HDL Compiler™ for Verilog
Reference Manual

Version 2000.05, May 2000

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

SYNOPSYS®

Printed in the U.S.A.

Document Order Number: 00039-000 IA
HDL Compiler for Verilog Reference Manual, v2000.05

Contents

4. Expressions

Figures

Tables

Examples

About This Manual

This manual describes the Synopsys HDL Compiler for Verilog tool, a member of the Synopsys HDL Compiler family. HDL Compiler software translates a high-level Verilog language description into a gate-level netlist.

This preface includes the following sections:

- Audience
- Related Publications
- SOLV-IT! Online Help
- Customer Support
- Conventions

Audience

The *HDL Compiler for Verilog Reference Manual* is written for logic designers and electronic engineers who are familiar with the Synopsys Design Compiler tool. Knowledge of the Verilog language is required, and knowledge of a high-level programming language is helpful.

Related Publications

For additional information about HDL Compiler for Verilog, see

- Synopsys Online Documentation (SOLD), which is included with the software
- Documentation on the Web, which is available through SolvNET on the Synopsys Web page at http://www.synopsys.com
- The Synopsys Print Shop, from which you can order printed copies of Synopsys documents, at http://docs.synopsys.com

You might also want to refer to the documentation for the following related Synopsys products:

- Design Analyzer
- Design Compiler
- DesignWare
- Library Compiler
- VHDL System Simulator (VSS)

SOLV-IT! Online Help

SOLV-IT! is the Synopsys electronic knowledge base, which contains information about Synopsys and its tools and is updated daily.

To obtain more information about SOLV-IT!,

1. Go to the Synopsys Web page at http://www.synopsys.com and click SolvNET.
2. If prompted, enter your user name and password. If you do not have a SOLV-IT! user name and password, you can obtain them at http://www.synopsys.com/registration.

Customer Support

If you have problems, questions, or suggestions, contact the Synopsys Technical Support Center in one of the following ways:

- Open a call to your local support center from the Web.
  a. Go to the Synopsys Web page at http://www.synopsys.com and click SolvNET (SOLV-IT! user name and password required).
  b. Click “Enter a Call.”
- Send an e-mail message to support_center@synopsys.com.
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at http://www.synopsys.com/support/support_ctr.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates command syntax. In command syntax and examples, shows system prompts, text from files, error messages, and reports printed by the system.
italic	Indicates a user specification, such as `object_name`
bold	In interactive dialogs, indicates user input (text you type).
[ ]	Denotes optional parameters, such as `pin1 [pin2 ... pinN]`
		Indicates a choice among alternatives, such as `low	medium	high` (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
_	Connects terms that are read as a single term by the system, such as `set_annotated_delay`
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

1

Introducing HDL Compiler for Verilog

The Synopsys HDL Compiler for Verilog tool (referred to as HDL Compiler) translates Verilog HDL descriptions into internal gate-level equivalents and optimizes them. The Synopsys Design Compiler products compile these representations to produce optimized gate-level designs in a given ASIC technology.

This chapter introduces the main concepts and capabilities of the HDL Compiler tool. It includes the following sections:

- What's New in This Release
- Hardware Description Languages
- HDL Compiler and the Design Process
- Using HDL Compiler With Design Compiler

- Design Methodology
- Verilog Example

What's New in This Release

Version 2000.05 of HDL Compiler includes solutions to Synopsys Technical Action Requests (STARs) filed in previous releases. Information about resolved STARs is available in the *HDL Compiler Release Note* in SolvNET.

To see the *HDL Compiler Release Note*,

1. Go to the Synopsys Web page at http://www.synopsys.com and click SolvNET.
2. If prompted, enter your user name and password. If you do not have a SOLV-IT! user name and password, you can obtain them at http://www.synopsys.com/registration.
3. Click Release Notes then open the *HDL Compiler Release Note*.

New Verilog Netlist Reader

The Verilog netlist reader incorporates algorithms that reduce the memory usage and CPU run time of the `read` command.

To use the new reader,

1. Set the following hidden variable (whose default is false) as shown:

```
enable_verilog_netlist_reader = true
```

2. Invoke the read command with the -netlist option as shown:

```
read -netlist -f verilog <file.v>
```

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 and then incrementally refine a design's detailed gate-level implementation.

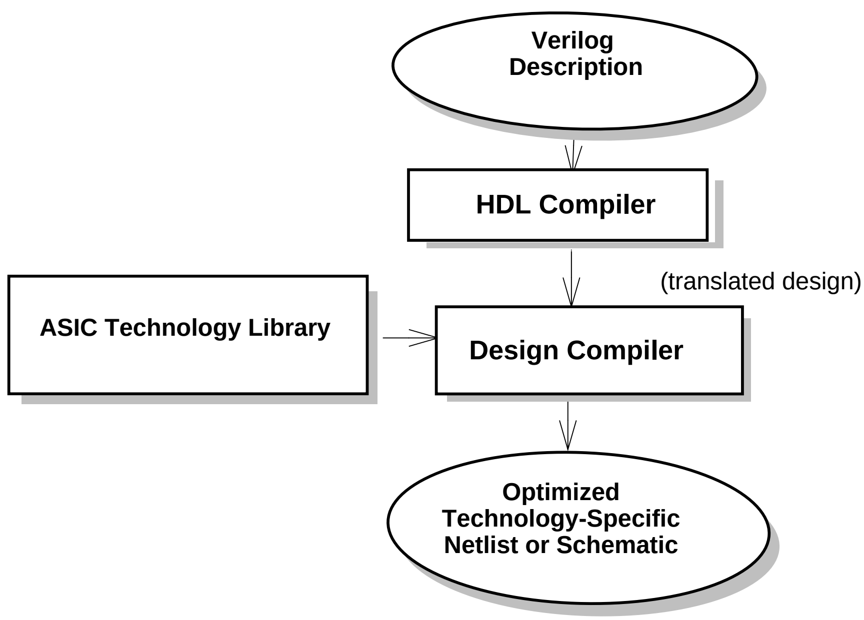
HDL descriptions play an important role in modern design methodology, for four main reasons:

- Verification of design functionality can happen 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.
- Coupling HDL Compiler with Synopsys logic synthesis, you can automatically convert an HDL description to a gate-level implementation in a target technology. This step eliminates the former gate-level design bottleneck, the majority of circuit design time, and the errors that occur when you hand-translate an HDL specification to gates.
- With Synopsys logic optimization, you can automatically transform a synthesized design into a smaller or faster circuit. Logic synthesis and optimization are provided by Synopsys Design Compiler.
- HDL descriptions provide technology-independent documentation of a design and its functionality. An HDL description is easier to read and understand than a netlist or a schematic description. Because the initial HDL design description is technology-independent, you can reuse it to generate the design in a different technology, without having to translate from the original technology.

HDL Compiler and the Design Process

HDL Compiler translates Verilog language hardware descriptions to the Synopsys internal design format. Design Compiler can then optimize the design and map it to a specific ASIC technology library, as Figure 1-1 shows.

Figure 1-1   HDL Compiler and Design Compiler



HDL Compiler supports a majority of the Verilog constructs. (For exceptions, see "Unsupported Verilog Language Constructs" on page B-21.)

Using HDL Compiler With Design Compiler

The process of reading a Verilog design into HDL Compiler involves converting the design to an internal database format so Design Compiler can synthesize and optimize the design. When Design Compiler optimizes a design, it might restructure part or all of the design. You control the degree of restructuring. Options include

- Fully preserving a design's hierarchy
- Allowing the movement of full modules up or down in the hierarchy
- Allowing the combination of certain modules with others
- Compressing the entire design into one module (called flattening the design)

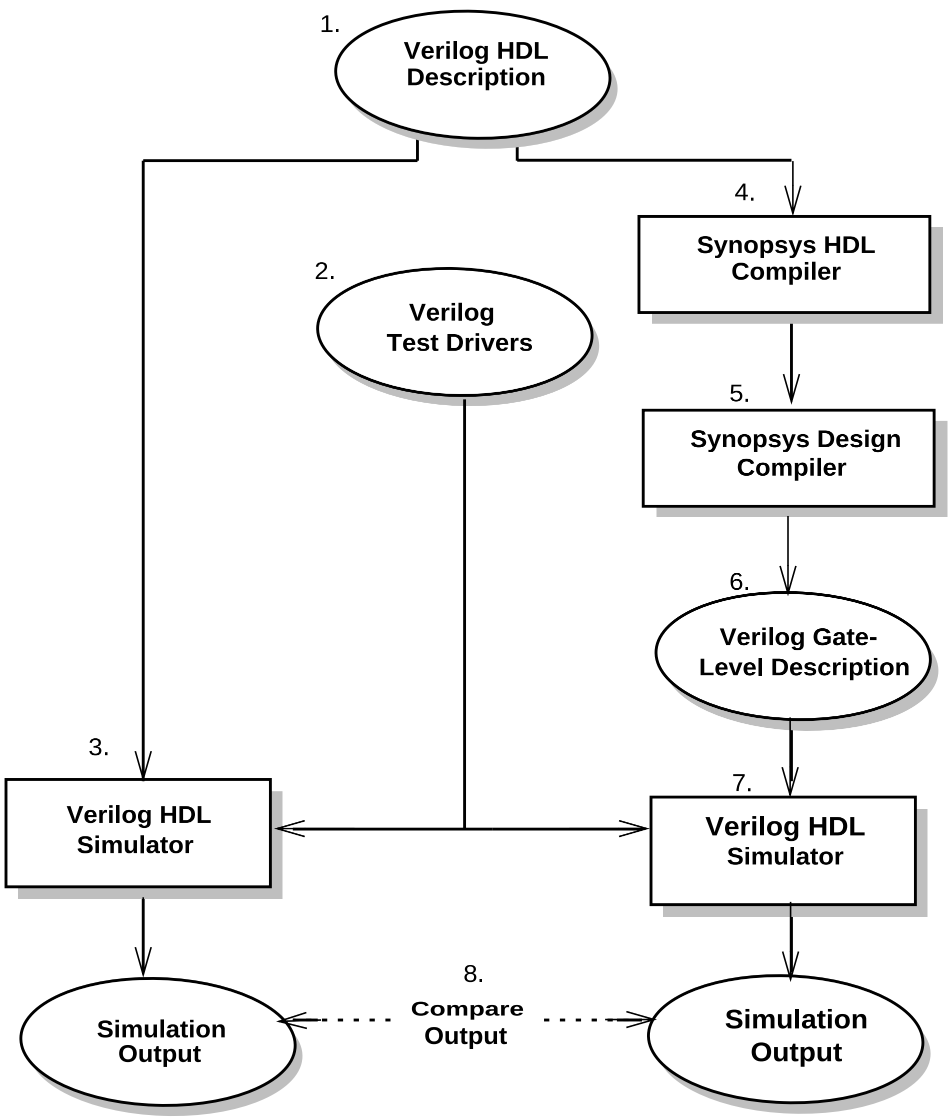
Synopsys Design Compiler can produce netlists and schematics in many commercial formats, including Verilog. It can convert existing gate-level netlists, sets of logic equations, or technology-specific circuits in another supported format to Verilog descriptions. The new Verilog descriptions document the original designs. In addition, a Verilog HDL Simulator can use the Verilog descriptions to provide circuit timing information.

The following section describes the design process that uses HDL Compiler and Design Compiler with a Verilog HDL Simulator.

Design Methodology

Figure 1-2 shows a typical design process that uses HDL Compiler, Design Compiler, and a Verilog HDL Simulator.

Figure 1-2 Design Flow



The steps in the design flow shown in Figure 1-2 are

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 for use with both Synopsys HDL Compiler and the 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. Translate the HDL description with HDL Compiler. HDL Compiler performs architectural optimizations and then creates an internal representation of the design.
5. Use Synopsys Design Compiler to produce an optimized gate-level description in the target ASIC library. You can optimize the generated circuits to meet the timing and area constraints wanted. This optimization step must follow the translation (step 4) to produce an efficient design.
6. Use Synopsys Design Compiler to output a Verilog gate-level description. This netlist-style description uses ASIC components as the leaf-level cells of the design. The gate-level description has the same port and module definitions as the original high-level Verilog description.
7. Pass the gate-level Verilog description from step 6 through the Verilog HDL simulator. You can use the original Verilog simulation test drivers from step 2, because module and port definitions are preserved through the translation and optimization processes.

8. Compare the output of the gate-level simulation (step 7) with 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 design session includes the following elements:

- A description of the design problem (count the 0s in a sequentially input 8-bit value)
- A listing of a Verilog design description
- A schematic of the synthesized circuit

  Note:

  The “Count Zeros—Sequential Version” example in this section is from Appendix A, “Examples.”

Verilog Design Description

The Count Zeros example illustrates a design that takes an 8-bit value and determines that the value has exactly one sequence of 0s and counts the 0s in that sequence.

A value is valid if it contains only one series of consecutive 0s. If more than one series appears, the value is invalid. A value consisting entirely of 1s is 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; the value `11000111` is valid and has three 0s; the value `00111100` is invalid, however.

The circuit accepts the 8-bit data value serially, 1 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`

True if the data is a valid value.

`data_ready`

True at the first invalid bit or when all 8 bits have been processed.

`zeros`

The number of 0s 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

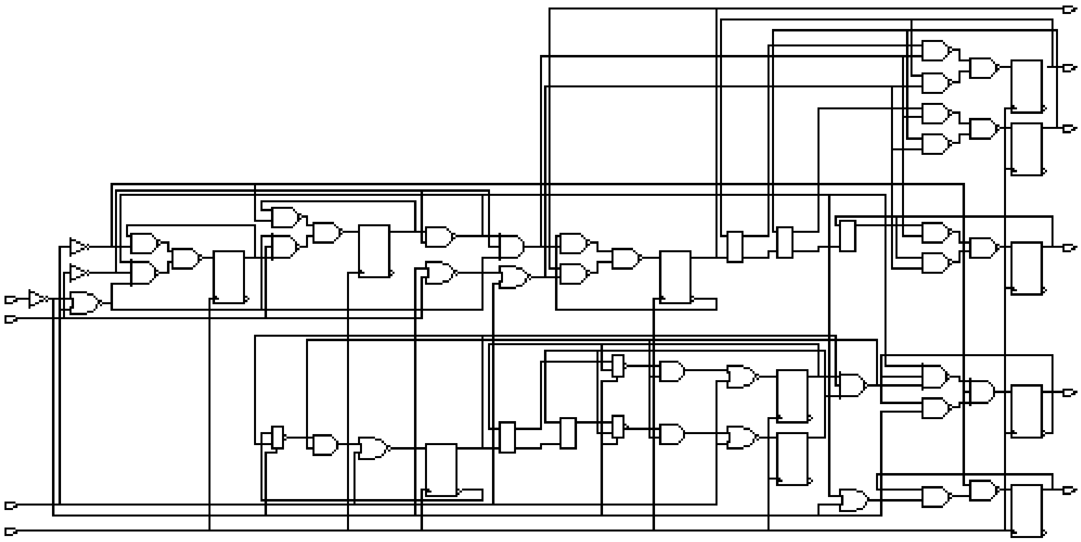
```
module count_zeros(data,reset,read,clk,zeros,is_legal,
data_ready);

    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 or zeros or bits_seen ) 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
```

Synthesizing the Verilog Design

Synthesis of the design description in Example 1-1 results in the circuit shown in Figure 1-3.

Figure 1-3   Count Zeros—Sequential Version



2

Description Styles

A Verilog circuit description can be one of two types: structural or functional. A structural description explains the physical makeup of the circuit, detailing gates and the connections between them. A functional description, also referred to as an RTL (register transfer level) description, describes what the circuit does.

This chapter covers the following topics:

- Design Hierarchy
- Structural Descriptions
- Functional Descriptions
- Mixing Structural and Functional Descriptions
- Design Constraints

- Register Selection
- Asynchronous Designs

Design Hierarchy

Synopsys HDL Compiler maintains the hierarchical boundaries you define when you use structural Verilog. These boundaries have two major effects:

- 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 Design Compiler.
- 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, “Structural Descriptions,” discusses modules and module instantiations.

Note:

HDL Compiler 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 reading in a Verilog design, you can use the `group -hdl_block` command to group the gates in a block, function, or task. For information on how to use the `group` command with Verilog designs, see the Synopsys `group` man page.

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

Structural Descriptions

The structural elements of a Verilog structural description are 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 in 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. See Chapter 3, “Structural Descriptions,” for more information.

Functional Descriptions

The functional elements of a Verilog description are function declarations, task statements, and always blocks. These elements describe the function of the circuit but do not describe its physical makeup or layout. The choice of gates and components is left entirely to Design Compiler.

You can construct functional descriptions with the Verilog functional constructs described in Chapter 5, “Functional Descriptions.” 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 (for example, for loops and multiple assignments to the same variable) appear sequential in nature, they describe combinational-logic networks. Other functional constructs imply sequential-logic networks. Latches and registers are inferred from these constructs. See Chapter 6, “Register, Multibit, Multiplexer, and Three-State Inference,” for details.

Mixing Structural and Functional Descriptions

When you use a functional description style in a design, you typically describe the combinational portions of the design 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 making this determination, 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.

You can specify elements of a design directly as module instantiations at the structural level. For example, see the three-state buffer `t1` in Example 2-1. (Note that three-states can be inferred. For more information, refer to "Three-State Inference" on page 6-73.) 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 that uses HDL Compiler, Design Compiler, and Verilog HDL Simulator. This process is described in Chapter 1, “Introducing HDL Compiler for Verilog.”

Description Style

Use the HDL design and coding style that makes the best use of the synthesis process to obtain high-quality results from HDL Compiler and Design Compiler. See Chapter 8, “Writing 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 uses HDL constructs that maximize coding flexibility while producing consistently good results. Although HDL Compiler 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 Language Constructs” on page B-21 lists these constructs.

Design Constraints

You can describe the area and performance constraints for a design module directly in your Verilog description. HDL Compiler inputs constraints specified for a design when they are embedded in a Synopsys-defined HDL Compiler directive. By specifying constraints with your HDL description,

- You can control the optimization of a design module from within the Verilog description. Design Compiler attempts to optimize each module so that all design constraints are met.
- You can use the Verilog description to document important specification information.

Chapter 9, “HDL Compiler Directives,” covers HDL Compiler directives in detail.

Register Selection

The clocking scheme and the placement of registers are important architectural factors. 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 ASIC 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 that

- 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) and then connect it to the rest of the design.

- The description is more difficult to write.

• You can use some Verilog constructs to direct HDL Compiler to infer registers from the description.

  The advantages to this approach directly counter the disadvantages of the previous approach. With register inference, the Verilog description is much easier to write and is technology-independent. This method allows Design Compiler to select the type of component inferred, based on constraints. Therefore, if a specific component is necessary, use instantiation. Some types of registers and latches cannot be inferred.

See “Register Inference” on page 6-2 for a discussion of latch and register inference.

Asynchronous Designs

You can use HDL Compiler to construct asynchronous designs that use multiple or gated clocks. However, although these designs are logically and statistically correct, they may not simulate or operate correctly, because of race conditions.

“Synthesis Issues” on page 8-36 describes how to write Verilog descriptions of asynchronous designs.

3

Structural Descriptions

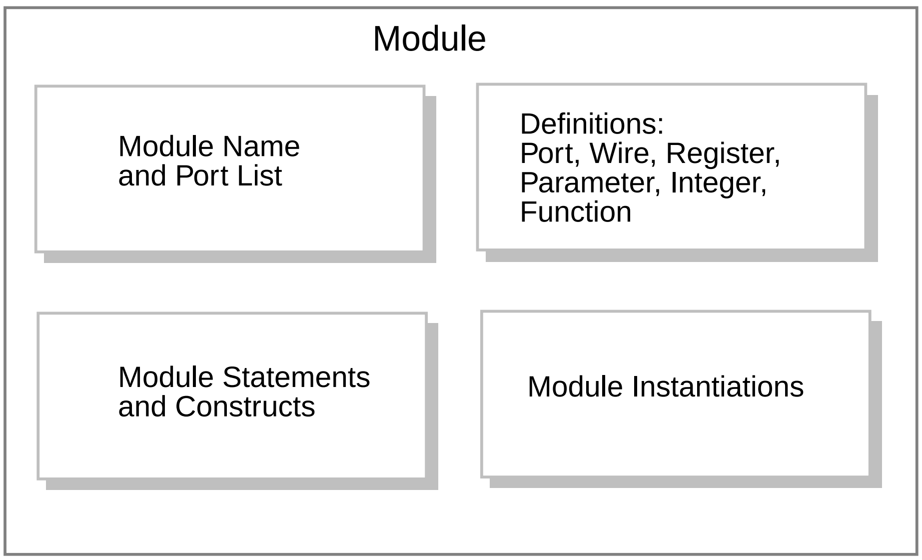
A Verilog structural description defines a connection of components that form a physical circuit. This chapter details the construction of structural descriptions, in the following major sections:

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

Modules

The principal design entity in the Verilog language is the 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



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 gives 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.)

The fourth line of the description creates a `wire` variable. The next two lines instantiate the two components, creating copies named `instance1` and `instance2` of the components `AND` and `INV`. These components connect to the ports of the module and are finally connected by use of 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
```

Macromodules

The macromodule construct makes simulation more efficient, by merging the macromodule definition with the definition of the calling (parent) module. However, HDL Compiler treats the macromodule construct as a module construct. Whether you use module or macromodule, the synthesis process, the hierarchy that synthesis creates, and its 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 Design Compiler instantiates a macromodule, a new level of hierarchy is created. To eliminate this new level of hierarchy, use the `ungroup` command. See the *Design Compiler User Guide* for information on the `ungroup` command.

Port Definitions

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

```
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, see “Concatenation Operators” on page 4-13.

Declare each port in a port list as input, output, or bidirectional in the module by use of an `input`, `output`, or `inout` statement. (See “Concatenation Operators” on page 4-13.) For example, the module definition in Example 3-1 on page 3-3 shows that module NAND has three ports: a, b, and z, connected to 1-bit nets a, b, and z. Declare these connections 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, a 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`, named a, b, and z, 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, unnamed, 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
```

Renaming Ports

You can rename a port by explicitly assigning a name to a port expression by using 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` and 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 Renaming 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

The Synopsys HDL Compiler tool 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 in “Module Instantiations” on page 3-16. Function definitions, always blocks, and task statements 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 most significant bit (`msb`) and least significant bit (`lsb`) must be nonnegative constant-valued expressions. Constant-valued expressions are composed only of constants, Verilog parameters, and operators.

parameter

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 constructions of different logic. For more information, see “Parameterized Designs” on page 3-19.

A parameter 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 integer or Boolean expression. 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. See “Constant-Valued Expressions” on page 4-2 for a discussion of constant formats.

You can use a parameter wherever a number is allowed, except when declaring the number of bits in an assignment statement, which will generate a syntax error as shown in Example 3-5.

Example 3-5 parameter Declaration Syntax Error

```
parameter size = 4;
assign out = in ? 4’b0000 : size’b0101;  // syntax error
```

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 3-6 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 of 3, 1, 0, and 2, respectively, and are stored as 2-bit quantities.

Example 3-6 parameter Declarations

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

wire

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 value from a `wire` from within a function or a `begin...end` block, but you cannot assign a value to a `wire` 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, HDL Compiler leaves undriven `wires` unconnected. Multiple connections or assignments to a `wire` simply short the `wires` together.

In Example 3-7, two `wires` are declared: a is a single-bit `wire`, and b is a 3-bit vector of `wires`. Its most significant bit (`msb`) has an index of 2, and its least significant bit (`lsb`) has an index of `0`.

Example 3-7 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. HDL Compiler accepts the syntax of these constructs, but they are ignored when the circuit is synthesized.

Note:

> You can use delay information for modeling, but Design Compiler ignores delay information. If the functionality of your circuit depends on the delay information, Design Compiler might create logic whose behavior does not agree with the behavior of the simulated circuit.

wand

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

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

Example 3-8   wand (wired-AND)

```
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. HDL Compiler accepts the syntax of these constructs but ignores the constructs during synthesis of the circuit.

wor

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

In Example 3-9, two variables drive the variable `c`. The value of `c` is determined by the logical `OR` of `a` and `b`.

Example 3-9   wor (wired-OR)

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

     wor c;

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

tri

The `tri` (`three-state`) data type is a specific type of wire. All variables that drive the `tri` must have a value of `Z` (high-impedance), except one. This single variable determines the value of the `tri`.

Note:

> HDL Compiler does not enforce the previous condition. You must ensure that no more than one variable driving a `tri` has a value other than `Z`.

In Example 3-10, three variables drive the variable `out`.

Example 3-10   tri (Three-State)

```
module tri_test (out, condition);
        input [1:0] condition;
        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 and supply1

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 3-11, power is tied to logic 1 and gnd (ground) is tied to logic 0.

Example 3-11 supply0 and supply1 Constructs

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

reg

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 and least significant bit of the vector. Both must be nonnegative constants, parameters, or constant-valued expressions. Example 3-12 shows some reg declarations.

Example 3-12 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 (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

You declare all input ports of a module 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 in the case of input b in the following example. The `input` statements can appear in any order in the description, but you must declare them before using them. For example,

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

output

You declare all output ports of a module 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 the statement before you use it.

You can use a range specification to declare an output 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

You can declare bidirectional ports with the `inout` statement. An inout is a type of `wire` and is governed by the syntax of `wire`. HDL Compiler 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 3-13 shows two equivalent methods for specifying a continuous assignment for `wire` a.

Example 3-13 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. You cannot assign a value to a `reg` in a continuous assignment.

Verilog allows you to assign drive strength for each continuous assignment statement. HDL Compiler 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 done bitwise, 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 in a module that defines component interconnections.

```
module_name instance_name1 (terminal, terminal, ...),
            instance_name2 (terminal, terminal, ...);
```

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. Module instantiations have this syntax:

```
(terminal1, terminal2, ...),
(terminal1, terminal2, ...);
```

Terminals connected to input ports can be any arbitrary expression. Terminals connected to output and inout ports can be identifiers, single- 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 3-14 shows the declaration for the module SEQ with two instantiations (SEQ_1 and SEQ_2).

Example 3-14 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 each 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 3-14, `SEQ_2` is instantiated by the use of 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 the use of 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.

Parameterized Designs

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

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

HDL Compiler does not support the defparam statement but does support the previous syntax.

The module in Example 3-15 contains a parameter declaration.

Example 3-15 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 3-15, 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.

Parameterized designs are read into dc_shell as templates with the `read` command. 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 will be read in as a template, in either of two ways:

- Add the pseudocomment `// synopsys template` to your code.
- Set the dc_shell variable `hdlin_auto_save_templates = true`.

  Note:

  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 the template in your Verilog code. Example 3-16 shows how to do this.

Example 3-16 Instantiating a Parameterized Design in 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 3-16 instantiates the parameterized design foo, which has one parameter, 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 Design Compiler links module `param`, it searches for template `foo` in memory. If `foo` is found, it is automatically built with the specified parameters. HDL Compiler 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.

Another way to build a parameterized design is with the `elaborate` command in dc_shell. The syntax of the command is

`elaborate` *`template_name`* `-parameters` *`parameterized`*

Using Templates—Naming

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"`

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.

`template_parameter_style = "%s%d"`

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.

`template_separator_style = "_"`

The `template_separator_style` variable contains a string that separates parameter names. This variable is used only for templates that have 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` has parameters `N`, `M`, and `Z`. You can build a design where `N = 8`, `M = 6`, and `Z` is left at its 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.

Using Templates—list -templates Command

To see which templates are available, use the `list -templates` command. The `report_templates` command lists all templates that reside in memory and the parameters you can select for each. The `remove_template` command deletes a template from memory.

Gate-Level Modeling

Verilog provides several 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. HDL Compiler 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`, `not`, and `tran` 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 Design Compiler ignores them. Example 3-17 shows two gate-level instantiations.

Example 3-17 Gate-Level Instantiations

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

Note:

HDL Compiler parses but ignores delay options for gate primitives. Because Design Compiler ignores the delay information, it can create logic whose behavior does not agree with the simulated behavior of the circuit. See “D Flip-Flop With Asynchronous Set or Reset” on page 6-28.

Three-State Buffer Instantiation

HDL Compiler 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 3-18 shows a three-state gate instantiation with an active-high enable and no inverted output.

Example 3-18 Three-State Gate Instantiation

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

    bufif1 (out1,in1,cntrl1);

endmodule
```

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, using

- 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. HDL Compiler 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
// performs  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 in an expression influence the way a design is synthesized. HDL Compiler 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. HDL Compiler does synthesize circuitry for an expression that contains variables, however.

Operators

Operators identify the operation to be performed on their operands to produce a new value. Most operators are either unary operators, which apply to only one operand, or binary operators, which apply to two operands. Two exceptions are conditional operators, which take three operands, and concatenation operators, which take any number of operands.

HDL Compiler supports the types of operations listed in Table 4-1, which also lists the Verilog language operators HDL Compiler supports. A description of the operators and their order of precedence appears in the sections that follow the table.

Table 4-1 Verilog Operators Supported by HDL Compiler

Operator type	Operator	Description
Arithmetic operators	+ – * /	Arithmetic
	%	Modules
Relational operators	> >= < <=	Relational
Equality operators	==	Logical equality
	!=	Logical inequality
Logical operators	!	Logical NOT
	&&	Logical AND
	||	Logical OR
Bitwise operators	~	Bitwise NOT
	&	Bitwise AND
	|	Bitwise OR

Table 4-1 Verilog Operators Supported by HDL Compiler (continued)

Operator type	Operator	Description
	^	Bitwise XOR
	^~ ~^	Bitwise XNOR
Reduction operators	&	Reduction AND
	|	Reduction OR
	~&	Reduction NAND
	~|	Reduction NOR
	^	Reduction XOR
	~^ ^~	Reduction XNOR
Shift operators	<<	Shift left
	>>	Shift right
Conditional operator	? :	Conditions
Concatenation operator	{ }	Concatenation

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 (/)
- Modules (%)

You can use the +, –, and * operators with any operand form (constants or variables). The + and – operators can be used as either unary or binary operators. HDL Compiler requires that the / and % 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 bitwise 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 by bit.

The Verilog equality operators are

- Equality (==)
- Inequality (`!=`)

Example 4-4 shows the equality operator testing 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

HDL Compiler always ignores comparisons to an X or a Z. If your code contains a comparison to an X or a Z, a warning message displays, indicating that the comparison is always evaluated to false, which might cause simulation to disagree with synthesis.

Example 4-5 shows code from a file called test2.v. HDL Compiler always assigns the variable B 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 HDL Compiler reads this code, it generates the following warning message:

```
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 alternative method of handling comparisons to X or Z, use the translate_off and translate_on directives to comment out the condition and its first branch (the true clause) so that only the else branch goes through 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);

    parameterIMMEDIATE=2'b00, DIRECT=2'b01;
    parameterSUBA_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
```

Bitwise Operators

Bitwise operators act on the operand bit by bit. The Verilog bitwise operators are

- Unary negation (~)
- Binary `AND` (&)
- Binary `OR` (|)
- Binary `XOR` (^)
- Binary `XNOR` (`^~ or ~^`)

Example 4-7 shows some bitwise operators.

Example 4-7 Bitwise 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

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.

The Verilog shift operators are

- Shift left (<<)
- Shift right (>>)

After the shift, vacated bits fill with zeros. Shifting by a constant results in minor circuitry modification (because only rewiring is required). Shifting by a variable causes a general shifter to be synthesized. Example 4-9 shows use of a shift-right operator to perform 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 Operator

The conditional operator (`? :`) evaluates an expression and returns a value that is based on the truth of the expression.

Example 4-10 shows how to use the conditional operator. 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
```

You can nest conditional operators 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 );

    parameterADD=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
```

Concatenation Operators

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

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>[ ]</td><td>Bit-select or part-select</td></tr>
<tr><td>( )</td><td>Parentheses</td></tr>
<tr><td>! ~</td><td>Logical and bitwise negation</td></tr>
<tr><td>& | ~& ~| ^ ~^ ^~</td><td>Reduction operators</td></tr>
<tr><td>+ –</td><td>Unary arithmetic</td></tr>
<tr><td>{ }</td><td>Concatenation</td></tr>
<tr><td>* / %</td><td>Arithmetic</td></tr>
<tr><td>+ -</td><td>Arithmetic</td></tr>
<tr><td><< >></td><td>Shift</td></tr>
<tr><td>> >= < <=</td><td>Relational</td></tr>
<tr><td>== !=</td><td>Logical equality and inequality</td></tr>
<tr><td>&</td><td>Bitwise AND</td></tr>
<tr><td>^ ^~ ~^</td><td>Bitwise XOR and XNOR</td></tr>
<tr><td>|</td><td>Bitwise OR</td></tr>
<tr><td>&&</td><td>Logical AND</td></tr>
<tr><td>||</td><td>Logical OR</td></tr>
<tr><td>? :</td><td>Conditional</td></tr>
</table>

Operands

You can use the following kinds of operands in an expression:

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

The following sections explain each of these operands.

Numbers

A number is either a constant value or a value specified as a parameter. The expression `size-1` in Example 4-1 on page 4-2 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. See “Numbers” on page B-14 for a discussion of the number format.

Wires and Registers

Variables that represent wires as well as registers are allowed in an expression. If the variable is a multiple-bit 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.

Wires are described in “Module Statements and Constructs” on page 3-6, and registers are described in “Function Declarations” on page 5-3.

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 evaluation of 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 or descending indexes, the part-select (when applied to that variable) must be in the same order.

You can also write the expression in Example 4-14 on page 4-18 with part-select operands, 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 1 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 uses the function call `legal` in an expression.

Example 4-17 Function Call Used as an Operand

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

Functions are described in “Function Declarations” on page 5-3.

Concatenation of Operands

Concatenation is the process of combining several single- or multiple-bit operands into one large bit vector. The use of the concatenation operator, a pair of braces (`{}`), is described in “Concatenation Operators” on page 4-13.

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 appear in Table 4-3.

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.

Table 4-3 Expression Bit-Widths

Expression	Bit length	Comments
unsized constant	32 bits	Self-determined
sized constant	as specified	Self-determined
i + j	max(L(i),L(j))	Context-determined

Table 4-3 Expression Bit-Widths (continued)

Expression	**Bit length**	**Comments**
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
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

Table 4-3 Expression Bit-Widths (continued)

Expression	Bit length	Comments
~|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}	L(i)+...+L(j)	Self-determined
{i {j,...,k}}	i*(L(j)+...+L(k))	Self-determined
i ? j : k	Max(L(j),L(k))	i is self-determined

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 4 bits and a value of `4’b1111`. The resulting width of the concatenation is 8 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 the 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 some 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 1-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 in 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`).

5

Functional Descriptions

A Verilog functional description defines a circuit in terms of what it does.

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

- Sequential Constructs
- Function Declarations
- Function Statements
- task Statements
- always Blocks

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;
```

HDL Compiler determines the combinational equivalent of this description. In fact, it treats the statements in Example 5-1 exactly as 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 outputs you want. For example, suppose the addition 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, HDL Compiler 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

Using a function declaration is one of three methods for describing combinational logic. The other two methods are to use the `always` block, described in “always Blocks” on page 5-33, and to use the continuous assignment, described in “Continuous Assignment” on page 3-15. 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.

HDL Compiler supports the following Verilog function declarations:

- Input declarations
- Output from a function
- Register 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 the following syntax shows.

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

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

The function statements HDL Compiler supports are discussed in "Function Statements" on page 5-9.

Input Declarations

The 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:

```
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.

Output From a Function

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.

Register Declarations

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

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

A `reg` can be a single-bit quantity or a vector of bits. The *range* specifies the most significant bit (`msb`) and the least significant bit (`lsb`) of the vector enclosed in square brackets ([ ]). Both bits must be nonnegative constants, parameters, or constant-valued expressions. 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` can hold its value across separate calls to a function. In some cases, HDL Compiler emulates this behavior by inserting flow-through latches. In other cases, it emulates this behavior without a latch. The concept of holding state is elaborated on in "Inferring Latches" on page 6-10 and in several examples in Appendix A, "Examples."

Memory Declarations

The memory declaration models a bank of registers or memory. In Verilog, the memory declaration is 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 8 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 being local to a function. However, you cannot use a local variable outside that function. Parameter declarations in a function are identical to parameter declarations in a module. 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 integers is 32 bits. HDL Compiler determines bit-widths, except in the case of a don't care condition resulting during 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 HDL Compiler supports 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 “Continuous Assignment” on page 3-15, 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 “Structural Data Types” on page 3-7, 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 (must be constant-valued)
- Integers
- Concatenations of the previous data types

HDL Compiler assigns the low bit on the right side 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. HDL Compiler allows multiple procedural assignments.

Example 5-10 shows some examples of procedural assignments.

Example 5-10 Procedural Assignments

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

RTL Assignments

HDL Compiler handles variables driven by an RTL (nonblocking) assignment differently than those driven by a procedural (blocking) assignment.

In procedural assignments, a value passed along from variable A to variable B to variable C results in all three variables having the same value in every clock cycle. In the netlist, procedural assignments are indicated when the input net of one flip-flop is connected to the input net of another flip-flop. Both flip-flops input the same value in the same clock cycle.

In RTL assignments, however, values are passed on in the next clock cycle. Assignment from variable A to variable B occurs after one clock cycle, if variable A has been a previous target of an RTL assignment. Assignment from variable B to variable C always takes place after one clock cycle, because B is the target when RTL assigns variable A’s value to B. In the netlist, an RTL assignment shows flip-flop B receiving its input from the output net of flip-flop A. It takes one clock cycle for the value held by flip-flop A to propagate to flip-flop B.

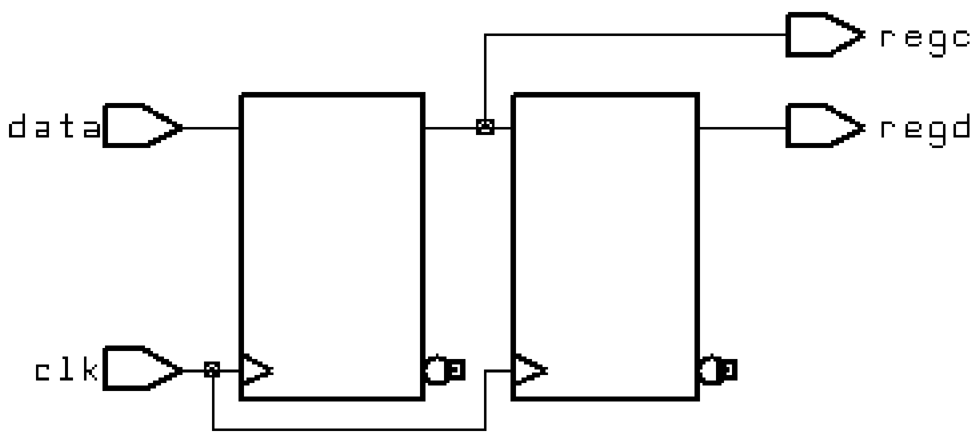
A variable can follow only one assignment method and therefore cannot be the target of RTL as well as procedural assignments.

