VHDL Reference Guide

About This Manual

This manual describes how to use the Xilinx Foundation Express program to compile VHDL designs.

Before using this manual, you should be familiar with the operations that are common to all Xilinx software tools. These operations are covered in the *Quick Start Guide*.

Additional Resources

For additional information, go to http://support.xilinx.com. The following table lists some of the resources you can access from this page. You can also directly access some of these resources using the provided URLs.

Resource	Description/URL
Tutorial	Tutorials covering Xilinx design flows, from design entry to verification and debugging http://support.xilinx.com/support/techsup/tutorials/index.htm
Answers Database	Current listing of solution records for the Xilinx software tools Search this database using the search function at http://support.xilinx.com/support/searchtd.htm
Application Notes	Descriptions of device-specific design techniques and approaches http://support.xilinx.com/apps/appsweb.htm
Data Book	Pages from *The Programmable Logic Data Book*, which describe device-specific information on Xilinx device characteristics, including read-back, boundary scan, configuration, length count, and debugging http://support.xilinx.com/partinfo/databook.htm

Resource	Description/URL
Xcell Journals	Quarterly journals for Xilinx programmable logic users http://support.xilinx.com/xcell/xcell.htm
Tech Tips	Latest news, design tips, and patch information on the Xilinx design environment http://support.xilinx.com/support/techsup/journals/index.htm

Manual Contents

This manual covers the following topics.

- Chapter 1, “Using Foundation Express with VHDL,” discusses general concepts about VHDL and the Foundation Express design process and methodology.
- Chapter 2, “Design Descriptions,” describes the use and importance of hierarchy in VHDL design entities.
- Chapter 3, “Data Types,” describes VHDL data types and their uses.
- Chapter 4, “Expressions,” discusses individual components of expressions and how to use them.
- Chapter 5, “Sequential Statements,” describes and illustrates the various types of sequential statements.
- Chapter 6, “Concurrent Statements,” defines and illustrates concurrent statements and their function.
- Chapter 7, “Register and Three-State Inference,” illustrates how to write VHDL descriptions to produce efficient synthesized circuits.
- Chapter 8, “Writing Circuit Descriptions,” describes how to write a VHDL description to ensure an efficient implementation.
- Chapter 9, “Foundation Express Directives,” explains how to use Foundation Express directives and Xilinx-defined VHDL attributes to provide circuit design information directly into the VHDL source code.
- Chapter 10, “Foundation Express Packages,” discusses the contents of three VHDL packages included with this release that

are a combination of standard IEEE packages and Synopsys packages that have been added to the standard IEEE package.

- Chapter 11, “VHDL Constructs,” provides a list of all VHDL language constructs with the level of support for each one and a list of VHDL reserved words.
- Appendix A, “Examples,” presents examples that demonstrate basic concepts of Foundation Express.

Conventions

This manual uses the following typographical and online document conventions. An example illustrates each typographical convention.

Typographical

The following conventions are used for all documents.

- `Courier font` indicates messages, prompts, and program files that the system displays.

  ```
  speed grade: -100
  ```

- **`Courier bold`** indicates literal commands that you enter in a syntactical statement. However, braces "{ }" in Courier bold are not literal and square brackets "[ ]" in Courier bold are literal only in the case of bus specifications, such as bus [7:0].

  `rpt_del_net=`

  `Courier bold` also indicates commands that you select from a menu.

  `File` → **`Open`**

- *Italic font* denotes the following items.

  - Variables in a syntax statement for which you must supply values

    `edif2ngd` *design_name*

  - References to other manuals

    See the *Development System Reference Guide* for more information.

- Emphasis in text

  If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.

- Square brackets "[ ]" indicate an optional entry or parameter. However, in bus specifications, such as bus [7:0], they are required.

  edif2ngd [*option_name*] *design_name*

- Braces "{ }" enclose a list of items from which you must choose one or more.

  lowpwr ={**on** | **off**}

- A vertical bar " | " separates items in a list of choices.

  lowpwr ={**on** | **off**}

- A vertical ellipsis indicates repetitive material that has been omitted.

```
IOB #1: Name = QOUT'
IOB #2: Name = CLKIN'
.
.
.
```

- A horizontal ellipsis ". . ." indicates that an item can be repeated one or more times.

  allow block *block_name loc1 loc2* . . . *locn;*

Online Document

The following conventions are used for online documents.

- Red-underlined text indicates an interbook link, which is a cross-reference to another book. Click the red-underlined text to open the specified cross-reference.
- Blue-underlined text indicates an intrabook link, which is a cross-reference within a book. Click the blue-underlined text to open the specified cross-reference.

Contents

Chapter 3 Data Types

Chapter 7 Register and Three-State Inference

Chapter 8 Writing Circuit Descriptions

Chapter 1

Using Foundation Express with VHDL

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

This chapter discusses concepts that you need to work with VHDL. These concepts are covered in the following sections.

- "Hardware Description Languages"
- "About VHDL"
- "Foundation Express Design Process"
- "Using Foundation Express to Compile a VHDL Design"
- "Design Methodology"

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 programing languages, but includes many hardware-specific constructs.

Foundation Express reads and parses the supported VHDL syntax. The "VHDL Constructs" chapter lists all VHDL constructs and includes the level of 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 development of 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 component-level or gate-level implementation. Alternatively, you can read an HDL design description into Foundation 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. Some of the advantages are listed below.

- 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.

- Foundation Express synthesizes and optimizes logic 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 a VHDL specification is hand-translated to gates. With Foundation 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 supply technology-independent documentation of a design and its functionality.

  An HDL description is more easily read and understood than a netlist or schematic description. Because 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. Incorrectly using types is 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 a few HDLs in widespread use today. VHDL is recognized as a standard HDL by the Institute of Electrical and Electronics Engineers (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, relative 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 to use with many designs or a family of designs.

A VHDL hardware model is shown in the following figure.

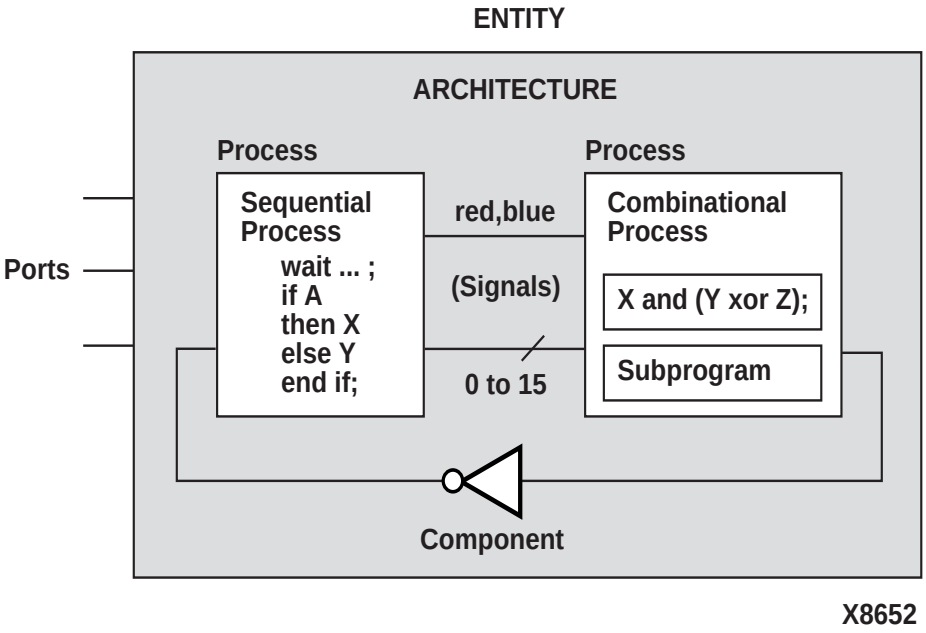


Figure 1-1 VHDL Hardware Model

A VHDL *entity* (design) has one or more input, output, or inout ports that are connected (wired) to neighboring systems. An entity is 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 combinatorial (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, or 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.

Foundation Express Design Process

Foundation 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 Xilinx technology library

Foundation Express synthesizes VHDL descriptions according to the VHDL *synthesis policy* defined in the "Design Descriptions" chapter. The Xilinx 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 Foundation Express to Compile a VHDL Design

When a VHDL design is read into Foundation Express, the design is converted to an internal database format so that Foundation Express can synthesize and optimize the design. Foundation Express restructures part or all of the design to optimize it. You control the degree of restructuring. Foundation Express includes the following restructuring options.

- Fully preserves a design's hierarchy
- Moves full modules up or down in the hierarchy
- Combines certain modules with others
- Compresses the entire design into one module (called flattening the design)

The "Design Methodology" section describes the design process that uses Foundation Express with a VHDL simulator.

Design Methodology

The following figure shows a typical design process that uses Foundation Express and a VHDL simulator.

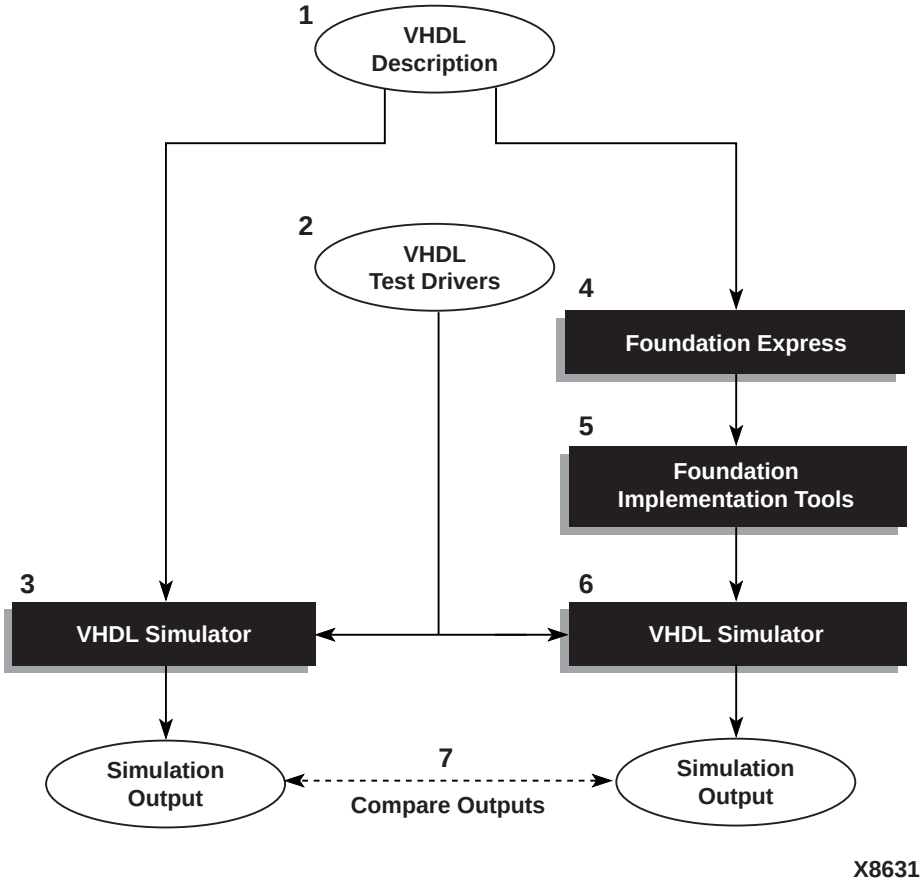


Figure 1-2 Design Flow

The numbers in the above figure are explained below.

1. Write a design description in VHDL.

   This description can be a combination of structural and functional elements (as shown in the "Design Descriptions" chapter). Both Foundation Express and a VHDL simulator use this design description.

2. Provide VHDL test drivers for the simulator.

The drivers supply test vectors for simulation and gather output data. To learn about writing these drivers, see the appropriate simulator manual.

3. Simulate the design by using a VHDL simulator and verify that the description is correct.
4. Using Foundation Express, synthesize and optimize the VHDL design descriptions into a gate-level netlist.

   Foundation Express generates optimized netlists to satisfy timing constraints for a targeted FPGA architecture.
5. Using your Foundation development system, 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

Design Descriptions

Each VHDL structural design can have four parts, which this chapter discusses in the following major sections.

- “Entities”
- “Architecture”
- “Configurations”
- “Packages”
- “Resolution Functions”

Entities

An entity defines the input and output ports of a design. A design can contain more than one entity. Each entity has its own architecture statement.

The syntax follows.

```
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.
- port_declarations determine the number and type of input and output ports.

You cannot use the declaration of other in the entity specification.

An entity serves as an interface to other designs, by defining entity characteristics that must be known to Foundation Express before it can connect the entity 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 input and output ports, as shown in the following example.

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

Entity Generic Specifications

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

A generic can have a default value. It receives a nondefault value only when the entity is instantiated (see the "Declarations" section of this chapter) or configured (see the "Configurations" section of this chapter). Inside an entity, a generic is a constant value.

The syntax follows.

```
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.
- Optional value is the default value of constant_name.

Entity Port Specifications

Port specifications define the number and type of ports in the entity. The syntax follows.

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

- port_name is the name of the port.
- mode is any of these four values.
  - 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 is similar to out but can be read. The value read is the assigned value. It can have only one driver. For more information about drivers, see "Driving Signals."
- port_type is a previously defined data type.

The following example shows an entity specification for a 2-input N-bit comparator with a default bit-width of 8.

```
-- 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;
```

Architecture

Architecture, which determines the implementation of an entity, 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 follows.

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

- architecture_name is the name of the architecture.
- entity_name is the name of the entity being implemented.
- block_declarative_item is any of the following statements.
  - use statement (See the "Type Declarations" section of this chapter.)
  - Subprogram Declarations

- Subprogram Body
- Type Declarations
- Subtype Declarations
- Constant Declarations
- Signal Declarations
- Concurrent Statements
  Define a unit of computation that reads signals, performs computations, and assigns values to signals

The following example shows a description for a 3-bit counter that contains an entity specification and an architecture statement.

- Entity specification for COUNTER3
- Architecture statement, MY_ARCH

```
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
                     -- Check 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;
```

The following figure shows a schematic of the previous example.

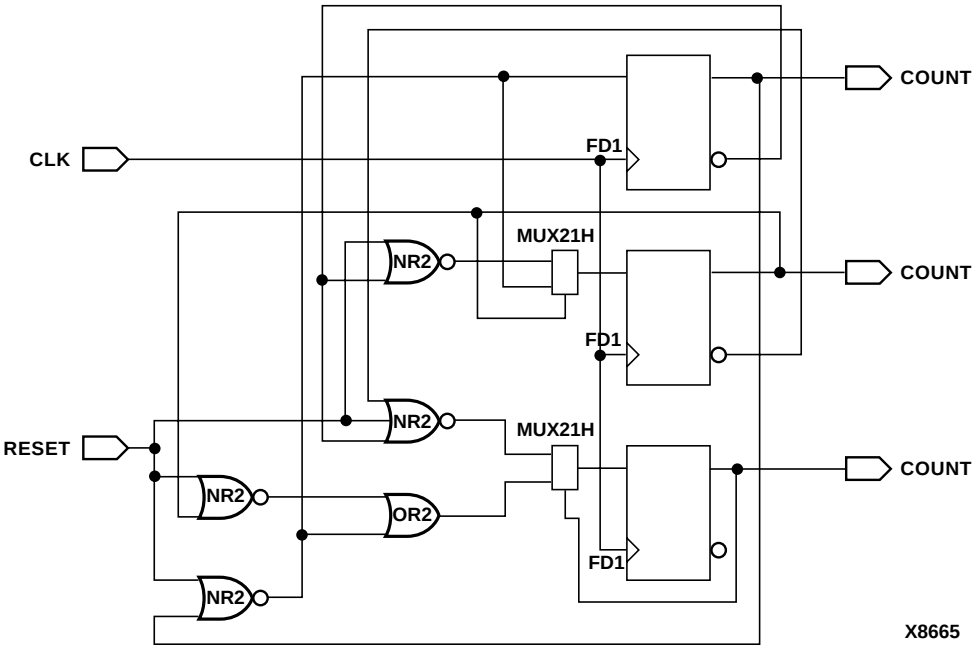


Figure 2-1 3-Bit Counter Synthesized Circuit

Note: In an architecture, you must not give constants or signals the same name as any of the entity's ports in the entity specification.

If you declare a constant or signal with a port's name, the new declaration hides that port name. If the new declaration lies directly in the architecture declaration (as shown in the following example) and not in an inner block, Foundation Express reports an error.

```
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 the previous example follow.

```
  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.
```

Declarations

An architecture consists of a declaration section where you declare the following.

- Components
- Concurrent Statements
- Constant Declarations
- Processes
- Signal Declarations
- Subprograms
- Type Declarations

Components

If your design consists only of VHDL entity statements, every component declaration in the architecture or package statement has to correspond to an entity.

Components declared in an architecture are local to that architecture.

The syntax follows.

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

- identifier is the name of the component.

  You cannot use names preceded by GTECH_ for components other than ones provided by Foundation Express. However, you

can use GTECH to precede a name if it is used without an underscore, as in GTECHBUSTBUF.

- generic_declaration determines local constants used for sizing or timing the component.
- port_declartion determines the number and type of input and output ports.

The following example shows a simple component declaration statement.

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

The following example shows a component declaration statement that uses a generic parameter.

```
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;
```

The component declaration makes a design entity (AND2 in the example of the 2-input AND gate and ADD in the example of the N-bit adder) usable within an architecture. You must declare a component in an architecture or package before you can instantiate it.

Sources of Components A declared component can come from the following.

- The same VHDL source file
- A different VHDL source file
- Another format, such as EDIF or XNF.
- A component from a technology library

Consistency of Component Ports Foundation Express checks for consistency among its VHDL entities. For other entities, the port names are taken from the original design description as follows.

- 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.

- For a VHDL component, Foundation Express verifies matching.
- For components from other sources, Foundation Express checks when linking the component to the VHDL description.

Component Instantiation Statement You use a component instantiation statement to define a design hierarchy or build a netlist in VHDL. A netlist is a structural description of a design.

To form a netlist, use component instantiation statements to instantiate and connect components. A component instantiation statement creates a new level of design hierarchy.

The syntax of the component instantiation statement follows.

```
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 as in the following.

  ```
  U1 : ADD
  ```

- generic map (optional) maps 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.

  ```
  U1 : ADD generic map (N => 4)
  ```

- port map maps 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.

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

Foundation Express uses the following two rules to select which entity and architecture to associate with a component instantiation.

- Each component declaration must have an entity—a VHDL entity, a design entity from another source or format, or a library component—with the same name. This entity is used for each component instantiation associated with the component declaration.
- A VHDL entity may have only one architecture associated with it. If multiple architectures are available, add only one of these files to the Design Sources window.

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 in the following example might use the following generic map.

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

Mapping Port Connections The port map maps component ports to actual signals.

Use named or positional association to specify port connections in component instantiation statements, as follows.

- To identify the specific ports of the component, use named association. The port_name => construction identifies the ports.
- To list the component port expressions in the declared port order, use positional association.

The first example that follows shows named and positional association for the U5 component instantiation statement in the second example.

```
EU5: 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 ports in the component declaration statement.

The following example shows a structural netlist description for the COUNTER3 design entity.

```
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;
```

Concurrent Statements

Each concurrent statement in an architecture defines a unit of computation that does the following.

- Reads signals
- Performs a computation that is based on the values of the signals
- Assigns the computed values to the signals

Concurrent statements all compute their values at the same time. Although the order of concurrent statements has no effect on the order in which Foundation Express executes them, concurrent statements coordinate their processing by communicating with each other through signals.

The five kinds of concurrent statements follow.

- Block

  Groups a set of concurrent statements.

- Component instantiation

  Creates an instance of an entity, connecting its interface ports to signals or interface ports of the entity being defined. See the "Component Instantiation Statement" section of this chapter.

- Procedure call

  Calls algorithms that compute and assign values to signals.

- Process

  Defines sequential algorithms that read the values of signals and compute new values to assign to other signals. For a discussion of processes, see the "Declarations" section.

- Signal assignments

  Assign computed values to signals or interface ports.

Concurrent statements are described further in the "Concurrent Statements" chapter.

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. Constants declared in an architecture are local to that architecture. An example of constant declarations follows.

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

You can use constants in expressions, as described in the "Identifiers" section and"Literals" section of the "Expressions" chapter and as source values in assignment statements, as described in the "Assignment Statements and Targets" section of the "Sequential Statements" chapter.

Processes

A process, which is declared within an architecture, is a concurrent statement. But it is made up of sequentially executed statements that define algorithms. The sequential statements can be any of the following, all of which are discussed in the "Sequential Statements" chapter.

- case statement
- exit statement
- if statement
- loop statement
- next statement
- null statement
- Procedure call
- Signal assignment
- Variable assignment
- wait statement

Processes, like all other concurrent statements, read and write signals and the values of interface ports to communicate with the rest of the architecture and with the enclosing system.

Processes are unique in that they behave like concurrent statements to the rest of the design, but they are internally sequential. In addi-

tion, 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.

Variable Declarations Variable declarations define a named value of a given type. An example of variable declarations follows.

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

You can use variables in expressions, as described in the "Expressions" chapter. You assign values to variables by using variable assignment statements, as described in the "Variable Assignment Statements" section of the "Sequential Statements" chapter.

Foundation Express does not support variable initialization. If you try to initialize a variable, Foundation Express generates the following message.

```
Warning: Initial values for signals are not supported
for synthesis. They are ignored on line %n (VHDL-2022)
```

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

Signal Declarations

Signals connect the separate concurrent statements of an architecture to each other, and to other parts of a design, through interface ports.

Signal declarations create new named signals (wires) of a given type. Signals can be given default (initial) values, but these initial values are ignored for synthesis.

Signals with multiple drivers (signals driven by wired logic) can have associated resolution functions, as described in the "Package Body" section. An example of signal declarations follows.

```
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 with port declarations or with signal declarations. You create ports only with 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 the "Sequential Statements" chapter. Signals are assigned values by signal assignment statements, as described in the "Signal Assignment Statements" section of the "Sequential Statements" chapter.

Subprograms

Subprograms use sequential statements to define algorithms and are useful for performing repeated calculations, often in different parts of an architecture. (See the "Subprograms" section of the "Sequential Statements" chapter.) Subprograms declared in an architecture are local to that architecture.

Subprograms differ from processes in that subprograms 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.

Signal declarations create new named signals (wires) of a given type. Signals can be given default (initial) values, but these initial values are ignored for synthesis.

Signals with multiple drivers (signals driven by wired logic) can have associated resolution functions, as described in the "Resolution Functions" section of this chapter..

Subprograms also differ from component instantiation statements, in that the use of a subprogram by an entity or another subprogram does not create a new level of design hierarchy.

There are two types of subprograms, which can have zero or more parameters.

- Procedure Subprogram
  A procedure returns zero or more values through its interface.
- Function Subprogram
  A function returns a single value directly.

A subprogram has two parts.

- Declaration
- Body

Note: 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.

When you declare a subprogram in an architecture, the program body must be in the architecture body but there is no corresponding subprogram declaration.

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

Procedure Declaration Syntax

The syntax of a procedure declaration follows.

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

- proc_name is the name of the procedure.
- parameter_declarations specify the number and type of input and output ports. The syntax follows.

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

  - parameter_name is the name of a parameter.
  - mode is procedure parameters that can be any of the following four modes.

    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 is similar to out but can be read. The value read is the assigned value. A buffer can have only one driver. For more information about drivers, see the "Driving Signals" section of the "Concurrent Statements" chapter.

  - parameter_type is a previously defined data type.

Function Declaration Syntax

The syntax of a function declaration follows.

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

- func_name is the name of the function
- type_name is the type of the function's returned value. Signal parameters of type range cannot be passed to a subprogram.
- parameter_declarations specify the number and type of input and output ports. The syntax follows.

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

  - parameter_name is the name of a parameter.
  - mode: Function parameters can only use the in mode.

    in can only be read.

  - parameter_type is a previously defined data type.

Declaration Examples

The following example shows sample subprogram declarations for a function and a procedure.

```
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.
```

When Foundation Express calls a subprogram, it substitutes actual parameters for the declared formal parameters. Actual parameters are the following.

- Constant values
- Names of signals, variables, constants, or ports

An actual parameter must support the formal parameter's type and mode. For example, Foundation Express does not accept an input port as an out parameter and uses a constant only as an in actual parameter.

The following example shows some calls to the subprogram declarations from the example above.

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

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

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

Procedure Body Syntax

The syntax of a procedure body follows.

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

- procedure_name is the name of the procedure.
- subprogram_declarative_item can be any of the following statements.
  - use clause
  - type declaration
  - subtype declaration
  - constant declaration
  - variable declaration
  - attribute declaration
  - attribute specification
  - subprogram declaration (for local, or nested subprograms)
  - subprogram body (for locally declared subprograms)

Function Body Syntax

The syntax of a function body follows.

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

- function_name is the name of the function.
- subprogram_declarative_item can be any of the following statements.
  - use clause
  - type declaration
  - subtype declaration
  - constant declaration
  - variable declaration
  - attribute declaration
  - attribute specification
  - subprogram declaration (for local, or nested subprograms)
  - subprogram body (for locally declared subprograms)

The following example shows subprogram bodies for the sample subprogram declarations for a function and a procedure.

```
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 which means that one or more subprograms 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.

The following example shows two subprograms with the same name, but different parameter profiles.

```
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 You can overload predefined operators such as +, and, and mod. 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 overload the operators for the types.

The following example shows how some predefined operators are overloaded for a new logic type.

```
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.

Variable Declarations Variable declarations define a named value of a given type.

You can use variables in expressions, as described in the "Identifiers" section and "Literals" section of the "Expressions" chapter. You assign values to variables by using variable assignment statements, as described in the "Variable Assignment" section of the "Sequential Statements" chapter.

Foundation Express does not support variable initialization. If you try to initialize a variable, Foundation Express generates the following message.

```
Warning: Initial values for signals are not supported
for synthesis. They are ignored on line %n (VHDL-2022)
```

The following example shows some 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.

To use these declarations in more than one entity or architecture, place them in a package, as described in the "Examples of Architectures for NAND2 Entity" section.

Type Declarations

You declare each signal with a type that determines the kind of data it carries. Types declared in an architecture are local to that architecture.

You can use type declarations in architectures, packages, entities, blocks, processes, and subprograms.

Type declarations define the name and characteristics of a type. Types and type declarations are fully described in the "Data Types" chapter. 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.

The following example shows a type declaration for type NEW_BIT and some functions and variables of that type.

```
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;
```

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.

The following subtype declaration (NEW_LOGIC) is a subrange of the type declaration in the previous example.

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

You can use subtype declarations wherever you use type declarations: in architectures, packages, entities, blocks, processes, and subprograms.

Examples of Architectures for NAND2 Entity

The following three examples show three different architectures for the entity NAND2. The three examples define equivalent implementations of NAND2. After optimization and synthesis, they all produce the same circuit, a 2-input NAND gate in the target technology. The architecture description style you use for this entity depends on your own preferences.

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

```
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;
```

The following example shows how you can define the entity NAND2 by its logical function.

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

The following example shows another implementation of 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;
```

Configurations

Configurations are not currently supported by Foundation Express.

Packages

A package is a collection of declarations that more than one design can use.

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. A package must contain at least one of the following constructs.

- Constants

  Declare system-wide parameters, such as data-path widths.

- VHDL data type declarations

  Define data types used throughout a design. All entities in a design must use common interface types, such as 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 follows.

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

- LIBRARY_NAME is the name of a VHDL library
- 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.

Note: Foundation Express 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 that one or more entities in a design need. 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 follows.

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

- package_name is the name of this package.
- A package_declarative_item is any of the following statements.
  - use clause (to include other packages)
  - type declaration
  - subtype declaration
  - constant declaration

- signal declaration
- subprogram declaration
- component declaration

The following example shows some 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 previous example declarations, add a use statement at the beginning of your design description as follows.

```
use WORK.EXAMPLE.ALL;
entity . . .
architecture . . .
```

The “Foundation Express Packages” chapter contains more examples of packages and their declarations.

Package Body

A package body includes the following.

- The implementations (bodies) of subprograms declared in the package declaration.
- Internal support subprograms

But designs or entities that use the package never see this information.

The syntax of a package body follows.

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

- package_name is the name of the associated package.
- package_body_declarative_item is any of the following statements.
  - use clause
  - subprogram declaration
  - subprogram body
  - type declaration
  - subtype declaration
  - constant declaration

The “Foundation Express Packages” chapter shows a package declaration and body example that comes with Foundation Express.

Resolution Functions

Resolution functions are used with signals that can be connected (wired together). For example, if two drivers directly connect 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.

1. Declare the signal’s base type.

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

2. Declare the resolution function.

   ```
   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
```

3. Declare the resolved signal's subtype as a subtype of the base type, which includes the name of the resolution function.

```
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)
```

4. Declare resolved signals as resolved subtypes.

```
signal resolved_signal_name:res_type;
-- resolved_signal_name is a resolved signal
```

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

Foundation Express considers the directive only when creating hardware. The body of the resolution function is parsed but ignored. Using unsupported VHDL constructs generates errors. (See the "VHDL Constructs" chapter.)

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

The three resolution function directives follow.

- synopsys resolution_method wired_and
- synopsys resolution_method wired_or
- synopsys resolution_method three_state

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

The following example shows how to create and use a resolved signal and how to use Foundation Express directives for resolution functions. The signal's base type is the predefined type BIT.

```
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
    -- synopsis resolution_method wired_and
  begin
  -- The code in this function is ignored by
  -- the program
  -- 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;
```

The following figure shows the design.

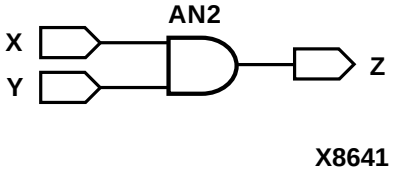


Figure 2-2 Design Using Resolved Signal

Chapter 3

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 supplied with all VHDL implementations.

This chapter describes VHDL data types and their uses. Data type information is included in the following sections.

- “Type Overview”
- “Enumeration Types”
- “Integer Types”
- “Array Types”
- “Record Types”
- “Record Aggregates”
- “Predefined VHDL Data Types”
- “Unsupported Data Types”
- “Express Data Types”
- “Subtypes”

Type Overview

The advantage of strong typing is that VHDL tools can detect many common design errors, such as assigning an 8-bit value to a 4-bit-wide signal or incrementing an array index out of its range.

The following example code shows the definition of a new type, BYTE, as an array of 8 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 in this chapter's examples. Constant, function, and parameter declarations are shown in the "Declarations" section of the "Design Descriptions" chapter.

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 the "Integer Types" section and "Subtypes" section of this chapter).

Enumeration Types

You define an enumeration type by listing (enumerating) all possible values of that type.

The syntax of an enumeration type definition follows.

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

- type_name is an identifier
- 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 the "VHDL Construct Support" section of the "VHDL Constructs" chapter.

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

The following example shows two enumeration type definitions and the corresponding variable and signal declarations.

```
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, Foundation 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 the "Enumeration Literals" section of the "Expressions" chapter.) The following example shows how you can qualify an overloaded enumeration literal.

```
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.

Foundation 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 3-bit encoding vector.

The following example shows the default encoding of an enumeration type with five values.

```
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. The interpretation of the ENUM_ENCODING attribute is specific to Foundation Express.

Several VHDL synthesis-related attributes are declared in the ATTRIBUTES package supplied with Foundation Express. For more information about this package, see the “ATTRIBUTES Package” section of the “Foundation Express Packages” chapter.

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

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 0s, 1s, Ds, Us, and Zs separated by blank spaces. The meaning of these encoding vectors is described in the “Enumeration Encoding Values” section of this chapter.

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.

The following example illustrates how the default encodings from the previous example can be changed with 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 Foundation Express. Other VHDL tools, such as simulators, use the standard encoding (ordering).

Enumeration Encoding Values

The possible encoding values for the ENUM_ENCODING attribute follow.

- '0'—bit value '0'
- '1'—bit value '1'
- 'D'—don't-care (can be either '0' or '1')

  To use don't care information, see the "Don't Care Inference" section of the "Writing Circuit Descriptions" chapter

- '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 the "Three-State Inference" section of the "Register and Three-State Inference" chapter for more information.

Integer Types

The maximum range of a VHDL integer type is - (2^{31}-1) to 2^{31}-1 (-2_147_483_647 .. 2_147_483_647). Integer types are defined as subranges of this anonymous built-in type. Multi-digit numbers in VHDL can include underscores (_) to make them easier to read.

Foundation 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 2s-complement bit vectors.

The syntax of an integer type definition follows.

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.

An example of integer type definitions follows.

```
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
```

You cannot directly access the bits of an INTEGER or explicitly state the bit width of the type. For these reasons, Express provides overloaded functions for arithmetic. These functions are defined in the std_logic_signed and std_logic_unsigned packages, described in the "std_logic_arith Package" section of the "Foundation Express Packages" chapter.

Array Types

An array is an object that is a collection of elements of the same type. VHDL supports N-dimensional arrays, but Foundation 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 the following example.

```
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 follows.

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
- type_name is the type of each array element

An example of a constrained array type definition follows.

```
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 follows.

```
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
- element_type_name is the type of each array element

An example of an unconstrained array type definition and a declaration that uses it follows.

```
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 can recall 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

Foundation Express supports the following predefined VHDL attributes for use with arrays.

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

The above attributes return a value corresponding to part of an array's range. The following table shows the values of the array attributes for the variable MY_VECTOR in the example of an uncon-

strained array type definition from the previous “Unconstrained Array” section.

Table 3-1 Array Index Attributes

Attribute Expression	Value
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)

The following example shows the use of array attributes in a function that ORs together all elements of a given bit vector (declared in the example of an unconstrained array type definition in the previous section) and returns that value.

```
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: 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 any previously defined type, including another record type.

The following example shows a record type declaration (BYTE_AND_IX), three signals of that type, and some assignments.

```
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 the above example, 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;
  ```

The individual fields of a record type object 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).

Record Aggregates

Record aggregates (constants) have the same syntax as array aggregates (see the "Aggregates" section of the "Expressions" chapter). They can appear anywhere records appear.

The following line illustrates a named record aggregate in a description.

```
X <= (BYTE => "11110000", IX => 2);
```

The following line illustrates a positional record aggregate in a description.

```
X <= ("11110000", 2);
```

You can use the others construct in a named or positional record aggregate, just as you can in an array aggregate (see the "Aggregates" section of the "Expressions" chapter).

You can mix named and positional aggregates in a description, with the positional items listed first.

You cannot have a named item that refers to a field covered in the positional aggregate. The following four examples illustrate this caveat.

The following example shows a simple record type.

```
type rec is
   record
      a: integer;
      b: integer;
      c: integer;
      d: integer;
      e: integer;
   end record
end
```

The following example shows a named aggregate for the previous example.

```
(a => 1, b => 2, c => 0, d => 3, e => 0)
```

In a named aggregate, the items can appear in any order as shown in the following example.

```
(1, 2, d => 3, others => 0)
```

The previous example is equivalent to the second example or the following example of positional aggregate.

```
(1, 2, 0, 3, 0)
```

You can supply a set of choices in a description of a record aggregate, but a choice cannot be a range. See the following two examples.

The following example shows a record aggregate equivalent to the next example after it.

```
(b => 2, c => 2, d => 2, a => 1, e => 3)
```

The following example shows a record aggregate with a set of choices.

```
(b | c | d => 2, a => 1, e =>3)
```

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). You do not need this package for synthesis, therefore, Foundation Express does not support it.

The Foundation Express implementation of the STANDARD package is illustrated in the following example. This STANDARD package is a subset of the IEEE VHDL STANDARD package. Differences are described in the "Unsupported Data Types" section of this chapter.

```
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.

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. The following sections list and describe these unsupported data types.

"VHDL Construct Support" section of the "VHDL Constructs" chapter describes the level of Foundation Express support for each VHDL construct.

Physical Types

Foundation Express does not support physical types, such as units of measure (for example, ns).

Floating-Point Types

Foundation Express does not support floating point types, such as REAL.

Access Types

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

File Types

Foundation Express does not support file (disk file) types, such as a hardware file type RAM or ROM.

Express 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 "std_logic_arith Package" section of the "Foundation Express Packages" chapter.

Subtypes

A subtype is a subset of a previously defined type or subtype. A subtype definition can appear anywhere a type definition is allowed.

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

You can also use subtypes for resolved signals to associate a resolution function with the signal type. (See the "Subtype Declarations" section in the "Design Descriptions" chapter for more information.)

In the example of the Foundation Express STANDARD Package (in the "Predefined VHDL Data Types" section of this chapter), 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.

The following example shows some valid and invalid assignments between NATURAL and POSITIVE values.

```
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 follows.

```
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 the following.

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

The following example shows that all functions and attributes that operate on BIT_VECTOR also operate on MY_VECTOR.

```
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 4

Expressions

In VHDL, 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.

The following sections of this chapter discuss the individual components and use of expressions in a design description.

- “Overview”
- “Operators”
- “Operands”

Overview

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 used in any of the following ways.

- Assign them to variables or signals or use them as the initial values of constants
- Use them as operands to other operators
- Use them for the return value of functions
- Use them for the IN parameters in a subprogram call
- Assign them to the OUT parameters in a procedure body
- Use them to control the actions of statements such as 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 (Signed) Operators
- Multiplying Operators
- Miscellaneous Arithmetic Operators

Operands

- Computable Operands
- Literals
- Identifiers
- Indexed Names
- Slice Names
- Function Calls
- Qualified Expressions
- Type Conversions

Operators

A VHDL operator is characterized by the following.

- 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 the table below.

Table 4-1 Predefined 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.

The following example shows several expressions and their interpretations.

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

VHDL allows existing operators to be overloaded, that is, 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 the "Operator Overloading" section in the "Design Descriptions" chapter.

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 or 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.

The following example shows logical signal declarations and their logical operations.

```
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);
```

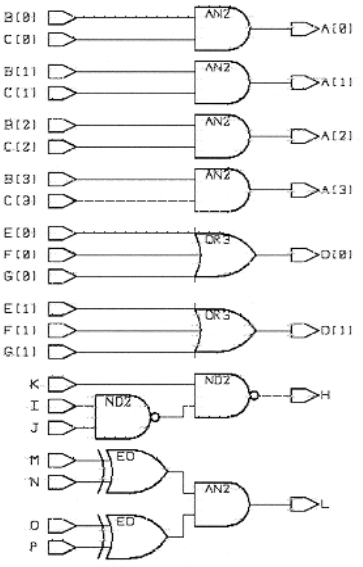


Figure 4-1 Design Schematic for Logical Operators

Normally, to use more than two operands in an expression, you must use parentheses to group the operands. An exception is that you can combine a sequence of AND, OR, XNOR, or XOR operators without parentheses, such as the following sequence that uses the same operator—AND.

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

However, a sequence that contains more than one of these operators requires parentheses to indicate which two operands are to be paired. In the following sequence, AND is the first operator, OR is the second.

```
A and B or C
```

Parentheses should be used in one of two ways, as shown in the following example.

```
(A and B) or C
```

or

```
A and (B or C)
```

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 (<, <=, >, and >=) 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 ("Enumeration Encoding" section of the "Data Types" chapter), the ordering operators compare your encoded value ordering, not the declaration ordering. Because this interpretation is specific to Foundation Express, a VHDL simulator continues to use the declaration's order of enumerated types.

Arrays are ordered alphabetically. Foundation Express determines the relative order of two array values 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 "10" is less than "101000."

Arrays are compared from left to right, regardless of their index ranges (to or downto).

The following example shows several expressions that evaluate to TRUE.

```
'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 std_logic_arith package (listed in the "Foundation Express Packages" chapter). The third line in the above example evaluates FALSE if the operands are of type UNSIGNED.

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

The following example shows some relational expressions. The resulting synthesized circuit follows the example.

```
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);
```

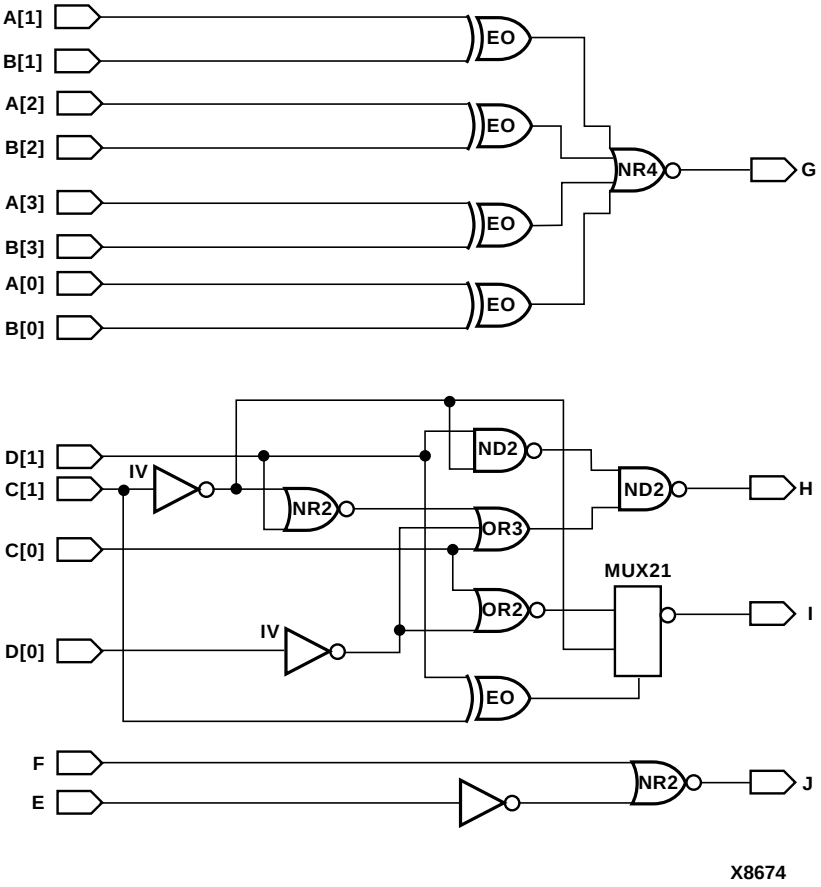


Figure 4-2 Circuit for Relational Operators

Adding Operators

Adding operators include arithmetic and concatenation operators.

The arithmetic operators + and - are predefined for all integer operands. These addition and subtraction operators perform conventional arithmetic. The following example uses the + operator.

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 the following examples. The schematic for the resulting circuits follow the examples.

