Multibit Register Synthesis and Physical Implementation Application Note

Version V-2023.12, December 2023

SYNOPSYS®

Copyright and Proprietary Information Notice

Destination Control Statement

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

Disclaimer

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

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

www.synopsys.com

Feedback

Contents

About This Application Note

The Design Compiler, DC Explorer, and IC Compiler II tools can replace single-bit register cells with multibit register cells if such cells are available in the logic library and physical library. Using multibit register cells in place of single-bit cells can reduce area, clock tree net length, and power. This application note describes the synthesis and physical implementation flows using multibit registers:

This preface includes the following sections:

New in This Release

Information about new features, enhancements, and changes, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the Design Compiler Release Notes on the SolvNetPlus site.

Related Products, Publications, and Trademarks

For additional information, see the documentation on the Synopsys SolvNetPlus support site at the following address:

https://solvnetplus.synopsys.com

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

- Design Compiler®
- DC Explorer
- IC Compiler™ II
- IC Compiler™

Conventions

The following conventions are used in Synopsys documentation.

Feedback

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`
`Purple`	• Within an example, indicates information of special interest. • Within a command-syntax section, indicates a default, such as `include_enclosing = true | false`
[ ]	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`
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Bold	Indicates a graphical user interface (GUI) element that has an action associated with it.
Edit > Copy	Indicates a path to a menu command, such as opening the **Edit** menu and choosing **Copy**.
Ctrl+C	Indicates a keyboard combination, such as holding down the Ctrl key and pressing C.

Customer Support

Customer support is available through SolvNetPlus.

Accessing SolvNetPlus

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

Feedback

To access the SolvNetPlus site, go to the following address:

https://solvnetplus.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 sign up for an account.

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

Contacting Customer Support

To contact Customer Support, go to https://solvnetplus.synopsys.com.

1

Multibit Register Synthesis and Physical Implementation

Synthesis and physical implementation tools can organize multiple register bits into groups called “multibit components” in the RTL bus inference flow or “banks” in the placement-aware flow. The register bits in a group are targeted for implementation using multibit registers. For example, a group of eight register bits can be implemented as a single 8-bit library register or two 4-bit library registers.

The tool initially represents register bits using single-bit registers. You can instruct the tool to replace groups of register bits with multibit register cells according to specified rules.

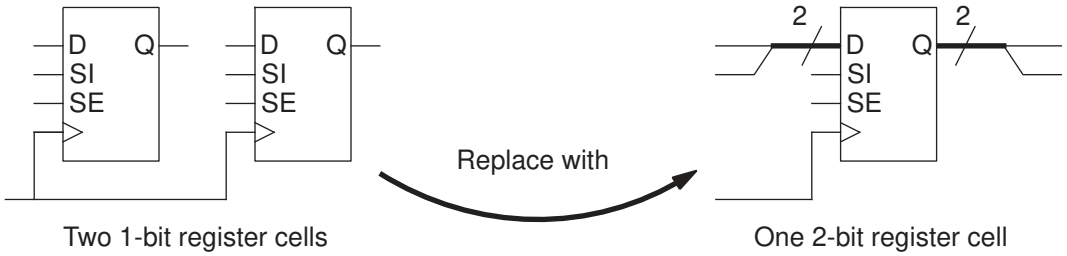
Replacing single-bit cells with multibit cells offers the following benefits:

- Reduction in area due to shared transistors and optimized transistor-level layout
- Reduction in the total length of the clock tree net
- Reduction in clock tree buffers and clock tree power

These benefits must be balanced against the lower flexibility in placing the registers and routing the connections to them.

Figure 1 shows how multiple single-bit registers can be replaced with a multibit register.

Figure 1 Replacing Multiple Single-Bit Register Cells With a Multibit Register Cell



The area of the 2-bit cell is less than that of two 1-bit cells due to transistor-level optimization of the cell layout, which might include shared logic, shared power supply connections, and a shared substrate well. The scan bits in the multibit register can be

connected together in a chain as in this example, or each bit can have its own scan input and scan output.

The Design Compiler tool offers two different methods for replacing single-bit register cells with multibit register cells:

- **RTL bus inference flow** in the Design Compiler (in wire load and topographical modes) and DC Explorer tools. In this flow, the tool groups the register bits belonging to each bus (as defined in the RTL) into multibit components. You can also group bits manually by using the `create_multibit` command. The bits in each multibit component are targeted for implementation using multibit registers. The actual replacement of register bits occurs during execution of the `compile_ultra` or `compile_exploration` command.
- **Placement-aware register banking flow** in the Design Compiler Graphical and IC Compiler II tools. In this flow, the tool groups single-bit register cells that are physically near each other into a register bank and replaces each register bank using one or more multibit register cells. This method works with both logically bused signals and unrelated register bits that meet the banking requirements. The register bits assigned to a bank must use the same clock signal and the same control signals, such as preset and clear signals.

Both types of flows support scan chain generation and design-for-test protocols. A design modified by mapping single-bit registers to multibit registers can be formally verified with the Formality tool.

To support multibit register flows, the logic library and physical library must contain both single-bit and multibit library cells, and the multibit cells must meet certain requirements so that the tools can recognize them as functionally equivalent to a group of single-bit cells.

The following topics describe the synthesis and physical implementation flows using multibit registers:

- Library Requirements
- RTL Bus Inference Flow
- Placement-Aware Multibit Register Banking in Design Compiler Graphical
- Enhanced Placement-Aware Multibit Register Banking
- Using Sequential Mapping to Map Single-Bit Registers
- Handling Timing Exceptions With Multibit Banking
- Debanking Registers in Design Compiler Graphical
- Placement-Aware Multibit Register Banking and Debanking in IC Compiler II
- Reporting Multibit Registers in Your Design

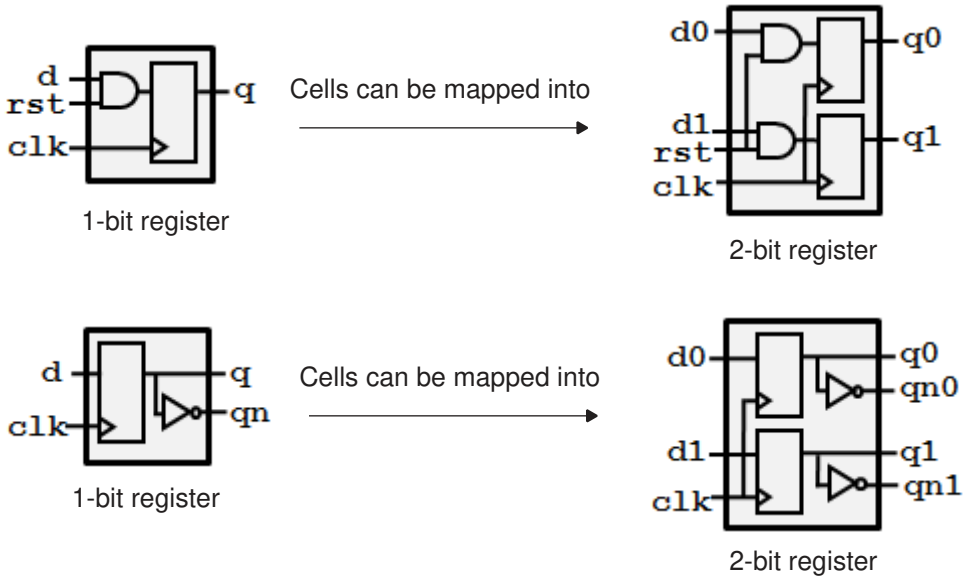
Feedback

- Reasons for Not Banking
- Multibit Flows

Library Requirements

To perform mapping from single-bit to multibit registers, the tool checks for matching pin functions and naming conventions in the multibit register pins, as shown in Figure 2.

Figure 2 Mapping of Single-Bit to Multibit Cells With the Same I/O Pins



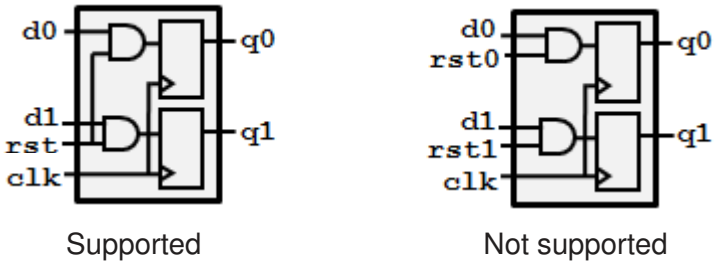
For example, if a single-bit register has Q and QN outputs, the tool can replace this register only with a multibit register that also has Q and QN outputs for each output register bit.

Feedback

In the placement-aware flow,

- The Design Compiler tool matches the single-bit cells with the multibit cells using the functionality of the cell in the library. The tool can match single-bit registers and multibit cells with different pin names.
- The Design Compiler Graphical and IC Compiler II tools do not support multibit registers that have a control pin for an individual bit, as shown in Figure 3.

Figure 3 Multibit Registers With a Control Pin For an Individual Bit Are Not Supported



For scan cells,

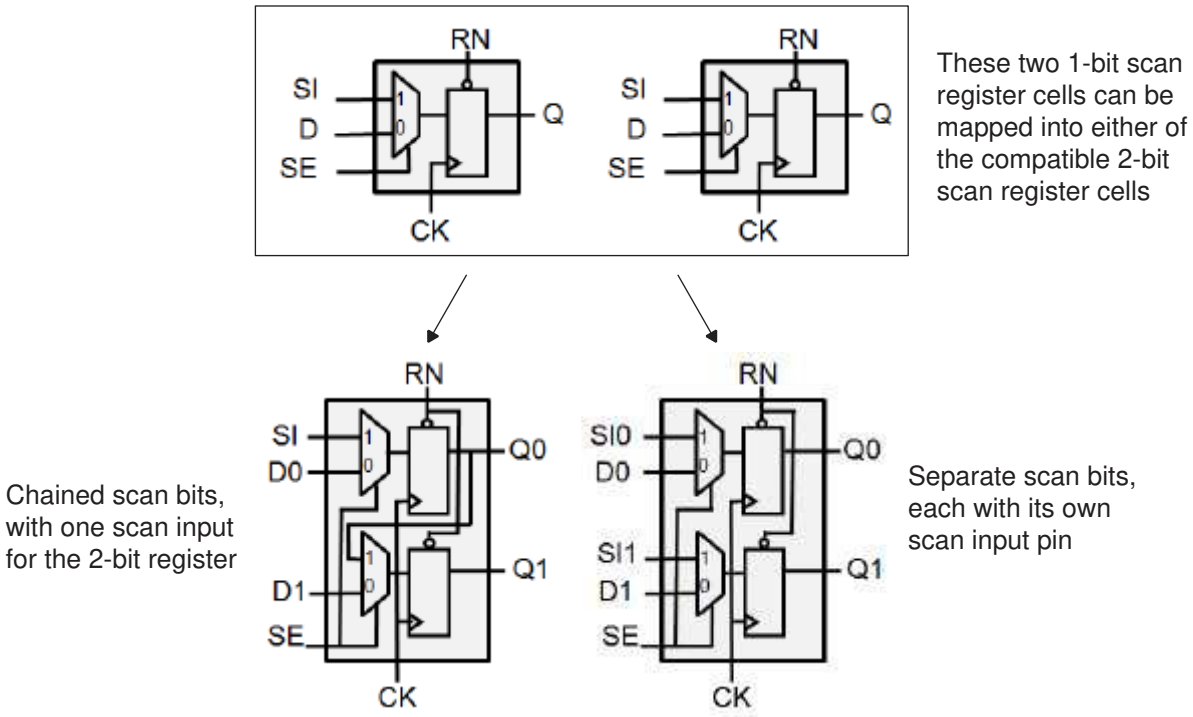
- You can use a multibit register cell with single scan input and single scan output for the whole cell, with the scan bits daisy-chained inside the cell; or with one scan input and one scan output for each register bit. Both types of multibit scan register configurations are supported. Dedicated scan output signals are also supported.

  Figure 4 shows how two single-bit scan cells can be mapped into either of two different compatible multibit scan cells, which have different scan configurations.

- In the Design Compiler and IC Compiler II tools,
  - You can map multibit registers with a dedicated scan-out pin even if the equivalent single-bit registers do not have a dedicated scan-out pin. This feature supports RTL inference and physically-aware multibit banking flows.
  - You can map multibit latches and flip-flops.

Feedback

Figure 4 Single-Bit Scan Cell and Compatible Multibit Scan Cells



In the Liberty syntax, the `ff_bank`, `latch_bank`, or `statetable` group describes a cell that is a collection of parallel, single-bit sequential parts. Each part shares control signals with the other parts and performs an identical function. Both groups are typically used in `cell` and `test_cell` groups to represent a multibit register.

In the Liberty-format description of a multibit register cell, the `ff_bank` or `latch_bank` group keyword defines the multibit characteristic. For details, see the *Library Compiler User Guide* available on SolvNetPlus:

- For the single-bit and multibit flip-flop definition syntax, see “Describing a Flip-Flop” and “Describing a Multibit Flip-Flop” in the “Defining Sequential Cells” chapter.
- For the single-bit and multibit latch definition syntax, see “Describing a Latch” and “Describing a Multibit Latch,” in the “Defining Sequential Cells” chapter.

For examples of Liberty-format descriptions of multibit cells, see Appendix A, Multibit Cell Modeling.

RTL Bus Inference Flow

In the RTL bus inference flow, the Design Compiler and DC Explorer tools infer multibit registers from the buses defined in the RTL and group the register bits of a bus into a

multibit component. A multibit component is a group of single-bit registers marked as candidates for implementation using one or more multibit library cells. The replacement of single-bit cells with multibit library cells occurs when you use the `compile_ultra` command.

The Design Compiler and DC Explorer tools initially represent all register bits using single-bit cells. To perform mapping from single-bit to multibit cells, you can use either or both the following methods:

- Before you read the RTL, set the `hdlin_infer_multibit` variable to specify how the tool performs multibit register inference from the RTL buses.
- After you read the RTL, use the `create_multibit` command to group specified register bits into multibit components.

To generate following reports on the multibit components in a design, both before and after using the `compile_ultra` command, use the `report_multibit` command:

- Before compile, the `report_multibit` command reports the groupings of single-bit register cells resulting from inference from the RTL and by using the `create_multibit` command.
- After compile, the `report_multibit` command reports the same multibit components, each component now replaced by one or more multibit cells, or still consisting of single-bit cells if multibit mapping was not successful.

Note:

- Commands used in the RTL bus inference flow do not work in the placement-aware multibit flow.
- RTL bus interference flow is not supported in the IC Compiler II tool.

The DC Explorer tool does not support the placement-aware register banking flow. Therefore, the tool does not support commands used in the flow, such as `identify_register_banks`, `create_register_bank`, and `split_register_bank`.

To infer multibit registers and handle multibit components in the RTL bus inference flow, see the following topics:

- Multibit Register Inference While Reading the RTL
- Creating Multibit Components After Reading the RTL
- Reporting Multibit Components
- Writing Multibit Components
- Multibit Register Naming in the RTL Inference Flow
- Removing Multibit Components

Feedback

- Multibit Synthesis Optimization
- Using Other Optimization Flows With RTL Inference Flow

See Also

- Using Sequential Mapping to Map Single-Bit Registers
- Handling Timing Exceptions With Multibit Banking

Multibit Register Inference While Reading the RTL

In the Design Compiler and DC Explorer tools, the `hdlin_infer_multibit` variable determines how the tool performs multibit register inference when it reads in the RTL. The variable can be set to any one of the following values:

- `never`: The tool does not infer any multibit registers when it reads in the RTL.
- `default_none`: The tool infers multibit registers only where enabled by directives embedded in the RTL. This is the default.
- `default_all`: The tool infers multibit registers from all bused registers, except where disabled by directives embedded in the RTL.

The `infer_multibit` and `dont_infer_multibit` directives in the RTL enable and disable multibit register inference. For example, the following block of code defines a single Verilog multibit flip-flop register q0 with the default setting of `default_none`:

```
module test (d0, d1, d2, rst, clk, q0, q1, q2);
  parameter d_width = 8;

  input [d_width-1:0] d0, d1, d2;
  input clk, rst;
  output [d_width-1:0] q0, q1, q2;
  reg [d_width-1:0] q0, q1, q2;

  //synopsys infer_multibit "q0"
  always @(posedge clk)
  begin
    if (!rst) q0 <= 0;
    else q0 <= d0;
  end

  always @(posedge clk or negedge rst)
  begin
    if (!rst) q1 <= 0;
    else q1 <= d1;
  end

  always @(posedge clk or negedge rst)
```

```
  begin
    if (!rst) q2 <= 0;
    else q2 <= d2;
  end

endmodule
```

For more information about using the `infer_multibit` and `dont_infer_multibit` directives, see the *HDL Compiler for SystemVerilog User Guide* or the *HDL Compiler for VHDL User Guide*.

Creating Multibit Components After Reading the RTL

In the Design Compiler or DC Explorer tool, the `create_multibit` command explicitly assigns registers by name into a multibit component. You specify the list of register cells to be included within a multibit component, and optionally a name for the new component.

For example,

```
dc_shell> create_multibit {x_reg[0] x_reg[1]} -name my_mult1
```

This command groups the single-bit registers xreg[0] and xreg[1] into a new multibit component called my_mult1.

You can use wildcard characters in the object list:

```
dc_shell> create_multibit {y_reg*}
```

In this example, the `create_multibit` command assigns all registers named y_reg* to a new multibit component. Design Compiler creates the name of the multibit component based on the name of the registers. To create a specific multibit component name, use the `-name` option.

By default, the command organizes the register bits in reverse alphanumeric order, which affects their order of mapping during synthesis with the `compile_ultra` command. To sort them in forward order instead, use the `-sort` option of the `create_multibit` command.

You can create multibit components in subdesigns as shown in the following example:

```
dc_shell> create_multibit {U1/xrg[0] U1/xrg[1] U1/xrg[2]}
```

All specified register cells must be in the same level of the hierarchy.

Feedback

Reporting Multibit Components

After you read in the design and allow the tool to infer multibit components from the RTL, or after you use the `create_multibit` command, you can report the multibit components with the `report_multibit` command. For example,

```
dc_shell> report_multibit y_reg
 ...

Attributes:
    b - black box (unknown)
    h - hierarchical
    n - noncombinational
    r - removable
    u - contains unmapped logic


Multibit Component : y_reg
Cell               Reference       Library      Area   Width  Attributes
--------------------------------------------------------------------------
y_reg[2]           **SEQGEN**                   0.00   1      n, u
y_reg[1]           **SEQGEN**                   0.00   1      n, u
y_reg[0]           **SEQGEN**                   0.00   1      n, u
--------------------------------------------------------------------------
Total 3 cells                                   0.00   3

Total 1 Multibit Components
```

The multibit component y_reg contains three register bits. The notation `**SEQGEN**` in the “Reference” column means “generic sequential” element, a technology-independent model of a register bit. The `u` in the “Attributes” column indicates a cell that contains unmapped logic. The three unmapped cells are targeted for mapping to a multibit library cell.

After you use the `compile_ultra` command, the same `report_multibit` command reports the library cells used to implement the generic sequential registers:

```
dc_shell> compile_ultra
 ...
dc_shell> report_multibit y_reg
 ...

Attributes:
    b - black box (unknown)
    h - hierarchical
    n - noncombinational
    r - removable
    u - contains unmapped logic

Multibit Component : y_reg
Cell             Reference      Library         Area    Width  Attributes
```

```
--------------------------------------------------------------------------
y_reg[2:1]       SDFF2          ump108v_125c    56.58   2       n, r
y_reg[0]         SDFF1          ump108v_125c    32.32   1       n
--------------------------------------------------------------------------
Total 2 cells                                   88.91   3

Total 1 Multibit Components
```

In this example, the `compile_ultra` command mapped the three generic sequential register bits into one 2-bit library cell and one 1-bit library cell.

To report multibit components in a subdesign, specify the hierarchical instance name of the subdesign as in the following example:

```
dc_shell> report_multibit U1/*
```

The following example generates a report of multibit components in all of the subdesigns:

```
dc_shell> report_multibit -hierarchical
```

Writing Multibit Components

You can write the multibit components to a Tcl file using the `write_multibit_components` command:

```
dc_shell> write_multibit_components -output mb_comp.tcl
```

This generates an mb_comp.tcl file that contains a list of `create_multibit` commands:

```
create_multibit -name "mb_reg1" { "mb_reg1[2]" "mb_reg1[1]" "mb_reg1[0]" }
create_multibit -name "mb_reg2" { "mb_reg2[2]" "mb_reg2[1]" "mb_reg2[0]" }
```

The Tcl file is useful if you intend to re-create the multibit components in a new Design Compiler session and your handoff is in ASCII format. You can source the file in the new Design Compiler session:

```
dc_shell> source -e -v mb_comp.tcl
dc_shell> report_multibit
 ...

Attributes:
    b - black box (unknown)
    h - hierarchical
    n - noncombinational
    r - removable
    u - contains unmapped logic

Multibit Component : mb_reg1
Cell               Reference       Library    Area   Width  Attributes
--------------------------------------------------------------------------
mb_reg1[2]           SDFF1           my_lib    2.32   1      n
mb_reg1[1]           SDFF1           my_lib    2.32   1      n
```

Feedback

```
mb_reg1[0]            SDFF1            my_lib     2.32    1       n
--------------------------------------------------------------------------------
Total 3 cells                                     6.96    3

Multibit Component : mb_reg2

Cell                Reference       Library       Area    Width   Attributes
--------------------------------------------------------------------------------
mb_reg2[2]            SDFF1             my_lib     2.32    1       n
mb_reg2[1]            SDFF1             my_lib     2.32    1       n
mb_reg2[0]            SDFF1             my_lib     2.32    1       n
--------------------------------------------------------------------------------
Total 3 cells                                     6.96    3

Total 2 Multibit Components
```

Multibit Register Naming in the RTL Inference Flow

The variables listed in Table 1 control the naming of multibit registers in the RTL inference flow.

