

3

Working With Design Files

Designs are stored in design files, which are ASCII files containing a description of one or more designs. Design files must have unique names. Each design in a design file must also have a unique name, but different files might contain designs with identical names.

This chapter contains the following sections:

- [Managing the Design Data](#)
- [Partitioning for Synthesis](#)
- [HDL Coding for Synthesis](#)

Managing the Design Data

Use systematic organizational methods to manage the design data. Two basic elements of managing design data are design data control and data organization.

Controlling the Design Data

As new versions of your design are created, you must maintain some data to provide a history of the design evolution and to use as a restart point if data is lost. Establishing controls for data creation, maintenance, overwriting, and deletion is a fundamental design management issue. Establish rules for each of these data operations before starting design development.

Establishing file naming conventions is one of the most important rules for data creation. Table 3-1 lists the recommended file name extensions for each design data type.

Table 3-1 File Name Extensions

Design Data Type	Extension	Description
Design source code	.v	Verilog
	.vhd	VHDL
	.edif	EDIF
Synthesis scripts	.con	Constraints
	.scr	Script
Reports and logs	.rpt	Report
	.log	Log
Design database	.db	Synopsys database format

Organizing the Design Data

Establishing and adhering to a method of organizing data is more important than which method you choose. After you place the essential design data under a consistent set of controls, you can create a meaningful data organization. To simplify data exchanges and data searches, designers must adhere to the adopted data organization system.

You can use a hierarchical directory structure to address the data organization issues. Your compile strategy influences your directory structure. The following figures show directory structures based on the top-down compile strategy (Figure 3-1) and the bottom-up compile strategy (Figure 3-2). See “Selecting a Compile Strategy” in Chapter 7 for details about these compile strategies.

Figure 3-1 Top-Down Compile Directory Structure

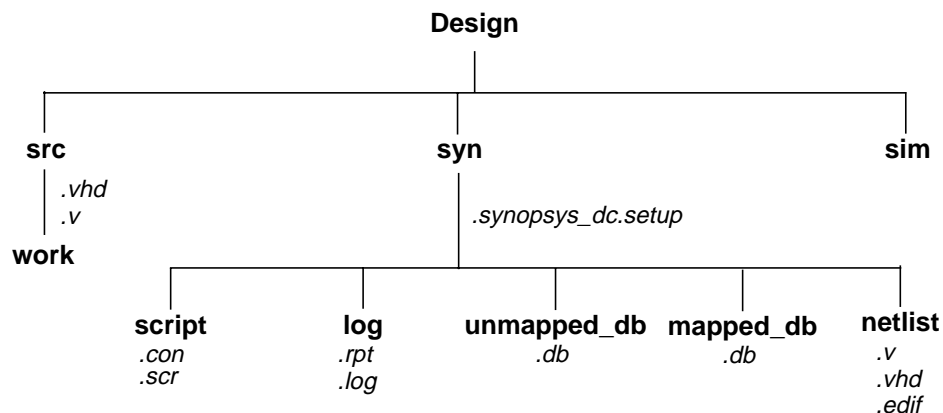
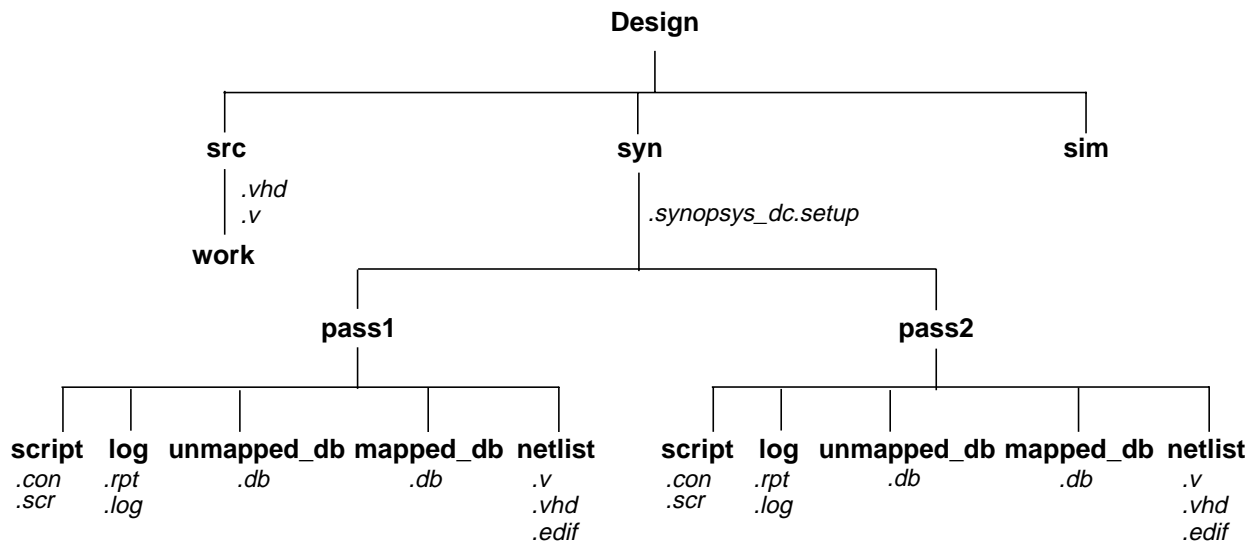