```
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
```

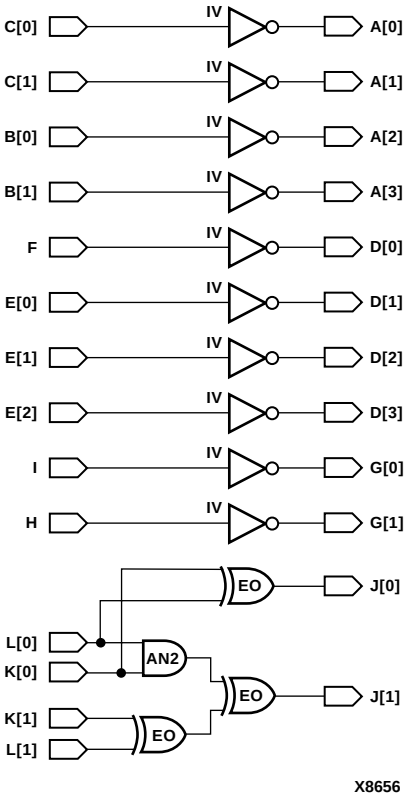


Figure 4-3 Circuits for Adding Operators

Unary (Signed) Operators

A unary operator has only one operand. Foundation Express predefines unary operators + and - for all integer types. The + operator has no effect. The - operator negates its operand as shown in the following example.

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

The following example shows how unary negation is synthesized. The resulting design follows the example.

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

```
A <= -B;
```

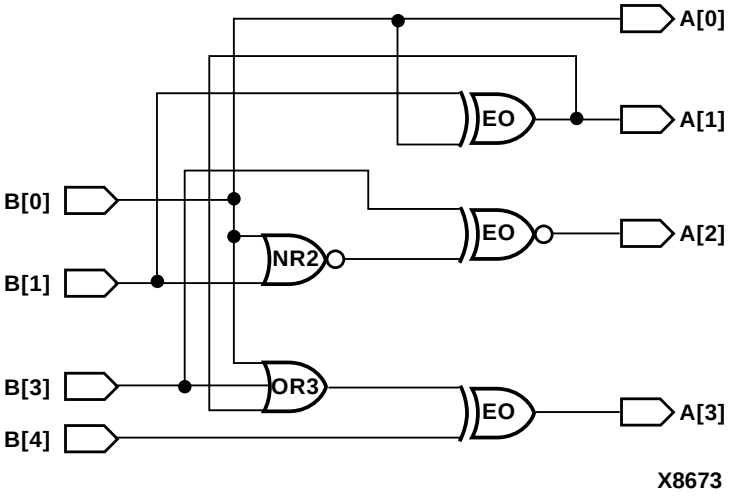


Figure 4-4 Design Illustrating Unary Negation

Multiplying Operators

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

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

- *—integer multiplication; no restrictions
- /—integer division; The right-hand operand must be a computable power of 2 and cannot be negative. (See the “Computable Operands” section of this chapter.) This operator is implemented as a bit shift.
- mod—modulus; same as /
- rem—remainder; same as /

The following example shows some uses of the multiplying operators whose right operands are all powers of 2. The resulting synthesized circuit design follows the example.

```
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;
```

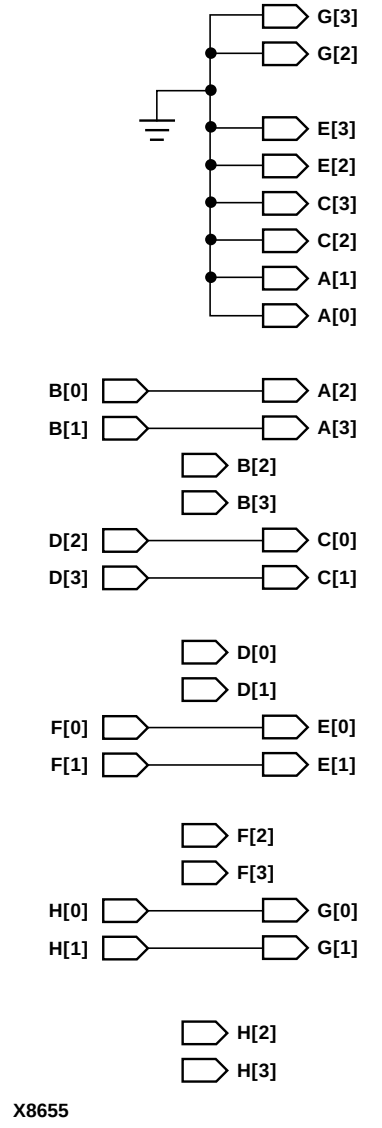


Figure 4-5 Design Illustrating Multiplying Operators

Miscellaneous Arithmetic Operators

Foundation Express predefines the absolute value (abs) and exponentiation (**) operators for all integer types. There is one restriction placed on the ** operator. When you are using ** exponentiation, the left operand must be the computable value 2 (see the “Computable Operands” section of this chapter).

The following example shows how these operators are used. The figure that illustrates the synthesized design follows the example.

```
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;
```

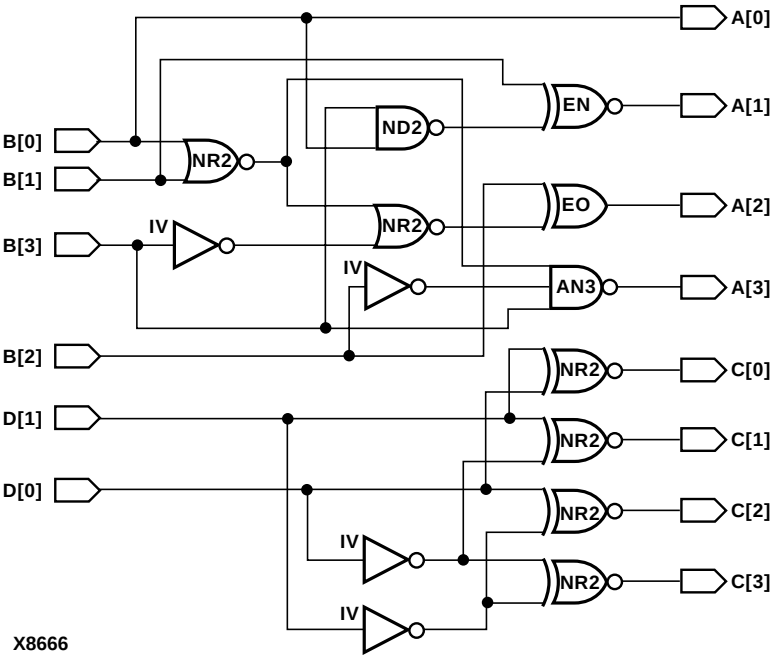


Figure 4-6 Design with Arithmetic Operators

Operands

Operands specify the data that the operator uses to compute its value. An operand returns 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 follow.

- Aggregates: my_array_type'(others => 1)
- Attributes: my_array'range
- Expressions: (A nand B)
- Function calls: LOOKUP_VAL(my_var_1, my_var_2)
- Identifiers: my_var, my_sig
- Indexed names: my_array(7
- Literals: '0', "101", 435, 16#FF3E#
- Qualified expressions: BIT_VECTOR'('1' & '0')
- Records and fields: my_record.a_field
- Slice names: my_array(7 to 11)
- Type conversions: THREE_STATE('0')

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

Operand Bit-Width

Foundation Express uses the bit-width of the largest operand to determine the bit-width needed to implement an operator in a circuit. For example, an INTEGER operand is 32 bits wide by default. An addition of two INTEGER operands causes Foundation 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, Foundation 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 Foundation Express. Computability is important because noncomputable expressions can require logic gates to determine their value.

Examples of computable operands follow.

- 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 when 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.

Examples of noncomputable operands follow.

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

- Variables assigned noncomputable values

The following example shows some definitions and declarations, followed by several 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
                         --   on iteration)
    end loop;

  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;
```

Aggregates

Aggregates create array literals by giving a value to each element of an instance of an array type. Aggregates can also be considered array literals, because they specify an array type and the value of each array element. The syntax follows.

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

type_name must be a constrained array type (as required by Foundation Express in the previous example), an element index, a sequence of indexes, or the others expression. Each expression provides a value for the chosen elements and must evaluate to a value of the element's type.

The following example shows an array type definition and an aggregate representing a literal of that array type. The two sets of assignments have the same result.

```
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 assignment
X(2) <= A nand B;
X(3) <= '1';
X(4) <= A or B;
```

You can specify an element's index by using either positional or named notation. With positional notation, each element receives the value of its expression in order, as shown in the example above.

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. 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.

The following example shows several aggregates representing the same value.

```
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. The following example shows two equivalent aggregates.

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

To use an aggregate as the target of an assignment statement, see the "Assignment Statements and Targets" section of the "Sequential Statements" chapter.

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 follows.

```
object'attribute
```

Foundation Express supports the following predefined VHDL attributes for use with arrays, as described in the "Array Types" section of the "Data Types" chapter.

- left
- right
- high
- low
- length

- range
- reverse_range

Foundation Express also supports the following predefined VHDL attributes to use with wait and if statements, as described in the "Register and Three-State Inference" chapter.

- event
- stable

In addition to supporting the predefined VHDL attributes listed above, Foundation Express has a defined set of synthesis-related attributes. You can include these Foundation Express-specific attributes in your VHDL design description to direct Foundation Express during optimization.

Expressions

Operands can themselves be expressions. You create expression operands by surrounding an expression with parentheses, such as (A nand B).

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 follows.

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

- function_name is the name of a defined function.
- The optional parameter_name is the name 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 if you put all positional expressions before named parameter expressions.

The following example shows a function declaration and several equivalent 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)
```

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.

Identifiers that contain special characters, begin with numbers, or have the same name as a keyword can be specified as an extended identifier. An extended identifier starts with a backslash character (\), followed by a sequence of characters, followed by another backslash character (\).

The following example shows some extended identifiers.

```
\a+b\        \3state\
\type\        \(a&b)|c\
```

The following example shows several kinds of identifiers and their usages. All identifiers appear in bold type.

```
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. The syntax of an indexed name follows.

identifier (expression)

identifier is the 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 the expression is computable (see the "Computable Operands" section of this chapter), the operand is synthesized directly. If the expression is not computable, a circuit is synthesized that extracts the specified element from the array.

The following example shows two indexed names—one computable and one not computable. The figure for the resulting synthesized circuit design follows the example.

```
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
```

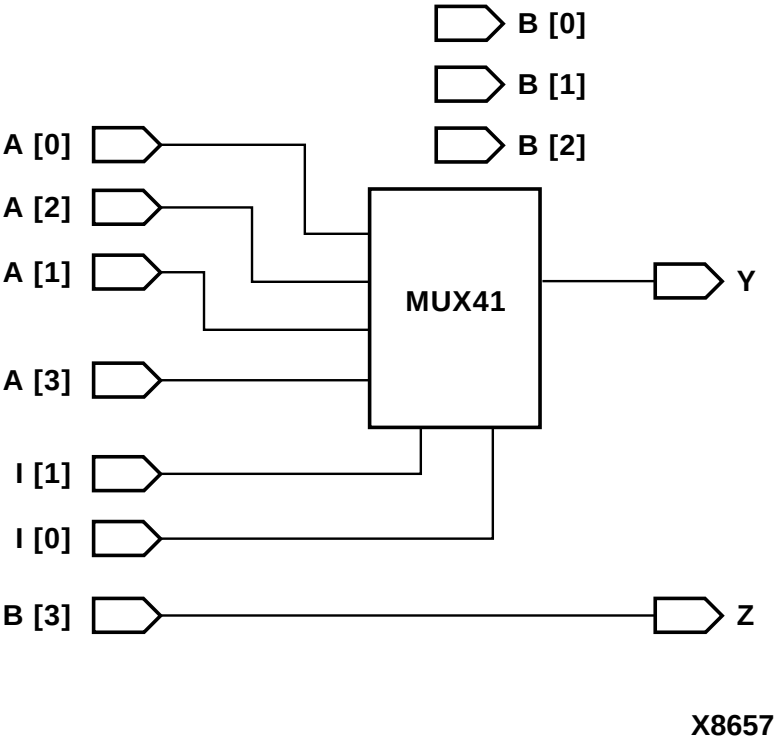


Figure 4-7 Design Illustrating Use of Indexed Names

You can also use indexed names as assignment targets; see the "Assignment Statements and Targets" section of the "Sequential Statements" chapter.

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 n underscores. The following example shows several different numeric literals, all representing the same value, which is 170.

```
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 are used both as values for operators and to define enumerated types, such as CHARACTER and BIT. See the “Enumeration Types” section of the “Data Types” chapter for the valid 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. The following example shows an enumeration type definition,

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

If two enumerated types use the same literals, those literals are overloaded. You must qualify overloaded enumeration literals when you use them in an expression, unless their type can be determined from context (See the “Qualified Expressions” section of this chapter.) See “Enumeration Types” section of the “Data Types” chapter for more information.

The example below defines two enumerated types and shows some enumeration literal values.

```
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 follow.

- Character strings which are sequences of characters in double quotation marks, 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 literal's type is a one-dimensional array of an enumerated type. Each of the characters in the string represents one element of the array. The following example shows character string literals.

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

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

Bit strings, like based numeric literals, are composed of a base specific 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. The following example shows three bit string literals that represent the value AAA.

```
X"AAA"
B"1010_1010_1010"
O"5252″
```

Qualified Expressions

Qualified expressions state the type of an ambiguous operand. You cannot use qualified expressions for type conversion. (See the “Type Conversions” section of this chapter.)

The syntax of a qualified expression follows.

type_name’(*expression*)

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

Note: Foundation Express requires a single quotation mark (tick) between type_name and (expression). If the single quotation mark is omitted, the construction is interpreted as a type conversion (described in the next section).

The following example shows a qualified expression that resolves an overloaded function by qualifying the type of a decimal literal parameter.

```
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
```

The following example shows how qualified expressions resolve ambiguities in 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
```

Records and Fields

Records are composed of named fields of any type. For more information, see the "Record Types" section of the "Data Types" chapter.

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

record_name.field_name

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

The example below shows a record type definition and 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 any type, including an array, record, or aggregate type. Refer to a field element by using that type's notation as in the following example.

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

Slice Names

Slice names identify a sequence of elements in an array variable or signal. The syntax follows.

identifier (expression direction expression)

identifier is the name of 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 the "Computable Operands" section of this chapter.

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

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

The following example uses slices to assign an 8-bit input to an 8-bit output, exchanging the lower and upper 4 bits. The figure for the resulting synthesized circuit design follows the example. Slices are also used as assignment targets. This usage is described in "Assignment Statements and Targets" section of the "Sequential Statements" chapter.

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

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

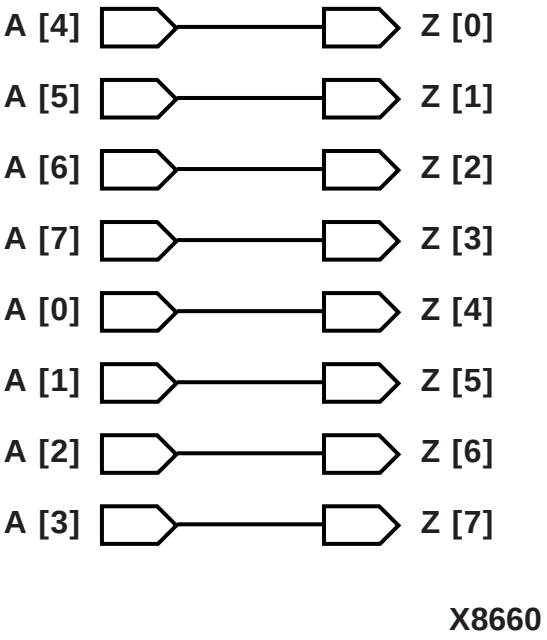


Figure 4-8 Design Illustrating Use of Slices

Limitations on Null Slices

Foundation Express does not support null slices, which are indicated by the following.

- A null range, such as (4 to 3)
- A range with the wrong direction, such as UP_VAR(3 downto 2) when the declared range of UP_VAR is ascending

The following example shows three null slices and one noncomputable slice.

```
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.

Type Conversions

Type conversions change an expression's type. The syntax of a type conversion follows.

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. The following conditions apply to type conversions.

- 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.

The following example shows some type definitions and associated signal declarations, followed by valid and invalid 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 5

Sequential Statements

Foundation Express interprets sequential statements, such as A:= 3, in the order in which they appear in code. VHDL sequential statements can appear only in processes and subprograms.

This chapter describes and illustrates the different types of sequential statements in the following sections.

- “Assignment Statements and Targets”
- “Variable Assignment Statements”
- “Signal Assignment Statements”
- “if Statements”
- “case Statements”
- “loop Statements”
- “next Statements”
- “exit Statements”
- “Subprograms”
- “return Statements”
- “wait Statements”
- “null Statements”

Assignment Statements and Targets

Use an assignment statement to assign a value to a variable or signal. The syntax follows.

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 the “Expressions” section of the “Expressions” chapter for more information.

There are 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)

The difference in syntax between variable assignments and signal assignments follows.

- Variables use the := operator.

  Variables are local to a process or subprogram, and their assignments take effect immediately.

- Signals use the <= operator.

  Signals need to be global in 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 the “Signal Assignment” section of this chapter.

Simple Name Targets

The syntax for an assignment to a simple name (identifier) target follows.

```
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.

The following example shows assignments to 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
                     -- bit value "1100"
```

Indexed Name Targets

The syntax for an assignment to an indexed name (identifier) target follows.

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 the "Expressions" chapter), but more hardware is synthesized if it is not.

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

In the following example, the elements for array variable A are assigned values as indexed names.

```
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.
```

The example below shows two indexed name targets. One of the targets is computable, and the other is not. The figure following the example illustrates the corresponding design.

```
entity example5 3 is
   port (
      signal A, B: BIT_VECTOR(0 to 3);
      signal I: INTEGER range 0 to 3;
      signal Y, Z: BIT;
      );
end example5 3;

architecture behave of example5 3 is
```

```
begin
process (I, Y, Z)
begin

A    <= “0000";
B    <= “0000";
A(I) <= Y;  -- Noncomputable index expression
B(3) <= Z;  -- Computable index expression

end process;
end behave
```

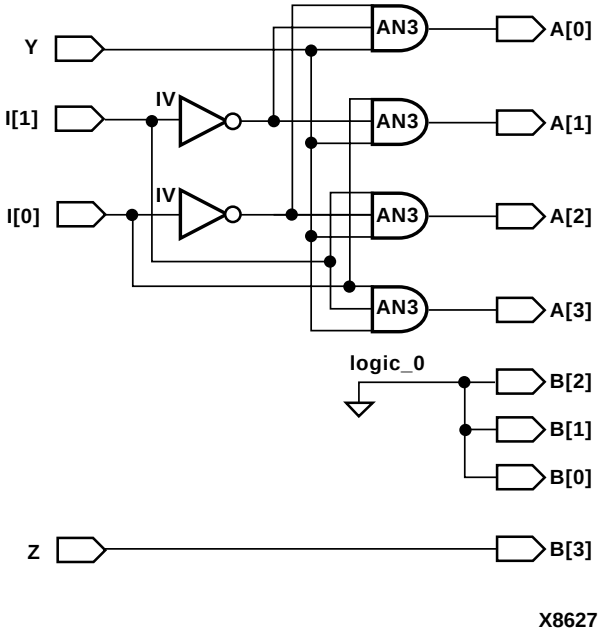


Figure 5-1 Design Illustrating Indexed Name Targets

Slice Targets

The syntax for an assignment to a slice target follows.

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 the “Expressions” chapter) and must lie within the bounds of the array. The direction must match the identifier array type’s direction, either to or downto.

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

In the following example, array variables A and B are assigned the same value.

```
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 follows.

identifier.*field_name*

identifier is the name of a record type signal or variable. 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 any type, including an array, record, or aggregate type.

The following example assigns values to the fields of record variables A and B.

```
type REC is
    record
        NUM_FIELD:   INTEGER range -16 to 15;
        ARRAY_FIELD: BIT_VECTOR(3 to 0);
    end record;

variable A, B: REC;

-- Target        Expression;
   A.NUM_FIELD   := -12;
     -- Assigns -12 to record A’s field NUM_FIELD
   A.ARRAY_FIELD := “0011";
     -- Assigns “0011" to record A’s field ARRAY_FIELD
   A.ARRAY_FIELD(3) := ’1’;
     -- Assigns ’1’ to the most significant bit of
     -- record A’s field ARRAY_FIELD
   B             := A;
-- Assigns values of record A to corresponding fields of B
```

For more information, see the “Record Types” section of the “Data Types” chapter.

Aggregate Targets

The syntax for an assignment to an aggregate target follows.

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

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

aggregate assignment assigns the array_expression element values to one or more variable or signal identifiers.

Each (optional) choice 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 element type. An identifier can be an array type.

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, as in the following example.

```
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
```

Variable Assignment Statements

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

target := *expression*;

target names the variables that receive the value of expression. See the "Assignment Statements and Targets" section of this chapter for a description of variable assignment targets.

Expression determines the assigned value; its type must be compatible with target. See the "Expressions" chapter for further information.

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

The following example shows the different effects of variable and signal 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;
```

Signal Assignment Statements

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

target := expression;

target names the signals that receive the value of expression. See the "Assignment Statements and Targets" section of this chapter for a description of variable assignment targets.

expression determines the assigned value; its type must be compatible with target. For more information about expressions, see the "Expressions" chapter.

Signals and variables behave in different ways when they receive assigned values. The differences lie in the way the two kinds of assignments take effect and how that influences the value Foundation Express reads from either variables or signals.

Variable Assignment

When a variable is assigned a value, the assignment changes the value of the variable from that point on. That value is kept until the variable is assigned a different value.

Signal Assignment

When a signal receives an assigned 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 the signal.

- 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. The circuit might be invalid, wired AND, wired OR, or a three-state bus. See the "Concurrent Statements" chapter for more information.

The following example shows the different effects of variable and signal 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 Statements

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

```
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 Conditions

An if statement evaluates each condition in order. Only the first true condition causes the execution of the if statement's branch statements. The remainder of the if statement is skipped.

If none of the conditions is true and the else clause is present, those statements are executed. If none of the conditions is true and no else clause is present, none of the statements is executed.

The following example shows an if statement. The figure following the example illustrates the corresponding circuit.

```
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;
```

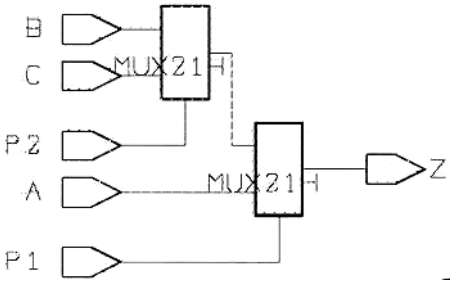


Figure 5-2 Schematic Design for if Statement

Using the if Statement to Infer 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 Foundation Express to infer registers or latches, as described in the "Register and Three-State Inference" chapter.

case Statements

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

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

expression must evaluate to an INTEGER, an enumerated type, or an array of enumerated types, such as BIT_VECTOR. Each of the choices must be in the following 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's type. The others choice, if present, matches expression only if no other choices match.

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

The following restrictions are placed on choices.

- No two choices can overlap.
- If an others choice is not present, all possible values of expression must be covered by the set of choices.

Using Different Expression Types

The following example shows a case statement that selects one of four signal assignment statements by using an enumerated expression type. The figure that follows the example illustrates the corresponding design with binary encoding specified.

```
library IEEE;
use IEEE.STD LOGIC 1164.all;

package case enum is
type ENUM is (PICK_A, PICK_B, PICK_C, PICK_D);
end case enum;

library work;
use work.case enum.all;

entity example5 9 is
   port (
      signal A, B, C, D: in BIT;
      signal VALUE: ENUM;
      signal Z:  out BIT;
   );
end example5 9;

architecture behave of example5 9 is

begin
process (VALUE)
begin
   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;
end process;
end behave;
```

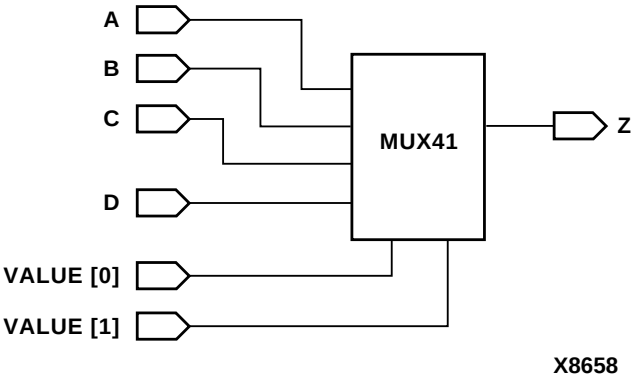


Figure 5-3 Circuit for case Statement with an Enumerated Type

The following example 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. The resulting circuit design is shown in the figure following the example.

```
entity example5_10 is
   port (
      signal VALUE: in INTEGER range 0 to 15;
      signal Z1, Z2, Z3, Z4:  out BIT
      );
end example5_10;
architecture behave of example5_10 is
begin
   process (VALUE)
begin
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;
        end process;
        end behave;
```

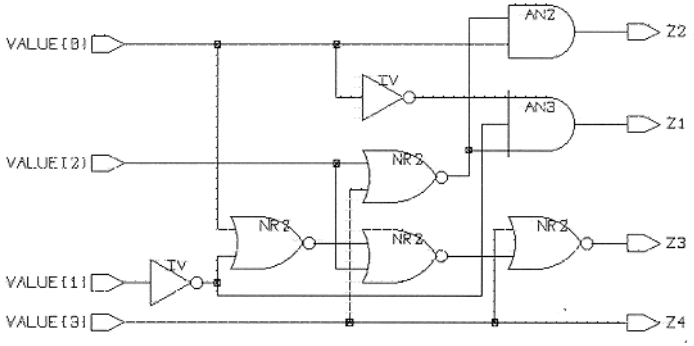


Figure 5-4 Circuit for case Statement with Integers

Invalid case Statements

The following example shows invalid case statements with explanatory comments.

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

case VALUE is   -- Must have at least one when clause
end case;

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

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

loop Statements

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

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

- label, which is optional, names the loop and is useful for building nested loops.
- iteration_scheme: There are three types of iteration_scheme; loop, while...loop, and for...loop. They are described in the next three sections.

The next and exit statements are sequential statements used only within loops.

- next statement skips the remainder of the current loop and continues with the next loop iteration.
- exit statement skips the remainder of the current loop and continues with the next statement after the exited loop.

  See the “next Statements” section and the “exit Statements” section of this chapter.

Basic loop Statement

The basic loop statement has no iteration scheme. Foundation Express executes enclosed statements repeatedly until it encounters an exit or next statement. The syntax statement follows.

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

- loop: The label, which is optional, names this loop.
- sequential_statement can be any statement described in this chapter. Two sequential statements are used only with loops.
- next statement skips the remainder of the current loop and continues with the next loop iteration.

- exit statement skips the remainder of the current loop and continues with the next statement after the exited loop.

See the “next Statements” section and “exit Statements” section of this chapter.

Note: Noncomputable loops (loop and while...loop statements) must have at least one wait statement in each enclosed logic branch. Otherwise, a combinatorial feedback loop is created. See the “wait Statements” section of this chapter for more information. Conversely, computable loops (for...loop statements) must not contain wait statements. Otherwise, a race condition may result.

while...loop Statements

The while...loop statement has a Boolean iteration scheme. If the iteration condition evaluates true, Foundation Express executes the enclosed statements once. The iteration condition is then reevaluated. As long as the iteration condition remains TRUE, the loop is repeatedly executed. When the iteration condition evaluates FALSE, the loop is skipped and execution continues with the next loop iteration. The syntax for a while...loop statement follows.

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

- label, which is optional, 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.
- next statement skips the remainder of the current loop and continues with the next loop iteration.
- exit statement skips the remainder of the current loop and continues with the next statement after the exited loop.

See the “next Statements” section and the “exit Statements” section of this chapter.

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

for...loop Statements

The for...loop statement has an integer iteration scheme. The integer range determines the number of repetitions The syntax for a for...loop statement follows.

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

- label, which is optional, names this loop.
- 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.

- range must be a computable integer range in either of the following two forms.

  integer_expression `to` *integer_expression*

  integer_expression `downto` *integer_expression*

  - integer_expression evaluates to an integer. For more informations, see the “Expressions” chapter.
  - sequential_statement can be any statement described in this chapter. Two sequential statements are used only with loops.
  - next statement skips the remainder of the current loop and continues with the next loop iteration.
  - exit statement skips the remainder of the current loop and continues with the next statement after the exited loop.

See the “next Statements” section and “exit Statements” section of this chapter.

Note: Computable loops (for...loop statements) must not contain wait statements. Otherwise, a race condition may result.

Steps in the Execution of a for...loop Statement

A for...loop statement executes as follows.

1. A new integer variable, which is local to the loop, is declared with the name identifier.
2. The identifier receives the first value of range, and the sequence of statements executes once.
3. The identifier receives the next value of range, and the sequence of statements executes once more.
4. Step 3 is repeated until identifier receives the last value in range. The sequence of statements then executes for the last time. Execution continues with the statement following the end loop. The loop is then inaccessible.

The following example shows two equivalent code fragments. The resulting circuit design is shown in the figure following the example.

```
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 statements
A(1) <= B(1);
A(2) <= B(2);
A(3) <= B(3);
```

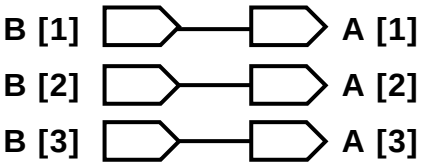


Figure 5-5 Circuit for for...loop Statement with Equivalent Fragments

for...loop Statements and Arrays

You can use a loop statement to operate on all elements of an array without explicitly depending on the size of the array. The following example shows how to use the VHDL array attribute 'range to invert each element of bit vector A. A figure of the resulting circuit follows the example. Unconstrained arrays and array attributes are described in "Array Types" section of the "Data Types" chapter.

```
entity example5_13 is
   port(
      A: out BIT_VECTOR(1 to 10);
      B: in BIT_VECTOR(1 to 10)
      );
end example5_13;

architecture behave of example5_13 is
begin
   process (B)
begin

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

end process;
end behave;
```

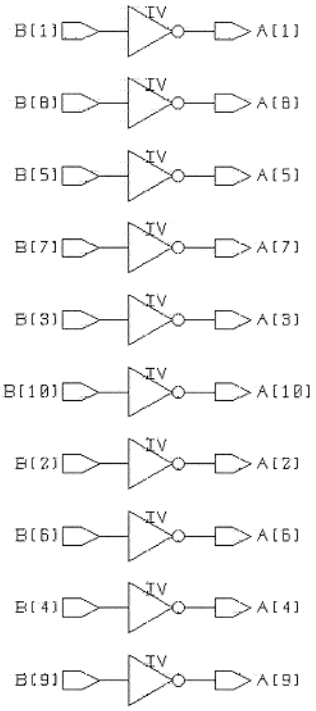


Figure 5-6 Circuit for for...loop Statement Operating on an Entire Array

next Statements

The next statement skips execution to the next iteration of an enclosing loop statement, called label in the syntax, as follows.

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

- label: 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.
- when is an optional clause that executes its next statement when its condition (a Boolean expression) evaluates TRUE.

The following example uses the next statement to copy bits conditionally from bit vector B to bit vector A only when the next condition

evaluates to TRUE. The corresponding design is shown in the figure following the example.

```
entity example5_14 is
   port(
      signal B, COPY_ENABLE: in BIT_VECTOR (1 to 8);
      signal A: out BIT_VECTOR (1 to 8)
      );
end example5_14;

architecture behave of example5_14 is

begin
process (B, Copy_ENABLE)
begin

A <= “00000000";

for I in 1 to 8 loop
  next when COPY_ENABLE(I) = ’0’;
  A(I) <= B(I);
end loop;

end process;
end behave;
```

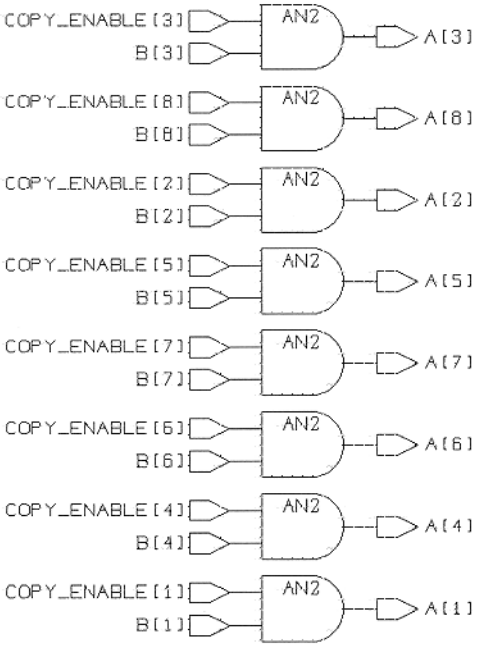


Figure 5-7 Circuit Design for next Statement

The example below shows the use of nested next statements in named loops. This example processes in the following manner.

- 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 manner until it is completed.

```
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 Statements

The exit statement completes execution of an enclosing loop statement, called label in the syntax. The completion is conditional if the statement includes a condition, such as the when condition in the following syntax.

exit [*label*] [**when** *condition*] ;

- label: An exit statement with no label terminates the innermost enclosing loop. When you specify a loop label, the current iteration of than named loop is terminated, as shown in the previous example of a named next statement.
- when is an optional clause that executes its next statement when its condition (a Boolean expression) evaluates TRUE.

Note: The exit and next statements have identical syntax, and they both skip the remainder of the enclosing (or named) loop. The difference between the two statements is that exit terminates its loop and, then, continues with the next loop iteration (if any).

The example below compares two bit vectors. An exit statement exits the comparison loop when a difference is found. The corresponding circuit design is shown in the figure following this example.

```
entity example5 16 is
   port(
      signal A, B: in BIT_VECTOR(1 downto 0);
      signal A_LESS_THAN_B: out Boolean;
      );
end example5 16;

architecture behave of example5 16 is

begin
process (A, B)
begin

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;
  end process;
  end behave;
```

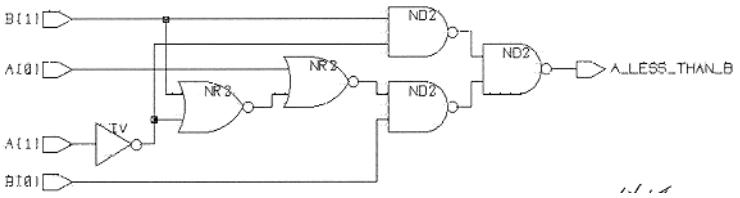


Figure 5-8 Circuit Design for Comparator Using the exit Statement

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 the "Combinatorial Versus Sequential Processes" section of this chapter) or concurrently (as described in the "Concurrent Statements" chapter).

Subprogram Always a Combinatorial Circuit

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

Subprogram Declaration and Body

Subprograms, like packages, have declarations and bodies. A subprogram declaration specifies its name, parameters, and return value (for

functions). The subprogram body then implements the operation you want.

Often, a package contains only type and subprogram declarations for other packages to use. 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.

Foundation Express implements procedure and function calls with combinatorial logic, unless you use the map_to_entity compiler directive (see the "Procedures and Functions as Design Components" section of this chapter). Foundation Express does not allow you to infer sequential devices, such as latches or flip-flops, in subprograms.

The following example shows a package containing some procedure and function declarations and bodies. The example itself cannot be synthesized; it just creates a template. Designs that instantiate procedure P, however, compile normally.

```
package EXAMPLE is
  procedure P (A: in INTEGER; B: inout INTEGER);
    -- Declaration of procedure P

  function INVERT (A: BIT) return BIT;
    -- Declaration of function INVERT
end EXAMPLE;

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

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

```
    end;
end EXAMPLE;
```

For more information about subprograms, see the “Subprograms” section of the “Design Descriptions” chapter.

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 receives 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 the following.

- read from (mode in)
- written to (mode out)
- both read from and written to (mode inout).

Actual parameters that use mode out and mode inout must be variables or signals and include indexed names (A(1)) and slices (A(1 to 3)). They cannot be constants or expressions.

Procedures and functions are two kinds of subprograms.

- Procedures

  Can have multiple parameters that use modes in, inout, and out Procedures do not 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.

- Functions

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

Use functions 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 follows.

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

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. A notation can mix named and positional notation, but positional parameters must precede named parameters.

A procedure call occurs in three steps.

1. Foundation Express assigns the values of the in and inout actual parameters to their associated formal parameters.
2. The procedure executes.
3. Foundation Express assigns the values of the inout and out formal parameters are assigned to the actual parameters.

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

The following example 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. The figure following the example illustrates the corresponding design.

```
library IEEE;
use IEEE.std_logic_1164.all;

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;
```

```
library IEEE;
use IEEEE.std_logic_1164.all;
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 1st and 2nd
         SWAP(MY_ARRAY, 2, 3); -- Swap 2nd and 3rd
         SWAP(MY_ARRAY, 1, 2); -- Swap 1st and 2nd
                               -- again
         OUT_ARRAY <= MY_ARRAY; -- Write result to
                                -- output
   end process;
end EXAMPLE;
```

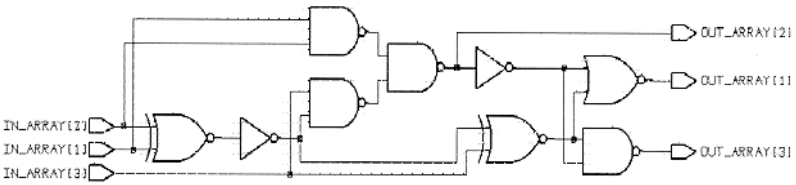


Figure 5-9 Circuit Design for Procedure Call to Sort an Array

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 follows.

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

- function_name is the name of a defined function.
- parameter_name, which is optional, is the name of a formal parameter 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 parameter values in positional or named notation, as you can 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 is 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 if you put all positional expressions before named parameter expressions.

The example below shows a simple function definition and two calls to that function.

```
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;
```

return Statements

The return statement terminates a subprogram. A function definition requires a return statement. In a procedure definition, a return statement is optional. The syntax follows.

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

- expression provides the return value of the function. Every function must have at least one return statement. The expression type must match the declared function type. Only one return statement is reached by a given function call.
- procedure can have one or more return statements, but no expression. A return statement, if present, is the last statement executed in a procedure.

In the following example, the function OPERATE returns either the AND logical operator or the OR logical parameters of its parameters A and B. The return depends on the value of the parameter OPERATION. The corresponding circuit design is shown in the figure following the example.

```
package test is
   function OPERATE (A, B, OPERATION: BIT) return BIT;
end test;

package body test is

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;
end test;

library IEEE;
use IEEE.std_logic_1164.all;
use WORK.test.all;

entity example5_20 is
   port(
      signal A, B, OPERATION: in BIT;
      signal RETURNED_VALUE: out BIT
```

```
      );
end example5_20;

architecture behave of example5_20 is

begin

RETURNED_VALUE <= OPERATE(A, B, OPERATION);
end behave;
```

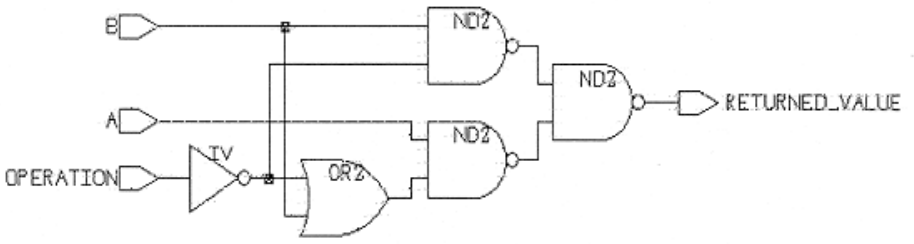


Figure 5-10 Circuit for Using Multiple return Statements

Procedures and Functions as Design Components

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 Foundation Express directive map_to_entity, which directs Foundation 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.

When you add a map_to_entity directive to a subprogram definition, Foundation Express assumes the existence of an entity with the identified name and the same interface. Foundation 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.The following example shows how to insert these directives.

```
   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 Foundation Express encounters the map_to_entity directive, it parses but ignores the contents of the subprogram definition. Use --- pragma synthesis_off and -- pragma synthesis_on to hide simulation-specific constructs in a map_to_entity subprogram (see “Translation Stop and Start Pragma Directives” section of the “Foundation Express Directives” chapter for more information about synthesis_off and synthesis_on).

The matching entity (entity_name) does not need to be written in VHDL. It can be in any format that Foundation Express supports.

Note: 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 with Component Implication Directives

The following example shows a function that uses component implication directives. The corresponding circuit design follows the example.

```
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 ARCH;

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 ARCH;
```

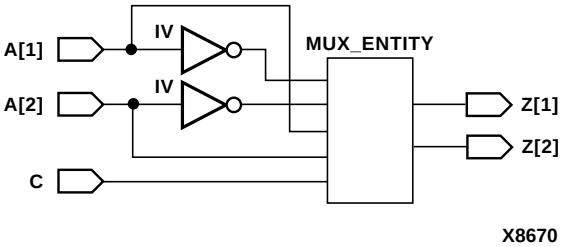


Figure 5-11 Circuit for Using Component Implication Directives on a Function

Example without Component Implication Directives

The following example shows the same design as the previous example but without the creation of an entity for the function. The component implication directives have been removed. The corresponding circuit design is shown in the figure following the example.

```
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 ARCH;
```

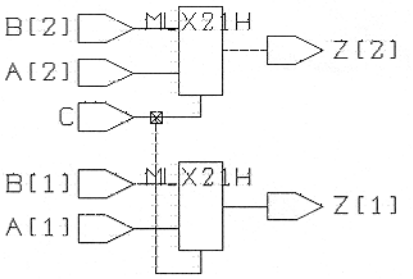


Figure 5-12 Circuit Design without Component Implication Directives

wait Statements

A wait statement suspends a process until Foundation Express detects a positive-going edge or negative-going edge on a signal. The syntax follows.

```
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 the "Data Types" chapter). The 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: Three forms of the wait statement (a subset of IEEE VHDL), shown in the previous syntax and in the following example, are specific to the current implementation of Foundation Express.

