate. All unassigned values are given a value by including `others =>` *expression* as the last element of the list.

Example 5-22 shows several aggregates representing the same value.

Example 5-22: Equivalent Aggregates

```
subtype MY_VECTOR is BIT_VECTOR(1 to 4);


MY_VECTOR'('1', '1', '0', '0');
MY_VECTOR'(2 => '1', 3 => '0', 1 => '1', 4 => '0');
MY_VECTOR'('1', '1', others => '0');
MY_VECTOR'(3 => '0', 4 => '0', others => '1');
MY_VECTOR'(3 to 4 => '0', 2 downto 1 => '1');
```

The `others` expression must be the only expression in the aggregate. Example 5-23 shows two equivalent aggregates.

Example 5-23: Equivalent Aggregates Using the others Expression

```
MY_VECTOR'(others => '1');
MY_VECTOR'('1', '1', '1', '1');
```

To use an aggregate as the target of an assignment statement, see ‘‘Aggregate Targets" in Chapter 6.

Attributes

VHDL defines attributes for various types. A VHDL attribute takes a variable or signal of a given type and returns a value. The syntax of an attribute is

object'attribute

FPGA Express supports the following predefined VHDL attributes for use with arrays, as described under ‘‘Array Types" in Chapter 4:

- left
- right
- high
- low
- length
- range
- reverse_range

FPGA Express also supports the following predefined VHDL attributes for use with `wait` and `if` statements, as described in Chapter 8, "Register and Three-State Inference":

- event
- stable

In addition to supporting predefined VHDL attributes listed above, FPGA Express has a defined set of synthesis-related attributes. These FPGA Express-specific attributes can be placed in your VHDL design description to direct optimization. See ‘‘Synthesis Attributes and Constraints" in Chapter 9 for more information.

Function Calls

A function call executes a named function with the given parameter values. The value returned to an operator is the function’s return value. The syntax of a function call is

function_name **(** [*parameter_name* **=>**] *expression*
{**,** [*parameter_name* **=>**] *expression* } **)**

function_name is the name of a defined function. The optional *parameter_name* is an expression of formal parameters, as defined by the function. Each *expression* provides a value for its parameter, and must evaluate to a type appropriate for that parameter.

You can specify parameters in positional or named notation, like aggregate values.

In positional notation, the *parameter_name* => construct is omitted. The first expression provides a value for the function's first parameter, the second expression provides a value for the second parameter, and so on.

In named notation, *parameter_name* => is specified before an expression; the named parameter gets the value of that expression.

You can mix positional and named expressions in the same function call, as long as all positional expressions appear before a named parameter expressions.

Function calls are implemented by logic unless you use the `map_to_entity` compiler directive. For more information, see "Mapping Subprograms to Components" in Chapter 6, and "Component Implication Directives" in Chapter 9.

Example 5-24 shows a function declaration and several equivalent function calls.

Example 5-24: Function Calls

```
function FUNC(A, B, C: INTEGER) return BIT;
. . .
FUNC(1, 2, 3)
FUNC(B => 2, A => 1, C => 7 mod 4)
FUNC(1, 2, C => -3+6)
```

Qualified Expressions

Qualified expressions state the type of an operand to resolve ambiguities in an operand's type. You cannot use qualified expressions for *type conversion* (see "Type Conversions").

The syntax of a qualified expression is

type_name'**(***expression***)**

type_name is the name of a defined type. *expression* must evaluate to a value of an appropriate type.

Note: A single quote, or tick, must appear between `type_name` and `(expression)`. If the single quote is omitted, the construction is interpreted as a type conversion (see "Type Conversions").

Example 5-25 shows a qualified expression that resolves an overloaded function by qualifying the type of a decimal literal parameter.

Example 5-25: A Qualified Decimal Literal

```
type R_1 is range 0 to 10;  -- Integer 0 to 10
type R_2 is range 0 to 20;  -- Integer 0 to 20

function FUNC(A: R_1) return BIT;
function FUNC(A: R_2) return BIT;

FUNC(5)         -- Ambiguous; could be of type R_1,
                --   R_2, or INTEGER

FUNC(R_1'(5))   -- Unambiguous
```

Example 5-26 shows how qualified expressions resolve ambiguities in aggregates and enumeration literals.

Example 5-26: Qualified Aggregates and Enumeration Literals

```
type ARR_1 is array(0 to 10) of BIT;
type ARR_2 is array(0 to 20) of BIT;
. . .
(others => '0')        -- Ambiguous; could be of
                       -- type ARR_1 or ARR_2

ARR_1'(others => '0')  -- Qualified; unambiguous
-----------------------------------------------------
type ENUM_1 is (A, B);
type ENUM_2 is (B, C);
. . .
B                      -- Ambiguous; could be of
                       -- type ENUM_1 or ENUM_2

ENUM_1'(B)             -- Qualified; unambiguous
```

Type Conversions

Type conversions change an expression's type. Type conversions are different from qualified expressions because they change the type of their expression; whereas qualified expressions simply resolve the type of an expression.

The syntax of a type conversion is

*type_name***(***expression***)**

type_name is the name of a defined type. The *expression* must evaluate to a value of a type that can be converted into type *type_name.*

- Type conversions can convert between integer types or between similar array types.
- Two array types are similar if they have the same length and if they have convertible or identical element types.
- Enumerated types cannot be converted.

Example 5-27 shows some type definitions and associated signal declarations, followed by legal and illegal type conversions.

Example 5-27: Legal and Illegal Type Conversions

```
type INT_1 is range 0 to 10;
type INT_2 is range 0 to 20;

type ARRAY_1 is array(1 to 10) of INT_1;
type ARRAY_2 is array(11 to 20) of INT_2;

subtype MY_BIT_VECTOR is BIT_VECTOR(1 to 10);
type BIT_ARRAY_10 is array(11 to 20) of BIT;
type BIT_ARRAY_20 is array(0 to 20) of BIT;

signal S_INT:      INT_1;
signal S_ARRAY:    ARRAY_1;
signal S_BIT_VEC:  MY_BIT_VECTOR;
signal S_BIT:      BIT;

       -- Legal type conversions

INT_2(S_INT)
  -- Integer type conversion

BIT_ARRAY_10(S_BIT_VEC)
  -- Similar array type conversion

       -- Illegal type conversions

BOOLEAN(S_BIT);
  -- Can't convert between enumerated types

INT_1(S_BIT);
  -- Can't convert enumerated types to other types
```

```
BIT_ARRAY_20(S_BIT_VEC);
  -- Array lengths not equal

ARRAY_1(S_BIT_VEC);
  -- Element types cannot be converted
```

Chapter 6
Sequential Statements

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 can be 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 following descriptions 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 but 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 contain the array's element type.

In Example 6-2, the elements for array variable A are assigned values as indexed names.

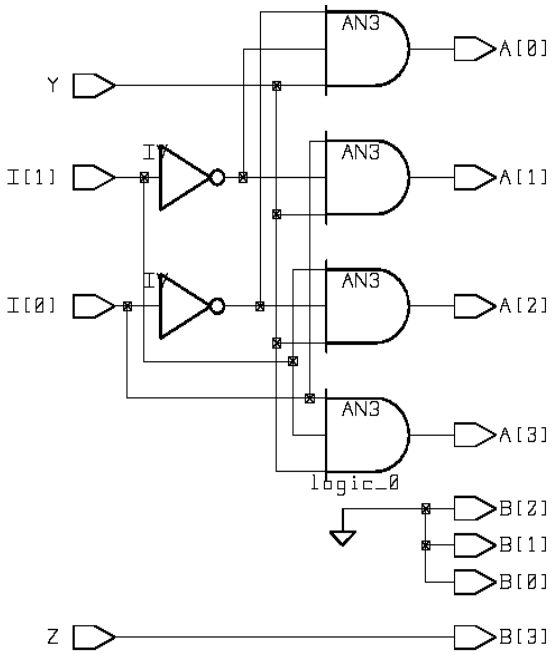
Example 6-2: Indexed Name Targets

```
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 of the targets is computable and 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;  -- Noncomputable index expression
B(3) <= Z;  -- Computable index expression
```



Slice Targets

The syntax for a slice target is

identifier(*index_expr_1 direction index_expr_2*)

identifier is the name of an array type signal or variable. Each *index_expr* expression must evaluate to an index value for the *identifier* array's index type and bounds. Both *index_expr* expressions must be computable (see -Computable Operands- in Chapter 5), and must lie within the bounds of the array. *direction* must match the *identifier* array type's direction—either `to` or `downto`.

The assigned expression must contain the array's element type.

In Example 6-4, array variables A and B are assigned the same value.

Example 6-4: Slice Targets

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


-- Target       Expression;
   A(1 to 2) := -11";  -- Assigns -11" to the first
                       -- two elements of array A
   A(3 to 4) := -00";  -- Assigns -00" to the last
                       -- two elements of array A
   B(1 to 4) := -1100";-- Assigns -1100" to array B
```

Field Targets

The syntax for a field target is

identifier.*field_name*

identifier is the name of a record type signal or variable, and *field_name* is the name of a field in that record type, preceded by a period (.). The assigned expression must contain 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 A and B.

Example 6-5: Field Targets

```
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 about field targets see -Record Types- in Chapter 4.

Aggregate Targets

The syntax for an assignment to an aggregate target is

```
([choice =>] identifier
 {,[choice =>] identifier}) := array_expression;
  -- Variable assignment

([choice =>] identifier
 {,[choice =>] identifier}) <= array_expression;
  -- Signal assignment
```

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

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

Example 6-6 shows some aggregate targets.

Example 6-6: Aggregate Targets

```
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 type of *identifier* must match the assigned expression's element type.

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

Variable Assignment Statement

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

target **:=** *expression*;

expression determines the assigned value; its type must be compatible with *target*. See Chapter 5 for further information about expressions. *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 it is assigned a new value.

Signal Assignment Statement

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

target **<=** *expression*;

expression determines the assigned value; its type must be compatible with *target*. See Chapter 5 for further information about expressions. *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 takes place immediately. A variable keeps its assigned value until it is assigned a new 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, read, and reassigned, 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

```
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

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.

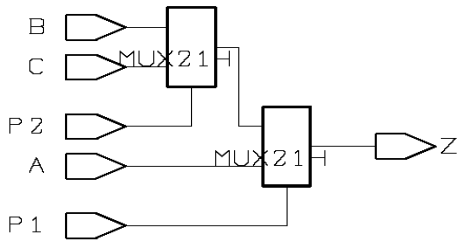
If none of the conditions `are TRUE`, and the `else` clause is present, those statements are executed.

If none of the conditions `are 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 FPGA Express to infer registers or latches, as described in Chapter 8, ‘‘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

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

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

```
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 the choice_expression type must be covered by one choice.

The final choice can be others, which matches all remaining (unchosen) values in the range of the expression 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 statements following each when clause is evaluated only if the choice value matches the expression value.

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.

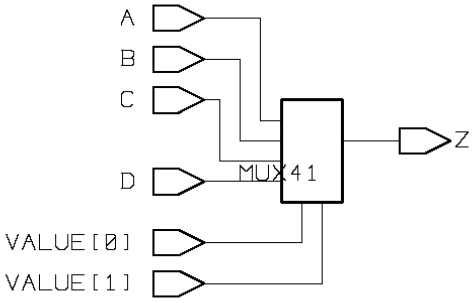
Using Different Expression Types

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

```
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 =>
```

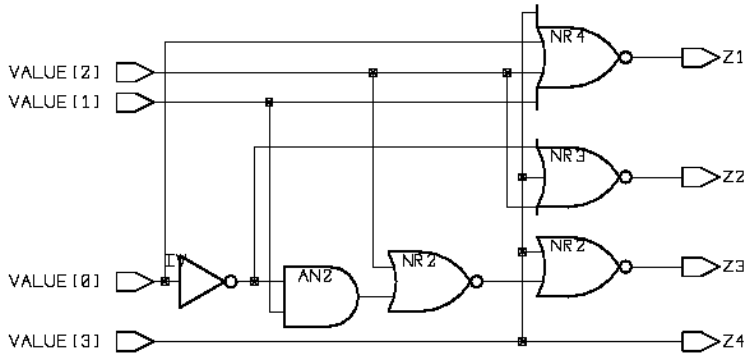
```
    Z <= A;
  when PICK_B =>
    Z <= B;
  when PICK_C =>
    Z <= C;
  when PICK_D =>
    Z <= D;
end case;
```



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

```
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;
```



Invalid *case* Statements

Example 6-11 shows four invalid case statements.

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
end case;              --   clause

case VALUE is          -- Values 2 to 15 are not
  when 0 =>            --   covered by choices
    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 to TRUE, enclosed statements are executed once. The iteration condition is then reevaluated. While the iteration condition remains true, the loop is repeatedly executed. When the iteration condition evaluates to 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.

Caution ***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 might result.

loop Statement

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

```
[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 to TRUE. The syntax is

```
[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.
- The value of *identifier* can be read only inside its loop (*identifier* does not exist outside the loop). You cannot assign a value to a loop identifier.

FPGA Express currently requires that *range* must be a computable integer range(see ‘‘Computable Operands- in Chapter 5), 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.

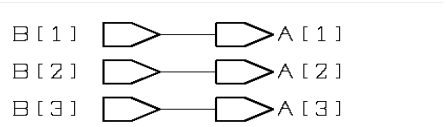
A `for .. loop` statement executes as follows:

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

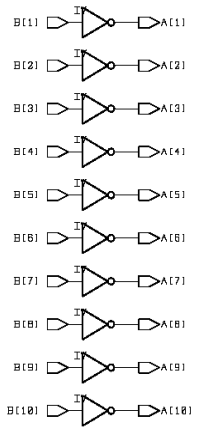
```
variable A, B: BIT_VECTOR(1 to 3);

-- 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

next *[label] [* **when** *condition]* **;**

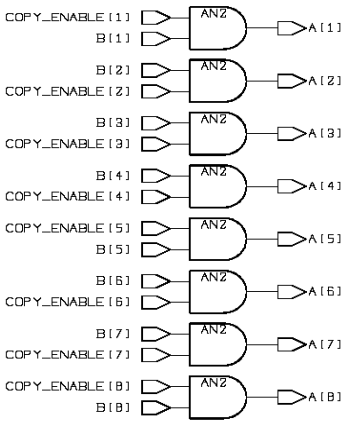
A `next` statement with no *`label`* terminates the current iteration of the innermost enclosing loop. When you specify a loop *`label`*, the current iteration of that named loop is terminated.

The optional `when` clause executes its `next` statement when its *`condition`* (a Boolean expression) evaluates to 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 to `TRUE`.

Example 6-14: next Statement

```
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 vector `X` against each of the first two elements of vector `Y`,
- The third element of vector `X` against each of the first three elements of vector `Y`,

The processing continues in this fashion until it is completed.

Example 6-15: Named next Statement

```
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

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 earlier in Example 6-15.

The optional `when` clause executes its `exit` statement when its *`condition`* (a Boolean expression) evaluates `TRUE`.

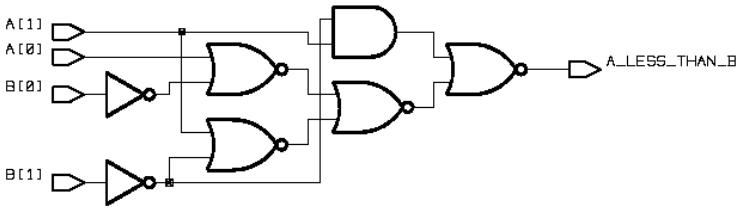
The `exit` and `next` statements are equivalent constructs. Both statements use identical syntax, and both skip the remainder of the enclosing (or named) loop. The only difference between the two statements is that `exit` terminates its loop, and `next` continues with the next loop iteration (if any).

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

Example 6-16: Comparator Using the exit Statement

```
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;
```



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 and 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, whereas 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.

FPGA Express implements procedure and function calls with combinational logic, unless you use the `map_to_entity` compiler directive (see ‘‘Mapping Subprograms to Components),- later in this chapter). FPGA Express 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

```
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 ‘‘Subprograms- 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) can 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.

Procedures and functions are two kinds of subprograms:

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 might 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 appear before named parameters.

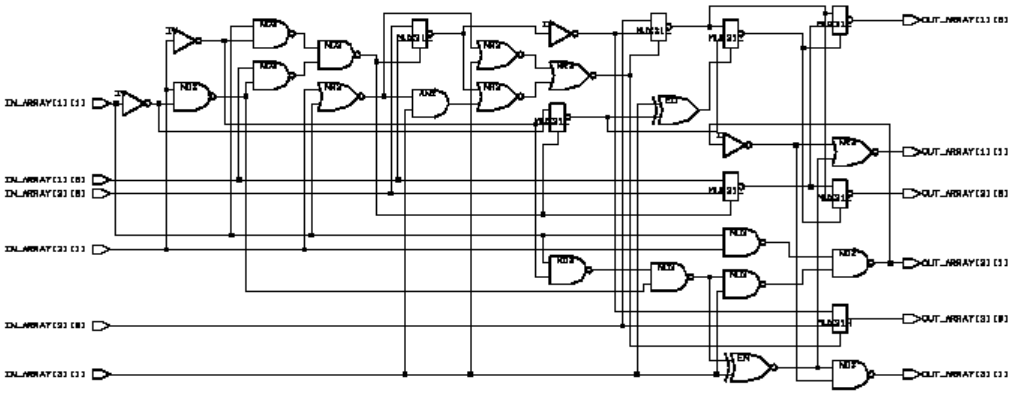
Conceptually, a procedure call is performed in three steps. First, the values of the `in` and `inout` actual parameters are assigned to their associated formal parameters. Second, the procedure is executed. Third, the values of the `inout` and `out` formal parameters are assigned to the actual parameters.

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 these elements if they are out of order. SWAP is repeatedly called 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 first and second 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 type 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

```
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. This statement is required in function definitions and is optional in procedure definitions. The syntax is

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

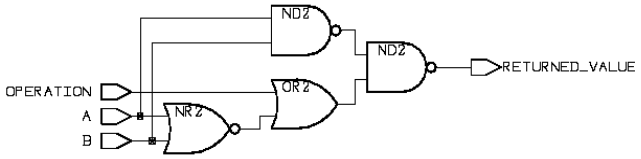
The required *`expression`* provides the function's return value. Every function must have at least one `return` statement. The expression's type must match the declared function type. A function can have more than one `return` statement. Only one `return` statement is reached by a given function call.

