Multibit Register Synthesis and Physical Implementation Application Note

Version L-2016.03-SP4, September 2016

SYNOPSYS®

Copyright Notice and Proprietary Information

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 http://www.synopsys.com/Company/Pages/Trademarks.aspx.
All other product or company names may be trademarks of their respective owners.

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.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Contents

1

Multibit Register Synthesis and Physical Implementation

The Design Compiler, DC Explorer, and IC Compiler 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.

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

- Multibit Register Overview
- Library Requirements
- RTL Bus Inference Flow
- Placement-Aware Multibit Register Banking
- Multibit Registers in Galaxy Flows
- Reporting Multibit Registers in the Design

Multibit Register Overview

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.

In the Synopsys Galaxy Implementation Platform, 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.

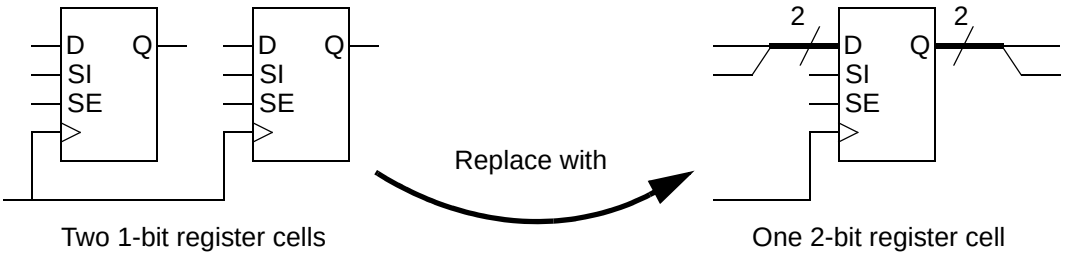
Replacing single-bit cells with multibit cells offers the following benefits:

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

Figure 1-1 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 Galaxy Implementation Platform 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` command.

- **Placement-aware register banking flow** in the Design Compiler Graphical and IC Compiler 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.

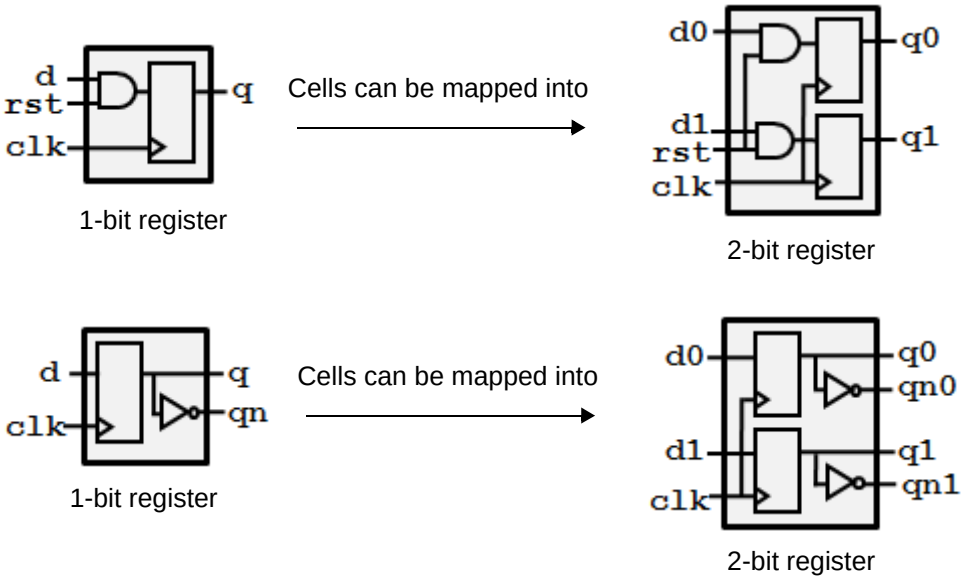
Both types of flows support scan chain generation and design-for-test protocols. A design modified by mapping single-bit registers to multibit registers can be formally verified with the Formality tool.

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.

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 1-2.

Figure 1-2 Mapping of Single-Bit to Multibit Cells With the Same I/O Pins



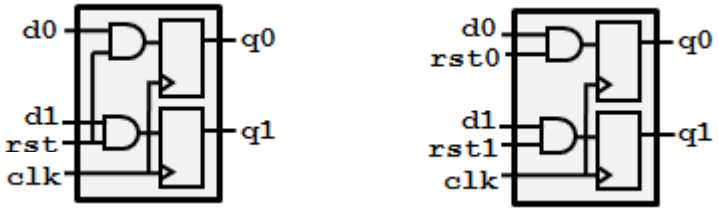
For example, if a single-bit register has Q and QN outputs, the tool can replace this register only with a multibit register that also has Q and QN outputs for each output register bit.

In the RTL inference flow, the Design Compiler tool matches the single-bit cells with the multibit cell using the functionality of the cell in the library. The tool can match single-bit registers and multibit cells with different pin names.

In the placement-aware flow,

- The Design Compiler Graphical and IC Compiler tools do not look at the functionality of the cells. The tools use the pin names of the cell to match the single-bit cells and the multibit cell. Single-bit registers and multibit registers must have the same pin names. For the pin name of bused or bundled input and output pins, use the pin name of the single-bit register as a base name followed by consecutive numbers. For example, if the single-bit pin name is D, the multibit pin names can be D0, D1, D2, or D[0], D[1], D[2].
- The Design Compiler Graphical and IC Compiler tools do not support multibit registers that have a control pin for an individual bit, as shown in Figure 1-3.

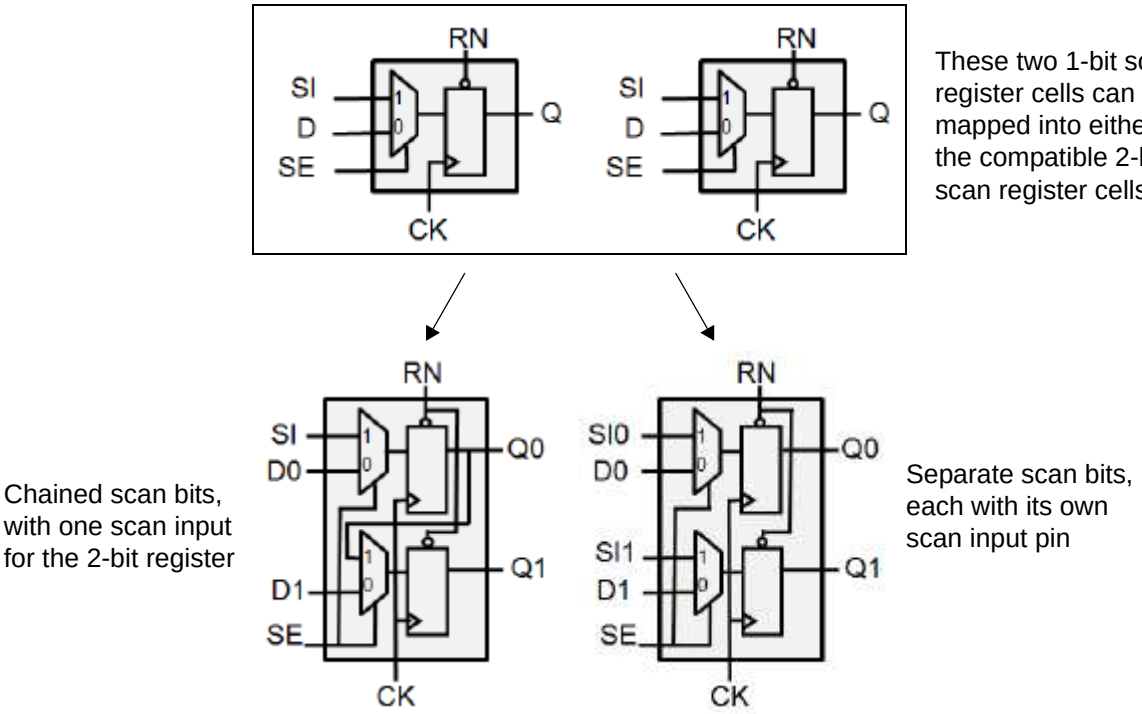
Figure 1-3 Multibit Registers With a Control Pin For an Individual Bit Are Not Supported



