SKILL Development of Parameterized Cells

Version 5.1.41

Lab Manual October 15, 2004

Cadence Trademarks

Other Trademarks

Confidentiality Notice

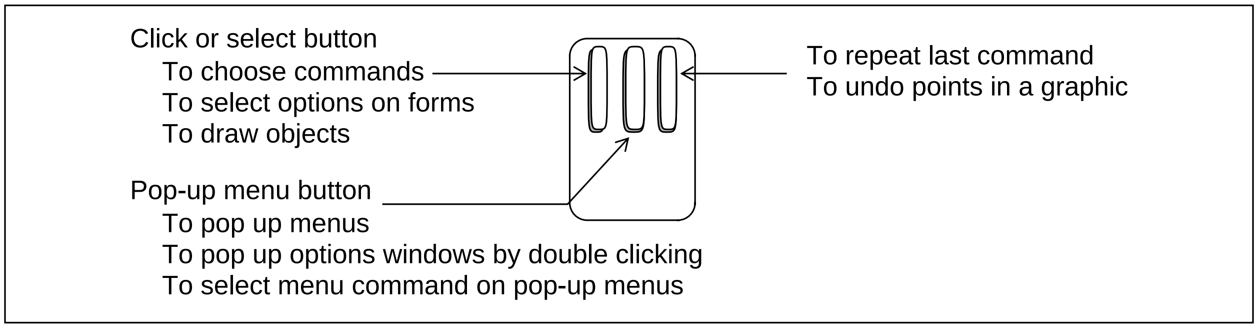
Table of Contents
SKILL Development of Parameterized Cells

Mouse Use and Terminology

Use the mouse to move the cursor on the screen, to make selections from a menu, and to draw.

Term	Action	Icon Example
click	Quickly press and release the specified mouse button. On menus and forms, you use the **left** mouse button most of the time.	Click **left**
double click	Rapidly press the specified mouse button twice.	2 Double click
Shift-click Control-click Shift-Control-click	Hold down the appropriate key or keys and click a specified mouse button.	**Shift** Shift- click **right**
draw through	Define a box by pressing the mouse button at one corner of the box, moving to the diagonally opposite corner of the box with the mouse button held down, and releasing the button.	Draw through
pop up	Press the **middle** mouse button.	Click **middle**
pull down	Move the mouse cursor to the menu name on the menu banner, press and hold the **left** mouse button, move the cursor down to highlight the menu selection, release the mouse button to execute the selection.	
Enter	Type a command in a window and press **Return** to execute the command.	
Select	Position the cursor over a command and press the **left** mouse button. Choose or pick are synonyms for select.	

The basic uses of mouse buttons are shown in this graphic.


Click or select button
To choose commands
To select options on forms
To draw objects
To repeat last command
To undo points in a graphic
Pop-up menu button
To pop up menus
To pop up options windows by double clicking
To select menu command on pop-up menus