Table 1 Multibit Register Naming in the RTL Inference Flow

Variable name	Default	Usage	Example
`bus_range_separator_style`	:	Determines the separator used to name a multibit cell that implements consecutive bits	If q_reg[0] and q_reg[1] are mapped to a multibit cell, the name of the resulting register would be q_reg[1:0]
`bus_multiple_separator_style`	,	Determines the separator used to name a multibit cell that implements nonconsecutive bits	If q_reg[2] and q_reg[4] are mapped to a multibit cell, the name of the resulting register would be q_reg[2,4]
`bus_multiple_name_separator_style`	,,	Determines the separator used to name a multibit cell that implements bits whose original base names differ.	If q_reg and p_reg are mapped to a multibit cell, the name of the resulting register would be q_reg,,p_reg

If you use the `change_names` command before compile, the Design Compiler tool uses double commas (,,), instead of colon (:) or single comma (,). Consider that the single-bit register names are *A_reg[1]* and *A_reg[0]* and the multibit register name in the default flow is *A_reg[1:0]*. When you use the `change_names` command before compile, the single-bit

register names are renamed to *A_reg_1_* and *A_reg_0_*, and the multibit register name is renamed to *A_reg_1_,,A_reg_0_*.

If you specify incorrect settings for the bus naming style variables, the tool uses the default for the variables and issues an OPT-916 warning.

For example, if you specify the same value for the `bus_multiple_name_separator_style` and `bus_multiple_separator_style` variables, the tool issues an OPT-916 warning during compile:

```
prompt> set_app_var bus_multiple_separator_style "_MB_"
prompt> set_app_var bus_multiple_name_separator_style "_MB_"
prompt> compile_ultra
Warning: Incorrect setting for bus naming style variables. (OPT-916)
…
report_multibit
Multibit Component : q_reg

Cell                  Reference    Library     Area   Width  Attributes
--------------------------------------------------------------------------
q_reg_1_,,q_reg_0_    MBFF         my_lib      4.62   2      n
--------------------------------------------------------------------------

Multibit Component : p_reg

Cell                 Reference      Library     Area   Width  Attributes
-------------------------------------------------------------------------
p_reg[2,4]           MBFF           my_lib      4.62   2      n
-------------------------------------------------------------------------
```

For more information, see the OPT-916 man page.

Removing Multibit Components

If you do not want a particular set of single-bit registers to be grouped into a multibit component, use the `remove_multibit` command to cancel the grouping if needed. For example,

```
dc_shell> remove_multibit y_reg
 ...
```

In this case, the command removes the multibit component named y_reg, causing its register bits to be excluded from grouping during multibit synthesis. It does not remove the register bits themselves.

The `remove_multibit` command works on the specified multibit components, whether they are created by RTL inference or by the `create_multibit` command.

You can specify a multibit component name directly, which removes the whole component. Alternatively, you can specify the names of cells or cell instances, which are individually removed from existing multibit components without affecting the remaining register cells.

You can also specify the design name to remove all multibit components or multibit registers in the specified design. This is useful in cases where you want to exclude specific subdesigns from multibit mapping.

If the design has already been compiled, the `remove_multibit` command removes the multibit component grouping but does not separate the multibit register into single-bit registers. To do that, you need to run an incremental compile operation after the `remove_multibit` command, as shown in the following example.

```
dc_shell> compile_ultra # Replaces single-bit cells with multibit cells
 ...
dc_shell> report_timing # Reports timing results after synthesis

dc_shell> report_multibit # Reports multibit cells used in synthesis
 ...
dc_shell> remove_multibit get_cells_y_reg[1:0] # Removes the y_reg[1:0]
                                               # multibit cell from the
                                               # multibit component
                                               # grouping
 ...

dc_shell> compile_ultra -incremental # Replaces multibit cells with
                                     # single-bit cells
```

Note:

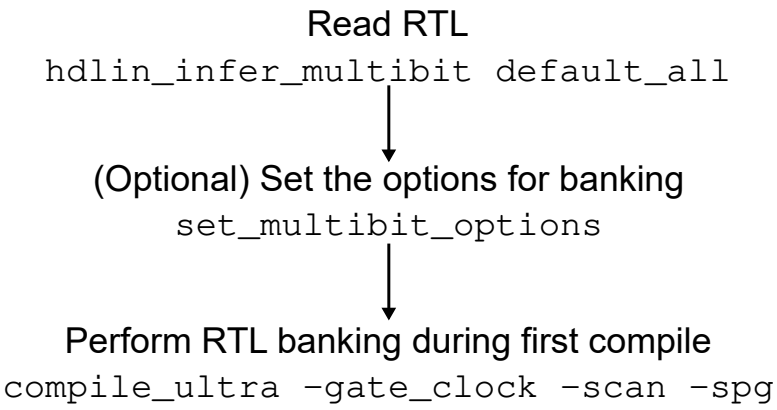
An incremental compile operation recognizes the `remove_multibit` command, but not the `create_multibit` command. It can decompose multibit cells, but not create new ones. Also, an incremental compile operation does not reconnect scan chains broken by decomposing multibit cells into single-bit cells.

Multibit Synthesis Optimization

Single-bit cells are grouped into multibit components by inference from the RTL or by using the `create_multibit` command, or both. The single-bit cells of a multibit component are targeted for replacement with one or more multibit cells. The `compile_ultra` command performs the actual replacement.

Feedback

Figure 5 Multibit Registers Inferred From Buses



The `compile_ultra` command has the flexibility to perform multibit replacement in multiple ways. For example, a 4-bit multibit component could be implemented as a single 4-bit register, two 2-bit registers, or one 2-bit register and two 1-bit registers.

The single-bit cells of a multibit component can be replaced with multibit registers if

- The type of registers is the same
- The same type of multibit register is available in the target library
- The registers are driven by the same clock and the same common control signal
- The registers do not have the `dont_touch` or `size_only` attributes
- The registers do not have timing exceptions (including group paths) on the clock pin or the common input pin
- The registers, if they are retention type registers, belong to the same retention strategy, as specified by the `-lib_cells` option of the `map_retention_cell` command

  If you use the `-elements` option with the `set_retention` and `map_retention_cell` commands to specify leaf cells to which the retention strategy applies, be sure to specify both the single-bit and multibit leaf cell instances.

To guide the `compile_ultra` command in deciding how to perform multibit register replacement, use the `set_multibit_options` command:

```
dc_shell> set_multibit_options -mode mode_name
```

The *mode_name* can be any one of the following multibit optimization modes:

- non_timing_driven (the default): The tool uses multibit cells whenever it can, resulting in the fewest possible remaining single-bit cells. Timing and area are allowed to become worse.
- timing_friendly: The tool replaces single-bit cells with multibit cells by choosing mulitibit library cells that are best for timing. However, timing might degrade during banking of multibit cells.
- timing_driven: The tool replaces single-bit cells with multibit cells only where timing does not get worse.
- none: The tool does not replace single-bit cells with multibit cells during compile, meaning that no multibit optimization takes place when you run the compile_ultra command.

When the compile_enable_physical_multibit_banking variable is set to true, the following modes are available.

See Also

- Handling Timing Exceptions With Multibit Banking

Using Other Optimization Flows With RTL Inference Flow

When you enable optimization flows with the RTL inference flow, the Design Compiler tool handles registers as follows:

- Retiming

  If you enable retiming on specific hierarchies, registers in these hierarchies are not replaced with multibit register cells during the RTL inference flow. Use the placement-aware register banking flow to bank such single-bit register cells to multibit registers cells. See Placement-Aware Multibit Register Banking in Design Compiler Graphical.

- Manual register replication

  During the RTL inference flow, if you set registers for replication by using the set_register_replication command, the registers can be replaced only with multibit register cells. However, after replacing registers with multibit registers, the tool does not replicate these registers.

If you want these registers to be replicated and not to be replaced with the multibit register cells, you should manually exclude these registers from multibit banking before using the `compile_ultra` command, as follows:

```
prompt> foreach_in_collection cell [get_flat_cells * \
-filter "defined(register_replication)"] \
{remove_multibit [get_object_name $cell]}
```

Placement-Aware Multibit Register Banking in Design Compiler Graphical

In the placement-aware register banking flow, the Design Compiler Graphical tool groups single-bit register cells that are physically near each other into a register bank and implements the register bank using one or more multibit register cells.

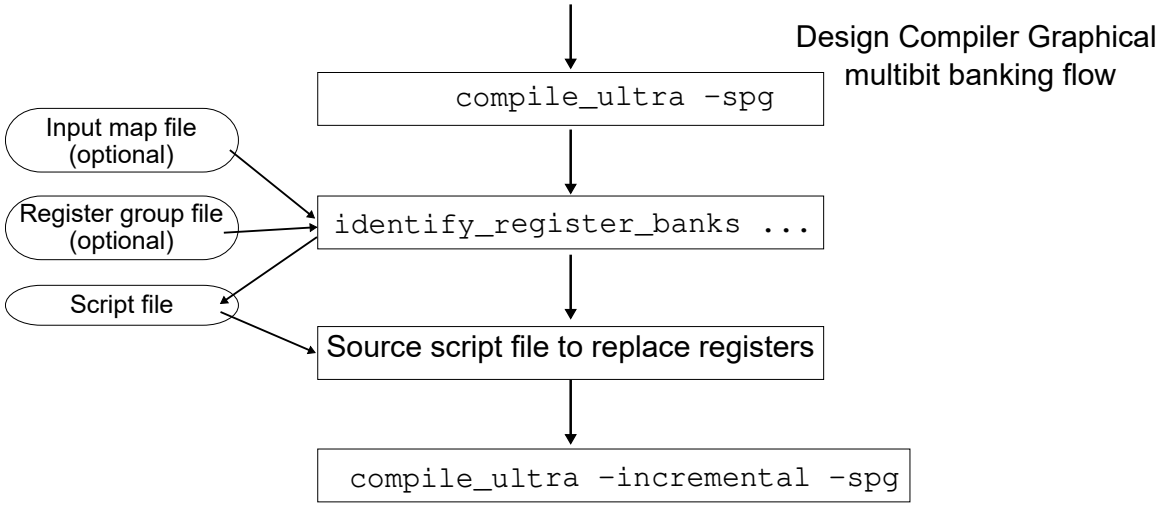
The placement-aware register banking flow offers the following advantages over the RTL bus inference flow:

- The tool uses physical location information to help decide which single-bit registers to map to multibit registers.
- The tool maps functionally unrelated registers, as well as bused registers, to multibit registers.
- The tool supports the usage of library cells that have more complex functionality.
- You can manually replace registers quickly, without running the `compile_ultra` command.

Figure 6 summarizes the flow in the Design Compiler Graphical tool.

Feedback

Figure 6 Multibit Banking Flow in the Design Compiler Graphical Tool



In the Design Compiler Graphical tool, the `identify_register_banks` command finds compatible groups of single-bit registers that are physically near each other and writes out a script containing `create_register_bank` commands. You can then execute the script, which performs the actual mapping of single-bit to multibit registers in the netlist. An incremental compile operation completes the register banking flow.

To group single-bit register cells and implement the register bank in the placement-aware register banking flow, see the following topics:

- Specifying Register Grouping
- Banking Registers
- Banking Multibit Components Using the identify_register_banks Command
- Multibit Register Naming Style
- Preventing Updating of Attributes during Multibit Banking and Debanking

See Also

- Input Map File and Register Group File
- Using Sequential Mapping to Map Single-Bit Registers
- Handling Timing Exceptions With Multibit Banking

Feedback

Specifying Register Grouping

To specify the grouping of single-bit registers into multibit register banks in the placement-aware multibit flow, use the `identify_register_banks` command in the Design Compiler tool.

The Design Compiler Graphical command performs the banking analysis and generates the script file immediately.

Single-bit registers can be grouped when they

- Are in the same logical hierarchy
- Are in the same physical bound
- Are driven by the same clock net
- Are driven by the same common control signal net (optional)

  You can control this with the `-common_net_pins` option.
- Belong to the same register group in the register group file
- Do not have the `dont_touch` or `fixed_placement` attributes
- Do not have the `size_only` attribute (optional)

  You can control this with the `-exclude_size_only_flops` option.
- Are not the start or stop register of a scan chain (optional)

  You can control this with the `-exclude_start_stop_scan_flops` option.

During grouping, the Design Compiler tool issues the following message:

```
Total number of ignored flops which are multi-bit flops   : 80
 Total ignored flops   : 80
 Total flops banked    : 5606
 Total flops in design : 6064
 Banking ratio         : 92.45%
 Effective banking ratio: 93.68%
```

Note:

The banking ratio is determined by the total number of registers banked divided by the total number of registers in the design. The following equation shows effective banking ratio is determined by the total number of registers banked divided by the total number of registers in the design minus the total number of ignored registers in the design.

$$\text{Effective banking ratio} = \frac{\text{Total number of registers banked}}{\text{Total number of registers in the design} - \text{Total number of ignored registers in the design}}$$

```
                                (Total number of registers in the design -
                              Total number of ignored registers in the design)
```

For information about using the script to report effective banking ratio, see Appendix C, Reporting Effective Banking Ratio.

The total number of registers banked is equivalent to the total bit number of the multibit registers.

This banking ratio is reported during grouping. Some of the banking commands could be rejected when executing the `create_register_bank` command. The banking ratio after executing the banking commands could be different from this report.

In the Design Compiler tool, to get the banking ratio, source the file generated by the `identify_register_banks` command and use the `report_multibit_banking` command.

identify_register_banks Command (Design Compiler)

The `identify_register_banks` command assigns the single-bit registers in the design into groups. It does not actually replace the single-bit registers. Instead, it writes out a banking script file containing `create_register_bank` commands. To perform register banking and change the design netlist, you need to execute the script file. Any existing multibit registers are excluded by the `identify_register_banks` command.

You do not need to specify the input map file and register group file with the `-input_map_file` and `-register_group_file` options. If you do not specify the files, the `identify_register_banks` command identifies the single-bit registers that can be replaced by available multibit registers in the target and link libraries based on the functional information of the library cells; the tool then uses that information to control mapping.

When you run the `identify_register_banks` command without specifying the `-input_map_file` and `-register_group_file` options, the command groups only registers that have the same net connection to the common pins. Note that you do not have to specify the `-common_net_pins` option with the `identify_register_banks` command.

To see which single-bit registers the tool has identified for replacement by multibit registers, use the `write_multibit_guidance_files` command. The command generates two mapping guidance files, an input map file and a register group file. After examining the contents of these mapping guidance files, you can change the way the tool performs multibit mapping by modifying the files and specifying them with the `-input_map_file` and `-register_group_file` options when you run the `identify_register_banks` command. The tool only accepts and uses user-specified input files when both the input map file and register group file are provided.

You can optionally restrict grouping by specifying that certain pins of the single-bit registers need to be connected to a shared net, such as the clock pin or reset pin. To do so, use the `-common_net_pins` option.

You can optionally exclude certain single-bit registers from consideration for multibit register banking by specifying their library cell names or instance names, or by specifying exclusion criteria such as a timing slack threshold or the size-only attribute.

To bank registers in a multibit component if they are physically near each other, use the `-multibit_components_only` option. The option banks cells belonging to the same multibit component to multibit registers. However, cells are not banked across different multibit components. For flow details, see Banking Multibit Components Using the identify_register_banks Command.

Banking Registers

To combine single-bit registers into multibit registers, use the `create_register_bank` command. This command performs the actual mapping of single-bit to multibit registers.

create_register_bank Command

To prepare for replacing single-bit registers with multibit registers, the tool writes out a banking script containing `create_register_bank` commands. You need to execute the script to carry out the actual replacement of single-bit cells with multibit cells. You can insert, delete, and edit the `create_register_bank` commands in the script file to modify the banking behavior of the script.

You can also execute the `create_register_bank` command by itself to perform a specific banking task.

Specifying the library name in the `-lib_cell` argument is mandatory in the Design Compiler tool .

For example, the following command replaces the single-bit register instances reg_u1 through reg_u4 with an instance of the my_mbit_lib/mreg_4bit library cell:

```
prompt> create_register_bank -name my_mregA \
                {reg_u1 reg_u2 reg_u3 reg_u4} \
                -lib_cell my_mbit_lib/mreg_4bit
...
************ CREATE BANK ************
Creating cell 'my_mregA' in design 'test'

Removing cell 'reg_u1'
Removing cell 'reg_u2'
Removing cell 'reg_u3'
Removing cell 'reg_u4'

********* CONNECTION SUMMARY ************
```

```
    Cell:                my_mregA
    Reference:           sdff_2bit
    Hierarchy:           test
    Library:             class

    Input Pins           Net
    -----------------    ------------
    D0                   d[0]
    CK                   net6
    D1                   d[1]
    D2                   d[2]
    D3                   d[3]

Output Pins              Net
-----------------        ------------
    Q0                   q[0]
    Q1                   q[1]
    Q2                   q[2]
    Q3                   q[3]

************ BANKING SUMMARY ************

The following 4 register instances: reg_u1 reg_u2 reg_u3 reg_u4
have been merged into 4-bit register 'my_mregA' (lib_cell: mreg_4bit)
1
```

Figure 7 Multibit Banking

During multibit banking, the `create_register_bank` command checks if the single-bit registers

- Are mapped to the equivalent library cells
- Are in the same logical hierarchy
- Are driven by the same clock net
- Are driven by the same common control signal net
- Do not have a `dont_touch` or `fixed_placement` attribute
- Have the same `size_only` attribute value
- Do not have timing exceptions (including group paths) on a clock pin

If the single-bit registers do not meet these conditions, they are not replaced with multibit registers and a PSYN message is generated. For example,

```
prompt> create_register_bank -name group0_1 \
           { data_a_reg data_b_reg} -lib_cell mb_lib/MB2FF

************ CREATE BANK ************
Warning: Inconsistent net connection between pin (data_a_reg/CLK) and pin
    (data_2_reg/CLK) (PSYN-1203)
```

After using the `create_register_bank` commands, placement of multibit registers might not be in optimal locations. During incremental compile, the multibit registers are placed to achieve improved QoR.

When the tool replaces single-bit registers with a multibit register, the tool sets a `register_list` attribute on the multibit cell. The attribute specifies the single-bit cells that were mapped to the multibit cell so you can identify the original single-bit register names of each multibit cell.

For example, if single-bit registers, reg_3, reg_2, reg_1, and reg_0, are mapped to a multibit register, the tool sets the `register_list` attribute with a {reg_3 reg_2 reg_1 reg_0} value on the resulting multibit register. The order of the list of single-bit registers is determined by the pin names of the multibit register, in ascending order. If the multibit register has D0, D1, D2, and D3 data input pins, the original reg_3 register is assigned to the D0 pin of the multibit register and the original reg_0 register is assigned to the D3 pin of the multibit register.

Banking Multibit Components Using the identify_register_banks Command

In this flow, you use the `identify_register_banks` command with the `-multibit_components_only` option to replace single-bit registers with multibit registers

in a multibit component. This flow is placement aware, meaning that the tool uses physical location information to help decide which single-bit registers to map to multibit registers.

1. Create the multibit components explicitly by using the `create_multibit` command or implicitly by using the `hdlin_infer_multibit` variable.
2. Disable multibit mapping during the `compile_ultra` run by running the `set_multibit_options` command with the `-mode none` option before running the `compile_ultra` command.

   This step is required only if you are running `compile_ultra` before performing placement aware multibit banking.

   (Optional) You can preserve the multibit components in ASCII format by using the `write_multibit_components` command. In this case, source the multibit component information in a subsequent Design Compiler session and run the `identify_register_banks` command as shown in Figure 9.
3. Perform placement-aware multibit mapping of bused registers using the `identify_register_banks` command with the `-multibit_components_only` option.

   Single-bit registers in a multibit component are banked only if they are physically near each other. Cells are not banked across different multibit components.

In this flow, you cannot specify the input map file and register group file guidance files.

Figure 8 shows the single session flow (running the `identify_register_banks` command in the same session rather than in a subsequent Design Compiler session).