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 FPGA Express to implement a function or procedure as a component instantiation. Procedures and functions that use `map_to_entity` are represented as components in designs in which they are called.

You can also use the FPGA Express Implementation Window to create a new level of hierarchy from a VHDL subprogram, as described in the *FPGA Express User's Guide.*

When you add a `map_to_entity` directive to a subprogram definition, FPGA Express assumes the existence of an entity with the identified name and the same interface. FPGA Express 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:

```
  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 FPGA Express encounters 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 that FPGA Express supports.

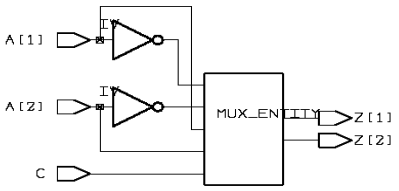
Caution ***The behavioral description of the subprogram is not checked against the functionality of the entity overloading it. Presynthesis and post-synthesis simulation results might not match if differences in functionality exist between the VHDL subprogram and the overloaded entity.***

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;
```

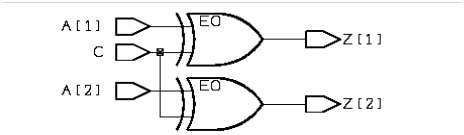
```
-- the following entity 'overloads' the function
-- MUX_FUNC above

entity MUX_ENTITY is
  port(A, B: in TWO_BIT; C: in BIT; Z: out TWO_BIT);
end;

architecture ARCH of MUX_ENTITY is
begin
  process
  begin
      case C is
          when '1' => Z <= A;
          when '0' => Z <= B;
      end case;
  end process;
end;
```



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;
```



wait Statement

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

```
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 FPGA Express.

Inferring Synchronous Logic

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

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

Example 6-23: Equivalent wait Statements

```
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

```
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 Using a wait Statement

```
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 A over four clock cycles.

Example 6-26: Using Multiple wait Statements

```
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 description uses implicit state logic, and the second uses 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 contains one or more `wait` statements, all paths must contain at least one `wait` statement.

Example 6-28 shows how a circuit with synchronous reset can be described with `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 Using 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 `wait` statements. These limitations are specific to FPGA Express.

Example 6-29: Invalid Uses of the wait Statement

```
...
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 wait statements
    ...
  end;
```

Combinational vs. Sequential Processes

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

If 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:

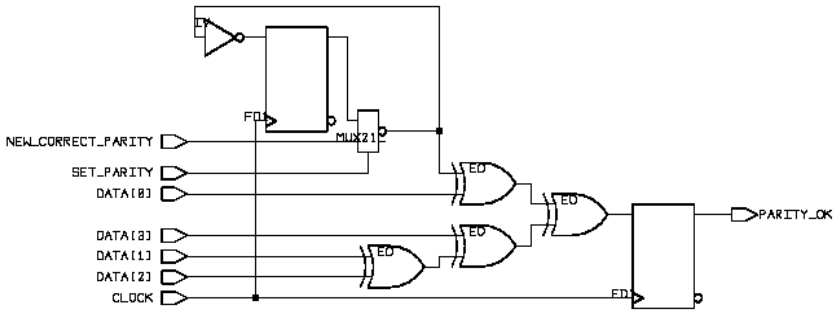
- Signals driven by the process; see ‘‘Signal Assignment Statement- at the beginning of this chapter.
- State vector values, where the state vector can be implicit or explicit (as in Example 6-27).
- Variables that *may* be read before they are set.

Note: Like the `wait` statement, some uses of the `if` statement can also imply synchronous logic, causing FPGA Express 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 Using 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 stores 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

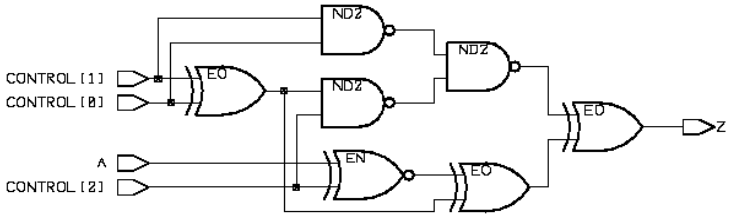
The null statement explicitly states that no action is required. The null statement is often used in case statements because all choices must be covered, even if some of the choices are ignored. The syntax is

```
null;
```

Example 6-31 shows a typical usage of the null statement.

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 =>
    null;            -- If not 0 or 7, then do nothing
end case;
```


ND2
ND2
ND2
CONTROL[1]
CONTROL[0]
EO
EO
A
CONTROL[2]
EN
EO
Z

Chapter 7
Concurrent Statements

A VHDL architecture contains a set of concurrent statements. Each concurrent statement defines one of the interconnected blocks or processes that describe the overall behavior or structure of a design. Concurrent statements in a design execute continuously, unlike sequential statements (see Chapter 6), which execute one after another.

The two main concurrent statements are

process statement
A process statement defines a process. Processes are composed of sequential statements (see Chapter 6), but processes are themselves concurrent statements. 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.

block statement
A block statement defines a block. Blocks are named collections of concurrent statements, optionally using locally defined types, signals, subprograms, and components.

VHDL provides two concurrent versions of sequential statements: concurrent procedure calls and concurrent signal assignments.

The component instantiation statement references a previously defined hardware component.

Finally, the `generate` statement creates multiple copies of any concurrent statement.

The concurrent statements consist of

- *process* Statements
- *block* Statement
- Concurrent Procedure Calls
- Concurrent Signal Assignments
- Component Instantiations
- *generate* Statements

process Statements

A `process` statement contains an ordered set of sequential statements. The syntax is

```
[ label: ] process [ ( sensitivity_list ) ]
     { process_declarative_item }
begin
     { sequential_statement }
end process [ label ] ;
```

An optional *`label`* names the process. The *`sensitivity_list`* is a list of all signals (including ports) read by the process, in the following format:

`signal_name` {, *`signal_name`*}

The hardware synthesized by FPGA Express is sensitive to all signals read by the process. To guarantee that a VHDL simulator sees the same results as the synthesized hardware, a process sensitivity list must contain all signals whose changes require resimulation of that process. FPGA Express checks sensitivity lists for completeness and issues warning messages for any signals that are read inside a process but are not in the sensitivity list. An error is issued if a clock signal is read as data in a process.

Note: IEEE VHDL does not allow a sensitivity list if the process includes a `wait` statement.

A *`process_declarative_item`* declares subprograms, types, constants, and variables local to the process. These items can be any of the following items:

- `use` clause
- Subprogram declaration
- Subprogram body
- Type declaration
- Subtype declaration
- Constant declaration
- Variable declaration

Each *`sequential_statement`* is described in Chapter 6.

Conceptually, the behavior of a process is defined by the sequence of its statements. After the last statement in a process is executed, execution continues with the first statement. The only exception is during simulation: if a process has a sensitivity list, the process is suspended (after its last statement) until a change occurs in one of the signals in the sensitivity list.

If a process has one or more `wait` statements (and therefore no sensitivity list), the process is suspended at the first `wait` statement whose wait condition is FALSE.

The hardware synthesized for a process is either combinational (not clocked) or sequential (clocked). If a process includes a `wait` or `if` *`signal'`*`event` statement, its hardware contains sequential components. The `wait` and `if` statements are described in Chapter 6.

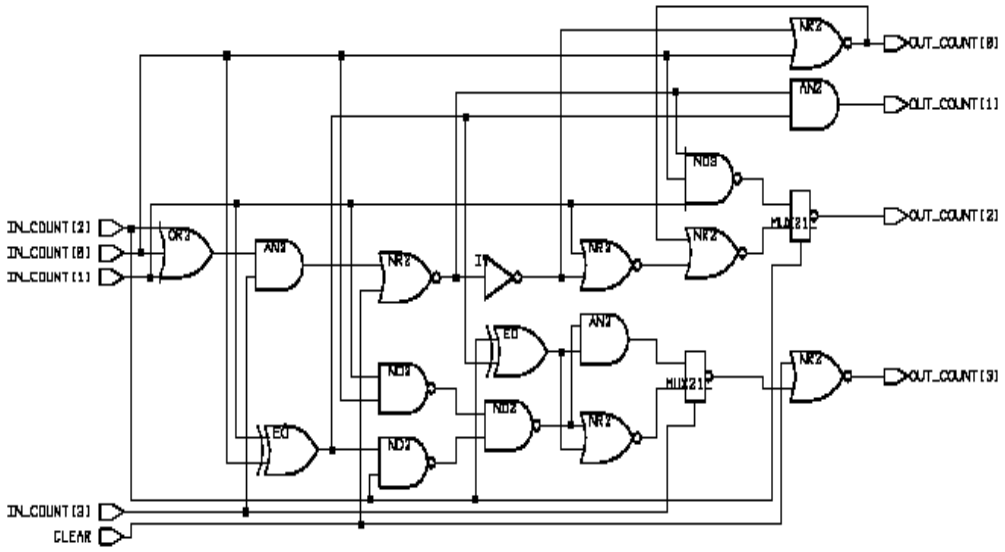
Note: The `process` statements provide a natural means for describing conceptually sequential algorithms. If the values computed in a process are inherently parallel, consider using concurrent signal assignment statements (see ‘‘Concurrent Signal Assignments," later in this chapter).

Combinational Process Example

Example 7-1 shows a process that implements a simple modulo-10 counter. The example process is sensitive to (reads) two signals: `CLEAR` and `IN_COUNT`. It drives one signal, `OUT_COUNT`. If `CLEAR` is `'1'` or `IN_COUNT` is 9, then `OUT_COUNT` is set to zero. Otherwise, `OUT_COUNT` is set to one more than `IN_COUNT`.

Example 7-1: Modulo-10 Counter Process

```
entity COUNTER is
   port (CLEAR:      in BIT;
         IN_COUNT:   in INTEGER range 0 to 9;
         OUT_COUNT: out INTEGER range 0 to 9);
end COUNTER;

architecture EXAMPLE of COUNTER is
begin
  process(IN_COUNT, CLEAR)
  begin
     if (CLEAR = '1' or IN_COUNT = 9) then
        OUT_COUNT <= 0;
     else
        OUT_COUNT <= IN_COUNT + 1;
     end if;
  end process;
end EXAMPLE;
```



Sequential Process Example

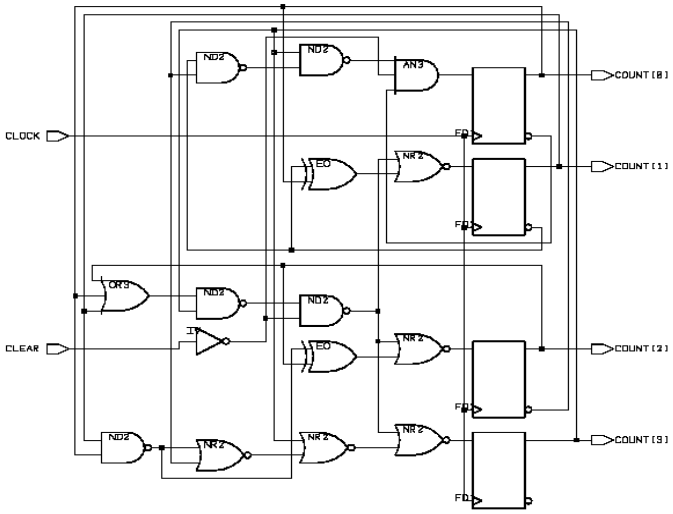
Because the process in Example 7-1 contains no `wait` statements, it is synthesized with combinational logic. An alternate implementation of the counter is to retain the count value internally in the process with a `wait` statement.

Example 7-2 shows an implementation of a counter as a sequential (clocked) process. On each 0-to-1 `CLOCK` transition, if `CLEAR` is `'1'` or `COUNT` is 9, `COUNT` is set to zero; otherwise, `COUNT` is incremented by 1.

Example 7-2: Modulo-10 Counter Process with wait Statement

```
entity COUNTER is
   port (CLEAR: in BIT;
         CLOCK: in BIT;
         COUNT: buffer INTEGER range 0 to 9);
end COUNTER;

architecture EXAMPLE of COUNTER is
begin
  process
  begin
     wait until CLOCK'event and CLOCK = '1';

     if (CLEAR = '1' or COUNT >= 9) then
        COUNT <= 0;
```

```
     else
        COUNT <= COUNT + 1;
     end if;
  end process;
end EXAMPLE;
```



In Example 7-2, the value of the variable `COUNT` is stored in four flip-flops. These flip-flops are generated because `COUNT` can be read before it is set, so its value must be maintained from the previous clock cycle. See ‘‘*wait* Statement" in Chapter 6 for more information.

Driving Signals

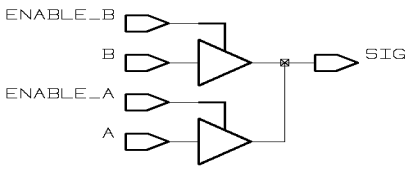
If a process assigns a value to a signal, the process is a *driver* of that signal. If more than one process or other concurrent statement drives a signal, that signal has *multiple drivers*.

Example 7-3 shows two three-state buffers driving the same signal (`SIG`). Chapter 8 shows how to describe a three-state device in technology-independent VHDL, in the section on ‘‘Three-State Inference."

Example 7-3: Multiple Drivers of a Signal

```
A_OUT <= A when ENABLE_A else 'Z';
B_OUT <= B when ENABLE_B else 'Z';

process(A_OUT)
begin
   SIG <= A_OUT;
end process;

process(B_OUT)
```

```
begin
   SIG <= B_OUT;
end process;
```



Bus resolution functions assign the value for a multiply-driven signal. See ‘‘Resolution Functions," under ‘‘Subprograms" in Chapter 3, for more information.

block Statement

A `block` statement names a set of concurrent statements. Use blocks to organize concurrent statements hierarchically.

The syntax is

```
label: block
  { block_declarative_item }
begin
  { concurrent_statement }
end block [ label ];
```

The required *`label`* names the block.

A *`block_declarative_item`* declares objects local to the block and can be any of the following items:

- `use` clause
- Subprogram declaration
- Subprogram body
- Type declaration
- Subtype declaration
- Constant declaration
- Signal declaration
- Component declaration

The order of each *`concurrent_statement`* in a block is not significant, because each statement is always active.

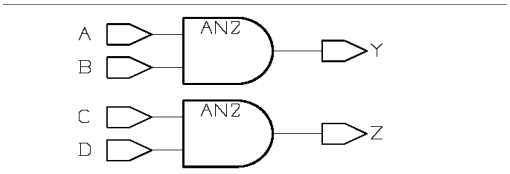
Note: FPGA Express does not support guarded blocks.

Objects declared in a block are visible to that block and to all blocks nested within. When a child block (inside a parent block) declares an object with the same name as an object in the parent block, the child's declaration overrides that of the parent (inside the child block).

Example 7-4 shows the use of nested blocks.

Example 7-4: Nested Blocks

```
B1: block
   signal S: BIT; -- Declaration of "S" in block B1
begin
   S <= A and B;  -- "S" from B1

   B2: block
      signal S: BIT; -- Declaration of "S" in block B2
   begin
      S <= C and D;  -- "S" from B2

      B3: block
      begin
         Z <= S;     -- "S" from B2
      end block B3;
   end block B2;

  Y <= S;         -- "S" from B1
end block B1;
```



Concurrent Procedure Calls

A concurrent procedure call is a procedure call used as a concurrent statement; it is used in an architecture or a block, rather than in a process. A concurrent procedure call is equivalent to a process containing a single sequential procedure call. The syntax is the same as that of a sequential procedure call:

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

The equivalent process is sensitive to all `in` and `inout` parameters of the procedure. Example 7-5 shows a procedure declaration, then a concurrent procedure call and its equivalent process.

Example 7-5: Concurrent Procedure Call and Equivalent Process

```
procedure ADD(signal A, B: in BIT;
              signal SUM: out BIT);
...
ADD(A, B, SUM);    -- Concurrent procedure call
...
process(A, B)      -- The equivalent process
begin
   ADD(A, B, SUM); -- Sequential procedure call
end process;
```

FPGA Express implements procedure and function calls with logic, unless you use the `map_to_entity` compiler directive (see ‘‘Mapping Subprograms to Components (Entities),'' in Chapter 6).

A common use for concurrent procedure calls is to obtain many copies of a procedure. For example, assume that a class of `BIT_VECTOR` signals must contain only one bit with value `1` and the rest of the bits value `0`. Suppose you have several signals of varying widths that you want monitored at the same time. One approach is to write a procedure to detect the error in a `BIT_VECTOR` signal, then make a concurrent call to that procedure for each signal.

Example 7-6 shows a procedure `CHECK` that determines whether a given bit vector contains exactly one element with value `'1'`; if this is not the case, `CHECK` sets its `out` parameter `ERROR` to `TRUE`.

Example 7-6: Procedure Definition for Example 7-7

```
procedure CHECK(signal A:      in BIT_VECTOR;
                signal ERROR: out Boolean) is

  variable FOUND_ONE: Boolean := FALSE;
                            -- Set TRUE when a '1'
                            -- is seen
begin
   for I in A'range loop    -- Loop across all bits
                            --   in the vector
      if A(I) = '1' then    -- Found a '1'
         if FOUND_ONE then  -- Have we already found one?
            ERROR <= TRUE;  -- Found two '1's
            return;         -- Terminate procedure
         end if;

         FOUND_ONE := TRUE; -- Note that we have
      end if;               --   seen a '1'
   end loop;