Example 5-11 is a description of a serial register implemented with RTL assignments. Figure 5-1 shows the resulting schematic for Example 5-11.

Example 5-11 RTL Nonblocking Assignments

```
module rtl (clk, data, regc, regd);
input data, clk;
output regc, regd;

reg regc, regd;

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

Figure 5-1 Schematic of RTL Nonblocking Assignments

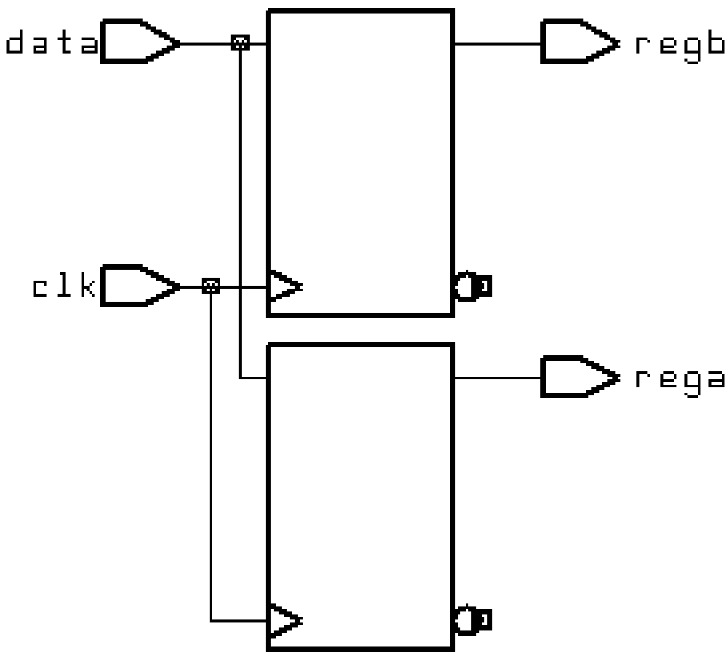


If you use a procedural assignment, as in Example 5-12, HDL Compiler does not synthesize a serial register. Therefore, the recently assigned value of `rega`, which is data, is assigned to `regb`, as the schematic in Figure 5-2 indicates.

Example 5-12   Blocking Assignment

```
module rtl (clk, data, rega, regb);
input data, clk;
output rega, regb;

reg rega, regb;

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

Figure 5-2   Schematic of Blocking Assignment



begin...end Block Statements

Using block statements is 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` pairs are commonly used in conjunction with `if`, `case`, and `for` statements to group several statements. Functions and `always` blocks that contain more than one statement require a `begin...end` pair to group the statements. Verilog also provides a construct called a named block, as 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` 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

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

The syntax of `if...else` statements is

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

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

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

The `if...else` statements can cause synthesis of registers. Registers are synthesized when you do not assign a value to the same `reg` in all branches of a conditional construct. Information on registers is in “Register Inference” on page 6-2.

HDL Compiler 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 place 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

HDL Compiler can synthesize a latch for a conditionally assigned variable. A variable is conditionally assigned if there is a path that does not explicitly assign a value to that variable.

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 in the subsequent sections on loops.

The syntax for a case statement is

```
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 with 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

HDL Compiler automatically determines whether a `case` statement is full or parallel. A `case` statement is full 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, HDL Compiler synthesizes a latch. See “parallel_case Directive” on page 9-8 and “full_case Directive” on page 9-10 for more information.

HDL Compiler synthesizes optimal logic for the control signals of a `case` statement. If HDL Compiler cannot determine that branches are parallel, it synthesizes hardware that includes a priority encoder. If HDL Compiler can determine that no cases overlap (parallel case), it synthesizes a multiplexer, because a priority encoder is not necessary. You can also declare a `case` statement as parallel case with the `//synopsys parallel_case` directive. See “full_case Directive” on page 9-10. 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 values of inputs `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 allows a multipath branch in logic, according to the value of an expression, just as the `case` statement does. 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

```
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, it means that the corresponding bit of the `casex` expression is not compared. Example 5-22 shows a case item that includes an `x`.

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
```

HDL Compiler allows `?`, `z`, and `x` bits in case items but not in *casex* expressions. Example 5-25 shows an invalid *casex* expression.

Example 5-25 Invalid casex Expression

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

casez Statements

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 `casex`, except that `x` is not allowed in case items; only `z` and `?` are accepted as special characters.

The syntax for a `casez` statement is

```
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 z or ?

When a `casez` statement is evaluated, the value z in the case item 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
```

HDL Compiler allows ? and z bits in case items but not in `casez` expressions. Example 5-27 shows an invalid expression in a `casez` statement.

Example 5-27 Invalid 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 appear in each `for` loop: `low_range` and `high_range`. In the syntax lines that follow, `high_range` is greater than or equal to `low_range`. HDL Compiler recognizes incrementing as well as decrementing loops. The statement to be duplicated is surrounded by `begin` and `end` statements.

Note:

HDL Compiler 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
```

The `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)
```

HDL Compiler supports `while` loops if you insert one of these 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 “Register Inference” on page 6-2 for more information on synchronous resets.

Using the style illustrated in Example 5-34 is not a good idea, because you cannot test it. The synthesized state machine does not reset to a known state; therefore, it is impossible to create a test program for it. Example 5-36 on page 5-31 illustrates how a synchronous reset for the state machine can be synthesized.

disable Statements

HDL Compiler 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
```

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: Block
        @ (posedge clk)
        if (Reset)
            begin
                z <= 1’b0;
                disable Block;
            end
                z <= a;
    end
```

The `disable` statement in Example 5-36 causes the block `Block` to terminate immediately and return to the beginning of the block.

task Statements

In Verilog, `task` statements are similar to functions, but `task` statements can have output and inout ports. You can use the `task` statement to structure your Verilog code so that a portion of code is reusable.

In Verilog, tasks can have timing controls and can take a nonzero time to return. However, HDL Compiler ignores all timing controls, so synthesis might disagree with simulation if timing controls are critical to the function of the circuit.

Example 5-37 shows how a `task` statement 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
```

Event Expression

The event expression declares the triggers or timing controls. The word *or* groups several triggers. 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, HDL Compiler ignores the event expression unless it is a synchronous trigger that infers a register. See Chapter 6, “Register, Multibit, Multiplexer, and Three-State Inference,” for details.

Example 5-38 shows a simple example of an always block with triggers.

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`. HDL Compiler 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, HDL Compiler 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, HDL Compiler does not require listing of all variables.

Any of the following types of event expressions can trigger an `always` block:

- A change in a specified value. For example,

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

  In the previous example, HDL Compiler 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.

Note:

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

Incomplete Event Specification

You risk misinterpretation of an `always` block if you do not list all the signals entering an `always` block in the event specification. Example 5-39 shows an incomplete event list.

Example 5-39 Incomplete Event List

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

HDL Compiler builds a 3-input AND gate for the description in Example 5-39, but in simulation of this description, `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 the signals in the 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 the signals in the event specification. Example 5-41 illustrates this problem.

Example 5-41 Incomplete Event List for Asynchronous Preload

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

In the logic synthesized for Example 5-41, if 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 in HDL Compiler only when you want changes in the load data to be reflected immediately 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, HDL Compiler prints a warning similar to the following:

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

6

Register, Multibit, Multiplexer, and Three-State Inference

HDL Compiler can infer registers (latches and flip-flops), multiplexers, and three-state cells. This chapter explains inference behavior and results, in the following sections:

- Register Inference
- Multibit Inference
- Multiplexer Inference
- Three-State Inference

Register Inference

Register inference allows you to use sequential logic in your designs and keep your designs technology-independent. A register is a simple, 1-bit memory device, either a latch or a flip-flop. A latch is a level-sensitive memory device. A flip-flop is an edge-triggered memory device.

The register inference capability can support coding styles other than those described in this chapter. However, for best results,

- Restrict each `always` block to a single type of memory-element inferencing:
  - Latch
  - Latch with asynchronous set or reset
  - Flip-flop
  - Flip-flop with asynchronous reset
  - Flip-flop with synchronous reset
- Use the templates provided in “Inferring Latches” on page 6-10 and “Inferring Flip-Flops” on page 6-25.

Reporting Register Inference

HDL Compiler provides the following controls for reporting register inference:

- Configuring the inference report
- Selecting the latch inference warnings

The following sections describe these controls.

Configuring the Inference Report

HDL Compiler can generate an inference report that shows the information HDL Compiler passes on to Design Compiler about the inferred devices. Use the following variables to configure an inference report:

`hdlin_report_inferred_modules = true`

This variable controls the generation of the inference report. You can select from the following settings for this variable:

`false`
HDL Compiler does not generate an inference report.

`true`
HDL Compiler generates a general inference report when building a design. This is the default setting. Example 6-1 shows a general inference report for a JK flip-flop.

`verbose`
HDL Compiler generates a verbose inference report when building a design. It provides the asynchronous set or reset, synchronous set or reset, and synchronous toggle conditions of each latch or flip-flop, expressed as Boolean formulas. Example 6-2 shows a verbose inference report for a JK flip-flop.

`hdlin_reg_report_length = 60`

This variable indicates the length of the Boolean formulas reported in the verbose inference report. You must specify an integer value for this variable. The default setting is 60.

Example 6-1 General Inference Report for a JK Flip-Flop

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

Example 6-2 Verbose Inference Report for a JK Flip-Flop

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

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

In the inference reports in Example 6-1 and Example 6-2,

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

In the verbose inference report (Example 6-2), the last part of the report lists the objects that control the synchronous reset and set conditions. In this example, a synchronous reset occurs when J is low (logic 0) and K is high (logic 1). The last line of the report indicates the register output value when both set and reset are active:

zero(0)

Indicates that the reset has priority and that the output goes to logic 0.

one (1)

Indicates that the set has priority and that the output goes to logic 1.

X

Indicates that there is no priority and that the output is unstable.

“Inferring Latches” on page 6-10 and “Inferring Flip-Flops” on page 6-25 provide inference reports for each register template. After you input a Verilog description, check the inference report to verify that HDL Compiler passes the correct information to Design Compiler.

Selecting Latch Inference Warnings

Use the `hdlin_check_no_latch` variable to control whether HDL Compiler generates warning messages when inferring latches.

If `hdlin_check_no_latch` is set true, HDL Compiler generates a warning message when it infers a latch. This is useful for verifying that a combinational design does not contain memory components. The default setting of the `hdlin_check_no_latch` variable is false.

Controlling Register Inference

Use HDL Compiler directives or dc_shell variables to direct HDL Compiler to the type of sequential device you want inferred. HDL Compiler directives give you control over individual signals, and dc_shell variables apply to an entire design.

Attributes That Control Register Inference

HDL Compiler provides the following directives for controlling register inference:

`async_set_reset`

When a signal has this directive set to true, HDL Compiler searches for a branch that uses the signal as a condition. HDL Compiler then checks whether the branch contains an assignment to a constant value. If the branch does, the signal becomes an asynchronous reset or set.

Attach this directive to single-bit signals, using the following syntax:

```
// synopsys async_set_reset ”signal_name_list”
```

`async_set_reset_local`

HDL Compiler treats listed signals in the specified block as if they have the `async_set_reset` directive set to true.

Attach this directive to a block label, using the following syntax:

```
/* synopsys async_set_reset_local block_label
   ”signal_name_list” */
```

`async_set_reset_local_all`

HDL Compiler treats all signals in the specified blocks as if they have the `async_set_reset` directive set to true.

Attach this directive to block labels, using the following syntax:

```
/* synopsys async_set_reset_local_all
   ”block_label_list” */
```

`sync_set_reset`

When a signal has this directive set to true, HDL Compiler checks the signal to determine whether it synchronously sets or resets a register in the design.

Attach this directive to single-bit signals, using the following syntax:

```
//synopsys sync_set_reset ”signal_name_list”
```

`sync_set_reset_local`

HDL Compiler treats listed signals, in the specified block as if they have the `sync_set_reset` directive set to true.

Attach this directive to a block label, using the following syntax:

```
/* synopsys sync_set_reset_local block_label
   ”signal_name_list” */
```

`sync_set_reset_local_all`

HDL Compiler treats all signals in the specified blocks as if they have the `sync_set_reset` directive set to true.

Attach this directive to block labels, using the following syntax:

```
/* synopsys sync_set_reset_local_all
   ”block_label_list” */
```

`one_cold`

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

Add a check to the Verilog code to ensure that the group of signals has a one-cold implementation. HDL Compiler does not produce any logic to check this assertion.

Attach this directive to set or reset signals on sequential devices, using the following syntax:

```
// synopsys one_cold ”signal_name_list”
```

`one_hot`

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

Add a check to the Verilog code to ensure that the group of signals has a one-hot implementation. HDL Compiler does not produce any logic to check this assertion.

Attach this directive to set or reset signals on sequential devices, using the following syntax:

```
// synopsys one_hot ”signal_name_list”
```

Variables That Control Register Inference

You can use the following dc_shell variables to control register inference:

`hdlin_ff_always_async_set_reset = true`

When this variable is true, HDL Compiler automatically checks for asynchronous set and reset conditions of flip-flops.

`hdlin_ff_always_sync_set_reset = false`

When this variable is true, HDL Compiler automatically checks for synchronous set and reset conditions of flip-flops.

`hdlin_latch_always_async_set_reset = false`

When this variable is true, HDL Compiler automatically checks for asynchronous set and reset conditions of latches. When this variable is false, HDL Compiler interprets each control object of a latch as synchronous.

Setting the variable to true is equivalent to specifying every object in the design in the object list for the `async_set_reset` directive. When true 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. HDL Compiler does not limit checks for assignments to a constant 0 or constant 1 to a single block. That is, HDL Compiler performs checking across blocks.

`hdlin_keep_feedback = false`

When this variable is false, HDL Compiler removes all flip-flop feedback loops. For example, HDL Compiler removes feedback loops inferred from a statement such as `Q=Q`. Removing the state feedback from a simple D flip-flop creates a synchronous loaded flip-flop. Set this variable to true if you want to keep feedback loops.

`hdlin_keep_inv_feedback = true`

When this variable is false, HDL Compiler removes all inverted flip-flop feedback loops. For example, HDL Compiler removes feedback loops inferred from a statement such as `Q=Q`. Removing the inverted feedback from a simple D flip-flop creates a toggle flip-flop. Set this variable to true if you want to keep feedback loops.

Inferring Latches

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

- SR latch
- D latch
- Master-slave latch

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

Inferring SR Latches

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

Example 6-3 shows the Verilog code that implements the inferred SR latch shown in Figure 6-1 on page 6-12 and described in Table 6-1 on page 6-11. Because the output y is unstable when both inputs have a logic 0 value, you might want to include a check in the Verilog code to detect this condition during simulation. Synthesis does not support such checks, so you must put the `translate_off` and `translate_on` directives around the check. See “translate_off and translate_on Directives” on page 9-6 for more information about special comments in the Verilog source code.

Example 6-4 shows the inference report HDL Compiler generates.

Table 6-1 SR Latch Truth Table (NAND Type)

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

Example 6-3 SR Latch

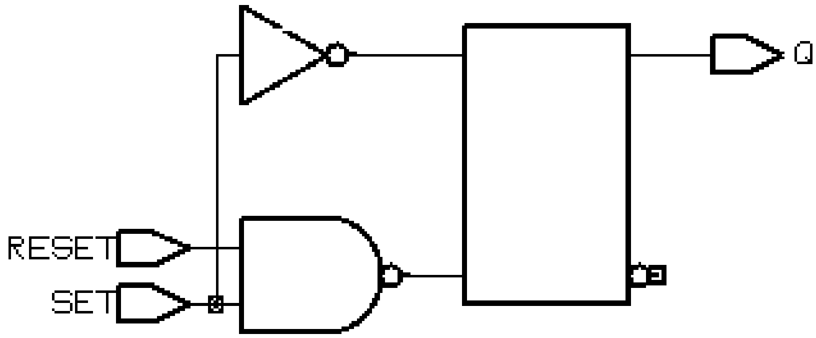
```
module sr_latch (SET, RESET, Q);
  input SET, RESET;
  output Q;
  reg Q;

//synopsys async_set_reset ”SET, RESET”
always @(RESET or SET)
  if (~RESET)
    Q = 0;
  else if (~SET)
    Q = 1;
endmodule
```

Example 6-4 Inference Report for an SR Latch

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

```
Q_reg
Async-reset: RESET’
Async-set: SET’
Async-set and Async-reset ==> Q: 1
```

Figure 6-1 SR Latch



Inferring D Latches

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

For example, the `if` statement in Example 6-5 infers a D latch because there is no `else` clause. The Verilog code specifies a value for output `Q` only when input enable has a logic 1 value. As a result, output `Q` becomes a latched value.

Example 6-5 Latch Inference Using an if Statement

```
always @ (DATA or GATE) begin
  if (GATE) begin
    Q = DATA;
  end
end
```

The `case` statement in Example 6-6 infers D latches, because the `case` statement does not provide assignments to decimal for values of I between 10 and 15.

Example 6-6 Latch Inference Using a case Statement

```
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
```

To avoid latch inference, assign a value to the signal under all conditions. To avoid latch inference by the `if` statement in Example 6-5, modify the block as shown in Example 6-7 or Example 6-8. To avoid latch inference by the `case` statement in Example 6-6, add the following statement before the `endcase` statement:

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

Example 6-7 Avoiding Latch Inference

```
always @ (DATA, GATE) begin
  Q = 0;
  if (GATE)
    Q = DATA;
end
```

Example 6-8 Another Way to Avoid Latch Inference

```
always @ (DATA, GATE) begin
  if (GATE)
    Q = DATA;
  else
    Q = 0;
end
```

Variables declared locally within a subprogram do not hold their value over time, because every time a subprogram is called, its variables are reinitialized. Therefore, HDL Compiler does not infer latches for variables declared in subprograms. In Example 6-9, HDL Compiler does not infer a latch for output `Q`.

Example 6-9 Function: No Latch Inference

```
function MY_FUNC
  input DATA, GATE;
  reg STATE;

  begin
    if (GATE) begin
      STATE = DATA;
    end
    MY_FUNC = STATE;
  end
end function
. . .
Q = MY_FUNC(DATA, GATE);
```

The following sections provide truth tables, code examples, and figures for these types of D latches:

- Simple D Latch
- D Latch With Asynchronous Set or Reset
- D Latch With Asynchronous Set and Reset

Simple D Latch

When you infer a D latch, make sure you can control the gate and data signals from the top-level design ports or through combinational logic. Controllable gate and data signals ensure that simulation can initialize the design.

Example 6-10 provides the Verilog template for a D latch. HDL Compiler generates the verbose inference report shown in Example 6-11. Figure 6-2 shows the inferred latch.

Example 6-10 D Latch

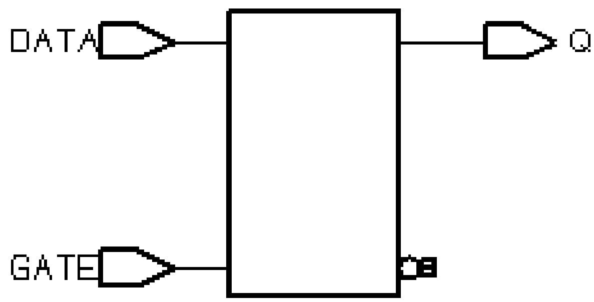
```
module d_latch (GATE, DATA, Q);
  input GATE, DATA;
  output Q;
  reg Q;

always @(GATE or DATA)
  if (GATE)
    Q = DATA;

endmodule
```

Example 6-11 Inference Report for a D Latch

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

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

Figure 6-2 D Latch



D Latch With Asynchronous Set or Reset

The templates in this section use the `async_set_reset` directive to direct HDL Compiler to the asynchronous set or reset pins of the inferred latch.

Example 6-12 provides the Verilog template for a D latch with an asynchronous set. HDL Compiler generates the verbose inference report shown in Example 6-13. Figure 6-3 shows the inferred latch.

Example 6-12 D Latch With Asynchronous Set

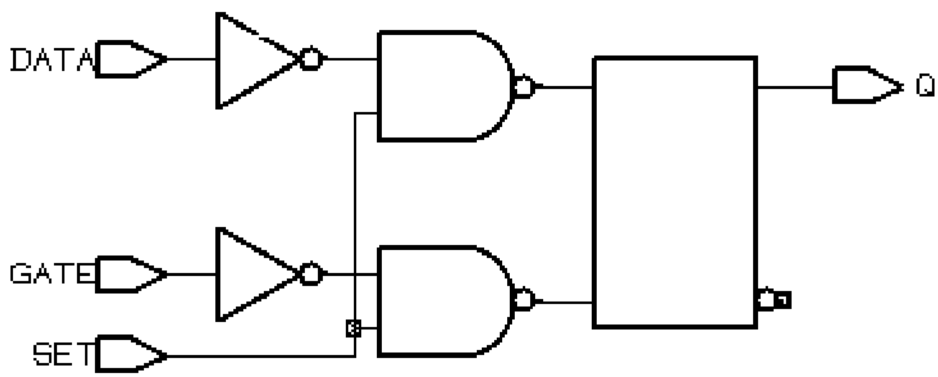
```
module d_latch_async_set (GATE, DATA, SET, Q);
  input GATE, DATA, SET;
  output Q;
  reg Q;

//synopsys async_set_reset ”SET”
always @(GATE or DATA or SET)
  if (~SET)
    Q = 1’b1;
  else if (GATE)
    Q = DATA;
endmodule
```

Example 6-13 Inference Report for D Latch With Asynchronous Set

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

```
Q_reg
Async-set: SET’
```

Figure 6-3 D Latch With Asynchronous Set



Note:

Because the target technology library does not contain a latch with an asynchronous set, Design Compiler synthesizes the set logic by using combinational logic.

Example 6-14 provides the Verilog template for a D latch with an asynchronous reset. HDL Compiler generates the verbose inference report shown in Example 6-15. Figure 6-4 shows the inferred latch.

Example 6-14   D Latch With Asynchronous Reset

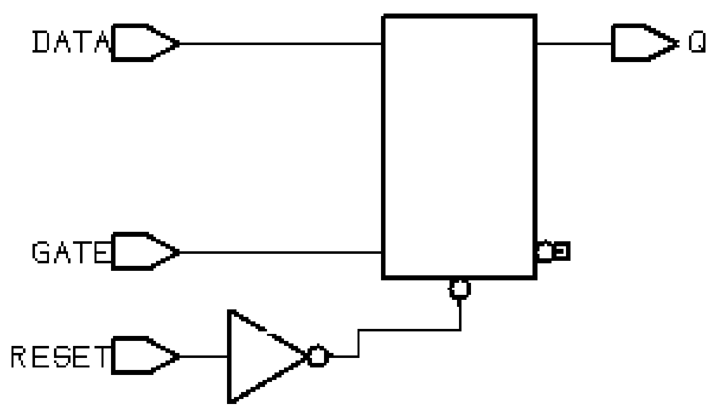
```
module d_latch_async_reset (RESET, GATE, DATA, Q);
  input RESET, GATE, DATA;
  output Q;
  reg Q;

//synopsys async_set_reset ”RESET”
always @ (RESET or GATE or DATA)
  if (~RESET)
    Q = 1’b0;
  else if (GATE)
    Q = DATA;
endmodule
```

Example 6-15   Inference Report for D Latch With Asynchronous Set

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

```
Q_reg
Async-reset: RESET’
```

Figure 6-4   D Latch With Asynchronous Reset



D Latch With Asynchronous Set and Reset

Example 6-16 provides the Verilog template for a D latch with an active-low asynchronous set and reset. This template uses the `async_set_reset_local` directive to direct HDL Compiler to the asynchronous signals in block infer. This template uses the `one_cold` directive to prevent priority encoding of the set and reset signals. For this template, if you do not specify the `one_cold` directive, the set signal has priority, because it serves as the condition for the if clause. HDL Compiler generates the verbose inference report shown in Example 6-17. Figure 6-4 shows the inferred latch.

Example 6-16   D Latch With Asynchronous Set and Reset

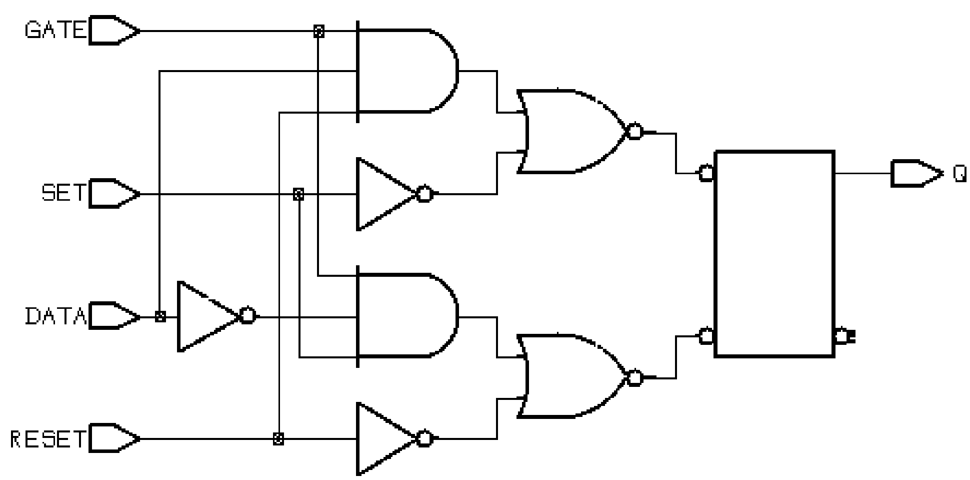
```
module d_latch_async (GATE, DATA, RESET, SET, Q);
  input GATE, DATA, RESET, SET;
  output Q;
  reg Q;

// synopsys async_set_reset_local infer ”RESET, SET”
// synopsys one_cold ”RESET, SET”
always @ (GATE or DATA or RESET or SET)
begin : infer
  if (!SET)
    Q = 1’b1;
  else if (!RESET)
   Q = 1’b0;
   else if (GATE)
    Q = DATA;
end

// synopsys translate_off
always @ (RESET or SET)
  if (RESET == 1’b0 & SET == 1’b0)
  $write (”ONE-COLD violation for RESET and SET.”);
// synopsys translate_on
endmodule
```

Example 6-17 Inference Report for D Latch With Asynchronous Set and Reset

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

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

Figure 6-5 D Latch With Asynchronous Set and Reset



Inferring Master-Slave Latches

HDL Compiler infers master-slave latches by using the `clocked_on_also` `signal_type` attribute.

In your Verilog description, describe the master-slave latch as a flip-flop by using only the slave clock. Specify the master clock as an input port, but do not connect it. In addition, attach the `clocked_on_also` attribute to the master clock port (called `MCK` in these examples).

This coding style requires that cells in the target technology library have slave clocks defined in the library with the `clocked_on_also` attribute in the cell's state declaration. (For more information, see the Synopsys Library Compiler documentation.)

If Design Compiler does not find any master-slave latches in the target technology library, the tool leaves the master-slave generic cell (`MSGEN`) unmapped. Design Compiler does not use D flip-flops to implement the equivalent functionality of the cell.

Note:

> Although the vendor's component behaves as a master-slave latch, Library Compiler supports only the description of a master-slave flip-flop.

Master-Slave Latch With Single Master-Slave Clock Pair

Example 6-19 provides the Verilog template for a master-slave latch. The template uses the `dc_script_begin` and `dc_script_end` compiler directives. See "Embedding Constraints and Attributes" on page 9-22 for more information. HDL Compiler generates the verbose inference report shown in Example 6-20. Figure 6-6 shows the inferred latch.

Example 6-18 Master-Slave Latch

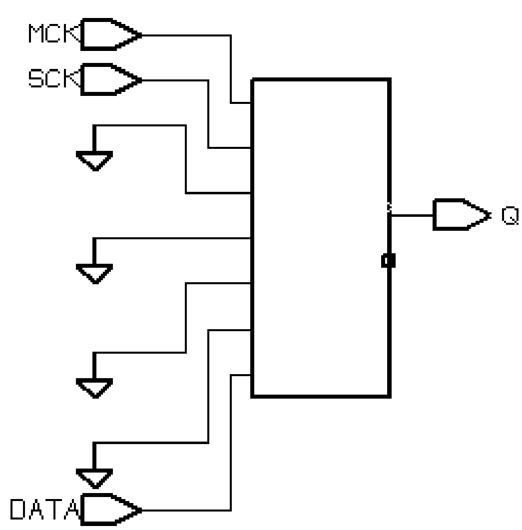
```
module mslatch (SCK, MCK, DATA, Q);
  input SCK, MCK, DATA;
  output Q;
  reg Q;

// synopsys dc_script_begin
// set_signal_type ”clocked_on_also” MCK
// synopsys dc_script_end

always @ (posedge SCK)
  Q <= DATA;
endmodule
```

Example 6-19 Inference Report for a Master-Slave Latch

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

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

Figure 6-6 Master-Slave Latch



Master-Slave Latch With Multiple Master-Slave Clock Pairs

If the design requires more than one master-slave clock pair, you must specify the associated slave clock in addition to the `clocked_on_also` attribute. Example 6-21 illustrates the use of the `clocked_on_also` attribute with the `-associated_clock` option.

Example 6-20 Inferring Master-Slave Latches With Two Pairs of Clocks

```
module mslatch2 (SCK1, SCK2, MCK1, MCK2, D1, D2, Q1, Q2);
  input SCK1, SCK2, MCK1, MCK2, D1, D2;
  output Q1, Q2;
  reg Q1, Q2;

// synopsys dc_script_begin
// set_signal_type ”clocked_on_also” MCK1 -associated_clock SCK1
// set_signal_type ”clocked_on_also” MCK2 -associated_clock SCK2
// synopsys dc_script_end

always @ (posedge SCK1)
  Q1 <= D1;

always @ (posedge SCK2)
  Q2 <= D2;
endmodule
```

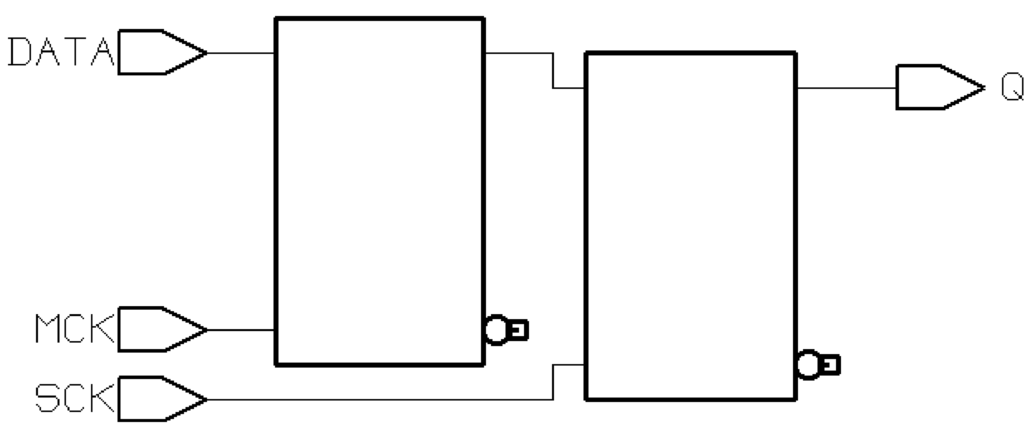
Master-Slave Latch With Discrete Components

If your target technology library does not contain master-slave latch components, you can infer two-phase systems by using D latches. Example 6-22 shows a simple two-phase system with clocks MCK and SCK. Figure 6-7 shows the inferred latch.

Example 6-21   Two-Phase Clocks

```
module latch_verilog (DATA, MCK, SCK, Q);
  input DATA, MCK, SCK;
  output Q;
  reg Q;

  reg TEMP;

always @(DATA or MCK)
  if (MCK)
    TEMP <= DATA;

always @(TEMP or SCK)
  if (SCK)
    Q <= TEMP;
endmodule
```

Figure 6-7   Two-Phase Clocks



Inferring Flip-Flops

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

Inferring D Flip-Flops

HDL Compiler infers a D flip-flop whenever the sensitivity list of an `always` block includes an `edge` expression (a test for the rising or falling edge of a signal). Use the following syntax to describe a rising edge:

```
posedge SIGNAL
```

Use the following syntax to describe a falling edge:

```
negedge SIGNAL
```

When the sensitivity list of an `always` block contains an `edge` expression, HDL Compiler creates flip-flops for all the variables that are assigned values in the block. Example 6-22 shows the most common use of an `always` block to infer a flip-flop.

Example 6-22 Using an always Block to Infer a Flip-Flop

```
always @(edge)
begin
  .
end
```

Simple D Flip-Flop

When you infer a D flip-flop, make sure you can control the clock and data signals from the top-level design ports or through combinational logic. Controllable clock and data signals ensure that simulation can initialize the design. If you cannot control the clock and data signals, infer a D flip-flop with an asynchronous reset or set or with a synchronous reset or set.

When you are inferring a simple D flip-flop, the `always` block can contain only one edge expression.

Example 6-23 provides the Verilog template for a positive-edge-triggered D flip-flop. HDL Compiler generates the verbose inference report shown in Example 6-24. Figure 6-8 shows the inferred flip-flop.

Example 6-23 Positive-Edge-Triggered D Flip-Flop

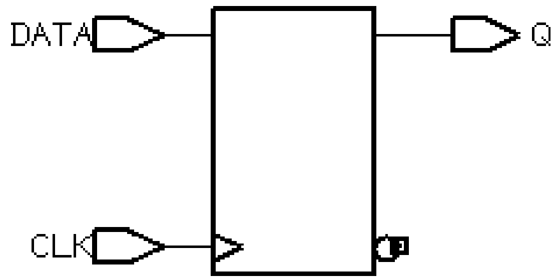
```
module dff_pos (DATA, CLK, Q);
  input DATA, CLK;
  output Q;
  reg Q;

always @(posedge CLK)
  Q <= DATA;
endmodule
```

Example 6-24 Inference Report for a Positive-Edge-Triggered D Flip-Flop

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

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

Figure 6-8 Positive-Edge-Triggered D Flip-Flop



Example 6-25 provides the Verilog template for a negative-edge-triggered D flip-flop. HDL Compiler generates the verbose inference report shown in Example 6-26. Figure 6-9 shows the inferred flip-flop.

Example 6-25 Negative-Edge-Triggered D Flip-Flop

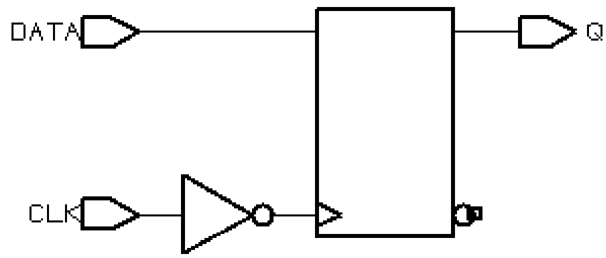
```
module dff_neg (DATA, CLK, Q);
  input DATA, CLK;
  output Q;
  reg Q;

always @(negedge CLK)
  Q <= DATA;
endmodule
```

Example 6-26 Inference Report for a Negative-Edge-Triggered D Flip-Flop

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

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

Figure 6-9 Negative-Edge-Triggered D Flip-Flop



D Flip-Flop With Asynchronous Set or Reset

When inferring a D flip-flop with an asynchronous set or reset, include edge expressions for the clock and the asynchronous signals in the sensitivity list of the `always` block. Specify the asynchronous conditions by using `if` statements. Specify the branches for the asynchronous conditions before the branches for the synchronous conditions.

Example 6-27 provides the Verilog template for a D flip-flop with an asynchronous set. HDL Compiler generates the verbose inference report shown in Example 6-28. Figure 6-10 shows the inferred flip-flop.

Example 6-27 D Flip-Flop With Asynchronous Set

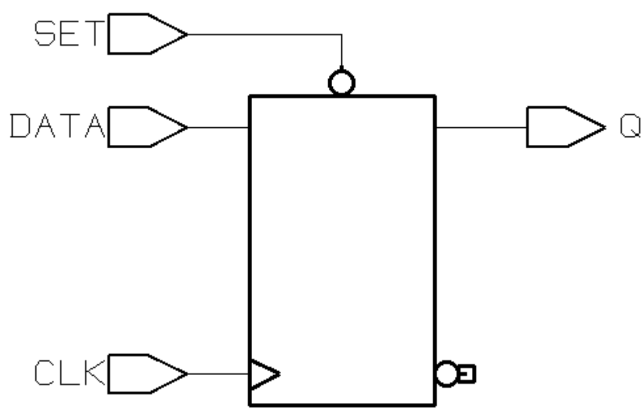
```
module dff_async_set (DATA, CLK, SET, Q);
  input DATA, CLK, SET;
  output Q;
  reg Q;

always @(posedge CLK or negedge SET)
  if (~SET)
    Q <= 1’b1;
  else
    Q <= DATA;
endmodule
```

Example 6-28   Inference Report for a D Flip-Flop With Asynchronous Set

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

```
Q_reg
Async-set: SET’
```

Figure 6-10   D Flip-Flop With Asynchronous Set



Example 6-29 provides the Verilog template for a D flip-flop with an asynchronous reset. HDL Compiler generates the verbose inference report shown in Example 6-30. Figure 6-11 shows the inferred flip-flop.

Example 6-29 D Flip-Flop With Asynchronous Reset

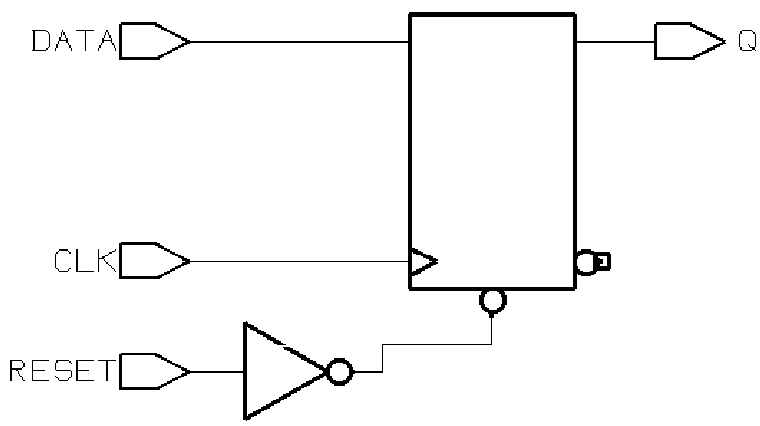
```
module dff_async_reset (DATA, CLK, RESET, Q);
  input DATA, CLK, RESET;
  output Q;
  reg Q;

always @(posedge CLK or posedge RESET)
  if (RESET)
    Q <= 1'b0;
  else
    Q <= DATA;
endmodule
```

Example 6-30 Inference Report for a D Flip-Flop With Asynchronous Reset

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

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

Figure 6-11 D Flip-Flop With Asynchronous Reset



D Flip-Flop With Asynchronous Set and Reset

Example 6-31 provides the Verilog template for a D flip-flop with active high asynchronous set and reset pins. The template uses the `one_hot` directive to prevent priority encoding of the set and reset signals. For this template, if you do not specify the `one_hot` directive, the reset signal has priority, because it is used as the condition for the if clause. HDL Compiler generates the verbose inference report shown in Example 6-32. Figure 6-12 shows the inferred flip-flop.