Labs for Module 1

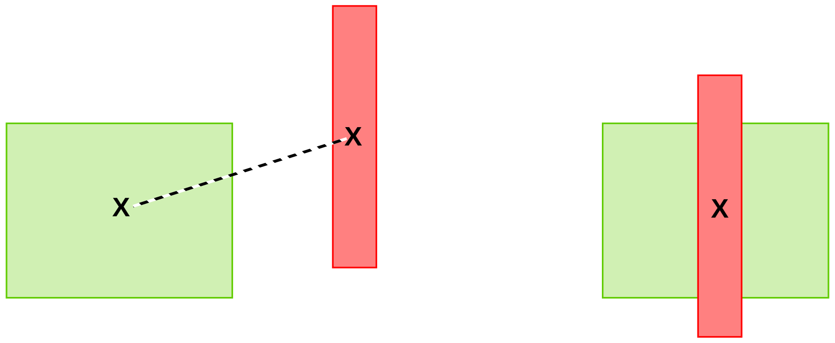
Introduction to Parameterized Cells

Lab 1-1 No Labs for this Module

End of Lab

Labs for Module 2

Introduction to Relative Object Design



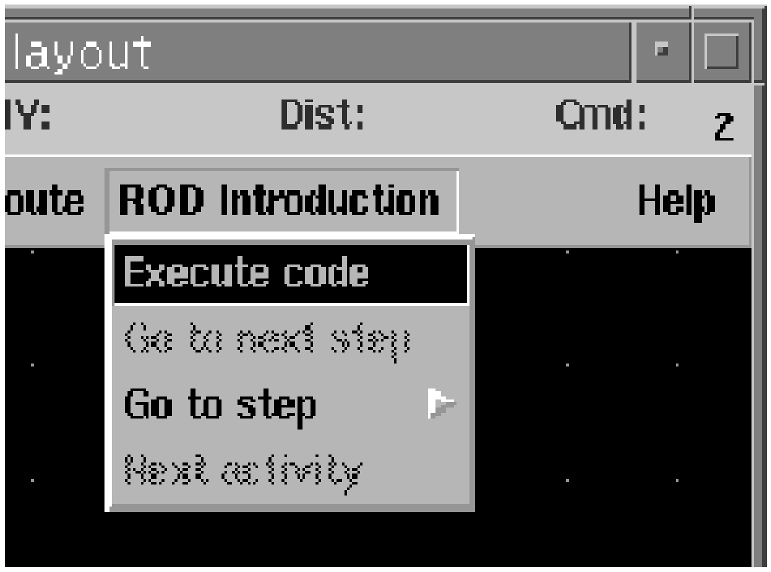
Before alignment After alignment

Before You Begin Module 2 Labs

The exercises in this lab have been specially streamlined to reduce the amount of typing required. When you start the Cadence® Design Framework II software, a Virtuoso® Layout Editor window appears. The window is associated with a sequence of steps called an *activity*. There are several activities in this lab. Each activity takes place in its own layout editor window.

Each step in an activity is associated with SKILL code that you execute. A text viewing window appears with the SKILL code for each step.

Each layout editor window contains a custom pull-down menu called **ROD Introduction**. This menu has several commands that allow you to navigate through the steps in the activity. The menu is depicted below:



Note: This menu is a custom feature. It is not a part of most Cadence courses.

The code viewing window allows you to study the SKILL code you execute in the current step. To execute the code and proceed to the next step, you choose from the **ROD Introduction** menu commands. Below is a description of the commands contained in this special menu:

- **Execute code**

  This command executes (loads) the code seen in the code viewing window into the Design Framework II session. You can choose this command only when you are at the beginning of a step in the activity.

- **Go to next step**

  This command closes the current code viewing window and prepares for the next step in the activity by opening a new code viewing window. You can choose this command only when you have executed the code for the current step and at least one step remains in the activity.

- **Go to step**

  This command is a slider menu that displays a sub-menu of the steps in the current activity. By selecting one of the steps listed in this sub-menu, you can go to any step in the activity. You can use this feature to repeat the current step or the entire activity.

Note: You can determine the name of the current step by looking in the banner of the code viewing window.

Note: You may see a dialog box asking if you wish to discard your edits when you use the **Go to step** menu command. Always answer **Yes** to this dialog box.

- **Next activity**

  This command closes the current code viewing window and the layout editor window for the current activity. The window for the next activity opens along with the code viewing window for the new activity's first step. You can choose this command only when you are at the end of the current activity and at least one more activity remains in the exercise sequence.

```
***                    ***
*** Activity Complete  ***
***                    ***
```

At this point, you can use **ROD Introduction—Go to step** to repeat any of the steps in the activity. If you do not wish to repeat any steps, choose **ROD Introduction—Next activity** to proceed to the next activity in the lab.

Lab 2-1 Creating Aligned Rectangles

Objective: This exercise demonstrates how to create simple rectangles and enforce alignment between them using relative object design constructs.

In this exercise, you will use relative object design (ROD) constructs to create several rectangles and enforce alignment between them. This demonstrates the application of two key ROD functions, rodCreateRect and rodAlign.

Start a Design Framework II Session

1. Enter these commands into a terminal window to start the Cadence Design Framework II session:

   ```
   cd ~/Skill_PCells_5_1_41/rodIntro
   layoutPlus &
   ```

   After a few moments the Command Interpreter Window (CIW) appears followed by a new (empty) layout editor window for the cellview, *example rectangles layout*.

   A text viewing window also appears with the SKILL code for the first step of the first activity.

Create Your First ROD Rectangle

In this section, you use *rodCreateRect* to create a ROD rectangle. You then examine the resulting ROD object using *rodGetObj*.

1. To execute the code for this step, select the menu command **ROD Introduction—Execute code**.

   This code assigns the current cellview object to a SKILL variable called *cv*. This variable will be used by the ROD functions called in this exercise.

   After you select the **Execute code** menu command, the source code displayed in the code viewing window is printed into the CIW indicating that this step is complete.

