

# **New HDL Compiler™**

## **Preliminary Information**

---

Version 2000.05, May 2000

Comments?

E-mail your comments about Synopsys  
documentation to [doc@synopsys.com](mailto:doc@synopsys.com)

# **SYNOPSYS®**

## Copyright Notice and Proprietary Information

Copyright © 2000 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

## Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

“This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of \_\_\_\_\_ and its employees. This is copy number \_\_\_\_\_.”

## Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

## Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

## Registered Trademarks

Synopsys, the Synopsys logo, AMPS, Arcadia, CMOS-CBA, COSSAP, Cyclone, DelayMill, DesignPower, DesignSource, DesignWare, dont\_use, Eagle Design Automation, Eagle*i*, EPIC, ExpressModel, Formality, in-Sync, Logic Automation, Logic Modeling, Memory Architect, ModelAccess, ModelTools, PathBlazer, PathMill, PowerArc, PowerMill, PrimeTime, RailMill, SmartLicense, SmartModel, SmartModels, SNUG, SOLV-IT!, SolvNET, Stream Driven Simulator, Synthetic Designs, TestBench Manager, and TimeMill are registered trademarks of Synopsys, Inc.

## Trademarks

ACE, Behavioral Compiler, BOA, BRT, CBA, CBAII, CBA Design System, CBA-Frame, Cedar, Chip Architect, Chronologic, CoreMill, DAVIS, DC Expert, DC Expert *Plus*, DC Professional, DC Ultra, DC Ultra Plus, Design Advisor, Design Analyzer, DESIGN (ARROWS), Design Compiler, DesignTime, DesignWare Developer, Direct RTL, Direct Silicon Access, dont\_touch, dont\_touch\_network, DW 8051, DWPCI, Eagle, EagleV, ECL Compiler, ECO Compiler, Falcon Interfaces, Floorplan Manager, Foundation, FoundryModel, FPGA Compiler, FPGA Compiler II, FPGA *Express*, Frame Compiler, Fridge, General Purpose Post-Processor, GPP, HDL Advisor, HDL Compiler, Integrator, Interactive Waveform Viewer, Liberty, Library Compiler, Logic Model, MAX, ModelSource, Module Compiler, MS-3200, MS-3400, Nanometer Design Experts, Nanometer IC Design, Nanometer Ready, Odyssey, PowerCODE, PowerGate, Power Compiler, ProFPGA, ProMA, Protocol Compiler, RMM, RoadRunner, RTL Analyzer, Schematic Compiler, Shadow Debugger, Silicon Architects, SmartModel Library, Source-Level Design, SWIFT, Synopsys Graphical Environment, Synopsys ModelFactory, Test Compiler, Test Compiler Plus, Test Manager, TestGen, TestSim, TimeTracker, Timing Annotator, Trace-On-Demand, VCS, VCS Express, VCSi, VERA, VHDL Compiler, VHDL System Simulator, Visualize, VMC, and VSS are trademarks of Synopsys, Inc.

## Service Marks

TAP-in is a service mark of Synopsys, Inc.

All other product or company names may be trademarks of their respective owners.

Printed in the U.S.A.

Document Order Number: (not applicable)  
New HDL Compiler, Preliminary Information, v2000, 05

# Table of Contents

---

## About this Manual

### 1. New HDL Compiler Overview

Using the New HDL Compiler .....	1-2
New HDL Compiler Features .....	1-3
New HDL Compiler Limitations .....	1-4
Verilog 2000 Features .....	1-5

### 2. New Feature Examples

generate Statements .....	2-2
Arrays of Instances .....	2-3
defparam Statement .....	2-5
Combinational while Loop .....	2-7
Enumerated Types .....	2-9
Enumerated Type Definition .....	2-9
Where Enumerated Types Occur .....	2-9

Working with Enumerated Types . . . . .	2-9
Enable Enumerated Type Variables. . . . .	2-10
Enumerated Type Example . . . . .	2-10
Enumerated Type Limitations . . . . .	2-13
Resource Sharing for Conditional Expressions. . . . .	2-14
Improved Logic for Mem[Addr] . . . . .	2-16
More Flexible Index Bounds Checking . . . . .	2-18
Divide Operator . . . . .	2-20
Blocking / Non-Blocking Procedural Assignments . . . . .	2-21
3. New HDL Compiler Differences	
Implementation Differences. . . . .	3-2
New Syntax Error Notifications . . . . .	3-3
Simulator Semantic Changes - Template Directive . . . . .	3-5
Appendix A. Variables	
New Public Variables. . . . .	D-2
Variables Not Supported . . . . .	D-5
Appendix B. New Verilog Netlist Reader	

## List of Tables

---

Table 1-1	New HDL Compiler Features — Verilog 2000 Support. .	1-5
Table A-1	New Public Variables . . . . .	D-2
Table A-2	Variables Not Supported in the New HDL Compiler . . . .	D-5



# About this Manual

---

This manual provides preliminary information about the new HDL Compiler.

This preface covers the following topics:

- Audience
- Supported Versions
- Other Sources of Information

---

## Audience

This preliminary product information is provided to customers who are interested in testing the initial version of the new HDL Compiler.