Example 6-31   D Flip-Flop With Asynchronous Set and Reset

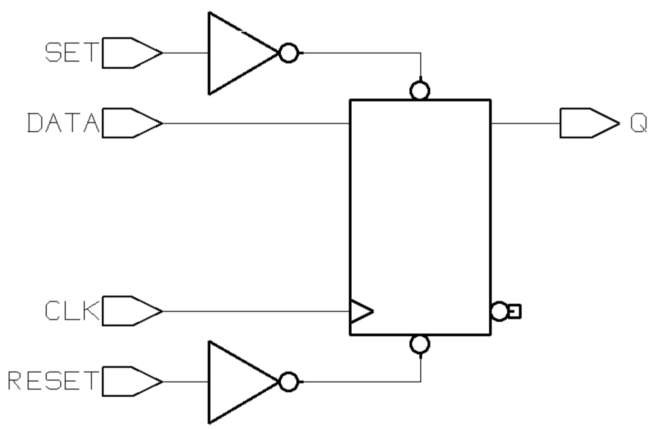
```
module dff_async (RESET, SET, DATA, Q, CLK);
  input CLK;
  input RESET, SET, DATA;
  output Q;
  reg Q;

// synopsys one_hot ”RESET, SET”
always @(posedge CLK or posedge RESET or
         posedge SET)
  if (RESET)
    Q <= 1’b0;
  else if (SET)
    Q <= 1’b1;
  else Q <= DATA;

// synopsys translate_off
always @ (RESET or SET)
  if (RESET + SET > 1)
  $write (”ONE-HOT violation for RESET and SET.”);
// synopsys translate_on
endmodule
```

Example 6-32 Inference Report for a D Flip-Flop With Asynchronous Set and Reset

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

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

Figure 6-12 D Flip-Flop With Asynchronous Set and Reset



D Flip-Flop With Synchronous Set or Reset

The previous examples illustrate how to infer a D flip-flop with asynchronous controls—one way to initialize or control the state of a sequential device. You can also synchronously reset or set a flip-flop (see Example 6-33 and Example 6-35). The `sync_set_reset` directive directs HDL Compiler to the synchronous controls of the sequential device.

When the target technology library does not have a D flip-flop with synchronous reset, HDL Compiler infers a D flip-flop with synchronous reset logic as the input to the D pin of the flip-flop. If the reset (or set) logic is not directly in front of the D pin of the flip-flop, initialization problems can occur during gate-level simulation of the design.

Example 6-33 provides the Verilog template for a D flip-flop with synchronous set. HDL Compiler generates the verbose inference report shown in Example 6-34. Figure 6-13 shows the inferred flip-flop.

Example 6-33 D Flip-Flop With Synchronous Set

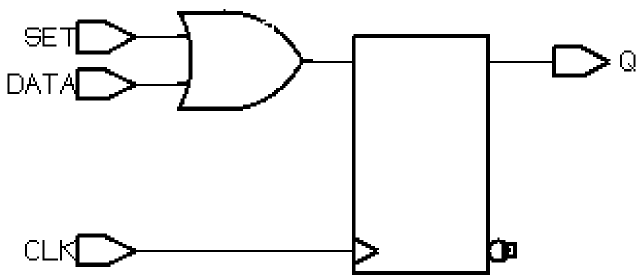
```
module dff_sync_set (DATA, CLK, SET, Q);
  input DATA, CLK, SET;
  output Q;
  reg Q;

//synopsys sync_set_reset ”SET”
always @(posedge CLK)
  if (SET)
    Q <= 1’b1;
  else
    Q <= DATA;
endmodule
```

Example 6-34 Inference Report for a D Flip-Flop With Synchronous Set

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

```
Q_reg
Sync-set: SET
```

Figure 6-13   D Flip-Flop With Synchronous Set



Example 6-35 provides the Verilog template for a D flip-flop with synchronous reset. HDL Compiler generates the verbose inference report shown in Example 6-36. Figure 6-14 shows the inferred flip-flop.

Example 6-35   D Flip-Flop With Synchronous Reset

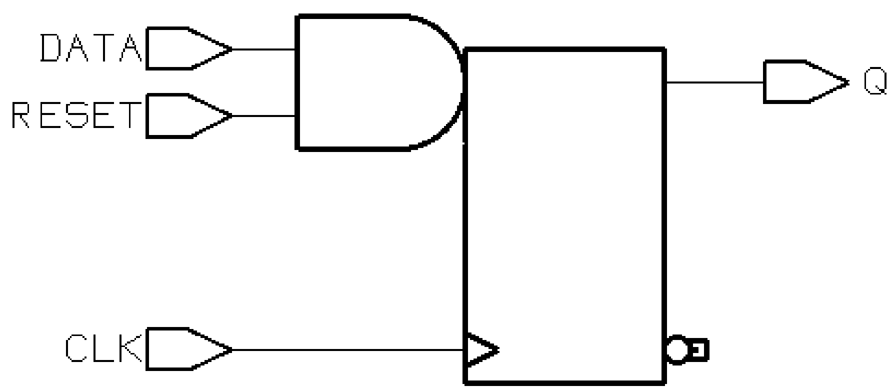
```
module dff_sync_reset (DATA, CLK, RESET, Q);
  input DATA, CLK, RESET;
  output Q;
  reg Q;

//synopsys sync_set_reset ”RESET”
always @(posedge CLK)
  if (~RESET)
    Q <= 1’b0;
  else
    Q <= DATA;
endmodule
```

Example 6-36   Inference Report for a D Flip-Flop With Synchronous Reset

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

```
Q_reg
Sync-reset: RESET'
```

Figure 6-14   D Flip-Flop With Synchronous Reset



D Flip-Flop With Synchronous and Asynchronous Load

D flip-flops can have asynchronous or synchronous controls. To infer a component with synchronous as well as asynchronous controls, you must check the asynchronous conditions before you check the synchronous conditions.

Example 6-37 provides the Verilog template for a D flip-flop with a synchronous load (called SLOAD) and an asynchronous load (called ALOAD). HDL Compiler generates the verbose inference report shown in Example 6-38. Figure 6-15 shows the inferred flip-flop.

Example 6-37 D Flip-Flop With Synchronous and Asynchronous Load

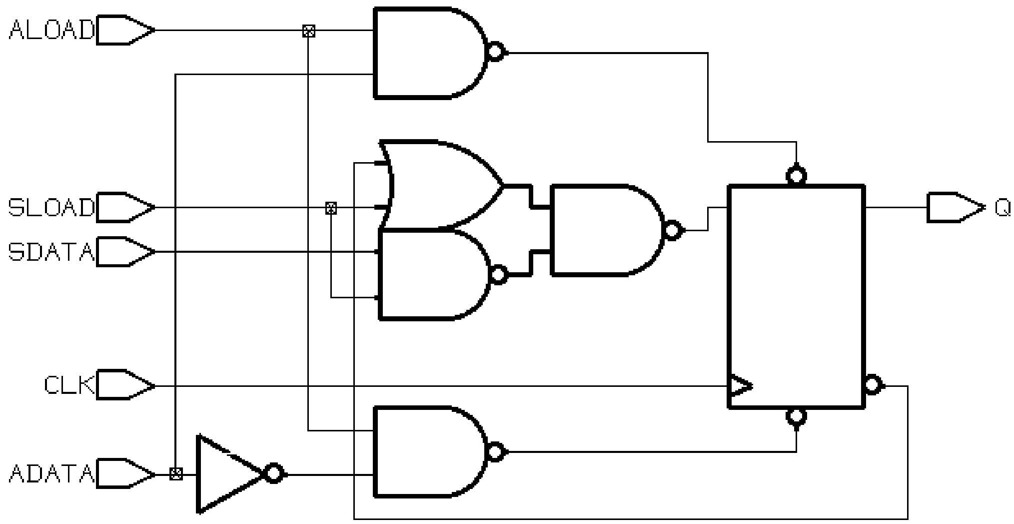
```
module dff_a_s_load (ALOAD, SLOAD, ADATA, SDATA, CLK, Q);
  input ALOAD, ADATA, SLOAD, SDATA, CLK;
  output Q;
  reg Q;

always @ (posedge CLK or posedge ALOAD)
  if (ALOAD)
    Q <= ADATA;
  else if (SLOAD)
    Q <= SDATA;
endmodule
```

Example 6-38 Inference Report for a D Flip-Flop With Synchronous and Asynchronous Load

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

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

Figure 6-15   D Flip-Flop With Synchronous and Asynchronous Load



Multiple Flip-Flops With Asynchronous and Synchronous Controls

If a signal is synchronous in one block but asynchronous in another block, use the `sync_set_reset_local` and `async_set_reset_local` directives to direct HDL Compiler to the correct implementation.

In Example 6-39, block `infer_sync` uses the reset signal as a synchronous reset and block `infer_async` uses the reset signal as an asynchronous reset. HDL Compiler generates the verbose inference reports shown in Example 6-40. Figure 6-16 shows the resulting design.

Example 6-39 Multiple Flip-Flops With Asynchronous and Synchronous Controls

```
module multi_attr (DATA1, DATA2, CLK, RESET, SLOAD,
                   Q1, Q2);
  input DATA1, DATA2, CLK, RESET, SLOAD;
  output Q1, Q2;
  reg Q1, Q2;

//synopsys sync_set_reset_local infer_sync ”RESET”
always @(posedge CLK)
begin : infer_sync
  if (~RESET)
    Q1 <= 1’b0;
  else if (SLOAD)
    Q1 <= DATA1;   // note: else hold Q
end

//synopsys async_set_reset_local infer_async ”RESET”
always @(posedge CLK or negedge RESET)
begin: infer_async
  if (~RESET)
    Q2 <= 1’b0;
  else if (SLOAD)
    Q2 <= DATA2;
end
endmodule
```

Example 6-40 Inference Reports for Example 6-39

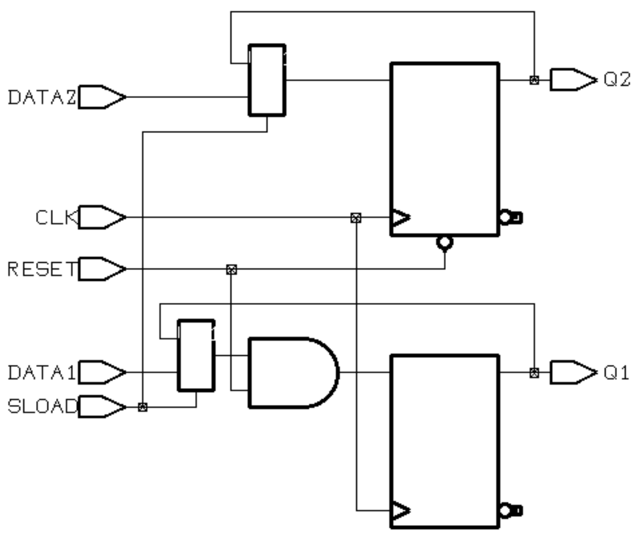
Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q1_reg	Flip-flop	1	-	-	N	N	Y	N	N

```
Q1_reg
Sync-reset: RESET’
```

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q2_reg	Flip-flop	1	-	-	Y	N	N	N	N

```
Q2_reg
Async-reset: RESET’
```

Figure 6-16 Multiple Flip-Flops With Asynchronous and Synchronous Controls



Understanding the Limitations of D Flip-Flop Inference

If you use an `if` statement to infer D flip-flops, your design must meet the following requirements:

- Set and reset conditions cannot use complex expressions.

  The following reset condition is invalid, because it uses a complex expression:

  ```
  always @(posedge clk and negedge reset)
    if (reset == (1-1))
    .
  end
  ```

  HDL Compiler generates the VE-92 message when you use a complex expression in a set or reset condition.

- An `if` statement must occur at the top level of the `always` block.

  The following example is invalid, because the `if` statement does not occur at the top level:

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

  HDL Compiler generates the following message when the `if` statement does not occur at the top level:

  ```
  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 HDL Compiler reference manual
  for ways to infer flip-flops and latches from ’always’
  blocks. (VE-93)
  ```

Inferring JK Flip-Flops

Use the `case` statement to infer JK flip-flops. Before reading in the Verilog description, set the `hdlin_keep_inv_feedback` variable to false.

Note:

If your inference report does not show a synchronous toggle (ST), check that you have set this variable correctly.

This section describes JK flip-flops and JK flip-flops with asynchronous set and reset.

JK Flip-Flop

When you infer a JK flip-flop, make sure you can control the J, K, and clock signals from the top-level design ports to ensure that simulation can initialize the design.

Example 6-41 provides the Verilog code that implements the JK flip-flop described in Table 6-2. In the JK flip-flop, the J and K signals act as active-high synchronous set and reset. Use the `sync_set_reset` directive to indicate that the J and K signals are the synchronous set and reset for the design. Example 6-42 shows the verbose inference report generated by HDL Compiler. Figure 6-17 shows the inferred flip-flop.

Table 6-2 Truth Table for JK Flip-Flop

J	K	CLK	Qn+1
0	0	Rising	Qn
0	1	Rising	0
1	0	Rising	1
1	1	Rising	QnB
X	X	Falling	Qn

Example 6-41 JK Flip-Flop

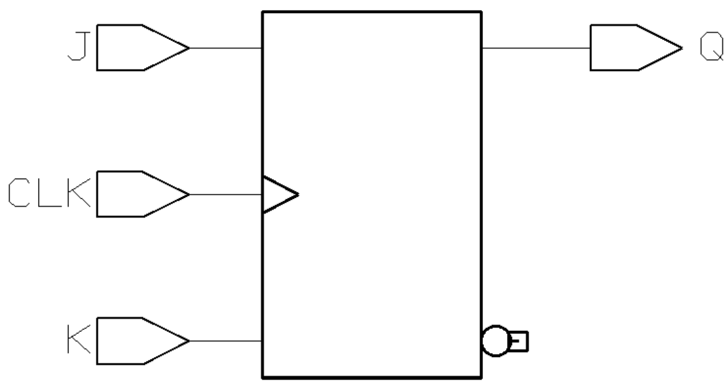
```
module JK(J, K, CLK, Q);
  input J, K;
  input CLK;
  output Q;
  reg Q;

// synopsys sync_set_reset ”J, K”
always @ (posedge CLK)
  case ({J, K})
    2’b01 : Q = 0;
    2’b10 : Q = 1;
    2’b11 : Q = ~Q;
  endcase
endmodule
```

Example 6-42 Inference Report for JK Flip-Flop

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

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

Figure 6-17 JK Flip-Flop



JK Flip-Flop With Asynchronous Set and Reset

Example 6-43 provides the Verilog template for a JK flip-flop with asynchronous set and reset. Use the `sync_set_reset` directive to indicate the JK function. Use the `one_hot` directive to prevent priority encoding of the J and K signals. HDL Compiler generates the verbose inference report shown in Example 6-44. Figure 6-18 shows the inferred flip-flop.

Example 6-43 JK Flip-Flop With Asynchronous Set and Reset

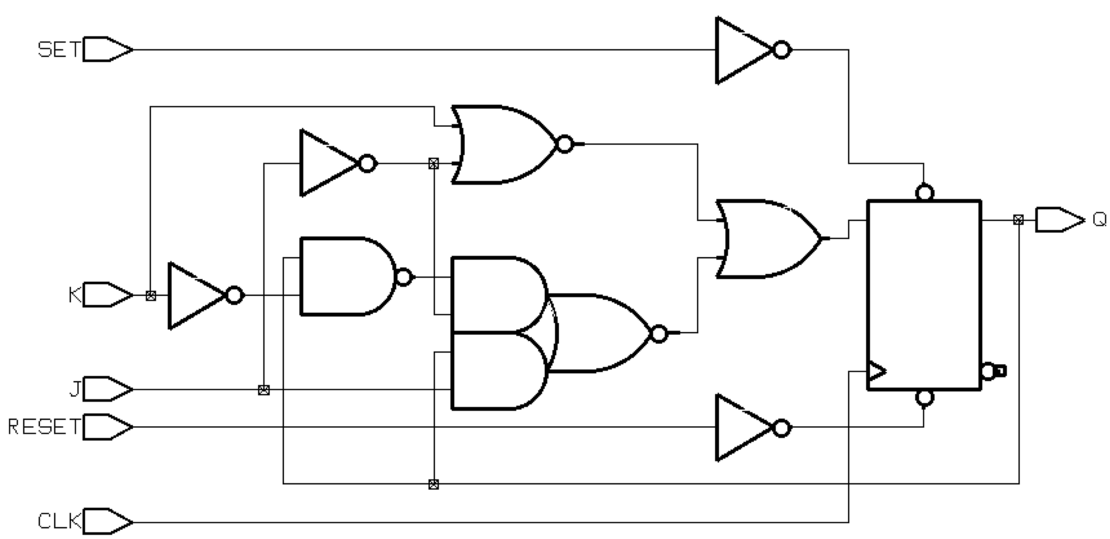
```
module jk_async_sr (RESET, SET, J, K, CLK, Q);
  input RESET, SET, J, K, CLK;
  output Q;
  reg Q;

// synopsys sync_set_reset ”J, K”
// synopsys one_hot ”RESET, SET”
always @ (posedge CLK or posedge RESET or
          posedge SET)
  if (RESET)
    Q <=1’b0;
  else if (SET)
    Q <=1’b1;
  else
    case ({J, K})
      2’b01 : Q = 0;
      2’b10 : Q = 1;
      2’b11 : Q = ~Q;
    endcase

//synopsys translate_off
always @(RESET or SET)
  if (RESET + SET > 1)
    $write (”ONE-HOT violation for RESET and SET.”);
// synopsys translate_on
endmodule
```

Example 6-44 Inference Report for JK Flip-Flop With Asynchronous Set and Reset

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

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

Figure 6-18 JK Flip-Flop With Asynchronous Set and Reset



Inferring Toggle Flip-Flops

To infer toggle flip-flops, follow the coding style in the following examples and set the `hdlin_keep_inv_feedback` variable to false.

Note:

If your inference report does not show a synchronous toggle (`ST`), check that you have set this variable correctly.

You must include asynchronous controls in the toggle flip-flop description. Without them, you cannot initialize toggle flip-flops to a known state.

This section describes toggle flip-flops with an asynchronous set or reset and toggle flip-flops with an enable and an asynchronous reset.

Toggle Flip-Flop With Asynchronous Set or Reset

Example 6-45 shows the template for a toggle flip-flop with asynchronous set. HDL Compiler generates the verbose inference report shown in Example 6-46. Figure 6-19 shows the flip-flop.

Example 6-45 Toggle Flip-Flop With Asynchronous Set

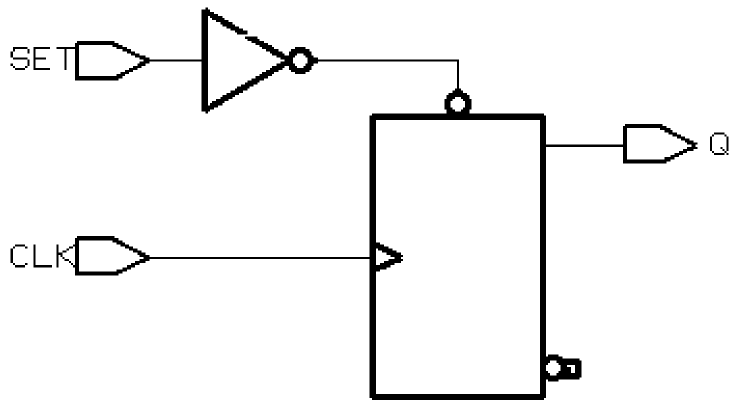
```
module t_async_set (SET, CLK, Q);
  input SET, CLK;
  output Q;
  reg Q;

always @ (posedge CLK or posedge SET)
  if (SET)
    Q <= 1;
   else
    Q <= ~Q;
endmodule
```

Example 6-46   Inference Report for a Toggle Flip-Flop With Asynchronous Set

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

```
TMP_Q_reg
Async-set: SET
Sync-toggle: true
```

Figure 6-19   Toggle Flip-Flop With Asynchronous Set



Example 6-47 provides the Verilog template for a toggle flip-flop with asynchronous reset. Example 6-48 shows the verbose inference report. Figure 6-20 shows the inferred flip-flop.

Example 6-47 Toggle Flip-Flop With Asynchronous Reset

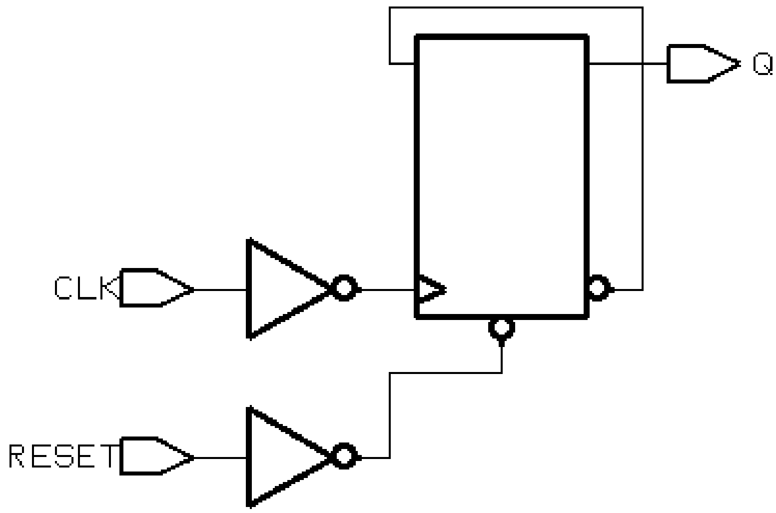
```
module t_async_reset (RESET, CLK, Q);
  input RESET, CLK;
  output Q;
  reg Q;

always @ (posedge CLK or posedge RESET)
  if (RESET)
    Q <= 0;
  else
    Q <= ~Q;
endmodule
```

Example 6-48 Inference Report: Toggle Flip-Flop With Asynchronous Reset

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

```
TMP_Q_reg
Async-reset: RESET
    Sync-toggle: true
```

Figure 6-20   Toggle Flip-Flop With Asynchronous Reset



Toggle Flip-Flops With Enable and Asynchronous Reset

Example 6-49 provides the Verilog template for a toggle flip-flop with an enable and an asynchronous reset. The flip-flop toggles only when the enable (TOGGLE signal) has a logic 1 value. HDL Compiler generates the verbose inference report shown in Example 6-50. Figure 6-21 shows the inferred flip-flop.

Example 6-49   Toggle Flip-Flop With Enable and Asynchronous Reset

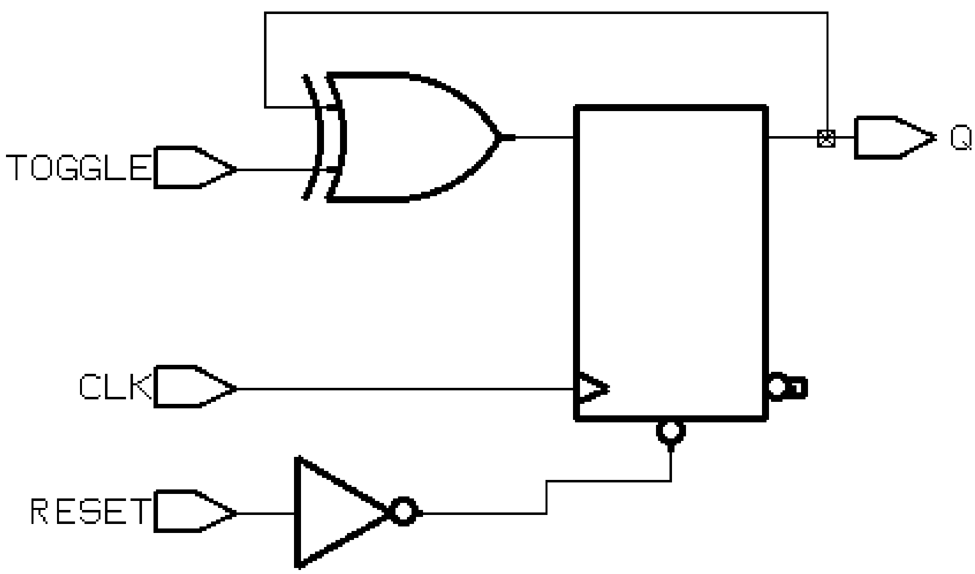
```
module t_async_en_r (RESET, TOGGLE, CLK, Q);
  input RESET, TOGGLE, CLK;
  output Q;
  reg Q;
always @ (posedge CLK or posedge RESET)
begin : infer
  if (RESET)
    Q <= 0;
  else if (TOGGLE)
    Q <= ~Q;
end
endmodule
```

Example 6-50 Inference Report: Toggle Flip-Flop With Enable and Asynchronous Reset

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

```
TMP_Q_reg
Async-reset: RESET
Sync-toggle: TOGGLE
```

Figure 6-21 Toggle Flip-Flop With Enable and Asynchronous Reset



Getting the Best Results

This section provides tips for improving the results you achieve during flip-flop inference. Topics include

- Minimizing flip-flop count
- Correlating synthesis results with simulation results

Minimizing Flip-Flop Count

An `always` block that contains a clock edge in the sensitivity list causes HDL Compiler to infer a flip-flop for each variable assigned a value in that `always` block. It might not be necessary to infer as flip-flops all variables in the `always` block. Make sure your HDL description builds only as many flip-flops as the design requires.

The description in Example 6-51 builds six flip-flops, one for each variable assigned a value in the `always` block (`COUNT(2:0)`, `AND_BITS`, `OR_BITS`, and `XOR_BITS`).

Example 6-51 Circuit With Six Implied Registers

```
module count (CLK, RESET, AND_BITS, OR_BITS, XOR_BITS);
  input CLK, RESET;
  output AND_BITS, OR_BITS, XOR_BITS;
  reg AND_BITS, OR_BITS, XOR_BITS;

  reg [2:0] COUNT;

always @(posedge CLK) begin
  if (RESET)
    COUNT <= 0;
  else
    COUNT <= COUNT + 1;

  AND_BITS <= & COUNT;
  OR_BITS <= | COUNT;
  XOR_BITS <= ^ COUNT;
end
endmodule
```

In this design, the outputs—`AND_BITS`, `OR_BITS`, and `XOR_BITS`—depend solely on the value of the variable `COUNT`. If the variable `COUNT` is inferred as a register, these three outputs are unnecessary.

To compute values synchronously and store them in flip-flops, set up an `always` block with a signal edge trigger. Put the assignments you want clocked in the `always` block with the signal edge trigger, `posedge CLK` in the synchronous block in Example 6-52.

To let other values change asynchronously, put these assignments in a separate `always` block with no signal edge trigger, as shown in the asynchronous block in Example 6-52.

You avoid inferring extra flip-flops by using this style in your description.

Example 6-52 Circuit With Three Implied Registers

```
module count (CLK, RESET,
              AND_BITS, OR_BITS, XOR_BITS);
  input CLK, RESET;
  output AND_BITS, OR_BITS, XOR_BITS;
  reg AND_BITS, OR_BITS, XOR_BITS;

  reg [2:0] COUNT;

//synchronous block
always @(posedge CLK) begin
  if (RESET)
    COUNT <= 0;
  else
    COUNT <= COUNT + 1;
end
//asynchronous block
always @(COUNT) begin
  AND_BITS <= & COUNT;
  OR_BITS <= | COUNT;
  XOR_BITS <= ^ COUNT;
end
endmodule
```

The technique of separating combinational logic from registered or sequential logic is useful for describing state machines. See the following examples in Appendix A:

- “Count Zeros—Combinational Version” on page A-2
- “Count Zeros—Sequential Version” on page A-5
- “Drink Machine—State Machine Version” on page A-8
- “Drink Machine—Count Nickels Version” on page A-13
- “Carry-Lookahead Adder” on page A-15

Correlating With Simulation Results

Using delay specifications with registered values can cause the simulation to behave differently from the logic HDL Compiler synthesizes. For example, the description in Example 6-53 contains delay information that causes Design Compiler to synthesize a circuit that behaves unexpectedly (the post-synthesis simulation results do not match the pre-synthesis simulation results).

Example 6-53 Delays in Registers

```
module flip_flop (D, CLK, Q);
     input D, CLK;
     output Q;
     .
endmodule

module top (A, C, D, CLK);
     .
     reg B;

always @ (A or C or D or CLK)
begin
     B <= #100 A;
     flip_flop F1(A, CLK, C);
     flip_flop F2(B, CLK, D);
end
endmodule
```

In Example 6-53, B changes 100 nanoseconds after A changes. If the clock period is less than 100 nanoseconds, output D is one or more clock cycles behind output C during simulation of the design. However, because 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 post-synthesis simulation.

When using delay information in your designs, make sure that the delays do not affect registered values. In general, you can safely include delay information in your description if it does not change the value that gets clocked into a flip-flop.

Understanding the Limitations of Register Inference

HDL Compiler cannot infer the following components. You must instantiate these components in your Verilog description.

- Flip-flops and latches with three-state outputs
- Flip-flops with bidirectional pins
- Flip-flips with multiple clock inputs
- Multiport latches
- Register banks

Note:

Although you can instantiate flip-flops with bidirectional pins, Design Compiler interprets these cells as black boxes.

Multibit Inference

A multibit component (MBC), such as a 16-bit register, reduces the area and power in a design. But the primary benefits of MBCs are the creation of a more uniform structure for layout during place and route and the expansion of the synthesized area of a design.

Multibit inference allows you to map registers, multiplexers, and three-state cells to regularly structured logic or multibit library cells. Multibit library cells (macro cells, such as 16-bit banked flip-flops) have these advantages:

- Smaller area and delay, due to shared transistors (as in select or set/reset logic) and optimized transistor-level layout

  In the use of single-bit components, the select or set/reset logic is repeated in each single-bit component

- Reduced clock skew in sequential gates, because the clock paths are balanced internally in the hard macro implementing the MBC
- Lower power consumption by the clock in sequential banked components, due to reduced capacitance driven by the clock net
- Better performance, due to the optimized layout within the MBC
- Improved regular layout of the data path

  Note:

  The term *multibit component* refers, for example, to a 16-bit register in your HDL description. The term *multibit library cell* refers to a library macrocell, such as a flip-flop cell.

Controlling Multibit Inference

To direct HDL Compiler to infer multibit components, do one of the following:

- Embed a directive in the Verilog description.

  The directive gives you control over individual wire and register signals.

- Use a dc_shell variable.

  dc_shell variables apply to an entire design.

Directives That Control Multibit Inference

The directives for Verilog are `infer_multibit` and `dont_infer_multibit`.

Set the Verilog directives on wire and register signals to infer multibit components (see Example 6-54 on page 6-61 and Example 6-55 on page 6-61).

Variable That Controls Multibit Inference

The following dc_shell variable controls multibit inference:

`hdlin_infer_multibit`

This variable controls multibit inference for all bused registers, multiplexers, and three-state cells you input in the same dc_shell session. Set this variable before reading in the HDL source. You can select from the following settings for this variable.

`default_none`
Infers multibit components for signals that have the `infer_multibit` directive in the Verilog description. This is the default value.

`default_all`
Infers multibit components for all bused registers, multiplexers, and three-state cells. Use the `dont_infer_multibit` directive to disable multibit mapping for certain signals.

Design Compiler infers multibit components for all bused register, multiplexer, or three-state cells that are larger than 2 bits. If you want to implement as single-bit components all buses that are more than 4 bits, use the following command:

```
set_multibit_options -minimum_width 4
```

This sets a `minimum_multibit_width` attribute on the design.

`never`
Does not infer multibit components, regardless of the attributes or directives in the HDL source.

Inferring Multibit Components

There are two methodologies for inferring multibit components:

- Directing multibit inference from the HDL source

  This is the best methodology for designers who are familiar with the design's layout and able to determine where multibit components have the largest impact.

- Directing multibit inference from a mapped design

  This is the best methodology for designers who complete an initial synthesis run and then a quick placement and routing.

  At that point, it is easier to determine

  - If multibit components would benefit certain areas of the design
  - If the multibit components already inferred are causing routing congestion

To adjust the design after layout, use the following commands:

`create_multibit`

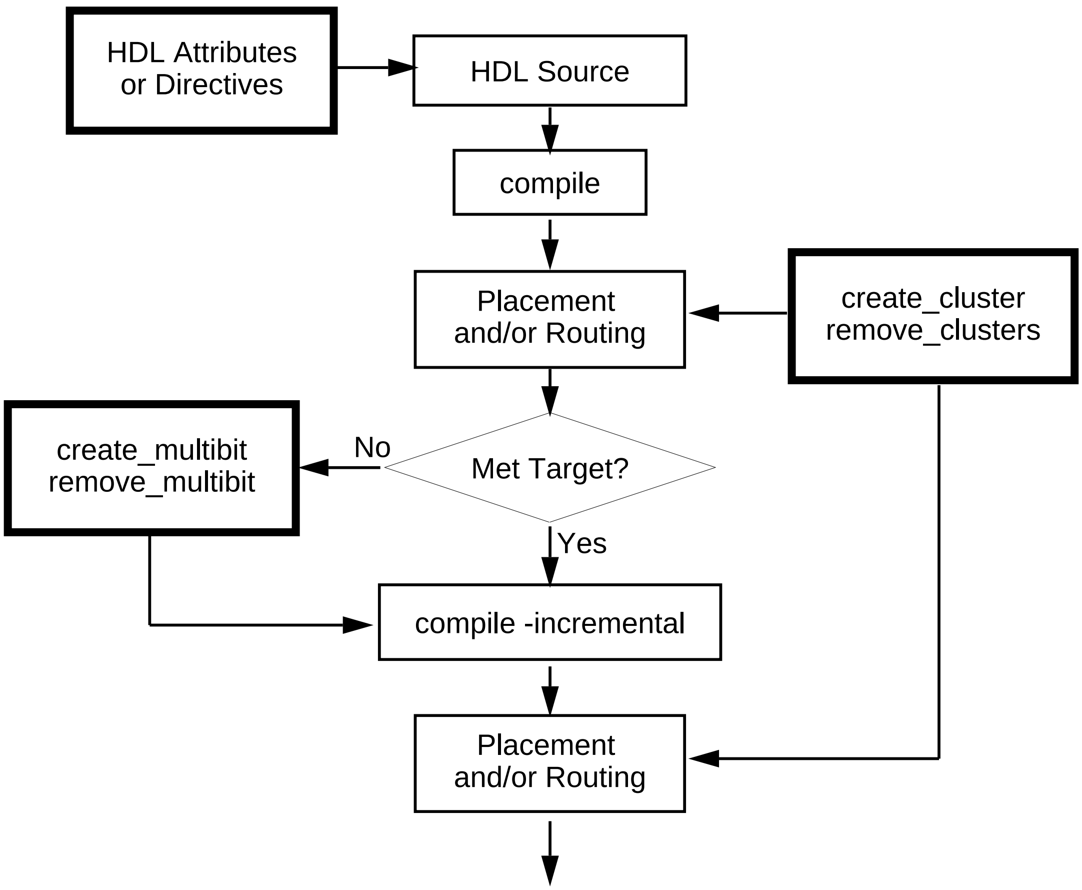
Infers multibit components in a mapped design

`remove_multibit`

Removes multibit components from a mapped design

Figure 6-22 illustrates these methodologies.

Figure 6-22 Design Flow of User-Directed Multibit Cell Inference



Inferring Multibit Cells From HDL Source With the hdlin_infer_multibit variable

If you know where multibit components will work well in your design, inferring multibit cells from HDL source is the best methodology to use.

Use the `hdlin_infer_multibit` variable to indicate the default behavior of all bused register, multiplexer, and three-state cells in your design.

Set the `hdlin_infer_multibit` variable before reading in the HDL source. Unless you change the variable, it will control multibit inferencing in all subsequent HDL files read in during the current dc_shell session.

Inferring Multibit Cells From HDL Source With infer_multibit and dont_infer_multibit Directives

In conjunction with the `hdlin_infer_multibit` variable (described in “Variable That Controls Multibit Inference” on page 6-57), use the `infer_multibit` and `dont_infer_multibit` directives to describe designs that are primarily multibit or primarily single-bit.

Multibit components may not be efficient in the following instances:

- As state machine registers
- In small bused logic that would benefit from single-bit design

Example 6-54 and Example 6-55 show the use of the `infer_multibit` and `dont_infer_multibit` directives.

Example 6-54 shows the use of `infer_multibit` to infer multibit inference of certain signals.

Example 6-55 shows the same HDL code but illustrates how to prevent multibit inference of certain signals.

Example 6-54 Inferring a 6-Bit 4-to-1 Multiplexer

```
module mux4to1_6 (select, a, b, c, d, z);
input [1:0] select;
input [5:0] a, b, c, d;
output [5:0] z;
reg [5:0] z;
//synopsys infer_multibit "z"

    always@(select or a or b or c or d)
    begin
        case (select)          // synopsys infer_mux
        2’b00: z <= a;
        2’b01: z <= b;
        2’b10: z <= c;
        2’b11: z <= d;
    endcase
    end
endmodule
```

Example 6-55 Not Inferring a 6-Bit 4-to-1 Multiplexer

```
module mux4to1_6 (select, a, b, c, d, z);
input [1:0] select;
input [5:0] a, b, c, d;
output [5:0] z;
reg [5:0] z;
//synopsys dont_infer_multibit "z"

    always@(select or a or b or c or d)
    begin
        case (select)          // synopsys infer_mux
        2’b00: z <= a;
        2’b01: z <= b;
        2’b10: z <= c;
        2’b11: z <= d;
    endcase
    end
endmodule
```

Reporting Multibit Inference

HDL Compiler generates an inference report, which shows the information HDL Compiler passes on to Design Compiler about the inferred devices.

Example 6-56 shows a multibit inference report.