Inferring Synchronous Logic

A wait statement implies synchronous logic, where signal is usually a clock signal. The "Combinatorial Versus Sequential Processes" section of this chapter describes how Foundation Express infers and implements this logic.

The following example shows three equivalent wait statements (all positive-edge triggered).

```
wait until CLK = '1';
wait until CLK' event and CL = '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.

The following example shows a wait statement that suspends a process until the next positive edge (a 0-to-1 transition) on signal CLK.

```
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. For more information, see the "process Statements" section of the "Concurrent Statements" chapter.

The following example shows how a wait statement is used to describe a circuit where a value is incremented on each positive clock edge.

```
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;
```

The following example shows how multiple wait statements describe a multicycle circuit. The circuit provides an average value of its input A over four clock cycles.

```
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;
```

The following example shows two equivalent descriptions. The first description uses implicit state logic, and the second uses explicit state logic.

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

```
-- Explicit State Logic
type STATE_TYPE is (SO, S1);
variable STATE : STATE_TYPE;
...
process
begin
  wait until CLK'event and CLK = '1';
  case STATE is
    when S0 =>
      if (CONDITION) then
         X <= A;
         STATE := S0;
      else
         STATE := S1;
      end if;
    when S1 =>
```

```
        STATE := S0;
    end case;
end process;
```

Note: You can use wait statements 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.

The following example shows how to describe a circuit with synchronous reset using wait statements in an infinite loop. Foundation Express checks the reset signal immediately after each wait statement. The assignment statements in the following example (X <= A; and Y <= B;) represent the sequential statements used to implement the circuit.

```
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;
```

The example below shows two invalid uses of wait statements that are specific to Foundation Express.

```
...
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 1 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;
```

Combinatorial Versus Sequential Processes

If a process has no wait statements, the process is synthesized with combinatorial logic. The computations the process performs react immediately to changes in input signals.

If a process uses one or more wait statements, it is synthesized with sequential logic. The process performs computations 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 the "Signal Assignment Statements" section of this chapter.

- State vector values, where the state vector can be implicit or explicit (as in the example of wait statements and state logic).

- Variables that might be read before they are set.

Note: Like the wait statement, some uses of the if statement can also imply synchronous logic, causing Foundation Express to infer registers or latches. These methods are described in the "Register and Three-State Inference" chapter.

The following example 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.

The corresponding circuit design is shown in the figure following the example.

```
entity example5 30 is
   port(
   signal CLOCK: in BIT;
   signal SET_PARITY: in Boolean;
   signal PARITY_OK: out BOOLEAN;
   signal NEW_CORRECT_PARITY: in BIT;
   signal DATA:in BIT_VECTOR(0 to 3);
   );
end example5 30;

architecture behave of example5 30 is

begin
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;
end behave;
```

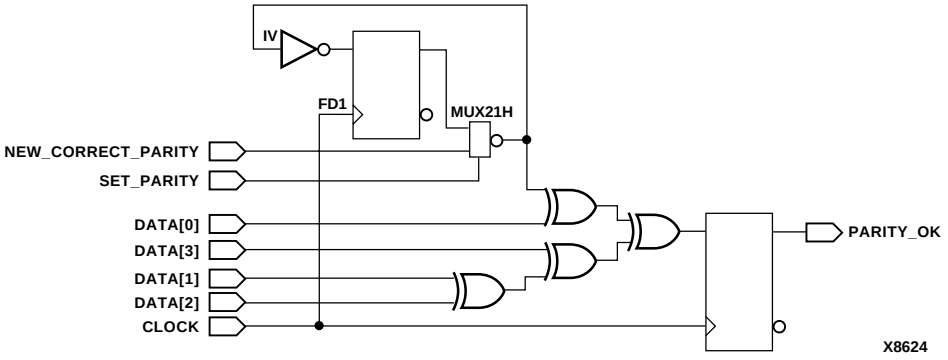


Figure 5-13 Circuit for Parity Tester Using the wait Statement

The previous figure shows two flip-flops are in the synthesized schematic from the example of a parity tester using the wait statement. 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 Statements

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

```
null;
```

The following example shows a typical use of the null statement. The corresponding circuit design is shown in the figure following the example.

```
entity example5 31 is
   port(
      signal CONTROL: INTEGER range 0 to 7;
      signal A: in BIT;
      signal Z: out BIT
      );
end example5 31;
```

```
architecture behave of example_5_31 is

begin

process (CONTROL, A)
begin

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;
end process;
end behave;
```

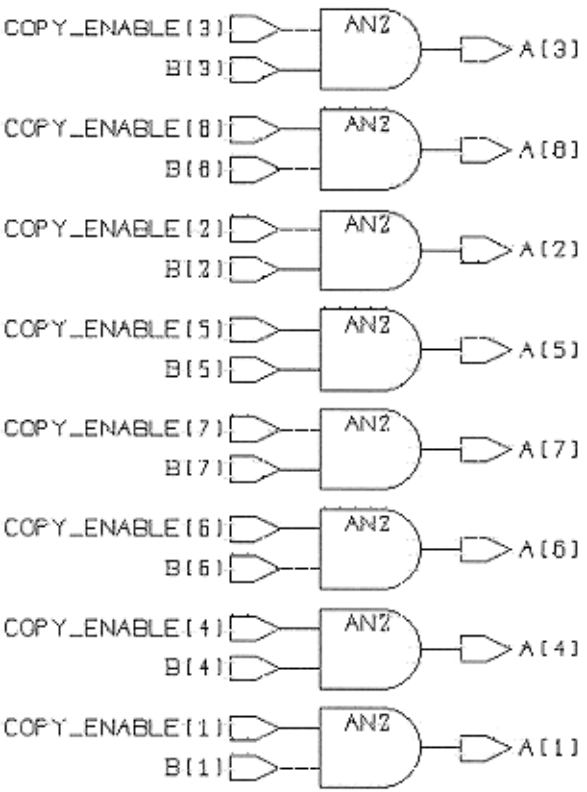


Figure 5-14 Circuit for null Statement

Chapter 6

Concurrent Statements

A VHDL architecture construct comprises a set of interconnected concurrent statements, such as blocks or processes, that describe an overall design in terms of behavior or structure. Concurrent statements in a design execute simultaneously, unlike sequential statement, which execute one after another.

This chapter describes concurrent statements and their function. The chapter is divided into the following sections.

- “Overview”
- “process Statements”
- “block Statements”
- “Concurrent Versions of Sequential Statements”
- “Component Instantiation Statements”
- “Direct Instantiation”
- “generate Statements”

Overview

The two main concurrent statements are the following.

- Process statements
- Block statements

VHDL provides two concurrent versions of sequential statements.

- Concurrent procedure calls
- Concurrent signal assignments

The component instantiation statement references a previously defined hardware component.

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

process Statements

A process statement (which is concurrent) contains a set of sequential statements. Although all processes in a design execute concurrently, Foundation Express interprets the sequential statements within each process one at a time.

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

The syntax of a process statement follows.

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

- label, which is optional, names the process.
- sensitivity_list is a list of all signals (including ports) read by the process, in the following format.

  signal_name {, signal_name}

  The circuit Foundation Express synthesizes is sensitive to all signals read the process reads. To guarantee the same results from a VHDL simulator and the synthesized circuit, a process sensitivity list has to contain all signals whose changes require simulating the process again.

  Follow these guidelines when developing the sensitivity list.

  - 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.

Foundation 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.

- process_declarative_item declares subprograms, types, constants, and variables local to the process. These items can be any of the following items, all of which are discussed in the "Design Descriptions" chapter.
  - use clause
  - Subprogram declaration
  - Subprogram body
  - Type declaration
  - Subtype declaration
  - Constant declaration
  - Variable declaration

The sequence of statements in a process defines the behavior of the process. After executing all the statements in a process, Foundation Express executes them all again.

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 circuit synthesized for a process is either combinatorial (not clocked) or sequential (clocked). If a process includes a wait or if signal'event statement, its circuit contains sequential components. The wait and if statements are described in the "Sequential Statements" chapter.

Process statements provide a natural means for describing sequential algorithms. If the values computed in a process are inherently parallel, consider using concurrent signal assignment statements.

(See the "Concurrent Versions of Sequential Statements" section of this chapter).

Combinatorial Process Example

The following example shows a process (with no wait statements) that implements a simple modulo-10 counter. The process reads two signals, CLEAR and IN_COUNT, and drives one signal, OUT_COUNT.

If CLEAR is '1' or IN_COUNT is '9', then OUT_COUNT is set to'0.' Otherwise, OUT_COUNT is set to one more than IN_COUNT. The resulting circuit design is shown in the figure following the example.

```
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;
```

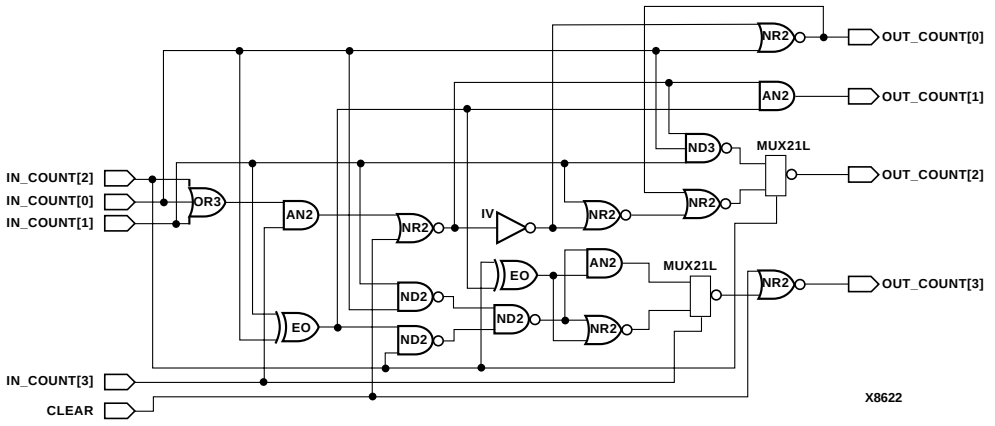


Figure 6-1 Modulo-10 Counter Process Design

Sequential Process Example

Another way to implement the counter in the previous example is to use a wait statement to contain the count value internally in the process.

The process in the following example implements the 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 ‘0.’
- Otherwise, Foundation Express increments the value of COUNT by one.
- The value of the variable COUNT is stored in four flip-flops, which Foundation Express generates because COUNT can be read before it is set. Thus, the value of COUNT has to be maintained from the previous clock cycle. For more information on using wait statements and count values, see “wait Statements” section of the “Sequential Statements” chapter.

The resulting circuit design is shown in the figure that follows the example.

```
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;
```

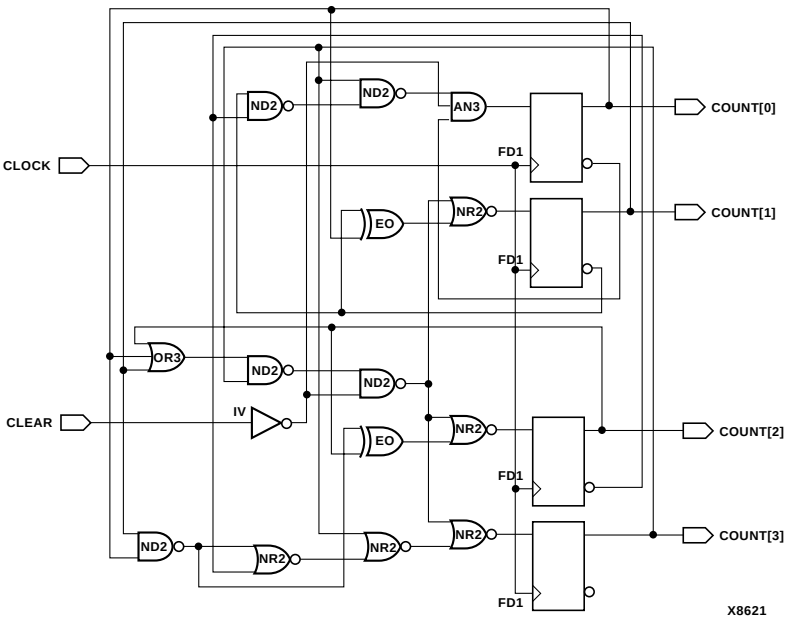


Figure 6-2 Modulo-10 Counter Process with wait Statement Design

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.

The following example shows two three-state buffers driving the same signal (SIG). The resulting circuit design is shown in the figure following the example. To learn to infer three-state devices in VHDL, see "Three-State Inference" section of the "Register and Three-State Inference" chapter.

```
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;
```

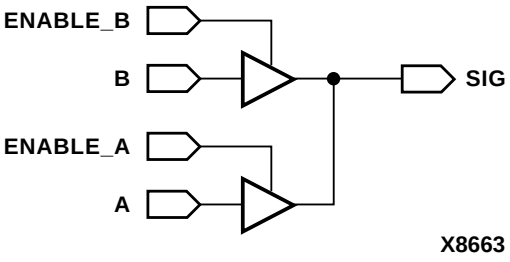


Figure 6-3 Two Three-State Buffers Driving the Same Signal

Bus resolution functions assign the value for a signal with multiple drivers. For more information, see "Resolution Functions" section of the "Design Descriptions" chapter.

block Statements

A block statement (which is concurrent) contains a set of concurrent statements. The order of the concurrent statements does not matter, because all statements are always executing.

Note: Foundation Express does not create a new level of design hierarchy from a block statement.

The syntax of a block statement follows.

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

- label, which is required, names the block.
- expression is the guard condition for the block. When this optional expression is present, Foundation Express evaluates the expression and creates a Boolean signal called GUARD.
- A block_declarative_item declares objects local to the block, which can be any of the following items.
  - use clause
  - subprogram declaration
  - subprogram body
  - type declaration
  - subtype declaration
  - constant declaration
  - signal declaration
  - component declaration

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 block's declaration overrides that of the parent.

Nested Blocks

The description in the following example uses nested blocks. The resulting circuit schematic is shown in the figure following the example.

```
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;
```

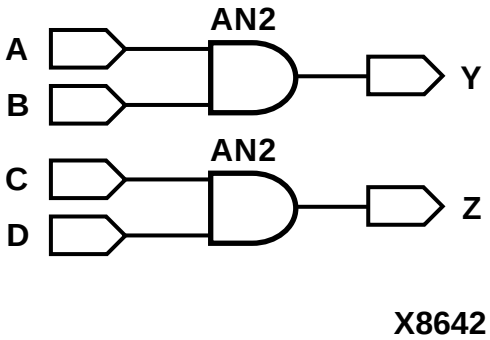


Figure 6-4 Schematic of Nested Blocks

Guarded Blocks

The description in the following example uses guarded blocks. In the example, z has the same value as a.

```
entity EG1 is
   port (a: in BIT; z: out BIT);
end;

architecture RTL of EG1 is
begin
   guarded_block: block (a = '1')
   begin
      z <= '1' when guard else '0';
```

```
   end block;
end RTL;
```

A concurrent assignment within a block statement can use the guarded keyword. In such a case, the guard expression conditions the signal assignment. The description in the following example produces a level-sensitive latch.

```
entity EG2 is
   port (d, g: in BIT; q: out BIT);
end;

architecture RTL of EG2 is
begin
   guarded_block: block (g = '1')
   begin
      q <= guarded d;
   end block;
end RTL;
```

Note: Do not use the 'event or 'stable attributes with the guard expression if you want to produce an edge-triggered latch using a guarded block. The presence of either attribute prevents it.

Concurrent Versions of Sequential Statements

This section describes concurrent versions of sequential statements in the following form.

- Concurrent Procedure Calls
- Concurrent Signal Assignments
  - Simple Concurrent Signal Assignments
  - Conditional Signal Assignments
  - Selected Signal Assignments

Concurrent Procedure Calls

A concurrent procedure call, which is used in an architecture construct or a block statement, is equivalent to a process with a single sequential procedure call in it (see the following example). The syntax is the same as that of a sequential procedure call.

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

The equivalent process reads all the in and inout parameters of the procedure. The following example shows a procedure declaration and a concurrent procedure call and its 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;
```

Foundation Express implements procedure and function calls with logic unless you use the map_to_entity compiler directive. (See the "Procedures and Functions as Design Components" section of the "Sequential Statements" chapter.)

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 have just 1 bit with value '1' and the rest of the bits with value '0' (as in the following example). Suppose you have several signals of varying widths that you want monitored at the same time (as the second example following). One approach is to write a procedure to detect the error in a BIT_VECTOR signal, and then make a concurrent call to that procedure for each signal.

The following example shows a procedure, CHECK, that determines whether a given bit vector has exactly one element with value '1.' If this is not the case, CHECK sets its out parameter ERROR to TRUE, as the example shows.

```
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;
        end if;
      end loop;

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

The following example shows the CHECK procedure called concurrently for four differently sized bit vector signals. The resulting circuit design is shown in the figure following the example.

```
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;
```

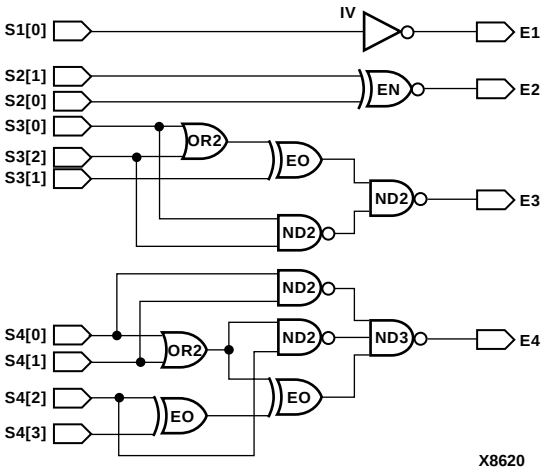


Figure 6-5 Concurrent CHECK Procedure Design

Concurrent Signal Assignments

A concurrent signal assignment is equivalent to a process containing a sequential assignment. Thus, each concurrent signal assignment defines a new driver for the assigned signal. This section discusses the three forms of concurrent signal assignment.

Simple Concurrent Signal Assignments

The syntax of the simplest form of the concurrent signal assignment follows.

target `<=` *expression*`;`

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

The following example shows the value of expressions A and B concurrently assigned to signal Z.

```
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 Assignments

The syntax of the conditional signal assignment follows.

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 Foundation Express executes a conditional signal assignment statement, it tests each condition in the order written.

- Foundation Express assigns to the target the expression of the first condition that evaluates to TRUE.
- If no condition evaluates to TRUE, Foundation Express assigns the final expression to the target.
- If two or more conditions are TRUE, Foundation Express assigns only the first one to the target.

The following example shows a conditional signal assignment. The target is the signal Z, which 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. The resulting design is shown in the figure following the example.

Note: The A assignment takes precedence over B, and B takes precedence over C, because the first TRUE condition controls the assignment.

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

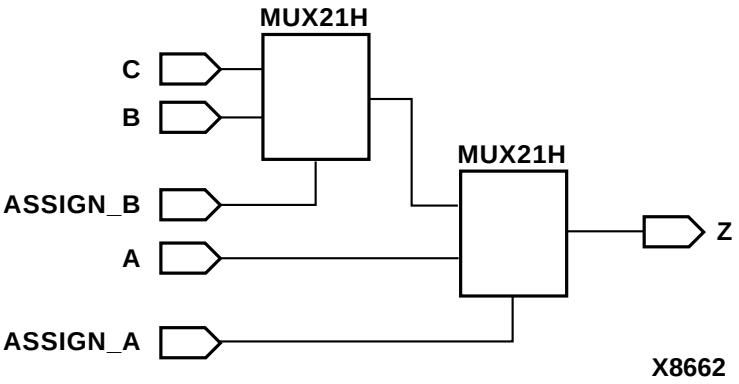


Figure 6-6 Conditional Signal Assignment Design

The following example shows a process equivalent to the example of the 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 Assignments

The syntax of the selected signal assignment follows.

```
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.

Each choice can be either of the following.

- A static expression (such as 3)
- A static range (such as 1 to 3).

The value of each choice the target signal receives has to match the value or values of choice_expression.

If the value of choice_expression is a static range, each value in the range must be covered by one choice in the expression.

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. You can use others as the final choice only if the value of choice_expression is a range.

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.

The following example shows target Z assigned from A, B, C, or D. The assignment depends on the current value of CONTROL. The resulting design is shown in the figure following the example.

```
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";
```

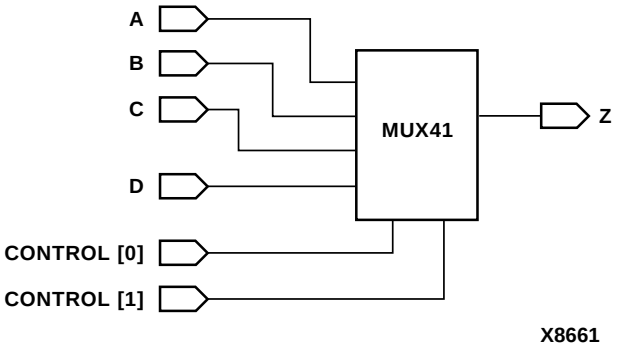


Figure 6-7 Circuit for Selected Signal Assignment

The following example shows a process equivalent to the previous example of selected signal assignment statement.

```
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 Instantiation Statements

The purpose of a component instantiation statement is to define a design hierarchy or build a netlist in VHDL by doing the following.

- Referencing a previously defined hardware component in the current design, at the current level of hierarchy

- Referencing components not defined in VHDL, such as the following.
  - Components from a technology library (FPGA vendor-specific)
  - Components defined in the Verilog hardware description language

The syntax follows.

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

- instance_name is the name of this instance of the component.
- component_name is the name of the component port map, which connects each port of this instance of component_name to a signal-valued expression in the current entity.
- port_name is the name of port.
- expression is the name of a signal, indexed name, slice name, or aggregate, to indicate the connection method for the component's ports.

  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 put 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.

The example below shows a component declaration (a 2-input NAND gate) followed by three equivalent component instantiation statements.

```
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
```

The following example 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 the example of component declaration and instantiations. The resulting circuit design is shown in the figure following the example.

```
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);
```

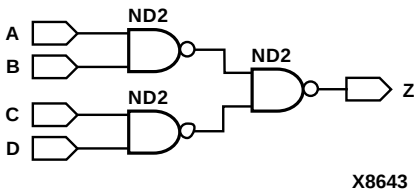


Figure 6-8 Simple Netlist Design

Direct Instantiation

A component instantiation statement

- Defines a subcomponent of the design entity in which it appears
- Associates signals or values with the ports of that subcomponent
- Associates values with generics of that subcomponent

The following two examples show the difference between a component instantiation statement and the more concise direct component instantiation statement.

```
ARCHITECTURE struct OF root IS
   COMPONENT leaf
      PORT (
         clk,data : in std_logic;
         Qout : out std_logic);
```

```
   END COMPONENT;
BEGIN
   u1 : leaf
      PORT MAP (
         clk => clk,
         data => d_in(0),
         Qout => q_out(0));
```

The following example shows how you can express the information in the previous example in a direct component instantiation statement.

```
ARCHITECTURE struct OF root IS
BEGIN
   u1 : entity work.leaf(rtl)
      port map (
         clk => clk,
         data => d_in(0),
         Qout => q_out(0));
```

generate Statements

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

- For...generate—the number of copies is determined by a discrete range.
- If...generate—zero or one copy is made, conditionally.

for...generate Statements

The syntax follows.

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

- label, which is required, names this statement and is useful for building nested generate statements.
- identifier is specific to the for...generate statement.
  - Identifier is not declared elsewhere. It is automatically declared by the generate statement itself and is local to the statement. A for...generate identifier overrides any other

identifier with the same name, but only within the for...generate statement.

- The value of identifier can be read only inside its for...generate statement (identifier does not exist outside the statement). You cannot assign a value to a for...generate identifier.
- The value of identifier cannot be assigned to any parameter whose mode is out or inout.

- range must be a computable integer range, in either of two forms.

  integer_expression `to` *integer_expression*
  integer_expression `downto` *integer_expression*

- integer_expression evaluates to an integer. Each concurrent_statement can be any of the statements described in this chapter, including other generate statements.

Steps in the Execution of a for...generate Statement

A for...generate statement executes as follows.

1. A new local integer variable is declared with the name identifier.
2. The identifier receives the first value of range, and each concurrent statement executes once.
3. The identifier receives the next value of range, and each concurrent statement executes once more.
4. Step 3 repeats until the identifier receives the last value in the range and each concurrent statement executes for the last time, Execution continues with the statement following end generate. The loop identifier is deleted.

The following example shows a code fragment that combines and interleaves two 4-bit arrays, A and B, into an 8-bit array, C. The resulting design is shown in the figure following the example.

```
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;
```

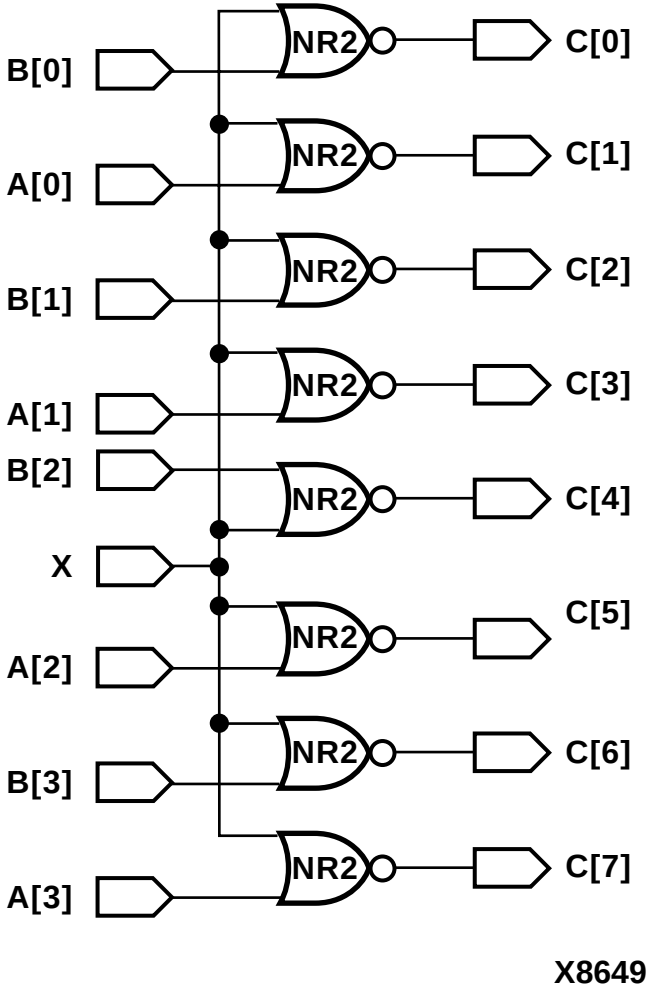


Figure 6-9 An 8-Bit Array Design

Common Usage of a for...generate Statement

The most common use of the generate statement is to create multiple copies of components, processes, or blocks. The following example and figure demonstrates this use with components. (The example and figure following this example and figure show this usage with processes.)

The following example 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. The resulting design follows each of the examples.

```
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;
```

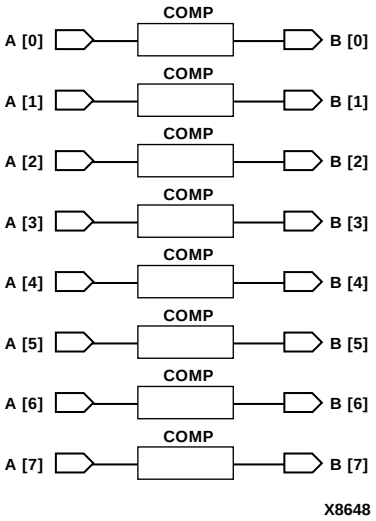


Figure 6-10 Design of COMP components Connecting Bit Vectors A and B

For more information about arrays, see "Array Types" section of the "Data Types" chapter.

if...generate Statements

The syntax follows.

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

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

Note: Unlike the if statement described in the "if Statements" section of the "Sequential Statements" chapter, 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 use a set of if...generate statements to define the beginning, middle, and ending sets of connections.

The following example shows a technology-independent description of an 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 1 bit, and the incoming DATA bit is moved into the low-order bit. The resulting design follows the example.

```
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;
```

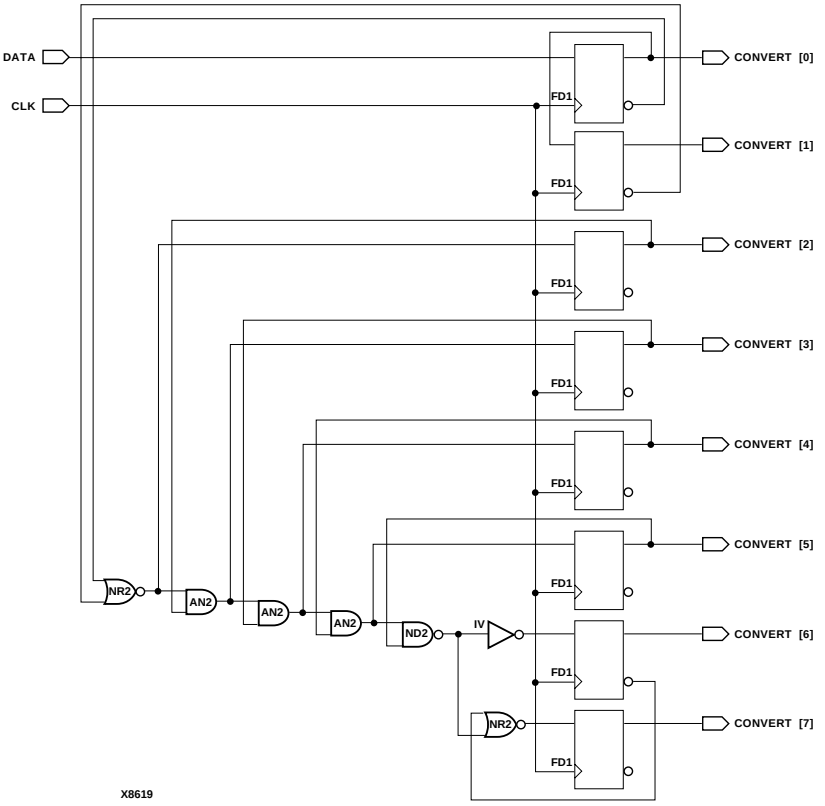


Figure 6-11 Design of N-Bit Serial-to-Parallel Converter

Chapter 7

Register and Three-State Inference

Foundation Express infers registers (latches and flip-flops) and three-state cells. This chapter explains inference behavior and results in the following sections.

- "Register Inference"
- "Three-State Inference"

Register Inference

By inferring registers, you can use sequential logic in your designs and keep your designs technology-independent. A register is a simple, one-bit memory device, either a latch or a flip-flop. A latch is a level-sensitive memory device. A flip-flop is an edge-triggered memory device.

Foundation Express' capability to infer registers supports coding styles other than those described in this chapter. However, for best results, do the following.

- Restrict each always block 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 templates provided in the "Inferring Latches" section and "Inferring Flip-Flops" section of this chapter.

The Inference Report

Foundation Express generates a general inference report when building a design. It provides the asynchronous set or reset, synchronous set or reset, and synchronous toggle conditions of each latch or

flip-flop, expressed in Boolean formulas. The following example shows the inference report for a JK flip-flop.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	Y	Y	N

```
Q_reg
   Sync-reset: J' K
   Sync-set: J K'
   Sync-toggle: J K
   Sync-set and Sync-reset ==> Q: X
```

The inference report shows the following.

- Y indicates that the flip-flop has a synchronous reset (SR) and a synchronous set (SS).
- N indicates that the flip-flop does not have an asynchronous reset (AR), an asynchronous set (AS), or a synchronous toggle (ST).

In the inference report, the last section of the report lists the objects that control the synchronous reset and set conditions. In this example (Inference Report for a JK Flip-Flop), a synchronous reset occurs when J is low (logic 0) and K is high (logic 1). The last line of the report indicates the register output value when both the set and reset are active.

- zero (0)—Indicates that the reset has priority and the output goes to logic 0
- one (1)—Indicates that the set has priority and the output goes to logic 1
- X—Indicates that there is no priority and that the output value is unstable

The “Inferring Latches” section and “Inferring Flip-Flops” section of this chapter provide inference reports for each register template. After you read a description in Foundation Express, check the inference report.

Latch Inference Warnings

Foundation Express generates a warning message when it infers a latch. The warning message is useful to verify that a combinatorial design does not contain memory components.

Controlling Register Inference

Use directives to direct the type of sequential device you want inferred. The default is to implement the type of latch described in the HDL code. These attributes override this behavior.

The ATTRIBUTES package in the VHDL library defines the following attributes for controlling register inference.

- async_set_reset

  When this is set to TRUE on a signal, Foundation Express searches for a branch that uses the signal as a condition. Foundation Express then checks whether the branch contains an assignment to a constant value. If the branch does, the signal becomes an asynchronous reset or set.

  Attach the async_set_reset attribute to 1-bit signals using the following syntax.

```
attribute async_set_reset of signal_name_list : signal is "true";
```

- async_set_reset_local

  Foundation Express treats listed signals in the specified process as if they have the async_set_reset attribute set to TRUE.

  Attach the async_set_reset_local attribute to a process label using the following syntax.

```
attribute async_set_reset_local of process_label : label is "signal_name_list";
```

- async_set_reset_local_all

  Foundation Express treats all signals in the specified processes as if they have the async_set_reset attribute set to TRUE.

  Attach the async_set_reset_local_all attribute to process labels by using the following syntax.

```
attribute async_set_reset_local_all of process_label_list : label is "true";
```

- sync_set_reset

  When this is set to TRUE on a signal, Foundation Express checks the signal to determine whether it synchronously sets or resets a register in the design.

  Attach the sync_set_reset attribute to 1-bit signals by using the following syntax.

```
attribute sync_set_reset of signal_name_list :  signal is "true";
```

- sync_set_reset_local

  Foundation Express treats listed signals in the specified process as if they have the sync_set_reset attribute set to TRUE.

  Attach the sync_set_reset_local attribute to a process label by using the following syntax.

```
attribute sync_set_reset_local of process_label : label is "signal_name_list";
```

- sync_set_reset_local_all

  Foundation Express treats all signals in the specified processes as if they have the sync_set_reset attribute set to TRUE.

  Attach the sync_set_reset_local_all attribute to process labels by using the following syntax.

```
attribute sync_set_reset_local_all of process_label_list : label is "true";
```

- one_cold

  A one-cold implementation means that all signals in a group are active low and that only one signal can be active at a given time. The one_cold directive prevents Foundation Express from implementing priority encoding logic for the set and reset signals.

  Add an assertion to the VHDL code to ensure that the group of signals has a one-cold implementation. Foundation Express does not produce any logic to check this assertion.

  Attach the one_cold attribute to set or reset signals on sequential devices by using the following syntax.

```
attribute one_cold signal_name_list : signal is "true";
```

- one_hot

  A one_hot implementation means that all signals in a group are active-high and that only one signal can be active at a given time. The one_hot attribute prevents Foundation Express from implementing priority encoding logic for the set and reset signals.

  Add an assertion to the VHDL code to ensure that the group of signals has a one_hot implementation. Foundation Express does not produce any logic to check this assertion.

  Attach the one_hot attribute to set or reset signals on sequential devices using the following syntax.

```
attribute one_hot signal_name_list : signal is "true";
```

Inferring Latches

In simulation, a signal or variable holds its value until that output is reassigned. In hardware, a latch implements this holding-of-state capability. Foundation Express supports inference of the following types of latches.

- SR latch
- D latch
- Master-slave latch

The following sections provide details about each of these latch types.

Inferring Set/Reset (SR) Latches

Use SR latches with caution, because they are difficult to test. If you decide to use SR latches, you must verify that the inputs are hazard-free (do not glitch). During synthesis, Foundation Express does not ensure that the logic driving the inputs is hazard-free.

The following example of an SR latch provides the VHDL code that implements the SR latch described in the truth table. The inference report following the truth table for an SR latch shows the inference report that Foundation Express generates.

set	reset	y
0	0	Not stable
0	1	1
1	0	0
1	1	y

The following example shows an SR latch.

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity sr_latch is
  port (SET, RESET : in std_logic;
        Q : out std_logic );
  attribute async_set_reset of SET, RESET :
```

```
    signal is "true";
end sr_latch;

architecture rtl of sr_latch is
begin

infer: process (SET, RESET) begin
  if (SET = '0') then
    Q <= '1';
  elsif (RESET = '0') then
    Q <= '0';
  end if;
end process infer;

end rtl;
```

The example below shows an inference report for an SR latch and its schematic.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	Y	Y	-	-	-

```
y_reg
   Async-reset: RESET'
   Async-set: SET'
   Async-set and Async-reset ==> Q: 1
```

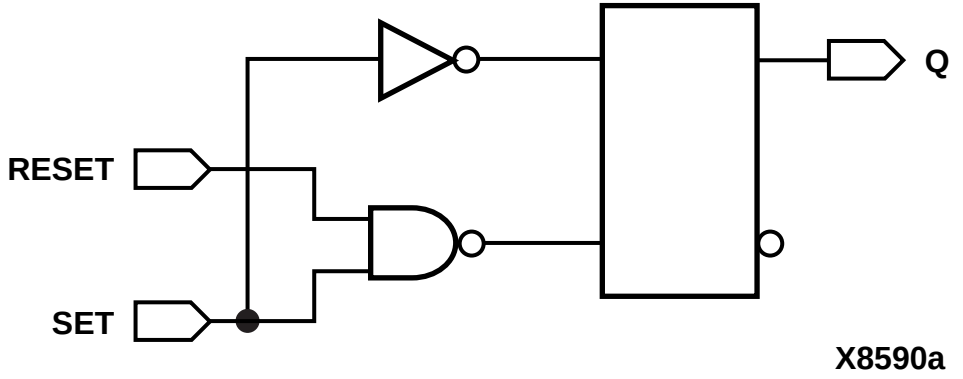


Figure 7-1 SR Latch

Inferring D Latches

When you do not specify the resulting value for an output under all conditions, as in an incompletely specified if statement, Foundation Express infers a D latch.

For example, the if statement in the following example infers a D latch because there is no else clause. The resulting value for output Q is specified only when input enable has a logic 1 value. As a result, output Q becomes a latched value.

```
process(DATA, GATE) begin
  if (GATE = '1') then
    Q <= DATA;
  end if;
end process;
```

To avoid latch inference, assign a value to the signal under all conditions, as shown in the following example.

```
process(DATA, GATE) begin
  if (GATE = '1') then
    Q <= DATA;
  else
    Q <= '0';
  end if;
end process;
```

Variables declared locally within a subprogram do not hold their value over time, because each time a subprogram is called, its variables are reinitialized. Therefore, Foundation Express does not infer latches for variables declared in subprograms. In the following example, Foundation Express does not infer a latch for output Q.

```
function MY_FUNC(DATA, GATE : std_logic) return
                 std_logic is
     variable STATE: std_logic;
begin
  if (GATE = '1') then
    STATE <= DATA;
  end if;
  return STATE;
end;
. . .
Q <= MY_FUNC(DATA, GATE);
```

The following sections provide code examples, inference reports, and figures for these types of D latches.

- Simple D latch
- D latch with asynchronous set
- D latch with asynchronous reset
- D latch with asynchronous set and reset

Simple D Latch When you infer a D latch, control the gate and data signals from the top-level design ports or through combinatorial logic. Gate and data signals that can be controlled ensure that simulation can initialize the design.

The following example provides the VHDL template for a D latch. Foundation Express generates the inference report shown after the example for a D latch. The figure “D Latch” shows the inferred latch.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity d_latch is
  port (GATE, DATA: in std_logic;
        Q : out std_logic );
end d_latch;

architecture rtl of d_latch is
begin

infer: process (GATE, DATA) begin
  if (GATE = '1') then
    Q <= DATA;
  end if;
end process infer;

end rtl;
```

The example below shows an inference report for a D latch.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	N	N	-	-	-

```
Q_reg
   reset/set:none
```

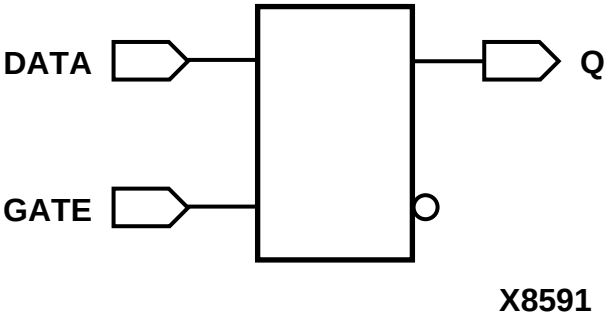


Figure 7-2 D Latch

D Latch with Asynchronous Set The template in this section uses the async_set_reset attribute to direct Foundation Express to the asynchronous set (AS) pins of the inferred latch.

The following example provides the VHDL template for a D latch with an asynchronous set. Foundation Express generates the inference report shown following the example for a D latch with asynchronous set. The figure “D Latch with Asynchronous Set” shows the inferred latch.

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity d_latch_async_set is
  port (GATE, DATA, SET : in std_logic;
        Q : out std_logic );
  attribute async_set_reset of SET :
    signal is ”true”;
end d_latch_async_set;

architecture rtl of d_latch_async_set is
begin

infer: process (GATE, DATA, SET) begin
  if (SET = ’0’) then
    Q <= ’1’;
  elsif (GATE = ’1’) then
    Q <= DATA;
```

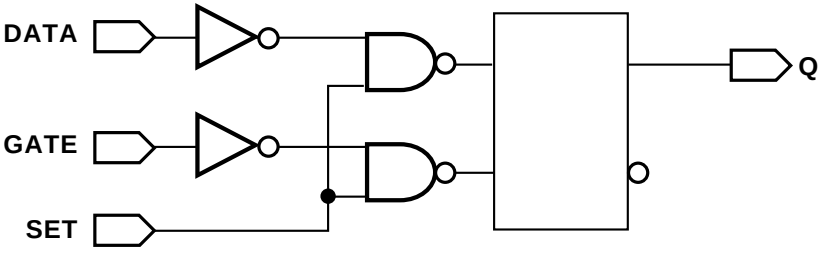
```
  end if;
end process infer;

end rtl;
```

The following example shows an inference report for a D latch with asynchronous set.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	N	Y	-	-	-

```
Q_reg
   Async-set: SET'
```



X8592

Figure 7-3 D Latch with Asynchronous Set

Note: Because the target technology library does not contain a latch with an asynchronous set, Foundation Express synthesizes the set logic by using combinatorial logic.