---

## Supported Versions

This manual supports Version 2000.05 of the new HDL Compiler.

---

## Other Sources of Information

For more information about the new HDL Compiler and other Synopsys products, refer to the following sections.

---

### Related Publications

Refer to the *HDL Compiler for Verilog Reference Manual 2000.05, Synthesis* for additional information on the HDL Compiler.

---

### SOLV-IT! Online Help

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

Access SOLV-IT! through e-mail or through the World Wide Web (WWW). For more information about SOLV-IT!, send e-mail to

`solvitfb@synopsys.com`

or view the Synopsys Web page at

`http://www.synopsys.com`



---

## Customer Support

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

- Send e-mail to  
`support_center@synopsys.com`
- Call (650) 694-4200 outside the continental United States or call (800) 245-8005 inside the continental United States, from 7 AM to 5:30 PM Pacific Standard Time, Monday through Friday.
- Send a fax to (650) 965-2539.



# 1

## New HDL Compiler Overview

---

The new HDL Compiler is a completely redesigned HDL Compiler offering the following new features and functionality:

- Faster and more predictable elaboration time  
Approximately five times faster in elaborating a design  
(The speedup in elaboration time will vary depending on hardware and design used.)
- Higher capacity  
Uses approximately one-third less memory  
(The reduction in memory usage will vary depending on hardware and design used.)
- Increased language support

This document describes the initial implementation of the new HDL Compiler for Verilog and provides new feature examples.

---

## Using the New HDL Compiler

To enable the new HDL Compiler, follow the steps below:

1. From `dc_shell`, set the following switch as shown:

```
hdlin_enable_presto = true
```

2. Invoke the applicable command:

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

```
analyze -f verilog <file.v>
```

```
elaborate <top design name>
```

3. To work with this initial version of the new Verilog HDL Compiler, use the *HDL Compiler for Verilog Reference Manual 2000.05* for all tasks except those described in this document.

---

## New HDL Compiler Features

Initially, the new HDL Compiler supports only Verilog designs. Later releases will support both Verilog and VHDL. Included features are listed below. See “New Feature Examples” in Chapter 2.

- generate Statements
- Arrays of Instances
- defparam Statement
- Combinational while Loop
- Enumerated Types
- Resource Sharing for Conditional Expressions
- Improved Logic for Mem[Addr]
- More Flexible Index Bounds Checking
- Divide Operator
- Blocking / Non-Blocking Procedural Assignments

---

## New HDL Compiler Limitations

The following features are not in the initial release but will be included in later releases:

- Support for VHDL files
- Support for use with other Synopsys products, such as Behavior Compiler, BC View, and Power Compiler
- Manual resource sharing
- Manual implementation selection
- Implicit state machines

---

## Verilog 2000 Features

Table 1-1 lists the features in the new HDL Compiler and identifies if the feature supports the Verilog 2000 LRM.

Note:

All Verilog 1995 LRM features are a subset of the Verilog 2000 LRM.

*Table 1-1 New HDL Compiler Features — Verilog 2000 Support*

New HDL Compiler Features	Verilog 1995 LRM	New Feature in Verilog 2000 LRM
generate Statements		X
Arrays of Instances	X	
defparam Statement	X	
Combinational while Loop	X	
Enumerated Types	X	
Resource Sharing for Conditional Expressions	X	
Improved Logic for Mem[Addr]	X	
More Flexible Index Bounds Checking	X	
Divide Operator	X	
Blocking / Non-Blocking Procedural Assignments	X	





# 2

## New Feature Examples

---

This chapter provides the following new feature examples:

- generate Statements on page 2-2
- Arrays of Instances on page 2-3
- defparam Statement on page 2-5
- Combinational while Loop on page 2-7
- Resource Sharing for Conditional Expressions on page 2-14
- Enumerated Types on page 2-9
- Improved Logic for Mem[Addr] on page 2-16
- More Flexible Index Bounds Checking on page 2-18
- Divide Operator on page 2-20
- Blocking / Non-Blocking Procedural Assignments on page 2-21

---

## generate Statements

The generate statement is a new IEEE 1364-2000 Verilog feature that makes the language easier to use by simplifying the code needed to repeat logic equations or component instantiations.

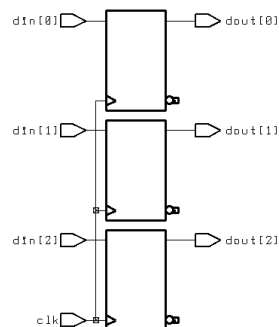
In Example 2-1, a generate loop is constructed around a flip-flop and executed three times making three flip-flops. See Figure 2-1. This example illustrates generate statement usage.

For additional information, see the IEEE Std P1364-2000, *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*.

### Example 2-1 generate Statement

```
module gen (clk, din, dout);  
    parameter N = 3;  
    input clk;  
    input [N-1:0] din;  
    output [N-1:0] dout;  
    reg [N-1:0] dout;  
    genvar i;  
    generate  
        for (i = 0; i < N; i = i+1)  
            always @ (posedge clk)  
                dout[i] <= din[i];  
    endgenerate  
endmodule
```

Figure 2-1 generate Statement



---

## Arrays of Instances

The arrays of instances is a IEEE 1364-1995 synthesis feature fully supported by the new HDL Compiler. This feature enables instantiations of modules that contain a range specification, which in turn, allows an array of instances to be created making the language easier to use by simplifying and reducing the code required to generate gates.

