FPGA Express
HDL Reference Manual

September 1996

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

Chapter 1
FPGA Express with Verilog HDL

FPGA Express translates and optimizes a Verilog HDL description into an internal gate-level equivalent, then compiles this representation to produce an optimized gate-level design in a given FPGA technology.

This chapter introduces the main concepts and capabilities of FPGA Express in the following sections:

- Hardware Description Languages
- FPGA Express and the Design Process
- Design Methodology

Hardware Description Languages

Hardware description languages (HDLs) describe the architecture and behavior of discrete electronic systems. Modern HDLs and their associated simulators are very powerful tools for integrated circuit designers.

A typical HDL supports a mixed-level description in which gate and netlist constructs are used with functional descriptions. This mixed-level capability enables you to describe system architectures at a very high level of abstraction, then incrementally refine a design's detailed gate-level implementation.

HDL descriptions play an important role in modern design methodology for three main reasons:

- Design functionality can be verified early in the design process. A design written as an HDL description can be simulated immediately. Design simulation at this higher level, before implementation at the gate-level, allows you to evaluate architectural and design decisions.

- FPGA Express provides Verilog compilation and *logic synthesis*, allowing you to automatically convert an HDL description to a gate-level implementation in a target FPGA technology. This step eliminates the former gate-level design bottleneck, the majority of circuit design time, and the errors introduced when you hand translate an HDL specification to gates.

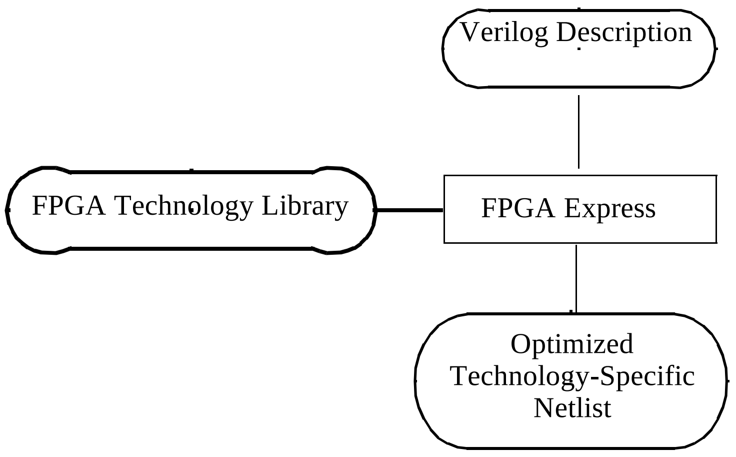
  With FPGA Express *logic optimization*, you can automatically transform a synthesized design into a smaller or faster circuit. FPGA Express provides both logic synthesis and optimization. For further information, refer to the *FPGA Express User's Guide*.

- HDL descriptions provide technology-independent documentation of a design and its functionality. An HDL description is more easily read and understood than a netlist or schematic description. Since the initial HDL design description is technology-independent, you can use it again to generate the design in a different technology, without having to translate from the original technology.

The FPGA Express Design Process

FPGA Express translates Verilog language hardware descriptions to a Synopsys internal design format. The design can then be optimized and mapped to a specific FPGA technology library by FPGA Express, as shown in Figure 1-1.

Figure 1-1 FPGA Express Design Process



FPGA Express supports a majority of the Verilog constructs.

Using FPGA Express to Compile a Verilog HDL Design

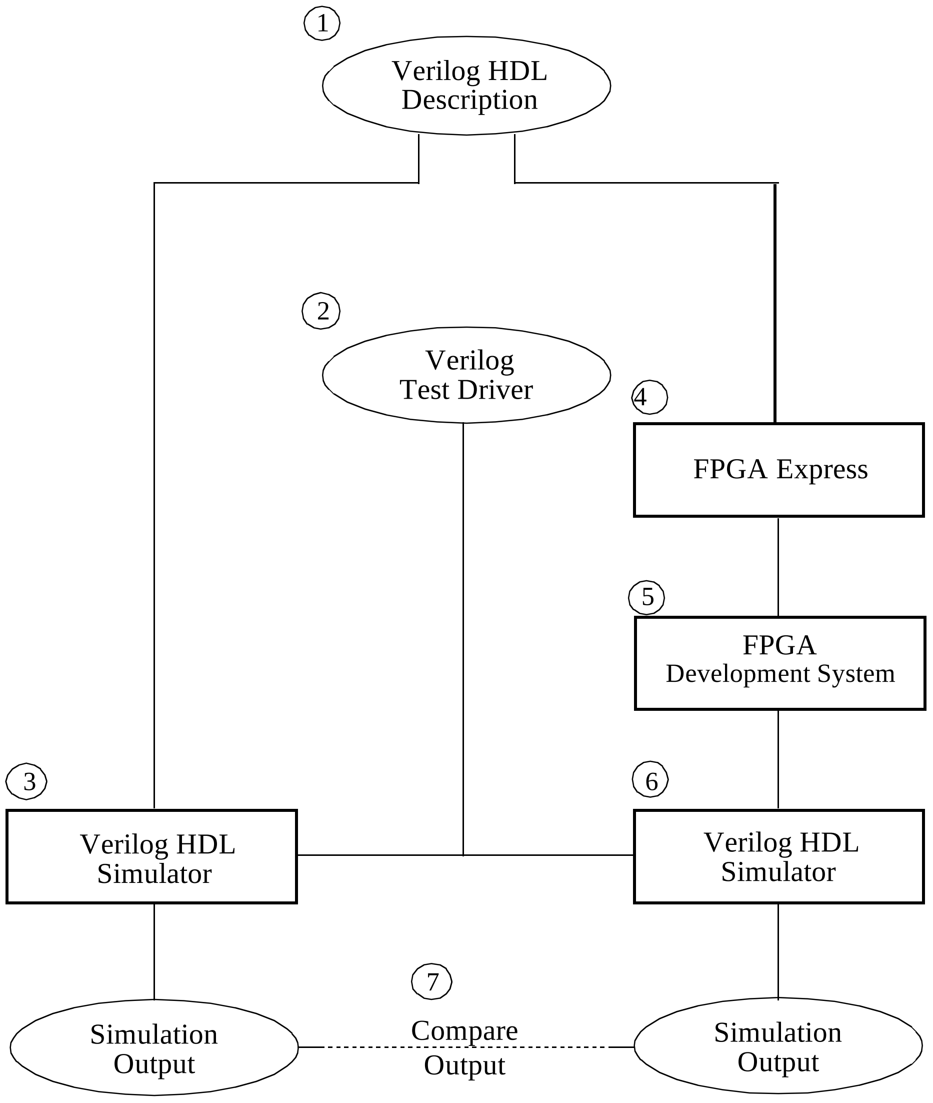
When a Verilog design is read into FPGA Express, it is converted to an internal database format so FPGA Express can synthesize and optimize the design. When FPGA Express optimizes a design, it may restructure part or all the design. You control the degree of restructuring. Options include

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

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

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

Figure 1-2 Design Flow



The steps in Figure 1-2 are explained below.

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

Verilog Example

This section takes you through a sample Verilog design session, starting with a Verilog description (source file). The ‘‘Count Zeros — Sequential Version“ example in this section is from Appendix A. The design session covers the following topics:

- A description of the design problem (count the number of zeros in a sequentially input 8-bit value)
- A listing of a Verilog design description

Verilog Design Description

The Count Zeros example illustrates a design that takes an 8-bit value and determine two things: first, that the value has exactly one sequence of 0’s in the value; and second, the number of 0’s in that sequence (if any).

A valid value is one that contains only one consecutive series of 0s. If more than one series of 0s appears, the value is invalid. A value consisting entirely of 1's is defined as a valid value. If a value is invalid, the zero counter is reset (to 0). For example, the value 00000000 is valid and has eight 0s; value 11000111 is valid and has three 0's; value 00111100 is invalid.

The circuit accepts the 8-bit data value serially, one bit per clock cycle, by using the data and clk inputs. The other two inputs are reset, which resets the circuit, and read, which causes the circuit to begin accepting the data bits.

The circuit's three outputs are

- is_legal, which is true if the data is a valid value.
- data_ready, which is true at the first invalid bit or when all eight bits have been processed.
- zeros, which is the number of zeros if is_legal is true.

Example 1-1 shows the Verilog source description for the Count Zeros circuit.

Example 1-1 Count Zeros-Sequential Version

```
module count_zeros(data,reset,read,clk,zeros,is_legal,
               data_ready);

  parameter TRUE=1, FALSE=0;

  input  data, reset, read, clk;
  output is_legal, data_ready;
  output [3:0] zeros;
  reg  [3:0] zeros;

  reg is_legal, data_ready;
  reg seenZero, new_seenZero;
  reg seenTrailing, new_seenTrailing;
  reg new_is_legal;
  reg new_data_ready;
  reg [3:0] new_zeros;
  reg [2:0] bits_seen, new_bits_seen;

always @ ( data or reset or read or is_legal
         or data_ready or seenTrailing or
          seenZero ) begin
     if ( reset ) begin
        new_data_ready   = FALSE;
        new_is_legal     = TRUE;
        new_seenZero     = FALSE;
        new_seenTrailing = FALSE;
        new_zeros        = 0;
        new_bits_seen    = 0;
     end
     else begin
        new_is_legal     = is_legal;
        new_seenZero     = seenZero;
        new_seenTrailing = seenTrailing;
        new_zeros        = zeros;
        new_bits_seen    = bits_seen;
        new_data_ready   = data_ready;
         if ( read ) begin
           if ( seenTrailing  && (data == 0) )
              begin
              new_is_legal   = FALSE;
              new_zeros      = 0;
              new_data_ready = TRUE;
              end
           else if ( seenZero && (data == 1'b1) )
              new_seenTrailing = TRUE;
           else if ( data == 1'b0 ) begin
              new_seenZero = TRUE;
              new_zeros = zeros + 1;
              end

           if ( bits_seen == 7 )
              new_data_ready = TRUE;
           else
              new_bits_seen = bits_seen+1;
        end
     end
  end
```

```
always @ ( posedge clk) begin
    zeros = new_zeros;
    bits_seen = new_bits_seen;
    seenZero = new_seenZero;
    seenTrailing = new_seenTrailing;
    is_legal = new_is_legal;
    data_ready = new_data_ready;
end
endmodule
```

Chapter 2
Description Styles

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

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

- Design Hierarchy
- Structural Descriptions
- Functional Descriptions
- Mixing Structural and Functional Descriptions
- Design Constraints
- Register Selection
- Asynchronous Designs

Design Hierarchy

FPGA Express maintains the hierarchical boundaries you define when you use structural Verilog. These boundaries have two major effects:

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

Chapter 3 discusses modules and module instantiations.

Note: FPGA Express does not automatically maintain (create) the hierarchy of other nonstructural Verilog constructs such as blocks, loops, functions, and tasks. These elements of an HDL description are translated in the context of their design. After analyzing and implementing a design, you can use the FPGA Express Implementation Window to group the gates in a block, function, or task. Refer to the FPGA Express User's Guide for further information.

The choice of hierarchical boundaries has a significant effect on the quality of the synthesized design. Using FPGA Express, you can optimize a design while preserving these hierarchical boundaries. However, FPGA Express only partially optimizes logic across hierarchical modules. Full optimization is possible across those parts of the design hierarchy that are collapsed in FPGA Express.

Structural Descriptions

The structural elements of a Verilog structural description consist of generic logic gates, library-specific components, and user-defined components connected by wires. In one way, a structural description can be viewed as a simple netlist composed of nets that connect instantiations of gates. However, unlike a netlist, nets in the structural description can be driven by an arbitrary expression that describes the value assigned to the net. A statement that drives an arbitrary expression onto a net is called a *continuous assignment*. Continuous assignments are convenient links between pure netlist descriptions and functional descriptions.

A Verilog structural description can define a range of hierarchical and gate-level constructs, including module definitions, module instantiations, and netlist connections. Refer to Chapter 3, “Structural Descriptions,” for more information.

Functional Descriptions

The functional elements of a Verilog description consist of `function` declarations, `task` statements, and `always` blocks. These elements describe the function of the circuit but do not describe its physical makeup, layout, or choice of gates and components.

You can construct functional descriptions with the Verilog functional constructs described in Chapter 5. These constructs can appear within functions or `always` blocks. Functions imply only combinational logic. `always` blocks can imply either combinational or sequential logic.

Although many Verilog functional constructs appear sequential in nature (for example, `for` loops and multiple assignments to the same variable), these constructs describe combinational-logic networks. Other functional constructs imply sequential-logic networks. Latches and registers are inferred from these constructs. Refer to Chapter 6 for details.

Mixing Structural and Functional Descriptions

When you use a functional description style in a design, the combinational portions of a design are typically described in Verilog functions, `always` blocks, and assignments. The complexity of the logic determines whether you use one or many functions.

Example 2-1, shows how structural and functional description styles are mixed in a design specification. In Example 2–1, the function `detect_ logic` determines whether the input bit is a 0 or a 1. After this determination is made, `detect_logic` sets `ns` to the next state of the machine. An `always` block infers flip-flops to hold the state information between clock cycles.

Elements of a design can be specified directly as module instantiations at the structural level. For example, see the three-state buffer, `t1`, in Example 2-1. (Note that three-state buffers *can* be inferred. For more information, refer to “Three-State Inference” in Chapter 6.) You can also use this description style to identify the wires and ports that carry information from one part of the design to another.

Example 2-1 Mixed Structural and Functional Descriptions

```
// This finite state machine (Mealy type) reads one
// bit per clock cycle and detects three or more
// consecutive 1s.

module three_ones( signal, clock, detect, output_
enable);
input signal, clock, output_enable;
output detect;

// Declare current state and next state variables.
reg [1:0] cs;
reg [1:0] ns;
wire ungated_detect;

// declare the symbolic names for states
parameter NO_ONES = 0, ONE_ONE = 1,
          TWO_ONES = 2, AT_LEAST_THREE_ONES = 3;

// ************* STRUCTURAL DESCRIPTION
****************
// Instance of a three-state gate that enables output
three_state t1 (ungated_detect, output_enable,
detect);

// *************I***  ALWAYS BLOCK
********************
// always block infers flip-flops to hold the state
of
// the FSM.
always @ ( posedge clock ) begin
     cs = ns;
end

// ************* FUNCTIONAL DESCRIPTION
****************
function detect_logic;
    input [1:0] cs;
    input signal;

    begin
        detect_logic = 0;   // default value

        if ( signal == 0 )  // bit is zero
            ns = NO_ONES;
        else                // bit is one, increment
state
            case (cs)
                NO_ONES: ns = ONE_ONE;
                ONE_ONE: ns = TWO_ONES;
                TWO_ONES, AT_LEAST_THREE_ONES:
                         begin
                           ns = AT_LEAST_THREE_ONES;
                            detect_logic = 1;
                         end
            endcase
    end
endfunction
```

```
// **************  assign STATEMENT  **************
assign ungated_detect = detect_logic( cs, signal );
endmodule
```

For a structural or functional HDL description to be synthesized, it must follow the Synopsys synthesis policy, which has three parts:

- Design methodology
- Description style
- Language constructs

Design Methodology

Design methodology refers to the synthesis design process described in Chapter 1, “Design Methodology.”

Description Style

Use the HDL design and coding style that makes the best use of the synthesis process to obtain high-quality results from FPGA Express. See Chapter 8, “Writing Efficient Circuit Descriptions,” for guidelines.

Language Constructs

The third component of the Verilog synthesis policy is the set of Verilog constructs that describe your design, determine its architecture, and give consistently good results.

Synopsys has chosen HDL constructs that maximize coding flexibility while producing consistently good results. Although FPGA Express can read the entire Verilog language, a few HDL constructs cannot be synthesized. These constructs are unsupported, because they cannot be realized in logic. For example, you cannot use simulation time as a trigger, because time is an element of the simulation process and cannot be realized. Unsupported Verilog constructs are listed in Appendix C.

Design Constraints

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

Register Selection

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

- You can directly instantiate registers into a Verilog description, selecting from any element in your FPGA library. Clocking schemes can be arbitrarily complex. You can choose between a flip-flop and a latch-based architecture. The main disadvantages to this approach are
  - The Verilog description is specific to a given technology because you choose structural elements from that technology library. However, you can isolate the portion of your design with directly instantiated registers as a separate component (module), then connect it to the rest of the design.
  - The description is more difficult to write.
- You can use some Verilog constructs to direct FPGA Express to infer registers from the description. The advantages of this approach directly counter the disadvantages of the previous approach. With register inference, the Verilog description is much easier to write, and it is technology independent. This method allows FPGA Express to select the type of component inferred, based on constraints. Therefore, if a specific component is necessary, instantiation should be used. Some types of registers and latches cannot be inferred.

See Chapter 6 for a discussion of latch and register inference.

Asynchronous Designs

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

Chapter 8 describes how to write Verilog descriptions of asynchronous designs in the section “Synthesis Issues.”

Chapter 3
Structural Descriptions

A Verilog circuit description can be one of two types: a structural description or a functional description, also referred to as an Register Transfer Level (RTL) description. A *structural* description defines the exact physical makeup of the circuit, detailing components and the connections between them. A *functional* or RTL description describes a circuit in terms of its registers and the combinational logic between the registers.

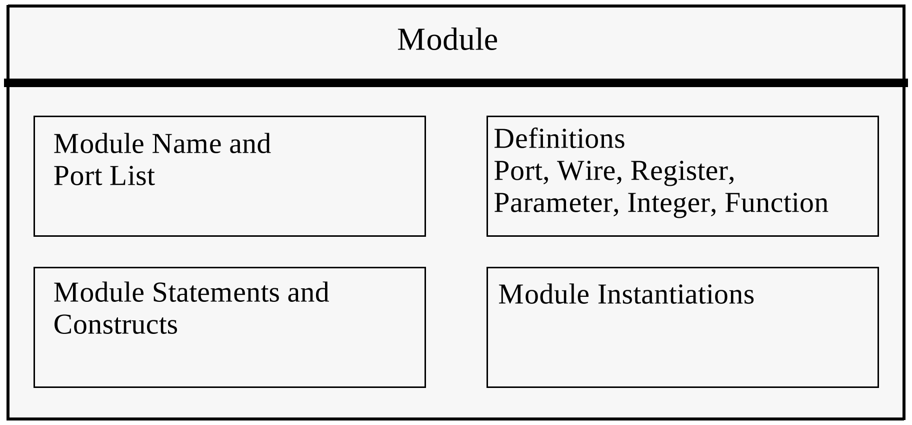
This chapter describes the construction of structural descriptions in the following sections:

- Modules
- Macromodules
- Port Definitions
- Module Statements and Constructs
- Module Instantiations

Modules

The principal design entity in the Verilog language is a *module*. A module consists of the module name, its input and output description (port definition), a description of the functionality or implementation for the module (module statements and constructs), and named instantiations. Figure 3-1 illustrates the basic structural parts of a module.

Figure 3-1 Structural Parts of a Module


Module
Module Name and Port List
Definitions Port, Wire, Register, Parameter, Integer, Function
Module Statements and Constructs
Module Instantiations

Example 3-1 shows a simple module that implements a 2-input NAND gate by instantiating an AND gate and an INV gate. The first line of the module definition provides the name of the module and a list of ports. The second and third lines give the direction for all ports. (Ports are either inputs, outputs, or bidirectionals.) A wire variable is created in the fourth line of the description. Next, the two components are *instantiated*; copies named instance1 and instance2 of the components AND and INV are created. These components are connected to the ports of the module, and are finally connected by using the variable and_out.

Example 3-1 Module Definition

```
module NAND(a,b,z);
  input  a,b;     // Inputs to nand gate
  output z;       // Outputs from nand gate
  wire   and_out; // Output from and gate

  AND instance1(a,b,and_out);
  INV instance2(and_out, z);
endmodule
```

macromodule Constructs

The macromodule construct makes simulation more efficient by merging the macromodule definition with the definition of the calling (parent) module. However, FPGA Express treats the macromodule construct as a module construct. Whether you use module or macromodule the synthesis process, the hierarchy it creates, and the end result are the same. Example 3-2 shows how to use the macromodule construct.

