

IC Validator LVS User Guide

Version O-2018.06, June 2018

SYNOPSYS®

Copyright Notice and Proprietary Information

©2018 Synopsys, Inc. All rights reserved. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

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.

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

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

Contents

About This User Guide	x
Customer Support	xii
1. LVS Overview	
LVS Compare	1-2
Global Netlist Modifications	1-4
The push_down_pins Argument	1-4
Using the remove_dangling_nets Argument	1-6
Equivalence Point Generation	1-6
Equivalence Point Comparison	1-7
Hierarchical Treatment of Failed Equivalence Cells	1-8
Filtering Devices	1-9
Merging Devices	1-9
Parallel Merging	1-9
Series Merging	1-10
Shorting Equivalent Nodes	1-12
Comparing the Schematic and Layout Netlists	1-14
Defining Match Conditions	1-14
Handling Circuit Symmetry	1-14
Compare Results	1-20
Net Mismatches	1-22
Instance Mismatches	1-25
Property Errors	1-28

2. Netlist Formats

NetTran.....	2-2
Translating Netlists to IC Validator Format Using NetTran	2-2
Netlist Translation	2-3
Skeletal Equivalence File	2-3
Wire Resolution Log File	2-3
Command-Line Syntax for NetTran	2-4
Translating Standard Netlists to IC Validator Format	2-12
Property Values	2-13
SPICE Translation Examples	2-13
CDL Map File	2-14
Verilog Translation Example	2-15
Creating a Skeletal Equivalence File	2-15
Comments in Output Netlist	2-15
Cell-Specified Comments	2-15
Instance-Specified Comments	2-17
Global Comments	2-17
SPICE Netlist Format	2-18
Devices	2-18
Bipolar Transistor	2-18
Capacitor	2-19
Diode	2-21
Inductor	2-22
JFET	2-22
MOSFET	2-23
Resistor	2-25
Cell Definition	2-26
Nested Cells Definition	2-27
Cell Instance Definition	2-28
String Parameters	2-31
Double-Quoted String Usage	2-31
Duplicate Ports Definition	2-32
Identical Cells Definition	2-33
Duplicate Cells Definition	2-33
Duplicate Cell Instance Definition	2-33
Control Statements	2-34
.GLOBAL	2-34
.CONNECT	2-35

.INCLUDE	2-35
.PARAM	2-36
*.BUSDELIMITER.....	2-37
*.CAPA	2-37
*.CONNECT	2-37
*.DIODE	2-37
*.EQUIV	2-37
*.RESI	2-38
*.SCALE	2-39
.OPTION SCALE	2-40
Classifying Nonstandard Numeric SPICE Parameters.....	2-40
Line Continuation.....	2-41
Comments	2-42
Verilog Netlist Format	2-42
Cell Definition.....	2-42
Port	2-42
Net.....	2-43
Instance	2-43
Assign Statements.....	2-44
Bus (Vector).....	2-44
Defining Global Supply Nets	2-45
Defining Local Supply Nets	2-45
Supply Net Translation Precedence	2-45
The -verilog-busLSB Option	2-47
Verilog Compiler Directives	2-49
3. Building the Substrate	
Overview.....	3-2
Guidelines for Building the Substrate	3-5
4. Hierarchical Device Extraction	
Hierarchical Device Extraction	4-2
Layers for Device Extraction	4-2
Device Terminals	4-3
Device Extraction and Hierarchy	4-4
Bipolar Junction Transistors	4-4

Vertical BJT Extraction	4-4
Lateral BJT Extraction	4-5
Capacitors	4-6
Metal-to-Metal Capacitor	4-7
MOS Capacitors	4-8
Diode Extraction	4-10
Inductor Extraction	4-11
MOSFETs	4-12
Resistor Extraction	4-13
5. Device Property Calculations	
Property Calculations	5-2
Compare Properties	5-2
Simulation Properties	5-2
Default Properties	5-4
Device Extraction and Property Calculation Functions	5-5
Device Extraction Function Prototype	5-5
Default Property Calculation Function	5-6
Device Extraction Function Calls	5-6
Coefficient and Body Position Arguments	5-7
Device Utility Functions	5-8
Measuring Polygons for Device Property Calculation	5-8
Custom Device Extraction Properties and Calculations	5-9
6. Layout-Versus-Schematic Flows	
Creating a Black-Box LVS Flow	6-2
Setting Up a Black-Box Flow	6-2
Usage Models	6-3
Example 1: All Ports Match Between Netlists	6-4
Example 2: Port Names Do Not Match Between Netlists	6-4
Example 3: Extra Ports in Layout Black-Box Cell	6-5
Example 4: Extra Schematic Ports on Black-Box Cells	6-8
Example 5: Extra Untexted Layout Ports on Black-Box Cells	6-10
Example 6: Untexted Layout Ports on Black-Box Cells	6-11
Example 7: Symmetry and Black-Box Cells	6-13

Netlist-Versus-Netlist Flow	6-14
Modes for Running NVN	6-14
Device Mapping Functions	6-15
NVN Flow With a Runset	6-16
Full Runset Flow	6-17
Partial Runset Flow	6-18
Netlist-Versus-Netlist Without a Runset	6-20
Output Files	6-20
7. Compare Functions Basics	
Predefined Name Matches	7-2
Complementary Functions	7-2
Precedence Rule	7-3
8. Netlist Modification	
Merging Devices	8-2
Merging Parallel Devices	8-2
Merged Device Properties	8-6
Excluding Merging by Tolerance	8-8
Custom Merging Equations for Device Properties	8-9
Series Merging Devices	8-11
Parallel Device Merging Short Equivalent Nodes	8-18
Filtering	8-21
Filtering Standard Options	8-22
Custom Filtering	8-24
9. Table-Based Lookup Functionality	
Overview	9-2
Using Table-Based Lookup	9-3
Lookup Table Structure	9-3
Table-Lookup Extraction Function	9-4
Table-Based Lookup Example	9-5

Preface

This preface includes the following sections:

- [About This User Guide](#)
- [Customer Support](#)

About This User Guide

The *IC Validator LVS User Guide* assists the designer in running layout-versus-schematic (LVS) with the IC Validator tool. Use this guide in conjunction with the *IC Validator Reference Manual* and *IC Validator User Guide*.

Audience

This user guide is designed to enhance both beginning and advanced users' knowledge of LVS.

Related Publications

For additional information about the IC Validator tool, see the documentation on the Synopsys SolvNet[®] online support site at the following address:

<https://solvnet.synopsys.com/DocsOnWeb>

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

- Custom Compiler[™]
- IC Compiler[™]
- IC Compiler II[™]
- StarRC[™]

Release Notes

Information about new features, enhancements, changes, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the *IC Validator Release Notes* on the SolvNet site.

To see the *IC Validator Release Notes*,

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

<https://solvnet.synopsys.com/DownloadCenter>

2. Select IC Validator, and then select a release in the list that appears.

Conventions

The following conventions are used in Synopsys documentation.

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

Customer Support

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

Accessing SolvNet

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

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

<https://solvnet.synopsys.com>

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to sign up for an account.

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

Contacting the Synopsys Technical Support Center

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

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

1

LVS Overview

The IC Validator Layout Versus Schematic (LVS) process compares the extracted netlist from the layout to the original schematic netlist to determine if they match.

The comparison check is considered clean if all of the devices and nets of the schematic match the devices and the nets of the layout. Optionally, the device properties can also be compared to determine if they match within a tolerance. When properties are compared, all of the properties must match to achieve a clean comparison.

Two main processes make up the LVS flow. The first process in the flow is extraction, in which the IC Validator tool analyzes the layers within the layout database and extracts all of the devices and nets. The second process in the flow is LVS compare, in which the actual comparison of devices and nets occurs. The LVS runset contains a series of function calls that control both extraction and netlist comparison.

This chapter has the following sections:

- [LVS Compare](#)
- [Global Netlist Modifications](#)
- [Equivalence Point Generation](#)
- [Equivalence Point Comparison](#)
- [Compare Results](#)

LVS Compare

The IC Validator layout versus schematic process checks to see if the netlist extracted from the layout matches the original schematic netlist. The comparison is considered clean if all of the devices and nets of the schematic match all the devices and nets in the layout. Optionally, the device properties can also be checked to determine if they match within a user-defined tolerance. In this case, all of the properties must match to get a clean comparison.

The compare flow begins with reading in both the schematic and layout netlists, as shown in [Figure 1-1](#). To obtain compare results,

- First, the IC Validator tool makes global netlist modifications, which facilitates the comparison of netlists. For example, if two or more ports are connected to the same net across all instances, the ports are merged.
- Next, the IC Validator tool considers a list of cell pairs (one from the schematic and one from the layout) to be compared. You can specify this list of cell pairs, or it can be determined automatically by the tool. The pair of cells used for comparison is called an equivalence cell pair. After determining the equivalence pairs, the tool begins the actual comparison process.
- Finally, the tool performs optional device filtering and merging operations. At this stage, the tool compares the logic of the schematic and layout cell.

Figure 1-1 LVS Flow

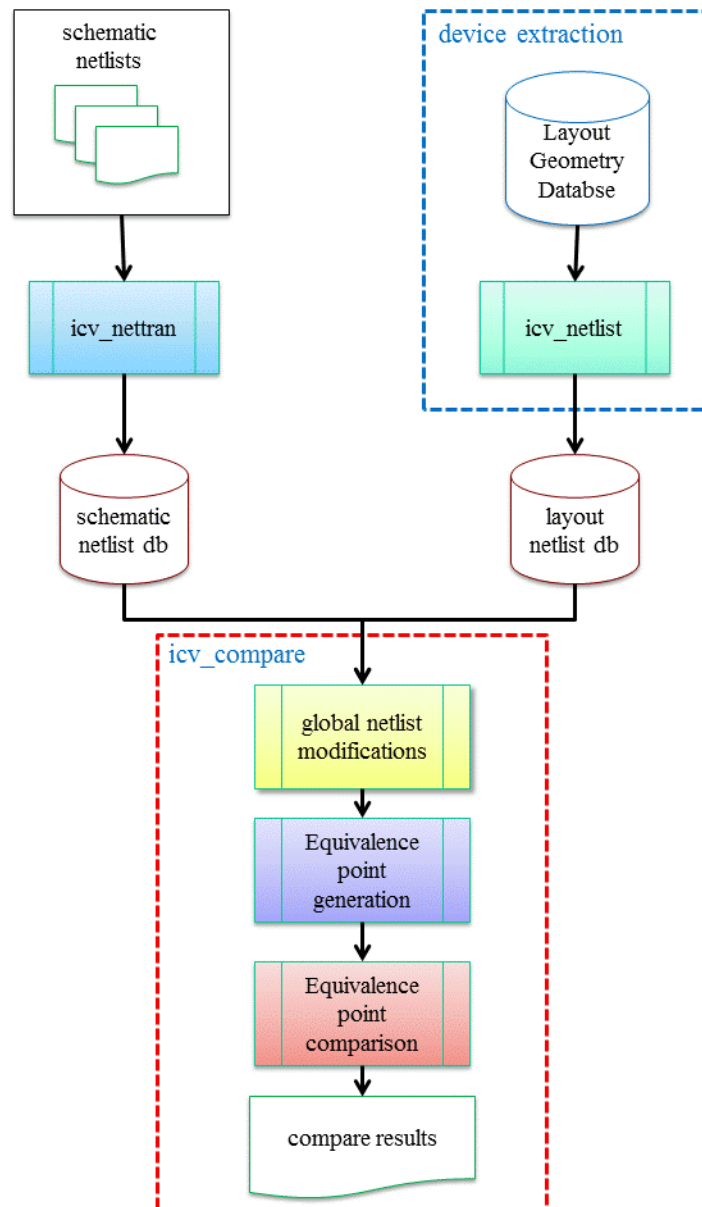
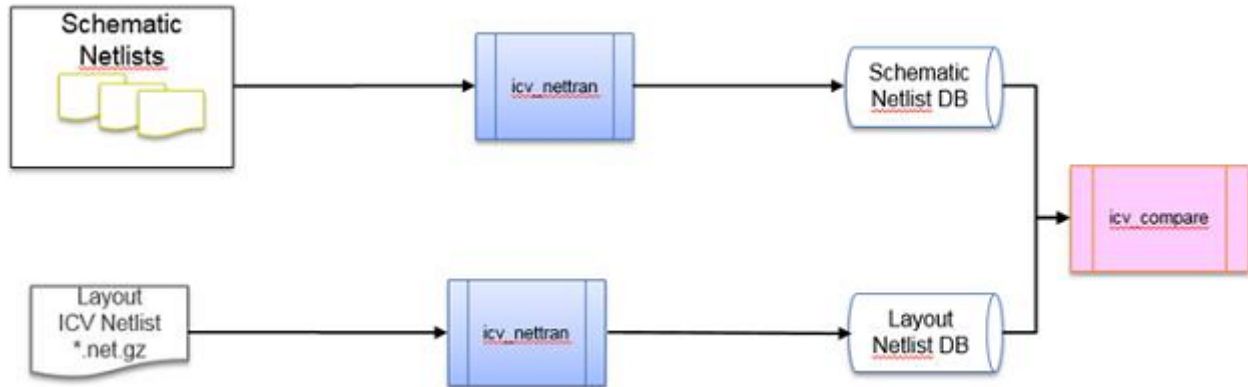


Figure 1-2, highlights the compare step of the LVS flow.

Figure 1-2 Compare-Only LVS Flow



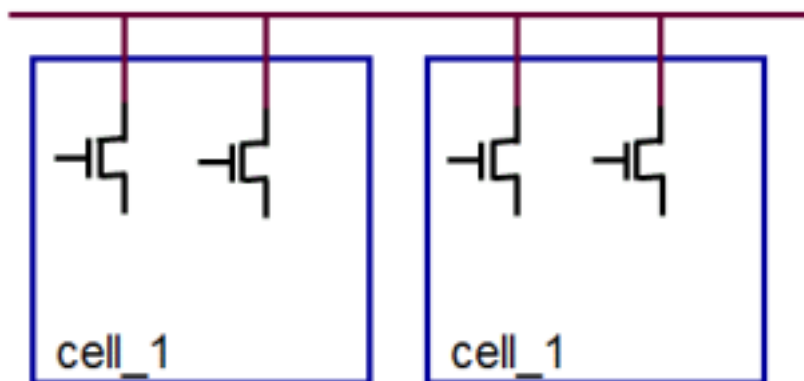
Global Netlist Modifications

During global netlist modification, the IC Validator tool uses the `push_down_pins` and `remove_dangling_nets` arguments of the `compare()` function to facilitate comparison by simplifying the topology and removing extraneous information. These arguments are described in the following sections.

The `push_down_pins` Argument

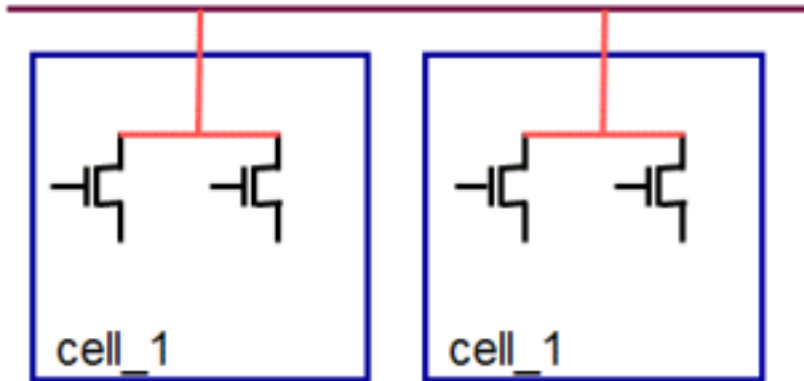
When the `push_down_pins` argument is `true` and two or more ports are connected to the same net across all instances, the ports are merged. In Figure 1-3, both ports are connected to the same net in each instance of `cell_1`.

Figure 1-3 Ports Connected to the Same Nets



The ports are merged as a result of applying this argument. See [Figure 1-4](#).

Figure 1-4 Merged Ports Using the push_down_pins Argument



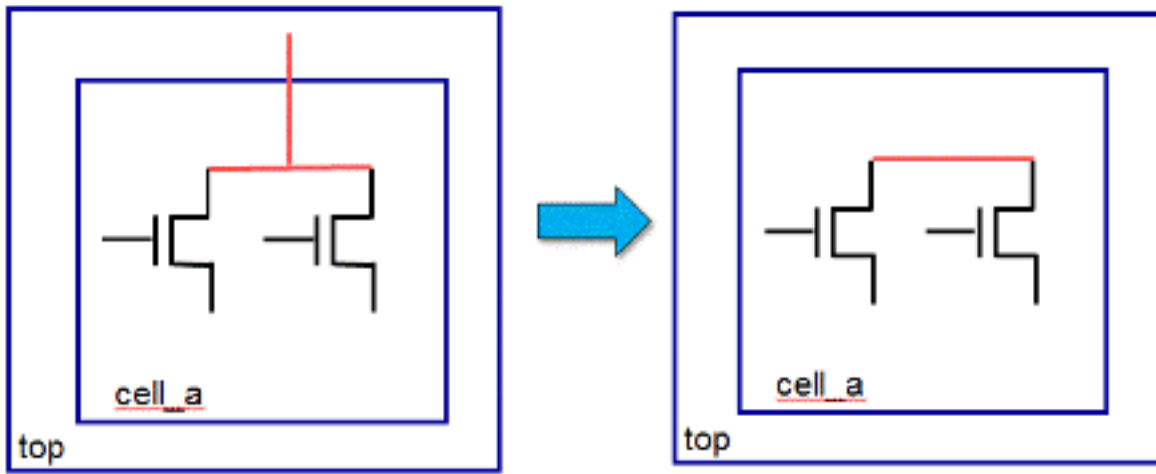
When ports are merged, the resulting port has a name such as SchMergedNet#N or LayMergedNet#N. The following scenarios do not have merged ports:

- Power and ground nets are not pushed down when they are shorted.
- For a schematic or layout equivalence cell pair, hierarchically connected pins are not pushed down for the pair.
- A pair participates in a multiple equivalence point (that is, multiple schematic cells paired with one layout cell or vice versa) does not have merged ports.

Using the `remove_dangling_nets` Argument

When the `remove_dangling_nets` argument is used, the netlists are preprocessed to find all cases where a port net in a cell never connects to any extracted device throughout the hierarchy. The IC Validator tool removes all such dangling nets throughout the hierarchy by changing the net in the originating cell from a port net to an internal net, as shown in [Figure 1-5](#).

Figure 1-5 Effect of the `remove_dangling_nets` Argument



Equivalence Point Generation

The equivalence point generation flow selects pairs of cells (one from the schematic and one from the layout) that potentially have the same logic and serve as points of comparison. These pairs of cells are called equivalence cells.

You can specify equivalence cells, or they can be generated by the IC Validator tool. The better the selection of equivalence cells, the faster the compare runs. Therefore, it is best to exclude as much as possible from the equivalence cell list pairs of cells that are never meant to be logically exactly the same.

Use the `equiv_options()` function to specify a list of equivalence cells. The IC Validator tool uses the `generate_system_equivs` argument of the `lvs_options()` function to generate additional equivalence cells. The tool uses heuristics based on the following information:

- Name matching
- Device counts
- Device types

- Instance counts
- Port counts

Note:

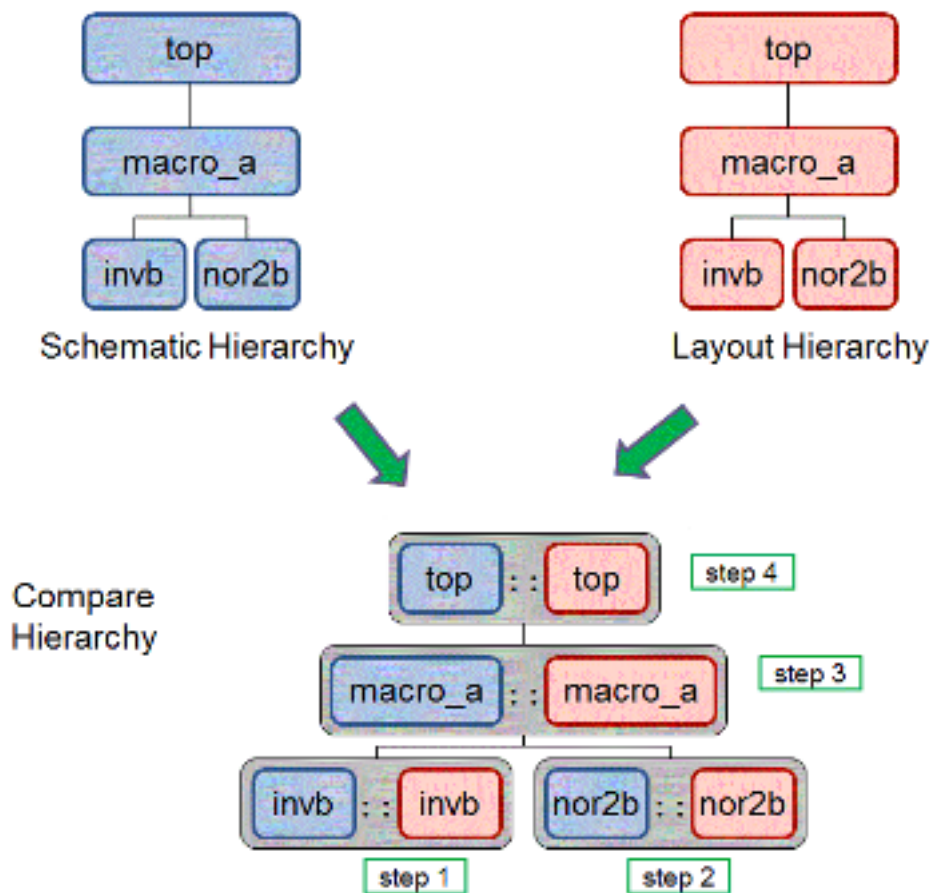
System-generated equivalence cell pairs are used only to inject more hierarchy into the comparison and improve performance. If compare errors are detected in a system-generated equivalence pair, that cell pair is exploded into the parent cell.

Equivalence Point Comparison

After the IC Validator tool has an equivalence list, the tool performs a bottom-up comparison, that is, it begins the cell comparison starting as low in the hierarchy as possible and works its way up the hierarchy until it finally compares the top block.

Figure 1-6 illustrates how the IC Validator tool matches cells from the schematic and the layout to establish a compare hierarchy.

Figure 1-6 Establishing a Compare Hierarchy



In this figure, the compare engine compares the equivalence cell pairs in the following order:

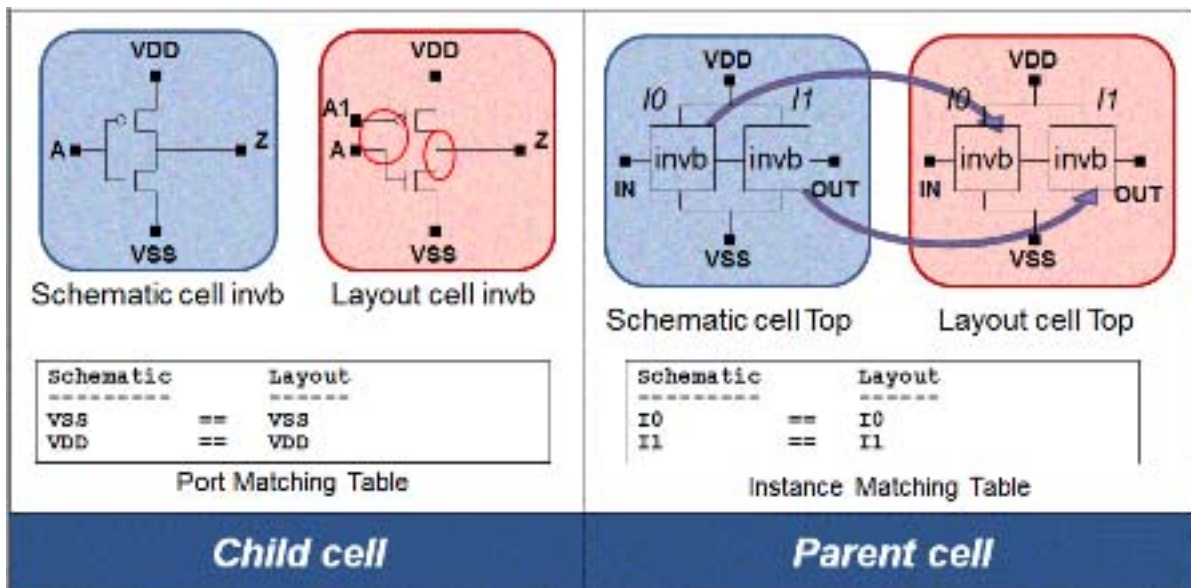
1. Compare invb::invb
2. Compare nor2b::nor2b
3. Compare macro_a::macro_a
4. Compare top::top

Hierarchical Treatment of Failed Equivalence Cells

The `action_on_error` argument of the `compare()` function determines how the IC Validator tool reports errors hierarchically for user-specified equivalence cell pairs. By default, `action_on_error = NO_EXPLODE`, which means that all details of the comparison failure are reported at the user-equivalence cell level. When the comparison continues further up the hierarchy, an abstracted instance of the cell is used in the parent cell.

In [Figure 1-7](#), the left side of the diagram illustrates the comparison of an inverter at the child level. There are differences in the nets and ports between the schematic and the layout. Only ports VSS and VDD are compared. Errors are reported at the child level.

Figure 1-7 Treatment of Failed Equivalence Cells



When the comparison continues at the parent level, the IC Validator tool creates an abstracted instance of the cell (invb on the right side of diagram) and

1. Uses matched ports information from child cell comparison (VDD, VSS)

2. Assumes match among previously-unmatched, like-named ports (A, Z)
3. Discards extra ports (A1)

In this figure, there are compare errors inside the child cell, however, it is possible to compare the connections to the ports of `invb` in the parent cell.

The system-generated equivalence cell pairs are used only to inject additional hierarchy into the comparison. If compare errors are detected in a system-generated equivalence pair, that cell pair is exploded into the parent cell.

Filtering Devices

The IC Validator tool uses the `filter()` function to remove devices that do not need to be compared. Filtering is based on the predefined filter options. You can customize the conditions for filtering by defining your own filter functions.

In the following example, all NMOS devices that have their gate, source, and drain pins shorted are filtered using a predefined filter option called `NMOS_1`.

```
filter(compare_state,  
      device_type = NMOS,  
      filter_options = {NMOS_1});
```

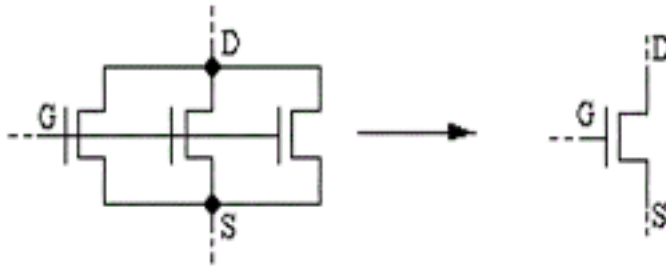
Merging Devices

Device merging facilitates comparisons by reducing the configuration of devices to their simplest form. Device merging is an iterative cycle alternating between passes of parallel merging and series merging. The merging process does not merge power and ground nets, port nets, or static nets. Merging continues until no further parallel or series merging occurs.

The different types of merging are presented in the following sections.

Parallel Merging

Devices are parallel if every corresponding pin pair between the devices is connected to the same net. Two or more parallel devices are combined into a single merged device with the same device type and net connections, as shown in [Figure 1-8](#).

Figure 1-8 Parallel Merging

For example, to merge p1 devices, type the following:

```
merge_parallel(
    compare_settings,
    device_type = PMOS,
    device_names = {"p1"}
);
```

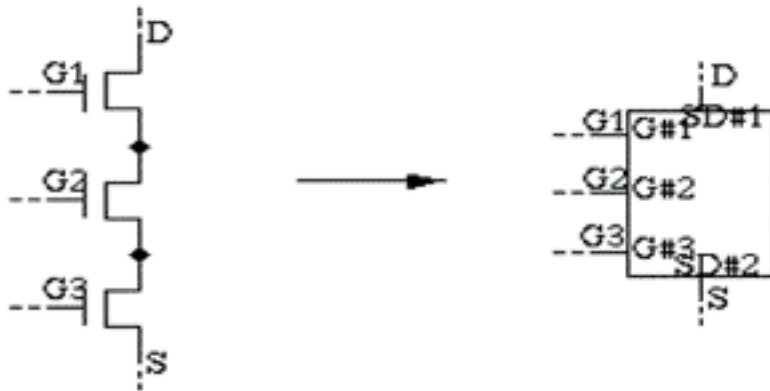
In some designs, it might be necessary to limit the merging based on the device properties. In the following example, the PMOS p1 devices are parallel merged if the widths of the devices are within 20 percent and the length of the devices are within 30 percent.

```
merge_parallel(
    compare_settings,
    device_type = PMOS,
    device_names = {"p1"},
    exclude_tolerances = {{ "w", [-20, +20] }, { "l", [-30, +30] } },
);
```

Series Merging

NMOS and PMOS devices are in series when their drain and source pins are connected sequentially. All devices in the chain must be of the same device type. The nets that connect the drain or source pins of neighboring devices cannot have any other connections or be a port of the cell. A single merged device is created with multiple gate pin connections corresponding to the number of devices in the chain, and the gate pins on the merged device are automatically designated as swappable, as shown in [Figure 1-9](#).

Figure 1-9 Series Merging



The following call to the `merge_series()` function merges all NMOS devices. Notice that each chain of MOS devices that gets merged must have the same device name.

```
merge_series(
    compare_settings,
    device_type = NMOS
);
```

If MOS devices in series have connected gates, you can merge them by using the `merge_connected_gates` argument, as shown in the following example:

```
merge_series(
    compare_settings,
    device_type = NMOS,
    merge_connected_gates = true
);
```

Figure 1-10 `merge_connected_gates` Argument

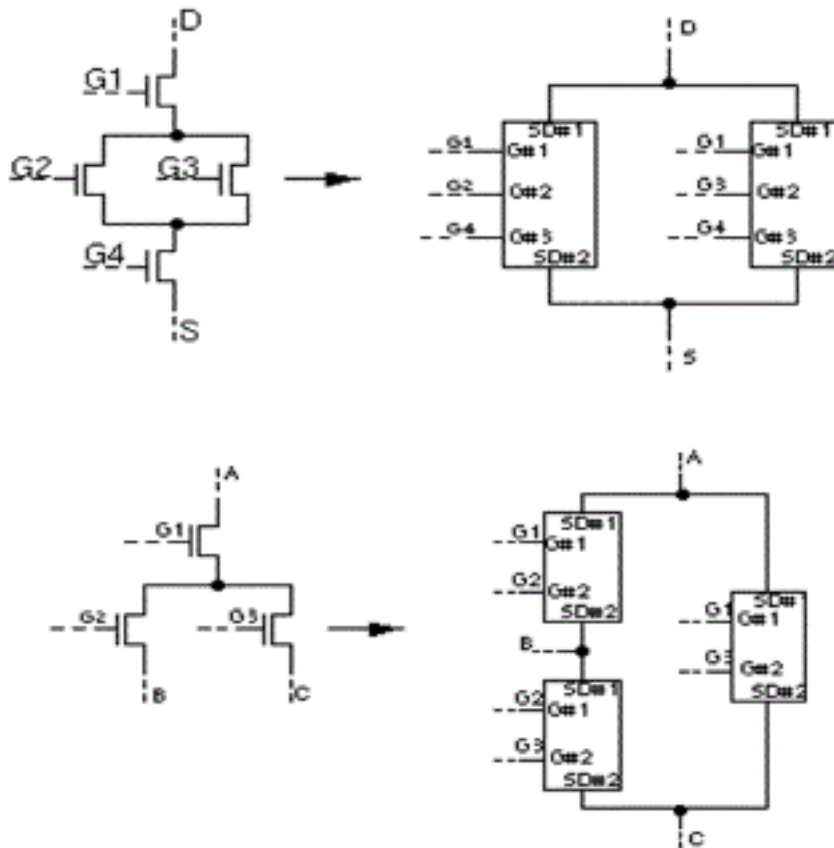
Path merging is used for complex structures where groups of devices are stacked. When two or more transistors are found in a path, they are combined into a single device with logically equivalent gate-pin inputs. Recognition of these constructs allows devices in the schematic to be equated to devices in the layout that are logically equivalent even though

their implementations might not match. To turn on path merging, set the `multiple_paths` argument, as shows below:

```
merge_series(
    compare_settings,
    device_type = NMOS,
    multiple_paths = true
);
```

Figure 1-11 shows examples of these complex structures.

Figure 1-11 Path Merging for Complex Structures



Shorting Equivalent Nodes

The `short_equivalent_nodes()` function facilitates additional merging by shorting equivalent nodes between two series chains, if the width ratios of gates meet the width ratio tolerances. There are two modes in which this function operates. When the `short_nodes`

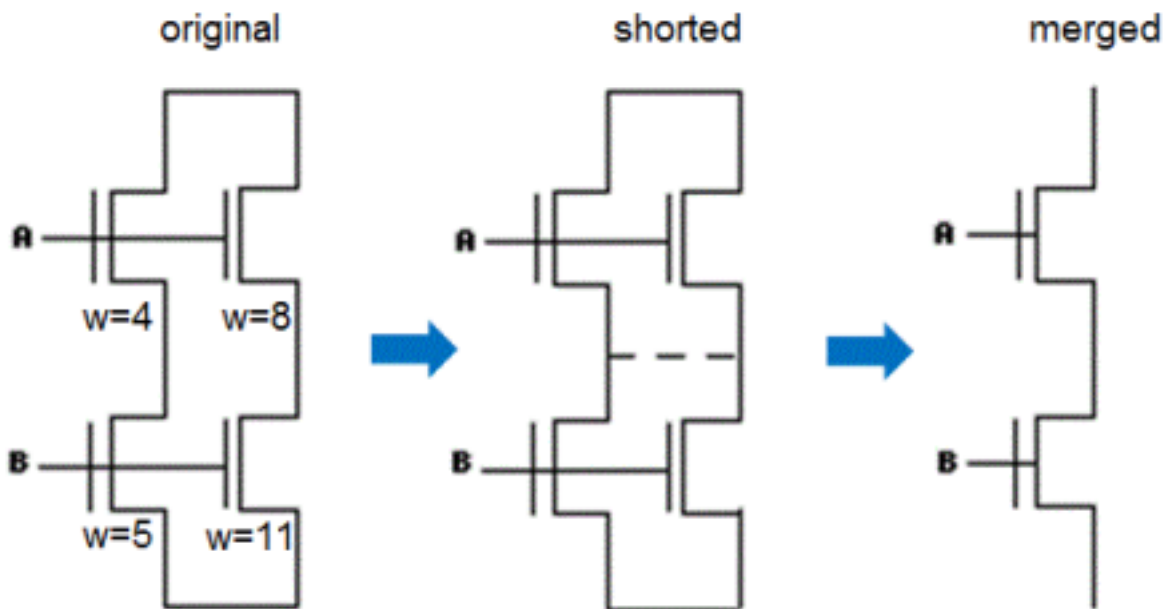
argument is `SAME_DEVICE_NAME_ONLY`, the nodes are shorted only when the MOS devices belong to the same `device_name`. To short any device of the same device type, set `short_nodes` to `ANY_DEVICE_NAME`.

In the following example, all NMOS devices with `device_name` equal to "nm1" have equivalent nodes shorted, if they meet the width ratio tolerance of within 15 percent.

```
short_equivalent_nodes(state = my_compare_state,
    device_type = NMOS,
    device_names = {"nm1"},
    short_nodes = SAME_DEVICE_NAME_ONLY,
    width_ratio_tolerance = { [-15,+15], RELATIVE }
);
```

In [Figure 1-12](#), the width ratio of the NMOS devices connected to net A is $8/4 = 2$ and the width ratio of the gates connected to net B is $11/5 = 2.2$. IC Validator uses the smallest width ratio as the base ratio. Therefore, the relative difference calculation is $(2.2 - 2) / 2 = 10$ percent. This meets the tolerance of within 15 percent and the equivalent nodes are merged.

Figure 1-12 Shorted Equivalent Nodes



Comparing the Schematic and Layout Netlists

After the filtering and merging stages are completed, the IC Validator tool compares the schematic and layout netlists. The tool matches the devices and nets within the equivalent cell pair it is processing. Using the `match()` function, you control key aspects of the comparison, such as controlling aspects of the compare algorithm, defining match conditions, and detecting swappable ports when equivalence cells are exploded.

Defining Match Conditions

By default, when comparing netlists, the IC Validator tool checks that the topologies (nets and devices) of the two netlists match, and that the properties of the devices are the same. Any differences in topology or device properties result in an error. By default, the tool issues warning messages for conditionals that are of interest, but that are not typically considered errors. You can use the `match_condition` argument to tailor the match conditions as well as the types of error and warning messages to be issued during a compare run.

The `match_condition` argument contains approximately thirty types of match conditions that can be set to `ERROR`, `WARNING`, or `NONE`. These match conditions cover a wide range of conditions such as undefined properties, equated nets failures, and port mismatches.

For example, if you want to flag any texted port nets in the layout that match port nets having different net names in the schematic, add the following statement to the runset:

```
match : published function (  
    state = compare_settings,  
    match_condition = { ports_matched_with_different_name = ERROR }  
);
```

For a list of match conditions, see the *IC Validator Reference Manual*.