Figure 8 Single Session Flow

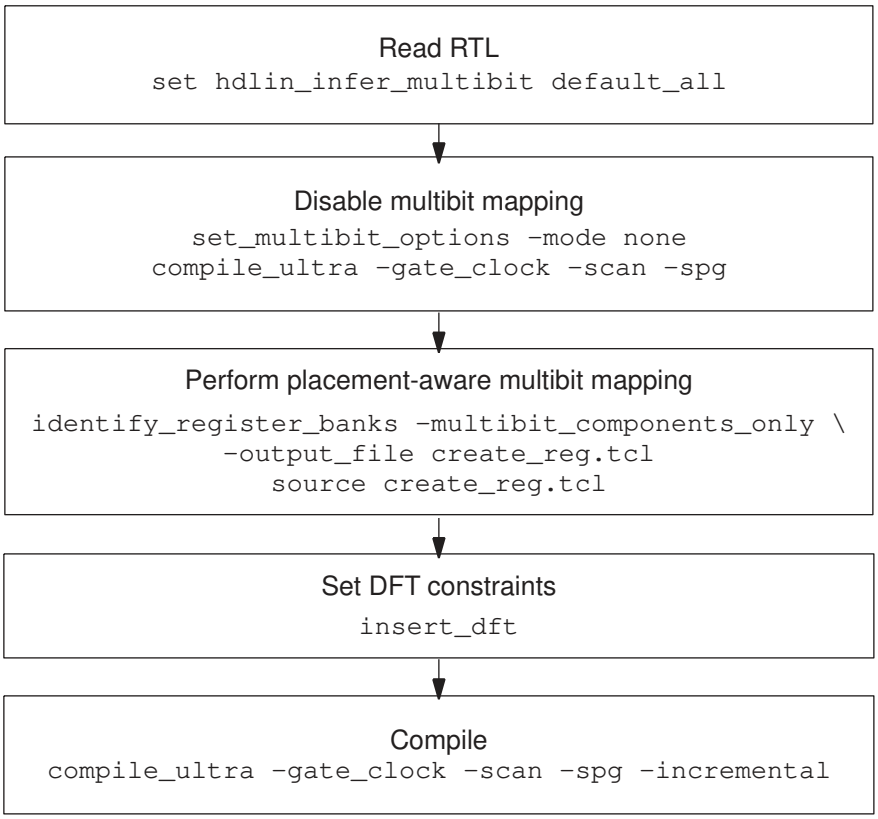
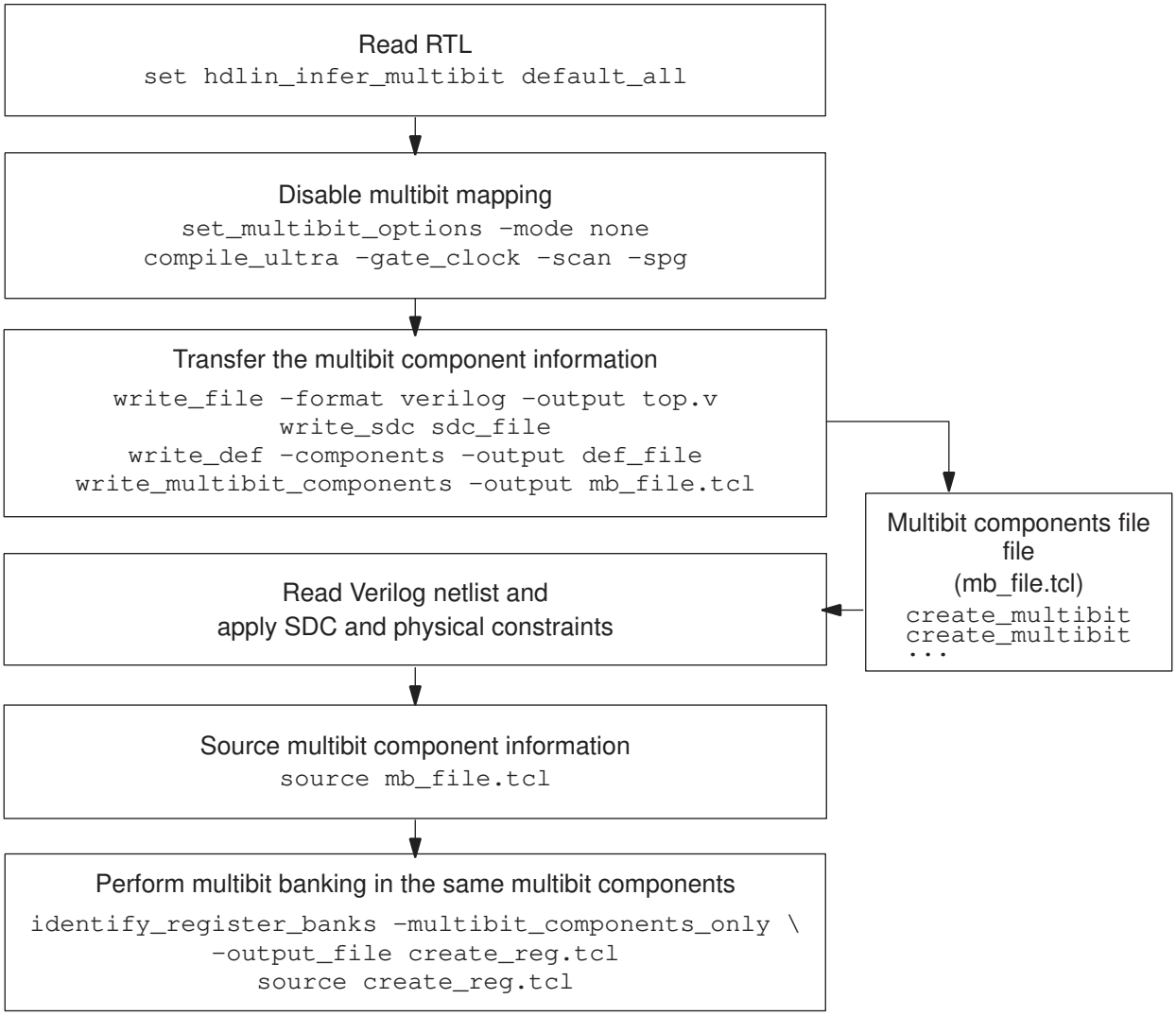


Figure 9 shows an alternative flow where you preserve the multibit components in ASCII format and then run the `identify_register_banks` command in a subsequent Design Compiler session.

Feedback

Figure 9 Two Session Flow With an ASCII Handoff



See Also

- Enhanced Placement-Aware Multibit Register Banking

Multibit Register Naming Style

In the Design Compiler Graphical placement-aware multibit flows, the `banking_enable_concatenate_name` variable controls the name of the multibit registers created by the banking script file. When the variable is set to `true` (the default), the

tool uses the following naming style for multibit cells: The name of the original single-bit registers concatenated with an underscore (_), such as reg_1_reg_0.

The following example shows the `create_register_bank` command in the banking script file when the `banking_enable_concatenate_name` variable is set to `true`. In this example, the tool is grouping two single-bit registers, reg_1 and reg_0. The tool specifies the new multibit register name as reg_1_reg_0 using the `-name` option. When you execute the script, the tool creates a multibit register named reg_1_reg_0:

```
create_register_bank -name reg_1_reg_0 {reg_1 reg_0} \
-lib_cell multibit_lib/REG_2_BIT
```

You can add a prefix to the multibit cell name by using the `-name_prefix` option when you run the `identify_register_banks` command. If you specify an MBIT prefix with the `-name_prefix` option, the tool specifies the new multibit register name as MBIT_reg_1_reg0, as shown in the following banking script:

```
create_register_bank -name MBIT_reg_1_reg0 {reg_1 reg_0} -lib_cell \
 multibit_lib/REG_2_BIT
```

To revert to the naming style used before the Design Compiler J-2014.09-SP3 release, set the `banking_enable_concatenate_name` variable to `false`.

Preventing Updating of Attributes during Multibit Banking and Debanking

You can prevent updating of the attributes in the netlist file for each use of the `create_register_bank` and `split_register_bank` commands.

To prevent updating of the attributes in the netlist file, set the `banking_enable_attribute_spreading` variable to `false`. The default is `true`. The feature is available only in the Design Compiler Graphical tool.

You can preferably use the variable when you see a long runtime with the `create_register_bank` or `split_register_bank` command, which is used multiple times in certain corner situations.

Enhanced Placement-Aware Multibit Register Banking

In Design Compiler NXT in topographical mode, you can enable enhanced placement-aware multibit banking as follows:

- Within the `compile_ultra` command

  This helps to improve runtime as you can avoid performing additional incremental compile after multibit banking.

Feedback

Set the `compile_enable_physical_multibit_banking` variable as follows before using the `compile_ultra` command:

```
set_app_var compile_enable_physical_multibit_banking true
```

Example 1 Flow Within compile_ultra

```
# read RTL

set_multibit_options -exclude exclude_list -stage physical \
      -mode mode_name

set_app_var compile_enable_physical_multibit_banking true
compile_ultra -scan -gate_clock -spg

# perform DFT setup
# apply multibit debanking settings

compile_ultra -incremental -spg
```

- With the `identify_register_banks` command

  Set the `enable_enhanced_physical_multibit_banking` variable as follows before using the `identify_register_banks` command:

```
set_app_var enable_enhanced_physical_multibit_banking true
```

Example 2 Flow With the identify_register_banks Command

```
# read RTL

compile_ultra -scan -gate_clock -spg

set_app_var enable_enhanced_physical_multibit_banking true

identify_register_banks -exclude exclude_list
 -output_file $TOP.snps_mbit_map.tcl
source $TOP.snps_mbit_map.tcl

# perform DFT setup
# apply multibit debanking settings

compile_ultra -incremental -spg
```

Feedback

In Placement-Aware Multibit Register Banking in Design Compiler Graphical with the `identify_register_banks` command (Figure 9), the following steps are performed:

1. The tool writes the Tcl file that includes many `create_register_bank` commands.
2. You can modify the Tcl file if needed and source the file to perform multibit banking.

In enhanced placement-aware multibit banking, the tool performs both the steps at once and the generated Tcl file includes the following comment:

```
# The netlist transformations were applied by identify_register_banks
```

Use the following options of the `set_multibit_options` command with enhanced placement-aware multibit banking:

- `-stage rtl | physical | banking_all | debanking | none` to distinguish between RTL and physical banking: The default is `rtl`.
- `-exclude` to exclude a specified list of cells from banking.
- `-multibit_components_only` to bank bus registers only.
- `-slack_threshold` to specify a slack threshold in percentage to determine which registers to bank and debank.
- `-name_prefix` to add a prefix to the name of the banked register.

  Note:

  In this flow, the names of multibit registers are generated by using the same convention as in the RTL inference flow. For information, see Multibit Register Naming in the RTL Inference Flow.

- `-mode` *`mode_name`* to specify one of the following multibit optimization modes:
  - `non_timing_driven` (the default): The tool uses multibit cells whenever it can, resulting in the fewest possible remaining single-bit cells and providing maximum banking ratio.
  - `timing_friendly`: The tool replaces single-bit cells with multibit cells after delay optimization, resulting in better timing than the `non_timing_driven` mode. However, timing might degrade during banking of multibit cells.
  - `timing_driven`: The tool replaces single-bit cells with multibit cells after delay optimization, similar to the `timing_friendly` mode. However, the tool rejects

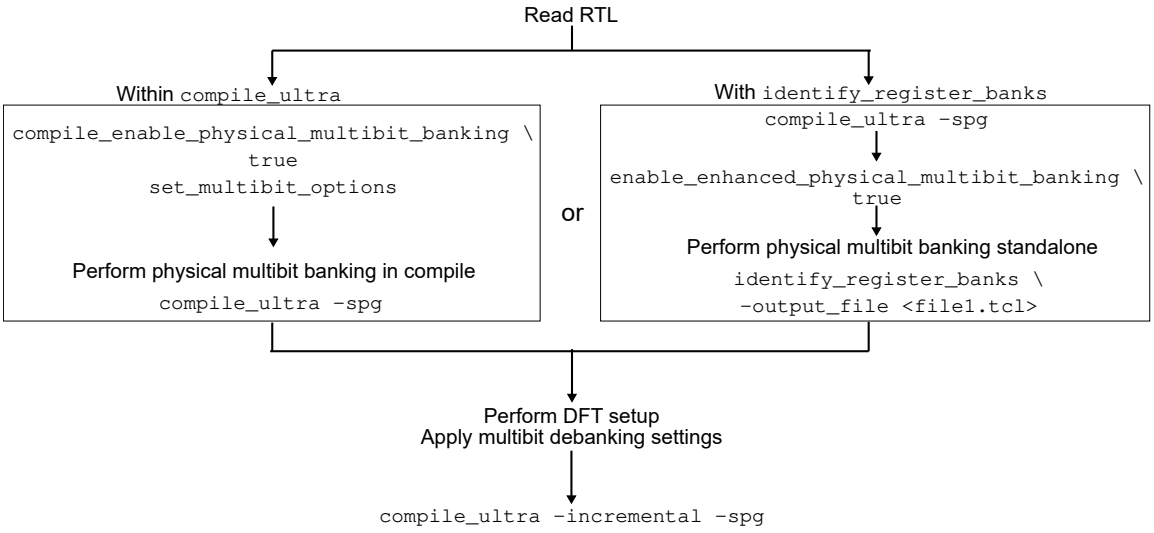
Feedback

banking if timing degrades and also performs debanking during delay optimization. This mode gives the best timing QoR but might impact banking-ratio significantly.

- `none`: The tool does not replace single-bit cells with multibit cells during compile, meaning that no multibit optimization takes place when you run the `compile_ultra` command.

For information about using this option in the RTL bus inference flow, see Multibit Synthesis Optimization.

Figure 10 Enhanced Placement-Aware Multibit Register Banking Flow Within compile_ultra or With identify_register_banks



See Also

- Banking Multibit Components Using the identify_register_banks Command
- Using Sequential Mapping to Map Single-Bit Registers
- Handling Timing Exceptions With Multibit Banking

Using Sequential Mapping to Map Single-Bit Registers

RTL banking and physical banking perform banking by replacing single-bit registers with multibit registers in the Design Compiler tool. Therefore, library cells selected for mapping single-bit registers also impact banking of multibit registers. During the first compile, if the selected library cell for single-bit register mapping does not have an equivalent multibit register in the library, the tool cannot bank the single-bit register to a multibit register.

Feedback

To enable sequential mapping during the first compile for mapping as many registers in the library to the single-bit registers that have functionally equivalent multibit registers, set the `seqmap_prefer_registers_with_multibit_equivalent` variable to `true` during the first compile. This helps to get a better banking ratio but might affect the quality of results (QoR).

For example, consider that the library has three single-bit cells (SB1, SB2, and SB3) and a 2-bit cell (MB). SB1 is equivalent to MB. The following figure shows the default mapping and mapping using the `seqmap_prefer_registers_with_multibit_equivalent` variable.

The default mapping	Mapping using `seqmap_prefer_registers_with_multibit_equivalent`
out_reg[0] → SB1 out_reg[1] → SB1 out_reg[2] → SB2 out_reg[3] → SB3 out_reg[4] → SB2	out_reg[0] → SB1 out_reg[1] → SB1 out_reg[2] → SB1 out_reg[3] → SB1 out_reg[4] → SB2
The tool can replace only SB1 to MB as follows: - out_reg[1:0] to MB, replaced with multibit register - out_reg[2] to SB2, remains as single-bit register - out_reg[3] to SB3, remains as single-bit register - out_reg[4] to SB2, remains as single-bit register	The tool can replace only SB1 to MB as follows: - out_reg[1:0] to MB, replaced with multibit register - out_reg[3:2] to MB, replace with multibit register - out_reg[4] to SB2, remains as single-bit register

See Also

- RTL Bus Inference Flow
- Placement-Aware Multibit Register Banking in Design Compiler Graphical
- Enhanced Placement-Aware Multibit Register Banking

Handling Timing Exceptions With Multibit Banking

To handle timing exceptions registers during multibit banking, see the following topics:

- Default Behavior
- Excluding Timing Exception Registers From Banking

Feedback

See Also

- RTL Bus Inference Flow
- Placement-Aware Multibit Register Banking in Design Compiler Graphical
- Enhanced Placement-Aware Multibit Register Banking

Default Behavior

The default behavior of the tool is to bank registers with exceptions. The tool banks registers based on whether the exception is set on shared or unshared pins, as follows:

- If the exception is set on unshared pins like D or Q, the tool can bank and transfer the exception.
- If the exception is set on shared pins like CLK or RST, the tool banks only if the same exception exists on all candidate of single bit register pins.

Excluding Timing Exception Registers From Banking

You can also use the following options of the `set_multibit_options` command to exclude registers with timing exceptions from banking in the Design Compiler tool:

- `-exclude_registers_with_timing_exceptions`: Excludes registers with timing exceptions from banking. The default is `false` to include all registers for banking. When the option is set to `true`, the tool excludes registers from banking with the `group_path`, `set_multicycle_path`, `set_false_path`, `set_max_delay`, and `set_min_delay` timing exceptions.
- `-ignore_timing_exception`: Ignores registers with the specified exceptions when excluding registers with exceptions. The default is `none` to exclude registers with all timing exceptions. The option supports the `group_path`, `path_group`, `false_path`, `multicycle_path`, `min_delay`, and `max_delay` timing exceptions.

  The `-exclude_registers_with_timing_exceptions` option must be set to `true` to use this option. For example, if you want registers with any exception other than `group_path` to be excluded from banking, use the command as follows:

  ```
  dc_shell> set_multibit_options -mode non_timing_driven \
  -exclude_registers_with_timing_exceptions true \
  -ignore_timing_exception GROUP_PATH
  ```

  When you set the `set_multibit_option -exclude_registers_with_timing_exceptions` command to `true`, the tool does not bank registers with timing exception on any pin whether it is compatible or incompatible with the shared or unshared pins.

Feedback

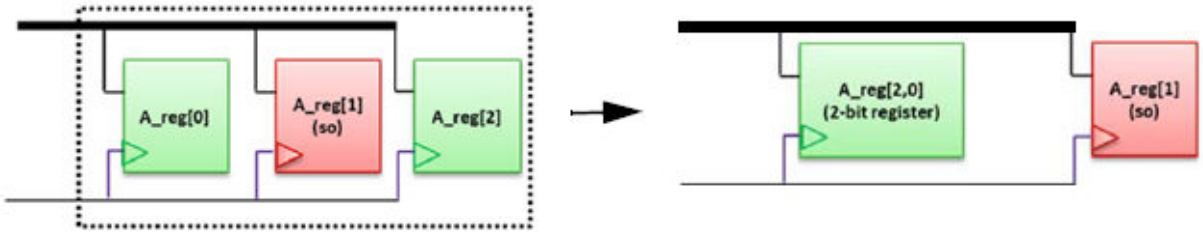
Note:

Make sure that the specified list of exceptions is valid for the register to be excluded correctly by the `-exclude_registers_with_timing_exceptions` option.

For example, `set_multicycle_path -through A_reg/Q` is a valid exception. However, `set_multicycle_path -to A_reg/Q` is not a valid exception, and the `-exclude_registers_with_timing_exceptions` option cannot exclude the A_reg/Q register from banking.

To report invalid exceptions set on a register in a design, use the `report_timing_requirements -ignored` command.

The Design Compiler and DC Explorer tools can also map the noncontiguous bits of a multibit component to a multibit register. In the following example, the noncontiguous A_reg[0] and A_reg[2] registers are packed into a two-bit register during multibit mapping. The A_reg[1] register is marked with a `size_only` attribute and therefore is preserved as a single-bit register.



Debanking Registers in Design Compiler Graphical

You can use the following methods for debanking in the Design Compile Graphical tool:

- Using the split_register_bank Command for Non-Scan Stitched Designs
- Using Incremental Compile for Scan Stitched and Non-Scan Stitched Designs

Using the split_register_bank Command for Non-Scan Stitched Designs

To split a multibit register into smaller register cells (having fewer bits), including single-bit registers, use the `split_register_bank` command:

```
split_register_bank bank_name
  -lib_cells {library_name/cell_name}
```

Feedback

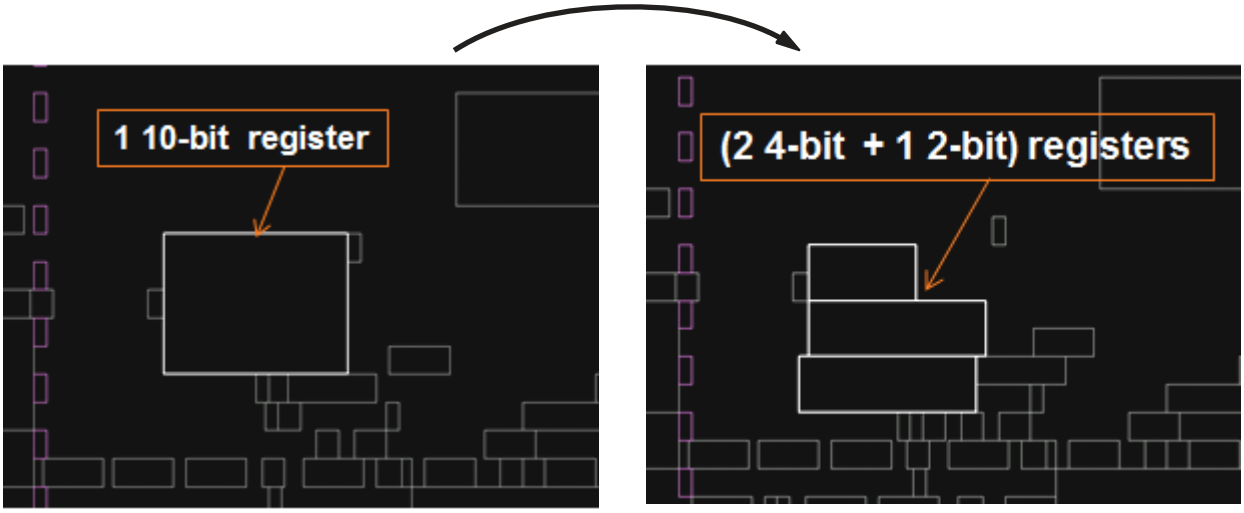
Specifying the library name in the `-lib_cell` argument is mandatory in the Design Compiler tool .