Example 3-2 macromodule Construct

```
macromodule adder (in1,in2,out1);
input [3:0] in1,in2;
output [4:0] out1;

assign out1 = in1 + in2;
endmodule
```

Note: When a *macromodule* ***is instantiated, a new level of hierarchy is created. You can ungroup this new level of hierarchy in the FPGA Express Implementation Window.***

Port Definitions

A port list consists of port expressions that describe the input and output interface for a module. Define the port list in parentheses after the module name, as shown below.

`module` ***`name`*** `(` ***`port_list`*** `) ;`

A port expression in a port list can be any of the following:

- An identifier
- A single bit selected from a bit vector declared within the module
- A group of bits selected from a bit vector declared within the module
- A concatenation of any of the above

Concatenation is the process of combining several single-bit or multiple-bit operands into one large bit vector. For more information on concatenation, refer to the section “Concatenations” in Chapter 4.

Each port in a port list must be declared explicitly as input, output, or bidirectional in the module with an input, output, or inout statement. (See “Port Declarations” later in this chapter.) For example, the module definition in Example 3–1 shows that module NAND has three ports, a, b, and z, connected to 1-bit nets a, b, and z. These connections are declared in the input and output statements.

Port Names

Some port expressions are *identifiers*. If the port expression is an identifier, the port name is the same as the identifier. A port expression is not an identifier if the expression is a single bit or group of bits selected from a vector of bits, or a concatenation of signals. In these cases, the port is unnamed unless you explicitly name it.

Example 3-3 shows some module definition fragments that illustrate the use of port names. The ports for module ex1 are named a, b, and z, and are connected to nets a, b, and z, respectively. The first two ports of module ex2 are unnamed; the third port is named z. The ports are connected to nets a[1], a[0], and z respectively. Module ex3 has two ports: the first port is unnamed and is connected to a concatenation of nets a and b; the second port, named z, is connected to net z.

Example 3-3 Module Port Lists

```
module ex1( a, b, z );
input a, b;
output z;
endmodule

module ex2( a[1], a[0], z );
input [1:0] a;
output z;
endmodule

module ex3( {a,b}, z );
input a,b;
output z;
endmodule
```

You can rename a port by explicitly assigning a name to a port expression with the dot (.) operator. The module definition fragments in Example 3-4 show how to rename ports. The ports for module ex4 are explicitly named in_a, in_b, and out These ports are connected to nets a, b, and z. Module ex5 shows ports named i1, i0, and z connected to nets a[1], a[0], and z, respectively. The first port for module ex6 (the concatenation of nets a and b) is named i.

Example 3-4 Naming Ports in Modules

```
module ex4( .in_a(a), .in_b(b), .out(z) );
  input a, b;
  output z;
endmodule

module ex5( .i1(a[1]), .i0(a[0]), z );
  input [1:0] a;
  output z;
endmodule

module ex6( .i({a,b}), z );
  input a,b;
  output z;
endmodule
```

Module Statements and Constructs

FPGA Express recognizes the following Verilog statements and constructs when they are used in a Verilog module:

- parameter declarations
- wire, wand, wor, tri, supply0, and supply1 declarations
- reg declarations
- input declarations
- output declarations
- inout declarations
- Continuous assignments
- Module instantiations
- Gate instantiations
- Function definitions
- always blocks
- task statements

Data declarations and assignments are described in this section. Module and gate instantiations are described later in this chapter. Function definitions, task statements, reg variables, and always blocks are described in Chapter 5, “Functional Descriptions.”

Structural Data Types

Verilog structural data types include wire, wand, wor, tri, supply0, and supply1. Although parameter does not fall into the category of structural data types, it is presented here because it is used with structural data types.

You can define an optional range for all the data types presented in this section. The range provides a means for creating a bit-vector. The syntax for a range specification is

```
[msb : lsb]
```

Expressions for msb (most significant bit) and lsb (least significant bit) must be nonnegative constant-valued expressions. Constant-valued expressions are composed only of constants, Verilog parameters, and operators.

parameter Definitions

Verilog parameters allow you to customize each instantiation of a module. By setting different values for the parameter when you instantiate the module, you can cause different logic to be constructed. For more information, see “Building Parameterized Designs,” later in this chapter.

A parameter definition represents constant values symbolically. The definition for a parameter consists of the parameter name and the value assigned to it. The value can be any constant-valued expression of integer or Boolean type, but not of type real. If you do not set the size of the parameter with a range definition or a sized constant, the parameter is unsized and defaults to a 32-bit quantity. Refer to Appendix C for a discussion of constant formats.

You can use a parameter wherever a number is allowed, and you can define a parameter anywhere within a module definition. However, the Verilog language requires that you define the parameter before you use it.

Example 3–5 shows two parameter declarations. Parameters TRUE and FALSE are unsized, and have values of 1 and 0, respectively. Parameters S0, S1, S2, and S3 have values 3, 1, 0, and 2, respectively, and are stored as 2-bit quantities.

Example 3-5 parameter Declarations

```
parameter TRUE=1, FALSE=0;
parameter [1:0] S0=3, S1=1, S2=0, S3=2;
```

wire Data Types

A wire data type in a Verilog description represents the physical wires in a circuit. A wire connects gate-level instantiations and module instantiations. The Verilog language allows you to *read* a wire value from within a function or a begin...end block, but you cannot *assign* a wire value from within a function or a begin...end block**.** (An always block is a specific type of begin...end block).

A wire does not store its value. It must be driven in one of two ways:

- By connecting the wire to the output of a gate or module.
- By assigning a value to the wire in a *continuous assignment*.

In the Verilog language, an undriven wire defaults to a value of Z (high impedance). However, FPGA Express leaves undriven wires unconnected. Multiple connections or assignments to a wire short the wires together.

In Example 3–6, two wire data types are declared: a and b. a is a single-bit wire, while b is a 3-bit vector of wires (the most significant bit (MSB) has an index of 2 and the least significant bit (LSB) has an index of 0.)

Example 3-6 wire Declarations

```
wire a;
wire [2:0] b;
```

You can assign a delay value in a wire declaration, and you can use the Verilog keywords scalared and vectored for simulation. FPGA Express accepts the syntax of these constructs, but they are ignored when the circuit is synthesized.

Note: You can use delay information for modeling, but FPGA Express ignores this delay information. If the functionality of your circuit depends on the delay information, FPGA Express might create logic with behavior that does not agree with the behavior of the simulated circuit.

wand Data Types

The wand (wired AND) data type is a specific type of wire data type.

In Example 3–7, two variables drive the variable c. The value of c is determined by the logical AND of a and b.

Example 3-7 wand (wired AND) Data Types

```
module wand_test(a, b, c);
  input a, b;
  output c;

  wand c;

  assign c = a;
  assign c = b;
 endmodule
```

You can assign a delay value in a wand declaration, and you can use the Verilog keywords scalared and vectored for simulation. FPGA Express accepts the syntax of these constructs, but they are ignored when the circuit is synthesized.

wor Data Types

The wor (wired OR) data type is a specific type of wire data type.

In Example 3–8, two variables drive the variable c. The value of c is determined by the logical OR of a and b.

Example 3-8 wor (wired-OR) Data Types

```
module wor_test(a, b, c);
  input a, b;
  output c;

  wor c;

  assign c = a;
  assign c = b;
 endmodule
```

tri Data Types

The tri (three-state) data type is a specific type of wire data type. Only one of the variables that drive the tri data type can have a non-Z (high-impedance) value. This single variable determines the value of the tri data type

Note: FPGA Express does not enforce the above condition. You must ensure that no more than one variable driving a tri data type has a value other than Z.

In Example 3-9, three variables drive the variable out.

Example 3-9 tri (Three-State) Data Types

```
module tri_test (out, condition);
  input [1:0] conditon;
  output out;

  reg a, b, c;
  tri out;

  always @ ( condition ) begin
    a = 1’bz;// set all variables to Z
    b = 1’bz;
    c = 1’bz;
     case ( condition )   // set only one variable to
non-Z
      2’b00 : a = 1’b1;
      2’b01 : b = 1’b0;
      2’b10 : c = 1’b1;
    endcase
  end

  assign out = a;         // make the tri connection
  assign out = b;
  assign out = c;
endmodule
```

supply0 / supply1 Data Types

The supply0 and supply1 data types define wires tied to logic 0 (ground) and logic 1 (power). Using supply0 and supply1 is the same as declaring a wire and assigning a 0 or a 1 to it. In Example 3–10, power is tied to logic 1 and gnd is tied to logic 0.

Example 3-10 supply0 and supply1 Constructs

```
supply0 gnd;
supply1 power;
```

reg Data Types

A reg represents a variable in Verilog. A reg can be a 1-bit quantity or a vector of bits. For a vector of bits, the range indicates the most significant bit (MSB) and least significant bit (LSB) of the vector. Both bits must be nonnegative constants, parameters, or constant-valued expressions. Example 3–11 shows some reg declarations.

Example 3-11 reg Declarations

```
reg x;// single bit
reg a,b,c;// 3 1-bit quantities
reg [7:0] q;// an 8-bit vector
```

Port Declarations

You must explicitly declare the direction (whether input, output, or bidirectional) of each port that appears in the port list of a port definition. Use the input, output, and inout statements, as described in the following sections.

input Declarations

All input ports of a module are declared with an input statement. An input is a type of wire and is governed by the syntax of wire. You can use a range specification to declare an input that is a *vector* of signals, as for input b in the following example. The input statements can appear in any order in the description but must be declared before they are used. For example:

```
input a;
input [2:0] b;
```

output Declarations

All output ports of a module are declared with an output statement. Unless otherwise defined by a reg, wand, wor, or tri declaration, an output is a type of wire and is governed by the syntax of wire. An output statement can appear in any order in the description, but you must declare it before you use it.

You can use a range specification to declare an output value that is a *vector* of signals. If you use a reg declaration for an output, the reg must have the *same range* as the vector of signals. For example:

```
output a;
output [2:0]b;
reg [2:0] b;
```

inout Declarations

You can declare bidirectional ports with the inout statement. An inout is a type of wire and is governed by the syntax of wire. FPGA Express allows you to connect only inout ports to module or gate instantiations. You must declare an inout before you use it. For example:

```
inout a;
inout [2:0]b;
```

Continuous Assignment

If you want to drive a value onto a wire, wand, wor, or tri, use a continuous assignment to specify an expression for the wire value. You can specify a continuous assignment in two ways:

- Use an explicit continuous assignment statement after the wire, wand, wor, or tri declaration.
- Specify the continuous assignment in the same line as the declaration for a wire.

Example 3–12 shows two equivalent methods for specifying a continuous assignment for wire a.

Example 3-12 Two Equivalent Continuous Assignments

```
wire a;             // declare
assign a = b & c;   // assign

wire a = b & c;     // declare and assign
```

The left side of a continuous assignment can be

- A wire, wand, wor, or tri.
- One or more bits selected from a vector.
- A concatenation of any of these.

The right side of the continuous assignment statement can be any supported Verilog operator, or any arbitrary expression that uses previously declared variables and functions. Note that you cannot assign a value to a reg in a continuous assignment.

Verilog allows you to assign drive strength for each continuous assignment statement. FPGA Express accepts drive strength, but it does not affect the synthesis of the circuit. Keep this in mind when you use drive strength in your Verilog source.

Assignments are performed bit-wise, with the low bit on the right side assigned to the low bit on the left side. If the number of bits on the right side is greater than the number on the left side, the high-order bits on the right side are discarded. If the number of bits on the left side is greater than the number on the right side, operands on the right side are zero-extended.

Module Instantiations

Module instantiations are copies of the logic that define component interconnections in a module.

```
module_name instance_name1 (terminal1, terminal2),
            instance_name2 (terminal1, terminal2);
```

A module instantiation consists of the name of the module (*module_name*), followed by one or more instantiations. An *instantiation* consists of an instantiation name (*instance_name*) and a connection list. A *connection list* is a list of expressions called terminals, separated by commas. These terminals are connected to the ports of the instantiated module.

Terminals connected to input ports can be any arbitrary expression. Terminals connected to output and inout ports can be identifiers, single-bit or multiple-bit slices of an array, or a concatenation of these. The bit widths for a terminal and its module port must be the same.

If you use an undeclared variable as a terminal, the terminal is implicitly declared as a scalar (1-bit) wire. After the variable is implicitly declared as a wire, it can appear wherever a wire is allowed.

Example 3–13 shows the declaration for the module SEQ with two instances (SEQ_1 and SEQ_2).

Example 3-13 Module Instantiations

```
module SEQ(BUS0,BUS1,OUT); // description of module SEQ
  input BUS0, BUS1;
  output OUT;
  ...
endmodule

module top( D0, D1, D2, D3, OUT0, OUT1 );
  input  D0, D1, D2, D3;
  output OUT0, OUT1;

  SEQ SEQ_1(D0,D1,OUT0), // instantiations of module SEQ
      SEQ_2(.OUT(OUT1),.BUS1(D3),.BUS0(D2));
endmodule
```

Named and Positional Notation

Module instantiations can use either named or positional notation to specify the terminal connections.

In name-based module instantiation, you explicitly designate which port is connected to each terminal in the list. Undesignated ports in the module are unconnected.

In position-based module instantiation, you list the terminals and specify connections to the module according to the terminal’s position in the list. The first terminal in the connection list is connected to the first module port, the second terminal to the second module port, and so on. Omitted terminals indicate that the corresponding port on the module is unconnected.

In Example 3-13, SEQ_2 is instantiated with named notation, as follows:

- Signal OUT1 is connected to port OUT of the module SEQ.
- Signal D3 is connected to port BUS1.
- Signal D2 is connected to port BUS0.

SEQ_1 is instantiated by using positional notation, as follows:

- Signal D0 is connected to port BUS0 of module SEQ.
- Signal D1 is connected to port BUS1.
- Signal OUT0 is connected to port OUT.

Building Parameterized Designs

The Verilog language allows you to create parameterized designs by overriding parameter values in a module during instantiation. In Verilog, you can do this with the defparam statement or with the following syntax.

```
module_name #(parameter_value,parameter_value,...)
instance_name
(terminal_list)
```

FPGA Express does not support the defparam statement but does support the syntax above.

The module in Example 3-14 contains a parameter declaration.

Example 3-14 parameter Declaration in a Module

```
module foo (a,b,c);

parameter width = 8;

input [width-1:0] a,b;
output [width-1:0] c;

assign c = a & b;

endmodule
```

In Example 3–14, the default value of the parameter width is 8, unless you override the value when the module is instantiated. When you change the value, you build a different version of your design. This type of design is called a *parameterized design*.

FPGA Express reads parameterized designs as templates. These designs are stored in an intermediate format so that they can be built with different (nondefault) parameter values when they are instantiated.

If your design contains parameters, you can indicate that the design should be read in as a template by adding the pseudo comment //synopsys template to your code.

If you use parameters as constants that never change, do *not* read in your design as a template. One way to build a template into your design is by instantiating it in your Verilog code. Example 3–15 shows how to do this.

Example 3-15 Instantiating a Parameterized Design in your Verilog Code

```
module param (a,b,c);

input [3:0] a,b;
output [3:0] c;

foo #(4) U1(a,b,c); // instantiate foo

endmodule
```

Example 3–15 instantiates the parameterized design, foo, which has one parameter that is assigned the value 4.

Because module foo is defined outside the scope of module param, errors such as port mismatches and invalid parameter assignments are not detected until the design is linked. When FPGA Express links module param, it searches for template foo in memory. If foo is found, it is automatically built with the specified parameters. FPGA Express checks that foo has at least one parameter and three ports, and that the bit widths of the ports in foo match the bit-widths of ports a, b, and c. If template foo is not found, the link fails.

Templates instantiated with different parameter values are different designs and require unique names. Three variables control the naming convention for the templates:

- template_naming_style = “%s_%p”
- template_parameter_style = “%s%d”
- template_separator_style = “_”

The template_naming_style variable is the master variable for renaming a template. The %s field is replaced by the name of the original design, and the %p field is replaced by the names of all the parameters.

The template_parameter_style variable determines how each parameter is named.   The %s field is replaced by the parameter name, and the %d field is replaced by the value of the parameter.

The template_separator_style variable contains a string that separates parameter names. This variable is used only for templates that contain more than one parameter.

When a template is renamed, only the parameters you select when you instantiate the parameterized design are used in the template name. For example, template ADD contains parameters N, M, and Z. You can build a design where N = 8, M = 6, and Z is the default value. The name assigned to this design is ADD_N8_M6. If no parameters are selected, the template is built with default values, and the name of the created design is the same as the name of the template.

Gate-Level Modeling

Verilog provides a number of basic logic gates that enable modeling at the gate level. *Gate-level modeling* is a special case of positional notation for module instantiation that uses a set of predefined module names. FPGA Express supports the following gate types:

- and
- nand
- or
- nor
- xor
- xnor
- buf
- not
- tran

Connection lists for instantiations of a gate-level model use positional notation. In the connection lists for and, nand, or, nor, xor, and xnor gates, the first terminal connects to the output of the gate, and the remaining terminals connect to the inputs of the gate. You can build arbitrarily wide logic gates with as many inputs as you want.

Connection lists for buf, tran, and not gates also use positional notation. You can have as many outputs as you want, followed by only one input. Each terminal in a gate-level instantiation can be a 1-bit expression or signal.

In gate-level modeling, instance names are optional. Drive strengths and delays are allowed, but they are ignored by FPGA Express. Example 3–16 shows two gate-level instantiations.

Example 3-16 Gate-Level Instantiations

```
buf (buf_out,e);
and and4(and_out,a,b,c,d);
```

Note: Delay options for gate primitives are parsed but ignored by FPGA Express. Because FPGA Express ignores the delay information, it might create logic whose behavior does not agree with the simulated behavior of the circuit. See Chapter 6 for more information.

Three-State Buffer Instantiation

FPGA Express supports the following gate types for instantiation of three-state gates:

- bufif0 (active low enable line)
- bufif1 (active high enable line)
- notif0 (active low enable line; output inverted)
- notif1 (active high enable line; output inverted)

Connection lists for bufif and notif gates use positional notation. Specify the order of the terminals as follows:

- The first terminal connects to the output of the gate.
- The second terminal connects to the input of the gate.
- The third terminal connects to the control line.

Example 3–17 shows a three-state gate instantiation with an active high enable and no inverted output.

Example 3-17 Three-State Gate Instantiation

```
module three_state (in1,out1,cntrl1);
input in1,cntrl1;
output out1;

bufif1 (out1,in1,cntrl1);

endmodule
```

Chapter 4
Expressions

In Verilog, *expressions* consist of a single operand or multiple operands separated by operators. Use expressions where a value is required in Verilog.

This chapter explains how to build and use expressions in the following sections:

- Constant-Valued Expressions
- Operators
- Operands
- Expression Bit Widths

Constant-Valued Expressions

A *constant-valued expression* is an expression whose operands are either constants or parameters. FPGA Express determines the value of these expressions.

In Example 4–1, `size-1` is a constant-valued expression. The expression `(op == ADD) ? a+b : a-b` is not a constant-valued expression, because the value depends on the variable `op`. If the value of `op` is `1`, `b` is added to `a`; otherwise, `b` is subtracted from `a`.

Example 4-1 Valid Expressions

```
// all expressions are constant-valued,
// except in the assign statement.
module add_or_subtract( a, b, op, s );
 // performs  s = a+b  if op is ADD
 //           s = a-b  if op is not ADD
parameter size=8;
parameter ADD=1'b1;

 input  op;
 input  [size-1:0] a, b;
 output [size-1:0] s;
 assign s = (op == ADD) ? a+b : a-b; // not a
constant-
// valued expression
endmodule
```

The operators and operands used in an expression influence the way a design is synthesized. FPGA Express evaluates constant-valued expressions and does not synthesize circuitry to compute their value. If an expression contains constants, they are propagated to reduce the amount of circuitry required. FPGA Express does synthesize circuitry for an expression that contains variables, however.

Operators

Operators represent an operation to be performed on one or two operands to produce a new value. Most operators are either unary operators that apply to only one operand, or binary operators that apply to two operands. Two exceptions are conditional operators, which take three operands and concatenation operators, which take any number of operands. The Verilog language operators supported by FPGA Express are listed in Table 4–1. A description of the operators and their order of precedence is given in the following sections.