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 1-4 shows how two single-bit scan cells can be mapped into either of two different compatible multibit scan cells, which have different scan configurations.

Figure 1-4 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 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 SolvNet:

- 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 during execution of the `compile_ultra` command.

The Design Compiler and DC Explorer tool 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 in the RTL, set the `hdlin_infer_multibit` variable to specify how the tool performs multibit register inference from the RTL buses.
- After you read in the RTL, use the `create_multibit` command to group specified register bits into multibit components.

You can use the `report_multibit` command to generate reports on the multibit components in the design, both before and after using the `compile_ultra` command:

- Before a compile operation, the `report_multibit` command reports the groupings of single-bit register cells resulting from inference from the RTL and by usage of the `create_multibit` command.
- After a compile operation, 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.

Commands used in the RTL bus inference flow do not work in the placement-aware multibit flow.

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`.

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.

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 recreate 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
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 the following table control the naming of multibit registers in the RTL inference flow.

Table 1-1 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 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, you can cancel that grouping by using the `remove_multibit` command. 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 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 Options

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.

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.

  Note:

  Multibit optimization of retention registers is supported in Design Compiler only.

To guide the `compile_ultra` command in deciding when 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 strings:

- 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_driven – The tool replaces single-bit cells with multibit cells only where timing and area are not made any worse.
- timing_only – The tool replaces single-bit cells with multibit cells only when timing is not made any worse. The area is allowed to become 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.

Placement-Aware Multibit Register Banking

In the placement-aware register banking flow, the Design Compiler Graphical or IC Compiler 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.

The Design Compiler Graphical and IC Compiler tools use a map file to specify the mapping of single-bit registers to multibit registers. The map file contains lines of text similar to the following:

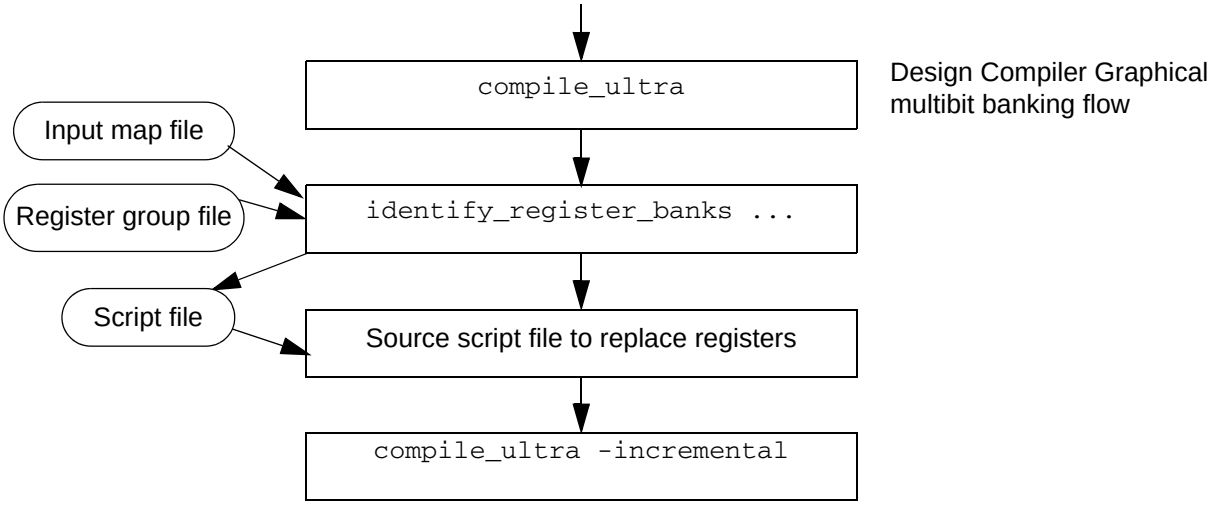
```
4 {1 MREG4}
6 {1 MREG2}{1 MREG4}
...
```

In this example, the first line says that any four compatible single-bit registers can be mapped into one MREG4 type multibit register. The second line says that any six compatible single-bit registers can be mapped into one MREG2 type register and one MREG4 type register.

After you create the map file, you can use it to guide the Design Compiler Graphical tool or IC Compiler tool in mapping single-bit to multibit registers.

Figure 1-5 summarizes the flow in the Design Compiler Graphical tool.

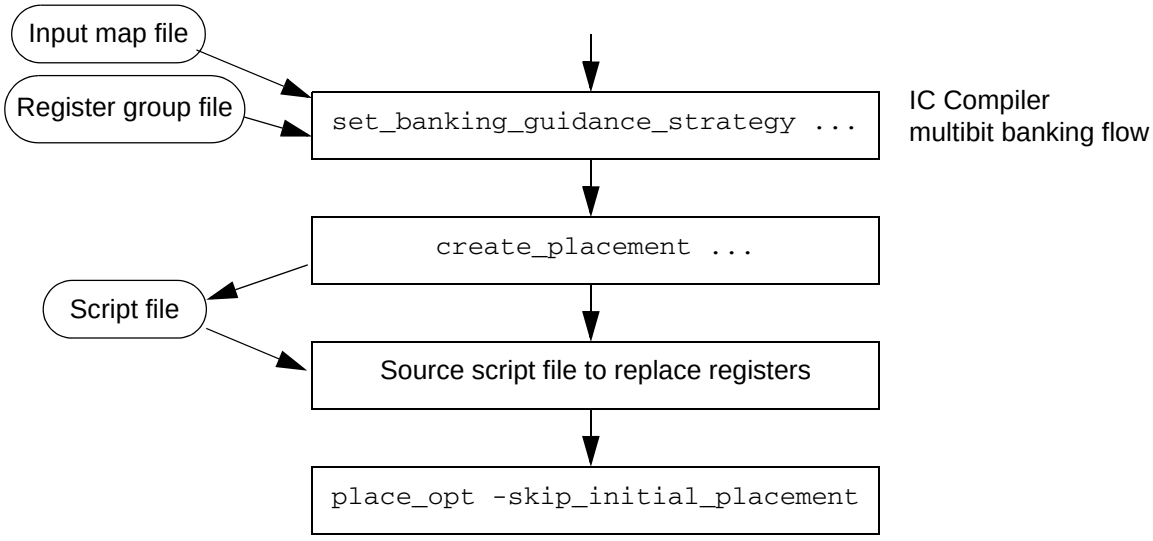
Figure 1-5 Multibit Banking Flow in the Design Compiler Graphical Tool



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.

