Design Compiler® Register Retiming Reference Manual

Version F-2011.09-SP2, December 2011

SYNOPSYS®

Contents

Preface

This preface includes the following sections:

What's New in This Release

Information about new features, enhancements, and changes, along with known problems and limitations and resolved Synopsys Technical Action Requests (STARs), is available in the *Design Compiler Release Notes* in SolvNet.

To see the *Design Compiler Release Notes*,

1. Go to the Download Center on SolvNet located at the following address:

   https://solvnet.synopsys.com/DownloadCenter

   If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.
2. Select Design Compiler, and then select a release in the list that appears.

About This Manual

The *Design Compiler Register Retiming Reference Manual* describes the concepts of register retiming and shows you how to use this capability to retime circuits. Register retiming performs a sequential optimization that moves registers to optimize timing and area. It optimizes gate-level netlists to meet timing while trying to use as few registers as possible.

This manual supports the Synopsys synthesis tools, whether they are running under the UNIX operating system or the Linux operating system. The main text of this manual describes UNIX operation.

Audience

This manual is intended for logic designers and engineers who use the Synopsys synthesis tools to design ASICs, ICs, and FPGAs. Knowledge of high level techniques, a hardware description language, such as VHDL or Verilog is required. A working knowledge of UNIX is assumed.

Related Publications

For additional information about *Using Tcl With Synopsys Tools*, see the Design Compiler and IC Compiler documentation on SolvNet at the following address:

https://solvnet.synopsys.com/DocsOnWeb

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

- Design Vision
- DesignWare components
- DFT Compiler
- PrimeTime
- Power Compiler
- HDL Compiler
- IC Compiler

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates syntax, such as `write_file`.
`Courier italic`	Indicates a user-defined value in syntax, such as `write_file` *`design_list`*.
`Courier bold`	Indicates user input—text you type verbatim—in examples, such as `prompt>` **`write_file top`**
[ ]	Denotes optional arguments in syntax, such as `write_file [-format` *`fmt`*`]`
...	Indicates that arguments can be repeated as many times as needed, such as *`pin1 pin2 ... pinN`*
|	Indicates a choice among alternatives, such as `low | medium | high`
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

SolvNet includes a knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation, and technical support.

To access SolvNet, go to the following address:

https://solvnet.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.

If you need help using SolvNet, click HELP in the top-right menu bar.

Contacting the Synopsys Technical Support Center

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

- Open a support case to your local support center online by signing in to SolvNet at https://solvnet.synopsys.com, clicking Support, and then clicking “Open A Support Case.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at http://www.synopsys.com/Support/GlobalSupportCenters/Pages
- Telephone your local support center.
  - Call (800) 245-8005 from within North America.
  - Find other local support center telephone numbers at http://www.synopsys.com/Support/GlobalSupportCenters/Pages

1

Introduction to Register Retiming

In this document, the term “register” refers to both edge-triggered registers and level-sensitive latches unless stated otherwise. Both types of sequential cells can be retimed. However, the `balance_registers` command can only be applied to edge-triggered registers.

This chapter contains the following sections:

Understanding Register Retiming

Register retiming is a sequential optimization technique that moves registers through the combinational logic gates of a design to optimize timing and area. Other optimization techniques, such as those implemented in the `compile_ultra` command or `compile` command, optimize the combinational logic by performing Boolean optimization and mapping to cells in the technology library. These techniques leave unchanged the location and number of any registers present in the design. Register retiming adds an opportunity for improving circuit timing.

When you describe circuits at the RT level before logic synthesis, it is usually very difficult and time consuming, if not impossible, to find the optimal register locations and code them into the HDL description. With register retiming, the locations of the flip-flops in a sequential design can be automatically adjusted to equalize as nearly as possible the delays of the stages. This capability is particularly useful when some stages of a design exceed the timing goal while other stages fall short. If no path exceeds the timing goal, register retiming can be used to reduce the number of flip-flops, where possible.

Purely combinational designs can also be retimed by introducing pipelining into the design. In this case, you first specify the desired number of pipeline stages and the preferred flip-flop from the target library. The appropriate number of registers are added at the outputs of the design. Then the registers are moved through the combinational logic to retime the design for optimal clock period and area.

Register retiming leaves the behavior of the circuit at the primary inputs and primary outputs unchanged (unless you choose special options that do not preserve the reset state of the design or add pipeline stages). Therefore you do not need to change any simulation test benches developed for the original RTL design.

Retiming does, however, change the location, contents, and names of registers in the design. A verification strategy that uses internal register inputs and outputs as reference points will no longer work. Retiming can also change the function of hierarchical cells inside a design and add clock, clear, set, and enable pins to the interfaces of the hierarchical cells.

A Register Retiming Example

During retiming, registers are moved forward or backward through the combinational logic of a design. Figure 1-1 and Figure 1-2 illustrate an example of delay reduction through backward retiming of a register.

Figure 1-1 Circuit Before Retiming

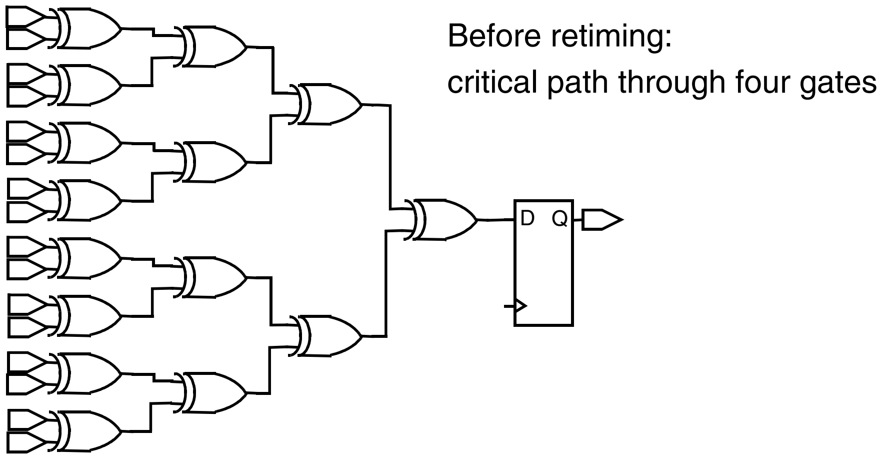
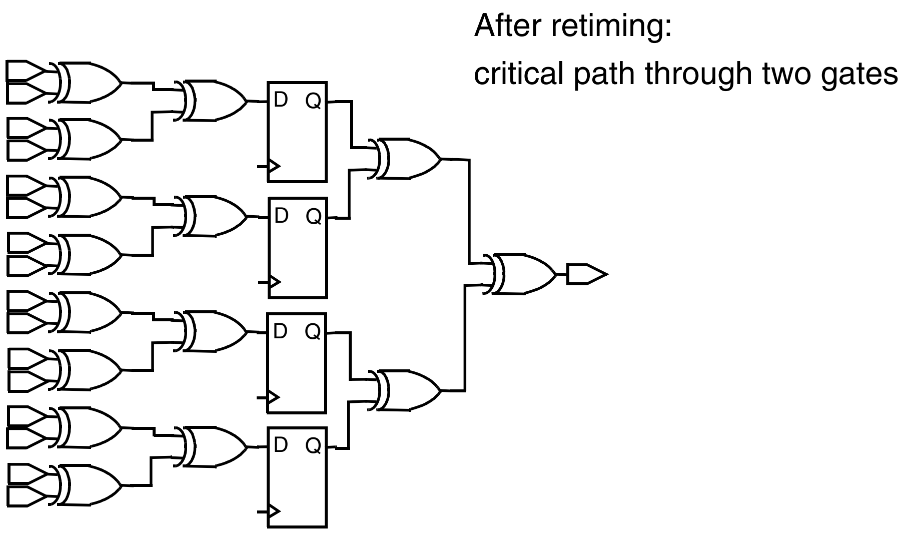


Figure 1-2 Circuit After Retiming

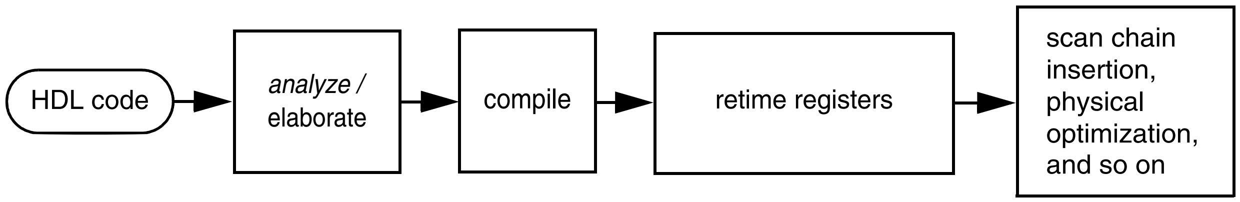


In this example, before register retiming there are four levels of combinational logic and only one register at the endpoint of the critical path. After retiming, the register, which has been replaced by four registers, has been moved back through two levels of logic, and the critical path now consists of two stages. The critical path delay in each stage is less than the critical path delay in the initial single stage design. As in this example, delay reduction through retiming often leads to an increase in the number of registers in the design, but usually this increase is small.

Design Flow Using Register Retiming

You optimize registers after you have compiled the design; that is, register retiming is performed on mapped netlists. Figure 1-3 shows the position at which register retiming is used in a typical design flow.

Figure 1-3 Design Flow With Register Retiming



As part of the register retiming functionality, an incremental compile is usually carried out automatically after retiming. Note, however, that there are three register retiming commands (see "Register Retiming Commands" on page 1-4), two of which include the incremental compile capability and one that does not. You can prevent the automatic incremental compile in the two commands by specifying the appropriate option.

Register Retiming Commands

Register retiming consists of three major retiming commands and a number of support commands for setting retiming-related attributes. The principal commands are

- `optimize_registers`
- `balance_registers`

The `optimize_registers` command offers the most convenient way to perform all retiming tasks for sequential designs. In general, this command optimizes both synchronous and asynchronous registers with respect to timing and area (minimum number of registers). An incremental compile is automatically done after the retiming, unless you specify otherwise. The command supports the retiming of level-sensitive latches and also includes analysis options.

The `balance_registers` command moves the existing registers of a sequential design to minimize the clock period. This command does not minimize the number of registers or retime asynchronous registers. Also, no incremental compile is carried out, and no analysis options are available.

Note:
Typically you apply these commands globally to a design. However, you can assign retiming attributes to individual cells. For more information, see "Setting Retiming Attributes on Individual Cells" on page A-2.

The optimize_registers Command

The `optimize_registers` command performs the following principal actions on a sequential design:

1. Minimizes the clock period

   The registers are moved to ensure the smallest clock period for the retimed circuit.

2. Minimizes the register count

   The minimum clock period determined in the first step is compared to a user-defined clock period target. If the minimum clock period is smaller than or equal to the target clock period, the target clock period is used and a register distribution is computed that accommodates the target clock period with the smallest number of registers possible. If the target clock period is smaller than the minimum clock period, the number of registers is minimized for the minimum clock period.

3. Executes an incremental optimization of the combinational logic

   Because of the new distribution of registers, the loads for cells and the location of the critical paths change. An incremental logic optimization step identical to the `compile -incremental` command optimizes the combinational logic to reflect these changes.

The balance_registers Command

The `balance_registers` command performs the following principal actions on a sequential design:

1. Minimizes the clock period

   The registers are moved to ensure the smallest clock period for the retimed circuit and to balance the pipeline stage delays.

2. Performs a sequential mapping after moving the registers

2

Register Retiming Concepts

In this document, the term “register” refers to both edge-triggered registers and level-sensitive latches unless stated otherwise. Both types of sequential cells can be retimed. However, the `balance_registers` command can only be applied to edge-triggered registers.

This chapter discusses fundamental register retiming concepts. A good understanding of these concepts will help you make the best use of the Design Compiler register retiming capabilities.

This chapter contains the following sections:

- Basic Definitions and Concepts
- Forward Retiming
- Backward Retiming
- Asynchronous Control Inputs of Registers
- Synchronous Control Inputs of Registers
- Multiclass Retiming
- Pipeline and Nonpipeline Circuits
- Reset State Justification

Basic Definitions and Concepts

To understand how register retiming works, you need first to understand certain basic definitions and concepts. In particular, you must understand what sequential generic elements (SEQGENs), control nets, and register classes are. These are important because during retiming, mapped registers are temporarily replaced by SEQGENs according to their classifications as determined by their control nets.

Note:
In this manual, the terms design, cell, leaf cell, hierarchical cell, combinational cell, and sequential cell are used in the same sense as in other Synopsys manuals.

Flip-Flops and Registers

Flip-flop, register, synchronous register, asynchronous register, and latch are familiar terms; however, with respect to register retiming as discussed in this manual, these terms have the following specialized usage:

- A *flip-flop* is an element of a technology library (target library) that has, unlike the combinational cells, a state and a distinguished clock input. Flip-flops can be edge triggered or level sensitive.
- A *register* is technically an instance of an edge-triggered flip-flop in the design. But for the purposes of this manual, because both edge-triggered flip-flops and level-sensitive latches can be retimed, the term *register* refers to both types of sequential devices unless stated otherwise.
- A *synchronous register* is a register that can change its state only at the active edge of the clock signal.
- An *asynchronous register* is a register that, in addition to changing its state on a clock edge, can also change its state according to the control levels of asynchronous signals, which are independent of its clock signal.
- A *latch* is an instance of a level-sensitive flip-flop in the design. Register retiming supports designs with latches and retimes them instead of the registers if the `-latch` option is used.

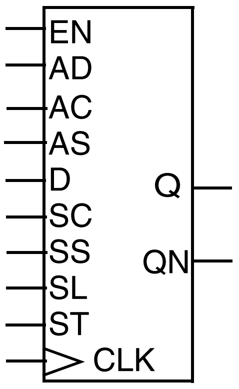
The essential point here to understand is that flip-flops are technology library elements, while registers and latches are their design instances.

SEQGENs

A SEQGEN is a generic sequential element that is used by Synopsys tools to represent registers and latches in a design. SEQGENs are created during elaboration and are usually mapped to flip-flops during compilation. Because mapped flip-flops are temporarily replaced by SEQGENs during register retiming, it is important that you understand the basic functionality of these elements.

Figure 2-1 shows the pins that are used when the SEQGEN cell describes a synchronous or asynchronous register. (Additional pins occur when the SEQGEN is operated as a latch.)