Table 4-1 Verilog Operators Supported by FPGA Express

Operator	**Description**
{ }	concatenation
+ - * /	arithmetic
%	modulus
> >= < <=	relational
!	logical NOT

Operator	Description
&&	logical AND
| |	logical OR
==	logical equality
! =	logical inequality
~	bit-wise NOT
&	bit-wise AND
|	bit-wise OR
^	bit-wise XOR
^~ ~^	bit-wise XNOR
&	reduction AND
|	reduction OR
~ &	reduction NAND
~ |	reduction NOR
^	reduction XOR
~^ ^~	reduction XNOR
<<	left shift
> >	right shift
? :	conditional

In the following descriptions, the terms *variable* and *variable operand* refer to operands or expressions that are not constant-valued expressions. This group includes wires and registers, bit-selects and part-selects of wires and registers, function calls, and expressions that contain any of these elements.

Arithmetic Operators

Arithmetic operators perform simple arithmetic on operands. The Verilog arithmetic operators are

- addition (+)
- subtraction (-)

- multiplication (*)
- division (/)
- modulus (%)

You can use the addition (+), subtraction (-), and multiplication (*) operators with any operand form (constants or variables). The addition (+) and subtraction (-) operators can be used as either unary or binary operators. FPGA Express requires that division (/) and modulus (%) operators have constant-valued operands.

Example 4-2 shows three forms of the addition operator. The circuitry built for each addition operation is different because of the different operand types. The first addition requires no logic, the second synthesizes an incrementer, and the third synthesizes an adder.

Example 4-2 Addition Operator

```
parameter size=8;
wire [3:0] a,b,c,d,e;

assign c = size + 2; // constant + constant
assign d = a + 1;    // variable + constant
assign e = a + b;    // variable + variable
```

Relational Operators

Relational operators compare two quantities and yield a 0 or 1 value. A true comparison evaluates to 1; a false comparison evaluates to 0. All comparisons assume unsigned quantities. The circuitry synthesized for relational operators is a bit-wise comparator whose size is based on the sizes of the two operands.

The Verilog relational operators are

- less than (<)
- less than or equal to (<=)
- greater than (>)
- greater than or equal to (>=)

Example 4-3 shows the use of a relational operator.

Example 4-3 Relational Operator

```
function [7:0] max( a, b );
input  [7:0] a,b;
   if ( a >= b )  max = a;
   else           max = b;
endfunction
```

Equality Operators

Equality operators generate a `0` if the expressions being compared are not equal and a `1` if the expressions are equal. Equality and inequality comparisons are performed bit-wise.

The Verilog equality operators are

- equality (`==`)
- inequality (`!=`)

Example 4–4 shows the equality operator used to test for a `JMP` instruction. The output signal `jump` is set to `1` if the two high-order bits of `instruction` are equal to the value of parameter `JMP`; otherwise, `jump` is set to `0`.

Example 4-4 Equality Operator

```
module is_jump_instruction ( instruction, jump );
   parameter JMP = 2’h3;

   input  [7:0] instruction;
   output jump;
   assign jump = (instruction[7:6] == JMP);

endmodule
```

Handling Comparisons to *X* or *Z*

Comparisons to an `X` or a `Z` are always ignored. If your code contains a comparison to an `X` or a `Z`, a warning message is displayed indicating that the comparison is always evaluated to false, which might cause simulation to disagree with synthesis.

For example, the variable `B` in the following code (from a file called `test2.v`) is always assigned to the value `1`, because the comparison to `X` is ignored.

Example 4-5 Comparison to X Ignored

```
always begin
if (A == 1’bx)   // this is line 10
B = 0;
else
B = 1;
end
```

When FPGA Express reads this code, the following warning message is generated.

```
Warning:Comparisons to a “don’t care” are treated as
always being false in routine test2 line 10 in file
‘test2.v’. This may cause simulation to disagree with
synthesis. (HDL-170)
```

For an alternate method of handling comparisons to `X` or `Z`, insert the `// synopsys translate_off` directive before the comparison and insert the `// synopsys translate_on` directive after the comparison. Inserting these directives might cause simulation to disagree with synthesis.

Logical Operators

Logical operators generate a `1` or a `0`, according to whether an expression evaluates to `true` (`1`) or `false` (`0`). The Verilog logical operators are

- logical NOT (`!`)
- logical AND (`&&`)
- logical OR (`||`)

The logical `not` operator produces a value of `1` if its operand is zero and a value of `0` if its operand is nonzero. The logical `and` operator produces a value of `1` if both operands are nonzero. The logical `or` operator produces a value of `1` if either operand is nonzero.

Example 4-6 shows some logical operators.

Example 4-6 Logical Operators

```
module is_valid_sub_inst(inst,mode,valid,unimp);

   parameter IMMEDIATE=2’b00, DIRECT=2’b01;
   parameter SUBA_imm=8’h80, SUBA_dir=8’h90,
             SUBB_imm=8’hc0, SUBB_dir=8’hd0;
   input  [7:0] inst;
   input  [1:0] mode;
   output valid, unimp;

   assign valid = (((mode == IMMEDIATE) && (
                 (inst == SUBA_imm) ||
                 (inst == SUBB_imm))) ||
                 ((mode == DIRECT) && (
                     (inst == SUBA_dir) ||
                     (inst == SUBB_dir))));

   assign unimp = !valid;

endmodule
```

Bit-Wise Operators

Bit-wise operators act on the operand bit by bit. The Verilog bit-wise operators are

- unary negation (~)
- binary AND (&)
- binary OR (|)
- binary XOR (^)
- binary XNOR (^~ or ~^)

Example 4-7 shows some bit-wise operators.

Example 4-7 Bit-Wise Operators

```
module full_adder( a, b, cin, s, cout );
  input  a, b, cin;
  output s, cout;

  assign s    = a ^ b ^ cin;
  assign cout = (a&b) | (cin & (a|b));
endmodule
```

Reduction Operators

Reduction operators take one operand and return a single bit. For example, the reduction and operator takes the and value of all the bits of the operand and returns a 1-bit result. The Verilog reduction operators are

- reduction AND (&)
- reduction OR (|)
- reduction NAND (~&)
- reduction NOR (~|)
- reduction XOR (^)
- reduction XNOR (^~ or ~^)

Example 4-8 shows the use of some reduction operators.

Example 4-8 Reduction Operators

```
module check_input ( in, parity, all_ones );
  input  [7:0] in;
  output parity, all_ones;

  assign parity   = ^ in;
  assign all_ones = & in;
endmodule
```

Shift Operators

The Verilog shift operators are

- shift left (<<)
- shift right (>>)

A shift operator takes two operands and shifts the value of the first operand right or left by the number of bits given by the second operand.

After the shift, vacated bits are filled with zeros. Shifting by a constant results in trivial circuitry (because only rewiring is required). Shifting by a variable causes a general shifter to be synthesized. Example 4-9 shows how a right-shift operator is used to perform a division by 4.

Example 4-9 Shift Operator

```
module divide_by_4( dividend, quotient );
  input  [7:0] dividend;
  output [7:0] quotient;

  assign quotient = dividend >> 2; // shift right 2
bits
endmodule
```

Conditional Operators

Conditional operators (`? :`) evaluate an expression and return a value that is based on the truth of the expression. Example 4-10 shows how to use conditional operators. If the expression `(op == ADD)` evaluates to `true`, the value `a+b` is assigned to `result`; otherwise, the value `a-b` is assigned to `result`.

Example 4-10 Conditional Operator

```
module add_or_subtract( a, b, op, result );

  parameter ADD=1'b0;
  input  [7:0] a, b;
  input  op;
  output [7:0] result;

    assign result = (op == ADD) ? a+b : a-b;
endmodule
```

Conditional operators can be nested to produce an `if . . . else if` construct. Example 4-11 shows the conditional operators `? :` used to evaluate the value of `op` successively and perform the correct operation.

Example 4-11 Nested Conditional Operator

```
module arithmetic( a, b, op, result );

  parameter ADD=3'h0,SUB=3'h1,AND=3'h2,
            OR=3'h3, XOR=3'h4;

  input  [7:0] a,b;
  input  [2:0] op;
  output [7:0] result;

  assign result = ((op == ADD) ? a+b : (
                   (op == SUB) ? a-b : (
                   (op == AND) ? a&b : (
                   (op ==  OR) ? a|b : (
                   (op == XOR) ? a^b : (a))))));
endmodule
```

Concatenations

Concatenation combines one or more expressions to form a larger vector. In the Verilog language, you indicate concatenation by listing all expressions to be concatenated, separated by commas, in curly braces (`{}`). Any expression except an unsized constant is allowed in a concatenation. For example, the concatenation `{1’b1,1’b0,1’b0}` yields the value `3’b100`.

You can also use a constant-valued repetition multiplier to repeat the concatenation of an expression. The concatenation `{1’b1,1’b0,1’b0}` can also be written as `{1’b1,{2{1’b0}}}` to yield `3’b100`. The expression `{2{`*`expr`*`}}` within the concatenation repeats *`expr`* two times.

Example 4-12 shows a concatenation that forms the value of a condition-code register.

Example 4-12 Concatenation Operator

```
output [7:0] ccr;
wire  half_carry, interrupt, negative, zero,
               overflow, carry;
...
assign ccr = { 2’b00, half_carry, interrupt,
               negative, zero, overflow, carry };
```

Example 4-13 shows an equivalent description for the concatenation.

Example 4-13 Concatenation Equivalent

```
output [7:0] ccr;
...
assign ccr[7] = 1’b0;
assign ccr[6] = 1’b0;
assign ccr[5] = half_carry;
assign ccr[4] = interrupt;
assign ccr[3] = negative;
assign ccr[2] = zero;
assign ccr[1] = overflow;
assign ccr[0] = carry;
```

Operator Precedence

Table 4-2 lists the precedence of all operators, from highest to lowest. All operators at the same level in the table are evaluated from left to right, except the conditional operator (`?:`), which is evaluated from right to left.

Table 4-2 Operator Precedence

Operator	**Description**
[ ]	bit-select or part-select
( )	parentheses
!, ~	logical and bit-wise negation
&, |, ~&, ~|, ^, ~^, ^~	reduction operators
+, -	unary arithmetic
{ }	concatenation
*, /, %	arithmetic
+, -	arithmetic
<<, >>	shift
>, >= , <, <=	relational
==, !=	logical equality
&	bit-wise AND
^, ^~ , ~^	bit-wise XOR and XNOR
|	bit-wise OR
& &	logical AND
| |	logical OR
? :	conditional

Operands

The following kinds of operands can be used in an expression:

- Numbers
- Wires and registers
- Bit-selects
- Part-selects
- Function calls

Each of these operands is explained in the following subsections.

Numbers

A number is either a constant value or a value specified as a parameter. The expression `size-1` in Example 4-1 illustrates how you can use both a parameter and a constant in an expression.

You can define constants as sized or unsized, in binary, octal, decimal, or hexadecimal bases. The default size of an unsized constant is 32 bits. Refer to Appendix C for a discussion of the format for numbers.

Wires and Registers

Variables that represent both wires and registers are allowed in an expression. (Wires are described in the section "Module Statements and Constructs" in Chapter 3. Registers are described in "Function Declarations" in Chapter 5.) If the variable is a multibit vector, and you use only the name of the variable, the entire vector is used in the expression. Bit-selects and part-selects allow you to select single or multiple bits, respectively, from a vector. These are described in the next two sections.

In the Verilog fragment shown in Example 4-14, `a`, `b`, and `c` are 8-bit vectors of wires. Because only the variable names appear in the expression, the entire vector of each `wire` is used in evaluating the expression.

Example 4-14 Wire Operands

```
wire [7:0] a,b,c;
assign c = a & b;
```

Bit-Selects

A bit-select is the selection of a single bit from a `wire`, `register`, or `parameter` vector. The value of the expression in brackets (`[]`) selects the bit you want from the vector. The selected bit must be within the declared range of the vector. Example 4-15 shows a simple example of a bit-select with an expression.

Example 4-15 Bit-Select Operands

```
wire [7:0] a,b,c;
assign c[0] = a[0] & b[0];
```

Part-Selects

A part-select is the selection of a group of bits from a `wire`, `register`, or `parameter` vector. The part-select expression must be constant-valued in the Verilog language, unlike the bit-select operator. If a variable is declared with ascending indices or descending indices, the part-select (when applied to that variable) must be in the same order.

The expression in Example 4-14 can also be written (with descending indices) as shown in Example 4-16.

Example 4-16 Part-Select Operands

```
assign c[7:0] = a[7:0] & b[7:0]
```

Function Calls

Verilog allows you to call one function from inside an expression and use the return value from the called `function` as an operand. Functions in Verilog return a value consisting of one or more bits. The syntax of a function call is the function name followed by a comma-separated list of function inputs enclosed in parentheses. Example 4-17 shows the function call `legal` used in an expression.

Example 4-17 Function Call Used as an Operand

```
assign error = ! legal(in1, in2);
```

Functions are described in Chapter 5, ‘‘Functional Descriptions.“

Concatenation of Operands

Concatenation is the process of combining several single-bit or multiple-bit operands into one large bit vector. The use of the concatenation operators, a pair of braces (`{}`), is described in the section ‘‘Concatenations“ earlier in this chapter.

Example 4-18 shows two 4-bit vectors (`nibble1` and `nibble2`) that are joined to form an 8-bit vector that is assigned to an 8-bit `wire` vector (`byte`).

Example 4-18 Concatenation of Operands

```
wire [7:0] byte;
wire [3:0] nibble1, nibble2;
assign byte = {nibble1,nibble2};
```

Expression Bit Widths

The bit width of an expression depends on the widths of the operands and the types of operators in the expression.

Table 4-3 shows the bit width for each operand and operator. In the table, *i*, *j*, and *k* are expressions; `L(`*i*`)` is the bit width of expression *i*.

To preserve significant bits within an expression, Verilog fills in zeros for smaller-width operands. The rules for this zero-extension depend on the operand type. These rules are also listed in Table 4-3.

Table 4-3 Expression Bit-Widths

Expression	**Bit Length**	**Comments**
unsized constant	32-bit	self-determined
sized constant	as specified	self-determined
i + j	max(L(*i*),L(*j*))	context-determined
i - j	max(L(*i*),L(*j*))	context-determined
*i * j*	max(L(*i*),L(*j*))	context-determined
i / j	max(L(*i*),L(*j*))	context-determined
i % j	max(L(*i*),L(*j*))	context-determined
i & j	max(L(*i*),L(*j*))	context-determined
i | j	max(L(*i*),L(*j*))	context-determined
i ^ j	max(L(*i*),L(*j*))	context-determined
i ^~ j	max(L(*i*),L(*j*))	context-determined
~*i*	L(*i*)	context-determined
i == j	1-bit	self-determined
i !== j	1-bit	self-determined
i && j	1-bit	self-determined
i || j	1-bit	self-determined
i > j	1-bit	self-determined
i >= j	1-bit	self-determined

Expression	**Bit Length**	**Comments**
i < *j*	1-bit	self-determined
i <= *j*	1-bit	self-determined
&*i*	1-bit	self-determined
|*i*	1-bit	self-determined
^*i*	1-bit	self-determined
~&*i*	1-bit	self-determined
~|*i*	1-bit	self-determined
~^*i*	1-bit	self-determined
i >> *j*	L(*i*)	*j* is self-determined
{i{j}}	i*L(j)	*j* is self-determined
i << *j*	L(*i*)	*j* is self-determined
i ? *j* : *k*	Max(L(*j*),L(*k*))	*j* is self-determined
{*i*,...,*j*}	L(*i*)+...+L(*j*)	self-determined
{*i* {*j*,...,*k*}}	/*(L(*j*)+...+L(k))	self-determined

Verilog classifies expressions (and operands) as either self-determined or context-determined. A *self-determined* expression is one in which the width of the operands is determined solely by the expression itself. These operand widths are never extended.

Example 4-19 shows a self-determined expression that is a concatenation of variables with known widths.

Example 4-19 Self-Determined Expression

```
output [7:0] result;
wire   [3:0] temp;

assign temp = 4’b1111;
assign result = {temp,temp};
```

The concatenation has two operands. Each operand has a width of four bits and a value of `4’b1111` . The resulting width of the concatenation is eight bits, which is the sum of the width of the operands. The value of the concatenation is `8’b11111111` .

A *context-determined* expression is one in which the width of the expression depends on all operand widths in the expression. For example, Verilog defines the resulting width of an addition as the greater of the widths of its two operands. The addition of two 8-bit quantities produces an 8-bit value; however, if the result of the addition is assigned to a 9-bit quantity, the addition produces a 9-bit result. Because the addition operands are context-determined, they are zero-extended to the width of the largest quantity in the entire expression.

Example 4-20 shows context-determined expressions.

Example 4-20 Context-Determined Expressions

```
if ( ((1’b1 << 15) >> 15) == 1’b0 )
  // This expression is ALWAYS true.

if ( (((1’b1 << 15) >> 15) | 20’b0) == 1’b0 )
  // This expression is NEVER true.
```

The expression `((1’b1 << 15) >> 15)` produces a one-bit `0` value `(1’b0)`. The `1` is shifted off the left end of the vector, producing a value of `0`. The right shift has no additional effect. For a shift operator, the first operand (`1’b1`) is context-dependent; the second operand (`15`) is self-determined.

The expression `(((1’b1 << 15) >> 15) | 20’b0)` produces a 20-bit `1` value `(20’b1)`. `20’b1` has a `1` in the least significant bit position and `0`s in the other 19 bit positions. Because the largest operand within the expression has a width of 20, the first operand of the shift is zero-extended to a 20-bit value. The left shift of 15 does not drop the `1` value off the left end; the right shift brings the `1` value back to the right end, resulting in a 20-bit `1` value `(20’b1)`.

Chapter 5
Functional Descriptions

A Verilog circuit description can be one of two types: a structural description or a functional description, also referred to as a Register Transfer Level (RTL) description. A *structural* description explains the exact physical makeup of the circuit, detailing components and the connections between them. A *functional* or RTL description describes a circuit in terms of its registers and the combinational logic between the registers.

This chapter describes the construction and use of functional descriptions in the following sections:

- Using Sequential Constructs
- *function* Declarations
- Function Statements
- *task* Statements
- *always* Blocks

Using Sequential Constructs

Although many Verilog constructs appear sequential in nature, they describe combinational circuitry. A simple description that appears to be sequential is shown in Example 5-1.

Example 5-1 Sequential Statements

```
x = b;
if (y)
x = x + a;
```

FPGA Express determines the combinational equivalent of this description. In fact, FPGA Express treats the statements in Example 5-1 the same way it treats the statements in Example 5-2.

Example 5-2 Equivalent Combinational Description

```
if (y)
x = b + a;
else
x = b;
```

To describe combinational logic, you write a sequence of statements and operators to generate the output values you want. For example, suppose the + operator is not supported, and you want to create a combinational, ripple-carry adder. The easiest way to describe this circuit is as a cascade of full adders, as in Example 5-3. The example has eight full adders, with each adder following the one before. From this description, FPGA Express generates a fully combinational adder.

Example 5-3 Combinational Ripple-Carry Adder

```
function [7:0] adder;
input [7:0] a, b;
    reg c;
    integer i;
    begin
        c = 0;
        for (i = 0; i <= 7; i = i + 1) begin
            adder[i] = a[i] ^ b[i] ^ c;
            c = a[i] & b[i] | a[i] & c | b[i] & c;
        end
    end
endfunction
```

function Declarations

Verilog function declarations are one of the two primary methods for describing combinational logic. The other method is the `always` block, described later in this chapter. You must declare and use Verilog functions within a module. You can call functions from the structural part of a Verilog description by using them in a continuous assignment statement or as a terminal in a module instantiation. You can also call functions from other functions or from `always` blocks.

FPGA Express supports the following Verilog function declarations:

- `input` declarations
- `reg` declarations
- memory declarations
- `parameter` declarations
- `integer` declarations

Functions begin with the keyword `function` and end with the keyword `endfunction`. The width of the function's return value (if any) and the name of the function follow the `function` keyword, as shown in the syntax below.

```
function [ range] name_of_function ;
            [ func_declaration]*
             statement_or_null
endfunction
```

Defining the bit ***range*** of the return value is optional. Specify ***range*** inside square brackets ([ ]). If you do not define ***range***, a 1-bit quantity is returned by default. The function's output is set by assigning it to the function name. A function can contain one or more statements. If you use multiple statements, enclose the statements between a `begin...end` pair.