Figure 3-2 Bottom-Up Compile Directory Structure



Partitioning for Synthesis

Partitioning a design effectively can enhance the synthesis results, reduce compile time, and simplify the constraint and script files.

Use the following strategies to partition your design:

- Partition for design reuse.
- Keep related combinational logic together.
- Register the block outputs.
- Partition by design goal.
- Partition by compile technique.
- Keep sharable resources together.
- Keep user-defined resources with the logic they drive.

- Isolate special functions, such as pads, clocks, boundary scans, and asynchronous logic.
- Size blocks based on available resources.

The following sections describe each of these strategies.

Partitioning for Design Reuse

Design reuse decreases time to market by reducing the design, integration, and testing effort.

When reusing existing designs, partition the design to enable instantiation of the designs.

To enable future design reuse, follow these guidelines during partitioning and block design:

- Thoroughly define and document the design interface.
- Standardize interfaces whenever possible.
- Parameterize the HDL code.

Keeping Related Combinational Logic Together

By default, Design Compiler cannot move logic across hierarchical boundaries. Dividing related combinational logic into separate blocks introduces artificial barriers that restrict logic optimization.

For best results,

- Group related combinational logic and its destination register together.

When working with the complete combinational path, Design Compiler has the flexibility to merge logic, resulting in a smaller, faster design.

Grouping combinational logic with its destination register simplifies the timing constraints and enables sequential optimization.

- Eliminate glue logic.

Glue logic is the combinational logic that connects blocks. Moving this logic into one of the blocks improves synthesis results by providing Design Compiler with additional flexibility. Eliminating glue logic also reduces compile time, because Design Compiler has fewer logic levels to optimize.

For example, assume that you have a design containing three combinational clouds on or near the critical path. Figure 3-3 shows poor partitioning of this design. Each of the combinational clouds occurs in a separate block, so Design Compiler cannot fully exploit its combinational optimization techniques.

Figure 3-3 Poor Partitioning of Related Logic

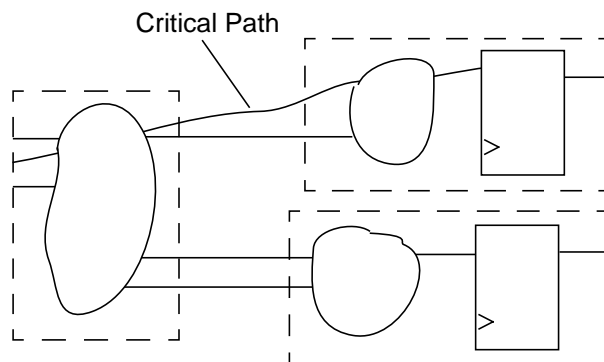
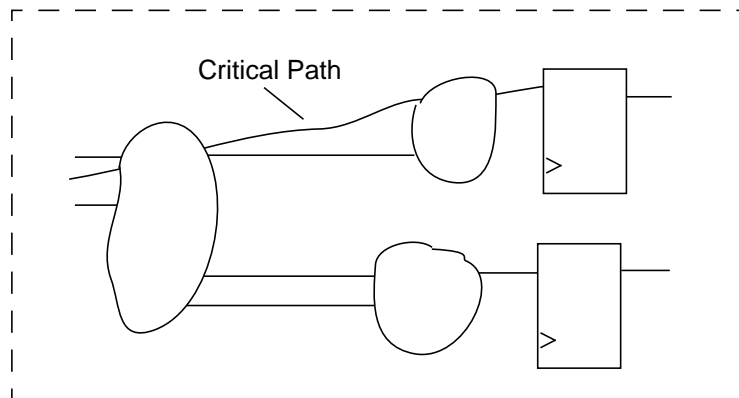


Figure 3-4 shows the same design with no artificial boundaries. In this design, Design Compiler has the flexibility to combine related functions in the combinational clouds.