Handling Circuit Symmetry

Circuit symmetries influence the netlist comparison process by introducing ambiguity to the mapping of a layout net or instance to its logically equivalent component in the schematic. The netlist comparison engine's treatment of this ambiguity is a function of several factors:

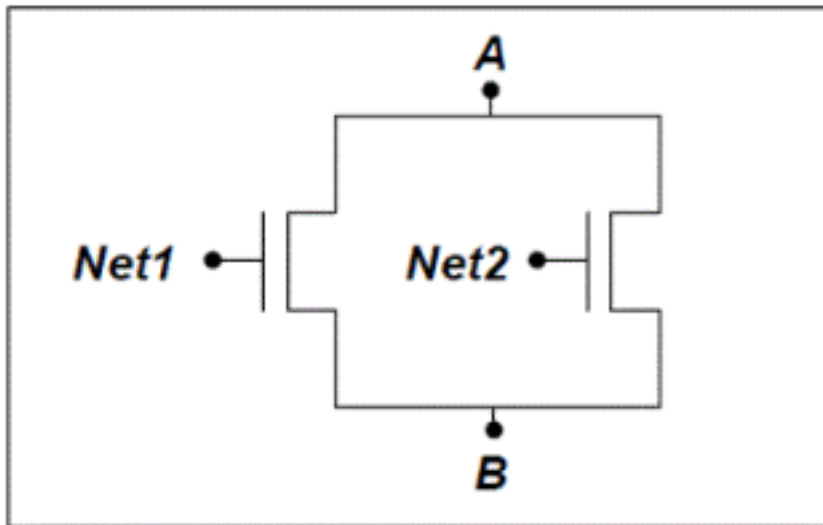
- Hierarchical location of the symmetry
- Availability of layout text to break the symmetry
- Compare engine runset configuration
- Design flow style with respect to layout texting and swapped connectivity of layout nets versus schematic nets

Impact of Symmetry on Circuit Comparisons

Circuit symmetry is defined as any group of nets or instances that cannot be logically differentiated from one another based on connectivity, net names, or device properties.

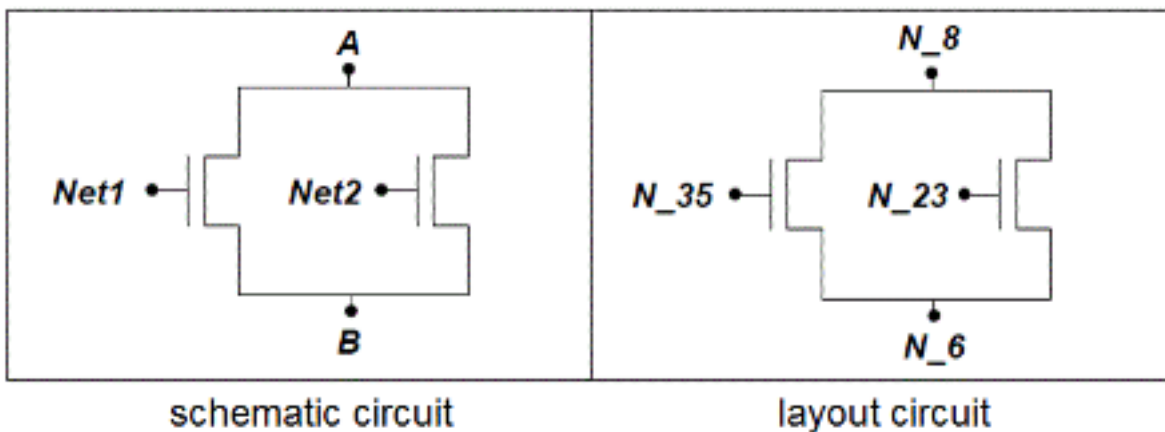
[Figure 1-13](#) illustrates examples of circuit symmetries. In this figure, net A is symmetric with net B and net Net1 is symmetric with Net2.

Figure 1-13 Circuit Symmetries



Symmetries complicate netlist comparisons by introducing ambiguity into the matching of nets and instances from the layout netlist to the source netlist. For example, consider how nets might be matched between the source and candidate netlist circuits in [Figure 1-14](#).

Figure 1-14 Schematic and Layout Circuits



There are four different mappings that occur when you compare the schematic and layout circuits. [Table 1-1](#) defines these mappings.

Table 1-1 Mapping Definitions

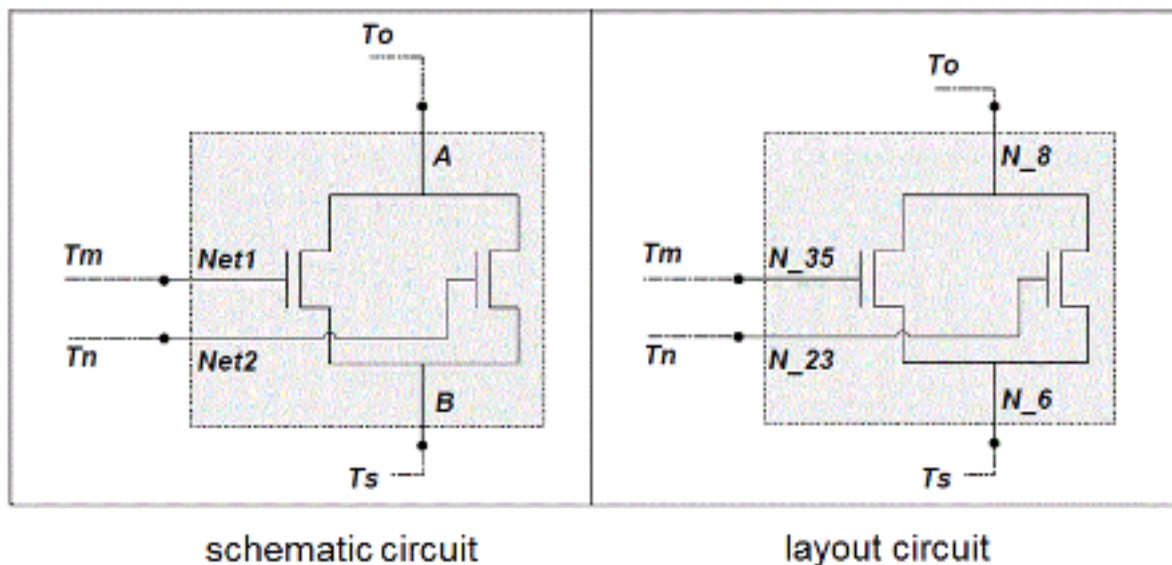
	Schematic net		Layout net
Mapping#1	Net1	=	N_35
	Net2	=	N_23
	A	=	N_8
	B	=	N_6
Mapping#2	Net1	=	N_35
	Net2	=	N_23
	A	=	N_6
	B	=	N_8
Mapping#3	Net1	=	N_23
	Net2	=	N_35
	A	=	N_8
	B	=	N_6
Mapping#4	Net1	=	N_23
	Net2	=	N_35
	A	=	N_6
	B	=	N_8

The selection of mappings by the netlist comparison tool among the four legal mapping sets is random, that is, any of the four mappings could be reported by the netlist comparison tool. Such randomness in net matching is acceptable in two situations:

- The symmetric nets occur directly within the top cell of the netlists being compared.
- The symmetric nets are internal, non-port nets within any level of hierarchy.

However, such randomness is not acceptable when comparing child cell ports. The reason is that child cell port matching tables are used to validate the connectivity of parent cell nets connecting to placements of the child cell. To illustrate this, look at the circuit block in the table and place it in a parent cell. You see the following circuits, as shown in [Figure 1-15](#).

Figure 1-15 Net Names in the Schematic and Layout Circuits



Assume that the net names in the parent cell in the schematic and layout are intended to be matched. Refer to [Table 1-1](#) that showed four possible mappings for the child cell. There is a potential randomness in the tool findings that the parent compares. For example, if the tool chooses Mapping #1, the parent cell is a clean comparison. However, if the tool chooses any of the other mappings, the compare fails. For successful LVS comparison, there must be a mechanism for resolving the symmetry of child cell ports.

There are two methods used for resolving symmetry:

- Calculate port swappability rules
- Match by net name

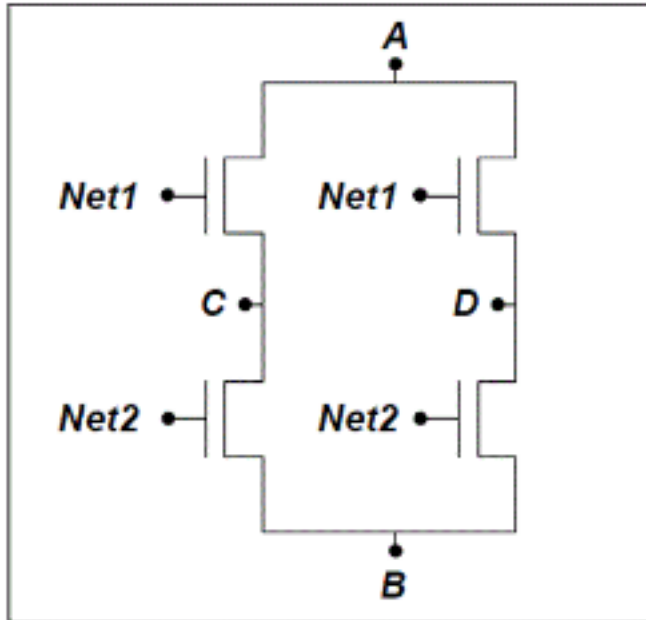
Resolving Symmetry Using Port Swappability Rules

When symmetry occurs for child cell port nets, port swappability rules are derived based on their symmetry. These rules describe various sets of logically equivalent connections to the port nets when the cell of interest is placed inside a parent cell. Port swappability rules take two forms:

- Independently swappable ports. A group of ports by which parent connections might be swapped while maintaining logical equivalence, regardless of connectivity to all other ports of the cell.
- Dependently swappable ports. A group of ports by which parent connections might be swapped while maintaining logical equivalence, but only if connections are also swapped to a secondary group of ports at the same time.

In [Figure 1-16](#), nets C and D can be swapped without altering the functionality of the circuit, resulting in net A and B being independently swappable ports. However, net A and net B are dependently swappable because in order to swap A and B, you must also swap Net1 with Net2.

Figure 1-16 Port Swappability



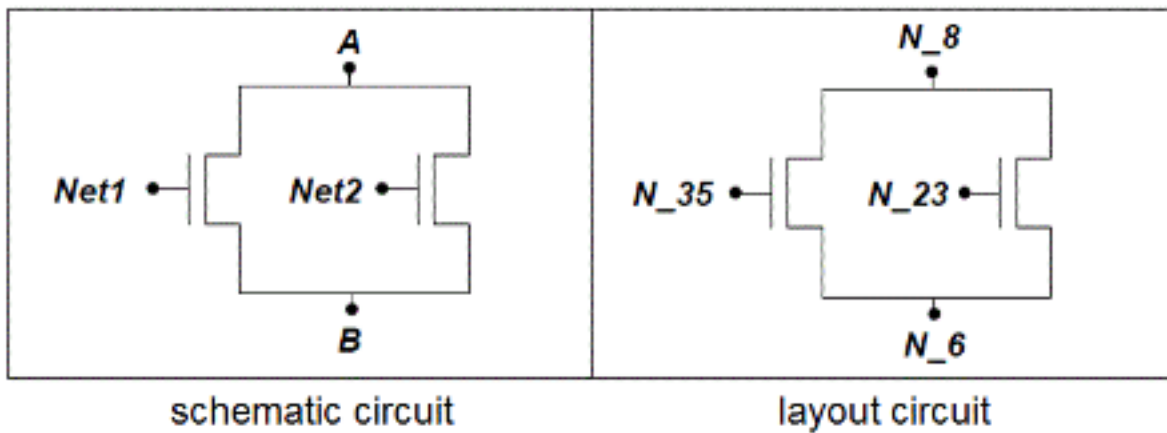
Port swappability rules are used to compare parent cell connections to child cell symmetric ports. To turn on the port swappability rules, set the `detect_permutable_ports` argument as follows:

```
match : published function (
    state = compare_settings,
    detect_permutable_ports = true
);
```

Resolving Symmetry With Net Naming

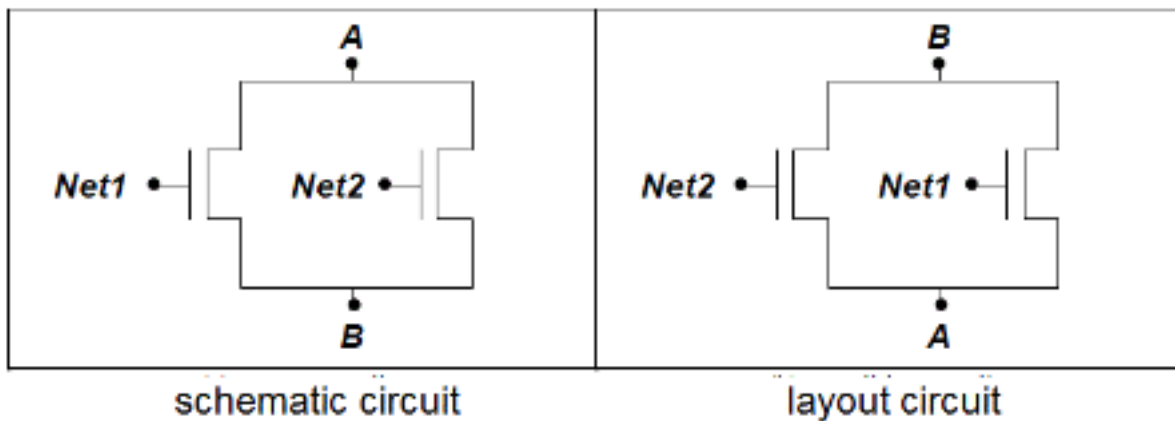
Logically equivalent nets can be differentiated using net names. Consider the schematic and layout circuits in [Figure 1-17](#), noting that there is no correspondence in net names between the two circuits. There is no opportunity to map a specific schematic net to a specific layout net, since all nets have unique names.

Figure 1-17 No Corresponding Net Names



Next, consider the same circuits, but with names that correspond between the schematic and layout circuits, as shown in [Figure 1-18](#).

Figure 1-18 Corresponding Net Names



The presence of corresponding net names is used by the netlist comparison algorithm to reliably match a specific symmetric layout net to a specific schematic net, as shown in [Table 1-2](#)

Table 1-2 Schematic and Layout Matching

Schematic		Layout
A	=	A
B	=	B
Net1	=	Net1

Table 1-2 Schematic and Layout Matching (Continued)

Schematic		Layout
Net2	=	Net2

Net names are also used to differentiate symmetric child cell ports for the purpose of checking parent cell connectivity to child cell ports.

To use text to resolve symmetry, turn on the swappability rules by setting the `match_by_net_name` argument:

```
match : published function (
    state = compare_settings,
    match_by_net_name = true
);
```

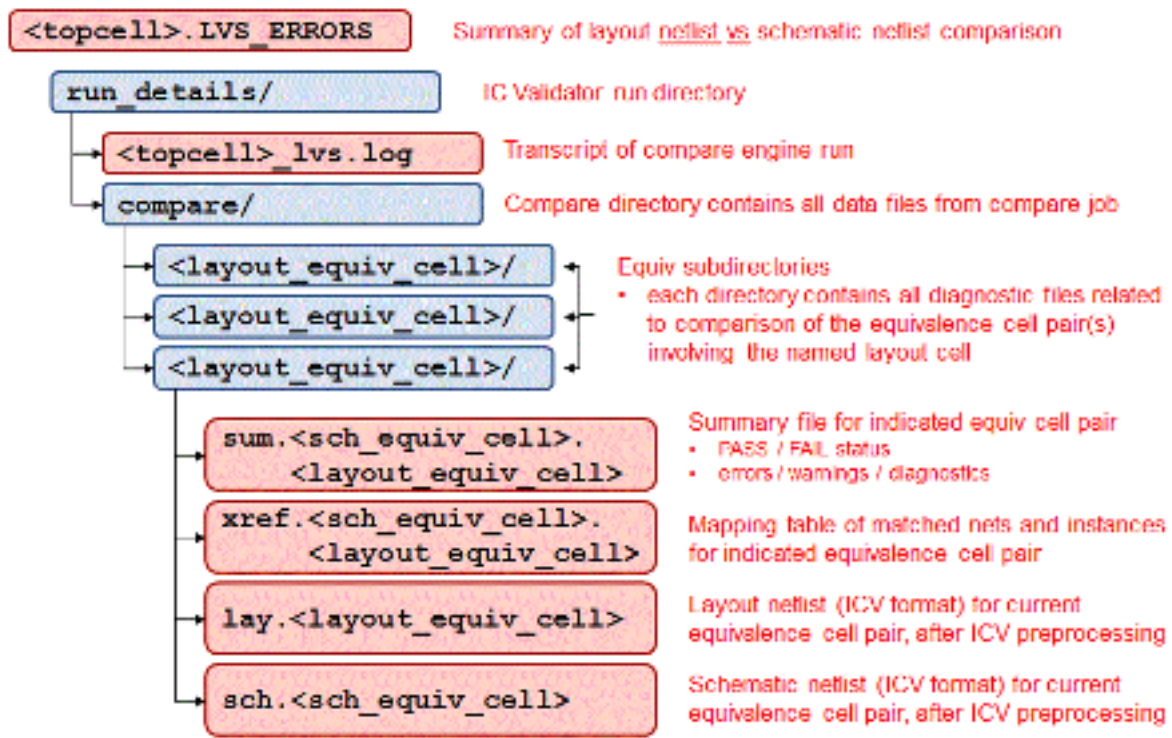
Note:

The `equate_by_net_name_fails` argument of the `match_conditions()` function reports net names found in both the schematic and layout, but cannot be used as an initial match reference point because the number of connections to the net is not the same between the two cells.

Compare Results

The list of output files generated by IC Validator during a compare run are shown in [Figure 1-19](#). In the current directory, the `topcell.LVS_ERRORS` summary file contains the summary of the comparison, which includes the final comparison result (PASS or FAIL), the number of failed and passed equivalences, debugging messages, and diagnostics.

In the `run_details` directory, the `topcell_lvs.log` file contains detailed messages from the compare engine run.

Figure 1-19 *run_details* Directory

In the *run_details* directory, there is a *compare* directory that contains one subdirectory for each equivalence cell pair. Each of these equivalence cell subdirectories contains four files: the summary of the comparison for the equivalence cell pair, a mapping table, the schematic netlist specific to the equivalence cell pair, and the layout netlist specific to the equivalence cell pair. Of these four files, the *sum.sch_equiv_cell.layout_equiv_cell* file is particularly useful for debugging, as it contains the PASS and FAIL statuses, error and warning information, and diagnostics.

Mismatches between the layout and schematic netlists occur in three main categories, as shown in [Table 1-3](#).

Table 1-3 *Summary File Messages*

Category	IC Validator summary file message
Net mismatches	N potential shorted net(s) in layout
	N potential open net(s) in layout
	N unmatched net group(s)

Table 1-3 Summary File Messages (Continued)

Category	IC Validator summary file message
Instance mismatches	N missing device(s) in layout N extra device(s) in layout
Property errors	N device(s) have mismatch property

All errors are reported in the *topcell.LVS_ERRORS* file and *sum.sch_cell.lay_cell* file.

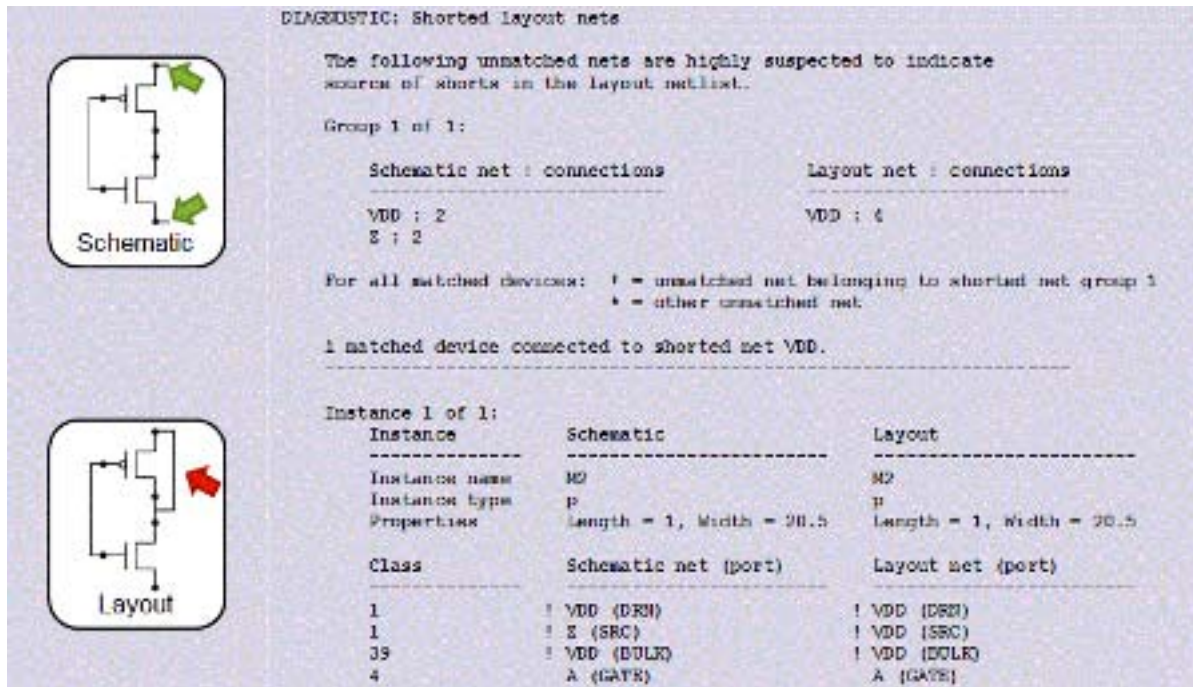
Net Mismatches

For compare opens and shorts, the compare failure report lists:

- Number of open or shorted nets
- Nets that participate in the short or open
- Number of instance connections on each shorted or open net
- Matched instances connected to each shorted or open net
- Instance pin connections for all matched instances connected to each net

Figure 1-20 shows the diagnostic message issued for a short detected in the layout netlist. In the schematic, VDD and Z connect to two instance pins. In the layout, VDD connects to four instance pins and therefore reveals that there is a short.

Figure 1-20 Diagnostic Messages



The second part of the diagnostic shows that there is a matched device on the shorted VDD net called M2. See the list of connections for each pin, which indicates the shorted nets by using an exclamation point (!). Here you see that the schematic M2 SRC pin connects to Z whereas the layout M2 SRC pin connects to VDD. Therefore the short must be near the SRC pin of the device. The Class column is an integer value that indicates instance pin swappability. If two pins have the same class value, they are legally swappable. Since pins DRN and SRC have a class value of 1, these two pins are swappable.

Not all net mismatches are pure opens or shorts. Nets might connect to incorrect pins of a cell or device. Disconnected nets could have unequal or equal connection counts. Such errors are generically reported as unmatched net groups. Consider the schematic and layout circuits shown in Figure 1-21.

Figure 1-21 Net Mismatches



For this case, the IC Validator tool issues a diagnostic summary, as shown in [Figure 1-22](#). In the first part of the summary, the connection counts are equivalent for each unmatched net: SUM has one connection and X has two connections.

The next two summaries compare the connection information for matched devices. Instance 1 is an and2b cell found in the schematic and in the layout. They are each connected to net X which is unmatched. Instance 2 is an invb cell that contains two unmatched nets. The names of nets reveal that there is a connection error. Net X on the layout must be connected to the input of the inverter and net SUM must be connected to the output of the inverter.

Figure 1-22 Diagnostic Summary

```
Diagnostic summary.

1 unmatched net group

DIAGNOSTIC: Unmatched nets

Unmatched schematic and layout nets are grouped for cross probing.

Group 1 of 1:

Schematic net : connections          Layout net : connections
-----
SUM : 1                               SUM : 1
X : 2                               X : 2

For all matched devices:  ! = unmatched net belonging to net group 1
                        * = other unmatched net

2 matched devices connected to unmatched nets

-----

Instance 1 of 2:
  Instance      Schematic          Layout
-----
  Instance name  x34               X3
  Instance type  and2b             and2b

  Class          Schematic net (port)  Layout net (port)
-----
  1              ! X (OUT)              ! X (OUT)

Instance 2 of 2:
  Instance      Schematic          Layout
-----
  Instance name  x38               X12
  Instance type  invb             invb

  Class          Schematic net (port)  Layout net (port)
-----
  2              ! X (A)               ! SUM (A)
  3              ! SUM (S)             ! X (S)
```

Instance Mismatches

Instance mismatches might impact either device or cell instances. There are two types of mismatches: missing instances or extra instances. In other words, a layout equivalence cell either has fewer or more instances of a particular device as compared with the schematic.

Missing or extra instances are flagged only after filtering and device merging has occurred. Device filtering and merging are configured by the LVS runset using the `filter()`, `merge_parallel()`, and `merge_series()` functions.

Figure 1-23 shows a diagnostic message for missing layout devices.

Figure 1-23 Diagnostic Summary for Missing Layout Devices

```
Diagnostic summary:

  2 missing devices in layout

DIAGNOSTIC: Missing layout instances

The following schematic instances are missing in the
layout netlist.

      Schematic instance (type)                Layout instance
      -----
      M4 (p)                                * Missing instance
      M3 (n)                                * Missing instance
```

Missing layout devices occur in the following situations:

- Devices are formed incorrectly in the layout, that is, a missing terminal.
- Devices are promoted up the hierarchy.
- Devices match the compare filtering rule and are filtered.
- Devices are unexpectedly merged (series or parallel) with other devices.

Figure 1-24 shows the format of a diagnostic message for extra layout devices.

Figure 1-24 Format of a Diagnostic Message

```

Diagnostic summary:

    3 extra devices in layout

DIAGNOSTIC: Extra layout instances

The following layout instances are extra compared to the
schematic netlist.

      Schematic instance      Layout instance (type) (x, y)
      -----
      * Missing instance      M4 (p) (17.000, 35.750)
      * Missing instance      M3 (n) (7.500, 10.500)
      * Missing instance      M2 (n) (11.500, 10.500)

```

In this figure, the “p” device and the two “n” devices have no match in the schematic. Notice that the coordinates of the extra devices are provided.

Extra layout devices are created in the following situations:

- Extra devices are erroneously drawn in the layout.
- Devices are not filtered as expected (connectivity problem).
- Devices failed to merge with neighboring series or parallel devices (connectivity error or property error).

To debug instance mismatches, use the netlist statistics tables in the `sum.sch_cell.lay_cell` file. [Figure 1-25](#) shows the statistics report. Notice that there are statistics tables specific to the schematic and layout netlists.

Figure 1-25 Statistics Report for Schematic and Layout Netlists

Statistics Report								
Schematic netlist statistics after filtering and merging								
Initial	PushDown	Filter	Parallel	Path/Ser	RecogGate	Final	Device type	
3	0	0	0	0	0	3	n	
3	0	0	0	0	0	3	p	
6	0	0	0	0	0	6	Total devices	
Initial	PushDown	Dangle	0 Connect	Path/Ser	RecogGate	Shorted	Total nets	
8	0	0	0	0	0	0	8	
Layout netlist statistics after filtering and merging								
Initial	PushDown	Filter	Parallel	Path/Ser	RecogGate	Final	Device type	
11	0	-1	-8	0	0	2	n	
6	0	-1	-3	0	0	2	p	
17	0	-2	-11	0	0	4	Total devices	
Initial	PushDown	Dangle	0 Connect	Path/Ser	RecogGate	Shorted	Total nets	
8	0	0	0	0	0	0	8	

There are three key components in the statistics tables:

- Initial column. Displays the netlist instance counts before merging and filtering.
- Filter, Parallel, and Path/Ser columns. Reports the device reduction due to filtering, parallel merging, and path and series merging.
- Final column. Reports the final counts after filtering, parallel merging, and path and series merging.

Note:

If the schematic final count does not match the layout final count for any device type, a mismatch error occurs. In [Figure 1-25](#), the layout is missing one “n” device and one “p” device.

Property Errors

Property errors result from mismatched property values for matched devices. By default, property checking occurs only when the cells being compared are topologically equivalent, that is, when there are no mismatched nets and no mismatched devices. However, the `check_property_for_failed_equiv` argument of the `compare()` function enables properties to be checked for matched devices, even if there are other topological errors.

In [Figure 1-26](#), two devices contain property mismatches. In the first instance, the “n” device has a length mismatch. In the second instance, the “p” device has length and width mismatches.

Figure 1-26 Property Mismatches

ERROR: Property mismatch

Device 1 of 2:		
Instance	Schematic	Layout
-----	-----	-----
Instance name	M2	M2
Instance type	n	n
Properties	Length = 0.045, Width = 0.09	Length = 0.048, Width = 0.09
Mismatched properties:		
Tolerance		

[-0.1%,0.1%]	Length = 0.045	Length = 0.048
Device 2 of 2:		
Instance	Schematic	Layout
-----	-----	-----
Instance name	M5	M5
Instance type	p	p
Properties	Length = 0.045, Width = 0.09	Length = 0.048, Width = 0.10
Mismatched properties:		
Tolerance		

[-0.1%,0.1%]	Length = 0.045	Length = 0.048
[-0.5%,0.5%]	Width = 0.09	Width = 0.10

2

Netlist Formats

This chapter describes NetTran (a netlist translation utility) and the SPICE and Verilog netlist formats.

This chapter has the following sections:

- [NetTran](#)
- [SPICE Netlist Format](#)
- [Verilog Netlist Format](#)

NetTran

NetTran is a netlist translation utility. You can run it from the UNIX command line, or it can be called within an IC Validator runset.

The IC Validator `schematic()` and `read_layout_netlist()` functions read input netlists and translate them to IC Validator netlist format if the netlists are not already in that format. NetTran does the translation. The results of the `schematic()` and `read_layout_netlist()` functions are used by the `compare()` function.

NetTran has the following capabilities:

- Translates a standard netlist format to an IC Validator netlist format
- Creates a skeletal equivalence file during any translation
- Creates a wire resolution log file during any translation
- Duplicates the instances that have multiplication factor (m) to expand the instance, thereby ensuring the correctness of the device number

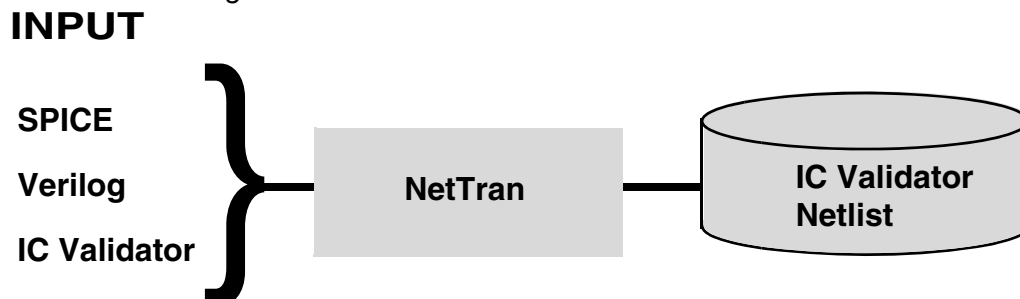
NetTran is described in the following sections:

- [Translating Netlists to IC Validator Format Using NetTran](#)
- [Command-Line Syntax for NetTran](#)
- [Translating Standard Netlists to IC Validator Format](#)
- [Creating a Skeletal Equivalence File](#)
- [Wire Resolution Log File](#)
- [Comments in Output Netlist](#)

Translating Netlists to IC Validator Format Using NetTran

[Figure 2-1](#) shows types of netlists that can be converted using NetTran.

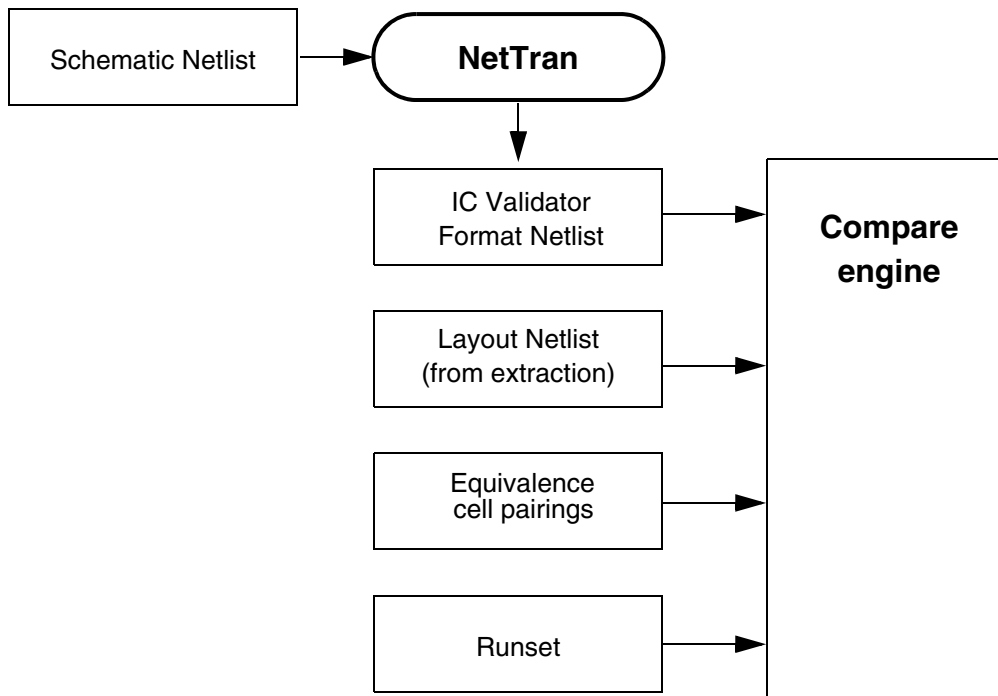
Figure 2-1 Converting Netlists



Netlist Translation

Schematic netlists are compared to the extracted layout netlist during LVS. The schematic netlist must be translated to the IC Validator format before the IC Validator tool can use it for LVS comparison, as shown in [Figure 2-2](#). The IC Validator netlist format contains a hierarchical description of schematic devices and their interconnections.

Figure 2-2 LVS Process Flow



Skeletal Equivalence File

A skeletal equivalence file contains an entry for each schematic cell definition found in the netlist during translation. Each associated layout structure entry is named the same as the schematic cell name. You must edit the layout name entries to set the proper equivalence point for any cases where the schematic name and layout name are not identical.

Wire Resolution Log File

The wire resolution log file (or wireLog file) is a record of all of the shorted nets for each cell. If two nets are shorted, the nets are replaced by one net name. The file can be used to cross-reference new net names with the original net names. NetTran automatically resolves the wire constructs in IC Validator netlists, but if you want to have a log of all the wire constructs, you must use the `-wireLog` NetTran command-line option.

Command-Line Syntax for NetTran

The command-line syntax for NetTran is

```
icv_nettran [-V] [-Version] [-usage]
  [-icv file ...] [-sp file ...] [-verilog file ...]
  [-globalNets netName ...] [-uppercase]
  [-sp-chopDevPrefix] [-sp-chopXPrefix]
  [-sp-devMap map_file] [-sp-dollarPins] [-sp-dummyNets]
  [-sp-resolveDupInstances] [-sp-fshort model_name...]
  [-sp-ignoreCdlResi] [-sp-multiplier name ...]
  [-sp-dupPort WARNING | ABORT] [-sp-voltThresh number]
  [-verilog-b0 netName] [-verilog-b1 netName] [-verilog-busLSB]
  [-verilog-localizeGlobalSupply] [-verilog-localizeModuleSupply]
  [-verilog-preferModuleSupply] [-verilog-addDummyDev devName]
  [-verilog-R] [-verilog-voltmap filename]
  [-acctFile file] [-cell cell] [-cellPorts portName] [-clf file]
  [-comp] [-dupCell USE_MULTIPLE | USE_ONE | ABORT]
  [-dupCellReportFile file] [-noFloatingPins]
  [-equiv file] [-finst instance_name ...]
  [-fopen model_name...] [-forceGlobalsOn] [-logFile file]
  [-mprop] [-noflatten]
  [-undefFile file] [-wireLog file]
  [-outType netlist_type]
  [-retainComments netlist_type]
  -outName file
```

[Table 2-1](#) describes the general purpose NetTran command-line options. [Table 2-2](#) and [Table 2-3](#) describe the command-line options specific to SPICE and Verilog.