Figure 2-1 Relevant Pins of the SEQGEN Cell



EN	: enable (clock if operated as latch)
AD	: asynch_data (data if operated as latch)
AC	: asynch_clear
AS	: asynch_set
D	: next_state
SC	: synch_clear
SS	: synch_set
SL	: synch_load
ST	: synch_toggle
CLK	: clocked_on

In register retiming, the operation of SEQGEN cells is as follows:

- The synchronous toggle pin (ST) has an inactive value of 0. Therefore, the SEQGEN cell is retimed only if the ST pin is connected to a constant net with value 0.
- The clock (CLK) pin is always connected to the clock net of the design.
- The synchronous state changes occur at the rising edge of the clock signal.
- If set to 1, the synchronous load (SL) pin enables the next-state D input to become the next state. The SL pin should be tied to a constant 1 net when unused.
- The synchronous clear (SC) pin sets the state to 0 if active. This pin preempts the SL input and, to be inactive, must be tied to a constant 0 net.
- The synchronous set (SS) pin sets the state to 1 if active. This pin preempts the SL input and, to be inactive, must be tied to a constant 0 net.
- If both SC and SS are active, the constant set as an attribute on the particular SEQGEN instance becomes the new state.

- The asynchronous inputs AC and AS override all settings of the synchronous inputs; these pins change the state and output of a SEQGEN instance, independent of the clock input.
- The AC input sets the Q output to 0 if active; the AS input sets the Q output to 1 if active. Both inputs must be 0 to be inactive.
- The EN input replaces the CLK clock pin if the SEQGEN cell operates as a level-sensitive latch.
- The AD pin input replaces the D pin input if the SEQGEN cell operates as a level-sensitive latch.

Control Nets

A control net is a net connected to one of the SL, SC, SS, AC, or AS pins of a SEQGEN instance. The equivalence of control nets plays a crucial role in the movement of registers during retiming.

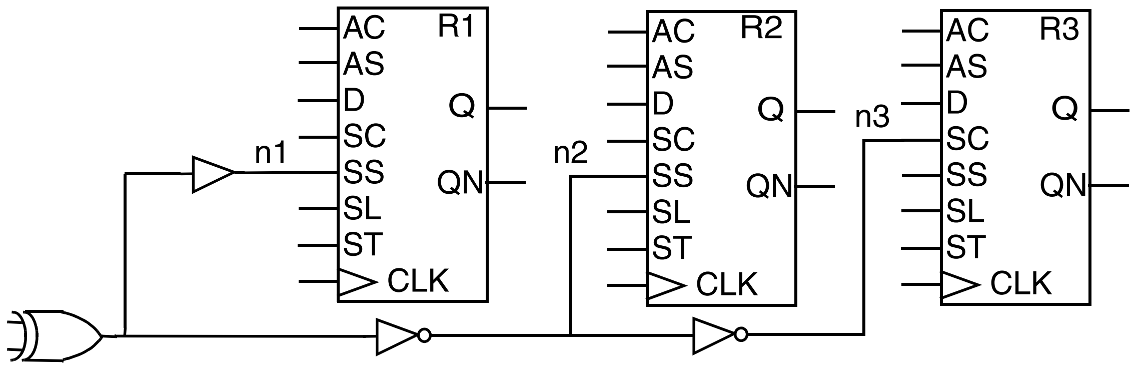
By definition, a set of control nets are equivalent if they meet either of the following conditions:

- All the nets can be reached from a common source, and between this common source and the SEQGEN instances, all the nets have an odd number of inverters or all the nets have an even number of inverters. (A net with no inverters is regarded as having an even number of inverters.)
- The control nets are constant and have the same constant values (0 or 1).

Thus, two nonconstant nets with a common source, one that includes an odd number of inverters and the other an even number of inverters, are not equivalent nets. Note that any number of buffers is allowed between the common source and the SEQGEN pin of an equivalent net.

Figure 2-2 shows an example of equivalent and nonequivalent nets. Nets n1 and n3, which have even number of inverters (0 and 2), are equivalent, while net n2, which has an odd number of inverters (1), is not equivalent to either of them.

Figure 2-2 Equivalent Control Net Example



Register Classes

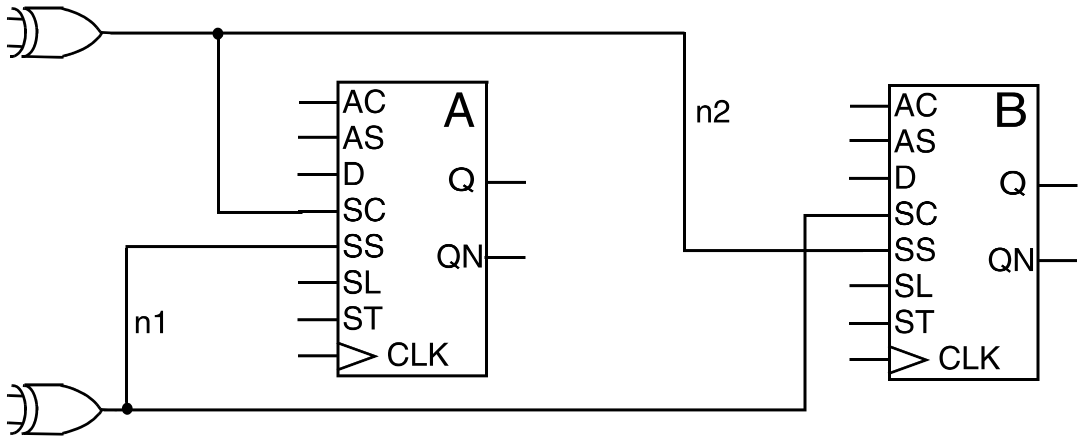
The SEQGEN instances are grouped into register classes according to their connections to control nets. Grouping is necessary because only registers belonging to the same group can be moved together across a combinational gate without violating the circuit logic.

Two SEQGEN instances belong to the same register class if the following conditions are fulfilled:

- Their SL pins are connected to equivalent control nets.
- Their SC and SS pins are connected to equivalent control nets. That is, a given control net can be connected to the corresponding SC or SS pins in the SEQGEN instances or to the SC pin of one SEQGEN instance and to the SS pin of the other.
- Their AC and AS pins are connected to equivalent control nets. The same conditions hold for the asynchronous pins as for the synchronous pins.

Figure 2-3 shows an example in which registers A and B belong to the same class.

Figure 2-3 Swapping of Control Net for Registers in the Same Class



In Figure 2-2 on page 2-5, if all pins not connected to nets n1, n2, and n3 are connected to their inactive constants, registers R1 and R3 belong to the same class, but register R2 belongs to a different class.

When registers belonging to the same class are moved, it is possible to swap their control nets as needed to accomplish the retiming. This swapping capability is true for both synchronous and asynchronous register pins.

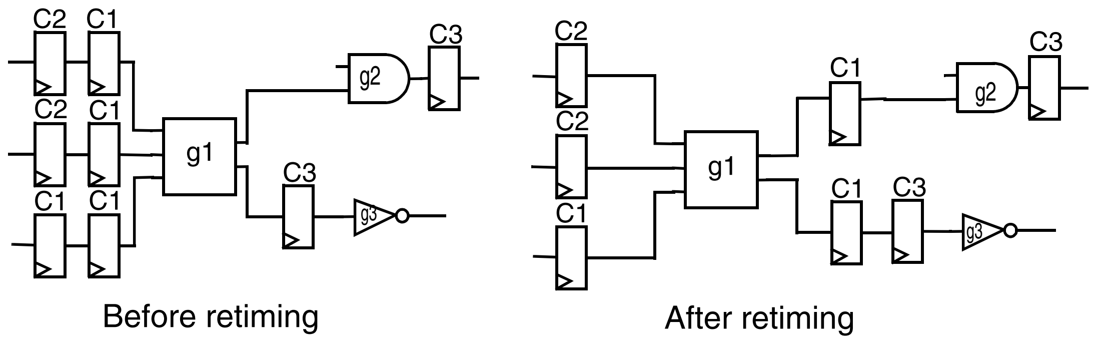
Forward Retiming

To retime forward across a combinational cell, each net in the immediate fanin of the cell must be driven by the Q pin of a register, and all these registers must belong to the same class. After the forward retiming move, the registers in the fanout of the cell belong to the same class as those in the fanin before the move.

Figure 2-4 shows an example of retiming forward across the combinational cell g1.

Note:
In this and the following sections, the explicit control nets for registers are not drawn unless there is a special reason to do so. Register classes are denoted by class names (for example, C1 and C2).

Figure 2-4 Forward Retiming Example



Before the retiming move is executed, all the registers in the immediate fanin of the g1 cell belong to class C1. Notice that after the move the three registers have been replaced by two registers belonging to the same class C1. It is not possible to retime forward the next slice of registers in the fanin of the g1 cell because not all these registers belong to the same class. Also, after the first retiming, it is not possible to retime forward the class C1 register that drives one of the g2 cell inputs because only one input pin of the cell has a register driving it.

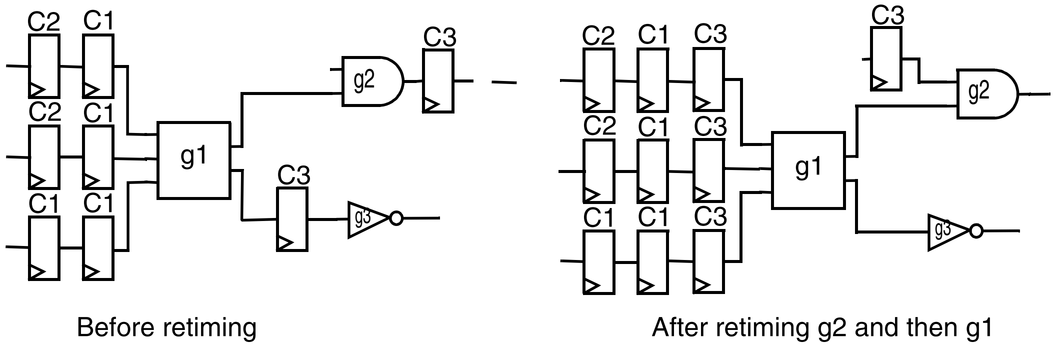
If during retiming the maximum number of forward retiming moves across a cell has been performed, the cell has reached its forward retiming boundary limit.

Backward Retiming

Rules similar to the forward retiming rules govern backward retiming across a combinational cell is possible. All nets in the fanout of a combinational cell must fan out to the D pin of the registers, and all these registers must belong to the same class. After the backward retiming move, the registers in the fanin of the cell belong to the same class as those in the fanout before the move.

Figure 2-5 shows how the combinational cell g1 can be retimed backward *after* the cell g2 has been retimed backward. Note that two backward timing moves have been carried out.

Figure 2-5 Backward Retiming Example

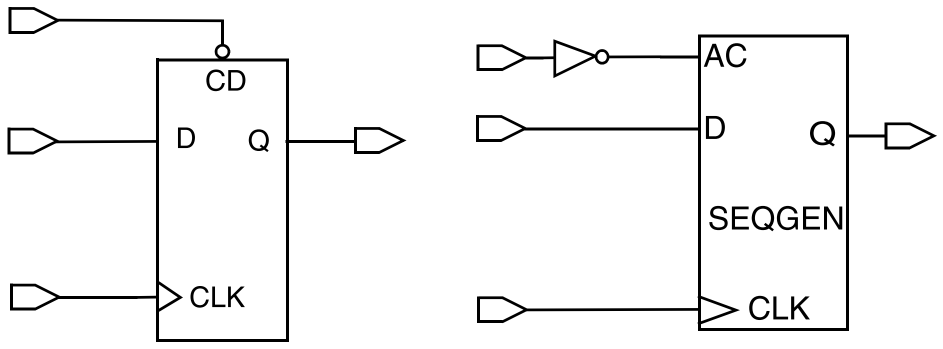


If during retiming the maximum possible number of backward retiming moves across a cell has been performed, the cell has reached its backward retiming boundary limit.

Asynchronous Control Inputs of Registers

Before the registers of a circuit can be classified for retiming, all of them have to be represented by SEQGEN cells. For registers with typical asynchronous set or clear pins, this is a relatively straightforward and unambiguous procedure. In Figure 2-6, a register with an active-low asynchronous clear input is transformed into an equivalent SEQGEN instance. (All inputs of the SEQGEN not shown in the figure are connected to a constant net with their inactive value.)

Figure 2-6 Cell Transformation With Asynchronous Clear Input



Synchronous Control Inputs of Registers

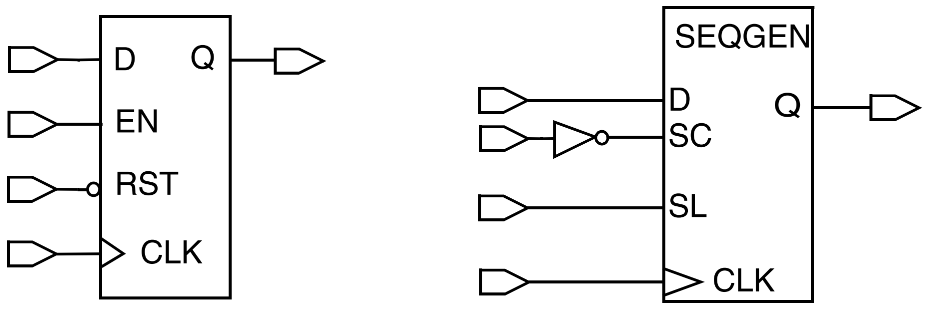
A register with typical synchronous control inputs can be represented with an equivalent SEQGEN instance in two ways:

- The synchronous control pins are directly translated to equivalent SEQGEN pins.
- The synchronous control nets are transformed through combinational decomposition of the register.

Translating Synchronous Input Pins to Equivalent SEQGEN Pins

As with asynchronous pins, this method carries out a straightforward translation of the register pins to equivalent SEQGEN pins of the SEQGEN cell. Figure 2-7 shows an example of a register with an active-low synchronous reset signal and an active-high synchronous enable signal. That is, the RST pin has been translated to an active-low SC pin, and the EN pin has been translated to an active-high SL pin.

Figure 2-7 Cell Transformation With Synchronous Clear and Enable



Note:
This method of transforming the original registers of a design leads to *multiclass* retiming because the resulting SEQGEN instances will probably not all belong to the same class.