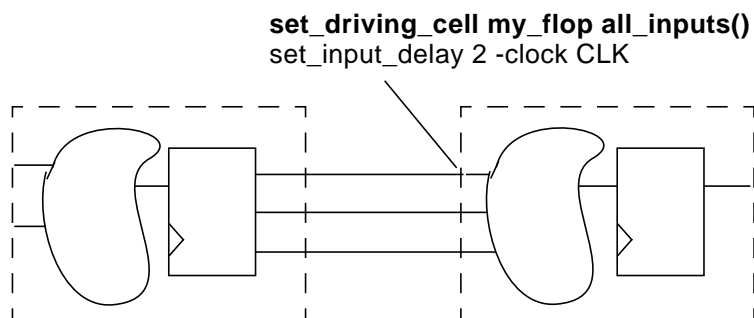
Figure 3-4 Keep Related Logic in the Same Block



Registering Block Outputs

To simplify the constraint definitions, make sure that registers drive the block outputs, as shown in Figure 3-5.

Figure 3-5 Register All Outputs



This method allows you to constrain each block easily because

- The drive strength on the inputs to an individual block always equals the drive strength of the average flip-flop
- The input delays from the previous block always equal the path delay through the flip-flop

Because no combinational-only paths exist when all outputs are registered, time-budgeting the design and using the `set_output_delay` command are easier. Given that one clock cycle occurs within each module, the constraints are simple and identical for each module.

This partitioning method of partitioning can increase simulation speed. With all outputs registered, a module can be described with only edge-triggered processes. The sensitivity list contains only the clock and, perhaps, a reset pin. A limited sensitivity list speeds simulation by having the process triggered only once in each clock cycle.

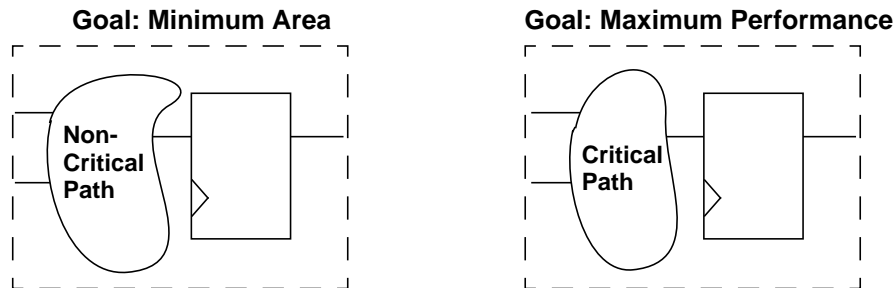
Partitioning by Design Goal

Partition logic with different design goals into separate blocks. Use this method when the design has both a critical speed constraint and a critical area constraint.

To achieve the best synthesis results, isolate the noncritical speed constraint logic from the critical speed constraint logic. By isolating the noncritical logic, you can apply different constraints, such as a maximum area constraint, on the block.

Figure 3-6 shows how to separate logic with different design goals.

Figure 3-6 Blocks With Different Constraints



Partitioning by Compile Technique

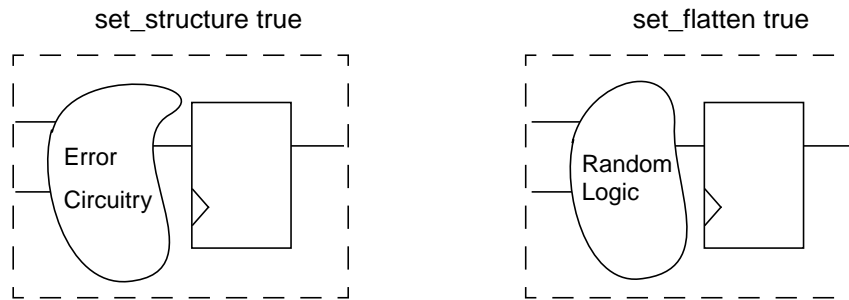
Partition logic that requires different compile techniques into separate blocks. Use this method when the design contains highly structured logic along with random logic.

- Highly structured logic, such as error detection circuitry, which usually contains large exclusive OR trees, is better suited to structuring.
- Random logic is better suited to flattening.

See “Logic-Level Optimization” in Chapter 8 for more information about these compile techniques.

Figure 3-7 shows the logic separated into different blocks.

Figure 3-7 Blocks With Different Compile Techniques



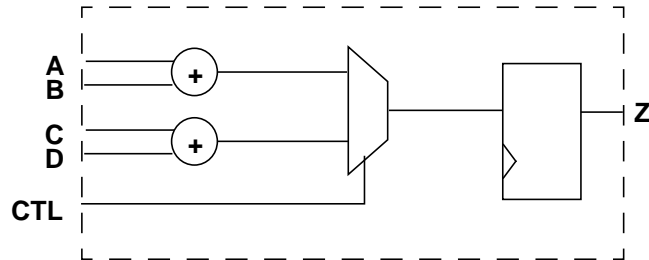
Keeping Sharable Resources Together

Design Compiler can share large resources, such as adders or multipliers, but resource sharing can occur only if the resources belong to the same VHDL process or Verilog always block.