   ERROR <= not FOUND_ONE;  -- Error will be TRUE
                            --   if no '1' found
end;
```

Example 7-7 shows the CHECK procedure called concurrently for four different-sized bit vector signals.

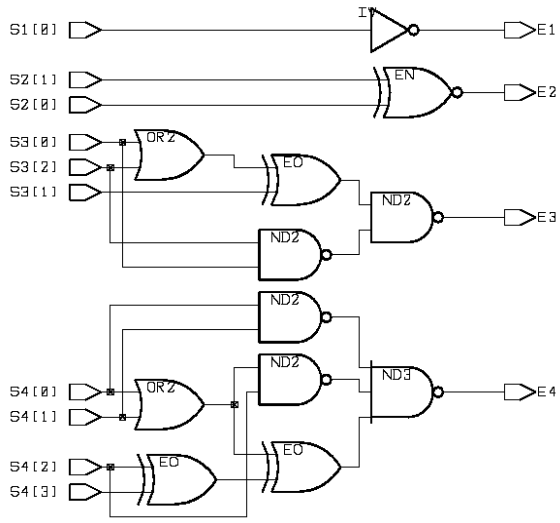
Example 7-7: Concurrent Procedure Calls

```
BLK: block
  signal S1: BIT_VECTOR(0 to 0);
  signal S2: BIT_VECTOR(0 to 1);
  signal S3: BIT_VECTOR(0 to 2);
  signal S4: BIT_VECTOR(0 to 3);


  signal E1, E2, E3, E4: Boolean;


begin
  CHECK(S1, E1);  -- Concurrent procedure call
  CHECK(S2, E2);
  CHECK(S3, E3);
  CHECK(S4, E4);
end block BLK;
```

Concurrent Signal Assignments

A concurrent signal assignment is equivalent to a process containing that sequential assignment. Thus, each concurrent signal assignment defines a new driver for the assigned signal. The simplest form of the concurrent signal assignment is

target **<=** *expression*;



target is a signal that receives the value of *expression.*

Example 7-8 shows the value of the expression `A and B` concurrently assigned to signal Z.

Example 7-8: Concurrent Signal Assignment

```
BLK: block
  signal A, B, Z: BIT;
begin
  Z <= A and B;
end block BLK;
```

The other two forms of concurrent signal assignment are conditional signal assignment and selected signal assignment.

Conditional Signal Assignment

Another form of concurrent signal assignment is the conditional signal assignment. The syntax is

```
target <= { expression when condition else }
          expression;
```

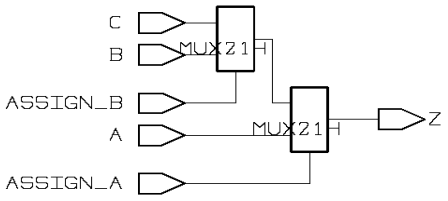
target is a signal that receives the value of an *expression.* The *expression* used is the first one whose Boolean *condition* is TRUE.

When a conditional signal assignment statement is executed, each *condition* is tested in order as written. The first *condition* that evaluates TRUE has its *expression* assigned to *target*. If no *condition* is TRUE, the final *expression* is assigned to the *target*. If two or more *conditions* are TRUE, only the first one is effective, just like the first TRUE branch of an `if` statement.

Example 7-9 shows a conditional signal assignment, where the target is the signal Z. The signal Z is assigned from one of the signals A, B, or C. The signal depends on the value of the expressions ASSIGN_A and ASSIGN_B. Note that the assignment of A takes precedence over that of B, and the assignment of B takes precedence over that of C, because the first TRUE condition controls the assignment.

Example 7-9: Conditional Signal Assignment

```
Z <= A when ASSIGN_A = '1' else
     B when ASSIGN_B = '1' else
     C;
```



Example 7-10 shows a process equivalent to the conditional signal assignment in Example 7-9.

Example 7-10: Process Equivalent to Conditional Signal Assignment

```
process(A, ASSIGN_A, B, ASSIGN_B, C)
begin
   if ASSIGN_A = '1' then
      Z <= A;
   elsif ASSIGN_B = '1' then
      Z <= B;
   else
      Z <= C;
   end if;
end process;
```

Selected Signal Assignment

The final kind of concurrent signal assignment is the selected signal assignment. The syntax is

```
with choice_expression select
   target <= { expression when choices, }
             expression when choices;
```

target is a signal that receives the value of an *expression.* The *expression* selected is the first one whose *choices* include the value of *choice_expression*. The syntax of *choices* is the same as that of the `case` statement:

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 the *choice_expression* type must be covered by one *choice*.

The final *choice* can be `others`, which matches all remaining (unchosen) values in the range of the *choice_expression* type. The `others` choice, if present, matches *choice_expression* only if none of the other choices match.

The `with..select` statement evaluates *choice_expression* and compares that value to each *choice* value. The `when` clause with the matching *choice* value has its *expression* assigned to *target*.

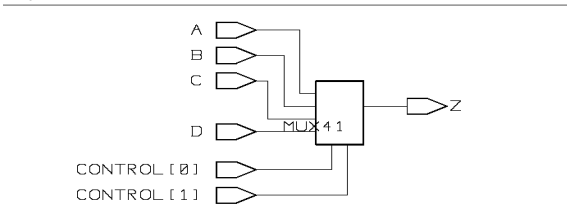
The following restrictions are placed on choices:

- No two choices can overlap.
- If no `others` choice is present, all possible values of *choice_expression* must be covered by the set of choices.

Example 7-11 shows target `Z` assigned from `A`, `B`, `C`, or `D`. The assignment depends on the current value of `CONTROL`.

Example 7-11: Selected Signal Assignment

```
signal A, B, C, D, Z: BIT;
signal CONTROL:  bit_vector(1 down to 0);
. . .
with CONTROL select
   Z <= A when "00",
        B when "01",
        C when "10",
        D when "11";
```



Example 7-12 shows the process equivalent to the selected signal assignment statement in Example 7-11.

Example 7-12: Process Equivalent to Selected Signal Assignment

```
process(CONTROL, A, B, C, D)
begin
   case CONTROL is
      when 0 =>
         Z <= A;
      when 1 =>
         Z <= B;
      when 2 =>
         Z <= C;
      when 3 =>
         Z <= D;
    end case;
end process;
```

Component Instantiations

A component instantiation references a previously defined hardware component, in the current design, at the current level of hierarchy. You can use component instantiations to define a design hierarchy. You can also use parts not defined in VHDL, such as components from an FPGA technology library, parts defined in the Verilog hardware description language, or the generic technology library. Component instantiation statements can be used to build netlists in VHDL.

A component instantiation statement indicates

- A name for this instance of the component.
- The name of a component to include in the current entity.
- The connection method for a component's ports.

The syntax is

```
instance_name : component_name port map (
                [ port_name => ] expression
                {, [ port_name => ] expression } );
```

instance_name names this instance of the component type *component_name*.

The port map connects each port of this instance of *component_name* to a signal-valued *expression* in the current entity. The value of *expression* can be a signal name, an indexed name, a slice name, or an aggregate. If *expression* is the VHDL reserved word open, the corresponding port is left unconnected.

You can map ports to signals by named or positional notation. You can include both named and positional connections in the port map, but you must place *all* positional connections before *any* named connections.

Note: For named association, the component port names must exactly match the declared component's port names. For positional association, the actual port expressions must be in the same order as the declared component's port order.

Example 7-13 shows a component declaration (a 2-input NAND gate) followed by three equivalent component instantiation statements.

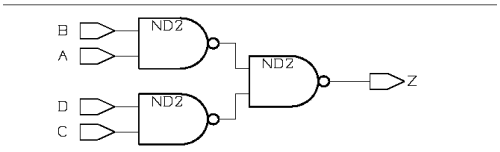
Example 7-13: Component Declaration and Instantiations

```
component ND2
   port(A, B: in BIT; C: out BIT);
end component;
. . .
signal X, Y, Z:  BIT;
. . .
U1: ND2 port map(X, Y, Z);                -- positional
U2: ND2 port map(A => X, C => Z, B => Y);-- named
U3: ND2 port map(X, Y, C => Z);           -- mixed
```

Example 7-14 shows the component instantiation statement defining a simple netlist. The three instances, U1, U2, and U3, are instantiations of the 2-input NAND gate component declared in Example 7-13.

Example 7-14: A Simple Netlist

```
signal TEMP_1, TEMP2: BIT;
. . .
  U1: ND2 port map(A, B, TEMP_1);
  U2: ND2 port map(C, D, TEMP_2);
  U3: ND2 port map(TEMP_1, TEMP_2, Z);
```



generate Statements

A `generate` statement creates zero or more copies of an enclosed set of concurrent statements. The two kinds of `generate` statements are

`for... generate`
the number of copies is determined by a discrete range

`if... generate`

zero or one copy is made, conditionally

for .. generate Statement

The syntax is

```
label: for identifier in range generate
     { concurrent_statement }
end generate [ label ] ;
```

The required *`label`* names this statement (useful for nested `generate` statements).

The use of the *`identifier`* in this construct is similar to that of the `for..loop` statement:

- *`identifier`* is not declared elsewhere. It is automatically declared by the `generate` statement itself and is entirely local to the loop. A loop identifier overrides any other identifier with the same name but only within the loop.
- The value *`identifier`* can be read only inside its loop, but you cannot assign a value to a loop identifier. In addition, the value of *`identifier`* cannot be assigned to any parameter whose mode is `out` or `inout`.

FPGA Express requires that *`range`* must be a *computable* integer range, in either of these forms:

`integer_expression` `to` *`integer_expression`*

`integer_expression` `downto` *`integer_expression`*

Each *`integer_expression`* evaluates to an integer.

Each *`concurrent_statement`* can be any of the statements described in this chapter, including other `generate` statements.

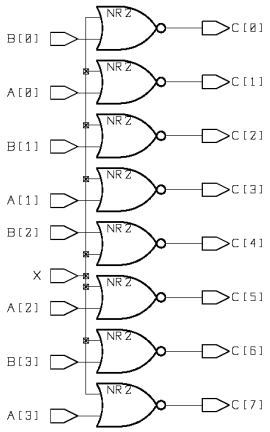
A `for..generate` statement executes as follows:

1. A new local integer variable is declared with the name *`identifier`*.
2. *`identifier`* is assigned the first value of *`range`*, and each concurrent statement is executed once.
3. *`identifier`* is assigned the next value in *`range`*, and each concurrent statement is executed once more.
4. Step 3 is repeated until *`identifier`* is assigned the last value in *`range`*. Each concurrent statement is then executed for the last time, and execution continues with the statement following `end generate`. The loop *`identifier`* is deleted.

Example 7-15 shows a code fragment that combines and interleaves two four-bit arrays A and B into an eight-bit array C.

Example 7-15: for..generate Statement

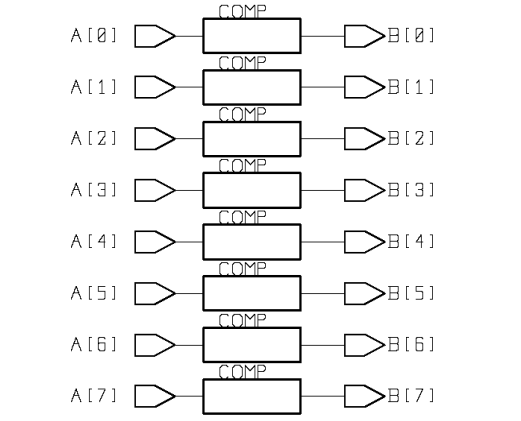
```
signal A, B : bit_vector(3 downto 0);
signal C    : bit_vector(7 downto 0);
signal X    : bit;
. . .
GEN_LABEL: for I in 3 downto 0 generate
  C(2*I + 1) <= A(I) nor X;
  C(2*I)     <= B(I) nor X;
end generate GEN_LABEL;
```



The most common usage of the `generate` statement is to create multiple copies of components, processes, or blocks. Example 7-16 demonstrates this usage with components. Example 7-17 shows how to generate multiple copies of processes.Example 7-16 shows VHDL array attribute `'range` used with the `for..generate` statement to instantiate a set of `COMP` components that connect corresponding elements of bit vectors A and B.

Example 7-16: for..generate Statement Operating on an Entire Array

```
component COMP
  port (X :  in bit;
        Y : out bit);
end component;
. . .
signal A, B: BIT_VECTOR(0 to 7);
. . .
GEN: for I in A'range generate
  U: COMP port map (X => A(I),
                    Y => B(I));
end generate GEN;
```



Unconstrained arrays and array attributes are described under ‘‘Array Types" in Chapter 4. Array attributes are shown in Example 4-9.

if . . generate Statement

The syntax is

```
label: if expression generate
     { concurrent_statement }
end generate [ label ] ;
```

label identifies (names) this statement. *expression* is any expression that evaluates to a Boolean value. A *concurrent_statement* is any of the statements described in this chapter, including other `generate` statements.

Note: Unlike the `if` statement described in Chapter 6, the `if..generate` statement has no `else` or `elsif` branches.

You can use the `if..generate` statement to generate a regular structure that has different circuitry at its ends. Use a `for..generate` statement to iterate over the desired width of a design, and a set of `if..generate` statements to define the beginning, middle, and ending sets of connections.

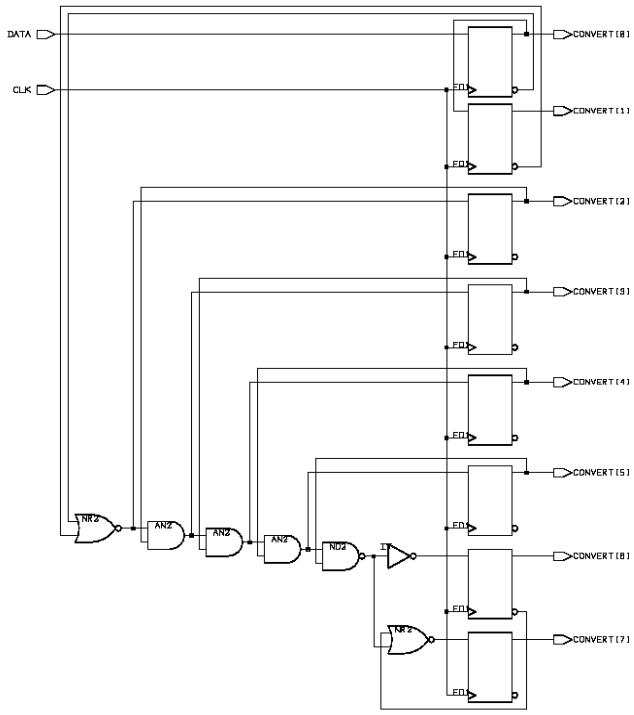
Example 7-17 shows a technology-independent description of the following *N*-bit serial-to-parallel converter. Data is clocked into an *N*-bit buffer from right to left. On each clock cycle, each bit in an *N*-bit buffer is shifted up one bit, and the incoming `DATA` bit is moved into the low-order bit.

Example 7-17 Typical Use of if..generate Statements

```
entity CONVERTER is
  generic(N: INTEGER := 8);

  port(CLK, DATA:   in BIT;
       CONVERT: buffer BIT_VECTOR(N-1 downto 0));
end CONVERTER;

architecture BEHAVIOR of CONVERTER is
  signal S : BIT_VECTOR(CONVERT'range);
begin

  G: for I in CONVERT'range generate

    G1: -- Shift (N-1) data bit into high-order bit
      if (I = CONVERT'left) generate
        process begin
          wait until (CLK'event and CLK = '1');
          CONVERT(I) <= S(I-1);
        end process;
    end generate G1;

    G2: -- Shift middle bits up
      if (I > CONVERT'right and
          I < CONVERT'left) generate

        S(I) <= S(I-1) and CONVERT(I);

        process begin
          wait until (CLK'event and CLK = '1');
          CONVERT(I) <= S(I-1);
        end process;
    end generate G2;

    G3:  -- Move DATA into low-order bit
      if (I = CONVERT'right) generate
        process begin
          wait until (CLK'event and CLK = '1');
```

```
          CONVERT(I) <= DATA;
        end process;
        S(I) <= CONVERT(I);
    end generate G3;

  end generate G;
end BEHAVIOR;
```

Example 7-17: (Continued) Typical Use of if..generate Statements



Chapter 8
Register and Three-State Inference

You can generally use several different, but logically equivalent, VHDL descriptions to describe a circuit.

To write VHDL descriptions to produce efficient synthesized circuits, consider the following topics:

- Register Inference
- Three-State Inference

You can use VHDL to make your design more efficient in terms of the synthesized circuit's area and speed, as follows:

- A design that needs some, but not all, of its variables or signals stored during operation can be written to minimize the number of latches or flip-flops required.
- A design that is described more easily with several levels of hierarchy can be synthesized more efficiently if part of the design hierarchy is collapsed during synthesis.

Register Inference

FPGA Express provides register inferencing using the `wait` and `if` statements.

A *register* is a simple, one-bit memory device, either a flip-flop or a latch. A flip-flop is an edge-triggered memory device. A latch is a level-sensitive memory device.

Use the `wait` statement to imply flip-flops in a synthesized circuit. FPGA Express creates flip-flops for all signals, and some variables assigned values in a process with a `wait` statement.

The `if` statement can be used to imply registers (flip-flops or latches) for signals and variables in the branches of the `if` statement.

To use register inferences, describe latches and flip-flops, and learn efficient use of registers, familiarize yourself with

- Using register inference
- Describing latches
- Describing flip-flops
- Efficient use of registers

Using Register Inference

Using register inference involves describing clock signals and using `wait` and `if` statements for register inferencing. Recommended models for different types of inferred registers and current Synopsys restrictions must also be considered.

Describing Clocked Signals

FPGA Express can infer asynchronous memory elements from VHDL descriptions written in a natural style.

Use the `wait` and `if` statements to test for the rising or falling edge of a signal. The most common usages are

```
process
begin
  wait until (edge);
  ...
end process;
...

process (sensitivity_list)
begin
  if (edge)
    ...
  end if;
end process;
```

Another form is

```
process (sensitivity_list)
begin
  if (...) then
    ...
  elsif (...)
    ...
  elsif (edge) then
    ...
  end if;
end process;
```

edge refers to an expression that tests for the positive or negative edge of a signal. The syntax of an *edge* expression is

```
SIGNAL'event      and SIGNAL = '1'  -- rising edge
NOT SIGNAL'stable and SIGNAL = '1'  -- rising edge

SIGNAL'event      and SIGNAL = '0'  -- falling edge
NOT SIGNAL'stable and SIGNAL = '0'  -- falling edge
```

In a `wait` statement, *edge* can also be

```
signal = '1'  -- rising edge
signal = '0'  -- falling edge
```

An *edge* expression must be the only condition of an `if` or an `elsif` statement. You can have only one *edge* expression in an `if` statement, and the `if` statement must not have an `else` clause. An *edge* expression cannot be part of another logical expression nor used as an argument.