Transforming Synchronous Input Pins Through Combinational Decomposition

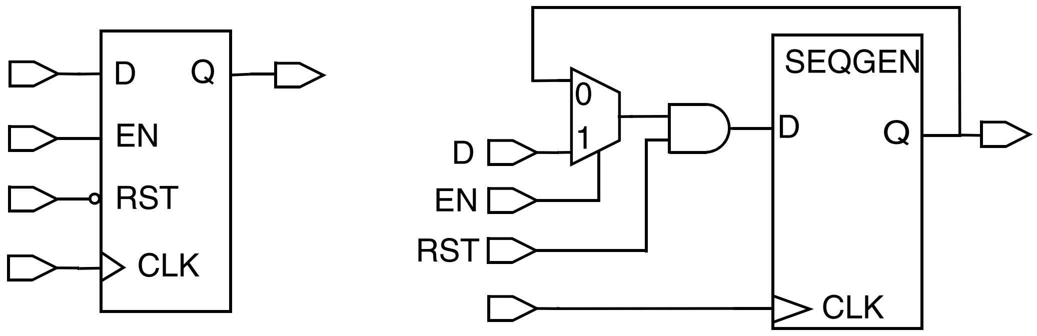
Combinational decomposition transforms synchronous input pins by

- Using combinational cells to implement the synchronous functionality, and
- Connecting the output of the combinational cells to the D pin of a SEQGEN cell

Note:
Combinational decomposition can be applied to the synchronous pins of asynchronous registers.

Figure 2-8 shows the combinational decomposition of the register example of Figure 2-7.

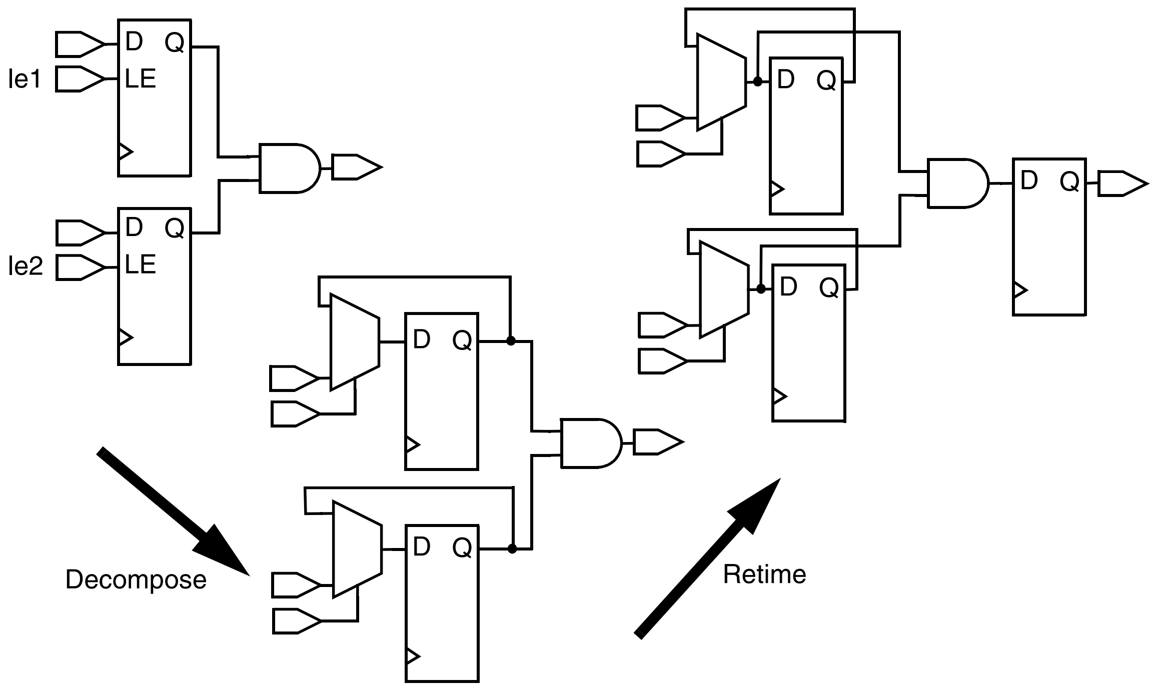
Figure 2-8 Transformation by Decomposition Example



An advantage of combinational decomposition is that all purely synchronous registers belong to the same class after they are transformed to SEQGENs. Consequently, there are no limits to the number of forward or backward moves possible at multiple input or output gates due to registers belonging to different classes. Not using decomposition (multiclass retiming) can lead to register class conflicts, which ultimately limit the number of forward or backward retiming moves possible.

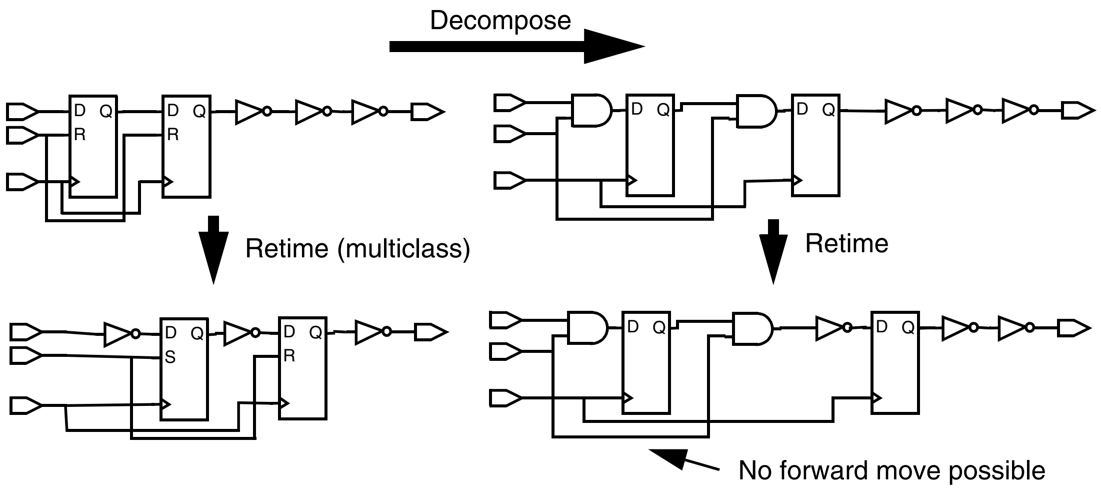
Figure 2-9 shows how registers with *different* enable control nets can be moved forward after decomposition. (To simplify the figure, the clock net is not drawn.) These registers could not be moved after multiclass retiming. Notice, however, that two D flip-flops remain with the fanouts belonging to synchronous combinational logic and cannot be forward retimed; the third flip-flop is free to move by forward retiming.

Figure 2-9 Forward Retiming of Decomposed Cells With Load Enable



Combinational decomposition can also limit the movability of registers. Figure 2-10 shows how decomposition applied to a sequence of two synchronous clear registers leaves the left register without the possibility of a forward move because the newly introduced AND gate does not have a register at its second port. (The right register is forward retimed through an inverter.) Alternatively, multiclass retiming allows both registers to move forward: The right register can be moved across two inverters and the left register across one buffer.

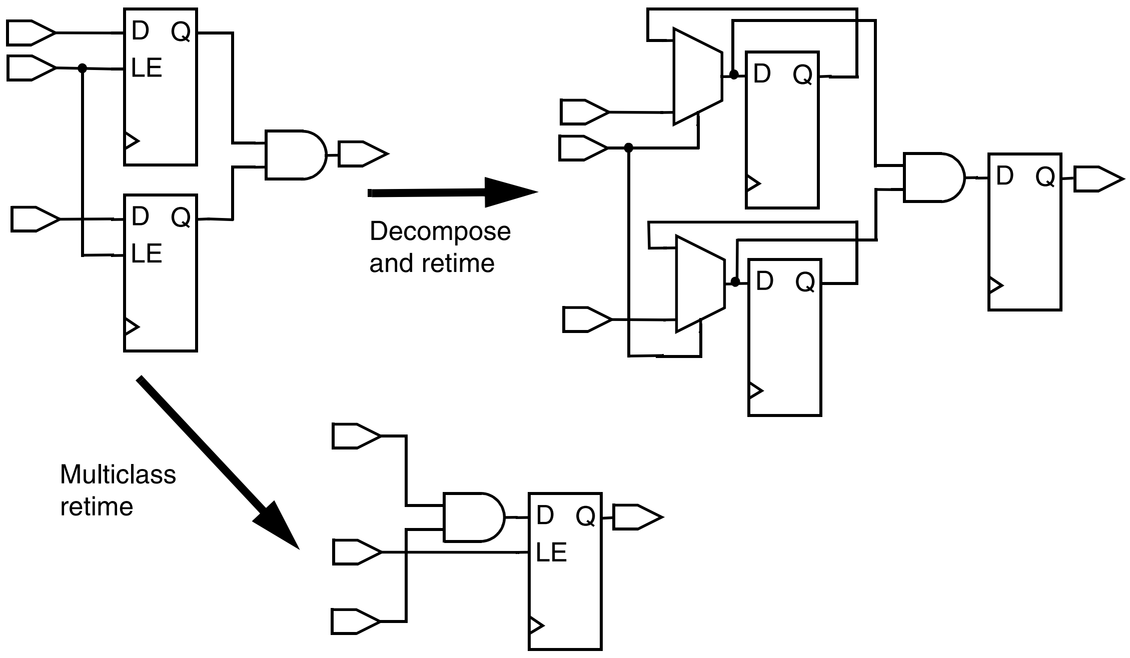
Figure 2-10 Reduced Mobility After Decomposition



Multiclass Retiming

Multiclass retiming can offer significant area savings compared with retiming after decomposition. A multiclass example is shown in Figure 2-11. The situation is similar to the example in Figure 2-9 except that the two load enable registers belong to the same class and therefore can be moved across the AND gate, leading to a single register. Using decomposition leads to a higher number of registers and additional cells after retiming.

Figure 2-11 Reduced Area Through Multiclass Retiming



Pipeline and Nonpipeline Circuits

In deciding which retiming method (multiclass or decomposition) to apply to a design, it is useful to classify circuits or parts of circuits by their topology, as well as by the types, connections, and locations of registers within the design. Circuit topologies can be classified as pipeline or nonpipeline, as defined later in this section.

For pipeline circuits, multiclass retiming is recommended. For nonpipeline circuits, decomposition or a combination of decomposition and multiclass retiming works best.

Defining Pipeline Circuits

You can understand the concept of a pipelined circuit, including register indexes, slices, and stages, in the following way:

Circuit registers can be reached by different paths from the primary data inputs. (A primary data input is an input that does not drive any control pin of a register.) During the transversal of a path, cells are passed from their input pins to their output pins.

For a particular register, the number of other registers encountered along each path from the primary inputs to the given register can be counted. If this number is independent of the path chosen (that is, the same for every path to the register), each register can be assigned this number as a *register index* of the circuit. All registers in such a circuit that have the same index form a *slice*.

If all the registers in each slice belong to the same class and if there is the same number of registers on any path from a primary input to a primary output, the circuit is classified as a *pipeline*. All other circuits not satisfying this definition are defined as nonpipelines.

The largest index occurring for the registers in a pipeline plus one is the number of pipeline *stages*. For example, a pipeline with just one slice of registers has two stages.

Note:
This definition of pipeline does not require that registers be located at the primary outputs or primary inputs.

Figure 2-12 and Figure 2-13 show examples for pipeline circuits and nonpipeline circuits.

Figure 2-12 Nonpipeline Circuit Examples

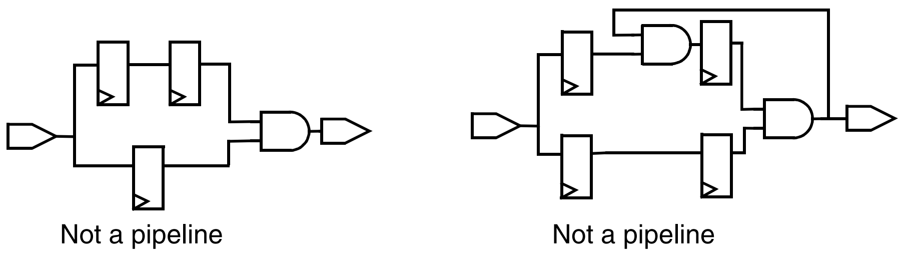
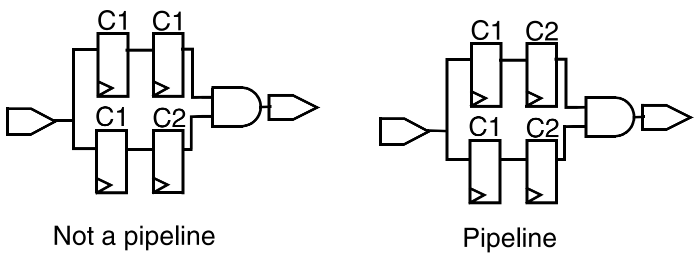


Figure 2-13 Pipeline and Nonpipeline Circuit Examples



Reset State Justification

When you move the registers in a circuit, it is not sufficient to follow only the rules for retiming of a single gate. In addition, it is usually necessary to preserve an equivalent reset state.

The circuit state is defined by the values of all the registers in the circuit at a given point in time. A circuit and its retimed version have an equivalent state if they produce the same sequences of values at corresponding primary outputs for identical sequences of values at the corresponding primary inputs.

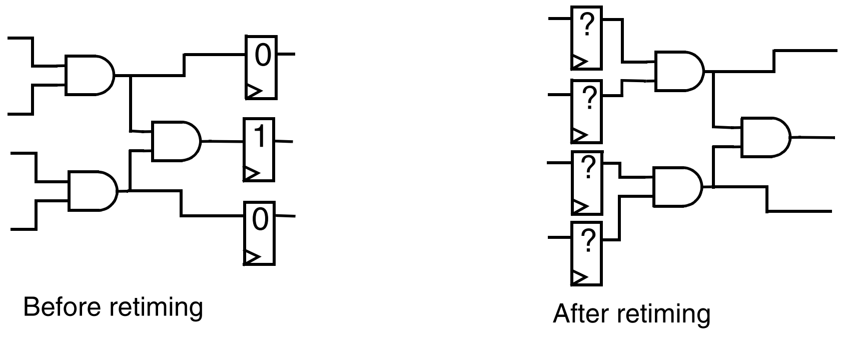
When power is switched on in a circuit, its state is unknown. Depending on the type of circuit, you might need to have an external reset or set input to reset the registers to a known value. This step ensures that the circuit has reproducible behavior after the input becomes active for the first time.