See the IEEE 1364-1995 standard, section 7.1, for additional information.

In Example 2-2,

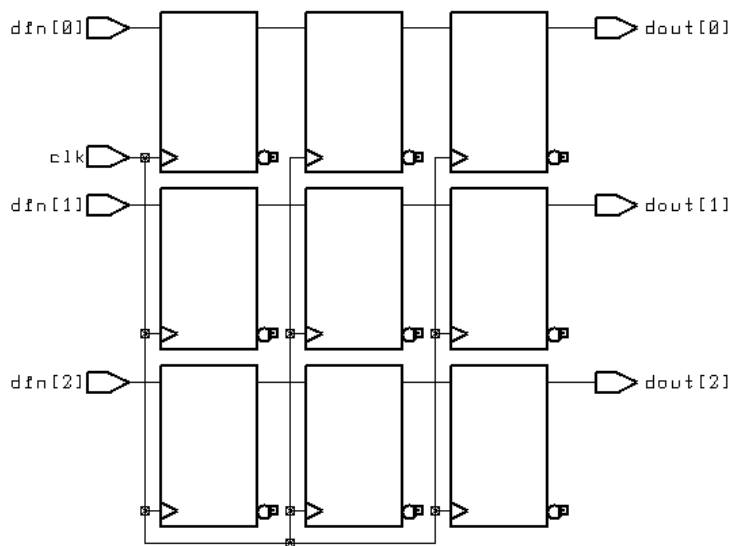
1. The module `pipe` pipelines two registers resulting in one pipe two deep.
2. The module `array_inst` uses the `pipe` module to make an array three wide adding one more pipe to each output and resulting in an array of three pipes three deep. See Figure 2-2.

## Example 2-2 Arrays of Instances

```
module array_inst (clk, din, dout);
    parameter N=3;
    input clk;
    input [N-1:0] din;
    output [N-1:0] dout;
    reg [N-1:0] dout;
    wire [N-1:0] idout;
    pipe pipe1[N-1:0] (.D(din), .CK(clk), .Q(idout));
    always @ (posedge clk)
        begin
            dout <= idout;
        end
endmodule

module pipe ( D, CK, Q );
    input D, CK;
    output Q;
    reg Q0, Q;
    always @ (posedge CK)
        begin
            Q0 <= D;
            Q <= Q0;
        end
    end
endmodule
```

Figure 2-2 Arrays of Instances



---

## defparam Statement

The defparam statement is an IEEE 1364-1995 synthesis feature fully supported by the new HDL Compiler. This feature makes the language easier to use by allowing parameter values to be changed in any module instance.

In Example 2-3, the original value of parameter p3 is 300. Using the defparam statement, the value is changed as it is passed from the top to the middle and finally to the bottom design in which p3 changes from 300 to 2. The defparam statement enables each module instantiation to customize declared parameters. The result of the code in Example 2-3 is:

- p1=0
- p2=1
- p3=2
- p4=3

See the IEEE 1364-1995 Verilog standard section 12.2 for additional information.

### *Example 2-3 Defparam Statement*

```
module top (x, y, q);

    input x, y;
    output q;

    defparam a.b.p3 = 2, a.b.p4=3;
    middle a (x, y, q);

endmodule

module middle (x, y, q);

    input x, y;
    output q;

    bottom #(0,1) b (x, y, q);

endmodule

module bottom (a, b, q);
    input a, b;
    output q;

    parameter p1 = 100;
    parameter p2 = 200;
    parameter p3 = 300;
    parameter p4 = 400;

    assign q = a & b;

endmodule
```

---

## Combinational while Loop

The combinational while loop statement is an IEEE 1364-1995 synthesis feature supported by the new HDL Compiler. This feature makes the language easier to use by enabling a more flexible coding style.

In Example 2-4, the current HDL Compiler produces error HDL-98 and stops because an event control signal (such as a clock edge) is needed. With the new HDL Compiler, this code produces gates (see Figure 2-3) and an event control signal is not necessary. This example illustrates combinational while loop statement usage.

**Note:**

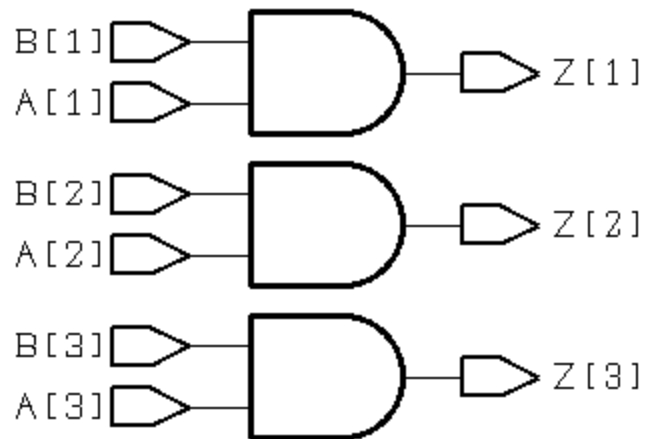
The loop iterative bound must be statically determinable or an error is reported. If the loop bound can not be determined, the program will limit the maximum number of iterations to the value set by the `hdlin_loop_iteration_limit` variable and report an ELAB-900 error.