```
if (edge and RST = '1')
  -- Illegal usage; edge must be only condition

Any_function(edge);
  -- Illegal usage; edge cannot be an argument

if X > 5 then
  sequential_statement;
elsif edge then
  sequential_statement;
else
  sequential_statement;
end if;
  -- Illegal usage; do not use edge as an intermediate expression.
```

These lines illustrate three incorrect uses of the *edge* expression. In the first group, the *edge* expression is part of a larger Boolean expression. In the second group, the *edge* expression is used as an argument. In the third group, the *edge* expression is used as an intermediate condition.

wait vs *if* Statements

Sometimes you can use the `wait` and `if` statements interchangeably. The `if` statement is usually preferred, because it provides greater control over the inferred register's capabilities, as described in the next section.

IEEE VHDL requires that a process with a `wait` statement must not have a sensitivity list.

An `if` *edge* statement can appear anywhere in a process. The sensitivity list of the process must contain all signals read in the process, including the *edge* signal. In general, the following guidelines apply:

- Synchronous processes (processes that compute values only on clock edges) must be sensitive to the clock signal.
- Asynchronous processes (processes that compute values on clock edges and when asynchronous conditions are TRUE) must be sensitive to the clock signal (if any), and to inputs that affect asynchronous behavior.

Recommended Use of Register Inference Capabilities

The register inference capability can support styles of description other than those described here. However, for best results:

- Restrict each process to a single type of memory-element inferencing: latch, latch with asynchronous set or reset, flip-flop, flip-flop with asynchronous reset, or flip-flop with synchronous reset.
- Use the following templates.

```
LATCH:  process(sensitivity_list)
          begin
            if LATCH_ENABLE then
                   ...
            end if;
          end process;


LATCH_ASYNC_SET:
                   ...
attribute async_set_reset of SET : signal is "true";
                   ...
        process(sensitivity_list)
          begin
            if SET then
               Q <= '1';
            elsif LATCH_ENABLE then
                   ...
            end if;
          end process;


FF:     process(CLK)
          begin
            if edge then
                ...
            end if;
          end process;
```

```
FF_ASYNC_RESET:
        process(RESET, CLK)
          begin
            if RESET then
               Q <= '0';
            elsif edge then
               Q <= ...;
            end if;
          end process;

FF_SYNC_RESET:
        process(RESET, CLK)
          begin
            if edge then
              if RESET then
                Q <= '0';
              else
                Q <= ...;
              end if;
            end if;
          end process;
```

Examples of these templates are provided in ‘‘Describing Latches" and ‘‘Describing Flip-Flops," later in this chapter.

Restrictions on Register Capabilities

Do not use more than one `if` *edge* expression in a process.

```
process(CLK_A, CLK_B)
begin
  if(CLK_A'event and CLK_A = '1') then
    A <= B;
  end if;

  if(CLK_B'event and CLK_B = '1') then  -- Illegal
    C <= B;
  end if;
end process;
```

Do not assign a value to a variable or signal on a FALSE branch of an `if` *edge* statement. This assignment is equivalent to checking for the *absence* of a clock edge, which has no hardware counterpart.

```
process(CLK)
begin
  if(CLK'event and CLK = '1') then
    SIG <= B;
  else
    SIG <= C;      -- Illegal
  end if;
end process;
```

If a variable is assigned a value inside an *edge* construct, do not read that variable later in the same process.

```
process(CLK)
 variable EDGE_VAR, ANY_VAR:  BIT;

begin
      if (CLK'event and CLK = '1') then
EDGE_SIGNAL <= X;
      EDGE_VAR    := Y;
      ANY_VAR     := EDGE_VAR; -- Legal
end if;

ANY_VAR := EDGE_VAR;        -- Illegal
end process;
```

Do not use an *edge* expression as an operand.

```
if not(CLK'event and CLK = '1') then  -- Illegal
```

Delays in Registers

If you use delay specifications with values that may be registered, the simulation to behave differently from the logic synthesized by FPGA Express. For example, the description in Example 8-1 contains delay information that causes FPGA Express to synthesize a circuit that behaves unexpectedly.

Example 8-1: Delays in Registers

```
component flip_flop (
    D, clock: in BIT;
    Q:        out BIT;);
end component;

process ( A, C, D, clock );
  signal B: BIT;
begin
      B <= A after 100ns;

      F1: flip_flop port map ( A, C, clock ),
      F2: flip_flop port map ( B, D, clock );
end process;
```

In Example 8-1, `B` changes 100 nanoseconds after `A` changes. If the clock period is fewer than 100 nanoseconds, output `D` is one or more clock cycles behind output `C` when the circuit is simulated. However, because FPGA Express ignores the delay information, `A` and `B` change values at the same time, and so do `C` and `D`. This behavior is *not* the same as in the simulated circuit.

When you use delay information in your designs, make sure the delays do not affect registered values. In general, you can safely include delay information in your description if it does not change the value that gets clocked into a flip-flop.

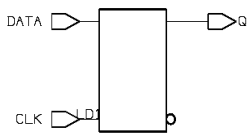
Describing Latches

FPGA Express infers latches from incompletely specified conditional expressions. In Example 8-2, the `if` statement infers a latch because there is no `else` clause:

Example 8-2: Latch Inference

```
process(GATE, DATA)
begin
  if (GATE = '1') then
    Q <= DATA;
  end if;
end process;
```

Figure 8-1: Latch Inference



The inferred latch uses `CLK` as its clock and `DATA` as its data input, as shown in Example 8-2.

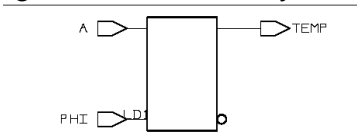
Automatic Latch Inferencing

A signal or variable that is not driven under all conditions becomes a latched value. As shown in Example 8-3, TEMP becomes a latched value because it is assigned only when PHI is 1.

Example 8-3: Automatically Inferred Latch

```
if(PHI = '1') then
  TEMP <= A;
end if;
```

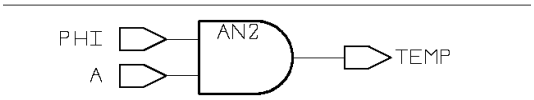
Figure 8-2: Automatically Inferred Latch



To avoid inferred latches, assign a value to the signal under all conditions, as shown in Example 8-4.

Example 8-4: Fully Specified Signal: No Latch Inference

```
if (PHI = '1') then
  TEMP <= A;
else
  TEMP <= '0';
end if;
```



Restrictions on Latch Inference Capabilities

You cannot read a conditionally assigned variable after the `if` statement in which it is assigned. A conditionally assigned variable is assigned a new value under some, but not all, conditions.

Therefore, a variable must always have a value before it is read.

```
signal X, Y: BIT;
. . .
process
  variable VALUE: BIT;
begin

  if (condition) then
    VALUE := X;
  end if;

  Y <= VALUE;  -- Illegal
end;
```

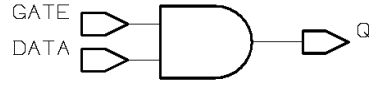
In simulation, latch inference occurs because signals and variables can hold state over time. A signal or variable holds its value until that value is reassigned. FPGA Express inserts a latch to duplicate this holding of state in hardware.

Variables declared locally within a subprogram do not hold their value over time. Every time a subprogram is used, its variables are reinitialized. Therefore, FPGA Express does not infer latches for variables declared in subprograms. In Example 8-5, no latches are inferred.

Example 8-5: Function without Inferred Latch

```
function MY_FUNC(DATA, GATE : BIT) return BIT is
     variable STATE: BIT;
begin
     if GATE then
          STATE := DATA;
     end if;
     return STATE;
end;
. . .
Q <= MY_FUNC(DATA, GATE);
```

Figure 8-3: Function without Inferred Latch



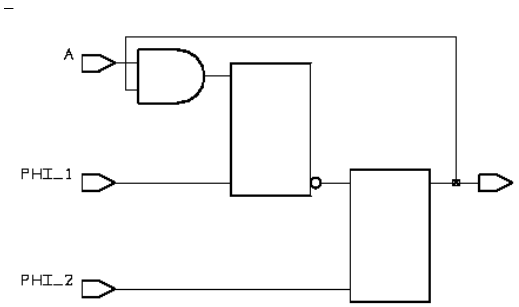
Example—Design with Two-Phase Clocks

By using the latch inference capability, you can describe network structures, such as two-phase systems in a technology-independent manner. Example 8-6 shows a simple two-phase system with clocks PHI_1 and PHI_2.

Example 8-6: Two-Phase Clocks

```
entity LATCH_VHDL is
  port(PHI_1, PHI_2, A : in BIT;
       t: out BIT);
end LATCH_VHDL;

architecture EXAMPLE of LATCH_VHDL is
  signal TEMP, LOOP_BACK: BIT;
begin
  process(PHI_1, A, LOOP_BACK)
  begin
    if(PHI_1 = '1') then
      TEMP <= A and LOOP_BACK;
    end if;
  end process;

  process(PHI_2, TEMP)
  begin
    if(PHI_2 = '1') then
      LOOP_BACK <= not TEMP;
    end if;
  end process;
  t <= LOOP_BACK;
end EXAMPLE;
```

Figure 8-4: Two-Phase Clocks



FPGA Express does not automatically infer dual-phase latches (devices with master and slave clocks). To use these devices, you must instantiate them as components, as described in Chapter 3.

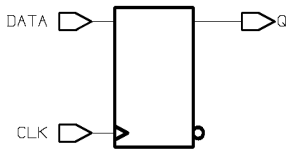
Describing Flip-Flops

Example 8-7 shows how an *edge* construct creates a flip-flop.

Example 8-7: Inferred Flip-Flop

```
process(CLK, DATA)
begin
  if (CLK'event and CLK = '1') then
    Q <= DATA;
  end if;
end process;
```

Figure 8-5: Inferred Flip-Flop

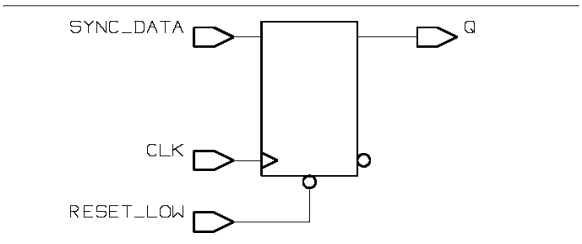


Flip-Flop with Asynchronous Reset

Example 8-8 shows how to specify a flip-flop with an asynchronous reset.

Example 8-8: Inferred Flip-Flop with Asynchronous Reset

```
process(RESET_LOW, CLK, SYNC_DATA)
begin
  if RESET_LOW = '0' then
    Q <= '0';
  elsif (CLK'event and CLK = '1') then
    Q <= SYNC_DATA;
  end if;
end process;
```



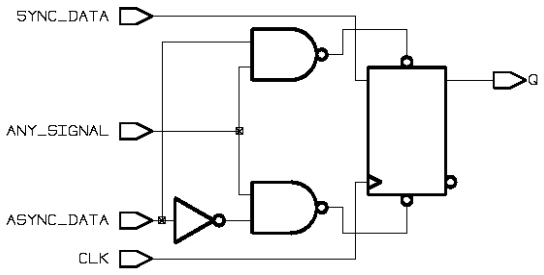
Note how the flip-flop in Example 8-8 is wired.

- The D input of the flip-flop is wired to SYNC_DATA.
- If the reset condition is computable (see "Computable Operands" in Chapter 5), either the SET or CLEAR pin of the flip-flop is wired to the RESET (or RESET_LOW) signal, as shown in Example 8–8.
- If the reset condition (ANY_SIGNAL in Example 8–9) is not computable, SET is wired to (ANY_SIGNAL AND ASYNC_DATA) and CLEAR is wired to (ANY_SIGNAL AND NOT(ASYNC_DATA)), as shown in Example 8–9.

Example 8-9 shows an inferred flip-flop with an asynchronous reset, where the reset condition is not computable.

Example 8-9: Inferred Flip-Flop with Asynchronous Set or Clear

```
process (CLK, ANY_SIGNAL, ASYNC_DATA, SYNC_DATA)
  begin
    if (ANY_SIGNAL) then
      Q <= ASYNC_DATA;
    elsif (CLK'event and CLK = '1') then
      Q <= SYNC_DATA;
    end if;
  end process;
```



Example—Synchronous Design with Asynchronous Reset

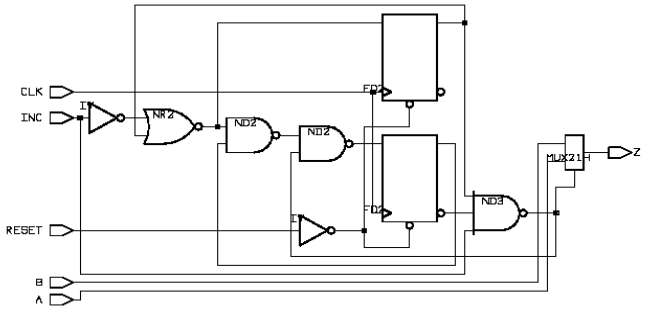
Example 8-10 describes a synchronous finite state machine (FSM) with an asynchronous reset.

Example 8-10: Synchronous Finite State Machine with Asynchronous Reset

```
package MY_TYPES is
  type STATE_TYPE is (S0, S1, S2, S3);
end MY_TYPES;

use WORK.MY_TYPES.ALL;

entity STATE_MACHINE is
  port(CLK, INC, A, B: in BIT; RESET: in Boolean;
       t: out BIT);
end STATE_MACHINE;

architecture EXAMPLE of STATE_MACHINE is
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
begin
  SYNC: process(CLK, RESET)
  begin
    if (RESET) then
      CURRENT_STATE <= S0;
    elsif (CLK'event and CLK = '1') then
      CURRENT_STATE <= NEXT_STATE;
    end if;
  end process SYNC;

  FSM: process(CURRENT_STATE, A, B)
  begin
    t <= A;            -- Default assignment
    NEXT_STATE <= S0;  -- Default assignment

    if (INC = '1') then
      case CURRENT_STATE is
        when S0 =>
          NEXT_STATE <= S1;
        when S1 =>
          NEXT_STATE <= S2;
          t <= B;
        when S2 =>
          NEXT_STATE <= S3;
        when S3 =>
          null;
      end case;
    end if;
  end process FSM;
```

```
end EXAMPLE;
```

Figure 8-6: Synchronous Finite State Machine with Asynchronous Reset



Attributes

New attributes used to assist register inference are discussed in this section. The attributes are defined in a VHDL library called Synopsys Attribute's package.

```
attribute async_set_reset : string;
attribute sync_set_reset : string;
attribute async_set_reset_local : string;
attribute sync_set_reset_local : string;
attribute async_set_reset_local_all : string;
attribute sync_set_reset_local_all : string;
attribute one_hot : string;
attribute one_cold : string;
```

async_set_reset

The `async_set_reset` attribute is attached to single-bit signals using the attribute construct. FPGA Express checks signals with the `async_set_reset` attribute set to `TRUE` to determine whether these signals asynchronously set or reset a latch in the entire design.

The syntax of `async_set_reset` is

```
attribute async_set_reset of signal_name,. : signal is "true";
```

Latch with Asynchronous Set or Clear Inputs

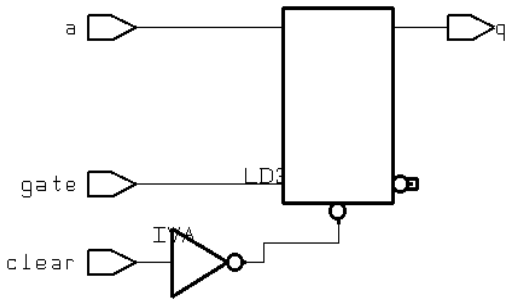
The asynchronous clear signal for a latch is inferred by driving the "Q" pin of your latch to `0`. The asynchronous set signal for a latch is inferred by driving the "Q" pin of your latch to `1`. Although FPGA Express does not require that the clear (set) be the first condition in your conditional branch, it is best to write your VHDL in this manner.

Example 8-11 shows how to specify a latch with an asynchronous clear input. To specify a latch with an asynchronous set, change the logic as indicated by the comments.

Example 8-11: Inferred Latch with Asynchronous Clear Input

```
attribute async_set_reset of clear : signal is "true";
process(clear, gate, a)
begin
  if (clear = '1') then
    q <= '0';
  elsif (gate = '1') then
    q <= a;
  end if;
end process;
```

Figure 8-7: Inferred Latch with Asynchronous Clear



sync_set_reset

The sync_set_reset attribute is attached to single-bit signals with the attribute constructs. FPGA Express checks signals with the sync_set_reset attribute set to TRUE to determine whether these signals synchronously set or reset a flip-flop in the entire design.

The syntax of sync_set_reset is

```
attribute sync_set_reset of signal_name,... : signal is "true";
```

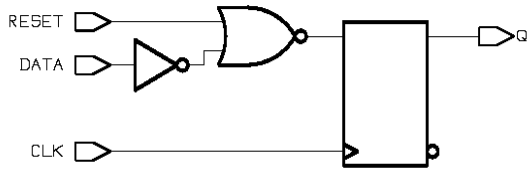
Flip-Flop with Synchronous Reset Input

Example 8-12 shows how to specify a flip-flop with a synchronous reset.

Example 8-12: Inferred Flip-Flop with Synchronous Reset Input

```
attribute sync_set_reset of RESET, SET : signal is "true";
process(RESET, CLK)
begin
  if (CLK'event and CLK = '1') then
    if RESET = '1' then
      Q <= '0';
```