If a circuit is designed this way, by default, register retiming ensures that the reset state of the retimed circuit is equivalent to that of the original circuit, and that the behavior is identical when a finite number of clock cycles has passed after the activation of the reset. If all registers are properly reset, output sequences should match immediately.

However, a typical case where the first few output values might not match is that of a pipelined datapath where the registers do not have any set or clear connections. The maximum duration of the mismatch is the number of stages of the pipeline multiplied by the clock period.

The computation of the equivalent reset state is called *justification*. Justification for registers that have been moved forward across combinational gates is always possible and does not require significant amounts of CPU time. Justification for registers moved backward across combinational gates can be more complicated. Figure 2-14 shows an example of the difficulty with backward justification.

Figure 2-14 Impossible Backward Justification Example



The numbers inside the register symbols are the given reset values. When the registers are moved to the post-retiming positions, it is not possible to find an equivalent state for the circuit. Register retiming handles this case by finding a position for the registers where an equivalent state is found that is as close to the optimal position as possible.

Backward justification also can cost more CPU time than forward justification. If the circuit to be retimed has a reset but does not need to have an equivalent reset state after retiming, there is a method available that does not perform justification. This method can be applied to pipelined datapaths, but it is not suitable for controllers.

You can use the `-justification_effort` option of the `optimize_registers` or `set_optimize_registers` command to specify the justification effort level during backward justification. The option can take one of the following values: low, high, or medium. For more information, see Appendix B, “Command Syntax and Variable Syntax.”

3

Writing HDL Code for Retiming

In this document, the term “register” refers to both edge-triggered registers and level-sensitive latches unless stated otherwise. Both types of sequential cells can be retimed. However, the `balance_registers` command can only be applied to edge-triggered registers.

Successful register retiming very much depends on a compatible HDL description of the original design. In particular, certain design limitations must be observed, including ones specific to pipelined designs.

This chapter contains the following sections:

Allowed Circuits

For the retiming `optimize_registers` command to work successfully, the circuit HDL description must conform to certain rules. Therefore you should check the original HDL code and constraints against the following rules:

- If the `-latch` option is used, registers are treated as fixed and only level-sensitive latches are retimed. (By default, only edge-triggered registers are moved.)
- The clock distribution network can contain only buffer, inverter, and clock-gating cells. Note that clock-gating cells should be unate at their clock input pin, contain only one output, and have no more than one clock signal at their inputs. (Avoid using nonunate cells, such as XORs and XNORs, in clock networks.)
- The design can contain only master-slave or standard edge-triggered registers.
- Combinational feedback loops are not allowed.

The `balance_registers` command has additional restrictions. See Appendix B, "Command Syntax and Variable Syntax," for more information about this command.

Timing exceptions such as false paths, multicycle paths, and maximum delay constraints are supported only in a limited way by retiming. Sequential cells named explicitly as startpoints or endpoints of these exceptions are not moved. The quality of results for designs with timing exceptions might be negatively affected. Therefore, it is better to avoid having the timing exceptions of a design retimed.

When retiming latches with the `optimize_registers` command, latches must be driven by a symmetric, two-phase clock system. This system can be created by using two clock ports, one for each phase, or by using the clock signal directly for one phase and inverting this signal for the other phase.

Writing HDL Code for Pipelines

This section describes how to write HDL code in the following sections:

- Calculating the Number of Pipeline Stages
- Determining the Initial Location of the Registers
- Using DesignWare Pipeline Register Component

Calculating the Number of Pipeline Stages

There are three cases to consider when you are setting the number of pipeline stages for your design. In two of the cases, you use formulas provided in this section to compute an approximate number. Note that the actual number of stages needed is often one less or one greater than this estimated number. (If the formulas yield a fractional result, you should round the number up to the next integer value.) In the third case, the number of stages is given, so no calculation is required.

The formulas that follow use these parameters with the following definitions:

- N is the number of pipeline stages in the design. It is one more than the number of registers on any path from the primary inputs to the primary outputs.
- T_c is the clock period of the design *after* retiming.
- T_s is the setup time of a typical flip-flop used for pipelining. Select a flip-flop with the same capabilities (load enable, reset) as needed for the final pipelined design.
- T_q is the clock-to-Q delay of the typical flip-flop. (Use the same selection criteria as for the setup time.)
- T_d is the maximum delay from inputs to outputs of the compiled combinational design *before* pipelining, including input and output delays. To obtain a value for this number, perform a separate compile before adding the pipeline stages.

The target clock period cannot be less than the sum of setup time and clock-to-Q delay for the flip-flop, that is, $T_s + T_q < T_c$. Furthermore, $T_d > T_c$ is assumed. Otherwise, no pipeline registers are needed.

In the first case, the objective is to minimize the number of pipeline stages, N, when the clock period, T_c, is given. The combinational design should be compiled with tight timing constraints to achieve the smallest possible delay. If the outputs are not registered, the number of stages needed can be estimated as follows:

$$N = \frac{T_d - T_s - T_q}{T_c - T_s - T_q}$$

On the other hand, if the outputs are registered, the following estimate can be used:

$$N = \frac{T_d + T_c - T_s - 2T_q}{T_c - T_s - T_q}$$

In the second case, the objective is to minimize the overall area of the design when the clock period T_c is given. In this case, the combinational design should be compiled with an area constraint of 0 and no timing constraints. The resulting number for T_d can then be used in the same formulas as in the first case to obtain an estimate of the number of stages needed.

In the third case, the objective is to minimize the clock period when the number of stages, N, is given. Therefore nothing has to be calculated in this case. The design should be compiled with tight timing constraints, as in the first case. The retiming should then be performed with a very small target clock period that cannot be achieved (for example, T_s). Based on the negative slack values you obtain, you can find the minimum clock period possible for the design.

Determining the Initial Location of the Registers

The initial location you specify for the registers influences the movability of the registers and the CPU time needed for retiming.

If none of the registers in the pipeline is connected to any synchronous or asynchronous set or clear signal or if you plan to use the don't-care state attribute for all registers, it is usually convenient to place the registers at the primary outputs. This is easy to code, and no CPU time-consuming backward justification is necessary. Even if you choose decomposition during retiming, only the forward movability through AND gates is limited, and this is irrelevant because the registers are not moved forward during retiming.

In all other cases (that is, when there is a set or clear functionality for some of the registers and the equivalent reset state is important), you should register the primary inputs of the design. However, you should not register the clock, set, clear, or load enable inputs while registering all other inputs. Otherwise, the forward movability of the registers is limited. Placing the registers at the inputs ensures short justification times.

Using the DesignWare Pipeline Register Component

To infer pipeline registers in RTL is to use the DesignWare pipeline register component. The DesignWare library provides the DW_pl_reg pipeline register component, which makes it easy to pipeline the arbitrary logic of arithmetic structures using register retiming.

When you instantiate the DW_pl_reg component in your RTL, you specify the parameters to control the width, enable/reset, and the number of stages of the pipeline register component.

For more information about how to use the DesignWare pipeline register component, see the DesignWare Building Block IP Datasheet.

Writing HDL Code for Nonpipelines

No special restrictions or recommendations can be made for nonpipelined designs because the register locations relative to primary inputs and outputs are difficult to change in a way that still provides the required functionality. Often a complex hierarchical design contains parts that can be considered as pipelined, although the rest of the design is not. You can apply the previous guidelines to the pipelined parts of a nonpipelined design.

4

Performing Analysis and Elaboration for Retiming

In this document, the term “register” refers to both edge-triggered registers and level-sensitive latches unless stated otherwise. Both types of sequential cells can be retimed. However, the `balance_registers` command can only be applied to edge-triggered registers.

The chapter includes the following section:

- Inferring Registers for Pipelines and Nonpipelines

Note:
If any of the variables discussed in this chapter are to be set to nondefault values, they should be set before the `analyze`, `elaborate`, `read`, `read_verilog`, or `read_vhdl` commands for the HDL code are issued.

Inferring Registers for Pipelines and Nonpipelines

For pipelines, the only HDL variable that should be set differently from its default value is `hdlin_ff_always_sync_set_reset`. This variable controls whether the synchronous clear (SC) and synchronous set (SS) pins of the SEQGEN cells in the elaborated netlist are used instead of combinational cells to create set and clear. Set the variable as follows:

```
dc_shell> set hdlin_ff_always_sync_set_reset true
```

To enable multiclass retiming in pipelines, move the registers with their synchronous clear and set. Because all registers in one slice are in the same class, no moves are limited.

For nonpipelines, keep the HDL variable set to its default as follows:

```
dc_shell> set hdlin_ff_always_sync_set_reset false
```

For nonpipelines, setting this variable to true can create registers belonging to too many different classes, which in turn limits the movability of registers and consequently the delay reduction.

5

Setting Attributes and Constraints for Retiming

In this document, the term “register” refers to both edge-triggered registers and level-sensitive latches unless stated otherwise. Both types of sequential cells can be retimed. However, the `balance_registers` command can only be applied to edge-triggered registers.

This chapter shows you how to set constraints before issuing the `compile_ultra` command or `compile` command. It also discusses which of the steps often associated with compilation in the design flow should be deferred until after register retiming.

This chapter includes the following sections:

Setting Timing Constraints

For any type of design, you should set the input and output delays to realistic values by using the commands `set_input_delay` and `set_output_delay`. (For information about how to use these commands, see the appropriate man page.) How you set timing constraints depends on whether the design is pipelined or nonpipelined.

Note:
Be careful to choose the correct clock and clock edge when you set specific input and output delays.

Setting Timing Constraints for Pipelines

How you set timing constraints for pipelines depends on whether the registers are located at the primary inputs and outputs of the design or are already distributed throughout the design.

If the registers are still located at the primary inputs and outputs, the delay of the combinational circuit is usually larger than the clock period with registers at inputs and outputs. To avoid unnecessarily long processing time, set the clock period to a value greater than the target clock period for the retimed design. If the combinational delay, T_d, has already been determined as part of the computation of the number of stages, N, for a given final clock period, T_c, the combinational delay can be used as the target clock period for the initial compilation.

Otherwise, if the number of stages, N, the final clock period, T_c, the setup time, T_s, the clock-to-Q delay, T_q, and the output delay, T_o, or the input delay, T_i, are known, a target clock period, T, for the initial compilation can be computed as follows:

If the registers are located at the primary outputs,

$$T = N * (T_c - T_q - T_s) + T_q + 2T_s - T_o$$

If the registers are located at the primary inputs,

$$T = N * (T_c - T_q - T_s) + 2T_q + T_s - T_i$$

If the registers are already approximately in their final positions after you have retimed the clock period, the first compile clock period should be the same as the target clock period after retiming.

Setting Timing Constraints for Nonpipelines

For nonpipelines, the clock period for the first compile should be the same as the target clock period after retiming.

Setting the Compile Command Option on SEQGEN Cells

During compile, Design Compiler performs a step known as sequential mapping. This step maps the generic SEQGEN cells of the elaborated netlist to cells from the technology library. However, depending on the flip-flops available in the technology library and the optimizations that can be performed by sequential mapping, information about control nets can be lost in the process.

For example, if no synchronous set flip-flop is available, a SEQGEN cell whose synchronous set pin is connected to a nonconstant net might be mapped to a simple D flip-flop with an OR gate feeding its D pin. Another possibility is that the synchronous clear and the next state input might be swapped by sequential mapping, greatly increasing the number of classes and reducing the number of possible forward moves.

If you want to perform multiclass retiming, you need to restrict the sequential mapping to map the SEQGEN to the technology library cell that exactly matches the functionality of the SEQGEN cell. You can achieve this by using the `-exact_map` option of the `compile_ultra` command.

Netlist Modifications to Avoid

Certain design modifications and optimizations are often performed together with the first compilation. Some of them can still be performed before register retiming, but others must be deferred until the retiming has been performed. You should note carefully the following guidelines.

Test-Related Modifications

Test-ready compile by using the `-scan` option can be performed before register retiming. Test flip-flops and feedback loops are again inserted after retiming. You must perform retiming before scan-chain insertion; the presence of scan chains makes moving the registers impossible.

In some libraries, there are no scan equivalents for load-enable registers. Therefore, test-ready compile before retiming introduces feedback loops around the registers that retiming cannot remove. In this case, use the following commands:

```
....
#no scan option initially

compile_ultra
optimize_registers -no_compile
compile -incremental -scan
....
```

Physical Design-Related Modifications

Avoid creating a clock tree before retiming. Register retiming can retime designs with clock trees consisting of inverters, buffers, and any clock-gating cell that is unate at its clock input pin. However, the tree is no longer balanced after retiming.

Defer other physical design-related optimization options until after retiming because register locations and the delays change significantly during retiming.

Using multibit flip-flops to realize registers is possible. However, if no `dont_touch` attribute is put on these flip-flops before retiming, they are split up into single-bit registers and moved individually.

6

Retiming the Mapped Netlist

In this document, the term “register” refers to both edge-triggered registers and level-sensitive latches unless stated otherwise. Both types of sequential cells can be retimed. However, the `balance_registers` command can only be applied to edge-triggered registers.

After the preparation and initial compilation, you can retime your design by using the `optimize_registers` command. However, before using this retiming command, you must set or reset certain constraints and attributes. This chapter discusses the steps you take to use the `optimize_registers` command.

This chapter contains the following sections:

Preventing Retiming

Sometimes it is best to avoid retiming some of the registers in a design. For example, registers driving primary outputs that have to stay in place because of a particular design style should not be retimed. In this case, you should set the `dont_retime` attribute on these output registers.

Another example of not moving certain registers occurs when you want to keep the controller registers of a design in place while allowing the datapath registers to move. Keeping the controller registers in place lets you easily identify these registers and relate them to the original HDL code. In this case, you can set the `dont_retime` attribute on the registers in the controller or on the controller cell itself (if it is a hierarchical cell).

The `dont_retime` attribute prevents a register from moving during retiming but allows sequential mapping to map the register to a different flip-flop.