Table 2-1 General Purpose NetTran Command-Line Options

Option	Description
<code>-acctFile file</code>	Writes an accounting file.
<code>-cell cell</code>	Sets the top cell for output netlist. Only cells contained in the hierarchy of the top cell are output. If <code>-uppercase</code> is also specified, <code>cell</code> can be specified using either the original cell name or the cell name in uppercase characters.
<code>-cellPorts portName ...</code>	Adds ports to top-cell port list. You must use the <code>-cell</code> option to specify the top cell.
<code>-clf file</code>	<p>Adds command-line options to the NetTran run using a text file. It performs a left-to-right replacement with the options from the specified file. This command-line option takes only one file, but you can use multiple <code>-clf</code> options on a command line.</p> <p>The content of the text file is not preprocessed by the UNIX shell, such as for wildcard expansion.</p> <p>NetTran expands text in the file that has a \$ sign at the start of a variable as an environment variable. For example, the MY_PATH environment variable set on the command line as</p> <pre>% setenv MY_PATH /u/path</pre> <p>could be used in the text file to set the path for the input netlist in SPICE format:</p> <pre>-sp \$MY_PATH/M0.sp</pre>
<code>-comp</code>	Compresses the output netlist.
<code>-dupCell USE_MULTIPLE USE_ONE ABORT</code>	<p>Specifies how to handle multiple cell definitions that have the same name. The default is <code>USE_MULTIPLE</code>.</p> <p>See the <code>duplicate_cell</code> argument of the <code>read_layout_netlist()</code> and <code>schematic()</code> functions for more information.</p>
<code>-dupCellReportFile file</code>	Specifies the duplicate cell report file name. The default name is <code>dupCell.log</code> .
<code>-equiv file</code>	Specifies the skeletal output equivalence file name.
<code>-first instance_name</code>	Filters out devices with a specific instance name; the wildcard <code>*</code> is supported. If <code>-uppercase</code> is also specified, <code>instance_name</code> can be specified using either the original instance name or the instance name in uppercase characters.

Table 2-1 General Purpose NetTran Command-Line Options (Continued)

Option	Description
<code>-fopen model_name ...</code>	<p>Filters out devices with the specified model names. The nets that used to be connected to the device terminal are left unconnected, that is, left open. This option applies to all devices. If <code>-uppercase</code> is also specified, <code>model_name</code> can be specified using either the original model name or the model name in uppercase characters.</p> <p><code>-fopen</code> takes a list of model names. For example,</p> <pre>icv_nettran -fopen nch pch nch_dw pch_dw</pre> <p>In the following example, NetTran filters out all devices with the model name EFG.</p> <pre>icv_nettran -sp myfile -fopen EFG</pre>
<code>-forceGlobalsOn</code>	<p>If a cell port has a global name, forces the port to be part of the global net with the same name. The global net name propagates up the hierarchy; that is, NetTran replaces the local name of an instance pin connecting to such a port with the global net name itself.</p> <p>For example,</p> <pre>*.global VSS .SUBCKT top X1 a one .SUBCKT one VSS</pre> <p>The default result, which is when <code>-forceGlobalsOn</code> is not used, is</p> <pre>pin a=VSS</pre> <p>The force global result, which is when <code>-forceGlobalsOn</code> is used, is</p> <pre>pin VSS=VSS</pre>
<code>-globalNets netName ...</code>	<p>Inserts a global net tag into the output netlist for the specified <code>netName</code> nets. Global net names are also propagated onto ports of instances exploded out of a cell by NetTran. If <code>-uppercase</code> is also specified, <code>netName</code> can be specified using either the original net name or the net name in uppercase characters. See “.GLOBAL” on page 2-34 for more information about the <code>.GLOBAL</code> statement.</p>
<code>-icv file ...</code>	Specifies the input netlists in IC Validator format.

Table 2-1 General Purpose NetTran Command-Line Options (Continued)

Option	Description
<code>-logFile file</code>	Specifies the summary log file name. The default name is <code>icv_nettran.log</code> .
<code>-mprop</code>	<p>Netlists the device elements, including primitive SUBCKT keywords, the number of times specified by the multiplication factor (<i>m</i>), each with <i>m</i> equal to 1. (A primitive SUBCKT is an x-card device that refers to an empty SUBCKT definition.) All of the duplicated devices are connected in parallel and are identical except for their names. The duplicated devices inherit the original device name followed by the suffix "<code>_NETTRAN_m</code>", where <i>m</i> starts at 1.</p> <p>This feature takes effect only when the <code>-outType</code> command-line option is <code>ICV</code>. The default is <code>ICV</code>.</p> <p>Note: The SUBCKT calls (x-cards or x-elements) with a multiplication factor are always expanded and, therefore, are not affect by this option.</p>
<code>-noflatten</code>	Prevents flattening of the hierarchy by NetTran to resolve parameters. By default, NetTran flattens the cells whose parameters refer to other cells so that no parameters are included in the output netlist. If you use this option, NetTran cannot handle equations with parameters.
<code>-noFloatingPins</code>	<p>Reports an error if there are floating pins; otherwise, a warning is reported.</p> <p>See the <code>floating_pins</code> argument of the <code>read_layout_netlist()</code> and <code>schematic()</code> functions for more information.</p>
<code>-outName file</code>	Specifies the output netlist name.
<code>-outType netlist_type</code>	<p>Specifies the output netlist type: <code>ICV</code>, <code>SPICE</code>, <code>VERILOG</code>. The default is <code>ICV</code>.</p> <p>Limitation: Only use the <code>SPICE</code> netlist type to translate <code>VERILOG+SPICE</code>. Other input combinations, such as <code>ICV+SPICE</code>, might not give correct results.</p>

Table 2-1 General Purpose NetTran Command-Line Options (Continued)

Option	Description
<code>-retainComments</code> <code>netlist_type</code>	Retains comments from the input netlist in the output netlist. The netlist type can be <code>SPICE</code> or <code>ICV</code> . See “Comments in Output Netlist” on page 2-15 for more information about how comments appear in the output file. You can also use the <code>retain_comments</code> argument of the <code>read_layout_netlist()</code> and <code>schematic()</code> functions to retain comments. This feature is available when the <code>-outType</code> command-line option is <code>SPICE</code> or <code>ICV</code> . The default is <code>ICV</code> .
<code>-sp file ...</code>	Specifies the input netlists in <code>SPICE</code> format.
<code>-undefFile file</code>	Specifies the file name to which undefined cells are written.
<code>-uppercase</code>	Converts NetTran input to uppercase characters.
<code>-usage</code>	Prints NetTran usage.
<code>-V</code>	Prints product version.
<code>-verilog file ...</code>	Specifies the input netlists in Verilog format.
<code>-Version</code>	Prints product and library version.
<code>-wireLog file</code>	Specifies the dissolved nets log file name.

[Table 2-2](#) describes the NetTran command-line options for the translation of `SPICE` netlists.

Table 2-2 SPICE Translation Command-Line Options

Option	Description
<code>-sp-chopDevPrefix</code>	Removes the leading R, L, C, Q, D, and M prefixes in <code>SPICE</code> instance names during netlist read-in and before any netlist processing, such as flattening.
<code>-sp-chopXPrefix</code>	Trims the prefix x of instance names.
<code>-sp-devMap map_file</code>	Maps device model names to new names using a predefined mapping file format.
<code>-sp-dollarPins</code>	Writes <code>SPICE</code> instance names using the <code>\$PINS</code> statement.

Table 2-2 SPICE Translation Command-Line Options (Continued)

Option	Description
<code>-sp-dummyNets</code>	Generates dummy nets for unconnected pins when writing SPICE instance names using the \$PINS statement.
<code>-sp-dupPort WARNING ABORT</code>	Controls the duplicate ports in the SPICE cell port list (implicitly defined). For a \$PINS statement, duplicate pins are not allowed. The default is <code>WARNING</code> . See the <code>duplicate_port</code> option in the <code>spice_settings</code> argument of the <code>read_layout_netlist()</code> and <code>schematic()</code> functions for more information.
<code>-sp-fshort model_name ...</code>	Filters resistor, capacitor, and inductor passive devices with the specified model name. Nets connecting the A and B pins of each device instance are shorted in the translated netlist. If <code>-uppercase</code> is also specified, <code>model_name</code> can be specified using either the original model name or the model name in uppercase characters. When using this option, net-naming rules for resolved nets are as follows, in order of highest to lowest priority: <ol style="list-style-type: none"> 1. Global nets are selected. 2. Port nets are selected. 3. Nets are selected, based on alphabetical order.
<code>-sp-ignoreCdlResi</code>	Ignores *.RESI statements in CDL and SPICE input.
<code>-sp-resolveDupInstances</code>	Resolves duplicate instance names found within the same cell.

Table 2-2 SPICE Translation Command-Line Options (Continued)

Option	Description
<code>-sp-multiplier name ...</code>	<p>Lists user-defined multiplier factors. When using this option, different instances and devices can have unique multiplier factors. When this option is not used, the default is "M".</p> <p>See the <code>spice_multiplier_names</code> argument of the <code>lvs_options()</code> function for more information.</p> <p>In the following example, the multiplier option is not used:</p> <pre>icv_nettran -sp in.sp -outName out</pre> <p>Therefore, for the following netlist input, M is a multiplier:</p> <pre>X1 G D S B nch M=2</pre> <p>For the following netlist input, MULT is a common parameter, not a multiplier:</p> <pre>X2 G D S B nch MULT=2</pre> <p>For the following netlist input, MULT is renamed to M and considered a multiplier:</p> <pre>M1 G D S B nch MULT=2</pre> <p>In the following example, the multiplier option is used:</p> <pre>icv_nettran -sp in.sp -sp-multiplier MA MB -outName out</pre> <p>For the following netlist input, an error occurs because both MA and MB multipliers are used:</p> <pre>X1 G D S B nch MA=2 MB=3</pre> <p>The following is another example of using the multiplier option:</p> <pre>icv_nettran -sp in.sp -sp-multiplier MA MB -outName out</pre> <p>For the following netlist input, MA is a multiplier but no expansion is performed:</p> <pre>M1 G D S B nch MA=2</pre> <p>In the following example, the multiplier and expansion options are used:</p> <pre>icv_nettran -sp in.sp -sp-multiplier MA MB -outName out -mprop</pre> <p>For the following netlist input, MA is a multiplier and expansion is performed:</p> <pre>M1 G D S B nch MA=2</pre>

Table 2-2 SPICE Translation Command-Line Options (Continued)

Option	Description
<code>-sp-scale scalefactor</code>	Overwrites the scaling factor (<code>*.SCALE</code> and <code>.OPTION SCALE</code>) for all input netlists in the SPICE format.
<code>-sp-voltThresh number</code>	Specifies the voltage source threshold value for shorts. The default is 0. <ul style="list-style-type: none"> Shorts voltage sources with a value \leq <i>number</i>. Strips voltage sources with a value $>$ <i>number</i>.

Table 2-3 describes the NetTran command-line options for the translation of Verilog netlists.

Table 2-3 Verilog Translation Command-Line Options

Option	Description
<code>-verilog-addDummyDev devName</code>	Adds dummy resistors to ports of the top cell in the Verilog netlist.
<code>-verilog-b0 netName</code>	Specifies the Verilog global ground net.
<code>-verilog-b1 netName</code>	Specifies the Verilog global power net.
<code>-verilog-busLSB</code>	Specifies that the Verilog bus starts with the least significant bit (0).
<code>-verilog-localizeGlobal Supply</code>	Disables the global net generation of global supply nets. See “Supply Net Translation Precedence” on page 2-45 . You can disable all global net generation in this numeric value translation by specifying the <code>-verilog-localizeModuleSupply</code> and <code>-verilog-localizeGlobalSupply</code> command-line options.
<code>-verilog-localizeModule Supply</code>	Changes the precedence of the net names used to replace numeric values so as to use local supply nets first. Global net generation of local supply nets is disabled as well. See “Supply Net Translation Precedence” on page 2-45 . Prompts an error if the <code>-verilog-preferModuleSupply</code> and <code>-verilog-localizeModuleSupply</code> command-line options are specified, as they are exclusive.
<code>-verilog-preferModule Supply</code>	Changes the precedence of the net names used to replace numeric values so as to use local supply nets first. See “Supply Net Translation Precedence” on page 2-45 .

Table 2-3 Verilog Translation Command-Line Options (Continued)

Option	Description
<code>-verilog-R</code>	Retains backslash (\) on Verilog net names.
<code>-verilog-voltmap filename</code>	Specifies the Verilog global net mapping file. If <code>-uppercase</code> is also specified, names of cells, instances, and nets in the file can be specified using either the original names or the names in uppercase characters.

Translating Standard Netlists to IC Validator Format

SPICE and Verilog netlists can be translated to the IC Validator format. The unit for geometric device properties in the IC Validator format is micron, whereas for SPICE it is meter.

SPICE netlists are different from IC Validator netlists in that SPICE netlists have implicit referencing. The order of the ports is fixed. In contrast, in the IC Validator netlist format, referencing is explicit because the port names are given, and therefore, the ports can exist in any order.

NetTran supports structural, not behavioral, Verilog. A Verilog netlist uses modules to define cells. It requires that every port and every net be previously defined as input, output, inout, or wire before being used. The IC Validator tool does not make use of this definition.

NetTran reports a warning message when it detects a floating pin, unless you use the `-noFloatingPins` command-line option. For the port that does not have a corresponding pin connection, a dummy net, with the name `icv_floatnet_xx`, is added. For example,

```
module nor3b(GND, VDD, A, B, QN)
    ...
endmodule

nor3b  x54  (.GND(GND), .A(A), .B(B), .QN(n32)); // missing VDD port

{inst x54=nor3b {TYPE CELL}
{pin GND=GND icv_floatnet_1=VDD A=A B=B n32=QN}}
```

The syntax for a Verilog netlist is of the form `.BULK(VDD)`, whereas the syntax for an IC Validator netlist is of the form `VDD=BULK`.

Property Values

Although standard netlist formats allow property values in several measurement units, NetTran accepts only specific units, as shown in [Table 2-4](#).

Table 2-4 Property Value

Character	Name	Value
f	femto	1e-15
p	pico	1e-12
n	nano	1e-9
u	micro	1e-6
m	milli	1e-3
K	kilo	1e+3
M	mega	1e+6
G	giga	1e+9
T	tera	1e+12

SPICE Translation Examples

In the following example, the `-uppercase` option forces all identifier names to uppercase characters, making the netlist case-insensitive. The `-equiv` option generates an equivalence file for the IC Validator tool.

```
% icv_nettran -sp fname.sp -uppercase -equiv equivfile -outName fname.nt
```

In the following example, NetTran converts a SPICE netlist to an IC Validator netlist and resolves all wires or shorts. The wireLog file contains all the nets that are dissolved in the process.

```
% icv_nettran -sp filename.sp -outName filename.nt -wireLog filename.log
```

Creating a WireLog File Using NetTran

The wireLog file reports lists of nets that are shorted by one of three SPICE shorting operations, `.CONNECT`, `*.CONNECT`, or `*.EQUIV`. NetTran automatically resolves all shorted net names.

To generate a log file of all wire construction, set the `-wireLog` NetTran command-line option. The wireLog file reports the original preshorted net name and the new postshorted net name.

The syntax is

```
icv_nettran -icv infile -wireLog wire_log_file -outName netlist
```

The structure of the wireLog file is

```
cell origNetName newNetName
. . .
cell origNetName newNetName
```

For example,

```
# wire.txt - Dissolved nets log file
# format: cellName origNetName newNetName
add4      VDD2      VDD
cs_add    VDD5      VDD
cs_add    n36       VDD
nor2b     VDD6      VDD5
```

CDL Map File

A map file is created by the Virtuoso Schematic Editor CDL out process. The file contains name translations of Virtuoso schematic database names to legal CDL names. You can map net, instance, and SUBCKT names using this method. The resulting netlist from NetTran contains the Virtuoso schematic database names, as shown in the first item in any list of two items in [Example 2-1](#).

Example 2-1 Example Map File

```
(( nil
version "2.1"
mapType "hierarchical"
repList "cdl schematic cmos.sch gate.sch symbol"
stopList "cdl"
globalList "vbb! vpp! vbleq! vblh! vint! gnd!"
  hierDelim "."
)
( net
( "vbleq!" "vbleq" )
( "vpp!" "vpp" )
( "vbb!" "vbb" )
)
( inst
( "IA<4>" "XIA4" )
( "IB<37>" "XIB37" )
( "IA<365>" "XIA365" )
)
( model
( "nfetlo" "M1" )
```

```
)
)
```

In this example, the `-sp-devMap` option loads a Cadence® name mapping file.

```
icv_nettran -sp filename.cdl -sp-devMap mapfile -outName filename.nt
```

Net, instance, and model names in the right column of the associated map file structures are replaced in the NetTran output netlist using names from the left column.

Verilog Translation Example

An example of the Verilog translation syntax is

```
icv_nettran -verilog infile infile2 -outName netlist
```

Creating a Skeletal Equivalence File

An equivalence file is used during LVS compare to list each schematic cell and the corresponding layout cell. NetTran can create a skeletal equivalence file during any netlist translation. The equivalence file that NetTran creates is skeletal because you must edit the layout name entries to set the equivalence points for any instances where the schematic name and layout name are not identical. The syntax is

```
icv_nettran -icv infile -equiv file -outName netlist
```

Comments in Output Netlist

When you use the `-retainComments` command-line option, comments specified in the input netlist are output to the ICV format netlist. Because to the order of cells and instances in the output are different from the input, the comments are attached to either a cell or an instance. NetTran classifies the comments into three types:

- [Cell-Specified Comments](#)
- [Instance-Specified Comments](#)
- [Global Comments](#)

Cell-Specified Comments

The cell-specified comments are related to a SUBCKT definition. The comments are written to the immediately previous line or the same line of the SUBCKT definition. Blank lines are ignored.

For example, the input SPICE netlist has these comments:

```

* this is my cell
.SUBCKT my_cell port1 port2 port3

.SUBCKT my_cell port1 port2 port3 * this is my cell
.SUBCKT my_cell port1 * this is my cell
+ port2 port3

```

NetTran attaches the comment "this is my cell" to the my_cell cell for the three cases. The output netlist has this comment:

```

/* this is my cell */
{cell my_cell
{port port1 port2 port3 }
... Cell instances ...
}

```

NetTran merges cell-specified comments that are more than one line and removes the blank lines. For example, the input SPICE netlist has these comments:

```

* this is my cell
*
* edited by Joe
.SUBCKT my_cell port1 port2 port3 * there are 3 ports

```

The output netlist has this comment:

```

/*
this is my cell
edited by Joe
there are 3 ports
*/
{cell my_cell
{port port1 port2 port3 }
... Cell instances ...
}

```

When comments are at the end of subckt definitions after the last instance definition in the subckt scope or on the same line as the .ends statement, NetTran prints the comments after the cell definition. For example, the input SPICE netlist has these comments:

```

.SUBCKT MY_CELL a b
M1 a b c d nch w=5 l=2
.ENDS * the end of my cell

.SUBCKT MY_CELL a b
M1 a b c d nch w=5 l=2
* the end of my cell
.ENDS

```

The output netlist has one comment:

```

{cell MY_CELL
{port a b}

```



```
{inst M1=nch {TYPE MOS}
{prop L=2e6 W=5e6}
{pin a=DRN b=GATE c=SRC d=BULK}}
/* the end of my cell */
}
```

Instance-Specified Comments

The instance-specified comments are related to an instance or a device. The comments are written to the immediately previous line or the same line of the instance or device definition. Blank lines are ignored. For example, the input SPICE netlist has these comments:

```
* this is my input resistor
R1 1 2 10MEG

R1 1 2 10MEG * this is my input resistor

R1 1 2 * this is my input resistor
+ 10MEG
```

The output netlist has one comment:

```
/* this is my input resistor */
{inst R4=R {TYPE RES}}
```

If there is more than one comment attached to an instance, NetTran merges the comments and removes blank lines. NetTran does not translate the SPICE power source, such as the E-card and V-card devices, so it puts the comments immediately below the instance. For example, the input SPICE netlist has this comment:

```
* DC GAIN (100K) AND POLE 1 (100HZ)
EP1 3 0 1 2 100K
RP1 3 4 1K
```

Because the EP1 instance is ignored by NetTran, NetTran attaches the comment to the next available element, that is, RP1. The output netlist has the comment:

```
/* DC GAIN (100K) AND POLE 1 (100HZ) */
{inst RP1=R {TYPE RES}
{pin 3=A 4=B}}
{prop R=1000}
}
```

Global Comments

If a comment cannot be attached to any instance, it is treated as a global comment at the end of the output netlist. For example, the input SPICE netlist has these comments:

```
* ANALYSIS
.TRAN 0.01MS 0.2MS
* VIEW RESULTS
.PLOT TRAN V(1) V(5)
```

```
<end of file>
```

Because NetTran does not translate the .TRAN and .PLOT statements, the two comments ANALYSIS and VIEW RESULTS are put in the last part of the output netlist. To distinguish the global comments from multiple input files, NetTran prints the file name before the comment. The output netlist has the comment:

```
<cell/instance of ICV netlist>
```

```
/* test.sp */
/*
  ANALYSIS
  VIEW RESULTS
*/
```

SPICE Netlist Format

All geometric properties have a default unit of meter or meter².

Devices

The SPICE element definitions are:

Bipolar Transistor

The syntax is:

```
Qxxx coll base emit [bulk] mname
+ [[AREA=]area] [L=l] [W=w] [M=m]
+ [$SUB=bulk] [$EA=area] [$L=l] [$W=w] [$X=x] [$Y=y]
```

For example,

```
Q1 a b c gnd npn9 AREA=9p
Q2 a b c npn9 9p
Q3 a b c npn10x10 AREA=9p W=10u L=10u M=9 $SUB=GND
```

Table 2-5 *Bipolar Transistor Arguments*

Argument	Description
<i>Qxxx</i>	Bipolar element name. It must begin with a Q.
<i>coll</i>	Bipolar collector node name.
<i>base</i>	Bipolar base node name.

Table 2-5 Bipolar Transistor Arguments (Continued)

Argument	Description
<i>emit</i>	Bipolar emitter node name.
<i>bulk</i>	Optional. Represents the bipolar bulk node name.
<i>mname</i>	Represents the element model name.
[AREA=] <i>area</i>	Optional. Represents the bipolar emitter area value. It must be a numeric value or a parameter.
L= <i>l</i>	Optional. Represents the bipolar emitter length.
W= <i>w</i>	Optional. Represents the bipolar emitter width.
M= <i>m</i>	Optional. Represents the multiplier to simulate multiple parallel elements. The default is 1.
\$SUB= <i>bulk</i>	Optional. Represents the bulk node for the element. It overrides the regular optional bulk node name, if present.
\$EA= <i>area</i>	Optional. Represents the bipolar emitter area value. It overrides the regular [AREA=] <i>area</i> , if present.
\$L= <i>l</i>	Optional. Represents the length. It overrides the [L= <i>l</i>], if present.
\$W= <i>w</i>	Optional. Represents the width. It overrides the [W= <i>w</i>], if present.
\$X= <i>x</i> \$Y= <i>y</i>	Optional. Represents the device coordinates.

Capacitor

Syntax 1:

```
Cxxx a b [[capacitance] mname]
+ [L=l] [W=w] [A=a] [P=p] [M=m]
+ [$[mname] | $.MODEL=mname]
+ [$SUB=bulk] [$A=a] [$P=p] [$X=x] [$Y=y]
```

Syntax 2:

```
Cxxx a b [[mname] C=capacitance]
+ [L=l] [W=w] [A=a] [P=p] [M=m]
+ [$[mname] | $.MODEL=mname]
+ [$SUB=bulk] [$A=a] [$P=p] [$X=x] [$Y=y]
```

Note:

In Syntax 1, *mname* and *capacitance* can be swapped. The *mname* argument is chosen from one of them, as it is nonnumeric.

For example,

```
C1 a b ncap C=10
C2 a b 20 $.MODEL=ncap
C3 a b C=30 W=5 L=10 $[ncap] $SUB=GND
```

Table 2-6 Capacitor Arguments

Argument	Description
Cxxx	Capacitor element name. It must begin with a C.
a	Capacitor positive node name.
b	Capacitor negative node name.
mname	Optional. Represents the element model name. It must follow after the capacitor negative node name.
[C=]capacitance	Optional. Represents the capacitor value. It must be a numeric value or a parameter.
L=l	Optional. Represents the capacitor length.
W=w	Optional. Represents the capacitor width.
A=a	Optional. Represents the capacitor area.
P=p	Optional. Represents the capacitor perimeter.
M=m	Optional. Represents the multiplier to simulate multiple parallel elements. The default is 1.
\$[mname] or \$.MODEL=mname	Optional. Represents the element model name. It overrides the regular optional <i>mname</i> parameter.
\$SUB=bulk	Optional. Represent the first optional node for the element.
\$A=a	Optional. Represents the area. It overrides the regular [A=a], if present.
\$P=p	Optional. Represents the perimeter. It overrides the [P=p], if present.
\$X=x \$Y=y	Optional. Represents the device coordinates.

Diode

Syntax 1:

```
Dxxx a b mname [AREA=area] [PJ=pj]
+ [M=m]
+ [$SUB=bulk]
+ [$X=x] [$Y=y]
```

Syntax 2:

```
Dxxx a b mname [area [pj]]
+ [M=m]
+ [$SUB=bulk]
+ [$X=x] [$Y=y]
```

For example,

```
D1 a b nd 10
D2 a b pdio AREA=20
D3 a b ndio AREA=1 PJ=4 $SUB=GND
```

Table 2-7 Diode Arguments

Argument	Description
Dxxx	Diode element name. It must begin with a D.
a	Diode anode node name.
b	Diode cathode node name.
mname	Represents the element model name.
[AREA=]area	Optional. Represents the diode area value. It must be a numeric value or a parameter.
[PJ]=pj	Optional. Represents the diode periphery value. It must be a numeric value or a parameter.
M=m	Optional. Represents the multiplier to simulate multiple parallel elements. The default is 1.
\$SUB=bulk	Optional. Represent the first optional node for the element.
\$X=x \$Y=y	Optional. Represents the device coordinates.

Inductor

The syntax is

```
Lxxx a b [mname] [[L=]inductance]
+ [M=m]
+ [$[mname] | $.MODEL=mname] [$SUB=bulk]
+ [$X=x] [$Y=y]
```

For example,

```
L1 a b iname 50
L2 a b 100 $.MODEL=iname
L3 a b L=200 $[iname] $SUB=GND
```

Table 2-8 Inductor Arguments

Argument	Description
<i>Lxxx</i>	Inductor element name. It must begin with an L.
<i>a</i>	Inductor positive node name.
<i>b</i>	Inductor negative node name.
<i>mname</i>	Optional. Represents the element model name. It must follow after the inductor negative node name.
<i>[L=]inductance</i>	Optional. Represents the unit of inductance in henry (H). It must be a numeric value or a parameter.
<i>M=m</i>	Optional. Represents the multiplier to simulate multiple parallel elements. The default is 1.
<i>[\$[mname] or \$.MODEL=mname</i>	Optional. Represents the element model name. It overrides the regular optional <i>mname</i> parameter.
<i>\$SUB=bulk</i>	Optional. Represents the first optional node for the element.
<i>\$X=x \$Y=y</i>	Optional. Represents the device coordinates.

JFET

The syntax is

```
Jxxx drn gate src [bulk] mname [AREA=area]
+ [L=l] [W=w] [M=m]
+ [$SUB=bulk]
+ [$X=x] [$Y=y]
```

For example,

```
J1 d g s b JN L=2u W=4u
J2 d g s b JN L=2u W=4u M=3 $SUB=gnd
J3 d g s b JN AREA=20 W=5 L=10 $SUB=GND
```

Table 2-9 JFET Arguments

Argument	Description
<i>Jxxx</i>	JFET (Junction gate Field Effect Transistor) element name. It must begin with a J.
<i>drn</i>	JFET drain node name.
<i>gate</i>	JFET gate node name.
<i>src</i>	JFET source node name.
<i>bulk</i>	Optional. JFET bulk node name.
<i>mname</i>	Represents the element model name.
<i>AREA=area</i>	Optional. Represents the JFET area value. It must be a numeric value or a parameter.
<i>L=l</i>	Optional. Represents the length.
<i>W=w</i>	Optional. Represents the width.
<i>M=m</i>	Optional. Represents the multiplier to simulate multiple parallel elements. The default is 1.
<i>\$SUB=bulk</i>	Optional. Represents the first optional node for the element.
<i>\$X=x \$Y=y</i>	Optional. Represents the device coordinates.

MOSFET

The syntax is

```
Mxxx drn gate src [bulk [bulk2 [bulk3 [bulk4]]]] mname
+ [L=l] [W=w] [AD=ad] [AS=as] [PD=pd] [PS=ps] [NRD=nrd] [NRS=nrs]
+ [RDC=rdc] [RSC=rsc] [M=m]
+ [$X=x] [$Y=y]
```

For example,

```
M1 d1 g1 s1 b1 mn W=0.8u L=0.35u
M2 d2 g2 s2 b1 b2 b3 b4 mn W=0.8u L=0.35u M=4
```

M3 d3 g3 s3 gnd mn L=2u W=2u AD=0.5u AS=0.7u PS=0.2u PD=0.3u NRS=0.15

Table 2-10 MOSFET Arguments

Argument	Description
Mxxx	MOSFET (Metal-Oxide-Semiconductor Field Effect Transistor) element name. It must begin with an M.
drn	MOSFET drain node name.
gate	MOSFET gate node name.
src	MOSFET source node name.
bulk[n]	Optional. MOSFET bulk node name. Up to four optional bulk nodes are supported.
mname	Represents the element model name.
L=l	Optional. Represents the MOSFET length.
W=w	Optional. Represents the MOSFET width.
AD=ad	Optional. Represents the MOSFET drain area.
AS=as	Optional. Represents the MOSFET source area.
PD=pd	Optional. Represents the MOSFET drain perimeter.
PS=ps	Optional. Represents the MOSFET source perimeter.
NRD=nrd	Optional. Represents the MOSFET number of squares of the drain diffusion area.
NRS=nrs	Optional. Represents the MOSFET number of squares of source diffusion area.
RDC=rdc	Optional. Represents the MOSFET additional drain resistance due to contact resistance.
RSC=rsc	Optional. Represents the MOSFET additional source resistance due to contact resistance.
M=m	Optional. Represents the multiplier to simulate multiple parallel elements. The default is 1.
\$X=x \$Y=y	Optional. Represents the device coordinates.

Resistor

Syntax 1:

```
Rxxx a b [[resistance] mname]
+ [L=l] [W=w] [M=m]
+ [$[mname] | $.MODEL=mname]
+ [$SUB=bulk] [$L=l] [$W=w] [$X=x]
+ [$Y=y]
```

Syntax 2:

```
Rxxx a b [[mname] R=resistance]
+ [L=l] [W=w] [M=m]
+ [$[mname] | $.MODEL=mname]
+ [$SUB=bulk] [$L=l] [$W=w] [$X=x]
+ [$Y=y]
```

Note:

In Syntax 1, *mname* and *resistance* can be swapped. The *mname* argument is chosen from one of them, as it is nonnumeric.

For example,

```
R1 a b ndres R=10
R2 a b 20 $.MODEL=pdres
R3 a b R=30 W=5 L=10 $[ndres] $SUB=GND
```

Table 2-11 Resistor Arguments

Argument	Description
<i>Rxxx</i>	Resistor element name. It must begin with an R.
<i>a</i>	Resistor positive node name.
<i>b</i>	Resistor negative node name.
<i>mname</i>	Optional. Represents the element model name. It must follow after the resistor negative node name.
<i>R=resistance</i>	Optional. Represents the resistor value. It must be a numeric value or a parameter.
<i>L=l</i>	Optional. Represents the resistor length.
<i>W=w</i>	Optional. Represents the resistor width.
<i>M=m</i>	Optional. Represents the multiplier to simulate multiple parallel elements. The default is 1.

Table 2-11 Resistor Arguments (Continued)

Argument	Description
<code>\$[mname]</code> or <code>\$.MODEL=mname</code>	Optional. Represents the element model name. It overrides the regular optional <i>mname</i> parameter.
<code>\$SUB=bulk</code>	Optional. Represent first optional node for the element.
<code>\$L=l</code>	Optional. Represents the length. It overrides the <code>[L=l]</code> , if present.
<code>\$W=w</code>	Optional. Represents the width. It overrides the <code>[W=w]</code> , if present.
<code>\$X=x \$Y=y</code>	Optional. Represents the device coordinates.

Cell Definition

The SPICE format uses `.SUBCKT` syntax to define cells in a netlist, and it ends with an `.ENDS` statement. The syntax is

```
.SUBCKT cell_name [n1 ...] [param=val]
...
.ENDS [cell_name]
```

For example,

Example 1:

```
.SUBCKT INVB1 GND VDD I O
M1 O I GND GND n L=0.7u W=13.5u
M2 O I VDD VDD p L=0.8u W=20.5u
.ENDS
```

Result of Example 1:

```
M1 L=0.7u W=13.5u
M2 L=0.8u W=20.5u
```

Example 2:

```
.SUBCKT INVB2 GND VDD I O L=0.5u
M1 O I GND GND n L=L W=13.5u
M2 O I VDD VDD p L=0.8u W=20.5u
.ENDS
```

Result of Example 2:

```
M1 L=0.5u W=13.5u, where L=L is substituted for SUBCKT param list L=0.5u
M2 L=0.8u W=20.5u, where L=0.8u is not substituted for SUBCKT param list
L=0.5u
```

Example 3:

```
.PARAM WIDTH=0.8u
.SUBCKT INVB3 GND VDD I O L=0.5u
M1 O I GND GND n L=L W=WIDTH
M2 O I VDD VDD p L=0.8u W=20.5u
.ENDS
```

Result of Example 3:

M1 L=0.5u, W=0.8u, where L=L is substituted for SUBCKT param list L=0.5u and W is substituted for WIDTH value 0.8U that is assigned in .PARAM
M2 L=0.5U, W=20.5U, where L=0.8u is substituted by SUBCKT param list L=0.5U

Table 2-12 Cell Definition Arguments

Argument	Description
<code>.SUBCKT <i>cell_name</i></code>	<i>cell_name</i> is the reference name for the cell instance.
<code><i>n1</i> ...</code>	Optional. Node names for external reference. Any element nodes that are in the cell, but are not in this list are strictly local, except nodes assigned using the <code>.GLOBAL</code> or <code>*.GLOBAL</code> commands.
<code><i>param=val</i></code>	Optional. Parameter name set to a value that is applied only in the cell. To override this value, assign the value in the cell instance.
<code>.ENDS <i>cell_name</i></code>	Use the <code>.ENDS</code> statement command to end all cell definitions that begin with a <code>.SUBCKT</code> . Use the <code>.ENDS <i>cell_name</i></code> statement to terminate a cell named <i>cell_name</i> .

Nested Cells Definition

NetTran supports nested cell definitions. With nested cells, the cell definition is inside another cell definition. The nested cell has only a local scope; it can be used from its parent of the same nesting hierarchy and from other cells nested under the same parent. It is not visible from above or below the parent in the nesting hierarchy.

The syntax is:

```
.SUBCKT cell_name [n1 ...] [param=val]
...
    .SUBCKT nested_cell_name [n1 ...] [param=val]
    .ENDS nested_cell_name
...
.ENDS [cell_name]
```

Note:

The `nested_cell_name` statement after the `.ENDS` command is required when the cell is nested.

For example,

```
.SUBCKT IP1 IN OUT
  .SUBCKT CELL1 I O
    M1 O I GND GND N L=0.7U W=13.5U
  .ENDS CELL1
  .SUBCKT CELL2 P N
    C1 P N C=2
    X1 P N CELL1
  .ENDS CELL2
  X2 IN OUT CELL1

.ENDS IP1

.SUBCKT IP2 IN OUT
  .SUBCKT CELL1 I O
    M2 O I VDD VDD P L=0.8U W=20.5U
  .ENDS CELL1 X3
  IN OUT CELL1
.ENDS IP2

.SUBCKT TOP IN OUT
  X4 IN OUT IP1
  X5 IN OUT IP2
.ENDS TOP
```

Inside the IP1 cell statement, the X1 cell instance inside CELL2 is a legal statement, because the CELL1 and CELL2 nested cells are in the same hierarchical scope of cell IP1.

There are two nested cells and two other cells named CELL1. Because cells IP1 and IP2 are not in the same hierarchy, they have different local scopes. Therefore the definitions of CELL1 in this example are legal definitions and do not result in duplicate cells.

Cell Instance Definition

The SPICE format uses `.SUBCKT` syntax to define a cell in the netlist and uses an X followed by characters as the cell instance name.

The syntax is:

```
Xxxx [n1 ...] cell_name [param=val]... [M=m]
Xxxx cell_name [param=val]... [M=m] [$PINS [port1=net1] ...]
```

For example,

```
.SUBCKT CELLA A B C L=4.166u M=4
Q1 A B C PNP L=L M=M
.ENDS
X1 P1 P2 P3 CELLA M=3
X2 N1 N2 N3 CELLA L=5u
```

The following statement shows the equivalent netlist:

```
.SUBCKT CELLA A B C L=4.166u
Q1 A B C PNP L=L M=4
.ENDS
X1_1 P1 P2 P3 CELLA M=1
X1_2 P1 P2 P3 CELLA M=1
X1_3 P1 P2 P3 CELLA M=1
X2 N1 N2 N3 CELLA L=5u
```

The $M=M$ inside cell `CELLA` is directly substituted by statement `SUBCKT param M=4`.

The $L=L$ of cell instance `X1` is directly substituted by cell param $L=4.166u$ of `CELLA`; the $M=3$ of cell instance `X1` statement indicates there are three `X1` instances connected in parallel, and they can be represented as `X1_1`, `X1_2` and `X1_3`. The M value of the original cell instance `X1` is reset to $M=1$ for `X1_1`, `X1_2` and `X1_3`; the M value cannot be passed through the cell `CELLA`.

The $L=5u$ of cell instance `X2` directly substitutes the original cell param $L=4.166u$.

