

Design Compiler Technology Backgrounder

April 2006

Introduction

Synopsys' RTL synthesis solution has been the No. 1 choice of ASIC designers worldwide since 1987. Countless numbers of chips have been designed using Synopsys' synthesis solution and over 60 semiconductor and library vendors offer hundreds of libraries supporting Synopsys synthesis. Synopsys' relentless focus on the needs of the IC designer has yielded a complete synthesis solution that addresses all of today's design challenges. At the core of Synopsys' RTL synthesis solution is Design Compiler™ (DC), which synthesizes your high-level design description, written in Verilog, SystemVerilog or VHDL language, into optimized gate-level designs. Design Compiler supports a wide range of flat and hierarchical design styles and can optimize both synchronous and asynchronous designs for timing, test, power and area. With Design Compiler you can:

- Generate fast, area-efficient ASIC designs by employing user-specified standard cell
- Explore design tradeoffs involving design constraints such as timing, area, and power under various loading, temperature, and voltage conditions
- Synthesize and optimize finite state machines, synchronous and asynchronous designs
- Manage complexity by creating and partitioning hierarchical designs automatically, optimize designs faster and utilize higher capacity with Automated Chip Synthesis
- Benefit from the easy-to-use, customizable UNIX style user interface, TCL, and intuitive visual interface
- Enjoy support for industry standard languages
- Design using hundreds of libraries offered by over 60 semiconductor and library vendors

Basic design implementation flow (Figure 1) involves defining your design goals, selecting compilation strategy, optimizing the design, analyzing results, driving place and route to achieve timing closure.

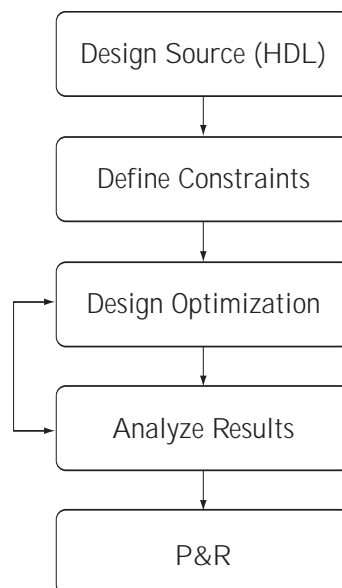


Figure 1: Basic design implementation flow

Following are some of the key features available in all Design Compiler products:

Hierarchical Compile

- Compile strategy
- Top level only compilation
- Incremental compile

Design Optimization

- High-level optimization
 - Arithmetic expressions optimization
- Full compile
 - Flattening and structuring
 - Mapping:
 - Delay optimization
 - Design rule fixing
 - Area specific optimization
- Sequential optimization for complex flip-flops and latches
- Finite State Machine (FSM) optimization
- Time borrowing for latch-based designs

Timing Analysis Features

- Timing exceptions
- Case analysis to set constant paths
- Incremental timing update
- Support for synchronous and asynchronous designs

Intuitive User Interface/Ease-of-Use

- Report generation
- Command log files
- Scripting

Budgeting

Automated Chip Synthesis

Hierarchical Compile

Compile Strategy

DC allows two different compile strategies to process your design in the most appropriate ways, top down or bottom up. In addition, you have flexibility in choosing different compile strategies for your sub blocks. In the top-down approach, the top-level design and all its sub-designs are compiled together. All environment and constraint settings are defined with respect to the top-level design. Therefore, this strategy automatically takes care of interblock dependencies. But for the larger designs or for the designs where sub-designs are done by different teams, individual sub-designs can be constrained and compiled separately. After successful compilation, these sub-designs can be assembled to compose the designs of the next higher level of the hierarchy and designs at the higher level are compiled. This compilation process is continued up through the hierarchy until the top-level design is synthesized. You can use either strategy to process the entire design, or you can mix strategies, using the most appropriate strategy for each sub-design.

Top Level Only Compilation

Design Compiler allows you to compile the top level design only to fix constraint violations occurring at the top level after the sub-blocks in a design are assembled. These violations might occur due to changes in the environment around the sub-blocks as a result of the optimizations that have been performed in the sub-blocks. You can compile the sub-design blocks separately and use this feature to fix violations at the top level nets. Any design rule violations present in the design gets fixed also regardless of where the violation occurs.