D Latch with Asynchronous Reset The template in this section uses the async_set_reset attribute to direct Foundation Express to the asynchronous reset (AR) pins of the inferred latch.

The following example provides the VHDL template for a D latch with an asynchronous reset. Foundation Express generates the inference report shown following the example for a D latch with asynchronous reset. The figure "D Latch with Asynchronous Reset" shows the inferred latch.

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity d_latch_async_reset is
  port (GATE, DATA, RESET : in std_logic;
        Q : out std_logic );
  attribute async_set_reset of RESET :
    signal is "true";
end d_latch_async_reset;

architecture rtl of d_latch_async_reset is
begin

infer : process (GATE, DATA, RESET) begin
  if (RESET = '0') then
    Q <= '0';
  elsif (GATE = '1') then
    Q <= DATA;
  end if;
end process infer;

end rtl;
```

The following example shows an inference report for a D latch with asynchronous reset.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	Y	N	-	-	-

```
Q_reg
   Async-reset: RESET'
```

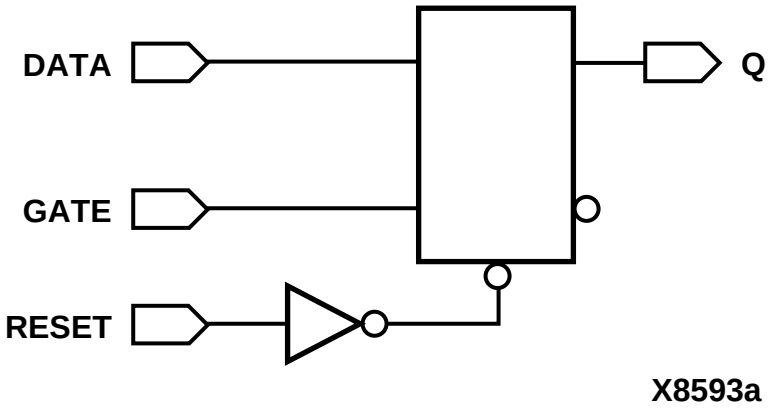


Figure 7-4 D Latch with Asynchronous Reset

D Latch with Asynchronous Set and Reset The following example provides the VHDL template for a D latch with an active-low asynchronous set and reset. This template uses the async_set_reset_local attribute to direct Foundation Express to the asynchronous signals in the infer process.

The template in the following example uses the one_cold attribute to prevent priority encoding of the set and reset signals. If you do not specify the one_cold attribute, the set signal has priority, because it is used as the condition for the if clause. Foundation Express generates the inference report shown following the example for a D latch with asynchronous set and reset. The figure "D Latch with Asynchronous Set and Reset" shows the inferred latch.

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity d_latch_async is
  port (GATE, DATA, SET, RESET :in  std_logic;
        Q : out std_logic );
attribute one_cold of SET, RESET :
    signal is ”true”;
end d_latch_async;

architecture rtl of d_latch_async is
  attribute async_set_reset_local of infer :
```

```
    label is "SET, RESET";
begin

infer : process (GATE, DATA, SET, RESET) begin
  if (SET = '0') then
    Q <= '1';
  elsif (RESET = '0') then
    Q <= '0';
  elsif (GATE = '1') then
    Q <= DATA;
  end if;
end process infer;
end rtl;
```

The following example shows an inference report for a D latch with asynchronous set and reset.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	Y	Y	-	-	-

```
Q_reg
   Async-reset: RESET'
   Async-set: SET'
   Async-set and Async-reset ==> Q: X
```

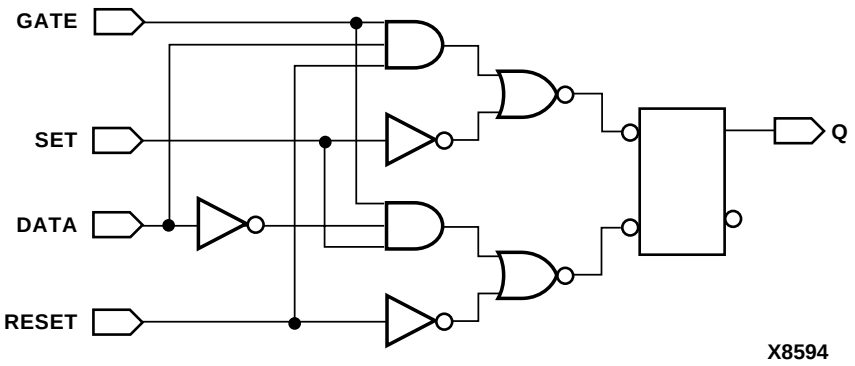


Figure 7-5 D Latch with Asynchronous Set and Reset

Understanding the Limitations of D Latch Inference A variable must always have a value before it is read. As a result, a conditionally assigned variable cannot be read after the if statement in which it is assigned. A conditionally assigned variable is assigned a new value

under some, but not all, conditions. The following example shows an invalid use of the conditionally assigned variable VALUE.

```
signal X, Y : std_logic;
. . .
process
  variable VALUE : std_logic;
begin
  if (condition) then
    VALUE <= X;
  end if;
  Y <= VALUE;  -- Invalid read of variable VALUE
end process;
```

Inferring Master-Slave Latches

You can infer two-phase systems by using D latches.The following example shows a simple two-phase system with clocks MCK and SCK. The inference reports follow the example. The figure “Two-Phase Clocks” shows the inferred latch.

```
library IEEE;
use IEEE.std_Logic_1164.all;

entity LATCH_VHDL is
  port(MCK, SCK, DATA: in std_logic;
       Q : out std_logic );
end LATCH_VHDL;

architecture rtl of LATCH_VHDL is
  signal TEMP : std_logic;
begin

process (MCK, DATA) begin
  if (MCK = '1') then
    TEMP <= DATA;
  end if;
end process;

process (SCK, TEMP) begin
  if (SCK = '1') then
    Q <= TEMP;
  end if;
end process;
```

```
end rtl;
```

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
TEMP_reg	Latch	1	-	-	N	N	-	-	-

```
TEMP_reg
   reset/set: none
```

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	N	N	-	-	-

```
Q_reg
  reset/set: none
```

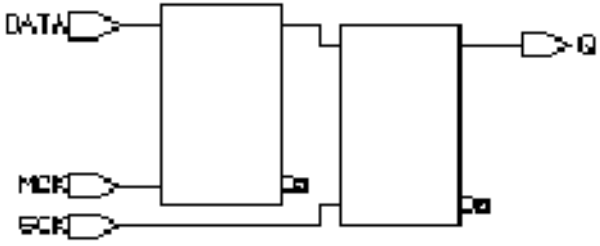


Figure 7-6 Two-Phase Clocks

Inferring Flip-Flops

Foundation Express can infer D flip-flops, JK flip-flops, and toggle flip-flops. The following sections provide details about each of these flip-flop types.

Many FPGA devices have a dedicated set/reset hardware resource that should be used. For this reason, you should infer asynchronous set/reset signals for all flip-flops in the design. Foundation Express will then use the global set/reset lines.

Inferring D Flip-Flops

Foundation Express infers a D flip-flop whenever the condition of a wait or if statement uses an edge expression (a test for the rising or falling edge of a signal). Use the following syntax to describe a rising edge.

```
SIGNAL'event and SIGNAL = '1'
```

Use the following syntax to describe a falling edge.

```
SIGNAL'event and SIGNAL = '0'
```

If you are using the IEEE std_logic_1164 package, you can use the following syntax to describe a rising edge and a falling edge.

```
if (rising_edge (CLK)) then
```

```
if (falling_edge (CLK)) then
```

If you are using the IEEE std_logic_1164 package, you can use the following syntax for a bused clock. You can also use a member of a bus as a signal.

```
sig (3)'event and sig (3) = '1'
```

```
rising_edge (sig(3))
```

A wait statement containing an edge expression causes Foundation Express to create flip-flops for all signals, and some variables are assigned values in the process. The following example shows the most common usage of the wait statement to infer a flip-flop.

```
process
begin
  wait until (edge);
  ...
end process;
```

An if statement implies flip-flops for signals and variables in the branches of the if statement. The following example shows the most common usages of the if statement to infer a flip-flop.

```
process (sensitivity_list)
begin
  if (edge)
    ...
  end if;
end process;
```

```
process (sensitivity_list)
begin
  if (...) then
    ...
  elsif (...)
    ...
  elsif (edge) then
    ...
  end if;
end process;
```

You can sometimes use wait and if statements interchangeably. If possible, use the if statement, because it provides greater control over the inferred registers.

The following sections provide code examples, inference reports, and figures for these types of D flip-flops.

- Positive edge-triggered D flip-flop
- Positive edge-triggered D flip-flop using rising_edge
- Negative edge-triggered D flip-flop
- Negative edge-triggered D flip-flop using falling_edge
- D flip-flop with asynchronous set
- D flip-flop with asynchronous reset
- D flip-flop with asynchronous set and reset
- D flip-flop with synchronous set
- D flip-flop with synchronous reset
- D flip-flop with synchronous and asynchronous load
- Multiple flip-flops with asynchronous and synchronous controls

Positive Edge-Triggered D Flip-Flop When you infer a D flip-flop, control the clock and data signals from the top-level design ports or through combinatorial logic. Clock and data signals that can be controlled ensure that simulation can initialize the design. If you cannot control the clock and data signals, infer a D flip-flop with asynchronous reset or set or with a synchronous reset or set.

The following example provides the VHDL template for a positive edge-triggered D flip-flop. Foundation Express generates the inference report shown following the example for a positive edge-trig-

gered D flip-flop. The figure “Positive-Edge-Triggered D Flip-flop” shows the inferred flip-flop.

```
library IEEE ;
use IEEE.std_logic_1164.all;

entity dff_pos is
  port (DATA, CLK : in std_logic;
        Q : out std_logic );
end dff_pos;

architecture rtl of dff_pos is
begin

infer : process (CLK) begin
  if (CLK'event and CLK = '1') then
    Q <= DATA;
  end if;
end process infer;

end rtl;
```

The following example shows an inference report for a positive edge-triggered D flip-flop.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	N	N	N

```
Q_reg
   set/reset/toggle: none
```

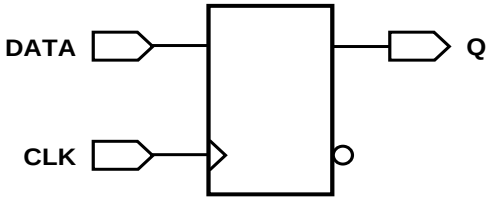


Figure 7-7 Positive Edge-Triggered D Flip-Flop

Positive Edge-Triggered D Flip-Flop Using rising_edge The following example provides the VHDL template for a positive edge-

triggered D flip-flop using the IEEE_std_logic_1164 package and rising_edge.

Foundation Express generates the inference report shown after the example. The figure following the inference report shows the inferred flip-flop.

```
library IEEE ;
use IEEE.std_logic_1164.all;

entity dff_pos is
  port (DATA, CLK : in std_logic;
        Q : out std_logic );
end dff_pos;

architecture rtl of dff_pos is
begin

infer : process (CLK) begin
  if (rising_edge (CLK)) then
    Q <= DATA;
  end if;
end process infer;

end rtl;
```

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	N	N	N

```
Q_reg
   set/reset/toggle: none
```

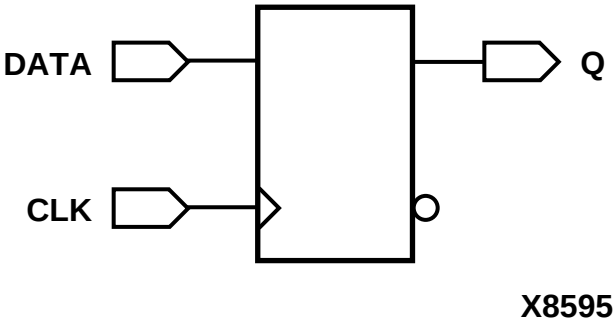


Figure 7-8 Positive Edge-Triggered D Flip-Flop Using rising_edge

Negative Edge-Triggered D Flip-Flop The following example provides the VHDL template for a negative edge-triggered D flip-flop. Foundation Express generates the inference report following the example for a negative edge-triggered D flip-flop. The figure "Negative Edge-Triggered D Flip-Flop" shows the inferred flip-flop.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff_neg is
  port (DATA, CLK : in std_logic;
        Q : out std_logic );
end dff_neg;

architecture rtl of dff_neg is
begin

infer : process (CLK) begin
  if (CLK'event and CLK = '0') then
    Q <= DATA;
  end if;
end process infer;

end rtl;
```

The following example shows an inference report for a negative edge-triggered D flip-flop.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	N	N	N

```
Q_reg
   set/reset/toggle: none
```

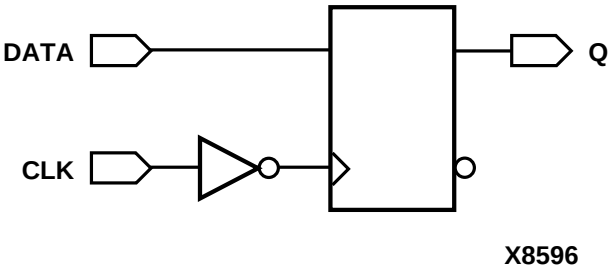


Figure 7-9 Negative Edge-Triggered D Flip-Flop

Negative Edge-Triggered D Flip-Flop Using falling_edge The following example provides the VHDL template for a negative edge-triggered D flip-flop using the IEEE_std_logic_1164 package and falling_edge.

Foundation Express generates the inference report shown after the following example. The figure following the inference report shows the inferred flip-flop.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff_neg is
  port (DATA, CLK : in std_logic;
        Q : out std_logic );
end dff_neg;

architecture rtl of dff_neg is
begin

infer : process (CLK) begin
  if (falling_edge (CLK)) then
    Q <= DATA;
```

```
  end if;
end process infer;

end rtl;
```

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	N	N	N

```
Q_reg
   set/reset/toggle: none
```

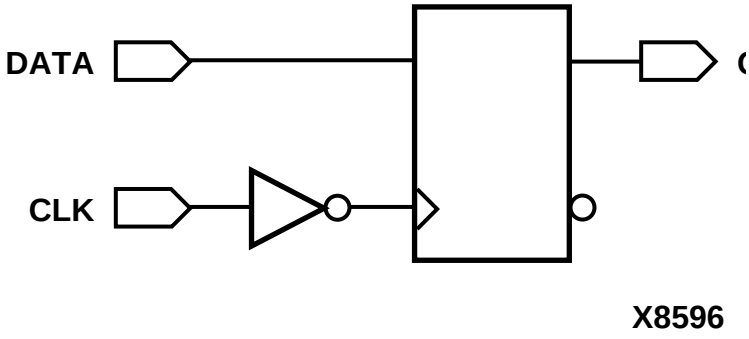


Figure 7-10 Negative Edge-Triggered D Flip-Flop Using falling_edge

D Flip-Flop with Asynchronous Set The following example provides the VHDL template for a D flip-flop with an asynchronous set. Foundation Express generates the inference report shown following the example for a D flip-flop with asynchronous set. The figure “D Flip-Flop with Asynchronous Set” shows the inferred flip-flop.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff_async_set is
  port (DATA, CLK, SET : in std_logic;
        Q : out std_logic );
end dff_async_set;
```

```
architecture rtl of dff_async_set is
begin

infer : process (CLK, SET) begin
  if (SET = '0') then
    Q <= '1';
  elsif (CLK'event and CLK = '1') then
    Q <= DATA;
  end if;
end process infer;
end rtl;
```

The following example shows an inference report for a D flip-flop with asynchronous set.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	Y	N	N	N

```
Q_reg
   Async-set: SET'
```

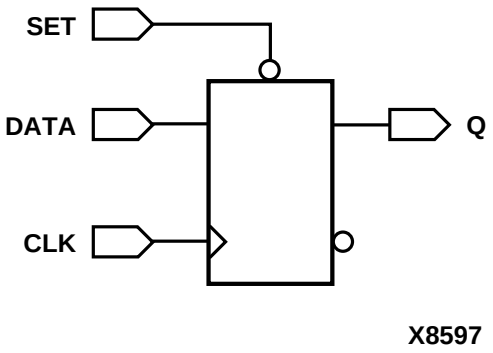


Figure 7-11 D Flip-Flop with Asynchronous Set

D Flip-Flop with Asynchronous Reset The following example provides the VHDL template for a D flip-flop with an asynchronous reset. Foundation Express generates the inference report following the example for a D flip-flop with asynchronous reset. The figure "D Flip-Flop with Asynchronous Reset" shows the inferred flip-flop.

```
library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity dff_async_reset is
  port (DATA, CLK, RESET : in std_logic;
        Q : out std_logic );
end dff_async_reset;

architecture rtl of dff_async_reset is
begin

infer : process ( CLK, RESET) begin
  if (RESET = '1') then
    Q <= '0';
  elsif (CLK'event and CLK = '1') then
    Q <= DATA;
  end if;
end process infer;

end rtl;
```

The following example shows an inference report for a D flip-flop with asynchronous reset.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	Y	N	N	N	N

```
Q_reg
   Async-reset: RESET
```

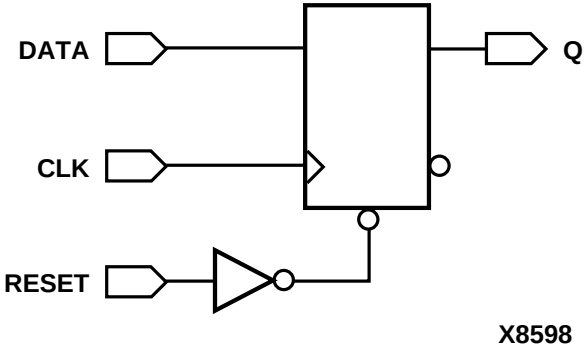


Figure 7-12 D Flip-Flop with Asynchronous Reset

D Flip-Flop with Asynchronous Set and Reset The following example provides the VHDL template for a D flip-flop with active high asynchronous set and reset pins. The template uses the one_hot

attribute to prevent priority encoding of the set and reset signals. If you do not specify the one_hot attribute, the reset signal has priority, because it is used as the condition for the if clause. Foundation Express generates the inference report following the example for a D flip-flop with asynchronous set and reset. The figure “D Flip-Flop with Asynchronous Set and Reset” shows the inferred flip-flop.

Note: Most FPGA architectures do not have a register with an asynchronous set and asynchronous reset cell available. For this reason, avoid this construct.

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity dff_async is
  port (DATA, CLK, SET, RESET : in std_logic;
        Q : out std_logic );
  attribute one_hot of SET, RESET : signal is ”true”;
end dff_async;

architecture rtl of dff_async is
begin
infer : process (CLK, SET, RESET) begin
  if (RESET = ’1’) then
    Q <= ’0’;
  elsif (SET = ’1’) then
    Q <= ’1’;
  elsif (CLK’event and CLK = ’1’) then
    Q <= DATA;
  end if;
end process infer;

end rtl;
```

The following example shows an inference report for a D flip-flop with asynchronous set and reset.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	Y	Y	N	N	N

```
Q_reg
   Async-reset: RESET
   Async-set: SET
   Async-set and Async-reset ==> Q: X
```

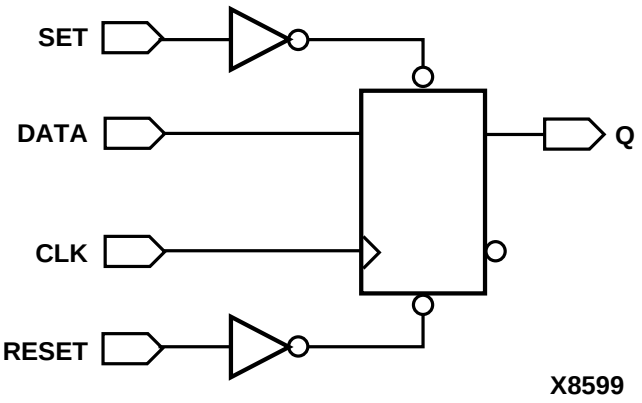


Figure 7-13 D Flip-flop with Asynchronous Set and Reset

D Flip-Flop with Synchronous Set or Reset The previous examples illustrate how to infer a D flip-flop with asynchronous controls—one way to initialize or control the state of a sequential device. You can also synchronously reset or set the flip-flop (see the following two examples in the next section). The sync_set_reset attribute directs Foundation Express to the synchronous controls of the sequential device.

When the target technology library does not have a D flip-flop with synchronous reset, Foundation Express infers a D flip-flop with synchronous reset logic as the input to the D pin of the flip-flop. If the reset (or set) logic is not directly in front of the D pin of the flip-flop, initialization problems can occur during gate-level simulation of the design.

D Flip-Flop with Synchronous Set The following example provides the VHDL template for a D flip-flop with synchronous set. Foundation Express generates the inference report shown following the example for a D flip-flop with synchronous set. The figure "D Flip-Flop with Synchronous Set" shows the inferred flip-flop.

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;
entity dff_sync_set is
  port (DATA, CLK, SET : in std_logic;
```

```
        Q : out std_logic );
  attribute sync_set_reset of SET : signal is "true";
end dff_sync_set;

architecture rtl of dff_sync_set is
begin

infer : process (CLK) begin
  if (CLK'event and CLK = '1') then
    if (SET = '1') then
      Q <= '1';
    else
      Q <= DATA;
    end if;
  end if;
end process infer;

end rtl;
```

The following example shows an inference report for a D flip-flop with synchronous set.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	N	Y	N

```
Q_reg
   Sync-set: SET
```

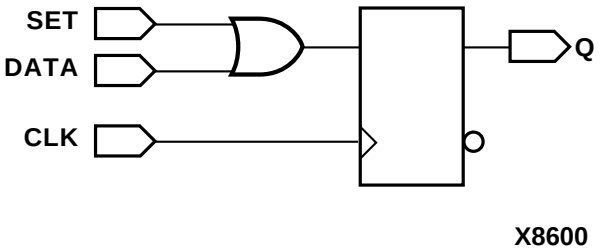


Figure 7-14 D Flip-Flop with Synchronous Set

D Flip-Flop with Synchronous Reset The following example provides the VHDL template for a D flip-flop with synchronous reset. Foundation Express generates the inference report shown following the example for a D flip-flop with synchronous reset. The figure "D Flip-Flop with Synchronous Reset" shows the inferred flip-flop.

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity dff_sync_reset is
  port (DATA, CLK, RESET : in std_logic;
        Q : out std_logic );
attribute sync_set_reset of RESET :
    signal is "true";
end dff_sync_reset;

architecture rtl of dff_sync_reset is
begin

infer : process (CLK) begin
  if (CLK'event and CLK = '1') then
    if (RESET = '0') then
      Q <= '0';
    else
      Q <= DATA;
    end if;
  end if;
end process infer;

end rtl;
```

The following example shows an inference report for a D flip-flop with synchronous reset.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	Y	N	N

```
Q_reg
   Sync-reset: RESET'
```

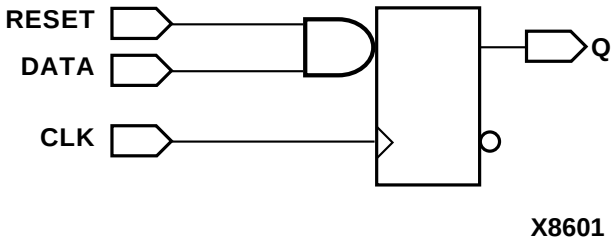


Figure 7-15 D Flip-Flop with Synchronous Reset

D Flip-Flop with Synchronous and Asynchronous Load D flip-flops can have asynchronous or synchronous controls. You must check the asynchronous conditions before you check the synchronous conditions.

The following example provides the VHDL template for a D flip-flop with synchronous load (called SLOAD) and an asynchronous load (called ALOAD). Foundation Express generates the inference report shown following the example for a D flip-flop with synchronous and asynchronous load. The figure "D Flip-Flop with Synchronous and Asynchronous Load" shows the inferred flip-flop.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff_a_s_load is
  port(SLOAD, ALOAD, ADATA, SDATA,
       CLK : in std_logic;
       Q : out std_logic );
end dff_a_s_load;

architecture rtl of dff_a_s_load is
begin

infer: process (CLK, ALOAD) begin
  if (ALOAD = '1') then
    Q <= ADATA;
  elsif (CLK'event and CLK = '1') then
    if (SLOAD = '1') then
      Q <= SDATA;
    end if;
  end if;
end process infer;

end rtl;
```

The following example shows an inference report for a D flip-flop with synchronous and asynchronous load.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	N	N	N

```
Q_reg
   set/reset/toggle: none
```

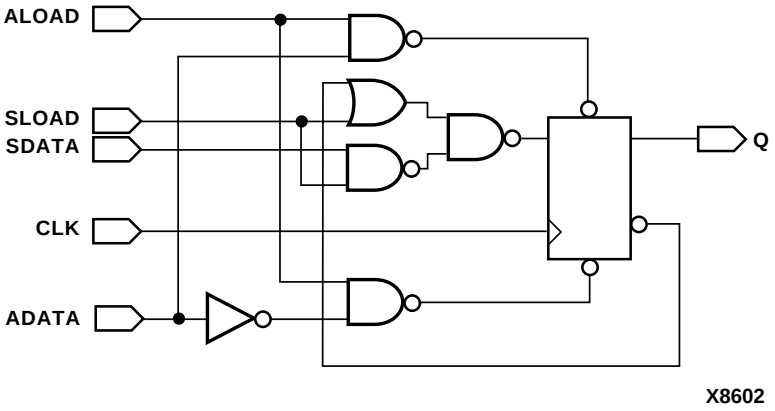


Figure 7-16 D Flip-Flop with Synchronous and Asynchronous Load

Multiple Flip-Flops with Asynchronous and Synchronous Controls If a signal is synchronous in one process but asynchronous in another, use the sync_set_reset_local and async_set_reset_local attributes to direct Foundation Express to the correct implementation.

In the following example, block infer_sync uses the reset signal as a synchronous reset, and the process infer_async uses the reset signal as an asynchronous reset. Foundation Express generates the inference report shown following the example for multiple flip-flops with asynchronous and synchronous controls. The figure "Multiple Flip-flops with Asynchronous and Synchronous Controls" shows the resulting design.

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity multi_attr is
  port (DATA1, DATA2, CLK, RESET, SLOAD : in
              std_logic;
        Q1, Q2 : out std_logic );
end multi_attr;

architecture rtl of multi_attr is
  attribute async_set_reset_local of infer_async :
```

```
    label is "RESET";
  attribute sync_set_reset_local of infer_sync :
    label is "RESET";
begin

infer_sync: process (CLK) begin
  if (CLK'event and CLK = '1') then
    if (RESET = '0') then
      Q1 <= '0';
    elsif (SLOAD = '1') then
      Q1 <= DATA1;
    end if;
  end if;
end process infer_sync;

infer_async: process (CLK, RESET) begin
  if (RESET = '0') then
    Q2 <= '0';
  elsif (CLK'event and CLK = '1') then
    if (SLOAD = '1') then
      Q2 <= DATA2;
    end if;
  end if;
end process infer_async;

end rtl;
```

The following example shows inference reports for multiple flip-flops with asynchronous and synchronous controls.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q1_reg	Flip-flop	1	-	-	N	N	Y	N	N

```
Q1_reg
   Sync-reset: RESET'
```

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q2_reg	Flip-flop	1	-	-	Y	N	N	N	N

```
Q2_reg
   Async-reset: RESET'
```

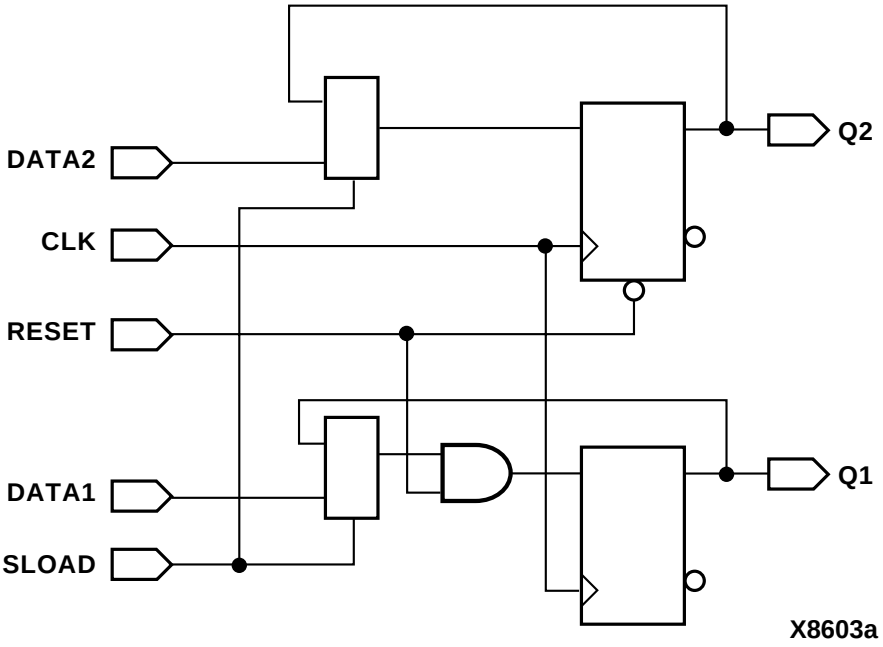


Figure 7-17 Multiple Flip-Flops with Asynchronous and Synchronous Controls

A flip-flop inference has specific limitations. See the "Understanding Limitations of Register Inference" section of this chapter.

Inferring JK Flip-Flops

When you infer a JK flip-flop, make sure you can control the J, K, and clock signals from the top-level design ports to ensure that simulation can initialize the design.The following sections provide code examples, inference reports, and figures for these types of JK flip-flops.

- JK flip-flop
- JK flip-flop with asynchronous set and reset

JK Flip-Flop When you infer a JK flip-flop, make sure you can control the J, K, and clock signals from the top-level design ports to ensure that simulation can initialize the design.

In the JK flip-flop, the J and K signals act as active-high synchronous set and reset. Use the sync_set_reset directive to indicate that the J and K signals are the synchronous set and reset for the design.

Table 7-1 Truth Table for JK Flip-Flop

J	K	CLK	Q_{n+1}
0	0	Rising	Q_n
0	1	Rising	0
1	0	Rising	1
1	1	Rising	Q_nB
X	X	Falling	Q_n

The following example provides the VHDL code that implements the JK flip-flop described in the truth table.

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity jk is
  port(J, K, CLK : in std_logic;
       Q_out : out std_logic );
  attribute sync_set_reset of J, K :
    signal is ”true”;
end jk;

architecture rtl of jk is
  signal Q : std_logic;
begin
infer: process
  variable JK : std_logic_vector ( 1 downto 0);
begin
  wait until (CLK’event and CLK = ’1’);
  JK <= (J & K);
  case JK is
    when ”01” => Q <= ’0’;
    when ”10” => Q <= ’1’;
    when ”11” => Q <= not (Q);
    when ”00” => Q <= Q;
    when others => Q <= ’X’;
```

```
  end case;
end process infer;

Q_out <= Q;
end rtl;
```

The following example shows the inference report generated by Foundation Express for a JK flip-flop, and the figure following the report, “JK Flip-Flop,” shows the inferred flip-flop.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	Y	Y	Y

```
Q_reg
   Sync-reset: J’ K
   Sync-set: J K’
   Sync-toggle: J K
   Sync-set and Sync-reset ==> Q: X
```

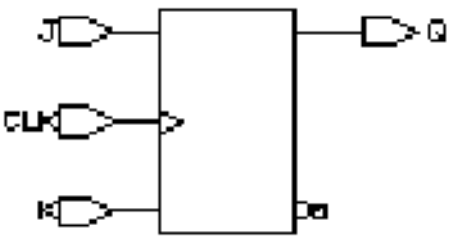


Figure 7-18 JK Flip-Flop

JK Flip-Flop With Asynchronous Set and Reset Use the sync_set_reset attribute to indicate the JK function. Use the one_hot attribute to prevent priority encoding of the J and K signals.

The following example provides the VHDL template for a JK flip-flop with asynchronous set and reset.

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity jk_async_sr is
  port (SET, RESET, J, K, CLK : in std_logic;
        Q_out : out std_logic );
```

```
  attribute sync_set_reset of J, K :
    signal is "true";
  attribute one_hot of SET,RESET : signal is "true";
end jk_async_sr;

architecture rtl of jk_async_sr is
  signal Q : std_logic;
begin

infer : process (CLK, SET, RESET)
  variable JK : std_logic_vector (1 downto 0);
begin
  if (RESET = '1') then
    Q <= '0';
  elsif (SET = '1') then
    Q <= '1';
  elsif (CLK'event and CLK = '1') then
    JK <= (J & K);
    case JK is
      when "01" => Q <= '0';
      when "10" => Q <= '1';
      when "11" => Q <= not(Q);
      when "00" => Q <= Q;
      when others => Q <= 'X';
    end case;
  end if;
end process infer;
Q_out <= Q;

end rtl;
```

The following table shows the inference report Foundation Express generates for a JK flip-flop with asynchronous set and reset, and the figure following the report, "JK Flip-Flop with Asynchronous Set and Reset," shows the inferred flip-flop.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	Y	Y	Y	Y	Y

```
Q_reg
    Async-reset: RESET
    Async-set: SET
    Sync-reset: J' K
    Sync-set: J K'
    Sync-toggle: J K
```

```
Async-set and Async-reset ==> Q: X
Sync-set and Sync-reset ==> Q: X
```

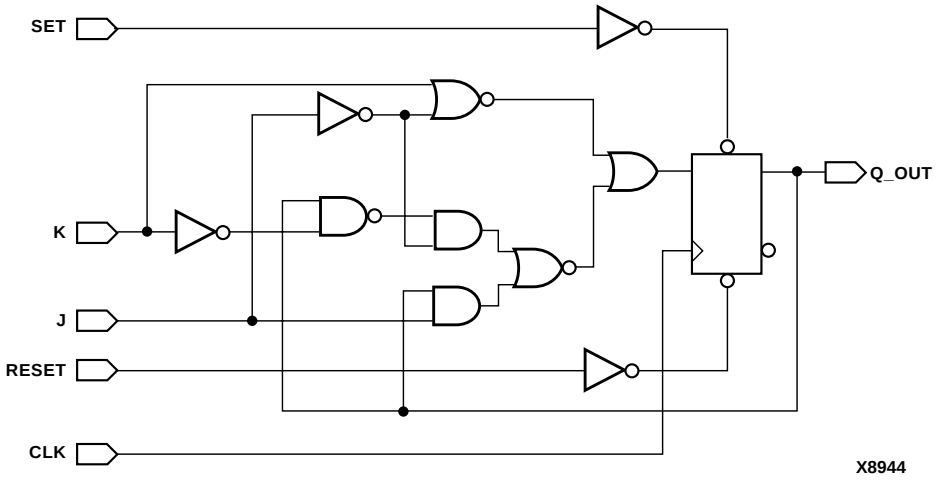


Figure 7-19 JK Flip-Flop with Asynchronous Set and Reset

Inferring Toggle Flip-Flops

To infer toggle flip-flops, follow the coding style in the following examples. You must include asynchronous controls in the toggle flip-flop description. Without them, you cannot initialize toggle flip-flops to a known state.

The following sections provide code examples, inference reports, and figures for these types of toggle flip-flops.

- Toggle flip-flop with asynchronous set
- Toggle flip-flop with asynchronous reset
- Toggle flip-flop with enable and asynchronous reset

Toggle Flip-Flop With Asynchronous Set The following example shows the VHDL template for a toggle flip-flop with asynchronous set. Foundation Express generates the inference report shown following the example, and the figure “Toggle Flip-Flop with Asynchronous Set” shows the flip-flop.

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
entity t_async_set is
  port(SET, CLK : in std_logic;
       Q : out std_logic );
end t_async_set;
architecture rtl of t_async_set is
  signal TMP_Q : std_logic;
begin

infer: process (CLK, SET) begin
  if (SET = '1') then
    TMP_Q <= '1';
  elsif (CLK'event and CLK = '1') then
    TMP_Q <= not (TMP_Q);
  end if;
  Q <= TMP_Q;
end process infer;

end rtl;
```

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
TMP_Q_reg	Flip-flop	1	-	-	N	Y	N	N	Y

```
TMP_Q_reg
   Async-set: SET
   Sync-toggle: true
```

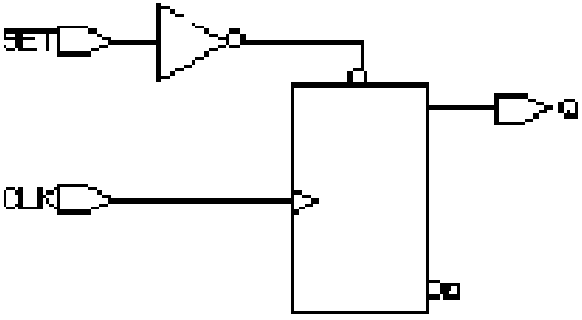


Figure 7-20 Toggle Flip-Flop with Asynchronous Set

Toggle Flip-Flop With Asynchronous Reset The following example provides the VHDL template for a toggle flip-flop with asynchronous reset. The table following the example shows the inference report, and the figure following the report, "Toggle Flip-Flop with Asynchronous Reset," shows the inferred flip-flop.

```
library IEEE ;
use IEEE.std_logic_1164.all;

entity t_async_reset is
  port(RESET, CLK : in std_logic;
       Q : out std_logic );
end t_async_reset;

architecture rtl of t_async_reset is
  signal TMP_Q : std_logic;
begin

infer: process (CLK, RESET) begin
  if (RESET = '1') then
    TMP_Q <= '0';
  elsif (CLK'event and CLK = '1') then
    TMP_Q <= not (TMP_Q);
  end if;
  Q <= TMP_Q;
end process infer;

end rtl;
```

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
TMP_Q_reg	Flip-flop	1	-	-	Y	N	N	N	Y

```
TMP_Q_reg
    Async-reset: RESET
    Sync-toggle: true
```

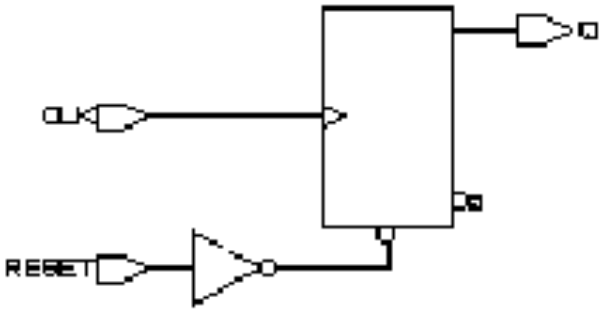


Figure 7-21 Toggle Flip-Flop with Asynchronous Reset

Toggle Flip-Flop With Enable and Asynchronous Reset The following example provides the VHDL template for a toggle flip-flop with an enable and an asynchronous reset. The flip-flop toggles only when the enable (TOGGLE signal) has a logic 1 value.

Foundation Express generates the inference report shown following the example, and the figure following the report, "Toggle Flip-Flop with Enable and Asynchronous Reset," shows the inferred flip-flop.

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity t_async_en_r is
  port(RESET, TOGGLE, CLK : in std_logic;
       Q : out std_logic );
end t_async_en_r;

architecture rtl of t_async_en_r is
  signal TMP_Q : std_logic;
begin

infer: process (CLK, RESET) begin
  if (RESET = '1') then
    TMP_Q <= '0';
  elsif (CLK'event and CLK = '1') then
    if (TOGGLE = '1') then
      TMP_Q <= not (TMP_Q);
    end if;
  end if;
end process infer;

Q <= TMP_Q;
```

```
end rtl;
```

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
TMP_Q_reg	Flip-flop	1	-	-	Y	N	N	N	Y

```
TMP_Q_reg
   Async-reset: RESET
   Sync-toggle: TOGGLE
```

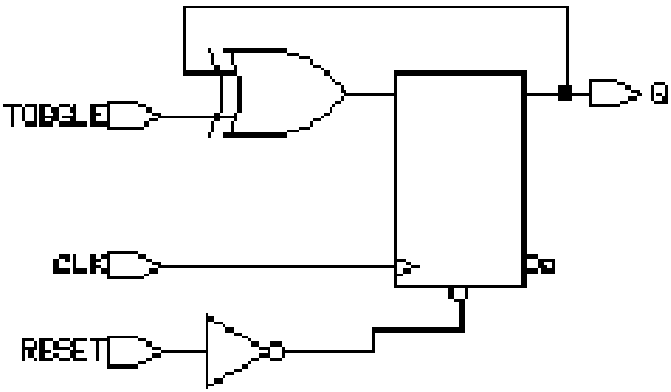


Figure 7-22 Toggle Flip-Flop with Enable and Asynchronous Reset

Getting the Best Results

This section provides tips for improving the results you achieve during flip-flop inference. The following topics are covered.

- Minimizing flip-flop count
- Correlating synthesis results with simulation results

Minimizing Flip-Flop Count HDL descriptions should build only as many flip-flops as the design requires.