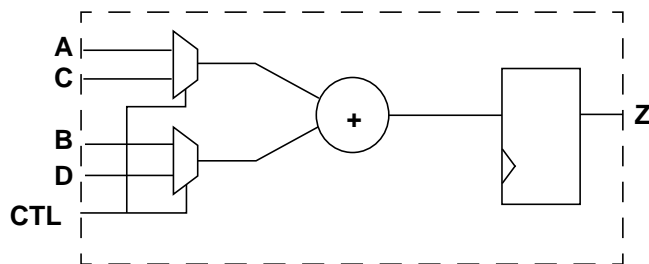
For example, if two separate adders have the same destination path and have multiplexed outputs to that path, keep the adders in one VHDL process or Verilog always block. This approach allows Design Compiler to consider resource sharing (using one adder instead of two) if the constraints allow sharing. Figure 3-8 shows possible implementations of this example logic.

Figure 3-8 Keep Sharable Resources in the Same Process

Unshared Resources



Shared Resources



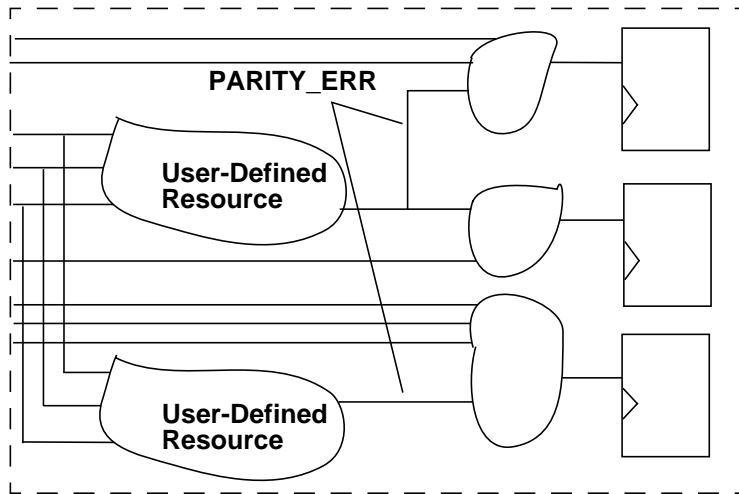
For more information about resource sharing, see the *HDL Compiler for Verilog Reference Manual* or the *VHDL Compiler Reference Manual*.

Keeping User-Defined Resources With the Logic They Drive

User-defined resources are user-defined functions, procedures, macro cells, or user-created DesignWare components. Design Compiler cannot automatically share or create multiple instances of user-defined resources. Keeping these resources with the logic they drive, however, gives you the flexibility to split the load by manually inserting multiple instantiations of a user-defined resource if timing goals cannot be achieved with a single instantiation.

Figure 3-9 illustrates splitting the load by multiple instantiation when the load on the signal `PARITY_ERR` is too heavy to meet constraints.

Figure 3-9 Duplicating User-Defined Resources



Isolating Special Functions

Isolate special functions (such as I/O pads, clock generation circuitry, boundary scan logic, and asynchronous logic) from the core logic. Figure 3-10 shows the recommended partitioning for the top level of the design.

If you make blocks very big, compile runtimes can be lengthy. Although Design Compiler has no inherent block size limit, you can improve runtimes by tailoring the block size to the performance and memory capacity of your workstation.

HDL Coding for Synthesis

HDL coding is the foundation for synthesis, because it implies the initial structure of the design. When writing your HDL source code, always consider the hardware implications of the code. A good coding style can generate smaller and faster designs. This section provides information to help you write efficient code so you can achieve your design target in the shortest possible time.

Topics include

- Writing technology-independent HDL
- Using HDL constructs
- Writing effective code

Writing Technology-Independent HDL

The goal of high-level design using a completely automatic synthesis process is to have no instantiated gates or flip-flops. Meeting this goal results in readable, concise, and portable high-level HDL code that can be transferred to other vendors or to future processes.

In some cases, the (V)HDL Compiler product requires compiler directives to provide implementation information while still maintaining technology independence. HDL Compiler implements all

compiler directives as special comments (indicated by `// synopsis`). VHDL Compiler implements some compiler directives as special comments (indicated by `-- synopsis`) and some as Synopsys attributes. Synopsys provides a VHDL package called `synopsys` that defines all the Synopsys attributes. When using Synopsys attributes in VHDL code, you must include the `synopsys` package.

The following sections discuss various methods for keeping your HDL code technology-independent.

Inferring Components

(V)HDL Compiler provides the capability to infer the following components:

- Multiplexers
- Registers
- Three-state drivers
- Multibit components

The following sections provide information about these inference capabilities. For additional information and examples, see the *HDL Compiler for Verilog Reference Manual*, the *VHDL Compiler Reference Manual*, and the *HDL Coding Styles: Synthesis Application Note*.