Example 6-56 Multibit Inference Report

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
q_reg	Latch	4	Y	Y	N	N	-	-	-

block name/line	Inputs	Outputs	# sel inputs	MB
proc1/23	4	7	2	Y

Three-State Device Name	Type	MB
q_tri_0	Three-State Buffer	Y
s_tri_3	Three-State Buffer	N
q_tri_1	Three-State Buffer	Y
s_tri_0	Three-State Buffer	N

Example 6-56 indicates which cells are inferred as multibit components. The column MB, for sequential cells, indicates whether the vectored component is inferred as a multibit component. The MB column also appears in inference reports for three-state cells and multiplexer cells.

Using the report_multibit Command

In addition to receiving the inference report HDL Compiler generates, you can issue the `report_multibit` command, which lets you report all multibit components in the current design. The report, viewable before and after compile, shows the multibit group name and what cells implement each bit.

Example 6-57 shows a multibit component report.

Example 6-57   Multibit Component Report

Multibit Component : alt178/syn11718					
Cell	Reference	Library	Area	Width	Attributes
U813	mx4a1x16	cba_core_mb	96.00	16	
U9101	mx4a1x16	cba_core_mb	96.00	16	
Total 2 cells			192.00	32	

Multibit Component : data_reg					
Cell	Reference	Library	Area	Width	Attributes
data_reg[0:15]	1d1a2x16	cba_core_mb	48.00	16	n
data_reg[16:31]	1d1a2x16	cba_core_mb	48.00	16	n
Total 2 cells			96.00	32	

The multibit group name for registers and three-state cells is set to the name of the bus. In the cell names of the multibit registers with consecutive bits, a colon separates the outlying bits.

If the colon conflicts with the naming requirements of your place and route tool, you can change the colon to another delimiter by using the `bus_range_separator_style` variable.

For multibit library cells with nonconsecutive bits, a comma separates the nonconsecutive bits. This delimiter is controlled by the `bus_multiple_separator_style` variable. For example, a 4-bit banked register that implements bits 0, 1, 2, and 5 of bus `data_reg` is named `data_reg[0:2,5]`.

For multiplexer cells, the name is set to the cell name of the MUX_OP before optimization.

Listing All Multibit Cells in a Design

To generate a list of all multibit cells in the design, use the new Design Compiler object multibit in a find command, as shown here:

```
find (multibit, “*”)
```

Understanding the Limitations of Multibit Inference

You can infer as multibit components only register, multiplexer, and three-state cells that have identical structures for each bit.

Note:

> Multibit inference of other combinational multibit cells occurs only during sequential mapping of multibit registers. Multibit sequential mapping does not pull in as many levels of logic as single-bit sequential mapping. Thus, Design Compiler might not infer a complex multibit sequential cell, such as a JK flip-flop, which could adversely affect the quality of the design.

See the *Design Compiler Reference Manual: Optimization and Timing Analysis* for more information about how Design Compiler handles multibit components.

Multiplexer Inference

Hardware designers often use multiplexers to implement conditional assignments to signals. HDL Compiler can infer a generic multiplexer cell (MUX_OP) from case statements in your Verilog description. Unlike with register inference, HDL Compiler also can infer MUX_OPs from case statements in subprograms. Design Compiler maps inferred MUX_OPs to multiplexer cells in the target technology library.

Note:

> If you want to use the multiplexer inference feature, the target technology library must contain at least a 2-to-1 multiplexer.

MUX_OPs are hierarchical cells similar to Synopsys DesignWare components. Design Compiler determines the MUX_OP implementation during compile, based on the design constraints. See the *Design Compiler Reference Manual: Optimization and Timing Analysis* for information about how Design Compiler maps MUX_OPs to multiplexers in the target technology library.

Reporting Multiplexer Inference

HDL Compiler generates an inference report that shows the information the compiler passes on to Design Compiler about the inferred devices. The `hdlin_report_inferred_modules` variable has no effect on the multiplexer inference report.

Example 6-58 shows a MUX_OP inference report.

Example 6-58   MUX_OP Inference Report

Statistics for MUX_OPs

block name/line	Inputs	Outputs	# sel inputs	MB
blk1/20	2	1	1	N

The first column of the report indicates the block that contains the case statement for which the MUX_OP is inferred. The line number of the case statement in Verilog also appears in this column. The remaining columns indicate the number of inputs, outputs, and select lines on the inferred MUX_OP.

Controlling Multiplexer Inference

You can embed an HDL Compiler directive in the Verilog description or use dc_shell variables to direct HDL Compiler to infer MUX_OPs. The directive gives you control over individual `case` statements, whereas dc_shell variables apply to an entire design.

HDL Compiler Directive That Controls Multiplexer Inference

Set the `infer_mux` directive on a block to direct HDL Compiler to infer MUX_OPs for all case statements in that block. You can also set the `infer_mux` directive on specific `case` statements to limit MUX_OP inference to that `case` statement.

Attach the `infer_mux` directive to a block, by using the following syntax:

```
// synopsys infer_mux block_label_list
```

Attach the infer_mux directive to a case statement by using the following syntax:

```
case (var) //synopsys infer_mux
```

Variables That Control Multiplexer Inference

The following dc_shell variables control multiplexer inference:

hdlin_infer_mux = default

This variable controls MUX_OP inference for all designs you input in the same dc_shell session. You can select from the following settings for this variable:

Use default to infer MUX_OPs for case statements in blocks that have the infer_mux directive attached. This is the default value.

The value of none does not infer MUX_OPs, regardless of the directives set in the Verilog description. HDL Compiler generates the following message during MUX_OP inference when this variable is set to none:

```
Warning: A mux for process %s was not inferred because
the variable hdlin_infer_mux was set to none. (HDL-384)
```

The value of all treats each case statement in the design as if the infer_mux directive is attached to it. This can negatively affect the quality of results, because it might be more efficient to implement the MUX_OPs as random logic instead of using a specialized multiplexer structure. Use this setting only if you want MUX_OPs inferred for every case statement in your design.

`hdlin_dont_infer_mux_for_resource_sharing = true`

When this variable is true, HDL Compiler does not infer a MUX_OP when two or more synthetic operators drive the data inputs of the MUX_OP. HDL Compiler generates the following message during MUX_OP inference when this variable is true:

```
Warning: No mux inferred for the case %s because it would
lose the benefit of resource sharing. (HDL-380)
```

When this variable is false, HDL Compiler infers a MUX_OP but resource sharing does not share the data pins that the synthetic operators drive. This can have a negative impact on the area of the final implementation. HDL Compiler generates the following message during MUX_OP inference when this variable is false:

```
Warning: A mux has been inferred for case %s which may
lose the benefit of resource sharing. (HDL-381)
```

`hdlin_mux_size_limit = 32`

If the number of branches in a `case` statement exceeds the maximum size specified by this variable, HDL Compiler generates the following message:

```
Warning: A mux was not inferred because case statement
%s has a very large branching factor. (HDL-383)
```

This variable sets the maximum size of a MUX_OP that HDL Compiler can infer. If you set this variable to a value greater than 32, HDL Compiler takes longer to process the design.

The following dc_shell variables control how Design Compiler maps the MUX_OPs:

`compile_create_mux_op_hierarchy = true`

When this variable is true, the `compile` command creates all MUX_OP implementations with their own level of hierarchy. When it is false, the `compile` command removes this level of hierarchy.

`compile_mux_no_boundary_optimization = false`

When this variable is false, the `compile` command performs boundary optimization on all MUX_OP implementations. When true, the `compile` command does not perform these boundary optimizations.

For more information about these variables, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*.

Inferring Multiplexers

This section contains Verilog examples that infer MUX_OPs.

The size of the inferred MUX_OP depends on the number of unique values that are read in the `case` statement. During compilation, Design Compiler attempts to map the MUX_OP to an appropriately sized multiplexer in the target technology library. If the library does not contain a large enough multiplexer, Design Compiler builds the multiplexer with smaller multiplexer cells (such as 4-to-1 multiplexer cells).

Example 6-59 attaches the `infer_mux` directive to the block blk1. HDL Compiler infers a MUX_OP for each case statement in the block. The first `case` statement reads eight unique values and infers an

8-to-1 MUX_OP. The second `case` statement reads four unique values and infers a 4-to-1 MUX_OP. HDL Compiler generates the inference report shown in Example 6-60.

Example 6-59 Multiplexer Inference for a Block

```
module muxtwo(DIN1, DIN2, SEL1, SEL2, DOUT1, DOUT2);
  input [7:0] DIN1;
  input [3:0] DIN2;
  input [2:0] SEL1;
  input [1:0] SEL2;
  output DOUT1, DOUT2;
  reg DOUT1, DOUT2;

//synopsys infer_mux ”blk1”

always @(SEL1 or SEL2 or DIN1 or DIN2)
begin: blk1
  // this case statement infers an 8-to-1 MUX_OP
  case (SEL1)
    3’b000: DOUT1 <= DIN1[0];
    3’b001: DOUT1 <= DIN1[1];
    3’b010: DOUT1 <= DIN1[2];
    3’b011: DOUT1 <= DIN1[3];
    3’b100: DOUT1 <= DIN1[4];
    3’b101: DOUT1 <= DIN1[5];
    3’b110: DOUT1 <= DIN1[6];
    3’b111: DOUT1 <= DIN1[7];
  endcase

  // this case statement infers an4-to-1 MUX_OP
  case (SEL2)
    2’b00: DOUT2 <= DIN2[0];
    2’b01: DOUT2 <= DIN2[1];
    2’b10: DOUT2 <= DIN2[2];
    2’b11: DOUT2 <= DIN2[3];
  endcase
end
endmodule
```

Example 6-60 Inference Report for a Block

Statistics for MUX_OPs

block name/line	Inputs	Outputs	# sel inputs	MB
blk1/53	4	1	2	N
blk1/29	8	1	3	N

Example 6-61 uses the `infer_mux` directive for a specific case statement. This `case` statement reads eight unique values, and HDL Compiler infers an 8-to-1 MUX_OP. HDL Compiler generates the inference report shown in Example 6-62.

Example 6-61 Multiplexer Inference for a Specific case Statement

```
module mux8to1 (DIN, SEL, DOUT);
  input [7:0] DIN;
  input [2:0] SEL;
  output DOUT;
  reg DOUT;

always@(SEL or DIN)
begin: blk1
  case (SEL)  // synopsys infer_mux
    3'b000: DOUT <= DIN[0];
    3'b001: DOUT <= DIN[1];
    3'b010: DOUT <= DIN[2];
    3'b011: DOUT <= DIN[3];
    3'b100: DOUT <= DIN[4];
    3'b101: DOUT <= DIN[5];
    3'b110: DOUT <= DIN[6];
    3'b111: DOUT <= DIN[7];
  endcase
end
endmodule
```

Example 6-62 Inference Report for case Statement

Statistics for MUX_OPs

block name/line	Inputs	Outputs	# sel inputs	MB
blk1/19	8	1	3	N

Understanding the Limitations of Multiplexer Inference

HDL Compiler does not infer MUX_OPs for

- `if...else` statements
- `case` statements that contain two or more synthetic operators, unless you set the following variable to false before inputting the Verilog description: `hdlin_dont_infer_mux_for_resource_sharing`
- `case` statements in `while` loops

HDL Compiler does infer MUX_OPs for incompletely specified `case` statements, but the resulting logic might not be optimal. HDL Compiler generates the following message when inferring a MUX_OP for an incompletely specified `case` statement:

```
Warning: A mux has been inferred for case %s which has either
a default clause or an incomplete mapping. (HDL-382)
```

HDL Compiler considers the following types of `case` statements incompletely specified:

- `case` statements that have a missing `case` statement branch or a missing assignment in a `case` statement branch
- `case` statements that contain an `if` statement or `case` statements that contain other `case` statements

Three-State Inference

HDL Compiler infers a three-state driver when you assign the value of z to a variable. The z value represents the high-impedance state. HDL Compiler infers one three-state driver per block. You can assign high-impedance values to single-bit or bused variables.

Reporting Three-State Inference

HDL Compiler can generate an inference report that shows the information the compiler passes on to Design Compiler about the inferred devices. The `hdlin_report_inferred_modules` variable controls the generation of the three-state inference report. See "Reporting Register Inference" on page 6-2 for more information about the `hdlin_report_inferred_modules` variable. For three-state inference, HDL Compiler generates the same report for the default and the verbose reports.

Example 6-63 shows a three-state inference report.

Example 6-63   Three-State Inference Report

Three-State Device Name	Type	MB
OUT1_tri	Three-State Buffer	N

The first column of the report indicates the name of the inferred three-state device. The second column indicates the type of three-state device HDL Compiler inferred.

Controlling Three-State Inference

HDL Compiler always infers a three-state driver when you assign the value of z to a variable. HDL Compiler does not provide any means of controlling the inference.

Inferring Three-State Drivers

This section contains Verilog examples that infer the following types of three-state drivers:

- Simple three-state drivers
- Registered three-state drivers

Simple Three-State Driver

This section provides a template for a simple three-state driver. In addition, it provides examples of how allocating high-impedance assignments to different blocks affects three-state inference.

Example 6-64 provides the Verilog template for a simple three-state driver. HDL Compiler generates the inference report shown in Example 6-65. Figure 6-23 shows the inferred three-state driver.

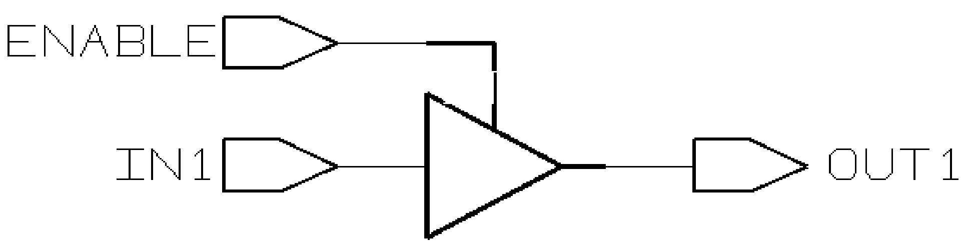
Example 6-64 Simple Three-State Driver

```
module three_state (ENABLE, IN1, OUT1);
  input IN1, ENABLE;
  output OUT1;
  reg OUT1;

always @(ENABLE or IN1) begin
  if (ENABLE)
    OUT1 = IN1;
  else
    OUT1 = 1'bz;  //assigns high-impedance state
end
endmodule
```

Example 6-65 Inference Report for Simple Three-State Driver

Three-State Device Name	Type	MB
OUT1_tri	Three-State Buffer	N

Figure 6-23 Schematic of Simple Three-State Driver



Example 6-66 provides an example of placing all high-impedance assignments in a single block. In this case, the data is gated and HDL Compiler infers a single three-state driver. Example 6-67 shows the inference report. Figure 6-24 shows the schematic the code generates.

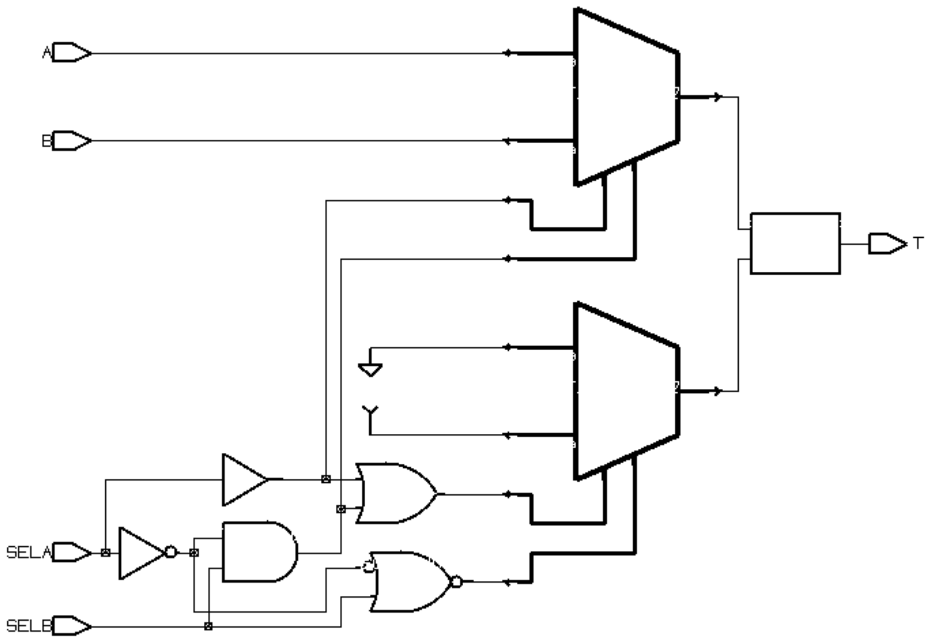
Example 6-66 Inferring One Three-State Driver From a Single Block

```
module three_state (A, B, SELA, SELB, T);
  input  A, B, SELA, SELB;
  output T;
  reg T;

always @(SELA or SELB or A or B) begin
  T = 1'bz;
  if (SELA)
    T = A;
  if (SELB)
    T = B;
end
endmodule
```

Example 6-67 Single Block Inference Report

Three-State Device Name	Type	MB
T_tri	Three-State Buffer	N

Figure 6-24 One Three-State Driver Inferred From a Single Block



Example 6-68 provides an example of placing each high-impedance assignment in a separate block. In this case, HDL Compiler infers multiple three-state drivers. Example 6-69 shows the inference report. Figure 6-25 shows the schematic the code generates.

Example 6-68 Inferring Three-State Drivers From Separate Blocks

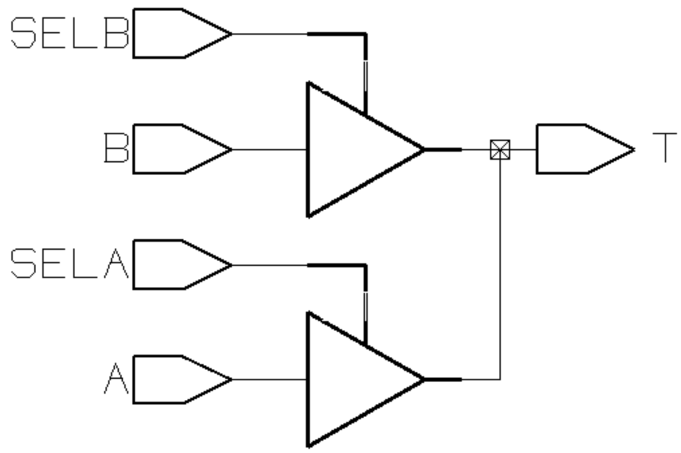
```
module three_state (A, B, SELA, SELB, T);
  input  A, B, SELA, SELB;
  output T;
  reg T;

always @(SELA or A)
  if (SELA)
    T = A;
  else
    T = 1'bz;

always @(SELB or B)
  if (SELB)
    T = B;
  else
    T = 1'bz;
endmodule
```

Example 6-69 Inference Report for Two Three-State Drivers

Three-State Device Name	Type	MB
T_tri	Three-State Buffer	N

Three-State Device Name	Type	MB
T_tri2	Three-State Buffer	N

Figure 6-25 Two Three-State Drivers Inferred From Separate Blocks



Registered Three-State Drivers

When a variable is registered in the same block in which it is defined as three-state, HDL Compiler also registers the enable pin of the three-state gate. Example 6-70 shows an example of this type of code. Example 6-71 shows the inference report. Figure 6-26 shows the schematic generated by the code.

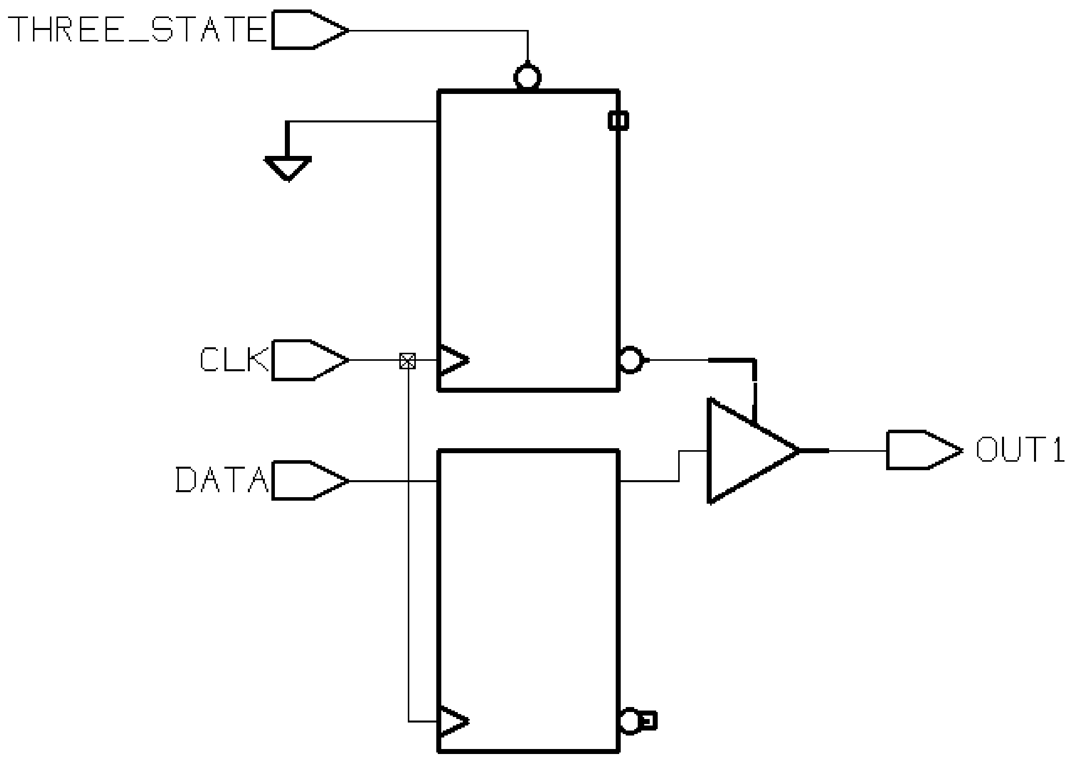
Example 6-70 Three-State Driver With Registered Enable

```
module ff_3state (DATA, CLK, THREE_STATE, OUT1);
  input DATA, CLK, THREE_STATE;
  output OUT1;
  reg OUT1;

always @ (posedge CLK) begin
  if (THREE_STATE)
    OUT1 = 1'bz;
  else
    OUT1 = DATA;
end
endmodule
```

Example 6-71 Inference Report for Three-State Driver With Registered Enable

Three-state Device Name	Type	MB
OUT1_tri	Three-State Buffer	N
OUT1_tr_enable_reg	Flip-flop (width 1)	N

Figure 6-26 Three-State Driver With Registered Enable



In Figure 6-26, the three-state gate has a register on its enable pin. Example 6-72 uses two blocks to instantiate a three-state gate, with a flip-flop only on the input. Example 6-73 shows the inference report. Figure 6-27 shows the schematic the code generates.

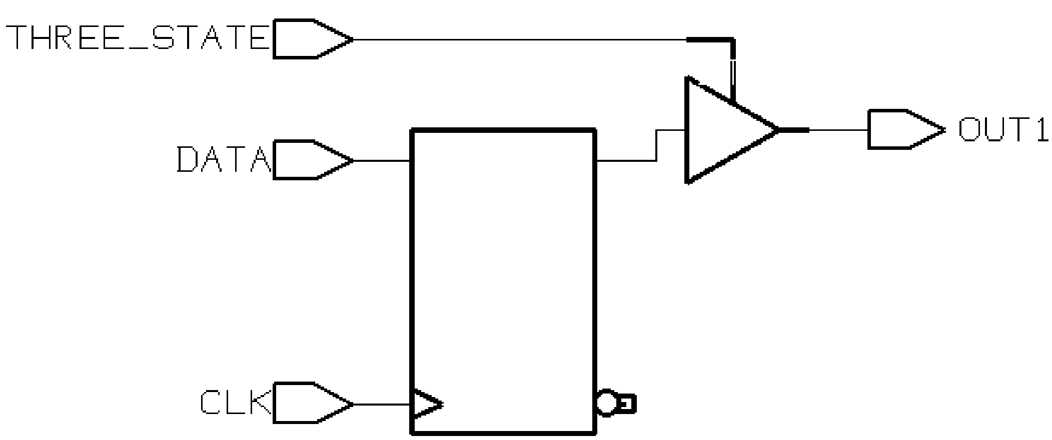
Example 6-72 Three-State Driver Without Registered Enable

```
module ff_3state (DATA, CLK, THREE_STATE, OUT1);
  input DATA, CLK, THREE_STATE;
  output OUT1;
  reg OUT1;

  reg TEMP;

always @(posedge CLK)
  TEMP <= DATA;

always @(THREE_STATE or TEMP)
  if (THREE_STATE)
    OUT1 = TEMP;
  else
    OUT1 = 1'bz;
endmodule
```

Example 6-73 Inference Report for Three-State Driver Without Registered Enable

Three-State Device Name	Type	MB
OUT1_tri	Three-State Buffer	N

Figure 6-27 Three-State Driver Without Registered Enable



Understanding the Limitations of Three-State Inference

You can use the z value in the following ways:

- Variable assignment
- Function call argument
- Return value

You cannot use the z value in an expression, except for comparison with z. Be careful when using expressions that compare with the z value. Design Compiler always evaluates these expressions to false, and the pre-synthesis and post-synthesis simulation results might differ. For this reason, HDL Compiler issues a warning when it synthesizes such comparisons.

This is an example of incorrect use of the z value in an expression:

```
OUT_VAL = (1'bz && IN_VAL);
```

This is an example of correct use of the z value in an expression:

```
if (IN_VAL == 1'bz) then
```

7

Resource Sharing

Resource sharing is the assignment of similar Verilog operations (for example, +) to a common netlist cell. Netlist cells are the resources—they are equivalent to built hardware. Resource sharing reduces the amount of hardware needed to implement Verilog operations.

Without resource sharing, each Verilog operation is built with separate circuitry. For example, every + with noncomputable operands causes a new adder to be built. This repetition of hardware increases the area of a design.

In contrast, with resource sharing, several Verilog + operations can be implemented with a single adder, which reduces the amount of hardware required. Also, different operations, such as + and –, can be assigned to a single adder or subtracter to further reduce a design's circuit area.

This chapter explains resource sharing, in the following sections:

- Scope and Restrictions
- Resource Sharing Methods
- Resource Sharing Conflicts and Error Messages
- Reports

Scope and Restrictions

Not all operations in your design can be shared. This section describes how to tell whether operations are candidates for sharing hardware.

The following operators can be shared with other like operators (such as * with *) and with the operators shown on the same line.

```
*
+    –
>    >=    <    <=
```

Operations can be shared only if they lie in the same `always` block. These blocks are usually implemented as synthetic library elements. See “Synthetic Libraries” on page 10-12 for more information.

Example 7-1 shows several possible sharing operations.

Example 7-1 Scope for Resource Sharing

```
always @(A1 or B1 or C1 or D1 or COND_1)
begin
     if(COND_1)
        Z1 = A1 + B1;
     else
        Z1 = C1 + D1;
end

always @(A2 or B2 or C2 or D2 or COND_2)
begin
     if(COND_2)
        Z2 = A2 + B2;
     else
        Z2 = C2 + D2;
end
```

Table 7-1 summarizes the possible sharing operations in Example 7-1. A no indicates that sharing is not allowed because the operations lie in different always blocks. A yes means sharing is allowed.

Table 7-1 Allowed and Disallowed Sharing for Example 7-1

	A1 + B1	**C1 + D1**	**A2 + B2**	**C2 + D2**
A1 + B1		yes	no	no
C1 + D1	yes		no	no
A2 + B2	no	no		yes
A2 + B2	no	no	yes	

The next two sections describe two types of conflicts, control flow conflicts and data flow conflicts, where sharing is not allowed.

Control Flow Conflicts

Two operations can be shared only if no execution path exists from the start of the block to the end of the block that reaches both operations. For example, if two operations lie in separate branches of an `if` or `case` statement, they are not on the same path (and can be shared). Example 7-2 illustrates control flow conflicts for `if` statements.

Example 7-2   Control Flow Conflicts for if Statements

```
always begin
     Z1 = A + B;

     if(COND_1)
        Z2 = C + D;

     else begin
        Z2 = E + F;
        if(COND_2)
           Z3 = G + H;
        else
           Z3 = I + J;
     end

     if(! COND_1)
        Z4 = K + L;
     else
        Z4 = M + N;
 end
```

Table 7-2 summarizes the possible sharing operations in Example 7-2. A no indicates that sharing is not allowed because of the flow of control (execution path) through the block. A yes means sharing is allowed.

Table 7-2 Allowed and Disallowed Sharing for Example 7-2

	A + B	**C + D**	**E + F**	**G + H**	**I + J**	**K + L**	**M +N**
A + B		no	no	no	no	no	no
C + D	no		yes	yes	yes	no	no
E + F	no	yes		no	no	no	no
G + H	no	yes	no		yes	no	no
I + J	no	yes	no	yes		no	no
K + L	no	no	no	no	no		yes
M + N	no	no	no	no	no	yes	

Note that the C + D addition cannot be shared with the K + L addition, even though no set of input values causes both to execute. When HDL Compiler evaluates the ability to share, it assumes that the values of expressions that control `if` statements are unrelated. The same rule applies to `case` statements, as shown in Example 7-3.

Example 7-3 Control Flow Conflicts for case Statement

```
always begin
    Z1 = A + B;

    case(OP)
         2'h0: Z2 = C + D;
         2'h1: Z2 = E + F;
         2'h2: Z2 = G + H;
         2'h3: Z2 = I + J;
    endcase
end
```

Table 7-3 summarizes the possible sharing operations in Example 7-3. A no indicates that sharing is not allowed because of the flow of control (execution path) through the circuit. A yes means sharing is allowed.

Table 7-3 Allowed and Disallowed Sharing for Example 7-3

	A + B	**C + D**	**E + F**	**G + H**	**I + J**
A + B		no	no	no	no
C + D	no		yes	yes	yes
E + F	no	yes		yes	yes
G + H	no	yes	yes		yes
I + J	no	yes	yes	yes	

Although operations in separate branches of an `if` statement can be shared, operations in separate branches of a `?:` (conditional) construct cannot share the same hardware, even if they are on separate lines.

Consider the following line of code, where `expression_`*n* represents any expression.

```
z = expression_1 ? expression_2 : expression_3;
```

HDL Compiler interprets this code as

```
temp_1 = expression_1;
temp_2 = expression_2;
temp_3 = expression_3;

z = temp_1 ? temp_2 : temp_3;
```

HDL Compiler evaluates both `expression_2` and `expression_3`, regardless of the value of the conditional. Therefore, operations in `expression_2` cannot share the same resource as operations in `expression_3`.

If you want operations in separate branches of `?:` constructs to share hardware, rewrite your code with an `if` statement. You can rewrite the previous expression as

```
if (expression_1)
     z = expression_2;
else
     z = expression_3;
```

The operations in a `?:` construct cannot share hardware with each other, but they can share hardware with operations in separate branches of an `if` statement or a `case` statement. The code fragment in Example 7-4 illustrates which operations can be shared when you use the `?:` construct in separate branches of an `if` statement.

Example 7-4   Code Fragment With ?: Operator and if...else Statement

```
if (cond_1)
     z = cond_2 ? (a + b) : (c + d);
else
   z = cond_3 ? (e + f) : (g + h);
```

Table 7-4 summarizes which operations can be shared in the previous code fragment.

Table 7-4 Allowed and Disallowed Sharing for Example 7-4

	a + b	**c + d**	**e + f**	**g + h**
a + b		no	yes	yes
c + d	no		yes	yes
e + f	yes	yes		no
g + h	yes	yes	no	

To allow resource sharing in separate branches of the `?:` operations in Example 7-4, rewrite the code fragment as shown in Example 7-5.

Example 7-5 Rewritten Code Fragment With if...else Statements

```
if (cond_1) begin
     if (cond_2)
          z = (a + b);
     else
          z = (c + d);
end else begin
     if (cond_3)
          z = (e + f);
     else
          z = (g + h);
end
```

Data Flow Conflicts

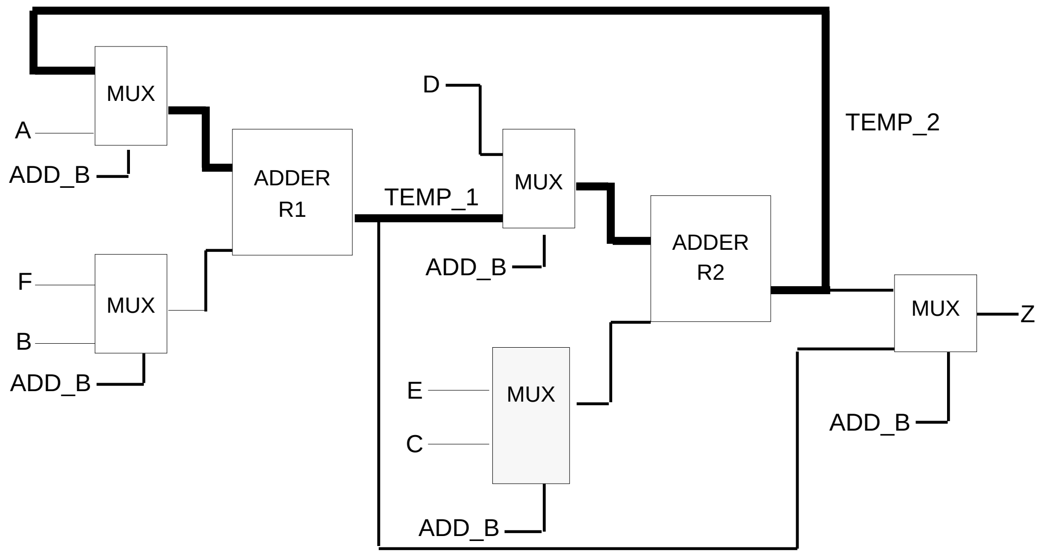
Operations cannot be shared if doing so causes a combinational feedback loop. To understand how sharing can cause a feedback loop, consider Example 7-6.

Example 7-6 Data Flow Conflict

```
always @(A or B or C or D or E or F or Z or ADD_B)

begin
     if(ADD_B) begin
          TEMP_1 = A + B;
          Z = TEMP_1 + C;
     end
     else begin
          TEMP_2 = D + E;
          Z = TEMP_2 + F;
     end
end
```

When the `A + B` addition is shared with the `TEMP_2 + F` addition on an adder called `R1` and the `D + E` addition is shared with the `TEMP_1 + C` addition on an adder called `R2`, a feedback loop results. The variable `TEMP_1` connects the output of `R1` to the input of `R2`. The variable `TEMP_2` connects the output of `R2` to the input of `R1`, resulting in a feedback loop. Figure 7-1 shows the circuit with the feedback loop highlighted.

Figure 7-1 Feedback Loop for Example 7-6



The circuit in Figure 7-1 is not faulty, because the multiplexing conditions never allow the entire path to be activated simultaneously. Still, the HDL Compiler resource sharing mechanism does not allow combinational feedback paths to be created, because most timing verifiers cannot handle them properly.

Errors

When HDL Compiler runs in automatic mode, the automatic sharing algorithm respects sharing restrictions. However, in manual mode or automatic sharing with manual controls mode, a directive can violate one of these restrictions. When a violation occurs, HDL Compiler displays an error message and ignores the directive. See “Resource Sharing Conflicts and Error Messages” on page 7-44 for more details.

Resource Sharing Methods

HDL Compiler offers three resource sharing methods:

- Automatic sharing
- Automatic sharing with manual controls
- Manual sharing

Automatic Resource Sharing

Using automatic resource sharing is the simplest way to share components and reduce the design area. This method is ideal if you do not know how you want to map the operations in your design onto hardware resources. In automatic sharing, HDL Compiler identifies the operations that can be shared. Design Compiler uses this information to minimize the area of your design, taking your constraints into consideration. If you want to override the automatically determined sharing, use automatic sharing with manual controls or manual sharing.

When resource sharing is enabled for a design, resources are allocated automatically the first time you compile that design. After the first compile, you can manually change the implementation of a resource with the `change_link` command.

To enable automatic sharing for all designs, set the dc_shell variable as shown before you execute the compile command.

```
dc_shell> hlo_resource_allocation = constraint_driven
```

The default value for this variable is `constraint_driven`.

To disable automatic sharing for uncompiled designs and enable resource sharing only for selected designs, enter the following commands:

```
dc_shell> hlo_resource_allocation = none
dc_shell> current_design = MY_DESIGN
dc_shell> set_resource_allocation constraint_driven
```

Source Code Preparation

You do not need to modify your Verilog source code.

Functional Description

The automatic sharing method minimizes the area of your design when it tries to meet your timing constraints. It identifies which operators are eligible to share resources and then evaluates various sharing configurations according to the area criteria.

Resource Area

Resource sharing reduces the number of resources in your design, which reduces the resource area. The area of a shared resource is a function of the types of operations that are shared on the resource and their bit-widths. The shared resource is made large enough to handle the largest of the bit-widths and powerful enough to perform all the operations.

Multiplexer Area

Resource sharing usually adds multiplexers to a design to channel values from different sources into a common resource input. In some cases, resource sharing reduces the number of multiplexers in a design. Example 7-7 shows a case in which shared operations have the same output targets, which results in fewer multiplexers.

Example 7-7 Shared Operations With the Same Output Target

```
always @(A or B or COND)
    begin
        if(COND)
            Z = A + B;
        else
            Z = A - B;
    end
```

When the addition and subtraction in Example 7-7 are not shared, a multiplexer selects whether the output of the adder or that of the subtracter is fed into Z. When they are shared, Z is fed from a single adder or subtracter and no multiplexing is necessary. If the inputs to the operations are different, multiplexers are added on the inputs of the adder or subtracter. HDL Compiler tends to share operations with common inputs and outputs to minimize multiplexer area.

Multiplexer area is a function of both the number of multiplexed values and the bit-widths of the values. Therefore, HDL Compiler tends to share operations with similar bit-widths.

Example of Shared Resources

Example 7-8 shows a simple Verilog program that adds either A and B or A and C; the addition depends on whether the condition ADD_B is true.