The following example splits a 10-bit register into two 4-bit registers and one 2-bit register:

```
prompt> split_register_bank my_regA \
   -lib_cells {mylib/reg_4bit mylib/reg_4bit mylib/reg_2bit}
```

The command preserves the order of the bits according to the order of the specified list of library cells. In this example, it replaces the first eight bits of the 10-bit register with two instances of the reg_4bit library cell and replaces the remaining two bits with an instance of the reg_2bit library cell. It breaks the connections to the removed 10-bit cell and appropriately reconnects the new 4-bit and 2-bit cells.

Figure 11 Multibit Splitting



When the `split_register_bank` command splits the multibit cell into single-bit cells, it uses the original single-bit cell name that is stored in the `register_list` attribute that is set on the multibit cell.

For example, the following command splits a 4-bit register into four single-bit registers:

```
prompt> split_register_bank my_regA \
 -lib_cells {mylib/reg_1bit mylib/reg_1bit mylib/reg_1bit mylib/reg_1bit}
```

If the multibit register in this example has a `register_list` attribute value of {reg_3 reg_2 reg_1 reg_0}, the command creates four single-bit registers with the following register names: reg3, reg_2, reg_1, reg_0

If there is no `register_list` attribute set on the multibit register, the command creates four single-bit registers with the following register names:

reg_3_reg_2_reg_1_reg_0_bank[0], reg_3_reg_2_reg_1_reg_0_bank[1], reg_3_reg_2_reg_1_reg_0_bank[2], reg_3_reg_2_reg_1_reg_0_bank[3]

The value of the `register_list` attribute can be used only for single-bit registers. For example, the following command splits a 4-bit register into one 2-bit register and two single-bit registers:

```
prompt> split_register_bank my_regA \
 -lib_cells {mylib/reg_2bit mylib/reg_1bit mylib/reg_1bit}
```

In this case, the command creates one 2-bit register and two single-bit registers with the following register names:

reg_3_reg_2_reg_1_reg_0_bank[0:1], reg_1, reg_0

In the ASCII flow, if you want to restore the `register_list` attribute, you can use the `set_attribute` command to reapply the attribute, as follows:

```
prompt> set_attribute [get_cells multibit_cell] \
        register_list {reg_3 reg_2 reg_1 reg_0}
```

The tool uses the `register_list` attribute as follows:

- If the A_reg and B_reg registers get banked to a multibit register, the registers are renamed to A_reg_B_reg, and the `register_list` attribute is applied to the renamed A_reg B_reg register.
- If the renamed A_reg_B_reg register is debanked, the register becomes two single-bit registers, and the `register_list` attribute restores the original register names, A_reg and B_reg respectively. However, if the `register_list` attribute is not applied on the multibit register (A_reg_B_reg), the two single-bit registers are renamed to A_reg_B_reg[0] and A_reg_B_reg[1].

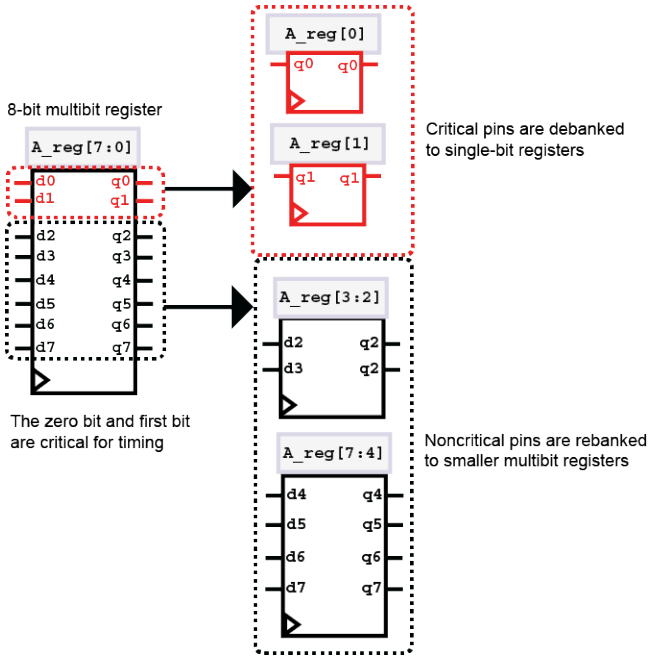
Using Incremental Compile for Scan Stitched and Non-Scan Stitched Designs

Replacing single-bit registers with multibit registers can sometimes impact the timing QoR of a design. Conversely, debanking multibit registers in the critical path to single-bit registers can improve timing QoR.

The Design Compiler Graphical tool can identify critical bits in a multibit register during incremental compile, debank the critical bits to single-bit registers, and rebank the noncritical bits to smaller multibit registers as shown in Figure 12. The tool debanks the critical bits only if it improves timing on the path.

Feedback

Figure 12 Debanking Critical Bits to Single-Bit Registers



The Design Compiler Graphical tool considers user-instantiated multibit registers, tool-inferred multibit registers, and multibit registers along infeasible paths for debanking. However, multibit registers with optimization restrictions, including `dont_touch` and `size_only` attributes, are excluded from debanking.

Constraints on the multibit registers are transferred to the corresponding debanked single-bit registers.

To enable the timing-aware multibit optimization flow, use the `-critical_range` and `-path_groups` options with the `set_multibit_options` command.

Specify a value for the `-critical_range` option using the following guidelines:

- 0.0—Debanks multibit registers on the most critical paths only.
- Nonzero integer—Debanks multibit registers on the near-critical paths within that value of the worst violator.
- `default`—The tool automatically determines a critical range value and debanks multibit registers near the critical path.

Specify a value for the `-path_groups` option using the following guidelines:

- If you specify the option, the tool debanks multibit registers belonging to a path group list only. Other path groups are ignored.
- If you do not specify the option, the tool considers all path groups for debanking.

Feedback

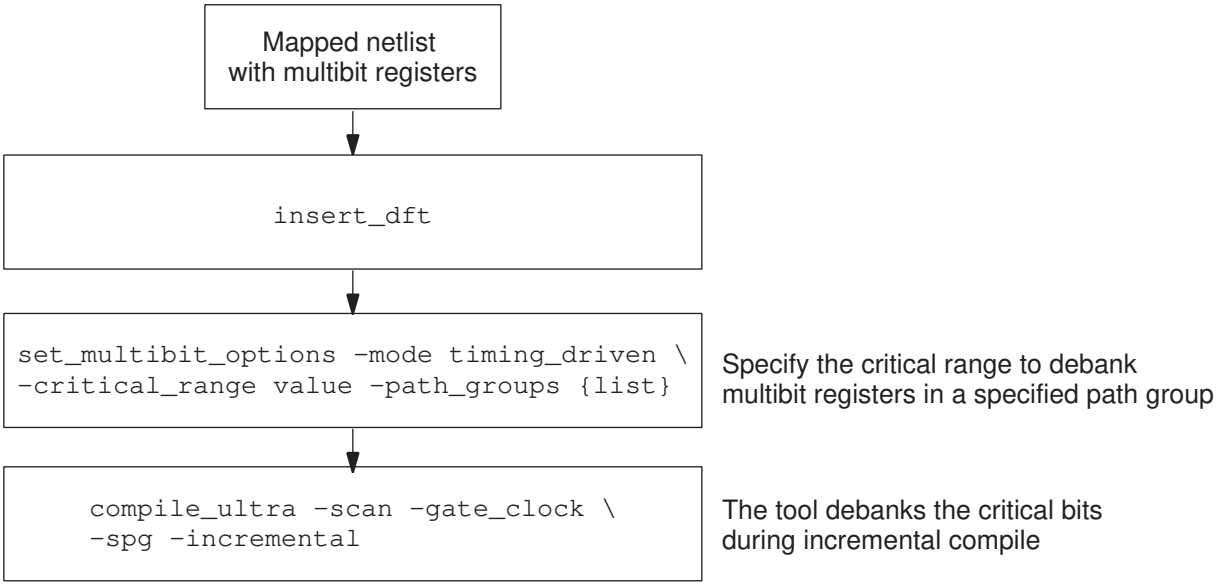
For example, the following command debanks multibit registers within a critical range of 2.0 for path group CLK:

```
prompt> set_multibit_options -mode timing_driven -critical_range 2.0 \
     -path_groups {CLK}
```

If the worst negative slack of this path group is -3.0, the tool debanks the multibit registers on the sub-critical paths with a slack ranging from -3.0 to -1.0 during incremental compile.

Figure 13 shows the timing-aware multibit optimization flow.

Figure 13 Timing-Aware Multibit Optimization Flow



The debanked bits are restored to the original register names based on the `register_list` attribute. If you're using an ASCII flow, it is recommended that you reapply the attribute:

```
prompt> set_attribute [get_cells reg_1_reg_0] register_list {reg_1 reg_0}
```

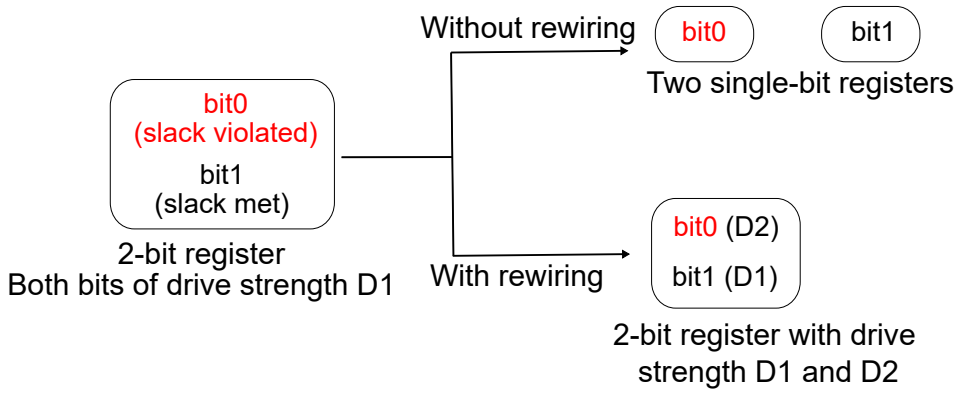
The flow supports both scan stitched and non-scan stitched designs. The debanked and rebanked registers become part of the same scan chain. If the multibit register has internal scan chains,

- The noncritical bits are not rebanked to internal scan chain multibit registers.
- The multibit register can be rebanked to an external scan chain multibit register if it is available in the library.

Feedback

Power-Aware Mixed Driving Strength Multibit Optimization

You can use mixed-drive-strength multibit library cells in which one or more bits of the multibit library cell has higher drive strength compared to the other bits in the same multibit library cell. For example, when one of the bits in the multibit library cell violates timing in a design, the Design Compiler NXT tool can use the mixed-drive-strength multibit library cell and rewire the critical bits with the bits of higher drive strength instead of debanking the multibit library cell, as shown in the following figure:



Using the mixed-drive-strength multibit library cell helps to preserve the banking ratio, resulting in improved power with the same or better timing QoR when compared to debanking the multibit cells to a single-bit or smaller multibit registers.

As shown in the figure, the Design Compiler NXT tool either

- Debanks or upsizes the cell to improve timing if a 2-bit register has one bit that violates timing

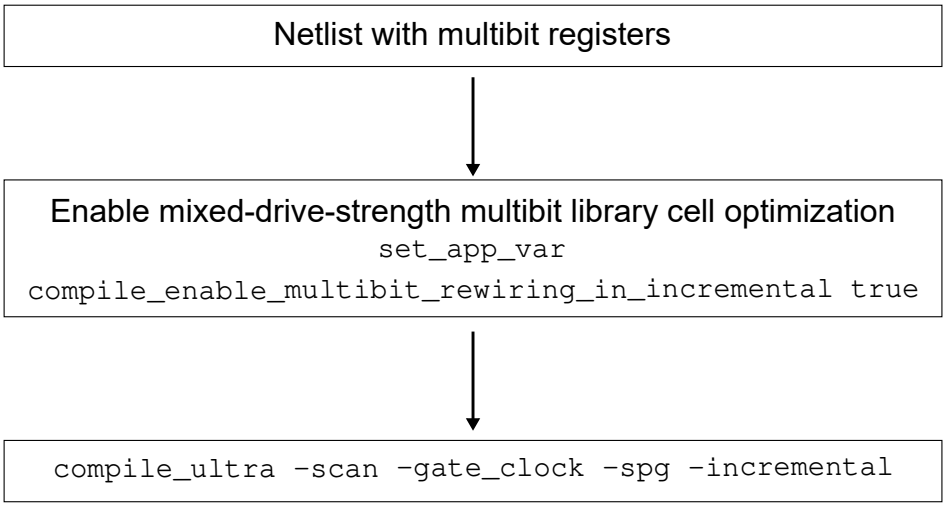
  or

- Rewires the violating path through the bit with higher drive strength if a mixed-drive-strength multibit library cell is available

To enable rewiring of mixed-drive-strength multibit cells during incremental compile and with the `optimize_netlist -area` command, use the `compile_enable_multibit_rewiring_in_incremental` variable in the Design Compiler NXT tool.

Figure 14 shows how to enable mixed-drive-strength multibit library cell optimization during incremental compile.

Figure 14 Enabling Mixed-Drive-Strength Multibit Library Cell Optimization



Placement-Aware Multibit Register Banking and Debanking in IC Compiler II

To group single-bit register cells and implement the register bank in the placement-aware register banking flow using the IC Compiler II tool, see the following topics:

- Performing Integrated Multibit Register Optimization
- Performing Multibit Register Optimization Using Discrete Commands
- Banking Multibit Retention Registers

Performing Integrated Multibit Register Optimization

To perform multibit register optimization during the `place_opt` command,

1. Enable multibit register banking by setting the `place_opt.flow.enable_multibit_banking` application option to `true`.
2. (Optional) Enable multibit register debanking by setting the `place_opt.flow.enable_multibit_debanking` application option to `true`.
3. (Optional) Enable banking of registers driven by equivalent integrated-clock-gating (ICG) cells by setting the `multibit.banking.across_equivalent_icg` application option to `true`.

   By default, the tool only banks registers that are driven by the same clock net. When you enable this feature, the tool banks registers driven by equivalent clock-gating cells

and merges the clock-gating cells. However, when doing so, it might lose nondefault routing rules applied to the nets connected to the clock-gating cells. Therefore, check and reapply these nondefault routing rules after optimization.

4. (Optional) Enable multibit register rewiring by setting the `place_opt.flow.enable_multibit_rewiring` application option to `true`.
5. (Optional) Specify settings for multibit register banking by using the `set_multibit_options` command.

   For example, the following command excludes the reg[1] cell from multibit optimization:

   ```
   prompt> set_multibit_options -exclude [get_cells "reg[1]"]
   ```

6. (Optional) Control multibit register banking and debanking as shown in the following table.

Table 2 Application Options for Controlling Multibit Register Banking and Debanking

To do this	Set this application option to true
Ignore scan flip-flops that are not included in the SCANDEF information during banking	`multibit.banking.enable_strict_scan_check-`
Consider multibit register library cells with multiple scan-in pins and internal scan-in pins during banking	`multibit.banking.lcs_consider_multi_si_cells`
Ignore user-specified size-only cells during banking and debanking	`multibit.common.ignore_sizeonly_cells`

7. Perform placement and optimization by using the `place_opt` command.
8. Report multibit components by using the `report_multibit` command.

Performing Multibit Register Optimization Using Discrete Commands

You can perform multibit banking using the following discrete steps:

1. Perform initial placement by using the `create_placement` command.
2. (Optional) Specify settings for multibit register banking by using the `set_multibit_options` command.
3. Identify the groups of cells that can be replaced by multibit cells by using the `identify_multibit` command, as described in Identifying Multibit Banks.
4. Optimize the design by using the `place_opt -from initial_drc` command.

5. (Optional) Report multibit banking information by using the `report_multibit` command.
6. (Optional) If multibit banking does not improve QoR, split specific multibit banks into individual registers or smaller multibit banks by using the `split_multibit` command, as described in Splitting Multibit Banks.

Identifying Multibit Banks

To identify groups of registers, isolation cells, or level-shifter cells that can be replaced by multibit cells, use the `identify_multibit` command.

When you use the `identify_multibit` command, you must specify:

- How to apply the multibit banking information by using one of the following methods:
  - To have the tool apply the banking information to the design, use the `-apply` option.
  - To manually apply the banking information, generate an output file containing `create_multibit` commands by using the `-output_file` option.

    You can view the output, edit it if necessary, and apply it to the design by using the `source` command.
- The type of cells you want to bank by using one of the following methods:
  - To identify groups of registers that can be banked, use the `-register` option.
  - To identify groups of isolation or level-shifter cells that can be banked, use the `-mv_cell` option.

When banking registers with the `-register` option, optionally, you can

- Control the mapping of single bits to multibit banks by specifying an input map file using the `-input_map_file` option. This file contains mapping information about which multibit register bank replaces which single-bit registers, as shown in the following example:

```
reg_group_1 {REGX1 REGX2 REGX4}
2 {1 MREG2}
4 {1 MREG4}
6 {1 MREG2}{1 MREG4}
```

This map file contains a functional group named reg_group_1, which specifies how single-bit registers of reference cell types REGX1, REGX2, and REGX4 can be grouped to form multibit registers as follows:

- The first line specifies that two single-bit registers can be combined to form a register bank consisting of one MREG2 reference cell.
- The second line specifies that four single-bit registers can be combined to form a register bank consisting of one MREG4 reference cell.
- The third line specifies that six single-bit registers can be combined to form a register bank consisting of one MREG2 and one MREG4 reference cells.

- Exclude registers based on their slack by specifying a slack threshold with the `-slack_threshold` option

  Registers with a slack less than the specified threshold are ignored. By default, the tool considers all registers for banking.

In addition, you can optionally control the cells that are banked by specifying:

- A list of instance to consider for banking by using the `-cells` option.

  When you use this option, the tool does not consider any other cells for banking.
- A list of instances to exclude by using the `-exclude_instance` option.
- A list of library cells to exclude all instances by using the `-exclude_library_cells` option.

By default, the `identify_multibit` command performs DFT optimization, which repartitions scan chains such that registers driven by the same clock driver or gating cell are put in the same scan chain. This increases the number of single-bit registers that can be combined to form multibit register banks. If you do not want the tool to repartition the scan chains, use the `-no_dft_opt` option with the `identify_multibit` command.

Splitting Multibit Banks

To improve local congestion or path slack, you can split a multibit bank into smaller multibit banks or single-bit cells by using the `split_multibit` command. By default, the command splits all multibit cells on timing paths with negative slack.

To split

- A specific cell, specify the cell instance name.

  You can specify the reference names of the smaller cells to use after splitting by using the `-lib_cells` option.

The following example splits the multibit cell instance named reg_gr0, which is a four-bit register bank, into two two-bit multibit banks:

```
prompt> split_multibit reg_gr0 -lib_cells {lib1/MREG2 lib1/MREG2}
```

- Multibit cells on paths with slack less than a specific value, use the `-slack_threshold` option.

  The following example splits the multibit cells on paths with a slack worse than -1:

  ```
  prompt> split_multibit -slack_threshold -1
  ```

- Multibit cells in a specific timing path group, use the `-path_groups` option.

  When you use this option, the tool splits the multibit cells on the pats with negative slack in the specified path group. You can specify a slack threshold limit for the path group by using the `-slack_threshold` option with the `-path_groups` option.

  The following example splits the multibit cells on paths with a slack worse than -0.5 in the timing path group CLKA:

  ```
  prompt> split_multibit -path_groups CLKA \
     -slack_threshold -0.5
  ```

You can further control splitting as follows:

- Exclude specific cells by using the `-exclude_instance` option
- Consider only specific cells by using the -`cells` option

The `split_multibit` command only splits multibit cells that were created by the `create_multibit` or `identify_multibit` command.

Banking Multibit Retention Registers

The tool can combine single-bit retention registers and create a multibit retention register if

- The single-bit retention registers are associated with the same retention strategy
- The retention pins of the single-bit retention registers have the same driver
- The cell library has a matching multibit retention register
- The library cell for the multibit retention register is specified as a library cell for the retention strategy by using the `map_retention_cell -lib_cells` command

Feedback

The tool can split a multibit retention register if

- The cell library has matching single-bit retention registers
- The library cells for the single-bit retention registers are specified as library cells to use for the retention strategy by using the `map_retention_cell -lib_cells` command

Note:

If you are using the `identify_multibit` command, to identify multibit retention registers, use the `-register` option, and not the `-mv_cell` option.