A simple `function` declaration is shown in Example 5-4.

Example 5-4 Simple Function Declaration

```
function [7:0] scramble;
input [7:0] a;
input [2:0] control;
integer i;
    begin
        for (i = 0; i <= 7; i = i + 1)
            scramble[i] = a[ i ^ control ];
    end
endfunction
```

Function statements supported by FPGA Express are discussed under "Function Statements" later in this chapter.

input Declarations

Verilog input declarations specify the input signals for a function.

You must declare the inputs to a Verilog function immediately after you declare the function name. The syntax of `input` declarations for a function is the same as the syntax of `input` declarations for a module, as shown below.

```
input [ range] list_of_variables ;
```

The optional `range` specification declares an input as a vector of signals. Specify `range` inside square brackets ([ ]).

Note: The order in which you declare the inputs must match the order of the inputs in the function call.

Function Output

The output from a function is assigned to the function name. A Verilog function has only one output, which can be a vector. For multiple outputs from a function, use the concatenation operation to bundle several values into one return value. This single return value can then be unbundled by the caller. Example 5-5 shows how unbundling is done.

Example 5-5 Many Outputs from a Function

```
function [9:0] signed_add;
input [7:0] a, b;
    reg [7:0] sum;
    reg carry, overflow;

    begin
        ...
        signed_add = {carry, overflow, sum};
    end
endfunction
...
assign {C, V, result_bus} = signed_add(busA, busB);
```

The `signed_add` function bundles the values of `carry`, `overflow`, and `sum` into one value. This new value is returned in the `assign` statement following the function. The original values are then unbundled by the function that called the `signed_add` function.

reg Declarations

A register represents a variable in Verilog. The syntax for a register declaration is

```
reg [ range] list_of_register_variables ;
```

A reg declaration can be a single-bit quantity or a vector of bits. The *range* parameter specifies the most significant bit (msb) and least significant bit (lsb) of the vector. Both must be nonnegative constants, parameters, or constant-valued expressions, and are enclosed in square brackets ([ ]). Example 5-6 shows some reg declarations.

Example 5-6 Register Declarations

```
reg x;              /* single bit */
reg a, b, c;        /* 3 single-bit quantities */
reg [7:0] q;        /* an 8-bit vector */
```

The Verilog language allows you to assign a value to a reg variable only within a function or an always block.

In the Verilog simulator, reg variables can hold state information. A reg variable can hold its value across separate calls to a function. In some cases, FPGA Express emulates this behavior by inserting flow-through latches. In other cases, this behavior is emulated without a latch. The concept of holding state is elaborated in Chapter 6 and in several examples in Appendix A.

Memory Declarations

The memory construct models a bank of registers or memory. In Verilog, the memory construct is actually a two-dimensional array of reg variables. Sample memory declarations are shown in Example 5-7.

Example 5-7 Memory Declarations

```
reg [7:0] byte_reg;
reg [7:0] mem_block [255:0];
```

In Example 5-7, byte_reg is an 8-bit register and mem_block is an array of 256 registers, each of which is eight bits wide. You can index the array of registers to access individual registers, but you cannot access individual bits of a register directly. Instead, you must copy the appropriate register into a temporary one-dimensional register. For example, to access the fourth bit of the eighth register in `mem_block`, enter

```
byte_reg = mem_block [7];
individual_bit = byte_reg [3];
```

parameter Declarations

Parameter variables are local or global variables that hold values. The syntax for a `parameter` declaration is

```
parameter [ range] identifier = expression,
identifier = expression;
```

The range specification is optional.

You can declare parameter variables as local to a function. However, you cannot use a local variable outside of that function. Parameter declarations in a function are identical to parameter declarations in a module. (See Chapter 3 for more information.) The function in Example 5-8 contains a `parameter` declaration.

Example 5-8 Parameter Declaration in a Function

```
function gte;
parameter width = 8;
input [width-1:0] a,b;
gte = (a >= b);
endfunction
```

integer Declarations

Integer variables are local or global variables that hold numeric values. The syntax for an `integer` declaration is

```
integer  identifier_list;
```

You can declare `integer` variables locally at the function level or globally at the module level. The default size for `integer` variables is 32 bits. FPGA Express determines bit widths, except in the case of a dont-care resulting from a compile.

Example 5-9 illustrates `integer` declarations.

Example 5-9 Integer Declarations

```
integer a;       /* single 32 bit integer */
integer b, c;    /* two integers */
```

Function Statements

The function statements supported by FPGA Express are

- Procedural assignments
- RTL assignments
- `begin . . . end` block statements
- `if. . . else` statements
- `case`, `casex`, and `casez` statements
- `for` loops
- `while` loops
- `forever` loops
- `disable` statements

Procedural Assignments

Procedural assignments are assignment statements used inside a function. They are similar to the continuous assignment statements described in Chapter 3, "Module Statements and Constructs", except that the left side of a procedural assignment can contain only `reg` variables and integers. Assignment statements set the value of the left side to the current value of the right side. The right side of the assignment can contain any arbitrary expression of the data types described in Chapter 3, including simple constants and variables.

The left side of the procedural assignment statement can contain only the following data types:

- `reg` variables
- Bit-selects of `reg` variables
- Part-selects of `reg` variables
- Integers
- Concatenations of the above

The expressions in the part-select of a left side must be constant-valued.

Assignments are made bit-wise, with the low bit on the right side assigned to the low bit on the left side. If the number of bits on the right side is greater than the number on the left side, the high-order bits on the right side are discarded. If the number of bits on the left side is greater than the number on the right side, the right side bits are zero-extended. Multiple procedural assignments are allowed.

Some examples of procedural assignments are shown in Example 5-10.

Example 5-10 Procedural Assignments

```
sum = a + b;
control[5] = (instruction == 8'h2e);
{carry_in, a[7:0]} = 9'h 120;
```

RTL Assignments

Procedural assignments in Verilog can be blocking in nature. For example, you can assign a delay of five time units with the following statement.

```
rega = #5 arg1 + arg2;
```

The expression, arg1 + arg2 is evaluated, then execution is suspended for five time units before the assignment is performed and the next statement is processed. Execution of the next statement is blocked until the current statement's execution is completed.

On the other hand, RTL assignments let you define nonblocking procedural assignments with timing controls. If you use a nonblocking RTL assignment statement instead of the procedural assignment, the sum is computed immediately, but the assignment is done after the five time-unit delay.

```
rega <= #5 arg1 + arg2;
```

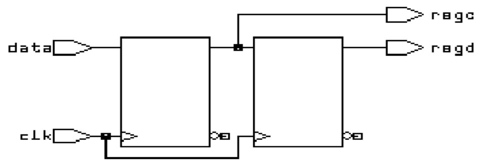
However, execution proceeds without waiting for the assignment to finish. FPGA Express ignores intra-assignment and interassignment delays; therefore, the RTL assignment behaves like the blocking procedural assignment in this case.

To illustrate the difference in behavior between RTL assignments and blocking procedural assignments, consider Example 5-11 and Example 5-12, where there are multiple assignments.

Example 5-11 RTL Assignments

```
always @(posedge clk) begin
regc <= data;
regd <= regc;
end
```

Figure 5-1 Schematic of RTL Assignments

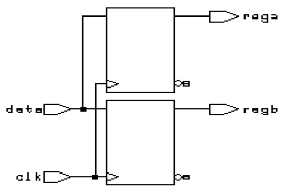


Example 5-11 is a description of a serial register implemented with RTL assignments. The recently assigned value of `regc`, which is data, is assigned to `regd` as the schematic indicates. If blocking assignments are used, as in Figure 5-2, a serial register is not synthesized, because assignments are executed before proceeding.

Example 5-12 Blocking Assignment

```
always @(posedge clk) begin
 rega = data;
 regb = rega;
end
```

Figure 5-2 Schematic of Blocking Assignment



The following restrictions apply to RTL assignments:

- •You cannot use procedural assignments with blocking delays and RTL assignments at the same time. The following example is not allowed.

```
reg b,c;

always begin
b <= #4a; // RTL assignment
c = #3b; // procedure assignment with
// blocking delay
end
```

- Because FPGA Express ignores delay information, synthesis might not agree with simulation.

- If you first assign a value to a `reg` variable with a procedural assignment, you cannot use an RTL assignment on that `reg` anywhere in the module.
- If you first assign a value to a `reg` variable with an RTL assignment, you cannot use a procedural assignment on that `reg` anywhere in the module.

begin . . . end Block Statements

Block statements are a way of syntactically grouping several statements into a single statement.

In Verilog, sequential blocks are delimited by the keywords `begin` and `end`. These `begin...end` blocks are commonly used in conjunction with `if`, `case`, and `for` statements to group several statements together. Functions and `always` blocks that contain more than one statement require a `begin...end` block to group the statements. Verilog also provides a construct called a *named block*, as shown in Example 5-13.

Example 5-13 Block Statement with a Named Block

```
begin : block_name
   reg local_variable_1;
integer local_variable_2;
parameter local_variable_3;
    ... statements ...
end
```

In Verilog, no semicolon (`;`) follows the `begin` or `end` keywords. You identify named blocks by following the `begin` keyword with a colon (`:`) and a ***`block_name`***, as shown. Verilog syntax allows you to declare variables locally in a named block. You can include `reg`, `integer` , and `parameter` declarations within a named block, but not in an unnamed block. Named blocks allow you to use the `disable` statement.

if . . . else Statements

`if...else` statements execute a block of statements according to the value of one or more expressions.

The syntax of an `if...else` statement is

```
if ( expr )
     begin
      ... statements ...
     end
else
     begin
      ... statements ...
     end
```

The `if` statement consists of the keyword `if`, followed by an expression enclosed in parentheses. This expression is followed by a statement or block of statements enclosed with the `begin` and `end` keywords. If the value of the expression is nonzero, it is `true`, and the statement block that follows is executed. If the value of the expression is zero, it is `false`, and the statement block that follows is *not* executed.

An optional `else` statement can follow an `if` statement. If the expression following the `if` keyword is `false`, the statement or block of statements following the `else` keyword is executed.

The `if...else` statement can cause registers to be synthesized. Registers are synthesized when you do not assign a value to the same `reg` variable in all branches of a conditional construct. Information on registers is provided in Chapter 6.

FPGA Express synthesizes multiplexer logic (or similar select logic) from a single `if` statement. The conditional expression in an `if` statement is synthesized as a control signal to a multiplexer, which determines the appropriate path through the multiplexer. For example, the statements in Example 5-14 create multiplexer logic controlled by `c` and places either `a` or `b` in the variable `x`.

Example 5-14 if Statement that Synthesizes Multiplexer Logic

```
if (c)
x = a;
else
x = b;
```

Example 5-15 illustrates how `if` and `else` can be used to create an arbitrarily long `if...else if...else` structure.

Example 5-15 if . . . else if . . . else Structure

```
if (instruction == ADD)
    begin
        carry_in = 0;
        complement_arg = 0;
    end
else if (instruction == SUB)
    begin
        carry_in = 1;
        complement_arg = 1;
    end
else
    illegal_instruction = 1;
```

Example 5-16 shows how to use nested `if` and `else` statements.

Example 5-16 Nested if and else Statements

```
if (select[1])
    begin
        if (select[0])  out = in[3];
        else out = in[2];
    end
else
    begin
        if (select[0])  out = in[1];
        else  out = in[0];
    end
```

Conditional Assignments

FPGA Express can synthesize a latch for a conditionally assigned variable. If a path exists that does not explicitly assign a value to a variable, the variable is conditionally assigned. See the section on “Latch Inference” in Chapter 6 for more information.

In Example 5-17, the variable `value` is conditionally driven. If `c` is not `true`, `value` is not assigned and retains its previous value.

Example 5-17 Synthesizing a Latch for a Conditionally Driven Variable

```
always begin
if ( c ) begin
value = x;
end
Y = value; //causes a latch to be synthesized for
value
end
```

case Statements

The `case` statement is similar in function to the `if...else...` conditional statement. The `case` statement allows a multipath branch in logic that is based on the value of an expression. One way to describe a multicycle circuit is with a `case` statement (see Example 5-18). Another way is with multiple `@` (clock-edge) statements, which are discussed later in this section.

The syntax for a `case` statement is shown below.

```
case ( expr )
     case_item1 : begin

     ... statements ...

     end
     case_item2 : begin

     ... statements ...

     end
     default : begin

     ... statements ...
     end
endcase
```

The `case` statement consists of the keyword `case`, followed by an expression in parentheses, followed by one or more case-items (and associated statements to be executed), followed by the keyword `endcase`. A *case-item* consists of an expression (usually a simple constant) or a list of expressions separated by commas, followed by a colon (`:`).

The expression following the `case` keyword is compared against each case-item expression, one by one. When the expressions are equal, the condition evaluates to `true`. Multiple expressions separated by commas can be used in each case-item. When multiple expressions are used, the condition is said to be `true` if any of the expressions in the case-item match the expression following the `case` keyword.

The first case-item that evaluates to `true` determines the path. All subsequent case-items are ignored, even if they are `true`. If no case-item is `true`, no action is taken. You can define a default case-item with the expression `default`, which is used when no other case-item is `true`.

An example of a `case` statement is shown in Example 5-18.

Example 5-18 case Statement

```
case (state)
    IDLE: begin
        if (start)
            next_state = STEP1;
        else
            next_state = IDLE;
    end
    STEP1: begin
        /* do first state processing here */
        next_state = STEP2;
    end
    STEP2: begin
        /* do second state processing here */
        next_state = IDLE;
    end
endcase
```

Full Case and Parallel Case

FPGA Express automatically determines whether a `case` statement is full or parallel. A `case` statement is referred to as*full case* if all possible branches are specified. If you do not specify all possible branches, but you know that one or more branches can never occur, you can declare a `case` statement as full case with the `// synopsys full_case` directive. Otherwise, FPGA Express synthesizes a latch. See " *full_case* Directive" in Chapter 9 for more information.

FPGA Express synthesizes optimal logic for the control signals of a `case` statement. If FPGA Express cannot statically determine that branches are parallel, it synthesizes hardware that includes a priority encoder. If FPGA Express can determine that no cases overlap (*parallel case*), a multiplexer is synthesized, because a priority encoder is not necessary. You can also declare a `case` statement as parallel case with the `//synopsys parallel_case` directive. Refer to the section "*parallel_case* Directive" in Chapter 9.

Example 5-19 does not result in either a latch or a priority encoder.

Example 5-19 A case Statement that is Both Full and Parallel

```
input [1:0] a;
always @(a or w or x or y or z) begin
case (a)
2’b11:
    b = w ;
2’b10:
    b = x ;
2’b01:
    b = y ;
2’b00:
    b = z ;
endcase
end
```

Example 5-20 shows a case statement that is missing branches for the cases 2’b01 and 2’b10. Example 5-20 infers a latch for b.

Example 5-20 A case Statement that is Parallel but Not Full

```
input [1:0] a;
always @(a or w or z) begin
case (a)
2’b11:
    b = w ;
2’00:
    b = z ;
endcase
end
```

The `case` statement in Example 5-21 is not parallel or full because the input values of `w` and `x` cannot be determined. However, if you know that only one of the inputs equals `2’b11` at a given time, you can use the `// synopsys parallel_case` directive to avoid synthesizing a priority encoder. If you know that either `w` or `x` always equals `2’b11` (a situation known as a one-branch tree), you can use the `// synopsys full_case` directive to avoid synthesizing a latch.

Example 5-21 A case Statement that is Not Full or Parallel

```
always @( w or x) begin
case (2’b11)
w:
b = 10 ;
x:
    b = 01 ;
endcase
end
```

casex Statements

The `casex` statement is a type of `case` statement. The `casex` statement allows a multipath branch in logic according to the value of an expression, just like the `case` statement. The differences between the `case` statement and the `casex` statement are the keyword and the processing of the expressions.

The syntax for a `casex` statement is shown below.

```
casex ( expr )
     case_item1 : begin
     ... statements ...
     end
     case_item2 : begin
     ... statements ...
     end
     default : begin
     ... statements ...
     end
endcase
```

A *case-item* can have expressions consisting of

- A simple constant
- A list of identifiers or expressions separated by commas, followed by a colon (`:`)
- Concatenated, bit-selected, or part-selected expressions
- A constant containing `z`, `x`, or `?`

When a `z`, `x`, or `?` appears in a case-item expression, it means that the corresponding bit of the `casex` expression is not compared. For example:

Example 5-22 casex Statement with x

```
reg [3:0] cond;
casex (cond)
    4'b100x: out = 1;
    default: out = 0;
endcase
```

In Example 5-22, `out` is set to `1` if `cond` is equal to `4'b1000` or `4'b1001`, because the last bit of `cond` is defined as `x`.

Example 5-23 shows a complicated section of code that can be simplified with a `casex` statement that uses the `?` value.

Example 5-23 Before Using casex with ?

```
if (cond[3]) out = 0;
else if (!cond[3] & cond[2] ) out = 1;
else if (!cond[3] & !cond[2] & cond[1] ) out = 2;
else if (!cond[3] & !cond[2] & !cond[1] & cond[0] )
out = 3;
else if (!cond[3] & !cond[2] & !cond[1] & !cond[0] )
out = 4;
```

Example 5-24 shows the simplified version of the same code.

Example 5-24 After Using casex with ?

```
casex (cond)
 4’b1???: out = 0;
  4’b01??: out = 1;
  4’b001?: out = 2;
  4’b0001: out = 3;
4’b0000: out = 4;
endcase
```

`?`, `z`, and `x` bits are allowed in case-item expressions, but are not allowed in `casex` expressions. Example 5-25 shows comparison in an illegal expression.

Example 5-25 Illegal casex Expression

```
express = 3’bxz?;
    ...
casex (express) /* illegal testing of an expression
*/
    ...
endcase
```

casez Statements

The `casez` statement is a type of `case` statement. The `casez` statement allows a multipath branch in logic according to the value of an expression, just like the `case` statement. The differences between the `case` statement and the `casez` statement are the keyword and the way the expressions are processed. The `casez` statement acts exactly the same as the `casex` statement, except that `x` is not allowed in case-item expressions. Only `z` and `?` are accepted as special characters.

The syntax for a `casez` statement is shown below.

```
casez ( expr )
      case_item1 : begin
      ... statements ...
      end
      case_item2 : begin
      ... statements ...
      end
default : begin
      ... statements ...
      end
endcase
```

A case-item can have expressions consisting of

- A simple constant
- A list of identifiers or expressions separated by commas, followed by a colon (`:`)
- Concatenated, bit-selected, or part-selected expressions
- A constant containing a `z` or `?`
- When a `casez` statement is evaluated, the value `z` in the case-item expression is ignored. An example of a `casez` statement with `z` in the case-item is shown in Example 5-26.

Example 5-26 casez Statement with z

```
casez (what_is_it)
  2’bz0: begin
     /* accept anything with least significant bit
zero */
     it_is = even;
  end
  2’bz1: begin
     /* accept anything with least significant bit
one */
     it_is = odd;
  end
endcase
```

`?` and `z` bits are allowed in case-items, but are not allowed in `casez` expressions. Example 5-27 shows an illegal expression in a `casez` statement.

Example 5-27 Illegal casez Expression

```
express = 1’bz;
    ...
casez (express) /* illegal testing of an expression
*/
    ...
endcase
```

for Loops

The `for` loop repeatedly executes a single statement or block of statements. The repetitions are performed over a range determined by the range expressions assigned to an index. Two range expressions are used in each `for` loop: `low_range` and `high_range` . Note that in the syntax lines that follow, `high_range` is greater than or equal to `low_range`. FPGA Express recognizes both incrementing and decrementing loops. The statement to be duplicated is surrounded by `begin` and `end` statements.

Note: FPGA Express allows four syntax forms for a `for` loop. They are

```
for (index= low_range;index < high_range;index= index
+ step)
for (index= high_range;index > low_range;index= index
- step)
for (index= low_range;index <= high_range;index=
index + step)
for (index= high_range;index >= low_range;index=
index - step)
```

Example 5-28 shows a simple `for` loop.