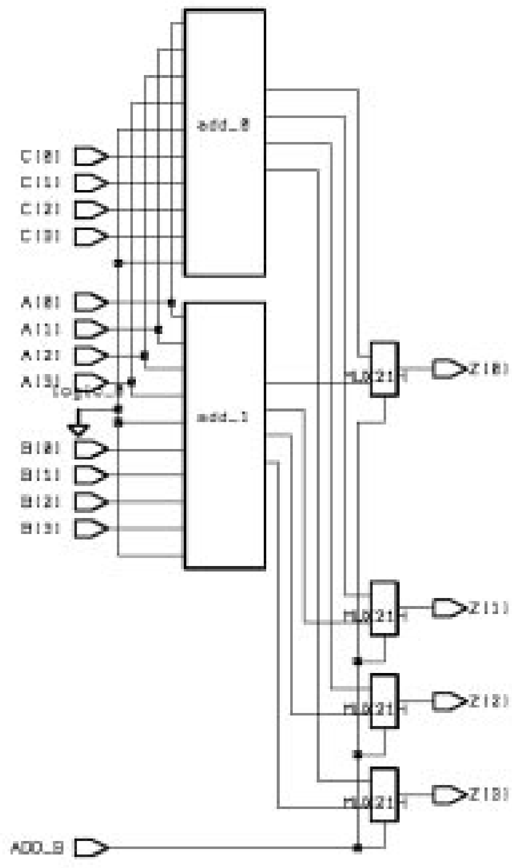
Example 7-8 Verilog Design With Two + Operators

```
module resources(A,B,C,ADD_B,Z);
input [4:0] A,B,C;
input ADD_B;
output [4:0] Z;

reg [4:0] Z;
```

```
always @(A or B or C or ADD_B)
begin
     if(ADD_B)
          Z = B + A;
     else
          Z = A + C;
end
endmodule
```

Figure 7-2 shows the schematic for Example 7-8 without resource sharing. Notice that two adders are built and that the outputs are multiplexed into Z.

Figure 7-2   Example 7-8 Design Without Resource Sharing



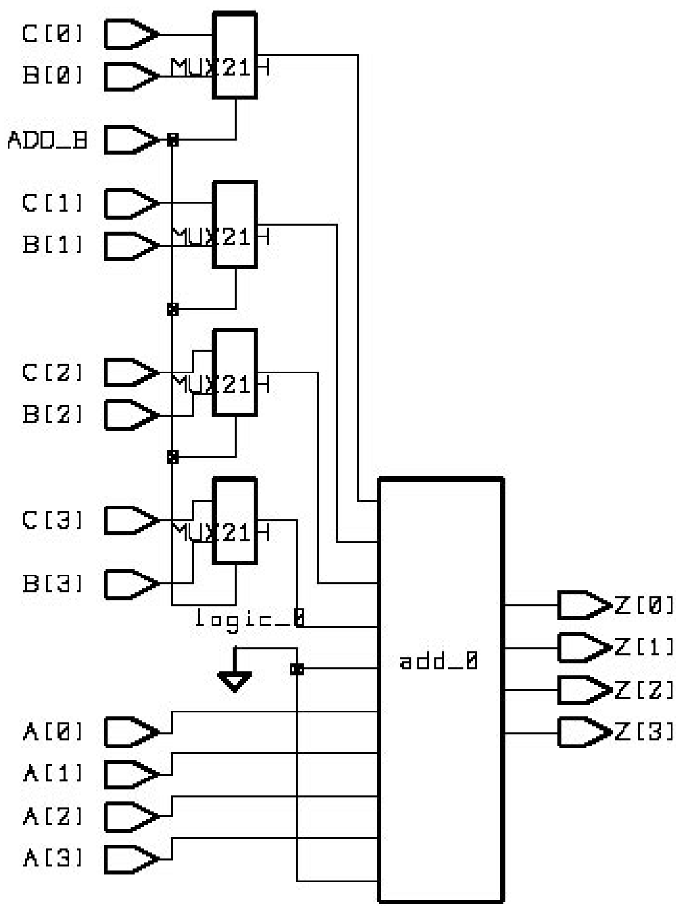
Input Ordering

Automatic sharing picks the best ordering of inputs to the resources to reduce the number of multiplexers required. In the following case, automatic sharing permutes `B + A` to `A + B`, then multiplexes `B` and `C`, and adds the output to `A`.

```
if (ADD_B) then
   Z = B + A
else
   Z = A + C...
```

Figure 7-3 shows the schematic for Example 7-8 that is produced by the use of automatic resource sharing. Notice that one adder is built with B and C multiplexed into one input and A fed directly into the other.

Figure 7-3 Example 7-8 Design With Automatic Resource Sharing



Automatic Resource Sharing With Manual Controls

In the automatic sharing with manual controls method, user directives influence the sharing configuration that HDL Compiler chooses automatically. You can control which sharing configurations are created or not created (regardless of their area savings or cost). You can use this method to solve a specific problem such as a violated timing constraint.

Manual controls allow you explicitly to

- Force the sharing of specified operations
- Prevent the sharing of specified operations
- Force specified operations to use a particular type of resource (such as an adder or a subtracter)

To control the sharing configuration, declare resources, and then specify whether the operations in your source file can be, must be, or must not be implemented on the resource. You can also indicate the type of hardware that implements the resource.

When you assign operations to the same resource, they are implemented on the same hardware. Operations you assign to a particular resource can also share hardware with other operations. Such sharing depends on the attributes you specify on the resource. You can force operations that are assigned to different resources to share the same hardware. To do this, declare a new resource that contains the resources you want to merge.

To use automatic resource sharing with manual controls, add the necessary resource sharing directives to your source files, then set the dc_shell variable as shown before you execute the `compile` command.

```
dc_shell> hlo_resource_allocation = constraint_driven
```

The default value for this variable is `constraint_driven`.

Source Code Preparation

Manual controls are incorporated in your Verilog source. Example 7-9 shows the code from Example 7-8 with the minimum controls you need in order to assign two operators to a resource. Two backslashes introduce each manual control statement.

Example 7-9   Sharing With Manual Controls

```
module TWO_ADDS_6 (A, B, C, Z, ADD_B);
     input[3:0] A, B, C;
     input ADD_B;
     output[3:0] Z;

reg[3:0] Z;
always @(A or B or C or ADD_B)
     begin : b1

     /* synopsys resource r0 : ops = ”A1 A2”;
     */

          if(ADD_B)
               Z = A + B;//synopsys label A1
          else
               Z = A + C;//synopsys label A2
     end
endmodule
```

To modify Example 7-8 for manual sharing, make the following changes (shown in the manual control statements in Example 7-9):

- Declare an identifier for an individual resource.

```
synopsys resource r0:
```

- Place labels on the operations.

```
if(ADD_B) then
     Z = A+B;// synopsys label A1
else
     Z = A+C;// synopsys label A2
```

- Use the `ops` directive to bind the labeled operations to the resource they share.

```
ops = ”A1 A2”;
```

Resources can be applied only to named blocks, such as

```
begin : b1
```

Note:

You cannot define resources in synchronous blocks. To use resource sharing with manual controls in a clocked design, put resource sharing directives in combinational blocks and assign states in synchronous (sequential) blocks.

Example 7-10 shows a resource defined within a synchronous block, and the resulting error message.

Example 7-10   Incorrectly Defining a Resource in a Synchronous Block

```
module adder (clk, rst, a, b, c);
input clk,rst;
input [5:0] a, b;
output [5:0] c;
reg [5:0] a, b, c;
always @(posedge clk or negedge rst)
begin : b0

    /* synopsys resource r1 :
         map_to_module = "add",
         implementation = "cla_add", ops = "op1";
    */

    if (!rst) c = 6'b0;
    else c = fctn (a, b);
end
function [5:0] fctn;
input [5:0] a, b;
begin : b1
fctn = a + b; //synopsys label op1
end
endfunction
endmodule
Error: syntax error at or near token 'resource' (File: /am/
remote/design/bad_res_share.v Line: 11) (VE-0)
```

The next section describes all the manual controls, along with their use and syntax.

Functional Description

In the automatic sharing with manual controls method, you add directives to your source file that influence which operations are shared and which are not shared. Next, HDL Compiler determines the exact sharing configuration that minimizes the area of your design and respects your directives.

The following descriptions explain how to use manual controls.

Verilog Resource Declarations and Identifiers

To make resource sharing directives, declare resources and identify attributes on those resources. You can make a resource declaration in a module, block, or function. In Verilog, you declare a resource with a compiler directive. The syntax is

```
//synopsys resource identifier
```

The identifier becomes the netlist cell name, unless the resource is merged with another resource.

Label Directive

Before operations in your source can be associated with resources, they must have unique labels. Assign a label with the `label` compiler directive. The syntax is

```
// synopsys label identifier
```

You can insert label directives in the following places: after a procedure call statement, after function calls, or after infix operations, as shown.

```
SWAP (IN_1, OUT_1);//synopsys label PROC_1
Z = ADD (B,C);//synopsys label FUNC_1
Z = A+B;      //synopsys label OP_1
```

You can also insert `label` directives after the name and left parenthesis of a task or function call, as shown.

```
SWAP (/*synopsys label PROC_1*/IN_1, OUT_1,);

Z = ADD (/*synopsys label FUNC_1*/B,C);

Z = A+ /*synopsys label OP_1*/ B;
```

The `label` directive applies to the operator most recently parsed. The operator to which a label applies is obvious in simple cases such as

```
a = b + c; //synopsys label my_oper
```

In an expression with multiple operators, the rules of precedence and associativity specified in the language determine the operator to which a label applies. In the following example, the label applies to the +, not the –, because the expression in the parentheses is evaluated first, so the + operator is parsed just before the label.

```
a = b + (c - d); //synopsys label my_oper
```

If you want the label to apply to the – operator, rewrite the expression as shown.

```
a = b + (c - /* synopsys label my_oper */ d);
```

To place multiple labels on a single statement, you can break your statement into multiple lines, as shown.

```
Z = a+       /* synopsys label ADD_1 */
      b+     /* synopsys label ADD_2 */
        c;

Z = ADD (     /* synopsys label PROC_1 */
      ADD (   /* synopsys label PROC_2 */
        A, B), C);
```

You can also use the /* synopsys label */ format to insert the label in the middle of a statement.

Keep labels unique within a function call or task.

Operations Directive

Assigning operations to resources in manual sharing is called binding. The `ops` directive binds operations and resources to a resource. It appears after the resource identifier declaration. The syntax is

```
/* synopsys resource resource_name:
    ops   = ”OP_ID RES_ID”;
*/
```

OP_ID and RES_ID are part of a list of operator or resource identifiers, separated by spaces, called the ops list. Example 7-11 shows how to use the `ops` directive.

Example 7-11   Using the ops Directive

```
always @(A or B or C or ADD_B)
    begin : b1
    /* synopsys resource r0 :
        ops = ”A1 A2”;
    */

        if(ADD_B)
            Z = A + B;// synopsys label A1
        else
            Z = A + C;// synopsys label A2
    end
```

If you use the same resource or operator identifier on more than one ops list, HDL Compiler generates an error message. One resource (the parent) can include another (the child) in its ops list, but only if the child resource (and any of its children) does not include the parent in its ops list. Example 7-12 shows an invalid ops list cycle with three resources.

Example 7-12 Invalid ops List Cycle

```
// synopsys resource r0 : ops = ”A1 r1”;
// synopsys resource r1 : ops = ”A2 r2”;
// synopsys resource r2 : ops = ”A0 r0”;
```

When you include a resource on the ops list, it is bound to the resource being declared, called the parent. The operations on the bound resource are realized on the parent resource, and the parent resource identifier is used for the name of the netlist cell. Example 7-21 on page 7-33 shows a resource contained in another resource.

map_to_module Directive

The `map_to_module` directive forces a resource to be implemented by a specific type of hardware module. Declare this directive after the resource declaration. The syntax is

```
/* synopsys resource resource_name:
     map_to_module = ”module_name”;
*/
```

`module_name` is the name of the module. You can set the implementation of a module with the `implementation` attribute, as described in the next section.

To list the module names and implementations in a synthetic library, use the command

```
dc_shell> report_synlib synthetic_library
```

synthetic_library is the name of a Synopsys synthetic library, such as standard.sldb. See the *DesignWare Databook* for more information.

Example 7-13 shows how to use the `map_to_module` directive.

Example 7-13   Using the map_to_module Directive

```
always @(A or B or C or ADD_B)
     begin : b1
          /* synopsys resource r0 :
               map_to_module = ”DW01_addsub”,
               ops = ”A1 A2”;
          */
```

HDL Compiler generates an error message if the indicated module cannot execute all operations bound to the resource. If you do not use `map_to_module` or if you do not give a module name, HDL Compiler selects the module as described in “Automatic Resource Sharing” on page 7-11.

implementation Attribute

The `implementation` attribute sets the initial implementation of a resource. If you use this attribute, it must follow the `map_to_module` directive. The syntax is

```
implementation = ”implementation_name”
```

implementation_name is the name of one of the implementations of the corresponding `map_to_module` `module`.

To list the module names and implementations in a synthetic library, use the command

```
dc_shell> report_synlib synthetic_library
```

synthetic_library is the name of a Synopsys synthetic library, such as standard.sldb.

Example 7-14 shows how to use the implementation attribute.

Example 7-14 Using the implementation Attribute

```
always @(A or B or C or ADD_B)
    begin : b1
        /* synopsys resource r0 :
            map_to_module = ”DW01_addsub”,
            implementation = ”rpl”,
            ops = ”A1 A2”;
      */
```

If `implementation` is not used or an implementation name is not given, HDL Compiler selects the module's implementation, as described in “Automatic Resource Sharing” on page 7-11. HDL Compiler reports an error if the associated module does not have the named implementation.

add_ops Directive

HDL Compiler guarantees that all operations in the ops list of a resource share the same hardware. Whether the hardware cell for a resource has additional operations bound to it depends on the area benefit of the additional sharing.

To direct HDL Compiler to evaluate whether to add more operations to a particular resource, use the `add_ops` directive. This directive must follow the declaration of the resource and can be applied only to a top-level resource. A top-level resource is one that is not included in another resource's ops list. The syntax is

```
/* synopsys resource resource_name:
    add_ops = ”true”|”false”;
*/
```

The default value is true. By default, HDL Compiler can merge the operations of a resource with other operations onto the same hardware. HDL Compiler merges additional operations if it reduces the area of your design. Additional operations can also be merged onto a resource by the addition of individual operations that are not bound to other resources (called free operations) or by the merging of two or more resources.

If you set the `add_ops` directive to false, the resource is assigned its own hardware, which cannot be used by other operations. In the code fragment in Example 7-15, resource r0 does not share hardware with operations other than `A1` and `A2`.

Example 7-15 Using the add_ops Directive

```
always @(A or B or C or ADD_B)
     begin : b1
     /* synopsys resource r0 :
          ops = ″A1 A2″,
          add_ops = ″false″;
   */
```

When `add_ops` is set to true, the resource can merge with any other resource that does not disallow sharing. To prevent automatic binding on a resource, set `add_ops` to false.

Note, however, that the `may_merge_with` and `dont_merge_with` directives override the `add_ops = ″false″` and `add_ops = ″true″` statements, respectively. These directives are discussed in detail in the following sections.

may_merge_with Directive

The `may_merge_with` directive overrides `add_ops = ″false″` for specific resources. The syntax is

```
/* synopsys resource resource_name:
     may_merge_with = ″{RES_2}″ | ″*″;
*/
```

RES_2 is a resource identifier, and `*` indicates all resources. The `may_merge_with` directive can be set either before or after `RES_2` is declared, but it must be set after *`resource_name`* is declared.

Note:

> You cannot use operation labels with the `may_merge_with` directive. To control the sharing of a labeled operation, put it in a resource.

In Example 7-16, resource `R1` can be shared only with resources `R2` and `R3`.

Example 7-16 Restricting Sharing With the may_merge_with Directive

```
always @(A or B or C or ADD_B)
begin : b1
     /* synopsys resource R1 :
          ops = ″A1 A2″,
          add_ops = ″false″,
          may_merge_with = ″R2 R3″;
     */
```

In Example 7-17, merging with resources is allowed but merging with free operations is not.

Example 7-17   Using the may_merge_with Directive

```
always @(A or B or C or ADD_B)
 begin : b1
    /* synopsys resource r1 :
         ops = "A1 A2",
         add_ops = "false",
         may_merge_with ="*";
    */
```

dont_merge_with Directive

The `dont_merge_with` directive overrides `add_ops = "true"` (the default). The syntax is

```
/* synopsys resource resource_name:
     dont_merge_with = "RES_ID" | "*";
*/
```

`RES_ID` is a resource identifier, and `*` indicates all resources. The `dont_merge_with` directive can be set either before or after `RES_ID` is declared but must be set after *`resource_name`* is declared.

Note:

> Do not use operation labels with the `dont_merge_with` directive. To control the sharing of a labeled operation, put it in a resource.

In Example 7-18, resource `R1` is allowed to share all resources except `R2` and `R3`.

Example 7-18 Restricting Sharing With the dont_merge_with Directive

```
always @(A or B or C or ADD_B)
begin : b1
    /* synopsys resource R1 :
        ops = ”A1 A2”,
        add_ops = ”true”,
        dont_merge_with = ”R2 R3”;
    */
```

In Example 7-19, merging with free operations is allowed but merging with resources is not.

Example 7-19 Using the dont_merge_with Directive

```
always @(A or B or C or ADD_B)
begin : b1
    /* synopsys resource r1 :
        ops = ”A1 A2”,
        add_ops = ”true”,
        dont_merge_with =”*”;
    */
```

If `may_merge_with` and `dont_merge_with` conflict, HDL Compiler issues an error message. Refer to “User Directive Conflicts” on page 7-44.

Operations and Resources

When you include a simple identifier in an ops list, HDL Compiler assumes that you are referring to a resource or a labeled operation in the current block or function. To refer to operations and resources declared in other functions that are called by the current block or function, use hierarchical naming or the `label_applies_to` directive. To refer to lower-level operations and resources directly, name the labels on the function calls that invoke the lower-level functions.

Hierarchical Naming

Hierarchical naming allows you to refer to resources and operations that are not defined in the current scope. You can use hierarchical names to share operations that occur in different functions if the functions are called from a single block. The syntax for a hierarchical name is

NAME/NAME

The first *NAME* identifies a labeled operation in the function or block in which the name is placed. The next *NAME* identifies a labeled operation in the called function. This can continue through an arbitrary number of function calls. The last *NAME* can refer to either a labeled operation or a resource.

Example 7-20 shows two + operations from different functions that are put in the same resource, which causes them to be shared.

Example 7-20 Hierarchical Naming for Two Levels

```
always @(A or B or C or COND)
begin : b1
    /* synopsys resource r0 :
        ops = ”L_1/ADD_1 L_2/ADD_2”;
    */
    if(COND)
        Z = CALC_1(A,B,C);// synopsys label L_1
    else
        Z = CALC_2(A,B,C);// synopsys label L_2
end

function [3:0] CALC_1;
    input [3:0] A, B, C;
    CALC_1 = (A +     // synopsys label ADD_1
                 B -  // synopsys label SUB_1
                 C);
endfunction

function [3:0] CALC_2;
    input [3:0] A, B, C;
    CALC_2 = (A -     // synopsys label SUB_2
                 B +  // synopsys label ADD_2
                 C);
endfunction
```

Example 7-21 shows a three-level hierarchical name.

Example 7-21 Hierarchical Naming for Three Levels

```
always @(A or B or C or COND)
  begin : b1
    /* synopsys resource R1 :
         ops = ”L_1/L_2/f1/R0”;
    */
    Z = CALC_1(A,B,C); // synopsys label L_1
  end

function [3:0] CALC_1;
  input [3:0] A, B, C;
  CALC_1 = CALC_2(A,B,C); // synopsys label L_2
endfunction

function [3:0] CALC_2;
  input [3:0] A, B, C;
  begin : f1
    /* synopsys resource R0 :
         ops = ”ADD_1 SUB_1”;
    */
    if (A < B)
         CALC_2 = (B + C);// synopsys label ADD_1
    else
       CALC_2 = (B - C); // synopsys label SUB_1
  end
endfunction
```

In Example 7-21, the function `CALC_2` has resources within a block. To refer to these resources, include the name of the block in the path name to the resource.

Each time a function is called, the operations in the function are replicated. To avoid extra hardware, you can refer to operations with hierarchical names and put them on the same resource. Example 7-22 shows how you can use `ops` attribute bindings with hierarchical names to reduce the number of cells created by function calls. If resource sharing is not used, each function call (L5, L6) creates a cell for each of the lower-level function operations (L1, L2, and L3), for a total of seven cells.

Example 7-22 Resource Sharing With Hierarchical Naming

```
module TOP (A, B, ADD, SUB, INC, SWITCH, Z);
input[3:0] A, B;
input ADD, SUB, INC, SWITCH;
output[3:0] Z;

reg[3:0] Z;
always begin : b1
        /* synopsys resource R2 :
             ops = ”L4 L5/f1/L1 L6/f1/L1 L5/R1 L6/R1”;
        */
        if(ADD)
             Z = A+B;                        // synopsys label L4
        else if (SWITCH)
             Z = sub_inc_dec (A, B, SUB, INC);  // synopsys label L5
        else
             Z = sub_inc_dec (B, A, SUB, INC);  // synopsys label L6
end

function [3:0] sub_inc_dec;
        input [3:0] A, B;
        input SUB, INC;
        /* synopsys resource R1 :
             ops = ”f1/L2 f1/L3”;
        */
        begin : f1
             if (SUB)
                  sub_inc_dec = (A-B);       // synopsys label L1
             else if (INC)
                  sub_inc_dec = (A+1’b1);   // synopsys label L2
             else
                  sub_inc_dec = (A-1’b1);   // synopsys label L3
        end
endfunction
endmodule
```

Example 7-22 has the following hierarchical naming details:

- The ops list for R1 binds the operations labeled L2 and L3 in the function `sub_inc_dec`.

- The ops list for R2 contains the operation L4, which is at the current scope.
- The ops list for R2 also contains L5/L1 and L6/L1, which identify each invocation of the A-B operation in the function `sub_inc_dec`.
- Finally, the ops list for R2 uses the names L5/R1 and L6/R1. You can use R1 as a shorthand notation to refer to all operations bound to R1. For example, L5/R1 refers to L5/L2 and L5/L3. When you use resource identifiers in hierarchical names, you avoid having to enter the labels under that resource.

label_applies_to Directive

As an alternative to using hierarchical naming, you can refer to lower-level operations and resources directly with the `label_applies_to` directive. Insert the `label_applies_to` directive in the declarations section of a function definition. Use this directive to name the label on the function call that invokes the lower-level function. The syntax is

```
// synopsys label_applies_to LABEL
```

LABEL identifies an operation or resource.

When you put a `label_applies_to` directive in a function definition, the label on any call to the function is equivalent to the operation or resource the label names. This is shown in Example 7-23.

Example 7-23 Using the label_applies_to Directive

```
module EX_D_14(A, B, Z);

input [3:0] A, B;
output[3:0] Z;

reg[3:0] Z;
always begin : b1
     /* synopsys resource r1 :
          ops = "L2";
     */
     Z = FUNC(A, B);        // synopsys label L2
end

function [3:0] FUNC;
     input [3:0] A, B;
     //synopsys label_applies_to L1
     begin
     FUNC = (A + B);        // synopsys label L1
endfunction
endmodule
```

In Example 7-23, resource R1 includes the A + B operation in its ops list by referring only to L2. The `label_applies_to` directive makes the L2 label apply to the L1 operation. Without the `label_applies_to` directive, the reference in the ops list is expressed hierarchically as L2/L1.

The `label_applies_to` directive can be used to make wrapper functions easier to use. A wrapper function computes its return value by calling another function. In some cases, the wrapper function makes a minor modification to the input or output. A simple example of a wrapper function is one that defines a new name for a function.

Suppose you have a function called `FOO`. Example 7-24 shows how you can define a function BAR that is equivalent to `FOO`.

Example 7-24 Using the label_applies_to Directive for Wrapper Functions

```
function [3:0] FOO;
 input [3:0] A, B;
 FOO = A+B;
endfunction

function [3:0] BAR;
  input [3:0] A, B;
  // synopsys label_applies_to REAL
  begin
    BAR = FOO(A,B);       // synopsys label REAL
  end
endfunction
```

Without the `label_applies_to` directive, FOO and BAR are not equivalent, because a hierarchical name that goes through the BAR function needs an additional reference to the REAL label. With the directive, this extra reference is not needed and FOO and BAR are equivalent.

Wrappers are often used to sign-extend data or to change data in some other way. Example 7-25 shows how `label_applies_to` connects a string of user-defined function calls to a single Verilog operator.

Example 7-25 Using the label_applies_to Directive With User-Defined Functions

```
module EX_D_20(A, B, Z);
input [3:0] A, B;
output[3:0] Z;

reg[3:0] Z;

always begin :b1
     /* synopsys
        resource R1 : ops = ”L1”;
     */
  Z = FUNC_1(A, B);         // synopsys label L1
end
function [3:0] FUNC_1;
  input [3:0] A, B;
  //synopsys label_applies_to L2
  begin
    FUNC_1 = FUNC_2(A,B);  // synopsys label L2
  end
endfunction

function [3:0] FUNC_2;
  input [3:0] A, B;
  //synopsys label_applies_to L3
  begin
    FUNC_2 = FUNC_3(A,B);  // synopsys label L3
  end
endfunction

function [3:0] FUNC_3;
  input [3:0] A, B;
  //synopsys label_applies_to L4
  begin
    FUNC_3 = A+B;          // synopsys label L4
  end
endfunction
endmodule
```

Example 7-25 has the following characteristics:

- Function FUNC_1 calls FUNC_2, which calls FUNC_3, and so on.
- A label_applies_to directive connects each level of the hierarchy to the next-lower level.
- The L1 identifier in the ops list points at L4. The equivalent hierarchical name is /L1/L2/L3/L4.

Example 7-26 uses a hierarchical name in a label_applies_to directive. The name A1/PLUS in the label_applies_to directive in function MY_ADD means that a reference to a label on a call to the function MY_ADD is equivalent to the L + R operation in the function MY_ADD_1.

Example 7-26 Using the label_applies_to Directive With Hierarchical Naming

```
module EX_D_21(A, B, C);
input [3:0] A, B;
output[3:0] C;

reg[3:0] C;

always begin :b1
        /* synopsys
           resource R0 : ops = ”A”;
        */
             C = MY_ADD(A, B);          // synopsys label A
end

function [3:0] MY_ADD;
        input [3:0] A, B;
        //synopsys label_applies_to A1/PLUS
        begin
             MY_ADD = MY_ADD_1(A,B);    // synopsys label A1
        end
endfunction

function [3:0] MY_ADD_1;
        input [3:0] L, R;
        begin
             MY_ADD_1 = L+R;            // synopsys label PLUS
        end
endfunction
endmodule
```

Manual Resource Sharing

Use manual sharing when you want to assign Verilog operators to resources but you do not want HDL Compiler to perform further sharing optimizations.

In manual sharing, you indicate all resource sharing with manual controls. As in automatic sharing with manual controls, these controls consist of

- Resource declarations that bind operators to resources and map resources to specific modules
- Compiler directives that label operations

You can bind as many operations to as many resources as you like. All operations bound to the same resource share the same hardware. The remaining operations are implemented with separate hardware.

To use the manual sharing method, add resource sharing directives to your source files and set the dc_shell variable as follows before you execute the `compile` command.

```
dc_shell> hlo_resource_allocation = none
```

This command disables automatic sharing. The default value for this variable is `constraint_driven`.

Source Code Preparation

Manual controls are incorporated in your Verilog source code.

Functional Description

In manual sharing, you are limited to a subset of the manual controls available for automatic sharing with manual controls. This subset of controls includes

- `label` directive
- `ops` directive
- `map_to_module` directive
- `label_applies_to` directive

See “Functional Description” on page 7-12 and “Operations and Resources” on page 7-30 for descriptions of these controls.

The following manual controls, used in automatic sharing with manual controls, are ignored in manual sharing:

- `add_ops` directive
- `may_merge_with` directive
- `dont_merge_with` directive

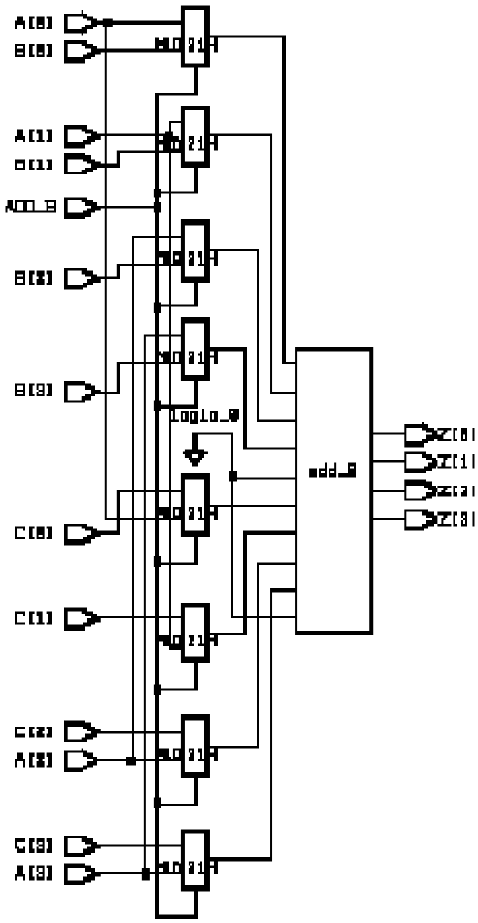
Input Ordering

In automatic sharing mode, HDL Compiler picks the best ordering of inputs to the cells to reduce the number of multiplexers required. In the following case, automatic sharing permutes B + A to A + B, then multiplexes B and C, and adds the output to A. (See Figure 7-3 on page 7-16.)

```
if (ADD_B)
    Z = B + A;
else
    Z = A + C;
end
...
```

In contrast, manual sharing does not optimize input ordering for resources. For example, suppose a resource is declared that forces the additions in the previous example onto the same adder. Under manual sharing, one input of the adder is fed by a multiplexer that chooses between A and B. The other input is fed by a multiplexer that chooses between A and C. This process is shown in Figure 7-4.

Figure 7-4   Manual Sharing With Unoptimized Inputs



To optimize input ordering with manual sharing, permute the inputs in the source code by rewriting B + A as A + B.

Remember that in manual sharing mode, operator instances that are not explicitly shared on resources are instantiated as new cells.

Resource Sharing Conflicts and Error Messages

Note:
Read “Resource Sharing Methods” on page 7-11 before you read this section.

For resource sharing, operators must be in the same `always` block. If they are not, a sharing conflict exists.

Other kinds of sharing conflicts can also prevent a resource from being shared—for example,

- User directive conflicts
- Module conflicts
- Control flow conflicts
- Data flow conflicts

With manual resource sharing, if the manual controls in your source create conflicts, they are reported as errors or warnings. In fully automatic sharing, HDL Compiler resolves these conflicts before the design is built, so no errors are reported.

User Directive Conflicts

User directive conflicts occur when manual controls that permit sharing contradict manual controls that prevent sharing. Note the user directive conflicts for resources R0 and R1 in the following example:

```
// synopsys resource R0: may_merge_with = ”R1”;
...
// synopsys resource R1: dont_merge_with = ”R0”;
```

HDL Compiler generates the following error message:

```
may_merge_with and dont_merge_with conflict
in resource ’R0’. may_merge_with ignored
```

However, the following directives for R0, R1, and R2 do not generate an error message:

```
// synopsys resource R0: may_merge_with = ”R1”;
...
// synopsys resource R1: may_merge_with = ”R2”;
...
// synopsys resource R2: dont_merge_with = ”R0”;
```

These directives do not conflict, because a `may_merge_with` directive does not mean that the resource will merge. The user directives are all satisfied if

- No sharing is done
- R0 and R1 are merged
- R1 and R2 are merged

The directives do not permit all three to be merged, because of the `dont_merge_with` directive on R2.

Module Conflicts

If a hardware module cannot implement all the operations bound to a resource assigned to it, a module conflict occurs. This conflict happens for two reasons:

- Inappropriate operations are mapped to a module that has a `map_to_module` directive, as shown in Example 7-27.

- Operators are bound to a resource that cannot be implemented by a single module.

Example 7-27 Module Conflict

```
always @(A or B or ADD_B)
begin : b1
    /* synopsys
        resource R0 :
            ops = ”A0”,
            map_to_module = ”sub”;
    */
    if (ADD_B)
        Z = A + B;        // synopsys label A0
    else
        Z = A - B;
end
```

In Example 7-27, a conflict occurs because the subtracter, sub, cannot perform addition. The error message is

```
Error: Module ’sub’ cannot implement all of the operations
in resource ’R0’
```

When resources are not mapped but operators are bound to a resource and no module can implement all the operations on that resource, the error message is

```
Error: There is no module which can implement all of the
operations in the resource ’R0’ in routine ADDER_1 line 12
in file ’/home/verilog/adder_1.v’
```

User-defined functions cannot be shared. If you attempt to share such functions, HDL Compiler generates an error message. Refer to “Scope and Restrictions” on page 7-2 for supported Verilog operators.

Control Flow Conflicts

As discussed in “Scope and Restrictions” on page 7-2, two operations can be shared only if no execution path exists from the start of the block to the end of the block that reaches both operations. Example 7-28 shows a control flow conflict.

Example 7-28 Control Flow Conflict

```
always @(A or B or C or D or ADD_B)
begin : b1
     /* synopsys
          resource R0 :
          ops = ”A1 A2”;
     */
     if(ADD_B) begin
          Y = A + B;         // synopsys label A1
          Z = C + D;         // synopsys label A2
     end
     else
          Z = A + C;
end
```

In Example 7-28, the + operations labeled A1 and A2 cannot be shared, because of a control flow conflict. HDL Compiler generates the following error message:

```
Error: Operations in resource ’R0’ can not be shared because
they may execute in the same control step in routine control
line 15 in file ’CONTROL.v’
```

If operations are in the same path in software (which creates a control flow conflict), they occur at the same time in hardware. Operations that occur at the same time require separate resources. Only disjoint operations can share resources.

Data Flow Conflicts

Combinational feedback that occurs as a result of resource sharing is not permitted. Example 7-29 shows a data flow conflict.

Example 7-29 Data Flow Conflict

```
always @(A or B or C or D or E or F or ADD_B)
begin : b1
     /* synopsys
          resource R0 : ops = ”K1 M4”;
          resource R1 : ops = ”K2 M3”;
     */
     if (ADD_B) begin
          X = A + B;    // synopsys label K1
          Y = X + C;    // synopsys label K2
     end
     else begin
          X = D + E;    // synopsys label M3
          Y = X + F;    // synopsys label M4
     end
end
```

In Example 7-29, the sharing mandated by resources R0 and R1 creates a feedback loop, as described in “Scope and Restrictions” on page 7-2. HDL Compiler generates the following error message:

```
Error: Operations in resource are part of a data flow cycle
in routine data line 15 in file ’DATA.v’
```

Reports

HDL Compiler generates reports that show the resource sharing configuration for a design. The resource report lists the resource name, the module, and the operations contained in the resource. You can generate this report for any resource sharing method. If you use manual controls, the information in the report makes it easier to explore design alternatives.

Generating Resource Reports

To display resource reports, read your design, compile it, then use the `report_resources` command as shown.

```
dc_shell> read -f verilog myfile.v
dc_shell> compile
dc_shell> report_resources
```

Interpreting Resource Reports

Example 7-30 shows the report that is generated for the following code. Resource sharing is not used.

```
always @(A or B or C or ADD_B)
 begin
    if(ADD_B)
       Z = B + A;
    else
       Z = A + C;
 end
```

Example 7-30 Resource Report Without Sharing

```
dc_shell> hlo_resource_allocation = none
dc_shell> read -f verilog example.v
dc_shell> compile
dc_shell> report_resources
```

Number of resource = 2

Resource	Module (impl)	Parameters	Contained Resources	Contained Operations
r30	DW01_add (cla)	n=4		add_9
r31	DW01_add (cla)	n=4		add_11

Example 7-31 shows the report for the same example after use of automatic sharing with manual controls.

Example 7-31 Resource Report Using Automatic Sharing With Manual Controls

Number of resource = 1

Resource	Module (impl)	Parameters	Contained Resources	Contained Operations
r23	DW01_add (cla)	n=4		add_11 add_9

Each report has five categories:

Resource

Identifies the cell in the final netlist. Where resources are bound to other resources, the parent resource name appears. In Example 7-30, two adders are created and two resource identifiers are shown in the report. Example 7-31, which uses resource sharing, has only one resource identifier. Both examples show the lines in the source code where the operations occur.

Module

Gives the name of the hardware module used by the resource. Example 7-30 has two adders; Example 7-31 has only one. The implementation name is shown as (impl) in the report and indicates the implementation that Design Compiler selected for the module.

Parameters

Identifies the bit-widths of the modules.

Contained Resources

Lists the names of resources bound to the parent resource, if any.

Contained Operations

Lists the operations that are shared on the resource.

8

Writing Circuit Descriptions

You can write many logically equivalent descriptions in Verilog to describe a circuit design. However, some descriptions are more efficient than others in terms of the synthesized circuit's area and speed. The way you write your Verilog source code can affect synthesis.

This chapter describes how to write a Verilog description to ensure an efficient implementation. Topics include

- How Statements Are Mapped to Logic
- Don't Care Inference
- Propagating Constants
- Synthesis Issues
- Designing for Overall Efficiency

Here are some general guidelines for writing efficient circuit descriptions:

- Restructure a design that makes repeated use of several large components, to minimize the number of instantiations.
- In a design that needs some, but not all, of its variables or signals stored during operation, minimize the number of latches or flip-flops required.
- Consider collapsing hierarchy for more-efficient synthesis.

How Statements Are Mapped to Logic

Verilog descriptions are mapped to logic by the creation of blocks of combinational circuits and storage elements. A statement or an operator in a Verilog function can represent a block of combinational logic or, in some cases, a latch or register.

When mapping complex operations, such as adders and subtracters, Design Compiler inserts arithmetic operators into the design as levels of hierarchy.

The description fragment shown in Example 8-1 represents four logic blocks:

- A comparator that compares the value of `b` with `10`
- An adder that has `a` and `b` as inputs
- An adder that has `a` and `10` as inputs
- A multiplexer (implied by the `if` statement) that controls the final value of `y`

Example 8-1   Four Logic Blocks

```
if (b < 10)
     y = a + b;
else
     y = a + 10;
```

The logic blocks created by HDL Compiler are custom-built for their environment. That is, if a and b are 4-bit quantities, a 4-bit adder is built. If a and b are 9-bit quantities, a 9-bit adder is built. Because HDL Compiler incorporates a large set of these customized logic blocks, it can translate most Verilog statements and operators.

Note:

> If the inputs to an adder or other operator resources are 4 bits or less, the hierarchy is automatically collapsed during the execution of the `compile` command.