See the IEEE 1364-1995 Verilog standard, section 9.6, for additional information.

### Example 2-4 Combination while Loop

```
module s2 (a, b, z);  
  
    parameter N = 3;  
    input [N:1] a, b;  
    output [N:1] z;  
  
    reg [N:1] z;  
    integer i;  
    always @(a or b or z)  
    begin  
        i = N;  
        while (i)  
            begin  
                z[i] = b[i] & a[i];  
                i = i - 1;  
            end  
    end  
  
endmodule
```

Figure 2-3 Combination while Loop





---

## Enumerated Types

The new HDL Compiler simplifies equality comparisons and detection of full cases in designs that contain enumerated types.

---

### ENUMERATED TYPE DEFINITION

A variable has an enumerated type when it can only take on a subset of the values that it could possibly represent. For example, if a two-bit value can be set to 0, 1, or 2, but is never assigned to 3, then it has the enumerated type {0, 1, 2}.

---

### WHERE ENUMERATED TYPES OCCUR

Enumerated types commonly occur in finite state machine state encodings. When the number of states needed is not a power of two, certain state values can never occur. In finite state machines with one-hot encodings, many values can never be assigned to the state vector. For example, for a vector of length  $n$ , there are  $n$  one-hot values, so there are  $(2^n - n)$  values that will never be used.

---

### WORKING WITH ENUMERATED TYPES

The new HDL Compiler can uncover some enumerated types automatically; user directives can be used in other situations. When all variable assignments are within a module, the new HDL Compiler usually detects if the variable has an enumerated type. If the variable is assigned a value that depends on an input port, or if the design

assigns individual bits of the variable separately, the new HDL Compiler requires the `/* synopsys enum */` directive in order to consider the variable as having an enumerated type.

---

## Enable Enumerated Type Variables

To enable the new HDL Compiler to find enumerated types, set the following variable as shown:

```
hdlin_infer_enumerated_types=true
```

To enable the new HDL Compiler to perform simplification of equality comparisons, set the following variable as shown:

```
hdlin_optimize_enum_types=true
```

---

## Enumerated Type Example

Example 2-5 illustrates how the new HDL Compiler optimizes enumerated types better than the current HDL Compiler.

In Example 2-5, the `current_state` register has a one-hot enumerated type and can only be assigned the parameters 'red', 'yellow', and 'green'. The value does not depend on input port values. The new HDL Compiler uses this information to determine that the case statement is a `full_case`. All possible values of `current_state` are covered because `current_state` has the enumerated type which contains only {001, 010, 100}.

To implement the case statement, the compiler needs to compare `current_state` against the values 001, 010, and 100 and decide which branch should execute. Because `current_state` is known to have a one-hot type, the new HDL Compiler can simplify the comparisons.

Rather than using the comparison (`current_state == 3'b001`), the new HDL Compiler substitutes (`current_state[0]`) because there is only one possible value for `current_state` for which `current_state[0] == 1'b1` and that is 3'b001. Furthermore, if `current_state` were compared against any values except {001, 010, 100}, the new HDL Compiler would simplify the comparison to false. See Figure 2-4 and Figure 2-5.

#### *Example 2-5 Enumerated Type*

```
module enum (clk, rst, current_state);

    input clk, rst;
    output [2:0] current_state;

    reg [2:0] current_state;
    parameter red = 3'b001,
               green = 3'b010,
               yellow = 3'b100;
    always @ (posedge clk or posedge rst)
        if(rst)
            current_state = red;
        else
            case (current_state)
                red:
                    current_state = green;
                green:
                    current_state = yellow;
                yellow:
                    current_state = red;
            endcase

endmodule
```