Example 5-28 A Simple for Loop

```
for (i = 0; i <= 31; i = i + 1) begin
    s[i] = a[i] ^ b[i] ^ carry;
    carry = a[i] & b[i]  |  a[i] & carry  |
                            b[i] & carry;
end
```

Note that `for` loops can be nested, as shown in Example 5-29.

Example 5-29 Nested for Loops

```
for (i = 6; i >= 0; i = i - 1)
    for (j = 0; j <= i; j = j + 1)
        if (value[j] > value[j+1]) begin
            temp = value[j+1];
            value[j+1] = value[j];
            value[j] = temp;
        end
```

You can use for loops as duplicating statements. Example 5-30 shows a for loop that is expanded into its longhand equivalent in Example 5-31.

Example 5-30 Example for Loop

```
for ( i=0; i < 8; i=i+1 )
    example[i] = a[i] & b[7-i];
```

Example 5-31 Expanded for Loop

```
example[0] = a[0] & b[7];
example[1] = a[1] & b[6];
example[2] = a[2] & b[5];
example[3] = a[3] & b[4];
example[4] = a[4] & b[3];
example[5] = a[5] & b[2];
example[6] = a[6] & b[1];
example[7] = a[7] & b[0];
```

while Loops

The `while` loop executes a statement until the controlling expression evaluates to `false`. A `while` loop creates a conditional branch that must be broken by one of the following statements to prevent combinational feedback

`@ (posedge clock)` or `@ (negedge clock)`

FPGA Express supports `while` loops, if you insert one of the following expressions in every path through the loop

`@ (posedge clock)` or `@ (negedge clock)`

Example 5-32 shows an unsupported `while` loop that has no *`event-expression`*.

Example 5-32 Unsupported while Loop

```
always
while (x < y)
x = x + z;
```

If you add @ (posedge clock) expressions after the while loop in Example 5-32, you get the supported version shown in Example 5-33.

Example 5-33 Supported while Loop

```
always
begin @ (posedge clock)
while (x < y)
begin
@ (posedge clock);
x = x + z;
end
end;
```

forever Loops

Infinite loops in Verilog use the keyword `forever`. You must break up an infinite loop with an `@ (posedge clock)` or `@ (negedge clock)` expression to prevent combinational feedback, as shown in Example 5-34.

Example 5-34 Supported forever Loop

```
always
forever
begin
@ (posedge clock);
x = x + z;
end
```

You can use `forever` loops with a `disable` statement to implement synchronous resets for flip-flops. The disable statement is described in the next section. See Chapter 6, “Register and Three-State Inference,” for more information on synchronous resets.

The style illustrated in Example 5-34 is not recommended because it is not testable. The synthesized state machine does not reset to a known state. Therefore, it is impossible to create a test program for the state machine. Example 5-36 illustrates how a synchronous reset for the state machine can be synthesized.

disable Statements

FPGA Express supports the `disable` statement when you use it in named blocks. When a `disable` statement is executed, it causes the named block to terminate. A comparator description that uses `disable` is shown in Example 5-35.

Example 5-35 Comparator Using disable

```
begin : compare
for (i = 7; i >= 0; i = i - 1) begin
    if (a[i] != b[i]) begin
         greater_than = a[i];
         less_than = ~a[i];
         equal_to = 0;
         /* comparison is done so stop looping */
         disable compare;
      end
end

/* If we get here a == b
If the disable statement is executed, the next three
       lines will not be executed */
   greater_than = 0;
   less_than = 0;
   equal_to = 1;
end
```

Note that Example 5-35 describes a combinational comparator. Although the description appears sequential, the generated logic runs in a single clock cycle.

You can also use a `disable` statement to implement a synchronous reset, as shown in Example 5-36.

Example 5-36 Synchronous Reset of State Register Using disable in a forever Loop

```
always
forever
begin: reset_label
@ (posedge clock);
if (reset) disable reset_label;
z = a;

@ (posedge clock);
if (reset) disable reset_label;
z = b;
end
```

The `disable` statement in Example 5-36 causes the block `reset_label` to immediately terminate and return to the beginning of the block. Therefore, the first state in the loop is synthesized as the reset state.

task Statements

The `task` statements are similar to functions in Verilog, except they can have `output` and `inout` ports. You can use the `task` statement to structure your Verilog code so that a portion of code can be reused.

In Verilog, `task` statements can have timing controls, and they can take a nonzero time to return. However, FPGA Express ignores all timing controls, so synthesis might disagree with simulation if the timing controls are critical to the function of the circuit.

Example 5–37 shows how a `task` construct is used to define an adder function.

Example 5-37 Using the task Statement

```
module task_example (a,b,c);
input [7:0] a,b;
output [7:0] c;
reg [7:0] c;

task adder;
input [7:0] a,b;
output [7:0] adder;
reg c;
integer i;

begin
c = 0;
for (i = 0; i <= 7; i = i+1) begin
adder[i] = a[i] ^ b[i] ^ c;
c = (a[i] & b[i]) | (a[i] & c) | (b[i] & c);
end
end
endtask
always
adder (a,b,c); // c is a reg

endmodule
```

Note: Only `reg` variables can receive output values from a `task`; `wire` variables cannot.

always Blocks

An `always` block can imply latches or flip-flops, or it can specify purely combinational logic. An `always` block can contain logic triggered in response to a change in a level or the rising or falling edge of a signal. The syntax of an `always` block is

```
always @ (  event-expression [or event-expression*] )
begin
    ...  statements ...
end
```

The *`event-expression`* declares the triggers, or timing controls. The word `or` groups several triggers together. The Verilog language specifies that if triggers in the *`event-expression`* occur, the block is executed. Only one trigger in a group of triggers needs to occur for the block to be executed. However, FPGA Express ignores the *`event-expression`* unless it is a *synchronous* trigger that infers a register. Refer to Chapter 6 for details.

A simple example of an `always` block with triggers is

Example 5-38 A Simple always Block

```
always @ ( a or b or c ) begin
    f = a & b & c
end
```

In Example 5-38, `a`, `b`, and `c` are asynchronous triggers. If any triggers change, the simulator resimulates the `always` block and recalculates the value of `f`. FPGA Express ignores the triggers in this example because they are not synchronous. However, you must indicate all variables that are read in the `always` block as triggers. If you do not indicate all the variables as triggers, FPGA Express gives a warning message similar to the following.

```
Warning: Variable ‘foo’ is being read in block ‘bar’
declared on line 88 but does not occur in the
timing control of the block.
```

For a *synchronous* always block, FPGA Express does not require all variables to be listed.

An `always` block is triggered by any of the following types of *`event-expressions`*:

- The change in a specified value. For example:

```
always @ (  identifier ) begin
   ...   statements ...
end
```

  In the example above, FPGA Express ignores the trigger.

- The rising edge of a clock. For example:

```
always @ ( posedge   event ) begin
   ...  statements ...
end
```

- The falling edge of a clock. For example:

```
always @ ( negedge   event ) begin
   ...  statements ...
end
```

- A clock or an asynchronous preload condition. For example:

```
always @ ( posedge   CLOCK or negedge  reset ) begin
   if ! reset begin
    ...  statements ...
   end
   else begin
    ...  statements ...
   end
end
```

- An asynchronous preload that is based on two events joined by the word `or`. For example:

```
always @ ( posedge   CLOCK or posedge  event1 or
           negedge   event2 ) begin
    if (  event1 ) begin
     ...  statements ...
    end
    else if ( ! event2 ) begin
     ...  statements ...
    end
    else begin
     ...  statements ...
    end
end
```

When the *event-expression* does not contain posedge or negedge, combinational logic (no registers) is usually generated, although flow-through latches can be generated. Refer to the section “Latch Inference” in Chapter 6.

Note: The statements `@ (posedge clock)` and `@ (negedge clock)` are not supported in functions or tasks.

Incomplete Event Specification

An `always` block can be misinterpreted if you do not list all signals entering an `always` block in the event specification.

As expected, FPGA Express builds a 3-input AND gate for the description in Example 5-39.

Example 5-39 Incomplete Event List

```
always @(a or b) begin
   f = a & b & c;
end
```

When this description is simulated, `f` is not recalculated when `c` changes, because `c` is not listed in the ***`event-expression`***. The simulated behavior is *not* that of a 3-input AND gate.

The simulated behavior of the description in Example 5-40 is correct because it includes all signals in ***`event-expression`***.

Example 5-40 Complete Event List

```
always @(a or b or c) begin
    f = a & b & c;
end
```

In some cases, you cannot list all signals in the event specification. Example 5-41 illustrates this problem.

Example 5-41 Incomplete Event List for Asynchronous Preload Condition

```
always @ (posedge c or posedge p)
if (p)
z = d;
else
z = a;
```

In the logic synthesized for Example 5-41, if data (`d`) changes while `p` is high, the change is reflected immediately in the output (`z`). However, when this description is simulated, `z` is not recalculated when `d` changes because `d` is not listed in the event specification. As a result, synthesis might not match simulation.

Asynchronous preloads can be correctly modeled only when you want changes in the load data to be immediately reflected in the output. In Example 5-41, data `d` must change to the preload value before preload condition `p` transits from low to high. If you attempt to read a value in an asynchronous preload, FPGA Express prints a warning similar to the one shown below.

```
Warning: Variable ‘d’ is being read asynchronously in
          routine reset line 21 in file
          ‘/usr/tests/hdl/asyn.v’. This    might  cause
          simulation-synthesis mismatches.
```

Chapter 6
Register and Three-State Inference

FPGA Express can infer latches and flip-flops. A *register* is a simple, one-bit memory device, either a flip-flop or a latch. A *flip-flop* is an edge-triggered memory device. A *latch* is a level-sensitive memory device. Register inference allows you to use sequential logic in your design descriptions and keep your designs technology independent.

This chapter discusses different types of register and three-state inference in the following sections:

- Latch Inference
- Simple Flip-Flop Inference
- Flip-Flop Inference with Asynchronous Reset
- Additional Types of Register Inference
- FPGA Express Latch and Flip-Flop Inference
- Delays in Registers
- Efficient Use of Registers
- Three-State Inference
- Registered and Latched Three-State Enables

Latch Inference

Because variables can hold state over time in simulation, FPGA Express needs to duplicate this condition in hardware. It does this by inserting a D-type flow-through latch. The latch allows a variable to hold its value (state) until that value is reassigned.

A variable must hold its state when its previous value might change because of a condition in an `if` statement. When the condition is `true`, the value is reassigned. Because the condition might be `false`, the variable must be able to hold its state. Therefore, a latch is created to hold the previous value of the variable. For example:

Example 6-1 Creating a Latch

```
always @ ( PHI_1 or A ) begin
   if ( PHI_1 ) begin
      Y = A;
   end
end
```

In Example 6-1, the variable `Y` is not assigned a new value when `PHI_1` is `false`. A latch is synthesized with its D input connected to `A`, its `Q` output connected to `Y`, and its gate controlled by `PHI_1`.

A latch can also be created when you use a `case` statement. For example, the code in Example 6-2 creates a latched binary-coded decimal (BCD) decoder.

Example 6-2 Creating a Latch with a case Statement

```
module decoder(I,decimal);
input [3:0] I;
output [9:0] decimal;
reg  [9:0] decimal;

always @(I) begin
case(I)
4'h0: decimal= 10'b0000000001;
4'h1: decimal= 10'b0000000010;
4'h2: decimal= 10'b0000000100;
4'h3: decimal= 10'b0000001000;
4'h4: decimal= 10'b0000010000;
4'h5: decimal= 10'b0000100000;
4'h6: decimal= 10'b0001000000;
4'h7: decimal= 10'b0010000000;
4'h8: decimal= 10'b0100000000;
4'h9: decimal= 10'b1000000000;
endcase
end
endmodule
```

The four bits from the input are passed to the `case` statement. The `case` statement assigns an appropriate binary expression of the input's decimal value to the `decimal` output and latches that value in register `decimal`.

To avoid creating latches, assign a value to all variables. The code in Example 6-2 does not create latches if you add the following statement to the `case` statement.

```
default: decimal= 10'b0000000000;
```

Variables declared within a function do not hold their values over time because every time a function is called, its variables are reinitialized. Therefore, FPGA Express does not infer latches for these variables. In Example 6-3, no latches are inferred.

Example 6-3 Variable Declared within a Function—No Latches Inferred

```
function my_func;
input data, gate;
reg state;
begin
if (gate) begin
state = data;
end
my_func = state;
end

endfunction
```

Both Example 6-4 and Example 6-5 assign all their variables under all circumstances and avoid creating latches in FPGA Express.

Example 6-4 Avoiding Latch Inference

```
always @ ( PHI_1 or A ) begin
Y = 0;
if ( PHI_1 ) begin
Y = A;
end
end
```

Example 6-5 Another Way to Avoid Creating Latches

```
always @ ( PHI_1 or A ) begin
if( PHI_1 ) begin
Y = A;
end else begin
Y = 0;
end
end
```

Simple Flip-Flop Inference

A flip-flop is implied when you use the `posedge` or `negedge` `clock` constructs in an `always` block, as shown below.

```
always @ ( posedge clock ) begin
    ...
end
```

A variable that is assigned a value in this `always` block is synthesized as a D-type edge-triggered flip-flop. The flip-flop is clocked on the rising (or falling) edge of the signal (*`clock`*) following the `posedge` (or `negedge`) keyword. With simple flip-flops (with no asynchronous set or reset), the block's *`event-expression`* may contain only one `posedge` (or `negedge)` statement, as shown in Example 6-6.

Example 6-6 Creating a Flip-Flop

```
always @ ( posedge CLK ) begin
Y = A & B;
end
```

This code is synthesized into a D-type positive-edge triggered flip-flop with the `D` input connected to `A & B`, the `Q` output connected to `Y`, and the clock input connected to `CLK`.

Flip-Flop Inference with Asynchronous Reset

The actual clock used for flip-flops is derived from the *`event-expression`* for the `always` block. In the *`event-expression`*, test for the `posedge` or `negedge` edges for all reset conditions and your clock.

When you build an asynchronous reset, the `always` block has a specific format. Each reset condition must be a single-bit quantity.

To reset when the condition is high, follow these steps:

1. Use the clause `posedge` *`condition`* in the *`event-expression`* at the beginning of the `always` block.

2. Test the condition in an `if` or `else if` statement. For example:

   `if (`*`condition`*`)`

To reset when the condition is low, follow these steps:

3. Use the clause `negedge` ***`condition`*** in the *event-expression* at the beginning of the `always` block.

4. Test the condition's complement in an `if` or `else if` statement. For example:

   ```
   if (!condition)
   ```

The first reset condition must appear in the first `if` statement. This statement must be of the form

```
if ( condition )

if ( condition == 1'b1)

if ( ~condition )

if ( condition == 1'b0)
```

or

```
if ( ! condition )
```

In the first two cases, a corresponding `posedge` *`condition`* clause must appear in the *`event-expression`* at the beginning of the `always` block. In the following cases, a corresponding `negedge` *`condition`* must appear there.

If subsequent optional reset conditions are used, they are placed in `else if` clauses of the form

```
else if ( condition2 )
```

or

```
else if ( ! condition2 )
```

These conditions also require corresponding `posedge` and `negedge` entries in the event-expression at the beginning of the `always` block. More information about this type of flip-flop is provided in the section "Additional Types of Register Inference."

The clock for the flip-flop is determined by default when FPGA Express reaches the final `else` clause. Remember that this clause has no condition to test. The clocked event is assumed. The flip-flop is clocked on the rising (falling) edge of the signal following the `posedge` (`negedge`) keyword in the *`event-expression`* at the beginning of the `always` block. See Example 6-7.

Example 6-7 Flip-Flop with Asynchronous Reset

```
module example (a,b,clk,reset,c);
input a,b,clk,reset;
output c;
reg c;

always @ (posedge clk or negedge reset) begin
if (!reset) // asynchronous reset
c = 0;
else // posedge clk is assumed
c = a & b;

end
endmodule
```

Refer to Examples A–3 and A–4 in Appendix A for more examples of register use.

Restrictions on Register Capabilities

Indexed expressions are not allowed in the predicate of an *`event-expression`*. The following example shows an indexed expression and the error message generated by FPGA Express.

```
always @ (posedge clk[1])

Error: In an event expression with ‘posedge’ and
‘negedge’
 qualifiers, only simple identifiers are allowed %s.
(VE-91)
```

- Set and reset conditions must be 1-bit variables. If you use an expression for a multibit variable (a bus), FPGA Express generates an error message, as shown in the following example.

```
always @ (posedge clk or negedge reset_bus) begin
if (!reset_bus[1])
.
.
end

Error: The expression for the reset condition of the
‘if’ statement in this ‘always’ block can only be a
simple identifier or its negation (%s). (VE-92)
```

  You can use an expression for the reset condition, such as

```
if (reset == 1’b0)
```

  or

```
if (~reset)
```

  but you cannot use a complex expression, such as

```
if (reset == (1-1))
```

- Use an `if` statement at the top level of an `always` block. The following example results in an error message.

```
always @ (posedge clk or posedge reset) begin
#1;
if (reset) ...
.
.
end

Error: The statements in this ‘always’ block are
outside the scope of the synthesis policy (%s). Only
an ‘if’ statement is allowed at the top level in this
‘always’ block. Please refer to the    FPGA Express
Verilog Reference Manual   for ways to infer flip-flops
and latches from ‘always’ blocks. (VE-93)
```

- To correctly model the loading of asynchronous data to a flip-flop, make the load condition `false` every time the asynchronous data changes, then return the load condition to `true` to latch the new data. See Example 5-41.

Additional Types of Register Inference

For examples describing various types of latches and flip-flops that use directives and variables introduced in the following sections, see the *HDL Coding Style: Sequential Devices Application Note*.

Directives

The following FPGA Express directives can assist with the inference of more complex sequential devices.

```
// synopsys async_set_reset
// synopsys sync_set_reset
// synopsys async_set_reset_local
// synopsys sync_set_reset_local
// synopsys async_set_reset_local_all
// synopsys sync_set_reset_local_all
// synopsys one_hot
// synopsys one_cold
```

async_set_reset Directive

`async_set_reset` takes one argument of a double-quoted list of single-bit signals separated by commas. FPGA Express checks whether an object specified by the `async_set_reset` directive asynchronously sets or resets a latch or flip-flop in the entire design.

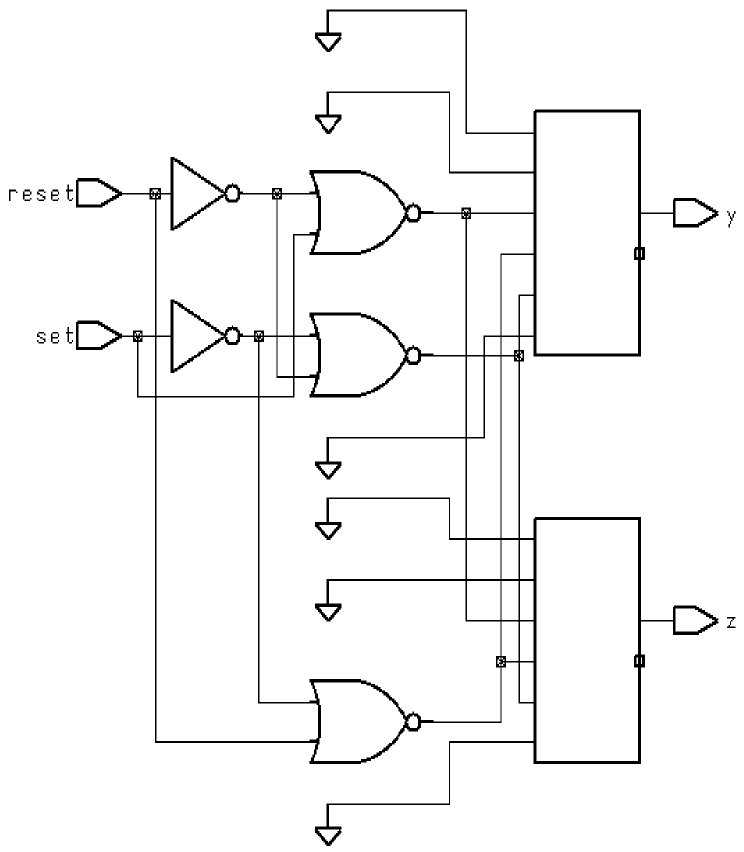
The syntax of `async_set_reset` is

```
// synopsys async_set_reset "object_name,..."
```

Example 6-8 Asynchronous Set/Reset of a Design

```
module async_set_reset(reset, set, d, gate, y, t) ;

input reset, set, gate, d ;
output y, t ;

// synopsys async_set_reset "reset, set"

reg y, t ;

always @ (reset or set)
begin : direct_set_reset
    if (reset)
      y = 1'b0; // asynchronous reset
    else if (set)
      y = 1'b1; // synchronous set
end

always @ (gate or reset) // for set : (gate or set)

    if (reset) // for set : if (set)
      t = 1'b0; // for set : t = 1'b1
    else if (gate)
      t = d ;

endmodule
```

Figure 6-1 Asynchronous Set/Reset of a Design



sync_set_reset Directive

The `sync_set_reset` directive takes one argument of a double-quoted list of single-bit signals separated by commas. FPGA Express checks whether an object specified by the `sync_set_reset` directive synchronously sets or resets a latch or flip-flop in the entire design.