2. To verify this step, enter the following into the CIW:

   ```
   cv
   ```

   You see the printed representation of the cellview returned as the value for the variable. It will look similar to this:

   ```
   db:17099820
   ```

   Note: The number following the *db:* prefix will be different from one Design Framework II session to the next.

3. To proceed to the next step, choose the menu command **ROD Introduction—Go to next step**.

   Notice that another menu command is now enabled under the **ROD Introduction** menu. You can choose **ROD Introduction—Go to step** to select any step in the activity.

4. After you select the **Go to next step** command, a new window appears with the SKILL code for this step.

   Examine the code which creates a new ROD object, a simple rectangle, in the cellview window. Try to answer these questions as you read the code:

   a. Which ROD function is used here?

   b. How does the function know in which cellview to add the rectangle?

   c. How are the dimensions and location for the rectangle specified?

   d. How is the layer specified?

5. Select **ROD Introduction—Execute code** to execute the ROD function shown in the code viewing window.

   A rectangle appears in the layout editor window. Congratulations! You have created your first ROD object.

6. The resulting ROD rectangle is called *gate*. You now use its name to retrieve the rectangle’s ROD object. Enter the following into the CIW:

   ```
   rodGetObj(“gate” cv)
   ```

   You will see the ROD object’s printed representation, which looks something like this:

   ```
   rodObj:18972696
   ```

   Note: The number following the *rodObj:* prefix will be different from one Design Framework II session to the next.

7. Enter the following into the CIW:

   ```
   rodGetObj(“gate” cv)~>??
   ```

   Inspect the results to gain an understanding of the information contained in a ROD object.

Create Your First Alignment Between ROD Objects

You now create two more ROD rectangles and align them to the upper and lower extremes of the *gate* rectangle using *rodAlign*.

1. Select **ROD Introduction—Go to next step**.

   A new code viewing window appears.

2. Examine the code and try to answer these questions:

   a. What type of ROD objects will be created?

   b. Where will they appear and on which layer?

3. Select **ROD Introduction—Execute code** and observe what happens in the layout editor window.

   Are the results as you expected?

4. Select one of the newly created shapes. Now, move the shape to a new location in the layout editor window.

   What happens to the other ROD shapes?

5. Select **ROD Introduction—Go to next step**. Examine the code in the viewer window and try to discern what this ROD function will do.

6. Select **ROD Introduction—Execute code** and observe what happens to the shapes in the layout editor window.

7. Select the shape that moved. As in the previous step, move this shape to a new location in the layout editor window.

   What happens?

   The alignment created by *rodAlign* causes the two shapes to behave as a single entity. The smaller shape is now constantly aligned to a specific point on the original rectangle. Congratulations! You have created your first set of aligned ROD objects.

8. Select **ROD Introduction—Go to next step**. Examine the code in the viewer window.

   Which shape will move in this alignment? Which points on the two shapes will be aligned?

9. Select **ROD Introduction—Execute code** and observe what happens to the shapes in the layout editor window.

   Is this the expected result?

10. Now select any of the shapes and move it to a new location.

    What happens?

    You now have a rectangle of poly with a pin aligned to each end. These three ROD objects now move together as a single object.

Create a Collection of Prealigned Rectangles

The collection of rectangles in the previous section were created and aligned individually. You now use *rodCreatePath* to create a similar structure with one command.

1. Select **ROD Introduction—Go to next step**. Examine the code in the viewer window and try to answer these questions:

   a. What will be the name of the new object?

b. What portion of the call to *rodCreatePath* specifies the pins?

c. How is connectivity established for the pins?

2. Select **ROD Introduction—Execute code** and observe what happens in the layout editor window.

   Is the resulting object similar to the initial collection of rectangles? In what ways is it different?

3. Select the new object and move it to a different location.

   Does it behave as you expected?

4. Select **ROD Introduction—Go to next step**. Examine the code in the viewer window and try to determine what will happen to the entire collection of rectangles in the layout editor window.

5. Select **ROD Introduction—Execute code** and observe what happens.

6. Left-click on any rectangle in the editor window and then drag it as before.

   What happens? Is this what you expected?

   You now have two structures, each made of three rectangles. As any one (or more) of these objects is moved to a new location, both structures will relocate relative to the moved rectangle(s). Spacing and orientation of the structures will remain fixed as this happens.

7. You will see a message like this in the CIW:

```
***                     ***
*** Activity Complete   ***
***                     ***
```

   This marks the end of this portion of the lab. You may use the **ROD Introduction—Go to step** menu command to repeat any or all of this exercise if you desire.