Design Structure

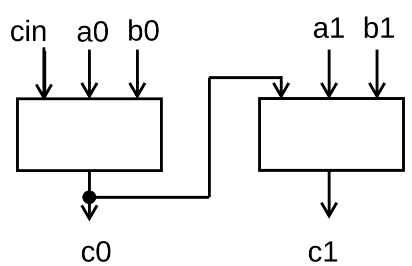
HDL Compiler provides significant control over the preoptimization structure, or organization of components, in your design. Whether or not your design structure is preserved after optimization depends on the Design Compiler options you select. Design Compiler automatically chooses the best structure for your design. You can view the preoptimized structure in the Design Analyzer window and then correlate it back to the original HDL source code.

You control structure by the way you order assignment statements and the way you use variables. Each Verilog assignment statement implies a piece of logic. The following examples illustrate two possible descriptions of an adder's carry chain. Example 8-2 results in a ripple carry implementation, as in Figure 8-1. Example 8-3 has more structure (gates), because the HDL source includes temporary registers, and it results in a carry-lookahead implementation, as in Figure 8-2.

Example 8-2   Ripple Carry Chain

```
// a is the addend
// b is the augend
// c is the carry
// cin is the carry in
c0 = (a0 & b0) |
     (a0 | b0) & cin;
c1 = (a1 & b1) |
     (a1 | b1) & c0;
```

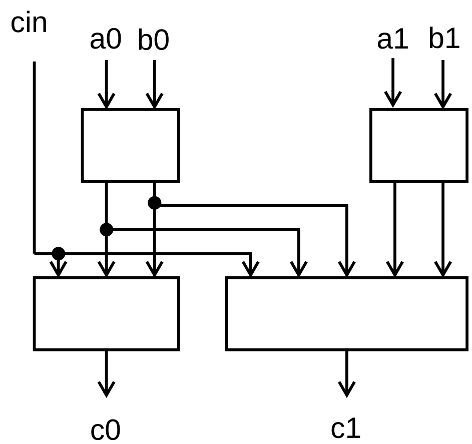
Figure 8-1   Ripple Carry Chain Implementation



Example 8-3   Carry-Lookahead Chain

```
// p’s are propagate
// g’s are generate
p0 = a0 | b0;
g0 = a0 & b0;
p1 = a1 | b1;
g1 = a1 & b1;
c0 = g0 | p0 & cin;
c1 = g1 | p1 & g0 |
         p1 & p0 & cin;
```

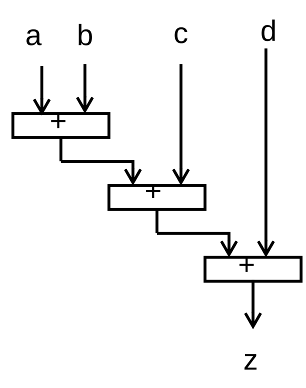
Figure 8-2 Carry-Lookahead Chain Implementation



You can also use parentheses to control the structure of complex components in a design. HDL Compiler uses parentheses to define logic groupings. Example 8-4 and Example 8-5 illustrate two groupings of adders. The circuit diagrams show how grouping the logic affects the way the circuit is synthesized. When Example 8-4 is parsed, `(a + b)` is grouped together by default, then c and d are added one at a time.

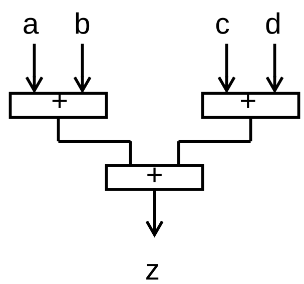
Example 8-4 4-Input Adder

```
z = a + b + c + d;
```



Example 8-5 4-Input Adder With Parentheses

```
z = (a + b) + (c + d);
```



Design Compiler considers other factors, such as signal arrival times, to determine which implementation is best for your design.

Note:

Manual or automatic resource sharing can also affect the structure of a design.

Using Design Knowledge

In many circumstances, you can improve the quality of synthesized circuits by better describing your high-level knowledge of a circuit. HDL Compiler cannot always derive details of a circuit architecture. Any additional architectural information you can provide to HDL Compiler can result in a more efficient circuit.

Optimizing Arithmetic Expressions

Design Compiler uses the properties of arithmetic operators (such as the associative and commutative properties of addition) to rearrange an expression so that it results in an optimized implementation. You can also use arithmetic properties to control the choice of implementation for an expression. Three forms of arithmetic optimization are discussed in this section:

- Merging cascaded adders with a carry
- Arranging expression trees for minimum delay
- Sharing common subexpressions

Merging Cascaded Adders With a Carry

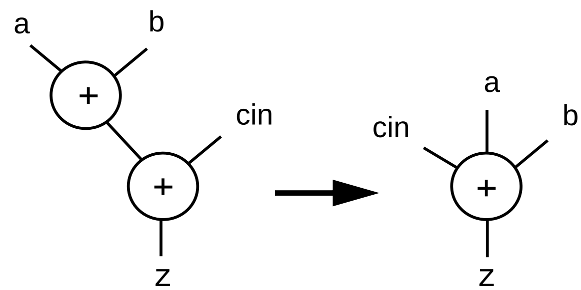
If your design has two cascaded adders and one has a bit input, HDL Compiler replaces the two adders with a simple adder that has a carry input. Example 8-6 shows two expressions in which `cin` is a bit variable connected to a carry input. Each expression results in the same implementation.

To infer cascaded adders with a carry input, set the variable to true (the default is false):

```
hdlin_use_cin = true
```

Example 8-6   Cascaded Adders With Carry Input

```
z <= a + b + cin;

t <= a + b;
z <= t + cin;
```



Arranging Expression Trees for Minimum Delay

If your goal is to speed up your design, arithmetic optimization can minimize the delay through an expression tree by rearranging the sequence of the operations. Consider the statement in Example 8-7.

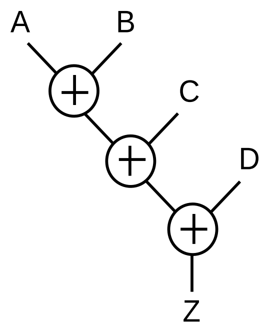
Example 8-7   Simple Arithmetic Expression

```
Z <= A + B + C + D;
```

The parser performs each addition in order, as though parentheses were placed as shown, and constructs the expression tree shown in Figure 8-3:

```
Z <= ((A + B) + C) + D;
```

Figure 8-3   Default Expression Tree



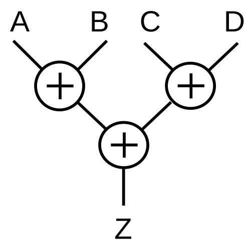
Considering Signal Arrival Times

To determine the minimum delay through an expression tree, Design Compiler considers the arrival times of each signal in the expression. If the arrival times of each signal are the same, the length of the critical path of the expression in Example 8-7 equals three adder delays. The critical path delay can be reduced to two adder delays if you add parentheses to the first statement as shown.

```
Z <= (A + B) + (C + D);
```

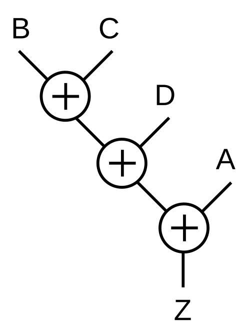
The parser evaluates the expressions in parentheses first and constructs a balanced adder tree, as shown in Figure 8-4.

Figure 8-4   Balanced Adder Tree (Same Arrival Times for All Signals)



Suppose signals `B`, `C`, and `D` arrive at the same time and signal `A` arrives last. The expression tree that produces the minimum delay is shown in Figure 8-5.

Figure 8-5   Expression Tree With Minimum Delay (Signal A Arrives Last)



Using Parentheses

You can use parentheses in expressions to exercise more control over the way expression trees are constructed. Parentheses are regarded as user directives that force an expression tree to use the groupings inside the parentheses. The expression tree cannot be rearranged to violate these groupings. If you are not sure about the best expression tree for an arithmetic expression, leave the expression ungrouped. Design Compiler can reconstruct the expression for minimum delay.

To illustrate the effect of parentheses on the construction of an expression tree, consider Example 8-8.

Example 8-8 Parentheses in an Arithmetic Expression

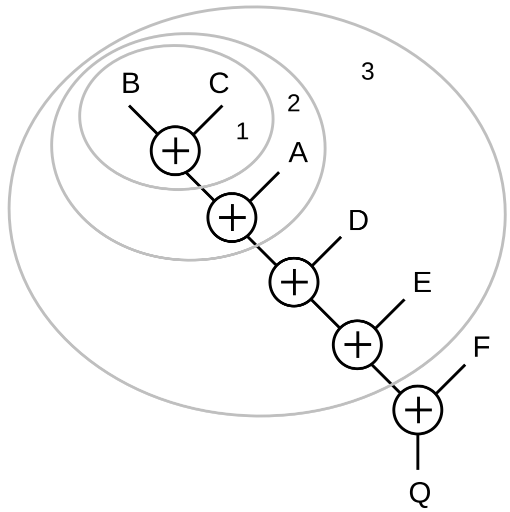
```
Q <= ((A + (B + C)) + D + E) + F;
```

The parentheses in the expression in Example 8-8 define the following subexpressions, whose numbers correspond to those in Figure 8-6:

1 (B + C)
2 (A + (B + C))
3 ((A + (B + C)) + D + E)

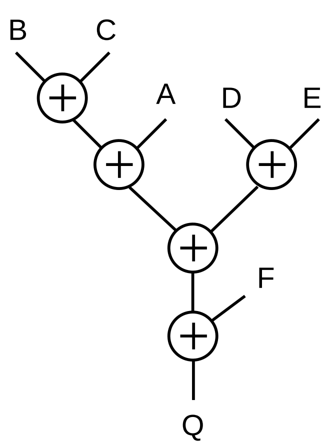
These subexpressions must be preserved in the expression tree. The default expression tree for Example 8-8 is shown in Figure 8-6.

Figure 8-6   Expression Tree With Subexpressions Dictated by Parentheses



Design Compiler restructures the expression tree in Figure 8-6 to minimize the delay and still preserve the subexpressions dictated by the parentheses. If all signals arrive at the same time, the result is the expression tree shown in Figure 8-7.

Figure 8-7   Restructured Expression Tree With Subexpressions Preserved



Design Compiler automatically optimizes expression trees to produce minimum delay. If you do not want HDL Compiler to optimize the expression trees in your design, enter the following command:

```
dc_shell> set_minimize_tree_delay false
```

The `set_minimize_tree_delay` command applies to the current design. The default for the command is true.

Considering Overflow Characteristics

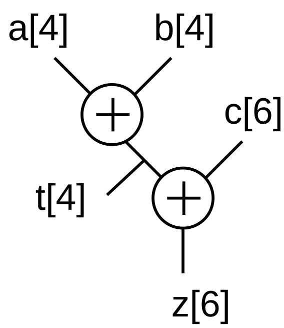
When Design Compiler performs arithmetic optimization, it considers how to handle the overflow from carry bits during addition. The optimized structure of an expression tree is affected by the bit-widths you declare for storing intermediate results. For example, suppose you write an expression that adds two 4-bit numbers and stores the result in a 4-bit register. If the result of the addition overflows the 4-bit output, the most significant bits are truncated. Example 8-9 shows how HDL Compiler handles overflow characteristics.

Example 8-9 Adding Numbers of Different Bit-Widths

```
t <= a + b;  // a and b are 4-bit numbers
z <= t + c;  // c is a 6-bit number
```

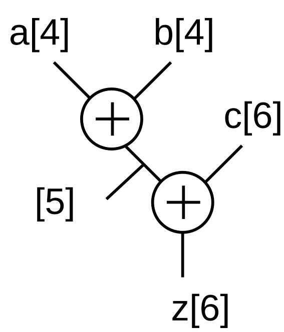
In Example 8-9, three variables are added (`a + b + c`). A temporary variable, `t`, holds the intermediate result of `a + b`. Suppose `t` is declared as a 4-bit variable so the overflow bits from the addition of `a + b` are truncated. The parser determines the default structure of the expression tree, which is shown in Figure 8-8.

Figure 8-8 Default Expression Tree With 4-Bit Temporary Variable



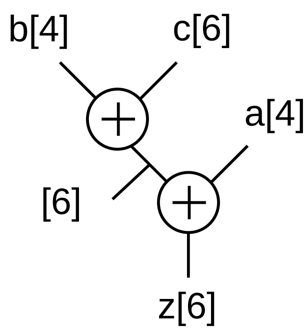
Now suppose the addition is performed without a temporary variable (z = a + b + c). HDL Compiler determines that five bits are needed to store the intermediate result of the addition, so no overflow condition exists. The results of the final addition might be different from the first case, where a 4-bit temporary variable is declared that truncates the result of the intermediate addition. Therefore, these two expression trees do not always yield the same result. The expression tree for the second case is shown in Figure 8-9.

Figure 8-9 Expression Tree With 5-Bit Intermediate Result



Now suppose the expression tree is optimized for delay and that signal a arrives late. The tree is restructured so that b and c are added first. Because c is declared as a 6-bit number, Design Compiler determines that the intermediate result must be stored in a 6-bit variable. The expression tree for this case, where signal a arrives late, is shown in Figure 8-10. Note how this tree differs from the expression tree in Figure 8-8.

Figure 8-10   Expression Tree for Late-Arriving Signal



Sharing Common Subexpressions

Subexpressions consist of two or more variables in an expression. If the same subexpression appears in more than one equation, you might want to share these operations to reduce the area of your circuit. You can force common subexpressions to be shared by declaring a temporary variable to store the subexpression, then use the temporary variable wherever you want to repeat the subexpression. Example 8-10 shows a group of simple additions that use the common subexpression (`a + b`).

Example 8-10   Simple Additions With a Common Subexpression

```
temp <= a + b;
x <= temp;
y <= temp + c;
```

Instead of manually forcing common subexpressions to be shared, you can let Design Compiler automatically determine whether sharing common subexpressions improves your circuit. You do not need to declare a temporary variable to hold the common subexpression in this case.

In some cases, sharing common subexpressions results in more adders being built. Consider Example 8-11, where `A + B` is a common subexpression.

Example 8-11   Sharing Common Subexpressions

```
if cond1
    Y <= A + B;
else
    Y <= C + D;
end;
if cond2
    Z <= E + F;
else
    Z <= A + B;
end;
```

If the common subexpression `A + B` is shared, three adders are needed to implement this section of code:

```
(A + B)
(C + D)
(E + F)
```

If the common subexpression is not shared, only two adders are needed: one to implement the additions `A + B` and `C + D` and one to implement the additions `E + F` and `A + B`.

Design Compiler analyzes common subexpressions during the resource sharing phase of the `compile` command and considers area costs and timing characteristics. To turn off the sharing of common subexpressions for the current design, enter the following command:

```
dc_shell> set_share_cse false
```

The default is true.

The HDL Compiler parser does not identify common subexpressions unless you use parentheses or write them in the same order. For example, the two equations in Example 8-12 use the common subexpression `A + B`.

Example 8-12   Unidentified Common Subexpressions

```
Y = A + B + C;
Z = D + A + B;
```

The parser does not recognize `A + B` as a common subexpression, because it parses the second equation as `(D + A) + B`. You can force the parser to recognize the common subexpression by rewriting the second `assignment` statement as

```
Z <= A + B + D;
```

or

```
Z <= D + (A + B);
```

Note:

> You do not have to rewrite the `assignment` statement, because Design Compiler recognizes common subexpressions automatically.

Using Operator Bit-Width Efficiently

You can improve circuits by using operators more carefully. In Example 8-13, the adder sums the 8-bit value of a with the lower 4 bits of `temp`. Although `temp` is declared as an 8-bit value, the upper 4 bits of `temp` are always `0`, so only the lower 4 bits of `temp` are needed for the addition.

You can simplify the addition by changing `temp` to `temp [3:0]`, as shown in Example 8-13. Now, instead of using eight full adders to perform the addition, four full adders are used for the lower 4 bits and four half adders are used for the upper 4 bits. This yields a significant savings in circuit area.

Example 8-13   More Efficient Use of Operators

```
module all (a,b,y);
input  [7:0] a,b;
output  [8:0] y;
function  [8:0] add_lt_10;
input  [7:0] a,b;
reg  [7:0] temp;
    begin
        if  (b < 10)
            temp = b;
        else
            temp = 10;
        add_lt_10 = a + temp [3:0]; // use [3:0] for temp
    end
endfunction
assign y = add_lt_10(a,b);
endmodule
```

Using State Information

When you build finite state machines, you can often specify a constant value of a signal in a particular state. You can write your Verilog description so that Design Compiler produces a more efficient circuit.

Example 8-14 shows the Verilog description of a simple finite state machine.

Example 8-14 A Simple Finite State Machine

```
module machine (x, clock, current_state, z);

input   x, clock;
output [1:0] current_state;
output  z;

reg [1:0]  current_state;
reg     z;
/* Redeclared as reg so they can be assigned to in always
statements. By default, ports are wires and cannot be
assigned to in 'always'
*/
reg [1:0] next_state;
reg previous_z;

parameter [1:0] set0  = 0,
     hold0 = 1,
     set1  = 2;

always @ (x or current_state) begin
    case (current_state)       //synopsys full_case
     /* declared full_case to avoid extraneous latches */
     set0:
         begin
         z = 0 ;       //set z to 0
         next_state = hold0;
         end
     hold0:
         begin
```

```
        z = previous_z;          //hold value of z
        if (x == 0)
             next_state = hold0;
        else
             next_state = set1;
        end
    set1:
        begin
        z = 1;                   //set z to 1
        next_state = set0;
        end
    endcase
end
always @ (posedge clock) begin
    current_state = next_state;
    previous_z    = z;
end
endmodule
```

In the state `hold0`, the output `z` retains its value from the previous state. To synthesize this circuit, a flip-flop is inserted to hold the state `previous_z`. However, you can make some assertions about the value of `z`. In the state `hold0`, the value of `z` is always `0`. This can be deduced from the fact that the state `hold0` is entered only from the state `set0`, where `z` is always assigned the value `0`.

Example 8-15 shows how the Verilog description can be changed to use this assertion, resulting in a simpler circuit (because the flip-flop for previous_z is not required). The changed line is shown in bold.

Example 8-15 Better Implementation of a Finite State Machine

```
module machine (x, clock, current_state, z);

input   x, clock;
output [1:0]current_state;
output  z;

reg [1:0] current_state;
reg     z;
/* Redeclared as reg so they can be assigned to in always
statements. By default, ports are wires and cannot be
assigned to in 'always'
*/
reg [1:0] next_state;

parameter [1:0] set0  = 0,
     hold0 = 1,
     set1  = 2;

always @ (x or current_state) begin
    case (current_state)   //synopsys full_case
     /* declared full_case to avoid extraneous latches */
     set0:
         begin
         z = 0 ;       //set z to 0
         next_state = hold0;
         end
     hold0:
         begin
         z = 0;    //hold z at 0
         if (x == 0)
              next_state = hold0;
         else
              next_state = set1;
         end
     set1:
         begin
         z = 1;         //set z to 1
         next_state = set0;
         end
    endcase
```

```
end
always @ (posedge clock) begin
    current_state = next_state;
end
endmodule
```

Describing State Machines

You can use an implicit state style or an explicit state style to describe a state machine. In the implicit state style, a clock edge (negedge or posedge) signals a transition in the circuit from one state to another. In the explicit state style, you use a constant declaration to assign a value to all states. Each state and its transition to the next state are defined under the `case` statement. Use the implicit state style to describe a single flow of control through a circuit (where each state in the state machine can be reached only from one other state). Use the explicit state style to describe operations such as synchronous resets.

Example 8-16 shows a description of a circuit that sums data over three clock cycles. The circuit has a single flow of control, so the implicit style is preferable.

Example 8-16 Summing Three Cycles of Data in the Implicit State Style (Preferred)

```
module sum3 ( data, clk, total );
input [7:0] data;
input clk;
output [7:0] total;

reg total;

always
begin
 @ (posedge clk)
          total = data;
 @ (posedge clk)
          total = total + data;
 @ (posedge clk)
          total = total + data;
end
endmodule
```

Note:

> With the implicit state style, you must use the same clock phase (either posedge or negedge) for each event expression. Implicit states can be updated only if they are controlled by a single clock phase.

Example 8-17 shows a description of the same circuit in the explicit state style. This circuit description requires more lines of code than Example 8-16 does, although HDL Compiler synthesizes the same circuit for both descriptions.

Example 8-17 Summing Three Cycles of Data in the Explicit State Style (Not Advisable)

```
module sum3 ( data, clk, total );
input [7:0] data;
input clk;
output [7:0] total;

reg total;
reg [1:0] state;

parameter S0 = 0, S1 = 1, S2 = 2;

always @ (posedge clk)
begin
   case (state)
   S0: begin
          total = data;
          state = S1;
       end
   S1: begin
          total = total + data;
          state = S2;
       end
   default : begin
          total = total + data;
          state = S0;
       end
   endcase
end
endmodule
```

Example 8-18 shows a description of the same circuit with a synchronous reset added. This example is coded in the explicit state style. Notice that the reset operation is addressed once before the `case` statement.

Example 8-18 Synchronous Reset—Explicit State Style (Preferred)

```
module SUM3 ( data, clk, total, reset );
input [7:0] data;
input clk, reset;
output [7:0] total;

reg total;
reg [1:0] state;

parameter S0 = 0, S1 = 1, S2 = 2;

always @ (posedge clk)
begin
   if (reset)
      state = S0;
   else
      case (state)
      S0: begin
             total = data;
             state = S1;
          end
      S1: begin
             total = total + data;
             state = S2;
          end
      default : begin
             total = total + data;
             state = S0;
          end
      endcase;
end
endmodule
```

Example 8-19 shows how to describe the same function in the implicit state style. This style is not as efficient for describing synchronous resets. In this case, the reset operation has to be addressed for every `always @` statement.

Example 8-19   Synchronous Reset—Implicit State Style (Not Advisable)

```
module SUM3 ( data, clk, total, reset );
input [7:0] data;
input clk, reset;
output [7:0] total;

reg total;

    always
        begin: reset_label

            @ (posedge clk)
            if (reset)
                begin
                    total = 8’b0;
                    disable reset_label;
                end
            else
                total = data;

            @ (posedge clk)
            if (reset)
                begin
                    total = 8’b0;
                    disable reset_label;
                end
            else
                total = total + data;

            @ (posedge clk)
            if (reset)
                begin
                    total = 8’b0;
                    disable reset_label;
                end
            else
                total = total + data;
        end
endmodule
```

Minimizing Registers

In an `always` block that is triggered by a clock edge, every variable that has a value assigned has its value held in a flip-flop.

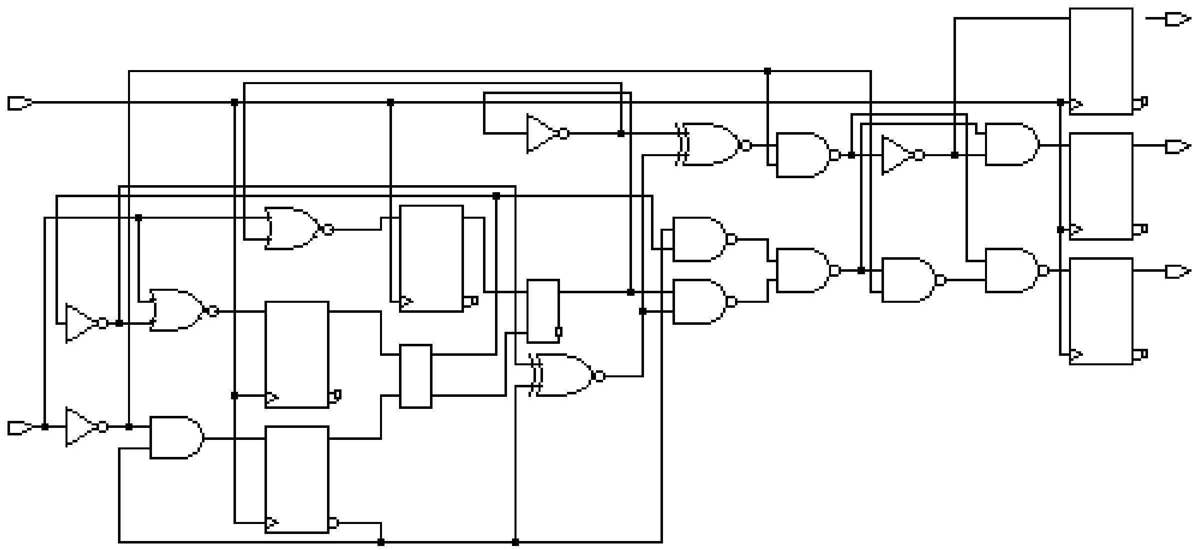
Organize your Verilog description so you build only as many registers as you need. Example 8-20 shows a description where extra registers are implied.

Example 8-20 Inefficient Circuit Description With Six Implied Registers

```
module count (clock, reset, and_bits, or_bits, xor_bits);
input clock, reset;
output and_bits, or_bits, xor_bits;
reg and_bits, or_bits, xor_bits;

reg [2:0] count;

    always @(posedge clock) begin
        if (reset)
            count = 0;
        else
            count = count + 1;

        and_bits = & count;
        or_bits  = | count;
        xor_bits = ^ count;
    end
endmodule
```

This description implies the use of six flip-flops: three to hold the values of `count` and one each to hold `and_bits`, `or_bits`, and `xor_bits`. However, the values of the outputs `and_bits`, `or_bits`, and `xor_bits` depend solely on the value of `count`. Because `count` is registered, there is no reason to register the three outputs. The synthesized circuit is shown in Figure 8-11.

Figure 8-11   Synthesized Circuit With Six Implied Registers



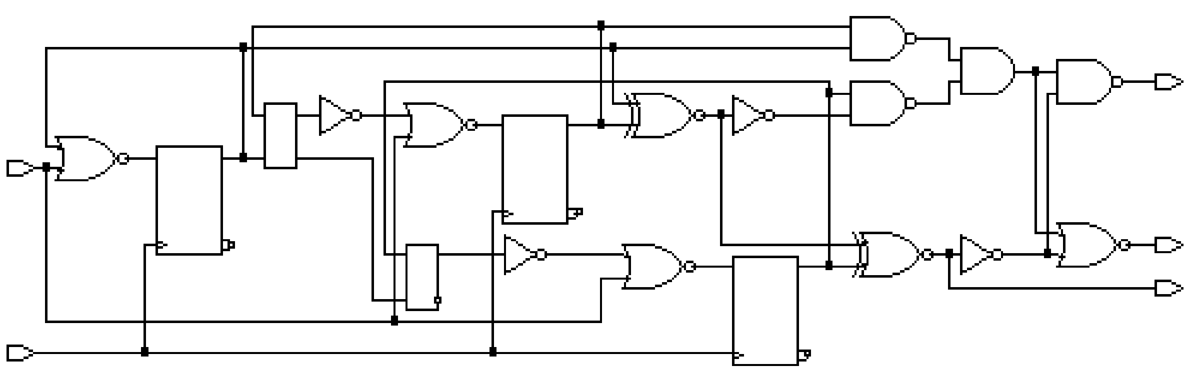
To avoid implying extra registers, you can assign the outputs from within an asynchronous `always` block. Example 8-21 shows the same logic described with two `always` blocks, one synchronous and one asynchronous, which separate registered or sequential logic from combinational logic. This technique is useful for describing finite state machines. Signal assignments in the synchronous `always` block are registered. Signal assignments in the asynchronous `always` block are not. Therefore, this version of the design uses three fewer flip-flops than the version in Example 8-20.

Example 8-21   Circuit With Three Implied Registers

```
module count (clock, reset, and_bits, or_bits, xor_bits);
input clock, reset;
output and_bits, or_bits, xor_bits;
reg and_bits, or_bits, xor_bits;

reg [2:0] count;

     always @(posedge clock) begin//synchronous
          if (reset)
               count = 0;
          else
               count = count + 1;
     end
     always @(count) begin//asynchronous
          and_bits = & count;
          or_bits  = | count;
          xor_bits = ^ count;
     end
endmodule
```

The more efficient version of the circuit is shown in Figure 8-12.

Figure 8-12   Synthesized Circuit With Three Implied Registers



Separating Sequential and Combinational Assignments

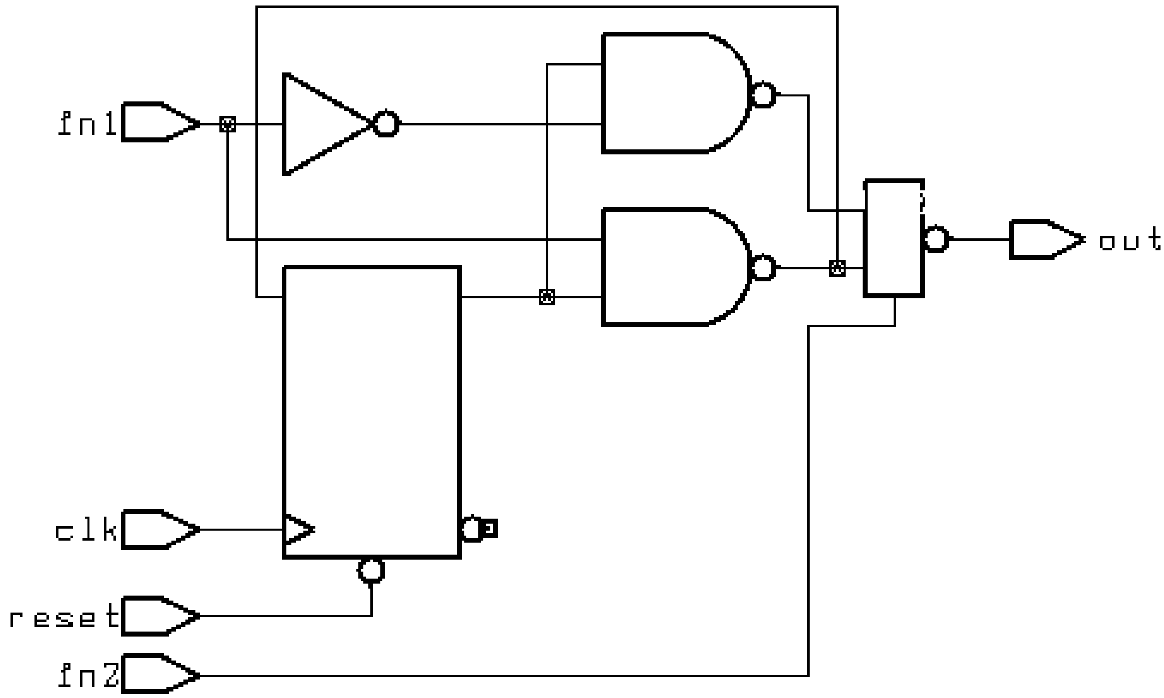
To compute values synchronously and store them in flip-flops, set up an `always` block with a signal edge trigger. To let other values change asynchronously, make a separate `always` block with no signal edge trigger. Put the assignments you want clocked in the `always` block with the signal edge trigger and the other assignments in the other `always` block. This technique is used for creating Mealy machines, such as the one in Example 8-22. Note that `out` changes asynchronously with `in1` or `in2`.

Example 8-22   Mealy Machine

```
module mealy (in1, in2, clk, reset, out);
     input in1, in2, clk, reset;
     output out;
     reg current_state, next_state, out;

     always @(posedge clk or negedge reset)
     // state vector flip-flops (sequential)
          if (!reset)
               current_state = 0;
          else
               current_state = next_state;

     always @(in1 or in2 or current_state)
     // output and state vector decode (combinational)

          case (current_state)
               0: begin
                        next_state = 1;
                        out = 1’b0;
                  end
               1: if (in1) begin

                        next_state = 1’b0;
                        out = in2;
                   end
                   else begin
                        next_state = 1’b1;
                        out = !in2;
                   end
          endcase

endmodule
```

The schematic for this circuit is shown in Figure 8-13.

Figure 8-13 Mealy Machine Schematic



Design Compiler Optimization

After HDL Compiler translates your design description, you then use Design Compiler to optimize the HDL description and synthesize the design.

Chapter 10, "Design Compiler Interface," describes how to use Design Compiler to read HDL descriptions through HDL Compiler. For a complete description of the Design Compiler `compile` command, see the Design Compiler documentation. For the syntax of Design Compiler commands, see the Synopsys man pages.

The Design Compiler commands `set_flatten` and `set_structure` set flatten and structure attributes for the compiler. Flattening reduces a design's logical structure to a set of two-level (and/or) logic equations. Structuring attempts to find common factors in the translated design's set of logic equations.

Don't Care Inference

You can greatly reduce circuit area by using don't care values. To use a don't care value in your design, create an enumerated type for the don't care value.

Don't care values are best used as default assignments to variables. You can assign a don't care value to a variable at the beginning of a module, in the default section of a `case` statement, or in the else section of an `if` statement.

To take advantage of don't care values during synthesis, use the Design Compiler command `set_flatten`. For information on embedding this command in your description, see "Embedding Constraints and Attributes" on page 9-22.

Limitations of Using Don’t Care Values

In some cases, using don’t care values as default assignments can cause these problems:

- Don’t care values create a greater potential for mismatches between simulation and synthesis.
- Defaults for variables can hide mistakes in the Verilog code.

  For example, you might assign a default don’t care value to VAR. If you later assign a value to VAR, expecting VAR to be a don’t care value, you might have overlooked an intervening condition under which VAR is assigned.

Therefore, when you assign a value to a variable (or signal) that contains a don’t care value, make sure that the variable (or signal) is really a don’t care value under those conditions. Note that assignment to an `x` is interpreted as a don’t care value.

Differences Between Simulation and Synthesis

Don’t care values are treated differently in simulation and in synthesis, and there can be a mismatch between the two. To a simulator, a don’t care is a distinct value, different from a `1` or a `0`. In synthesis, however, a don’t care becomes a `0` or a `1` (and hardware is built that treats the don’t care value as either a `0` or a `1`).

Whenever a comparison is made with a variable whose value is don’t care, simulation and synthesis can differ. Therefore, the safest way to use don’t care values is to

- Assign don’t care values only to output ports
- Make sure that the design never reads output ports

These guidelines guarantee that when you simulate within the scope of the design, the only difference between simulation and synthesis occurs when the simulator indicates that an output is a don't care value.

If you use don't care values internally to a design, expressions Design Compiler compares with don't care values (X) are synthesized as though values are not equal to X.

For example,

```
if A == 'X' then
...
```

is synthesized as

```
if FALSE then
...
```

If you use expressions comparing values with X, pre-synthesis and post-synthesis simulation results might not agree. For this reason, HDL Compiler issues the following warning:

```
Warning: A partial don't-care value was read in routine test
line 24 in file 'test.v'  This may cause simulation to
disagree with synthesis. (HDL-171)
```

Propagating Constants

Constant propagation is the compile-time evaluation of expressions that contain constants. HDL Compiler uses constant propagation to reduce the amount of hardware required to implement complex operators. Therefore, when you know that a variable is a constant, specify it as a constant. For example, a + operator with a constant of `1` as one of its arguments causes an incrementer, rather than a general adder, to be built. If both arguments of an operator are constants, no hardware is constructed, because HDL Compiler can calculate the expression’s value and insert it directly into the circuit.

Comparators and shifters also benefit from constant propagation. When you shift a vector by a constant, the implementation requires only a reordering (rewiring) of bits, so no logic is needed.

Synthesis Issues

The next two sections describe feedback paths and latches that result from ambiguities in signal or variable assignments, and asynchronous behavior.

Feedback Paths and Latches

Sometimes your Verilog source can imply combinational feedback paths or latches in synthesized logic. This happens when a signal or a variable in a combinational logic block (an `always` block without a `posedge` or `negedge clock` statement) is not fully specified. A variable or signal is fully specified when it is assigned under all possible conditions.

Synthesizing Asynchronous Designs

In a synchronous design, all registers use the same clock signal. That clock signal must be a primary input to the design. A synchronous design has no combinational feedback paths, one-shots, or delay lines. Synchronous designs perform the same function regardless of the clock rate, as long as the rate is slow enough to allow signals to propagate all the way through the combinational logic between registers.

Synopsys synthesis tools offer limited support for asynchronous designs. The most common way to produce asynchronous logic in Verilog is to use gated clocks on registers. If you use asynchronous design techniques, synthesis and simulation results might not agree. Because Design Compiler does not issue warning messages for asynchronous designs, you are responsible for verifying the correctness of your circuit.

The following examples show two approaches to the same counter design: Example 8-23 is synchronous, and Example 8-24 is asynchronous.

Example 8-23 Fully Synchronous Counter Design

```
module COUNT (RESET, ENABLE, CLK, Z);

     input RESET, ENABLE, CLK;
     output [2:0] Z;
     reg [2:0] Z;

always @ (posedge CLK) begin
     if (RESET) begin
          Z = 1’b0;
     end else if (ENABLE == 1’b1) begin
          if (Z == 3’d7) begin
               Z = 1’b0;
          end else begin
               Z = Z + 1’b1;
          end
     end
end

endmodule
```

Example 8-24 Asynchronous Counter Design

```
module COUNT (RESET, ENABLE, CLK, Z);

     input RESET, ENABLE, CLK;
     output [2:0] Z;
     reg [2:0] Z;
     wire GATED_CLK = CLK & ENABLE;

     always @ (posedge GATED_CLK or posedge RESET) begin
          if (RESET) begin
               Z = 1'b0;
          end else begin
               if (Z == 3'd7) begin
                    Z = 1'b0;
               end else begin
                    Z = Z + 1'b1;
               end
          end
     end
endmodule
```

The asynchronous version of the design uses two asynchronous design techniques. The first technique is to enable the counter by ANDing the clock with the enable line. The second technique is to use an asynchronous reset. These techniques work if the proper timing relationships exist between the asynchronous control lines (ENABLE and RESET) and the clock (CLK) and if the control lines are glitch-free.

Some forms of asynchronous behavior are not supported. For example, you might expect the following circuit description of a one-shot signal generator to generate three inverters (an inverting delay line) and a NAND gate.

```
X = A ~& (~(~(~ A)));
```

However, this circuit description is optimized to

```
     X = A ~& (~ A);
then
     X = 1;
```

Designing for Overall Efficiency

The efficiency of a synthesized design depends primarily on how you describe its component structure. The next two sections explain how to describe random logic and how to share complex operators.

Describing Random Logic

You can describe random logic with many different shorthand Verilog expressions. HDL Compiler often generates the same optimized logic for equivalent expressions, so your description style for random logic does not affect the efficiency of the circuit. Example 8-25 shows four groups of statements that are equivalent. (Assume that a, b, and c are 4-bit variables.) HDL Compiler creates the same optimized logic in all four cases.