Table 2-13 Cell Instance Arguments

Argument	Description
<code>Xxxx</code>	Cell instance name; must begin with X.
<code>n1 ...</code>	Node names for external reference by pin order. Any element nodes that are in the cell, but are not in this list, are strictly local. Unreferenced node names are considered to be floating pins. The IC Validator default allows floating pins with a warning message. For the port that does not have a corresponding pin connection, a dummy net, with the name <code>icv_floatnet_xx</code> , is added.
<code>cell_name</code>	Cell reference name for the cell instance.

Table 2-13 Cell Instance Arguments (Continued)

Argument	Description
<code>\$PINS port1=net1 ...</code>	<p><code>\$PINS</code> followed by ports assignment is another way to specify a cell ports connection by nets.</p> <ul style="list-style-type: none"> Port connection swap: A cell instance's ports connections swap is allowed. For example: <pre>.SUBCKT CELL A B CENDS X1 CELL \$PINS A=n1 B=n3 C=n2 X2 CELL \$PINS C=n2 A=n1 B=n3</pre> where <code>x1</code> and <code>x2</code> are functionally equivalent; they both have the same port connection assignments, but with different placement orders. Floating pin: A cell instance refers fewer pins than the specified cell definition allows; those pins are not referenced in the cell instance and are considered to be floating pins. For example: <pre>.SUBCKT CELL A B CENDS X1 CELL \$PINS A=n1 C=n2</pre> where the cell definition port <code>B</code> is floating because it is not in the cell instance <code>x1</code> pin assignment list. Unreferenced node names are considered to be floating pins. The IC Validator default allows floating pins with a warning message. For the port that does not have a corresponding pin connection, a dummy net, with the name <code>icv_floatnet_xx</code>, is added. Duplicate ports are not supported within a <code>\$PINS</code> statement. For example: <pre>.SUBCKT CELL PORT1 PORT2 PORT1ENDS X1 CELL \$PINS PORT1=A PORT2=B PORT1=A</pre> where <code>PORT1</code> has duplicate port assignments and IC Validator reports it as an error.
<code>param=val</code>	Optional. Parameter name set to a value that overrides the value in the cell definition.

Table 2-13 Cell Instance Arguments (Continued)

Argument	Description
<i>m</i>	Optional. Represents the multiplier to multiple parallel cell instances. The $M=m$ indicates that there are m individual cell instances connected in parallel, and each of the cell instance parameters, $M=1$. However, parameter M is not considered as other parameters are passed down through the cell, <i>cell_name</i> . The multiplier names can be specified with the NetTran <code>-sp-multiplier</code> command-line option.
<i>"string"</i>	The standard string parameter is indicated by double quotation marks.

String Parameters

The string parameter, like a number parameter, is a basic element to a property value. The string parameter is atomic. Therefore, it cannot be divided or replaced through interpretation. The standard string parameter is indicated by double quotation marks.

The syntax is:

"string"

For example, *"ABC"*, *"-5"*, and *"A56"* are valid string parameter values.

If the double quotation marks are not paired properly, a parse error occurs. For example:

ABC", *"-5*

Double-Quoted String Usage

Double-quoted string parameters are defined with the parameter value: *"val"* of the format *[param="val"]*. The IC Validator tool supports double quotation marks as string properties in addition to the SPICE standard.

Here is an example of the double-quoted strings in the SPICE netlist:

```
.PARAM str1="mode0"
.SUBCKT CELL_1 P1 P2 P3 str2="1.2v"
M1 P1 P2 P3 P4 PMOS W=1 L=2str3 ="-1"
X1 P1 P2 P3 CELL_2 str4="1"
.END
```

The double-quoted strings, *"mode0"*, cell initial parameter, *"1.2v"*, device property, *"-1"*, and cell instance property, *"1"* are all defined as string parameters.

Here is an example of double-quoted strings in an IC Validator netlist.

```
{param_global str1="mode0"}
{cell CELL_X
{param_init str2="1.2v"}
{inst M1=NMOS {TYPE MOS}
{prop str3="-1"}
...}
{inst X1=MODEL_X {TYPE CELL}
{param str4="1"}
...
}
```

The double-quoted strings, "mode0", cell initial parameter, "1.2v", device property, "-1", and cell instance property, "1" are all defined as string parameters.

Duplicate Ports Definition

Duplicate ports have the same name in one cell. NetTran supports duplicate ports in the SPICE format. The default is that NetTran shorts the duplicate ports hierarchically with a warning message.

Note:

The IC Validator tool allows duplicate ports to be represented as a WARNING or ERROR in the SPICE cell port lists. To specify duplicate ports, set the `duplicate_port` argument of the `schematic()` and `read_layout_netlist()` functions, or use the `-sp-dupPort` NetTran command-line option.

For example,

```
.SUBCKT INVB GND VDD Z I Z
M1 Z I GND GND n L=0.7u W=13.5u
M2 Z I VDD VDD p L=0.8u W=20.5u
.ENDS

.SUBCKT MYCELL GND VDD
X1 GND VDD N1 N2 N3 INVB
.ENDS
```

There is a duplicate port name, Z, in cell INVB, and the nets N1 and N3 of the X1 instance cell are shorted together hierarchically. The following statement shows the equivalence netlist:

```
.SUBCKT INVB GND VDD Z I Z
M1 Z I GND GND n L=0.7u W=13.5u
M2 Z I VDD VDD p L=0.8u W=20.5u
.ENDS

.SUBCKT MYCELL GND VDD
X1 GND VDD N1 N2 N1 INVB
```



```
.ENDS
```

Identical Cells Definition

Identical cells are duplicate cells with the same content. NetTran always preserves one cell definition and discards the remaining cell definitions.

For example,

```
.SUBCKT INVB GND VDD Z I
M1 Z I GND GND n L=0.7u W=13.5u
M2 Z I VDD VDD p L=0.8u W=20.5u
.ENDS
```

```
.SUBCKT INVB GND VDD Z I
M1 ZI GND GND n L=0.7u W=13.5u
M2 Z I VDD VDD p L=0.8u W=20.5u
.ENDS
```

The following statement shows the equivalence netlist where only one cell is preserved:

```
.SUBCKT INVB GND VDD Z I
M1 Z I GND GND n L=0.7u W=13.5u
M2 Z I VDD VDD p L=0.8u W=20.5u
.ENDS
```

Duplicate Cells Definition

Duplicate cells have the same cell names, port counts, and port names.

The IC Validator tool provides some mechanisms for duplicate cell handling. To specify duplicate cell handling, set the `duplicate_cell` in `schematic()` and `read_layout_netlist()` functions, or use the `-dupCell` NetTran command-line option.

Cells that have the same cell name but different port counts or port names are not regarded as duplicate cells. However, NetTran does not allow this situation. Therefore, only one cell name is preserved; the other duplicate cells and cell names of their cell instances are automatically renamed and a WARNING message is issued.

Duplicate Cell Instance Definition

Duplicate cell instances have the same instance and reference cell name. The default behavior is to issue the following error:

```
GNFError: Duplicate instance "xxx" in cell "xxx"
```

The IC Validator tool provides mechanisms for handling duplicate cell instances. For example, you specify a duplicate cell instance by using the `resolve_duplicate_instances` argument of the `schematic()` and `read_layout_netlist()` functions or the `-sp-resolveDupInstances` command-line option.

For example,

```
.SUBCKT child  GND VDD QN A B
...
.ENDS
.SUBCKT top  GND VDD A1 B1 A2 B2
X1 A B C D E child
X1 C B D A E child
.ENDS
```

This example shows two instance cells, `x1`, which refer to the same cell, `child`, in cell definition, `top`. The IC Validator tool issues the following error message:

```
GNSError: Duplicate instance "X1" in cell "top".
```

When resolving duplicate instances, the IC Validator tool renames these instances by adding a suffix, as indicated by the following message:

```
Renaming duplicate instance from "X1" to "X1_DUP#1" in cell "top"
```

Control Statements

The IC Validator tool supports the following control statements in the SPICE netlist.

.GLOBAL

The `.GLOBAL` statement globally assigns a node name. All references to a global node name used in the circuit at any level of the hierarchy are connected to the same node.

The `.GLOBAL` statement is most often used when subcircuits are included in a netlist file. This statement assigns a common node name to internal, nonport subcircuit nodes. Power supply connections of all subcircuits are often assigned using a `.GLOBAL` statement, so that power supply nodes do not have to be specified in the port list of each subcircuit. For example, `.GLOBAL VCC` connects all subcircuits with the internal node name `VCC`.

The syntax is

```
.GLOBAL node ... node
```

where *node* specifies global nodes, such as supply and clock names, and overrides local subcircuit definitions.

Note:

The `*.GLOBAL` statement is equivalent to the `.GLOBAL` statement.

.CONNECT

The `.CONNECT` statement connects two or more nodes. NetTran uses the first node name to replace all other listed node names during netlist translation.

- If a `.CONNECT` statement is defined outside a subcircuit, the statement applies to the top cell of the netlist.
- If a `.CONNECT` statement is defined within a subcircuit, the statement applies only to that subcircuit. However, if a global net is connected using a `.CONNECT` statement inside a subcircuit, the `.CONNECT` statement is applied to the entire global net.
- When a `*.CONNECT` or `.CONNECT` statement is specified for global nets, the `*.CONNECT` or `.CONNECT` statement shorts (merges) global nets everywhere that global net names occur in the netlist hierarchy, regardless of the position of the `*.CONNECT` or `.CONNECT` statement.

The syntax is

```
.CONNECT net1 net2
```

Note:

The `*.CONNECT` statement is equivalent to the `.CONNECT` statement.

.INCLUDE

The syntax is

```
.INCLUDE filename
```

Argument	Description
<i>filename</i>	Specifies the file name. The name can be full a path or relative name. The file name can contain a dollar sign (\$) followed by a UNIX environment variable name. NetTran expands the text automatically.

For example,

```
.INCLUDE /home/janet/incdir/xyz.cdl
.INCLUDE $DIR/abc.cdl
```

If NetTran encounters multiple `.INCLUDE` instructions to the same file, it processes the first instruction and ignores the other instructions.

.PARAM

The `.PARAM` statement defines parameters or tokens that have associated numeric values. During netlist translation, NetTran replaces tokens with the corresponding numeric values defined by `.PARAM` statements. The following rules govern the interpretation of SPICE

`.PARAM` statements:

- If a redefinition of a `.PARAM` statement occurs, NetTran uses the first `.PARAM` statement and ignores subsequent definitions. This behavior applies regardless of whether the `.PARAM` statements all occur in a single file or occur across multiple files joined by `.INCLUDE` instructions.
- When NetTran is invoked with multiple SPICE netlists on the command line, any `.PARAM` statement within any netlist is interpreted only within the local scope of that netlist.
- If a `.PARAM` statement occurs within a subcircuit definition, NetTran interprets the `.PARAM` statement only within the context of that subcircuit definition.
- NetTran applies a `.PARAM` statement to tokens referenced either before or after the `.PARAM` statement.

The syntax is

```
.PARAM paramname = number
```

Parameter Priority of SPICE Netlists

NetTran processes parameters in the following order:

- Caller passing parameter

```
.PARAM w=3u
.subckt TOP
    x1 a b SUB w=1u
.ends TOP
.subckt SUB a b w=2u
    m1 a b c d W=w
.ends SUB
```

where "w" of m1 is 1u passed from the caller.

- Cell initial parameter

```
.PARAM w=3u
.subckt SUB a b w=2u
    m1 a b c d W=w
.ends SUB
```

where "w" of m1 is 2u from the subcircuit parameter.

- Global parameter

```
.PARAM w=3u
.subckt SUB a b
    m1 a b c d W=w
.ends SUB
```

where "w" of m1 is 3u from the global parameter.

*.BUSDELIMITER

The `*.BUSDELIMITER` statement specifies delimiter characters for bus pins inside a SPICE netlist. This syntax is used by NetTran when it must apply a bus defined inside a separate netlist format, such as Verilog, to underlying bus pins used inside a SPICE netlist. Valid delimiter characters understood by NetTran include a bracket ([), a curly brace ({}), a less-than sign (<), or an underscore (_). Only the leftmost character is specified within the `*.BUSDELIMITER` statement. If unspecified, the default bus delimiter character understood by NetTran is a bracket ([).

*.CAPA

The `*.CAPA` statement omits capacitors from the schematic netlist.

*.CONNECT

The `.CONNECT` statement connects two or more nodes together.

This statement has the same functionality as `.CONNECT`. See [.CONNECT](#) for more information.

*.DIODE

The `*.DIODE` statement omits diodes from the schematic netlist.

*.EQUIV

The `*.EQUIV` statement replaces an old node name or device model name with a new node name or device model name.

Use the `*.EQUIV` statement to short nets in the netlist. For example, a schematic netlist contains a standard cell with two ports, gnd! and gnda!, and a RAM cell with one port, vss. In a layout, gnd!, gnda!, and vss are connected to a global power net, vss. In the layout netlist during LVS, vss is properly connected to gnd!, gnda!, and the vss pin of the standard cell and RAM. To match layout nets to schematic nets, gnd!, gnda!, and vss have to be connected. You can connect these nets by adding the `*.EQUIV` statement in the schematic netlist and retranslating it using NetTran.

The syntax is

```
*.EQUIV new_name = old_name
```

For example,

```
*.EQUIV VSS=gnd! VSS=gnda! VSS=vss
```

*.RESI

The `*.RESI` statement specifies the threshold value (*tvalue*) or model name (*mname*) of the shorted resistors. If the resistance between any two nodes in the resistor statement is less than or equal to the threshold value, the two nodes are shorted. You can also short a resistor by specifying its model name (*modelName*).

The default threshold value is 2000 for `*.RESI` statements that have no listed threshold value or model name. The threshold value specified by a `*.RESI` statement applies to all of the resistor elements in the same netlist, and the value can be overwritten by another `*.RESI` statement (including `*.RESI` statement with the default of 2000).

The syntax is

```
*.RESI [[=] [tvalue]] [[mname1]] [[mname2]] ...
```

Argument	Description
<i>tvalue</i>	Optional. Represents the threshold value of the shorted resistors. The default threshold value is 2000.
[<i>mname</i>]	Optional. Represents the model name of the shorted resistors. Must be placed within square brackets []. Multiple model names are also supported within different square brackets [] as well as with a space between the model names.

For example,

- `*.RESI` \$shorts resistors with resistance value ≤ 2000
- `*.RESI = 10` \$shorts resistors with resistance value ≤ 10
- `*.RESI [res1] [res2]` \$shorts resistors with model name *res1*, or *res2*
- `*.RESI 10 [res1]` \$shorts resistors with resistance value ≤ 10 , or with model name *res1*

*.SCALE

The `*.SCALE` statement specifies the base unit of the netlist. The IC Validator tool supports the parsing of the input SPICE netlist and generates the SPICE layout netlists (cell.net) with this syntax. If this CDL syntax is not defined, the SPICE netlist format unit is meter.

The syntax is

```
*.SCALE meter | micron
```

Units such as nano and pico are not allowed. The IC Validator tool and NetTran report an error if these values are used.

The following is an example of a meter-based SPICE netlist:

```
*.SCALE meter

.SUBCKT invb GND VDD A Z
M1 GND A Z GND n L=1u W=13u
M2 VDD A Z VDD p L=1u W=20.5u
.ENDS
```

In this netlist, the properties of M1 are L=1 micron and W=13 micron. The syntax, `*.SCALE meter` is optional, as the SPICE format is meter-based.

Micron-based SPICE netlist 1:

```
*.SCALE micron

.SUBCKT invb GND VDD A Z
M1 GND A Z GND n L=1 W=13
M2 VDD A Z VDD p L=1 W=20.5
.ENDS
```

In this netlist, the properties of M1 are L=1 micron and W=13 micron.

Micron-based SPICE netlist 2:

```
*.SCALE micron

.SUBCKT invb GND VDD A Z
M1 GND A Z GND n L=1u W=13u
M2 VDD A Z VDD p L=1u W=20.5u
.ENDS
```

In this netlist, the properties of M1 are L=0.000001 micron and W=0.000013 micron.

When the `netlist()` function extracts cell.net in the SPICE flow with micron units, `*.SCALE micron` must be applied in the netlist.

Note:

Properties are generated by user functions. The IC Validator tool does not know the real units of individual properties. Therefore, additional translation of a generated SPICE netlist is not recommended.

.OPTION SCALE

The IC Validator tool supports both the `.OPTION SCALE` and `.OPTIONS SCALE` statements to scale geometric properties of instances in the netlist. One-dimensional geometric property values, such as length, width, and perimeter, are multiplied by the specified scale factor; two-dimensional property values, such as area, are multiplied by the square of the specified scale factor. Nonstandard properties can be reclassified as one-dimensional geometric, two-dimensional geometric, or nongeometric properties by using the `*.LENGTH_UNIT`, `*.AREA_UNIT`, and `*.NON_UNIT` statements.

Multiple `.OPTION SCALE` statements are allowed, but a later `.OPTION SCALE` overrides the previous one.

If the `.OPTION SCALE` statement is inside a `.SUBCKT`, it must be the first line or NetTran reports an error. Even inside a `.SUBCKT`, its scope is still global within the netlist.

If NetTran merges multiple netlists into a single netlist, each netlist uses its own definition of `.OPTION SCALE`.

If `.OPTION SCALE` is set to a scale factor, the value is resolved to a nonscale number, with a warning message from NetTran. For example, 1u is resolved to 1e-06.

The syntax is

```
.OPTION SCALE=scalefactor
```

Classifying Nonstandard Numeric SPICE Parameters

Parameter names are case-insensitive. Parameter names that are prefixed with W, L, or P are scaled with a length unit such as W, L, and P, respectively. Parameter names that are prefixed with A are scaled with an area unit of A. The `*.SCALE` and `.OPTION SCALE` statements are netlist unit transition statements. The IC Validator tool supports the `*.LENGTH_UNIT`, `*.AREA_UNIT`, and `*.NON_UNIT` nonstandard SPICE statements, which should be used only during NetTran conversion.

- `*.LENGTH_UNIT`:

In addition to devices parameter names prefixed with W, L, and P, the `*.LENGTH_UNIT` statement treats additional parameter names such as length scaling during NetTran unit conversion. The syntax is

```
*.LENGTH_UNIT parameter_1(device_type_1) ...  
                parameter_n(device_type_n)
```

- `*.AREA_UNIT`:

In addition to devices parameter names prefixed with A, the `*.AREA_UNIT` statement treats additional parameter names such as area scaling during NetTran unit conversion. The syntax is

```
*.AREA_UNIT parameter_1(device_type_1) ... parameter_n(device_type_n)
```


- `*.NON_UNIT:`

The `*.NON_UNIT` statement excludes parameter names prefixed with W, L, P, or A from NetTran unit conversion. The syntax is

```
*.NON_UNIT parameter_1(device_type_1) ... parameter_n(device_type_n)
```

The supported device types are: MOS, RES, CAP, IND, BJT, DIODE, and JFET.

Example 1

To apply `LENGTH_UNIT` to `S` only in MOSFET devices,

```
*.LENGTH_UNIT S(MOS)
```

Example 2

To exclude the `prop_1` parameter name from a MOSFET device,

```
*.NON_UNIT prop_1(MOS)
```

Use the property name without specifying a device type to apply the parameter to all device types. For example, to apply `LENGTH_UNIT` to `S` for all devices,

```
*.LENGTH_UNIT S
```

These statements can also be used to classify parameters on subcircuits by specifying the subcircuit name. For example, to apply `AREA_UNIT` to `S` for an instance with the subcircuit name `syfns_res`,

```
*.AREA_UNIT S(syfns_res)
```

Line Continuation

In a SPICE netlist format, the plus sign (+) is a line continuation character. Text on a line that starts with a plus sign goes with the last valid line above it. For example,

```
.SUBCKT mycell a b
+c
R1 a b res
.ENDS
```

translates to:

```
{Cell mycell
{ports a b c}
{inst...}
}
```

The line continuation character, however, is not applied to comments except in the header section. For example,

```
<file beginning>
```

```

* comment
+ I'm still a comment
$ comment
+ I'm still a comment, $ is the same as *

.SUBCKT mycell a b
* here is the comment
+c                               Note: c is not a comment
R1 a b res
.ENDS

```

Comments

There are two kinds of comments in a SPICE netlist format:

- Asterisk (*). The asterisk is always placed at the beginning of a line, following the comment strings.
- Dollar sign (\$). The dollar sign is used for comments that do not begin at the first character position in a line. The strings after the dollar sign in a line are comments.

Verilog Netlist Format

Cell Definition

The Verilog format uses the keyword module to define cells in a netlist. A module can be an element or a collection of lower-level design cells. Each module must have a cell name, which is the identifier for the module, and a port list, which describes the input and output terminals of the module.

The syntax is

```

module cell_name(port1, ..., portN);
    ...
endmodule

```

Port

The syntax for port declaration is

Format 1:

```

module cell_name list_of_port_name;
    list_of_port_declaration1
endmodule

```

```
list_of_port_name ::= (port_name{, port_name})|()
list_of_port_declaration1 ::= (port_declaration;{ port_declaration;})|()
port_declaration ::= Input/output/inout port_name{, port_name}
```

Format 2:

```
module cell_name list_of_port_name_declaration2;
endmodule

list_of_port_declaration2 ::= (port_declaration{, port_declaration})|()
```

For example,

Format 1:

```
module AND( A, B, C);
    input A;
    input B;
    output C;
    ...
endmodule
```

Format 2:

```
module OR2T( inout VSS,
             output X,
             input A, B,
             inout VDD);
    ...
endmodule
```

Net

An example of the syntax is

```
wire a;           // Declare net a for the circuit
wire b,c;         // Declare net b and c for the circuit
wire d = 1'b0;    // Net d is fixed to logic value 0 at declaration
```

where wire describes internal nets.

Instance

The syntax is

```
model_name instance_name (connections);
```

Connections can be made by order or by name. For example,

- Connecting by order:

```
AND I1 (net1, net2, net3);
```

- net1 connects to port #1 in the module definition of AND
- net2 connects to port #2 in the module definition of AND
- net3 connects to port #3 in the module definition of AND

- Connecting by name:

```
AND I1 (.B(net2), .C(net3), .A(net1));
```

- net1 connects to port name "A" in the module definition of AND
- net2 connects to port name 'B' in the module definition of AND
- net3 connects to port name 'C' in the module definition of AND

Assign Statements

NetTran shorts two nets if there is a Verilog assign statement. The syntax is

```
assign old_name = new_name;
```

For example,

```
assign net1 = net2;
```

NetTran shorts two nets, net1 and net2, and uses the name net2 to substitute for the name net1. If the shorted net is connected to a port, it retains the port name as the retained net and, in other cases, the net name.

Bus (Vector)

Nets can be declared as vectors, that is, multiple bit-widths. If the bit-width is not specified, the default is scalar (1 bit).

For example, to declare the vectors:

```
output [3:0] X // 4-bit X
input [0:2] Y
input [2:12] Z
```

Then, the vectors would be used as follows:

```
X[3], X[2], X[1], X[0]
Y[0], Y[1], Y[2]
Z[2], Z[3], ..., Z[12]
```

Defining Global Supply Nets

Global supply nets are VDD and VSS, by default. You can specify your own names for global supply nets using the NetTran `-verilog-b1` and `-verilog-b0` command-line options.

Defining Local Supply Nets

Local supply nets are defined in a Verilog module using the `supply1` and `supply0` net types, or with a mapping file from the NetTran `-verilog-voltmap` command-line option. The syntax of the mapping file is

```
cell-name inst-name B0 voltage-name B1 voltage-name
```

If the instance name is not specified, the assigned name mapping applies to the entire cell. If the instance name is assigned, the name mapping is performed only on that specific instance.

Here is an example of a Verilog voltage mapping file:

```
sub1 B1 VDD1 B0 GND1
sub3 B1 VDD2 B0 GND2
top_io top_io1 B0 GND1 B1 VDD1
top_io top_io2 B0 GND2 B1 VDD2
```

Supply Net Translation Precedence

The net names used to replace numeric values are chosen as follows (with precedence from high to low):

1. User-specified global supply nets (`-verilog-b1` and `-verilog-b0`)

In this case, NetTran generates corresponding global nets for these names, and all local supply nets are shorted with these nets.
2. Local supply nets (net type `supply1` and `supply0`, or `-verilog-voltmap`)

If more than one candidate is found, the net to be used is chosen arbitrarily. In this case, NetTran generates the corresponding global nets for all local supply nets.
3. Default global supply nets VDD and VSS

In this case, NetTran generates the corresponding global nets for them.

Available options for altering the preceding behavior:

- `-verilog-preferModuleSupply` changes the precedence to 2) > 1) > 3).

- `-verilog-localizeModuleSupply` changes the precedence to 2) > 1) > 3) and also disables the global net generation for local supply nets in 2).
- `-verilog-localizeGlobalSupply` disables the global net generation for global supply nets in 1) and 3).

You can disable all global net generation in this numeric value translation by specifying both the `-verilog-localizeModuleSupply` and `-verilog-localizeGlobalSupply` options.

Table 2-14 illustrates the 1'b1 translation under various combinations of conditions.

Table 2-14 1'b1 Translation

Specified options	Net name to replace 1'b1 in modules with supply1 net L_P	Net name to replace 1'b1 in modules with no supply1 net	Generated global nets
(NONE)	L_P	VDD	VDD L_P
<code>-verilog-preferModuleSupply</code>	L_P	VDD	VDD L_P
<code>-verilog-localizeGlobalSupply</code>	L_P	VDD	L_P
<code>-verilog-preferModuleSupply</code> <code>-verilog-localizeGlobalSupply</code>	L_P	VDD	L_P
<code>-verilog-localizeGlobalSupply</code>	L_P	VDD	VDD
<code>-verilog-localizeModuleSupply</code> <code>-verilog-localizeGlobalSupply</code>	L_P	VDD	(None)
<code>-verilog-b1 G_P</code>	G_P Nets L_P and G_P are shorted	G_P	G_P
<code>-verilog-b1 G_P</code> <code>-verilog-preferModuleSupply</code>	L_P	G_P	G_P L_P
<code>-verilog-b1 G_P</code> <code>-verilog-localizeModuleSupply</code>	L_P	G_P	G_P
<code>-verilog-b1 G_P</code> <code>-verilog-localizeGlobalSupply</code>	G_P Nets L_P and G_P are shorted	G_P	(None)

Table 2-14 1'b1 Translation (Continued)

Specified options	Net name to replace 1'b1 in modules with supply1 net L_P	Net name to replace 1'b1 in modules with no supply1 net	Generated global nets
-verilog-b1 G_P -verilog-preferModuleSupply -verilog-localizeGlobalSupply	L_P	G_P	L_P
-verilog-b1 G_P -verilog-localizeModuleSupply -verilog-localizeGlobalSupply	L_P	G_P	(None)

Note:

The translation of 1'b0 and other numeric values can be derived similarly.

The -verilog-busLSB Option

The NetTran -verilog-busLSB command-line option starts the Verilog bus with the least significant bit (LSB).

Note:

The -verilog-busLSB option takes effect only when an explicit pin-port binding is specified in the netlist file.

The following example shows how the -verilog-busLSB option changes the translated netlist. In a Verilog netlist:

```
test I1 (
    .byte_sel_n (byte_sel_n)
);
```

The net (wire) that is defined as a bus is

```
wire    [4:0]    byte_sel_n;
```

The net contains 5 lines:

```
byte_sel_n[4]
byte_sel_n[3]
byte_sel_n[2]
byte_sel_n[1]
byte_sel_n[0]
```

The port `byte_sel_n` from the cell test is not defined. The default is to treat the port as the net, from bus [4] to bus [0]. When the `-verilog-busLSB` option is set, NetTran treats the port from bus [0] to bus [4].

If the Verilog netlist is well defined, that is, it has no undefined cells except primitives, the `-verilog-busLSB` option does not need to be set. NetTran translates the netlist to the correct order.

For example, with the Verilog netlist shown in [Example 2-2](#), the IC Validator netlist is as shown in [Example 2-3](#), and the IC Validator netlist translated with the `-verilog-busLSB` option is as shown in [Example 2-4](#).

Example 2-2 Verilog Netlist

```
module top ();

    wire [1:0] s_top;
    wire [1:0] bus;

    mid X1 ( .i(bus), .s(s_top) );
endmodule

module mid ( .i({\i[0][0] , \i[1][1] }), s );

input [1:0] s;
input \i[0][0] , \i[1][1] ;

    nd02d4 X1 ( .a1(\i[0][0] ), .a2(\i[1][1] ), .zn(s[1]) );
    nd02d4 X2 ( .a1(\i[0][0] ), .a2(\i[1][1] ), .zn(s[0]) );

endmodule
```

Example 2-3 IC Validator Netlist

```
{netlist out
{version 2 1 0 }
/* ICV_Nettran: RELEASE OMITTED */
/* Created: DATA OMITTED */
/* Options: -verilog busLSB.v -outName out */

{net_global }
{cell mid
{port s[0] s[1] i[1][1] i[0][0]}
{inst X2=nd02d4 {TYPE CELL}
{pin i[0][0]=a1 i[1][1]=a2 s[0]=zn}}
{inst X1=nd02d4 {TYPE CELL}

{cell top
{inst X1=mid {TYPE CELL}
{pin bus[1]=i[0][0] bus[0]=i[1][1] s_top[1]=s[1] s_top[0]=s[0]}}
}

}
```


Example 2-4 IC Validator Netlist: Translated With the -verilog-busLSB Option

```

{netlist out
{version 2 1 0 }
/* ICV_Nettran: RELEASE OMITTED */
/* Created: DATA OMITTED */
/* Options: -verilog busLSB.v -outName out -verilog-busLSB */

{net_global }
{cell mid
{port s[0] s[1] i[1][1] i[0][0]}
{inst X2=nd02d4 {TYPE CELL}
{pin i[0][0]=a1 i[1][1]=a2 s[0]=zn}}
{inst X1=nd02d4 {TYPE CELL}
{pin i[0][0]=a1 i[1][1]=a2 s[1]=zn}}
}

{cell top
{inst X1=mid {TYPE CELL}
{pin bus[0]=i[0][0] bus[1]=i[1][1] s_top[0]=s[1] s_top[1]=s[0]}}
}

}

```

Verilog Compiler Directives

All Verilog compiler directives are preceded by the (`) character. This character is called grave accent (ASCII 0x60). It is different from the character, ('), which is the apostrophe character (ASCII 0x27).

NetTran supports the following directive:

```
`include
```

The file inclusion (`include) is used to insert the contents of the entire file into the current file during compilation.

NetTran parses and ignores the following directives:

```

`accelerate
`celldefine
`define
`endcelldefine
`endprotect
`protect
`remove_gatenames
`timescale

```

Directives that are not listed cause a NetTran error.

```
`include
```

The syntax is

```
`include "file_name"
```

The file name can be a relative or absolute path. Inclusions are ignored.

For example:

```
`include "sources/FileA.v"  
`include "FileB.v"
```

3

Building the Substrate

This chapter describes the different methods the IC Validator tool uses to create substrate layers for different processes.

This chapter contains the following sections:

- [Overview](#)
- [Guidelines for Building the Substrate](#)

Overview

To build full-chip connectivity for ERC, DRC, and device extraction runsets, you need the correct substrate definition. A less than optimal substrate definition might cause device extraction errors, extra ports in the extracted netlist, and hierarchically complex layers that can degrade performance.

The IC Validator tool uses different methods to create substrate layers depending on the requirements of the process. The following runset methods are used to generate substrates:

`cell_extent` : `not`, `buildsub`, and `get_substrate()`.

Note:

The `get_substrate` method is a legacy method suitable only for processes that do not have multiple potential regions. This method is not covered in this user guide.

Choosing a substrate definition method requires balancing accuracy and performance.

[Example 3-1](#) shows the `cell_extent` : `not` method. [Example 3-2](#) shows the `buildsub` method.

Example 3-1 `cell_extent` : `not` Method

```
sub1 = cell_extent(  
    cell_list = {"*"}  
);  
psub = sub1 not NWELL
```

Example 3-2 `buildsub` Method

```
psub=buildsub(NWELL);
```

When accuracy is a concern and the process demands a generated substrate layer from only input polygons that form isolated substrate regions, the `buildsub` method must be used. The `cell_extent` : `not` method does not check for input polygons that form isolated regions, and it uses all of the input polygons to generate the substrate layer. The following figures illustrate the differences between the `cell_extent` : `not` and `buildsub` methods.

[Figure 3-1](#) shows four green input `NWELL` polygons to the `cell_extent` : `not` and `buildsub` methods.

Note:

The top left `NWELL` ring polygon and bottom left `NWELL` polygon are the only polygons that form isolated substrate regions.

Figure 3-1 Input NWELL Polygons

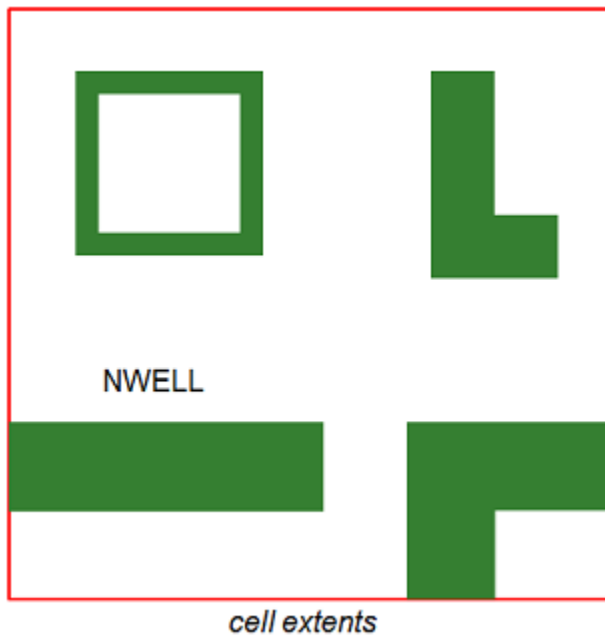


Figure 3-2 shows the gray output layer from the `cell_extent : not` method. This method uses all of the `NWELL` shapes to create the final `psub` substrate output layer. Therefore, all of the `NWELL` polygons are subtracted from the output layer.

Figure 3-2 Gray Output Layer From `cell_extent : not` Method

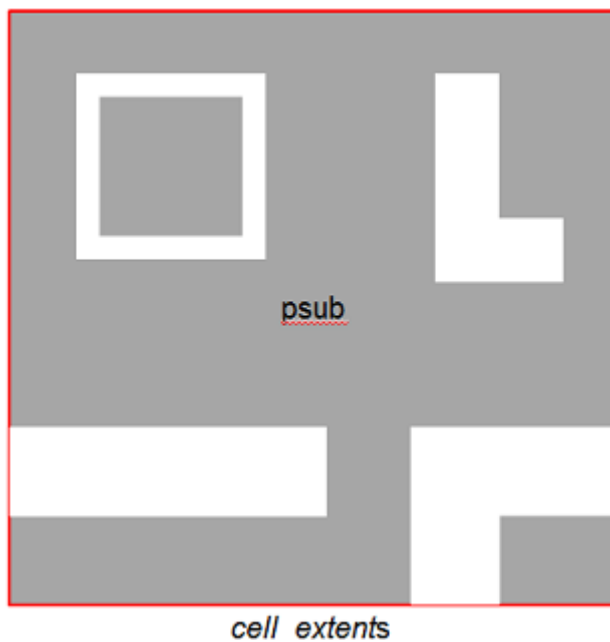
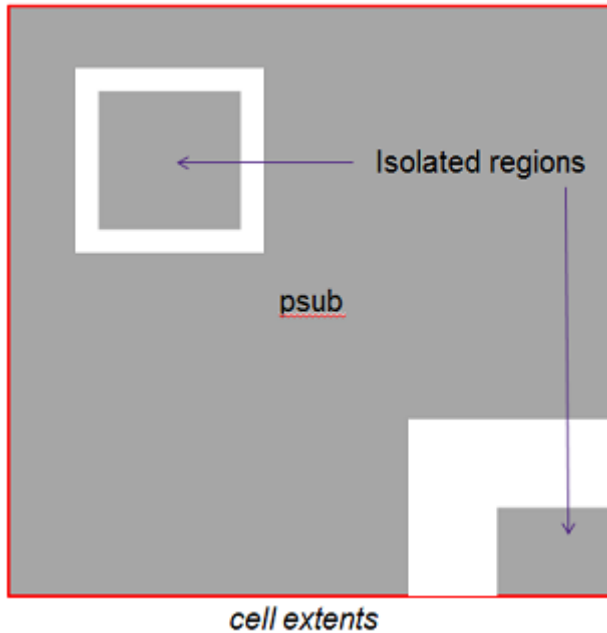


Figure 3-3 shows the gray output from the `buildsub` method. This method uses only input `NWELL` polygons that form isolated substrate regions. Therefore, only the `NWELL` polygon that formed a ring and the `NWELL` that divided the chip were subtracted from the output layer.

Figure 3-3 Gray Output Layer From `buildsub` Method



Therefore, use the `buildsub` method if the process requires a generated bulk layer from the input well polygons that form the isolated regions.

Use the `cell_extent : not` method if well polygons that form isolated regions are not a concern and you need all well polygons to be subtracted from the generated substrate layer.

Select “don’t care” if all well polygons in the layout form isolated regions.