Inferring Multiplexers

(V)HDL Compiler can infer a generic multiplexer cell (`MUX_OP`) from case statements in your HDL code. If your target technology library contains at least a 2-to-1 multiplexer cell, Design Compiler maps the inferred `MUX_OP`s to multiplexer cells in the target technology library.

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.

Use the `infer_mux` compiler directive to control multiplexer inference. When attached to a block, the `infer_mux` directive forces multiplexer inference for all case statements in the block. When attached to a case statement, the `infer_mux` directive forces multiplexer inference for that specific case statement.

Inferring Registers

Register inference allows you to specify technology-independent sequential logic in your designs. 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.

(V)HDL Compiler infers a D latch whenever you do not specify the resulting value for an output under all conditions, as in an incompletely specified if or case statement. (V)HDL Compiler can also infer SR latches and master-slave latches.

(V)HDL Compiler infers a D flip-flop whenever the sensitivity list of a Verilog always block or VHDL process includes an edge expression (a test for the rising or falling edge of a signal). (V)HDL Compiler can also infer JK flip-flops and toggle flip-flops.

Mixing Register Types

For best results, restrict each Verilog always block or VHDL process to a single type of register inferencing: latch, latch with asynchronous set or reset, flip-flop, flip-flop with asynchronous set or reset, or flip-flop with synchronous set or reset.

Be careful when mixing positive- and negative-edge-triggered flip-flops in your design. If a module infers both positive- and negative-edge-triggered flip-flops and the target technology library does not contain a negative-edge-triggered flip-flop, Design Compiler generates an inverter in the clock tree for the negative-edge clock.

Inferring Registers Without Control Signals

When inferring registers without control signals, make the data and clock pins controllable from the input ports or through combinational logic. If a gate-level simulator cannot control the data or clock pins from the input ports or through combinational logic, the simulator cannot initialize the circuit and the simulation fails.

Inferring Registers With Control Signals

You can initialize or control the state of a flip-flop, using either an asynchronous or a synchronous control signal.

To infer asynchronous control signals on latches, use the `async_set_reset` compiler directive (attribute in VHDL) to identify the asynchronous control signals. (V)HDL Compiler automatically identifies asynchronous control signals when inferring flip-flops.

To infer synchronous resets, use the `sync_set_reset` compiler directive (attribute in VHDL) to identify the synchronous controls.

Inferring Three-State Drivers

Assign the high-impedance value (1'bz in Verilog, 'Z' in VHDL) to the output pin to infer three-state gates. Three-state logic reduces the testability of the design and makes debugging difficult. Where possible, replace three-state buffers with a multiplexer.

Never use high-impedance values in a conditional expression. (V)HDL Compiler always evaluates expressions compared to high-impedance values as false, which can cause the gate-level implementation to behave differently than the RTL description.

The *HDL Compiler for Verilog Reference Manual* and the *VHDL Compiler Reference Manual* contain additional information about three-state inference.

Inferring Multibit Components

Multibit inference allows you to map multiplexers, registers, and three-state drivers to regularly structured logic or multibit library cells. Using multibit components can result in

- Smaller area and delay, due to shared transistors and optimized transistor-level layout
- Reduced clock skew in sequential gates
- Lower power consumption by the clock in sequential banked components
- Improved regular layout of the data path

Multibit components might not be efficient in the following instances:

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

You must weigh the benefits of multibit components against the loss of optimization flexibility when deciding whether to map to multibit or single-bit components.

Attach the `infer_multibit` compiler directive to bused signals to infer multibit components. You can also change between a single-bit and a multibit implementation after optimization by using the `create_multibit` and `remove_multibit` commands.

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

Using Synthetic Libraries

To help you achieve optimal performance, Synopsys supplies a synthetic library. This library contains efficient implementations for adders, incrementers, comparators, and signed multipliers.

Design Compiler selects a synthetic component to meet the given constraints. After Design Compiler assigns the synthetic structure, you can always change to another type of structure by modifying your constraints. If you ungroup the synthetic cells or write the netlist to a text file, however, Design Compiler can no longer recognize the synthetic component and cannot perform implementation reselection.

The *HDL Compiler for Verilog Reference Manual* and the *VHDL Compiler Reference Manual* contain additional information about using compiler directives to control synthetic component use. The *DesignWare Foundation Library Databook* contains additional information about synthetic libraries and provides examples of how to infer and instantiate synthetic components.

The following examples use the `label`, `ops`, `map_to_module`, and implementation compiler directives to implement a 32-bit carry-lookahead adder.