Example 8-25   Equivalent Statements

```
c = a & b;

c[3:0] = a[3:0] & b[3:0];

c[3] = a[3] & b[3];
c[2] = a[2] & b[2];
c[1] = a[1] & b[1];
c[0] = a[0] & b[0];

for (i = 0; i <= 3; i = i + 1)
    c[i] = a[i] & b[i];
```

Sharing Complex Operators

You can use automatic resource sharing to share most operators. However, some complex operators can be shared only if you rewrite your source description more efficiently. These operators are

- Noncomputable array index
- Function call
- Shifter

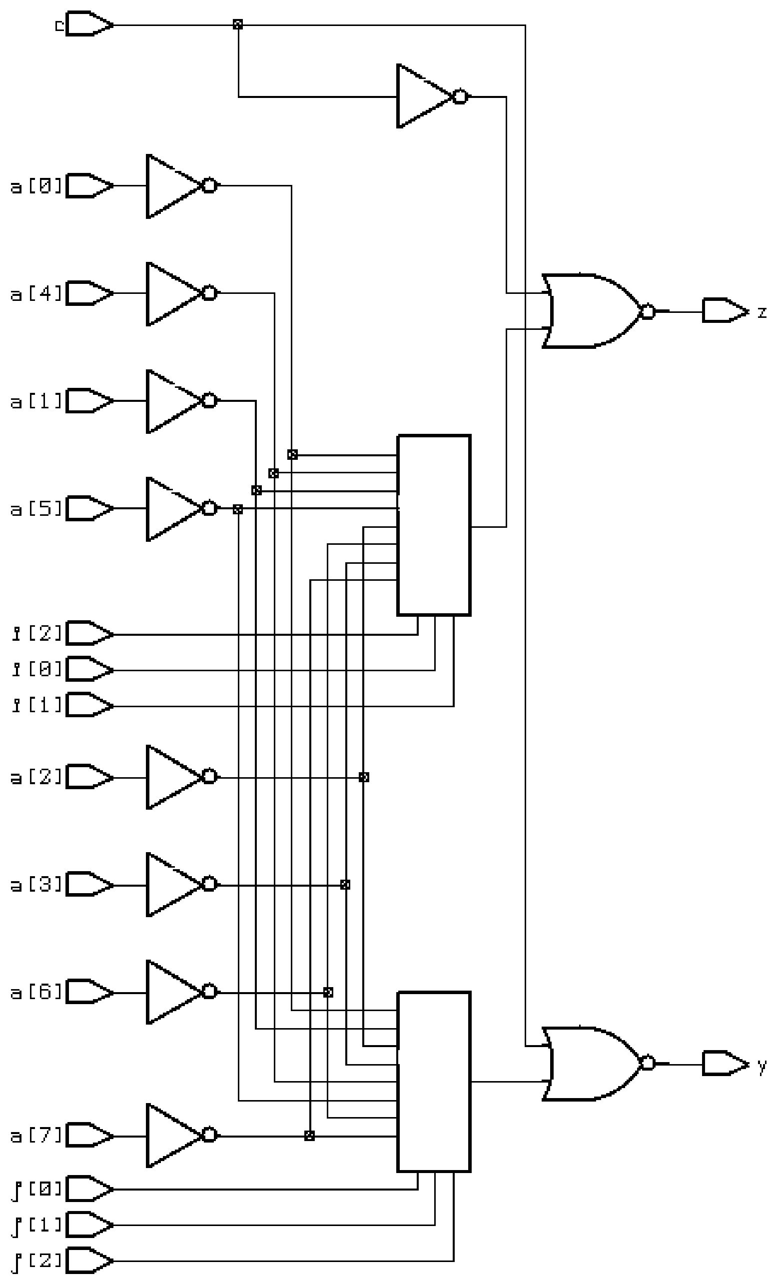
Example 8-26 shows a circuit description that creates more functional units than necessary when automatic resource sharing is turned off.

Example 8-26   Inefficient Circuit Description With Two Array Indexes

```
module rs(a, i, j, c, y, z);

 input [7:0] a;
 input [2:0] i,j;
 input c;

 output y, z;
 reg y, z;

 always @(a or i or j or c)
     begin
     z=0;
     y=0;
     if(c)
          begin
          z = a[i];
          end
     else
          begin
          y = a[j];
          end
     end
endmodule
```

The schematic for this code description is shown in Figure 8-14.

Figure 8-14 Circuit Schematic With Two Array Indexes



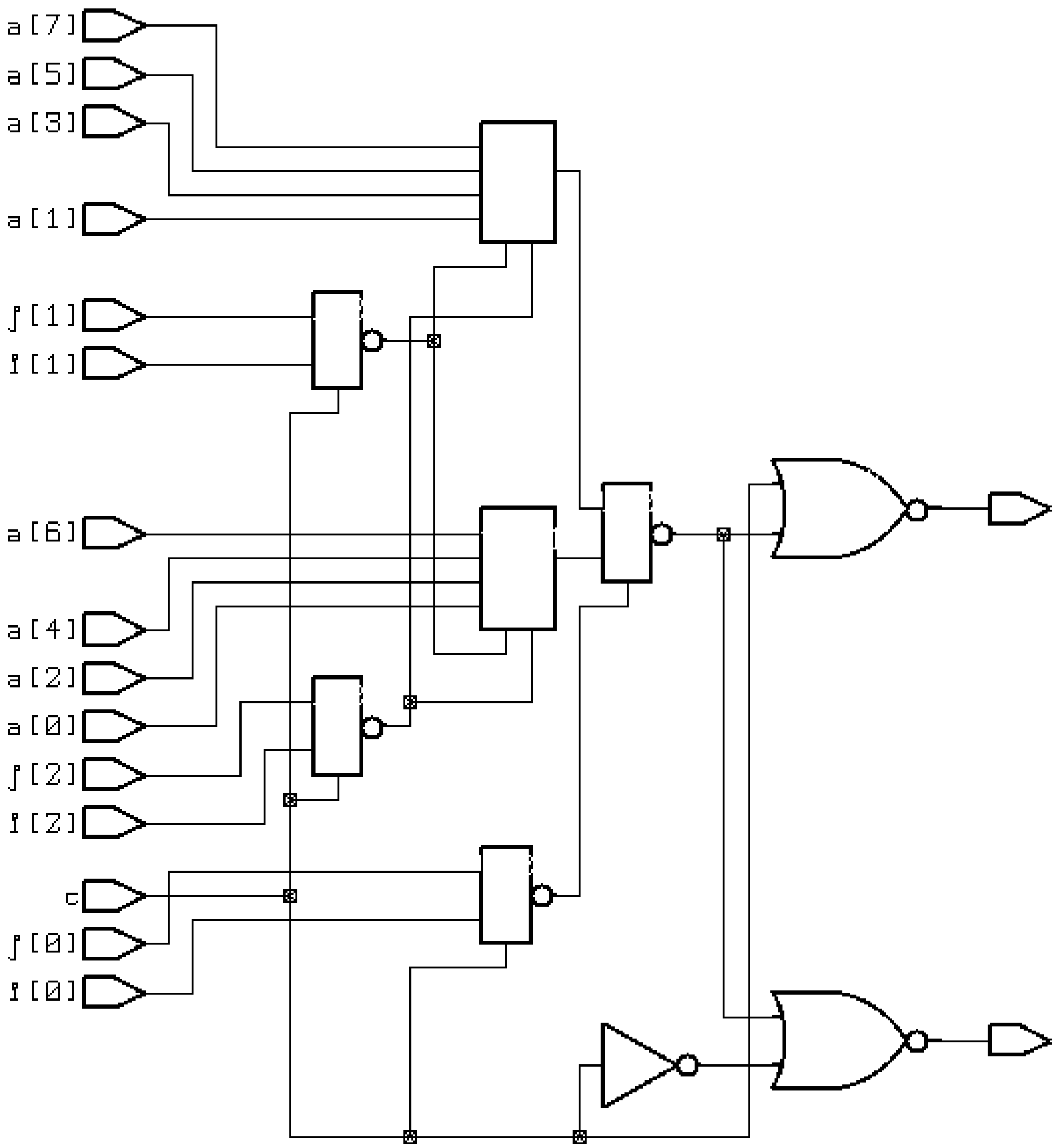
You can rewrite the circuit description in Example 8-26 so that it contains only one array index, as shown in Example 8-27.

Example 8-27 Efficient Circuit Description With One Array Index

```
module rs1(a, i, j, c, y, z);

    input [7:0] a;
    input [2:0] i,j;
    input c;

    output y, z;
    reg y, z;

    reg [3:0] index;
    reg temp;

    always @(a or i or j or c) begin
    if(c)
        begin
        index = i;
        end
    else
        begin
        index = j;
        end

    temp = a[index];

    z=0;
    y=0;
    if(c)
        begin
        z = temp;
        end
    else
        begin
        y = temp;
        end
    end

endmodule
```

The circuit description in Example 8-27 is more efficient than the one in Example 8-26 because it uses a temporary register, `temp`, to store the value evaluated in the `if` statement. The resulting schematic is shown in Figure 8-15.

Figure 8-15 Circuit Schematic With One Array Index



Consider resource sharing whenever you use a complex operation more than once. Complex operations include adders, multipliers, shifters (only when shifting by a variable amount), comparators, and most user-defined functions. If you use automatic resource allocation, adders, subtracters, and comparators can be shared. Chapter 7, “Resource Sharing,” covers these topics in detail.

9

HDL Compiler Directives

The Synopsys Verilog HDL Compiler translates a Verilog description to the internal format Design Compiler uses. Specific aspects of this process can be controlled by special comments in the Verilog source code called HDL Compiler directives. Because these directives are a special case of regular comments, they are ignored by the Verilog HDL Simulator and do not affect simulation.

This chapter describes HDL Compiler directives and their effect on translation, in the following sections:

- Verilog Preprocessor Directives
- Notation for HDL Compiler Directives
- translate_off and translate_on Directives
- parallel_case Directive
- full_case Directive

- state_vector Directive
- enum Directive
- template Directive
- Embedding Constraints and Attributes
- Component Implication

Verilog Preprocessor Directives

Verilog preprocessing provides the following features:

- `-define` option to the `analyze` command
- dc_shell variables
- The `‘ifdef`, `‘else`, and `‘endif` directives
- The DC Macro
- Extended capabilities for the `‘define` directive

Define Option to the analyze Command

An option to the `analyze` command, `-define` (or `-d`, abbreviated), allows macro definition on the command line.

You can use only one `-define` per `analyze` command. But the argument can be a list of macros, as shown in Example 9-1.

You do not need to use curly brackets to enclose one macro, as shown in Example 9-2.

Example 9-1 analyze Command With List of Defines

```
analyze -f verilog -d { RIPPLE, SIMPLE } mydesign.v
```

Example 9-2 analyze Command With One Define

```
analyze -f verilog -define ONLY_ONE mydesign.v
```

Note:

The `read` command does not accept the `-d` option.

The input to the `analyze` command continues to be a Verilog file. The output of the `analyze` command continues to be a .syn file.

dc_shell Variables

These variables perform the following functions:

`hdlin_preserve_vpp_files`

By default, this variable is false. When it is false, intermediate preprocessor files are deleted after use.

Preprocessor files are preserved (not deleted) when this flag is set to true.

`hdlin_vpp_temporary_directory`

Indicates where the intermediate preprocessor files are created. The default is to use the user’s WORK directory.

`hdlin_enable_vpp`

When set to true (the default), `hdlin_enable_vpp` allows interpretation of the `‘ifdef`, `‘else`, and `‘endif` directives.

It also activates `‘define` extensions, which allow macros with arguments.

‘ifdef, ‘else, and ‘endif Directives

The `‘ifdef`, `‘else`, and `‘endif` directives allow the conditional inclusion of code.

The macros that are arguments to the `‘ifdef` directives can also be defined in the Verilog source file by use of the `‘define` directive. In that case, there is no change in the invocation of the HDL Compiler to read in Verilog files. Example 9-3 shows a design that uses the directives.

Example 9-3   Design Using Preprocessor Directives and ‘define

```
‘ifdef SELECT_XOR_DESIGN

module selective_design(a,b,c);
input a, b;
output c;
  assign c = a ^ b;
endmodule

‘else

module selective_design(a,b,c);
input a, b;
output c;
  assign c = a | b;
endmodule

‘endif
```

DC Macro

The special macro DC is always defined, as in the following example:

Example 9-4 DC Macro

```
‘ifdef DC
     ...
     ...      /* Synthesis-only information */
     ...
‘else
     ...
     ...      /* Simulation-only information */
     ...
‘endif
```

The Verilog preprocessor directives are not affected by `translate_off` and `translate_on` (described in “translate_off and translate_on Directives” on page 9-6); that is, the preprocessor reads whatever is between `translate_off` and `translate_on`.

To suspend translation of the source code for synthesis, use the `‘ifdef, ‘else`, `‘endif` construct, not `translate_off` and `translate_on`.

‘define Verilog Preprocessor Directive

With the dc_shell variable `hdlin_enable_vpp` set to true, the `‘define` directive can specify macros that take arguments. For example,

```
‘define BYTE_TO_BITS(arg)((arg) << 3)
```

The `‘define` directive can do more than simple text substitution. It can also take arguments and substitute their values in its replacement text.

Notation for HDL Compiler Directives

The special comments that make up HDL Compiler directives begin, like all other Verilog comments, with the characters // or /*. The // characters begin a comment that fits on one line (most HDL Compiler directives do). 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 but only at the beginning of the first line. If the word following these characters is *synopsys* (all lowercase) or an alternative defined in Design Compiler with the `hdlin_pragma_keyword` variable, HDL Compiler treats the remaining comment text as a compiler directive.

Note:

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

translate_off and translate_on Directives

When the `// synopsys translate_off` and `// synopsys translate_on` directives are present, HDL Compiler suspends translation of the source code and restarts translation at a later point. Use these directives when your Verilog source code contains commands specific to simulation that HDL Compiler does not accept.

Note:

The Verilog preprocessor directives are not affected by `translate_off` and `translate_on`, and the preprocessor reads whatever is between them (see “DC Macro” on page 9-4).

You turn translation off by using

```
// synopsys translate_off
```

or

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

You turn translation back on by using

```
// synopsys translate_on
```

or

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

At the beginning of each Verilog file, translation is enabled. After that, you can use the `translate_off` and `translate_on` directives anywhere in the text. These directives must be used in pairs. Each `translate_off` must appear before its corresponding `translate_on`. Example 9-5 shows a simulation driver protected by a `translate_off` directive.

Example 9-5 // 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 “Full Case and Parallel Case” on page 5-19, 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 HDL Compiler to generate multiplexer logic instead.

The syntax for the `parallel_case` directive is

```
// synopsys parallel_case
```

or

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

In Example 9-6, the states of a state machine are encoded as one hot signal. If the `case` statement were implemented as a priority encoder, the generated logic would be unnecessarily complex.

Example 9-6 // 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, which could 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 “Full Case and Parallel Case” on page 5-19, 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, HDL Compiler 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 9-7 shows two uses of `full_case`. The `parallel_case` and `full_case` directives can be combined in one comment.

Example 9-7 // 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 Example 9-7) to indicate that other branch conditions do not occur. If you cover all possible conditions explicitly, HDL Compiler recognizes the `case` statement as full-case, so the `full_case` directive is not necessary.

The second `case` statement in Example 9-7 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 still must assign a value to each variable used in the case statement in all branches of the case statement. Example 9-8 illustrates a situation in which the full_case directive prevents a latch from being inferred for variable b but not for variable a.

Example 9-8 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 full_case 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 HDL Compiler assumes that the case statement is full_case.

state_vector Directive

The `// synopsys state_vector` directive labels a variable in a Verilog description as the state vector of an equivalent finite state machine.

The syntax for the `state_vector` directive is

```
// synopsys state_vector vector_name
```

or

```
/* synopsys state_vector vector_name */
```

The *`vector_name`* variable is the name chosen as a state vector. This declaration allows Synopsys Design Compiler to extract the labeled state vector from the Verilog description. Used with the `enum` directive, described in the next section, the `state_vector` directive allows you to define the state vector of a finite state machine (and its encodings) from a Verilog description. Example 9-9 shows one way to use the `state_vector` directive.

Caution!

Do not define two `state_vector` directives in one module. Although Design Compiler does not issue an error message, it recognizes only the first `state_vector` directive and ignores the second.

Example 9-9 // synopsys state_vector Example

```
reg [1:0] state, next_state;
// synopsys state_vector state

always @ (state or in) begin
     case (state) // synopsys full_case
          0: begin
               out = 3;
               next_state = 1;
               end
          1: begin
               out = 2;
               next_state = 2;
               end
          2: begin
               out = 1;
               next_state = 3;
               end
          3: begin
               out = 0
               if (in)
               next_state = 0;
               else
                    next_state = 3;
          endcase
     end

     always @ (posedge clock)
          state = next_state;
```

Note:

The `state_vector` directive works only with inferred flip-flops. You can also define the state vector and its encodings if you read in a state machine with instantiated flip-flops in HDL format and use embedded dc_shell scripts.

enum Directive

The `// synopsys enum` directive is designed for use with the Verilog parameter definition statement to specify state machine encodings. When a variable is marked as a `state_vector` (see “state_vector Directive” on page 9-13) and it is declared as an `enum`, the Synopsys HDL Compiler uses the enum values and names for the states of an extracted state machine.

The syntax of the `enum` directive is

```
// synopsys enum enum_name
```

or

```
/* synopsys enum enum_name */
```

Example 9-10 shows the declaration of an enumeration of type colors that is 3 bits wide and has the enumeration literals red, green, blue, and cyan with the values shown.

Example 9-10 Enumeration of Type Colors

```
parameter [2:0] // synopsys enum colors
red = 3’b000, green = 3’b001, blue = 3’b010, cyan = 3’b011;
```

The enumeration must include a size (bit-width) specification. Example 9-11 shows an invalid `enum` declaration.

Example 9-11 Invalid enum Declaration

```
parameter /* synopsys enum colors */
red = 3’b000, green = 1;
// [2:0] required
```

Example 9-12 shows a register, a wire, and an input port with the declared type of colors. In each of the following declarations, the array bounds must match those of the enumeration declaration. If you use different bounds, synthesis might not agree with simulation behavior.

Example 9-12 More enum Type Declarations

```
reg   [2:0]  /* synopsys enum colors */ counter;
wire  [2:0]  /* synopsys enum colors */ peri_bus;
input [2:0]  /* synopsys enum colors */ input_port;
```

Even though you declare a variable to be of type `enum`, it can still be assigned a bit value that is not one of the enumeration values in the definition. Example 9-13 relates to Example 9-12 and shows an invalid encoding for colors.

Example 9-13 Invalid Bit Value Encoding for Colors

```
counter = 3'b111;
```

Because `111` is not in the definition for colors, it is not a valid encoding. HDL Compiler accepts this encoding, because it is valid Verilog code, but Design Compiler recognizes this assignment as an invalid encoding and ignores it.

You can use enumeration literals just like constants, as shown in Example 9-14.

Example 9-14 Enumeration Literals Used as Constants

```
if (input_port == blue)
    counter = red;
```

You can also use enumeration with the `state_vector` directive. Example 9-15 shows how the `state_vector` variable is tagged by use of enumeration.

Example 9-15 Finite State Machine With // synopsys enum and // synopsys state_vector

```
// This finite-state machine (Mealy type) reads 1 bit
// per cycle and detects 3 or more consecutive 1s.

module enum2_V(signal, clock, detect);
input signal, clock;
output detect;
reg detect;

// Declare the symbolic names for states
parameter [1:0]//synopsys enum state_info
    NO_ONES = 2'h0,
    ONE_ONE = 2'h1,
    TWO_ONES = 2'h2,
    AT_LEAST_THREE_ONES = 2'h3;

// Declare current state and next state variables.
reg [1:0] /* synopsys enum state_info */   cs;
reg [1:0] /* synopsys enum state_info */   ns;

// synopsys state_vector cs

always @ (cs or signal)

    begin
        detect = 0;// default values
        if (signal == 0)
            ns = NO_ONES;
        else
            case (cs)              // synopsys full_case
                NO_ONES: ns = ONE_ONE;
                ONE_ONE: ns = TWO_ONES;
                TWO_ONES,
                AT_LEAST_THREE_ONES:
                    begin
                        ns = AT_LEAST_THREE_ONES;
                        detect = 1;
                    end
            endcase
    end
always @ (posedge clock) begin
    cs = ns;
end
endmodule
```

Enumerated types are designed to be used as whole entities. This design allows Design Compiler to rebind the encodings of an enumerated type more easily. You cannot select a bit or a part from a variable that has been given an enumerated type. If you do, the overall behavior of your design changes when Design Compiler changes the original encoding. Example 9-16 shows an unsupported bit-select.

Example 9-16 Unsupported Bit-Select From Enumerated Type

```
parameter [2:0] /* synopsys enum states */
     s0 = 3'd0, s1 = 3'd1, s2 = 3'd2, s3 = 3'd3,
     s4 = 3'd4, s5 = 3'd5, s6 = 3'd6, s7 = 3'd7;
reg [2:0] /* synopsys enum states */ state, next_state;

assign high_bit = state[2];// not supported
```

Because you cannot access individual bits of an enumerated type, you cannot use component instantiation to hook up single-bit flip-flops or three-states. Example 9-17 shows an example of this type of unsupported bit-select.

Example 9-17 Unsupported Bit-Select (With Component Instantiation) From Enumerated Type

```
DFF ff0 ( next_state[0], clk, state[0] );
DFF ff1 ( next_state[1], clk, state[1] );
DFF ff2 ( next_state[2], clk, state[2] );
```

To create flip-flops and three-states for `enum` values, you must imply them with the posedge construct or the literal `z`, as shown in Example 9-18.

Example 9-18 Using Inference With Enumerated Types

```
parameter [2:0] /* synopsys enum states */
    s0 = 3'd0, s1 = 3'd1, s2 = 3'd2, s3 = 3'd3,
    s4 = 3'd4, s5 = 3'd5, s6 = 3'd6, s7 = 3'd7;
reg [2:0] /* synopsys enum states */ state, next_state;

parameter [1:0] /* synopsys enum outputs */
    DONE = 2'd0, PROCESSING = 2'd1, IDLE = 2'd2;
reg [1:0] /* synopsys enum outputs */ out, triout;

always @ (posedge clk) state = next_state;
assign triout = trienable ? out : 'bz;
```

If you use the constructs shown in Example 9-18, you can change the enumeration encodings by changing the `parameter` and `reg` declarations, as shown in Example 9-19. You can also allow HDL Compiler to change the encodings.

Example 9-19 Changing the Enumeration Encoding

```
parameter [3:0] /* synopsys enum states */
    s0 = 4'd0, s1 = 4'd10, s2 = 4'd15, s3 = 4'd5,
    s4 = 4'd2, s5 = 4'd4, s6 = 4'd6, s7 = 4'd8;
reg [3:0] /* synopsys enum states */ state, next_state;

parameter [1:0] /* synopsys enum outputs */
    DONE = 2'd3, PROCESSING = 2'd1, IDLE = 2'd0;
reg [1:0] /* synopsys enum outputs */ out, triout;

always @ (posedge clk) state = next_state;
assign triout = trienable ? out : 'bz;
```

If you must select individual bits of an enumerated type, you can declare a temporary variable of the same size as the enumerated type. Assign the enumerated type to the variable, then select individual bits of the temporary variable. Example 9-20 shows how this is done.

Example 9-20 Supported Bit-Select From Enumerated Type

```
parameter [2:0] /* synopsys enum states */
    s0 = 3'd0, s1 = 3'd1, s2 = 3'd2, s3 = 3'd3,
    s4 = 3'd4, s5 = 3'd5, s6 = 3'd6, s7 = 3'd7;
reg [2:0] /* synopsys enum states */ state, next_state;
wire [2:0] temporary;

assign temporary = state;
assign high_bit = temporary[2]; //supported
```

Note:

Selecting individual bits from an enumerated type is not recommended.

If you declare a port as a `reg` and as an enumerated type, you must declare the enumeration when you declare the port. Example 9-21 shows the declaration of the enumeration.

Example 9-21 Enumerated Type Declaration for a Port

```
module good_example (a,b);

parameter [1:0] /* synopsys enum colors */
    green = 2'b00, white = 2'b11;
input a;
output [1:0] /* synopsys enum colors */ b;
reg [1:0] b;
.
.
endmodule
```

Example 9-22 shows the wrong way to declare a port as an enumerated type, because the `enumerated type` declaration appears with the `reg` declaration instead of with the `output port` declaration. This code does not export enumeration information to Design Compiler.

Example 9-22 Incorrect Enumerated Type Declaration for a Port

```
module bad_example (a,b);

parameter [1:0] /* synopsys enum colors */
     green = 2’b00, white = 2’b11;
input a;
output [1:0] b;
reg [1:0] /* synopsys enum colors */ b;
.
.
endmodule
```

template Directive

The `// synopsys template` directive overrides the setting of the `hdlin_auto_save_templates` variable. If you use this directive and your design contains parameters, the design is archived as a template. Example 9-23 shows how to use the directive.

Example 9-23 // synopsys template Directive in a Design With a Parameter

```
module template (a, b, c);
input a, b, c;
// synopsys template
parameter width = 8;
.
.
.
endmodule
```

See “Module Instantiations” on page 3-16 for more information.

Embedding Constraints and Attributes

Constraints and attributes, usually entered at the dc_shell prompt, can be embedded in your Verilog source code. Prefix the usual constraint or attribute statement with the Verilog comment characters `//`, and delimit the embedded statements with the compiler directives `// synopsys dc_script_begin` and `// synopsys dc_script_end`. The method is shown in Example 9-24.

Example 9-24   Embedding Constraints and Attributes With // Delimiters

```
...
// synopsys dc_script_begin
// max_area 0.0
// set_drive -rise 1 port_b
// max_delay 0.0 port_z
// synopsys dc_script_end
...
```

Constraints and attributes as shown in Example 9-25 can also be delimited with the characters `/*` and `*/`. When you use these delimiters, the `// synopsys dc_script_end` comment is not required or valid, because the attributes or constraints are terminated by `*/`.

Example 9-25   Embedding Constraints and Attributes With / and */ Delimiters*

```
/* synopsys dc_script_begin
   max_area 10.0
   max_delay 0.0 port_z
*/
```

The dc_shell script interprets the statements embedded between the `// synopsys dc_script_begin` and the `// synopsys dc_script_end` directives. If you want to comment out part of your dc_shell script, use the convention for comments that dc_shell uses.

Limitations on the Scope of Constraints and Attributes

The following limitations apply to the use of constraints and attributes in your design:

- Constraints and attributes declared outside a module apply to all subsequent modules declared in the file.
- Constraints and attributes declared inside a module apply only to the enclosing module.
- Any dc_shell scripts embedded in functions apply to the whole module.
- Include in your dc_shell script only commands that set constraints and attributes. Do not use action commands such as `compile`, `gen`, and `report`.
- The constraints or attributes set in the embedded script go into effect after the `read` command is executed. Therefore, variables that affect the read process itself are not in effect before the read. Thus, if you set the variable `hdlin_no_latches = true` in the embedded script, this variable does not influence latch inference in the current read.
- dc_shell performs error checking after the `read` command finishes. Syntactic and semantic errors in dc_shell strings are reported at this time.

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 HDL Compiler to recognize the netlist as a function being implemented by another module. When this subprogram is invoked in the behavioral code, HDL Compiler instantiates the module (see Example 9-26 on page 9-25).

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, `// synopsys return_port_name`, identifies a return port (functions in Verilog do not have output ports). To instantiate the function as a component, the return port must have a name. 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 whereas the indicated module is instantiated. Ensure that the functionality of the module instantiated in this way and the function it replaces are the same; otherwise, pre-synthesis and post-synthesis simulation do not match.

Example 9-26 illustrates the `map_to_module` and `return_port_name` directives.

Example 9-26   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
```

10

Design Compiler Interface

This chapter discusses the Design Compiler interface to Synopsys HDL Compiler for Verilog. It covers the following topics:

- Starting Design Compiler
- Reading In Verilog Source Files
- Optimizing With Design Compiler
- Busing
- Correlating HDL Source Code to Synthesized Logic
- Writing Out Verilog files
- Setting Verilog Write Variables

The Design Analyzer tool provides the graphic interface to the Synopsys synthesis tools. Design Analyzer reads in, synthesizes, and writes out Verilog source files, among others, calling Design Compiler for these functions. When you view a synthesized schematic in Design Analyzer, you can use the RTL Analyzer tool to see how the Verilog source code corresponds to its synthesized entities and gates. For more information, see the *RTL Analyzer User Guide*.

This chapter describes the commands and variables you use to read Verilog designs. It also explains how to specify synthesis attributes and constraints for compilation and how to write out designs in Verilog format.

Note:

> To understand this chapter, you must be familiar with Design Compiler concepts, especially synthesis attributes and constraints. For more information, see the Design Compiler documentation.

Starting Design Compiler

Design Compiler has two interfaces: a command-based interface (dc_shell) and a graphical user interface (Design Analyzer).

Starting the dc_shell Command Interface

Start the Design Compiler command interface by entering the invocation command `dc_shell` at your UNIX prompt.

```
% dc_shell
             Design Analyzer (TM)
           Behavioral Compiler (TM)
             DC Professional (TM)
                DC Expert (TM)
                 DC Ultra (TM)
              FPGA Compiler (TM)
              VHDL Compiler (TM)
               HDL Compiler (TM)
             Library Compiler (TM)
              Power Compiler (TM)
              Test Compiler (TM)
             Test Compiler Plus (TM)
                 CTV-Interface
               ECO Compiler (TM)
            DesignWare Developer (TM)
                DesignTime (TM)
                DesignPower (TM)

          Version 2000.05 -- May 18, 2000
   Copyright (c) 1999-2000 by Synopsys, Inc.
               ALL RIGHTS RESERVED

This program is proprietary and confidential information
of Synopsys, Inc., and may be used and disclosed only as
authorized in a license agreement controlling such use and
disclosure.
Initializing...
```

When Design Compiler has finished initializing, the command-line prompt appears.

```
Initializing...
dc_shell>
```

Starting Design Analyzer

Start Design Analyzer by entering the invocation command `design_analyzer` at your UNIX prompt, in an X windows command window. As in most UNIX programs, you can use the ampersand (&) to execute Design Analyzer in the background.

```
% design_analyzer &
```

The main Design Analyzer window appears. For complete information on using Design Analyzer, see the *Design Analyzer Reference Manual*.

Design Analyzer also provides access to the dc_shell command interface, through the Setup menu's Command Window selection.

The rest of this chapter describes the commands and the menu selections you use when working with Verilog source files and designs.

Reading In Verilog Source Files

Use the Design Compiler `read` command to read in Verilog design files.

```
dc_shell> read -format verilog {file_1, file_2, file_n}
```

Use the Design Analyzer File/Read dialog box to read in Verilog design files.

All of the `read` command options are described in the Design Compiler documentation and in the `read` man page. In the next section, “Reading Structural Descriptions,” however, we include a description of the `-netlist` option for reading structural Verilog files. You might want to use this option to save time.

Reading Structural Descriptions

To read in a Verilog structural description—that is, one that contains only module instantiations and no `always` blocks or continuous assignments—use the `-netlist` option with the `read` command in dc_shell. When the `-netlist` option is present, HDL Compiler reads structural descriptions faster and uses less memory. The syntax is

```
dc_shell> read -f verilog -netlist my_file.v
```

Note:

> To use the `-netlist` option with the `read` command, be sure your description is structural only. Do not use this option with any other type of description.
>
> Use the `-netlist` option only with the `read` command. It is not an option for any other command, such as `elaborate`.

Design Compiler Flags and dc_shell Variables

Several dc_shell variables affect how Verilog source files are read. Set these variables before you read in a Verilog file with the `read -format verilog` command or the File -> Read dialog box. You can set variables interactively or in your .synopsys_dc.setup file.

To list the `hdlin_` variables that affect reading in Verilog, enter

```
dc_shell> list -variables hdl
```

The following are explanations of the Verilog reading variables:

`hdlin_auto_save_templates`

If this variable is set to true, Design Compiler saves templates (designs that use generics) in memory. If this flag is false, it saves templates only as part of the calling (instantiating) design. For more information about templates, see “Using Templates—Naming” on page 3-21 and “template Directive” on page 9-21. Design Compiler automatically generates names for templates that are based on the values of the template naming variables (described later in this chapter).

The default is false.

`hdlin_hide_resource_line_numbers`

When HDL Compiler infers a synthetic library or a DesignWare part, the line number in the HDL source is not appended to the inferred cell’s name if this variable is set to true. (The default setting of `hdlin_hide_resource_line_numbers` is false.) This value makes the results of the Design Compiler `compile` command independent of the location of the inferred synthetic library or DesignWare parts in the HDL source.

To determine the current value of `hdlin_hide_resource_line_numbers`, type

```
dc_shell> list hdlin_hide_resource_line_numbers
```

`hdlin_report_inferred_modules`

If this variable is set to true, Design Compiler generates a report about inferred latches, flip-flops, and three-state and multiplexer devices. Redirect the report file by entering

```
dc_shell> read -f verilog my_file.v > my_file.report
```

`suppress_errors`

Indicates whether to suppress warning messages when reading Verilog source files. Warnings are nonfatal error messages. If this variable is set to true, warnings are not issued; if false, warnings are issued. This variable has no effect on fatal error messages, such as syntax errors, that stop the reading process.

The default is false.

You can also use this variable to disable specific warnings: set `suppress_errors` to a space-separated string of the error ID codes you want suppressed. Error ID codes are printed immediately after warning and error messages. For example, to suppress the following warning

```
Warning: Assertion statements are not supported. They are
ignored near symbol "assert" on line 24 (HDL-193).
```

set the variable to

```
suppress_errors = "HDL-193"
```

`hlo_resource_allocation`

When set to `constraint_driven`, this variable enables automatic resource sharing (see “Resource Sharing Methods” on page 7-11). When it is set to none, each operation in Verilog is implemented with separate circuitry.

Array Naming Variable

The `bus_naming_style` variable affects the way Design Compiler names elements of Verilog arrays.

This variable determines how to name the bits in port, cell, and net arrays. When a multiple-bit array is read in, Design Compiler converts the array to a set of individual single-bit names. The value is a string containing the characters `%s` and `%d`, which are replaced by the array name and bit (element) index, respectively. If the value is

```
bus_naming_style = "%s.%d"
```

the third element of an array called X_ARRAY, indexed from 0 to 7, is represented as `X_ARRAY.2`.

The default is `"%s[%d]"`.

To override the default value, set this variable before you issue the `read` command.

This variable is part of the io variable group; to list its current value, enter

```
dc_shell> list -variables io
```

Template Naming Variables

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

`template_naming_style = "%s_%p"`

> This is the master variable for naming a design built from 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.

`template_parameter_style = "%s%d"`

> This 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.

`template_separator_style = "_"`

> This variable contains a string that separates parameter names. This variable is used only for templates that have more than one parameter.

When a design is built from a template, only the parameters you indicate when you instantiate the parameterized design are used in the template name. For example, suppose the template ADD has parameters N, M, and Z. You can build a design where N = 8, M = 6, and Z is left at its default value. The name assigned to this design is `ADD_N8_M6`. If no parameters are listed, the template is built with default values and the name of the created design is the same as the name of the template.

Building Parameterized Designs

If your design has parameters, you can change the value of the parameters in a module each time that 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.

Parameterized designs are read into dc_shell as templates with the read command, just as other Verilog files are read. These designs are archived in a design library so they can be built with different (nondefault) values substituted for the parameters. You can also store a template in a design library with the `analyze` command.

If your design contains parameters, you can indicate that the design should be read in as a template in one of three ways:

- Add the pseudocomment `// synopsys template` to your code.
- Use the `analyze` command.
- Set the dc_shell variable `hdlin_auto_save_templates = true`.

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 10-1 shows how to do this.

Example 10-1 Instantiating a Parameterized Design in Verilog Code

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

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

foo #(4,5,4+6) U1(a,b,c); // instantiate foo

endmodule
```

In Example 10-1, the Verilog code instantiates the parameterized design foo, which has three parameters. The first parameter is assigned the value 4, the second parameter is assigned the value 5, and the third parameter takes the value 10.

Because module `foo` is defined outside the scope of module `param`, errors, such as port mismatches and invalid parameter assignments, are not detected until link time. When Design Compiler links module `param`, it searches for template `foo` in the design library `work`. If `foo` is found, it is automatically built with the specified parameters. Design Compiler checks that `foo` has at least three parameters 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.

Another way to instantiate a parameterized design is with the `elaborate` command in dc_shell. The syntax of the command is

```
elaborate template_name -parameters parameter_list
```

You can archive parameterized designs (templates) in design libraries. To verify that a template is stored in memory, use the `report_design_lib work` command. The `report_design_lib` command lists templates that reside in the indicated design library.

Synthetic Libraries

This section gives only basic information on synthetic libraries. For a complete explanation of how to use synthetic libraries, see the *DesignWare Components Databook*.

A synthetic library contains synthetic cells called operators. Operators resemble generic logic, as they have no netlist implementation and are not linked. Operators are visible from `report_synlib standard.sldb`. Table 10-1 shows all standard operators and a description of each.

Table 10-1 Synopsys Standard Operators

Operator	Description
ADD_TC_OP	Signed adder
ADD_UNS_OP	Unsigned adder
EQ_TC_OP	Signed equality
EQ_UNS_OP	Unsigned equality
GEQ_TC_OP	Signed greater than or equal to
GEQ_UNS_OP	Unsigned greater than or equal to
GT_TC_OP	Signed greater than
GT_UNS_OP	Unsigned greater than
LEQ_TC_OP	Signed less than or equal to
LEQ_UNS_OP	Unsigned less than or equal to
LT_UNS_OP	Unsigned less than
LT_TC_OP	Signed less than
MULT_TC_OP	Signed multiplier

Table 10-1 Synopsys Standard Operators (continued)

Operator	Description
NE_TC_OP	Signed inequality
NE_UNS_OP	Unsigned inequality
SELECT_OP	Selector
SUB_TC_OP	Signed subtracter
SUB_UNS_OP	Unsigned subtracter

The selector operator, `SELECT_OP`, functions as a multiplexer but has a control input for each data input. When the control input for a corresponding data input is high, that input is passed to the output.

When you issue the `compile` command, Design Compiler determines an appropriate implementation for the operators in your design. Design Compiler implements an operator in three steps:

1. It chooses a module, such as `add`, and its corresponding implementation, such as `rpl_add`. The function of an implementation is determined by the operator type, such as `ADD_UNS_OP`, and the width of the connections to it (the bit-width).
2. It creates a netlist for the implementation and inserts the netlist in the design.
3. It optimizes the netlist.