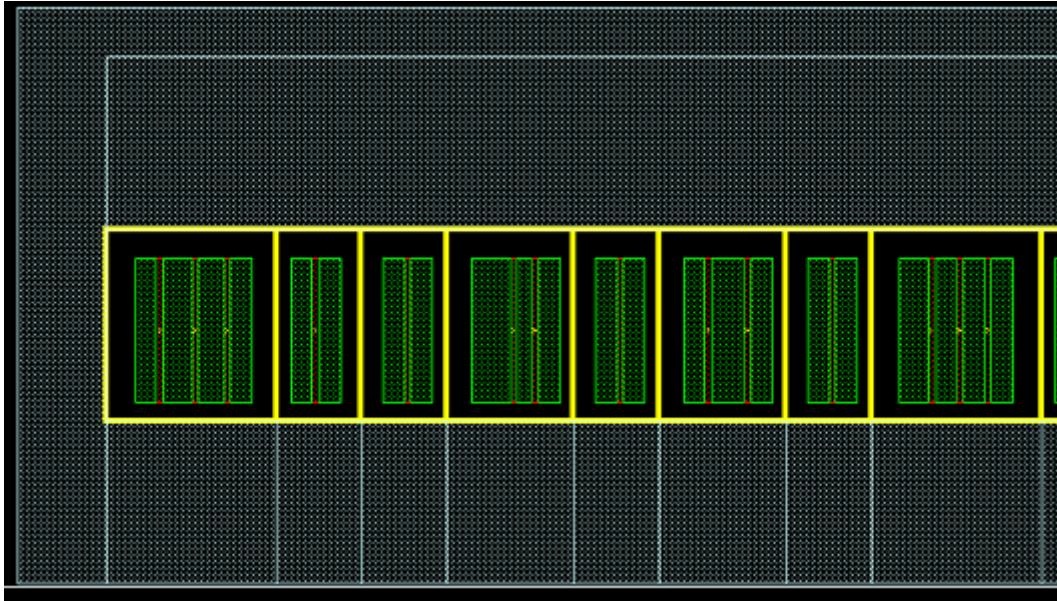
Example 3-3 shows a PMOS device with two bulk layers:

Example 3-3 PMOS Device

```
pmos(my_devices, "p", psd, pgate, psd, {{device_layer=NWELL,
pin_name="BULK1"}, {device_layer=psub, pin_name="BULK2"}})
```

Figure 3-4 shows the yellow `NWELL` layer is subtracted from the gray generated `psub` layer using the `cell_extent : not` method. If devices with two bulk layers are extracted, the `cell_extent : not` method causes device extraction errors.

Figure 3-4 Devices With Two Bulk Layers



When the IC Validator tool executes device extraction, the `pmos()` function shows the missing bulk connection errors, as shown in [Example 3-4](#). The rectangular `NWELL` shapes are subtracted incorrectly from the substrate derivation layer. With this case, the `buildsub` method is the better choice, as this `NWELL` structure is not subtracted from the generated substrate layer.

Example 3-4 Missing Bulk Connection

```
runset.rs:108:pmos[p]:device extraction errors
-----
Structure  Pin    Error Type          (position x, y)
-----
invb       BULK2 MISSING_TERMINALS (5.5000, 35.7500)
nor2b      BULK2 MISSING_TERMINALS (5.5000, 35.7500)
nor2b      BULK2 MISSING_TERMINALS (8.5000, 35.7500)
nor3b      BULK2 MISSING_TERMINALS (11.5000, 35.7500)
nor3b      BULK2 MISSING_TERMINALS (7.5000, 35.7500)
nor3b      BULK2 MISSING_TERMINALS (15.5000, 35.7500)
nand2b     BULK2 MISSING_TERMINALS (5.5000, 35.7500)
```

Guidelines for Building the Substrate

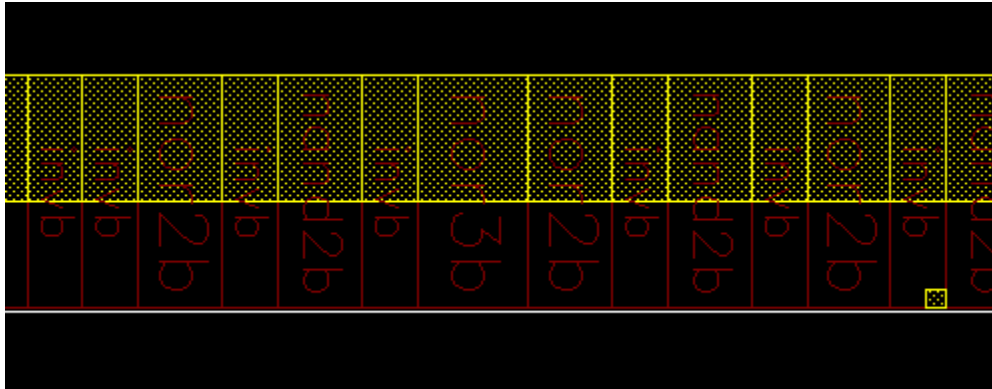
When you use the `buildsub` and `cell_extent : not` methods, be aware of the following guidelines and warnings:

- When using the `cell_extent : not` method, a `no_explode` list containing all of the cells in the hierarchy for the `cell_extent(cell_list={"*"})` function might cause

performance problems. The `cell_extent()` function might complicate the hierarchy with polygons in too many cells. These unnecessary polygons could adversely affect performance of data creation commands such as Boolean NOT.

- The `cell-extent : not` method might cause netlist quality issues for layouts having instance-specific NWELL structures. [Figure 3-5](#) shows the AD4FUL design with an instance-specific NWELL structure in the lower-right corner.

Figure 3-5 Instance-Specific NWELL Structure



In [Figure 3-5](#), the `cell_extent : not` method creates untexted ports to upper-level cells in the netlist with port 2.

Example 3-5 Untexted Ports to Upper-Level Cells

```
{CELL invb
{PORT 2 GND VDD Z A}
{PROP top=50.000000 bottom=0.000000 left=0.000000 right=12.000000}
{INST M1=n {TYPE MOS} {COORD x=5.5000 y=10.5000}
{PROP l=1.0000 w=13.0000}
{PIN Z=DRN A=GATE GND=SRC 2=BULK}}
{INST M2=p {TYPE MOS} {COORD x=5.5000 y=35.7500}
{PROP l=1.0000 w=20.5000}
{PIN Z=DRN A=GATE VDD=SRC VDD=BULK}}
}
```

The `buildsub` method is advantageous in this case because it does not subtract NWELL from the output that does not form isolated regions. As a result, the instance-specific NWELL structures do not create the extra ports in the netlist.

- Performance issues do exist when using the `buildsub` method. If a design contains many instance-specific NWELL rings in the designs, the `buildsub` method performs an aggressive pull-down to all cells, which complicates the hierarchy and causes runtime problems for downstream connections and device extraction functions.

If certain functions have performance issues and the pulled polygon count is high in the summary file, as shown in [Example 3-6](#), use the `cell_extent : not` method or use the manual `buildsub` method, as shown in [Example 3-7](#).

Example 3-6 *buildsub Method*

```

psub = buildsub(NWELL)
Function: buildsub
Inputs: NWELL.polygonlayer.0003
Outputs: psub.polygonlayer.0001
447 unique polygons written.
Pulled 303 polygons

```

The manual `buildsub` method, as shown in [Example 3-7](#) is logically equivalent to the `buildsub` method. However, this method uses the `pull_down_to()` and `copy_by_layout_equiv_cells()` functions to pull the output polygons to only the equivalence cells. This is a much less intrusive pull-down step, which outputs a cleaner, more efficient polygon than the polygon generated by the published `buildsub` method.

Example 3-7 *Manual buildsub Method*

```

manual_buildsub : function (
layer1 : polygon_layer,
oversize_value: double = 0.0
) returning isolated_regions : polygon_layer {
  all_cells_extent = cell_extent( cell_list = {"*"} );
  equiv_cell_extent = copy_by_layout_equiv_cells(all_cells_extent);
  all_cell_extent_pgon = copy(equiv_cell_extent);
  CHIP_LAYER=size( all_cell_extent_pgon, oversize_value);
  sized_chip_layer=size(CHIP_LAYER, 0.01);
  chip_ring=sized_chip_layer not CHIP_LAYER;
  layer1_touch_chip_edge = layer1 interacting chip_ring;
  new_chip_extent = ( CHIP_LAYER not layer1_touch_chip_edge);
  layer1_donuts = donuts(layer1);
  negate_layer1_layer =not (CHIP_LAYER, layer1_donuts);
  isolated_regions = new_chip_extent not ( new_chip_extent not
negate_layer1_layer);
  isolated_regions = pull_down_to( isolated_regions,
equiv_cell_extent, duplicate=ALL );
};

psub = manual_buildsub(NWELL);

```


4

Hierarchical Device Extraction

This chapter describes the device functions and arguments, which include device names, terminals, and extraction information.

This chapter has the following sections:

- [Hierarchical Device Extraction](#)
- [Bipolar Junction Transistors](#)
- [Capacitors](#)
- [Diode Extraction](#)
- [Inductor Extraction](#)
- [MOSFETs](#)
- [Resistor Extraction](#)

Hierarchical Device Extraction

IC Validator uses device functions, such as `nmos()` and `pmos()`, to define devices that are extracted from the layout and written to the layout netlist. These device functions have several arguments that consist of a device name, terminals, and extraction information. Their results are stored in a matrix of devices that are extracted using the `extract_devices()` function for LVS and the extraction flows. For detailed information about the device functions, see the Runset Functions chapters in the *IC Validator Reference Manual* for more information.

Layers for Device Extraction

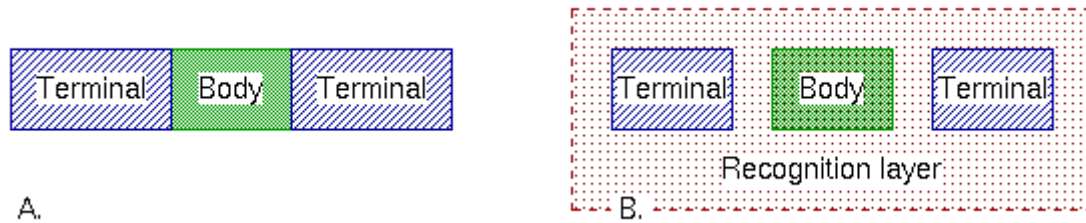
The IC Validator tool has five primary types of device layers:

- **Body layer.** Specifies the seed around which the device is formed.
- **Terminal layer.** Specifies the polygons that represent electrical connections to the devices.
- **Recognition layer.** Optional layer that specifies the device boundary. This layer is required when terminal layers do not touch or interact with the body layer. Use a recognition layer to indicate multifinger devices. [Figure 4-1](#) shows an example of a recognition layer.
- **Processing layer.** Specifies the polygons that are required to compute properties (optional). Use this layer to calculate device properties. [Figure 4-2](#) shows how the processing layer interacts with the device.
- **Reference layer.** Optional layer that specifies the intended location in the hierarchy for the extracted device.

The IC Validator tool extracts one device for each body layer that interacts with all of the other required polygons.

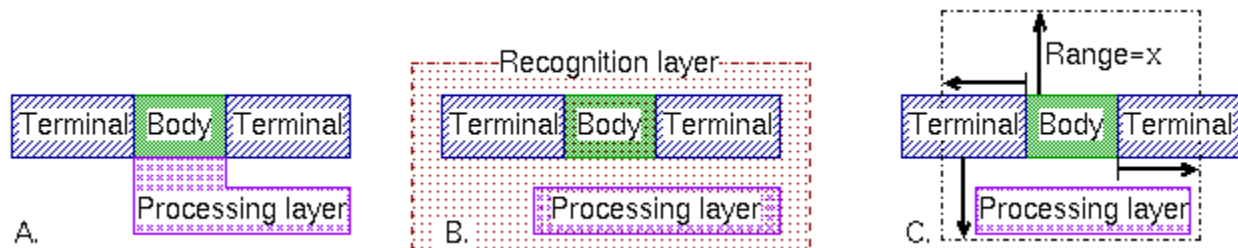
Devices are defined with a body layer, terminal layers, and a recognition layer in [Figure 4-1](#). All terminals touch the body layer (A) and use a recognition layer when terminals do not touch the body layer (B).

Figure 4-1 Defining Devices With a Body Layer, Terminal Layers, and a Recognition Layer



The processing layer interaction with the device: (A) body layer, (B) recognition layer, and (C) within the specified range of the body is shown in Figure 4-2.

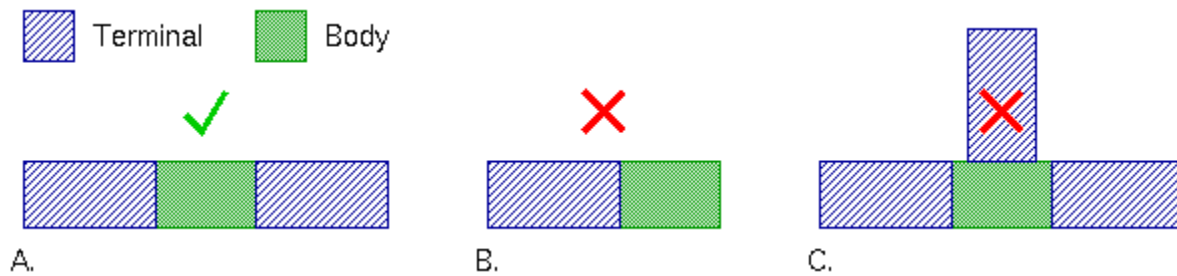
Figure 4-2 Processing Layer Interaction With the Device



Device Terminals

When you specify a device, the IC Validator extraction functions require a specific number of terminals. For example, resistors, capacitors, or inductors must have exactly two terminals. Otherwise, the IC Validator tool reports an extraction error for either too many terminals or too few terminals (see Figure 4-3). If a bipolar junction transistor (BJT) or diode is missing a terminal, the device might not be recognized. If the BJT or diode has an extra terminal, the IC Validator tool might extract additional devices. For a MOSFET, you can require either one or two source/drain terminals using the `source_drain_config` argument.

Examples of resistors with different numbers of terminals are shown in Figure 4-3. When two terminals connect to the body (A) the device is extracted. If there are too few (B) or too many (C) terminals, the IC Validator tool does not extract the device.

Figure 4-3 Resistors With Different Numbers of Terminals

Device Extraction and Hierarchy

The IC Validator tool performs hierarchical device extraction but does not require that all of the polygon data or device layers be in a single cell. For example, the poly and diffusion layers might be in different cells, but the IC Validator tool still recognizes their interaction and forms the gate, source, and drain layers for a MOSFET device. It is recommended, however, that all of the terminal layers are in the same cell, to optimize performance from the tool and simplify debugging.

Bipolar Junction Transistors

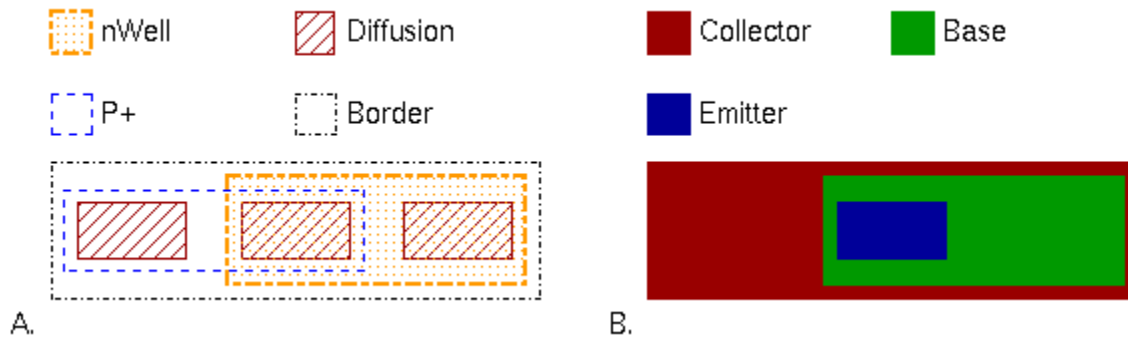
A bipolar junction transistor (BJT) consists of an emitter, base, collector, and one or more optional bulk terminals, and is created with either the `nnp()` or `pnnp()` functions. All of the terminals must be on different layers, for example, the emitter and collector. Specify the body using the `body_layer` argument.

See the following examples of vertical and lateral BJT layouts and the device extraction code.

Vertical BJT Extraction

[Figure 4-4](#) shows a vertical PNP BJT layout and its generated layers.

Figure 4-4 Vertical BJT (A. Layout, B. Generated Layers)



Example 4-1 shows how to generate the layers and the `pnp()` function.

Example 4-1 Vertical BJT

```

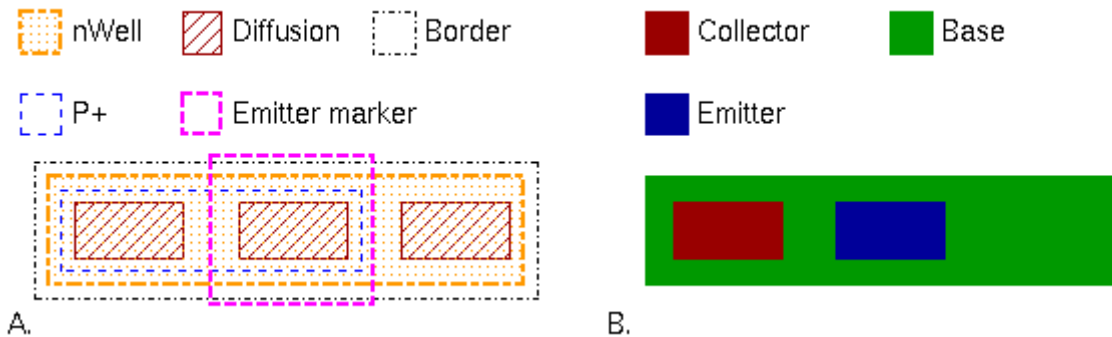
pnp_collector = BJT_BORDER;
pdiff = DIFFUSION and PPLUS;
pdev = pdiff and NWELL;
psd = pdev not POLY;
pnp_emitter = psd and BJT_BORDER;
pnp_base = NWELL and BJT_BORDER;
...
input_cdb = incremental_connect(
    connect_sequence = input_cdb,
    connect_items = {{pnp_collector}, psub},
                  {{pnp_base}, welltie},
                  {{pnp_emitter}, psd}}
);
...
pnp(
    matrix = my_devices,
    device_name = "PNP_VERTICAL",
    collector = pnp_collector,
    base = pnp_base,
    emitter = pnp_emitter,
    body_position = COLLECTOR,
    recognition_layer=BJT_BORDER
);

```

Lateral BJT Extraction

Figure 4-5 shows a lateral PNP BJT layout and its generated layers.

Figure 4-5 Lateral BJT (A. Layout, B. Generated Layers)



Example 4-2 shows how to generate the layers and the `pnp()` function.

Example 4-2 Lateral BJT

```

pdiff = DIFFUSION and PPLUS;
pdev = pdiff and NWELL;
psd = pdev not POLY;
pnp_collector = psd not EMITTER_MK;
pnp_emitter = psd and EMITTER_MK;
pnp_base = interacting(NWELL, p_emitter);
...
input_cdb = incremental_connect(
    connect_sequence = input_cdb,
    connect_items = {{pnp_collector}, psd},
                  {{pnp_base}, welltie},
                  {{pnp_emitter}, psd}}
);
...
pnpp(
    matrix = my_devices,
    device_name = "PNP_LATERAL",
    collector = pnp_collector,
    base = pnp_base,
    emitter = pnp_emitter,
    // Only the base touches all layers, so it is the body
    body_position = BASE
);

```

Capacitors

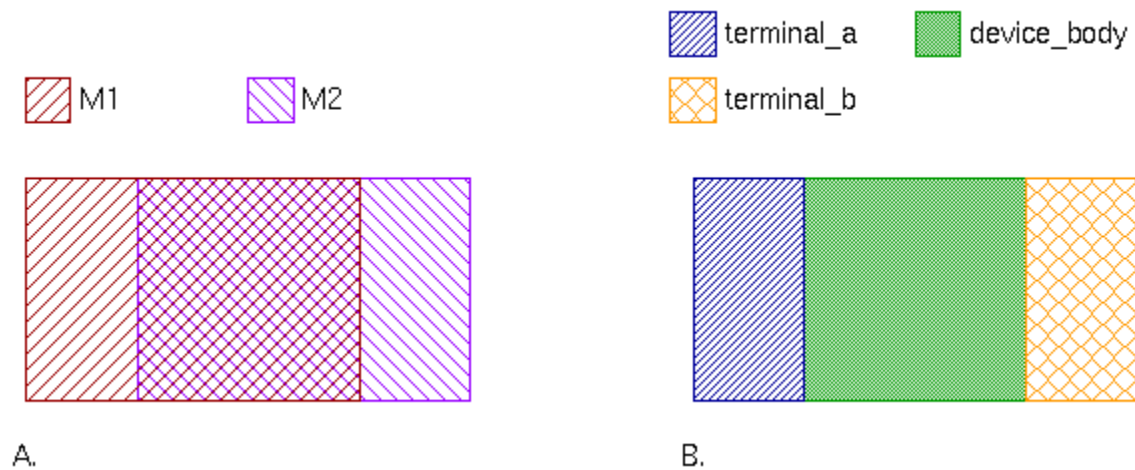
Capacitor devices consist of two overlapping layers, where the overlapping region is the device area.

Metal-to-Metal Capacitor

The metal-1 to metal-2 capacitor (A) and the generated layers for extraction (B) are shown in [Figure 4-6](#). You generate the device_body layer by performing a Boolean AND of the metal-1 and metal-2 layers.

You generate the terminal layers by performing a Boolean NOT of the device_body layer and the metal-1 and metal-2 layers.

Figure 4-6 Metal-to-Metal Capacitor (A. Layout, B. Generated Layers)



[Example 4-3](#) shows the metal-to-metal capacitor.

Example 4-3 Metal-to-Metal Capacitor

```

cap_body = M1 and M2;
m1_connect = interacting(M1, cap_body);
m1_terminal = m1_connect not cap_body;
m1 = M1 not m1_terminal;
m2_connect = interacting(M2, cap_body);
m2_terminal = m2_connect not cap_body;
m2 = M2 not m2_terminal;
...
input_cdb = incremental_connect(
    connect_sequence = input_cdb,
    connect_items = {{{m1_terminal}, m1},
                     {{m2_terminal}, m2}
    }
);
...
cap(
    matrix = my_devices,
    device_name = "MOM_M1_M2",
    device_body = cap_body,
    terminal_a = m1_terminal,
    terminal_b = m2_terminal,
);

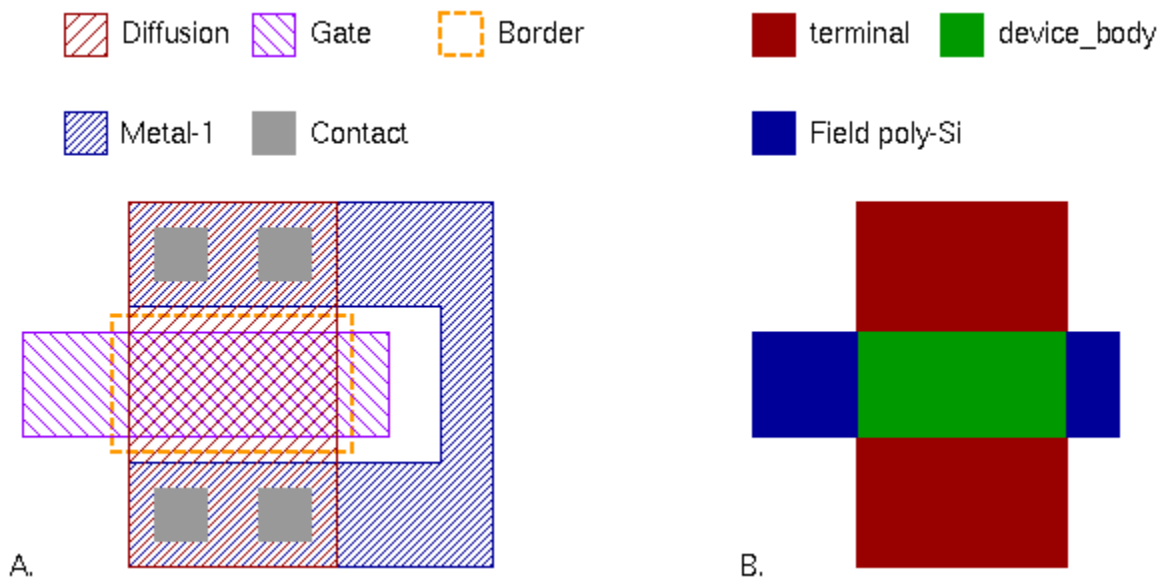
```

MOS Capacitors

You create decoupling capacitors by using a MOSFET with the source and drain electrically connected to the bulk, and the gate electrically connected to VCC. Even though the device is a MOSFET, circuit simulation is faster without a loss in accuracy by treating these devices as capacitors.

A layout and its generated layers are shown in [Figure 4-7](#).

Figure 4-7 MOS Capacitor (A. Layout - Source and Drain Are Connected, B. Generated Layers)



Example 4-4 shows the MOS capacitor.

Example 4-4 MOS Capacitor

```
ndiff = DIFFUSION not PPLUS;
ndev = ndiff not NWELL;
ngate = POLY and ndev;
field_poly = POLY not DIFFUSION;
ngate_cap = ngate and MOSCAP;
nsd = ndev not ngate;

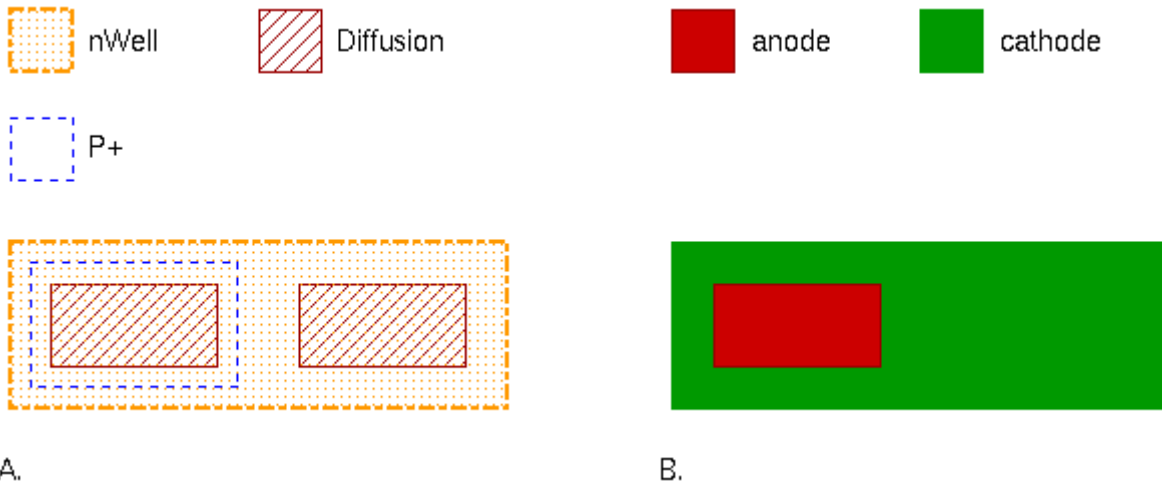
...
input_cdb = incremental_connect(
    connect_sequence = input_cdb,
    connect_items = {{{ngate_cap}, field_poly}}
);

...
cap(
    matrix = my_devices,
    device_name = "NMOS_CAP",
    device_body = ngate_cap,
    terminal_a = nsd,
    terminal_b = field_poly,
    optional_pins = {{psub}}
);
```

Diode Extraction

The diode device consists of two overlapping layers. The NP and PN diodes are extracted using the `np()` and `pn()` functions, respectively. You specify the body using the `body_layer` argument. An example of a `pn()` diode is shown in [Figure 4-8](#).

Figure 4-8 Diode Drawn in Layout (A) and Generated Layers for the `pn()` Function (B)



[Example 4-5](#) shows how to generate the layers and specify the device in the `pn()` function.

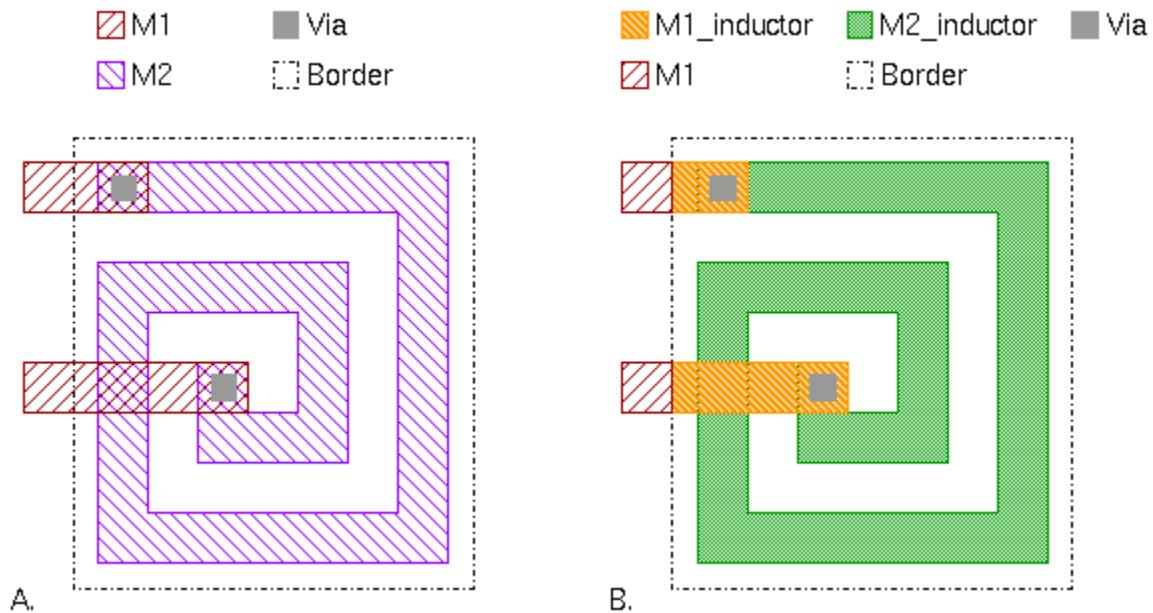
Example 4-5 Diode Extraction

```
ndiff = DIFFUSION not PPLUS;
pdiff = DIFFUSION not ndiff;
pdev = pdiff and NWELL;
anode = pdev not POLY;
cathode = NWELL;
...
input_cdb = incremental_connect(
    connect_sequence = input_cdb,
    connect_items = {{anode}, psd},
                  {{cathode }, welltie}
);
...
pn(
    matrix = my_devices,
    device_name = "PN_DIODE",
    device_body = cathode,
    anode = anode,
    cathode = cathode,
    optional_pins = {{psub}}
);
```

Inductor Extraction

The `inductor()` function extracts inductors that have a device layer and two terminal layers. You generate the inductor layers from a Boolean AND between a border layer and the metal layers, as shown in [Figure 4-9](#).

Figure 4-9 Inductor Drawn in Layout (A) and Generated Layers for inductor() Function (B)



[Example 4-6](#) shows how to create an inductor.

Example 4-6 Inductor Extraction

```

m1_induct = metall and INDUCTOR_MK;
m2_induct = metal2 and INDUCTOR_MK;
metall = metall not m1_induct;
metal2 = metal2 not m2_induct;

...
input_cdb = incremental_connect(
    connect_sequence = input_cdb,
    connect_items = {{m1_induct}, m1}
);

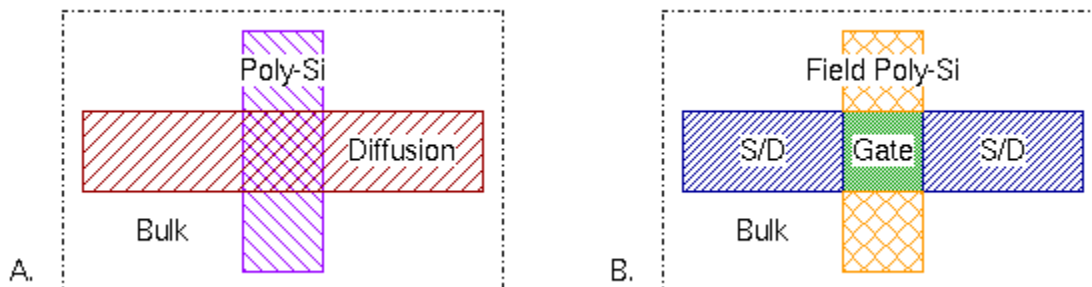
...
inductor(
    matrix = my_devices,
    device_name = "induct",
    device_body = INDUCTOR_MK,
    terminal_a = m1_induct,
    terminal_b = m1_induct,
    processing_layer_hash = { "VIA1" => {VIA1, 0.0},
        "m2_induct=> {m2_induct, 0.0}},
    connectivity = {{ {m1_induct, m2_induct}, VIA1}}
);

```

MOSFETs

A MOSFET consists of a gate polygon, two source/drain polygons, and optionally, one or more bulk terminals. You generate the gate from a Boolean AND between a polysilicon and diffusion layer, as shown in [Figure 4-10](#) and the source and drain polygons from a Boolean NOT. The gate layer must be connected by using the `connect()` or `incremental_connect()` function. You create different types of MOSFETs through Boolean operations between the gate and other layers, such as implants, nWell, and thick oxide.

Figure 4-10 MOSFET Drawn in Layout and Generated Layers for `nmos()` and `pmos()` Functions



For optimal IC Validator extraction:

- The gate polygon should exactly edge-touch each of the source/drain polygons.
- Enclose the gate polygon and the source/drain polygons by the bulk polygon, if specified.
[Example 4-7](#) shows how to create a MOSFET.

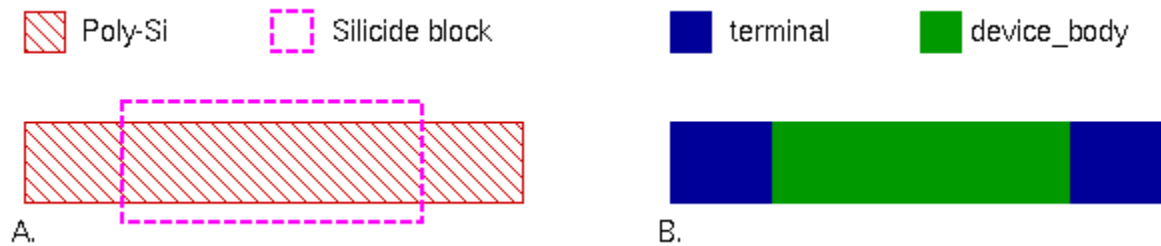
Example 4-7 MOSFET Extraction

```
ndiff = DIFFUSION not PPLUS;
pdiff = DIFFUSION not ndiff;
ndev = ndiff not NWELL;
pdev = pdiff and NWELL;
ngate = POLY and ndev;
pgate = POLY and pdev;
nsd = ndev not ngate;
psd = pdev not pgate;
sub_extent = cell_extent(cell_list = "*");
psub = sub_extent not NWELL;
...
nmos(
    matrix = my_devices,
    device_name = "NMOS",
    drain = nsd,
    gate = ngate,
    source = nsd,
    optional_pins = {{psub}}
);
pmos(
    matrix = my_devices,
    device_name = "PMOS",
    drain = psd,
    gate = pgate,
    source = psd,
    optional_pins = {{NWELL}}
);
```

Resistor Extraction

A resistor device consists of a resistive body polygon, usually path-like in shape, which has terminals at each end. Figure A of [Figure 4-11](#) shows an example a polysilicon resistor formed by a silicide block layer with silicided polysilicon terminals and the derived layers for the `resistor()` function. You create the body polygon by using a Boolean AND operation and the terminals by using a Boolean NOT operation. You specify an optional resistance value per square and an additional bulk terminal, which encloses the device.

Figure 4-11 Polysilicon Resistor Layout (A) and Generated Layers for Device Extraction (B)



Example 4-8 shows a polysilicon resistor formed by a silicide block layer with silicided polysilicon terminals.

Example 4-8 Resistor Extraction

```
poly_resistor = POLY and SILICIDE_BLK;
poly_field = POLY not SILICIDE_BLK;
...
input_cdb = incremental_connect(
    connect_sequence = input_cdb,
    connect_items = {{{poly_field}, contact}
}
);
...
resistor(
    matrix = my_devices,
    device_name = "poly_resistor",
    device_body = poly_resistor,
    terminal_a = poly_field,
    terminal_b = poly_field,
    optional_pins = {{psub}},
    resistor_value = 20
);
```


5

Device Property Calculations

This chapter describes device properties that can be attached to the primitive devices in a netlist. These properties can be used for simulation and LVS comparison between the schematic and layout netlists.

This chapter has the following section:

- [Property Calculations](#)

Property Calculations

Properties can be attached to the primitive devices in a netlist, and they can be used for simulation and LVS comparison between the schematic and layout netlists. Layout netlist device properties are derived from the polygons that are selected for each device instance during device extraction. These device properties can typically be divided into two categories based on their hierarchical characteristics and how they are used.

Compare Properties

Compare properties are found in both the schematic and layout netlists and are compared with the expectation that they should be the same within some tolerance to successfully pass LVS. These properties are also typically able to be extracted without flattening the hierarchy, therefore making it possible for the hierarchy to be preserved for LVS.