   When you are ready, make sure you are at the last step in the activity (look for the message in the CIW shown above), and then choose **ROD Introduction—Next Activity**. This will iconify the current editor window and open an editor window for the next activity.

Important

You must choose ***ROD Introduction—Next Activity*** *before proceeding to the next activity.*

Important

Do not close (destroy) the editor window for any activity.

End of Lab

Lab 2-2 Using ROD in a SKILL Procedure

Objective: This exercise demonstrates how to use ROD constructs in a SKILL procedure to create simple physical structures.

In this exercise you define a SKILL procedure that uses a ROD construct to create a transistor structure. The procedure accepts an argument to specify the type of transistor to create (N or P). This procedure also demonstrates accessing technology file data (such as layer spacing rules) and uses this data in construction of the transistor device.

As in the previous exercise, you use commands under the special **ROD Introduction** menu in the current layout editor window to navigate through the constituent steps.

Define the Transistor Generation Procedure

You see a layout editor window for the cellview *example ptran layout* and a code viewing window for the first step of this activity. You will load the definition of the *SPCtran* procedure before using it to create transistors.

1. Examine the code in the code viewing window, but do not be intimidated by it—xyou are not required to understand exactly how it functions. Nonetheless, see if you can determine any of the following:

   a. What arguments are there to *SPCtran*?

   b. How is the type of diffusion specified?

   c. How many ROD functions are used?

   d. What information is obtained from the library technology data and how is the information used?

2. Select **ROD Introduction—Execute code**. This loads the definition of *SPCtran* into the current design framework session, but does not execute it.

Create a P-Type Transistor Using tran

You now call *SPCtran* to create a P-type transistor.

1. Select **ROD Introduction—Go to next step**. Again, as in the first activity, this step assigns the cellview object in the current window to a global variable called *cv*. This variable will be used as you call the *SPCtran* function.

2. Select **ROD Introduction—Execute code**.

3. Select **ROD Introduction—Go to next step** and examine the code in the viewing window.

   The call to *SPCtran* is quite simple. Notice the arguments passed to it.

   a. What is the first argument?

   b. What will be the transistor's width and length?

   c. Which argument determines the type (P or N) of the transistor?

4. Select **ROD Introduction—Execute code** and observe the results in the layout editor window.

5. Select the transistor and move it to a new location in the layout editor window. *What happens?*

6. In the banner of the layout editor window, select **Edit—Other—Rotate**. You are prompted to choose a reference point for the operation.

7. Left-click in the center of the transistor. As you move the mouse pointer, a highlighted outline will show you the amount of rotation.

8. Left-click again when the rotation indicates 90 degrees.

   What happens to the rectangles comprising the transistor?

9. Choose **Edit—Undo** to restore the transistor to its original orientation.

10. You will see a message like this in the CIW:

```
***                    ***
*** Activity Complete  ***
***                    ***
```

This marks the end of this portion of the lab. You may use the **ROD Introduction—Go to step** menu command to repeat any or all of this exercise if you desire.

When you are ready, make sure you are at the last step in the activity, and then choose **ROD Introduction—Next Activity**. This will iconify the current editor window and open an editor window for the next activity.

Important

*You must choose **ROD Introduction—Next Activity** before proceeding to the next activity.*

Important

Do not close (destroy) the editor window for any activity.

Create an N-Type Transistor Using tran

A layout editor window appears for the cellview *example ntran layout* along with a code viewing window for the first step of this activity. You use *SPCtran* to create an N-type transistor in this window.

1. A familiar operation appears in the code viewing window. Again, as in the previous activity, this step assigns the cellview object in the current window to a global variable called *cv*. This variable will be used as you call the *SPCtran* function.

2. Select **ROD Introduction—Execute code**.

3. Select **ROD Introduction—Go to next step**. Since the *SPCtran* function was defined in the previous activity, you can simply call it. Hence, the code for this step is a call to *SPCtran*. Examine this code and determine the following:

   a. What will be the dimensions of the N-type transistor?

   b. What determines the type of the transistor?

4. Select **ROD Introduction—Execute code** and observe the results in the layout editor window. You may experiment with moving and rotating this transistor as in the previous activity if you so desire.

5. You will see a message like this in the CIW:

   ```
   ***                    ***
   *** Activity Complete  ***
   ***                    ***
   ```

   This marks the end of this portion of the lab. You may use the **ROD Introduction—Go to step** menu command to repeat any or all of this exercise if you desire.

   When you are ready, make sure you are at the last step in the activity, and then choose **ROD Introduction—Next Activity**. This will iconify the current editor window and open an editor window for the next activity.