Reporting Multibit Registers in Your Design

The following topics explain how to report multibit registers with the following tools:

- Using Design Compiler Graphical and DC Explorer
- Using IC Compiler II

Using Design Compiler Graphical and DC Explorer

You can generate a report that includes all multibit registers in a design and the banking ratio by using the `report_multibit_banking` command in the Design Compiler Graphical and DC Explorer tools. For example,

```
prompt> report_multibit_banking -hierarchical
Total number of sequential cells:                                  8
    (a) Number of single-bit flip-flops:                           2
    (b) Number of single-bit latches:                              0
    (c) Number of multi-bit flip-flops:                            6
    (d) Number of multi-bit latches:                               0

Total number of single-bit equivalent sequential cells:           64
    (A) Single-bit flip-flops:                                     2
    (B) Single-bit latches:                                        0
    (C) Multi-bit flip-flops:                                     62
    (D) Multi-bit latches:                                         0

Sequential cells banking ratio ((C + D) / (A + B + C + D)):    96.88%
Flip-Flop cells banking ratio  ((C) / (A + C)):                96.88%

Sequential bits per cell ((A + B + C + D) / (a + b + c + d)):   8.00
Flip-Flop bits per cell  ((A + C) / (a + c)):                   8.00

Multi-bit Register Decomposition:
-------------------------------------------------------------------
Bit-Width Reference       Number of instances   Single-bit Equivalent
-------------------------------------------------------------------
  2-bits                        1 ( 12.50%)             2 (  3.12%)
```

```
          MB2_lib                 1
  4-bits                          1 ( 12.50%)              4 (  6.25%)
          MB4_lib                 1
  8-bits                          1 ( 12.50%)              8 ( 12.50%)
          MB8_lib                 1
16-bits                           3 ( 37.50%)             48 ( 75.00%)
          MB16_lib                3
-------------------------------------------------------------------------
Hierarchical multi-bit distribution:
-------------------------------------------------------------------------
Hierarchical cell            Num. of MB   Num. of      Seq. cells
                             seq. cells   seq. cells   banking ratio
-------------------------------------------------------------------------
top                                 6            8         96.88%
b                                   6            8         96.88%
-------------------------------------------------------------------------
1
```

See Also

- Multibit Flows
- Reasons for Not Banking

Feedback

Using IC Compiler II

You can generate a report that includes all multibit registers in a design and the banking ratio by using the `report_multibit -hierarchical` command in the IC Compiler II tool.

```
icc2_shell> report_multibit -hierarchical
Total number of sequential cells                   4
   Number of single-bit flip-flops:                0
   Number of single-bit latches:                   0
   Number of multi-bit flip-flops:                 4
   Number of multi-bit latches:                    0

Total number of single-bit equivalent cells:            12
   (A)   Single-bit flip-flops:                          0
   (B)   Single-bit latches:                             0
   (C)   Multi-bit flip-flops:                          12
   (D)   Multi-bit latches:                              0

Sequential cells banking ratio ((C+D)/(A+B+C+D)):      100.00%
Flip-flop cells banking ratio ((C)/(A+C)):             100.00%
BitsPerflop:                                           3.00
BitsPerSequentialCell:                                 3.00


--------------------------------------------------------------------------
Bit-Width  Reference        Number of instances    Single-bit Equivalent
--------------------------------------------------------------------------
2 -bit                      2 ( 50.00% )           4 ( 33.33% )

           MB2_lib          2

4 -bit                      2 ( 50.00% )           8 ( 66.67% )

           MB4_lib          2
--------------------------------------------------------------------------
--------------------------------------------------------------------------
Hierarchical Distribution of Multibit banking:
--------------------------------------------------------------------------
Hierarchical cell                          Num. of MB   Num. of   Seq. cells
                                           seq. cells   seq. cells
                                                                  banking
 ratio
--------------------------------------------------------------------------
---
top                                        4            4         100.00%
--------------------------------------------------------------------------
---
```

Reasons for Not Banking

The following topics lists reasons with examples for not banking with the following tools:

- In Design Compiler and DC Explorer
- In IC Compiler II

In Design Compiler and DC Explorer

In the Design Compiler and DC Explorer tools, the `report_multibit` command reports the reasons for not banking a specific single-bit register to a multibit register. The reasons listed in the report can help you to debug and identify the settings that impact multibit banking. These settings can be modified to improve the banking ratio. Other reporting commands, such as `report_cell` and `report_attribute`, also report reasons for not banking.

Table 3 lists the reasons for not banking a single-bit register to a multibit register and the multibit banking method or flow in which the reason can be reported. The reasons for not banking are listed in the attribute section of the report.

Feedback

Table 3 Reasons For Not Banking a Single-Bit Register to a Multibit Register

Reported attribute	Reason for not mapping to multibit	Flows in which the reason can be reported
Constraint related		
r8	Register cannot be packed to multibit due to incompatible multi-cycle path timing exception	RTL Bus Inference Flow and Placement-Aware Multibit Register Banking in Design Compiler Graphical
r9	Register cannot be packed to multibit due to incompatible false path timing exception	
r10	Register cannot be packed to multibit due to incompatible group path timing exception	
r11	Register cannot be packed to multibit due to incompatible max delay timing exception	
r12	Register cannot be packed to multibit due to incompatible min delay timing exception	
Attribute related		
r21	Register cannot be packed to multibit due to a size_only attribute	RTL Bus Inference Flow and Placement-Aware Multibit Register Banking in Design Compiler Graphical
r20	Register cannot be packed to multibit due to a dont_touch attribute	
r14	Register was explicitly excluded from multibit optimizations	
r5	Register cannot be packed to multibit due to a timing size_only mismatch	
Library related		
r3	Register cannot be packed to multibit due to a voltage threshold group mismatch	RTL Bus Inference Flow and Placement-Aware Multibit Register Banking in Design Compiler Graphical
r22	Register cannot be packed to multibit due to a lack of compatible multibit library cells	
r7	Register cannot be packed to multibit due to a DFT wrapper chain role mismatch	

Feedback

Table 3 Reasons For Not Banking a Single-Bit Register to a Multibit Register (Continued)

Reported attribute	Reason for not mapping to multibit	Flows in which the reason can be reported
r18	Register is a single-bit cell that was left over from a multibit packing solution	
Optimization related		
r4	Register cannot be packed to multibit due to a shift register role mismatch	RTL Bus Inference Flow and Placement-Aware Multibit Register Banking in Design Compiler Graphical
r6	Register cannot be packed to multibit due to a retention strategy mismatch	
r13	Register cannot be packed to multibit due to a conflicting pin connection	
r15	Register cannot be grouped with others for multibit packing	
r1	Register cannot be packed to multibit due to a functionality mismatch	
r17	Register belonged to a group whose multibit packing was rejected due to QoR	RTL Bus Inference Flow only
r23	Register cannot be packed to multibit due to being retimed	
r24	Register cannot be packed to multibit due to being a duplicated register	
r30	Register was unpacked from multibit due to QoR	Using Incremental Compile for Scan Stitched and Non-Scan Stitched Designs in Debanking by incremental compile

The tools do not report a reason for not banking to a multibit register in the following cases:

- For .ddc files written from Design Compiler versions before the N-2017.09 release
- When all the multibit cells in the library are marked with a `dont_use` attribute
- When the register was excluded as a result of an RTL `//synopsys dont_infer_multibit` directive

Feedback

- When the `set_multibit_options -mode` command is set to `none`

  With this setting, multibit mapping is disabled when you run the `compile_ultra` command

Examples

The following example shows a report provided by the `report_multibit` command:

```
prompt> report_multibit
...
Attributes:
 r8 - register couldn't be packed to multi-bit due to incompatible
 multi-cycle path timing exception.
 r21 - register couldn't be packed to multi-bit due to a size_only
 attribute.
 r22 - register couldn't be packed to multi-bit due to lack of compatible
 multi-bit library cells.

Multibit Component : A_reg
Cell                    Reference    Library    Area   Width Attributes
--------------------------------------------------------------------------
A_reg[7:6]                MBFF       my_lib     46.50    2    n, r
A_reg[5]                  SBFF       my_lib     40.43    1    n, r8
A_reg[4:3]                MBFF       my_lib     46.50    2    n, r
A_reg[2]                  SBFF       my_lib     40.43    1    n, r21
A_reg[1]                  SBFF1      my_lib     41.50    1    n, r22
A_reg[0]                  SBFF1      my_lib     41.50    1    n, r22
--------------------------------------------------------------------------
Total 6 cells                                   256.86    8
```

If you have two registers in your multibit component (or bused register) and one of them is optimized away as a constant, the `report_multibit` report displays an r15 attribute on the register that was not optimized away, as shown:

```
prompt> report_multibit -hierarchical
...
Attributes:
    b - black box (unknown)
    h - hierarchical
    n - noncombinational
    r - removable
  r15 - register couldn't be grouped with others for multi-bit packing.
    u - contains unmapped logic

Multibit Component : out_reg
Cell          Reference     Library    Area   Width  Attributes
--------------------------------------------------------------------------
out_reg[1]    MBFF           my_lib    3.28   1      n, r15
--------------------------------------------------------------------------
Total 1 cells                          3.28   1
Total 1 Multibit Components
1
```

Feedback

See Also

- Reporting Multibit Registers in Your Design
- Multibit Flows

In IC Compiler II

You can report all the cells that are ignored during banking and debanking with the `report_multibit -ignored_cells` command in the IC Compiler II tool.

```
icc2_shell> report_multibit -ignored_cells
…
Sequential cells banking ratio ((C+D)/(A+B+C+D)):     100.00%
Flip-flop cells banking ratio ((C)/(A+C)):            100.00%
BitsPerflop:                                          3.00
BitsPerSequentialCell:                                3.00
…
Reasons for sequential cells not mapping to multibit during Physical
Banking:
------------------------------------------------------------------------
Explanations:
   pb33: Cell cannot be banked to multibit because timing constraints on
   its shared pins are incompatible with other cells.(Number of cells: 1)
------------------------------------------------------------------------
   pb37: Cell cannot be mapped to multibit because it cannot be grouped
   with other cells.(Number of cells: 1)
------------------------------------------------------------------------

-------------------------------------------
Sequential Cell                        Reason
-------------------------------------------
b1/my_mb_reg_out1_reg[3:0]              pb33
b1/my_mb_reg_out_reg[3:0]               pb37
-------------------------------------------

Reasons for multibit sequential cells not getting debanked:
------------------------------------------------------------------------
Explanations:
   db39: Multibit cell cannot be debanked because it was not critical
   when debanking was performed.(Number of cells: 4)
------------------------------------------------------------------------

-------------------------------------------------
Sequential Cell                           Reason
-------------------------------------------------
b1/my_mb_reg_out1_reg[3:0]                  db39
b1/my_mb_reg_out_reg[3:0]                   db39
my_mb_reg_my_inst_MB_0_my_inst_MB_3         db39
my_mb_reg_my_inst_MB_2_my_inst_MB_1         db39
```

Feedback

```
------------------------------------------------
1
```

Multibit Flows

The following topics demonstrate the full multibit synthesis, implementation, and verification flows across Synopsys tools:

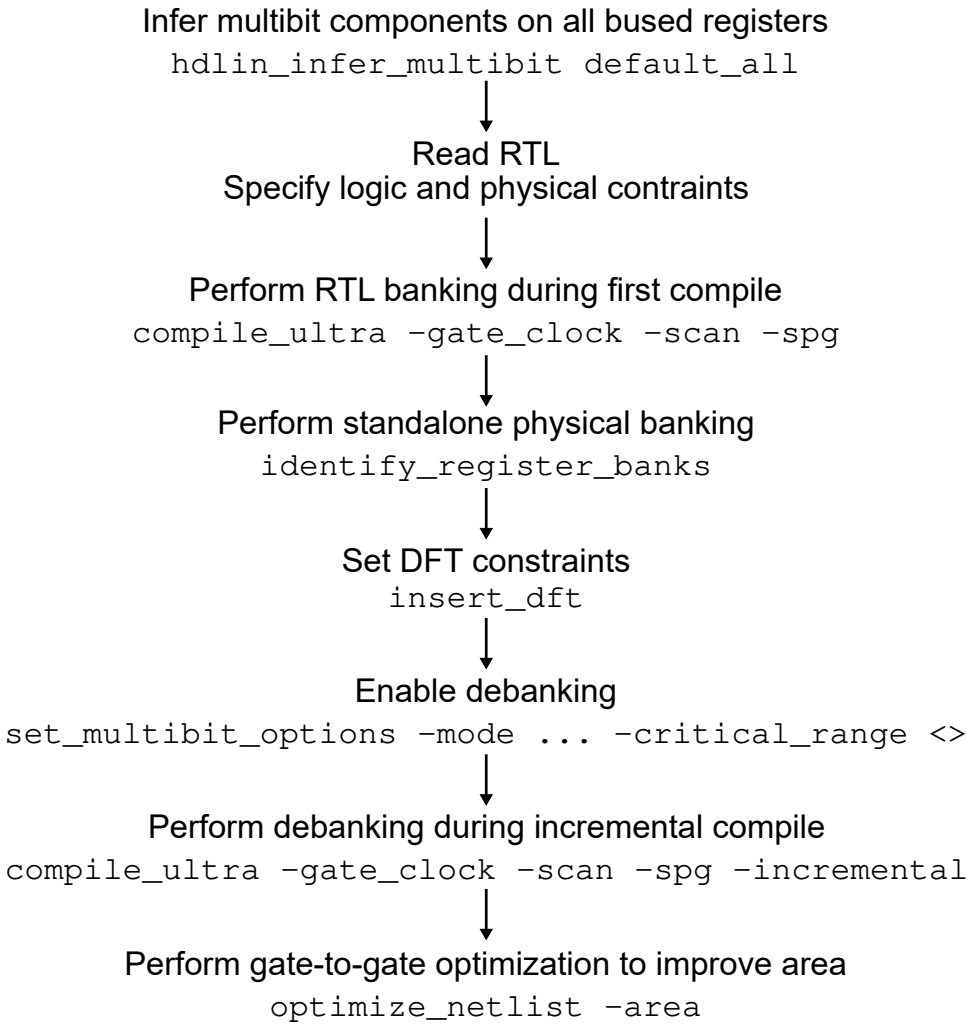
- Design Compiler Graphical
- IC Compiler II
- TestMAX DFT and Multibit Registers
- Clock Gating in the Multibit Flow
- Multibit SAIF Flow
- Multicorner-Multimode Flow
- Design Verification Flow in Formality

Design Compiler Graphical

The following table lists the flows that you can use with Design Compiler Graphical:

Title	Description	Flow
RTL bus inference flow	Multibit registers are inferred from the buses that are defined in the RTL	Figure 5
Placement-aware register banking flow	Banking cells that are physically near each other	Figure 6
Banking multibit components flow	Banking RTL buses and user-defined groups that are physically near each other	Figure 8
Enhanced placement-aware multibit register banking	Enhanced banking of cells that are physically near each other	Figure 10
Debanking timing critical multibit registers flow	Identify and debank critical bits in a multibit register	Figure 13
Power-aware mixed driving strength multibit optimization flow	Use the mixed-drive-strength multibit library cell and rewire the critical bits with the bits of higher drive strength instead of debanking	Figure 14

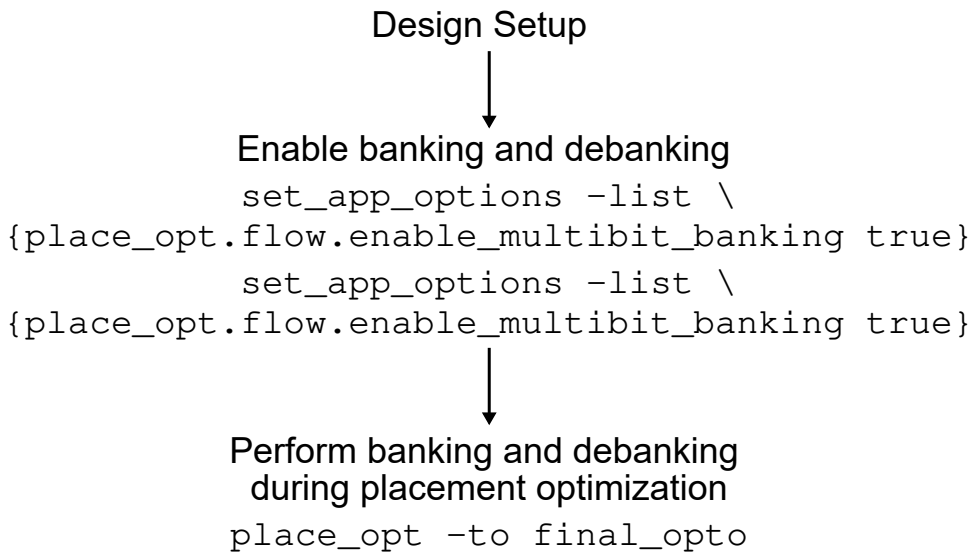
Use the following multibit flow with the Design Compiler Graphical tool:



Feedback

IC Compiler II

Use the following multibit flow use with the IC Compiler II tool.



TestMAX DFT and Multibit Registers

The TestMAX DFT tool supports design-for-test (DFT) insertion features with the `insert_dft` command on designs with multibit registers.

Scan Insertion With Multibit Registers

In the Design Compiler tool, single-bit registers must be replaced with scan cells before performing multibit mapping. This is done by using the `-scan` option with the `compile_ultra` command during the initial compile. Multibit mapping must be performed before scan insertion.

When you use TestMAX DFT commands on a design with multibit registers, ensure the following variables and commands are set appropriately:

- `set compile_seqmap_identify_shift_registers_with_synchronous_logic false`

  The default is `false`, starting from version I-2013.12-SP2.

  Leaving this variable set to `false` helps preserve multibit registers during DFT scan insertion.

Feedback

- `set_scan_configuration -preserve_multibit_segment false`

  The default is `false`, starting from version I-2013.12.

  You must have the value set to `false` for optimal scan chain balancing.

After scan insertion, run the `check_scan_def` command to ensure there are no failures in the scan chain structural checks.

Core Wrapping With Multibit Registers

You can core-wrap designs that use multibit registers. Both the RTL bus inference flow and placement-based register banking flow are supported.

In the maximized reuse flow, multibit registers associated with ports can be shared wrapper cells. For best results, enable and configure maximized reuse core wrapping *before* performing multibit banking with

- The initial `compile_ultra` command (RTL bus inference flow)
- The `identify_register_banks` command (placement-based banking flow)

This informs the multibit banking algorithms that you can perform maximized reuse core wrapping during DFT insertion. The tool builds each multibit register from single-bit registers of the same type—input registers, output registers, or core registers—so they can be stitched into the corresponding wrapper or core scan chains.

To confirm this, the tool issues the following message during multibit banking:

```
Information: DFT core wrapping client enabled; banking anticipates core
wrapping. (TEST-1291)
```

If you forget to enable and configure core wrapping *before* performing multibit banking, the tool issues the following message during multibit banking:

```
Information: DFT core wrapping client disabled; banking anticipates no
core wrapping. (TEST-1290)
```

In the simple core wrapper flow, multibit registers cannot be wrapper cells. Ports associated with multibit registers always get a dedicated wrapper cell.

For more information, see “Wrapping Cores With Multibit Registers” in the *TestMAX DFT User Guide*.

Clock Gating in the Multibit Flow

All clock gating features, such as multistage, balancing, instance-based, and global, are supported on multibit registers in both unmapped and mapped designs. For optimal clock gating and maximum multibit register usage ratios, performing clock gating during the initial compile is recommended.

Feedback

Starting with Design Compiler version J-2014.09, the fanout calculation for multibit registers with clock gating has been changed. For example, if there are 19 single-bit registers that are not gated and 16 are gated into 4 4-bit multibit registers, the fanout is 4. In previous releases, the fanout count was 16, which is the total number of gated bits even when only 4 registers were driven.

When using the `set_clock_gating_style -max_fanout` command, specify the actual load that the clock-gating cell drives. With the current fanout count, each clock-gating cell drives more multibit registers for a given value of the `max_fanout` option. The fanout count affects the `set_clock_gating_style` and `report_clock_gating` commands.

If the design was optimized using the `-gate_clock` option, the `report_clock_gating` command reports the summary of clock-gating and multibit registers. For example,

```
dc_shell> report_clock_gating
...
                    Clock Gating Summary
  ------------------------------------------------------------
  |    Number of Clock gating elements     |      151         |
  |      ...                               |                  |
  |    Total number of registers           |     3621         |
  ------------------------------------------------------------


    Clock Gating Multibit Decomposition
  +--------------------------------+------------+------------+
  |                                |   Actual   | Single-bit |
  |                                |   Count    | Equivalent |
  +--------------------------------+------------+------------+
  |   Number of Gated Registers    |            |            |
  |                    1-bit       |   1015     |   1015     |
  |                    2-bit       |   2203     |   4406     |
  |                  Total         |   3218     |   5421     |
  |   Number of Ungated Registers  |            |            |
  |                    1-bit       |    163     |    163     |
  |                    2-bit       |    240     |    480     |
  |                  Total         |    403     |    643     |
  |   Total Number of Registers    |            |            |
  |                    1-bit       |   1178     |   1178     |
  |                    2-bit       |   2443     |   4886     |
  |                  Total         |   3621     |   6064     |
  +--------------------------------+------------+------------+
```

For more information regarding placement-aware clock-gating, see the *Power Compiler User Guide*.

Feedback

Multibit SAIF Flow

Power Compiler supports SAIF-based power analysis on designs with multibit registers. You can use the same SAIF flow for designs that use multibit registers and designs that do not use multibit registers.

Here is a script example of the multibit SAIF flow:

```
#Enable bus-based multibit optimization in compile_ultra
set hdlin_infer_multibit default_all

#Improve annotation
set hdlin_enable_upf_compatible_naming true

#Initialize name-mapping database
saif_map -start

#Read the RTL design
read_verilog <RTL>
source sdc

#Perform RTL SAIF annotation using the name-mapping database
read_saif -input rtl.saif -instance_name tb/top -auto_map_names

#Initial compile
compile_ultra -scan -gate_clock

#Multibit register banking
identify_register_banks
source <create_register_bank> file

#DFT scan insertion
insert_DFT

#Incremental compile
compile_ultra -incremental -gate_clock

#Write out the map file and check power consumption
report_saif -missing -hier
saif_map -write-map map.ptpx -type ptpx
report_power
```

For more information about power optimization and analysis flow using SAIF, see the *Power Compiler User Guide*.

Multicorner-Multimode Flow

When you run placement-aware multibit mapping on designs with multiple scenarios, you must activate all scenarios before you run the `create_register_bank` or

`split_register_bank` commands to transfer scenario-specific information during banking operations. To do this, perform the following steps:

1. Run the `identify_register_banks` command:

   ```
   prompt> identify_register_banks -output_file create_reg.tcl
   ```

2. Activate all scenarios by using the `set_active_scenarios` command:

   ```
   prompt> set_active_scenarios -all
   ```

   This step ensures that the tool transfers scenario-specific information during the next step.