Figure 2-4 Current HDL Compiler Gates

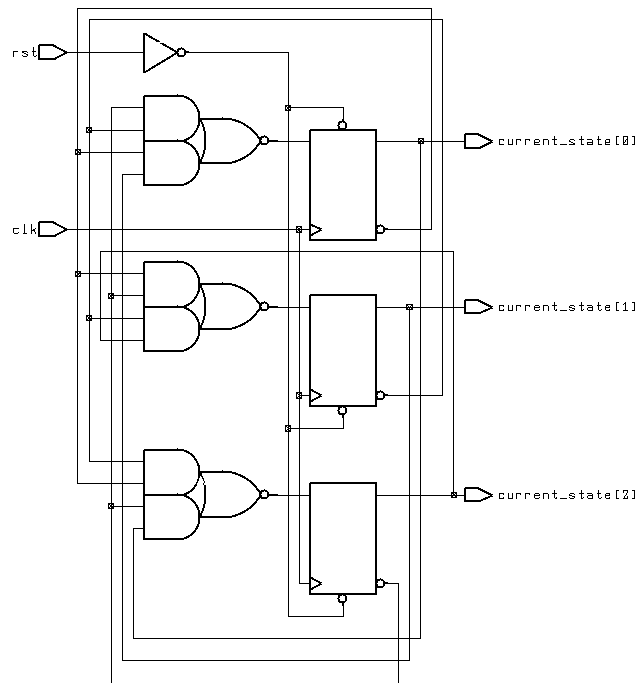
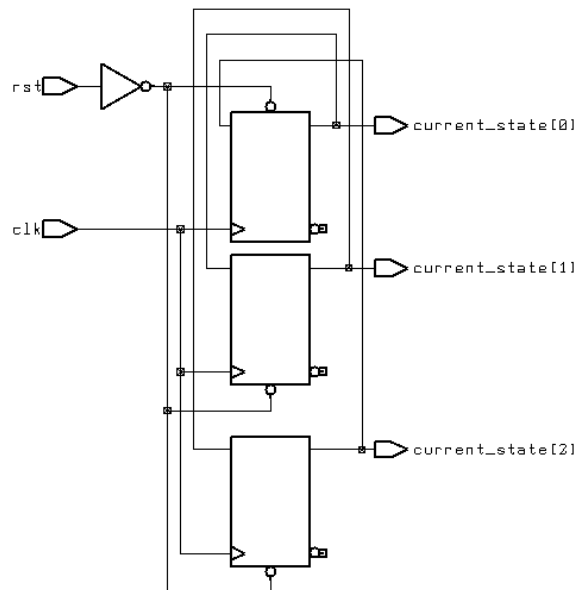


Figure 2-5 New HDL Compiler Gates



---

## Enumerated Type Limitations

Enumerated type optimizations may cause some spurious verification failures. One example of this is the comparison simplification from `(current_state == 3'b001)` to `(current_state[0])` in Example 2-5.

Most tools *do not know* that when `current_state[0]` is 1'b1, `current_state[1]` and `current_state[2]` *must* equal 0. This is difficult to figure out once the design is in netlist format. Formal verification tools will report that the new HDL Compiler design is not the same as the current HDL Compiler design because they are not checking bits 1 and 2 of `current_state`. Nonetheless, the new HDL Compiler design should operate exactly as the current HDL Compiler design but with better delay and area.

---

## Resource Sharing for Conditional Expressions

The new HDL Compiler makes the language easier to use by enabling the conditional operator to support resource sharing. This simplifies and reduces the code required to generate gates.

From the code in Example 2-6, the current HDL Compiler creates a circuit using an adder, subtractor, and mux (no resources sharing) while the new HDL Compiler circuit uses a DesignWare resource — a single device that optimizes the logic of the adder and subtractor into one and uses a select signal to choose addition or subtraction. See Figure 2-6 and Figure 2-7.

### *Example 2-6* Resource Sharing for Conditional Expressions

```
module datapath (a, b, sel, dout);

    parameter N = 7;
    input [N:0] a,b;
    input sel;
    output [N:0] dout;

    reg [N:0] dout;
    always @ (a or b)
        dout = sel ? (a + b) : (a - b);

endmodule
```

Figure 2-6 Current HDL Compiler Gates

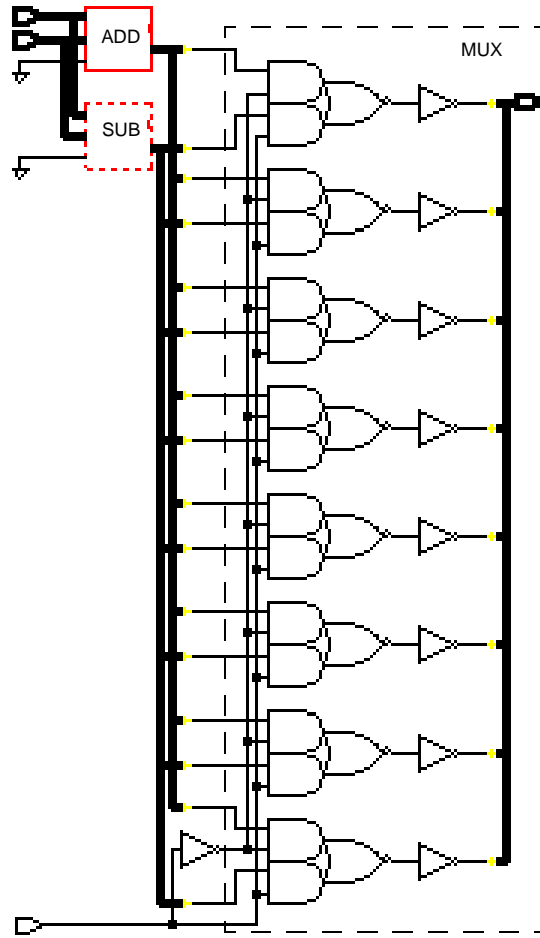
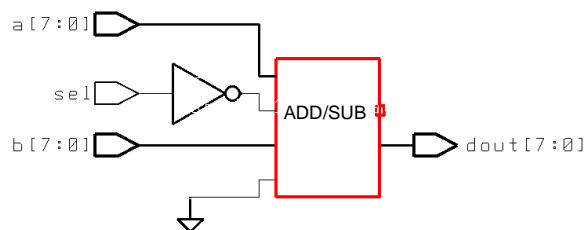


Figure 2-7 New HDL Compiler Gates



---

## Improved Logic for Mem[Addr]

Improved Mem[Addr] logic enables the new HDL Compiler to use the following two new DesignWare components.