Use the `set_dont_retime` command to control the designs or cells that can be retimed. When set to `true` (the default), the command sets the `dont_retime` attribute on specific cells and designs in the current design so that sequential cells are not moved during retiming optimizations. For example,

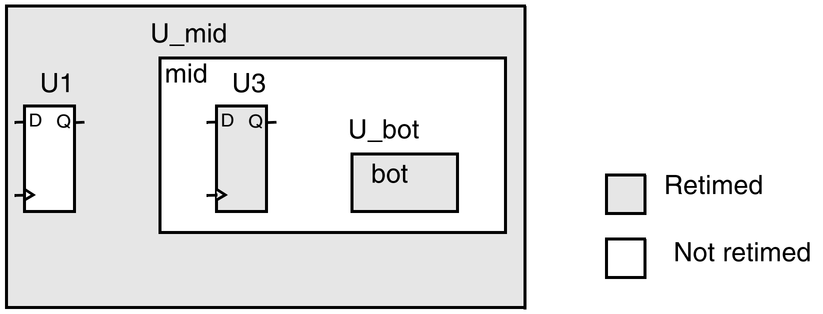
```
set_dont_retime [get_cells {z1_reg z2_reg}] true
```

Setting the `dont_retime` attribute on a hierarchical cell implies that the attribute is set on all sequential cells below it that do not have the `dont_retime` attribute set to `false`. A leaf-level cell that has the `dont_retime` attribute is not retimed if it is a sequential cell. When the `dont_retime` attribute is set on a design, all sequential cells that do not have the `dont_retime` attribute set to `false` are not retimed. For example, consider the following sequence of commands:

```
set_dont_retime [get_cells U1]
set_dont_retime [get_designs mid] true
set_dont_retime [get_cells U_mid/U3] false
set_dont_retime [get_cells U_mid/U_bot]false
optimize_registers
```

Figure 6-1 shows that cells U1 and design mid are not retimed.

Figure 6-1 Cells U1 and Design mid Are Not Retimed



Note:
The `set_dont_retime` command overrides the `set_transform_for_retiming` command. That is, if the value of the `dont_retime` attribute is `true`, setting the `transform_for_retiming` attribute to `decompose` or `multiclass` does not make a register retimable. A movable register cannot be moved across a register with the `dont_retime` or the `dont_touch` attribute.

Doing Timing Analysis During Retiming

When the registers are moved during retiming, their final location is unknown. Therefore the influence of the registers on the delay of the final circuit cannot be taken into account for each individual register. Instead, the combinational delay between registers is optimized. To set a realistic goal for the combinational delay, you must correct the target clock period by subtracting all the delay components related to the registers.

Because each register can have a different delay, you must find a representative delay value for all registers. You do this by selecting one flip-flop, called the preferred flip-flop, from the technology library. You select this flip-flop from the flip-flops instantiated in the design and similar ones available in the library. The selection criteria you should use, in the order of importance, are as follows:

1. Smallest setup time. The setup time is available directly from the library information.
2. Smallest average clock-to-Q delay. The average clock-to-Q delay is obtained by computing the average load of all nets in the design. This load is then driven by the flip-flop, and the resulting clock-to-Q delay is computed.
3. Smallest average load of all input pins. The average load of all input pins of the flip-flop is computed from the library information.

If there is a tie between several flip-flops after these criteria are applied, further selection criteria (not described here) are used. For more information about determining the selection of the preferred flip-flop, see Appendix A, “Additional Information About the Register Retiming Commands.”

After you select the preferred flip-flop, a clock correction is computed. The clock correction is determined from the following three factors:

- Setup time
- Median clock-to-Q delay of the design

  An instance of the preferred flip-flop is used to drive each net of the design. For each of these configurations, the clock-to-Q delay is computed. The median of all these values is taken as the clock-to-Q delay for the clock correction. Using the median value instead of the arithmetic average helps reduce the influence of a few extreme values on the outcome.
- Clock uncertainty

  This quantity is set by the user when determining the clock for the design.

The sum of these three values is the clock correction. The clock correction is subtracted from the target clock period, and the resulting number is used as the target value for the combinational delay during the register moving phases. If the number is less than 0, a warning is issued, and 0 is used instead.

When retiming latches with different clock ports for the two clock phases, the difference in the clock source latency of the two clock ports is also added to the clock correction.

You can prevent the clock correction, if necessary. For information about how to circumvent the computation of the clock correction, see Appendix A, “Additional Information About the Register Retiming Commands.”

During incremental compilation phase of the `optimize_registers` command, exact timing, including register delays, is used for the design.

Setting Timing Constraints

Setting accurate input and output delays for all primary inputs and primary outputs is very important. If the correct values have already been set for the compilation of the design, no change is necessary. Otherwise, you should set the values before using the `optimize_registers` command.

Input and output delays are set relative to clocks created for the design. If a port has an input or output delay set relative to a particular clock, the cells in the fanin or fanout of this port are part of the network for this particular clock if it is retimed. See "Retiming Designs With Multiple Clocks" on page 6-7.

For Design Compiler register retiming, the tool assumes that the input delay at each primary input is at least as large as the median clock-to-Q delay used for the clock correction. If the input delay is less, a warning is issued and the median clock-to-Q delay is used.

Similarly, for register retiming, the tool assumes that the output delay at each primary output is at least as large as the setup time used for the clock correction. If the output delay is less, a warning is issued and the setup time of the preferred flip-flop is used.

The clock period has to be set for the external clock port, using the `create_clock` command. The value of the clock period might be different from the one used during initial compilation, especially if the design is a pipeline whose registers are still concentrated at the outputs. The clock period value is used by the register moving phases and the incremental compilation phase of the `optimize_registers` command. For more information about how to set different constraints for the two phases, see Appendix A, "Additional Information About the Register Retiming Commands."

Selecting Transformation Options

The `optimize_registers` command lets you specify how the mapped registers are transformed to SEQGEN cells. There is a transformation option for synchronous registers and another for asynchronous registers. (Recall that a synchronous register does not have any asynchronous input pins and an asynchronous register has at least one asynchronous input pin.)

The transformation option for synchronous registers is

```
-sync_transform multiclass | decompose| dont_retime
```

The `multiclass` value specifies that the synchronous clear, set, and enable functionality is moved with the synchronous sequential cells (if they are moved during retiming). The `decompose` value specifies that any synchronous sequential cell is decomposed (transformed into an instance of a D flip-flop or latch and additional combinational logic to create the necessary synchronous functionality). The `dont_retime` value specifies that these registers are not to be moved. The default for this option is `multiclass`.

The transformation option for asynchronous registers is

```
-async_transform multiclass | decompose | dont_retime
```

The `multiclass` value specifies that the asynchronous clear and set as well as any synchronous clear, set, and enable functionality are moved with the asynchronous sequential cells (if they are moved during retiming). The `decompose` value specifies that any asynchronous sequential cell is decomposed. The `dont_retime` value specifies that these registers are not to be moved. The default for this option is `multiclass`.

Registers that are already SEQGEN instances are not affected by these settings. Their set, clear, and enable connections are controlled by HDL Compiler options, as described earlier in Chapter 3, “Writing HDL Code for Retiming.”

Recommended Transformation Options for Pipelines

For pipelined designs, the recommended transformation options for the `optimize_registers` command are as follows:

```
optimize_registers -sync_transform multiclass -async_transform multiclass
```

Because there are no class conflicts preventing registers in pipelines from being moved across combinational cells, using multiclass retiming for all types of registers is best. These settings give the best timing results with the smallest possible register count and area.

Note:
As described in Appendix A, individual attribute settings on cells or their parent cells override these option settings.

Recommended Transformation Options for Nonpipelines

For nonpipelined designs, the recommended transformation options for the `optimize_registers` command are as follows:

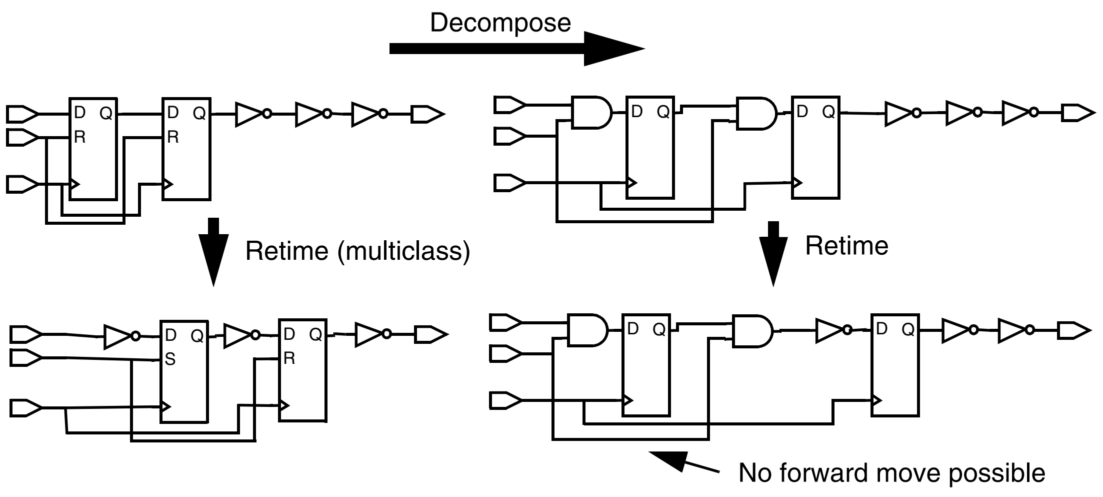
```
optimize_registers -sync_transform decompose -async_transform decompose
```

In most cases, decomposing all synchronous functionality ensures that no unnecessary class conflicts occur to limit the movability of the registers. The solution with the smallest possible delay or target delay should be found. An exception to this result can occur when the forward movability of registers is limited because of additional AND gates or OR gates as shown in Figure 6-2 on page 6-7. In this situation, setting the individual retiming attributes might help.

Note:

As described in Appendix A, individual attribute settings on cells or their parent cells override these option settings.

Figure 6-2 Reduced Mobility After Decomposition



Retiming Designs With Multiple Clocks

If the registers of the design are triggered by multiple different clocks or by both the rising and the falling edge of the same clocks, the retiming can be performed on only one clock at a time. The `optimize_registers` command offers you two ways to achieve this retiming, namely, by using the `-clock` option or not using this option.

With the -`clock` option you can specify that the registers for a single clock are retimed during one invocation of the `optimize_registers` command. By default, only the registers triggered by the rising edge of the clock are retimed. If you want to retime the registers triggered by the falling edge of the clock, you have to use the `-edge` option with the value *`fall`*.

If the `-clock` option is not used, registers for all clocks are retimed during the first (register moving) phase of retiming. The retiming is performed one clock at a time. Clocks with a larger clock period are retimed before clocks with a smaller clock period. If two clocks have the same clock period, the clock with the larger number of registers is retimed first. For each single clock, the registers triggered by the rising edge are retimed before those triggered by the falling edge. Note that this default order might not yield the best possible results. Also, retiming all clocks in the first phase means that there is no incremental optimization of the combinational logic when different clocks are retimed. Therefore it is recommended that you determine the best order for retiming clocks yourself and apply that order by using multiple runs of `optimize_registers` with the `-clock` option.

When retiming latches, a two-phase clock system is being retimed. This means that the rising and falling edges of a clock or two or more different clocks have to be retimed together. If you specify a clock using the `-clock` option while using the `-latch` option, the `optimize_registers` command retimes all the clocks and edges that need to be retimed with this clock. Note that even though the `-clock` option take only one argument, the command finds the other clock of the two-phase clock system.

Settings That Influence Register Retiming Runtime

In larger designs, using the `optimize_registers` command can result in increased CPU runtime. The following option settings can reduce the runtime while exploring the potential delay improvement resulting from register retiming. Note, however, that for the final optimization, you might not want to use some or any of the options.

- `-minimum_period_only`

  If this option is set, register count minimization is not performed. The registers are only moved to the locations that result in the smallest possible combinational delay between registers.

- `-justification_effort low | medium | high`

  Specifies the effort level to be used during backward justification of registers. Specifying a low effort ensures that justification terminates quickly; however, the quality of results (QoR) can be poor. A medium effort might provide better QoR but result in a larger runtime. A high effort could give provide the best QoR without considering runtime. The default is `medium`.

- `-no_compile`

  This argument omits the default incremental logic synthesis step normally performed after computation of the optimal sequential cell locations. If you specify this option, no design rule fixing is performed. Generic sequential cells might remain in the design.

  When you use this option, you can choose a logic compilation script adapted to your design instead of relying on the default used internally by `optimize_registers`. It is important to perform logic synthesis after sequential cell retiming to obtain the best possible timing results.

If the runtime of the `optimize_registers` command is too long, you can use all these settings at one time and then successively switch them off again to see when the runtime increases greatly.

Netlist Changes Performed by Register Retiming

Because retiming moves registers in the design, it is no longer possible to associate each register in the retimed design with exactly one register in the design before retiming. Therefore, new names have to be given to the registers. Registers that have the `dont_touch` or `dont_retime` attribute set on them are not retimed and not renamed.

Also, registers can be moved into hierarchical cells where there were no registers before retiming or into hierarchical cells where registers are not connected to the same set, clear, or enable signals. In these situations, additional pins have to be added to the hierarchical cells to have the necessary clock, set, clear, and enable nets. Adding these pins to a hierarchical cell changes its name and the name of its design.

Finally, if retiming cannot improve the delay or reduce the number of registers of the circuit, no register is moved or renamed and no incremental compilation is performed. The design is unchanged.

Delay Threshold Optimization

By default, retiming stops optimization when it detects paths that cannot be further improved by moving registers. Design Compiler reports these paths that limit further delay optimization as the critical loop in the log file when you specify the `-print_critical_loop` option with the `optimize_registers` command. For more information about critical loops, see “Displaying the Sequence of Cells That Limits Delay Optimization” in Chapter 7.

When you specify the `optimize_registers` command with the `-delay_threshold` option, retiming continues to improve paths until subcritical paths have delays less than or equal to the specified threshold.

For more information, see the `optimize_registers` man page.

7 Analyzing Retiming Results

In this document, the term “register” refers to both edge-triggered registers and level-sensitive latches unless stated otherwise. Both types of sequential cells can be retimed. However, the `balance_registers` command can only be applied to edge-triggered registers.

While it is running, the `optimize_registers` command displays information to dc_shell that can help you understand and often improve results. In addition to the standard information output to dc_shell, you can set various `optimize_registers` options to provide additional information.

This chapter contains the following sections:

Standard Output of the optimize_registers Command

By default, the `optimize_registers` command provides the following informational messages (not including warning or error messages):

- Name and setup time of the preferred flip-flop.
- Worst, best, and median clock-to-Q delay obtained, using the preferred flip-flop.
- A table with histogram information for the clock-to-Q delays found from computing the median clock-to-Q delay.
- Two values for the combinational delay between registers after registers are moved. These values, obtained from the retiming delay calculator, are referred to as the lower bound estimate and the critical path length. They are computed using slightly different methods.
- The value used for the clock correction and its components (setup time, clock-to-Q delay, and clock uncertainty).