```
    else
      Q <= DATA_A;
    end if;
  end if;
end process;

process (SET, CLK)
begin
  if (CLK'event and CLK = '1') then
    if SET = '1' then
      T <= '1';
    else
      T <= DATA_B;
    end if;
  end if;
end process;
```



async_set_reset_local

The `async_set_reset_local` attribute is attached to the label of a process with a value of a double-quoted list of single-bit signals. Every signal in the list is treated as though it has the `async_set_reset` attribute attached in the specified process.

The syntax of `async_set_reset_local` is

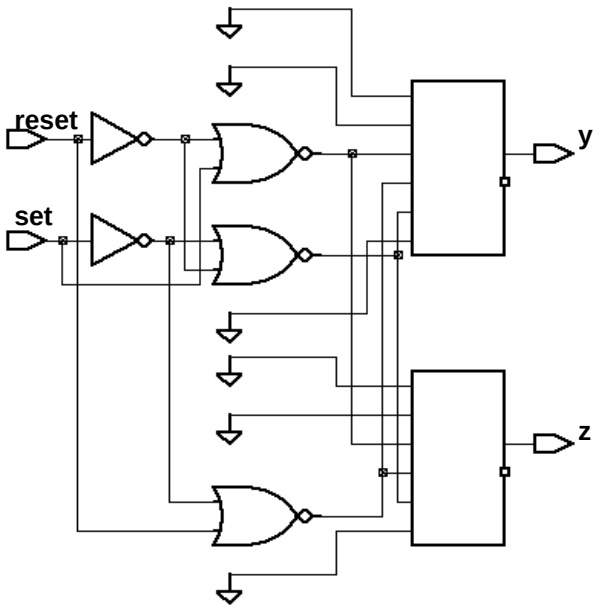
```
attribute async_set_reset_local of process_label : label is
                        "signal_name,...";
```

Example 8-13: Asynchronous Set/Reset on a Single Block

```
library IEEE;
library synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity e_async_set_reset_local is
port(reset, set, gate: in std_logic; y, t: out std_logic);
end e_async_set_reset_local;
```

```
architecture rtl of e_async_set_reset_local is
attribute async_set_reset_local of direct_set_reset : label
                    is "reset, set";
begin

  direct_set_reset: process (reset, set)
  begin
    if (reset = '1') then
      y <= '0';             -- asynchronous reset
    elsif (set = '1') then
      y <= '1';             -- asynchronous set
    end if;
  end process direct_set_reset;

  gated_data: process (gate, reset, set)
  begin
    if (gate = '1') then
      if (reset = '1') then
        t <= '0';             -- gated data
      elsif (set = '1') then
        t <= '1';             -- gated data
      end if;
    end if;
  end process gated_set_reset;

end rtl;
```

Figure 8-8: Asynchronous Set/Reset on a Single Block



sync_set_reset_local

The `sync_set_reset_local` attribute is attached to the label of a process with a value of a double-quoted list of single-bit signals. Every signal in the list is treated as though it has the `sync_set_reset` attribute attached in the specified process.

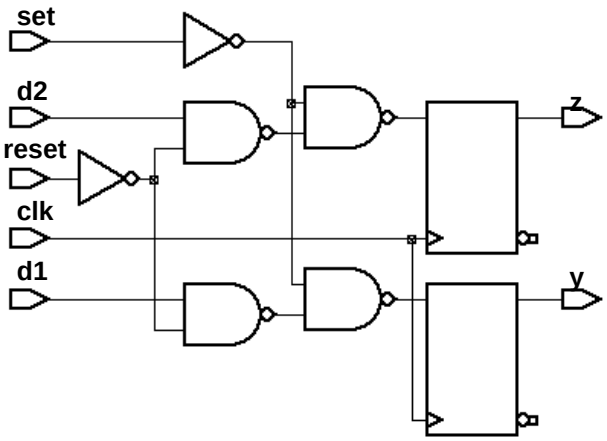
The syntax of `sync_set_reset_local` is

```
attribute sync_set_reset_local of process_label : label is
"signal_name,..."
```

Example 8-14: Synchronous Set/Reset on a Single Block

```
library IEEE;
library synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity e_sync_set_reset_local is
port(clk, reset, set, gate : in std_logic; y, t: out std_logic);
end e_sync_set_reset_local;

architecture rtl of e_sync_set_reset_local is
attribute sync_set_reset_local of clocked_set_reset : label is "reset,
set";
begin

  clocked_reset: process (clk, reset, set)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
        y <= '0';             -- synchronous reset
      else
        y <= '1';               -- synchronous set
      end if;
    end if;
  end process clocked_set_reset;

  gated_data: process (clk, gate, reset, set)
  begin
    if (clk'event and clk = '1') then
      if (gate = '1') then
        if (reset = '1') then
          t <= '0';             -- gated data
        elsif (set = '1') then
          t <= '1';             -- gated data
        end if;
      end if;
    end if;
  end process gated_set_reset;

end rtl;
```

Figure 8-9: Synchronous Set/Reset on a Single Block



async_set_reset_local_all

The `async_set_reset_local_all` attribute is attached to a process label. The attribute `async_set_reset_local_all` specifies that all the signals in the process are used to detect an asynchronous set or reset condition for inferred latches or flip-flops.

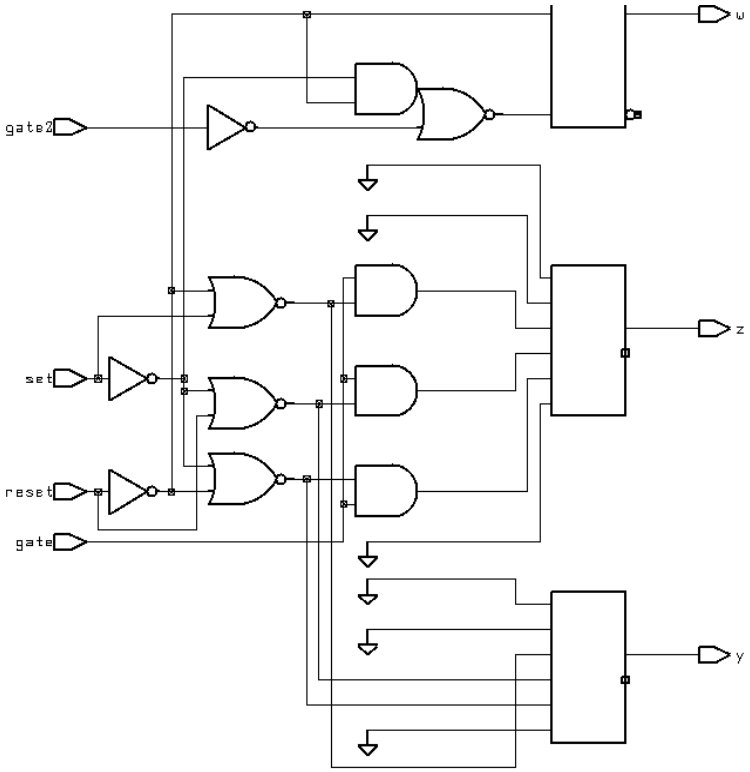
The syntax of `async_set_reset_local_all` is

```
attribute async_set_reset_local_all of process_label,... : label is "true";
```

Example 8-15: Asynchronous Set/Reset on Part of a Design

```
library IEEE;
library synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity e_async_set_reset_local_all is
port(reset, set, gate, gate2: in std_logic; y, t, w: out std_logic);
end e_async_set_reset_local_all;

architecture rtl of e_async_set_reset_local_all is
attribute async_set_reset_local_all of
          direct_set_reset, direct_set_reset_too: label is "true";
begin
  direct_set_reset: process (reset, set)
  begin
    if (reset = '1') then
      y <= '0';             -- asynchronous reset
    elsif (set = '1') then
      y <= '1';             -- asynchronous set
    end if;
  end process direct_set_reset;

  direct_set_reset_too: process (gate, reset, set)
  begin
    if (gate = '1') then
      if (reset = '1') then
        t <= '0';             -- asynchronous reset
      elsif (set = '1') then
        t <= '1';             -- asynchronous set
      end if;
    end if;
  end process direct_set_reset_too;

  gated_data: process (gate2, reset, set)
  begin
    if (gate = '1') then
      if (reset = '1') then
        w <= '0';             -- gated data
      elsif (set = '1') then
        w <= '1';             -- gated data
      end if;
    end if;
  end process gated_set_reset;

end rtl;
```

Figure 8-10: Asynchronous Set/Reset on Part of a Design



sync_set_reset_local_all

The `sync_set_reset_local_all` attribute is attached to a process label. The attribute `sync_set_reset_local_all` specifies that all the signals in the process are used to detect a synchronous set or reset condition for inferred latches or flip-flops.

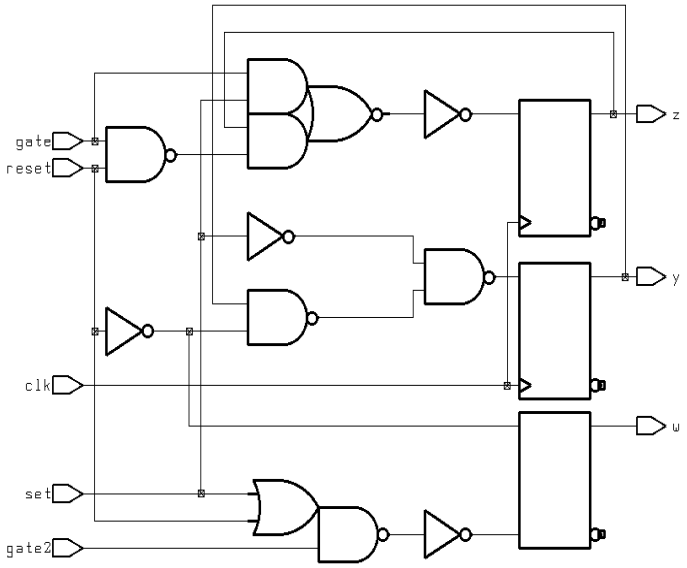
The syntax of `sync_set_reset_local_all` is

```
attribute sync_set_reset_local_all of process_label,... : label is "true";
```

Example 8-11: Example 8-16Synchronous Set/Reset on a Part of a Design

```
library IEEE;
library synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity e_sync_set_reset_local_all is
port(clk, reset, set, gate, gate2: in std_logic; y, t, w: out std_logic);
end e_sync_set_reset_local_all;

architecture rtl of e_sync_set_reset_local_all is
attribute sync_set_reset_local_all of
          clocked_set_reset, clocked_set_reset_too: label is "true";
begin

  clocked_set_reset: process (clk, reset, set)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
        y <= '0';             -- synchronous reset
      elsif (set = '1') then
        y <= '1';             -- synchronous set
      end if;
    end if;
  end process clocked_set_reset;

  clocked_set_reset_too: process (clk, gate, reset, set)
  begin
    if (clk'event and clk = '1') then
      if (gate = '1') then
        if (reset = '1') then
          t <= '0';             -- synchronous reset
        elsif (set = '1') then
          t <= '1';             -- synchronous set
        end if;
      end if;
    end if;
  end process clocked_set_reset_too;

  gated_data: process (clk, gate2, reset, set)
  begin
    if (clk'event and clk = '1') then
      if (gate = '1') then
        if (reset = '1') then
          w <= '0';             -- gated data
        elsif (set = '1') then
          w <= '1';             -- gated data
        end if;
      end if;
    end if;
  end process gated_set_reset;

end rtl;
```

Figure 8-11: Synchronous Set/Reset on a Part of a Design



Note: Use the `one_hot` and `one_cold` directives to implement D-type flip-flops with asynchronous set and reset signals. These two attributes tell FPGA Express that only one of the objects in the list are active at a time. If you are defining active high signals, use `one_hot`. For active low, use `one_cold`. Each attribute has two objects specified.

one_hot

The `one_hot` directive takes one argument of a double-quoted list of signals separated by commas. This attribute indicates that the group of signals are `one_hot`, in other words, at any time, no more than one signal can have a Logic 1 value. You must make sure that the group of signals are really `one_hot`. FPGA Express does not produce any logic to check this assertion.

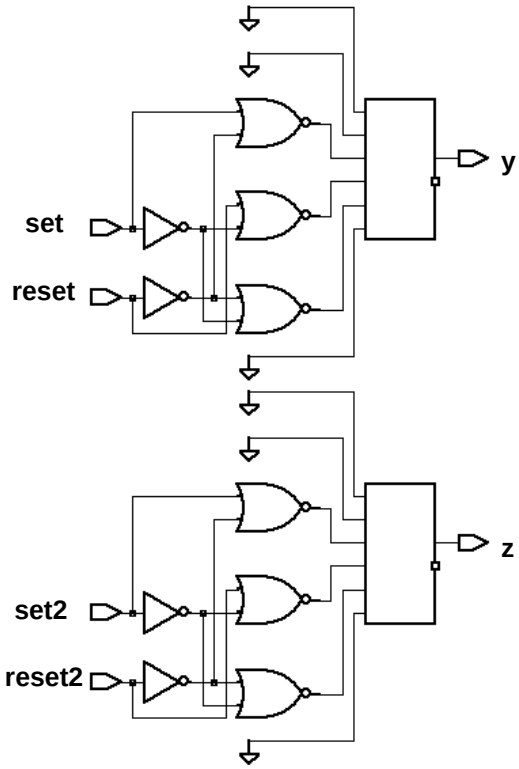
The syntax of `one_hot` is

```
attribute one_hot signal_name,... : label is "true";
```

Example 8-17: Using one_hot for Set and Reset

```
library IEEE;
library synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity e_one_hot is
port(reset, set, reset2, set2: in std_logic; y, t: out std_logic);
attribute async_set_reset of reset, set : signal is "true";
attribute async_set_reset of reset2, set2 : signal is "true";
attribute one_hot of reset, set : signal is "true";
end e_one_hot;

architecture rtl of e_one_hot is
begin
  direct_set_reset: process (reset, set )
  begin
    if (reset = '1') then
      y <= '0';             -- asynchronous reset by "reset"
    elsif (set = '1') then
      y <= '1';             -- asynchronous set by "set"
    end if;
  end process direct_set_reset;
  direct_set_reset_too: process (reset2, set2 )
  begin
    if (reset2 = '1') then
      t <= '0';             -- asynchronous reset by "reset2"
    elsif (set2 = '1') then
      t <= '1';             -- asynchronous set by "~reset2 set2"
    end if;
  end process direct_set_reset_too;

-- synopsys synthesis_off
process (reset, set)
begin
  assert not (reset='1' and set='1')
    report "One-hot violation"
    severity Error;
end process;
-- synopsys synthesis_on
end rtl;
```

Figure 8-12: Using *one_hot* for Set and Reset



one_cold

The one_cold directive is similar to the one_hot directive. one_cold indicates that no more than one signal in the group can have a Logic 0 value at any time.

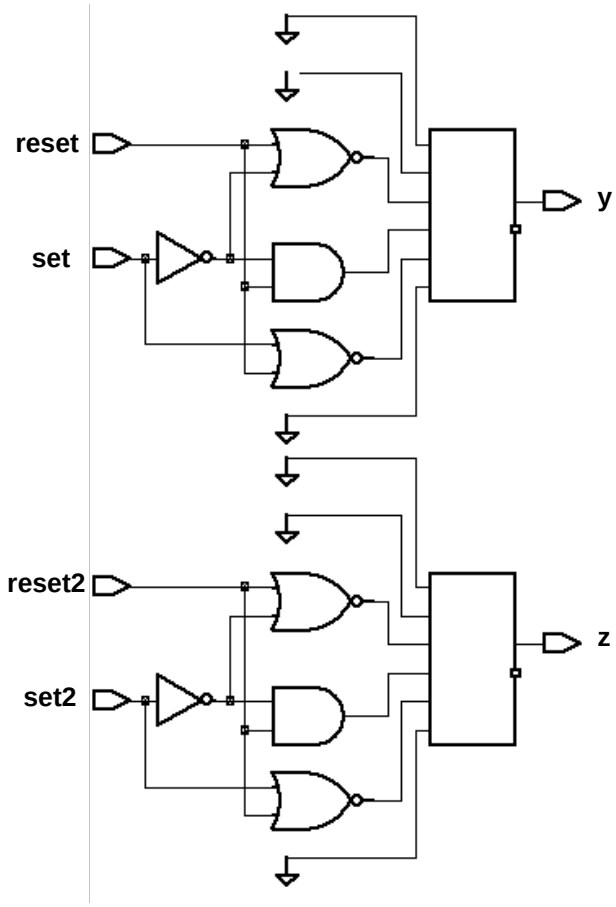
The syntax of one_cold is

```
attribute one_cold signal_name,... : label is "true";
```

Example 8-18 Using *one_cold* for Set and Reset

```
library IEEE;
library synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity e_one_cold is
port(reset, set, reset2, set2: in std_logic; y, t: out std_logic);
attribute async_set_reset of reset, set : signal is "true";
attribute async_set_reset of reset2, set2 : signal is "true";
attribute one_cold of reset, set : signal is "true";
end e_one_cold;

architecture rtl of e_one_cold is
begin

  direct_set_reset: process (reset, set )
  begin
    if (reset = '0') then
      y <= '0';             -- asynchronous reset by "not reset"
    elsif (set = '0') then
      y <= '1';             -- asynchronous set by "not set"
    end if;
  end process direct_set_reset;

  direct_set_reset_too: process (reset2, set2 )
  begin
    if (reset2 = '0') then
      t <= '0';             -- asynchronous reset by "not reset2"
    elsif (set2 = '0') then
      t <= '1';           -- asynchronous set by "(not reset2) (not set2)"
    end if;
  end process direct_set_reset_too;

