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.

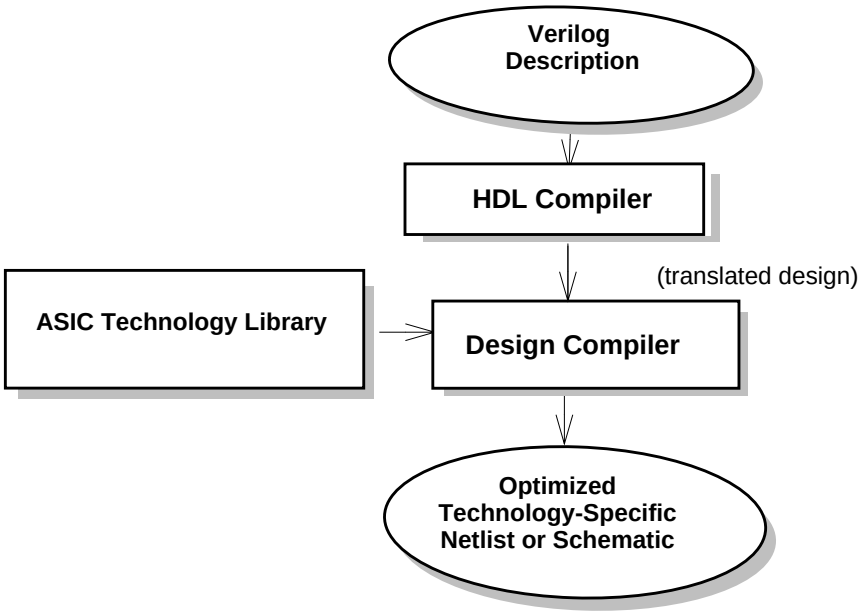
HDL descriptions play an important role in modern design methodology, for four main reasons:

- 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)

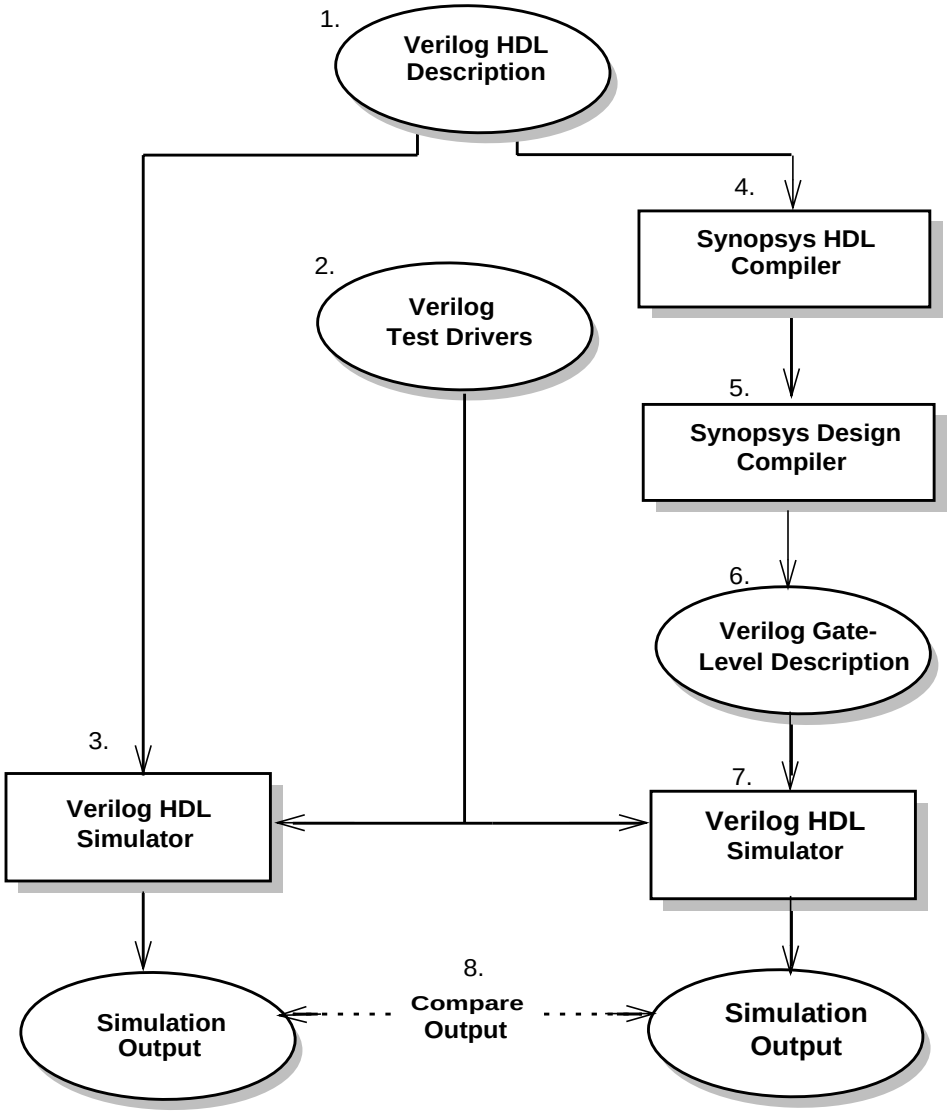
Synopsys Design Compiler can produce netlists and schematics in many commercial formats, including Verilog. It can convert existing gate-level netlists, sets of logic equations, or technology-specific circuits in another supported format to Verilog descriptions. The new Verilog descriptions document 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