3. Perform banking or debanking operations with the `create_register_bank` or `split_register_bank` commands:

   ```
   prompt> source create_reg.tcl
   ```

4. Reselect the active scenarios for the subsequent optimization:

   ```
   prompt> set_active_scenarios {scenario-1 secnario-2}
   ```

5. Run an incremental compile:

   ```
   prompt> compile_ultra -scan -incremental
   ```

Design Verification Flow in Formality

Whenever a Synopsys tool replaces a group of single-bit registers with a multibit register, or replaces a multibit register with single-bit registers, it records the changes in an .svf file. This file provides guidance to the Formality formal verification tool, allowing it to verify the equivalence of the single-bit and multibit replacement logic. This is true for all tools that perform multibit register banking: Design Compiler, TestMAX DFT, Power Compiler, and IC Compiler II.

Replacement of single-bit registers with multibit registers generates content for the .svf file similar to the following:

```
guide_multibit \
  -design { test } \
  -type { svfMultibitTypeBank } \
  -groups \
  { { q0_reg[0] 1 q0_reg[1] 1 q0_reg[0]_q0_reg[1] 2 } }
```

Splitting of a multibit register to single-bit registers generates content for the .svf file similar to the following:

```
guide_multibit \
  -design { test } \
  -type { svfMultibitTypeSplit } \
```

Feedback

```
-groups \
{ { q0_reg[0]_q0_reg[1] 2 q0_reg[0] 1 q0_reg[1] 1 } }
```

In the Design Compiler tool, when you enable multibit optimization (in either the RTL inference or placement-aware multibit flow), you must generate the verification guidance file using the `set_svf` command. In the Design Compiler tool, generate the verification guidance file with the `set_svf` command before reading the RTL. When you verify the design, use the .svf file generated by the Design Compiler tool.

In Design Compiler, checkpoint guidance provides a mechanism for the Formality and Design Compiler tools to synchronize using an intermediate netlist. A checkpoint guidance netlist is generated when a register that is retimed or replicated in the same dc_shell session is replaced by a multibit register using the `create_register_bank` command.

To enable checkpoint verification on designs with retiming and multibit mapping, you must perform multibit mapping by sourcing the file with the `create_register_bank` commands immediately after running the `compile_ultra` command with retiming enabled, as shown:

```
prompt> compile_ultra -scan -retime
prompt> identify_register_banks -output_file create_reg.tcl
prompt> source create_reg.tcl
```

Do not run any commands that perform design modification after running the `compile_ultra` command and before sourcing the script file with the `create_register_bank` commands.

If you run an incremental compile or any command that generates verification guidance, such as `ungroup` and `change_names`, between the `compile_ultra` and `create_register_bank` commands, you must verify the RTL synthesis using two separate steps:

1. Verify synthesis from the RTL to the netlist before multibit mapping.
2. Verify the netlist before multibit mapping to the netlist after multibit mapping.

Checkpoint guidance verification is supported for retiming performed by the following methods:

- Using the `compile_ultra` command with the `-retime` option
- Using the `set_optimize_registers` command followed by the `compile_ultra` command
- Using DesignWare pipelined components in the RTL

For more information about the checkpoint guidance flow, see SolvNetPlus article 024569, “Formality Verification Flow Using Checkpoint Guidance.”

Feedback

Note:

You must be logged in to SolvNetPlus for the link to connect directly to the article. If you are prompted to log in to SolvNetPlus upon clicking the link to the article, log in, then click the link again to reach the article.

A

Multibit Cell Modeling

Multibit synthesis and physical implementation flows require the single-bit and multibit registers to be properly defined in the logic library. The multibit characteristics are specified in Liberty format in the .lib file and compiled to .db format by Library Compiler.

This appendix describes the Liberty syntax for defining single-bit and multibit registers in the following sections:

Introduction

For multibit optimization flow to work smoothly, the library must include the single-bit version of each multibit cell. For example, if you need to have the tool infer a nonscan multibit cell, the corresponding single-bit nonscan cell must also exist in the target libraries.

In the Liberty-format description of a multibit register cell, the `ff_bank` or `latch_bank` group describes a cell that is a collection of parallel, single-bit sequential parts. Each part shares control signals with the other parts and performs the same function as the other parts. The `ff_bank` or `latch_bank` group is typically used to represent multibit registers. It can be used in `cell` and `test_cell` groups.

Note:

The only supported types of syntax inside a `test_cell` group are `ff`, `latch`, `ff_bank`, and `latch_bank` groups. It is not possible to model the functionality of a sequential element using the `statetable` Liberty syntax inside a `test_cell` group.

Feedback

For more information about library modeling of register cells, see the *Library Compiler User Guide*, available on SolvNetPlus:

- For the single-bit and multibit flip-flop definition syntax, see “Describing a Flip-Flop” and “Describing a Multibit Flip-Flop” in the chapter “Defining Sequential Cells.”
- For the single-bit and multibit latch definition syntax, see “Describing a Latch” and “Describing a Multibit Latch,” also in the chapter “Defining Sequential Cells.”
- For TestMAX DFT scan cell syntax, see “Describing a Scan Cell” and “Describing a Multibit Scan Cell,” in the chapter “Defining Test Cells.”

The following sections describe some examples of Liberty-format descriptions of multibit cells.

Nonscan Register Cell Models

The regular cell functionality of the multibit cells whose structures are parallel data-in and parallel data-out are modeled using the `ff_bank` or `latch_bank` Liberty syntax, instead of the `ff` or `latch` syntax used in a single-bit cell.

Single-Bit Nonscan Cell

The single-bit version of a multibit cell is simply a D flip-flop or D latch model having the same functionality as each bit in the multibit cell. This is the general Liberty syntax for a D flip-flop:

```
library (library_name) {
  …

  cell (cell_name) {
    ...
    ff(variable1, variable2) {
      clocked_on : "Boolean_expression" ;
      next_state : "Boolean_expression" ;
      clear : "Boolean_expression" ;
      preset : "Boolean_expression" ;
      clear_preset_var1 : value ;
      clear_preset_var2 : value ;
      clocked_on_also :"Boolean_expression";
      power_down_function : "Boolean_expression" ;
    }
  }
}
```

variable1 is the state of the noninverting output of the flip-flop. It is considered the 1-bit storage value of the flip-flop. *variable2* is the state of the inverting output.

You can name *variable1* and *variable2* anything except the name of a pin in the cell being described. Both variables are required, even if one of them is not connected to a primary output pin.

In an `ff` group, the `clocked_on` and `next_state` attributes are required; all other attributes are optional.

The syntax for a latch is similar to that of the flip-flop. For details, see “Describing a Latch” in “Defining Sequential Cells” in the *Library Compiler User Guide*, available on SolvNetPlus.

Multibit Nonscan Cell

The regular cell functionality of the multibit cells whose structures are parallel data-in and parallel data-out are modeled using the `ff_bank` or `latch_bank` Liberty syntax. This is the general Liberty syntax for a multibit flip-flop:

```
library(library_name) {
  …
  cell (cell_name) {
    ...
    pin (pin_name) {
      ...
    }
    bus (bus_name) {
      ...
    }

    ff_bank (variable1, variable2, bits) {
      clocked_on : "Boolean_expression" ;
       next_state : "Boolean_expression" ;
       clear : "Boolean_expression" ;
       preset : "Boolean_expression" ;
       clear_preset_var1 : value ;
       clear_preset_var2 : value ;
       clocked_on_also : "Boolean_expression" ;
    }
  }
}
```

Note that either `bus` or `bundle` Liberty syntax can be used to model these multibit signal pins in the library.

The syntax for a multibit latch is similar to that of the multibit flip-flop. For details, see “Describing a Multibit Latch” in “Defining Sequential Cells” in the *Library Compiler User Guide*, available on SolvNetPlus.

Feedback

Multibit Scan Register Cell Models

Multibit scan cells can have parallel or serial scan chains. The scan output of a multibit scan cell can reuse the data output pins or have a dedicated scan output (SO) pin in addition to the bus, or the bundle output pins.

The Library Compiler tool supports multibit scan cells with the structures described in Table 4. Use the Liberty syntax described in the table to model the functionality of the cells.

Table 4 Modeling Multibit Scan Cells

Multibit cell type	Liberty syntax
Parallel scan bits with a dedicated scan output bus	`statetable`
Parallel scan bits without a dedicated scan output bus	`ff_bank`
Serial scan chain with a dedicated scan output pin	`statetable`
Serial scan chain without a dedicated scan output pin	`statetable`

This is the general Liberty syntax for a multibit parallel scan cell:

```
library(library_name) {
  …
  cell (cell_name) {
    single_bit_degenerate : single_bit_scan_seq_cell_name;
    ...
    pin (pin_name) {
      ...
    }
    bus (bus_name) {
      direction : input;
      ...
    }
    bus (bus_name) {
      direction : output;
      pin(bus_name[0]) {
      input_map : "input_node_names";
      …
    }
    …
  }
       statetable( "input node names", "internal node names" ) {
table : "input node values : current int. values : next int. values, \
         input node values : current int. values : next int. values" ;
       power_down_function : "Boolean expression" ;
     }
```

Feedback

```
    }
}
```

Note:

The `single_bit_degenerate` attribute is needed only for complex serial scan cells that can also have a dedicated scan output pin, and in some complex parallel multibit sequential cell models for automatic inference in optimization tools. For details, see the *Library Compiler User Guide*.

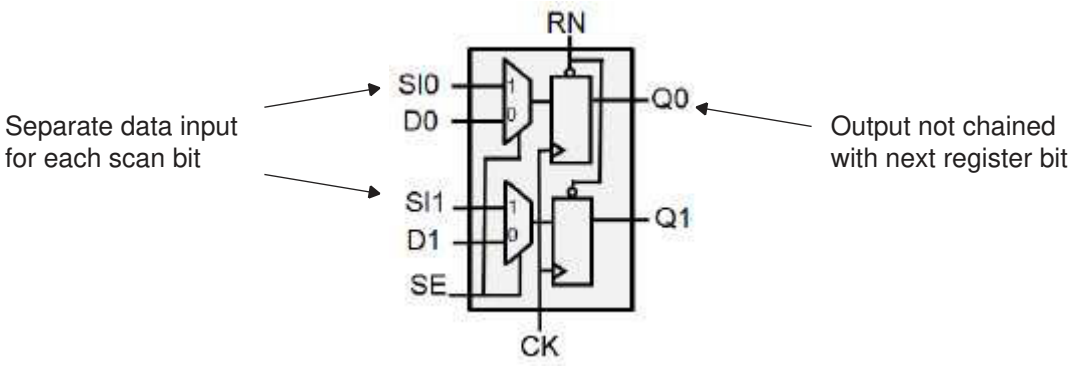
The following topics show detailed examples to demonstrate the multibit scan cell syntax.

- Multibit Scan Cell With Parallel Scan Bits
- Multibit Scan Cell With Internal Serial Scan Chain

Multibit Scan Cell With Parallel Scan Bits

A multibit scan cell with parallel scan bits has parallel data and scan inputs, and parallel functional and scan outputs. Figure 15 shows the logic diagram of a multibit scan cell with parallel input and output pins for both the normal operating mode and scan mode.

Figure 15 Multibit Register Cell With Parallel Scan Bits



In the normal operating mode, the cell uses the input and output buses, D0-D1 and Q0-Q1, respectively.

In the scan mode, the cell functions as a parallel-in, parallel-out register using the scan input bus, SI0-SI1 for the input scan data. For the output scan data, the cell either reuses the data output bus, Q0-Q1, as shown in Figure 15, or uses a separate dedicated scan output bus, SO0-SO1. The scan enable signal can be a bus, SE0-SE1, where each bit of the bus enables a corresponding sequential element of the cell, or it can be a single-bit pin that enables all sequential elements of the cell, as shown in Figure 15.

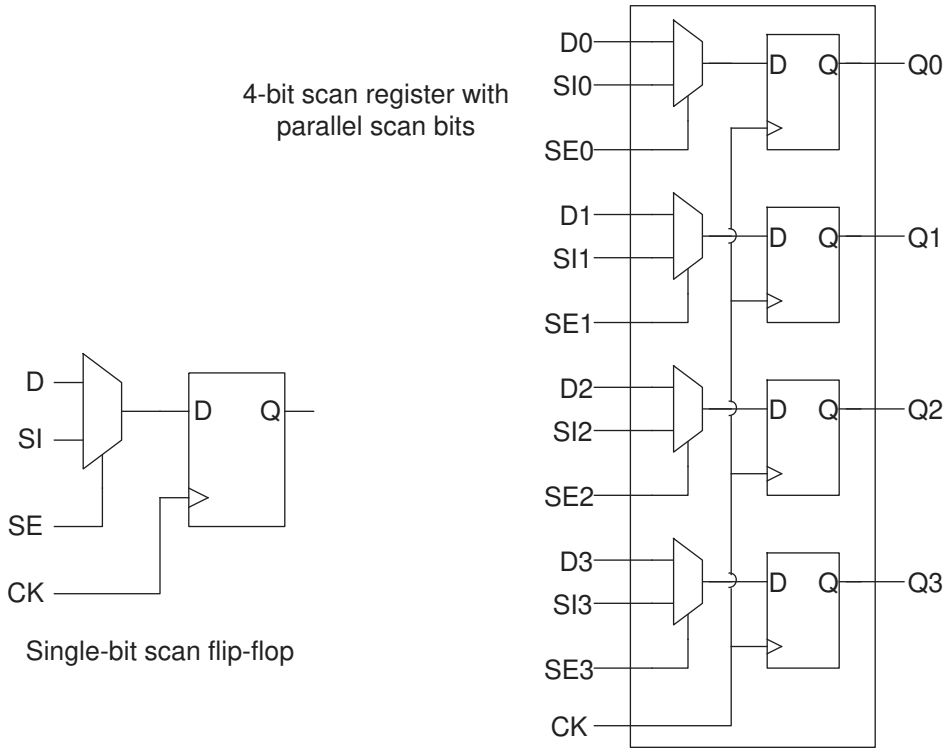
This is the general Liberty syntax for modeling a parallel scan multibit register:

```
library(library_name) {
...
cell(cell_name) {
  ff_bank (variable1, variable2, bits) {
    clocked_on : "Boolean_expression" ;
    next_state : "Boolean_expression" ;
    clear : "Boolean_expression" ;
    preset : "Boolean_expression" ;
    clear_preset_var1 : value ;
    clear_preset_var2 : value ;
    clocked_on_also : "Boolean_expression" ;
  }
   bus(scan_in_pin_name) {
/* cell scan in signal bus that has the signal_type as */
/* "test_scan_in" inside the test_cell group           */
    ...
  }
  bus(scan_out_pin_name) {
/* cell scan out signal bus with signal_type as */
/* "test_scan_out" inside the test_cell group   */
    ...
  }
  bus | bundle (bus_bundle_name) {
    direction : input | output;
  }
  test_cell() {
    pin(scan_in_pin_name) {
      signal_type : test_scan_in;
      ...
  }
  pin(scan_out_pin_name) {
    signal_type : test_scan_out | test_scan_out_inverted;
    ...
  }
...
}
 } /* end cell group */
} /* end library group */
```

The following example shows the Liberty syntax for modeling a 4-bit scan cell with parallel scan bits and the single-bit equivalent cell. Figure 16 shows the logic diagram for the 1-bit and 4-bit cells. During design optimization, the tool can map four single-bit cells to a single 4-bit cell.

Feedback

Figure 16 4-bit Register Cell With Parallel Scan Bits



The 4-bit cell is defined by the `ff_bank` group and the `function` attribute on the bus.

In the normal operating mode, the cell is a parallel shift register that uses the data input bus, D0 through D3, and the data output bus, Q0 through Q3.

In the scan mode, the cell functions as a 1-bit shift register with parallel scan input and scan enable signals, and parallel reused output signals. To define the cell behavior in the scan mode, set the `signal_type` attribute to `test_scan_out` on the bus Q in the `test_cell` group.

To see more multibit scan cell figures and examples, see the *Library Compiler User Guide*.

Parallel Scan Cell Examples

Example 3 and Example 4 show the Liberty description of a 4-bit parallel scan cell using bundle and bus syntax, respectively. These cell description examples are complete and working; they can be copied into a .lib file and compiled by the Library Compiler tool without modification.

Example 3 Liberty Model of 4-Bit Parallel Scan Cell Using "bundle" Syntax

```
library (mylib_using_bundle) {
  delay_model : table_lookup;
  time_unit                     : "1ns";
  current_unit                  : "1mA";
  voltage_unit                  : "1V";
  capacitive_load_unit           (1,pf);
  pulling_resistance_unit       : "1kohm";
  default_fanout_load           : 1.0;
  default_output_pin_cap        : 0.00;
  default_inout_pin_cap         : 0.00;
  default_input_pin_cap         : 0.01;
  leakage_power_unit            : "1nW";
  default_cell_leakage_power    : 0.00;
  default_leakage_power_density : 0.00;

  slew_lower_threshold_pct_rise : 30.00;
  slew_upper_threshold_pct_rise : 70.00;
  slew_lower_threshold_pct_fall : 30.00;
  slew_upper_threshold_pct_fall : 70.00;
  input_threshold_pct_rise      : 50.00;
  output_threshold_pct_rise     : 50.00;
  input_threshold_pct_fall      : 50.00;
  output_threshold_pct_fall     : 50.00;

  voltage_map(VDD, 1.0);
  voltage_map(VSS, 0.0);

/* operation conditions */
  nom_process     : 1;
  nom_temperature : 25;
  nom_voltage     : 1.0;
  operating_conditions(myoc) {
    process     : 1;
    temperature : 25;
    voltage     : 1.0;
    tree_type   : balanced_tree
  }
  default_operating_conditions : myoc;

cell (4-bit_parallel_scan_cell) {
  area : 4.0;
  pg_pin(VDD) {
    voltage_name : VDD;
    pg_type : primary_power;
  }
  pg_pin(VSS) {
    voltage_name : VSS;
    pg_type : primary_ground;
  }

    ff_bank (IQ,IQN,4) {
```

```
  next_state : "(D * !SE + SI * SE)";
  clocked_on : "CK";
}

/* functional output bundle pins*/
bundle(Q) {
  members (Q0, Q1, Q2, Q3);
  direction : output;
  function : IQ
  related_power_pin : VDD;
  related_ground_pin : VSS;
  timing() {
    related_pin : "CK" ;
    timing_type : rising_edge ;
    cell_fall (scalar) { values("0.0000"); }
    cell_rise (scalar) { values("0.0000"); }
    fall_transition (scalar) { values("0.0000"); }
    rise_transition (scalar) { values("0.0000"); }
  }
}
pin(CK) {
  direction : input;
  related_power_pin : VDD;
  related_ground_pin : VSS;
  capacitance : 1.0;
}
/* scan enable bundle pins */
bundle(SE) {
  members (SE0, SE1, SE2, SE3);
  direction : input;
  related_power_pin : VDD;
  related_ground_pin : VSS;
  capacitance : 1.0;
  nextstate_type : scan_enable;
}
/* scan input bus pins */
bundle(SI) {
  members (SI0, SI1, SI2, SI3);
  direction : input;
  related_power_pin : VDD;
  related_ground_pin : VSS;
  capacitance : 1.0;
  nextstate_type : scan_in;
}
/* data input bundle pins */
bundle(D) {
  members (D0, D1, D2, D3);
  direction : input;
  related_power_pin : VDD;
  related_ground_pin : VSS;
  capacitance : 1.0;
  nextstate_type : data;
```

```
    }
    test_cell () {
      pin(CK){
        direction : input;
      }
      bundle(D) {
      members (D0, D1, D2, D3);
        direction : input;
      }
      bundle(SI) {
        members (SI0, SI1, SI2, SI3);
        direction : input;
        signal_type : "test_scan_in";
      }
      bundle(SE) {
        members (SE0, SE1, SE2, SE3);
        direction : input;
        signal_type : "test_scan_enable";
      }
      ff_bank (IQ,IQN,4) {
        next_state : "D";
        clocked_on : "CK";
      }
      bundle(Q) {
        members (Q0, Q1, Q2, Q3);
        direction : output;
        function : "IQ";
        signal_type : "test_scan_out";
      }
    } /* end test_cell group */
   }/* end cell group */
  }/* end library group */
```