-- synopsys synthesis_off
process (reset, set)
begin
  assert not (reset='0' and set='0')
    report "One-cold violation"
    severity Error;
end process;
-- synopsys synthesis_on

end rtl;
```

Figure 8-13: Using *one_cold* for Set and Reset



FPGA Express Latch and Flip-Flop Inference

FPGA Express inferes latches and flip-flops as follows:

- Asynchronous Flip-Flop Resets
  FPGA Express reports asynchronous set and reset conditions of flip-flops.

- Asynchronous Latch Resets
  FPGA Express interprets each control object of a latch as synchronous. If you want to asynchronously set or reset a latch, set this variable to TRUE.

- Flip-Flop Feedback Loops
  FPGA Express removes all flip-flop feedback loops. For example, feedback loops inferred from a statement such as Q=Q are removed. With the state feedback removed from a simple D flip-flop, it becomes a synchronous loaded flip-flop.

- Flip-Flop Inverted Feedback Loops
  FPGA Express removes all inverted flip-flop feedback loops. For example, feedback loops inferred from a statement such as Q=Q are removed and synthesized as T flip-flops.

- Reporting Inferred Modules
  FPGA Express generates a brief report on inferred latches, flip-flops, or three-state devices.

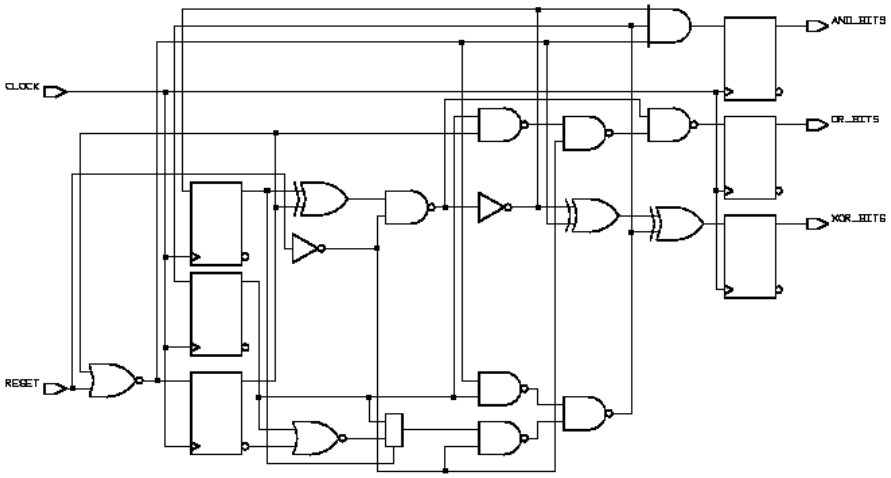
Efficient Use of Registers

Organize your HDL description so that you build only as many flip-flops as the design requires. Example 8-19 shows a description where too many flip-flops are implied.

Example 8-19: Circuit with Six Implied Registers

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity ex8_13 is
port ( clk , reset : in std_logic;
       and_bits , or_bits , xor_bits : out std_logic
);
end ex8_13;

architecture rtl of ex8_13 is
begin
process
variable count : std_logic_vector (2 downto 0);
begin
     wait until (clk'event and clk = '1');
     if (reset = '1') then
          count := "000";
     else count := count + 1;
     end if;
     and_bits <= count(2) and count(1) and count(0);
     or_bits <= count(2) or count(1) or count(0);
     xor_bits <= count(2) xor count(1) xor count(0);
end process;
end rtl;
```

Figure 8-14: Circuit with Six Implied Registers



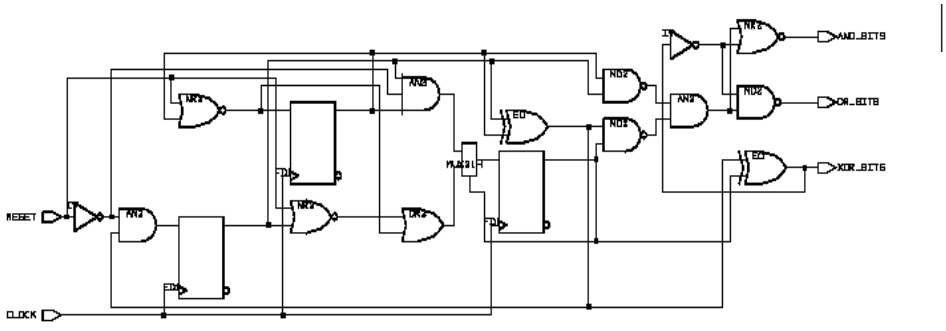
In Example 8-19, the outputs AND_BITS, OR_BITS, and XOR_BITS depend solely on the value of COUNT. Because COUNT is registered, the three outputs do not need to be registered. To avoid implying extra registers, assign the outputs from within a process that does not have a wait statement. Example 8-20 shows a description with two processes, one with a wait statement and one without. This description style lets you choose the signals that are registered and those that are not.

Example 8-20: Circuit with Three Implied Registers

```
use work.ARITHMETIC.all;
entity COUNT is
  port(CLOCK, RESET: in BIT;
       AND_BITS, OR_BITS, XOR_BITS : out BIT);
end COUNT;

architecture RTL of COUNT is
  signal COUNT : UNSIGNED (2 downto 0);
begin

  REG: process                    -- Registered logic
  begin
    wait until CLOCK'event and CLOCK = '1';
    if (RESET = '1') then
       COUNT <= "000";
    else
       COUNT <= COUNT + 1;
    end if;
  end process;
```

```
  COMBIN: process(COUNT)        -- Combinational logic
  begin
    AND_BITS <= COUNT(2) and COUNT(1) and COUNT(0);
    OR_BITS <= COUNT(2) or COUNT(1) or COUNT(0);
    XOR_BITS <= COUNT(2) xor COUNT(1) xor COUNT(0);
  end process;
end RTL;
```

Figure 8-15: Circuit with Three Implied Registers



This technique of separating combinational logic from registered or sequential logic is useful when describing finite state machines.

See the following examples in Appendix A:

- Moore machine
- Mealy machine
- Count zeros—sequential version
- Soft drink machine controller—state machine version

Example—Using Synchronous and Asynchronous Processes

You might want to keep some of the values computed by a process in flip-flops, while allowing other values to change between clock edges.

You can do this by splitting your algorithm between two processes, one with a `wait` statement and one without. Put the registered (synchronous) assignments into the `wait` process. Put the other (asynchronous) assignments into the other process. Use signals to communicate between the two processes.

For example, suppose you want to build a design with the following characteristics:

- Inputs `A_1`, `A_2`, `A_3` and `A_4` change asynchronously.
- Output `t` is driven from one of `A_1`, `A_2`, `A_3`, or `A_4`.
- Input `CONTROL` is valid only on the positive edge of `CLOCK`. The value at the edge determines which of the four inputs is selected during the next clock cycle.

- Output t must always reflect changes in the value of the currently selected signal.

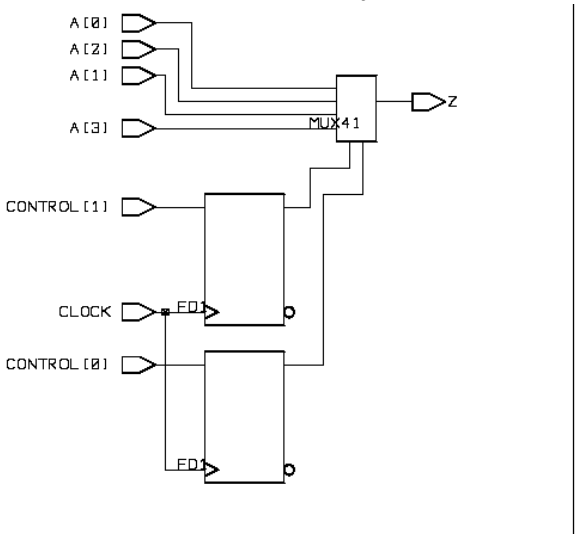
The implementation of this design requires two processes. The process with a wait statement synchronizes the CONTROL value. The other process multiplexes the output, based on the synchronized control. The signal SYNC_CONTROL communicates between the two processes.

Example 8-21 shows the code and a schematic of one possible implementation.

Example 8-21: Two Processes: One Synchronous, One Asynchronous

```
entity SYNC_ASYNC is
   port (CLOCK:   in BIT;
         CONTROL: in INTEGER range 0 to 3;
         A:       in BIT_VECTOR(0 to 3);
         t:      out BIT);
end SYNC_ASYNC;

architecture EXAMPLE of SYNC_ASYNC is
  signal SYNC_CONTROL: INTEGER range 0 to 3;
begin

  process
  begin
    wait until CLOCK'event and CLOCK = '1';
    SYNC_CONTROL <= CONTROL;
  end process;

  process (A, SYNC_CONTROL)
  begin
    t <= A(SYNC_CONTROL);
  end process;
end EXAMPLE;
```

Figure 8-16: Two Processes: One Synchronous, One Asynchronous



Three-State Inference

FPGA Express can infer three-state gates (high-impedance output) from enumeration encoding in VHDL. After inferrence, FPGA Express maps the gates to a specified technology library. See "Enumeration Encoding" in Chapter 4 for more information.

When a variable is assigned the value of `'Z'`, the output of the three-state gate is disabled. Example 8-22 shows the VHDL for a three-state gate

Example 8-22: Creating a Three-State Gate in VHDL

```
signal OUT_VAL, IN_VAL: std_logic;
...
if (COND) then
    OUT_VAL <= IN_VAL;
else
    OUT_VAL <= 'Z';     -- assigns high-impedance
end if;
```

You can assign a high impedance value to a four-bit wide bus with `"ZZZZ"`.

One three-state device is inferred from a single process. Example 8-23 infers only one three-state device.

Example 8-23: Inferring One Three-State Device from a Single Process

```
process (sela, a, selb, b) begin
  t <= 'z';
    if (sela = '1') then
      t <= a;
    if (selb = '1') then
      t <= b;
end process;
```

Example 8-24 infers two three-state devices.

Example 8-24: Inferring Two Three-State Devices

```
process (sela, a) begin
    if (sela = '1') then
      t = a;
    else t = 'z';
end process;

process (selb, b) begin
    if (selb = '1') then
      t = b;
    else t = 'z';
end process;
```

The VHDL conditional assignment may also be used for three-state inferencing.

Assigning the Value *Z*

Assigning variables the value Z is allowed. The value Z can also appear in function calls, return statements, and aggregates. However, except for comparisons to Z, you cannot use Z in an expression. Example 8-25 shows an incorrect use of Z (in an expression), and Example 8-26 shows a correct use of Z (in a comparison).

Example 8-25: Incorrect Use of the Value Z in an Expression

```
OUT_VAL <= 'Z' and IN_VAL;
...
```

Example 8-26: Correct Expression Comparing to Z

```
if IN_VAL = 'Z' then
...
```

Caution ***Expressions comparing to Z are synthesized as though values are not equal to Z.***

For example:

```
if X = 'Z' then
...
```

is synthesized as:

```
if FALSE then
...
```

If you use expressions comparing values to `'Z',` the presynthesis and postsynthesis simulation results might differ. For this reason, FPGA Express issues a warning when it synthesizes such comparisons.

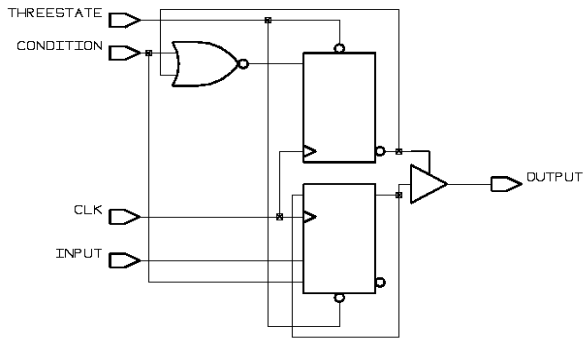
Latched Three-State Variables

When a variable is latched (or registered) in the same process in which it is three-stated, the enable of the three-state Z is also latched (or registered). This process is shown in Example 8-27.

Example 8-27: Three-State Inferred with Registered Enable

```
-- Creates a flip-flop on input and on enable
if (THREESTATE = '0') then
    OUTPUT <= 'Z';
elsif (CLK'event and CLK = '1') then
    if (CONDITION) then
        OUTPUT <= INPUT;
    end if;
end if;
```

Figure 8-17: Three-State Inferred with Registered Enable

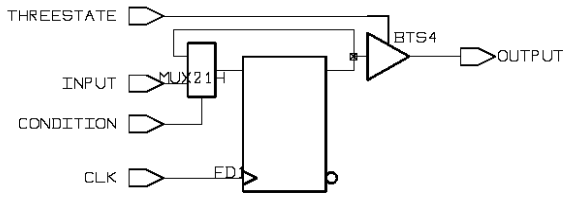


In Example 8-27, the three-state gate has a registered enable signal. Example 8-28 uses two processes to instantiate a three-state with a flip-flop only on the input.

Example 8-28: Example 8-28Latched Three-State with Flip-flop on Input

```
entity LATCH_3S is
  port(CLK, THREESTATE, INPUT: in std_logic;
       OUTPUT: out std_logic; CONDITION: in Boolean);
end LATCH_3S;

architecture EXAMPLE of LATCH_3S is
  signal TEMP: std_logic;
begin

  process(CLK, CONDITION, INPUT)
  begin      -- creates three-state
    if (CLK'event and CLK = '1') then
      if (CONDITION) then
          TEMP <= INPUT;
      end if;
    end if;
  end process;
  process(THREESTATE, TEMP)
  begin
    if (THREESTATE = '0') then
        OUTPUT <= 'Z';
    else
        OUTPUT <= TEMP;
    end if;
  end process;
end EXAMPLE;
```

Figure 8-18: Latched Three-State with Flip-Flop on Input



Chapter 9
FPGA Express Directives

Synopsys has defined several methods of providing circuit design information directly in your VHDL source code.

- Using FPGA Express directives, you can direct the translation from VHDL to components with special VHDL comments. These synthetic comments turn translation on or off, specify one of several hard-wired resolution methods, and provide a means to map subprograms to hardware components.
- Using Synopsys-defined VHDL attributes, you can add synthesis-related signal and constraint information to ports, components, and entities. This information is used by FPGA Express during synthesis.

To familiarize yourself with FPGA Express directives, consider the following topics:

- Notation for FPGA Express Directives
- FPGA Express Directives
- Synthesis Attributes and Constraints

Notation for FPGA Express Directives

FPGA Express directives are special VHDL comments (*synthetic comments*) that affect the actions of FPGA Express. These comments are just a special case of regular VHDL comments, so they are ignored by other VHDL tools. Synthetic comments are used only to direct the actions of FPGA Express.

Synthetic comments begin with two hyphens (`--`), just like a regular comment. If the word following these characters is `pragma` or `synopsys`, the remaining comment text is interpreted by FPGA Express as a directive.

Note: FPGA Express displays a syntax error if an unrecognized directive is encountered after `-- synopsys` or `-- pragma`.

FPGA Express Directives

The three types of directives are

- Translation stop and start Directives

```
-- pragma translate_off
-- pragma translate_on
-- pragma synthesis_off
```

```
-- pragma synthesis_on
```

- Resolution function directives

```
-- pragma resolution_method wired_and
-- pragma resolution_method wired_or
-- pragma resolution_method three_state
```

- Component implication directives

```
-- pragma map_to_entity entity_name
-- pragma return_port_name port_name
```

Other directives such as the `map_to` operator are used to drive inference of HDL operators such as `*`, `+`, and `-`.

Translation Stop and Start Directives

Translation directives stop and start the translation of a VHDL source file by FPGA Express.

```
-- pragma translate_off
-- pragma translate_on
```

The `translate_off` and `translate_on` directives instruct FPGA Express to stop and start synthesizing VHDL source code. The VHDL code between these two directives is, however, checked for syntax.

Translation is enabled at the beginning of each VHDL source file. You can use `translate_off` and `translate_on` directives anywhere in the text.

The `synthesis_off` and `synthesis_on` directives are the recommended mechanisms for hiding simulation-only constructs from synthesis. Any text between these directives is checked for syntax, but no corresponding hardware is synthesized. The behavior of the `synthesis_off` and `synthesis_on` directives is not affected by the variable `hdlin_translate_off_skip_text`.

Example 9-1 shows how you can use the directives to protect a simulation driver.

Example 9-1: Using synthesis_on and synthesis_off Directives

```
-- The following test driver for entity EXAMPLE
--   should not be translated:
--
-- pragma synthesis_off
--   Translation stops

entity DRIVER is
end;

architecture VHDL of DRIVER is
    signal A, B : INTEGER range 0 to 255;
    signal SUM  : INTEGER range 0 to 511;

    component EXAMPLE
        port (A, B: in INTEGER range 0 to 255;
              SUM: out INTEGER range 0 to 511);
    end component;

begin
    U1: EXAMPLE port map(A, B, SUM);
    process
    begin
        for I in 0 to 255 loop
            for J in 0 to 255 loop
                A <= I;
                B <= J;
                wait for 10 ns;
                assert SUM = A + B;
            end loop;
        end loop;
    end process;
end;

-- pragma synthesis_on
--   Code from here on is translated
entity EXAMPLE is
    port (A, B: in INTEGER range 0 to 255;
          SUM: out INTEGER range 0 to 511);
end;

architecture VHDL of EXAMPLE is
begin
    SUM <= A + B;
end;
```

Resolution Function Directives

Resolution function directives determine the resolution function associated with resolved signals (see ‘‘Signal Declarations“ in Chapter 3). FPGA Express does not currently support arbitrary resolution functions. It does support the following three methods:

```
-- pragma resolution_method wired_and
-- pragma resolution_method wired_or
-- pragma resolution_method three_state
```

Note: Do not connect signals that use different resolution functions. FPGA Express supports only one resolution function per network.

Component Implication Directives

Component implication directives map VHDL subprograms onto existing components or VHDL entities. These directives are described under ‘‘Mapping Subprograms to Components“ in Chapter 6:

```
-- pragma map_to_entity entity_name
-- pragma return_port_name port_name
```

Chapter 10
Synopsys Packages

Three Synopsys packages are included with this release:

- *std_logic_1164* Package

  Defines a standard for designers to use when describing the interconnection data types used in VHDL modeling.

- *std_logic_arith* Package

  Provides a set of arithmetic, conversion, and comparison functions for SIGNED, UNSIGNED, INTEGER, STD_ULOGIC, STD_LOGIC, and STD_LOGIC_VECTOR types.