Example 3-1 32-Bit Carry-Lookahead Adder (Verilog)

```

module add32 (a, b, cin, sum, cout);
  input [31:0] a, b;
  input cin;
  output [31:0] sum;
  output cout;
  reg [33:0] temp;

  always @(a or b or cin)
  begin : add1
    /* synopsys resource r0:
       ops = "A1",
       map_to_module = "DW01_add",
       implementation = "cla"; */
    temp = ({1'b0, a, cin} + // synopsys label A1
           {1'b0, b, 1'b1});
  end

  assign {cout, sum} = temp[33:1];

endmodule

```

Example 3-2 32-Bit Carry-Lookahead Adder (VHDL)

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

library synopsys;
use synopsys.attributes.all;

entity add32 is
  port (a,b : in std_logic_vector (31 downto 0);
        cin : in std_logic;
        sum : out std_logic_vector (31 downto 0);
        cout: out std_logic);
end add32;

architecture rtl of add32 is
  signal temp_signed : SIGNED (33 downto 0);

```

```
    signal op1, op2, temp : STD_LOGIC_VECTOR (33 downto 0);
    constant COUNT : UNSIGNED := "01";

begin
    infer: process ( a, b, cin )
        constant r0 : resource := 0;
        attribute ops of r0 : constant is "A1";
        attribute map_to_module of r0 : constant is "DW01_add";
        attribute implementation of r0 : constant is "cla";

        begin
            op1 <= '0' & a & cin;
            op2 <= '0' & b & '1';
            temp_signed <= SIGNED(op1) + SIGNED(op2); -- pragma
label A1
            temp <= STD_LOGIC_VECTOR(temp_signed);

            cout <= temp(33);
            sum <= temp(32 downto 1);
        end process infer;
end rtl;
```

Designing State Machines

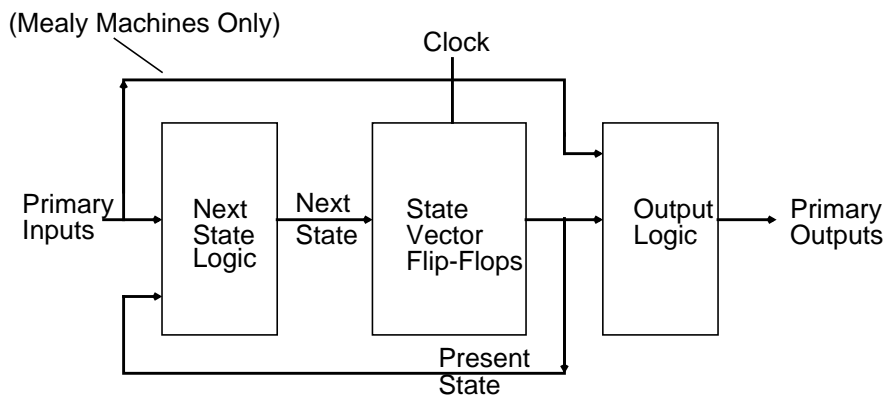
You can specify a state machine by using several different formats:

- Verilog
- VHDL
- State table
- PLA

If you use the `state_vector` and `enum` compiler directives in your HDL code, Design Compiler can extract the state table from a netlist. In the state table format, Design Compiler does not retain the `casex`, `casez`, and `parallel_case` information. Design Compiler does not optimize invalid input combinations and mutually exclusive inputs.

Figure 3-11 shows the architecture for a finite state machine.

Figure 3-11 Finite State Machine Architecture



Using an extracted state table provides the following benefits:

- State minimization can be performed.
- Trade offs between different encoding styles can be made.
- Don't care conditions can be used without flattening the design.
- Don't care state codes are automatically derived.

For information about extracting state machines and changing encoding styles, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*.

Using HDL Constructs

The following sections provide information guidelines about using specific HDL constructs. The information is divided into these sections:

- General HDL Constructs

- Verilog Macro Definitions
- VHDL Port Definitions

General HDL Constructs

The information in this section applies to both Verilog and VHDL.

Sensitivity Lists

Completely specify the sensitivity list for each Verilog always block or VHDL process. Incomplete sensitivity lists can result in simulation mismatches between the HDL and the gate-level design.

Example 3-3 Incomplete Sensitivity List (Verilog)

```
always @ (A)
  C <= A | B;
```

Example 3-4 Incomplete Sensitivity List (VHDL)

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

Value Assignments

Both Verilog and VHDL support the use of immediate and delayed value assignments in the RTL code. The hardware generated by immediate value assignments—implemented by Verilog blocking assignments (=) and VHDL variables (:=)—is dependent on the ordering of the assignments. The hardware generated by delayed value assignments—implemented by Verilog nonblocking assignments (<=) and VHDL signals (<=)—is independent of the ordering of the assignments.

For the most intuitive results,

- Use immediate value assignments within sequential Verilog always blocks or VHDL processes.
- Use delayed value assignments within combinational Verilog always blocks or VHDL processes.

if Statements

When an if statement used in a Verilog always block or VHDL process as part of a continuous assignment does not include an else clause, Design Compiler creates a latch. The following examples show if statements that generate latches during synthesis.

Example 3-5 Incorrect if Statement (Verilog)