Example 4 Liberty Model of 4-Bit Parallel Scan Cell Using “bus” Syntax

```
  library (mylib_using_bus) {
    delay_model                    : table_lookup;
    time_unit                      : "1ns";
    current_unit                   : "1mA";
    voltage_unit                   : "1V";
    capacitive_load_unit            (1,pf);
    pulling_resistance_unit        : "1kohm";
    default_fanout_load            : 1.0;
    default_output_pin_cap         : 0.00;
    default_inout_pin_cap          : 0.00;
    default_input_pin_cap          : 0.01;
    leakage_power_unit             : "1nW";
    default_cell_leakage_power     : 0.00;
    default_leakage_power_density : 0.00;

    slew_lower_threshold_pct_rise : 30.00;
    slew_upper_threshold_pct_rise : 70.00;
    slew_lower_threshold_pct_fall : 30.00;
```

```
    slew_upper_threshold_pct_fall : 70.00;
    input_threshold_pct_rise      : 50.00;
    output_threshold_pct_rise     : 50.00;
    input_threshold_pct_fall      : 50.00;
    output_threshold_pct_fall     : 50.00;

    voltage_map(VDD, 1.0);
    voltage_map(VSS, 0.0);

  /* operation conditions */
    nom_process     : 1;
    nom_temperature : 25;
    nom_voltage     : 1.0;
    operating_conditions(myoc) {
      process     : 1;
      temperature : 25;
      voltage     : 1.0;
      tree_type   : balanced_tree
    }
    default_operating_conditions : myoc;

    type(bus4){
     base_type : array;
     bit_from : 0;
     bit_to : 3;
     bit_width : 4;
     data_type : bit;
     downto : false;
    }

  cell (4-bit_parallel_scan_cell) {
    area : 4.0;
    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }

    ff_bank (IQ,IQN,4) {
      next_state : "(D * !SE + SI * SE)";
      clocked_on : "CK";
    }
    /* functional output bus pins */
    bus(Q) {
      bus_type : bus4;
      direction : output;
      function : IQ;
      related_power_pin : VDD;
      related_ground_pin : VSS;
```

```
    timing() {
      related_pin : "CK" ;
      timing_type : rising_edge ;
      cell_fall (scalar) { values("0.0000"); }
      cell_rise (scalar) { values("0.0000"); }
      fall_transition (scalar) { values("0.0000"); }
      rise_transition (scalar) { values("0.0000"); }
    }
  }
  pin(CK) {
    direction : input;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    capacitance : 1.0;
  }
  /* scan enable bus pins */
  bus(SE) {
    bus_type : bus4;
    direction : input;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    capacitance : 1.0;
    nextstate_type : scan_enable;
  }
  /* scan input bus pins */
  bus(SI) {
    bus_type : bus4;
    direction : input;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    capacitance : 1.0;
    nextstate_type : scan_in;
  }
  /* data input bus pins */
  bus(D) {
    bus_type : bus4;
    direction : input;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    capacitance : 1.0;
    nextstate_type : data;
  }
  test_cell () {
    pin(CK){
      direction : input;
    }
    bus(D) {
      bus_type : bus4;
      direction : input;
    }
    bus(SI) {
      bus_type : bus4;
      direction : input;
```

Feedback

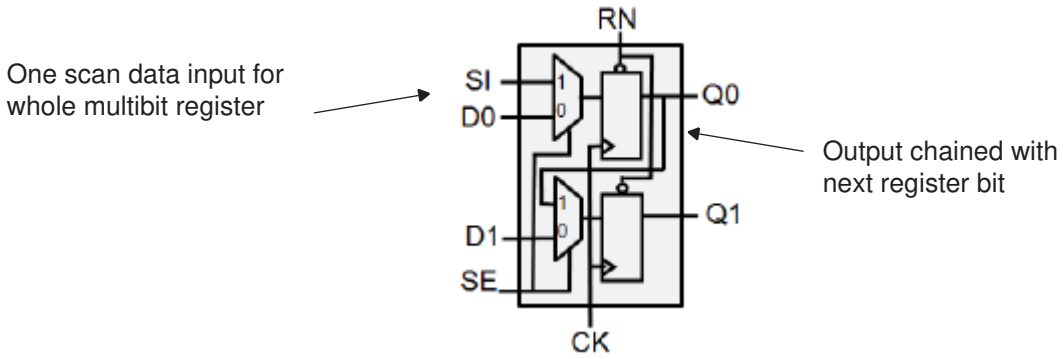
```
      signal_type : "test_scan_in";
    }
    bus(SE) {
      bus_type : bus4;
      direction : input;
      signal_type : "test_scan_enable";
    }
    ff_bank (IQ,IQN,4) {
      next_state : "D";
      clocked_on : "CK";
    }
    bus(Q) {
      bus_type : bus4 ;
      direction : output;
      function : "IQ";
      signal_type : "test_scan_out";
    }
  } /* end test_cell group */
 }/* end cell group */
}/* end library group */
```

Multibit Scan Cell With Internal Serial Scan Chain

A multibit scan cell with an internal scan chain has a serial scan chain, with a single shared or dedicated output pin to output the scan signal.

Figure 17 shows the schematic diagram of a generic multibit scan cell with an internal or serial scan chain and a shared-purpose output pin, Q1, for the scan chain output.

Figure 17 Multibit Register Cell With Internal Scan Chain



In the normal operating mode, the cell uses the input and output buses, D0-D1 and Q0-Q1, respectively. In the scan mode, the cell functions as a serial shift register from the scan input (SI) pin to the scan output pin, which can reuse the last Q (Q1) output as in

this example, or can use a dedicated scan output pin. The scan chain is stitched using the data output Q of each sequential element of the cell.

This is the general Liberty syntax for a multibit flip-flop with an internal serial scan chain:

```
library(library_name) {
  ...
   bus | bundle (bus_bundle_name) {
     direction : inout | output;
     scan_start_pin : pin_name;
     scan_pin_inverted : true | false;
  }
  test_cell() {
        pin(scan_in_pin_name) {
        signal_type : test_scan_in;
        ...
       }
         pin(scan_out_pin_name) {
           signal_type : test_scan_out | test_scan_out_inverted;
           ...
         }
       ...
  }
}
```

Attributes Defined in the “bus” or “bundle” Group

To model the internal scan chain in a multibit scan cell with a dedicated scan output pin, the cell definition can use certain attributes in the `bus` and `bundle` groups.

scan_start_pin

The optional `scan_start_pin` attribute specifies where the internal scan chain begins.

The tool supports only the following types of scan chains: from the least significant bit (LSB) to the most significant bit (MSB) of the output bus or bundle group; or from the MSB to the LSB of the output bus or bundle group. Therefore, for a multibit scan cell with an internal scan chain, the value of the `scan_start_pin` attribute can either be the LSB or MSB output pin.

Specify the `scan_start_pin` attribute in the `bus` or `bundle` group. You cannot specify this attribute in the `pin` group, even for pin definitions of the `bus` or `bundle` group.

scan_pin_inverted

The optional `scan_pin_inverted` attribute specifies that the scan signal is inverted after the first sequential element of the multibit scan cell. The valid values are `true` and `false`. The default is `false`. If you specify `true`, the `signal_type` attribute must be set to `test_scan_out_inverted`.

Feedback

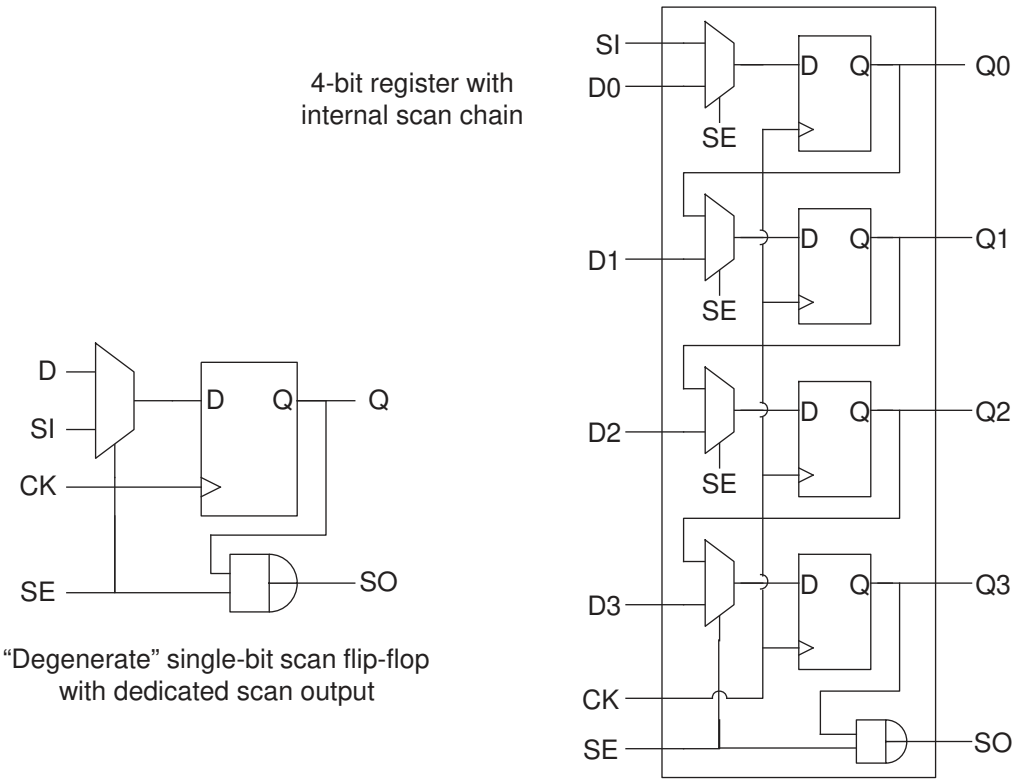
If you specify the `scan_pin_inverted` attribute, you must specify the `scan_start_pin` attribute in the same `bus` or `bundle` group. You cannot specify this attribute in the `pin` group, even for pin definitions in the `bus` or `bundle` group.

For the cell in Figure 17, the Library Compiler tool identifies the pins in the internal scan chain based on the values of the `scan_start_pin` and `scan_pin_inverted` attributes and performs the scan only in forward or reverse sequential order, such a 0-1-2-3 or 3-2-1-0. A random shift order such as 2-0-3-1 is not supported. The scan data can shift out through either the inverting or noninverting output pin.

Internal Serial Scan Cell Example

Figure 18 shows a schematic of a 4-bit scan cell with a serial scan chain and a single scan output pin, SO. The figure also shows the corresponding single-bit cell. During optimization, the Design Compiler tool maps single-bit cells to the multibit cell.

Figure 18 4-bit Register With Internal Scan Chain



The 4-bit cell has the output bus Q0-Q3 and a single scan output pin with combinational logic. The cell is defined using the `statetable` group and the `state_function` attribute on the serial output pin, SO.

Feedback

In the normal operating mode, the cell is a shift register that uses the output bus Q[0:3].

In scan mode, the cell functions as a shift register with the single-bit output pin SO. To define the scan mode for the cell, set the `signal_type` attribute to `test_scan_out` on the pin SO in the `test_cell` group. Do not define the function attribute of this pin.

Example 5 and Example 6 show the Liberty description of a 4-bit cell with an internal scan chain and a single dedicated scan output, using the `bus` and `bundle` syntax, respectively. These cell description examples are complete and working; they can be copied into a .lib file and compiled by the Library Compiler tool without modification.

Example 5 Liberty Model of 4-Bit Internal Serial Scan Cell Using "bus" Syntax

```
library (test_bus) {
  delay_model                    : table_lookup;
  time_unit                      : "1ns";
  current_unit                   : "1mA";
  voltage_unit                   : "1V";
  capacitive_load_unit            (1,pf);
  pulling_resistance_unit        : "1kohm";
  default_fanout_load            : 1.0;
  default_output_pin_cap         : 0.00;
  default_inout_pin_cap          : 0.00;
  default_input_pin_cap          : 0.01;
  slew_lower_threshold_pct_rise : 30.00;
  slew_upper_threshold_pct_rise : 70.00;
  slew_lower_threshold_pct_fall : 30.00;
  slew_upper_threshold_pct_fall : 70.00;
  input_threshold_pct_rise       : 50.00;
  output_threshold_pct_rise      : 50.00;
  input_threshold_pct_fall       : 50.00;
  output_threshold_pct_fall      : 50.00;
  leakage_power_unit             : "1nW";
  default_cell_leakage_power     : 0.00;
  default_leakage_power_density : 0.00;

  voltage_map(VDD, 1.0);
  voltage_map(VSS, 0.0);

/* operation conditions */
  nom_process     : 1;
  nom_temperature : 25;
  nom_voltage     : 1.0;
  operating_conditions(typical) {
    process     : 1;
    temperature : 25;
    voltage     : 1.0;
    tree_type   : balanced_tree
  }
  default_operating_conditions : typical;
  type(bus4){
   base_type : array;
```

```
      bit_from : 0;
      bit_to : 3;
      bit_width : 4;
      data_type : bit;
      downto : false;
    }
  /* Single-bit Scan DFF with gated_scanout */
    cell(SDFF_SO) {
      area : 1.0;
      pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
      }
      pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
      }
      ff("IQ", "IQN") {
        next_state : "D * !SE + SI * SE" ;
        clocked_on : "CK" ;
      }
      pin(Q) {
        direction : output ;
        related_power_pin  : VDD;
        related_ground_pin : VSS;
        function : "IQ" ;
        timing() {
          related_pin : "CK" ;
          timing_type : rising_edge ;
          cell_fall (scalar) { values("0.0000"); }
          cell_rise (scalar) { values("0.0000"); }
          fall_transition (scalar) { values("0.0000"); }
          rise_transition (scalar) { values("0.0000"); }
        }
      }
      pin(SO) {
        direction : output ;
        related_power_pin  : VDD;
        related_ground_pin : VSS;
        function : "SE * IQ" ;
        timing() {
          related_pin : "CK" ;
          timing_type : rising_edge ;
          cell_fall (scalar) { values("0.0000"); }
          cell_rise (scalar) { values("0.0000"); }
          fall_transition (scalar) { values("0.0000"); }
          rise_transition (scalar) { values("0.0000"); }
        }
      }
      pin(D) {
        direction : input ;
        related_power_pin  : VDD;
```

```
    related_ground_pin : VSS;
    capacitance : 0.1;
    nextstate_type : data;
    timing(){
      timing_type : setup_rising;
      fall_constraint (scalar) { values("0.0000"); }
      rise_constraint (scalar) { values("0.0000"); }
      related_pin : "CK" ;
    }
    timing(){
      timing_type : hold_rising;
      fall_constraint (scalar) { values("0.0000"); }
      rise_constraint (scalar) { values("0.0000"); }
      related_pin : "CK" ;
    }
  }
  pin(SI) {
    direction : input ;
    related_power_pin  : VDD;
    related_ground_pin : VSS;
    capacitance : 0.1;
    nextstate_type : scan_in;
    timing(){
      timing_type : setup_rising;
      fall_constraint (scalar) { values("0.0000"); }
      rise_constraint (scalar) { values("0.0000"); }
      related_pin : "CK" ;
    }
    timing(){
      timing_type : hold_rising;
      fall_constraint (scalar) { values("0.0000"); }
      rise_constraint (scalar) { values("0.0000"); }
      related_pin : "CK" ;
    }
  }
  pin(SE) {
    direction : input ;
    related_power_pin  : VDD;
    related_ground_pin : VSS;
    capacitance : 0.1;
    nextstate_type : scan_enable;
    timing(){
      timing_type : setup_rising;
      fall_constraint (scalar) { values("0.0000"); }
      rise_constraint (scalar) { values("0.0000"); }
      related_pin : "CK" ;
    }
    timing(){
      timing_type : hold_rising;
      fall_constraint (scalar) { values("0.0000"); }
      rise_constraint (scalar) { values("0.0000"); }
      related_pin : "CK" ;
    }
```

```
      }
      pin(CK) {
        direction : input ;
        related_power_pin  : VDD;
        related_ground_pin : VSS;
        capacitance : 0.1;
      }
      test_cell() {
        pin(CK){
          direction : input;
        }
        pin(D) {
          direction : input;
        }
        pin(SI) {
          direction : input;
          signal_type : "test_scan_in";
        }
        pin(SE) {
          direction : input;
          signal_type : "test_scan_enable";
        }
        ff(IQ,IQN) {
          next_state : "D";
          clocked_on : "CK";
        }
        pin(Q) {
          direction : output;
          function  : "IQ";
        }
        pin(SO) {
          direction : output;
          signal_type : "test_scan_out";
          test_output_only : "true";
        }
      }
    }
  /* 4-bit Scan DFF and gated_scanout */
    cell(SDFF4_SO) {
      area   : 4.0;

      pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
      }
      pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
      }
      pin(CK) {
        clock              : true;
        direction          : input;
```

```
    capacitance        :  1.0   ;
    related_power_pin  : VDD;
    related_ground_pin : VSS;
  }
  bus(D) {
    bus_type : bus4;
    direction          : input;
    capacitance        :  1.0 ;
    related_power_pin  : VDD;
    related_ground_pin : VSS;
    pin(D[0]) {
      timing() {
        related_pin        : "CK";
        timing_type        : "hold_rising";
        fall_constraint(scalar) { values("0.0"); }
        rise_constraint(scalar) { values("0.0"); }
      }
      timing() {
        related_pin        : "CK";
        timing_type        : "setup_rising";
        fall_constraint(scalar) { values("0.0"); }
        rise_constraint(scalar) { values("0.0"); }
      }
    }
    pin(D[1]) {
      timing() {
      related_pin        : "CK";
      timing_type        : "hold_rising";
      fall_constraint(scalar) { values("0.1"); }
      rise_constraint(scalar) { values("0.1"); }
    }
      timing() {
        related_pin        : "CK";
        timing_type        : "setup_rising";
        fall_constraint(scalar) { values("0.1"); }
        rise_constraint(scalar) { values("0.1"); }
      }
    }
    pin(D[2]) {
      timing() {
        related_pin        : "CK";
        timing_type        : "hold_rising";
        fall_constraint(scalar) { values("0.0"); }
        rise_constraint(scalar) { values("0.0"); }
      }
      timing() {
        related_pin        : "CK";
        timing_type        : "setup_rising";
        fall_constraint(scalar) { values("0.0"); }
        rise_constraint(scalar) { values("0.0"); }
      }
    }
    pin(D[3]) {
```

```
    timing() {
    related_pin        : "CK";
    timing_type        : "hold_rising";
    fall_constraint(scalar) { values("0.1"); }
    rise_constraint(scalar) { values("0.1"); }
  }
    timing() {
      related_pin        : "CK";
      timing_type        : "setup_rising";
      fall_constraint(scalar) { values("0.1"); }
      rise_constraint(scalar) { values("0.1"); }
    }
  }
}
pin(SE) {
  direction          : input;
  capacitance        :  1.0 ;
  related_power_pin  : VDD;
  related_ground_pin : VSS;
  timing() {
    related_pin        : "CK";
    timing_type        : "hold_rising";
    fall_constraint(scalar) { values("0.0"); }
    rise_constraint(scalar) { values("0.0"); }
  }
  timing() {
    related_pin        : "CK";
    timing_type        : "setup_rising";
    fall_constraint(scalar) { values("0.0"); }
    rise_constraint(scalar) { values("0.0"); }
  }
}
pin(SI) {
  direction          : input;
  capacitance        :  1.0 ;
  related_power_pin  : VDD;
  related_ground_pin : VSS;
  timing() {
    related_pin        : "CK";
    timing_type        : "hold_rising";
    fall_constraint(scalar) { values("0.0"); }
    rise_constraint(scalar) { values("0.0"); }
  }
  timing() {
    related_pin        : "CK";
    timing_type        : "setup_rising";
    fall_constraint(scalar) { values("0.0"); }
   rise_constraint(scalar) { values("0.0"); }
  }
}
statetable ( " D  CK  SE  SI " ,    "Q" ) {
  table :    " -  ~R   -   -  : - :  N,   \
              H/L  R   L   -  : - : H/L,  \
```

```
                -   R   H  H/L : - : H/L" ;
  }
  bus(Q) {
    bus_type : bus4;
    direction       : output;
    inverted_output    : false;
    internal_node : "Q" ;
    related_power_pin  : VDD;
    related_ground_pin : VSS;
    scan_start_pin : Q[0];
    pin (Q[0]) {
      input_map : " D[0]  CK  SE  SI";
      timing() {
        related_pin        : "CK";
        timing_type        : "rising_edge";
        cell_fall(scalar) { values("0.0"); }
        fall_transition(scalar) { values("0.0"); }
        cell_rise(scalar) { values("0.0"); }
        rise_transition(scalar) { values("0.0"); }
      }
    }
    pin (Q[1]) {
      input_map : " D[1]  CK  SE  Q[0]";
      timing() {
        related_pin        : "CK";
        timing_type        : "rising_edge";
        cell_fall(scalar) { values("0.0"); }
        fall_transition(scalar) { values("0.0"); }
        cell_rise(scalar) { values("0.0"); }
        rise_transition(scalar) { values("0.0"); }
      }
    }
    pin (Q[2]) {
      input_map : " D[2]  CK  SE  Q[1]";
      timing() {
        related_pin        : "CK";
        timing_type        : "rising_edge";
        cell_fall(scalar) { values("0.0"); }
        fall_transition(scalar) { values("0.0"); }
        cell_rise(scalar) { values("0.0"); }
        rise_transition(scalar) { values("0.0"); }
      }
    }
    pin (Q[3]) {
      input_map : " D[3]  CK  SE  Q[2]";
      timing() {
        related_pin        : "CK";
        timing_type        : "rising_edge";
        cell_fall(scalar) { values("0.0"); }
        fall_transition(scalar) { values("0.0"); }
        cell_rise(scalar) { values("0.0"); }
        rise_transition(scalar) { values("0.0"); }
```

```
      }
    }
  }
  pin(SO) {
    direction       : output;
    state_function : "SE * Q[3]" ;
    related_power_pin  : VDD;
    related_ground_pin : VSS;
    test_output_only : true;
    timing() {
      related_pin        : "CK";
      timing_type        : "rising_edge";
      cell_fall(scalar) { values("0.0"); }
      fall_transition(scalar) { values("0.0"); }
      cell_rise(scalar) { values("0.0"); }
      rise_transition(scalar) { values("0.0"); }
    }
  }

  test_cell() {
    pin(CK){
      direction : input;
    }
    bus(D) {
      bus_type : bus4;
      direction : input;
    }
    pin(SI) {
      direction : input;
      signal_type : "test_scan_in";
    }
    pin(SE) {
      direction : input;
      signal_type : "test_scan_enable";
    }
    ff_bank (IQ,IQN,4) {
      next_state : "D";
      clocked_on : "CK";
    }
    bus(Q) {
      bus_type : bus4;
      direction : output;
      function  : "IQ";
    }
    pin(SO) {
      direction : output;
      signal_type : "test_scan_out";
      test_output_only  : "true";
    }
  }
 } /* end cell group */
} /* end library group */
```