Checking for Design Features That Limit the Quality of Results

Sometimes it is useful to obtain more statistical information about the design being retimed. You can do this by using the `-check_design` argument for the `optimize_registers` command. The additional information can often help you find potential problems in the design.

The `-verbose` argument can be used only with the `-check_design` argument to display the cells by name. Using the `-verbose` argument might produce many lines of output for large designs, but it might help identify the exact cause of a problem.

To properly analyze retiming results, you need to examine the output before and after the registers are moved. Sections “Output Before Registers Are Moved” on page 7-2 and “Output After Registers Are Moved” on page 7-4 describe types of output.

Output Before Registers Are Moved

You should analyze the following output before the registers are moved by retiming:

- All base clocks in the design that trigger registers

  For each base clock, all gated clocks that are derived from this base clock are printed. And for each gated clock, its polarity relative to its base clock is provided. A positive polarity means that a rising edge of the base clock results in a rising edge of the gated clock. A negative polarity means that a rising edge of the base clock results in a falling edge of the gated clock. If the `-verbose` option is also used, then for each gated clock, all the registers derived from the gated clock and their polarities relative to the gated clock are printed.

- The five timing arcs with the largest delay in all the combinational cells

  If a single cell has a large delay, this can severely limit the smallest delay the retiming can achieve. Therefore cells with a delay larger than a particular percentage (for example, 10 percent) of the target clock period should be avoided. Two reasons such cells might exist are as follows:

  First, a `dont_touch` attribute was put on a combinational hierarchical cell. Such a hierarchical cell appears as a single cell during the register moving phases. Consider removing the `dont_touch` attribute if the cell's delay is too large.

  Second, the presence a combinational cell from the library that is either very complex or has low drive strength and therefore a large delay. Consider compiling the design again after putting a `dont_use` attribute on this particular type of library cell.

- Delay distribution for all timing arcs in the design

  The histogram information can indicate whether there are a few cells with a particularly large delay compared to others.

- Detailed description of the selection process for the preferred flip-flop
- Total number of combinational leaf cells in the design

  The larger this number, the more complex the retiming becomes, and as a result CPU times might increase.

- Number of hierarchy cells with the `dont_touch` attribute
- Number of black box cells

  Black box cells are cells without timing information. No registers are moved across them. If you do not want to have black box cells, check the linking of your design and the completeness of the library information.

- Total number of movable sequential cells
- Number of movable synchronous sequential cells with the `decompose` attribute
- Number of movable asynchronous sequential cells with the `decompose` attribute
- Number of movable synchronous sequential cells with the `multiclass` attribute
- Number of movable asynchronous sequential cells with the `multiclass` attribute
- Number of movable sequential cells with an asynchronous clear pin
- Number of movable sequential cells with an asynchronous set pin

- Total number of immovable sequential cells

  If this number is large relative to the number of movable cells or if you suspect that some registers are not movable because of attributes or constraints you are unaware of, check the categories next in this list to find and possibly change the movability of some cells.

- Number of sequential cells that are not movable due to having the `dont_touch` attribute set

  If the `dont_touch` attribute is not necessary, remove it.

- Number of sequential cells that are not movable due to having the `dont_touch` attribute set on one of their parent cells

  If the `dont_touch` attribute is not necessary, remove it.

- Number of sequential cells that are not movable due to point-to-point exceptions

  Check whether your design has to be implemented using multicycle or false paths. Change the design or timing constraints, if possible, to eliminate these immovable sequential cells.

- Number of sequential cells that are not movable due to insufficient technology library information

  The information given on the flip-flop in the technology library is not sufficient to transform the instances of these flip-flops to SEQGEN cells. Try to compile the design before retiming, after you put a `dont_use` attribute on these flip-flop library cells.

- Number of asynchronous sequential cells that have the `dont_retime` attribute set

  If the attribute is not necessary, remove it.

- Number of synchronous sequential cells that have the `dont_retime` attribute set

  If the attribute is not necessary, remove it.

Output After Registers Are Moved

You should analyze the following output after the registers are moved by retiming:

- Total number of movable sequential cells
- Number of movable sequential cells with an asynchronous clear pin
- Number of movable sequential cells with an asynchronous set pin

Displaying the Sequence of Cells That Limits Delay Optimization

In addition to using the `-check_design` argument of the `optimize_registers` command, you can also use the `-print_critical_loop` option to find the part of the design that is limiting delay improvement. This option is available for the `optimize_registers` command.

With this option, the command displays a sequence of combinational cells, ports, and nonmovable registers in the design. The location of the registers between these cells before and after retiming is also displayed. For each cell, the rise and fall delays and the total delays from the last register or port to the output of the cell are displayed. The names of the cells and output pins are those used in the netlist before retiming. Therefore you can recognize them by looking at a schematic for this netlist in a graphical display tool such as Design Vision.

The only exceptions are the cells inserted into the netlist when registers are decomposed. You cannot find these cells in the netlist before retiming, but some of them can show up in the critical loop display. Sometimes the same sequence of cells is displayed several times. This has no particular significance.

The three different classifications of critical loops that can be printed before the sequence of cells are as follows:

- Loop without primary Input/Outputs

  In this case, the loop does not contain a primary output, a primary input, or a pin of a register that cannot move. Such a register is regarded as fixed (for example, because it has the `dont_touch` attribute set). The total delay of the cells in the loop and the number of registers in the loop determine the minimal delay that retiming can achieve. To further reduce delay, you might have to reduce the delay inside the loop (for example, by recoding the design or recompiling with different constraints). If the design allows such a modification, you can also add an additional register to the loop.

- Loop from primary input to primary output

  This case includes loops that go from actual primary input to primary outputs as well as those that begin or end at a fixed register. If the loop begins or ends at a fixed register, you might want to check whether the `dont_touch` or `dont_retime` attribute placed on that register, or on any of its parent cells, is really needed. If the startpoint and endpoint are a primary input and a primary output, you can either reduce the combinational delay between the ports by recoding and recompiling the design, or you can add registers to the design at one or more of these ports (which increases the latency).

- Loop limited by node bounds

  This case occurs when different classes of registers are present in the fanin or fanout of a cell (also referred to as a node), limiting the delay that can be achieved by retiming. To find which cells contribute to the register class conflict, look at the cells at the beginning and the end of the sequence. If the cells are combinational cells (that is, not fixed registers, black boxes, or primary ports), they are the cells responsible for the class conflict.

If the cell at the beginning of the sequence is combinational, look at the registers in its fanin. Some of them will belong to different classes or be clocked by different base clocks. If this class difference is due to the control nets to synchronous pins, you might be able to reduce the delay further by using the `decompose` option or putting the decompose transformation attribute on these registers.

Alternatively, if the combinational cell at the end of the sequence has the bound, you can apply a similar process to the registers in the fanout of the cell.

Example 7-1 shows the critical loop output with a node bound.

Example 7-1 Critical Loop Output With a Node Bound

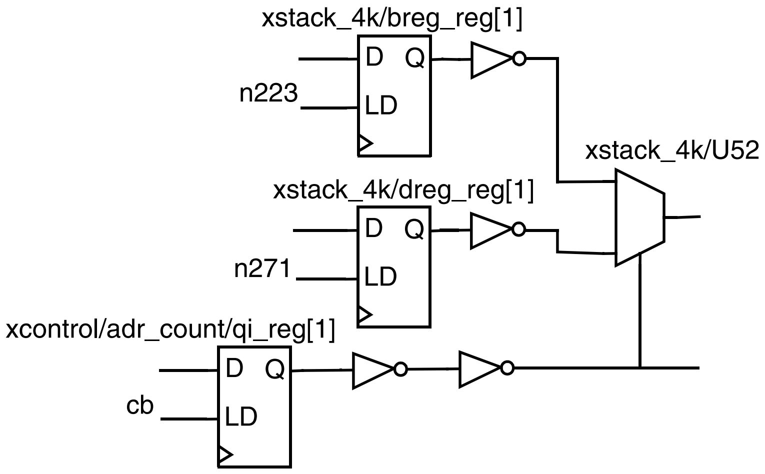
```
Critical Loop(s) for Minimum Period Retiming
--------------------------------------------
---- loop limited by node bound(s) ----

Point                        Incr          Path
-------------------------------------------------------------------------
xstack_4k/U52/Z (MX21LC)                  (  0.19   0.18)    0.19   0.18
xstack_4k/U47/Z (MX21LB)                  (  0.12   0.13)    0.30   0.32
xstack_4k/U65/Z (ND2C)                    (  0.06   0.09)    0.38   0.39
xalu/U90/Z (MX21LC)                       (  0.16   0.19)    0.55   0.57
xalu/U91/Z (N1F)                          (  0.08   0.10)    0.66   0.66
xalu/add_59/plus/plus/U24/Z (NR2C)        (  0.12   0.10)    0.78   0.76
xalu/add_59/plus/plus/U18/Z (EON1C)       (  0.07   0.10)    0.83   0.88
xalu/add_59/plus/plus/U31/Z (EN3A)        (  0.21   0.22)    1.09   1.10
xalu/U96/Z (N1B)                          (  0.05   0.06)    1.15   1.15
*** 1 register(s) AFTER min. period retiming here ***
xalu/U86/Z (OR2B)                         (  0.15   0.19)    0.15   0.19
xalu/U85/Z (ND2B)                         (  0.22   0.27)    0.41   0.42
*** 1 register(s) BEFORE min. period retiming here ***
xseg7dec/U6/Z (NR2L)                      (  0.22   0.11)    0.64   0.52
xseg7dec/U13/Z (AO2A)                     (  0.17   0.28)    0.69   0.92
xseg7dec/U30/Z (ND2B)                     (  0.06   0.08)    0.98   0.77
g/**outside** (**out_port**)              (  0.34   0.34)    1.32   1.32
(++)
-------------------------------------------------------------------------
(+) I/O port resulting from a pin of a fixed register or black box
(++) Delay may have been corrected by clock to Q (input) or setup

(output)
```

The cell xstack_4k/U52 is the one that has the conflict. Figure 7-1 shows the fanin of this cell.

Figure 7-1 Analyzing Node Bounds Example



The three registers in the fanin all belong to different classes because their LD (load enable) pins are driven by different nets. Because the design is not a pure pipeline, you could improve performance by following the general recommendation of using decomposition for all registers. If this increases the area too much, an alternative tactic is to set the `decompose` attribute on these registers individually while using the `multiclass` option for all other registers in the `optimize_registers` command.

```
dc_shell> set_transform_for_retiming \
              [get_cells "xstack_4k/breg_reg[1]"] decompose
dc_shell> set_transform_for_retiming \
              [get_cells "xstack_4k/dreg_reg[1]"] decompose
dc_shell> set_transform_for_retiming \
              [get_cells "xcontrol/adr_count/qi_reg[1]"] decompose
dc_shell> optimize_registers -sync_transform multiclass \
           -async_transform multiclass
```

You might have to repeat this process for class conflicts at other nodes, depending on the outcome of subsequent runs of the `optimize_registers` command.

A

Additional Information About the Register Retiming Commands

In this document, the term “register” refers to both edge-triggered registers and level-sensitive latches unless stated otherwise. Both types of sequential cells can be retimed. However, the `balance_registers` command can only be applied to edge-triggered registers.

This appendix describes additional options for the register retiming commands. This information is provided for use in special cases or for compatibility with previous versions of the retiming capability.

The commands for setting retiming attributes are explained in this appendix. Also, some other commands related to retiming are briefly explained.

This appendix contains the following sections:

Setting Retiming Attributes on Individual Cells

You can control the transformation and the equivalent reset state computation for individual registers by setting attributes on these cells. Setting attributes on individual registers instead of using command-line options can sometimes improve the retiming results or runtime. You can use the following `set_transform_for_retiming` command to set attributes on individual cells:

The command syntax is

```
set_transform_for_retiming cell_list \
      decompose | multiclass | dont_retime
```

This attribute determines how registers are transformed into generic SEQGEN cells and whether they are moved during retiming. A `dont_touch` attribute set on a given cell or its parent cells in the hierarchy overrides this attribute on that cell.

If no `dont_touch` attribute is set on the specified cell, `decompose` specifies that the registers affected by the attribute are decomposed when they are transformed to SEQGEN cells. The `multiclass` value specifies that the synchronous set, clear, and load pins of the SEQGEN cells are to be used in this transformation, if possible. The `dont_retime` value specifies that the registers are not to be transformed or moved.

See Appendix B, "Command Syntax and Variable Syntax" for syntax information about this command.

Other Commands Related to Retiming

The following commands relate to register retiming because they influence the behavior of the `optimize_registers` command or they perform some retiming function themselves:

- `set_register_type`

  This command lets you specify a particular flip-flop as the preferred flip-flop. You set this command before running the register retiming command as follows:

  ```
  set_register_type -exact -flip_flop flip_flop_name \
       [design_name]
  ```

  You can find potential candidates for the preferred flip-flop by running the register retiming command with the `-check_design` option but without the `set_register_type` option. However, keep in mind that this setting also restricts the flip-flops chosen by compile for the sequential mapping, including incremental compilation.

  See Appendix B, "Command Syntax and Variable Syntax" for syntax information about this command.

- `set_optimize_registers`

  This command sets the `optimize_registers` attribute on a design so that `compile_ultra` or compile automatically invoke retiming similar to the `optimize_registers` command during optimization on the instances of the design. The attribute is particularly useful for creating embedded dc_shell compilation scripts for HDL descriptions that are to be transformed to DesignWare synthetic library components.

  The `set_optimize_registers` command allows you to control retiming behavior with various options. The command can also be used together with the `optimize_registers` command and the `-only_attributed_designs` option.

  See Appendix B, "Command Syntax and Variable Syntax" for syntax information about this command.

- `balance_registers`

  This command performs register retiming on a mapped gate-level netlist that is similar to `optimize_registers` but with the following restrictions or exceptions:

  - No support is provided for multiple clocks.
  - No minimum register count retiming is performed, and the target clock period is always 0.
  - No retiming of asynchronous registers or latches is possible.
  - Synchronous register functionality is always decomposed.
  - No incremental compile is performed as part of the command.
  - No retiming analysis options are available.

  See Appendix B, "Command Syntax and Variable Syntax" for syntax information about this command.

- `set_balance_registers`

  This attribute-setting command is similar to the `set_optimize_registers` command. However, the `set_balance_registers` attribute invokes retiming functionality equivalent to the `balance_registers` command during compile time. This attribute setting command is intended only for developing pipelined synthetic library components.