```
if ((a == 1) && (b == 1))
    z = 1;
```

Example 3-6 Incorrect if Statement (VHDL)

```
if (a = '1' and b = '1') then
    z <= '1';
end if;
```

case Statements

If your if statement contains more than three conditions, consider using the case statement to improve the parallelism of your design and the clarity of your code. The following examples use the case statement to implement a 3-bit decoder.

Example 3-7 Using the case Statement (Verilog)

```
case ({a, b, c})
    3'b000: z = 8'b00000001;
    3'b001: z = 8'b00000010;
    3'b010: z = 8'b00000100;
    3'b011: z = 8'b00001000;
    3'b100: z = 8'b00010000;
```

```
3'b101: z = 8'b00100000;  
3'b110: z = 8'b01000000;  
3'b111: z = 8'b10000000;  
default: z = 8'b00000000;  
endcase
```

Example 3-8 Using the case Statement (VHDL)

```
case_value := a & b & c;  
CASE case_value IS  
  WHEN "000" =>  
    z <= "00000001";  
  WHEN "001" =>  
    z <= "00000010";  
  WHEN "010" =>  
    z <= "00000100";  
  WHEN "011" =>  
    z <= "00001000";  
  WHEN "100" =>  
    z <= "00010000";  
  WHEN "101" =>  
    z <= "00100000";  
  WHEN "110" =>  
    z <= "01000000";  
  WHEN "111" =>  
    z <= "10000000";  
  WHEN OTHERS =>  
    z <= "00000000";  
END CASE;
```

An incomplete case statement results in the creation of a latch. VHDL does not support incomplete case statements. In Verilog you can avoid latch inference by using either the default clause or the `full_case` compiler directive.

Although both the `full_case` directive and the default clause prevent latch inference, they have different meanings. The `full_case` directive asserts that all valid input values have been specified and no default clause is necessary. The default clause specifies the output for any undefined input values.

For best results, use the default clause instead of the `full_case` directive. If the unspecified input values are don't care conditions, using the default clause with an output value of x can generate a smaller implementation.

If you use the `full_case` directive, whenever the case expression evaluates to an unspecified input value, the gate-level simulation might not match the RTL simulation. If you use the default clause, simulation mismatches can occur only if you specified don't care conditions and the case expression evaluates to an unspecified value.

Constant Definitions

Use the Verilog ``define` statement or the VHDL constant statement to define global constants. Keep global constant definitions in a separate file. Use parameters (Verilog) or generics (VHDL) to define local constants.

Example 3-9 shows a Verilog code fragment that includes a global ``define` statement and a local parameter. Example 3-10 shows a VHDL code fragment that includes a global constant and a local generic.

Example 3-9 Using Macros and Parameters (Verilog)

```
// Define global constant in def_macro.v
`define WIDTH 128

// Use global constant in reg128.v
reg regfile[WIDTH-1:0];
```

```
// Define and use local constant in module foo
module foo (a, b, c);
    parameter WIDTH=128;
    input [WIDTH-1:0] a, b;
    output [WIDTH-1:0] c;
```

Example 3-10 Using Constants and Generics (VHDL)

```
-- Define global constant in synthesis_def.vhd
constant WIDTH : INTEGER := 128;

-- include global constants
library my_lib;
USE my_lib.synthesis_def.all;

-- Use global constant in entity foo
entity fool is
    port (a,b : in std_logic_vector(WIDTH-1 downto 0);
          c: out std_logic_vector(WIDTH-1 downto 0));
end fool;