Important

*You must choose **ROD Introduction —Next Activity** before proceeding to the next activity.*

Important

Do not close (destroy) the editor window for any activity.

End of Lab

Lab 2-3 Using ROD to Create a Cell

Objective: This exercise demonstrates the use of ROD constructs in creating the physical design for a small cell.

In this exercise, you will see how to create a simple inverter using instances of the transistors created in the previous lab. The resulting inverter cell will contain hierarchy. You will see how ROD information can be accessed through that hierarchy and used in the creation and alignment of other ROD objects.

A layout editor window appears for the cellview *example inv layout* along with a code viewing window for the first step of this activity. The layout window shows some existing layout data including instances of the N- and P-type transistors created in the previous sections along with supply rails.

Examine the Existing Layout

1. Select **ROD Introduction—Execute code**.

   The code viewing window shows that you are assigning the current cellview object to the SKILL variable *cv*.

2. Select the upper transistor instance (the P-type transistor) in the layout editor window. Select **Edit—Properties** from the menu items in the window banner.

   A property editor form appears containing information specific to this instance. Notice that the name of the instance is *I1*.

3. With the property editor form open, select the lower transistor instance (the N-type transistor) in the layout editor window.

   Notice that the name of the instance is *I2*.

4. Click the **Cancel** button on the property editor form.

   The property editor form disappears.

Create and Connect the Input Pin

1. Select **ROD Introduction—Go to next step**. Examine the code in the viewing window and consider these questions:

   a. What function will the new structure serve?

   b. On which layers will the structure's constituent shapes appear?

   c. With which object will this structure be aligned?

2. Select **ROD Introduction—Execute code** and observe the results in the editor window.

   Notice that this multishape object was created with a single call to *rodCreatePath*.

3. Select **ROD Introduction—Go to next step**. Examine the code in the viewer window. Look specifically at the list of points in the first call to *rodCreatePath*.

   Do you notice any access to information below the current level of hierarchy in the first call to *rodGetObj*? What data is used here?

   In this call to *rodCreatePath*, the *upperCenter* point on instance *I2* and the *lowerCenter* point on instance *I1* are used to define the extent of the poly stripe connecting the transistors. This demonstrates access of ROD information through hierarchy.

4. Scan the remaining code in the viewing window.

   Do you see other uses of ROD information through the hierarchy? What purpose is served by the rodPointX and rodPointY functions?

5. Select **ROD Introduction—Execute code** and observe the results in the editor window.

Create and Connect the Output Pin

1. Select **ROD Introduction—Go to next step**. Examine the code in the viewer window and answer these questions:

   a. Where in the code is information from the hierarchy obtained and used?

b. What purpose is served by the *rodAddToX* function?

c. Which new object will become the output pin for the inverter?

2. Select **ROD Introduction—Execute code** and observe the results in the editor window.

   Congratulations! Your inverter cell is complete.

3. You will see a message like this in the CIW:

```
***                        ***
*** Activity Complete  ***
***                        ***
```

   This marks the end of this portion of the lab. You may use the **ROD Introduction—Go to step** menu command to repeat any or all of this exercise if you desire.

   When you are ready, make sure you are at the last step in the activity, and then choose **ROD Introduction—Next Activity**. This will iconify the current editor window and open an editor window for the next activity.

Important

*You must choose **ROD Introduction—Next Activity** before proceeding to the next activity.*

Important

Do not close (destroy) the editor window for any activity.

End of Lab

Lab 2-4 Creating ROD Objects Interactively

Objective: This exercise demonstrates how to create ROD objects interactively within the layout editor.

In this activity, you load a simple SKILL program for interactively creating a guard-ring structure. The function employs one call to *rodCreatePath* to perform this otherwise tedious task. You then use various features of the layout editor to create other types of ROD objects. Finally, you tour the multipart path user interface available with Virtuoso XL.

A layout editor window appears for the cellview *example interactive layout* along with a code viewing window for the first step of this activity. In the code viewing window you find the definition of a SKILL function called *SPCdigitizeGuardring*, a SKILL function called *SPCcreateGuardring*, and a key binding for the **F10** key.

Load the Guard-ring Code

In this section, you load SKILL code that allows you to interactively define a guard-ring structure. This demonstrates how you can build custom interactive features for creating ROD structures in your design flow.