Figure 1-6 summarizes the flow in the IC Compiler tool.

Figure 1-6 Multibit Banking Flow in the IC Compiler Tool



In the IC Compiler tool, the `set_banking_guidance_strategy` command finds compatible groups of single-bit registers that are physically near each other, similar to the `identify_register_banks` command in the Design Compiler Graphical tool. However, it does not directly write out a `create_register_bank` script file. Instead, it prepares the subsequent placement command for automatic generation of the script file.

When you run a placement command, such as `create_placement`, the IC Compiler tool writes out the script file containing `create_register_bank` commands. After placement is complete, you can execute the script to perform the mapping of single-bit to multibit registers. Then, you can use the `place_opt -skip_initial_placement` command to incrementally optimize the placement of the multibit registers.

For designs that have already been placed, you can bypass the placement step and use the `create_banking_guidance` command after the `set_banking_guidance_strategy` command to generate the script file containing the `create_register_bank` commands.

Input Map File and Register Group File

The multibit banking flow uses two different types of mapping guidance files—both are plain ASCII text files:

- Input Map File – Specifies which multibit library cells will be 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.

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.

In IC Compiler, you must specify at least the input map file name. A register group file is necessary to get optimum banking ratios.

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
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 or IC Compiler 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*}

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.

To use these guidance files in IC Compiler, specify them with the `set_banking_guidance_strategy` command.

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 or the `set_banking_guidance_strategy` command in the IC Compiler tool.

The Design Compiler Graphical command performs the banking analysis and generates the script file immediately, whereas the IC Compiler command merely specifies the strategy, which is carried out later by the placement command.

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.
- Are in the same scan chain

During grouping, Design Compiler and IC Compiler generate the following message:

```
MB8SDFF_1              : 584
MB4SDFF_1              : 149
MB2SDFF_1              : 169
Excluded flops         : 80
Total flops banked     : 5606
Total flops in design  : 6064
Banking ratio          : 92.45%
```

Note:

The 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 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 Design Compiler, to get the banking ratio, source the command file generated by register grouping and then run the `report_multibit_banking` command.

In IC Compiler, to get the banking ratio, source the command file generated by register grouping and then use the Tcl script in the "Reporting Multibit Registers in the Design" section.

identify_register_banks Command (Design Compiler)

The `identify_register_banks` command assigns the single-bit registers in the design into groups according to the input map file and register group file. 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.

This is the full syntax of the `identify_register_banks` command:

```
identify_register_banks
  [-input_map_file file_name]
  -output_file file_name
  [-register_group_file file_name]
  [-maximum_flop_count integer]
  [-minimum_flop_count integer]
  [-exclude_instances exclude_cells]
  [-wns_threshold percentage]
  [-wns_threshold_file file_name]
  [-common_net_pins names]
  [-name_prefix prefix]
  [-exclude_library_cells library_cells]
  [-exclude_size_only_flops true | false]
  [-exclude_start_stop_scan_flops true | false]
  [-multibit_components_only]
```

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. You do not need to specify the `-common_net_pins` option.

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.

set_banking_guidance_strategy Command (IC Compiler)

The `set_banking_guidance_strategy` command prepares the `create_placement` or `create_banking_guidance` command for identifying banking opportunities. It specifies the name of the input map file, output script file, and other parameters used to guide the banking process.

This is the full syntax of the `set_banking_guidance_strategy` command:

```
set_banking_guidance_strategy
  -input_map_file file_name
  -output_file file_name
  [-register_group_file file_name]
  [-maximum_flop_count integer]
  [-minimum_flop_count integer]
  [-exclude_instances exclude_cells]
  [-wns_threshold percentage]
  [-wns_threshold_file file_name]
  [-common_net_pins names]
  [-name_prefix prefix]
  [-exclude_library_cells library_cells]
  [-exclude_size_only_flops true | false]
  [-exclude_start_stop_scan_flops true | false]
```

You must specify at least the input map file name and the name of the banking script file written out by the placement command or the `create_banking_guidance` command. A register group file is necessary to get optimum banking ratios.

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 scan chain start/stop attribute. For details, see the man page for the `set_banking_guidance_strategy` command.

To view the current register banking settings, use the following command:

```
icc_shell> report_banking_guidance_strategy
...
    input_map_file           : my_map_file.map
    output_file              : my_banking.tcl
    maximum_flop_count       : 16
    minimum_flop_count       : 2
    register_group_file      : my_group_file.grp
    exclude_instances        : U_179 U_259 U_334 U_227 U_238
    wns_threshold            : 50.000000
    common_net_pins          : CP RN
    name_prefix              : groupA
    wns_threshold_file       :
    exclude_library_cells    :
```

To remove the current register banking settings, use the following command:

```
icc_shell> remove_banking_guidance_strategy
```

The register banking settings that you specify apply only to the current tool session.

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 new `-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 1-8.

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 1-7 shows the single session flow (running the `identify_register_banks` command in the same session rather than in a subsequent Design Compiler session).