Simulation Properties

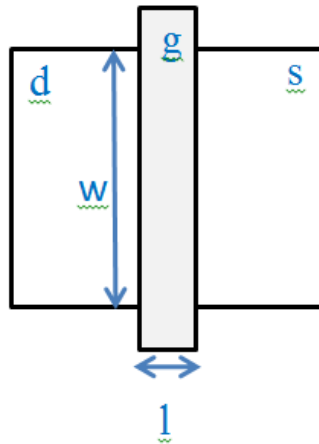
Simulation properties are needed for simulation, but are not found in the schematic netlist and are not compared during LVS. These properties often require a larger ambit around the device for accurate extraction and, therefore, might require some additional flattening to be extracted, which results in a loss of netlist hierarchy.

By default, the compare and simulation properties are attached to devices in the netlist that are used during LVS compare. The simulation properties might require flattening of the netlist hierarchy, which can make LVS more difficult to debug. For this reason, use the dual hierarchy flow to separate the simulation properties into a separate file called the annotation file. This file allows the simulation properties to be passed downstream to the StarRC tool to be combined in the flat netlist for simulation, while also maintaining the hierarchy in the netlist used for LVS comparison.

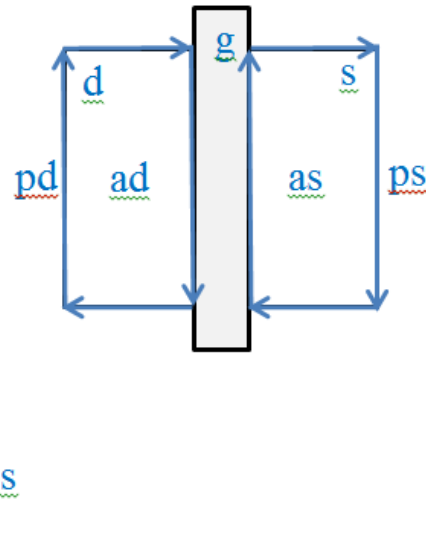
An example of MOS properties is shown in [Figure 5-1](#).

Figure 5-1 MOS Properties

Compare Properties



Simulation Properties



Compare properties:

```
M1 d g s b n l=1e-06 w=1.3e-05
```

The Length (l) and Width (w) properties for this MOS device are compared between the Schematic and Layout netlists.

Simulation Properties:

```
M1 d g s b n l=1e-06 w=1.3e-05
+ as=1.17e-10 ad=3.9e-11 ps=1.8e-05 pd=1.9e-05
```

The Source Area (as), Drain Area (ad), Source Perimeter (ps), and, Drain Perimeter (pd) properties for this MOS device are used for simulation only and might require device leveling for accuracy.

Default Properties

Each device extraction function has default properties and default property calculation functions, as shown in [Table 5-1](#). The default property and property calculation function definitions are in the function prototypes in \$ICV_HOME_DIR/include/device_public.rh.

Table 5-1 Device Extraction Function Default Properties

Device extraction function	Default property calculation function	Default properties	Description
<code>nmos()</code>	<code>calc_capacitor_properties()</code>	<code>w, l</code>	Gate width, gate length.
<code>pmos()</code>	<code>calc_capacitor_properties()</code>	<code>w, l</code>	gate width, gate length.
<code>np()</code>	<code>calc_diode_properties()</code>	<code>area, pj</code>	Junction area, junction perimeter.
<code>pn()</code>	<code>calc_diode_properties()</code>	<code>area, pj</code>	Junction area, junction perimeter.
<code>npn()</code>	<code>calc_bjt_properties()</code>	<code>area</code>	Area of emitter, base, or collector as defined by the body position argument.
<code>pnp()</code>	<code>calc_bjt_properties()</code>	<code>area</code>	Area of emitter, base, or collector as defined by the body position argument.
<code>capacitor()</code>	<code>calc_capacitor_properties()</code>	<code>c</code>	Capacitance. The capacitance property requires at least one additional capacitance coefficient argument; otherwise the capacitance is calculated as zero.
<code>resistor()</code>	<code>calc_resistor_properties()</code>	<code>l, w, r</code>	Body length, body width, and Resistance. The resistance property requires an additional sheet resistance argument; otherwise the resistance is calculated as zero.
<code>inductor()</code>	<code>calc_inductor_properties()</code>	<code>l</code>	Body length.

Table 5-1 Device Extraction Function Default Properties (Continued)

Device extraction function	Default property calculation function	Default properties	Description
gendev()	calc_gendev_properties()	none	Generic devices have no default properties.

Device Extraction and Property Calculation Functions

For each of the device extraction functions, the properties that are output for a device are defined by the `properties` argument. The function used to calculate the properties is defined by the `property_function` argument. The defaults for these arguments are in the prototype for each device extraction function. When a device extraction function is called without assigning values to these arguments, the default values are used.

Device Extraction Function Prototype

The default properties are “area” and “pj”. The default property calculation function is `calc_diode_properties()`.

```

np : published command_section function
    matrix : in_out device_matrix,
    device_name : string,
    device_body : polygon_layer,
    anode : polygon_layer,
    cathode : polygon_layer,
    optional_pins : list of optional_pin_layer_s = {},
    recognition_layer : polygon_layer = NULL_POLYGON_LAYER,
    reference_layer : polygon_layer = NULL_POLYGON_LAYER,
    processing_layer_hash : processing_layer_h = {},
    properties : list of device_property_s = {"area", DOUBLE, PICO},
                                           {"pj", DOUBLE, MICRO}},
    property_function : function (void) returning void =
                           calc_diode_properties,

    merge_parallel : boolean = false,
    bulk_relationship : bulk_relationship_e = ENCLOSE,
    swappable_pins : list of list of string = {},
    schematic_devices : list of schematic_diode_device_s = {},
    x_card : boolean = false,
    spice_netlist_function: string = "",
    extract_shorted_device: boolean = true,
    processing_mode : processing_mode_e = HIERARCHICAL,
    unique_identifier : string = "",
    swappable_properties : list of pin_property_s = {},
    simulation_model_name : string = "",

```

```

    dlink_libraries : list of dev_dlink_library_handle = {}
) returning void;

```

Default Property Calculation Function

The `calc_diode_properties()` function is the default property function for the `np()` and `pn()` diode extraction functions. The “area” and “pj” properties are calculated using device utility functions that make measurements of the device polygon data. The `dev_is_property()` function determines if the property is defined in the `properties` argument of the extraction function. The property is saved using the `dev_save_double_properties()` if `dev_is_property()` is true.

Example

```

calc_diode_properties: published function (void) returning void
{
    area : double = diode_area();
    pj : double = diode_perim();

    if (dev_is_property("area")) {
        dev_save_double_properties({"area", area});
    }
    if (dev_is_property("pj")) {
        dev_save_double_properties({"pj", pj});
    }
}

```

Device Extraction Function Calls

The device extraction function calls are defined as follows:

- Default property calculation function and default properties

```
np(my_dev_matrix, "my_diode_name", my_anode, my_cathode);
```

Default properties “area” and “pj” are output.

- Default property calculation function and nondefault properties

```
np(my_dev_matrix, "my_diode_name", my_anode, my_cathode,
   properties = {"area", DOUBLE, PICO});
```

The default property calculation function is used to calculate the properties, but only the “area” property is output.

- Custom property calculation function and nondefault properties

```
np(my_dev_matrix, "my_diode_name", my_anode, my_cathode,
   properties = {{ "my_custom_property", DOUBLE, PICO}},
   property_function = my_custom_property_function
);
```

A “my_custom_property” custom property is output and calculated using a custom property function. The “area” and “pj” properties are not output.

Coefficient and Body Position Arguments

The `resistor()`, `capacitor()`, `npn()`, and `pnnp()` device extraction functions have additional arguments that control how the default properties for these devices are calculated.

`resistor()`

`resistor_value`. Sheet resistance coefficient. The default is 0, so the resistance property “r” is 0 by default.

`capacitor()`

`area_capval`. Capacitance per unit area coefficient for parallel plate capacitance. The default is 0.

`coinedge_capval`. Capacitance per unit length for coincident edges between the first and second terminal layers. The default is 0.

`fringe_edge_capval`. Capacitance per unit length coefficient for the first terminal layer overlapping the edge of the second terminal layer. The default is 0.

`perim_capval`. Capacitance per unit length coefficient for edges of the overlapping area that are not otherwise accounted for by the coincident edge or fringe edge coefficients. The default is 0. When `coinedge_capval` is 0, the coincident edges are considered part of the perimeter. When `fringe_edge_capval` is 0, the fringe edges are considered part of the perimeter.

If all of the coefficients are 0, the default calculation for the capacitance property “c” is 0.

`npn()` and `pnnp()`

`body_position`. Argument that determines whether the `BASE`, `EMITTER`, or `COLLECTOR` layer are considered as the body layer. This argument also selects the layer that is used for calculating the default “area” for the device. The default is `EMITTER`, so by default, the “area” property is calculated based on the `EMITTER` area for the device.

Device Utility Functions

A number of device utility functions are available for use in device extraction property functions. Many of these utility functions appear in the default device property calculation functions and can be used in user-defined property functions. The device utility functions are named using a prefix convention that indicates where the functions can be used, as shown in [Table 5-2](#).

Table 5-2 Device Utility Functions

Utility function prefix	Device extraction function
dev_	All device extraction functions, including <code>gendev()</code>
cap_	<code>capacitor()</code>
gdm_	<code>gendev_manual()</code>
ind_	<code>inductor()</code>
mos_	<code>nmos()</code> , <code>pmos()</code>
diode_	<code>np()</code> , <code>pn()</code>
bjt_	<code>nnpn()</code> , <code>pnpn()</code>
res_	<code>resistor()</code>

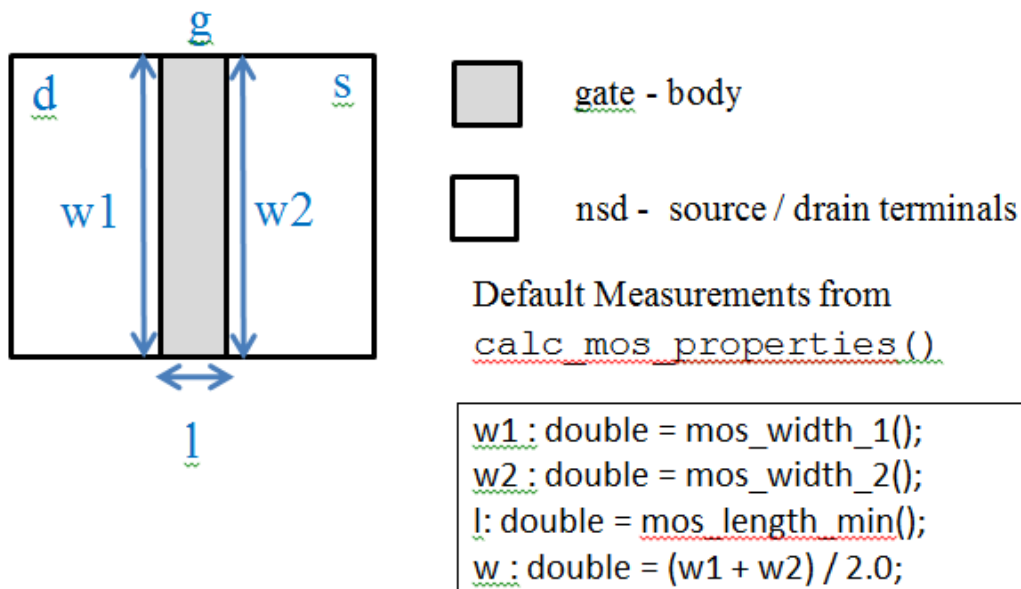
In addition to the device utility functions, the general-purpose functions, which include math functions, string functions, diagnostic functions, and number conversion functions, can be used in the device property calculation functions. One function that can be particularly useful when writing and debugging property functions is the `note()` diagnostic function.

Measuring Polygons for Device Property Calculation

As described in the “[Layers for Device Extraction](#)” section of the Hierarchical Device Extraction Chapter in this guide, the polygons for the device, and thus the polygons used for property calculations, are selected based on the device body layer or, if defined, the device recognition layer. These selected polygons can then be manipulated and measured in the device property function, using the available Utility Functions, to calculate the device properties.

An example NMOS property layer measurement is shown in [Figure 5-2](#)

Figure 5-2 NMOS Property Layer Measurement



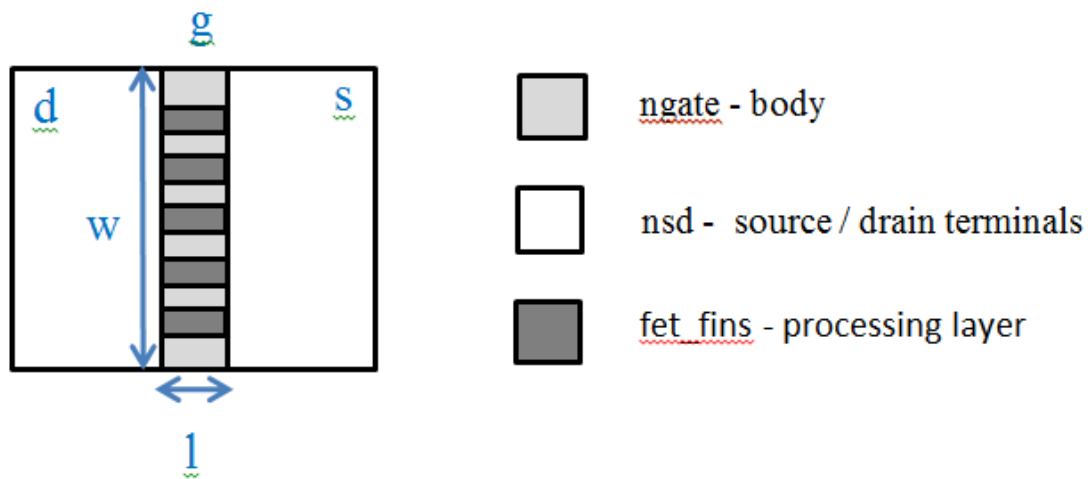
Dedicated utility functions are provided for the most common measurements on the device polygons. The `mos_length_min()` utility function measures the minimum length of the gate body for calculating the device length property “l”. The `mos_width_1()` and `mos_width_2()` utility functions measure the coincident edge lengths between the body and source/drain polygons for calculating the device with property “w”.

Custom Device Extraction Properties and Calculations

Often properties need to be calculated that are not defined by default. These properties can be defined and extracted by defining custom device property calculation functions. The following example demonstrates how to extract the gate area and fin count properties for an NMOS device in addition to the standard width and length properties. The fin count property requires an additional layer, the fin layer, which is passed in as a device processing layer. The fin polygons for each device are selected based on their interaction with the device body and are accessible in the property extraction function by calling `dev_processing_layer()`.

An example of custom property calculations (NMOS) is shown in [Figure 5-3](#)

Figure 5-3 Custom Property Calculations (NMOS)



The following properties are extracted for the preceding device:

- Default property width: "w"
- Default property length: "l"
- Custom property for gate area: "area"
- Custom property for the number of fet_fins polygons: "fins"

Example of a custom property function:

```

my_custom_mos_property_function: function (void) returning void
{
    // Calculate the "w" and "l" properties using the default property
    // function mos_length_min();

    // Calculate the custom "area" and "nfin" properties
    // Calculate the gate area
    gate_area = mos_gate_area();

    // Get the fin polygons that interact with the gate
    fin_polygons = dev_processing_layer("fet_fins");

    // Count the fin shapes
    fin_count = dev_count_polygons(fin_polygons);

    // Save the property only if the property is defined in the nmos()
    // properties argument

    if(dev_is_property("nfin")){
        dev_save_double_properties({{"nfin", fin_count}});
    }
    if(dev_is_property("area")){
        dev_save_double_properties({{"area", gate_area}});
    }
}

```

Example of an extraction function:

```

nmos(my_devices, "n", nsd, ngate, nsd, {{psub}},
     properties = {{{"l", DOUBLE, MICRO},
                    {"w", DOUBLE, MICRO},
                    {"area", DOUBLE, PICO},
                    {"nfin", DOUBLE }},
     property_function = my_custom_mos_property_function,
     processing_layer_hash = { "fet_fins" => {fet_fins} }
);

```

Example of an extracted device with “l”, “w”, “area”, and “nfin” properties:

```

M1 d g s b n l=1.0u w=13.0u area=13.0p nfin=5.0

```


6

Layout-Versus-Schematic Flows

Layout-versus-schematic (LVS) flows are described in the following sections:

- [Creating a Black-Box LVS Flow](#)
- [Netlist-Versus-Netlist Flow](#)

Creating a Black-Box LVS Flow

Setting up a black-box LVS flow allows you to validate designs before all of the building blocks are completed. This flow allows for a parallel development model where different groups can work concurrently on sections of a design. In this flow, IC Validator LVS can be run on a block where the connections to an incomplete cell are checked but the contents of the incomplete cell are ignored.

- [Setting Up a Black-Box Flow](#)
- [Usage Models](#)

Setting Up a Black-Box Flow

To set up a black-box LVS flow, start by identifying the blocks that you want to treat as black-box cells during the LVS run using the `lvs_black_box_options()` function. This function defines the schematic to layout cell mapping.

For example, if the 'invb' and 'nor2b' cells are not yet completed, you can specify them as black boxes:

```
lvs_black_box_options(  
    equiv_cells = {  
        {schematic_cell = "invb", layout_cell = "invb"},  
        {schematic_cell = "nor2b", layout_cell = "nor2b"}  
    }  
);
```

You can also provide LVS black-box options for each black-box cell separately. This is useful if the configuration for each black box is unique. For example,

```
lvs_black_box_options(  
    equiv_cells = {{ "invb", "invb" }},  
    equate_ports = {{ "A", "A1" }}  
);  
lvs_black_box_options(  
    equiv_cells = {{ "nor2b", "nor2b" }}  
);
```

The use of arguments available in the `lvs_black_box_options()` function that control the black-box configuration is discussed in the [“Usage Models”](#) section.

To provide the black-box information to the IC Validator run, the `lvs_black_box_options()` function can be

- Inserted directly in the runset.
- Inserted via a PXL formatted file with a preprocessor include directive in your IC Validator runset before the assign functions. For example,

```
#include <black_box_file>
```

- Passed to the IC Validator runset with the `-e` command-line option. The `-e` command-line option reads the `equiv_options()` and `lvs_black_box_options()` functions from the specified file. See Chapter 1, “IC Validator Basics” in the *IC Validator User Guide* for more information about the `-e` command-line option.

For example,

```
% icv -e black_box_file runset.rs
```

When a cell is designated as a black-box cell, the IC Validator tool matches the schematic ports to the layout ports by name. When a black-box cell is successfully matched, it is labeled as such in the summary details for the cells in which it was placed. For example,

Post-compare summary (* = unmatched devices or nets):

Matched	Schematic unmatched	Layout unmatched	Instance types [schematic, layout]
-----	-----	-----	-----
8	0	0	[invb, invb] (blackbox)
3	0	0	[nand2b, nand2b]
1	0	0	[nand3b, nand3b]
3	0	0	[nor2b, nor2b] (blackbox)
1	0	0	[nor3b, nor3b]
-----	-----	-----	-----
16	0	0	Total instances
21	0	0	Total nets

Usage Models

This section shows several black-box examples:

- [Example 1: All Ports Match Between Netlists](#)
- [Example 2: Port Names Do Not Match Between Netlists](#)
- [Example 3: Extra Ports in Layout Black-Box Cell](#)
- [Example 4: Extra Schematic Ports on Black-Box Cells](#)
- [Example 5: Extra Untexted Layout Ports on Black-Box Cells](#)
- [Example 6: Untexted Layout Ports on Black-Box Cells](#)

- [Example 7: Symmetry and Black-Box Cells](#)

Example 1: All Ports Match Between Netlists

In the following example, all of the port counts and names match:

Schematic: ADD4.sp	Layout: add4.net
<pre>.SUBCKT invb GND VDD A Z M1 GND A Z GND n L=1u W=13u M2 VDD A Z VDD p L=1u W=20.5u .ENDS .SUBCKT nor2b GND VDD QN A B M1 QN B GND GND n L=1u W=13u M2 QN A GND GND n L=1u W=13u M3 n11 B QN VDD p L=1u W=20.5u M4 n11 A VDD VDD p L=1u W=20.5u .ENDS</pre>	<pre>{CELL invb {PORT VDD GND Z A} {PROP top=50.000000 bottom=0.000000 left=0.000000 right=12.000000} } {CELL nor2b {PORT VDD GND QN A B} {PROP top=50.000000 bottom=0.000000 left=0.000000 right=18.000000} }</pre>

LVS successfully runs to completion because the port counts and names match. If the port counts or names do not match between the schematic and layout, then, by default, the IC Validator tool reports an error.

Example 2: Port Names Do Not Match Between Netlists

If the layout changes such that the layout netlist has a net named differently than for the schematic netlist, then the IC Validator tool reports an error. In the following example, in cell 'invb' the port name changes from 'A' to 'A1'.

Schematic: ADD4.sp	Layout: add4.net
<pre>.SUBCKT invb GND VDD A Z M1 GND A Z GND n L=1u W=13u M2 VDD A Z VDD p L=1u W=20.5u .ENDS .SUBCKT nor2b GND VDD QN A B M1 QN B GND GND n L=1u W=13u M2 QN A GND GND n L=1u W=13u M3 n11 B QN VDD p L=1u W=20.5u M4 n11 A VDD VDD p L=1u W=20.5u .ENDS</pre>	<pre>{CELL invb {PORT VDD GND Z A1} {PROP top=50.000000 bottom=0.000000 left=0.000000 right=12.000000} } {CELL nor2b {PORT VDD GND QN A B} {PROP top=50.000000 bottom=0.000000 left=0.000000 right=18.000000} }</pre>

The LVS run stops and prints the following message in the LVS log file (add4_lvs.log):

```
Processing black boxes' pins...
ERROR: Schematic port "A" does not have a corresponding port in the
layout in blackbox {invb, invb}
ERROR: Layout port "A1" does not have a corresponding port in the
schematic in blackbox {invb, invb}
```

You can fix this error by manually defining the port mapping for the pins that do not match using the `lvs_black_box_options()` function for this equivalence. For example,

```
lvs_black_box_options(
    equiv_cells = {"invb", "invb"},
    equiv_ports = {"A", "A1"}
);
```

Example 3: Extra Ports in Layout Black-Box Cell

When dealing with unfinished blocks, the IC Validator tool might create extra layout ports because of unintended hierarchical interactions. If the layout changed so that the layout netlist has an extra port compared to the schematic, then the IC Validator tool reports an error. In the following example, an extra port, 'B', was created in the 'invb' cell.

Schematic: ADD4.sp	Layout: add4.net
<pre>.SUBCKT invb GND VDD A Z M1 GND A Z GND n L=1u W=13u M2 VDD A Z VDD p L=1u W=20.5u .ENDS .SUBCKT nor2b GND VDD QN A B M1 QN B GND GND n L=1u W=13u M2 QN A GND GND n L=1u W=13u M3 n11 B QN VDD p L=1u W=20.5u M4 n11 A VDD VDD p L=1u W=20.5u .ENDS</pre>	<pre>{CELL invb {PORT VDD GND Z A B} {PROP top=50.000000 bottom=0.000000 left=0.000000 right=12.000000} } {CELL nor2b {PORT VDD GND QN A B} {PROP top=50.000000 bottom=0.000000 left=0.000000 right=18.000000} }</pre>

The LVS run stops and prints the following message in the LVS log file (add4_lvs.log):

```
Processing black boxes' pins...
ERROR: Layout port "B" does not have a corresponding port in the
schematic in blackbox {invb, invb}
```

You can fix this error in one of two ways:

1. Manually remove the port using the `remove_layout_ports` argument of the `lvs_black_box_options()` function. You can also use this argument if there are extra schematic ports that need to be removed. Unless the interaction is known ahead of time

and the ports are removed in the first run, a second IC Validator run is needed to complete the LVS run. For example,

```
lvs_black_box_options(
    equiv_cells      = {{"invb", "invb"}},
    remove_layout_ports = {"B"}
);
```

2. Use the `report_black_box_errors` argument of the `match()` function to downgrade the error to a less severe message. The default for the `extra_layout_ports` option is `ERROR`. If there is an extra layout port, an error is reported and the IC Validator run stops. Optionally, you can set the option to force the tool to report the error but continue to match the topology of the design, to issue a warning and continue, or to ignore the error and continue. The advantage of using the `report_black_box_errors` argument is that you do not need advanced knowledge of the design differences between the layout and schematic. The options of the `report_black_box_errors` argument are: `ERROR_NO_ABORT`, `WARNING`, and `NONE`.

- `ERROR_NO_ABORT`

The tool prints the error message but continues to compare the topology of the design. The tool ignores all connections to extra pins of black-box cells. For example,

```
match(
    ...
    report_black_box_errors = {
        extra_layout_ports      = ERROR_NO_ABORT,
        extra_schematic_ports = ERROR_NO_ABORT
    },
    ...
);
```

If the only problem found in the design is the extra layout black-box port error, the final LVS result is reported as `FAIL` with the following message:

```
ERROR: Following are 8 layout port-error blackbox instances.
Check lvs.log file for extra ports in the blackbox cells.
```

```
Instance name (type)
-----
599CF7565 (invb)(-108.000, 0.000)
599CF7566 (invb)(18.000, 0.000)
599CF7567 (invb)(108.000, 0.000)
599CF7568 (invb)(-24.000, 0.000)
599CF7569 (invb)(-84.000, 0.000)
599CF75610 (invb)(-54.000, 0.000)
599CF75611 (invb)(48.000, 0.000)
599CF75612 (invb)(90.000, 0.000)
```

The following message is printed in the LVS log file (add4_lvs.log):

```
Processing black boxes' pins...
ERROR: Layout port "B" does not have a corresponding port in the
schematic in blackbox {inbv, inbv}
```

○ WARNING

The tool reports a warning and continues to compare the topology of the design. The tool ignores all connections to extra pins of the black-box cell. For example,

```
match(
    ...
    report_black_box_errors = {
        extra_layout_ports    = WARNING,
        extra_schematic_ports = WARNING
    },
    ...
);
```

The following warning messages are printed in the LVS log file (add4_lvs.log):

```
Processing black boxes' pins ...
WARNING: Layout port "B" does not have a corresponding port in the
schematic in blackbox {inbv, inbv}
```

○ NONE

With this setting, the tool does not report a warning or error in the LVS log file and continues to compare the topology of the design. All connections to extra pins of the black-box cell are ignored.

Example 4: Extra Schematic Ports on Black-Box Cells

When you work with unfinished cells, the layout netlist is missing ports as compared to the schematic. The design is not complete enough to have all of the appropriate hierarchical interactions needed to create the ports. In the following example, the 'A' port of the 'invb' cell was not extracted.

Schematic: ADD4.sp	Layout: add4.net
<pre>.SUBCKT invb GND VDD A Z M1 GND A Z GND n L=1u W=13u M2 VDD A Z VDD p L=1u W=20.5u .ENDS .SUBCKT nor2b GND VDD QN A B M1 QN B GND GND n L=1u W=13u M2 QN A GND GND n L=1u W=13u M3 n11 B QN VDD p L=1u W=20.5u M4 n11 A VDD VDD p L=1u W=20.5u .ENDS</pre>	<pre>{CELL invb {PORT VDD GND Z } {PROP top=50.000000 bottom=0.000000 left=0.000000 right=12.000000} } {CELL nor2b {PORT VDD GND QN A B} {PROP top=50.000000 bottom=0.000000 left=0.000000 right=18.000000} }</pre>

The following error message is printed in the LVS log file (add4_lvs.log):

```
Processing black boxes' pins...
ERROR: Schematic port "A" does not have a corresponding port in the
layout in blackbox {invb, invb}
```

As in Example 3, which has extra layout ports, you can fix this error in one of two ways:

1. Manually remove the port using the `remove_schematic_ports` argument of the `lvs_black_box_options()` function. You can also use this argument if there are extra schematic ports that need to be removed. For example,

```
lvs_black_box_options(
    equiv_cells          = {{"invb", "invb"}},
    remove_schematic_ports = {"A"}
);
```

2. Use the `report_black_box_errors` argument of the `match()` function to downgrade the error to a less severe message. The default for the `extra_schematic_ports` option is `ERROR`. If there is an extra schematic port, an error is reported and the IC Validator run stops. Optionally, you can set the option to force the tool to report the error but continue to match the topology of the design, to issue a warning and continue, or to ignore the message and continue. The advantage of using the `report_black_box_errors` argument is that you do not need advanced knowledge of the design differences between the layout and schematic.

- ERROR_NO_ABORT

The tool prints the error message but continues to compare the topology of the design. The tool ignores all connections to extra pins of the black-box cells. For example,

```
match(
    ...
    report_black_box_errors = {
        extra_layout_ports    = ERROR_NO_ABORT,
        extra_schematic_ports = ERROR_NO_ABORT
    },
    ...
);
```

If the only problem found in the design is the extra schematic black-box port error, the final result is reported as FAIL with the following message:

```
ERROR: Following are 8 schematic port-error blackbox instances.
Check lvs.log file for extra ports in the blackbox cells.
```

```
Instance name (type)
-----
x33 (invb)
x37 (invb)
x39 (invb)
x40 (invb)
x47 (invb)
x48 (invb)
x50 (invb)
x52 (invb)
```

The following message is printed in the LVS log file (add4_lvs.log):

```
Processing black boxes' pins ...
ERROR: Schematic port "A" does not have a corresponding port in the
layout in blackbox {invb, invb}
```

- WARNING OR ERROR

The results are the same as in Example 3.

Example 5: Extra Untexted Layout Ports on Black-Box Cells

Untexted layout ports can be created during an IC Validator run if there is a hierarchical interaction between a parent net and an untexted child cell net. For black-box cells, these ports are treated separately from extra layout ports because they might be indicative of other design issues that might need to be investigated. In the following example, there is an extra port called '1' on the 'invb' cell.

Schematic: ADD4.sp	Layout: add4.net
<pre>.SUBCKT invb GND VDD A Z M1 GND A Z GND n L=1u W=13u M2 VDD A Z VDD p L=1u W=20.5u .ENDS .SUBCKT nor2b GND VDD QN A B M1 QN B GND GND n L=1u W=13u M2 QN A GND GND n L=1u W=13u M3 n11 B QN VDD p L=1u W=20.5u M4 n11 A VDD VDD p L=1u W=20.5u .ENDS</pre>	<pre>{CELL invb {PORT VDD GND Z A 1} {PROP top=50.000000 bottom=0.000000 left=0.000000 right=12.000000} } {CELL nor2b {PORT VDD GND QN A B} {PROP top=50.000000 bottom=0.000000 left=0.000000 right=18.000000} }</pre>

Even though the created port is an extra port, when the tool is processing the port, by default an untexted port error is reported and the IC Validator tool stops. For example,

```
Processing black boxes' pins...
ERROR: Layout port "1" is untexted in blackbox {invb, invb}
```

This error can be downgraded using the `untexted_layout_ports` option of the `report_black_box_errors` argument, as shown in Example 2. For example,

```
match(
  ...
  report_black_box_errors = {
    untexted_layout_ports = ERROR_NO_ABORT
  },
  ...
);
```

Now, the tool reports an error in the LVS log file and continues to compare the topology of the design but ignores all connections to extra, untexted pins of the black-box cell. For example,

```
ERROR: Following are 8 layout port-error blackbox instances.
Check lvs.log file for extra ports in the blackbox cells.
```

```
Instance name (type)
-----
599CF7565 (invb)(-108.000, 0.000)
599CF7566 (invb)(18.000, 0.000)
```

```

599CF7567 (invb)(108.000, 0.000)
599CF7568 (invb)(-24.000, 0.000)
599CF7569 (invb)(-84.000, 0.000)
599CF75610 (invb)(-54.000, 0.000)
599CF75611 (invb)(48.000, 0.000)
599CF75612 (invb)(90.000, 0.000)

```

The following message is printed in the LVS log file (add4_lvs.log):

```

Processing black boxes' pins...
ERROR: Layout port "1" is untexted in blackbox {invb, invb}

```

Note:

All untexted black-box ports are ignored if the error is downgraded to a warning. For example, even if you have the following settings, only the warning for the untexted layout ports is reported and no error for the extra port is reported:

```

match(
...
  report_black_box_errors = {
    untexted_layout_ports = WARNING,
    extra_layout_ports    = ERROR
  },
...
);

```

Example 6: Untexted Layout Ports on Black-Box Cells

In this example, the 'A' port of the 'invb' cell is replaced as untexted, and as a result, the number of ports matches between the schematic and layout. For example,

Schematic: ADD4.sp	Layout: add4.net
<pre> .SUBCKT invb GND VDD A Z M1 GND A Z GND n L=1u W=13u M2 VDD A Z VDD p L=1u W=20.5u .ENDS .SUBCKT nor2b GND VDD QN A B M1 QN B GND GND n L=1u W=13u M2 QN A GND GND n L=1u W=13u M3 n11 B QN VDD p L=1u W=20.5u M4 n11 A VDD VDD p L=1u W=20.5u .ENDS </pre>	<pre> {CELL invb {PORT VDD GND Z 1} {PROP top=50.000000 bottom=0.000000 left=0.000000 right=12.000000} } {CELL nor2b {PORT VDD GND QN A B} {PROP top=50.000000 bottom=0.000000 left=0.000000 right=18.000000} } </pre>

The default settings of the options of the `report_black_box_errors` argument are:

```

match(
...
  report_black_box_errors = {

```

```

        untexted_layout_ports = ERROR,
        extra_layout_ports    = ERROR
    },
    ...
);

```

The following error messages are printed in the LVS log file (add4_lvs.log):

```

Processing black boxes' pins ...
ERROR: Schematic port "A" does not have a corresponding port in the
layout in blackbox {invb, invb}
ERROR: Layout port "1" is untexted in blackbox {invb, invb}

```

Setting both options to a non-fatal value, such as `WARNING`, however, forces the IC Validator tool to report an error because the tool cannot find a match for the 'A' port. For example,

```

match(
    ...
    report_black_box_errors = {
        untexted_layout_ports = WARNING,
        extra_layout_ports    = WARNING
    },
    ...
);

```

The following error messages are printed:

```

Processing black boxes' pins ...
ERROR: Schematic port "A" does not have a corresponding port in the
layout in blackbox {invb, invb}
WARNING: Layout port "1" is untexted in blackbox {invb, invb}

```

You can avoid this issue by explicitly setting the port matching in the `lvs_black_box_options` argument, as shown in Example 1. For example,

```

lvs_black_box_options(
    equiv_cells = {{ "invb", "invb" }},
    equate_ports = {{ "A", "1" }}
);

```


Example 7: Symmetry and Black-Box Cells

Black-box cells require a one-to-one correspondence between schematic and layout ports. Each pin is treated as unique, and no symmetry is allowed. In the following example of a black-box setup, the 'invb' and 'nor2b' ports match in name and count. However, in the layout, one of the connections to the 'A' and 'B' ports of the 599CF75616 instance of the 'nor2b' cell is swapped.

Schematic: ADD4.sp	Layout: add4.net
<pre>.SUBCKT invb GND VDD A Z M1 GND A Z GND n L=1u W=13u M2 VDD A Z VDD p L=1u W=20.5u .ENDS .SUBCKT nor2b GND VDD QN A B M1 QN B GND GND n L=1u W=13u M2 QN A GND GND n L=1u W=13u M3 n11 B QN VDD p L=1u W=20.5u M4 n11 A VDD VDD p L=1u W=20.5u .ENDS .SUBCKT cs_add GND C SUM A B COUT VDD ... x49 GND VDD n28 A B nor2b x46 GND VDD n3 n29 n16 nor2b x34 GND VDD n34 n35 n4 nor2bENDS</pre>	<pre>{CELL invb {PORT VDD GND Z A} } {CELL nor2b {PORT VDD GND QN A B} } {CELL cs_add ... {INST 599CF75615=nor2b {TYPE CELL} {PIN VDD=VDD GND=GND 13=QN 15=A 4=B}} {INST 599CF75616=nor2b {TYPE CELL} {PIN VDD=VDD GND=GND 7=QN B=A A=B}} {INST 599CF75617=nor2b {TYPE CELL} {PIN VDD=VDD GND=GND 9=QN 16=A 18=B}} ... }</pre>

If these cells are not black-box cells, then the IC Validator tool recognizes that the 'A' and 'B' pins are logically swappable based on the topology of the 'nor2b' contents, and the LVS report is clean. However, in the following flow, this situation results in an LVS failure:

```
> cs_add != cs_add (level = 1)
```

Error summary:

```
0 unmatched schematic device
2 unmatched schematic nets
0 unmatched layout device
2 unmatched layout nets
```

```
16 matched devices
19 matched nets
```

```
...
```

Diagnostic summary:

```

1 wrongly connected net group

DIAGNOSTIC: Wrongly connected nets group

The most possible equated nets are grouped for cross probing.

Group 1 of 1:

Schematic net : connections          Layout net : connections
-----
A : 4                                ~=          A : 4
B : 4                                ~=          B : 4
...