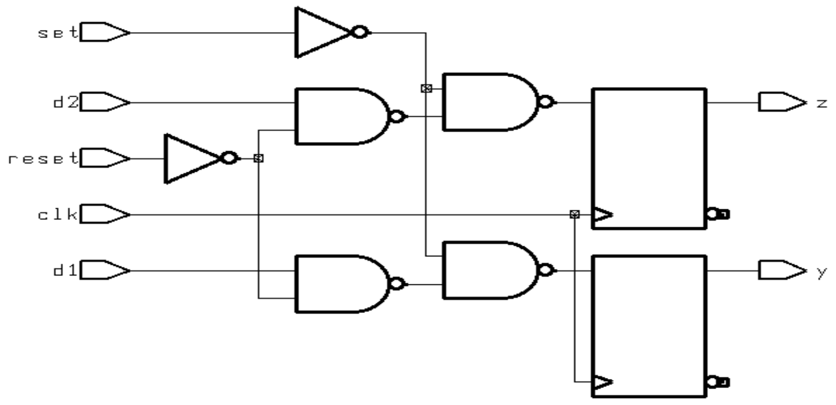
The syntax of `sync_set_reset` is

```
// synopsys sync_set_reset "object_name,..."
```

Example 6-9 Synchronous Set/Reset of a Design

```
module sync_set_reset(clk, reset, set, d1, d2, y, t)
;

input clk, reset, set, d1, d2 ;
output y, t ;

// synopsys sync_set_reset "reset, set"

reg y, t ;

always @ (posedge clk)
begin : synchronous_reset

    if (reset)
      y = 1’b0; // synchronous reset
    else
      y = d1;
end

always @ (posedge clk)
begin : synchronous_set

    if (set)
      t = 1’b1; // synchronous set
    else
      t = d2;
end

endmodule
```

Figure 6-2 *Synchronous Set/Reset of a Design*



async_set_reset_local Directive

The async_set_reset_local directive takes two arguments. The first argument is the label of a block. The second is a double-quoted list of single-bit signals separated by commas. Every signal in the list is treated as though the async_set_reset directive is attached in the specified block.

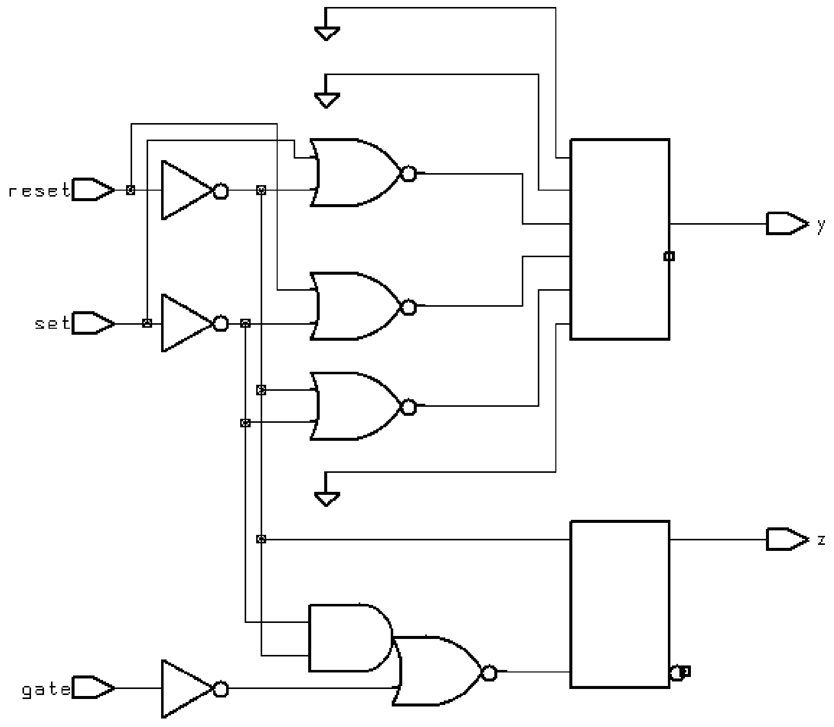
The syntax of async_set_reset_local is

```
// synopsys async_set_reset_local block_label "object_name,..."
```

Example 6-10 Asynchronous Set/Reset of a Single Block

```
module async_set_reset_local(reset, set, gate, y, t)
;

input gate, reset, set ;
output y, t ;

// synopsys async_set_reset_local direct_set_reset
"reset, set"

reg y, t ;

always @ (reset or set)
begin : direct_set_reset

    if (reset)
      y = 1'b0; // asynchronous reset
    else if (set)
      y = 1'b1; // asynchronous set
end

always @ (gate or reset or set)
begin : gated_data

    if (gate)
    begin
      if (reset)
        t = 1'b0; // gated data
      else if (set)
        t = 1'b1; // gated data
    end
end

endmodule
```

Figure 6-3 *Asynchronous Set/Reset of a Single Block*



sync_set_reset_local Directive

The `sync_set_reset_local` directive takes two arguments. The first is the label of a block. The second is a double-quoted list of single-bit signals separated by commas. Every signal in the list is treated as though the `sync_set_reset` directive is attached in the specified block.

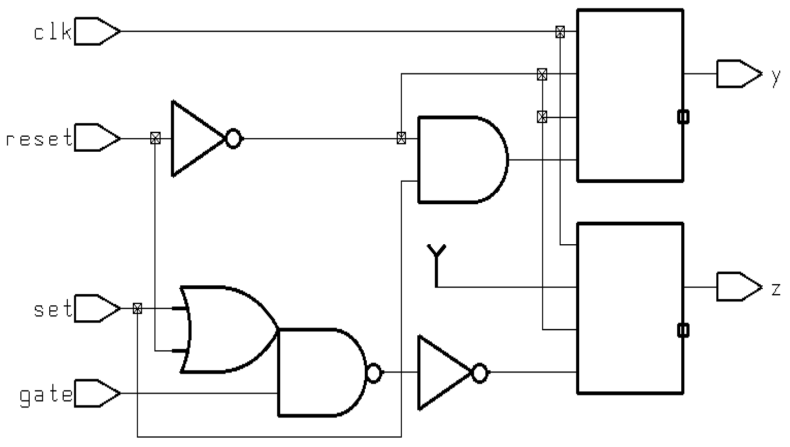
The syntax of `sync_set_reset_local` is

```
// synopsys sync_set_reset_local block_label
"signal_name,..."
```

Example 6-11 Synchronous Set/Reset of a Single Block

```
module sync_set_reset_local(clk, reset, set, gate, d,
y, t) ;

input clk, gate, reset, set, d ;
output y, t ;

// synopsys sync_set_reset_local clocked_set_reset
"reset"

reg y, t ;

always @ (posedge clk)
begin : clocked_reset

    if (reset)
      y = 1'b0; // synchronous reset
    else
      y = d ;
end

always @ (posedge clk)
begin : gated_data

    if (gate)
    begin
      if (reset)
        t = 1'b0; // gated data
      else if (set)
        t = 1'b1; // gated data
    end
end

endmodule
```

Figure 6-4 *Synchronous Set/Reset of a Single Block*



async_set_reset_local_all Directive

The `async_set_reset_local_all` directive takes only one argument, the list of block labels. The `async_set_reset_local_all` directive specifies that *all* the signals are treated as though the `async_set_reset` directive is attached in each of the blocks.

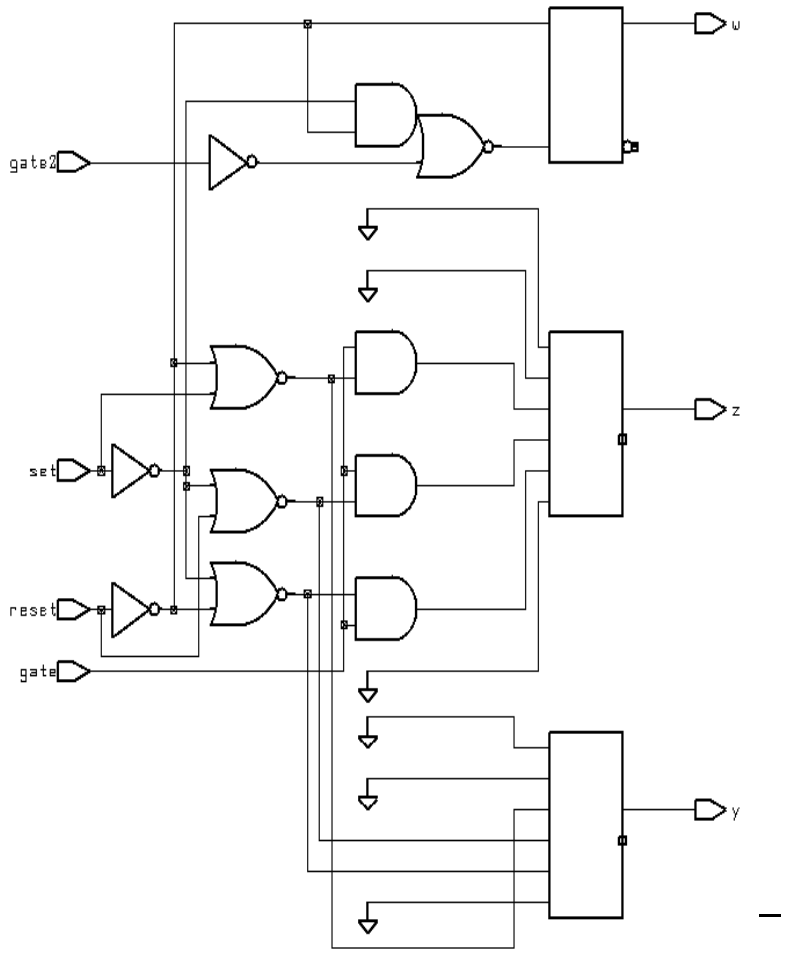
The syntax of `async_set_reset_local_all` is

```
// synopsys async_set_reset_local_all "block_
label,..."
```

Example 6-12 Asynchronous Set/Reset for Part of a Design

```
module async_set_reset_local_all(reset, set, gate,
gate2, y, t, w) ;
input gate, gate2, reset, set ;
output y, t, w ;
// synopsys async_set_reset_local_all "direct_set_
reset, direct_set_reset_too"
reg y, t, w ;
always @ (reset or set)
begin : direct_set_reset
    if (reset)
      y = 1'b0; // asynchronous reset
    else if (set)
      y = 1'b1; // asynchronous set
end

always @ (gate or reset or set)
begin : direct_set_reset_too
    if (gate)
    begin
      if (reset)
        t = 1'b0; // asynchronous reset
      else if (set)
        t = 1'b1; // asynchronous set
    end
end

always @ (gate2 or reset or set)
begin : gated_data
    if (gate2)
    begin
      if (reset)
        w = 1'b0; // gated data
      else if (set)
        w = 1'b1; // gated data
    end
end
endmodule
```

Figure 6-5 *Asynchronous Set/Reset for Part of a Design*



sync_set_reset_local_all Directive

The `sync_set_reset_local_all` directive takes only one argument, the list of block labels. The `sync_set_reset_local_all` directive specifies that a*ll* the signals are treated as though the `sync_set_reset` directive is attached in each of the blocks.

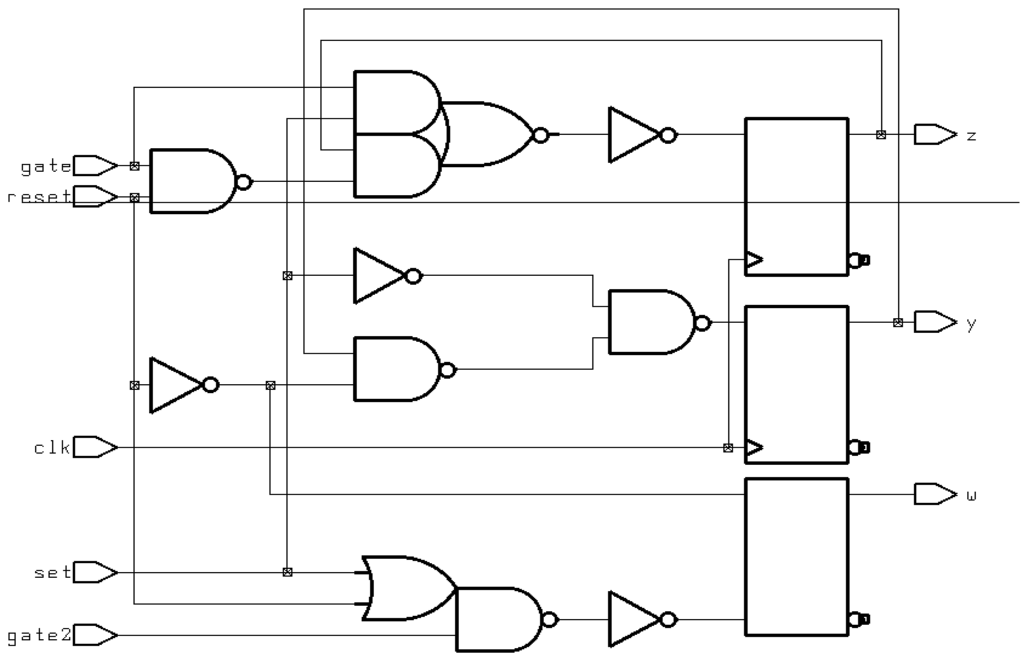
The syntax of `sync_set_reset_local_all` is

```
// synopsys sync_set_reset_local_all "block_
label,..."
```

Example 6-13 Synchronous Set/Reset for Part of a Design

```
module sync_set_reset_local_all(clk, reset, set,
gate, gate2, y, t, w) ;
input clk, gate, gate2, reset, set ;
output y, t, w ;
// synopsys sync_set_reset_local_all "clocked_set_
reset, clocked_set_reset_too"
reg y, t, w ;
always @ (posedge clk)
begin : clocked_set_reset
    if (reset)
      y = 1’b0; // synchronous reset
    else if (set)
      y = 1’b1; // synchronous set
end
always @ (posedge clk)
begin : clocked_set_reset_too
    if (gate)
    begin
      if (reset)
        t = 1’b0; // synchronous reset
      else if (set)
        t = 1’b1; // synchronous set
    end
end
always @ (gate2 or reset or set)
begin : gated_data
    if (gate2)
    begin
      if (reset)
        w = 1’b0; // gated data
      else if (set)
        w = 1’b1; // gated data
    end
end
endmodule
```

Figure 6-6 *Synchronous set/reset for Part of a Design*



Note: Use the **`one_hot`** ***and*** **`one_cold`** ***directives to implement D flip-flops with asynchronous set and reset signals. These two directives tell FPGA Express that only one of the objects in the list are active at a time. If you are defining active high signals, use the*** **`one_hot`** ***directive. For active low signals, use the*** **`one_cold`** ***directive. Each directive specifies two objects.***

one_hot Directive

The `one_hot` directive takes one argument of a double-quoted list of objects separated by commas. This directive indicates that the group of signals are `one_hot`. For example, no more than one signal has a Logic 1 value. Users are responsible to ensure that the group of objects are `one_ hot`. In Example 6-14, FPGA Express does not synthesize logic to check this assertion.

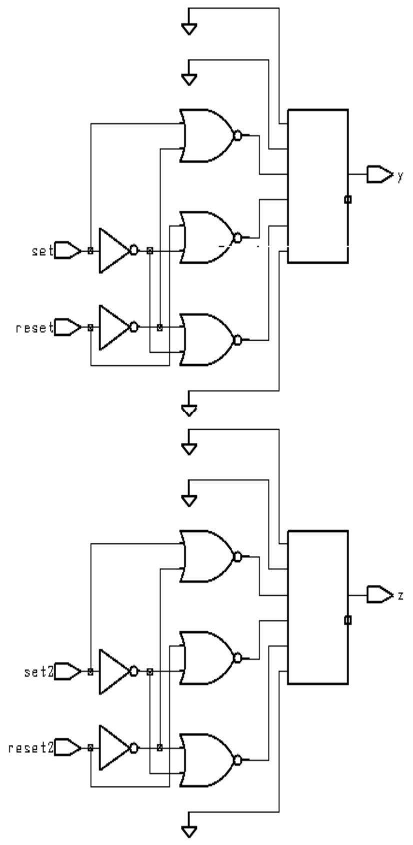
The syntax of `one_hot` is

```
// synopsys one_hot "object_name,..."
```

Example 6-14 Using the one_hot Directives for Set and Reset

```
module one_hot_example (reset, set, reset2, set2, y,
t) ;
input reset, set, reset2, set2 ;
output y, t ;
// synopsys async_set_reset "reset, set"
// synopsys async_set_reset "reset2, set2"
// synopsys one_hot "reset, set"
reg y, t ;

always @ (reset or set)
begin : direct_set_reset
    if (reset)
      y = 1’b0; // asynchronous reset by "reset"
    else if (set)
      y = 1’b1; // asynchronous set by "set"
end

always @ (reset2 or set2)
begin : direct_set_reset_too
    if (reset2)
      t = 1’b0; // asynchronous reset by "reset2"
    else if (set2)
      t = 1’b1; // asynchronous set by "~reset2 set2"
end

// synopsys translate_off
always @(reset or set)
    if (reset & set)
       $write("ONE-HOT violation for ’reset’,
’set’.");
// synopsys translate_on

endmodule
```

Figure 6-7 Using the one_hot Directive for Set and Reset



one_cold Directive

The `one_cold` directive is similar to the `one_hot` directive. `one_cold` indicates that no more than one object in the group has a Logic 0 value.

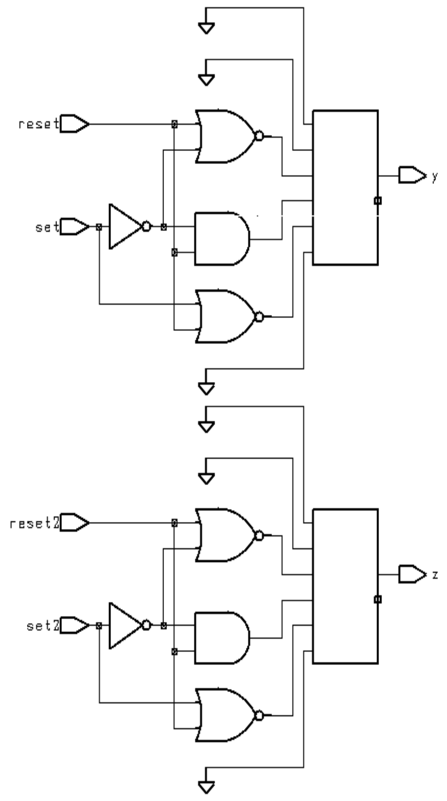
The syntax of the `one_cold` directive is

```
// synopsys one_cold "  signal_name,..."
```

Example 6-15 Using the one_cold Directive for Set and Reset

```
module one_cold(reset, set, reset2, set2, y, t) ;
input reset, set, reset2, set2 ;
output y, t ;
// synopsys async_set_reset "reset, set"
// synopsys async_set_reset "reset2, set2"
// synopsys one_cold "reset, set"
reg y, t ;

always @ (reset or set)
begin : direct_set_reset
    if (~reset)
      y = 1'b0; // asynchronous reset by "~reset"
    else if (~set)
      y = 1'b1; // asynchronous set by "~set"
end

always @ (reset2 or set2)
begin : direct_set_reset_too
    if (~reset2)
      t = 1'b0; // asynchronous reset by "~reset2"
    else if (~set2)
      t = 1'b1; // asynchronous set by "reset2 ~set2"
end

// synopsys translate_off
always @(reset or set)
    if (~reset & ~set)
       $write("ONE-COLD violation for 'reset',
'set'.");
// synopsys translate_on

endmodule
```

Figure 6-8 Using the one_cold Directive for Set and Reset



FPGA Express Latch and Flip-Flop Inference

For latches, FPGA Express interprets each control object as synchronous. For a design subsequently analyzed, every constant `0` loaded on a latch is used for asynchronous reset, and every constant `1` loaded on a latch is used for asynchronous set. FPGA Express does not limit checks for assignments to a constant `0` or constant `1` to a single process. That is, FPGA Express performs checking across processes and provides a brief report for inferred latches.