  See Appendix B, "Command Syntax and Variable Syntax" for syntax information about this command.

Examples of dc_shell Register Retiming Scripts

The following sections present examples of register retiming scripts. The examples apply the `optimize_registers` command to a nonpipelined design and a pipelined design.

Script for a Nonpipelined Design Using the optimize_registers Command

Example A-1 shows the dctcl script used to apply the `optimize_registers` command to a nonpipelined design.

Example A-1 Using optimize_registers on a Nonpipelined Design

```
set search_path [concat $search_path [list "." "../lib"]]
set synthetic_library [list standard.sldb]
set target_library [list lcbg10pv.db]
set link_library [list $target_library $synthetic_library]

analyze -f vhdl vhd/calc.vhd

elaborate -architecture rtl calc

create_clock clk -period 1.5
set_input_delay 0.50 -clock clk [remove_from_collection [all_inputs] clk]
set_output_delay 0.40 -clock clk [all_outputs]

set_max_area 1000000

compile -map_effort medium

report_timing

set_max_area 0

optimize_registers -sync_transform decompose -async_transform decompose

report_timing
```

Script for a Pipelined Design Using the optimize_registers Command

Example A-2 shows the dctcl script used to apply the `optimize_registers` command to a pipelined design.

Example A-2 Using optimize_registers on a Pipelined Design

```
set search_path [concat $search_path [list "." "../lib"]]
set synthetic_library [list standard.sldb]
set target_library [list lcbg10pv.db]
set link_library [list $target_library $synthetic_library]

set hdlin_ff_always_sync_set_reset true

read_verilog verilog/tsu_mul24x24_seq.v
create_clock clk -period 11.0
set_input_delay 0.50 -clock clk [remove_from_collection [all_inputs] clk]
set_output_delay 0.40 -clock clk [all_outputs]

/* keep SEQGENs during compile */
set_dont_touch [get_cells -hier -filter "@ref_name==**SEQGEN**" *] true

compile -map_effort medium

report_timing

/* remove dont_touch attribute for retiming */
remove_attribute [get_cells -hier -filter "@ref_name==**SEQGEN**" *]
dont_touch

create_clock clk -period 4.5

optimize_registers

report_timing
```

B

Command Syntax and Variable Syntax

In this document, the term “register” refers to both edge-triggered registers and level-sensitive latches unless stated otherwise. Both types of sequential cells can be retimed. However, the `balance_registers` command can only be applied to edge-triggered registers.

This appendix contains the following sections:

The balance_registers Command

The `balance_registers` command works by timing a design and moving registers through the logic levels of the design so that the delays between the register banks are equal. The `balance_registers` command affects the state of the flip-flops internal to a design but maintains cycle-to-cycle behavior at all outputs of the design.

The `balance_registers` command has the following requirements.

- Registers must be
  - Single bit registers
  - Edge-triggered flip-flops clocked by the same phase of the same clock, or
  - Master-slave elements with the same signals for the master and slave pins, or
  - Level-sensitive latches (latches are not moved during retiming).
- Flip-flops and master-slave elements cannot be present in the same design.
- Master and slave clock waveforms cannot overlap.
- All clock pins of the flip-flops in the design must be connected to the same clock port. The interconnection from the clock port to the clock pins can contain buffers, inverters.

  If inverters are used, all clock pins of the network must be connected to the clock either by an even (including 0) number of inverters or by an odd number of inverters.

  The outputs of the clock network can be connected only to sequential cells. Buffers and inverters might be removed from the clock network of the retimed design.
- Flip-flops that have asynchronous set or clear pins are not moved.

Subdesigns in the hierarchy are ungrouped into the design unless the `dont_touch` attribute is set. The `balance_registers` command does not ungroup an instance that has the `dont_touch` attribute, and it does not place any registers inside an instance that has the `dont_touch` attribute.

The `balance_registers` command includes a delay modeling capability. For the sake of predictability, the algorithm selects a register from the target library with a small setup time, good drive at the outputs, and low input loading as compared with other registers in the library. This register is designated as the preferred flip-flop. Sequential mapping is invoked in the final step to improve the circuit by remapping the registers. The preferred flip-flop helps provide tighter bounds on the performance variation after the `balance_registers` sequence. To disable delay modeling, set the shell variable `balance_reg_delay` to 0.

As part of the delay modeling capability, `balance_registers` provides some statistics on the delays in the circuit. If the preferred flip-flop appears as a driver for a net, `balance_registers` analyzes the circuit to compute the clock-to-pin-to-next-state-pin delay for that net. These estimates provide a bound on the final clock period after the `balance_registers` sequence.

To cause `balance_registers` to be automatically invoked during compile, use the `set_balance_registers` command to set the `balance_registers` attribute on the design.

See the `balance_registers` man page for more information.

The optimize_registers Command

The `optimize_registers` command performs retiming of sequential cells (edge-triggered registers or level-sensitive latches) on a mapped gate-level netlist. The command determines the placement of sequential cells in a design to achieve a target clock period and minimizes the number of sequential cells while maintaining that clock period.

The `optimize_registers` command syntax is as follows:

```
optimize_registers
   [-minimum_period_only]
   [-no_compile]
   [-sync_transform multiclass | decompose | dont_retime]
   [-async_transform multiclass | decompose | dont_retime]
   [-check_design [-verbose]]
   [-justification_effort low | medium | high]
   [-only_attributed_designs}
   [-print_critical_loop]
   [-clock clock_name [-edge rise | fall]]
   [-latch]
   [-delay_threshold target_clock_period]
```

`-minimum_period_only`

This argument indicates that only the minimum period step of the retiming algorithm (minimum clock period retiming), and not the minimum area (sequential cell count optimization) step is to be executed. By default, the minimum period and minimum area optimization steps are executed. The `-minimum_period_only` argument is useful if you want to have a fast turnaround when trying to optimize the design's timing. The runtime is reduced, but your area results will not be optimal. After you are satisfied with the timing, you can attempt to reduce area by running the retiming command without this argument.

`-no_compile`

This argument omits the default incremental logic synthesis step normally performed after computation of the optimal sequential cell locations. If you specify this option, no design rule fixing is performed. Generic sequential cells might remain in the design.

When you use this option, you can choose a logic compilation script adapted to your design instead of relying on the default used internally by `optimize_registers`. It is important to perform logic synthesis after sequential cell retiming to obtain the best possible timing results.

`-sync_transform multiclass | decompose | dont_retime`

Specifies which transformation is used for synchronous sequential cells in the design. An edge-triggered register is synchronous if none of its input pins change the outputs asynchronously. A level-sensitive latch is considered synchronous if none of its input pins can change the outputs during the clock phase where the latch is not transparent.

Selecting the `multiclass` transformation specifies that the identifiable synchronous clear, set, and enable functionality is moved with the synchronous sequential cells if they are moved during retiming. Sequential cells are classified according to their set, clear, and enable connections. The class of the sequential cells at the fanin or fanout of a combinational cell determines whether retiming across this cell can be performed.

Selecting the `decompose` argument specifies that the synchronous cells in the design are transformed into instances of a D flip-flop or D-latch respectively and the additional combinational logic necessary to create synchronous functionality. Only the D flip-flop or D-latch can be moved during retiming.

Selecting the `dont_retime` argument specifies that the synchronous sequential cells are not moved during retiming. Their mapping, however, might change to a different flip-flop or latch from the technology library.

To set the retiming transform attribute for individual sequential cells use the `set_transform_for_retiming` command.

The default for this option is `multiclass`.

`-async_transform multiclass | decompose | dont_retime`

Specifies which transformation method is used for asynchronous sequential cells in the design. An edge-triggered register is asynchronous if at least one of its input pins changes the outputs asynchronously. A level-sensitive latch is asynchronous if at least one of its inputs can change the outputs during the clock phase where the latch is not transparent.

Selecting the `multiclass` transformation specifies that the identifiable asynchronous clear and set functionality, as well as any synchronous set, clear, and enable functionality, is moved with the asynchronous sequential cells, if they are moved during retiming. Sequential cells are classified according to their set, clear, and enable connections. The class of the sequential cells at the fanin or fanout of a combinational cell determines whether retiming can be performed across this cell.

Selecting the `decompose` transformation specifies that asynchronous sequential cells in the design are transformed into an instance of a flip-flop or latch respectively with asynchronous set and clear inputs, as necessary, and additional combinational logic to create the necessary synchronous functionality. Only the flip-flop or latch instances can be moved during retiming. They are classified according to their synchronous set, clear, and enable functionality.

Selecting the `dont_retime` value specifies that asynchronous sequential cells will not be moved during retiming. Their mapping might still be changed to a different flip-flop or latch from the technology library. The `dont_touch` attributes and retiming transformation attributes set on individual sequential cells override the value set in this option.

To set the retiming transform attribute for individual sequential cells use the `set_transform_for_retiming` command.

The default for this option is `multiclass`.

`-check_design`

Indicates that additional information about the design is to be displayed before and after retiming. This information includes the number of cells in different categories (for example, hierarchy cells with `dont_touch` attributes or nonmovable sequential cells) and more detailed information about the selection of the preferred flip-flop or latch respectively. You use this information to help in troubleshooting if retiming does not show the expected results.

`-verbose`

For use only with the `-check_design` option. Indicates that the explicit names of the cells are to be displayed along with the number of cells in each category for most categories of the `-check_design` option. The explicit naming of cells can help to locate a problem; however, the lists of output names might be long.

`-justification_effort low | medium | high`

Specifies the effort level to be used during backward justification of registers. Specifying a low effort ensures that justification terminates quickly; however, the quality of results (QoR) can be poor. A medium effort might provide better QoR but result in a larger runtime. A high effort could give provide the best QoR without considering runtime. The default is medium.

`-only_attributed_designs`

Specifies that instead of the top-level design, only instances of those designs in the hierarchy that have the `optimize_registers` attribute are retimed.

`-print_critical_loop`

Indicates that the critical loop of the design, as seen during retiming, is to be displayed. The critical loop is defined as the sequence of directly-connected combinational and sequential cells whose total combinational delay divided by the number of registers in the loop has a higher value than any other loop in the design. The critical loop limits the minimum clock period that can be achieved by retiming. Use this option to help in troubleshooting problem areas of the design if the intended clock period cannot be achieved with the given number of sequential cells in the design. If you are pipelining a datapath, you might need to add pipeline stages in the HDL code.

`-clock` *`clock_name`*

Specifies the name of the clock whose sequential cells are to be retimed. The clock must not be a virtual clock, that is, it must have a clock port associated with it. The name of the clock is either the name specified in the `create_clock` command or the name of the clock port, if the `create_clock` command had no name specified. The registers of the clock are all sequential cells which are triggered by this clock. The connection from the clock port to the sequential cell can be through clock gating cells, buffer cells and inverter cells. If the `-clock` option is specified and edge-triggered registers are retimed, registers of other clocks are not retimed. If level-sensitive latches are retimed and the `-clock` is specified, the latches driven by this clock, as well as those driven by other clocks needed to complete a two-phase clock system, are retimed. If edge-triggered registers are retimed, only registers triggered by one specific edge of the clock are retimed. By default, the registers triggered by the rising edge are retimed. A different edge can be specified using the `-edge` option.

`-edge rise | fall`

Specifies whether the registers triggered by the rising or the falling edge of the clock are to be retimed. This option can only be used together with the `-clock` option. When level-sensitive latches are retimed this option does not matter.

`-latch`

Specifies that level-sensitive latches are to be retimed instead of edge-triggered sequential cells (flip-flops). If this option is used, the edge-triggered sequential cells in the design are not moved. In order to be able to retime latches, they must be driven by a symmetrical two-phase clocks system. Latches that are used to prevent glitches in gated clocks are not moved even if the `-latch` option is used. These latches are in the fanin of clock-gating cells.

`-delay_threshold` *`target_clock_period`*

Improves paths until subcritical paths have delays less than or equal to the threshold specified by the `-delay_threshold` option. Design Compiler makes additional retiming passes until all such paths are optimized to operate at the specified target clock period.

The `optimize_registers` command operates in two phases. During the first phase, it performs retiming by moving the sequential cells in the design to meet a target clock period and minimize the number of sequential cells while maintaining that clock period. The second phase consists of an incremental compile, which adjusts the design to the changed fanout structure. The `optimize_registers` command uses the clock period of the clocks being retimed. Otherwise, the command returns without moving registers.

If the design has multiple clocks that are not virtual clocks and the `-clock` option has not been used, sequential cells for all clocks are retimed during the first phase of retiming. When retiming edge-triggered registers, the retiming is performed one clock at a time. When retiming level-sensitive latches, the retiming is performed for sets of clocks that together with their latches form a symmetric two-phase clock system. Clocks with a larger clock period are retimed before clocks with a smaller clock period. If two clocks have the same clock period, the clock with the larger number of movable sequential cells is retimed first. When retiming edge-triggered registers, for each single clock the registers triggered by the rising edge are retimed before those triggered by the falling edge. This default order might not yield the best possible results. Also, retiming all clocks in phase one prevents the incremental optimization of the combinational logic when retiming different clocks. Therefore, it is recommended that you determine the best order for retiming clocks and apply it by using multiple `optimize_registers -clock` runs.

The `optimize_registers` command has the following requirements:

- A gated clock must be derived from one of the design's clock ports or another gated clock through a unate clock gating cell (usually a logic AND or logic OR gate). The clock gating control logic can contain latches. The clock network between the base clock port and gated clocks can contain buffer and inverter cells.
- Flip-flops and master-slave elements cannot be present in the same design. Master and slave clock waveforms cannot overlap.
- All clock pins of all flip-flops in the design must be connected to their base clock or a gated clock derived from the base clock in the following way: The connection from the clock origin (the clock port or the clock gating cell) to the clock pins can contain buffer and inverter cells with one input and one output pin. All clock pins must be connected either by an even number (including zero) or odd number of inverters to the clock origin. Buffer and inverter cells on the clock network might be removed during retiming. Therefore, any existing clock tree has to be resynthesized.
- All sequential cells must be single bit, or it must be possible to decompose them into single-bit registers.
- Only certain FPGA technologies are supported.
- Designs cannot contain a combinational loop.
- The timing constraints for the design should be set in the following way: The external clock ports of the design must have a clock constraint created by the `create_clock` command. These are called the base clocks. All primary inputs of the design should have a nonnegative input delay relative to one or more of the base clocks. All primary outputs of the design should have a nonnegative output delay relative to one of the base clocks. Negative input and output delays are tolerated, but the quality of the final retiming result might be worse than expected. Point-to-point timing exceptions as created by the `set_false_path`, `set_multicycle_path`, `set_max_delay` and `set_min_delay` commands are honored, but their presence might reduce the quality of results. In the presence of such point-to-point exceptions, timing constraint violations might be worsened by retiming. Therefore, it is strongly recommended to not apply retiming to designs with these exceptions. Case analysis constraints are also ignored when moving the sequential cells. The incremental compilation after moving the sequential cells takes all types of constraints into account.