- A decoder that decodes inputs
- A mux that multiplexes outputs

Example 2-7 illustrates Mem[addr] usage with the decoder decoding the input address and the mux multiplexing the output data. See Figure 2-8.

### *Example 2-7 Mem[Addr]*

```
module mem (addr, din, wr, dout);

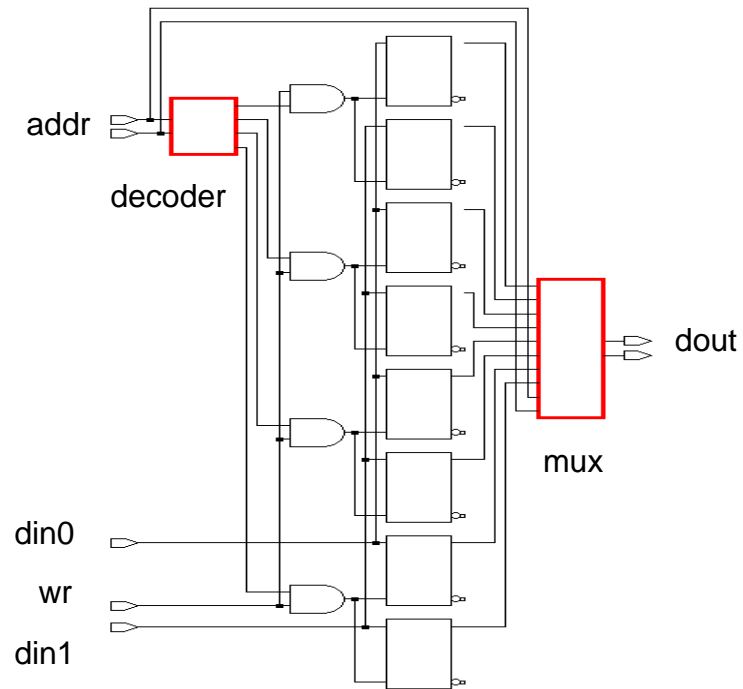
    input wr;
    input  [1:0] addr;
    input  [1:0] din;
    output [1:0] dout;

    reg [1:0] dout ;
    reg [1:0] mem [3:0];
    always @ (addr or wr or din)
        begin
            if (wr == 1) begin
                mem[addr] <= din;
                dout <= 2'bXX;
            end
            else
                dout <= mem[addr];
            end
        end

endmodule
```



Figure 2-8 Mem[Addr]



---

## More Flexible Index Bounds Checking

The new HDL Compiler improves index bounds checking.

Currently, the code in Example 2-8 produces an error and stops because “ $i + \text{offset} - 8$ ” is sometimes less than zero and “ $i + \text{offset}$ ” is sometimes greater than seven.

In the new HDL Compiler, the code runs error free and creates gates. See Figure 2-9. This example illustrates index bounds checking usage.

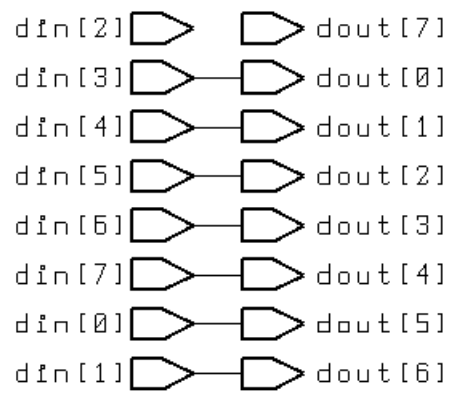
### *Example 2-8 Index Bounds Checking*

```
module index (din, dout);

    parameter offset = 3;
    input [7:0] din;
    output [7:0] dout;

    reg [7:0] dout;
    integer i;
    always @ (din)
        begin
            for (i=0; i<7; i=i+1) begin
                if (i + offset > 7)
                    dout[i]<= din[i + offset - 8];
                else
                    dout[i] = din[i + offset];
            end
        end
endmodule
```

*Figure 2-9 Index Bounds Checking*



---

## Divide Operator

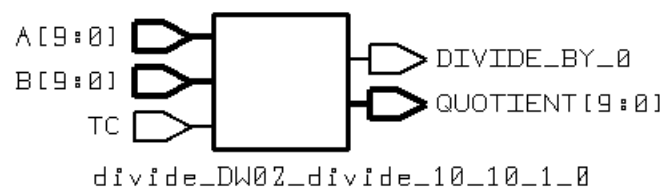
The divide operator is an IEEE 1364-1995 synthesis feature fully supported by the new HDL Compiler.

In the current HDL Compiler, the code in Example 2-9 generates the HDL-65 error message: “operands to divide must be a constant” and fails to produce gates. In the new HDL Compiler, this constraint is removed. See Figure 2-10. To use this feature you must have a license for DesignWare divide or DWOL library.