- *std_logic_misc* Package

  Defines supplemental types, subtypes, constants, and functions for the std_logic_1164 package.

  To understand the contents of each package, review the following sections.

std_logic_1164 Package

This package defines the IEEE standard for designers to use when describing the interconnection data types used in VHDL modeling. The logic system defined in this package might be insufficient for modeling switched transistors, because such a requirement is out of the scope of this effort. Furthermore, mathematics, primitives, and timing standards are considered orthogonal issues as they relate to this package and are therefore beyond the scope of this effort.

The `std_logic_1164` package contains Synopsys synthesis directives. Three functions, however, are not currently supported for synthesis: `rising_edge`, `falling_edge`, and `is_x`.

To use this package in a VHDL source file, include the following lines at the top of the source file:

```
library IEEE;
use IEEE.std_logic_1164.all;
```

When you analyze your VHDL source file, FPGA Express automatically finds the IEEE library and the `std_logic_1164` package. However, you must analyze the `use` packages not contained in the IEEE and Synopsys libraries before processing a source file that uses them.

std_logic_arith Package

Functions defined in the `std_logic_arith` package provide conversion to and from the predefined VHDL data type `INTEGER`, and arithmetic, comparison, and Boolean operations. This package lets you perform arithmetic operations and numeric comparisons on array data types. The package defines some arithmetic operators (+, -, *, and `abs`) and the relational operators (<, >, <=, >=, =, and /=). Note that IEEE VHDL does not define arithmetic operators for arrays and defines the comparison operators in a manner inconsistent with an arithmetic interpretation of array values.

The package also defines two major data types of its own: `UNSIGNED` and `SIGNED`. Details can be found in ‘‘Synopsys Data Types" later in this appendix. The `std_logic_arith` package is legal VHDL; you can use it for both synthesis and simulation.

The `std_logic_arith` package can be configured to work on any array of single-bit types. You encode single-bit types in one bit with the `ENUM_ENCODING` attribute.

You can make the vector type (for example, `std_logic_vector`) synonymous with either `SIGNED` or `UNSIGNED`. This way, if you plan to use mostly `UNSIGNED` numbers, you do not need to convert your vector type to call `UNSIGNED` functions. The disadvantage of making your vector type synonymous with either `UNSIGNED` or `SIGNED` is that it causes the standard VHDL comparison functions (=, /=, <, >, <=, and >=) to be redefined.

Table 10-1 shows that the standard comparison functions for `BIT_VECTOR` do not match the `SIGNED` and `UNSIGNED` functions.

Table 10-1: UNSIGNED, SIGNED and BIT_VECTOR Comparison Functions

ARG1	op	ARG2	UNSIGNED	SIGNED	BIT_VECTOR
"000"	=	"000"	TRUE	TRUE	TRUE
"00"	=	"000"	TRUE	TRUE	FALSE
"100"	=	"0100"	TRUE	FALSE	FALSE
"000"	<	"000"	FALSE	FALSE	FALSE
"00"	<	"000"	FALSE	FALSE	TRUE
"100"	<	"0100"	FALSE	TRUE	FALSE

Using the Package

The `std_logic_arith` package is in the `$synopsys/packages/IEEE/src/std_logic_arith.vhd` subdirectory of the Synopsys root directory. To use this package in a VHDL source file, include the following lines at the top of the source file:

```
library IEEE;
use IEEE.std_logic_arith.all;
```

Synopsys packages are preanalyzed and do not require further analyzing.

Modifying the Package

The std_logic_arith package is written in standard VHDL. You can modify or add to it. The appropriate hardware is then synthesized.

For example, to convert a vector of multivalued logic to an INTEGER, you can write the function shown in Example 10-1. This MVL_TO_INTEGER function returns the integer value corresponding to the vector when the vector is interpreted as an unsigned (natural) number. If unknown values are in the vector, the return value is -1.

Example 10-1: New Function Based on a std_logic_arith Package Function

```
library IEEE;
use IEEE.std_logic_1164.all;

function MVL_TO_INTEGER(ARG : MVL_VECTOR)
  return INTEGER is
  -- pragma built_in SYN_FEED_THRU
  variable uns: UNSIGNED (ARG'range);
begin
    for i in ARG'range loop
        case ARG(i) is
            when '0' | 'L' => uns(i) := '0';
            when '1' | 'H' => uns(i) := '1';
            when others    => return -1;
        end case;
    end loop;
    return CONV_INTEGER(uns);
end;
```

Note the use of the CONV_INTEGER function in Example 10-1.

FPGA Express performs almost all synthesis directly from the VHDL descriptions. However, several functions are hard wired for efficiency. These functions can be identified by the following comment in their declarations

```
-- pragma built_in
```

This statement marks functions as special, causing the body to be ignored. Modifying the body does *not* change the synthesized logic unless you remove the built_in comment. If you want new functionality, use the built_in functions; this is more efficient than removing the built_in and modifying the body.

Data Types

The std_logic_arith package defines two data types, UNSIGNED and SIGNED:

```
type UNSIGNED is array (natural range <>) of std_logic;
type SIGNED is array (natural range <>) of std_logic;
```

These data types are similar to the predefined VHDL type BIT_VECTOR, but the std_logic_arith package defines the interpretation of variables and signals of these types as numeric values. With the install_vhdl conversion script, you can change these data types to arrays of other one-bit types.

UNSIGNED

The UNSIGNED data type represents an unsigned numeric value. FPGA Express interprets the number as a binary representation, with the farthest left bit being most significant. For example, the decimal number 8 can be represented as

```
UNSIGNED'("1000")
```

When you declare variables or signals of type UNSIGNED, a larger vector holds a larger number. A four-bit variable holds values up to decimal 15; an eight-bit variable holds values up to 255, and so on. By definition, negative numbers cannot be represented in an UNSIGNED variable. Zero is the smallest value that can be represented.

Example 10-2 illustrates some UNSIGNED declarations. Note that the most significant bit is the farthest left array bound, rather than the high or low range value.

Example 10-2: UNSIGNED Declarations

```
variable VAR: UNSIGNED (1 to 10);
  -- 11-bit number
  -- VAR(VAR'left) = VAR(1) is the most significant bit

signal SIG: UNSIGNED (5 downto 0);
  -- 6-bit number
  -- SIG(SIG'left) = SIG(5) is the most significant bit
```

SIGNED

The SIGNED data type represents a signed numeric value. FPGA Express interprets the number as a 2's complement binary representation, with the farthest left bit as the sign bit. For example, you can represent decimal 5 and -5 as

```
SIGNED'("0101")  -- represents +5
SIGNED'("1011")  -- represents -5
```

When you declare SIGNED variables or signals, a larger vector holds a larger number. A four-bit variable holds values from -8 to 7; an eight-bit variable holds values from –128 to 127. Note that a SIGNED value cannot hold as large a value as an UNSIGNED value with the same bit width.

Example 10-3 shows some SIGNED declarations. Note that the sign bit is the farthest left bit, rather than the highest or lowest.

Example 10-3: SIGNED Declarations

```
variable S_VAR: SIGNED (1 to 10);
  -- 11-bit number
  -- S_VAR(S_VAR'left) = S_VAR(1) is the sign bit

signal S_SIG: SIGNED (5 downto 0);
  -- 6-bit number
  -- S_SIG(S_SIG'left) = S_SIG(5) is the sign bit
```

Conversion Functions

The std_logic_arith package provides three sets of functions to convert values between its UNSIGNED and SIGNED types, and the predefined type INTEGER. This package also provides the std_logic_vector.

Example 10-4 shows the declarations of these conversion functions. BIT and BIT_VECTOR types are shown.

Example 10-4: Conversion Functions

```
subtype SMALL_INT is INTEGER range 0 to 1;

function CONV_INTEGER(ARG: INTEGER)  return INTEGER;
function CONV_INTEGER(ARG: UNSIGNED) return INTEGER;
function CONV_INTEGER(ARG: SIGNED)   return INTEGER;
function CONV_INTEGER(ARG: STD_ULOGIC) return SMALL_INT;

function CONV_UNSIGNED(ARG: INTEGER;
                       SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: UNSIGNED;
                       SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: SIGNED;
                       SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: STD_ULOGIC;
                       SIZE: INTEGER) return UNSIGNED;

function CONV_SIGNED(ARG: INTEGER;
                     SIZE: INTEGER)   return SIGNED;
function CONV_SIGNED(ARG: UNSIGNED;
                     SIZE: INTEGER)   return SIGNED;
function CONV_SIGNED(ARG: SIGNED;
```

```
                    SIZE: INTEGER)   return SIGNED;
function CONV_SIGNED(ARG: STD_ULOGIC;
                    SIZE: INTEGER)   return SIGNED;

function CONV_STD_LOGIC_VECTOR(ARG: INTEGER;
                    SIZE: INTEGER)   return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: UNSIGNED;
                    SIZE: INTEGER)   return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: SIGNED;
                    SIZE: INTEGER)   return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: STD_ULOGIC;
                    SIZE: INTEGER)   return STD_LOGIC_VECTOR;
```

Note that there are four versions of each conversion function.

The operator overloading mechanism of VHDL determines the correct version from the function call's argument types.

The CONV_INTEGER functions convert an argument of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an INTEGER return value. The CONV_UNSIGNED and CONV_SIGNED functions convert an argument of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an UNSIGNED or SIGNED return value whose bit width is SIZE.

The CONV_INTEGER functions have a limitation on the size of operands. VHDL defines INTEGER values as between -2147483647 and 2147483647. This range corresponds to a 31-bit UNSIGNED value or a 32-bit SIGNED value. You cannot convert an argument outside this range to an INTEGER.

The CONV_UNSIGNED and CONV_SIGNED functions require two operands. The first operand is the value converted. The second operand is an INTEGER that specifies the expected size of the converted result. For example, the following function call returns a 10-bit UNSIGNED value representing the value in sig.

```
ten_unsigned_bits := CONV_UNSIGNED(sig, 10);
```

If the value passed to CONV_UNSIGNED or CONV_SIGNED is smaller than the expected bit width (such as representing the value 2 in a 24-bit number), the value is bit-extended appropriately. FPGA Express places zeros in the more significant (left) bits for an UNSIGNED return value and uses sign extension for a SIGNED return value.

You can use the conversion functions to extend a number's bit width even if conversion is not required. For example:

```
CONV_SIGNED(SIGNED'("110"), 8) % "11111110"
```

An UNSIGNED or SIGNED return value is truncated when its bit width is too small to hold the ARG value. For example:

```
CONV_SIGNED(UNSIGNED'("1101010"), 3) % "010"
```

Arithmetic Functions

The `std_logic_arith` package provides arithmetic functions for use with combinations of Synopsys' `UNSIGNED` and `SIGNED` data types and the predefined types `STD_ULOGIC` and `INTEGER`. These functions produce adders and subtracters.

There are two sets of arithmetic functions: binary functions with two arguments, such as `A+B` or `A*B`, and unary functions with one argument, such as `-A`. The declarations for these functions are shown in Examples 10-5 and 10-6.

Example 10-5: Binary Arithmetic Functions

```
function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED;   R: SIGNED)   return SIGNED;
function "+"(L: UNSIGNED; R: SIGNED)   return SIGNED;
function "+"(L: SIGNED;   R: UNSIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: INTEGER)  return UNSIGNED;
function "+"(L: INTEGER;  R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED;   R: INTEGER)  return SIGNED;
function "+"(L: INTEGER;  R: SIGNED)   return SIGNED;
function "+"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
function "+"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED;   R: STD_ULOGIC) return SIGNED;
function "+"(L: STD_ULOGIC; R: SIGNED)   return SIGNED;

function "+"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "+"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "+"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: STD_ULOGIC) return
 STD_LOGIC_VECTOR;
function "+"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "+"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED;   R: SIGNED)   return SIGNED;
function "-"(L: UNSIGNED; R: SIGNED)   return SIGNED;
function "-"(L: SIGNED;   R: UNSIGNED) return SIGNED;
function "-"(L: UNSIGNED; R: INTEGER)  return UNSIGNED;
function "-"(L: INTEGER;  R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED;   R: INTEGER)  return SIGNED;
function "-"(L: INTEGER;  R: SIGNED)   return SIGNED;
function "-"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
function "-"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED;   R: STD_ULOGIC) return SIGNED;
function "-"(L: STD_ULOGIC; R: SIGNED)   return SIGNED;

function "-"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "-"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "-"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: STD_ULOGIC) return
 STD_LOGIC_VECTOR;
function "-"(L: STD_ULOGIC; R: UNSIGNED) return
 STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "-"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;

function "*"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
```

```
function "*"(L: SIGNED;   R: SIGNED)   return SIGNED;
function "*"(L: SIGNED;   R: UNSIGNED) return SIGNED;
function "*"(L: UNSIGNED; R: SIGNED)   return SIGNED;
```

Example 10-6: Example 9-6Unary Arithmetic Functions

```
function "+"(L: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED)   return SIGNED;
function "-"(L: SIGNED)   return SIGNED;
function "ABS"(L: SIGNED) return SIGNED;
```

These functions determine the width of their return values as follows:

1. When only one `UNSIGNED` or `SIGNED` argument is present, the width of the return value is the same as that argument.
2. When both arguments are either `UNSIGNED` or `SIGNED`, the width of the return value is the larger of the two argument widths. An exception is that when an `UNSIGNED` number is added to or subtracted from a `SIGNED` number of the same size or smaller, the return value is a `SIGNED` number one bit wider than the `UNSIGNED` argument. This size guarantees that the return value is large enough to hold any (positive) value of the `UNSIGNED` argument.

The number of bits returned by + and - is illustrated in Table 10-2.

```
signal U4: UNSIGNED (3 downto 0);
signal U8: UNSIGNED (7 downto 0);
signal S4: SIGNED (3 downto 0);
signal S8: SIGNED (7 downto 0);
```

Table 10-2: Number of Bits Returned by + and -

+ or -	U4	U8	S4	S8
U4	4	8	5	8
U8	8	8	9	9
S4	5	9	4	8
S8	8	9	8	8

In some circumstances, you might need to obtain a carry-out bit from the + or - operation. To do this, extend the larger operand by one bit. The high bit of the return value is the carry-out bit, as illustrated in Example 10-7.

Example 10-7: Using the Carry-Out Bit

```
process
    variable a, b, sum: UNSIGNED (7 downto 0);
    variable temp: UNSIGNED (8 downto 0);
    variable carry: BIT;
begin
    temp  := CONV_UNSIGNED(a,9) + b;
    sum   := temp(7 downto 0);
    carry := temp(8);
end process;
```

Comparison Functions

The `std_logic_arith` package provides functions to compare `UNSIGNED` and `SIGNED` data types to each other and to the predefined type `INTEGER`. FPGA Express compares the *numeric* values of the arguments, returning a Boolean value. For example, the following expression evaluates to `TRUE`.

```
UNSIGNED'("001") > SIGNED'("111")
```

The `std_logic_arith` comparison functions are similar to the built-in VHDL comparison functions. The only difference is that the `std_logic_arith` functions accommodate signed numbers and varying bit widths. The predefined VHDL comparison functions perform bit-wise comparisons and so do not have the correct semantics for comparing numeric values (see ‘‘Relational Operators" in Chapter 5).

These functions produce comparators. The function declarations are listed in two groups, ordering functions (<, <=, >, and >=) and equality functions (= and /=), in Examples 10-8 and 10-9.

Example 10-8: Ordering Functions

```
function "<"(L: UNSIGNED; R: UNSIGNED) return Boolean;
function "<"(L: SIGNED;   R: SIGNED)   return Boolean;
function "<"(L: UNSIGNED; R: SIGNED)   return Boolean;
function "<"(L: SIGNED;   R: UNSIGNED) return Boolean;
function "<"(L: UNSIGNED; R: INTEGER)  return Boolean;
function "<"(L: INTEGER;  R: UNSIGNED) return Boolean;
function "<"(L: SIGNED;   R: INTEGER)  return Boolean;
function "<"(L: INTEGER;  R: SIGNED)   return Boolean;

function "<="(L: UNSIGNED; R: UNSIGNED) return Boolean;
function "<="(L: SIGNED;   R: SIGNED)   return Boolean;
function "<="(L: UNSIGNED; R: SIGNED)   return Boolean;
function "<="(L: SIGNED;   R: UNSIGNED) return Boolean;
function "<="(L: UNSIGNED; R: INTEGER)  return Boolean;
function "<="(L: INTEGER;  R: UNSIGNED) return Boolean;
function "<="(L: SIGNED;   R: INTEGER)  return Boolean;
function "<="(L: INTEGER;  R: SIGNED)   return Boolean;

function "" functions">">"(L: UNSIGNED; R: UNSIGNED) return Boolean;
function ">"(L: SIGNED;   R: SIGNED)   return Boolean;
function ">"(L: UNSIGNED; R: SIGNED)   return Boolean;
function ">"(L: SIGNED;   R: UNSIGNED) return Boolean;
function ">"(L: UNSIGNED; R: INTEGER)  return Boolean;
function ">"(L: INTEGER;  R: UNSIGNED) return Boolean;
function ">"(L: SIGNED;   R: INTEGER)  return Boolean;
function ">"(L: INTEGER;  R: SIGNED)   return Boolean;

function ="" functions">">="(L: UNSIGNED; R: UNSIGNED) return Boolean;
function ">="(L: SIGNED;   R: SIGNED)   return Boolean;
function ">="(L: UNSIGNED; R: SIGNED)   return Boolean;
function ">="(L: SIGNED;   R: UNSIGNED) return Boolean;
function ">="(L: UNSIGNED; R: INTEGER)  return Boolean;
function ">="(L: INTEGER;  R: UNSIGNED) return Boolean;
function ">="(L: SIGNED;   R: INTEGER)  return Boolean;
function ">="(L: INTEGER;  R: SIGNED)   return Boolean;
```

Example 10-9: Equality Functions

```
function "="(L: UNSIGNED; R: UNSIGNED) return Boolean;
function "="(L: SIGNED;   R: SIGNED)   return Boolean;
function "="(L: UNSIGNED; R: SIGNED)   return Boolean;
function "="(L: SIGNED;   R: UNSIGNED) return Boolean;
function "="(L: UNSIGNED; R: INTEGER)  return Boolean;
function "="(L: INTEGER;  R: UNSIGNED) return Boolean;
function "="(L: SIGNED;   R: INTEGER)  return Boolean;
function "="(L: INTEGER;  R: SIGNED)   return Boolean;

function "/="(L: UNSIGNED; R: UNSIGNED) return Boolean;
function "/="(L: SIGNED;   R: SIGNED)   return Boolean;
function "/="(L: UNSIGNED; R: SIGNED)   return Boolean;
function "/="(L: SIGNED;   R: UNSIGNED) return Boolean;
function "/="(L: UNSIGNED; R: INTEGER)  return Boolean;
function "/="(L: INTEGER;  R: UNSIGNED) return Boolean;
function "/="(L: SIGNED;   R: INTEGER)  return Boolean;
function "/="(L: INTEGER;  R: SIGNED)   return Boolean;
```

Shift Functions

The `std_logic_arith` package provides functions for shifting the bits in `SIGNED` and `UNSIGNED` numbers. These functions produce shifters. Example 10-10 shows the shift function declarations.