Figure 1-7 Single Session Flow

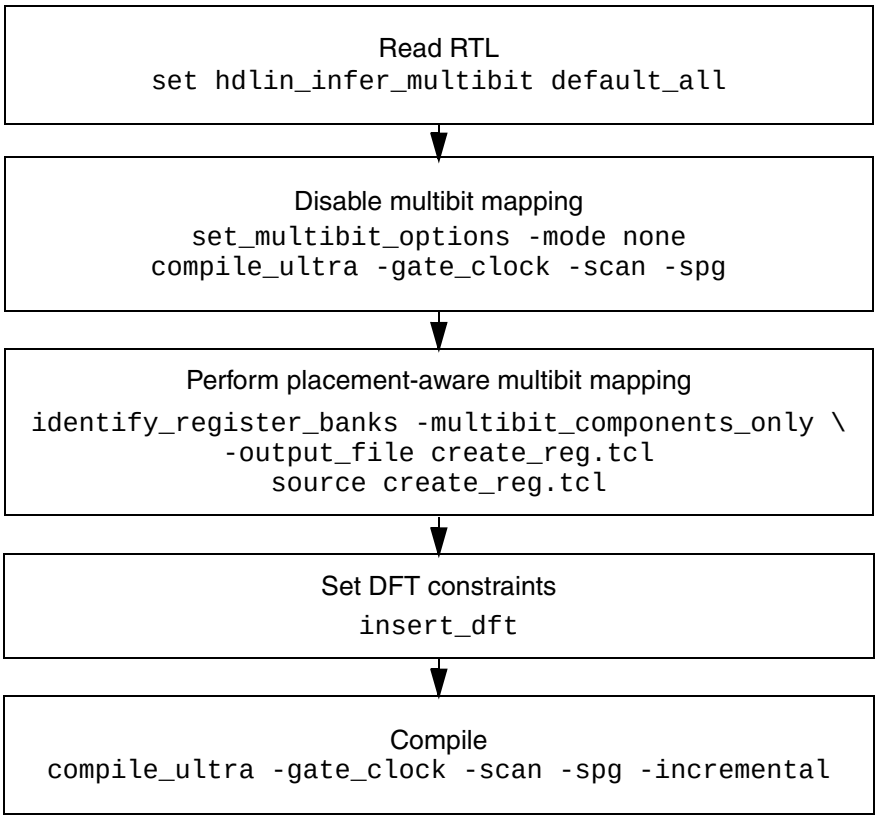
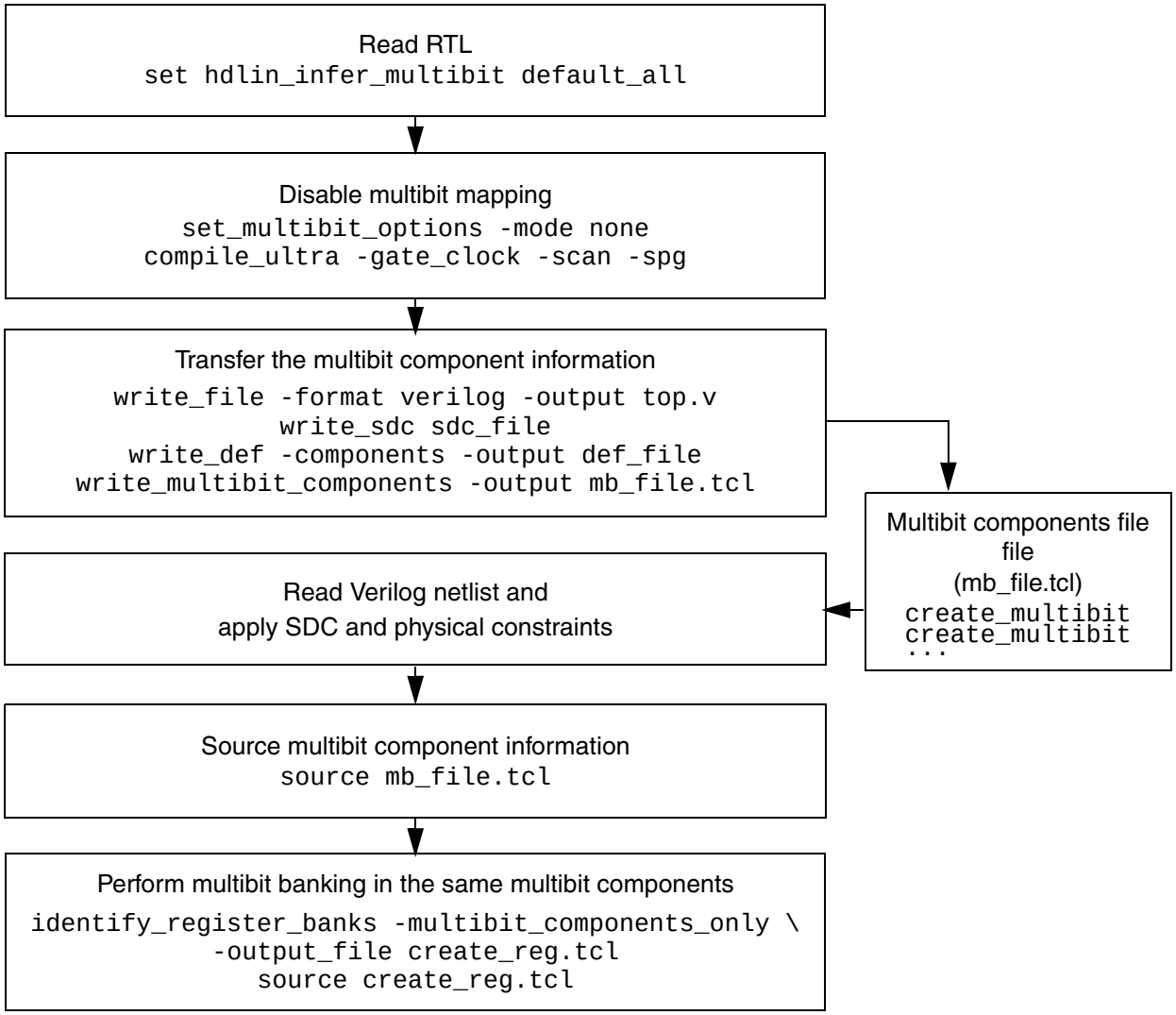


Figure 1-8 shows an alternative flow where you preserve the multibit components in ASCII format and then run the `identify_register_banks` command in a subsequent Design Compiler session.

Figure 1-8 Two Session Flow With an ASCII Handoff



Multibit Register Naming Style

In the Design Compiler Graphical and IC Compiler 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 or set_banking_guidance_strategy 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 or IC Compiler J-2014.09-SP4 releases, set the banking_enable_concatenate_name variable to false.

Banking and Debanking Registers

In both the Design Compiler Graphical and IC Compiler tools, 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. If the result of banking is not satisfactory, you can split a multibit register into smaller registers by using the split_register_bank command.

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.

This is the syntax of the command:

```
create_register_bank
  single_bit_register_object_list
  [-name bank_name]
  -lib_cell library_name/multibit_lib_cell_name
```

Specifying the library name in the -lib_cell argument is mandatory in the Design Compiler tool and optional in the IC 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 1-9 Multibit Banking

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

- Have the same input and output pins
- 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
- Are in the same scan chain

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)
```

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.

split_register_bank Command

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}
```

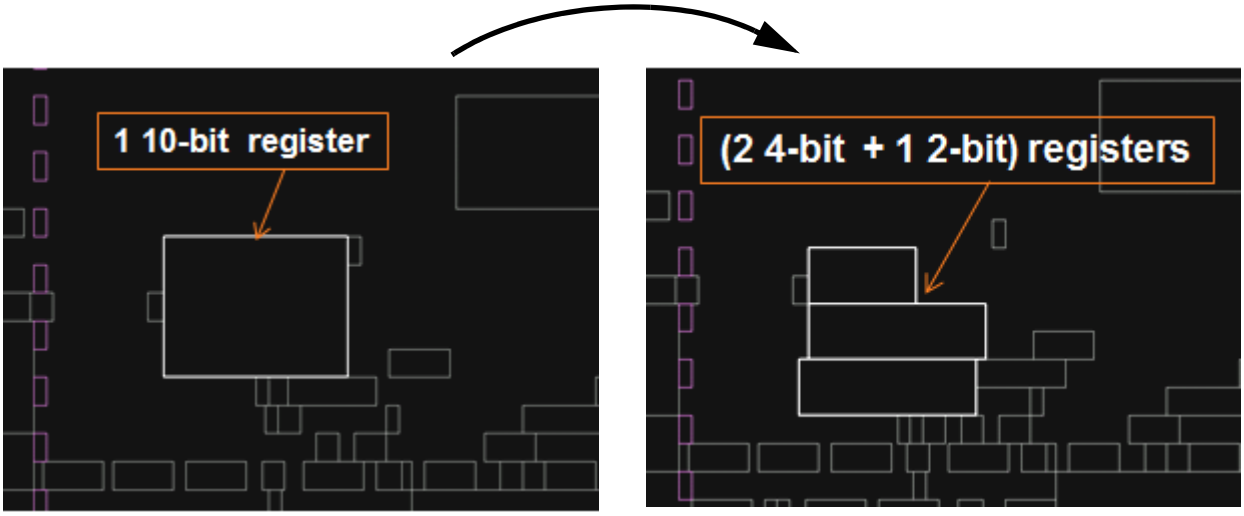
Specifying the library name in the `-lib_cell` argument is mandatory in the Design Compiler tool and optional in the IC Compiler tool.