Circuit Description Inferring Too Many Flip-Flops The following example shows a description that infers too many flip-flops. The inference report is shown following the example. The figure “Circuit with Six Inferred Flip-Flops” shows the inferred flip-flops.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity count is
  port (CLK, RESET : in std_logic;
        AND_BITS, OR_BITS,
        XOR_BITS : out std_logic );
end count;

architecture rtl of count 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;
```

The following example has only one process, which contains a wait statement and six output signals. Foundation Express infers six flip-flops, one for each output signal in the process.

- COUNT(2:0) (three inferred flip-flops)
- AND_BITS (one inferred flip-flop)
- OR_BITS (one inferred flip-flop)
- XOR_BITS (one inferred flip-flop)

However, because the outputs AND_BITS, OR_BITS, and XOR_BITS depend solely on the value of variable COUNT, and variable COUNT is registered, these three outputs do not need to be registered. Therefore, assign AND_BITS, OR_BITS, and XOR_BITS within a process

that does not have a wait statement (see the next section, “Circuit Description Inferring Correct Number of Flip-Flops”).

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
AND_BITS_reg	Flip-flop	1	-	-	N	N	N	N	N
COUNT_reg	Flip-flop	3	Y	N	N	N	N	N	N
OR_BITS_reg	Flip-flop	1	-	-	N	N	N	N	N
XOR_BITS_reg	Flip-flop	1	-	-	N	N	N	N	N

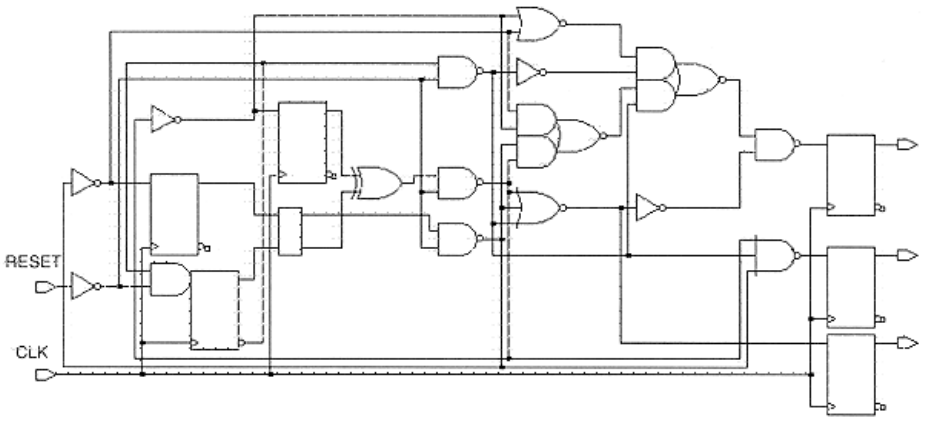


Figure 7-23 Circuit with Six Inferred Flip-Flops

Circuit Description Inferring Correct Number of Flip-Flops To avoid inferring extra flip-flops, assign the output signals from within a process that does not have a wait statement.

The following example shows a description with two processes, one with a wait statement and one without. The registered (synchronous) assignments are in the first process, which contains the wait statement. The other (asynchronous) assignments are in the second process. Signals communicate between the two processes.

This description style lets you choose the signals that are registered and those that are not. The inference report is shown following the example. The figure “Circuit with Three Inferred Flip-Flops” shows the resulting circuit.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity count is
  port(CLK, RESET : in std_logic;
       AND_BITS, OR_BITS, XOR_BITS : out std_logic);
end count;

architecture rtl of count is
  signal COUNT : std_logic_vector (2 downto 0);
begin

reg : process begin
  wait until (CLK'event and CLK = '1');
  if (RESET = '1') then
    COUNT <= "000";
  else
    COUNT <= COUNT + 1;
  end if;
end process reg;
combine : process(count) 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 combine;

end rtl;
```

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
COUNT_reg	Flip-flop	3	Y	N	N	N	N	N	N

```
COUNT_reg (width 3)
   set/reset/toggle: none
```

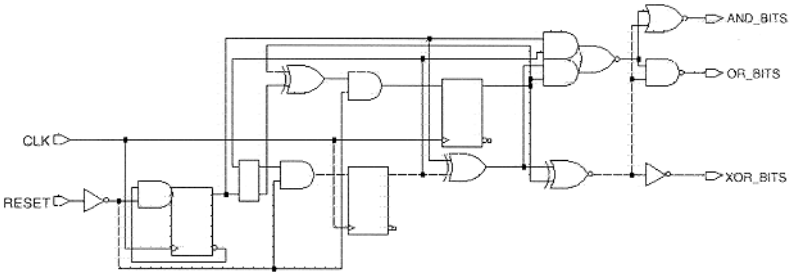


Figure 7-24 Circuit with Three Inferred Flip-Flops

This technique of separating combinatorial logic from registered or sequential logic in your design is useful when describing finite state machines. See these in the "Examples" appendix.

- "Moore Machine"
- "Mealy Machine"
- "Count Zeros—Sequential Version""
- "Soft Drink Machine—State Machine Version"

Correlating Synthesis Results with Simulation Results Using delay specifications with registered values can cause the simulation to behave differently from the logic Foundation Express synthesizes. For example, the description in the following example contains delay information that causes Foundation Express to synthesize a circuit that behaves unexpectedly (the post-synthesis simulation results do not match the pre-synthesis simulation results).

```
component flip_flop (D, CLK : in std_logic;
                     Q : out std_logic );
end component;

process (A, CLK);
  signal B: std_logic;
begin
  B <= A after 100ns;

  F1: flip_flop port map (A, CLK, C),

  F2: flip_flop port map (B, CLK, D);
end process;
```

In the above example, B changes 100 nanoseconds after A changes. If the clock period is less than 100 nanoseconds, output D is one or

more clock cycles behind output C during simulation of the design. However, because Foundation 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 post-synthesis simulation.

When using delay information in your designs, make sure that 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.

Understanding Limitations of Register Inference

Foundation Express cannot infer the following components. You must instantiate these components in your VHDL description.

- Flip-flops and latches with three-state outputs
- Flip-flops with bidirectional pins
- Flip-flips with multiple clock inputs
- Multiport latches
- Register banks

Note: Although you can instantiate flip-flops with bidirectional pins, Foundation Express interprets these cells as black boxes.

If you use an if statement to infer D flip-flops, your design must meet the following requirements.

- An edge expression must be the only condition of an if or an elsif clause.

  The following if statement is invalid because it has multiple conditions in the if clause.

  ```
  if (edge and RST = '1')
  ```

- You can have only one edge expression in an if clause, and the if clause must not have an else clause.

  The following if statement is invalid, because you cannot include an else clause when using an edge expression as the if or elsif condition.

  ```
  if X > 5 then
    sequential_statement;
  elsif edge then
  ```

```
  sequential_statement;
else
  sequential_statement;
end if;
```

- An edge expression cannot be part of another logical expression or be used as an argument.

  The following function call is invalid, because you cannot use the edge expression as an argument.

  `any_function(`*edge*`);`

Three-State Inference

Foundation Express infers a three-state driver when you assign the value of Z to a variable. The Z value represents the high-impedance state. Foundation Express infers one three-state driver per process. You can assign high-impedance values to single-bit or bused signals (or variables).

Reporting Three-State Inference

The following example shows a three-state inference report.

Three-State Device Name	Type	MB
OUT1_tri	Three-State Buffer	N

The first column of the report indicates the name of the inferred three-state device. The second column of the report indicates the type of three-state device that Foundation Express inferred. The third column indicates whether the three-state device has multiple bits.

Controlling Three-State Inference

Foundation Express always infers a three-state driver when you assign the value of Z to a signal or variable. Foundation Express does not provide any means of controlling the inference.

Inferring Three-State Drivers

This section contains VHDL examples that infer the following types of three-state drivers.

- Simple three-state driver

- Three-state driver with registered enable
- Three-state driver without registered enable

Inferring a Simple Three-State Driver

This section provides a template for a simple three-state driver. In addition, this section supplies examples of how allocating high-impedance assignments to different processes affects three-state inference.

The following example provides the VHDL template for a simple three-state driver. Foundation Express generates the inference report shown following the example for a simple three-state driver. The figure “Three-State Driver” shows the inferred three-state driver.

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
entity three_state is
port(IN1, ENABLE : in std_logic;
     OUT1 : out std_logic );
end;

architecture rtl of three_state is
begin

process (IN1, ENABLE) begin
  if (ENABLE = '1') then
    OUT1 <= IN1;
  else
    OUT1 <= 'Z';   -- assigns high-impedance state
  end if;
end process;

end rtl;
```

The following example shows an inference report for a simple three-state driver.

Three-State Device Name	Type	MB
OUT1_tri	Three-State Buffer	N

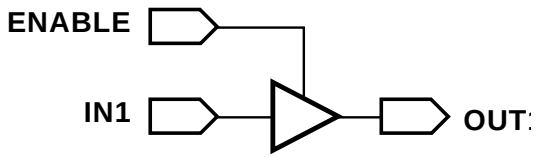


Figure 7-25 Simple Three-State Driver

Inferring One Three-State Driver from a Single Process The following example shows how to place all high-impedance assignments in a single process. In this case, the data is gated and Foundation Express infers a single three-state driver. An inference report for a single process follows the example. The figure "Inferring One Three-State Driver" shows the schematic the code generates.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity three_state is
  port ( A, B, SELA, SELB : in std_logic ;
         T : out std_logic );
end three_state;

architecture rtl of three_state is
begin
infer : process (SELA, A, SELB, B) begin
  T <= 'Z';
  if (SELA = '1') then
    T <= A;
  elsif (SELB = '1') then
    T <= B;
  end if;
end process infer;

end rtl;
```

The following example shows a single block inference report.

Three-State Device Name	Type	MB
T_tri	Three-State Buffer	N

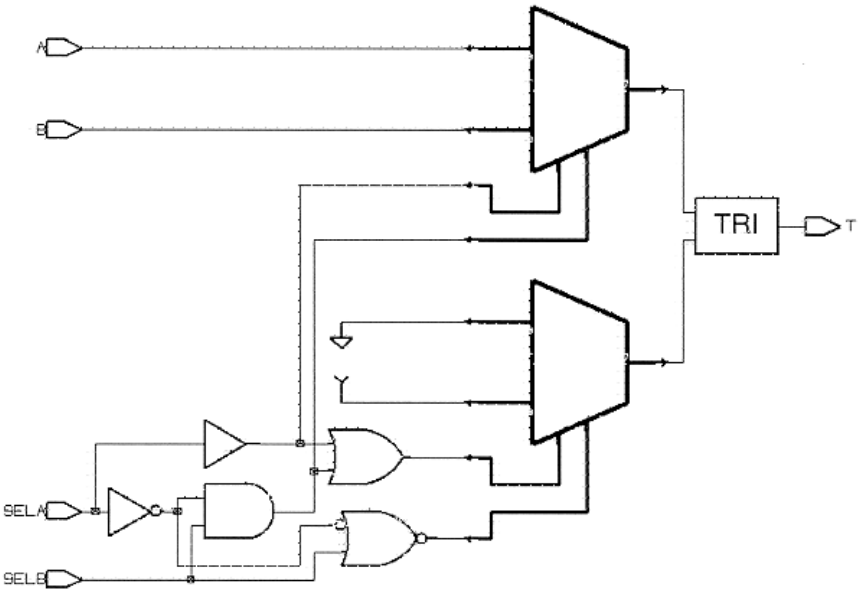


Figure 7-26 Inferring One Three-State Driver

Inferring Three-State Drivers from Separate Processes The following example shows how to place each high-impedance assignment in a separate process. In this case, Foundation Express infers multiple three-state drivers.

The inference report for two three-state drivers follows the example. The figure “Inferring Two Three-State Drivers” shows the schematic the code generates.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity three_state is
  port ( A, B, SELA, SELB : in std_logic ;
         T : out std_logic );
end three_state;

architecture rtl of three_state is
begin
infer1 : process (SELA, A) begin
  if (SELA = '1') then
    T <= A;
```

```
  else
    T <= 'Z';
  end if;
end process infer1;

infer2 : process (SELB, B) begin
  if (SELB = '1') then
    T <= B;
  else
    T <= 'Z';
  end if;
end process infer2;

end rtl;
```

The following example shows an inference report for two three-state drivers from separate processes.

Three-State Device Name	Type	MB
T_tri	Three-State Buffer	N

Three-State Device Name	Type	MB
T_tri2	Three-State Buffer	N

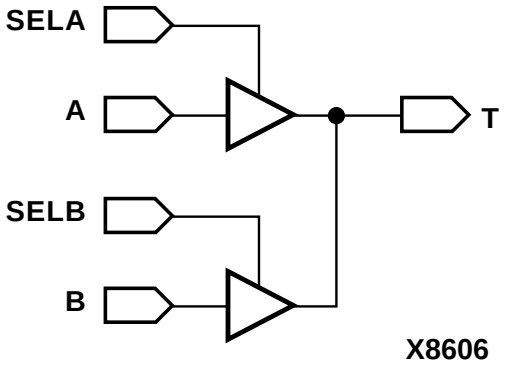


Figure 7-27 Inferring Two Three-State Drivers

Three-State Driver with Registered Enable

When a variable, such as THREE_STATE in the following example, is assigned to a register and defined as a three-state gate within the same process, Foundation Express also registers the enable pin of the three-state gate.

The following example shows this type of code, and the inference report for a three-state driver with registered enable follows the example. The figure “Three-State Driver with Registered Enable” shows the schematic the code generates, a three-state gate with a register on its enable pin.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity three_state is
   port ( DATA, CLK, THREE_STATE : in std_logic ;
          OUT1 : out std_logic );
end three_state;

architecture rtl of three_state is
begin
infer : process (THREE_STATE, CLK) begin
   if (THREE_STATE = '0') then
      OUT1 <= 'Z';
   elsif (CLK'event and CLK = '1') then
      OUT1 <= DATA;
   end if;
end process infer;

end rtl;
```

The following example shows an inference report for a three-state driver with registered enable.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
OUT1_reg	Flip-flop	1	-	-	N	N	N	N	N

Three-State Device Name	Type	MB
OUT1_tri	Three-State Buffer	N
OUT1_tr_enable_reg	Flip-Flop (width 1)	N

```
OUT1_reg
   set/reset/toggle: none
```

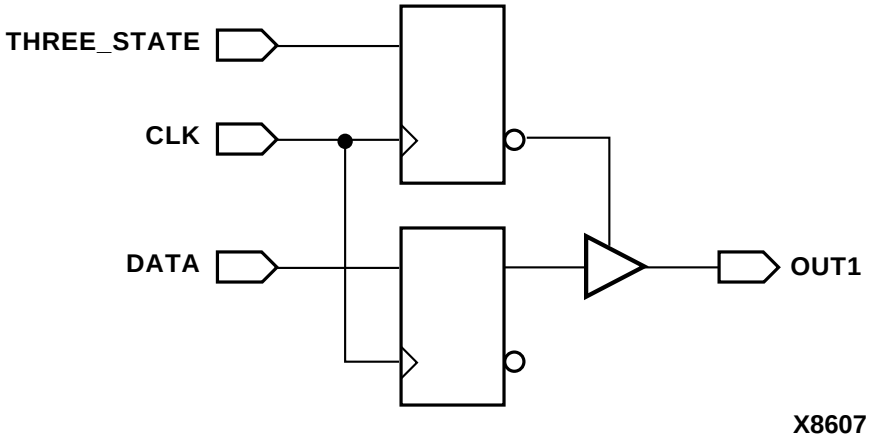


Figure 7-28 Three-State Driver with Registered Enable

Three-State Driver Without Registered Enable

The following example uses two processes to instantiate a three-state gate with a flip-flop on the input. The inference report for a three-state driver without registered enable follows the example. The figure “Three-State Driver without Registered Enable” shows the schematic the code generates.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity ff_3state2 is
   port ( DATA, CLK, THREE_STATE : in std_logic ;
          OUT1 : out std_logic );
end ff_3state2;

architecture rtl of ff_3state2 is
   signal TEMP : std_logic;
begin

process (CLK) begin
   if (CLK'event and CLK = '1') then
```

```
      TEMP <= DATA;
   end if;
end process;

process (THREE_STATE, TEMP) begin
   if (THREE_STATE = '0') then
      OUT1 <= 'Z';
   else
      OUT1 <= TEMP;
   end if;
end process;

end rtl;
```

The following example shows an inference report for a three-state driver without registered enable.

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
TEMP_reg	Flip-flop	1	-	-	N	N	N	N	N

.

Three-State Device Name	Type	MB
OUT1_tri	Three-State Buffer	N

```
TEMP_reg
   set/reset/toggle: none
```

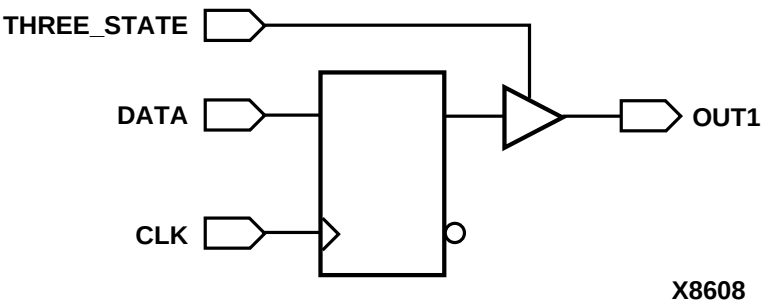


Figure 7-29 Three-State Driver without Registered Enable

Understanding the Limitations of Three-State Inference

You can use the Z value in the following ways.

- Signal assignment
- Variable assignment
- Function call argument
- Return value
- Aggregate definition

You cannot use the Z value in an expression, except for comparison with Z. Be careful when using expressions that compare with the Z value. Foundation Express always evaluates these expressions to FALSE, and the pre- and post-synthesis simulation results might differ. For this reason, Foundation Express issues a warning when it synthesizes such comparisons.

The following example shows the incorrect use of the Z value in an expression.

```
OUT_VAL = (1'bz && IN_VAL);
```

The following example shows the correct use of the Z value in an expression.

```
if (IN_VAL == 1'bz) then
```

Chapter 8

Writing Circuit Descriptions

To understand Foundation Express and to write VHDL descriptions that produce efficient synthesized circuits, study the information presented in the following sections of this chapter.

- "How Statements Are Mapped to Logic"
- "Asynchronous Designs"
- "Don't Care Inference"
- "Synthesis Issues"

Here are some general guidelines for writing efficient circuit descriptions:

- Restructure a design that makes repeated use of several large components, to minimize the number of instantiations.
- In a design that needs some, but not all, of its variables or signals stored during operation, minimize the number of latches or flip-flops required.
- Consider collapsing hierarchy for more efficient synthesis.

How Statements Are Mapped to Logic

VHDL descriptions are mapped to combinatorial logic by the creation of blocks of logic. A statement or an operator in a VHDL function can represent a block of combinatorial logic or, in some cases, a latch or register.

The statements shown in the following example represent four logic blocks.

- A comparator that compares the value of B with 10
- An adder that has A and B as inputs

- An adder that has A and 10 as inputs
- A multiplexer (implied by the if statement) that controls the final value of Y

```
if (B < 10)
   Y = A + B;
else
   Y = A+ 10;
```

The logic blocks created by Foundation Express are custom-built for their environment. That is, if A and B are 4-bit quantities, a 4-bit adder is built. If A and B are 9-bit quantities, a 9-bit adder is built. Because Foundation Express incorporates a large set of these customized logic blocks, it can translate most VHDL statements and operators.

Design Structure

A design's structure influences the size and complexity of the resulting synthesized circuit. These sections help you understand the following concepts.

- Adding Structure
- Using Design Knowledge
- Optimizing Arithmetic Expressions
- Changing an Operator Bit-Width
- Using State Information
- Propagating Constants
- Sharing Complex Operators

Adding Structure

Foundation Express gives you significant control over the preoptimization structure, or organization of components, in your design. Whether or not your design structure is preserved after optimization depends on the options you select.

Using Variables and Signals

You control design structure with your ordering of assignment statements and your use of variables. Each VHDL signal assignment,

process, or component instantiation implies a piece of logic. Each variable or signal implies a wire. By using these constructs, you can connect entities in any configuration.

The following two examples show two possible descriptions of an adder's carry chain. The figure following the examples illustrates the resulting design.

```
-- A is the addend
-- B is the augend
-- C is the carry
-- Cin is the carry in
C0 <= (A0 and B0) or
      ((A0 or B0) and Cin);
C1 <= (A1 and B1) or
      ((A1 or B1) and C0);
```

The following example shows a carry-lookahead chain.

```
-- Ps are propagate
-- Gs are generate
p0 <= a0 or b0;
g0 <= a0 and b0;
p1 <= a1 or b1;
g1 <= a1 and b1;
c0 <= g0 or (p0 and cin);
c1 <= g1 or (p1 and g0) or
      (p1 and p0 and cin);
```

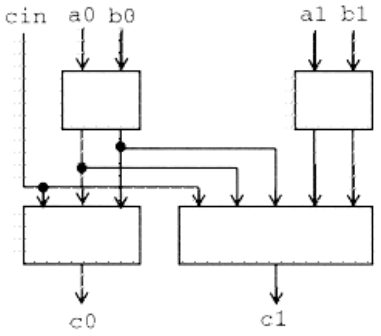


Figure 8-1 Ripple Carry and Carry-Lookahead Chain Design

Using Parentheses

Another way to control the structure of a design is to use parentheses to define logic groupings. The following example describes a 4-input adder grouping. The figure following the example illustrates the resulting design.

```
Z <= (A + B) + C + D;
```

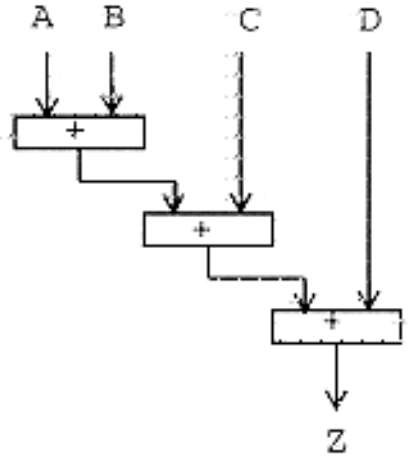


Figure 8-2 Diagram of 4-Input Adder

The following example describes a 4-input adder grouping that is structured with parentheses. The figure following the example illustrates the design.

```
Z <= (A + B) + (C + D);
```

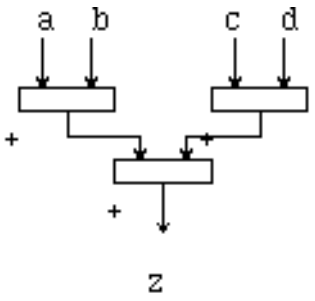


Figure 8-3 Diagram of 4-Input Adder With Parentheses

Using Design Knowledge

In many circumstances, you can improve the quality of synthesized circuits by better describing your high-level knowledge of a circuit. Foundation Express cannot always derive details of a circuit architecture. Any additional architectural information you can provide to Foundation Express can result in a more efficient circuit.

Optimizing Arithmetic Expressions

Foundation Express uses the properties of arithmetic operators (such as the associative and commutative properties of addition) to rearrange an expression so that it results in an optimized implementation. You can also use arithmetic properties to control the choice of implementation for an expression. Three forms of arithmetic optimization are discussed in this section.

- Arranging Expression Trees for Minimum Delay
- Sharing Common Subexpressions

Arranging Expression Trees for Minimum Delay

If your goal is to speed up your design, arithmetic optimization can minimize the delay through an expression tree by rearranging the sequence of the operations. Consider the statement in the following example.

```
Z <= A + B + C + D;
```

The parser performs each addition in order, as though parentheses were placed within the expression as follows.

```
Z <= ((A + B) + C) + D);
```

The parser constructs the expression tree shown in the following figure.

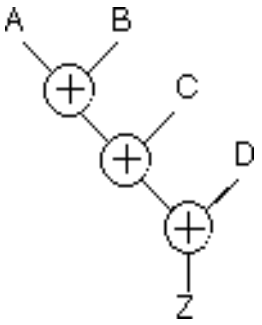


Figure 8-4 Default Expression Tree

Considering Signal Arrival Times To determine the delay through an expression tree, Foundation Express considers the arrival times of each signal in the expression. If the arrival times of all the signals are the same, the length of the critical path of the expression in the previous example of a simple arithmetic expression equals three adder delays. The critical path delay can be reduced to two adder delays if you insert parentheses as follows.

```
Z <= (A + B) + (C + D);
```

The parser constructs the subexpression tree as shown in the following figure.

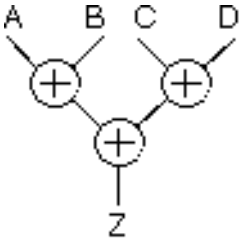


Figure 8-5 Balanced Adder Tree (Same Arrival Times for All Signals)

Suppose signals B, C, and D arrive at the same time and signal A arrives last. The expression tree that produces the minimum delay is shown in the following figure.

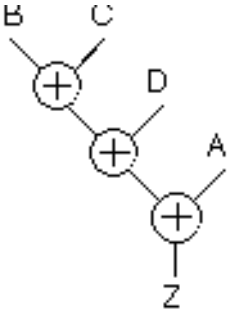


Figure 8-6 Expression Tree With Minimum Delay (Signal A Arrives Last)

Using Parentheses You can use parentheses in expressions to exercise more control over the way expression trees are constructed. Parentheses are regarded as user directives that force an expression tree to use the groupings inside the parentheses. The expression tree cannot be rearranged in a way that violates these groupings.

To illustrate the effect of parentheses on the construction of an expression tree, consider the following example.

```
Q <= ((A + (B + C)) + D + E) + F;
```

The parentheses in the expression in the above example define the following subexpressions.

```
1 (B + C)
2 (A + (B + C))
3 ((A + (B + C)) + D + E)
```

These subexpressions must be preserved in the expression tree. The default expression tree for the above example is shown in the following figure.

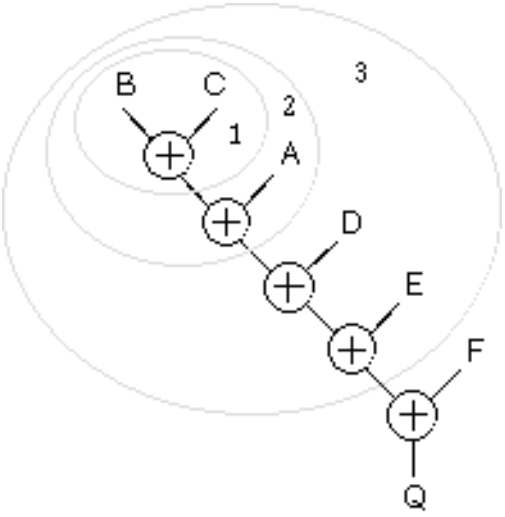


Figure 8-7 Expression Tree With Subexpressions Dictated by Parentheses

Considering Overflow Characteristics When Foundation Express performs arithmetic optimization, it considers how to handle the overflow from carry bits during addition. The optimized structure of an expression tree is affected by the bit-widths you declare for storing intermediate results. For example, suppose you write an expression that adds two 4-bit numbers and stores the result in a 4-bit register. If the result of the addition overflows the 4-bit output, the most significant bits are truncated. The following example shows how Foundation Express handles overflow characteristics.

```
t <= a + b;     --a and b are 4-bit numbers
z <= t + c;     --c is a 6-bit number
```

In the above example, three variables are added (a + b + c). A temporary variable, t, holds the intermediate result of a + b. If t is declared as a 4-bit variable, the overflow bits from the addition of a + b are truncated. The parser determines the default structure of the expression tree, which is shown in the following figure.

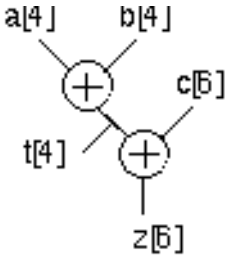


Figure 8-8 Default Expression Tree With 4-Bit Temporary Variable

Now suppose the addition is performed without a temporary variable (z = a + b + c). Foundation Express determines that five bits are needed to store the intermediate result of the addition, so no overflow condition exists. The results of the final addition might be different from the first case, where a 4-bit temporary variable is declared that truncates the result of the intermediate addition. Therefore, these two expression trees do not always yield the same result. The expression tree for the second case is shown in the following figure.

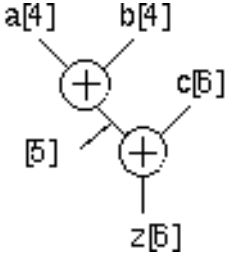


Figure 8-9 Expression Tree With 5-Bit Intermediate Result

Sharing Common Subexpressions

Subexpressions consist of two or more variables in an expression. If the same subexpression appears in more than one equation, you might want to share these operations to reduce the area of your circuit.

You can force common subexpressions to be shared by declaring a temporary variable to store the subexpression, then use the temporary variable wherever you want to repeat the subexpression. The following example shows a group of simple additions that use the common subexpression (a + b).

```
temp <= a + b;
x <= temp;
y <= temp + c;
```

Instead of manually forcing common subexpressions to be shared, you can let Foundation Express automatically determine whether sharing common subexpressions improves your circuit. You do not need to declare a temporary variable to hold the common subexpression in this case.

In some cases, sharing common subexpressions results in more adders being built. Consider the following example, where A + B is a common subexpression.

```
if cond1
   Y <= A + B;
else
   Y <= C + D;
end;
if cond2
   Z <= E + F;
else
   Z <= A + B;
end;
```

If the common subexpression A + B is shared, three adders are needed to implement this section of code.

```
(A + B)
(C + D)
(E + F)
```

If the common subexpression is not shared, only two adders are needed: one to implement the additions A + B and C + D and one to implement the additions E + F and A + B.

Foundation Express analyzes common subexpressions during the resource sharing phase of the compile process and considers area costs and timing characteristics. To turn off the sharing of common subexpressions for the current design, use the constraint manager. The default is TRUE.

```
Y <= A + B + C;
Z <= D + A + B;
```

The parser does not recognize A + B as a common subexpression, because it parses the second equation as (D + A) + B. You can force the parser to recognize the common subexpression by rewriting the second assignment statement as follows.

```
Z <= A + B + D;
```

or

```
Z <= D + (A + B);
```

Note: You do not have to rewrite the assignment statement, because Foundation Express recognizes common subexpressions automatically.

Changing an Operator Bit-Width

The adder in the following example sums the 8-bit value of A (a BYTE) with the 8-bit value of TEMP. TEMP's value is either B, which is used only when it is less than 16, or C, which is a 4-bit value (a NIBBLE).Therefore, the upper four bits of TEMP are always 0. Foundation Express cannot derive this fact, because TEMP is declared with type BYTE.

You can simplify the synthesized circuit by changing the declared type of TEMP to NIBBLE (a 4-bit value). With this modification, half adders, rather than full adders, are required to implement the top four bits of the adder circuit, which figure, "Function with One Adder Schematic," illustrates.

```
function ADD_IT_16 (A, B: BYTE; C: NIBBLE) return BYTE is
   variable TEMP: BYTE;
begin
   if B < 16 then
      TEMP <= B;
   else
      TEMP <= C;
    end if;
  return A + TEMP;
end;
```

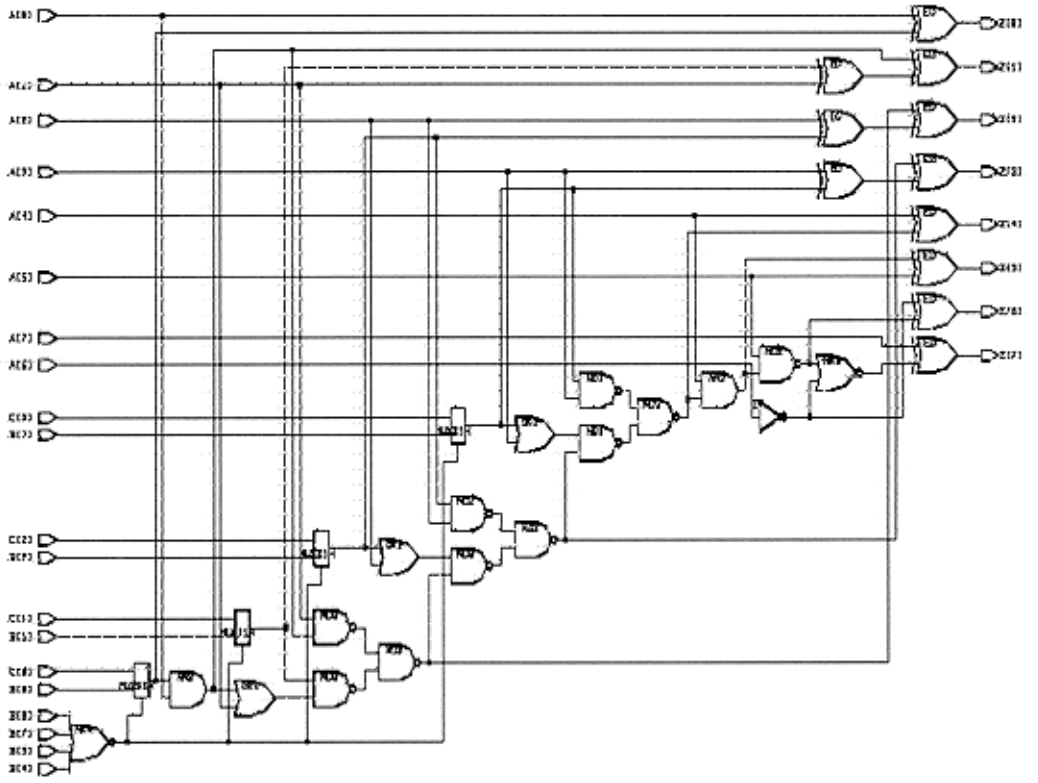


Figure 8-10 Function With One Adder Schematic

The following example shows how this change in TEMP's declaration can yield a significant savings in circuit area, which the figure following the example illustrates.

```
function ADD_IT_16 (A, B: BYTE; C: NIBBLE)
    return BYTE is
  variable TEMP: NIBBLE;   -- Now only 4 bits
begin
  if B < 16 then
    TEMP <= NIBBLE(B);     -- Cast BYTE to NIBBLE
  else
    TEMP <= C;
  end if;
  return A + TEMP;         -- Single adder
end;
```

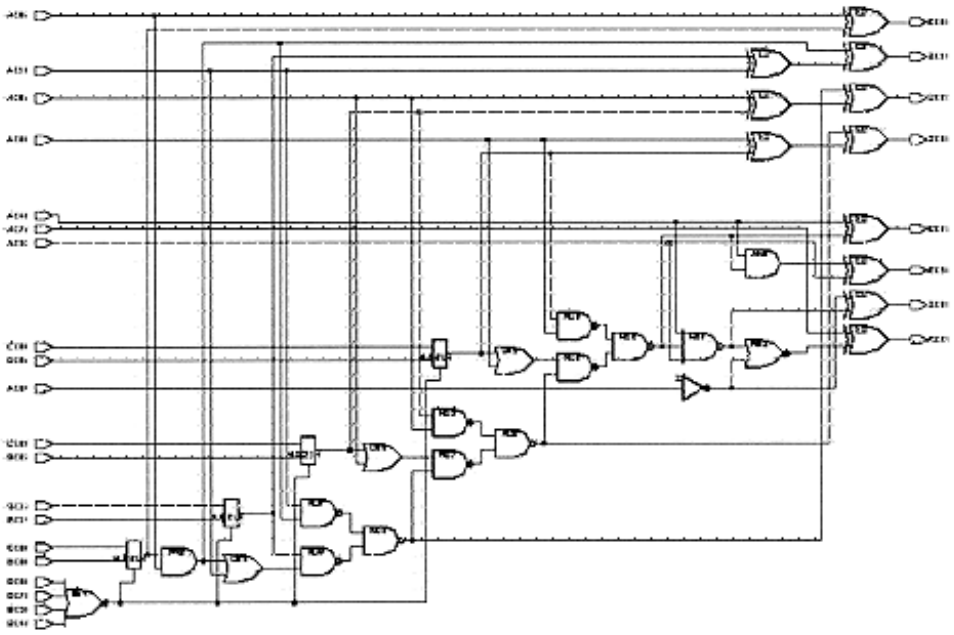


Figure 8-11 Using TEMP Declaration to Save Circuit Area

Using State Information

You can also apply design knowledge in sequential designs. Often you can make strong assertions about the value of a signal in a particular state of a finite-state machine. You can describe this information to Foundation Express. The following example shows the VHDL description of a simple state machine that uses two processes.

```
package STATES is
  type STATE_TYPE is (SET0, HOLD0, SET1);
end STATES;

use work.STATES.all;

entity MACHINE is
  port(X, CLOCK: in BIT;
       CURRENT_STATE: buffer STATE_TYPE;
       Z: buffer BIT);
end MACHINE;
```

```
architecture BEHAVIOR of MACHINE is
  signal NEXT_STATE: STATE_TYPE;
  signal PREVIOUS_Z: BIT;begin


  -- Process to hold combinatorial logic.
  COMBIN: process(CURRENT_STATE, X, PREVIOUS_Z)
  begin
    case CURRENT_STATE is
      when SET0 =>
        Z <= '0';                 -- Set Z to '0'
        NEXT_STATE <= HOLD0;

      when HOLD0 =>
        Z <= PREVIOUS_Z;          -- Hold value of Z
        if X = '0' then
          NEXT_STATE <= HOLD0;
        else
          NEXT_STATE <= SET1;
        end if;

      when SET1 =>                -- Set Z to '1'
        Z <= '1';
        NEXT_STATE <= SET0;
    end case;
  end process COMBIN;

  -- Process to hold synchronous elements (flip-
flops).
  SYNCH: process
  begin
    wait until CLOCK'event and CLOCK = '1';
    CURRENT_STATE <= NEXT_STATE;
    PREVIOUS_Z <= Z;
  end process SYNCH;
end BEHAVIOR;
```

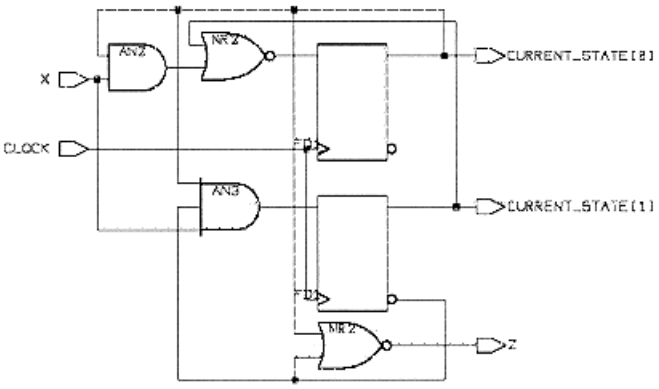


Figure 8-12 Schematic of Simple State Machine with Two Processes

The following figure shows a schematic of a simple state machine with two processes.In the state hold0, the output Z retains its value from the previous state. To accomplish this, you insert a flip-flop to hold the PREVIOUS_Z. However, you can make some assertions about the value of Z. In state HOLD0, the value of Z is always 0. You can deduce this from the fact that the state HOLD0 is entered only from the state SET0, where Z is always assigned '0.'

The following example shows how you can change the VHDL description to use this assertion, resulting in a simpler circuit. The figure following the example illustrates the circuit.

```
package STATES is
  type STATE_TYPE is (SET0, HOLD0, SET1);
end STATES;
use work.STATES.all;

entity MACHINE is
  port(X, CLOCK: in BIT;
       CURRENT_STATE: buffer STATE_TYPE;
       Z: buffer BIT);
end MACHINE;

architecture BEHAVIOR of MACHINE is
  signal NEXT_STATE: STATE_TYPE;
begin
  -- Combinatorial logic.
  COMBIN: process(CURRENT_STATE, X)
```

```
  begin
    case CURRENT_STATE is
      when SET0 =>
        Z <= '0';                  -- Set Z to '0'
        NEXT_STATE <= HOLD0;
      when HOLD0 =>
        Z <= '0';                  -- Hold Z at '0'
        if X = '0' then
          NEXT_STATE <= HOLD0;
        else
          NEXT_STATE <= SET1;
        end if;
      when SET1 =>                 -- Set Z to '1'
        Z <= '1';
        NEXT_STATE <= SET0;
    end case;
  end process COMBIN;
  -- Process to hold synchronous elements (flip-flops)
  SYNCH: process
  begin
    wait until CLOCK'event and CLOCK = '1';
    CURRENT_STATE <= NEXT_STATE;
  end process SYNCH;
end BEHAVIOR;
```

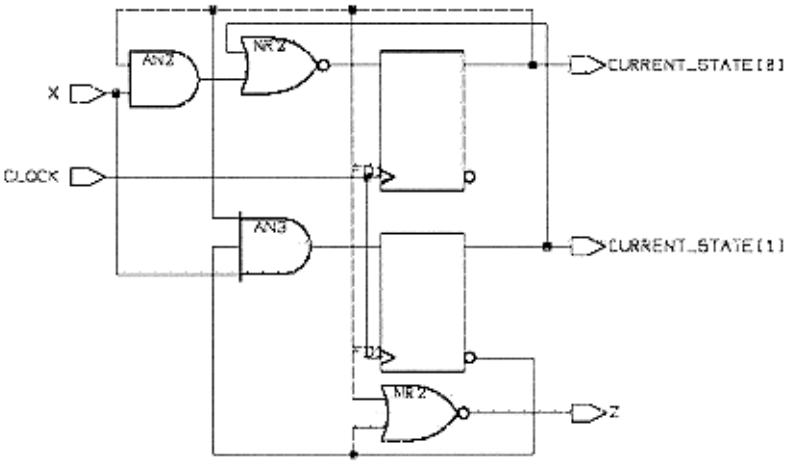


Figure 8-13 Schematic of an Improved State Machine

Propagating Constants

Constant propagation is the compile-time evaluation of expressions containing constants. Foundation Express uses constant propagation to reduce the amount of hardware required to implement operators. For example, a + operator with a constant 1 as one of its arguments causes an incrementer to be built, rather than a general adder. If both arguments of + or any other operator are constants, no hardware is constructed, because the expression's value is calculated by Foundation Express and inserted directly in the circuit.

Other operators that benefit from constant propagation include comparators and shifters. Shifting a vector by a constant amount requires no logic to implement; it requires only a reshuffling (rewiring) of bits.

Sharing Complex Operators

The efficiency of a synthesized design depends primarily on how you describe its component structure. The optimization of individual components, especially those made from random logic, produces similar results from two very different descriptions. Therefore, concentrate the majority of your design effort on the implied component hierarchy (as discussed in the preceding sections) rather than on the logical descriptions. The "Design Descriptions" chapter discusses how to define a VHDL design hierarchy.

Foundation Express supports many shorthand VHDL expressions. There is no benefit to using a verbose syntax when a shorter description is adequate. The following example shows four equivalent groups of statements.