- Designs cannot contain unmapped synthetic library components.

The movement of registers and the handling of hierarchical subdesigns can be controlled as follows (in addition to the `-async_transform` and `-sync_transform` options):

- If a sequential cell has the dont_touch attribute set, the `optimize_registers` command does not move the sequential cell itself, nor does it move any other sequential cell across that cell.
- If instances contain the dont_touch attribute, they are not ungrouped. The `optimize_registers` command does not ungroup and does not place any registers inside an instance that has the dont_touch attribute set. If a hierarchical cell does not contain sequential cells and has the dont_touch attribute set, sequential cells can move across the cell. If the cell does contain sequential cells and has the dont_touch attribute set, sequential cells cannot move across the cell.
- The `optimize_registers` command can move sequential cells into a level of hierarchy. In this case, a clock pin is inserted into the interface of the instance if there was no clock pin previously. The new clock pin is named after the clock pin of the enclosing hierarchical instance.

The `optimize_registers` command supports handling of black box cells (that is, cells for which no timing is specified, such as placeholders for RAMs). The `optimize_registers` command models a black box cell as if the cell is external to the current design, without actually changing the interface of the design itself.

The `optimize_registers` command includes a delay modeling capability. For the sake of predictability, the algorithm selects from the target library a flip-flop or latch that has a small setup time, good drive at the outputs, and low input loading as compared with other flip-flops or latches in the library. This flip-flop or latch, respectively, is designated as the preferred flip-flop or preferred latch. The preferred flip-flop or preferred latch helps to provide tighter bounds on the performance variation after the `optimize_registers` command sequence. To select a particular preferred flip-flop or latch, use the `set_register_type` command described in the section, "The set_register_type Command" on page B-13. To exclude certain flip-flops from being chosen, use the `set_dont_use` command.

After retiming, implementation selection is no longer performed on synthetic library components in the design by subsequent executions of compile.

To cause `optimize_registers` to be automatically invoked during compile, use the `set_optimize_registers` command to set the `optimize_registers` attribute on the design.

See the `optimize_registers` man page for more information.

The set_balance_registers Command

The `set_balance_registers` command sets the `balance_registers` attribute on the specified designs or on the current design so that the design is retimed during compile. The `set_balance_registers` command syntax is as follows:

```
set_balance_registers [true | false] [-design design_list]
```

`true | false`

This argument is the value with which to set the `balance_registers` attribute. The default is true.

`-design`

This argument specifies a list of designs to retime. The default is the current design.

If the `balance_registers` attribute is set to true (the default) on the design, compile automatically invokes the `balance_registers` command, which moves registers to minimize the maximum register-to-register delay. Subdesigns in the hierarchy are ungrouped into the design, unless the `dont_touch` attribute is set.

In addition, it is a mistake to invoke `balance_registers` on a design that contains generic logic. If the `balance_registers` attribute is set, compile attempts to optimize the design by invoking `balance_registers`. Be sure that your design contains no generic logic when `balance_registers` is called during compile.

To remove `balance_registers`, use `remove_attribute` or `reset_design`. You can achieve the same effect by setting the `balance_registers` attribute to false.

See the `set_balance_registers` man page for more information.

The set_optimize_registers Command

The `set_optimize_registers` command sets the `optimize_registers` attribute on the specified design or on the current design, so that compile automatically invokes the `optimize_registers` attribute to retime the design during optimization.

The `set_optimize_registers` command syntax is as follows:

```
set_optimize_registers
   [true | false]
   [-design design_list]
   [-minimum_period_only]
   [-sync_transform multiclass | decompose | dont_retime]
   [-async_transform multiclass | decompose | dont_retime]
   [-clock clock_name [-edge rise | fall]]
   [-check_design   [-verbose]]
   [-latch]
   [-justification_effort low | medium | high]
```

`true | false`

These arguments set the value with which to set the `optimize_registers` attribute. The default is true.

`-design`

This option specifies a list of designs to retime. The default is the current design.

`-minimum_period_only`

This argument indicates that only the minimum period step of the retiming algorithm (minimum clock period retiming), and not the minimum area (sequential cell count optimization) step is to be executed. By default, the minimum period and minimum area optimization steps are executed. The `-minimum_period_only` argument is useful if you want to have a fast turnaround when trying to optimize the design's timing. The runtime is reduced, but your area results will not be optimal. After you are satisfied with the timing, you can attempt to reduce area by running the retiming command without this argument.

`-sync_transform multiclass | decompose | dont_retime`

Specifies which transformation is used for synchronous sequential cells in the design. An edge-triggered register is synchronous if none of its input pins change the outputs asynchronously. A level-sensitive latch is considered synchronous if none of its input pins can change the outputs during the clock phase where the latch is not transparent.

Selecting the `multiclass` transformation specifies that the identifiable synchronous clear, set, and enable functionality is moved with the synchronous sequential cells if they are moved during retiming. Sequential cells are classified according to their set, clear, and enable connections. The class of the sequential cells at the fanin or fanout of a combinational cell determines whether retiming across this cell can be performed.

Selecting the `decompose` argument specifies that the synchronous cells in the design are transformed into instances of a D flip-flop or D-latch respectively and the additional combinational logic necessary to create synchronous functionality. Only the D flip-flop or D-latch can be moved during retiming.

Selecting the `dont_retime` argument specifies that the synchronous sequential cells are not moved during retiming. Their mapping, however, might change to a different flip-flop or latch from the technology library.

To set the retiming transform attribute for individual sequential cells use the `set_transform_for_retiming` command.

The default for this option is `multiclass`.

`-async_transform multiclass | decompose | dont_retime`

Specifies which transformation method is used for asynchronous sequential cells in the design. An edge-triggered register is asynchronous if at least one of its input pins changes the outputs asynchronously. A level-sensitive latch is asynchronous if at least one of its inputs can change the outputs during the clock phase where the latch is not transparent.

The default for the `-async_transform` option is `multiclass`. Selecting the `multiclass` transformation specifies that the identifiable asynchronous clear and set functionality, as well as any synchronous set, clear, and enable functionality, is moved with the asynchronous sequential cells, if they are moved during retiming. Sequential cells are classified according to their set, clear, and enable connections. The class of the sequential cells at the fanin or fanout of a combinational cell determines whether retiming can be performed across this cell.

Selecting the `decompose` transformation specifies that asynchronous sequential cells in the design are transformed into an instance of a flip-flop or latch respectively with asynchronous set and clear inputs, as necessary, and additional combinational logic to create the necessary synchronous functionality. Only the flip-flop or latch instances can be moved during retiming. They are classified according to their synchronous set, clear, and enable functionality.

Selecting the `dont_retime` value specifies that asynchronous sequential cells will not be moved during retiming. Their mapping might still be changed to a different flip-flop or latch from the technology library. The `dont_touch` attributes and retiming transformation attributes set on individual sequential cells override the value set in this option.

To set the retiming transform attribute for individual sequential cells use the `set_transform_for_retiming` command.

`-check_design`

Indicates that additional information about the design is to be displayed before and after retiming. This information includes the number of cells in different categories (for example, hierarchy cells with dont_touch attributes or nonmovable sequential cells) and more detailed information about the selection of the preferred flip-flop or latch respectively. You use this information to help in troubleshooting if retiming does not show the expected results.

`-verbose`

For use only with the `-check_design` option. Indicates that the explicit names of the cells are to be displayed along with the number of cells in each category for most categories of the `-check_design` option. The explicit naming of cells can help to locate a problem; however, the lists of output names might be long.

`-print_critical_loop`

Indicates that the critical loop of the design, as seen during retiming, is to be displayed. The critical loop is defined as the sequence of directly-connected combinational and sequential cells whose total combinational delay divided by the number of registers in the loop has a higher value than any other loop in the design. The critical loop limits the minimum clock period that can be achieved by retiming. Use this option to help in troubleshooting problem areas of the design if the intended clock period cannot be achieved with the given number of sequential cells in the design. If you are pipelining a datapath, you might need to add pipeline stages in the HDL code.

`-clock` *`clock_name`*

Specifies the name of the clock whose sequential cells are to be retimed. The clock must not be a virtual clock, that is, it must have a clock port associated with it. The name of the clock is either the name specified in the `create_clock` command or the name of the clock port, if the `create_clock` command had no name specified. The registers of the clock are all sequential cells which are triggered by this clock. The connection from the clock port to the sequential cell can be through clock gating cells, buffer cells and inverter cells. If the `-clock` option is specified and edge-triggered registers are retimed, registers of other clocks are not retimed. If level-sensitive latches are retimed and the `-clock` is specified, the latches driven by this clock, as well as those driven by other clocks needed to complete a two-phase clock system, are retimed. If edge-triggered registers are retimed, only registers triggered by one specific edge of the clock are retimed. By default, the registers triggered by the rising edge are retimed. A different edge can be specified using the `-edge` option.

`-edge rise | fall`

Specifies whether the registers triggered by the rising or the falling edge of the clock are to be retimed. This option can only be used together with the `-clock` option. When level-sensitive latches are retimed this option does not matter.

`-latch`

Specifies that level-sensitive latches are to be retimed instead of edge-triggered sequential cells (flip-flops). If this option is used, the edge-triggered sequential cells in the design are not moved. In order to be able to retime latches, they must be driven by a symmetrical two-phase clocks system. Latches that are used to prevent glitches in gated clocks are not moved even if the `-latch` option is used. These latches are in the fanin of clock-gating cells.

`-justification_effort low | medium | high`

Specifies the effort level to be used during backward justification of registers. Specifying a low effort ensures that justification terminates quickly; however, the quality of results (QoR) can be poor. A medium effort might provide better QoR but result in a larger runtime. A high effort could give provide the best QoR without considering runtime. The default is medium.

See the `set_optimize_registers` man page for more information.

The set_register_type Command

The `set_register_type` command specifies latch or flip-flop type information for the compile to use by setting appropriate attributes on the designs or cell instances.

The `set_register_type` command syntax is as follows:

```
set_register_type -latch example_latch | -flip_flop example_flip_flop
[-exact][cell_or_design_list]
```

`-latch` *`example_latch`* `-exact`

The `-exact -latch` *`example_latch`* specifies a latch from the target library to be used by compile as the exact latch for cells and as the exact default latch for designs. It sets the `default_latch_type_exact` attribute to the example latch on all designs in the *`cell_or_design_list`*, and the `latch_type_exact` attribute to the example latch on all cells in *`cell_or_design_list`*. Notice that you must use the `-exact` argument with the `-latch` argument. You can specify both the `-latch` argument and the `-flip_flop` argument; however, you must specify at least one.

`-flip_flop` *`example_flip_flop`* `-[exact]`

The `-exact -flip_flop` *`example_flip_flop`* argument specifies a flip-flop from the target library to be used by compile as the default flip-flop type. If you use `-exact`, this indicates that compile is to make an exact mapping to the example flip flop, if possible. The argument sets the `default_flip_flop_type` or `default_flip_flop_type_exact` attribute to the example flip-flop on all designs in *`cell_or_design_list`*; and the `flip_flop_type` or `flip_flop_type_exact` attribute to the example flip-flop on all cells in *`cell_or_design_list`*. You can specify both the `-latch` argument and the `-flip_flop` argument; however, you must specify at least one.

`cell_or_design_list`

The *`cell_or_design_list`* argument specifies a list of cells or designs in which the specified latch or flip-flop is to be used. The default is the current design.

See the `set_register_type` man page for more information.

The set_transform_for_retiming Command

The `set_transform_for_retiming` command sets the `transform_for_retiming` attribute on cells in the current design. This command can affect hierarchical cells and sequential leaf cells.

The `set_transform_for_retiming` command syntax is as follows:

```
set_transform_for_retiming cell_list

multiclass | decompose | dont_care
```

`cell_list`

The *cell_list* argument is a list of cells on which the `transform_for_retiming` attribute is to be set. If you specify more than one cell name, the names must be enclosed in quotation marks (“”) or in braces ({}). For more information about cell names, see to the Synopsys `find` command man page.

`multiclass | decompose | dont_care`

The `multiclass | decompose | dont_retime` argument specifies the value with which to set the `transform_for_retiming` attribute. There is no default. One of the values must be specified.

If the `transform_for_retiming attribute` is placed on a sequential, nonhierarchical cell, the attribute value determines the way in which that cell is transformed for retiming. The attribute does not affect nonsequential, nonhierarchical cells.

The `multiclass` value specifies that the cell is moved together with any synchronous or asynchronous preset or clear or synchronous load enable signals.

The `decompose` value specifies that synchronous load enable and synchronous reset logic are made explicit, and only the basic storage register is moved. For example, if a register has a data (D) input, a clock (CLK) input, and a synchronous clear (SD) input with active-low polarity and the `transform_for_retiming` attribute is set to `decompose`, it is decomposed into a simple register with D and CLK input and an AND gate driving the D input. The inputs of this AND gate are the original data net and the synchronous clear net. Only the simple register is moved for retiming, while the AND gate stays in place.

The `dont_retime` value specifies that the cell is not retimed. It can still be mapped to a different library cell during incremental mapping optimizations. If you want to disable retiming and sequential mapping optimizations for a cell, use the `dont_touch` attribute.

If the attribute is set on a hierarchical cell, it applies to all sequential cells in the hierarchy below this cell, unless the attribute is set on a hierarchical cell in between or on the sequential leaf cell itself. Values of the attribute set on lower levels of hierarchy override those set on higher levels.

If the attribute has not been set on a sequential leaf cell or on any of its hierarchical parent cells, the transform that is applied to this cell is determined by the corresponding command-line option chosen for the `optimize_registers` command, or by the default setting for the retiming command that you are using.

If the `dont_touch` attribute is set to true on a cell, the `transform_for_retiming` attribute does not come into effect on this cell or any of its child cells.

Using the attribute can improve timing results but at the cost of increased area for certain designs.

To remove the `transform_for_retiming` attribute, use the `remove_attribute` command.

See the `set_transform_for_retiming` man page for more information.

Index