Example 10-10: Shift Functions

```
function SHL(ARG: UNSIGNED;
             COUNT: UNSIGNED)  return UNSIGNED;
function SHL(ARG: SIGNED;
             COUNT: UNSIGNED)  return SIGNED;


function SHR(ARG: UNSIGNED;
             COUNT: UNSIGNED)  return UNSIGNED;
function SHR(ARG: SIGNED;
             COUNT: UNSIGNED)  return SIGNED;
```

The `SHL` function shifts the bits of its argument `ARG` to the *left* by `COUNT` bits. `SHR` shifts the bits of its argument `ARG` to the *right* by `COUNT` bits.

The `SHL` functions work the same for both `UNSIGNED` and `SIGNED` values of `ARG`, shifting in zero bits as necessary. The `SHR` functions treat `UNSIGNED` and `SIGNED` values differently. If `ARG` is an `UNSIGNED` number, vacated bits are filled with zeros; if `ARG` is a `SIGNED` number, the vacated bits are copied from the sign bit of `ARG`.

Example 10-11 shows some shift function calls and their return values.

Example 10-11: Shift Operations

```
variable U1, U2: UNSIGNED (7 downto 0);
variable S1, S2: SIGNED   (7 downto 0);
variable COUNT:  UNSIGNED (1 downto 0);
. . .
U1 := "01101011";
U2 := "11101011";

S1 := "01101011";
S2 := "11101011";

COUNT := CONV_UNSIGNED(ARG => 3, SIZE => 2);
. . .
SHL(U1, COUNT) = "01011000"
SHL(S1, COUNT) = "01011000"
SHL(U2, COUNT) = "01011000"
SHL(S2, COUNT) = "01011000"

SHR(U1, COUNT) = "00001101"
SHR(S1, COUNT) = "00001101"
SHR(U2, COUNT) = "00011101"
SHR(S2, COUNT) = "11111101"
```

Multiplication Using Shifts

You can use shift operations for simple multiplication and division of `UNSIGNED` numbers, if you multiply or divide by a power of two.

For example, to divide the following `UNSIGNED` variable `U` by 4:

```
variable U: UNSIGNED (7 downto 0) := "11010101";
variable quarter_U: UNSIGNED (5 downto 0);

quarter_U := SHR(U, "01");
```

ENUM_ENCODING Attribute

Place the synthesis attribute ENUM_ENCODING on your primary logic type (see ‘‘Enumeration Encoding" in Chapter 4). This attribute allows FPGA Express to interpret your logic correctly.

pragma *built_in*

Label your primary logic functions with the built_in pragma. This pragma allows FPGA Express to interpret your logic functions easily. When you use a built_in pragma, FPGA Express parses but ignores the body of the function. Instead, FPGA Express directly substitutes the appropriate logic for the function. You need not use built_in pragmas; however using these pragmas result in run times that are ten times faster.

Use built_in pragmas by placing a comment in the declaration part of a function. FPGA Express interprets a comment as a directive if the first word of the comment is pragma.

Example 10-12 shows the use of built_in pragmas.

Example 10-12: Using a built_in pragma

```
function "XOR" (L, R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
  -- pragma built_in SYN_XOR
    begin
        if (L = '1') xor (R = '1') then
            return '1';
        else
            return '0';
        end if;
end "XOR";
```

Two-Argument Logic Functions

Synopsys provides six built-in functions to perform two-argument logic functions:

- SYN_AND
- SYN_OR
- SYN_NAND
- SYN_NOR
- SYN_XOR
- SYN_XNOR

You can use these functions on single-bit arguments or equal-length arrays of single bits.

Example 10-13 shows a function that generates the logical AND of two equal-size arrays.

Example 10-13: Built-In AND for Arrays

```
function "AND" (L, R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
  -- pragma built_in SYN_AND
    variable MY_L: STD_LOGIC_VECTOR (L'length-1 downto 0);
    variable MY_R: STD_LOGIC_VECTOR (L'length-1 downto 0);
    variable RESULT: STD_LOGIC_VECTOR (L'length-1 downto 0);
begin
    assert L'length = R'length;
    MY_L := L;
    MY_R := R;
    for i in RESULT'range loop
        if (MY_L(i) = '1') and (MY_R(i) = '1') then
            RESULT(i) := '1';
        else
            RESULT(i) := '0';
        end if;
    end loop;
    return RESULT;
end "AND";
```

One-Argument Logic Functions

Synopsys provides two built-in functions to perform one-argument logic functions:

- SYN_NOT
- SYN_BUF

You can use these functions on single-bit arguments or equal-length arrays of single bits. Example 10-14 shows a function that generates the logical NOT of an array.

Example 10-14: Built-In NOT for Arrays

```
function "NOT" (L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
  -- pragma built_in SYN_NOT
     variable MY_L: STD_LOGIC_VECTOR (L'length-1 downto 0);
     variable RESULT: STD_LOGIC_VECTOR (L'length-1 downto 0);
begin
    MY_L := L;
    for i in result'range loop
        if (MY_L(i) = '0' or MY_L(i) = 'L') then
            RESULT(i) := '1';
```

```
        elsif (MY_L(i) = '1' or MY_L(i) = 'H') then
            RESULT(i) := '0';
        else
            RESULT(i) := 'X';
        end if;
    end loop;
    return RESULT;
end "NOT";
end;
```

Type Conversion

The built-in function SYN_FEED_THRU performs fast type conversion between unrelated types. The synthesized logic from SYN_FEED_THRU wires the single input of a function to the return value. This connection can save the CPU time required to process a complicated conversion function, as shown in Example 10-15.

Example 10-15: Use of SYN_FEED_THRU

```
type COLOR is (RED, GREEN, BLUE);
attribute ENUM_ENCODING : STRING;
attribute ENUM_ENCODING of COLOR : type is "01 10 11";
...

function COLOR_TO_BV (L: COLOR) return BIT_VECTOR is
  -- pragma built_in SYN_FEED_THRU
begin
    case L is
       when RED   => return "01";
       when GREEN => return "10";
       when BLUE  => return "11";
    end case;
end COLOR_TO_BV;
```

translate_off Directive

If there are constructs in your "types" package that are not supported for synthesis, or that produce warning messages, you may need to use the FPGA Express directive `-- synopsys translate_off`.

You can make liberal use of the `translate_off` directive when you use `built_in` pragmas because FPGA Express ignores the body of `built_in` functions. For examples of illustrating how to use the `translate_off` directive, see the `std_logic_arith.vhd` package.

std_logic_misc Package

The std_logic_misc package resides in the Synopsys libraries directory ($synopsys/packages/IEEE/src/std_logic_misc.vhd). This package declares the primary data types supported by the Synopsys VSS Family.

Boolean reduction functions use one argument, an array of bits, and return a single bit. For example, the and-reduction of "101" is "0", the logical AND of all three bits.

Several functions in the std_logic_misc package provide Boolean reduction operations for the pre-defined type STD_LOGIC_VECTOR. Example 10-16 shows the declarations of these functions.

Example 10-16: Boolean Reduction Functions

```
function AND_REDUCE  (ARG: STD_LOGIC_VECTOR) return UX01;
function NAND_REDUCE (ARG: STD_LOGIC_VECTOR) return UX01;
function OR_REDUCE   (ARG: STD_LOGIC_VECTOR) return UX01;
function NOR_REDUCE  (ARG: STD_LOGIC_VECTOR) return UX01;
function XOR_REDUCE  (ARG: STD_LOGIC_VECTOR) return UX01;
function XNOR_REDUCE (ARG: STD_LOGIC_VECTOR) return UX01;
function AND_REDUCE  (ARG: STD_ULOGIC_VECTOR) return UX01;
function NAND_REDUCE (ARG: STD_ULOGIC_VECTOR) return UX01;
function OR_REDUCE   (ARG: STD_ULOGIC_VECTOR) return UX01;
function NOR_REDUCE  (ARG: STD_ULOGIC_VECTOR) return UX01;
function XOR_REDUCE  (ARG: STD_ULOGIC_VECTOR) return UX01;
function XNOR_REDUCE (ARG: STD_ULOGIC_VECTOR) return UX01;
```

These functions combine the bits of the STD_LOGIC_VECTOR, as the name of the function indicates. For example, XOR_REDUCE returns the XOR value of all bits in ARG.

Example 10-17 shows some reduction function calls and their return values.

Example 10-17: Boolean Reduction Operations

```
AND_REDUCE("111") = '1'
AND_REDUCE("011") = '0'

OR_REDUCE("000")  = '0'
OR_REDUCE("001")  = '1'

XOR_REDUCE("100") = '1'
XOR_REDUCE("101") = '0'

NAND_REDUCE("111") = '0'
NAND_REDUCE("011") = '1'
```

```
NOR_REDUCE("000") = '1'
NOR_REDUCE("001") = '0'

XNOR_REDUCE("100") = '0'
XNOR_REDUCE("101") = '1'
```

Chapter 11
HDL Constructs

Many VHDL language constructs, although useful for simulation and other stages in the design process, are not relevant to synthesis. Because these constructs cannot be synthesized, they are not supported by FPGA Express.

This appendix provides a list of all VHDL language constructs with the level of support for each, followed by a list of VHDL reserved words.

This appendix describes

- VHDL Construct Support
- VHDL Reserved Words

VHDL Construct Support

A construct can be fully supported, ignored, or unsupported. Ignored and unsupported constructs are defined as follows:

- Ignored means that the construct is allowed in the VHDL source, but is ignored by FPGA Express.
- Unsupported means that the construct is not allowed in the VHDL source and that FPGA Express flags the construct as an error. If errors are found in a VHDL description, the description is not translated (synthesized).

Constructs are listed in the following order:

- Design units
- Data types
- Declarations
- Specifications
- Names
- Operators
- Operands and expressions
- Sequential statements
- Concurrent statements
- Predefined language environment

Design Units

entity

The entity statement part is ignored.

Generics are supported, but only of type `INTEGER`.

Default values for ports are ignored.

architecture

Multiple architectures are allowed.

Global signal interaction between architectures is unsupported.

configuration

Configuration declarations and block configurations are supported, but only to specify the top-level architecture for a top-level entity.

Attribute specifications, `use` clauses, component configurations, and nested block configurations are unsupported.

package

Packages are fully supported.

library

Libraries and separate compilation are supported.

subprogram

Default values for parameters are unsupported. Assigning to indexes and slices of unconstrained `out` parameters is unsupported, unless the actual parameter is an identifier.

Subprogram recursion is unsupported if the recursion is not bounded by a static value.

Resolution functions are supported for wired-logic and three-state functions only.

Subprograms can only be declared in packages and in the declaration part of an architecture.

Data Types

enumeration

Enumeration is fully supported.

integer

Infinite-precision arithmetic is unsupported.

Integer types are automatically converted to bit vectors whose width is as small as possible to accommodate all possible values of the type's range, either in unsigned binary for nonnegative ranges, or in 2's-complement form for ranges that include negative numbers.

physical

Physical type declarations are ignored. The use of physical types is ignored in delay specifications.

floating

Floating-point type declarations are ignored. The use of floating-point types is unsupported except for floating-point constants used with Synopsys-defined attributes (see Chapter 9).

array

Array ranges and indexes other than integers are unsupported.

Multidimensional arrays are unsupported, but arrays of arrays are supported.

record

Record data types are fully supported.

access

Access type declarations are ignored, and the use of access types is unsupported.

file

File type declarations are ignored, and the use of file types is unsupported.

incomplete type declarations

Incomplete type declarations are unsupported.

Declarations

constant

Constant declarations are supported, except for deferred constant declarations.

signal

`register` and `bus` declarations are unsupported.

Resolution functions are supported for wired and three-state functions only.

Declarations other than from a globally static type are unsupported.

Initial values are unsupported.

variable

Declarations other than from a globally static type are unsupported.

Initial values are unsupported.

file

File declarations are unsupported.

interface

`buffer` and `linkage` are translated to `out` and `inout`, respectively.

alias

Alias declarations are ignored.

component

Component declarations that list a name other than a valid entity name are unsupported.

attribute

Attribute declarations are fully supported. However, the use of user-defined attributes is unsupported.

Specifications

attribute

`others` and `all` are unsupported in attribute specifications.

User-defined attributes can be specified, but the use of user-defined attributes is unsupported.

configuration

Configuration specifications are unsupported.

disconnection

Disconnection specifications are unsupported.

Attribute declarations are fully supported. However, the use of user-defined attributes is unsupported.

Names

simple

Simple names are fully supported.

selected

Selected (*qualified*) names outside of a `use` clause are unsupported.

Overriding the scopes of identifiers is unsupported.

operator symbols

Operator symbols are fully supported.

indexed

Indexed names are fully supported, with one exception. Indexing an unconstrained `out` parameter in a procedure is unsupported.

slice

Slice names are fully supported, with one exception. Using a slice of an unconstrained `out` parameter in a procedure is unsupported unless the actual parameter is an identifier.

attribute

Only the following predefined attributes are supported: `base`, `left`, `right`, `high`, `low`, `range`, `reverse_range`, and `length.`

`event` and `stable` attributes are supported only as described with the `wait` and `if` statements (see Chapter 6).

User-defined attribute names are unsupported.

The use of attributes with selected names (`name.name'attribute`) is unsupported.

Operators

logical

Logical operators are fully supported.

relational

Relational operators are fully supported.

addition

Concatenation and arithmetic operators are both fully supported.

signing

Signing operators are fully supported.

multiplying

The * (multiply) operator is fully supported.

The / (division), mod, and rem operators are supported only when both operands are constant or when the right operand is a constant power of 2.

miscellaneous

The ** operator is supported only when both operands are constant or when the left operand is 2.

The abs operator is fully supported.

operator overloading

Operator overloading is fully supported.

short-circuit operations

The short-circuit behavior of operators is not supported.

Operands and Expressions

based literals

Based literals are fully supported.

null literals

Null slices, null ranges, and null arrays are unsupported.

physical literals

Physical literals are ignored.

strings

Strings are fully supported.

aggregates

The use of types as aggregate choices is unsupported.

Record aggregates are unsupported.

function calls

Function conversions on input ports are not supported, because type conversions on formal ports in a connection specification are unsupported.

qualified expressions

Qualified expressions are fully supported.

type conversion

Type conversion is fully supported.

allocators

Allocators are unsupported.

static expressions

Static expressions are fully supported.

universal expressions

Floating-point expressions are unsupported, except in a Synopsys-recognized attribute definition.

Infinite-precision expressions are not supported.

Precision is limited to 32 bits; all intermediate results are converted to integer.

Sequential Statements

wait

The `wait` statement is unsupported unless it is of one the following forms:

```
wait until                        clock = VALUE;
wait until     clock'event  and clock = VALUE;
wait until not clock'stable and clock = VALUE;
```

where `VALUE` is `0`, `1` or an enumeration literal whose encoding is `0` or `1`. A `wait` statement in this form is interpreted to mean “wait until the falling (`VALUE` is `0`) or rising (`VALUE` is `1`) edge of the signal named `clock`.”

`wait` statements cannot be used in subprograms or in `for` loops.

assertion

`assertion` statements are ignored.

signal

Guarded signal assignment is unsupported.

`transport` and `after` are ignored.

Multiple waveform elements in signal assignment statements are unsupported.

variable

`variable` statements are fully supported.

procedure call

Type conversion on formal parameters is unsupported.

Assignment to single bits of vectored ports is unsupported.

if

`if` statements are fully supported.

case

`case` statements are fully supported.

loop

`for` loops are supported, with two constraints: the loop index range must be globally static, and the loop body must not contain a `wait` statement.

`while` loops are supported, but the loop body must contain at least one `wait` statement.

`loop` statements with no iteration scheme (infinite loops) are supported, but the loop body must contain at least one `wait` statement.

next

`next` statements are fully supported.

exit

`exit` statements are fully supported.

return

`return` statements are fully supported.

null

`null` statements are fully supported.

Concurrent Statements

block

Guards on `block` statements are unsupported.

Ports and generics in `block` statements are unsupported.

process

Sensitivity lists in `process` statements are ignored.

concurrent procedure call

Concurrent procedure call statements are fully supported.

concurrent assertion

Concurrent assertion statements are ignored.

concurrent signal assignment

The `guarded` and `transport` keywords are ignored. Multiple waveforms are unsupported.

component instantiation

Type conversion on the formal port of a connection specification is unsupported.

generate

`generate` statements are fully supported.

Predefined Language Environment

severity_level type

`severity_level` type is unsupported.

time type

`time` type is unsupported.

now function

`now` function is unsupported.

TEXTIO package

The `TEXTIO` package is unsupported.

predefined attributes

Predefined attributes are unsupported, except for `base`, `left`, `right`, `high`, `low`, `range`, `reverse_range`, and `length`.

The `event` and `stable` attributes are supported only in the `if` and `wait` statements, as described in Chapter 6.

VHDL Reserved Words

The following words are reserved for the VHDL language and cannot be used as identifiers:

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

Index