Incremental Compile

You can improve your design's performance by using the incremental compile option. Incremental compile performs gate-level optimization only ensuring the resulting design's performance is the same or better than the original design's. An incremental compilation improves the existing design cost by focusing on the areas of the design that do not meet constraints. The existing structure is preserved if all constraints are already met. Incremental mapping uses the existing gates from an earlier compilation as a starting point for the mapping process. Mapping optimizations are accepted only if they improve the circuit speed or area. Incremental mapping guarantees that a circuit can only be improved.

Design Optimization

High Level Optimization

Design Compiler automatically selects the best structure for your design based on your high level architecture. You can create more efficient synthesized circuits by providing additional architecture information. One of the techniques that Design Compiler employs is Arithmetic Expression Optimization.

Arithmetic Expressions Optimization

Design Compiler uses the properties of arithmetic operators to rearrange an expression so that it results in an optimized implementation. For example, Design Compiler can recognize and optimize the design based on the associative and commutative properties of simple operator, an addition (+). You can also use arithmetic properties such as parenthesis to control the choice of implementation for an expression. Arithmetic optimization can be performed using one of the following:

- Merging cascaded adders with a carry

If your design has two cascaded adders and one has a bit input, Design Compiler replaces the two adders with a simple adder that has a carry input. Figure 2 shows two expressions in which cin is a bit variable connected to a carry input. Each expression results in the same implementation.

```
z <= a + b + cin;  
or  
t <= a + b;  
z <= t + cin;
```

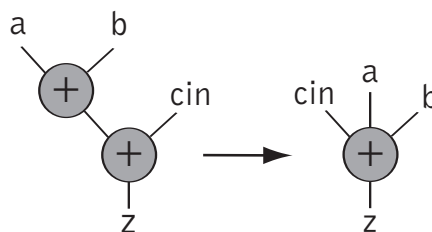


Figure 2. Cascaded adders with carry input

- Arranging expression trees for minimum delay

Arithmetic optimization can minimize the delay through an expression tree by rearranging the sequence of the operations. Consider the following statement:

$Z \leq A + B + C + D;$

The parser performs each addition in order, as though parentheses were placed as shown, and constructs the following expression tree shown in Figure 3.

$Z \leq ((A + B) + C) + D;$

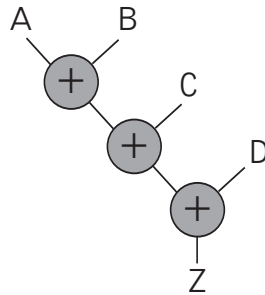


Figure 3. Default expression tree

- Considering signal arrival times

Design Compiler considers the arrival time of each signal in the expression to minimize the delay through an expression tree. For example, if the arrival times of each signal are the same, the length of the critical path of the expression:

$Z \leq A + B + C + D;$

will be equal to three adder delays. The critical path delay can be reduced to two adder delays if you add parentheses to the first statement as:

$Z \leq (A + B) + (C + D);$

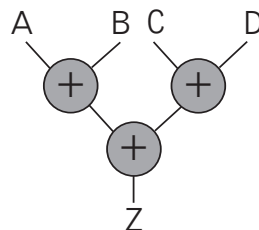


Figure 4. Balanced adder tree

The parser evaluates the expressions in parentheses first and constructs a balanced adder tree in Figure 4.

- Resource sharing

Resource sharing reduces the amount of hardware needed to implement operators such as addition (+) in your Verilog or VHDL description. Resource sharing assigns similar operations to a common netlist cell. Netlist cells are the resources—they are equivalent to built hardware. Without resource sharing, each Verilog or VHDL operation is built with separate circuitry. For example, every + with non-computable operands causes a new adder to be built. This repetition of hardware increases the area of a design. In contrast, with resource sharing, several + 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 subtractor to further reduce a design's circuit area. Design Compiler offers automatic resource sharing.

In automatic resource sharing, Design Compiler identifies the operations that can be shared and uses this information to minimize the area of your design, taking your constraints into consideration. You have an option of overriding the automatically determined sharing by using automatic sharing with manual controls or manual sharing. Automatic resource sharing is the simplest way to share components and reduce the design area especially if you do not know how you want to map the operations in your design onto hardware resources.