Example 6 Liberty Model of 4-Bit Internal Serial Scan Cell Using "bundle" Syntax

```
library (test_bundle) {
  delay_model                    : table_lookup;
  time_unit                      : "1ns";
  current_unit                   : "1mA";
  voltage_unit                   : "1V";
  capacitive_load_unit            (1,pf);
  pulling_resistance_unit        : "1kohm";
  default_fanout_load            : 1.0;
  default_output_pin_cap         : 0.00;
  default_inout_pin_cap          : 0.00;
  default_input_pin_cap          : 0.01;
  slew_lower_threshold_pct_rise : 30.00;
  slew_upper_threshold_pct_rise : 70.00;
  slew_lower_threshold_pct_fall : 30.00;
  slew_upper_threshold_pct_fall : 70.00;
  input_threshold_pct_rise       : 50.00;
  output_threshold_pct_rise      : 50.00;
  input_threshold_pct_fall       : 50.00;
  output_threshold_pct_fall      : 50.00;
  leakage_power_unit             : "1nW";
  default_cell_leakage_power     : 0.00;
  default_leakage_power_density : 0.00;

  voltage_map(VDD, 1.0);
  voltage_map(VSS, 0.0);

/* operation conditions */
  nom_process     : 1;
  nom_temperature : 25;
  nom_voltage     : 1.0;
  operating_conditions(typical) {
    process     : 1;
    temperature : 25;
    voltage     : 1.0;
    tree_type   : balanced_tree
  }
  default_operating_conditions : typical;

/* Single-bit Scan DFF with gated_scanout */
  cell(SDFF_SO) {
    area : 1.0;

    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
```

```
ff("IQ", "IQN") {
  next_state : "D * !SE + SI * SE" ;
  clocked_on : "CK" ;
}

pin(Q) {
  direction : output ;
  related_power_pin  : VDD;
  related_ground_pin : VSS;
  function : "IQ" ;
  timing() {
    related_pin : "CK" ;
    timing_type : rising_edge ;
    cell_fall (scalar) { values("0.0000"); }
    cell_rise (scalar) { values("0.0000"); }
    fall_transition (scalar) { values("0.0000"); }
    rise_transition (scalar) { values("0.0000"); }
  }
}
pin(SO) {
  direction : output ;
  related_power_pin  : VDD;
  related_ground_pin : VSS;
  function : "SE * IQ" ;
  timing() {
    related_pin : "CK" ;
    timing_type : rising_edge ;
    cell_fall (scalar) { values("0.0000"); }
    cell_rise (scalar) { values("0.0000"); }
    fall_transition (scalar) { values("0.0000"); }
    rise_transition (scalar) { values("0.0000"); }
  }
}
pin(D) {
  direction : input ;
  related_power_pin  : VDD;
  related_ground_pin : VSS;
  capacitance : 0.1;
  nextstate_type : data;
  timing(){
    timing_type : setup_rising;
    fall_constraint (scalar) { values("0.0000"); }
    rise_constraint (scalar) { values("0.0000"); }
    related_pin : "CK" ;
  }
  timing(){
    timing_type : hold_rising;
    fall_constraint (scalar) { values("0.0000"); }
    rise_constraint (scalar) { values("0.0000"); }
    related_pin : "CK" ;
  }
}
```

```
pin(SI) {
  direction : input ;
  related_power_pin  : VDD;
  related_ground_pin : VSS;
  capacitance : 0.1;
  nextstate_type : scan_in;
  timing(){
    timing_type : setup_rising;
    fall_constraint (scalar) { values("0.0000"); }
    rise_constraint (scalar) { values("0.0000"); }
    related_pin : "CK" ;
  }
  timing(){
    timing_type : hold_rising;
    fall_constraint (scalar) { values("0.0000"); }
    rise_constraint (scalar) { values("0.0000"); }
    related_pin : "CK" ;
  }
}
pin(SE) {
  direction : input ;
  related_power_pin  : VDD;
  related_ground_pin : VSS;
  capacitance : 0.1;
  nextstate_type : scan_enable;
  timing(){
    timing_type : setup_rising;
    fall_constraint (scalar) { values("0.0000"); }
    rise_constraint (scalar) { values("0.0000"); }
    related_pin : "CK" ;
  }
  timing(){
    timing_type : hold_rising;
    fall_constraint (scalar) { values("0.0000"); }
    rise_constraint (scalar) { values("0.0000"); }
    related_pin : "CK" ;
  }
}
pin(CK) {
  direction : input ;
  related_power_pin  : VDD;
  related_ground_pin : VSS;
  capacitance : 0.1;
}
test_cell() {
  pin(CK){
    direction : input;
  }
  pin(D) {
    direction : input;
  }
  pin(SI) {
    direction : input;
```

```
        signal_type : "test_scan_in";
      }
      pin(SE) {
        direction : input;
        signal_type : "test_scan_enable";
      }
      ff(IQ,IQN) {
        next_state : "D";
        clocked_on : "CK";
      }
      pin(Q) {
        direction : output;
        function  : "IQ";
      }
      pin(SO) {
        direction : output;
        signal_type : "test_scan_out";
        test_output_only : "true";
      }
    }
  }
/* 4-bit Scan DFF and gated_scanout */
  cell(SDFF4_SO) {
    area   : 2.0;

    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }

    pin(CK) {
      clock              : true;
      direction          : input;
      capacitance        :  1.0   ;
      related_power_pin  : VDD;
      related_ground_pin : VSS;
    }

    bundle(D) {
      members(D0, D1, D2, D3);
      direction          : input;
      capacitance        :  1.0 ;
      related_power_pin  : VDD;
      related_ground_pin : VSS;
      pin(D0) {
        timing() {
          related_pin        : "CK";
          timing_type        : "hold_rising";
```

```
        fall_constraint(scalar) { values("0.0"); }
        rise_constraint(scalar) { values("0.0"); }
      }
      timing() {
        related_pin        : "CK";
        timing_type        : "setup_rising";
        fall_constraint(scalar) { values("0.0"); }
        rise_constraint(scalar) { values("0.0"); }
      }
    }
    pin(D1) {
      timing() {
      related_pin        : "CK";
      timing_type        : "hold_rising";
      fall_constraint(scalar) { values("0.1"); }
      rise_constraint(scalar) { values("0.1"); }
    }
      timing() {
        related_pin        : "CK";
        timing_type        : "setup_rising";
        fall_constraint(scalar) { values("0.1"); }
        rise_constraint(scalar) { values("0.1"); }
      }
    }
    pin(D2) {
      timing() {
        related_pin        : "CK";
        timing_type        : "hold_rising";
        fall_constraint(scalar) { values("0.0"); }
        rise_constraint(scalar) { values("0.0"); }
      }
      timing() {
        related_pin        : "CK";
        timing_type        : "setup_rising";
        fall_constraint(scalar) { values("0.0"); }
        rise_constraint(scalar) { values("0.0"); }
      }
    }
    pin(D3) {
      timing() {
      related_pin        : "CK";
      timing_type        : "hold_rising";
      fall_constraint(scalar) { values("0.1"); }
      rise_constraint(scalar) { values("0.1"); }
    }
      timing() {
        related_pin        : "CK";
        timing_type        : "setup_rising";
        fall_constraint(scalar) { values("0.1"); }
        rise_constraint(scalar) { values("0.1"); }
      }
    }
  }
```

```
pin(SE) {
  direction          : input;
  capacitance        :  1.0 ;
  related_power_pin  : VDD;
  related_ground_pin : VSS;
  timing() {
    related_pin        : "CK";
    timing_type        : "hold_rising";
    fall_constraint(scalar) { values("0.0"); }
    rise_constraint(scalar) { values("0.0"); }
  }
  timing() {
    related_pin        : "CK";
    timing_type        : "setup_rising";
    fall_constraint(scalar) { values("0.0"); }
    rise_constraint(scalar) { values("0.0"); }
  }
}

pin(SI) {
  direction          : input;
  capacitance        :  1.0 ;
  related_power_pin  : VDD;
  related_ground_pin : VSS;
  timing() {
    related_pin        : "CK";
    timing_type        : "hold_rising";
    fall_constraint(scalar) { values("0.0"); }
    rise_constraint(scalar) { values("0.0"); }
  }
  timing() {
    related_pin        : "CK";
    timing_type        : "setup_rising";
    fall_constraint(scalar) { values("0.0"); }
   rise_constraint(scalar) { values("0.0"); }
  }
}
statetable ( " D  CK  SE  SI " ,    "Q" ) {
  table :    " -  ~R   -   -  : - :  N,   \
               H/L  R   L   -  : - : H/L,  \
               -   R   H  H/L : - : H/L" ;
}

bundle(Q) {
  members(Q0, Q1, Q2, Q3)
  direction       : output;
  inverted_output    : false;
  internal_node : "Q" ;
  related_power_pin  : VDD;
  related_ground_pin : VSS;
```

```
    pin (Q0) {
      input_map : " D0  CK  SE  SI";
      timing() {
        related_pin        : "CK";
        timing_type        : "rising_edge";
        cell_fall(scalar) { values("0.0"); }
        fall_transition(scalar) { values("0.0"); }
        cell_rise(scalar) { values("0.0"); }
        rise_transition(scalar) { values("0.0"); }
      }
    }
    pin (Q1) {
      input_map : " D1  CK  SE  Q0";
      timing() {
        related_pin        : "CK";
        timing_type        : "rising_edge";
        cell_fall(scalar) { values("0.0"); }
        fall_transition(scalar) { values("0.0"); }
        cell_rise(scalar) { values("0.0"); }
        rise_transition(scalar) { values("0.0"); }
      }
    }
    pin (Q2) {
      input_map : " D2  CK  SE  Q1";
      timing() {
        related_pin        : "CK";
        timing_type        : "rising_edge";
        cell_fall(scalar) { values("0.0"); }
        fall_transition(scalar) { values("0.0"); }
        cell_rise(scalar) { values("0.0"); }
        rise_transition(scalar) { values("0.0"); }
      }
    }
    pin (Q3) {
      input_map : " D3  CK  SE  Q2";
      timing() {
        related_pin        : "CK";
        timing_type        : "rising_edge";
        cell_fall(scalar) { values("0.0"); }
        fall_transition(scalar) { values("0.0"); }
        cell_rise(scalar) { values("0.0"); }
        rise_transition(scalar) { values("0.0"); }
      }
    }
  }
  pin(SO) {
    direction       : output;
    state_function : "SE * Q1" ;
    related_power_pin  : VDD;
    related_ground_pin : VSS;
    test_output_only : true;
    timing() {
```

```
        related_pin        : "CK";
        timing_type        : "rising_edge";
        cell_fall(scalar) { values("0.0"); }
        fall_transition(scalar) { values("0.0"); }
        cell_rise(scalar) { values("0.0"); }
        rise_transition(scalar) { values("0.0"); }
      }
    }

    test_cell() {
      pin(CK){
        direction : input;
      }
      bundle(D) {
        members(D0, D1, D2, D3);
        direction : input;
      }
      pin(SI) {
        direction : input;
        signal_type : "test_scan_in";
      }
      pin(SE) {
        direction : input;
        signal_type : "test_scan_enable";
      }
      ff_bank (IQ,IQN,4) {
        next_state : "D";
        clocked_on : "CK";
      }
      bundle(Q) {
        members(Q0, Q1, Q2, Q3);
        direction : output;
        function  : "IQ";
      }
      pin(SO) {
        direction : output;
        signal_type : "test_scan_out";
        test_output_only  : "true";
      }
    }
  } /* end cell group */
} /* end library group */
```

B

Multibit Mapping Guidance Files

The multibit banking flow uses the input map and register group files for mapping guidance files—both are plain ASCII text files.

The following sections explains how you can use these types of mapping guidance files:

- Input Map File – Specifies which multibit library cells are used to replace a given number of single-bit cells.
- Register Group File – Defines groups of related single-bit and multibit library cells, so that only single-bit cells of a given type are grouped together, and these groups are replaced only by multibit cells of the same type.

Input Map File and Register Group File

In Design Compiler Graphical, specifying these files is optional. If you do not specify the files, the `identify_register_banks` command identifies the single-bit registers that can be replaced by available multibit registers in the target and link libraries based on the functional information of the library cells; the tool then uses that information to control mapping. To see which single-bit registers the tool has identified for replacement by multibit registers and guide the tool to perform multibit mapping, see the "Examining the Guidance Files and Controlling Mapping in Design Compiler" section.

The following sections explain how to use the mapping guidance files in the multibit banking flow.

Input Map File

The input map file specifies which multibit library cells are to be used to replace a given number of single-bit cells. The file consists of multiple lines of text, each in the following form:

```
number_of_bits {number_of_instances multibit_lib_cell_name} [ { ... } ]
```

For example,

```
2 {1 dff_2bit}    ;map 2 single bits to 1 dff_2bit register
3 {1 dff_2bit}    ;map 3 single bits to 1 dff_2bit register and 1 single
4 {1 dff_4bit}    ;map 4 single bits to 1 dff_4bit register
```

Feedback

```
5 {1 dff_4bit}   ;map 5 single bits to 1 dff_4bit register and 1 single
6 {1 dff_4bit} {1 dff_2bit} ;map 6 single bits to 1 dff_4bit register
                            ;and 1 dff_2bit register
7 (1 dff_4bit} {1 dff_2bit} ;map 7 single bits to 1 dff_4bit register
                            ;and 1 dff_2bit register and 1 single
8 {2 dff_4bit}              ;map 8 single bits to 2 dff_4bit registers
...
```

The list should continue up to the maximum number of single register bits that you want the tool to group into multibit registers, typically 32.

By default, all single-bit registers can be mapped to any of the specified multibit registers. If you want the tool to group together single-bit registers of the same type, and to replace such groups only with multibit cells of the same type, you need to create a register group file as described in the next section, “Register Group File.”

If the library has different types of multibit registers that differ only in their physical size and drive strength, the input map file should list only the smallest-area library cells. In that case, the Design Compiler Graphical tool tool always replaces single-bit cells with the smallest-area multibit cells. During subsequent optimization, the tool might resize these multibit cells to larger, stronger-drive cells where needed to meet the timing constraints.

Specifying only the smallest-area multibit register of each bit-width is recommended. If you want to use different multibit registers of the same bit-width, you need to repeat each line for each available library cell. For example,

```
2 {1 dff_2bitA}  ;map 2 single bits to 1 dff_2bitA register, or
2 {1 dff_2bitB}  ;map 2 single bits to 1 dff_2bitB register
3 {1 dff_2bitA}  ;map 3 single bits to 1 dff_2bitA register + single, or
3 {1 dff_2bitB}  ;map 3 single bits to 1 dff_2bitB register + single
4 {1 dff_4bitA}   ;map 4 single bits to 1 dff_4bitA register, or
4 {1 dff_4bitB}   ;map 4 single bits to 1 dff_4bitB register
...
```

When you specify multiple combinations for the same number of bits, the tool uses the first definition and ignores the subsequent definitions. In the following example, the tool uses 1 dff_4bitA for 4-bit registers from the first line, ignoring the definition in the second line.

```
4 {1 dff_4bitA} ;map 4 single bits to 1 dff_4bitA register
4 {2 dff_2bitA} ;map 4 single bits to 2 dff_2bitA register
```

Register Group File

The register group file specifies which registers are allowed to be grouped together and which multibit registers can be used for replacement in each group.

The file consists of multiple lines of text, each in the following form:

reg_group_name number {1_bit_lib_cells} {multibit_lib_cells}

Feedback

where *number* is the number of single-bit library cells listed in the following braces.

For example,

```
reg_group_plain       3 {dff_A dff_B dff_C}     {dff_2bit dff_4bit}
reg_group_scan        3 {sdff_A sdff_B sdff_C}  {sdff_2bit sdff_4bit}
reg_group_Reset       2 {Rdff_A Rdff_B}         {Rdff_2bit Rdff_4bit}
reg_group_Reset_scan  2 {Rsdff_A Rsdff_B}       {Rsdff_2bit Rsdff_4bit}
```

The first line specifies that instances of the `dff_A`, `dff_B`, and `dff_C` single-bit library cells can be grouped together in any combination (but not with other types of single-bit cells), and these groups can be replaced only by instances of the `dff_2bit` and `dff_4bit` multibit library cells. The other three lines each define a register group for scan cells, for cells with reset inputs, and for scan cells with reset inputs, respectively.

This register group file, together with the following input map file, fully describe the allowed grouping and mapping of single-bit registers to multibit registers.

```
; input map file
2 {1 dff_2bit}
2 {1 sdff_2bit}
2 {1 Rdff_2bit}
2 {1 Rsdff_2bit}

3 {1 dff_2bit}
3 {1 sdff_2bit}
3 {1 Rdff_2bit}
3 {1 Rsdff_2bit}
...
6 {1 dff_2bit 1 dff_4bit}
6 {1 sdff_2bit 1 sdff_4bit}
6 {1 Rdff_2bit 1 Rdff_4bit}
6 {1 Rsdff_2bit 1 Rsdff_4bit}
...
32 {8 dff_4bit}
32 {8 sdff_4bit}
32 {8 Rdff_4bit}
32 {8 Rsdff_4bit}
```

A register group file is necessary to get optimum banking ratios.

Examining the Guidance Files and Controlling Mapping in Design Compiler

To see which single-bit registers the tool has identified for replacement by multibit registers and guide the tool to perform multibit mapping in Design Compiler Graphical, use the `write_multibit_guidance_files` command. The `write_multibit_guidance_files` command uses the same process as the `identify_register_banks` command to identify

the multibit and single-bit registers and then generates the following guidance files based on the functional information of the library cells:

- multibit_guidance.input_map (input map file)
- multibit_guidance.register_group (register group file)

Examine these files to see which multibit cells are available in the library and which single-bit cells can be replaced by multibit cells.

If you do not need to change the way these files define multibit mapping, you do not need to specify the input map file and the register group file when you run the `identify_register_banks` command. In this case, Design Compiler automatically uses the guidance from the files generated by the `write_multibit_guidance_files` command. To change the way multibit mapping is performed, modify these guidance files and provide additional multibit mapping guidance. You must specify the modified files when you run the `identify_register_banks` command.

C

Reporting Effective Banking Ratio

This appendix shows how to use the script to report the effective banking ratio in your design.

The effective banking ratio is determined by the total number of registers banked divided by the total number of registers in the design - the total number of ignored registers in the design, as shown in the following equation:

$$\text{Effective banking ratio} - \frac{\text{Total number of registers banked}}{(\text{Total number of registers in the design} - \text{Total number of ignored registers in the design})}$$

The following script reports effective banking ratio for both RTL inference flow and placement-aware multibit banking. You can source the script after completing banking of mulitibit registers.

```
set mb_lib_cell_list [remove_from_collection [get_lib_cells
-f "multibit_width >1 && is_sequential == true" */*] [get_lib_cells
-f "multibit_width >1" gtech/*]]
set bit_width [get_attribute $mb_lib_cell_list multibit_width]

set num_of_mb_bit 0
set num_of_mb_cell 0
foreach mb_size [lsort -unique $bit_width] {
 set cell_name [get_attribute [get_lib_cells -f "multibit_width ==
 $mb_size && is_sequential == true" */*] name]
 foreach cell_name_unique [lsort -unique [lsearch -all -inline -not
 $cell_name GTECH*]] {
  set num_of_mb_temp [sizeof [get_flat_cell -f "ref_name ==
  $cell_name_unique"]]
  if {$num_of_mb_temp != 0} {
  set num_of_mb_bit [expr ($num_of_mb_temp * $mb_size) +
  $num_of_mb_bit]
  set num_of_mb_cell [expr ($num_of_mb_temp + $num_of_mb_cell)]
     }
  }
}

set num_of_ff [sizeof [all_registers ]]
set num_of_sb [expr $num_of_ff - $num_of_mb_cell]
puts "  "
puts "Total number of sequential cells: $num_of_ff"
```

```
puts "    (x) Number of single-bit sequential cells: $num_of_sb"
puts "    (y) Number of multi-bit sequential cells: $num_of_mb_cell"
puts "  "

set num_of_bit [expr $num_of_mb_bit + $num_of_sb]
set pack_ratio [expr ($num_of_mb_bit*100 /  $num_of_bit)]
redirect -var rpt_cell {report_cell [get_flat_cells -filter
 "is_sequential == true"]}
set num_of_rejected_bit [regexp -all -line -- {n,.*r\d} $rpt_cell]
set effective_pack_ratio [expr ($num_of_mb_bit*100 /  ($num_of_bit -
$num_of_rejected_bit))]
puts "  "
puts "Total number of single-bit equivalent sequential cells:
$num_of_bit"
puts "    (X) Single-bit sequential cells: $num_of_sb"
puts "    (Y) Multi-bit sequential cells: $num_of_mb_cell"
puts "  "

puts "(Z) Total number of rejected single-bit cells:
$num_of_rejected_bit"
puts "  "

puts "  Banking ratio ( Y / (X + Y)): $pack_ratio %"
puts "  Effective Banking Ratio ( Y / (X + Y - Z)): $effective_pack_ratio
%"
```