For flip-flops, FPGA Express removes all feedback loops. For example, feedback loops inferred from a statement such as `Q=Q` are removed. With the state feedback removed from a simple D flip-flop, it becomes a synchronous loaded flip-flop. In addition, FPGA Express removes all inverted flip-flop feedback loops. For example, feedback loops inferred from a statement such as $Q=\overline{Q}$ are removed and synthesized as T flip-flops.

Delays in Registers

If you use delay specifications with values that may be registered, they may cause the simulation to behave differently from the logic synthesized by HDL Compiler. For example, the module in Example 6–24 contains delay information that causes Design Compiler to synthesize a circuit that behaves unexpectedly.

Delays in Registers

```
module problem ( A, C, D, clock );
input A, clock;
output C, D;
wire B;
assign B = #100 A;

flip-flop f1 ( A, C, clock ),
          f2 ( B, D, clock );
endmodule

module flip-flop ( D, Q, clock );
input D, clock
output q;
always @ ( posedge clock ) Q = #5 D;
endmodule
```

In Example 6–24, `B` changes 100 time units after `A` changes. If the clock period is less than 100, output `D` is one or more clock cycles behind output `C` when the circuit is simulated. However, because HDL Compiler ignores the delay information, `A` and `B` change values at the same time, and so do `C` and `D`. This behavior is *not* the same as in the simulated circuit.

When you use delay information in your designs, make sure 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.

Efficient Use of Registers

All variables that are assigned values in an `always` block containing either a `posedge` or `negedge` clock are synthesized with flip-flops. To avoid the flip-flop inference, keep combinational logic in a separate `always` block that does not have a `posedge` or `negedge` clock. See the section "Minimizing Registers" in Chapter 8.

Three-State Inference

FPGA Express can infer three-state gates from the value `z` (high impedance) in the Verilog language. When a variable is assigned the value `z`, the output of the three-state gate is disabled.

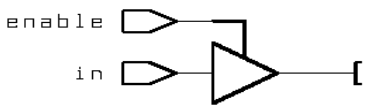
Example 6-16 shows a three-state gate described in Verilog.

Example 6-16 Creating a Three-State Gate in Verilog

```
module simple_threestate ( enable, in, out );
  input  in, enable;
  output out;
  reg    out;

  always @(enable or in) begin
    if (enable)
      out = in;
    else
      out = 1'bz;  // assigns high-impedance
  end
endmodule
```

Figure 6-9 shows the three-state gate from Example 6-16 in a schematic.

Figure 6-9 A Three-State Gate in a Schematic



A 4-bit-wide bus can be assigned high impedance values with `4'bzzzz` just as a bit value is assigned `1'bz` in Example 6-16.

One three-state device is inferred from a single `always` block. Example 6-17 infers only one three-state device.

Example 6-17 One Three-State Device

```
always @(sela or selb or a or b) begin
  t = 1'bz;
    if (sela)
      t = a;
    if (selb)
      t = b;
end
```

The value `z` can also appear in function calls, return statements, and aggregates, although it is valid to use `z` in an expression such as

```
if (value == 1’bz)
```

Expressions that compare a value to `z` are always evaluated as `false` during synthesis. This evaluation might cause a difference between presynthesis and postsynthesis simulations.

Example 6-18 infers two three-state devices.

Example 6-18 Inferring Two Three-State Devices

```
always @(sel_a or a)
  if (sel_a)
    t = a
  else t = 1’bz;
always @(sel_b or b)
  if (sel_b)
    t = b;
  else t = 1’bz;
```

The Verilog conditional statement can also be used to infer three states.

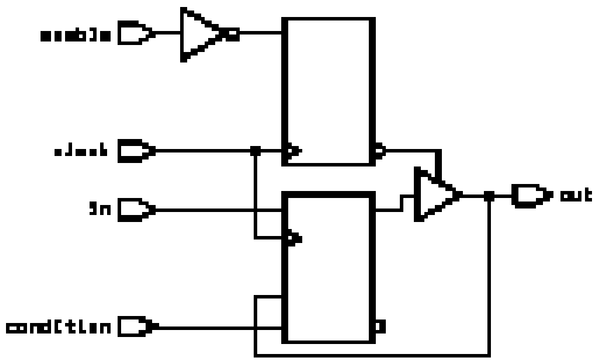
Registered and Latched Three-State Enables

When a variable is registered (or latched) in the same process in which it is three-stated, the enable of the three-state is also registered (or latched). Example 6-19 shows an example of this code and Figure 6-10 shows the schematic generated by the code.

Example 6-19 Three-State with Registered Enable (Inefficient Description)

```
module enable_ff ( clock, condition, enable, in, out
);
  input  in, enable, condition, clock;
  output out;
  reg    out;

  always @ ( posedge clock ) begin
    if ( enable )
       out = ( ~condition ) ? in : out;
   else
       out = 1’bz;
  end
endmodule
```

Figure 6-10 Schematic for a Three-State with a Registered Enable (Inefficient Version)

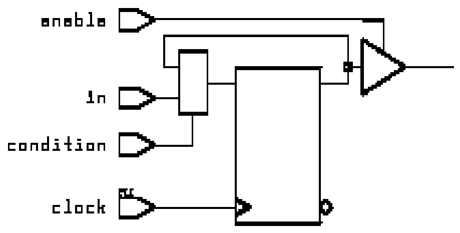


In Example 6-19, the three-state gate has a register on its enable. To remove the register from the enable, use two `always` blocks to separate the register inference from the three-state gate inference, and add a register `temp`. Refer to Example 6-20 and Figure 6-11.

Example 6-20 Three-State without a Registered Enable

```
module no_enable_ff (clock, condition, enable, in,
out);
  input  in, enable, condition, clock;
  output out;
  reg  out;
  reg  temp;

  always @(posedge clock) begin // flip-flop on input
    if ( condition )
      temp = in;
  end

  always @(enable or temp) begin
    if (enable)                // three-state
      out = temp;
    else
      out = 1'bz;
  end
endmodule
```

Figure 6-11 Schematic for a Three-State without a Registered Enable



Chapter 7
FPGA Express Directives

FPGA Express translates a Verilog description to a Synopsys internal format. Specific aspects of this process can be controlled by special *FPGA Express directives* in the Verilog source code. These directives are treated as comments by Verilog simulators and do not affect simulation.

This chapter describes FPGA Express directives and their effect on translation in the following sections:

- Notation for HDL Compiler Directives
- *translate_off* and *translate_on* Directives
- *parallel_case* Directive
- *Full_case* Directive
- Component Implication

Note: Begin each of the above directives with `//synopsys`. You can also use `$s` in place of `synopsys`.

Notation for FPGA Express Directives

The special comments that make up FPGA Express directives begin, like all Verilog comments, with the characters `//` or `/*`. The `//` characters begin a comment that fits on one line (most FPGA Express directives fit on one line). If you use the `/*` characters to begin a multiline comment, you must end the comment with `*/`. You do not need to use the `/*` characters at the beginning of each line, only at the beginning of the first line. The word `synopsys` (all lowercase) following the comment characters tells FPGA Express to treat the text following the word `synopsys` as a compiler directive.

Note: You cannot use `// synopsys` in a regular comment. In addition, the compiler displays a syntax error if Verilog code is in a `// synopsys` directive.

translate_off and *translate_on* Directives

The `// synopsys translate_off` and `// synopsys translate_on` directives tell FPGA Express to suspend translation of the source code and restart translation at a later point. Use these directives when your Verilog source code contains commands specific to simulation that are not accepted by FPGA Express.

You turn translation off with

```
// synopsys translate_off
```

or

```
/* synopsys translate_off */
```

You turn translation back on with

```
// synopsys translate_on
```

or

```
/* synopsys translate_on */
```

At the beginning of each Verilog file, translation is enabled. Subsequently, you can use the `translate_off` and `translate_on` directives anywhere in the text. These directives must be used in pairs. Each `translate_off` directive must appear before its corresponding `translate_on` directive. Example 7-1 shows a simulation driver protected by a `translate_off` directive.

Example 7-1 // synopsys translate_on and // synopsys translate_off Directives

```
module trivial (a, b, f);
input a,b;
output f;
    assign f = a & b;

    // synopsys translate_off
    initial $monitor (a, b, f);
    // synopsys translate_on
endmodule

/* synopsys translate_off */
module driver;
    reg [1:0] value_in;
    integer i;

    trivial triv1(value_in[1], value_in[0]);

    initial begin
        for (i = 0; i < 4; i = i + 1)
            #10 value_in = i;
    end
endmodule
/* synopsys translate_on */
```

parallel_case Directive

The `// synopsys parallel_case` directive affects the way logic is generated for the `case` statement. As presented in Chapter 5, a `case` statement generates the logic for a priority encoder. Under certain circumstances, you might not want to build a priority encoder to handle a `case` statement. You can use the `parallel_case` directive to force FPGA Express to generate multiplexer logic instead.

The syntax for the `parallel_case` directive is

```
// synopsys parallel_case
```

or

```
/* synopsys parallel_case */
```

In Example 9–2, the states of a state machine are encoded as *one hot* signals. If the `case` statement in the example were implemented as a priority encoder, the generated logic would be more complex than necessary.

Example 7-2 // synopsys parallel_case Directives

```
reg [3:0] current_state, next_state;
parameter state1 = 4'b0001, state2 = 4'b0010,
state3 = 4'b0100, state4 = 4'b1000;

case (1) //synopsys parallel_case

    current_state[0] : next_state = state2;
    current_state[1] : next_state = state3;
    current_state[2] : next_state = state4;
    current_state[3] : next_state = state1;

endcase
```

Use the `parallel_case` directive immediately after the `case` expression, as shown. This directive makes all case-item evaluations in parallel. *All* case items that evaluate to `true` are executed (not just the first one, which might give you unexpected results.)

In general, use `parallel_case` when you know that only one case item is executed. If only one case item is executed, the logic generated from a `parallel_case` directive performs the same function as the circuit when it is simulated. If two case items are executed, and you have used the `parallel_case` directive, the generated logic is not the same as the simulated description.

full_case Directive

The `// synopsys full_case` directive asserts that all possible clauses of a `case` statement have been covered and that no default clause is necessary. This directive has two uses: it avoids the need for default logic, and it can avoid latch inference from a `case` statement by asserting that all necessary conditions are covered by the given branches of the `case` statement. As shown in Chapter 5, a latch can be inferred whenever a variable is not assigned a value under all conditions.

The syntax for the `full_case` directive is

```
// synopsys full_case
```

or

```
/* synopsys full_case */
```

If the `case` statement contains a `default` clause, FPGA Express assumes that all conditions are covered. If there is no `default` clause, and you do not want latches to be created, use the `full_case` directive to indicate that all necessary conditions are described in the `case` statement.

Example 9–3 shows two uses of the `full_case` directive. Note that the `parallel_case` and `full_case` directives can be combined in one comment.

Example 7-3 // synopsys full_case Directives

```
reg [1:0] in, out;
reg [3:0] current_state, next_state;
parameter state1 = 4'b0001, state2 = 4'b0010,
          state3 = 4'b0100, state4 = 4'b1000;

case (in) // synopsys full_case
    0: out = 2;
    1: out = 3;
    2: out = 0;
endcase

case (1)  // synopsys parallel_case full_case
    current_state[0] : next_state = state2;
    current_state[1] : next_state = state3;
    current_state[2] : next_state = state4;
    current_state[3] : next_state = state1;
endcase
```

In the first `case` statement, the condition `in == 3` is not covered. You can either use a `default` clause to cover all other conditions, or use the `full_case` directive (as in this example) to indicate that other branch conditions do not occur. If you cover all possible conditions explicitly, FPGA Express recognizes the `case` statement as full case, so the `full_case` directive is not necessary.

The second `case` statement in Example 9–3 does not cover all 16 possible branch conditions. For example, `current_state == 4'b0101` is not covered. The `parallel_case` directive is used in this example because only one of the four case items can evaluate to `true` and be executed.

Although you can use the `full_case` directive to avoid creating latches, using this directive does not guarantee that latches will not be built. You must still assign a value to each variable used in the `case` statement in all branches of the `case` statement. Example 9–4 illustrates a situation where the `full_case` directive prevents a latch from being inferred for variable `b`, but not for variable `a`.

Example 7-4 Latches and // synopsys full_case

```
reg a, b;
reg [1:0] c;
case (c)   // synopsys full_case
    0: begin a = 1; b = 0; end
    1: begin a = 0; b = 0; end
    2: begin a = 1; b = 1; end
    3: b = 1;               // a is not assigned here
endcase
```

In general, use the `full_case` directive when you know that all possible branches of the `case` statement have been enumerated or at least all branches that can occur. If all branches that can occur are enumerated, the logic generated from the `case` statement performs the same function as the simulated circuit. If a `case` condition is not fully enumerated, the generated logic and the simulation are not the same.

Note: You do not need the `full_case` directive if you have a default branch or you enumerate all possible branches in a `case` statement because FPGA Express assumes that the `case` statement is `full_case`.

Component Implication

In Verilog, you cannot instantiate modules in behavioral code. To include an embedded netlist in your behavioral code, use the directives `// synopsys map_to_module` and `// synopsys return_port_name` for FPGA Express to recognize the netlist as a *function* being implemented by another module. When this subprogram is invoked in the behavioral code, the module is instantiated.

The first directive, `// synopsys map_to_module`, flags a function for implementation as a distinct component. The syntax is

`// synopsys map_to_module` *`modulename`*

The second directive identifies a return port, because functions in Verilog do not have output ports. A return port name must be identified to instantiate the function as a component. The syntax is

`// synopsys return_port_name` *`portname`*

Note: Remember that if you add a `map_to_module` directive to a function, the contents of the function are parsed and ignored and the indicated module is instantiated. You must ensure that the functionality of the module instantiated in this way and the function it replaces are the same; otherwise, presynthesis and postsynthesis simulation do not match.

Example 9–22 illustrates the `map_to_module` and `return_port_name` directives.

Example 7-19 Component Implication

```
module mux_inst (a, b, c, d, e);
input a, b, c, d;
output e;
function mux_func;
// synopsys map_to_module mux_module
// synopsys return_port_name mux_ret
input in1, in2, cntrl;
/*
** the contents of this function are ignored for
** synthesis, but the behavior of this function
** must match the behavior of mux_module for
** simulation purposes
*/
begin
if (cntrl) mux_func = in1;
else mux_func = in2;
end

endfunction

assign e = a & mux_func (b, c, d); // this function
call
// actually instantiates component (module) mux_
module

endmodule

module mux_module (in1, in2, cntrl, mux_ret);
input in1, in2, cntrl;
output mux_ret;

and and2_0 (wire1, in1, cntrl);
not not1 (not_cntrl, cntrl);
and and2_1 (wire2, in2, not_cntrl);
or or2 (mux_ret, wire1, wire2);

endmodule
```

Chapter 8
Flip-Flops

This appendix is for FPGA Express users whose current design descriptions include hand-instantiated flip-flops. It explains how to translate these flip-flops to `always` blocks that can be used with FPGA Express. Read this appendix after you have read Chapter 5, “Functional Descriptions.”

Some of the benefits of translating your hand-instantiated flip-flops to `always` blocks are

- Clearer code. The logic of the new module definitions is easier to understand.
- Continued compatibility. The new design descriptions can use the expanded capabilities of future versions of FPGA Express.
- Technology independence. Any FPGA library can be used as the target for synthesis of a Verilog description.
- Multiple-bit values. Such values can be registered with a single statement, rather than with multiple flip-flop instantiations.

Translating Flip-flops

The first step in translating a flip-flop to the `always` syntax is to be sure that you understand the function of the module. Next, determine what parts of the module description provide the flip-flop behavior.

Example B–1 shows a simple module that uses three manually inserted flip-flops.

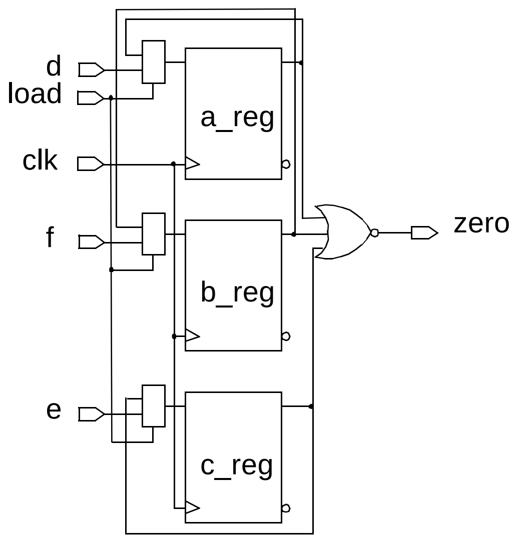
Example 8-1 Existing Module

```
module simple ( d, e, f, load, clk, zero );
  input d, e, f, load, clk;
  output zero;
  reg new_a, new_b, new_c;

function zilch ;
  input load, a, b, c;

begin
  if ( load ) begin
    new_a = d;
    new_b = e;
    new_c = f;
  end
  else begin
    new_a = a;
    new_b = b;
    new_c = c;
  end

  if ( a==0 & b==0 & c==0 )
    zilch=1;
  else
    zilch=0;
  end

endfunction

FD1S a_reg ( new_a, clk, a, );
FD1S b_reg ( new_b, clk, b, );
FD1S c_reg ( new_c, clk, c, );

assign zero = zilch ( load, a, b, c );
endmodule
```

This module evaluates the three state variables, `a`, `b`, and `c`, to determine whether all the values are 0. Additional input signals are `load`, which forces a synchronous reset, and `clk`, which is the module's clock. The functionality of the module is described in the function `zilch`. The input values are latched in the flip-flop described in the three statements beginning with `dFF` (a D-type edge-triggered flip-flop). A final `assign` statement assigns the returned value of the function `zilch` to the output `zero`.

Example B–1 generates the schematic shown in Figure B–1.

Figure 8-1 Schematic from Example B–1



To translate this description, find the combinational logic and determine the triggering events. In this case, the function `zilch` creates combinational logic.

Example 8-2 Existing Module Logic

```
function zilch ;
input load, a, b, c;

if ( load ) begin
new_a = d;
new_b = e;
new_c = f;
end
else begin
new_a = a;
new_b = b;
new_c = c;
end
if ( a==0 & b==0 & c==0 )
zilch=1;
else
zilch=0;
endfunction
```

In Example B–2, the values of `a`, `b`, `c`, `d`, `e`, `f`, and `load` are the triggers (signals that are read). You can rewrite this description as an `always` block with triggers, as shown in Example B–3.

Example 8-3 New Module Logic

```
always @ ( a or b or c or d or e or f or load ) begin
 if ( load ) begin
    new_a = d;
    new_b = e;
    new_c = f;
 end
 else begin
    new_a = a;
    new_b = b;
    new_c = c;
 end

 if ( a==0 & b==0 & c==0 )
   zero=1;
 else
   zero=0;
end
```

The next step is to build an `always` block that replaces the flip-flop instantiations—the three statements that begin with `dFF`.

Example 8-4 Existing Flip-flop Instantiations

```
dFF a_reg ( new_a, clk, a );
dFF b_reg ( new_b, clk, b );
dFF c_reg ( new_c, clk, c );
```

Use the clock signal, `clk`, as the event-expression of the new `always` block, as shown.