```
signal A, B, C: BIT_VECTOR(3 downto 0);
  . . .
  C <= A and B;
-------------------------------------
  C(3 downto 0) <= A(3 downto 0) and B(3 downto 0);
-------------------------------------
  C(3) <= A(3) and B(3);
  C(2) <= A(2) and B(2);
  C(1) <= A(1) and B(1);
  C(0) <= A(0) and B(0);
-------------------------------------
  for I in 3 downto 0 loop
```

```
    C(I) <= A(I) and B(I);
  end loop;
```

Asynchronous Designs

In a synchronous design, all flip-flops use a single clock that is a primary input to the design and there are no combinatorial feedback paths. Synchronous designs perform the same function regardless of the clock rate if all signals can propagate through the design's combinatorial logic during the clock's cycle time.

Foundation Express treats all designs as synchronous. It can therefore change the timing behavior of the combinatorial logic if the maximum and minimum delay requirements are met.

Foundation Express always preserves the Boolean function computed by logic, assuming that the clock arrives after all signals have propagated. Foundation Express' built-in timing verifier helps determine the slowest path (critical path) through the logic, which determines how fast the clock can run.

Foundation Express provides some support for asynchronous designs, but you must assume a greater responsibility for the accuracy of your circuits. Although fully synchronous circuits usually agree with their simulation models, asynchronous circuits might not. Foundation Express might not warn you when a design is not fully synchronous. Be aware of the possibility of asynchronous timing problems.

The most common way to produce asynchronous logic in VHDL is to use gated clocks on latches or flip-flops. The following figure shows a fully synchronous design, a counter with synchronous ENABLE and RESET inputs. Because it is synchronous, this counter works if the clock speed is slower than the critical path. The figure following the example illustrates the design.

```
entity COUNT is
  port(RESET, ENABLE, CLK: in     BIT;
       Z:                 buffer INTEGER range 0 to 7);
end;
architecture ARCH of COUNT is
begin
  process(RESET, ENABLE, CLK, Z)
  begin
    if (CLK'event and CLK = '1') then
```

```
        if (RESET = '1') then        -- occurs on clock
                                     --edge
           Z <= 0;
        elsif (ENABLE = '1') then    -- occurs on clock
                                     --edge
          if (Z = 7) then
            Z <= 0;
          else
            Z <= Z + 1;
          end if;
        end if;
      end if;
    end process;
  end ARCH;
```

The schematic for the synchronous counter is shown in the following figure.

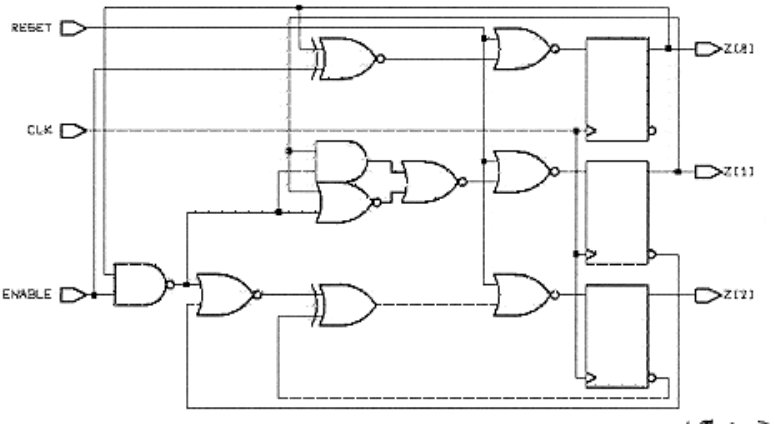


Figure 8-14 Schematic of Synchronous Counter with Reset and Enable

The following example shows an asynchronous version of the design in the previous example. The version in the following example uses two common asynchronous design techniques.

- The first technique, shown in the example of a better implementation of a state machine, enables the counter by using an AND gate on the clock and enable signals.
- The second technique, shown in the example of four equivalent groups of statements, uses an asynchronous reset.

These techniques work only when the proper timing relationships exist between the reset signal (RESET) and the clock signal (CLK) and there are no glitches in these signals.

The following example shows a design with gated clock and asynchronous reset.

```
entity COUNT is
  port(RESET, ENABLE, CLK: in     BIT;
       Z:                 buffer INTEGER range 0 to 7);
end;

architecture ARCH of COUNT is
  signal GATED_CLK: BIT;
begin
  GATED_CLK <= CLK and ENABLE; -- clock gated by
ENABLE

  process(RESET, GATED_CLK, Z)
  begin
    if (RESET = '1') then      -- asynchronous reset
      Z <= 0;
    elsif (GATED_CLK'event and GATED_CLK = '1') then
      if (Z = 7) then
        Z <= 0;
      else
        Z <= Z + 1;
      end if;
    end if;
  end process;
end ARCH;
```

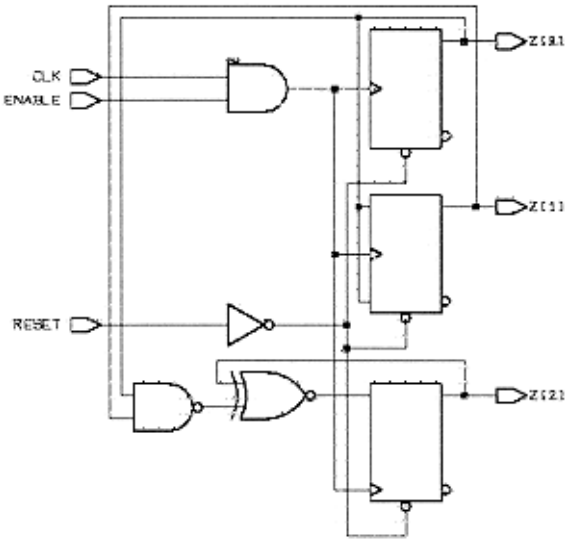


Figure 8-15 Design with AND Gate on Clock and Enable Signals

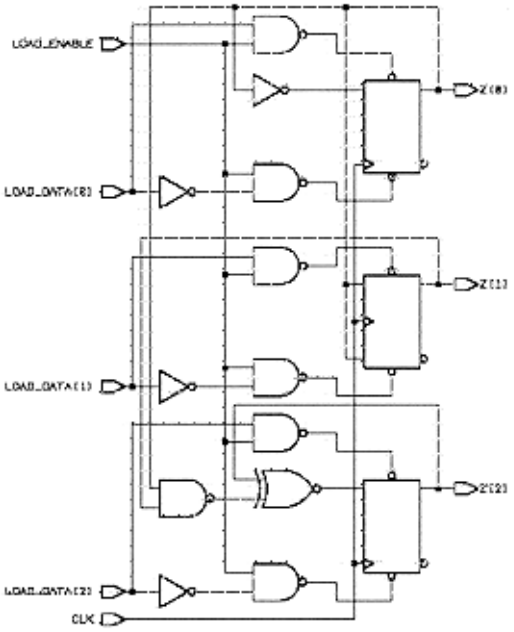


Figure 8-16 Design with Asynchronous Reset

The following example shows an asynchronous design that might not work, because Foundation Express does not guarantee that the combinatorial logic it builds has no hazards (glitches).

```
entity COUNT is
  port(LOAD_ENABLE, CLK: in     BIT;
       LOAD_DATA:        in     INTEGER range 0 to 7;
       Z:                buffer INTEGER range 0 to 7);
end;

architecture ARCH of COUNT is

begin
  process(LOAD_ENABLE, LOAD_DATA, CLK, Z)
  begin
    if (LOAD_ENABLE = '1') then
      Z <= LOAD_DATA;
    elsif (CLK'event and CLK = '1') then
      if (Z = 7) then
        Z <= 0;
      else
```

```
        Z <= Z + 1;
      end if;
    end if;
  end process;
end ARCH;
```

The design in the previous example works only when the logic driving the preset and clear pins of the flip-flops that hold Z is faster than the clock speed. If you use this design style, you must simulate the synthesized circuit thoroughly. You also need to inspect the synthesized logic, because potential glitches might not appear in simulation. For a safer design, use a synchronous LOAD_ENABLE.

A design synthesized with complex logic driving the gate of a latch rarely works. The following example describes an asynchronous design that never works. The figure following the example shows the resulting schematic.

```
entity COMP is
  port(A, B: in     INTEGER range 0 to 7;
       Z:    buffer INTEGER range 0 to 7);
end;
architecture ARCH of COMP is
begin
  process(A, B)
  begin
    if (A = B) then
      Z <= A;
    end if;
  end process;
end ARCH;
```

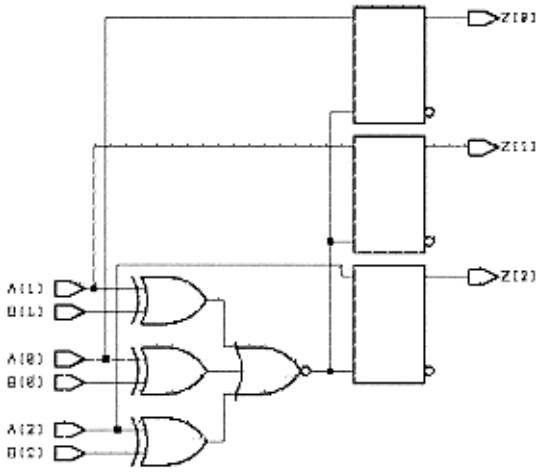


Figure 8-17 Schematic of Incorrect Asynchronous Design

In the previous example and figure, the comparator's output latches the value A onto the value Z. This design might work under behavioral simulation where the comparison happens instantly. However, the hardware comparator generates glitches that cause the latches to store new data when they should not.

Don't Care Inference

You can greatly reduce circuit area by using don't care values. To use a don't care value in your design, create an enumerated type for the don't care value.

Don't care values are best used as default assignments to variables. You can assign a don't care value to a variable at the beginning of a module, in the default section of a case statement, or in the else section of an if statement.

The following example shows don't care encoding for a seven-segment LED decoder. Enumeration encoding 'D' represents the don't care state. The figure following the example illustrates the design.

```
entity CONVERTER is
  port (BCD: in BIT_VECTOR(3 downto 0);
        LED: out BIT_VECTOR(6 downto 0));
```

```
  -- pragma dc_script_begin
  -- set_flatten true
  -- pragma dc_script_end
end CONVERTER;


architecture BEHAVIORAL of CONVERTER is
begin
CONV: process(BCD)
  begin
     case BCD is
       when "0000" => LED <= "1111110";
       when "0001" => LED <= "1100000";
       when "0010" => LED <= "1011011";
       when "0011" => LED <= "1110011";
       when "0100" => LED <= "1100101";
       when "0101" => LED <= "0110111";
       when "0110" => LED <= "0111111";
       when "0111" => LED <= "1100010";
       when "1000" => LED <= "1111111";
       when "1001" => LED <= "1110111";
       when others => LED <= "0000000";
     end case;
  end process CONV;
end BEHAVIORAL;
```

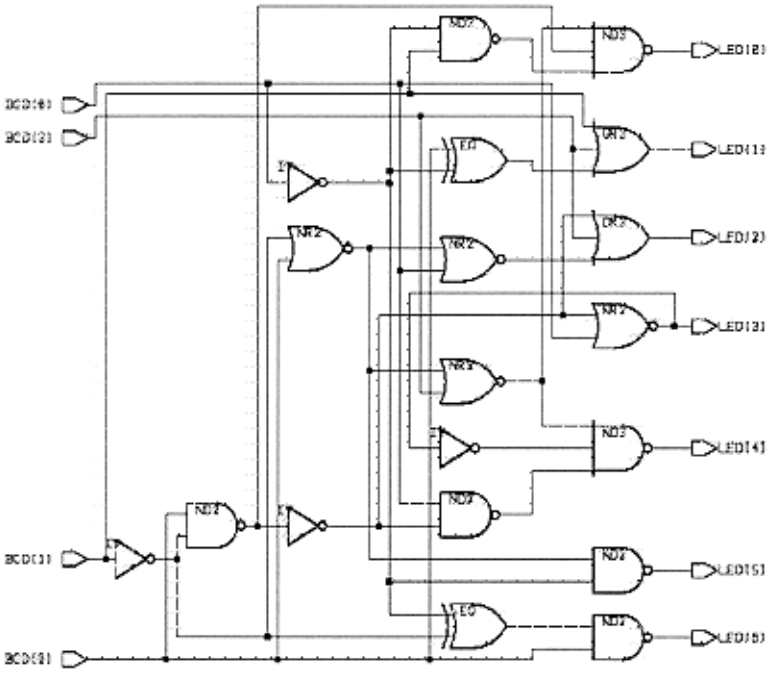


Figure 8-18 Seven-Segment LED Decoder with O LED Default

Using Don't Care Default Values

You do not always want to assign a default value or don't care, although it can be beneficial in some cases, as the seven-segment decoder in the previous example shows.

The reasons for not always defaulting to don't care follow.

- The potential for mismatches between simulation and synthesis is greater.
- Defaults for variables can hide mistakes in the VHDL code.

  For example, you might assign a default don't care value to VAR. If you later assign a value to VAR, expecting VAR to be a don't care value, you might have overlooked an intervening condition under which VAR is assigned.

Therefore, when you assign a value to a variable (or signal) that contains a don't care value, make sure that the variable (or signal) is really a don't care value under those conditions.

Differences Between Simulation and Synthesis

Don't care values are treated differently in simulation and in synthesis, and there can be a mismatch between the two. To a simulator, a don't care is a distinct value, different from a 1 or a 0. In synthesis, however, a don't care becomes a 0 or a 1 (and the hardware built treats the don't care value as either a 0 or a 1).

Whenever a comparison is made with a variable whose value is don't care, simulation and synthesis can differ. The safest way to use don't care values is to do the following.

- Assign don't care values only to output ports
- Make sure that the design never reads output ports

These guidelines guarantee that when you simulate within the scope of the design, the only difference between simulation and synthesis occurs when the simulator defines an output as a don't care.

Note: If you use don't care values internally to a design, expressions compared to don't care ('D') are synthesized as though their values are not equal to 'D.'

For example,

```
if X = 'D' then
...
```

is synthesized as

```
if FALSE then
```

If you use expressions comparing values with 'D,' there might be a difference between pre-synthesis and post-synthesis simulation results. For this reason, Foundation Express issues a warning when it synthesizes such comparisons.

```
Warning: A partial don't-care value was read in
routine test line 24 in file 'test.vhdl'  This may
cause simulation to disagree with synthesis. (HDL-171)
```

Synthesis Issues

Feedback paths and latches result from ambiguities in signal or variable assignments and language supersets or the differences between a VHDL simulator view and the Foundation Express use of VHDL.

Feedback Paths and Latches

Implied combinatorial feedback paths or latches in synthesized logic can occur when a signal or variable in a combinatorial process (one without a wait or if signal'event statement) is not fully specified in the VHDL description. A variable or signal is fully specified when it is assigned under all possible conditions. A variable or signal is not fully specified when a condition exists under which the variable is not assigned.

Fully Specified Variables

The following example shows several variables. A, B, and C are fully specified; X is not.

```
process (COND1)
  variable A, B, C, X : BIT;
begin
  A <= '0'     -- A is hereby fully specified
  C <= '0'     -- C is hereby fully specified


  if (COND1) then
    B <= '1';    -- B is assigned when COND1 is TRUE
    C <= '1';    -- C is already fully specified
    X <= '1';    -- X is assigned when COND1 is TRUE
  else
    B <= '0';    -- B is assigned when COND1 is FALSE
  end if;
  -- A is assigned regardless of COND1, so A is fully
  --   specified.


  -- B is assigned under all branches of if (COND1),
  --   that is, both when COND1 is TRUE and when
  --   COND1 is FALSE, so B is fully specified.


  -- C is assigned regardless of COND1, so C is fully
  -- specified.  (The second assignment to C does
```

```
    -- not change this.)


    -- X is not assigned under all branches of
    --   if (COND1), namely, when COND1 is FALSE,
    --   so X is not fully specified.
end process;
```

The conditions of each if and else statement are considered independent in the previous example. A is considered not fully specified in the following fragment.

```
if (COND1) then
  A <= '1';
end if;

if (not COND1) then
  A <= '0';
end if;
```

A variable or signal that is not fully specified in a combinatorial process is considered conditionally specified. In this case, a flow-through latch is implied. You can conditionally assign a variable, but you cannot read a conditionally specified variable. You can, however, both conditionally assign and read a signal.

If a fully specified variable is read before its assignment statements, combinatorial feedback might exist. For example, the following fragment synthesizes combinatorial feedback for VAL.

```
process(NEW, LOAD)
  variable VAL: BIT;
begin
  if (LOAD) then
    VAL <= NEW;
  else
    VAL <= VAL;
  end if;

  VAL_OUT <= VAL;
end process;
```

In a combinatorial process, you can ensure that a variable or signal is fully specified by providing an initial (default) assignment to the variable at the beginning of the process. This default assignment assures that the variable is always assigned a value, regardless of conditions. Subsequent assignment statements can override the

default. A default assignment is made to variables A and C in the example of fully specified variables.

Another way to ensure that you do not imply combinatorial feedback is to use a sequential process (one with a wait or if signal'event statement). In such a case, variables and signals are registered. The registers break the combinatorial feedback loop.

See the "Register and Three-State Inference" chapter for more information about sequential processes and the conditions under which Foundation Express infers registers and latches.

Asynchronous Behavior

Some forms of asynchronous behavior are not supported. An example is a circuit description of a one-hot signal generator of the following form.

```
X <= A nand (not(not(not A)));
```

You might expect this circuit description to generate three inverters (an inverting delay line) and a NAND gate, but it is optimized to the following.

```
X <= A nand (not A);
```

Then, it is optimized to the following.

```
X <= 1;
```

```
c[0] = a[0] & b[0];

for (i = 0; i <= 3; i = i + 1)
   c[i] = a[i] & b[i];
```

Understanding Superset Issues and Error Checking

The Foundation Express VHDL Analyzer is a full IEEE 1076 VHDL analyzer.

When Foundation Express reads in a VHDL design, it first calls the VHDL Analyzer to check the VHDL source for errors and then translates the VHDL source to an intermediate form for synthesis. If an error is in the VHDL source, you get a VHDL Analyzer message and possibly a VHDL Compiler message.

VHDL Compiler allows globally static objects where only locally static objects are allowed, without issuing an error message.

Chapter 9

Foundation Express Directives

The Foundation Express HDL compiler has several methods of writing circuit design information directly in your VHDL source code.

Using Foundation 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.

To familiarize yourself with Foundation Express directives, consider the following topics presented in this chapter.

- "Notation for Foundation Express Directives"
- "Foundation Express Directives"

Notation for Foundation Express Directives

Foundation Express directives are special (synthetic) VHDL comments that affect the actions of Foundation Express. These comments are a special case of regular VHDL comments, which are ignored by other VHDL tools. Synthetic comments are used only to direct the actions of Foundation Express.

Synthetic comments begin with two hyphens (--) like a regular comment. If the word following these characters is *pragma* or *synopsys*, Foundation Express treats the remaining comment text as a directive.

Note: Foundation Express displays a syntax error if an unrecognized directive is encountered after -- synopsys or -- pragma.

Foundation Express Directives

The three types of directives follow.

- Translation stop and start directives

```
-- pragma synthesis_off
-- pragma synthesis_on
```

```
-- pragma translate_off   Use not recommended.
-- pragma translate_on    Use not recommended.
```

- 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
```

Translation Stop and Start Pragma Directives

Foundation Express supports the synthesis_off and synthesis_on pragma directives.

Note: It is recommended that you not use the following directives.

```
-- pragma translate_off
-- pragma translate_on
```

The use of these directives in Foundation Express can lead to errors in your design.

synthesis_off and synthesis_on Directives

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 example below shows how you can use the directives to protect a simulation driver.

```
-- The following test driver for entity EXAMPLE
-- should not be translated:
```

```
-- pragma synthesis_off
-- Translation stops

entity DRIVER is
end DRIVER;
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 VHDL;

-- 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 EXAMPLE;

architecture VHDL of EXAMPLE is
begin
    SUM <= A + B;
end VHDL;
```

Resolution Function Directives

Resolution function directives determine the resolution function associated with resolved signals. (See the “Resolution Functions” section of the “Design Descriptions” chapter.) Foundation Express

does not 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. Foundation 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 the "Procedures and Functions as Design Components" section of the "Sequential Statements" chapter.

```
-- pragma map_to_entity entity_name
-- pragma return_port_name port_name
```

Chapter 10

Foundation Express Packages

Three VHDL packages are included with this release. This chapter discusses the contents of each package. Each section of this chapter explains one of these packages.

- "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

- "numeric_std Package"

  The numeric_std package is an alternative to the std_logic_arith package. It is the IEEE standard 1076.3-1997, and documentation about it is available from IEEE.

- "std_logic_misc Package"

  Defines supplemental types, subtypes, constants, and functions for the std_logic_1164 package.

- "ATTRIBUTES Package"

  Declares synthesis attributes and the resource sharing subtype and its attributes.

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 package. Furthermore, mathematics, primitives, and timing standards are considered orthogonal issues as they relate to this package and are, therefore, beyond its scope.

The std_logic_1164 package has been updated with Foundation Express synthesis directives.

To use this package in a VHDL source file, include the following lines at the beginning of the source file.

```
library IEEE;
use IEEE.std_logic_1164.all;
```

When you analyze your VHDL source file, Foundation 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 Foundation Express 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. With this package, you can perform arithmetic operations and numeric comparisons on array data types. The package defines some arithmetic operators (+, -, *, and ABS) and the relational operators (<, >, <=, >=, =, and /=). (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. Find details in the “Data Types” section of this chapter. The std_logic_arith package is legal VHDL; you can use it for both synthesis and simulation.

You can configure the std_logic_arith package to work on any array of single-bit types. You encode single-bit types in 1 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.

The table below 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

To use this package in a VHDL source file, include the following lines at the beginning of the source file.

```
library IEEE;
use IEEE.std_logic_arith.all;
```

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 the following example. 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.

```
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 MLV TO INTEGER;
```

Notice how the CONV_INTEGER function is used in the above example.

Foundation 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 of the function 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 of the function.

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.

UNSIGNED

The UNSIGNED data type represents an unsigned numeric value. Foundation 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 by the following.

```
UNSIGNED'("1000")
```

When you declare variables or signals of type UNSIGNED, a larger vector holds a larger number. A 4-bit variable holds values up to decimal 15; an 8-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.

The following example illustrates some UNSIGNED declarations. The most significant bit is the farthest-left array bound, rather than the high or low range value.

```
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. Foundation 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 by the following.

```
SIGNED'("0101")  -- represents +5
SIGNED'("1011")  -- represents -5
```

When you declare SIGNED variables or signals, a larger vector holds a larger number. A 4-bit variable holds values from -8 to 7; an 8-bit variable holds values from –128 to 127. Notice that a SIGNED value cannot hold as large a value as an UNSIGNED value with the same bit-width.

The following example shows some SIGNED declarations. The sign bit is the farthest-left bit, rather than the highest or lowest.

```
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.

The following example shows the declarations of these conversion functions, with BIT and BIT_VECTOR types.

```
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;
```

There are four versions of each conversion function.

The VHDL 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 each 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. Foundation 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. An example follows.

```
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. An example follows.

```
CONV_SIGNED(UNSIGNED'("1101010"), 3) % "010"
```

Arithmetic Functions

The std_logic_arith package provides arithmetic functions for use with combinations of Xilinx's 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 the following examples.

Example 10-1: 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 10-2: Unary Arithmetic Functions

```
function "+"(L: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED)   return SIGNED;
function "-"(L: SIGNED)   return SIGNED;
function "ABS"(L: SIGNED) return SIGNED;
```

These unary arithmetic functions in the previous two examples determine the width of their return values as follows.

- When only one UNSIGNED or SIGNED argument is present, the width of the return value is the same as that argument's.
- 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 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 the following table.

```
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 the example below.

```
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 with each other and with the predefined type INTEGER. Foundation Express compares the numeric values of the arguments, returning a BOOLEAN value. For example, the following expression evaluates 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 the “Relational Operators” section of the “Expressions” chapter.)

These functions produce comparators. The function declarations are listed in two groups, ordering functions (<, <=, >, and >=) and equality functions (= and /=) in the following examples.

Example 10-3: 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 10-4: 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. See the following example for shift function declarations. For a list of shift and rotate operators, see the "Operators" section of the "VHDL Constructs" chapter.

```
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. The 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.

The following example shows some shift function calls and their return values.

```
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 2.

For example, to divide the following UNSIGNED variable U by 4, see the following.

```
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 the “Enumeration Encoding” section of the “Data Types” chapter.) This attribute allows Foundation Express to interpret your logic correctly.

pragma built_in

Label your primary logic functions with the built_in pragma. Pragmas allow Foundation Express to interpret your logic functions easily. When you use a built_in pragma, Foundation Express parses but ignores the body of the function. Instead, Foundation Express

directly substitutes the appropriate logic for the function. You are not required to use built_in pragmas; however, using these pragmas can result in run times that are ten times faster.

Use built_in pragmas by placing a comment in the declaration part of a function. Foundation Express interprets a comment as a directive if the first word of the comment is pragma.

The following example shows how to use 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

Xilinx 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.

The following example shows a function that takes the logical AND of two equal-size 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

Foundation Express 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. The following example shows a function that takes the logical NOT of an array.

```
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 the following example.

```
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;
```

numeric_std Package

Foundation Express supports nearly all of numeric_std, the IEEE Standard VHDL Synthesis Package, which defines numeric types and arithmetic functions.

Warning: The numeric_std package and the std_logic_arith package have overlapping operations. Using these two packages simultaneously during analysis could cause type mismatches.

Understanding the Limitations of numeric_std package

The 1999.05 version of Foundation Express does not support the following numeric_std package components:

- divide, rem, or mod operators

  If your design contains these operators, use the std_logic_arith package.

- TO_01 function as a simulation construct

Using the Package

Access numeric_std package with the following statement in your VHDL code.

```
library IEEE;
use IEEE.numeric_std.all;
```

These VHDL packages are pre-analyzed and do not require further analyzing. To list the packages currently in memory, use the following command.

```
report_design_lib
```

Data Types

The numeric_std package defines the following two data types in the same way that the std_logic_arith package does.

- UNSIGNED

  ```
  type UNSIGNED is array (NATURAL range <>) of
  STD_LOGIC;
  ```

  See the “UNSIGNED” section of this chapter for more information.

- SIGNED

  ```
  type SIGNED is array (NATURAL range <>) of STD_LOGIC;
  ```

  See the “SIGNED” section of this chapter for more information.

Conversion Functions

The numeric_std package provides functions to convert values between its UNSIGNED and SIGNED types. The following example shows the declarations of these conversion functions.

```
function TO_INTEGER (ARG: UNSIGNED) return NATURAL;
function TO_INTEGER (ARG: SIGNED) return INTEGER;
function TO_UNSIGNED (ARG, SIZE: NATURAL) return UNSIGNED;
function TO_SIGNED (ARG: INTEGER; SIZE: NATURAL) return SIGNED;
```

TO_INTEGER, TO_SIGNED, and TO_UNSIGNED are similar to CONV_INTEGER, CONV_SIGNED, and CONV_UNSIGNED in std_logic_arith (see the “Conversion Functions” section of this chapter).

Resize Function

The resize function numeric_std supports is shown in the declarations in the following example.

```
function RESIZE (ARG: SIGNED; NEW_SIZE: NATURAL) return SIGNED;
function RESIZE (ARG: UNSIGNED; NEW_SIZE: NATURAL) return UNSIGNED;
```

Arithmetic Functions

The numeric_std package provides arithmetic functions for use with combinations of 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, which the numeric_std package defines in the same way that the std_logic_arith package does.

- Binary functions having two arguments, such as the following.

  A+B

  A*B

- Unary functions having one argument, such as the following.

  –A

  abs A

The following example shows the declarations for binary functions having two arguments.

```
function "+" (L, R: UNSIGNED) return UNSIGNED;
function "+" (L, R: SIGNED) return SIGNED;
function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
function "+" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
function "+" (L: INTEGER; R: SIGNED) return SIGNED;
function "+" (L: SIGNED; R: INTEGER) return SIGNED;

function "-" (L, R: UNSIGNED) return UNSIGNED;
function "-" (L, R: SIGNED) return SIGNED;
function "-" (L: UNSIGNED;R: NATURAL) return UNSIGNED;
function "-" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
function "-" (L: SIGNED; R: INTEGER) return SIGNED;
function "-" (L: INTEGER; R: SIGNED) return SIGNED;
```

```
function "*" (L, R: UNSIGNED) return UNSIGNED;
function "*" (L, R: SIGNED) return SIGNED;
function "*" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
function "*" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
function "*" (L: SIGNED; R: INTEGER) return SIGNED;
function "*" (L: INTEGER; R: SIGNED) return SIGNED;
```

The following example shows the declarations for unary functions having one argument.

```
function "abs" (ARG: SIGNED) return SIGNED;
function "-" (ARG: SIGNED) return SIGNED;
```

Comparison Functions

The numeric_std package provides functions to compare UNSIGNED and SIGNED data types to each other and to the predefined type INTEGER. Foundation Express compares the numeric values of the arguments and returns a BOOLEAN value.

These functions produce comparators. The function declarations are listed in two groups.

- Ordering functions ("<", "<=", ">", ">="), shown in the following example
- Equality functions ("=", "/="), shown in the second example

```
function ">" (L, R: UNSIGNED) return BOOLEAN;
function ">" (L, R: SIGNED) return BOOLEAN;
function ">" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function ">" (L: INTEGER; R: SIGNED) return BOOLEAN;
function ">" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function ">" (L: SIGNED; R: INTEGER) return BOOLEAN;

function "<" (L, R: UNSIGNED) return BOOLEAN;
function "<" (L, R: SIGNED) return BOOLEAN;
function "<" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function "<" (L: INTEGER; R: SIGNED) return BOOLEAN;
function "<" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function "<" (L: SIGNED; R: INTEGER) return BOOLEAN;

function "<=" (L, R: UNSIGNED) return BOOLEAN;
function "<=" (L, R: SIGNED) return BOOLEAN;
function "<=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function "<=" (L: INTEGER; R: SIGNED) return BOOLEAN;
function "<=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function "<=" (L: SIGNED; R: INTEGER) return BOOLEAN;
```

```
function ">=" (L, R: UNSIGNED) return BOOLEAN;
function ">=" (L, R: SIGNED) return BOOLEAN;
function ">=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function ">=" (L: INTEGER; R: SIGNED) return BOOLEAN;
function ">=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function ">=" (L: SIGNED; R: INTEGER) return BOOLEAN;
```

The following example shows numeric_std equality functions.

```
function "=" (L, R: UNSIGNED) return BOOLEAN;
function "=" (L, R: SIGNED) return BOOLEAN;
function "=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function "=" (L: INTEGER; R: SIGNED) return BOOLEAN;
function "=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function "=" (L: SIGNED; R: INTEGER) return BOOLEAN;
```

```
function "/=" (L, R: UNSIGNED) return BOOLEAN;
function "/=" (L, R: SIGNED) return BOOLEAN;
function "/=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;
function "/=" (L: INTEGER; R: SIGNED) return BOOLEAN;
function "/=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;
function "/=" (L: SIGNED; R: INTEGER) return BOOLEAN;
```

Defining Logical Operators Functions

The numeric_std package provides functions that define all of the logical operators: NOT, AND, OR, NAND, NOR, XOR, and XNOR. These functions work just like similar functions in std_logic_1164, except that they operate on SIGNED and UNSIGNED values rather than on STD_LOGIC_VECTOR values. The following example shows these function declarations.

```
function "not" (L: UNSIGNED) return UNSIGNED;
function "and" (L, R: UNSIGNED) return UNSIGNED;
function "or" (L, R: UNSIGNED) return UNSIGNED;
function "nand" (L, R: UNSIGNED) return UNSIGNED;
function "nor" (L, R: UNSIGNED) return UNSIGNED;
function "xor" (L, R: UNSIGNED) return UNSIGNED;
function "xnor" (L, R: UNSIGNED) return UNSIGNED;
```

```
function "not" (L: SIGNED) return SIGNED;
function "and" (L, R: SIGNED) return SIGNED;
function "or" (L, R: SIGNED) return SIGNED;
function "nand" (L, R: SIGNED) return SIGNED;
function "nor" (L, R: SIGNED) return SIGNED;
function "xor" (L, R: SIGNED) return SIGNED;
function "xnor" (L, R: SIGNED) return SIGNED;
```

Shift Functions

The numeric_std package provides functions for shifting the bits in UNSIGNED and SIGNED numbers. These functions produce shifters. The following example shows the shift function declarations.

```
function SHIFT_LEFT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
function SHIFT_RIGHT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
function SHIFT_LEFT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
function SHIFT_RIGHT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;

function ROTATE_LEFT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
function ROTATE_RIGHT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
function ROTATE_LEFT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
function ROTATE_RIGHT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;
```

The SHIFT_LEFT function shifts the bits of its argument ARG left by COUNT bits. SHIFT_RIGHT shifts the bits of its argument ARG right by COUNT bits.

The SHIFT_LEFT functions work the same for both UNSIGNED and SIGNED values of ARG, shifting in zero bits as necessary. The SHIFT_RIGHT 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 ARG sign bit

The example in the “Shift and Rotate Operators” section of this chapter shows some shift functions calls and their return values.

Rotate Functions

ROTATE_LEFT and ROTATE_RIGHT are similar to the shift functions.

The following example shows rotate function declarations.

```
ROTATE_LEFT (U1, COUNT) = "01011011"

ROTATE_LEFT (S1, COUNT) = "01011011"

ROTATE_LEFT (U2, COUNT) = "01011111"

ROTATE_LEFT (S2, COUNT) = "01011111"
```

```
ROTATE_RIGHT (U1, COUNT) = "01101101"
ROTATE_RIGHT (S1, COUNT) = "01101101"
ROTATE_RIGHT (U2, COUNT) = "01111101"
ROTATE_RIGHT (S2, COUNT) = "01111101"
```

Shift and Rotate Operators

The numeric_std package provides shift operators and rotate operators, which work in the same way that shift functions and rotate functions do. The shift operators are sll, srl, sla, and sra.

The following example shows some shift and rotate operator declarations.

```
function "sll" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
function "sll" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
function "srl" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
function "srl" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
function "rol" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
function "rol" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
function "ror" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;
function "ror" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;
```

The following example includes some shift and rotate operators.

```
Variable U1, U2: UNSIGNED (7 downto 0);
Variable S1, S2: SIGNED (7 downto 0);
Variable COUNT: NATURAL;
...
U1 <= "01101011";
U2 <= "11101011";
S1 <= "01101011";
S2 <= "11101011";
COUNT <= 3;
...
SHIFT_LEFT (U1, COUNT) = "01011000"
SHIFT_LEFT (S1, COUNT) = "01011000"
SHIFT_LEFT (U2, COUNT) = "01011000"
SHIFT_LEFT (S2, COUNT) = "01011000"

SHIFT_RIGHT (U1, COUNT) = "00001101"
SHIFT_RIGHT (S1, COUNT) = "00001101"
SHIFT_RIGHT (U2, COUNT) = "00011101"
SHIFT_RIGHT (S2, COUNT) = "11111101"

U1 sll COUNT = "01011000"
S1 sll COUNT = "01011000"
```

```
U2 sll COUNT = "01011000"
S2 sll COUNT = "01011000"

U1 srl COUNT = "00001101"
S1 srl COUNT = "00001101"
U2 srl COUNT = "00011101"
S2 srl COUNT = "11111101"

U1 rol COUNT = "01011011"
S1 rol COUNT = "01011011"
U2 rol COUNT = "01011111"
S2 rol COUNT = "01011111"

U1 ror COUNT = "01101101"
S1 ror COUNT = "01101101"
U2 ror COUNT = "01111101"
S2 ror COUNT = "01111101"
```

std_logic_misc Package

This package resides in the Xilinx Foundation synthesis libraries directory ($XILINX/synth/lib/packages/IEEE/src/std_logic_misc.vhd). The std_logic_misc package declares the primary data types the Foundation Express VSS tools support.

Boolean reduction functions take 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 predefined type STD_LOGIC_VECTOR. The following example shows the declarations of these 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 STD_LOGIC_VECTOR, as the name of the function indicates. For example, XOR_REDUCE returns the XOR value of all bits in ARG.

The following example shows some reduction function calls and their return values.

```
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'
```

ATTRIBUTES Package

The ATTRIBUTES package declares all the supported synthesis (and simulation) attributes. These include the following.

- Foundation Express constraints and attributes
- State vector attributes
- Resource sharing attributes
- General attributes for interpreting VHDL (described in the "Data Types" chapter)
- Attributes to use with the Foundation Express VSS tools

Reference this package when you use synthesis attributes.

```
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;
```

Chapter 11

VHDL 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, Foundation Express does not support them.

This chapter provides a list of all VHDL language constructs with the level of support for each. At the end of the chapter is a list of VHDL reserved words.

The chapter is divided into the following sections.

- "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 Foundation Express.
- Unsupported means that the construct is not allowed in the VHDL source and that Foundation 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 subprograms 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 Express-defined attributes.

- 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. Only declarations from a globally static type are supported. Initial values are unsupported.

- variable

  Only declarations from a globally static type are supported. Initial values are unsupported.

- shared variable

  Variable shared by different processes. Shared variables are fully supported.

- file

  File declarations are unsupported.

- interface

  Buffer and linkage are translated to out and inout, respectively.

- alias

  Alias declarations are supported, with the following exceptions.

  - An alias declaration that lacks a subtype indication
  - A nonobject alias—such as an alias that refers to a type.

- component

  Only component declarations that list a valid entity name are supported.

- 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 symbol

  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. The event and stable attributes are supported only as described with the wait and if statements. (See the "wait Statements" section of the "Sequential Statements" chapter.) User-defined attribute names are unsupported. The use of attributes with selected names (name.name'attribute) is unsupported.

Identifiers and Extended Identifiers

An identifier in VHDL is a user-defined name for any of the following: constant, variable, function, signal, entity, port, subprogram, parameter, and instance.

Specifics of Identifiers

The characteristics of identifiers follow.

- They can be composed of letters, digits, and the underscore character (_).
- Their first character cannot be a number, unless it is an extended identifier (see the example in the next section).
- They can be of any length.
- They are case-insensitive.
- All of their characters are significant.

Specifics of Extended Identifiers

The characteristics of extended identifiers follow.

- Any of the following can be defined as an extended identifier.
  - Identifiers that contain special characters
  - Identifiers that begin with numbers
  - Identifiers that have the same name as a keyword
- They start with a backslash character (\ ), followed by a sequence of characters, followed by another backslash (\ ).
- They are case-sensitive.

The following example shows some extended identifiers.

```
\a+b\       \3state\
\type\       \(a&b)|c\
```

For more information about identifiers and extended identifiers, see the “Identifiers” section of the “Expressions” chapter.

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.

Shift and Rotate Operators

You can define shift and rotate operators for any one-dimensional array type whose element type is either of the predefined types, BIT or Boolean. The right operand is always of type integer. The type of the result of a shift operator is the same as the type of the left

operand. The shift and rotate operators are included in the list of VHDL reserved words in the “VHDL Construct Support” section of this chapter. There is more information about the shift and rotate operators that numeric_std supports in the “Shift and Rotate Operators” section of the “Foundation Express Packages” chapter. The shift operators follow.

- sll

  Shift left logical

- srl

  Shift right logical

- sla

  Shift left arithmetic

- sra

  Shift right arithmetic

The rotate operators follow.

- rol

  Rotate left logical

- ror

  Rotate right logical

The following example illustrates the use of shift and rotate operators.

```
architecture arch of shft_op is
begin
   a <= "01101";
   q1 <= a sll 1;-- q1 = "11010"
   q2 <= a srl 3;-- q2 = "00001"
   q3 <= a rol 2;-- q3 = "10101"
   q4 <= a ror 1;-- q4 = "10110"
   q5 <= a sla 2;-- q5 = "10100"
   q6 <= a sra 1;-- q6 = "00110"
end;
```

xnor Operator

You can define the binary logical operator xnor for predefined types BIT and Boolean, as well as for any one-dimensional array type whose element type is BIT or Boolean. The operands must be the same type and length. The result also has the same type and length. The xnor operator is included in the list of VHDL reserved words in the “VHDL Reserved Words” section of this chapter.

```
a <= "10101";
b <= "11100";
c <= a xnor b; -- c = "10110"
```

Operands and Expressions

- based literal

  Based literals are fully supported.

- null literal

  Null slices, null ranges, and null arrays are unsupported.

- physical literal

  Physical literals are ignored.

- string

  Strings are fully supported.

- aggregate

  The use of types as aggregate choices is unsupported. Record aggregates are supported.

- function call

  Function calls are supported, with one exception: Function conversions on input ports are not supported, because type conversions on formal ports in a connection specification (port map) are not supported.

- qualified expression

  Qualified expressions are fully supported.

- type conversion

  Type conversion is fully supported.

- allocator

  Allocators are unsupported.

- static expression

  Static expressions are fully supported.

- universal expression

  Floating-point expressions are unsupported, except in a Express-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 one of the following forms.

  ```
  wait until        clock = VALUE;
  wait until        clock'event and clock = VALUE;
  wait until        not clock'stable and clock = VALUE;
  ```

  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."

  You cannot use wait statements in subprograms.

- assert

  Assert statements are ignored.

- report

  Report statements are ignored.

- statement label

  Statement labels are ignored.

- signal

  Guarded signal assignment is unsupported. The Transport and after signals 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

  The 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. The 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 supported. 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 keyword is supported. The transport keyword is ignored. Multiple waveforms are unsupported.

- component instantiation

  Type conversion on the formal port of a connection specification is unsupported.

- generate

  The generate statements are fully supported.

Predefined Language Environment

- severity_level type

  The severity_level type is unsupported.

- time type

  The time type is ignored if time variables and constants are used only in after clauses. In the following two code fragments, both the after clause and TD are ignored.