```

To have the IC Validator tool recognize these pins as swappable in the black-box flow, you need to manually define the swappable pins in the `lvs_black_box_options()` function. For example,

```

lvs_black_box_options(
    equiv_cells          = {{"nor2b", "nor2b"}},
    schematic_swappable_ports = {{"A", "B"}}
);

```

Netlist-Versus-Netlist Flow

The IC Validator tool has a netlist-versus-netlist (NVN) flow that you can use to compare two netlists of any origin. An NVN flow differs from an LVS flow in that the NVN flow does not perform device extraction from a layout netlist; instead, the IC Validator tool reads and compares two standalone netlists.

Modes for Running NVN

There are three modes for running the NVN flow with the IC Validator tool:

1. No runset
Uses command-line switches for a simple topological comparison.
2. Partial runset
Uses a simple runset to define only non-default behaviors. Device mapping functions are optional in this flow.
3. Full runset
Uses the full flexibility of an LVS runset. The tool replaces device configuration functions with mapping functions.

[Table 6-1](#) summarizes the modes. Details of the modes are discussed in following sections.

Table 6-1 *Netlist-Versus-Netlist Modes*

Mode	Usage model	Runset configuration
No runset	Only standard devices are defined in netlists. This is a topological comparison without merging, filtering, or checking of properties.	Command-line only.
Partial runset	Mostly standard devices are defined in netlists. This is a topological comparison with merging, filtering, and checking of properties.	Device mapping functions are required for nonstandard devices. The runset is fully configurable for compare settings.
Full runset	A combination of standard and nonstandard devices are defined in the netlists. This is a topological comparison with merging, filtering, and property checking	Device mapping functions are required for all devices. The runset is fully configurable for compare settings.

Note:

In an NVN flow, the term *schematic* refers to the reference netlist and *layout* refers to the candidate netlist. This terminology is used because the IC Validator NVN flow uses the same compare engine as is used for other LVS flows. All output is formatted in terms of *schematic* and *layout*.

Device Mapping Functions

Device mapping functions define compare-relevant information that would otherwise be obtained from device configuration functions during LVS. The information includes device type and name, pin names, mapping to schematic devices, and swappable pins and properties.

[Table 6-2](#) shows the correspondence between the device configuration functions and NVN mapping functions.

Table 6-2 *Device Configuration and NVN Mapping Functions Correspondence*

LVS device configuration functions	NVN mapping functions
<code>capacitor()</code>	<code>map_capacitor()</code>
<code>gendev()</code>	<code>map_gendev()</code>

Table 6-2 Device Configuration and NVN Mapping Functions Correspondence (Continued)

LVS device configuration functions	NVN mapping functions
<code>inductor()</code>	<code>map_inductor()</code>
<code>nmos()</code>	<code>map_nmos()</code>
<code>np()</code>	<code>map_np()</code>
<code>pmos()</code>	<code>map_pmos()</code>
<code>npn()</code>	<code>map_npn()</code>
<code>pn()</code>	<code>map_pn()</code>
<code>pnnp()</code>	<code>map_pnp()</code>
<code>resistor()</code>	<code>map_resistor()</code>

The `netlist_vs_netlist` argument of the `init_compare_matrix()` function specifies how device instances are mapped. The options are

- `FULL_RUNSET`
NVN mapping function calls are required for every device instance in each imported netlist. This option is the default.
- `PARTIAL_RUNSET`
The mapping of device instance to device type can be inferred from the netlist without an explicit mapping function call for each instance.

NVN Flow With a Runset

Whether you are using a partial or a full runset flow, during an NVN run, you can call the following functions:

- `compare()`
- `init_compare_matrix()`
- `read_layout_netlist()`
- `schematic()`

In addition, you can use any compare setting functions that have `device_type` and `device_name` arguments. These functions are: `check_property()`, `check_property_off()`, `filter()`, `filter_off()`, `fopen()`, `merge_parallel()`, `merge_parallel_off()`, `merge_series()`, `merge_series_off()`, and `recalculate_property()`.

Full Runset Flow

If the `netlist_vs_netlist` argument is set to `FULL_RUNSET`, the IC Validator tool does not perform automatic device mappings. A full NVN runset containing NVN mapping function calls for every device instance in each imported netlist is required. An error occurs for any imported device instance that is not mapped to an IC Validator device type by a mapping function.

The following is an example of an NVN flow with a full runset:

- Input schematic netlist (ADD4_sch.sp):

```
.SUBCKT invb GND VDD A Z
M1 GND A Z GND n L=1u W=13u
M2 VDD A Z VDD p L=1u W=20.5u
.ENDS
```

- Input layout netlist (ADD4_layout.sp):

```
.SUBCKT invb GND VDD A Z
M1 GND A Z GND n L=1u W=13u
M2 VDD A Z VDD p L=1u W=20.5u
.ENDS
```

- Runset (full_nvn.rs):

```
#include <icv.rh>

// import netlists
sch = schematic({"ADD4_sch.sp", SPICE});
lay = read_layout_netlist({"ADD4_layout.sp", SPICE});

compare_state = init_compare_matrix(
    netlist_vs_netlist = FULL_RUNSET
);

// map functions
map_nmos(compare_state, "n");
map_pmos(compare_state, "p");

// compare specific behaviors
merge_parallel(compare_state, NMOS, {"n"});
merge_parallel(compare_state, PMOS, {"p"});

check_property(compare_state, NMOS, {"n"}, {"l"}, {"w"});
```

```

check_property(compare_state, PMOS, {"p"}, {"l"}, {"w"}));

compare(compare_state, sch, lay,
        schematic_top_cell = "add4",
        layout_top_cell = "add4"
);

```

In the full runset flow example, the schematic and layout input netlists are defined and the compare state is initialized with the `FULL_RUNSET` option of the `netlist_vs_netlist` argument. The netlists are small, so only one NMOS function and one PMOS function are needed. Parallel merging is enabled on each MOS device, and the top cell names are defined in the `compare()` function. Everything needed to run NVN is set.

Run the NVN flow as follows:

```
%icv full_nvn.rs
```

Partial Runset Flow

If the `netlist_vs_netlist` argument is set to `PARTIAL_RUNSET`, the IC Validator tool performs automatic device mappings. This means that NVN mapping function calls for every device instance in each imported netlist are not needed in the NVN runset. Device mapping is required for standard devices with non-default characteristics, such as extra pins, or generic devices declared as X-cards in SPICE.

If an explicit device mapping function call exists in the runset for a particular device name, then instances corresponding to the device name are handled according to the mapping function call. However, if an explicit device mapping function call does not exist in the runset for a particular device name, then instances corresponding to the device name can be implicitly mapped by type as follows:

1. The device type is implicitly determined by the `TYPE` field inside the IC Validator format netlist input. The `TYPE` field is derived from the leading character in instance names read from SPICE netlists.
2. However, because the `TYPE` field does not differentiate N-type from P-type devices, the device type of doped devices like MOS, bipolar transistors, and diodes cannot be uniquely determined based on the `TYPE` field alone. For these devices, the device type is implicitly determined by the `device_type` argument of any compare-related function call that also specifies a non-empty list of the `device_names` argument.

Because of automatic mapping, a compare-related function call that specifies a `device_type` argument value equal to NMOS, PMOS, NP, PN, NPN, or PNP, must also specify a non-empty value for the `device_names` argument. In this situation, an error results if an empty list value is specified for the `device_names` argument. Note that this restriction only exists for the partial NVN runset mode (`PARTIAL_RUNSET` option).

Implicit type mappings must not conflict. This means that the mapping of device name to device type must be consistent across all calls to compare-related functions. When a conflict occurs, an error occurs.

The following is an example partial NVN runset:

- Input schematic netlist (ADD4_sch.sp):

```
.SUBCKT invb GND VDD A Z
M1 GND A Z GND n L=1u W=13u
M2 VDD A Z VDD p L=1u W=20.5u
.ENDS
```

- Input layout netlist (ADD4_layout.sp):

```
.SUBCKT invb GND VDD A Z
M1 GND A Z GND n L=1u W=13u
M2 VDD A Z VDD p L=1u W=20.5u
.ENDS
```

- Runset (partial_nvn.rs):

```
#include <icv.rh>

// import netlists
sch = schematic({{"ADD4_sch.sp", SPICE}});
lay = read_layout_netlist({{"ADD4_layout.sp", SPICE}});

compare_state = init_compare_matrix(
    netlist_vs_netlist = PARTIAL_RUNSET
);

// map functions
// no map functions required

// compare specific behaviors
merge_parallel(compare_state, NMOS, {"n"});
merge_parallel(compare_state, PMOS, {"p"});

check_property(compare_state, NMOS, {"n"}, {"l"}, {"w"});
check_property(compare_state, PMOS, {"p"}, {"l"}, {"w"});

compare(compare_state, sch, lay
    schematic_top_cell = "add4",
    layout_top_cell = "add4"
);
```

Netlist-Versus-Netlist Without a Runset

To run the NVN flow without a runset, use the `-nvn` command-line option. The schematic and layout netlists must be specified on the command line along with the formats. The cell names must also be defined. For devices to be appropriately matched, they must have the same name, same number of pins, and the same standard pin names.

The following example shows the command line for an NVN run without a runset:

```
%icv -nvn -ln layout_filename -lnf layout_format -s schematic_filename \  
-sf schematic_format -c layout_top_cell [-stc schematic_top_cell]
```

Note:

If the layout or schematic format is ICV, it must be IC Validator format 2.0 or greater.

Output Files

For the no runset mode, the IC Validator tool default settings are used. Therefore, the `compare()` function writes the default output files.

For the partial and full runset modes, the settings of the `compare()` function determine the output files.

7

Compare Functions Basics

This chapter explains some of the basics of the compare functions.

The information described in this chapter is:

- [Predefined Name Matches](#)
- [Complementary Functions](#)
- [Precedence Rule](#)

Predefined Name Matches

During LVS compare, various mappings are made by IC Validator.

The alias names and associated matches that are shown in [Table 7-1](#) are predefined. Their values are compared between the schematic and layout netlists.

Note:

The names are case insensitive.

Table 7-1 Predefined Properties for Schematic and Layout Files

Property name	Alias names	Description
A	A, AREA	Area
C	C, CAP, CAPVAL, CVAL	Capacitance
L	L, LEN, LENG, LENGTH	Length
M	M, MULT	Multiplier
PJ	PJ	Perimeter of junction
R	R, RES, RESVAL, RVAL	Resistance
W	W, WIDTH	Width

Complementary Functions

Complementary functions are used to selectively deactivate compare functionality that was activated by a previous function for a set of devices. The complementary compare functions are

`check_property()` — `check_property_off()`

`filter()` — `filter_off()`

`merge_series()` — `merge_series_off()`

`merge_parallel()` — `merge_parallel_off()`

`short_equivalent_nodes()` — `short_equivalent_nodes_off()`

Precedence Rule

The following compare functions can be set at the device name level and the device type level.

```
check_property()  
check_property_off()  
filter()  
filter_off()  
merge_parallel()  
merge_parallel_off()  
merge_series()  
merge_series_off()  
recalculate_property()  
short_equivalent_nodes()  
short_equivalent_nodes_off()
```

The function settings use this precedence:

- A. Compare functions with the device type specified.
- B. Compare functions with the device name specified.

The precedence order is B > A.

To enable parallel merging for all RES devices except those devices formed from polygon layers, use syntax as shown here:

```
merge_parallel(my_devices, RESISTOR);  
merge_parallel_off(my_devices, RESISTOR, {"poly_res"});
```

The first statement turns parallel merging on for all resistors. The second function turns off parallel merging for only `poly_res`.

If you repeat a function for the same level, such as `device_name`, the second function replaces the first one. In the following example, `poly_res` is used twice:

```
merge_parallel_off(my_devices, RESISTOR, {"poly_res"});  
merge_parallel(my_devices, RESISTOR, {"poly_res"});
```

In this case, `merge_parallel()` is enabled for `poly_res` because both statements are specified at the same level of precedence and the second function call replaces the first.

8

Netlist Modification

To achieve a clean comparison result, the topologies and properties of the netlists must be modified. Topology modifications consist of device merging, recalculation of properties for merged devices, and device filtering. The functions used for netlist modification provide a number of different options by which the modifications can be controlled or customized.

This chapter has the following sections:

- [Merging Devices](#)
- [Custom Merging Equations for Device Properties](#)
- [Series Merging Devices](#)
- [Parallel Device Merging Short Equivalent Nodes](#)
- [Filtering](#)

Merging Devices

You use device merging to modify the connected device topologies by reducing multiple parallel or series connected devices into a single merged device instance. For example, a reduction of multiple devices makes it possible to match a single device in one netlist to an equivalent merged device in another netlist. Additionally, for series merged MOS devices, merged devices with logically equivalent gate connections can be generated for series connected transistor stacks to make it possible to match different, but logically equivalent, transistor stacks.

Merging is controlled based on various characteristics such as device type, device name, equivalence cells, and device property tolerances. Additionally, when devices are merged, the properties must be recalculated for the new merged device. Device merging functions have default equations that are used for recalculating the property values for commonly used properties for the merged devices. Default equations and properties are described in the *IC Validator Reference Manual*. Custom property merging functions can also be defined.

The merging functions, and how merging is controlled based on these various device characteristics, are discussed in this chapter. Most of the examples are provided in the context of parallel merged MOS devices, but the ability to control merging based on properties, property calculations, and tolerances extends to other devices and merge functions.

Merging Parallel Devices

The `merge_parallel()` function defines the criteria for merging devices in parallel.

Syntax

```
merge_parallel(
    state                = compare_state,
    device_type          = NMOS | PMOS | NPN | PNP | PN | NP |
                        RESISTOR | CAPACITOR | INDUCTOR | GENERIC,
    device_names         = {"string", ...}, //optional
    exclude_tolerances = {{property      = "string",
                           tolerance      = doubleconstraint,
                           tolerance_type = RELATIVE | ABSOLUTE},
                           ...}, //optional
    exclude_function    = "string", //optional
    property_functions  = {{property_function = "string",
                           property          = "string"}, ...}, //optional
    equiv_cells         = {{schematic_cell = "string",
                           layout_cell     = "string"},
                           ...} //optional
);
```

The `merge_parallel_off()` function defines the criteria for excluding certain devices from being merged in parallel by type, device name, or equivalence cells.

Syntax

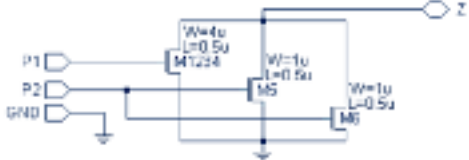
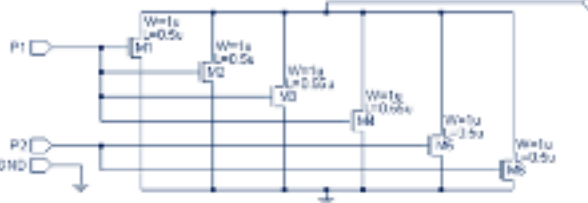
```
merge_parallel_off(
    state           = compare_state,
    device_type     = NMOS | PMOS | NPN | PNP | PN | NP |
                    RESISTOR | CAPACITOR | INDUCTOR | GENERIC,
    device_names    = {"string", ...}, //optional
    equiv_cells     = {{schematic_cell = "string",
                        layout_cell    = "string"},
                      ...} //optional
);
```

Example of Parallel Merging all NMOS Devices Using Defaults

In this example, the defaults are used for all optional settings. All NMOS devices are merged. The default property equations are used for all properties. Therefore, the W properties are summed and the L properties are averaged, as shown in [Figure 8-1](#).

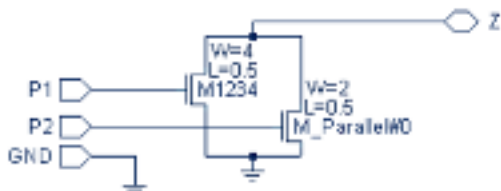
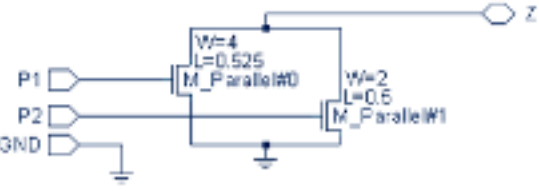
```
merge_parallel(compare_settings, NMOS);
```

Figure 8-1 Input Netlists

Schematic	Layout
<pre>.SUBCKT top P1 P2 & GND M1234 & P1 GND GND na L=0.5u W=4u M5 & P2 GND GND nb L=0.5u W=1u M6 & P2 GND GND nb L=0.5u W=1u .ENDS</pre>	<pre>.SUBCKT top P1 P2 & GND M1 & P1 GND GND na L=0.5u W=1u M2 & P1 GND GND na L=0.5u W=1u M3 & P1 GND GND na L=0.55u W=1u M4 & P1 GND GND na L=0.55u W=1u M5 & P2 GND GND nb L=0.5u W=1u M6 & P2 GND GND nb L=0.5u W=1u .ENDS</pre>
	

In the layout netlist, the na devices are merged into a single merged device with $W=4$ and $L=0.525$, as shown in [Figure 8-2](#). In the schematic, the na device has $L=0.5$, which might cause an LVS failure depending on the `check_property()` function tolerances.

Figure 8-2 Modified Netlists

Schematic (run_details/compare/top/sch.top)	Layout (run_details/compare/top/lay.top)
<pre>.SUBCKT top P1 P2 GND Z M1234 Z P1 GND GND na W=4u L=0.5u M_Parallel#0 Z P2 GND GND nb W=2u L=0.5u .ENDS</pre>	<pre>.SUBCKT top P1 P2 GND Z M_Parallel#0 Z P1 GND GND na W=4u L=0.525u M_Parallel#1 Z P2 GND GND nb W=2u L=0.5u .ENDS</pre>
	

Example of Parallel Merging of all NMOS Devices Except the nb Device

There are two methods for merging all of the devices except for the nb device. You can either list all of the devices by name in the `merge_parallel()` function. In this case, it is na.

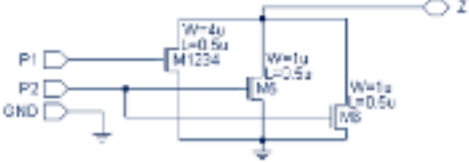
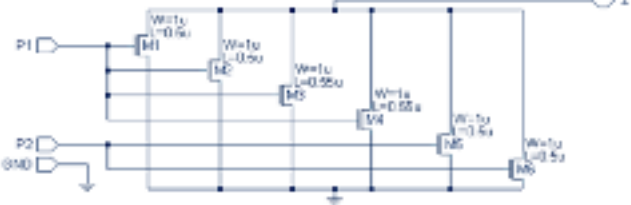
```
merge_parallel(compare_settings, NMOS, {"na"});
```

Or, you can leave out the name specification in the `merge_parallel()` function such that all devices are to be merged. Certain devices can be specified by name to not be merged using the `merge_parallel_off()` function.

```
merge_parallel(compare_settings, NMOS);
merge_parallel_off(compare_settings, NMOS, {"nb"});
```

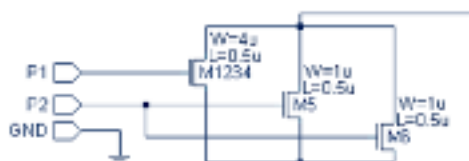
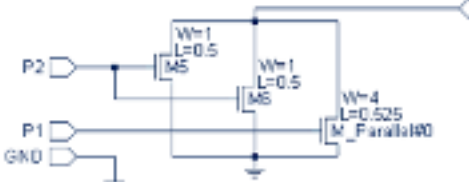
With either method, the resulting modified netlists are the same. The na devices are merged, and the nb devices are not merged, as shown in [Figure 8-3](#).

Figure 8-3 Input Netlists

Schematic	Layout
<pre>.SUBCKT top P1 P2 % GND M1234 2 P1 GND GND na L=0.5u W=4u M5 2 P2 GND GND nb L=0.5u W=1u M6 2 P2 GND GND nb L=0.5u W=1u .ENDS</pre>	<pre>.SUBCKT top P1 P2 % GND M1 2 P1 GND GND na L=0.5u W=1u M2 2 P1 GND GND na L=0.5u W=1u M3 2 P1 GND GND na L=0.55u W=1u M4 2 P1 GND GND na L=0.55u W=1u M5 2 P2 GND GND nb L=0.5u W=1u M6 2 P2 GND GND nb L=0.5u W=1u .ENDS</pre>
	

In the layout netlist, the na devices are merged and the nb devices are not merged, as shown in [Figure 8-4](#).

Figure 8-4 Modified Netlists

Schematic (run_details/compare/top/sch.top)	Layout (run_details/compare/top/lay.top)
<pre>.SUBCKT top P1 P2 % GND M1234 2 P1 GND GND na W=4u L=0.5u M5 2 P2 GND GND nb W=1u L=0.5u M6 2 P2 GND GND nb W=1u L=0.5u .ENDS</pre>	<pre>.SUBCKT top P1 P2 % GND M_Parallel#0 2 P1 GND GND na W=4u L=0.525u M5 2 P2 GND GND nb W=1u L=0.5u M6 2 P2 GND GND nb W=1u L=0.5u .ENDS</pre>
	

Merged Device Properties

Device properties for merged devices can be recalculated using functions other than the default functions by specifying the functions in the `property_functions` argument with the properties. These functions can be predefined functions or even user-defined functions.

The predefined functions are:

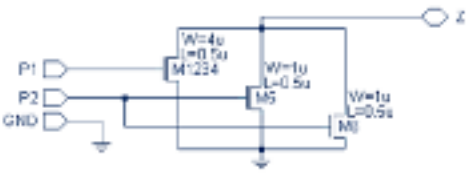
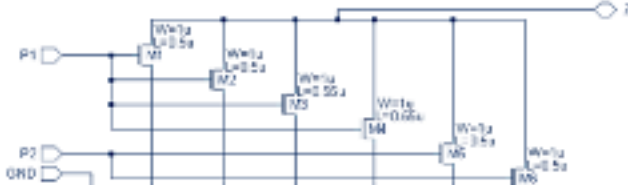
- `sum_merge_methods`. Specifies that the property values of the individual devices should be added together.
- `average_merge_method`. Specifies the averages of the property values of the individual devices.
- `min_merge_method`. Specifies the smallest property value of all individual devices.
- `max_merge_method`. Specifies the largest property value of all individual devices.

Example of Minimum Value of L for the Parallel Merged Device

Rather than calculating the value of L for the merged device as the average, use the minimum value of L in the original parallel connected group. The `min_merge_method` is specified in the `property_function` argument for L. See [Figure 8-5](#).

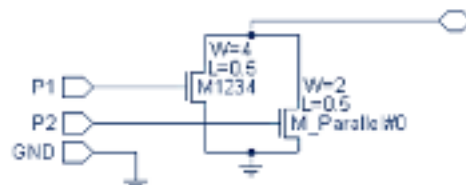
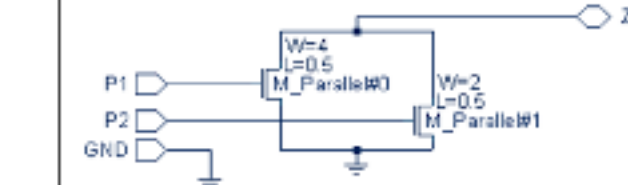
```
merge_parallel(compare_settings, NMOS,
    property_functions={
        {"min_merge_method", "L"},
        {"sum_merge_method", "W"}}
);
```

Figure 8-5 Input Netlists

Schematic	Layout
<pre>.SUBCKT top P1 P2 Z GND M1234 Z P1 GND GND na L=0.5u W=4u M5 Z P2 GND GND nb L=0.5u W=1u M6 Z P2 GND GND nb L=0.5u W=1u .ENDS</pre>	<pre>.SUBCKT top P1 P2 Z GND M1 Z P1 GND GND na L=0.5u W=1u M2 Z P1 GND GND na L=0.5u W=1u M3 Z P1 GND GND na L=0.55u W=1u M4 Z P1 GND GND na L=0.55u W=1u M5 Z P2 GND GND nb L=0.5u W=1u M6 Z P2 GND GND nb L=0.5u W=1u .ENDS</pre>
	

In the layout netlist, L=0.5 is the minimum instead of L=0.525 for the merged na device. See [Figure 8-6](#).

Figure 8-6 Modified Netlists

Schematic (run_details/compare/top/sch.top)	Layout (run_details/compare/top/lay.top)
<pre>.SUBCKT top P1 P2 Z GND M1234 Z P1 GND GND na W=4u L=0.5u M_Parallel#0 Z P2 GND GND nb W=2u L=0.5u .ENDS</pre>	<pre>.SUBCKT top P1 P2 Z GND M_Parallel#0 Z P1 GND GND na W=4u L=0.5u M_Parallel#1 Z P2 GND GND nb W=2u L=0.5u .ENDS</pre>
	

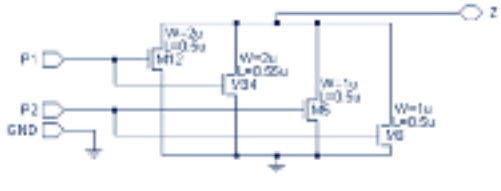
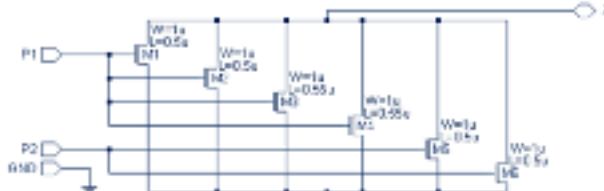
Excluding Merging by Tolerance

Merging is controlled based on property tolerances in potential merged device groups. In the following example, the L tolerance limits the merging of devices with lengths in a specific tolerance range. See [Figure 8-7](#).

Example of Merged Parallel Devices With a Tolerance of L = +/- 0.02

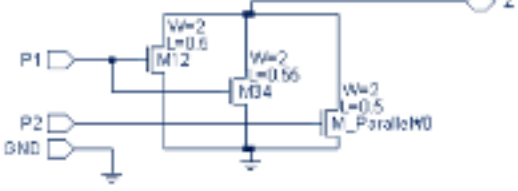
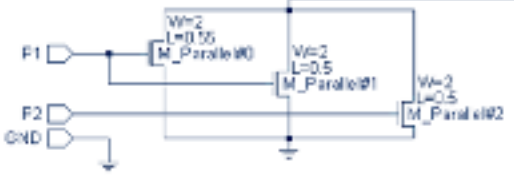
```
merge_parallel(compare_settings, NMOS,
    exclude_tolerances = [{"L", [-0.02,0.02], ABSOLUTE}]
);
```

Figure 8-7 Input Netlists

Schematic	Layout
<pre>.SUBCKT top P1 P2 3 GND M12 Z P1 GND GND na L=0.5u W=2u M34 Z P1 GND GND na L=0.55u W=2u M5 Z P2 GND GND nb L=0.5u W=1u M6 Z P2 GND GND nb L=0.5u W=1u .ENDS</pre> 	<pre>.SUBCKT top P1 P2 3 GND M1 Z P1 GND GND na L=0.5u W=1u M2 Z P1 GND GND na L=0.5u W=1u M3 Z P1 GND GND na L=0.55u W=1u M4 Z P1 GND GND na L=0.55u W=1u M5 Z P2 GND GND nb L=0.5u W=1u M6 Z P2 GND GND nb L=0.5u W=1u .ENDS</pre> 

In the layout netlist, there are two parallel merged na devices, one with W=2 L=0.5 and the other with W=2 L=0.55. The L properties have more than an 0.02 difference in length, as shown in [Figure 8-8](#).

Figure 8-8 Modified Netlists

Schematic (run_details/compare/top/sch.top)	Layout (run_details/compare/top/lay.top)
<pre> .SUBCKT top P1 P2 Z GND M12 Z P1 GND GND na W=2u L=0.5u M34 Z P1 GND GND na W=2u L=0.55u M_Parallel#0 Z P2 GND GND nb W=2u L=0.5u .ENDS </pre>	<pre> .SUBCKT top P1 P2 Z GND M_Parallel#1 Z P1 GND GND na W=2u L=0.5u M_Parallel#0 Z P1 GND GND na W=2u L=0.55u M_Parallel#2 Z P2 GND GND nb W=2u L=0.5u .ENDS </pre>
	

Custom Merging Equations for Device Properties

Custom functions can be defined for most of the netlist modification functions, such as the `merge_parallel()` and `merge_series()` functions. For the merging functions, custom functions can be defined for both post-merge property calculations as well as for merge exclusions. Custom functions that operate within the `compare()` function domain are callback functions and are declared with the entry point qualifier. The entrypoint functions are defined in a separate file that is referenced by the `user_functions_file` argument of the `compare()` function call. See [Figure 8-9](#) and [Figure 8-10](#).

Example of Custom Resistor Parallel Merged Equations

A custom function calculates the length and width values of parallel merged resistors when there are no common lengths or widths across all of the individual resistors in the parallel group.

The `res_merge_pq()` custom resistor function is called from the `merge_parallel()` function in `runset.rs`.

```
runset.rs
```

```
merge_parallel(compare_settings, RESISTOR, {"r_pq"},
property_functions = [{"res_merge_pq" /* W & L */ }]);
```

The `res_merge_pq()` function is defined in the `compare_entrypoint_functions.rs` user-defined function file that is referenced in the `compare()` in `runset.rs`.

`runset.rs`

```
compare(compare_settings, schematic_netlist_db, layout_netlist_db,
user_functions_file = "/path/to/file/compare_entrypoint_functions.rs",
... );
```

Prototypes for the intrinsic IC Validator utility functions used in the `res_merge_pq()` function, such as the `lvs_is_resistor()` and `lvs_sum_of_products()` functions, are in the `icv_compare.rh` header file, and they are defined in the in the Chapter 4, “Compare Utility Functions” in the *IC Validator Reference Manual*.

`compare_entrypoint_functions.rs`

```
#include "math.rh"
#include "icv_compare.rh"

res_merge_pq : entrypoint function ( void ) returning void
{
    device = lvs_current_device( );
    if ( lvs_is_resistor( device ) && lvs_is_parallel(device))
    {
        P : double = 0;
        Q : double = 0;
        Leff : double = 0;
        Weff : double = 0;

        result = lvs_sum_of_products( device, "W", "L", P );// Wi*Li
        result = lvs_sum_of_divisions( device, "W", "L", Q );// Wi/Li
        Leff = sqrt( P / Q );
        Weff = sqrt( P * Q );
        lvs_save_double_property( "L", Leff );
        lvs_save_double_property( "W", Weff );
    }
}
```

Figure 8-9 Input Netlists

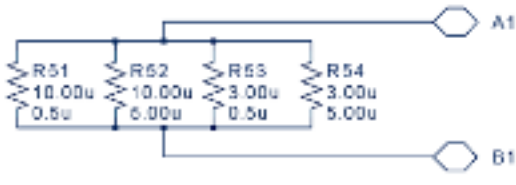
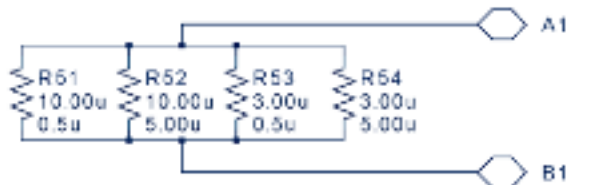


Schematic	Layout
R51 A1 B1 r_pq L=10.00u W=0.5u R52 A1 B1 r_pq L=10.00u W=5.00u R53 A1 B1 r_pq L=3.00u W=0.5u R54 A1 B1 r_pq L=3.00u W=5.00u	R51 A1 B1 r_pq L=10.00u W=0.5u R52 A1 B1 r_pq L=10.00u W=5.00u R53 A1 B1 r_pq L=3.00u W=0.5u R54 A1 B1 r_pq L=3.00u W=5.00u
	

Figure 8-10 Modified Netlists

Schematic	Layout
R_Parallel#0 A5 B5 r_pq W=13.054 L=5.478	R_Parallel#0 A5 B5 r_pq W=13.054 L=5.478
	

Series Merging Devices

The `merge_series()` function defines the criteria for merging devices in a series.

In addition to reducing the number of device instances, the `merge_series()` function creates instances with logically interchangeable ports, such as the gate pins on series merged MOS devices.

Syntax

```

merge_series(
    state                = compare_state,
    device_type          = NMOS | PMOS | NPN | PNP |
                        RESISTOR | CAPACITOR | INDUCTOR | GENERIC,
    device_names         = {"string", ...},                      //optional
    exclude_tolerances   = {{property      = "string",
                           tolerance       = doubleconstraint,
                           tolerance_type = RELATIVE | ABSOLUTE},
                           ...},                                //optional
    exclude_function     = "string",                             //optional
    property_functions    = {{property_function = "string",
                           property         = "string"}, ...},   //optional
    merge_connected_gates = true | false,                        //optional
    multiple_paths        = true | false,                        //optional
    equiv_cells           = {{schematic_cell = "string",
                           layout_cell      = "string"},
                           ...},                                //optional
    gendev_series_pins    = {pin_a = "string",
                           pin_b = "string"}                     //optional
);

```

The `merge_series_off()` function defines the criteria for excluding certain devices from being merged in series by type, device name, or equivalence cells.

Syntax

```

state                = compare_state,
device_type          = NMOS | PMOS | NPN | PNP | PN | NP |
                        RESISTOR | CAPACITOR | INDUCTOR | GENERIC,
device_names         = {"string", ...}                          //optional
equiv_cells           = {{schematic_cell = "string",
                           layout_cell    = "string"},
                           ...}                                //optional
);

```

Example of Series Merged MOS Devices Using Defaults

In this example, the series connected devices are transformed into a single, merged device instance with swappable gate connections. See [Figure 8-11](#).

```
merge_series(compare_settings, NMOS, {"na"});
```


Figure 8-11 Input Netlists

Schematic	Layout
M1 Z S1 VDD VDD pa L=0.5u W=1u M2 Z S2 VDD VDD pa L=0.5u W=1u M3 Z S1 Net1 GND na L=0.5u W=1u M4 Net1 S2 GND GND na L=0.5u W=1u	M1 Z S1 VDD VDD pa L=0.5u W=1u M2 Z S2 VDD VDD pa L=0.5u W=1u M3 Z S2 Net1 GND na L=0.5u W=1u M4 Net1 S1 GND GND na L=0.5u W=1u

Figure 8-12 Modified Netlists

Schematic	Layout
M1 Z S1 VDD VDD pa W=1u L=0.5u M2 Z S2 VDD VDD pa W=1u L=0.5u XSerChain#0 GND S1 S2 GND Z na=2	M1 Z S1 VDD VDD pa W=1u L=0.5u M2 Z S2 VDD VDD pa W=1u L=0.5u XSerChain#0 GND S2 S1 GND Z na=2

The gate connections for the series merged devices, G#0 and G#1, are logically equivalent and are swappable. This makes it possible to match the series transistors with different gate connection orders, as shown in [Figure 8-12](#).