Full Compile

Optimize your RTL design performing both technology-independent optimization as well as technology specific optimization with full compilation. During full compilation process, Design Compiler removes the existing gate structure from a design, and then rebuilds the design with the goal to improve the design's logic structure. Design Compiler strives to reduce the number of product terms. As a first-order approximation, decreasing the number of product terms relates to both area and delay reduction. The optimization of logic equations does not affect a particular part of a function; rather it has a global effect on the overall area or speed characteristics of a design. You can select the logic optimization strategy based on your design goals. Logic optimization has two components: flattening and structuring.

Flattening is an optional logic optimization step that removes all intermediate variables and uses Boolean distributive laws to remove all parentheses. Thus, flattening removes all logic structure from a design. Removing poor intermediate variables enables Design Compiler to choose more-efficient sub-functions. You can also obtain a two-level (sum of products) equation representation of a design using flattening only, with no structuring or mapping. The result of flattening is a two-level, sum-of-products form. Flattening can result in a faster design, because the two-level form generally has fewer levels of logic between inputs and outputs. It is not recommended to flatten regular or highly structured designs, such as adders and ALUs designed with an explicit structure.

Structuring is a logic optimization step that adds intermediate variables and logic structure to a design. During structuring, Design Compiler searches for sub-functions that can be factored out, and then evaluates these factors based on the size of the factor and the number of times the factor appears in the design. The sub-functions that most reduce the logic are turned into intermediate variables and factored out of the design equations. Design Compiler structures a design during compilation by default. Structuring provides a powerful way to improve the logic structure of your designs.

Design Compiler offers timing-driven structuring to minimize delays and Boolean structuring to reduce area. Timing-driven structuring considers a design's timing constraints during local and global structuring and improves critical paths as necessary. Timing-driven structuring is on by default. Timing-driven structuring occurs as part of the structuring step during compilation. When you use timing-driven structuring, you must define accurate timing and clock constraints. You can use Boolean optimization for reducing the area. Boolean optimization structuring uses Boolean algebra to capture don't care information and reduce circuit area. Boolean optimization is executed as part of the structuring step during compilation.

Mapping the technology-specific optimization synthesizes optimized circuit structure to a gate-level netlist in target technology. The mapping optimization involves:

- Delay optimization

During the delay optimization phase, Design Compiler creates optimal netlist representation in the target technology library that meets your timing goals. DC takes design rules into account for optimizing the delay performance of critical paths in your design. When two circuit solutions offer the same delay performance, Design Compiler implements the solution that has the lower design rule cost.

- Design rule fixing

In the initial design rule fixing, Design Compiler tries to fix whatever design rule violations it can—without increasing delay cost. Whenever possible, Design Compiler fixes design rule violations by resizing gates across multiple logic levels—as opposed to adding buffers to the circuitry.

If there are still design rule violations after the initial design phase, Design Compiler offers the flexibility to fix design rule violations even if doing so increases the delay cost. Design rules carry a higher priority than delay in the second phase of design rule fixing, which you may skip.

- Area optimization

Assuming that you have placed area constraints on your design, Design Compiler now attempts to minimize the number of gates in the design. You can direct Design Compiler to put a low, medium, or high effort into area optimization. If you do not place area constraints on your design, Design Compiler performs a limited series of downsizing and area cleanup steps.

Low Effort

Design Compiler does gate sizing and buffer and inverter cleanup. Design Compiler allocates limited CPU time to this effort level.

Medium Effort

Design Compiler adds phase assignment to gate sizing and buffer and inverter cleanup. Design Compiler allocates more CPU time to this effort than to a low-effort optimization.

High Effort

Design Compiler performs additional gate minimization strategies and increases number of iterations. The tool adds gate composition to the process and allocates even more CPU time.

Sequential Optimization

Design Compiler allows you to optimize sequential cells in two phases, initial and final sequential optimization. During the first phase of initial sequential optimization, information about the delay through the combinational logic is incomplete and Design Compiler does not have enough information to select the optimum sequential cell. You can map to either standard sequential cells or scan-equivalent cells in the library. Initial sequential optimization is in the first phase of gate-level optimization. Design Compiler optimizes the sequential cells, defining the following information:

- Locations of the islands of combinational logic between sequential cells
- Timing constraints on the combinational islands required to meet the setup and hold constraints on the sequential cells

Once Design Compiler has accurate values for all delays through the combinational logic the final sequential optimization phase occurs. Design Compiler optimizes timing-critical sequential cells (cells on the critical path). The tool examines each sequential cell and its surrounding combinational logic to determine if they might be replaced by more complex sequential cells from the target library. Final sequential optimization can:

- Improve design timing by choosing higher-performance sequential cells.
- Possibly reduce the design's area and delay if complex sequential cells exist in the library. The tool incorporates the combinational logic in front of the sequential cell into the sequential cell itself.
- Improve area by trying to re-map the sequential elements to recover area.