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 two bits of the 4-bit register with an instance of the mreg_2bit library cell, and replaces the remaining two bits with two instances of the mreg_1bit library cell. It breaks the connections to the removed 4-bit cell and appropriately reconnects the new 2-bit and single-bit cells.

Figure 1-10 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

When you pass the netlist from Design Compiler to IC Compiler, the `register_list` attribute that is set on the multibit register is stored in the .ddc or Milkyway file. When you run the `split_register_bank` command in IC Compiler, the command uses the single-bit cell names listed in the `register_list` attribute.

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}
```

Multibit Registers in Galaxy Flows

The following sections demonstrate the full multibit synthesis, implementation, and verification flows across Galaxy Design Platform tools:

- DFT and Multibit Registers
- Clock Gating in the Multibit Flow
- Multibit SAIF Flow
- Multicorner-Multimode Flow
- Design Verification Flow in Formality
- Synopsys Physical Guidance Flow
- Placement-Aware Flow in IC Compiler

DFT and Multibit Registers

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

In Design Compiler, 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. Both banking and debanking on a scan stitched design are not supported in Design Compiler. To perform multibit mapping after scan insertion, use IC Compiler.

When you use 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`

  (default: `false` starting from version I-2013.12-SP2)

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

- `set_scan_configuration -preserve_multibit_segment false`

  (default: `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.

In IC Compiler, you can perform banking or debanking after scan insertion. After registers in the scan chain are replaced, the tool automatically reconnects the scan chain. You must run the `check_scan_chain` command after banking or debanking to update the SCANDEF.

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.

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 ungated single-bit registers and 16 of them 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*.

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, DFT Compiler, Power Compiler, and IC Compiler.

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

In both Design Compiler and IC Compiler, 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 Design Compiler, generate the verification guidance file with the `set_svf` command before reading the RTL. In IC Compiler, use the `set_svf` command before using the banking or debanking command. When you verify the design, use the .svf file generated from Design Compiler or IC Compiler.

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

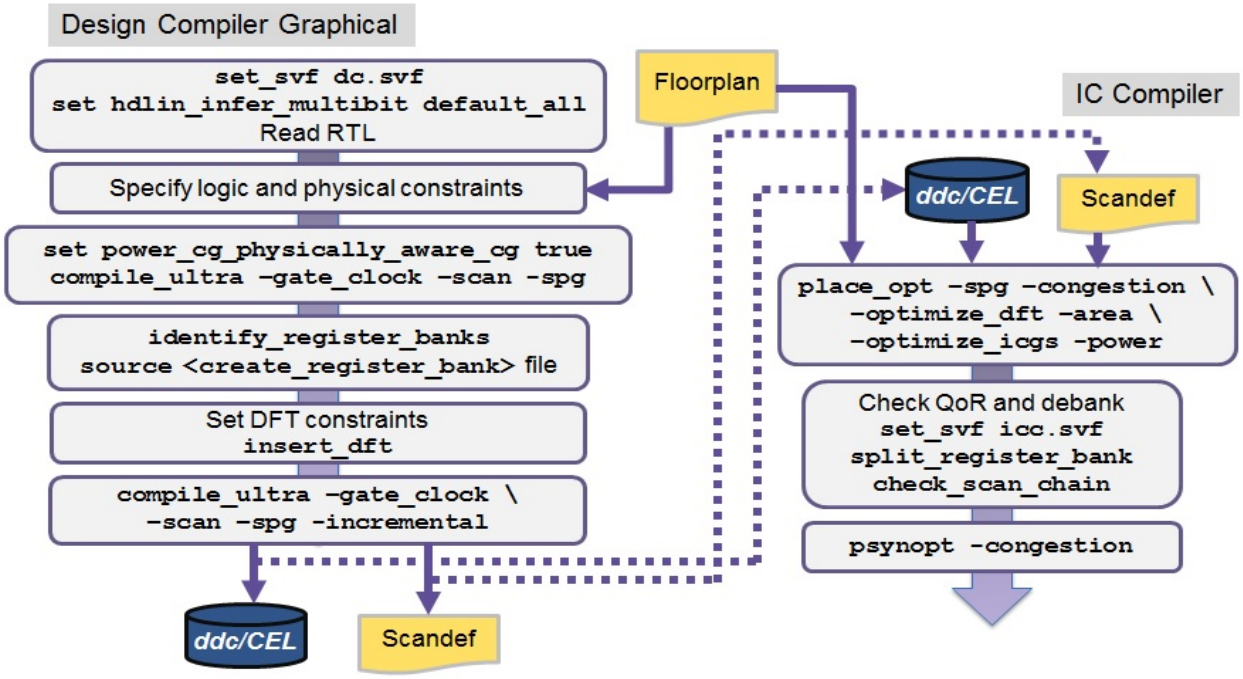
For more information about the checkpoint guidance flow, see SolvNet article 1703130, "Formality Verification Flow Using Checkpoint Guidance."

Synopsys Physical Guidance Flow

In the Synopsys physical guidance flow, the Design Compiler Graphical tool performs refined placement, delay optimization, and routability optimization to provide a better, more highly correlated starting point for physical implementation by the IC Compiler tool. You invoke this flow by using the `-spg` option with the `compile_ultra` command in the Design Compiler tool and also using the `-spg` option with the `place_opt` command in the IC Compiler tool.

Figure 1-11 shows the typical Synopsys physical guidance flow with multibit register synthesis. This flow uses both RTL bus inference and placement-aware register banking in the Design Compiler Graphical tool, and multibit register splitting for QoR improvement in the IC Compiler tool.

Figure 1-11 Synopsys Physical Guidance Flow With Multibit Register Synthesis



In the Synopsys physical guidance flow, perform the following steps in Design Compiler Graphical:

1. Create the input map file and the register group file as described in the section "Input Map File and Register Group File." These files specify the manner of mapping from single-bit to multibit cells.
2. Enable bus-based multibit optimization:

   ```
   dc_shell-topo> set hdlin_infer_multibit default_all
   ```

3. Read in the RTL for the design.
4. Apply the logical and physical constraints for the design, including the clock constraints.
5. Perform an initial compile operation:

   ```
   dc_shell-topo> compile_ultra -scan -gate_clock -spg ...
   ```

   Scan replacement is not supported after registers are mapped to multibit registers. You must use the `-scan` options with the `compile_ultra` command before multibit register banking. To get the optimal clock-gating and multibit replacement ratios, use the `-gate_clock` option during the initial compile.

6. Assign the single-bit registers into groups by using the `identify_register_banks` command:

   ```
   dc_shell-topo> identify_register_banks \
                  -output_file my_reg_file.tcl \
                  ...
   ```

   The command generates a script file containing `create_register_bank` commands that perform the actual mapping of single-bit to multibit registers. You can view and optionally modify this script file before you execute it.

7. Execute the script file created in the previous step:

   ```
   dc_shell-topo> source my_reg_file.tcl
   ```

   This replaces the groups of single-bit registers with multibit registers in the netlist.

   A this point, you can optionally break up multibit registers into smaller multibit registers or single-bit registers by using the `split_register_bank` command.

   After multibit register banking, you should run an incremental compile to adjust the register locations and sizing. This incremental compile can be done before or after you perform DFT scan insertion in Step 8.

8. Perform DFT scan chain insertion:

   ```
   dc_shell-topo> insert_dft
   ```

   In Design Compiler, you must run scan insertion after completing all multibit replacement. Running scan insertion before multibit mapping is not supported.

9. Perform an incremental compile operation:

   ```
   dc_shell-topo> compile_ultra -incremental -scan -gate_clock -spg
   ```

   This performs logic and timing optimization on the modified, now containing multibit registers in place of single-bit registers.

The following example shows a script for the Design Compiler Graphical tool in the Synopsys physical guidance flow:

```
# Set Formality guidance file for initial compile operation
set_svf svf1.svf

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

# Read the RTL design
# Apply logical and physical constraints

# Enable Placement-aware clock-gating
set power_cg_physically_aware_cg true
```

```
# Initial compile
compile_ultra –scan –gate_clock -spg
write_file –hierarchy –format ddc –output initial_compile.ddc

# Multibit register banking
identify_register_banks -input_map_file input.map \
  -register_group_file reg_group.txt \
  -output_file create_reg_bank.tcl \
source create_reg_bank.tcl

# DFT scan insertion
set_scan_configuration ...
insert_dft

# Incremental compile
compile_ultra –scan –incremental –gate_clock -spg
write_scan_def –output mapped.scandef
check_scan_def
write_file –hierarchy –format ddc –output mapped.ddc
```

The following example shows a script for the IC Compiler tool in the Synopsys physical guidance flow:

```
# Open Milkyway design
open_mw_cel my_design –library my_mw_lib.mw

# Read floorplan
read_def fp.def

# Run placement optimization with the -spg option and other
# recommended options
place_opt -spg -congestion -optimize_dft -area_recovery -optimize_icgs \
   -power

# Selectively split banked register to smaller registers (optional)
set_svf debanking.svf
split_register_bank -lib_cells {REG4 REG4 REG2}
check_scan_chain

# Continue flow
...
clock_opt ...
...
route_opt ...
...
```

Placement-Aware Flow in IC Compiler

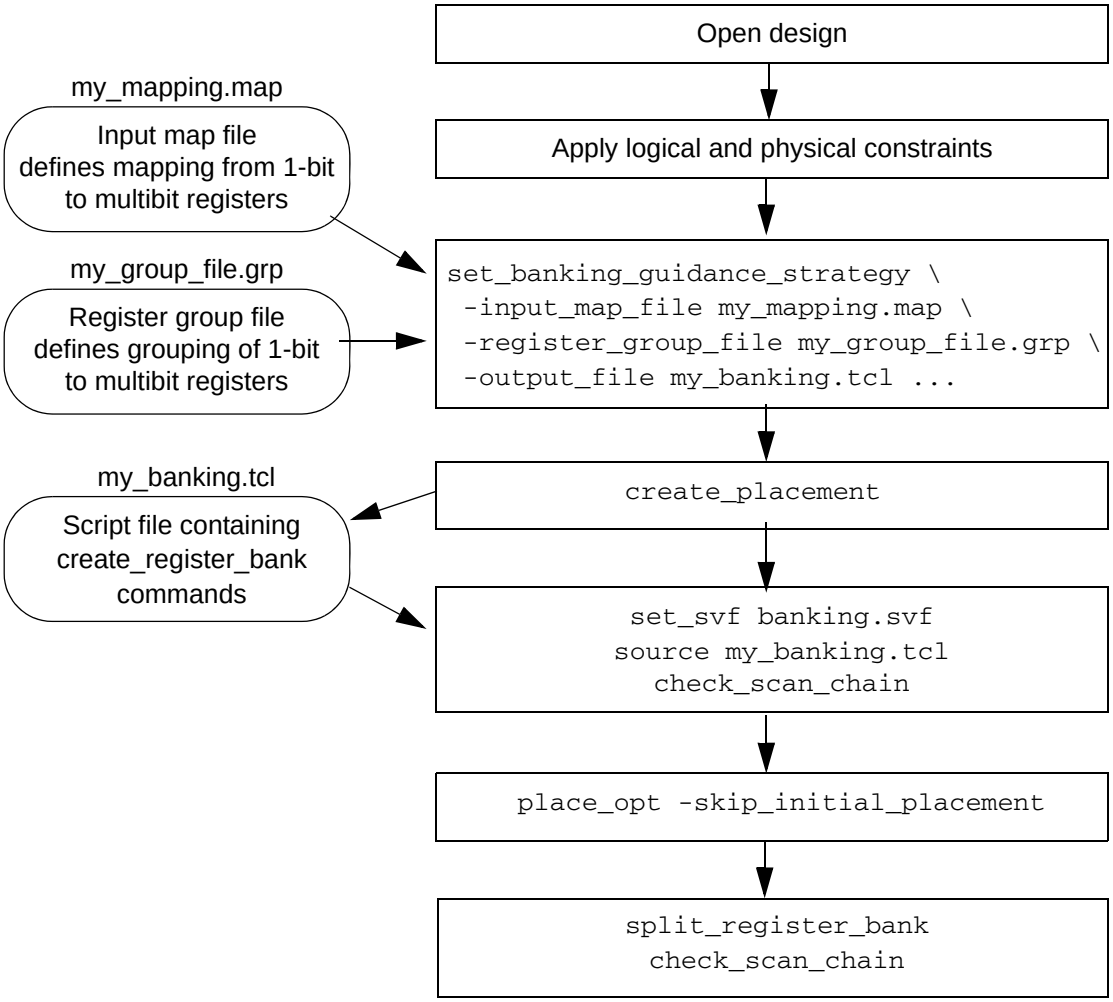
In the Synopsys physical guidance flow, the Design Compiler Graphical tool performs placement-aware multibit banking and the IC Compiler tool performs QoR-based debanking. Alternatively, you can perform placement-aware multibit banking entirely in the IC Compiler tool.

In the IC Compiler tool, the `set_banking_guidance_strategy` command prepares the `create_placement` command for identifying banking opportunities. The placement command, along with performing placement, also writes out a script file to perform the mapping of single-bit to multibit registers. After you execute the script, you optimize the modified design by running an incremental `place_opt` command.

For designs that have already been placed, you can bypass the placement step and use the `create_banking_guidance` command after the `set_banking_guidance_strategy` command to generate the script file that performs the mapping of single-bit to multibit registers.

The multibit banking flow in the IC Compiler tool is shown in Figure 1-12.

Figure 1-12 Multibit Banking Flow in IC Compiler Tool



These are the steps in the IC Compiler multibit banking flow:

1. Create the input map file and the register group file as described in the section "Input Map File and Register Group File." These files specify the manner of mapping from single-bit to multibit cells.
2. Open the Milkyway design.
3. Apply the logical and physical constraints for the design, including the clock constraints.

4. Specify the banking strategy for placement by using the `set_banking_guidance_strategy` command:

```
icc_shell> set_banking_guidance_strategy \
             -input_map_file my_map_file.map \
             -register_group_file my_group_file.grp \
             -output_file my_reg_file.tcl \
             ...
```

   The command has options to restrict the number of registers allowed in each group and to exclude some cell instances from banking by name, timing slack, or register cell characteristics.

5. Execute the `create_placement` command, which not only performs placement, but also generates a script file containing `create_register_bank` commands that perform the mapping of single-bit to multibit registers. You can view and optionally modify this script file before you execute it.

6. Enable the Formality setup guidance file:

```
icc_shell> set_svf banking.svf
```

7. Execute the script file created in the previous step:

```
icc_shell> source my_reg_file.tcl
```

   This replaces the groups of single-bit registers with multibit registers in the netlist.

   A this point, you can optionally break up multibit registers into smaller multibit registers or single-bit registers by using the `split_register_bank` command.

8. Update scan chains by running the `check_scan_chain` command:

```
icc_shell> check_scan_chain
```

9. Optimize the multibit register placement:

```
icc_shell> place_opt -skip_initial_placement
```

10. Check the QoR. If the multibit register creates timing or congestion issues, debank registers, then update the scan chains:

```
icc_shell> split_register_bank
icc_shell> check_scan_chain
```

11. Continue with the physical implementation flow:

```
icc_shell> clock_opt
...
icc_shell> route_opt
...
```

Script Example

The following script example demonstrates the placement-aware multibit banking flow in the IC Compiler tool.

```
# Open Milkway design
open_mw_cel my_design –library my_mw_lib.mw

# Set banking strategy
set_banking_guidance_strategy \
  -input_map_file my_mapping.map \
  -register_group_file reg_group.txt \
  -output_file my_banking.tcl

# Perform placement and generate banking script file
create_placement
remove_banking_guidance_strategy ;# optional to remove strategy

# Set Formality guidance file for multibit register banking process
set_svf banking.svf

# Source generated file to perform single-bit to multibit reg mapping
source -echo my_banking.tcl

# Update scan chain
check_scan_chain

# Run placement optimization
place_opt -skip_initial_placement ...

# Selectively split banked register to smaller registers (optional)
split_register_bank -lib_cells {REG4 REG4 REG2}
check_scan_chain

# Continue flow
...
clock_opt ...
...
route_opt ...
...
```

Reporting Multibit Registers in the Design

Use the following methods to report multibit registers:

- Reporting in Design Compiler and DC Explorer
- Reporting in IC Compiler

Reporting in Design Compiler and DC Explorer

In Design Compiler 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. For example,

```
dc_shell-topo> report_multibit_banking -hierarchical
Total number of sequential cells:                                  2422
    Number of single-bit flip-flops:                               1699
    Number of single-bit latches:                                    10
    Number of multi-bit flip-flops:                                 711
    Number of multi-bit latches:                                      2
Total number of single-bit equivalent sequential cells:           3147
    (A) Single-bit flip-flops:                                     1699
    (B) Single-bit latches:                                          10
    (C) Multi-bit flip-flops:                                      1434
    (D) Multi-bit latches:                                            4
Sequential cells banking ratio ((C + D) / (A + B + C + D)):      45.69%
Flip-Flop cells banking ratio  ((C) / (A + C)):                  45.77%

Multi-bit Register Decomposition:
--------------------------------------------------------------------------------
Bit-Width Reference                Number of instances    Single-bit Equivalent
--------------------------------------------------------------------------------
  2-bits                                707 (29.19%)          1414 (44.93%)
         MB2_DFF1                       705
         MB2_LD1                          2
  4-bits                                  6 (0.25%)             24 (0.76%)
         MB4_DFF1                         6
--------------------------------------------------------------------------------
Hierarchical multi-bit distribution:
--------------------------------------------------------------------------------
Hierarchical cell                     Num. of MB   Num. of      Seq. cells
                                      seq. cells   seq. cells   banking ratio
--------------------------------------------------------------------------------
top                                     713          2422         45.69%
sub_design1                             466          1055         61.28%
--------------------------------------------------------------------------------
```

Reporting in IC Compiler

In IC Compiler, you can report all multibit registers in the design by using the following get_mb_info.tcl script. The banking ratio is determined by the total bits of multibit registers divided by the total bits of all registers.

```
# © 2014 Synopsys, Inc.  All rights reserved.
#
# This script is proprietary and confidential information of
# Synopsys, Inc. and may be used and disclosed only as authorized
# per your agreement with Synopsys, Inc. controlling such use and
# disclosure.
#

puts "  Multibit register list"

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] {
  puts "   $mb_size-bit registers"
  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_cells -f "ref_name == $cell_name_unique"]]
     if {$num_of_mb_temp != 0} {
        puts "      $cell_name_unique x$num_of_mb_temp"
        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 -edge]]
set num_of_sb [expr $num_of_ff - $num_of_mb_cell]
puts "  "
puts "  Total Number of registers: $num_of_ff"
puts "     Single bit registers: $num_of_sb"
puts "     Multibit registers: $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)]
puts "  Banking Ratio: $pack_ratio %"
puts "     Total bits of all registers: $num_of_bit"
puts "     Total bits of multibit registers: $num_of_mb_bit"
```

Sourcing the get_mb_info.tcl provides results similar to the following:

```
prompt> source get_mb_info.tcl

Multibit register list
   2-bit registers
      MB2_FFRS_1 x885
      MB2_FFRS_2 x100
   4-bit registers
      MB4_FFRS_1 x674
      Mb4_FFRS_2 x135

Total Number of registers: 4235
   Single bit registers: 2441
   Multibit registers: 1794

Banking Ratio: 68 %
   Total bits of all registers: 7647
   Total bits of multibit registers: 5206
```

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.

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

- 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 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 SolvNet.

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 SolvNet.

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 A-1. Use the Liberty syntax described in the table to model the functionality of the cells.

Table A-1 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" ;
     }
  }
}
```

Note:

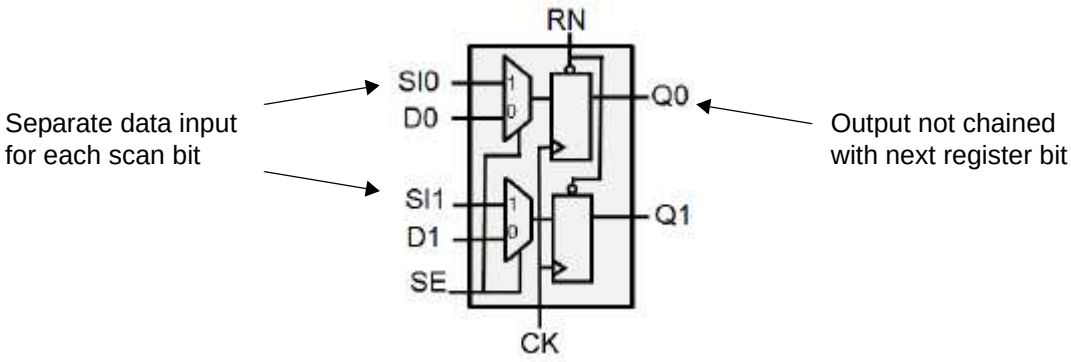
The `single_bit_degenerate` attribute is needed only for complex serial scan cells that can also have a dedicated scan output pin, and in some complex parallel multibit sequential cell models for automatic inference in optimization tools. For details, see the *Library Compiler User Guide*.