Example of Series Merged Paths for MOS Devices

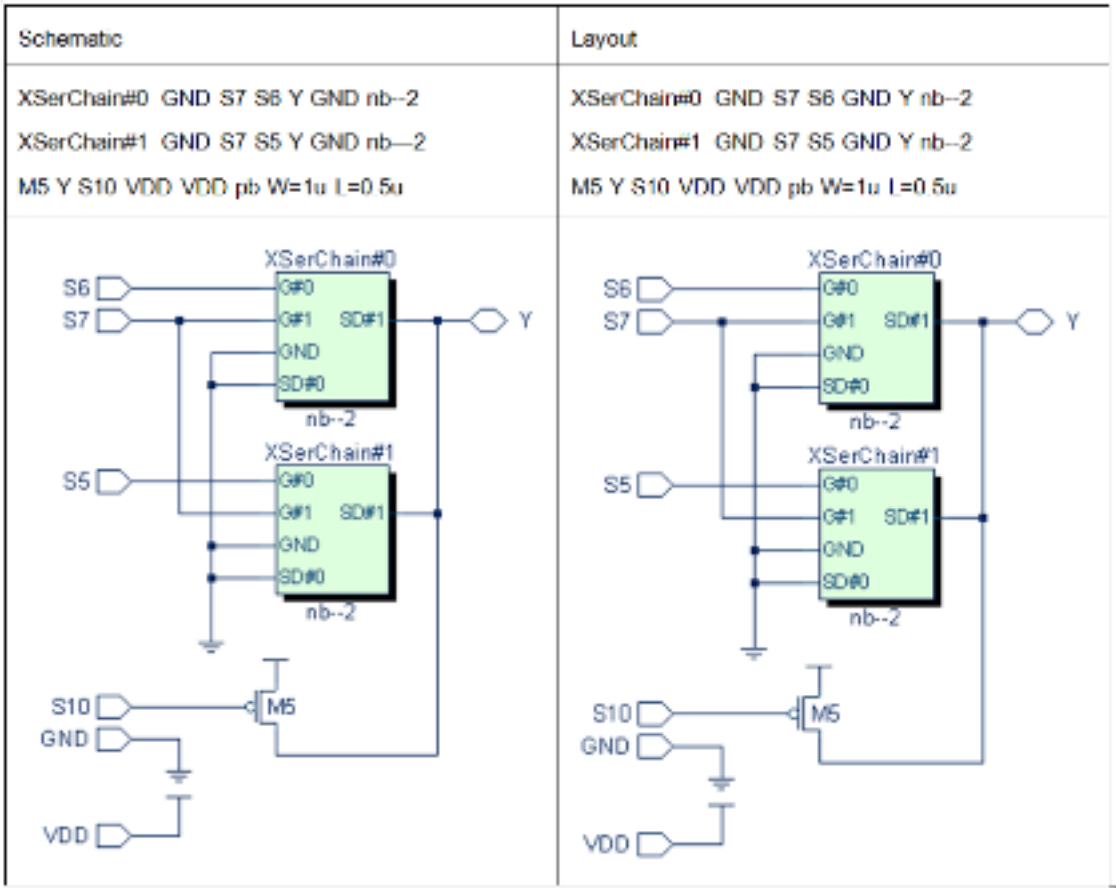
In this example, the `multiple_paths` argument expands stacks into multiple merged series devices, each with swappable ports. See [Figure 8-13](#).

```
merge_series(compare_settings, NMOS, {"nb"},
             multiple_paths = true);
```

Figure 8-13 Input Netlists

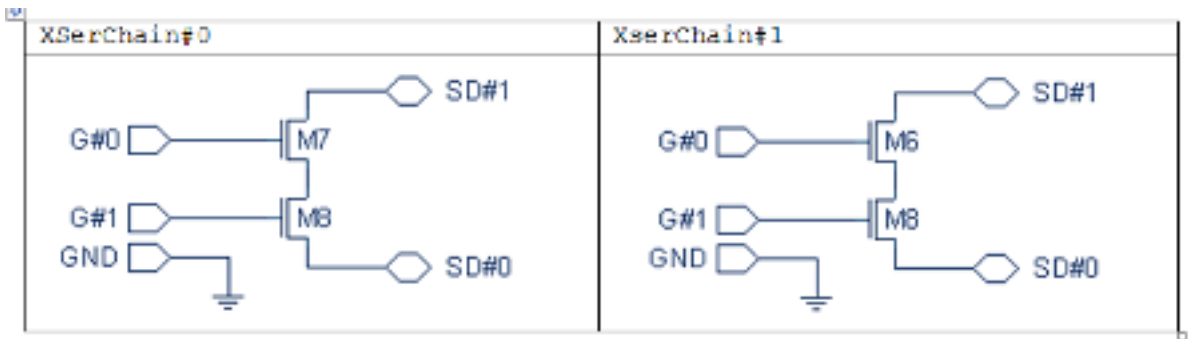
Schematic	Layout
<pre>M5 Y S10 VDD VDD pb L=0.5u W=1u M8 GND S5 Net4 GND nb L=0.5u W=1u M7 GND S6 Net4 GND nb L=0.5u W=1u M6 Net4 S7 Y GND nb L=0.5u W=1u</pre>	<pre>M5 Y S10 VDD VDD pb L=0.5u W=1u M8 Y S5 Net4 GND nb L=0.5u W=1u M7 Y S6 Net4 GND nb L=0.5u W=1u M6 Net4 S7 GND GND nb L=0.5u W=1u</pre>

Figure 8-14 Modified Netlists



The merged series instances in the modified netlists represent two devices in series. On the layout side, for example, XSerChain#0 contains M8 in series with M7, and XSerChain#1 contains M8 in series with M6, as shown in [Figure 8-14](#). Since the SD#0 and SD#1 ports are swappable, the merged series chains make it possible to match the different topologies for the Y and GND connections compared to the series chains in the schematic, as shown in [Figure 8-15](#).

Figure 8-15 Modified Layout Netlist Series Chains



Example of Series Merged Connected Gates for MOS Devices

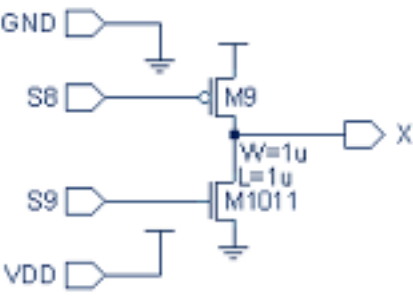
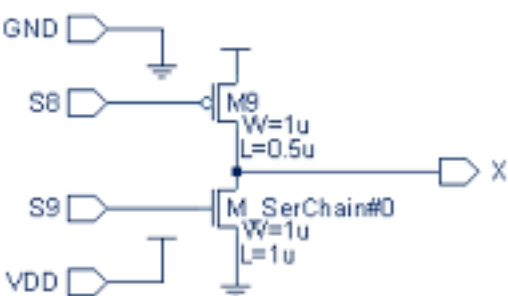
Series connected MOS devices with the same gate pin connection are reduced to a single, series merged device, therefore reducing the number of instances. See [Figure 8-16](#) and [Figure 8-17](#).

```
merge_series(compare_settings, NMOS, {"nc"},
             merge_connected_gates = true);
```

Figure 8-16 Input Netlists

Schematic	Layout
<div>M9 X S8 VDD VDD pb L=0.5u W=1u M1011 X S9 GND GND nc L=1u W=1u</div>	<div>M9 X S8 VDD VDD pb L=0.5u W=1u M10 X S9 Net5 GND nc L=0.5u W=1u M11 Net5 S9 GND GND nc L=0.5u W=1u</div>

Figure 8-17 Modified Netlists

Schematic	Layout
M1011 X S9 GND GND nc W=1u L=1u M9 X S8 VDD VDD pb Width=1u Length=0.5u	M_SerChain#0 X S9 GND GND nc W=1u L=1u M9 X S8 VDD VDD pb Width=1u Length=0.5u
	

Example of Series Merged Resistor Devices Using Defaults

Series connected resistors are reduced to a single, series merged device, therefore reducing the number of instances. See [Figure 8-18](#) and [Figure 8-19](#).

```
merge_series(compare_settings, RESISTOR);
```

Figure 8-18 Input Netlists


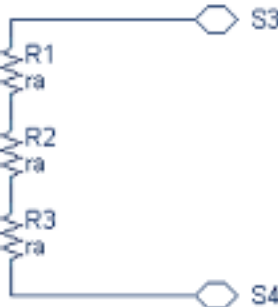


Schematic	Layout
R123 S3 S4 ra L=5.0u W=0.5u 12.0	R1 S3 Net2 ra L=2.0u W=0.5u 4.0 R2 Net2 Net3 ra L=2.0u W=0.5u 4.0 R3 Net3 S4 ra L=2.0u W=0.5u 4.0
	

Figure 8-19 Modified Netlists

Schematic	Layout
R123 S3 S4 ra Rval=12 W=0.5u L=6u	R_SerChain#0 S3 S4 ra Rval=12 W=0.5u L=6u
	

Parallel Device Merging Short Equivalent Nodes

The `short_equivalent_nodes()` function creates virtual connections (shorts) between electrically equivalent nodes in parallel-connected series transistor stacks. Connecting the electrically equivalent nodes makes it possible to merge the devices in parallel. The `short_equivalent_nodes()` function is typically used in conjunction with the `merge_parallel()` function.

Syntax

```
short_equivalent_nodes(
    state                = compare_state,
    device_type          = NMOS | PMOS,
    device_names         = {"string", ...}, //optional
    short_nodes          = SAME_DEVICE_NAME_ONLY | ANY_DEVICE_NAME, //optional
    width_ratio_tolerance = {tolerance      = doubleconstraint,
                             tolerance_type = RELATIVE | ABSOLUTE}, //optional
    equiv_cells          = [{schematic_cell = "string",
                             layout_cell    = "string"},
                             ...], //optional
    stack_type           = {SERIES_PARALLEL} //optional
);
```

The `short_equivalent_nodes_off()` function provides a mechanism for removing specific devices from consideration for `short_equivalent_nodes()` function based on device type, name, and the cell name that contains the device. See [Figure 8-20](#) and [Figure 8-22](#).

Syntax

```
short_equivalent_nodes_off(
    state                = compare_state,
```

```

device_type = NMOS | PMOS,
device_names = {"string", ...}, //optional
equiv_cells = {{schematic_cell = "string",
                 layout_cell    = "string"}, ...} //optional
);

```

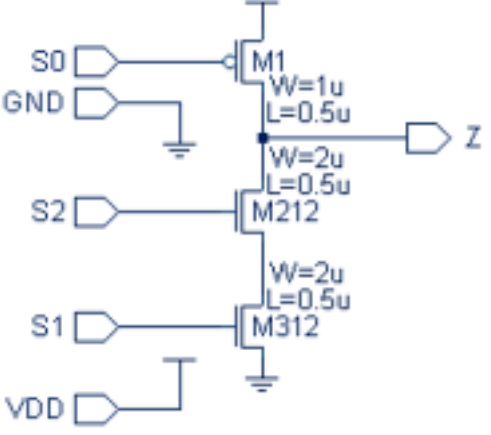
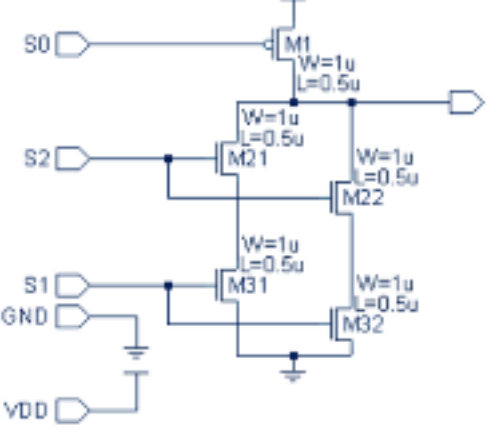
Example of Short Equivalent Nodes Using Defaults

```

short_equivalent_nodes(compare_settings, NMOS);
merge_parallel(compare_settings, NMOS);

```

Figure 8-20 Input Netlists

Schematic	Layout
M1 Z S0 VDD VDD pa L=0.5u W=1u M212 Z S2 Net12 GND na L=0.5u W=2u M312 Net12 S1 GND GND na L=0.5u W=2u	M1 Z S0 VDD VDD pa L=0.5u W=1u M21 Z S2 Net1 GND na L=0.5u W=1u M31 Net1 S1 GND GND na L=0.5u W=1u M22 Z S2 Net2 GND na L=0.5u W=1u M32 Net2 S1 GND GND na L=0.5u W=1u
	

The `short_equivalent_nodes()` function transforms the layout netlist by shorting the equivalent, intermediate node in the series transistor stacks, as shown in [Figure 8-21](#). After the equivalent node is shorted, the `merge_parallel()` function merge the devices. The M21 device is parallel merged with the M22 device, and the M31 device is parallel merged with the M32 device to form M_Parallel#0 and M_Parallel#1.

Figure 8-21 Short Equivalent Nodes

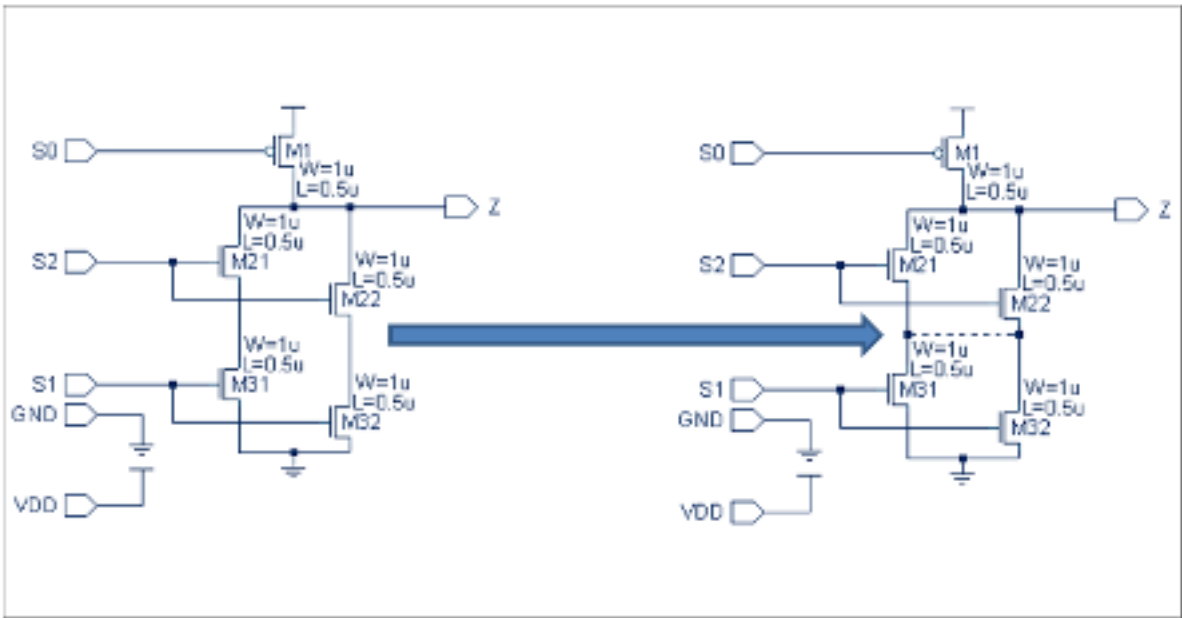


Figure 8-22 Modified Netlists

Schematic	Layout
<p>M212 Z S2 Net12 GND na W=2u L=0.5u</p> <p>M312 Net12 S1 GND GND na W=2u L=0.5u</p> <p>M1 Z S0 VDD VDD pa Width=1u L=0.5u</p>	<p>M_Parallel#0 Z S2 ShortedNet#0 GND na W=2u L=0.5u</p> <p>M_Parallel#1 ShortedNet#0 S1 GND GND na W=2u L=0.5u</p> <p>M1 Z S0 VDD VDD pa W=1u L=0.5u</p>

Filtering

The `filter()` function defines the criteria for removing devices from the netlist. Devices can be specified for filtering by type, name, and equivalence cell. Devices can be filtered using standard, predefined filtering criteria or by specifying user-defined filter functions. The connections that remain after the device is removed can be opened or shorted.

Syntax

```
filter(
    state                = compare_state,
    device_type          = NMOS | PMOS | NPN | PNP | PN | NP |
                          RESISTOR | CAPACITOR | INDUCTOR | GENERIC,
    device_name          = {"string", ...},                      //optional
    filter_options        = {option, ...},                      //optional
    filter_function       = "string",                            //optional
    equiv_cells          = {{schematic_cell = "string",
                           layout_cell    = "string"},
                           ...},                                //optional
    schematic_filter_options = {option, ...},                  //optional
    layout_filter_options  = {option, ...},                    //optional
    short_pins            = {"string", ...}                     //optional
);
```

The `filter_off()` function defines the criteria for excluding certain devices from being filtered by type, device name, or equivalence cells.

Syntax

```
filter_off(
    state                = compare_state,
    device_type          = NMOS | PMOS | NPN | PNP | PN | NP |
                          RESISTOR | CAPACITOR | INDUCTOR | GENERIC,
    device_names         = {"string", ...},                      //optional
    equiv_cells          = {{schematic_cell = "string",
                           layout_cell    = "string"},
                           ...},                                //optional
);
```

Filtering Standard Options

Standard filtering options are defined in Chapter 2 of the *IC Validator Reference Manual*. The standard filtering options are selected with an enumeration value that represents a particular filterable device configuration. The following excerpt from the Filtering Options table in the `filter()` function of the *IC Validator Reference Manual* shows some of the standard NMOS filter configuration definitions.

Table 8-1 Standard Filter Options

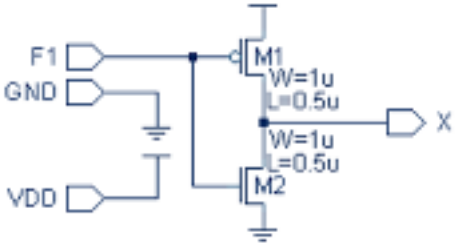
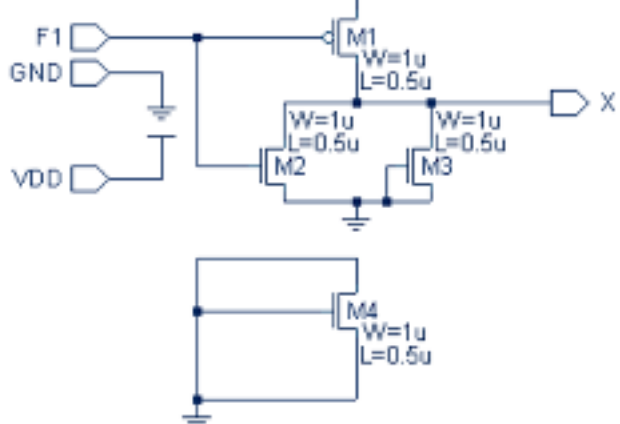
Option	Description
NMOS_1	Filters devices when gate, source, and drain pins are shorted.
NMOS_2	Filters devices when gate, source, and drain pins are floating.
NMOS_3	Filters devices when the gate pin is tied to ground.
NMOS_4	Filters devices when source and drain pins are tied to ground.
NMOS_5	Filters devices when source and drain pins are shorted.

Example of Standard Filtering for MOS Devices

```
filter(compare_settings, NMOS, {"na"}, {NMOS_1, NMOS_3});
```

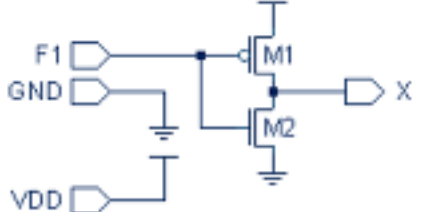
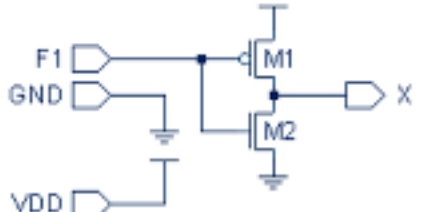
In [Figure 8-23](#), filterable devices with NMOS_1 and NMOS_3 are in the input layout netlist. Transistor M3 is filterable by NMOS_3. Transistor M4 is filterable by either NMOS_1 or NMOS_3.

Figure 8-23 Input Netlists

Schematic	Layout
M1 X F1 VDD VDD pa L=0.5u W=1u M2 X F1 GND GND na L=0.5u W=1u	M1 X F1 VDD VDD pa L=0.5u W=1u M2 X F1 GND GND na L=0.5u W=1u M3 X GND GND GND na L=0.5u W=1u M4 GND GND GND GND na L=0.5u W=1u
	

When the devices are filtered, the pins are left open, as shown in [Figure 8-24](#).

Figure 8-24 Modified Netlists

Schematic	Layout
M1 X F1 VDD VDD pa L=0.5u W=1u M2 X F1 GND GND na L=0.5u W=1u	M1 X F1 VDD VDD pa L=0.5u W=1u M2 X F1 GND GND na L=0.5u W=1u
	

Custom Filtering

Custom filtering can be accomplished by defining a custom filter function. As with all user-defined functions for compare-related functions, the filter functions are defined in a separate use-function file that is referenced in the `compare()` function.

Example of Custom Filtering for a Resistor

In the following example, resistors are filtered using a custom, user-defined function. Resistors are filtered if the device has the device name, `r_delete_me`, and is connected to a net called, `special_net`. The custom filter function is an `entrypoint` function. Therefore, it is defined in a separate `entrypoint` function file that is referenced by the `user_functions_file` argument of the `compare()` function. See [Figure 8-25](#) and [Figure 8-26](#).

```
runset.rs
```

```
compare(compare_settings, schematic_netlist_db, layout_netlist_db,
user_functions_file = "/path/to/file/compare_entrypoint_functions.rs",
... );
```

The `filter_res_by_net_and_device_name()` custom filter function is defined in the `entrypoint` function file as follows:

```
compare_entrypoint_functions.rs
```

```
#ifndef COMPARE_ENTRYPOINT_FUNCTIONS_RS
#define COMPARE_ENTRYPOINT_FUNCTIONS_RS
```

```
#include <icv_compare.rh>
#include <math.rh>
```

```
filter_res_by_net_and_device_name : entrypoint function (void) returning
void {
```

```
    dev_id : device = lvs_current_device();
```

```
    if(lvs_is_resistor(dev_id)){ // Is this a resistor?
```

```
        if((lvs_device_name(dev_id) == "r_delete_me")){ // Is this an //
            "r_delete_me" device?
```

```
            netA = lvs_net_name(lvs_get_device_nets_by_pin_name(dev_id, "A")); //
            Get the net names on the pins
```

```
            netB = lvs_net_name(lvs_get_device_nets_by_pin_name(dev_id, "B")); //
            if((netA == "special_net") || (netB == "special_net")){ //
                Is either net names "special_net"?

```

```
                lvs_remove_device(); // Delete/filter the device
            }
        }
    }
```

```
    }  
  }  
}  
#endif
```

The custom filter function is called in the `filter()` function in the main `runset.rs` before calling `compare()`, as follows:

```
runset.rs  
  
filter(compare_settings, RESISTOR,  
      filter_function = "filter_res_by_net_and_device_name"  
);  
  
compare(compare_settings, schematic_netlist_db, layout_netlist_db,  
user_functions_file = "/path/to/file/compare_entrypoint_functions.rs",  
... );
```

Figure 8-25 Input Netlists

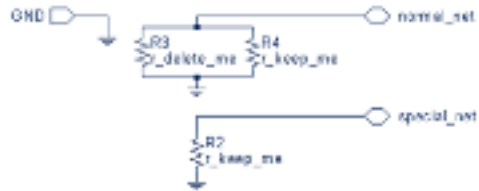
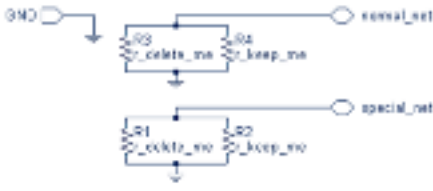
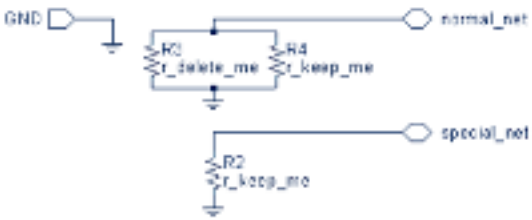
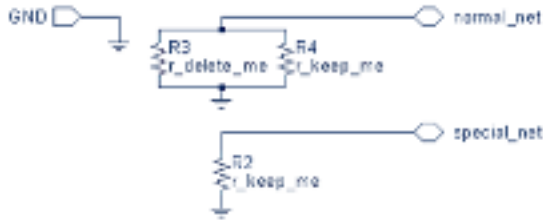
Schematic	Layout
<div>R2 special_net GND r_keep_me 1.0</div> <div>R3 normal_net GND r_delete_me 1.0</div> <div>R4 normal_net GND r_keep_me 1.0</div>	<div>R1 special_net GND r_delete_me 1.0</div> <div>R2 special_net GND r_keep_me 1.0</div> <div>R3 normal_net GND r_delete_me 1.0</div> <div>R4 normal_net GND r_keep_me 1.0</div>
	

Figure 8-26 Modified Netlists

Schematic	Layout
<div>R4 normal_net GND r_keep_me n='r_keep_me'</div> <div>R3 normal_net GND r_delete_me n='r_delete_me'</div> <div>R2 special_net GND r_keep_me n='r_keep_me'</div>	<div>R4 normal_net GND r_keep_me n='r_keep_me'</div> <div>R3 normal_net GND r_delete_me n='r_delete_me'</div> <div>R2 special_net GND r_keep_me n='r_keep_me'</div>
	

9

Table-Based Lookup Functionality

This chapter explains the table-based functionality for design for manufacturing (DFM).

The table-based functionality is described in the following sections:

- [Overview](#)
- [Using Table-Based Lookup](#)
- [Lookup Table Structure](#)
- [Table-Lookup Extraction Function](#)
- [Table-Based Lookup Example](#)

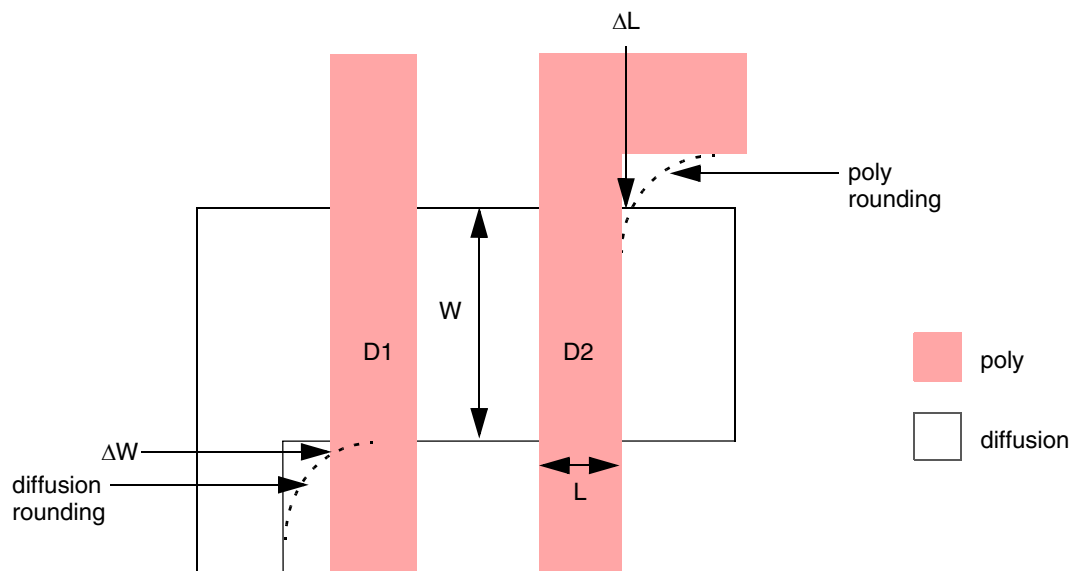
Overview

Table-based lookup functionality is a mechanism by which you import postmanufacturing effects into the design cycle for better simulation results. These postmanufacturing effects can be sampled, and then populated in lookup tables. The information in the lookup tables can be imported into the IC Validator flow by the remote functions called by the `nmos()` and `pmos()` functions.

One use of the table-based lookup functionality in DFM applications is to model the effective changes in a MOS device channel length and width due to postmanufacturing contour inaccuracies introduced by L-shaped poly or diffusion layers. In [Figure 9-1](#) the dotted line shows how the effective channel length and width can change after manufacturing from L-shaped poly and diffusion contours.

- W is the drawn channel width. L is the drawn channel length.
- ΔW and ΔL are the change in W and L after manufacturing.
- Effective channel width is $W' = W + \Delta W$. Effective channel length is $L' = L + \Delta L$.

Figure 9-1 Effective Channel Width and Length



You can save the width and length changes via a lookup table, and then use the `mos_get_dfm_double()` function in the remote functions called by the `nmos()` and `pmos()` functions to return the width and length changes at the runset level. You can then use these values to change the effective width and length for better simulation accuracy.

Using Table-Based Lookup

To use a lookup table:

1. Define a lookup table. See [“Lookup Table Structure”](#) for information about the table structure.
2. Define the path to the file that has the lookup table using the `dfm_files` argument of the `init_device_matrix()` function:

```
dfm_files = {"string", ...}
```

The IC Validator tool searches the files, in order, to find the table specified in the table-lookup extraction function. The table lookup file can be specified as either a relative or full path.

3. Use the table-lookup extraction function, `mos_get_dfm_double()`, within the remote functions called by the `nmos()` and `pmos()` functions. See [“Table-Lookup Extraction Function”](#) for information about the `mos_get_dfm_double()` function.

Lookup Table Structure

Within a table-lookup file:

- The `begin` statement is the first line in the table. The table name is defined on this line.
- The next line contains the table dimension following the number of data points for each dimension.
- Specify one input data row for each dimension. The data in each row must increase in value from left to right. The values are separated by a space.

Note:

Adjacent input values cannot be equivalent as this situation would cause divide by zero errors when interpolating to determine the return value.

- Specify return data.

The number of data points in each row of return values must equal the number of data points for the first dimension. In the following example, the first dimension (p) is 5; therefore, each row of return values must have five data points.

The number of rows containing return values must equal the product of the other dimensions. In the following example, the number of required rows of return values is $q \cdot r$ or $4 \cdot 2 = 8$. If a fourth dimension containing three input values was added, the number of required rows would be $q \cdot r \cdot s$ or $4 \cdot 2 \cdot 3 = 24$.

- The table can be of any dimension. The format requirements apply to all dimensions.

- Lines that start with # are comments. They are allowed anywhere.
- Empty lines are allowed anywhere.
- An `end` statement follows the last row of return values for any given table.
- Multiple tables can be defined, each with a unique name and delimited with `begin` and `end` statements.
- The table file can be encrypted using the IC Validator `pxlcrpt` executable.

```
% pxlcrpt plain_text.txt encrypted.txt
```

Figure 9-2 shows an example of a three-dimensional table structure:

Figure 9-2 Example Three-Dimensional Table Structure

```
# 3-D table example
begin rounding_effects_3d
3 5 4 2
0.00 0.12 0.16 0.18 0.22
0.00 0.80 0.90 0.96
1 2
0.00 0.00 0.00 0.00 0.00
0.00 0.01 0.02 0.04 0.08
0.00 0.02 0.04 0.08 0.10
0.00 0.04 0.08 0.10 0.16
0.00 0.00 0.00 0.00 0.00
0.00 0.02 0.03 0.05 0.09
0.00 0.03 0.05 0.09 0.11
0.00 0.05 0.09 0.11 0.17
end
```

← dimension p q r
 ← x1, x2, x3, x4, x5
 ← y1, y2, y3, y4
 ← z1, z2
 ← v_{y1}
 ← v_{y2}
 ← v_{y3}
 ← v_{y4} } v_{z1}
 ← v_{y1}
 ← v_{y2}
 ← v_{y3}
 ← v_{y4} } v_{z2}
 ↑ v_{x1} ↑ v_{x2} ↑ v_{x3} ↑ v_{x4} ↑ v_{x5}

Table-Lookup Extraction Function

The table-lookup extraction function returns a delta length (ΔL) or delta width (ΔW) based on the input table name and specified array values. You can use more than one extraction function in remote functions called by the `nmos()` and `pmos()` functions.

The table-lookup extraction function syntax is:

```
mos_get_dfm_double (
    table = "string",
    index = {double, ...}
);
```

table

Specifies the name used to locate the lookup table within the files specified by the `dfm_files` argument of the `init_device_matrix()` function. Table names are case-sensitive.

index

Values used in referencing the table to determine the return value. The input array size must match the table dimension. For example, an error message is reported if the lookup table is five-dimensional but the input array size is 6. The input array can contain numeric variables created within the remote functions called by the `nmos()` and `pmos()` functions.

Use the returned value to calculate the effective length (L') and width (W') within a device configuration function and netlist.

In the following example, the DFM input array contains the numeric values created from the `dev_touch_length()` function. The array is then used by the `mos_get_dfm_double()` function to find the delta width.

```
my_mos_prop_func : function(void) returning void
{
    delta_width : double = 0.0;
    DFM : list of double = {};

    src_drn = dev_pin("SRC");
    DFM.push_back (dev_touch_length(ox, src_drn));
    DFM.push_back (dev_touch_length(ox, src_drn));
    DFM.push_back (1.0);

    delta_width = mos_get_dfm_double("rounding_effects_3d", DFM);
    dev_save_double_properties({"delta_width", delta_width});
}

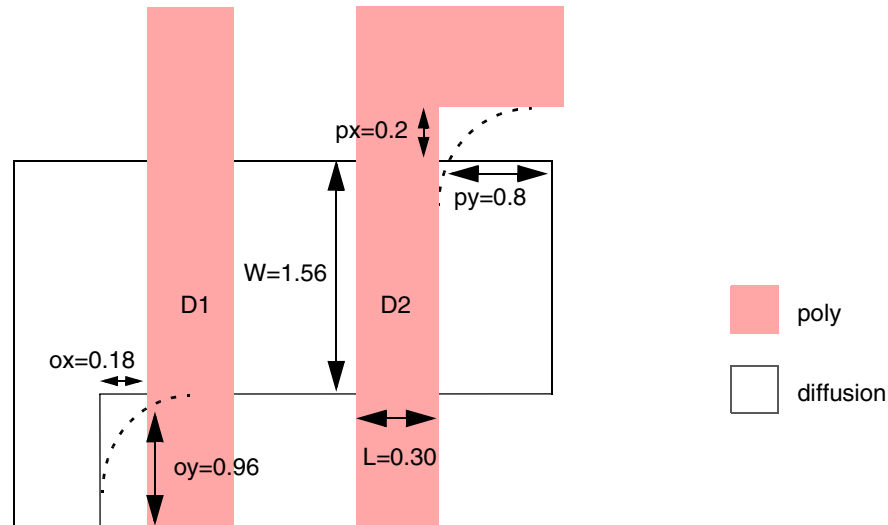
nmos(matrix,
    device_name = "nfet",
    drain = src_drn,
    gate = gate,
    source = src_drn,
    properties = {"delta_width", DOUBLE},
    property_function = my_mos_prop_func
);
```

Table-Based Lookup Example

In the example shown in [Figure 9-3](#), the IC Validator tool extracts the effective width and length of devices D1 and D2. IC Validator data creation functions extract polygons representing ox and oy edges (diffusion rounding effects), and px and py edges (poly

rounding effects). These layers are collected by device extraction and processed by layer-based functions to give numeric input values to the `mos_get_dfm_double()` function. The IC Validator tool returns a user-accessible numeric value from the table.

Figure 9-3 Example Effective Channel Width and Length



The IC Validator tool uses the following three-dimensional table to cross-reference input values x , y , and z for a return value V . The table contains five x input values, four y input values, and two z input values. The xy -dimension values are the extracted length measurements ox , oy , px , and py . The z -dimension value represents a flag to return values from either poly rounding or diffusion rounding.

```
# 3-D table example
begin rounding_effects_3d
3 5 4 2
0.00 0.12 0.16 0.18 0.22
0.00 0.80 0.90 0.96
1 2
0.00 0.00 0.00 0.00 0.00
0.00 0.01 0.02 0.04 0.08
0.00 0.02 0.04 0.08 0.10
0.00 0.04 0.08 0.10 0.16
0.00 0.00 0.00 0.00 0.00
0.00 0.02 0.03 0.05 0.09
0.00 0.03 0.05 0.09 0.11
0.00 0.05 0.09 0.11 0.17
end
```

For device D1, ox and oy are $0.18 \mu\text{m}$ and $0.96 \mu\text{m}$, respectively. The z flag is 1 for diffusion rounding. From the input table data shown earlier, $0.18 \mu\text{m}$ is x_4 , 0.96 is y_4 and 1 is z_1 . The IC Validator tool returns the value intersecting v_{x_4} , v_{y_4} , and v_{z_1} . This value is $0.1 \mu\text{m}$.

$$V(v_{x_4} \cap v_{y_4} \cap v_{z_1}) = 0.1$$

For device D2, px and py are 0.2 μm and 0.8 μm, respectively. The z flag is 2 for poly rounding. From the input table data, interpolation is required because 0.2 μm is between the input values x4 and x5. The value 0.8 μm, however, is y2. The z value is noted as z2. The IC Validator tool returns a value based on this interpolation equation:

$$V(v_x \cap v_{y2} \cap v_{z2}) = (v_x \cap v_{y2} \cap v_{z2}) + \frac{x - x_4}{x_5 - x_4} (v_{x5} - v_{x4})$$

$$V(v_x \cap v_{y2} \cap v_{z2}) = 0.05 + \frac{(0.20 - 0.18)}{(0.22 - 0.18)} (0.09 - 0.05)$$

$$V(v_x \cap v_{y2} \cap v_{z2}) = 0.07$$

In the following example, a three-dimensional table is used because different return values exist for diffusion and poly rounding. If a higher degree of table data hierarchy is needed, increase the table dimension. For example, a four-dimensional table might be needed if the return value differs based on whether the chip block was analog or digital.

The following NMOS extraction example demonstrates the runset usage of table-based extraction with a resultant layout netlist.

```
my_mos_prop_func : function(void) returning void
{
    DFM : list of double = {};
    EV_W1 = mos_width_1();
    EV_W2 = mos_width_2();
    EV_L1 = mos_length_1();
    EV_L2 = mos_length_2();

    py : polygon_set = dev_processing_layer("py");
    oy : polygon_set = dev_processing_layer("oy");
    poly_route : polygon_set = dev_processing_layer("poly_route");
    px : polygon_set = dev_processing_layer(px);
    ox : polygon_set = dev_processing_layer(ox);

    POx = dev_touch_length(px, poly_route);
    POy = dev_touch_length(py, poly_route);
    DIFFx = dev_touch_length(ox, poly_route);
    DIFFy = dev_touch_length(oy, poly_route);

    DFM.push_back(DIFFx);
    DFM.push_back(DIFFy);
    DFM.push_back(1.0);
    delta_width = mos_get_dfm_double("rounding_effects_3d", DFM);

    effective_width = ((EV_W1+EV_W2)/2) + delta_width;
    dev_save_double_properties({"effective_width", effective_width});

    DFM = {};
    DFM.push_back(POx);
    DFM.push_back(POy);
}
```

```

DFM.push_back(2.0);
delta_length = mos_get_dfm_double("rounding_effects_3d", DFM);

effective_length = ((EV_L1+EV_L2)/2) + delta_length;
dev_save_double_properties({"effective_length", effective_length});
}

nmos(matrix,
      device_name = "table_base_nmos",
      drain = src_drn,
      gate = gate,
      source = src_drn,
      processing_layer_hash = {
        "py" =>{py},
        "oy" =>{oy},
        "poly_route" =>{poly_route},
        "px" =>{px},
        "ox" =>{ox}
      },
      recognition_layer = gate_size,
      properties = {{"effective_width", DOUBLE},
                    {"effective_length", DOUBLE}},
      property_function = my_mos_prop_func
);

```