Example 8-5 First Line of the New always Block

```
always @ ( posedge clk ) begin
```

Put the values and the registers in the body of the `always` block. The `Q` output values in the old module (`a`, `b`, and `c`) become the assigned values in the new version. The clock from the old version is specified in the *event-expression* of the new `always` block. The `D` input values in the old module (`new_a`, `new_b`, and `new_c`) become the values read by the new version, as shown in Example B–6.

Example 8-6 New Clocked always Block

```
always @ ( posedge clk ) begin
a = new_a ;
b = new_b ;
c = new_c ;
end
```

Now, label the input and output signals in the module. Look at the variable declarations and determine which of the wires and functions serve the flip-flop and which serve the logic of the module.

Example 8-7 Existing Inputs and Outputs

```
module simple ( d, e, f, load, clk, zero );
input d, e, f, load, clk;
output zero;
reg new_a, new_b, new_c;
```

In this case, as in most cases, the module's inputs and outputs remain the same. However, you must change the `wire` values to `reg` values. Declare the output `zero` twice; once as the output and once as a `reg`, so it can be used in the `always` block. Make the former function variables `a`, `b`, and `c` into `reg` variables, because they are now assigned within the second `always` block. Example B–8 shows the new input and output declarations.

Example 8-8 New Input and Output Declarations

```
module new_simple ( d, e, f, load, clk, zero );
input d, e, f, load, clk;
output zero;
reg zero;
reg a, b, c;
reg new_a, new_b, new_c;
```

Example B–9 shows the complete new module with `always` blocks.

Example 8-9 Translated Module Using always Blocks

```
module new_simple ( d, e, f, load, clk, zero );
input d, e, f, load, clk;
output zero;
reg zero;
reg a, b, c;
reg new_a, new_b, new_c;

always @ ( a or b or c or d or e or f or load ) begin
if ( load ) begin
new_a = d;
new_b = e;
new_c = f;
end
else begin
new_a = a;
new_b = b;
new_c = c;
end

if ( a==0 & b==0 & c==0 )
zero=1;
else
zero=0;
end

always @ ( posedge clk ) begin
a = new_a ;
b = new_b ;
c = new_c ;
end
endmodule
```

Chapter 9
Verilog Syntax

This appendix contains a syntax description of the Verilog language as supported by FPGA Express. This appendix covers the following topics:

- Syntax
- Lexical Conventions
- Verilog Keywords
- Unsupported Verilog Language Constructs

Syntax

This section presents the syntax of the supported Verilog language in Backus Naur Form (BNF), and presents the syntax formalism.

Note: The BNF syntax convention used in this section differs from the Synopsys syntax convention used elsewhere in this manual.

BNF Syntax Formalism

White space separates lexical tokens.

`name` is a keyword.

`<name>` is a syntax construct definition.

`<name>` is a syntax construct item.

`<name>?` is an optional item.

`<name>*` is zero, one, or more items.

`<name>+` is one or more items.

`<port> <,<port>>*` is a comma-separated list of items.

`::=` gives a syntax definition to an item.

`||=` refers to an alternative syntax construct.

BNF Syntax

`<source_text>`

```
::= <description>*
```

`<description>`

```
::= <module>
```

`<module>`

```
::= module <name_of_module> <list_of_ports>? ;
           <module_item>*
    endmodule
```

`<name_of_module>`

```
::= <IDENTIFIER>
```

`<list_of_ports>`

```
::= ( <port> <,<port>>* )
           ||= ( )
```

`<port>`

```
::= <port_expression>?
||= . <name_of_port> ( <port_expression>? )
```

`<port_expression>`

```
::= <port_reference>
||= { <port_reference> <, <port_reference>>* }
```

<port_reference>

```
   ::= <name_of_variable>
   ||= <name_of_variable> [ <expression> ]
   ||= <name_of_variable> [ <expression> :
<expression> ]
```

<name_of_port>

```
   ::= <IDENTIFIER>
```

<name_of_variable>

```
   ::= <IDENTIFIER>
```

<module_item>

```
   ::= <parameter_declaration>
   ||= <input_declaration>
   ||= <output_declaration>
   ||= <inout_declaration>
   ||= <net_declaration>
   ||= <reg_declaration>
   ||= <integer_declaration>
   ||= <gate_instantiation>
   ||= <module_instantiation>
   ||= <continuous_assign>
   ||= <function>
```

<function>

```
   ::= function <range>? <name_of_function> ;
            <func_declaration>*
            <statement_or_null>
       endfunction
```

<name_of_function>

```
   ::= <IDENTIFIER>
```

<func_declaration>

```
   ::= <parameter_declaration>
   ||= <input_declaration>
   ||= <reg_declaration>
   ||= <integer_declaration>
```

<always>

```
   ::= always @ ( <identifier> or <identifier> )
   ||= always @ ( posedge <identifier> )
   ||= always @ ( negedge <identifier> )
   ||= always @ ( <egde> or <edge> or ... )
```

<edge>

```
   ::= posedge <identifier>
   ||= negedge <identifier>
```

<parameter_declaration>

```
   ::= parameter <range>? <list_of_assignments> ;
```

<input_declaration>

```
   ::= input <range>? <list_of_variables> ;
```

<output_declaration>

```
   ::= output <range>? <list_of_variables> ;
```

<inout_declaration>

```
   ::= inout <range>? <list_of_variables> ;
```

<net_declaration>

```
   ::= <NETTYPE> <charge_strength>? <expandrange>?
<delay>? <list_of_variables> ;
   ||= <NETTYPE> <drive_strength>? <expandrange>?
<delay>? <list_of_assignments> ;
```

<NETTYPE>

```
   ::= wire
   ||= wor
   ||= wand
   ||= tri
```

<expandrange>

```
   ::= <range>
   ||= scalared <range>
   ||= vectored <range>
```

<reg_declaration>

```
   ::= reg <range>? <list_of_register_variables> ;
```

<integer_declaration>

```
   ::= integer <list_of_integer_variables> ;
```

<continuous_assign>

```
   ::= assign <drive_strength>? <delay>?
             <list_of_assignments>;
```

<list_of_variables>

```
   ::= <name_of_variable> <, <name_of_variable>>*
```

<name_of_variable>

```
   ::= <IDENTIFIER>
```

<list_of_register_variables>

```
   ::= <register_variable> <, <register_variable>>*
```

<register_variable>

```
   ::= <IDENTIFIER>
```

<list_of_integer_variables>

```
   ::= <integer_variable> <, <integer_variable>>*
```

<integer_variable>

```
::= <IDENTIFIER>
```

<charge_strength>

```
::= ( small )
||= ( medium )
||= ( large )
```

<drive_strength>

```
::= ( <STRENGTH0> , <STRENGTH1> )
||= ( <STRENGHT1> , <STRENGTH0> )
```

<STRENGTH0>

```
::= supply0
||= strong0
||= pull0
||= weak0
||= highz0
```

<STRENGTH1>

```
::= supply1
||= strong1
||= pull1
||= weak1
||= highz1
```

<range>

```
::= [ <expression> : <expression> ]
```

<list_of_assignments>

```
::= <assignment> <, <assignment>>*
```

<gate_instantiation>

```
::= <GATETYPE> <drive_strength>? <delay>?
         <gate_instance> <, <gate_instance>>* ;
```

<GATETYPE>

```
::= and
||= nand
||= or
||= nor
||= xor
||= xnor
||= buf
||= not
```

<gate_instance>

```
::= <name_of_gate_instance>? ( <terminal>
                    <, <terminal>>* )
```

<name_of_gate_instance>

```
::= <IDENTIFIER>
```

<terminal>

```
::= <identifier>
||= <expression>
```

<module_instantiation>

```
::= <name_of_module> <parameter_value_assignment>?
    <module_instance> <, <module_instance>>* ;
```

<name_of_module>

```
::= <IDENTIFIER>
```

<parameter_value_assignment>

```
::= #( <expression> <,<expression>>*)
```

<module_instance>

```
::= <name_of_module_instance>
    ( <list_of_module_terminals>? )
```

<name_of_module_instance>

```
::= <IDENTIFIER>
```

<list_of_module_terminals>

```
   ::= <module_terminal>? <,<module_terminal>>*
   ||= <named_port_connection> <,<named_port_
connection>>*
```

<module_terminal>

```
::= <identifier>
||= <expression>
```

<named_port_connection>

```
::= . IDENTIFIER ( <identifier> )
||= . IDENTIFIER ( <expression> )
```

<statement>

```
   ::= <assignment>
   ||= if ( <expression> )
          <statement_or_null>
   ||= if ( <expression> )
          <statement_or_null>
       else
          <statement_or_null>
   ||= case ( <expression> )
          <case_item>+
       endcase
   ||= casex ( <expression> )
          <case_item>+
       endcase
   ||= casez ( <expression> )
          <case_item>+
       endcase
   ||= for ( <assignment> ; <expression> ;
<assignment> )
          <statement>
   ||= <seq_block>
   ||= disable <IDENTIFIER> ;
   ||= forever <statement>
   ||= while ( <expression> ) <statement>
```

<statement_or_null>

```
   ::= statement
   ||= ;
```

<assignment>

```
   ::= <lvalue> = <expression>
```

<case_item>

```
   ::= <expression> <,<expression>>* : <statement_or_
null>
   ||= default : <statement_or_null>
   ||= default <statement_or_null>
```

<seq_block>

```
::= begin
        <statement>*
    end
||= begin : <name_of_block>
        <block_declaration>*
        <statement>*
    end
```

<name_of_block>

```
::= <IDENTIFIER>
```

<block_declaration>

```
::= <parameter_declaration>
||= <reg_declaration>
||= <integer_declaration>
```

<lvalue>

```
::= <IDENTIFIER>
||= <IDENTIFIER> [ <expression> ]
||= <concatenation>
```

<expression>

```
::= <primary>
||= <UNARY_OPERATOR> <primary>
||= <expression> <BINARY_OPERATOR>
||= <expression> ? <expression> : <expression>
```

<UNARY_OPERATOR>

```
::= !
||= ~
||= &
||= ~&
||= |
||= ~|
||= ^
||= ~^
||= -
||= +
```

<BINARY_OPERATOR>

```
::=  +
||=  -
||=  *
||=  /
||=  %
||=  ==
||=  !=
||=  &&
||=  ||
||=  <
||=  <=
||=  >
||=  >=
||=  &
||=  |
||=  <<
||=  >>
```

<primary>

```
::= <number>
||= <identifier>
||= <identifier> [ <expression> ]
||= <identifier> [ <expression> : <expression> ]
||= <concatenation>
||= <multiple_concatenation>
||= <function_call>
||= ( <expression> )
```

<number>

```
::= <NUMBER>
||= <BASE> <NUMBER>
||= <SIZE> <BASE> <NUMBER>
```

<NUMBER>

A number can have any of the following characters:
`0123456789abcdefxzABCDEFXZ`

<SIZE>

```
::= ’b
||= ’B
||= ’o
||= ’O
||= ’d
||= ’D
||= ’h
||= ’H
```

<SIZE>

Any number of the following digits: `0123456789`

<concatenation>

```
   ::= { <expression> <,<expression>>* }
```

<multiple_concatenation>

```
  ::= { <expression> { <expression> <,<expression>>*
} }
```

<function_call>

```
   ::= <name_of_function> ( <expression>
<,<expression>>*)
```

<name_of_function>

```
   ::= <IDENTIFIER>
```

<identifier>

An identifier is any sequence of letters, digits, and the underscore character (_), where the first character is a letter or underscore. Uppercase and lowercase letters are treated as different characters. Identifiers can be any size and all characters are significant. Escaped identifiers start with the backslash character (\) and end with a space. The leading backslash character (\) is not part of the identifier. Use escaped identifiers to include any printable ASCII characters in an identifier.

<delay>

```
   ::= # <NUMBER>
   ||= # <identifier>
   ||= # ( <expression> <,<expression>>* )
```

Lexical Conventions

The lexical conventions used by FPGA Express are nearly identical to those of the Verilog language. The types of lexical tokens used by FPGA Express are described in the following subsections:

- White Space
- Comments
- Numbers
- Identifiers
- Operators
- Macro Substitutions
- `include` Directive
- Simulation Directives
- Verilog System Functions

White Space

White space separates words in the input description, and can contain spaces, tabs, new lines, and form feeds. You can place white space anywhere in the description. FPGA Express ignores white space.

Comments

You can enter comments anywhere in a Verilog description in two forms:

- Beginning with two backslashes `//.`

  FPGA Express ignores all text between these characters and the end of the current line.

- Beginning with the two characters `/*` and ending with `*/`.

  FPGA Express ignores all text between these characters, so you can continue comments over more than one line.

Note: You cannot nest comments.

Numbers

You can declare numbers in several different radices and bit-widths. A *radix* is the base number on which a numbering system is built. For example, the binary numbering system has a radix of 2, octal has a radix of 8, and decimal has a radix of 10.

You can use these three number formats:

1. A simple decimal number that is a sequence of digits between 0 and 9. All constants declared in this way are assumed to be 32-bit numbers.

2. A number that specifies the bit width, as well as the radix. These numbers are exactly the same as the previous format, except they are preceded by a decimal number that specifies the bit width.

3. A number followed by a two-character sequence prefix that specifies the number's size and radix. The radix determines which symbols you can include in the number. Constants declared this way are assumed to be 32-bit numbers. Any of these numbers can include underscores (_). The underscores improve readability and do not affect the value of the number. Table C–1 summarizes the available radices and valid characters for the number.

Table B-1 Verilog Radices

Name	Character Prefix	Valid Characters
binary	’b	0 1 x X z Z _ ?
octal	’o	0–7 x X z Z _ ?
decimal	’d	0–9 _
hexadecimal	’h	0–9 a–f A–F x X z Z _ ?

Example C–1 shows some valid number declarations.

Example B-1 Valid Verilog Number Declarations

```
391              //  32-bit decimal number
’h3a13           //  32-bit hexadecimal number
10’o1567         //  10-bit octal number
3’b010           //  3-bit binary number
4’d9             //  4-bit decimal number
40’hFF_FFFF_FFFF //  40-bit hexadecimal number
2’bxx            //  2-bits don’t care
3’bzzz           //  3-bits high-impedance
```

Identifiers

Identifiers are user-defined words for variables, function names, module names, and instance names. Identifiers can be composed of letters, digits, and the underscore character (_). The first character of an identifier cannot be a number. Identifiers can be any length. Identifiers are case-sensitive and all characters are significant.

An identifier that contains special characters, begin with numbers, or have the same name as a keyword can be specified as an *escaped identifier*. An escaped identifier starts with the backslash character (\), followed by a sequence of characters, followed by white space.

Some escaped identifiers are shown in Example C–2.

Example B-2 Sample Escaped Identifiers

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

The Verilog language supports the concept of *hierarchical names*, which can be used to access variables of submodules directly from a higher-level module. Hierarchical names are partially supported by FPGA Express.

Operators

Operators are one-character or two-character sequences that perform operations on variables. Some examples of operators are `+`, `~^`, `<=`, and `>>`. Operators are described in detail in Chapter 4.

Macro Substitutions

Macro substitution assigns a string of text to a macro variable. The string of text is inserted into the code where the macro is encountered. The definition begins with the back quote character (`‘`), followed by the keyword `define` , followed by the name of the macro variable. All text from the macro variable until the end of the line is assigned to the macro variable.

You can declare and use macro variables anywhere in the description. The definitions can carry across several files that are read into FPGA Express at the same time. To make a macro substitution, type a back quotation mark (`‘`) followed by the macro variable name.

Some sample macro variable declarations are shown in Example C–3.

Example B-3 Macro Variable Declarations

```
‘define highbits      31:29
‘define bitlist       {first, second, third}
wire [31:0] bus;

‘bitlist = bus[‘highbits];
```

include Construct

The `include` construct in Verilog is similar to the `#include` directive in C. You can use this construct to include Verilog code, such as type declarations and functions, from one module into another. Example C–4 shows an application of the `include` construct.

Example B-4 Including a File Within a File

```
Contents of file1.v

‘define WORDSIZE 8
function [WORDSIZE-1:0] fastadder;
.
.
endfunction

Contents of secondfile

module secondfile (in1,in2,out)
‘include “file1.v”
wire [WORDSIZE-1:0] temp;
assign temp = fastadder (in1,in2);
.
.
endmodule
```

Included files can include other files, up to 24 levels of nesting. You cannot use the `include` construct recursively.

Simulation Directives

Simulation directives (not to be confused with FPGA Express directives described in Chapter 6) refer to special commands that affect the operation of the Verilog HDL Simulator. You can include these directives in your design description, because FPGA Express parses and ignores them.

```
‘accelerate ‘celldefine ‘default_nettype
‘endcelldefine ‘endprotect ‘expand_vectornets
‘noaccelerate ‘noexpand_vectornets  ‘noremove_netnames
‘nounconnected_drive   ‘protect ‘remove_netnames
‘resetall ‘timescale ‘unconnected_drive
```

Verilog System Functions

Verilog system functions are implemented by the Verilog HDL Simulators to generate input or output during simulation. Their names start with a dollar sign ($). These functions are parsed and ignored by FPGA Express.

Verilog Keywords

Verilog uses keywords to interpret an input file. You cannot use these words as user variable names unless you use an escaped identifier. For more information, see the section “Identifiers,” earlier in this chapter.

`always`	`and`	`assign`	`begin`
`buf`	`bufif0`	`bufif1`	`case`
`casex`	`casez`	`cmos`	`deassign`
`default`	`defparam`	`disable`	`else`
`end`	`endcase`	`endfunction`	`endmodule`
`endprimitive`	`endtable`	`endtask`	`event`
`for`	`force`	`forever`	`fork`
`function`	`highz0`	`highz1`	`if`
`initial`	`inout`	`input`	`integer`
`join`	`large`	`medium`	`module`
`nand`	`negedge`	`nmos`	`nor`
`not`	`notif0`	`notif1`	`or`
`output`	`parameter`	`pmos`	`posedge`
`primitive`	`pulldown`	`pullup`	`pull0`
`pull1`	`rcmos`	`reg`	`release`
`repeat`	`rnmos`	`rpmos`	`rtran`
`rtranif0`	`rtranif1`	`scalared`	`small`
`strong0`	`strong1`	`supply0`	`supply1`
`supply1`	`table`	`task`	`time`
`tran`	`tranif0`	`tranif1`	`tri`
`triand`	`trior`	`trireg`	`tri0`
`tri1`	`vectored`	`wait`	`wand`
`weak0`	`weak1`	`while`	`wire`
`wor`	`xnor`	`xor`	

Unsupported Verilog Language Constructs

The following Verilog constructs are not supported by FPGA Express.

- Unsupported definitions and declarations
- Unsupported statements
- Unsupported operators
- Unsupported gate-level constructs
- Unsupported miscellaneous constructs

Constructs added to the Verilog Simulator in versions after Verilog 1.6 might not be supported.

If you use an unsupported construct in a Verilog description, FPGA Express issues a syntax error such as

```
event is not supported
```

Unsupported Definitions and Declarations

The following Verilog definitions and declarations are not supported by FPGA Express.

- *primitive* definition
- *time* declaration
- *event* declaration
- *triand*, *trior*, *tri1*, *tri0*, and *trireg* net types
- Ranges and arrays for integers

Unsupported Statements

The following Verilog statements are not supported by FPGA Express.

- *defparam* statement
- *initial* statement
- *repeat* statement
- *delay* control
- *event* control
- *wait* statement
- *fork* statement
- *deassign* statement
- *force* statement
- *release* statement
- Assignment statement with a variable used as a bit-select on the left side of the equal sign

Unsupported Operators

The following Verilog operators are not supported by FPGA Express.

- Case equality and inequality operators (`===` and `!==`)
- Division and modulus operators for variables

Unsupported Gate-Level Constructs

The following Verilog gate-level constructs are not supported by FPGA Express.

- *nmos*, *pmos*, *cmos*, *rnmos*, *rpmos*, *rcmos*, *pullup*, *pulldown*, *tranif0*, *tranif1*, *rtran*, *rtranif0*, and *rtranif1* gate types

Unsupported Miscellaneous Constructs

The following Verilog miscellaneous constructs are not supported by FPGA Express.

- Hierarchical names within a module
- *‘ifdef*, *‘endif* and *‘else* compiler directives