The following detailed examples demonstrate the multibit scan cell syntax.

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 A-1 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 A-1 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 A-1, 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 A-1.

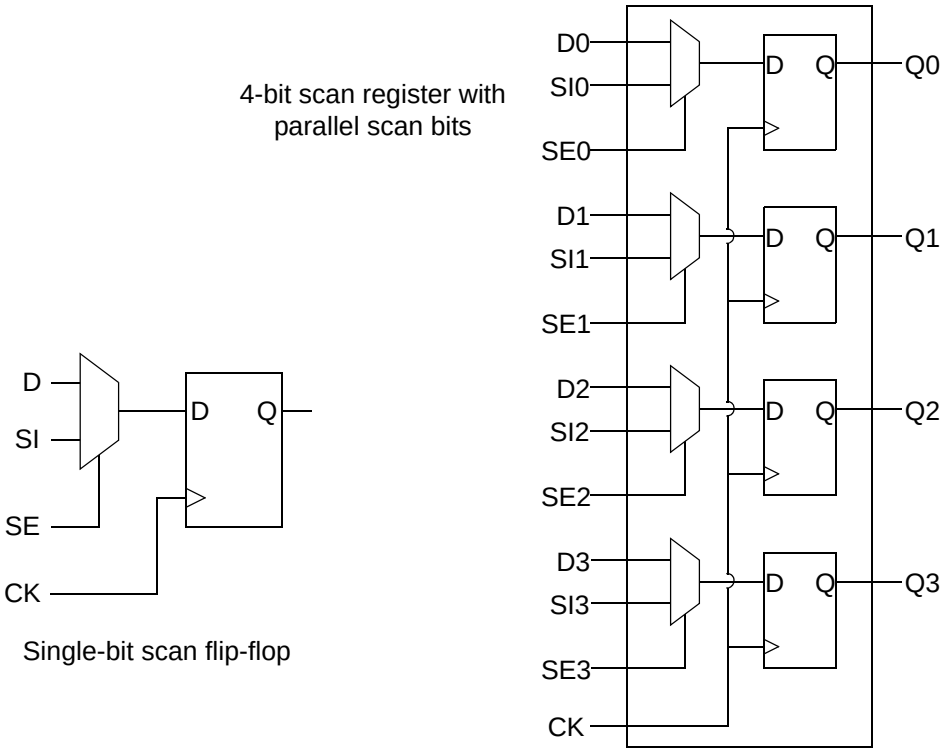
This is the general Liberty syntax for modeling a parallel scan multibit register:

```
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 A-2 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.

Figure A-2 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 A-1 and Example A-2 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 A-1 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 A-2 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 */
```

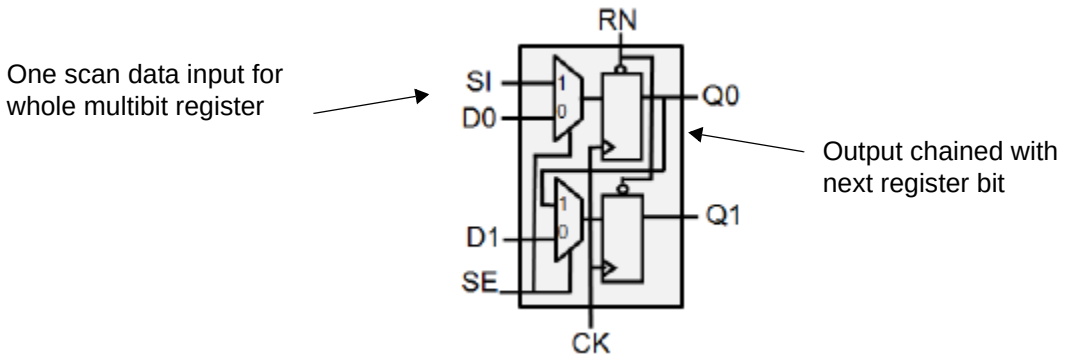
```
  bus(D) {
    bus_type : bus4;
    direction : input;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    capacitance : 1.0;
    nextstate_type : data;
  }
  test_cell () {
    pin(CK){
      direction : input;
    }
    bus(D) {
      bus_type : bus4;
      direction : input;
    }
    bus(SI) {
      bus_type : bus4;
      direction : input;
      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 A-3 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 A-3 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`.

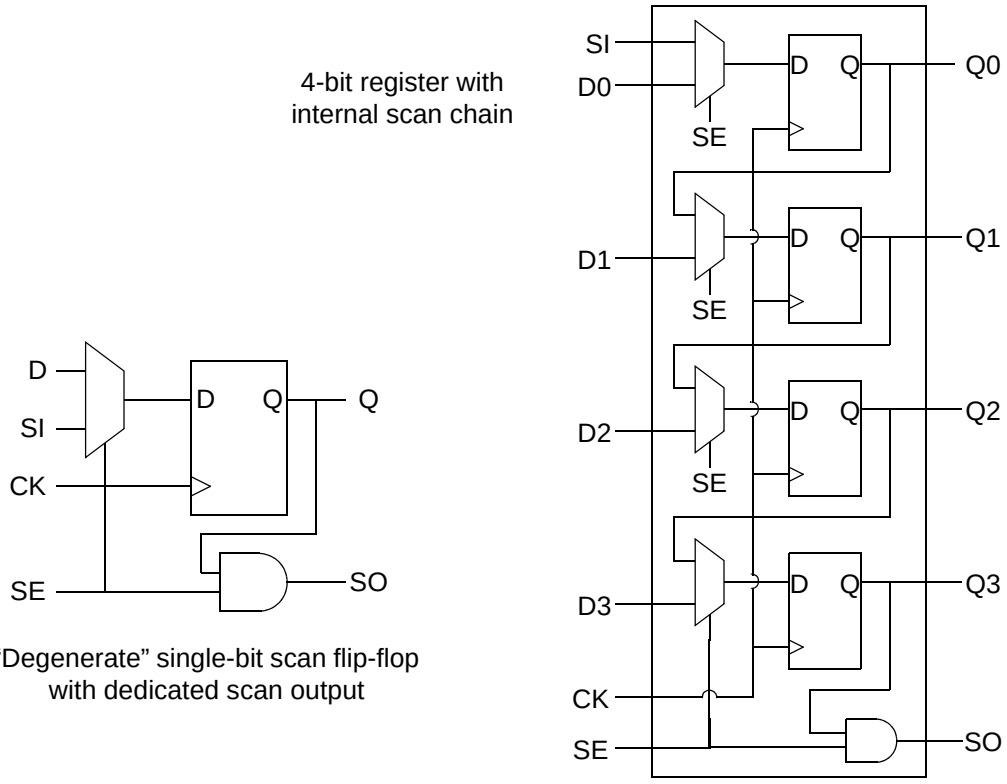
If you specify the `scan_pin_inverted` attribute, you must specify the `scan_start_pin` attribute in the same `bus` or `bundle` group. You cannot specify this attribute in the `pin` group, even for pin definitions in the `bus` or `bundle` group.

For the cell in Figure A-3, 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 A-4 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 A-4 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.

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 A-3 and Example A-4 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 A-3 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 A-4 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 */
```