For example, HDL Compiler generates an operator called `ADD_UNS_OP_3_4_5` when you read in the following code

```
z[4:0] = a[2:0] + b[3:0];
```

One way to implement the `ADD_UNS_OP` operator is with a 5-bit ripple carry adder. This implementation is called `rpl_add_n5`.

To see a list of modules and their implementations, enter

```
dc_shell> report_synlib standard.sldb
```

Optimizing With Design Compiler

After HDL Compiler translates a Verilog description, it passes the description to Design Compiler for optimization and synthesis. When you read a Verilog design into Design Compiler, the design is converted to the Design Compiler internal database format. When Design Compiler performs logic optimization on a design, it can restructure all or part of the design. You have control over the degree of restructuring. You can keep your design’s hierarchy intact, move modules up or down the design hierarchy, combine modules, or compress the entire design into one module.

After you read your design into Design Compiler, you can write it out in a variety of formats, including Verilog. You can convert existing gate-level netlists, sets of logic equations, or technology-specific circuits to a Verilog description. You can use the new Verilog description as documentation for the original design and as a starting point for reimplementing the design in a new technology. In addition, you can give the Verilog description to a Verilog simulator to extract circuit timing information.

This section describes some uses of the `compile` command in Design Compiler. For a complete description, refer to the `compile` man page.

Flattening and Structuring

Design Compiler uses two optimization strategies: flattening and structuring. Flattening tries to reduce a design's logical structure to a set of two-level logic equations. Structuring tries to find the common factors in the translated design's set of logic equations.

When a design is flattened, the original structure of its Verilog description is lost. Flattening is useful when a description is written at a high level without regard to the use of constructs or resource allocation. Random control logic often falls into this category. In general, flattening consolidates logic; it also often speeds up the final implementation. Not all logic can be flattened: For example, large adders, XOR networks, and comparators of two variables cannot be flattened. If you use these elements in a design, place them in separate modules that will not be flattened.

If you build structure into the Verilog description through user-defined operators (such as carry-lookahead adders) or resource sharing, do not flatten the design. You can still use structuring, which attempts to improve the design's logical structure without destroying the existing structure. The Design Compiler defaults of `-no_flatten` and `-structure` are appropriate for almost all Verilog descriptions. For more information about flattening and structuring a design, see the *Design Compiler User Guide*.

Grouping Logic

Design Compiler performs optimization on designs. All constraints and compile directives are applied at the design level. If you intend to optimize two pieces of logic differently, they must be in separate designs.

Designs in Design Compiler have a one-to-one correspondence with modules in the input Verilog description. Functions and operators in a Verilog module are grouped with that module for optimization. At times, you might regroup logic in a Verilog description to achieve the optimization you want. For example, you might want to optimize part of your design for speed and part for area. You can group the speed-critical logic and optimize it independently. You can regroup logic with the `group` command. For more information on the `group` command, see the Design Compiler documentation or the `group` man page.

Busing

Design Compiler maintains types throughout a design, including types for buses (vectors). Example 10-2 shows a Verilog design read into HDL Compiler containing a bit vector that is NOTed into another bit vector.

Example 10-2 Bit Vector in Verilog

```
module test_busing_1 ( a, b );
  input  [3:0] a;
  output [3:0] b;

  assign b = ~a;

endmodule
```

Example 10-3 shows the same description written out by HDL Compiler. The description contains the original Verilog types of ports. Internal nets do not maintain their original bus types. Also, the NOT operation is instantiated as single bits.

Example 10-3   Bit Blasting

```
module test_busing_2 ( a, b );
input  [3:0] a;
output [3:0] b;
    assign b[0] = ~a[0];
    assign b[1] = ~a[1];
    assign b[2] = ~a[2];
    assign b[3] = ~a[3];
endmodule
```

Correlating HDL Source Code to Synthesized Logic

By using RTL Analyzer, you can display the text in your source HDL code that corresponds to gates in the synthesized design. For more information, see the *RTL Analyzer User Guide*.

Writing Out Verilog Files

To write out Verilog design files, use the File/Write dialog box or the `write` command.

```
dc_shell> write -format verilog -output my_file.verilog
```

The `write -format verilog` command is valid whether or not the current design originated as a Verilog source file. Any design, regardless of initial format (equation, netlist, and so on), can be written out as a Verilog design.

For more information about the `write` command, see the Design Compiler documentation.

Setting Verilog Write Variables

Several dc_shell variables affect how designs are written out as Verilog files. To override the default settings, set these variables before you write out the design with the `write -format verilog` command or the File/Write dialog box. You can set the variables interactively or set them in your .synopsys_dc.setup file.

To list the current values of the variables that affect writing out Verilog (`verilogout_` variables), enter

```
dc_shell> list -variables hdl
```

The `verilogout_` variables are

`verilogout_equation`

When this is set to true, Verilog assign statements (Boolean equations) are written out for combinational gates, instead of for gate instantiations. Flip-flops and three-state cells are left instantiated. The default is false.

`verilogout_higher_designs_first`

When this is set to true, Verilog modules are ordered so that higher-level designs come before lower-level designs, as defined by the design hierarchy. The default is false.

`verilogout_no_tri`

When this is set to true, three-state nets are declared as Verilog `wire` instead of `tri`. This variable eliminates `assign` primitives and `tran` gates in your Verilog output, by connecting an output port directly to a component instantiation. The default is false.

`verilogout_single_bit`

When this variable is set to true, vectored ports (or ports that use record types) are bit-blasted; if a port's bit vector is `N` bits wide, it is written out to the Verilog file as `N` separate single-bit ports. When it is set to false, all ports are written out with their original data types. The default is true.

`verilogout_time_scale`

This variable determines the ratio of library time to simulator time and is used only by the `write_timing` command. The default is 1.0.

A

Examples

This appendix presents five examples that demonstrate basic concepts of Synopsys HDL Compiler:

Count Zeros—Combinational Version

Using this circuit is one possible solution to a design problem. Given an 8-bit value, the circuit must determine two things:

- The presence of a value containing exactly one sequence of zeros
- The number of zeros in the sequence (if any)

The circuit must complete this computation in a single clock cycle. The input to the circuit is an 8-bit value, and the two outputs the circuit produces are the number of zeros found and an error indication.

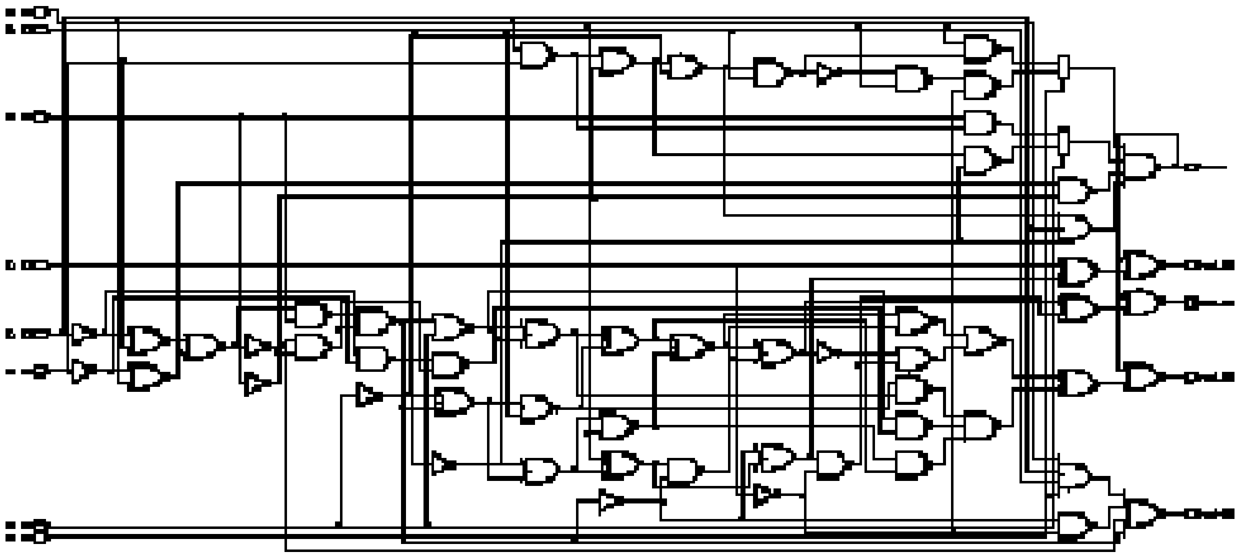
A valid value contains only one series of zeros. If more than one series of zeros appears, the value is invalid. A value consisting of all ones is a valid value. If a value is invalid, the count of zeros is set to zero. For example,

- The value 00000000 is valid, and the count is eight zeros.
- The value 11000111 is valid, and the count is three zeros.
- The value 00111110 is invalid.

A Verilog description and a schematic of the circuit are shown in Example A-1.

Example A-1   Count Zeros—Combinational

```
module count_zeros(in, out, error);
   input  [7:0] in;
   output [3:0] out;
   output error;
   function legal;
   input [7:0] x;
   reg seenZero, seenTrailing;
   integer i;
   begin : _legal_block
      legal = 1; seenZero = 0; seenTrailing = 0;
      for ( i=0; i <= 7; i=i+1 )
         if ( seenTrailing && (x[i] == 1'b0) ) begin
            legal = 0;
            disable _legal_block;
            end
         else if ( seenZero && (x[i] == 1'b1) )
            seenTrailing = 1;
         else if ( x[i] == 1'b0 )
            seenZero = 1;
      end
   endfunction

   function [3:0] zeros;
   input [7:0] x;
   reg   [3:0] count;
   integer i;

   begin
      count = 0;
      for ( i=0; i <= 7; i=i+1 )
         if ( x[i] == 1'b0 ) count = count + 1;
         zeros = count;
      end
   endfunction
   wire is_legal = legal(in);
   assign error = ! is_legal;
   assign out   = is_legal ? zeros(in) : 1'b0;
endmodule
```



This example shows two Verilog functions: legal and zeros. The function legal determines if the value is valid. It returns a 1-bit value: either 1 for a valid value or 0 for an invalid value. The function zeros cycles through all bits of the value, counts the number of zeros, and returns the appropriate value. The two functions are controlled by continuous assignment statements at the bottom of the module definition. This example shows a combinational (parallel) approach to counting zeros; the next example shows a sequential (serial) approach.

Count Zeros—Sequential Version

Example A-2 shows a sequential (clocked) solution to the “count zeros” design problem. The circuit specification is slightly different from the specification in the combinational solution. The circuit now accepts the 8-bit string serially, 1 bit per clock cycle, using the data and clk inputs. The other two inputs are

- `reset`, which resets the circuit
- `read`, which causes the circuit to begin accepting data

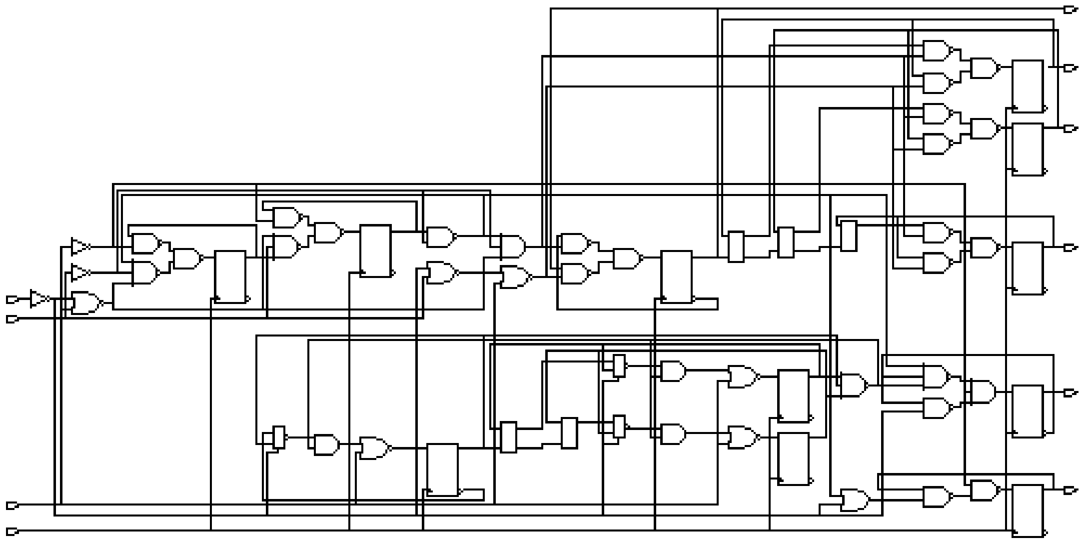
The circuit’s three outputs are

- `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 8 bits have been processed
- `zeros`, which is the number of zeros if `is_legal` is true

Example A-2 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 or zeros or bits_seen ) 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
```



Drink Machine—State Machine Version

The next design is a vending control unit for a soft drink vending machine. The circuit reads signals from a coin-input unit and sends outputs to a change-dispensing unit and a drink-dispensing unit.

Input signals from the coin-input unit are `nickel_in` (nickel deposited), `dime_in` (dime deposited), and `quarter_in` (quarter deposited).

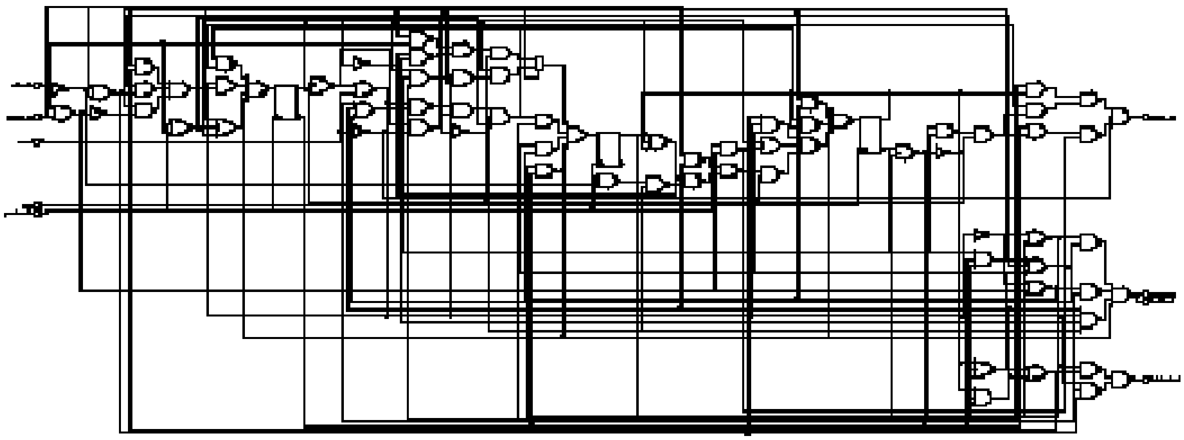
Outputs to the vending control unit are `collect` (collect coins), to the coin-input unit; `nickel_out` (nickel change) and `dime_out` (dime change), to the change-dispensing unit; and `dispense` (dispense drink), to the drink-dispensing unit.

The price of a drink is 35 cents. The Verilog description for this design, shown in Example A-3, uses a state machine description style. The description includes the `state_vector` directive, which enables Design Compiler to extract an equivalent state machine.

Example A-3 Drink Machine—State Machine Version

```
‘define vend_a_drink {D,dispense,collect} = {IDLE,2’b11}

module drink_machine(nickel_in, dime_in, quarter_in,
                     collect, nickel_out, dime_out,
                     dispense, reset, clk) ;
   parameter IDLE=0,FIVE=1,TEN=2,TWENTY_FIVE=3,
             FIFTEEN=4,THIRTY=5,TWENTY=6,OWE_DIME=7;

   input  nickel_in, dime_in, quarter_in, reset, clk;
   output collect, nickel_out, dime_out, dispense;

   reg collect, nickel_out, dime_out, dispense;
   reg [2:0] D, Q; /* state */
// synopsys state_vector Q

always @ ( nickel_in or dime_in or quarter_in or reset )
      begin
         nickel_out = 0;
         dime_out   = 0;
         dispense   = 0;
         collect    = 0;

         if ( reset ) D = IDLE;
         else begin
            D = Q;

            case ( Q )
            IDLE:
               if (nickel_in)       D = FIVE;
               else if (dime_in)    D = TEN;
               else if (quarter_in) D = TWENTY_FIVE;
            FIVE:
               if(nickel_in)        D = TEN;
               else if (dime_in)    D = FIFTEEN;
               else if (quarter_in) D = THIRTY;
            TEN:
               if (nickel_in)       D = FIFTEEN;
               else if (dime_in)    D = TWENTY;
               else if (quarter_in) ‘vend_a_drink;
            TWENTY_FIVE:
               if( nickel_in)       D = THIRTY;
               else if (dime_in)    ‘vend_a_drink;
               else if (quarter_in) begin

                  ‘vend_a_drink;
                   nickel_out = 1;
                   dime_out = 1;
```

```
            end

        FIFTEEN:
            if (nickel_in)       D = TWENTY;
            else if (dime_in)    D = TWENTY_FIVE;
            else if (quarter_in) begin
                ‘vend_a_drink;
                nickel_out = 1;
            end

        THIRTY:
            if (nickel_in)       ‘vend_a_drink;
            else if (dime_in)    begin
                ‘vend_a_drink;
                nickel_out = 1;
            end
            else if (quarter_in) begin
                ‘vend_a_drink;
                dime_out = 1;
                D = OWE_DIME;
            end

        TWENTY:
            if (nickel_in)       D = TWENTY_FIVE;
            else if (dime_in)    D = THIRTY;
            else if (quarter_in) begin
                ‘vend_a_drink;
                dime_out = 1;
            end

        OWE_DIME:
            begin
                dime_out = 1;
                D = IDLE;
            end
        endcase
    end
end

always @ (posedge clk ) begin
     Q = D;
end
endmodule
```



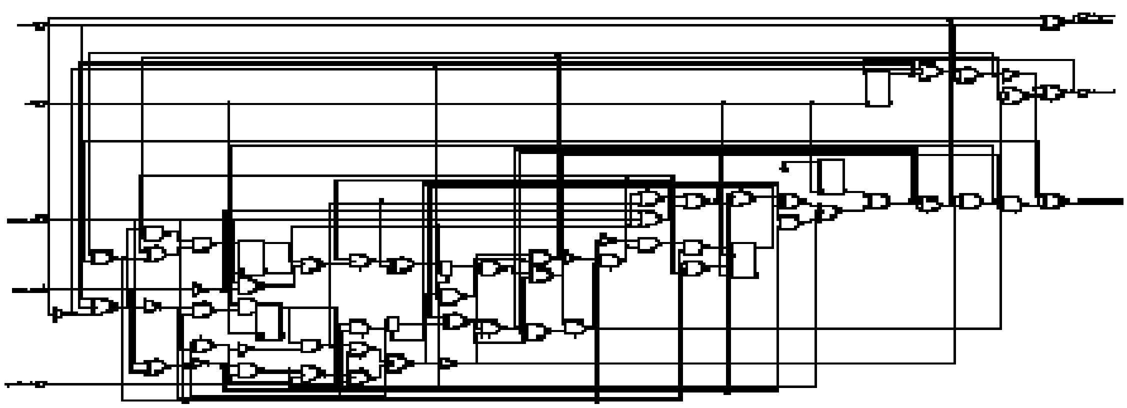
Drink Machine—Count Nickels Version

Example A-4 uses the same design parameters as Example A-3 with the same input and output signals. In this version, a counter counts the number of nickels deposited. This counter is incremented by one if the deposit is a nickel, by two if it's a dime, and by five if it's a quarter.

Example A-4 Drink Machine—Count Nickels Version

```
module drink_machine(nickel_in,dime_in,quarter_in,collect,
       nickel_out,dime_out,dispense,reset,clk);

        input nickel_in, dime_in, quarter_in, reset, clk;
        output nickel_out, dime_out, collect, dispense;

        reg nickel_out, dime_out, dispense, collect;
        reg [3:0] nickel_count, temp_nickel_count;
        reg temp_return_change, return_change;

             always @ ( nickel_in or dime_in or quarter_in or
             collect or temp_nickel_count or
             reset or nickel_count or return_change) begin
                   nickel_out = 0;
                   dime_out   = 0;
                   dispense   = 0;
                   collect    = 0;
                   temp_nickel_count = 0;
                   temp_return_change = 0;

                   // Check whether money has come in
                   if (! reset) begin
                        temp_nickel_count = nickel_count;
                        if (nickel_in)
                          temp_nickel_count = temp_nickel_count + 1;
                        else if (dime_in)
                          temp_nickel_count = temp_nickel_count + 2;
                        else if (quarter_in)
                          temp_nickel_count = temp_nickel_count + 5;

                   // correct amount deposited?
                   if (temp_nickel_count >= 7) begin
                        temp_nickel_count = temp_nickel_count - 7;
                        dispense = 1;
                        collect = 1;
                   end
```

```
                    // return change
                    if (return_change || collect) begin
                         if (temp_nickel_count >= 2) begin
                           dime_out = 1;
                           temp_nickel_count = temp_nickel_count - 2;
                           temp_return_change = 1;
                         end

                         if (temp_nickel_count == 1) begin
                           nickel_out = 1;
                           temp_nickel_count = temp_nickel_count - 1;
                         end
                    end
               end
          end
          always @ (posedge clk ) begin
               nickel_count = temp_nickel_count;
               return_change = temp_return_change;
          end
endmodule
```



Carry-Lookahead Adder

Figure A-1 on page A-17 and Example A-5 on page A-18 show how to build a 32-bit carry-lookahead adder. The adder is built by partitioning of the 32-bit input into eight slices of 4 bits each. The PG module computes propagate and generate values for each of the eight slices.

Propagate (output P from PG) is 1 for a bit position if that position propagates a carry from the next-lower position to the next-higher position. Generate (output G) is 1 for a bit position if that position generates a carry to the next-higher position, regardless of the carry-in from the next-lower position.

The carry-lookahead logic reads the carry-in, propagate, and generate information computed from the inputs. It computes the carry value for each bit position. This logic makes the addition operation an XOR of the inputs and the carry values.

The following list shows the order in which the carry values are computed by a three-level tree of 4-bit carry-lookahead blocks (illustrated in Figure A-1):

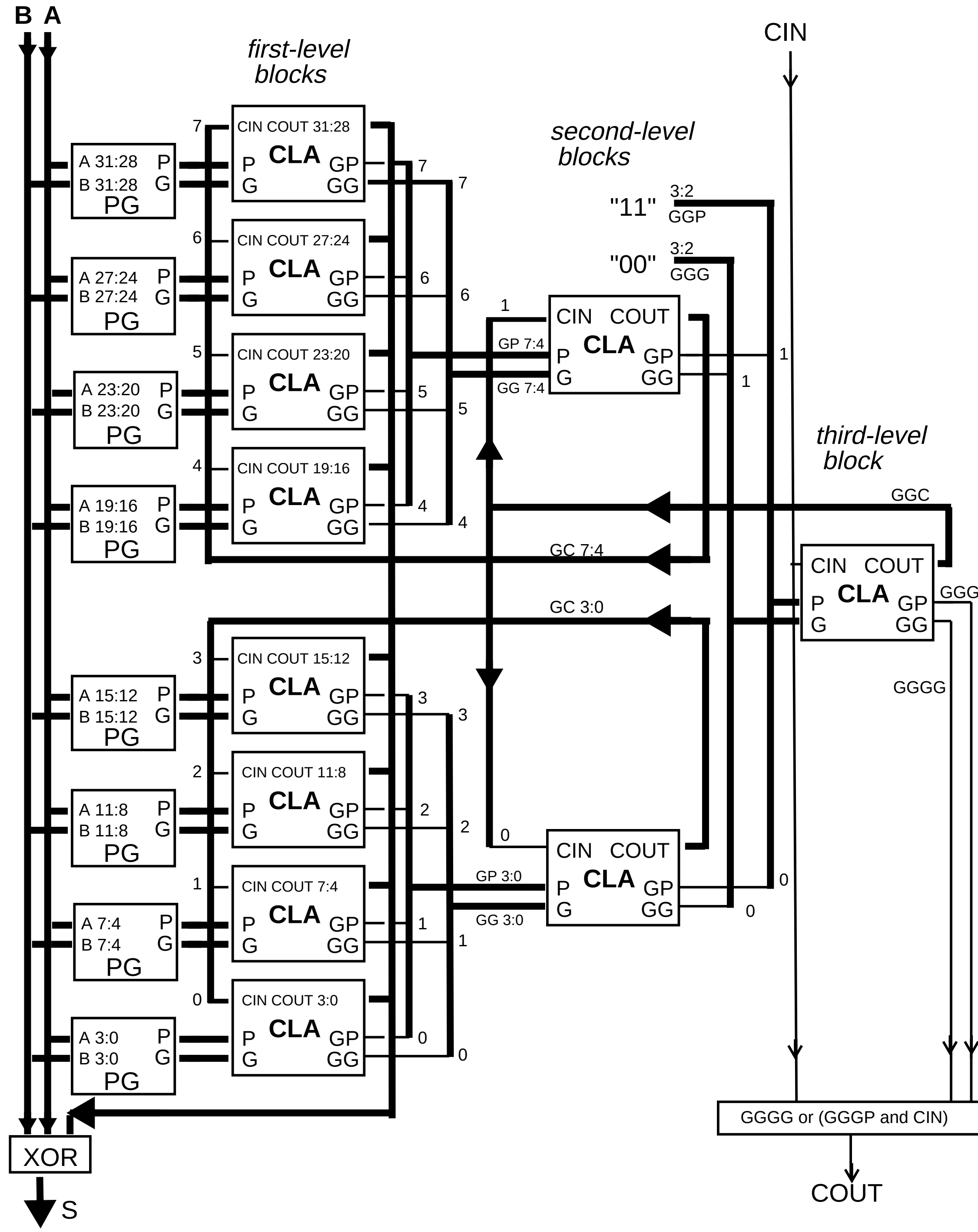
1. The first level of the tree computes the 32 carry values and the 8 group propagate and generate values. Each of the first-level group propagate and generate values tells if that 4-bit slice propagates and generates carry values from the next-lower group to the next-higher. The first-level lookahead blocks read the group carry computed at the second level.

2. At the second level of the tree, the lookahead blocks read the group propagate and generate information from the four first-level blocks and then compute their own group propagate and generate information. They also read group carry information computed at the third level to compute the carries for each of the third-level blocks.
3. At the third level of the tree, the third-level block reads the propagate and generate information of the second level to compute a propagate and generate value for the entire adder. It also reads the external carry to compute each second-level carry. The carry-out for the adder is 1 if the third-level generate is 1 or if the third-level propagate is 1 and the external carry is 1.

   The third-level carry-lookahead block can process four second-level blocks. Because there are only two second-level blocks in Figure A-1, the high-order 2 bits of the computed carry are ignored, the high-order 2 bits of the generate input to the third-level are set to 00 (zero), and the propagate high-order bits are set to 11. This causes the unused portion to propagate carries but not to generate them.

Figure A-1 shows the three levels of a block diagram of the 32-bit carry-lookahead adder. Example A-5 shows the code for the adder.

Figure A-1 Carry-Lookahead Adder Block Diagram



Example A-5 Carry-Lookahead Adder

```
‘define word_size 32
‘define word [‘word_size-1:0]

‘define n 4
‘define slice [‘n-1:0]

‘define s0 (1*‘n)-1:0*‘n
‘define s1 (2*‘n)-1:1*‘n
‘define s2 (3*‘n)-1:2*‘n
‘define s3 (4*‘n)-1:3*‘n
‘define s4 (5*‘n)-1:4*‘n
‘define s5 (6*‘n)-1:5*‘n
‘define s6 (7*‘n)-1:6*‘n
‘define s7 (8*‘n)-1:7*‘n

module cla32_4(a, b, cin, s, cout);
input ‘word a, b;
input cin;
output ‘word s;
output cout;

 wire [7:0] gg, gp, gc; // Group generate, propagate,
                                   // carry
 wire [3:0] ggg, ggp, ggc;// Second-level gen., prop.
 wire gggg, gggp; // Third-level gen., prop.

 bitslice i0(a[‘s0], b[‘s0], gc[0], s[‘s0], gp[0], gg[0]);
 bitslice i1(a[‘s1], b[‘s1], gc[1], s[‘s1], gp[1], gg[1]);
 bitslice i2(a[‘s2], b[‘s2], gc[2], s[‘s2], gp[2], gg[2]);
 bitslice i3(a[‘s3], b[‘s3], gc[3], s[‘s3], gp[3], gg[3]);

 bitslice i4(a[‘s4], b[‘s4], gc[4], s[‘s4], gp[4], gg[4]);
 bitslice i5(a[‘s5], b[‘s5], gc[5], s[‘s5], gp[5], gg[5]);
 bitslice i6(a[‘s6], b[‘s6], gc[6], s[‘s6], gp[6], gg[6]);
 bitslice i7(a[‘s7], b[‘s7], gc[7], s[‘s7], gp[7], gg[7]);

 cla c0(gp[3:0], gg[3:0], ggc[0], gc[3:0], ggp[0], ggg[0]);
 cla c1(gp[7:4], gg[7:4], ggc[1], gc[7:4], ggp[1], ggg[1]);

 assign ggp[3:2] = 2’b11;
 assign ggg[3:2] = 2’b00;
 cla c2(ggp, ggg, cin, ggc, gggp, gggg);
 assign cout = gggg | (gggp & cin);
endmodule

// Compute sum and group outputs from a, b, cin
```

```
module bitslice(a, b, cin, s, gp, gg);
input ‘slice a, b;
input cin;
output ‘slice s;
output gp, gg;

 wire ‘slice p, g, c;
 pg i1(a, b, p, g);
 cla i2(p, g, cin, c, gp, gg);
 sum i3(a, b, c, s);
endmodule

// compute propagate and generate from input bits

module pg(a, b, p, g);
input ‘slice a, b;
output ‘slice p, g;

 assign p = a | b;
 assign g = a & b;
endmodule

// compute sum from the input bits and the carries

module sum(a, b, c, s);
input ‘slice a, b, c;
output ‘slice s;

 wire ‘slice t = a ^ b;
 assign s = t ^ c;
endmodule

// n-bit carry-lookahead block

module cla(p, g, cin, c, gp, gg);
input ‘slice p, g;// propagate and generate bits
input cin;    // carry in
output ‘slice c; // carry produced for each bit
output gp, gg; // group generate and group propagate

 function [99:0] do_cla;
 input ‘slice p, g;
 input cin;

 begin : label
 integer i;
 reg gp, gg;
 reg ‘slice c;
```

```
 gp = p[0];
 gg = g[0];
 c[0] = cin;
 for(i = 1; i < ‘n; i = i+1) begin
        gp = gp & p[i];
        gg = (gg & p[i]) | g[i];
        c[i] = (c[i-1] & p[i-1]) | g[i-1];
 end
 do_cla = {c, gp, gg};
 end
 endfunction

 assign {c, gp, gg} = do_cla(p, g, cin);
endmodule
```

B

Verilog Syntax

This appendix contains a syntax description of the Verilog language as supported by Synopsys HDL Compiler. It 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 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 @ ( <edge> 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> )
   ||= ( <STRENGTH1> , <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 these characters:
0123456789abcdefxzABCDEFXZ.

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

<SIZE>
```

A size can have any number of these 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 an 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 HDL Compiler uses are nearly identical to those of the Verilog language. The types of lexical tokens HDL Compiler uses are described in the following subsections:

- White Space
- Comments
- Numbers
- Identifiers
- Operators
- Macro Substitution
- include Construct
- 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. HDL Compiler ignores white space.

Comments

You can enter comments anywhere in a Verilog description, in two forms:

- Beginning with two slashes //

  HDL Compiler ignores all text between these characters and the end of the current line.

- Beginning with the two characters /* and ending with */

  HDL Compiler 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:

- A simple decimal number that is a sequence of digits in the range of 0 to 9. All constants declared this way are assumed to be 32-bit numbers.
- A number that specifies the bit-width as well as the radix. These numbers are the same as those in the previous format, except that they are preceded by a decimal number that specifies the bit-width.

- 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 (_), which improve readability and do not affect the value of the number. Table B-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 B-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.

Identifiers that contain 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 B-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. These are partially supported by HDL Compiler. (For more information, see “Unsupported Verilog Language Constructs” on page B-21.)

Operators

Operators are one- or two-character sequences that perform operations on variables. Some examples of operators are +, ~^, <=, and >>. Operators are described in detail in “Operators” on page 4-3.

Macro Substitution

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 quotation mark (‘), 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 Design Compiler 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 B-3.

Example B-3   Macro Variable Declarations

```
‘define highbits      31:29
‘define bitlist       {first, second, third}
wire [31:0] bus;
‘bitlist = bus[‘highbits];
```

Text macros are not supported when used with sized constants, as shown in Example B-4.

Example B-4   Macro With Sized Constants

```
‘define SIZE 4

module test (in,out);
output [3:0] out;
input [3:0] in;

assign out = ‘SIZE’b0101; //text macro from ‘define statement
                          //cannot be used with a sized constant

endmodule
```

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 in another module. Example B-5 shows an application of the `include` construct.

Example B-5 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, with up to 24 levels of nesting. You cannot use the `include` construct recursively. Set the include directory with the `search_path` variable in dc_shell.

Simulation Directives

Simulation directives refer to special commands that affect the operation of the Verilog HDL Simulator. You can include these directives in your design description, because HDL Compiler 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 special functions Verilog HDL Simulators implement to generate input or output during simulation. Their names start with a dollar sign ($). These functions are parsed and ignored by HDL Compiler.

Verilog Keywords

Verilog uses keywords, shown in Table B-2, to interpret an input file. You cannot use these words as user variable names unless you use an escaped identifier. For more information, see “Identifiers” on page B-16.

Table B-2 Verilog Keywords

always	force	or	trireg
and	forever	output	table
assign	fork	parameter	task
begin	function	pmos	time
buf	highz0	posedge	tran
bufif0	highz1	primitive	tranif0
bufif1	if	pull0	tranif1
case	initial	pull1	tri
casex	inout	rcmos	triand
casez	input	reg	tri0
cmos	integer	release	tri1
deassign	join	repeat	vectored
default	large	rnmos	wait
defparam	medium	rpmos	wand
disable	module	rtran	weak0
end	nand	rtranif0	weak1
endcase	negedge	rtranif1	while

Table B-2 Verilog Keywords (continued)

endfunction	nmos	scalared	wire
endmodule	nor	small	wor
endprimitive	not	strong0	xnor
endtable	notif0	strong1	xor
endtask	notif1	supply0	
event	pulldown	supply1	
for	pullup	trior	

Unsupported Verilog Language Constructs

HDL Compiler does not support the following Verilog constructs:

- Unsupported definitions and declarations
  - primitive definition
  - `time` declaration
  - `event` declaration
  - `triand`, `trior`, `tri1`, `tri0`, and `trireg` net types
  - Ranges and arrays for integers
- Unsupported statements
  - `defparam` statement
  - `initial` statement
  - `repeat` statement

- delay control
- event control
- `wait` statement
- `fork` statement
- `deassign` statement
- `force` statement
- `release` statement

- Unsupported operators
  - Case equality and inequality operators (=== and !==)
  - Division and modulus operators for variables
- Unsupported gate-level constructs
  - nmos, pmos, cmos, rnmos, rpmos, rcmos
  - pullup, pulldown, tranif0, tranif1, rtran, rtrainf0, and rtrainf1 gate types
- Unsupported miscellaneous constructs, such as hierarchical names within a module

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, HDL Compiler issues a syntax error such as

```
event is not supported
```

Glossary

anonymous type
A predefined or underlying type with no name, such as universal integers.

ASIC
Application-specific integrated circuit.

behavioral view
The set of Verilog statements that describe the behavior of a design by using sequential statements. These statements are similar in expressive capability to those found in many other programming languages. See also the *data flow view*, *sequential statement*, and *structural view* definitions.

bit-width
The width of a variable, signal, or expression in bits. For example, the bit-width of the constant 5 is 3 bits.

character literal
Any value of type CHARACTER, in single quotation marks.

computable
Any expression whose (constant) value HDL Compiler can determine during translation.

constraints
The designer’s specification of design performance goals. Design Compiler uses constraints to direct the optimization of a design to meet area and timing goals.

convert
To change one type to another. Only integer types and subtypes are convertible, along with same-size arrays of convertible element types.

data flow view
The set of Verilog statements that describe the behavior of a design by using concurrent statements. These descriptions are usually at the level of Boolean equations combined with other operators and function calls. See also the *behavioral view* and *structural view* definitions.

Design Compiler
The Synopsys tool that synthesizes and optimizes ASIC designs from multiple input sources and formats.

design constraints
See *constraints*.

flip-flop
An edge-sensitive memory device.

HDL
Hardware Description Language.

HDL Compiler
The Synopsys Verilog synthesis product.

identifier
A sequence of letters, underscores, and numbers. An identifier cannot be a Verilog reserved word, such as type or loop. An identifier must begin with a letter or an underscore.

latch
A level-sensitive memory device.

netlist
A network of connected components that together define a design.

optimization
The modification of a design in an attempt to improve some performance aspect. Design Compiler optimizes designs and tries to meet specified design constraints for area and speed.

port
A signal declared in the interface list of an entity.

reduction operator
An operator that takes an array of bits and produces a single-bit result, namely the result of the operator applied to each successive pair of array elements.

register
A memory device containing one or more flip-flops or latches used to hold a value.

resource sharing
The assignment of a similar Verilog operation (for example, +) to a common netlist cell. Netlist cells are the resources—they are equivalent to built hardware.

RTL
Register transfer level, a set of structural and data flow statements.

sequential statement
A set of Verilog statements that execute in sequence.

signal
An electrical quantity that can be used to transmit information. A signal is declared with a type and receives its value from one or more drivers. Signals are created in Verilog through either wire or reg declarations.

signed value
A value that can be positive, zero, or negative.

structural view
The set of Verilog statements used to instantiate primitive and hierarchical components in a design. A Verilog design at the structural level is also called a netlist. See also *behavioral view* and *data flow view*.

subtype
A type declared as a constrained version of another type.

synthesis
The creation of optimized circuits from a high-level description. When Verilog is used, synthesis is a two-step process: translation from Verilog to gates by HDL Compiler and optimization of those gates for a specific ASIC library with Design Compiler.

technology library
A library of ASIC cells available to Design Compiler during the synthesis process. A technology library can contain area, timing, and functional information on each ASIC cell.

translation
The mapping of high-level language constructs onto a lower-level form. HDL Compiler translates RTL Verilog descriptions to gates.

type
In Verilog, the mechanism by which objects are restricted in the values they are assigned and the operations that can be applied to them.

unsigned
A value that can be only positive or zero.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z