Finite State Machine Optimization

You can represent, extract, and optimize finite state machines (FSM) using DC. An FSM is a special representation of a sequential design as shown in Figure 5. Design Compiler optimizes the state encoding of each FSM, then converts it to Boolean equations and technology-independent flip-flops. Design Compiler supports unique optimizations for finite state machines.

You can read in a design with FSM represented as state table or instruct DC to extract FSM from the design. Optionally, you can specify FSM attributes (state names and encodings, encoding generation style, and state vector flip-flops). Once you have specified circuit-level constraints and attributes for the design, you are ready to optimize designs with FSM using compile.

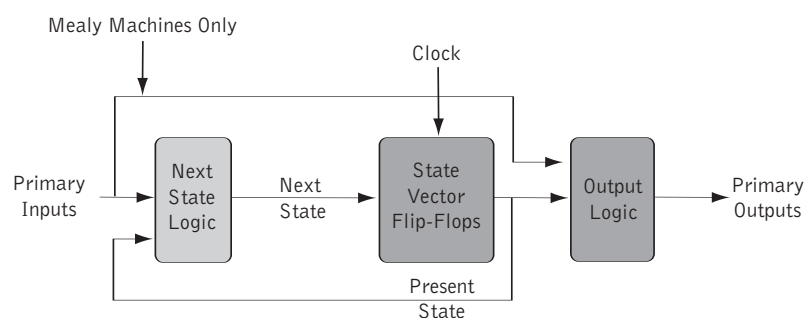


Figure 5. Finite State Machine architecture

Time Borrowing in Latch-based Designs

You can leverage Design Compiler's unique technique, time borrowing, to optimize near-critical paths and reduce delay costs in latch-based designs. A latch is a simple, 1-bit level sensitive memory device. Design Compiler allows borrowing time from next clock cycle to extend the time during which latch is enabled if the path leading to the data pin of a latch is too long. For example, in the following two-stage latch based design, the combinational logic block between Latch1 and Latch2 may have more delay than the delay between Latch2 and Latch3. To resolve this discrepancy, the first stage can borrow time from the second clock cycle. In this event, the second clock cycle is left with less time to accommodate the combinational logic block between Latch2 and Latch3.