### *Example 2-9 Divide Operator*

```
module divide (a, b, z);  
  
    input  [9:0] a, b;  
    output [8:0] z;  
  
    assign z= a / b;  
  
endmodule
```

*Figure 2-10 Divide Operator*



---

## Blocking / Non-Blocking Procedural Assignments

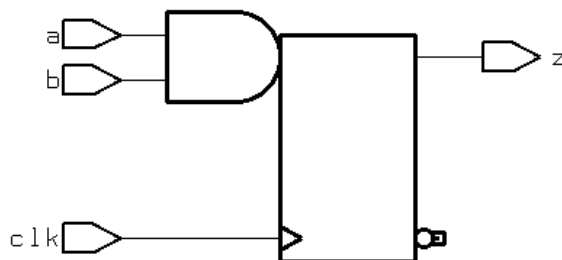
The new HDL Compiler supports both blocking and non-blocking assignments of the same variable.

In the current HDL Compiler, the code in Example 2-10 generates a VE-57 error and the code halts. In the new HDL Compiler, gates are generated with only a VER-35 warning. This example illustrates blocking and non-blocking assignment usage.

### *Example 2-10* Blocking and Non-Blocking Assignments

```
module blocking (clk, a, b, z);  
  
    input  clk, a, b;  
    output z;  
  
    reg    z;  
    always @ (posedge clk)  
    begin  
        z <= a & b;  
        z = a | b;  
    end  
  
endmodule
```

*Figure 2-11* Blocking and Non-Blocking Assignments





# 3

## New HDL Compiler Differences

---

The following sections describe implementation, syntax, and semantic differences between the current HDL Compiler and the new HDL Compiler:

- “Implementation Differences” on page 3-2
- “New Syntax Error Notifications” on page 3-3
- “Simulator Semantic Changes - Template Directive” on page 3-5

---

## Implementation Differences

Often a single HDL design source can result in several different design implementations. In such cases, the new HDL Compiler and the current HDL Compiler produce different but functionally equal designs.

In some cases, differences occur because the new HDL Compiler treatment of don't care values adheres closer to the IEEE Standard 1364-1995. Many portions of the current HDL Compiler were written before this standard was available. The new HDL Compiler also implements the generate statement which is in the upcoming IEEE 1364-2000 Verilog standard.

When generated logic is different between the designs, the new HDL Compiler typically provides switches so users can control behavior with an option.



---

## New Syntax Error Notifications

Under certain conditions, the new HDL Compiler outputs warnings when incorrect syntax is used; the current HDL Compiler generates no errors. See Example 3-1 through Example 3-3. Incorrect code is in the line marked with an error.

### *Example 3-1 Syntax Error Notification*

```
module dif_case (sel, a, b, c, d, dout);  
  
    input        a, b, c, d;  
    input  [1:0] sel;  
    output       dout;  
  
    reg          dout;  
    always @ (sel or a or b or c or d)  
    begin  
        case (sel)  
            2'b00: dout = a;  
            2'b01: dout = b;  
            2'b10: dout = c;  
            2'b11: dout = d;  
        endcase;  
    end  
endmodule
```

← Error: It is illegal to use a semicolon after endcase.  
The new HDL Compiler notifies the user.  
The current HDL Compiler does not notify the user

### Example 3-2 Syntax Error Notification

```
module dif_reg_in (clk, reset, a, b, y);  
  
    input    clk, reset, a, b;  
    output   y;  
  
    reg      y;  
    reg      clk;  
  
    always @ (posedge clk or posedge reset)  
        if ( reset ) begin  
            y <= 0;  
        end  
        else begin  
            y <= a & b;  
        end  
  
endmodule
```

← Error: it is illegal to code the clock as an input and as a register.  
The new HDL Compiler notifies the user.  
The current HDL Compiler does not notify the user.

### Example 3-3 Syntax Error Notification

```
module dif_ditto (clk, reset, a, b, y);  
  
    input    clk, reset, a, b;  
    output   y;  
  
    reg      y;  
    always @ (posedge clk or posedge reset)  
        if ( reset ) begin  
            y <= 0;;  
        end  
        else begin  
            y <= a & b;  
        end  
  
endmodule
```

← Error: Extra semicolon is illegal.  
The new HDL Compiler notifies the user.  
The current HDL Compiler does not notify the user

---

## Simulator Semantic Changes - Template Directive

The template directive works differently in the current HDL Compiler and the new HDL Compiler.

Under the Template directive:

- The read command does analyze in the current HDL Compiler.
- The read command does both analyze and elaborate in the new HDL Compiler.



# A

## Variables

---

The new HDL Compiler has new public variables that are in addition to the public variables in the current HDL Compiler. These variables are described in:

- “New Public Variables” on page A-2

There are also a small number of variables in the current HDL Compiler no longer supported by the new HDL Compiler. These are listed in:

- “Variables Not Supported” on page A-5

Variables not supported in the new HDL Compiler are ignored.

---

## New Public Variables

New public variable are described in Table A-1.