-- Define and use local constant in entity foo
entity foo is
    generic (WIDTH_VAR : INTEGER := 128);
    port (a,b : in std_logic_vector(WIDTH-1 downto 0);
          c: out std_logic_vector(WIDTH-1 downto 0));
end foo;
```

Verilog Macro Definitions

In Verilog, macros are implemented using the ``define` statement. Use these guidelines for ``define` statements:

- Use ``define` statements only to declare constants.
- Keep ``define` statements in a separate file.

- Do not use nested ``define` statements.

Reading a macro that is nested more than twice is difficult. To make your code readable, do not use nested ``define` statements.

- Do not use ``define` inside module definitions.

When you use a ``define` statement inside a module definition, the local macro and the global macro have the same reference name but different values. Use parameters to define local constants.

VHDL Port Definitions

When defining ports in VHDL source code,

- Use the `STD_LOGIC` and `STD_LOGIC_VECTOR` packages.

Using `STD_LOGIC` avoids the need for type conversion functions on the synthesized design.

- Do not use the buffer port mode.

When you declare a port as a buffer, it must be used as a buffer throughout the hierarchy. To simplify synthesis, declare the port as an output, then define an internal signal that drives the output port.

Writing Effective Code

This section provides guidelines for writing efficient, readable HDL source code for synthesis. The guidelines cover

- Identifiers
- Expressions

- Functions
- Modules

Guidelines for Identifiers

A good identifier name conveys the meaning of the signal, the value of a variable, or the function of a module; without this information, the hardware descriptions are difficult to read.

Observe the following naming guidelines to improve the readability of your HDL source code:

- Ensure that the signal name conveys the meaning of the signal or the value of a variable without being verbose.

For example, assume that you have a variable that represents the floating point opcode for rs1. A short name, such as `frs1`, does not convey the meaning to the reader. A long name, such as `floating_pt_opcode_rs1`, conveys the meaning, but its length might make the source code difficult to read. Use a name such as `fpop_rs1`, which meets both goals.

- Use a consistent naming style for capitalization and to distinguish separate words in the name.

Commonly used styles include C, Pascal, and Modula.

- C style uses lowercase names and separates words with an underscore, for example, `packet_addr`, `data_in`, and `first_grant_enable`.
- Pascal style capitalizes the first letter of the name and first letter of each word., for example, `PacketAddr`, `DataIn`, and `FirstGrantEnable`.

- Modula style uses a lowercase letter for the first letter of the name and capitalizes the first letter of subsequent words, for example, packetAddr, dataIn, and firstGrantEnable.

Choose one convention and apply it consistently.

- Avoid confusing characters.

Some characters (letters and numbers) look similar and are easily confused, for example, O and 0 (zero); l and 1 (one).

- Avoid reserved words.
- Use the noun or noun followed by verb form for names, for example, AddrDecode, DataGrant, PCI_interrupt.
- Add a suffix to clarify the meaning of the name.

Table 3-2 shows common suffixes and their meanings.

Table 3-2 Signal Name Suffixes and Their Meanings

Suffix	Meaning
_clk	Clock signal
_next	Signal before being registered
_n	Active low signal
_z	Signal that connects to a three-state output
_f	Register that uses an active falling edge
_xi	Primary chip input
_xo	Primary chip output
_xod	Primary chip open drain output
_xz	Primary chip three-state output
_xbio	Primary chip bidirectional I/O

Guidelines for Expressions

Use the following guidelines for expressions:

- Use parentheses to indicate precedence.

Expression operator precedence rules are confusing, so you should use parentheses to make your expression easy to read. Unless you are using DesignWare resources, using parentheses has little effect on the generated logic. An example of a logic expression without parentheses that is difficult to read is

```
bus_select = a ^ b & c~^d|b^~e&^f[1:0];
```

- Replace repetitive expressions with function calls or continuous assignments.

If you use a particular expression more than two or three times, consider replacing the expression with a function or a continuous assignment that implements the expression.

Guidelines for Functions

Use these guidelines for functions:

- Do not use global references within a function.

In procedural code, a function is evaluated when it is called. In a continuous assignment, a function is evaluated when any of its declared inputs changes.

Avoid using references to nonlocal names within a function because the function might not be reevaluated if the nonlocal value changes. This can cause a simulation mismatch between the HDL description and the gate-level netlist. For example, the following Verilog function references the nonlocal name `byte_sel`:

```
function byte_compare;
  input [15:0] vector1, vector2;
  input [7:0] length;

  begin
    if (byte_sel)
      // compare the upper byte
    else
      // compare the lower byte
      ...
    end
  endfunction // byte_compare
```

- Be aware that task and function local storage is static.

Formal parameters, outputs, and local variables retain their values after a function has returned. The local storage is reused each time the function is called. This storage can be useful for debugging, but it also means that functions and tasks cannot be called recursively.

- Be careful when using component implication.

You can map a function to a specific implementation by using the `map_to_module` and `return_port_name` compiler directives. Simulation uses the contents of the function. Synthesis uses the gate-level module in place of the function. When you are using component implication, the RTL model and the gate-level model might be different. Therefore, the design cannot be fully verified until simulation is run on the gate-level design.

The following areas might require component instantiation or functional implication:

- Clock gating circuitry for power savings
- Asynchronous logic with potential hazards

This includes asynchronous logic and asynchronous signals that are valid during certain states.

- Data-path circuitry

This includes large multiplexers; instantiated wide banks of multiplexers; memory elements, such as RAM or ROM; and black box macro cells.

For more information about component implication, see the *HDL Compiler for Verilog Reference Manual* or the *VHDL Compiler Reference Manual*.

Guidelines for Modules

Use these guidelines for modules:

- Avoid using logic expressions when you pass a value through ports.

The port list can include expressions, but expressions complicate debugging. In addition, isolating a problem related to the bit field is difficult, particularly if that bit field leads to internal port quantities that differ from external port.

- Define local references as generics (VHDL) or parameters (Verilog). Do not pass generics or parameters into modules.