  ```
  constant TD: time := 1.4 ns;
  X <= Y after TD;
  ```

  ```
  X <= Y after 1.4 ns;
  ```

- now function

  The now function is unsupported.

- TEXTIO package

  The TEXTIO package is unsupported.

- predefined attributes

  These predefined attributes are supported: base, left, right, high, low, range, reverse_range, ascending, and length. The event and stable attributes are supported only in the if and wait statements,

as described in the “wait Statements” section of the “Sequential Statements” chapter.

VHDL Reserved Words

The following words are reserved for the VHDL language and cannot be used as identifiers.

abs	if	reject
access	impure	rem
after	in	report
alias	inertial	return
all	inout	rol
and	is	ror
architecture		
array	label	select
assert	library	severity
attribute	linkage	shared
	literal	signal
begin	loop	sla
block		sll
body	map	sra
buffer	mod	srl
bus		subtype
	nand	
case	new	then
component	next	to
configuration	nor	transport
constant	not	type
	null	
disconnect		unaffected
downto	of	units
	on	until
else	open	use
elsif	or	
end	others	variable

entity
exit

file
for
function

generate
generic
guarded

out

package
port
procedure
process

range
record
register

wait
when
while
with

xnor
xor

Appendix A

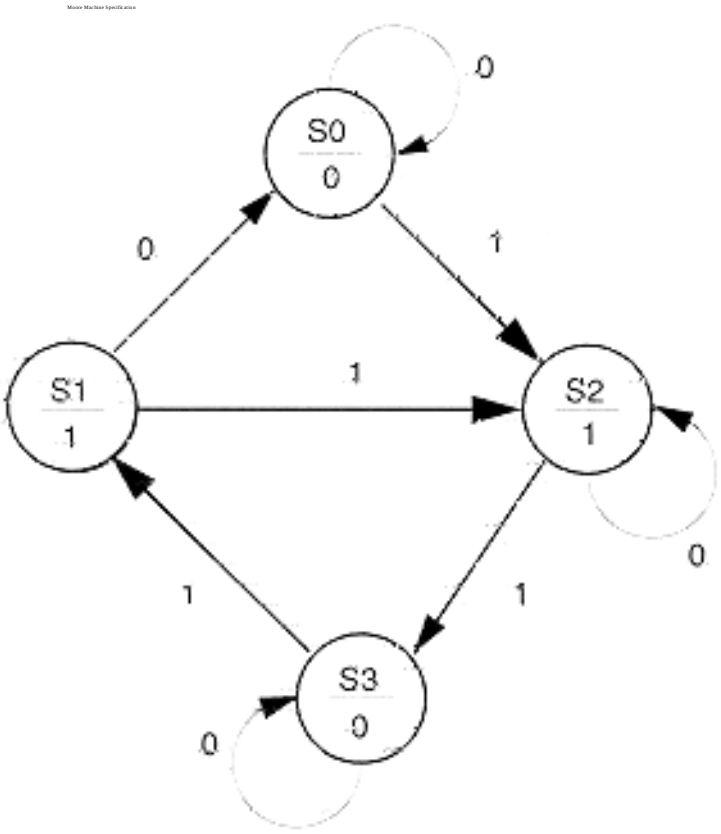
Examples

This appendix presents examples that demonstrate basic concepts of Foundation Express.

- "Moore Machine"
- "Mealy Machine"
- "Read-Only Memory"
- "Waveform Generator"
- "Smart Waveform Generator"
- "Definable-Width Adder-Subtracter"
- "Count Zeros—Combinatorial Version"
- "Count Zeros—Sequential Version"
- "Soft Drink Machine—State Machine Version"
- "Soft Drink Machine—Count Nickels Version"
- "Carry-Lookahead Adder"
- "Serial-to-Parallel Converter—Counting Bits"
- "Serial-to-Parallel Converter—Shifting Bits"
- "Programmable Logic Arrays"

Moore Machine

The following figure is a diagram of a simple Moore finite state machine. It has one input (X), four internal states (S0 to S3), and one output (Z).



Present state	Next state		Output (Z)
	X=0	X=1	X=0
S0	S0	S2	0
S1	S0	S2	1
S2	S2	S3	1
S3	S3	S1	0

Figure A-1 Moore Machine Specification

The VHDL code implementing this finite state machine is shown in the following example, which includes a schematic of the synthesized circuit.

The machine description includes two processes. One process defines the synchronous elements of the design (state registers); the other process defines the combinatorial part of the design (state assignment case statement). For more details on using the two processes, see the "Combinatorial Versus Sequential Processes" section of the "Sequential Statements" chapter.

```
entity MOORE is                -- Moore machine
  port(X, CLOCK: in BIT;
       Z: out BIT);
end MOORE;

architecture BEHAVIOR of MOORE is
  type STATE_TYPE is (S0, S1, S2, S3);
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
begin

  -- Process to hold combinatorial logic
  COMBIN: process(CURRENT_STATE, X)
  begin
    case CURRENT_STATE is
      when S0 =>
        Z <= '0';
        if X = '0' then
          NEXT_STATE <= S0;
        else
          NEXT_STATE <= S2;
        end if;
      when S1 =>
        Z <= '1';
        if X = '0' then
          NEXT_STATE <= S0;
        else
          NEXT_STATE <= S2;
        end if;
      when S2 =>
        Z <= '1';
        if X = '0' then
          NEXT_STATE <= S2;
        else
          NEXT_STATE <= S3;
        end if;
      when S3 =>
```

```
            Z <= '0';
            if X = '0' then
              NEXT_STATE <= S3;
            else
              NEXT_STATE <= S1;
            end if;
          end case;
        end process COMBIN;


        -- Process to hold synchronous elements (flip-flops)
        SYNCH: process
        begin
          wait until CLOCK'event and CLOCK = '1';
          CURRENT_STATE <= NEXT_STATE;
        end process SYNCH;
      end BEHAVIOR;
```

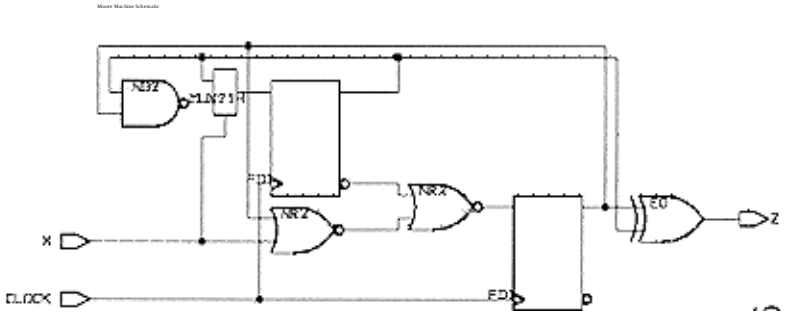


Figure A-2 Moore Machine Schematic

Mealy Machine

The following figure is a diagram of a simple Mealy finite state machine. The VHDL code for implementing this finite state machine is shown in the example following the diagram. The machine description includes two processes, as in the previous Moore machine example.

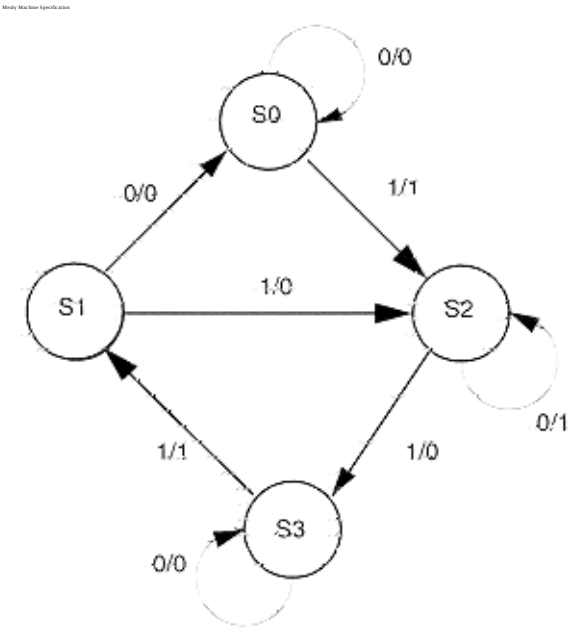


Figure A-3 Mealy Machine Specification-1

Present state	Next state		Output (Z)	
	X=0	X=1	X=0	X=1
S0	S0	S2	0	1
S1	S0	S2	0	0
S2	S2	S3	1	0
S3	S3	S1	0	1

Figure A-4 Mealy Machine Specification-2

```
entity MEALY is              -- Mealy machine
  port(X, CLOCK: in BIT;
       Z: out BIT);
end MEALY;

architecture BEHAVIOR of MEALY is
  type STATE_TYPE is (S0, S1, S2, S3);
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
begin
```

```
  -- Process to hold combinatorial logic.
  COMBIN: process(CURRENT_STATE, X)
  begin
    case CURRENT_STATE is
      when S0 =>
        if X = '0' then
          Z <= '0';
          NEXT_STATE <= S0;
        else
          Z <= '1';
          NEXT_STATE <= S2;
        end if;
      when S1 =>
        if X = '0' then
          Z <= '0';
          NEXT_STATE <= S0;
        else
          Z <= '0';
          NEXT_STATE <= S2;
        end if;
      when S2 =>
        if X = '0' then
          Z <= '1';
          NEXT_STATE <= S2;
        else
          Z <= '0';
          NEXT_STATE <= S3;
        end if;
      when S3 =>
        if X = '0' then
          Z <= '0';
          NEXT_STATE <= S3;
        else
          Z <= '1';
          NEXT_STATE <= S1;
        end if;
    end case;
  end process COMBIN;
  -- Process to hold synchronous elements (flip-flops)
  SYNCH: process
  begin
    wait until CLOCK'event and CLOCK = '1';
    CURRENT_STATE <= NEXT_STATE;
  end process SYNCH;
end BEHAVIOR;
```

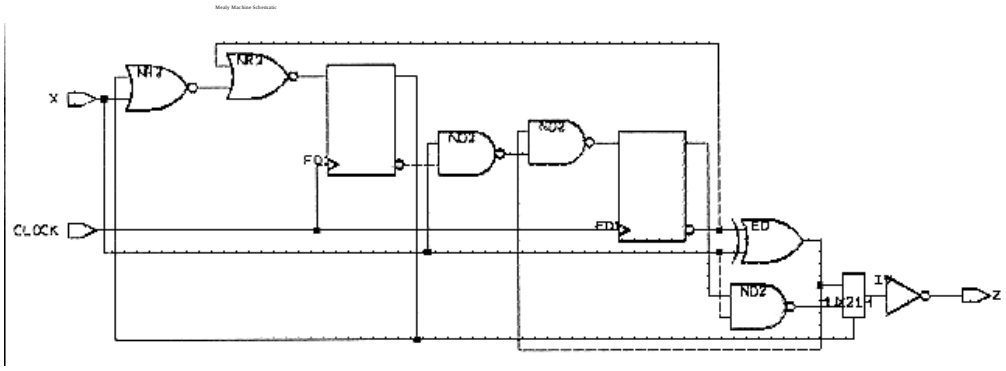


Figure A-5 Mealy Machine Schematic

Read-Only Memory

The following example shows how you can define a read-only memory in VHDL. The ROM is defined as an array constant, ROM. Each line of the constant array specification defines the contents of one ROM address. To read from the ROM, index into the array.

The number of ROM storage locations and bit-width is easy to change. The subtype ROM_RANGE specifies that the ROM contains storage locations 0 to 7. The constant ROM_WIDTH specifies that the ROM is 5 bits wide.

After you define a ROM constant, you can index into that constant many times to read many values from the ROM. If the ROM address is computable (see the “Computable Operands” section of the “Expressions” chapter), no logic is built and the appropriate data value is inserted. If the ROM address is not computable, logic is built for each index into the value. In the following example, ADDR is not computable, so logic is synthesized to compute the value.

Foundation Express does not actually instantiate a typical array-logic ROM, such as those available from ASIC vendors. Instead, it creates the ROM from random logic gates (AND, OR, NOT, and so on). This type of implementation is preferable for small ROMs and for ROMs that are regular. For very large ROMs, consider using an array-logic implementation supplied by your ASIC vendor.

The following example shows the VHDL source code and the synthesized circuit schematic.

```
package ROMS is
  -- declare a 5x8 ROM called ROM
  constant ROM_WIDTH: INTEGER := 5;
  subtype ROM_WORD is BIT_VECTOR (1 to ROM_WIDTH);
  subtype ROM_RANGE is INTEGER range 0 to 7;
  type ROM_TABLE is array (0 to 7) of ROM_WORD;
  constant ROM: ROM_TABLE := ROM_TABLE'(
      ROM_WORD'("10101"),              -- ROM contents
      ROM_WORD'("10000"),
      ROM_WORD'("11111"),
      ROM_WORD'("11111"),
      ROM_WORD'("10000"),
      ROM_WORD'("10101"),
      ROM_WORD'("11111"),
      ROM_WORD'("11111"));
end ROMS;
use work.ROMS.all;   -- Entity that uses ROM
entity ROM_5x8 is
  port(ADDR: in ROM_RANGE;
       DATA: out ROM_WORD);
end ROM_5x8;
architecture BEHAVIOR of ROM_5x8 is
begin
  DATA <= ROM(ADDR);      -- Read from the ROM
end BEHAVIOR;
```

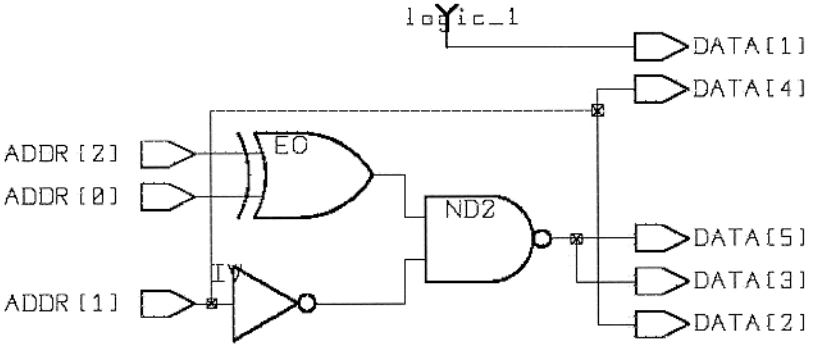


Figure A-6 ROM Schematic

Waveform Generator

The waveform generator example shows how to use the previous ROM example to implement a waveform generator.

Assume that you want to produce the waveform output shown in the following figure.

1. First, declare a ROM wide enough to hold the output signals (4 bits) and deep enough to hold all time steps (0 to 12, for a total of 13).
2. Next, define the ROM so that each time step is represented by an entry in the ROM.
3. Finally, create a counter that cycles through the time steps (ROM addresses), generating the waveform at each time step.

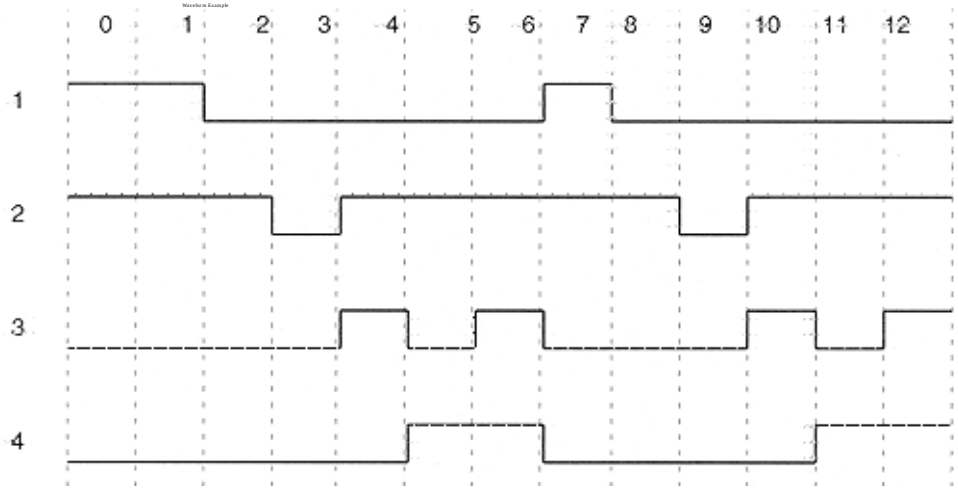


Figure A-7 Waveform Example

The following example shows an implementation for the waveform generator. It consists of a ROM, a counter, and some simple reset logic.

```
package ROMS is
  -- a 4x13 ROM called ROM that contains the waveform
  constant ROM_WIDTH: INTEGER := 4;
  subtype ROM_WORD is BIT_VECTOR (1 to ROM_WIDTH);
  subtype ROM_RANGE is INTEGER range 0 to 12;
  type ROM_TABLE is array (0 to 12) of ROM_WORD;
```

```
  constant ROM: ROM_TABLE := ROM_TABLE'(
      "1100",   -- time step 0
      "1100",   -- time step 1
      "0100",   -- time step 2
      "0000",   -- time step 3
      "0110",   -- time step 4
      "0101",   -- time step 5
      "0111",   -- time step 6
      "1100",   -- time step 7
      "0100",   -- time step 8
      "0000",   -- time step 9
      "0110",   -- time step 10
      "0101",   -- time step 11
      "0111");  -- time step 12
end ROMS;

use work.ROMS.all;
entity WAVEFORM is            -- Waveform generator
  port(CLOCK: in BIT;
       RESET: in BOOLEAN;
       WAVES: out ROM_WORD);
end WAVEFORM;

architecture BEHAVIOR of WAVEFORM is
  signal STEP: ROM_RANGE;
begin

  TIMESTEP_COUNTER: process   -- Time stepping process
  begin
    wait until CLOCK'event and CLOCK = '1';
    if RESET then             -- Detect reset
      STEP <= ROM_RANGE'low;  -- Restart
    elsif STEP = ROM_RANGE'high then  -- Finished?
      STEP <= ROM_RANGE'high;  -- Hold at last value
   -- STEP <= ROM_RANGE'low;   -- Continuous wave
    else
      STEP <= STEP + 1;        -- Continue stepping
    end if;
  end process TIMESTEP_COUNTER;

  WAVES <= ROM(STEP);
end BEHAVIOR;
```

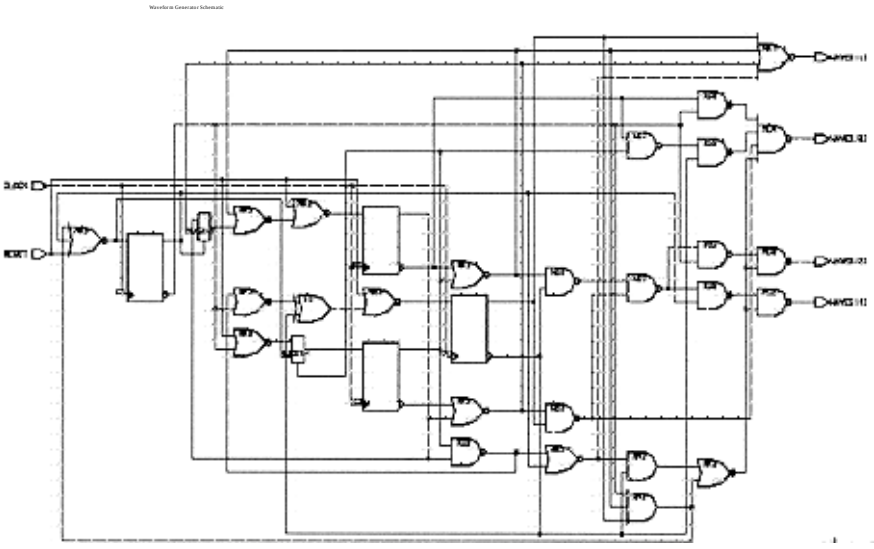


Figure A-8 Waveform Generator Schematic

When the counter STEP reaches the end of the ROM, STEP stops, generates the last value, then waits until a reset. To make the sequence automatically repeat, remove the following statement.

```
STEP <= ROM_RANGE'high;  -- Hold at last value
```

Use the following statement instead (commented out in the previous example).

```
STEP <= ROM_RANGE'low;   -- Continuous wave
```

Smart Waveform Generator

The smart waveform generator in the following figure is an extension of the waveform generator in the figure “Waveform Example.” But

this smart waveform generator is capable of holding the waveform at any time step for several clock cycles.

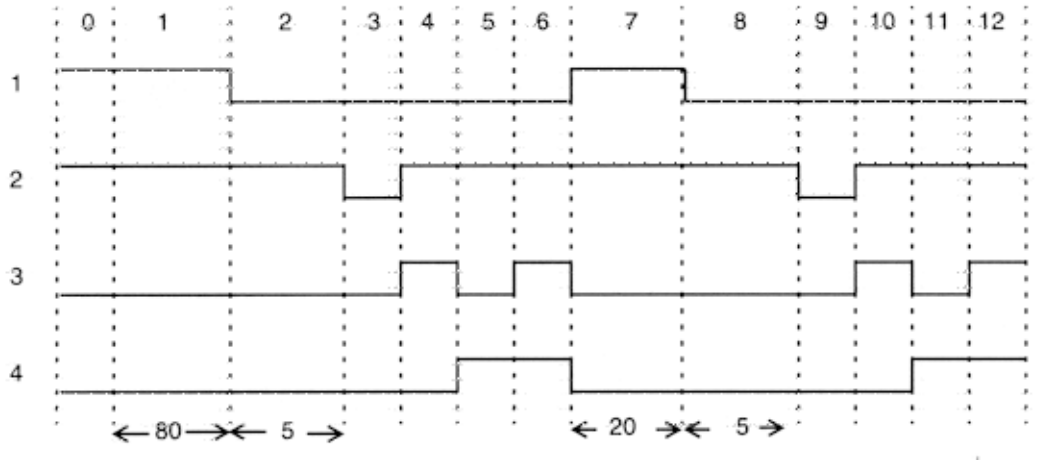


Figure A-9 Waveform for Smart Waveform Generator

The implementation of the smart waveform generator is shown in the following example. It is similar to the waveform generator in the "Mealy Machine Schematic" figure in the Mealy Machine section, but has two additions. A new ROM, D_ROM, has been added to hold the length of each time step. A value of 1 specifies that the corresponding time step should be one clock cycle long; a value of 80 specifies that the time step should be 80 clock cycles long. The second addition to the previous waveform generator is a delay counter that counts the clock cycles between time steps.

In the architecture of the following example, a selected signal assignment determines the value of the NEXT_STEP counter.

```
package ROMS is

-- a 4x13 ROM called W_ROM containing the waveform
  constant W_ROM_WIDTH: INTEGER := 4;
  subtype W_ROM_WORD is BIT_VECTOR (1 to W_ROM_WIDTH);
  subtype W_ROM_RANGE is INTEGER range 0 to 12;
  type W_ROM_TABLE is array (0 to 12) of W_ROM_WORD;
  constant W_ROM: W_ROM_TABLE := W_ROM_TABLE'(
    "1100",   -- time step 0
    "1100",   -- time step 1
    "0100",   -- time step 2
```

```
    ”0000”,   -- time step 3
    ”0110”,   -- time step 4
    ”0101”,   -- time step 5
    ”0111”,   -- time step 6
    ”1100”,   -- time step 7
    ”0100”,   -- time step 8
    ”0000”,   -- time step 9
    ”0110”,   -- time step 10
    ”0101”,   -- time step 11
    ”0111”);  -- time step 12

  -- a 7x13 ROM called D_ROM containing the delays
  subtype D_ROM_WORD is INTEGER range 0 to 100;
  subtype D_ROM_RANGE is INTEGER range 0 to 12;
  type D_ROM_TABLE is array (0 to 12) of D_ROM_WORD;
  constant D_ROM: D_ROM_TABLE := D_ROM_TABLE’(
      1,80,5,1,1,1,1,20,5,1,1,1,1);
end ROMS;

use work.ROMS.all;
entity WAVEFORM is        -- Smart Waveform Generator
  port(CLOCK: in BIT;
       RESET: in BOOLEAN;
       WAVES: out W_ROM_WORD);
end WAVEFORM;

architecture BEHAVIOR of WAVEFORM is
  signal STEP, NEXT_STEP: W_ROM_RANGE;
  signal DELAY: D_ROM_WORD;
begin

  -- Determine the value of the next time step
  NEXT_STEP <= W_ROM_RANGE’high when
                    STEP = W_ROM_RANGE’high
               else
                 STEP + 1;
  -- Keep track of which time step we are in
  TIMESTEP_COUNTER: process
  begin
    wait until CLOCK’event and CLOCK = ’1’;
    if RESET then             -- Detect reset
      STEP <= 0;              -- Restart waveform
    elsif DELAY = 1 then
      STEP <= NEXT_STEP;      -- Continue stepping
    else
      null;          -- Wait for DELAY to count down;
```

```
    end if;          -- do nothing here
  end process TIMESTEP_COUNTER;

   -- Count the delay between time steps
  DELAY_COUNTER: process
  begin
    wait until CLOCK'event and CLOCK = '1';
    if RESET then              -- Detect reset
      DELAY <= D_ROM(0);       -- Restart
    elsif DELAY = 1 then      -- Have we counted down?
      DELAY <= D_ROM(NEXT_STEP);  -- Next delay value
      else
      DELAY <= DELAY - 1;   -- decrement DELAY counter
      end if;
    end process DELAY_COUNTER;

  WAVES <= W_ROM(STEP);     -- Output waveform value
end BEHAVIOR;
```

Figure A-10 Smart Waveform Generator Schematic

Definable-Width Adder-Subtracter

VHDL lets you create functions for use with array operands of any size. This example shows an adder-subtracter circuit that, when called, is adjusted to fit the size of its operands.

The following example shows an adder-subtracter defined for two unconstrained arrays of bits (type BIT_VECTOR) in a package named

MATH. When an unconstrained array type is used for an argument to a subprogram, the actual constraints of the array are taken from the actual parameter values in a subprogram call.

```
package MATH is
  function ADD_SUB(L, R: BIT_VECTOR; ADD: BOOLEAN)
      return BIT_VECTOR;
    -- Add or subtract two BIT_VECTORs of equal length
end MATH;

package body MATH is
    function ADD_SUB(L, R: BIT_VECTOR; ADD: BOOLEAN)
        return BIT_VECTOR is
      variable CARRY: BIT;
      variable A, B, SUM:
          BIT_VECTOR(L'length-1 downto 0);
    begin
      if ADD then
          -- Prepare for an "add" operation
          A := L;
          B := R;
          CARRY := '0';
      else
          -- Prepare for a "subtract" operation
          A := L;
          B := not R;
          CARRY := '1';
      end if;

      -- Create a ripple carry chain; sum up bits
      for i in 0 to A'left loop
        SUM(i) := A(i) xor B(i) xor CARRY;
        CARRY := (A(i) and B(i)) or
                 (A(i) and CARRY) or
                 (CARRY and B(i));
      end loop;
      return SUM;        -- Result
    end;
end MATH;
```

Within the function ADD_SUB, two temporary variables, A and B, are declared. These variables are declared to be the same length as L (and necessarily, R) but have their index constraints normalized to L'length-1 downto 0. After the arguments are normalized, you can create a ripple carry adder by using a for loop.

No explicit references to a fixed array length are in the function ADD_SUB. Instead, the VHDL array attributes 'left and 'length are used. These attributes allow the function to work on arrays of any length.

The following example shows how to use the adder-subtracter defined in the MATH package. In this example, the vector arguments to functions ARG1 and ARG2 are declared as BIT_VECTOR(1 to 6). This declaration causes ADD_SUB to work with 6-bit arrays. A schematic of the synthesized circuit follows the example.

```
use work.MATH.all;

entity EXAMPLE is
    port(ARG1, ARG2: in BIT_VECTOR(1 to 6);
         ADD: in BOOLEAN;
         RESULT : out BIT_VECTOR(1 to 6));
end EXAMPLE;

architecture BEHAVIOR of EXAMPLE is
begin
  RESULT <= ADD_SUB(ARG1, ARG2, ADD);
end BEHAVIOR;
```

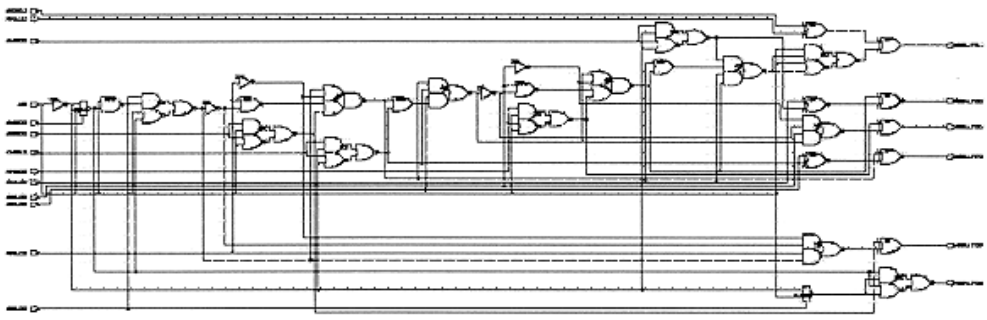


Figure A-11 6-Bit Adder-Subtracter Schematic

Count Zeros—Combinatorial Version

The count zeros—combinatorial example illustrates a design problem in which an 8-bit-wide value is given and the circuit determines two things.

- That no more than one sequence of zeros is in the value.

- The number of zeros in that sequence (if any). This computation must be completed in a single clock cycle.

The circuit produces two outputs: the number of zeros found and an error indication.

A valid input value can have at most one consecutive series of zeros. A value consisting entirely of ones is defined as a valid value. If a value is invalid, the zero counter resets to 0. For example, the value 00000000 is valid and has eight zeros; value 11000111 is valid and has three zeros; value 00111100 is invalid.

The following example shows the VHDL description for the circuit. It consists of a single process with a for loop that iterates across each bit in the given value. At each iteration, a temporary INTEGER variable (TEMP_COUNT) counts the number of zeros encountered. Two temporary Boolean variables (SEEN_ZERO and SEEN_TRAILING), initially false, are set to true when the beginning and end of the first sequence of zeros is detected.

If a zero is detected after the end of the first sequence of zeros (after SEEN_TRAILING is true), the zero count is reset (to 0), ERROR is set to true, and the for loop is exited.

The following example shows a combinatorial (parallel) approach to counting the zeros. The second example shows a sequential (serial) approach.

```
entity COUNT_COMB_VHDL is
  port(DATA:  in  BIT_VECTOR(7 downto 0);
       COUNT: out INTEGER range 0 to 8;
       ERROR: out BOOLEAN);
end COUNT_COMB_VHDL;

architecture BEHAVIOR of COUNT_COMB_VHDL is
begin
  process(DATA)
    variable TEMP_COUNT : INTEGER range 0 to 8;
    variable SEEN_ZERO, SEEN_TRAILING : BOOLEAN;
  begin
    ERROR <= FALSE;
    SEEN_ZERO <= FALSE;
    SEEN_TRAILING <= FALSE;
    TEMP_COUNT <= 0;
    for I in 0 to 7 loop
      if (SEEN_TRAILING and DATA(I) = '0') then
        TEMP_COUNT <= 0;
```

```
            ERROR <= TRUE;
            exit;
          elsif (SEEN_ZERO and DATA(I) = '1') then
            SEEN_TRAILING <= TRUE;
          elsif (DATA(I) = '0') then
            SEEN_ZERO <= TRUE;
            TEMP_COUNT <= TEMP_COUNT + 1;
          end if;
        end loop;

       COUNT <= TEMP_COUNT;
      end process;

    end BEHAVIOR;
```

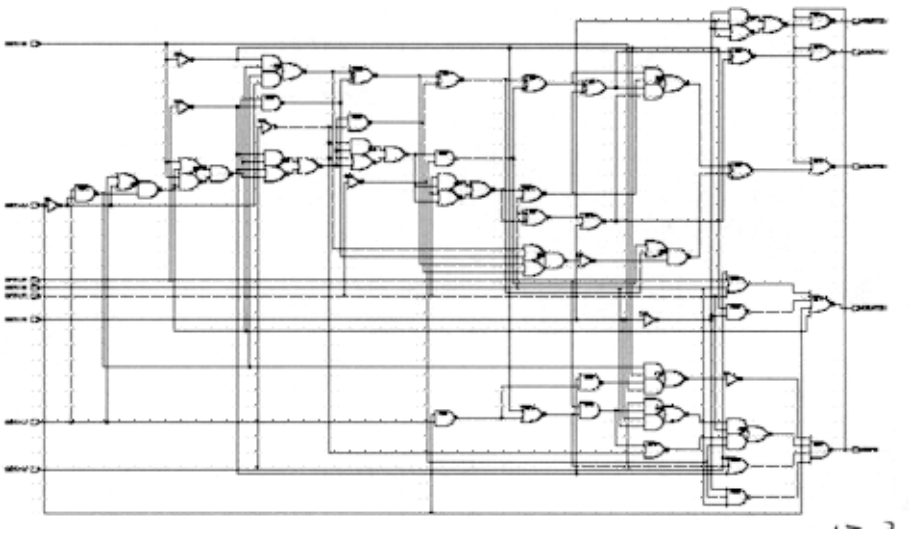


Figure A-12 Count Zeros—Combinatorial Schematic

Count Zeros—Sequential Version

The count zeros—sequential example shows a sequential (clocked) variant of the preceding design (Count Zeros—Combinatorial Version).

The circuit now accepts the 8-bit data value serially, 1 bit per clock cycle, by using the DATA and CLK inputs. The other two inputs follow.

- RESET, which resets the circuit
- READ, which causes the circuit to begin accepting data bits

The circuit's three outputs follow.

- IS_LEGAL, which is true if the data was a valid value
- COUNT_READY, which is true at the first invalid bit or when all 8 bits have been processed
- COUNT, the number of zeros (if IS_LEGAL is true)

Note: The output port COUNT is declared with mode BUFFER so that it can be read inside the process. OUT ports can only be written to, not read in.

```
entity COUNT_SEQ_VHDL is
  port(DATA, CLK: in BIT;
       RESET, READ: in BOOLEAN;
       COUNT: buffer INTEGER range 0 to 8;
       IS_LEGAL: out BOOLEAN;
       COUNT_READY: out BOOLEAN);
end COUNT_SEQ_VHDL;
architecture BEHAVIOR of COUNT_SEQ_VHDL is
begin
  process
    variable SEEN_ZERO, SEEN_TRAILING: BOOLEAN;
    variable BITS_SEEN: INTEGER range 0 to 7;
  begin
    wait until CLK'event and CLK = '1';

    if(RESET) then
      COUNT_READY<= FALSE;
      IS_LEGAL <= TRUE;-- signal assignment
      SEEN_ZERO<= FALSE;-- variable assignment
      SEEN_TRAILING <= FALSE;
      COUNT<= 0;
      BITS_SEEN<= 0;
    else
      if (READ) then
        if (SEEN_TRAILING and DATA = '0') then
          IS_LEGAL <= FALSE;
          COUNT <= 0;
          COUNT_READY <= TRUE;
        elsif (SEEN_ZERO and DATA = '1') then
          SEEN_TRAILING := TRUE;
        elsif (DATA = '0') then
```

```
            SEEN_ZERO <= TRUE;
            COUNT <= COUNT + 1;
          end if;

          if (BITS_SEEN = 7) then
            COUNT_READY <= TRUE;
          else
            BITS_SEEN <=  BITS_SEEN + 1;
          end if;

        end if;     -- if (READ)
      end if;       -- if (RESET)
    end process;
end BEHAVIOR;
```

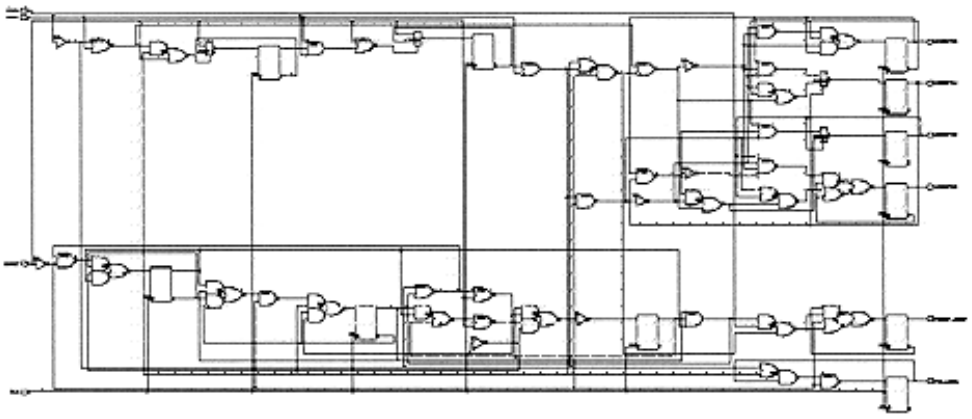


Figure A-13 Count Zeros—Sequential Schematic

Soft Drink Machine—State Machine Version

The soft drink machine—state machine example is a control unit for a soft drink vending machine.

The circuit reads signals from a coin input unit and sends outputs to a change dispensing unit and a drink dispensing unit.

Here are the design parameters for the following two examples.

- This example assumes that only one kind of soft drink is dispensed.
- This is a clocked design with CLK and RESET input signals.

- The price of the drink is 35 cents.
- The input signals from the coin input unit are NICKEL_IN (nickel deposited), DIME_IN (dime deposited), and QUARTER_IN (quarter deposited).
- The output signals to the change dispensing unit are NICKEL_OUT and DIME_OUT.
- The output signal to the drink dispensing unit is DISPENSE (dispense drink).
- The first VHDL description for this design uses a state machine description style. The second VHDL description is in the example after the following example.

```
library synopsys; use synopsys.attributes.all;

entity DRINK_STATE_VHDL is
  port(NICKEL_IN, DIME_IN, QUARTER_IN, RESET: BOOLEAN;
       CLK: BIT;
       NICKEL_OUT, DIME_OUT, DISPENSE: out BOOLEAN);
end DRINK_STATE_VHDL;

architecture BEHAVIOR of DRINK_STATE_VHDL is
  type STATE_TYPE is (IDLE, FIVE, TEN, FIFTEEN,
            TWENTY, TWENTY_FIVE, THIRTY, OWE_DIME);
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
  attribute STATE_VECTOR : STRING;
  attribute STATE_VECTOR of BEHAVIOR : architecture is
 "CURRENT_STATE";

attribute sync_set_reset of reset : signal is "true";
begin

   process(NICKEL_IN, DIME_IN, QUARTER_IN,
          CURRENT_STATE, RESET, CLK)
  begin
    -- Default assignments
    NEXT_STATE <= CURRENT_STATE;
    NICKEL_OUT <= FALSE;
    DIME_OUT <= FALSE;
    DISPENSE <= FALSE;

    -- Synchronous reset
    if(RESET) then
      NEXT_STATE <= IDLE;
    else
```

```
-- State transitions and output logic
case CURRENT_STATE is
    when IDLE =>
      if(NICKEL_IN) then
        NEXT_STATE <= FIVE;
      elsif(DIME_IN) then
        NEXT_STATE <= TEN;
      elsif(QUARTER_IN) then
        NEXT_STATE <= TWENTY_FIVE;
      end if;

    when FIVE =>
      if(NICKEL_IN) then
        NEXT_STATE <= TEN;
      elsif(DIME_IN) then
        NEXT_STATE <= FIFTEEN;
      elsif(QUARTER_IN) then
        NEXT_STATE <= THIRTY;
      end if;
    when TEN =>
      if(NICKEL_IN) then
        NEXT_STATE <= FIFTEEN;
      elsif(DIME_IN) then
        NEXT_STATE <= TWENTY;
      elsif(QUARTER_IN) then
        NEXT_STATE <= IDLE;
        DISPENSE <= TRUE;
      end if;
    when FIFTEEN =>
      if(NICKEL_IN) then
        NEXT_STATE <= TWENTY;
      elsif(DIME_IN) then
        NEXT_STATE <= TWENTY_FIVE;
      elsif(QUARTER_IN) then
        NEXT_STATE <= IDLE;
        DISPENSE <= TRUE;
        NICKEL_OUT <= TRUE;
      end if;
    when TWENTY =>
      if(NICKEL_IN) then
        NEXT_STATE <= TWENTY_FIVE;
      elsif(DIME_IN) then
        NEXT_STATE <= THIRTY;
      elsif(QUARTER_IN) then
        NEXT_STATE <= IDLE;
        DISPENSE <= TRUE;
```

```
            DIME_OUT <= TRUE;
          end if;

when TWENTY_FIVE =>
          if(NICKEL_IN) then
            NEXT_STATE <= THIRTY;
          elsif(DIME_IN) then
            NEXT_STATE <= IDLE;
            DISPENSE <= TRUE;
          elsif(QUARTER_IN) then
            NEXT_STATE <= IDLE;
            DISPENSE <= TRUE;
            DIME_OUT <= TRUE;
            NICKEL_OUT <= TRUE;
          end if;

        when THIRTY =>
          if(NICKEL_IN) then
            NEXT_STATE <= IDLE;
            DISPENSE <= TRUE;
          elsif(DIME_IN) then
            NEXT_STATE <= IDLE;
            DISPENSE <= TRUE;
            NICKEL_OUT <= TRUE;
          elsif(QUARTER_IN) then
            NEXT_STATE <= OWE_DIME;
            DISPENSE <= TRUE;
            DIME_OUT <= TRUE;
          end if;

        when OWE_DIME =>
          NEXT_STATE <= IDLE;
          DIME_OUT <= TRUE;


      end case;
    end if;
  end process;

  -- Synchronize state value with clock
  -- This causes it to be stored in flip-flops
  process
  begin
    wait until CLK'event and CLK = '1';
    CURRENT_STATE <= NEXT_STATE;
  end process;
end BEHAVIOR;
```

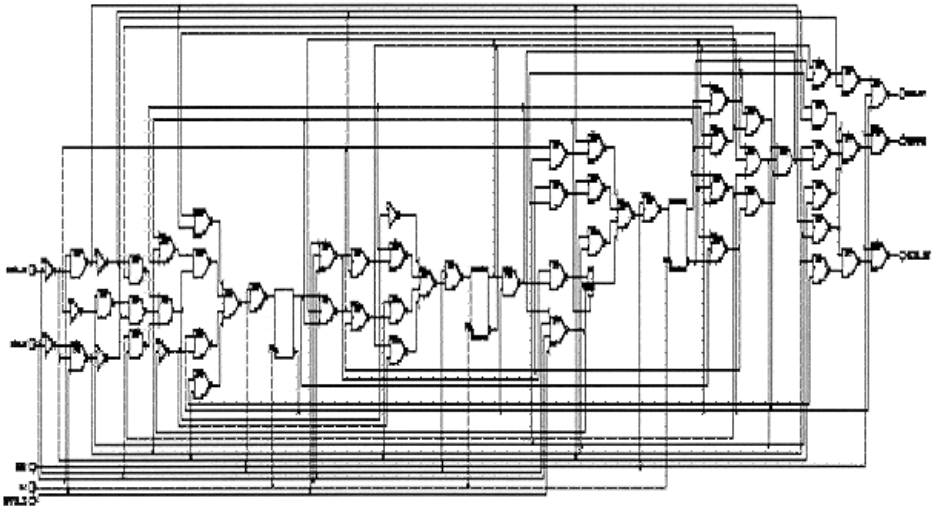


Figure A-14 Soft Drink Machine—State Machine Schematic

Soft Drink Machine—Count Nickels Version

The soft drink machine—count nickels example uses the same design parameters as the preceding example of a soft drink machine—state machine with the same input and output signals. In this version, a counter counts the number of nickels deposited. The counter in the following example is incremented by one if the deposit is a nickel, by two if it is a dime, and by five if it is a quarter.