*Table A-1 New Public Variables*

Name	Default	Description
hdlin_allow_mixed_blocking_and_nonblocking	TRUE	Allows mixed blocking and nonblocking assignments to the same variable
hdlin_link_design	TRUE	Links designs
hdlin_auto_full_case	TRUE	Performs automatic full-case detection
hdlin_auto_parallel_case_early	TRUE	Performs automatic parallel-case detection during IL passes
hdlin_black_box_pin_hdlc_style	TRUE	For black boxes, uses HDLC pin naming style
hdlin_check_user_full_case	TRUE	Warns if full_case is applied to non-full CASE statements
hdlin_check_user_parallel_case	TRUE	Warns if parallel_case is applied to non-parallel CASE statements
hdlin_compare_const_with_gates	TRUE	Do not use synthetic operators '==' and '!=' for equality comparisons with constants.
hdlin_compare_eq_with_gates	FALSE	Do not use synthetic operators for any '==' and '!=' operators
hdlin_dont_eliminate_latches	TRUE	Do not eliminate those latches
hdlin_dyn_array_bnd_check	FALSE	Checks array index against array bounds dynamically
hdlin_escape_special_names	FALSE	Prepends a backslash onto any Verilog name with special characters
hdlin_infer_enumerated_types	TRUE	Infers enumerated types
hdlin_infer_comparators	TRUE	Infers two- and six-output comparators (value numbering-dependent)
hdlin_infer_decoders	FALSE	Infers decoders (value numbering-dependent)
hdlin_infer_function_local_latches	FALSE	Allows latches to be inferred for function- and task-scope variables
hdlin_loop_invariant_code_motion	TRUE	Hoist loop-invariant code out of loops

*Table A-1 New Public Variables*

<b>Name</b>	<b>Default</b>	<b>Description</b>
hdlin_map_to_module	TRUE	Processes map_to_module pragma for Verilog function
hdlin_no_sequential_mapping	FALSE	Don't perform sequential mapping
hdlin_one_hot_one_cold_on	TRUE	Optimize according to one_hot and one_cold attributes
hdlin_optimize_enum_types	TRUE	Simplify comparisons based on enumerated type information
hdlin_print_modfiles	TRUE	Inform user of reads and writes to module files
hdlin_report_enumerated_types	TRUE	Report inferred enumerated types
hdlin_report_mux_op	TRUE	Print summary of MUX_OPs
hdlin_report_syn_cell	FALSE	Print summary of synthetic cells
hdlin_report_tri_state	TRUE	Print report for inferred three-state elements
hdlin_ssa_transforms	TRUE	Perform transformations that are dependent on the static single assignment (SSA) representation
hdlin_translate_off_on	TRUE	Heed the translate_off/translate_on directives
hdlin_upcase_names	FALSE	Convert all Verilog names to upper case
hdlin_use_syn_shifter	FALSE	Generate synthetic shift operators
hdlin_value_numbering	TRUE	Perform value numbering (SSA dependent)
hdlin_verbose_cell_naming	FALSE	Put verbose (descriptive) names on cells
hdlin_warn_array_bound	TRUE	Warn when an array index is out of bounds
hdlin_warn_implicit_wires	TRUE	Warn about implicitly declared wires
hdlin_warn_mixed_blocking_and_nonblocking	TRUE	Warn when variable is the target of both blocking and nonblocking assignments
hdlin_warn_sens_list	TRUE	Warn on incomplete sensitivity list
hdlin_call_stack_depth	1000	Maximum depth of call stack for functions and tasks
hdlin_decoder_max_input_width	31	Only infer a decoder when the number of inputs is less than this value
hdlin_decoder_min_input_width	5	Only infer a decoder when number of inputs exceeds this value

*Table A-1 New Public Variables*

<b>Name</b>	<b>Default</b>	<b>Description</b>
hdlin_decoder_min_use_percentage	90	Percentage of outputs that must be used in order to infer a decoder
hdlin_mux_size_min	2	Minimum number of data inputs for MUX inference
hdlin_seqmap_search_depth	3	Levels of nesting to search when looking for set/reset
hdlin_while_loop_iterations	1000	Iteration limit for indeterminate FOR and WHILE loops
hdlin_work_directory	"./WORK"	Directory for storing analyzed Verilog files
hdlin_template_naming_style_variable	"%s_%p"	Naming style for templates
hdlin_template_parameter_style_variable	"%s%d"	Naming style for parameters in template
hdlin_template_separator_style_variable	"_"	Style for separator in template



---

## Variables Not Supported

Variables not supported are listed in Table A-2. These variables are ignored by the new HDL Compiler.

*Table A-2 Variables Not Supported in the New HDL Compiler*

Name	Comments
hdlin_advisor_directory	
hdlin_auto_save_templates	
hdlin_check_no_latch	
hdlin_dont_check_param_width	
hdlin_enable_analysis_info	
hdlin_enable_analysis_info_for_analyze	
hdlin_enable_vpp	
hdlin_hide_resource_line_numbers	
hdlin_keep_inv_feedback	
hdlin_latch_always_async_set_reset	
hdlin_marge_nested_conditional_statements	
hdlin_preserve_vpp_files	
hdlin_reg_report_length	
hdlin_replace_synthetic	
hdlin_translate_off_skip_test	
hdlin_vpp_temporary_directory	
hdlin_write_gtech_design_directory	

---

# B

## New Verilog Netlist Reader

---

The new Verilog structural netlist reader incorporates algorithms that reduce memory usage and CPU run time of the read command. To use the new reader, set the following hidden variable as shown:

- `enable_verilog_netlist_reader = true` (default is false)

To invoke the read command with the -netlist option, read in the file as shown:

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