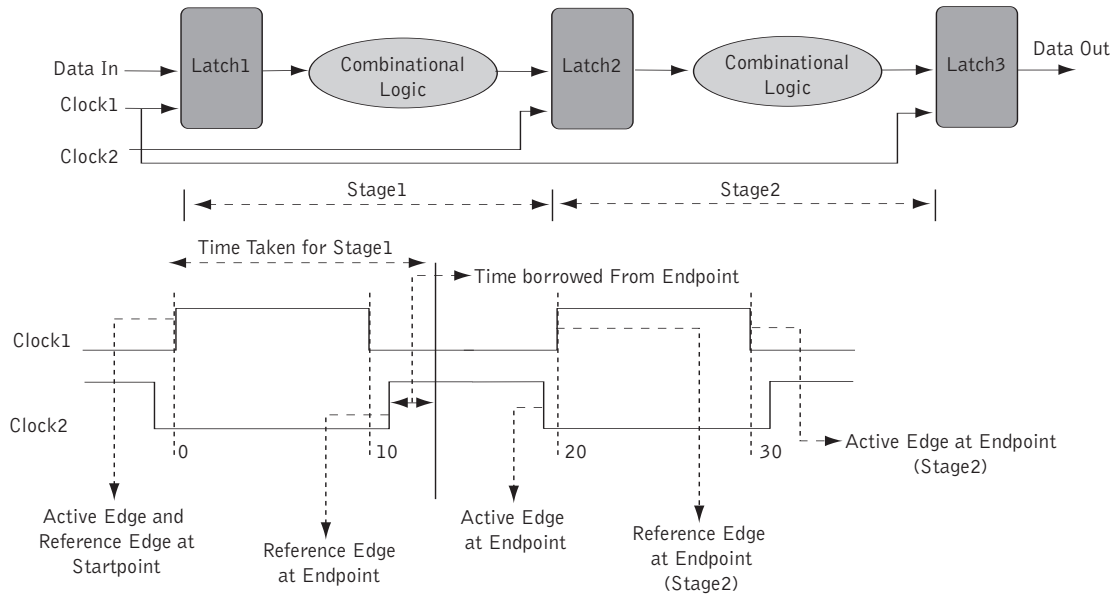


Figure 6. General structure of a latch-based design

Timing Analysis

The purpose of timing analysis is to understand the performance and ensure design meets the design goals after synthesis. You can identify the problem areas and bottlenecks in the design at this stage and plan next steps such as re-synthesize blocks based on problem areas. The design goals are specified using design constraints. Constraints are measurable circuit characteristics such as timing, area, and capacitance. You can drive the synthesis process by defining the following types of constraints:

- Design rule constraints: implicit constraints defined in technology library.
- Optimization constraints: explicit constraints; you define them during the duration of synthesis run

Design Compiler tries to meet both design rule constraints and optimization constraints, but design rule constraints take precedence. You can specify constraints interactively on the command line or in a constraints file.

Timing Exceptions

Design Compiler automatically determines the maximum and minimum path delay requirements for the design. It examines the clock waveforms at each timing path startpoint and endpoint. A timing path is a path through logic along which signals can propagate. Timing paths normally start at primary inputs or clock pins of registers and end at primary outputs or data pins of registers. You can drive the synthesis using path-based commands that operate on timing paths or endpoints. You have the flexibility to override the default behavior by using commands called timing exceptions such as the following:

- `set_false_path`
- `set_max_delay`
- `set_min_delay`
- `set_multicycle_path`
- `reset_path`

Timing exception commands operate on a "from" set, a "to" set, and several "through" sets. The -from option indicates a path startpoint, -to indicates a path endpoint, and-through enables control over a specific path or multiple paths between a given startpoint and endpoint.

False Paths

You can set a path to be a false path indicating that the particular timing path does not propagate a signal. Design Compiler removes the timing constraints from such paths and does not optimize them.

Multi-cycle Paths

Design Compiler by default makes single clock cycle timing a requirement for all the paths. Design Compiler automatically infers single-cycle timing from clock waveforms and from input delay and output delay information. Single-cycle timing means that data should reach from start point to end point in a single clock cycle.

You can change the default behavior for multi-cycle paths. A multi-cycle path is a timing path that is not expected to propagate a signal in one cycle. Multi-cycle paths are exceptions to the default single-cycle timing. You can set multi-cycle paths to direct Design Compiler to allow multiple clock cycles for data to propagate along a path. You can always reset paths to single-cycle timing.

Case Analysis to Set Constant Paths

Design Compiler allows you to define a path as constant path so that it can ignore the path during timing analysis and not over constraint the design knowing the path will remain constant during the operation of the design. These paths remain functional in the optimized design. The case analysis constant can be assigned to a port or pin and can take a value of 0 or 1. Design Compiler automatically propagates forward each case analysis constant from the specified port or pin to establish a constant path if either of the following two conditions is met:

- the constant value is a controlling value
- the constant value is not a controlling value, but all other inputs to the gate are constant

Case analysis constants can propagate through hierarchy. Constant path propagation ends at cells where a non-constant output can occur.

Incremental Timing Update

If you decide to change your timing constraints after the design has already been optimized, DC offers flexibility to re-time only the paths in the design that are affected by the changes in constraints. Timing updates are not performed on paths that are not affected by the changes in constraints. This enables you to assess your performance goal.

Support for Synchronous and Asynchronous Designs

Synopsys synthesis tools support both synchronous and asynchronous designs. You can define the constraints on both synchronous and asynchronous paths. Synchronous paths are constrained by clocks while asynchronous paths can be constrained by specifying minimum and maximum delays. In addition, you can synthesize designs that have multiple clocks with different cycle times. Design Compiler determines all the edge dependencies in such cases and analyzes the design to ensure all the paths meet most restrictive timing constraints. You have the flexibility to override the behavior. Design Compiler also applies a unique technique called time borrowing to designs containing level sensitive latches. You can analyze latch-based designs to further optimize your designs.

Intuitive User Interface/Ease-Of-Use

Design Compiler offers three user interfaces for reporting, scripting and analyzing the design:

- Design Compiler command language (dcsh)
- Tool Command Language (TCL User Interface (GUI))

Both dcsh and TCL modes are easy to use and very similar to UNIX operating shell including setting variables, conditional execution of commands, and control flow commands.