```
entity DRINK_COUNT_VHDL is
  port(NICKEL_IN, DIME_IN, QUARTER_IN, RESET:
            BOOLEAN;
       CLK: BIT;
       NICKEL_OUT, DIME_OUT, DISPENSE: out BOOLEAN);
end DRINK_COUNT_VHDL;

architecture BEHAVIOR of DRINK_COUNT_VHDL is
  signal CURRENT_NICKEL_COUNT,
         NEXT_NICKEL_COUNT: INTEGER range 0 to 7;
  signal CURRENT_RETURN_CHANGE, NEXT_RETURN_CHANGE :
BOOLEAN;
begin
```

```
process(NICKEL_IN, DIME_IN, QUARTER_IN, RESET, CLK,
         CURRENT_NICKEL_COUNT, CURRENT_RETURN_CHANGE)
    variable TEMP_NICKEL_COUNT: INTEGER range 0 to 12;
begin
    -- Default assignments
    NICKEL_OUT <= FALSE;
    DIME_OUT <= FALSE;
    DISPENSE <= FALSE;
    NEXT_NICKEL_COUNT <= 0;
    NEXT_RETURN_CHANGE <= FALSE;

-- Synchronous reset
    if (not RESET) then
      TEMP_NICKEL_COUNT <= CURRENT_NICKEL_COUNT;

      -- Check whether money has come in
      if (NICKEL_IN) then
        -- NOTE:  This design will be flattened, so
        --   these multiple adders will be optimized
        TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT + 1;
      elsif(DIME_IN) then
        TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT + 2;
      elsif(QUARTER_IN) then
        TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT + 5;
      end if;

      -- Enough deposited so far?
      if(TEMP_NICKEL_COUNT >= 7) then
        TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT - 7;
        DISPENSE <= TRUE;
      end if;

      -- Return change
      if(TEMP_NICKEL_COUNT >= 1 or
         CURRENT_RETURN_CHANGE) then
        if(TEMP_NICKEL_COUNT >= 2) then
          DIME_OUT <= TRUE;
          TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT - 2;
          NEXT_RETURN_CHANGE <= TRUE;
        end if;
        if(TEMP_NICKEL_COUNT = 1) then
          NICKEL_OUT <= TRUE;
          TEMP_NICKEL_COUNT <= TEMP_NICKEL_COUNT - 1;
        end if;
      end if;
```

```
            NEXT_NICKEL_COUNT <= TEMP_NICKEL_COUNT;
          end if;
        end process;

        -- Remember the return-change flag and
        -- the nickel count for the next cycle
        process
        begin
          wait until CLK'event and CLK = '1';
          CURRENT_RETURN_CHANGE <= NEXT_RETURN_CHANGE;
          CURRENT_NICKEL_COUNT <= NEXT_NICKEL_COUNT;
        end process;


    end BEHAVIOR;
```

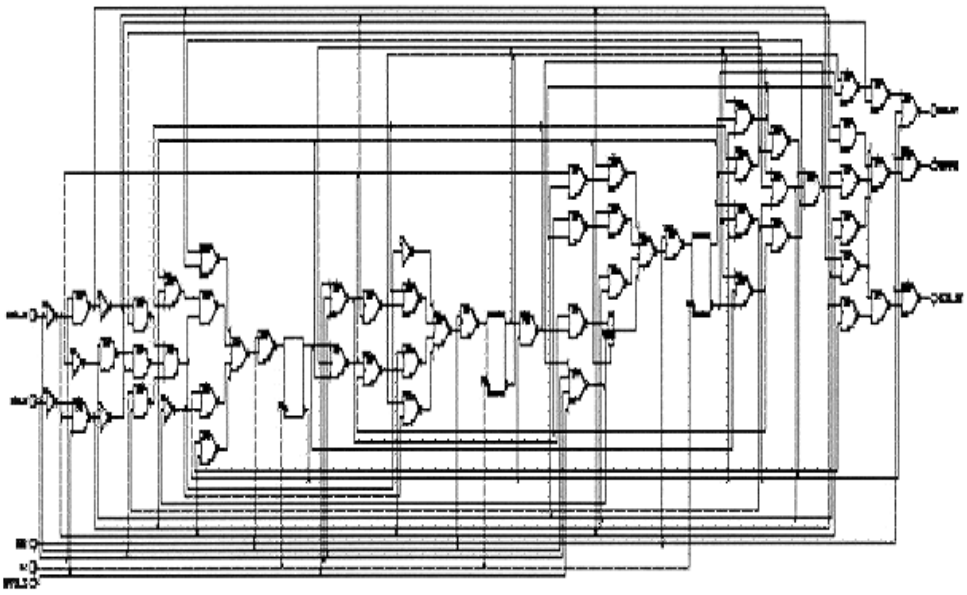


Figure A-15 Soft Drink Machine—Count Nickels Version Schematic

Carry-Lookahead Adder

This example of a carry-lookahead adder uses concurrent procedure calls to build a 32-bit carry-lookahead adder. The adder is built by

partitioning of the 32-bit input into eight slices of 4 bits each. Each of the eight slices computes propagate and generate values by using the PG procedure.

Propagate (output P from PG) is '1' for a bit position if that position propagates a carry from the next-lower position to the next-higher position. Generate (output G) is '1' for a bit position if that position generates a carry to the next-higher position, regardless of the carry-in from the next lower position. The carry-lookahead logic reads the carry-in, propagate, and generate information computed from the inputs. The logic computes the carry value for each bit position and makes the addition operation an XOR of the inputs and the carry values.

Carry Value Computations

The carry values are computed by a three-level tree of 4-bit carry-lookahead blocks.

- The first level of the tree computes the 32 carry values and the eight group-propagate and generate values. Each of the first-level group-propagate and generate values tells if that 4-bit slice propagates and generates carry values from the next-lower group to the next-higher group. The first-level lookahead blocks read the group carry computed at the second level.
- The second-level lookahead blocks read the group-propagate and generate information from the four first-level blocks and then compute their own group-propagate and generate information. The second-level lookahead blocks also read group carry information computed at the third level to compute the carries for each of the third-level blocks.
- The third-level block reads the propagate and generate information of the second level to compute a propagate and generate value for the entire adder. It also reads the external carry to compute each second-level carry. The carry-out for the adder is '1' if the third-level generate is '1' or if the third-level propagate is '1' and the external carry is '1'.

  The third-level carry-lookahead block is capable of processing four second-level blocks. But because there are only two second-level blocks, the high-order 2 bits of the computed carry are ignored; the high-order two bits of the generate input to the

third-level are set to zero, ”00”; and the propagate high-order bits are set to ”11”. These settings cause the unused portion to propagate carries but not to generate them. The following figure shows the overall structure for the carry-lookahead adder.

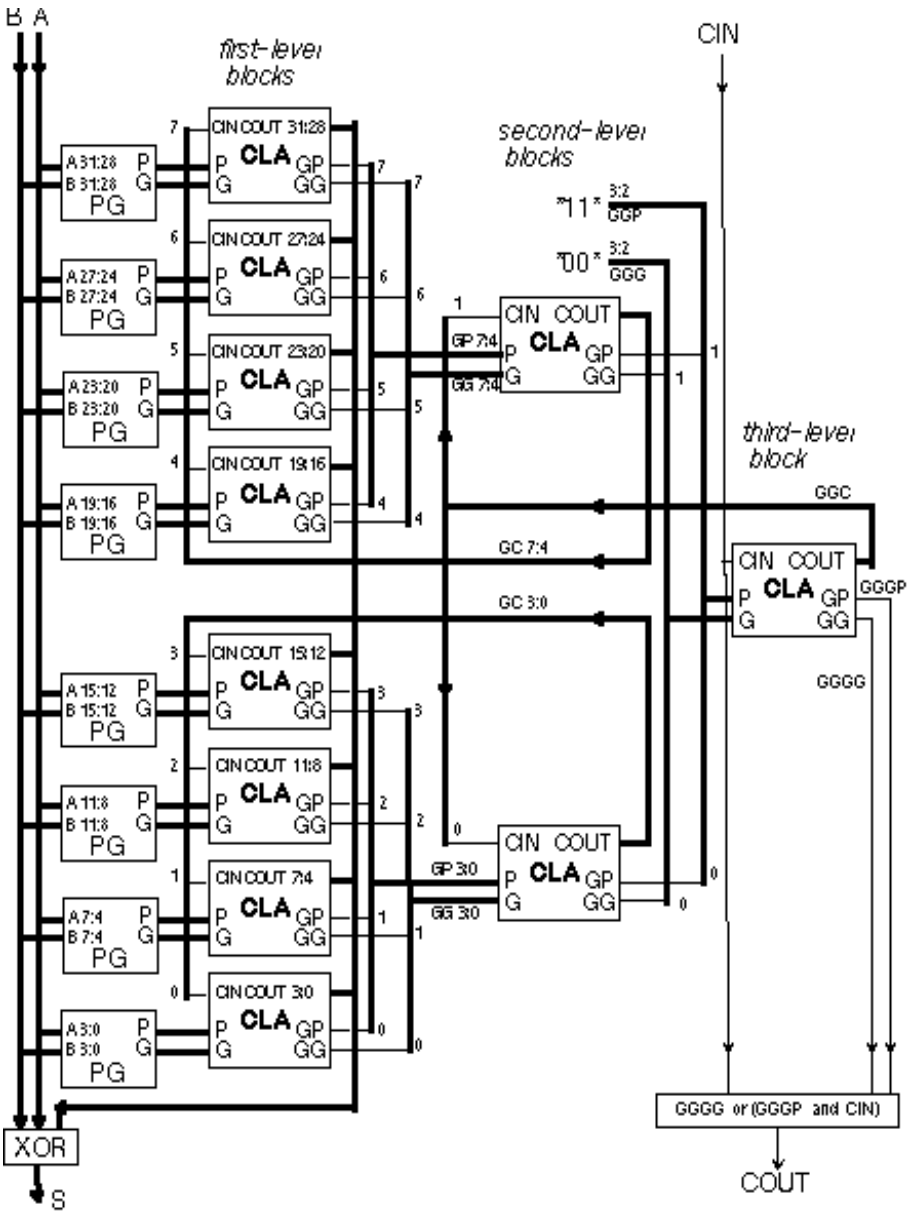


Figure A-16 Carry-Lookahead Adder Block Diagram

The VHDL implementation of the design in the previous figure is accomplished with four procedures:

- CLA—Names a 4-bit carry-lookahead block.
- PG—Computes first-level propagate and generate information.
- SUM—Computes the sum by adding the XOR values to the inputs with the carry values computed by CLA.
- BITSLICE—Collects the first-level CLA blocks, the PG computations, and the SUM. This procedure performs all the work for a 4-bit value except for the second- and third-level lookaheads.

The following example shows a VHDL description of the adder.

```
package LOCAL is
   constant N:    INTEGER := 4;

procedure BITSLICE(
      A, B: in BIT_VECTOR(3 downto 0);
      CIN: in BIT;
      signal S: out BIT_VECTOR(3 downto 0);
      signal GP, GG: out BIT);
  procedure PG(
      A, B: in BIT_VECTOR(3 downto 0);
      P, G: out BIT_VECTOR(3 downto 0));
  function SUM(A, B, C: BIT_VECTOR(3 downto 0))
      return BIT_VECTOR;
  procedure CLA(
      P, G: in BIT_VECTOR(3 downto 0);
      CIN: in BIT;
      C: out BIT_VECTOR(3 downto 0);
      signal GP, GG: out BIT);
end LOCAL;

package body LOCAL is
  ------------------------------------------------
  -- Compute sum and group outputs from a, b, cin
  ------------------------------------------------
procedure BITSLICE(
      A, B: in BIT_VECTOR(3 downto 0);
      CIN: in BIT;
      signal S: out BIT_VECTOR(3 downto 0);
      signal GP, GG: out BIT) is

    variable P, G, C: BIT_VECTOR(3 downto 0);
  begin
    PG(A, B, P, G);
```

```
    CLA(P, G, CIN, C, GP, GG);
    S <= SUM(A, B, C);
  end;

-----------------------------------------------------
  -- Compute propagate and generate from input bits
-----------------------------------------------------

procedure PG(A, B: in BIT_VECTOR(3 downto 0);
               P, G: out BIT_VECTOR(3 downto 0)) is

begin
    P <= A or B;
    G <= A and B;
end;

-----------------------------------------------------
  -- Compute sum from the input bits and the carries
-----------------------------------------------------

function SUM(A, B, C: BIT_VECTOR(3 downto 0))
      return BIT_VECTOR is

begin
    return(A xor B xor C);
  end;

--------------------------------
  -- 4-bit carry-lookahead block
--------------------------------

procedure CLA(
      P, G: in BIT_VECTOR(3 downto 0);
      CIN: in BIT;
      C: out BIT_VECTOR(3 downto 0);
      signal GP, GG: out BIT) is
    variable TEMP_GP, TEMP_GG, LAST_C: BIT;
begin
    TEMP_GP <= P(0);
    TEMP_GG <= G(0);
    LAST_C <= CIN;
    C(0) <= CIN;

    for I in 1 to N-1 loop
      TEMP_GP <= TEMP_GP and P(I);
      TEMP_GG <= (TEMP_GG and P(I)) or G(I);
      LAST_C <= (LAST_C and P(I-1)) or G(I-1);
      C(I) <= LAST_C;
    end loop;
```

```
    GP <= TEMP_GP;
    GG <= TEMP_GG;
  end;
end LOCAL;

use WORK.LOCAL.ALL;

--------------------------------
-- A 32-bit carry-lookahead adder
--------------------------------

entity ADDER is
  port(A, B: in BIT_VECTOR(31 downto 0);
       CIN: in BIT;
       S: out BIT_VECTOR(31 downto 0);
       COUT: out BIT);
end ADDER;
architecture BEHAVIOR of ADDER is

  signal GG,GP,GC: BIT_VECTOR(7 downto 0);
    -- First-level generate, propagate, carry
  signal GGG, GGP, GGC: BIT_VECTOR(3 downto 0);
    -- Second-level gen, prop, carry
  signal GGGG, GGGP: BIT;
    -- Third-level gen, prop

begin
  -- Compute Sum and 1st-level Generate and Propagate
  -- Use input data and the 1st-level Carries computed
  -- later.
  BITSLICE(A( 3 downto  0),B( 3 downto  0),GC(0),
           S( 3 downto  0),GP(0), GG(0));
  BITSLICE(A( 7 downto  4),B( 7 downto  4),GC(1),
           S( 7 downto  4),GP(1), GG(1));
  BITSLICE(A(11 downto  8),B(11 downto  8),GC(2),
           S(11 downto  8),GP(2), GG(2));
  BITSLICE(A(15 downto 12),B(15 downto 12),GC(3),
           S(15 downto 12),GP(3), GG(3));
  BITSLICE(A(19 downto 16),B(19 downto 16),GC(4),
           S(19 downto 16),GP(4), GG(4));
  BITSLICE(A(23 downto 20),B(23 downto 20),GC(5),
           S(23 downto 20),GP(5), GG(5));
  BITSLICE(A(27 downto 24),B(27 downto 24),GC(6),
           S(27 downto 24),GP(6), GG(6));
  BITSLICE(A(31 downto 28),B(31 downto 28),GC(7),
           S(31 downto 28),GP(7), GG(7));
```

```
  -- Compute first-level Carries and second-level
  -- generate and propagate.
  -- Use first-level Generate, Propagate, and
  -- second-level carry.
  process(GP, GG, GGC)
    variable TEMP: BIT_VECTOR(3 downto 0);
  begin
    CLA(GP(3 downto 0), GG(3 downto 0), GGC(0), TEMP,
        GGP(0), GGG(0));
    GC(3 downto 0) <= TEMP;
  end process;

process(GP, GG, GGC)
    variable TEMP: BIT_VECTOR(3 downto 0);
  begin
    CLA(GP(7 downto 4), GG(7 downto 4), GGC(1), TEMP,
        GGP(1), GGG(1));
    GC(7 downto 4) <= TEMP;
  end process;

  -- Compute second-level Carry and third-level
  --   Generate and Propagate
  -- Use second-level Generate, Propagate and Carry-in
  --   (CIN)
  process(GGP, GGG, CIN)
    variable TEMP: BIT_VECTOR(3 downto 0);
  begin
    CLA(GGP, GGG, CIN, TEMP, GGGP, GGGG);
    GGC <= TEMP;
  end process;

  -- Assign unused bits of second-level Generate and
  --   Propagate
  GGP(3 downto 2) <= "11";
  GGG(3 downto 2) <= "00";

  -- Compute Carry-out (COUT)
  -- Use third-level Generate and Propagate and
  --   Carry-in (CIN).
  COUT <= GGGG or (GGGP and CIN);
end BEHAVIOR;
```

Implementation

In the carry-lookahead adder implementation, procedures perform the computation of the design. The procedures can also be in the form of separate entities and used by component instantiation, producing

a hierarchical design. Foundation Express does not collapse a hierarchy of entities, but it does collapse the procedure call hierarchy into one design.

The keyword signal is included before some of the interface parameter declarations. This keyword is required for the out formal parameters when the actual parameters must be signals.

The output parameter C from the CLA procedure is not declared as a signal; thus, it is not allowed in a concurrent procedure call. Only signals can be used in such calls. To overcome this problem, subprocesses are used, declaring a temporary variable TEMP. TEMP receives the value of the C parameter and assigns it to the appropriate signal (a generally useful technique).

Serial-to-Parallel Converter—Counting Bits

This example shows the design of a serial-to-parallel converter that reads a serial, bit-stream input and produces an 8-bit output.

The design reads the following inputs.

- SERIAL_IN—The serial input data.
- RESET—The input that, when it is '1', causes the converter to reset. All outputs are set to 0, and the converter is prepared to read the next serial word.
- CLOCK—The value of RESET and SERIAL_IN, which is read on the positive transition of this clock. Outputs of the converter are also valid only on positive transitions.

The design produces the following outputs:

- PARALLEL_OUT—The 8-bit value read from the SERIAL_IN port.
- READ_ENABLE—The output that, when it is '1' on the positive transition of CLOCK, causes the data on PARALLEL_OUT to be read.
- PARITY_ERROR—The output that, when it is '1' on the positive transition of CLOCK, indicates that a parity error has been detected on the SERIAL_IN port. When a parity error is detected, the converter halts until restarted by the RESET port.

Input Format

When no data is being transmitted to the serial port, keep it at a value of ’0’. Each 8-bit value requires ten clock cycles to read it. On the eleventh clock cycle, the parallel output value can be read.

In the first cycle, a ’1’ is placed on the serial input. This assignment indicates that an 8-bit value follows. The next eight cycles transmit each bit of the value. The most significant bit is transmitted first. The tenth cycle transmits the parity of the 8-bit value. It must be ’0’ if an even number of ’1’ values are in the 8-bit data, and ’1’ otherwise. If the converter detects a parity error, it sets the PARITY_ERROR output to ’1’ and waits until the value is reset.

On the eleventh cycle, the READ_ENABLE output is set to ’1’ and the 8-bit value can be read from the PARALLEL_OUT port. If the SERIAL_IN port has a ’1’ on the eleventh cycle, another 8-bit value is read immediately; otherwise, the converter waits until SERIAL_IN goes to ’1’.

The following figure shows the timing of this design.

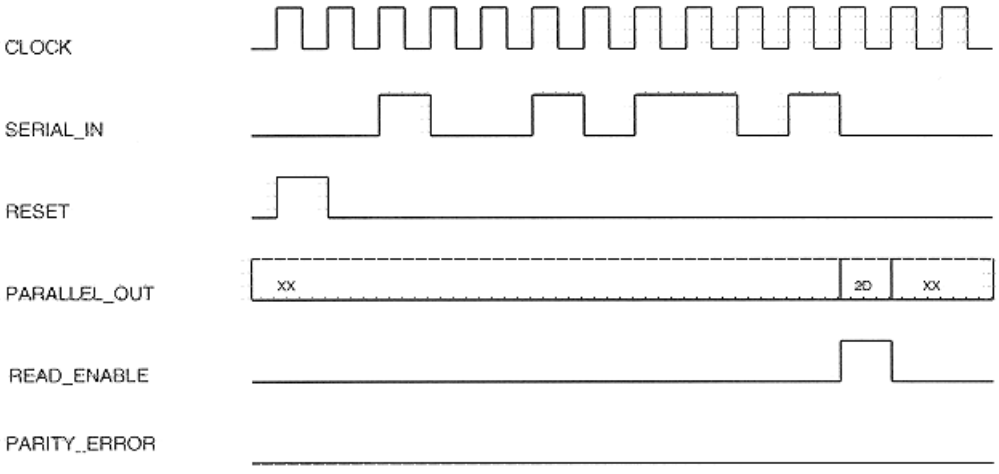


Figure A-17 Sample Waveform through the Converter

Implementation Details

The implementation of the converter is as a four-state finite-state machine with synchronous reset. When a reset is detected, the converter enters a WAIT_FOR_START state. The description of each state follows.

- WAIT_FOR_START

  Stay in this state until a '1' is detected on the serial input. When a '1' is detected, clear the PARALLEL_OUT registers and go to the READ_BITS state.

- READ_BITS

  If the value of the current_bit_position counter is 8, all 8 bits have been read. Check the computed parity with the transmitted parity. If it is correct, go to the ALLOW_READ state; otherwise, go to the PARITY_ERROR state.

  If all 8 bits have not yet been read, set the appropriate bit in the PARALLEL_OUT buffer to the SERIAL_IN value, compute the parity of the bits read so far, and increment the current_bit_position.

- ALLOW_READ

  This is the state where the outside world reads the PARALLEL_OUT value. When that value is read, the design returns to the WAIT_FOR_START state.

- PARITY_ERROR_DETECTED

  In this state, the PARITY_ERROR output is set to '1' and nothing else is done.

This design has four values stored in registers.

- CURRENT_STATE

  Remembers the state as of the last clock edge.

- CURRENT_BIT_POSITION

  Remembers how many bits have been read so far.

- CURRENT_PARITY

  Keeps a running XOR of the bits read.

- CURRENT_PARALLEL_OUT

  Stores each parallel bit as it is found.

The design has two processes: the combinatorial NEXT_ST containing the combinatorial logic and the sequential SYNCH that is clocked.

NEXT_ST performs all the computations and state assignments. The NEXT_ST process starts by assigning default values to all the signals it drives. This assignment guarantees that all signals are driven under all conditions. Next, the RESET input is processed. If RESET is not active, a case statement determines the current state and its computations. State transitions are performed by assigning the next state's value you want to the NEXT_STATE signal.

The serial-to-parallel conversion itself is performed by these two statements in the NEXT_ST process.

```
NEXT_PARALLEL_OUT(CURRENT_BIT_POSITION) <= SERIAL_IN;
NEXT_BIT_POSITION <= CURRENT_BIT_POSITION + 1;
```

The first statement assigns the current serial input bit to a particular bit of the parallel output. The second statement increments the next bit position to be assigned.

SYNCH registers and updates the stored values previously described. Each registered signal has two parts, NEXT_... and CURRENT_... :

- NEXT_...

  Signals hold values computed by the NEXT_ST process.

- CURRENT_...

  Signals hold the values driven by the SYNCH process. The CURRENT_... signals hold the values of the NEXT_... signals as of the last clock edge.

The following example shows a VHDL description of the converter.

```
-- Serial-to-Parallel Converter, counting bits

package TYPES is
  -- Declares types used in the rest of the design
  type STATE_TYPE is (WAIT_FOR_START,
                      READ_BITS,
                      PARITY_ERROR_DETECTED,
```

```
                        ALLOW_READ);
  constant PARALLEL_BIT_COUNT: INTEGER := 8;
  subtype PARALLEL_RANGE is INTEGER
      range 0 to (PARALLEL_BIT_COUNT-1);
  subtype PARALLEL_TYPE is
BIT_VECTOR(PARALLEL_RANGE);
end TYPES;

use WORK.TYPES.ALL;     -- Use the TYPES package

entity SER_PAR is       -- Declare the interface
  port(SERIAL_IN, CLOCK, RESET: in BIT;
       PARALLEL_OUT: out PARALLEL_TYPE;
       PARITY_ERROR, READ_ENABLE: out BIT);
end SER_PAR;

architecture BEHAVIOR of SER_PAR is
  -- Signals for stored values
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
  signal CURRENT_PARITY, NEXT_PARITY: BIT;
  signal CURRENT_BIT_POSITION, NEXT_BIT_POSITION:
      INTEGER range PARALLEL_BIT_COUNT downto 0;
  signal CURRENT_PARALLEL_OUT, NEXT_PARALLEL_OUT:
      PARALLEL_TYPE;
begin
NEXT_ST: process(SERIAL_IN, CURRENT_STATE, RESET,
                 CURRENT_BIT_POSITION, CURRENT_PARITY,
                  CURRENT_PARALLEL_OUT)
  -- This process computes all outputs, the next
  --   state, and the next value of all stored values
  begin
    PARITY_ERROR <= '0'; -- Default values for all
    READ_ENABLE <= '0';  --  outputs and stored values
    NEXT_STATE <= CURRENT_STATE;
    NEXT_BIT_POSITION <= 0;
    NEXT_PARITY <= '0';
    NEXT_PARALLEL_OUT <= CURRENT_PARALLEL_OUT;

if (RESET = '1') then      -- Synchronous reset
      NEXT_STATE <= WAIT_FOR_START;
    else
      case CURRENT_STATE is   -- State processing
        when WAIT_FOR_START =>
          if (SERIAL_IN = '1') then
            NEXT_STATE <= READ_BITS;
            NEXT_PARALLEL_OUT <=
                PARALLEL_TYPE'(others=>'0');
```

```
            end if;
          when READ_BITS =>
            if (CURRENT_BIT_POSITION =
                PARALLEL_BIT_COUNT) then
              if (CURRENT_PARITY = SERIAL_IN) then
                NEXT_STATE <= ALLOW_READ;
                READ_ENABLE <= '1';
              else
                NEXT_STATE <= PARITY_ERROR_DETECTED;
              end if;
            else
             NEXT_PARALLEL_OUT(CURRENT_BIT_POSITION) <=
                 SERIAL_IN;
              NEXT_BIT_POSITION <=
                 CURRENT_BIT_POSITION + 1;
              NEXT_PARITY <= CURRENT_PARITY xor
                             SERIAL_IN;
            end if;
          when PARITY_ERROR_DETECTED =>
            PARITY_ERROR <= '1';
          when ALLOW_READ =>
            NEXT_STATE <= WAIT_FOR_START;
        end case;
      end if;
    end process NEXT_ST;

  SYNCH: process
      -- This process remembers the stored values
      --    across clock cycles
  begin
      wait until CLOCK'event and CLOCK = '1';
      CURRENT_STATE <= NEXT_STATE;
      CURRENT_BIT_POSITION <= NEXT_BIT_POSITION;
      CURRENT_PARITY <= NEXT_PARITY;
      CURRENT_PARALLEL_OUT <= NEXT_PARALLEL_OUT;
  end process SYNCH;

  PARALLEL_OUT <= CURRENT_PARALLEL_OUT;

  end BEHAVIOR;
```

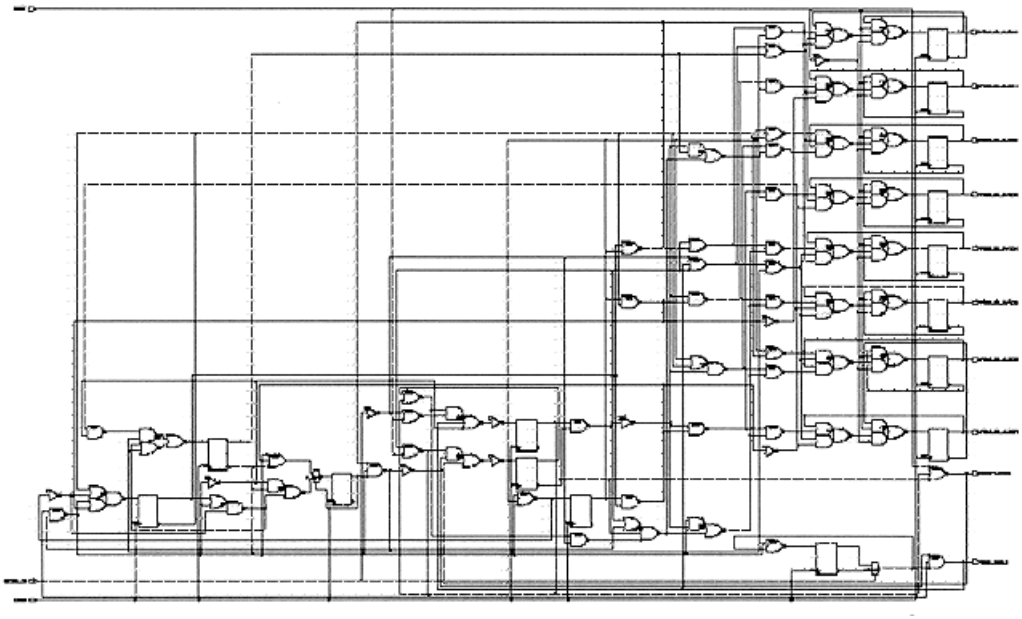


Figure A-18 Serial-to Parallet Converter—Counting Bits Schematic

Serial-to-Parallel Converter—Shifting Bits

This example describes another implementation of the serial-to-parallel converter in the last example. This design performs the same function as the previous one but uses a different algorithm to do the conversion.

The previous implementation used a counter to indicate the bit of the output that was set when a new serial bit was read. In this implementation, the serial bits are shifted into place. Before the conversion occurs, a '1' is placed in the least-significant bit position. When that '1' is shifted out of the most significant position (position 0), the signal NEXT_HIGH_BIT is set to '1' and the conversion is complete.

The following example shows the listing of the second implementation. The differences are highlighted in bold. The differences relate to the removal of the ..._BIT_POSITION signals, the addition of ..._HIGH_BIT signals, and the change in the way NEXT_PARALLEL_OUT is computed.

```
package TYPES is
  -- Declares types used in the rest of the design
  type STATE_TYPE is (WAIT_FOR_START,
                      READ_BITS,
                      PARITY_ERROR_DETECTED,
                      ALLOW_READ);
  constant PARALLEL_BIT_COUNT: INTEGER := 8;
  subtype PARALLEL_RANGE is INTEGER
      range 0 to (PARALLEL_BIT_COUNT-1);
  subtype PARALLEL_TYPE is
           BIT_VECTOR(PARALLEL_RANGE);
end TYPES;

use WORK.TYPES.ALL;      -- Use the TYPES package

entity SER_PAR is        -- Declare the interface
  port(SERIAL_IN, CLOCK, RESET: in BIT;
       PARALLEL_OUT: out PARALLEL_TYPE;
       PARITY_ERROR, READ_ENABLE: out BIT);
end SER_PAR;

architecture BEHAVIOR of SER_PAR is
  -- Signals for stored values
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;

  signal CURRENT_PARITY, NEXT_PARITY: BIT;
  signal CURRENT_HIGH_BIT, NEXT_HIGH_BIT: BIT;
  signal CURRENT_PARALLEL_OUT, NEXT_PARALLEL_OUT:
     PARALLEL_TYPE;
begin

NEXT_ST: process(SERIAL_IN, CURRENT_STATE, RESET,
                 CURRENT_HIGH_BIT, CURRENT_PARITY,
                 CURRENT_PARALLEL_OUT)
  -- This process computes all outputs, the next
  --   state, and the next value of all stored values
  begin
    PARITY_ERROR <= '0'; -- Default values for all
    READ_ENABLE <= '0';  --  outputs and stored values
    NEXT_STATE <= CURRENT_STATE;
    NEXT_HIGH_BIT <= '0';
    NEXT_PARITY <= '0';
    NEXT_PARALLEL_OUT <= PARALLEL_TYPE'(others=>'0');
    if(RESET = '1') then      -- Synchronous reset
      NEXT_STATE <= WAIT_FOR_START;
    else
      case CURRENT_STATE is   -- State processing
        when WAIT_FOR_START =>
```

```
                if (SERIAL_IN = '1') then
                  NEXT_STATE <= READ_BITS;
                  NEXT_PARALLEL_OUT <=
                      PARALLEL_TYPE'(others=>'0');
                end if;
              when READ_BITS =>
                if (CURRENT_HIGH_BIT = '1') then
                  if (CURRENT_PARITY = SERIAL_IN) then
                    NEXT_STATE <= ALLOW_READ;
                    READ_ENABLE <= '1';
                  else
                    NEXT_STATE <= PARITY_ERROR_DETECTED;
                  end if;
                else
                  NEXT_HIGH_BIT <= CURRENT_PARALLEL_OUT(0);
                  NEXT_PARALLEL_OUT <=
                      CURRENT_PARALLEL_OUT(
                          1 to PARALLEL_BIT_COUNT-1) &
                      SERIAL_IN;
                  NEXT_PARITY <= CURRENT_PARITY xor
                                 SERIAL_IN;
                end if;
              when PARITY_ERROR_DETECTED =>
                PARITY_ERROR <= '1';
              when ALLOW_READ =>
                NEXT_STATE <= WAIT_FOR_START;
            end case;
          end if;
        end process NEXT_ST;

        SYNCH: process
          -- This process remembers the stored values
          --    across clock cycles
        begin
          wait until CLOCK'event and CLOCK = '1';
          CURRENT_STATE <= NEXT_STATE;
          CURRENT_HIGH_BIT <= NEXT_HIGH_BIT;
          CURRENT_PARITY <= NEXT_PARITY;
          CURRENT_PARALLEL_OUT <= NEXT_PARALLEL_OUT;
        end process SYNCH;

        PARALLEL_OUT <= CURRENT_PARALLEL_OUT;

      end BEHAVIOR;
```

Note: The synthesized schematic for the shifter implementation is much simpler than that of the previous count implementation in the

example of the serial-to-parallel converter—counting bits. It is simpler because the shifter algorithm is inherently easier to implement.

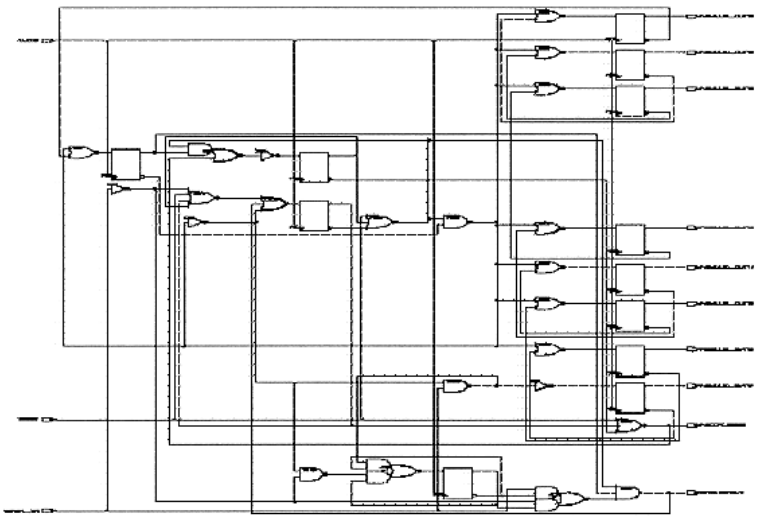


Figure A-19 Serial-to-Parallel Converter—Shifting Bits Schematic

With the count algorithm, each of the flip-flops holding the PARALLEL_OUT bits needed logic that decoded the value stored in the BIT_POSITION flip-flops to see when to route in the value of SERIAL_IN. Also, the BIT_POSITION flip-flops needed an incrementer to compute their next value.

In contrast, the shifter algorithm requires neither an incrementer nor flip-flops to hold BIT_POSITION. Additionally, the logic in front of most PARALLEL_OUT bits needs to read only the value of the previous flip-flop or '0'. The value depends on whether bits are currently being read. In the shifter algorithm, the SERIAL_IN port needs to be connected only to the least significant bit (number 7) of the PARALLEL_OUT flip-flops.

These two implementations illustrate the importance of designing efficient algorithms. Both work properly, but the shifter algorithm produces a faster, more area-efficient design.

Programmable Logic Arrays

This example shows a way to build programmable logic arrays (PLAs) in VHDL. The PLA function uses an input lookup vector as an index into a constant PLA table and then returns the output vector specified by the PLA.

The PLA table is an array of PLA rows, where each row is an array of PLA elements. Each element is either a one, a zero, a minus, or a space ('1', '0', '–', or ' '). The table is split between an input plane and an output plane. The input plane is specified by zeros, ones, and minuses. The output plane is specified by zeros and ones. The two planes' values are separated by a space.

In the PLA function, the output vector is first initialized to be all zeros. When the input vector matches an input plane in a row of the PLA table, the ones in the output plane are assigned to the corresponding bits in the output vector. A match is determined as follows.

- If a zero or one is in the input plane, the input vector must have the same value in the same position.
- If a minus is in the input plane, it matches any input vector value at that position.

The generic PLA table types and the PLA function are defined in a package named LOCAL. An entity PLA_VHDL that uses LOCAL needs only to specify its PLA table as a constant, then call the PLA function.

The PLA function does not explicitly depend on the size of the PLA. To change the size of the PLA, change the initialization of the TABLE constant and the initialization of the constants INPUT_COUNT, OUTPUT_COUNT, and ROW_COUNT. In the following example, these constants are initialized to a PLA equivalent to the ROM shown previously in the ROM example in the "Read-Only Memory" section of this appendix. Accordingly, the synthesized schematic is the same as that of the ROM, with one difference: in the example of the implementation of a ROM in random logic, the DATA output port range is 1 to 5; in the following example, the OUT_VECTOR output port range is 4 down to 0.

```
package LOCAL is
  constant INPUT_COUNT: INTEGER := 3;
  constant OUTPUT_COUNT: INTEGER := 5;
  constant ROW_COUNT: INTEGER := 6;
```

```
  constant ROW_SIZE: INTEGER := INPUT_COUNT +
                                OUTPUT_COUNT + 1;
  type PLA_ELEMENT is ('1', '0', '-', ' ');
  type PLA_VECTOR is
      array (INTEGER range <>) of PLA_ELEMENT;
  subtype PLA_ROW is
      PLA_VECTOR(ROW_SIZE - 1 downto 0);
  subtype PLA_OUTPUT is
      PLA_VECTOR(OUTPUT_COUNT - 1 downto 0);
  type PLA_TABLE is
      array(ROW_COUNT - 1 downto 0) of PLA_ROW;

  function PLA(IN_VECTOR: BIT_VECTOR;
               TABLE: PLA_TABLE)
      return BIT_VECTOR;
end LOCAL;

package body LOCAL is

   function PLA(IN_VECTOR: BIT_VECTOR;
                TABLE: PLA_TABLE)
      return BIT_VECTOR is
    subtype RESULT_TYPE is
        BIT_VECTOR(OUTPUT_COUNT - 1 downto 0);
    variable RESULT: RESULT_TYPE;
    variable ROW: PLA_ROW;
    variable MATCH: BOOLEAN;
    variable IN_POS: INTEGER;

   begin
    RESULT <= RESULT_TYPE'(others => BIT'( '0' ));
    for I in TABLE'range loop
      ROW <= TABLE(I);
      MATCH <= TRUE;
      IN_POS <= IN_VECTOR'left;

      -- Check for match in input plane
      for J in ROW_SIZE - 1 downto OUTPUT_COUNT loop
        if(ROW(J) = PLA_ELEMENT'( '1' )) then
          MATCH <= MATCH and
                   (IN_VECTOR(IN_POS) = BIT'( '1' ));
        elsif(ROW(J) = PLA_ELEMENT'( '0' )) then
          MATCH <= MATCH and
                   (IN_VECTOR(IN_POS) = BIT'( '0' ));
        else
          null;     -- Must be minus ("don't care")
        end if;
```

```
        IN_POS <= IN_POS - 1;
      end loop;

      -- Set output plane
      if(MATCH) then
        for J in RESULT'range loop
          if(ROW(J) = PLA_ELEMENT'( '1' )) then
            RESULT(J) <= BIT'( '1' );
          end if;
        end loop;
      end if;
    end loop;
    return(RESULT);
  end;
end LOCAL;

use WORK.LOCAL.all;
entity PLA_VHDL is
  port(IN_VECTOR: BIT_VECTOR(2 downto 0);
       OUT_VECTOR: out BIT_VECTOR(4 downto 0));
end PLA_VHDL;

architecture BEHAVIOR of PLA_VHDL is
  constant TABLE: PLA_TABLE := PLA_TABLE'(
       PLA_ROW'("--- 10000"),
       PLA_ROW'("-1- 01000"),
       PLA_ROW'("0-0 00101"),
       PLA_ROW'("-1- 00101"),
       PLA_ROW'("1-1 00101"),
       PLA_ROW'("-1- 00010"));

begin
  OUT_VECTOR <= PLA(IN_VECTOR, TABLE);
end BEHAVIOR;
```

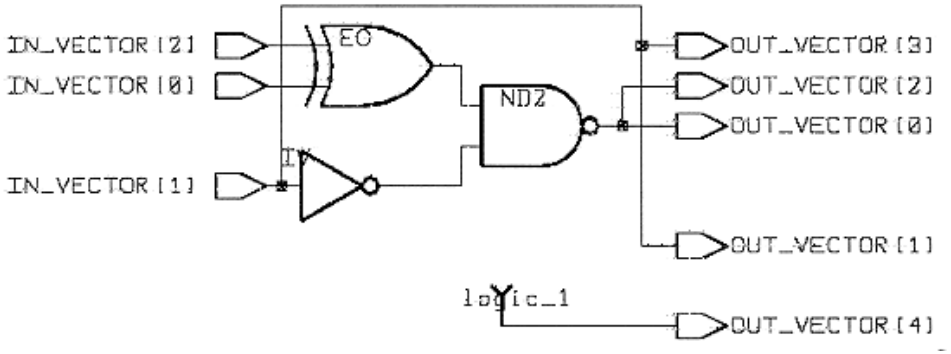


Figure A-20 Programmable Logic Array Schematic