Report Generation

After optimizing a design, you can generate reports and schematics to analyze your timing, area, and component selection results. The timing analyzer determines the actual path delays of the design and compares them with the required path delays. The timing analyzer computes each gate and interconnect delay, then traces critical paths, calculating minimum and maximum arrival times to points of interest. The timing analyzer uses the critical path values to evaluate design constraints and create timing reports. This helps you identify the constraint violations in the design that may require special compile techniques to meet the design goals.

Report commands display information about the current design. Most reports can also be run on the current instance. In addition to its unique command language, Design Compiler supports industry standard TCL language for report generation commands. Design Compiler builds an internal database of the netlist and the attributes it applied to the database. This database consists of several classes of objects, such as designs, libraries, ports, cells, nets, pins, clocks, and so on. Most Design Compiler commands operate on these objects. A collection is a group of objects referred to by a string identifier known as a collection handle. You can create collections of objects, then apply a set of commands to interact with those collections. Collections can be homogeneous (contain objects of one type) or heterogeneous (contain objects of many types). The TCL-mode in Design Compiler also supports a set of commands that allows you to create a collection. You can use this collection to analyze your design.

With Design Compiler you can customize report generation. You can create a collection of paths and assign a variable to this collection to be used in other commands. You can then iterate commands through these paths.

Command Log Files

Design Compiler records commands including setup file commands and variable assignments into a command log during a Design Compiler session. You can use the command log file to produce a script for a particular synthesis strategy, to record the design exploration process or document any problems during the synthesis.

Scripting

You can create a command script file by placing a sequence of dc commands in a text file. All the DC command can be executed within a script file. You can develop the script in DC command language (dcsh) or in TCL.

Budgeting

As designs become larger and larger, the compile runtimes becomes long. To solve this runtime problem, you can use a divide-and-conquer strategy. Such a strategy divides the design into several blocks, compiles each block in parallel and then reassembles the blocks. After dividing the design into several smaller blocks, you need a new environment file and an accurate constraint file for each block.

Design budgeting is the process that automatically generates the constraint file (including the timing requirements and compile environment) for each block from the top-level constraints and environment. Using the budgeter, you can do full interactive budgeting from within Design Compiler. Budgeting is an alternative methodology to manual constraining and the characterize compile strategy. Generating constraints manually is laborious and can lead to over-constraining, under-constraining, or missing constraints entirely. The characterize compile solution requires that you characterize a block, compile it, characterize another block, compile it, and so forth leading to long turn around times.

You can specify the design hierarchy together with chip-level timing constraints, and use the budgeter to allocate timing and environment constraints across lower-level blocks. The allocated budgets can be used to drive the synthesis of individual blocks. The budget generation and allocation process is iterative: You use the RTL budgeting flow to allocate initial budgets, then use the initial budgets to synthesize lower-level blocks. You can apply the synthesis results to the budgeter to derive more accurate budgets by using the gate-level or refinement budgeting flow, repeating this process until design goals are achieved. Budgeting addresses capacity and runtime limitations:

- Offers a divide-and-conquer synthesis methodology in which you divide the entire design into smaller synthesis runs leading to higher capacity
- Reduces runtime because you can compile all the sub-blocks in parallel after the allocated constraints are generated leading to faster synthesis

Automated Chip Synthesis

As designs become larger and larger, the compile runtimes might become impractical. You can reduce your top-down compile runtime by compiling portions of your design in parallel. This divide-and-conquer technique although reduces runtime, but it increases the data management requirements. The Automated Chip Synthesis tool addresses these challenges by providing a higher level of abstraction for synthesis; including compile strategy implementation and automatic generation of compile scripts and makefiles. Automated Chip Synthesis is a set of Design Compiler commands that provide a data management environment, automates the divide-and-conquer synthesis flows, and enables parallel compilation of the hierarchical design.

SYNOPSYS®

700 East Middlefield Road, Mountain View, CA 94043 T +1 650 584 5000 www.synopsys.com

Synopsys, and the Synopsys logo are registered trademarks and Design Compiler is a trademark of Synopsys, Inc.
All other products or service names mentioned herein are trademarks of their respective holders and should be treated as such.

All rights reserved. Printed in the U.S.A.
©2006 Synopsys, Inc. 4/06.PS WO.06-14263