1. Examine the code for *SPCdigitizeGuardring* and attempt to determine its purpose.

2. Examine the code for *SPCcreateGuardring* and try to discern the following:

   a. Is technology information used?

   b. On which layer is the *master* path drawn?

   c. On which layer is the enclosed subpath drawn?

   d. Which part of the guard-ring structure does the *subRect* keyword argument specify? Does this create a single polygon?

3. Select **ROD Introduction—Execute code**.

   The functions for digitizing and building the guard-ring are now loaded and the digitizing function is bound to the **F10** key.

Digitize a Guard-ring

You now create a guard-ring by entering points in the layout editor window. You then experiment with the **Chop** command to determine the guard-ring's behavior.

1. In the layout editor window, use the **F10** key and the mouse to digitize the guard-ring structure by clicking at various points of your choosing. Use the **Back Space** key if you wish to undo the most recent point. Click twice at the same point or press the **Return** key to complete the structure.

   Does the guard-ring appear as you expected?

2. Select **Edit—Stretch** in the layout editor window.

3. Use the **F4** key to toggle to partial select mode. Notice the *(P)* in the window banner rather than *(F)*.

4. Click on one end of the guard-ring and drag that end to a new location.

   Notice how the contact arrays and the metal and diffusion paths adjust accordingly.

5. Press the **Escape** (or **Esc**) key.

   This action cancels the **Stretch** mode initiated in step 2.

6. Select the new guard-ring object by clicking on it.

7. Select **Edit—Other—Chop** in the layout editor window.

8. Click the mouse near a portion of the guard-ring and drag the selection rectangle through the guard-ring.

9. Release the mouse button.

   What happened?

   Look at the code for *createGuardring* and try to determine which keyword arguments control the behavior of the metal and diffusion layers.

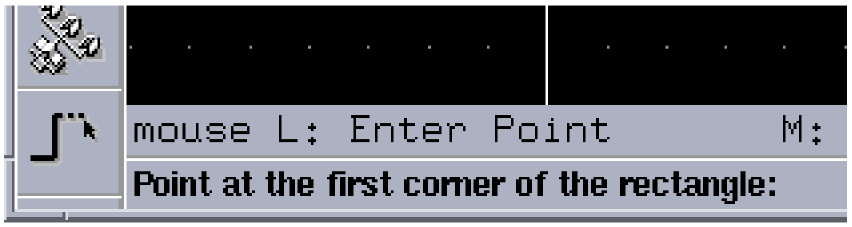
10. If you desire, you may use the **F10** key and mouse to create more guard-rings.

Create a ROD Rectangle Interactively

In this section, as you create a rectangle you use the rectangle option form to designate it as a ROD object. You then use the **ROD** category of the property editor to examine the resulting object.

1. Select **Create—Rectangle** from the menus in the layout editor window.

   Look at the lower left region of the layout editor window. Notice that the system is prompting you to enter the first corner point of the new rectangle:

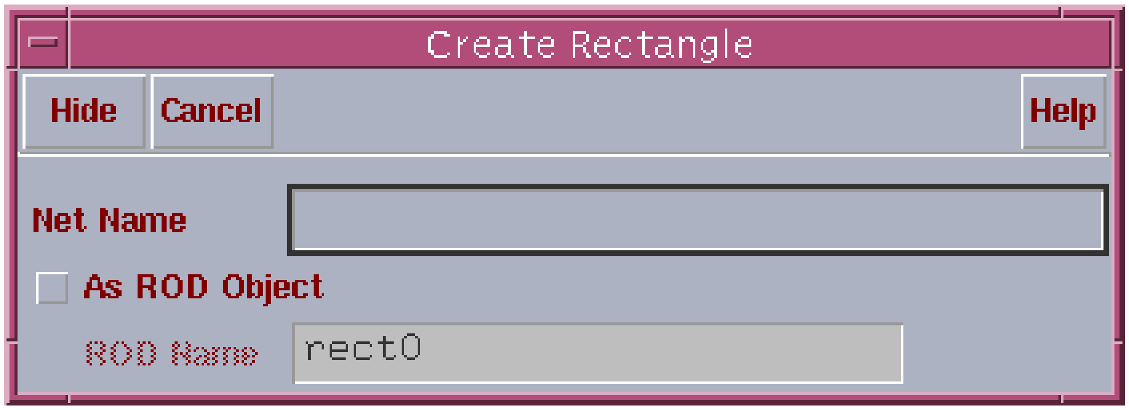


mouse L: Enter Point M:

Point at the first corner of the rectangle:

2. Press the **F3** key to display the object creation options form.

   The options form appears as shown below:



Create Rectangle

Net Name

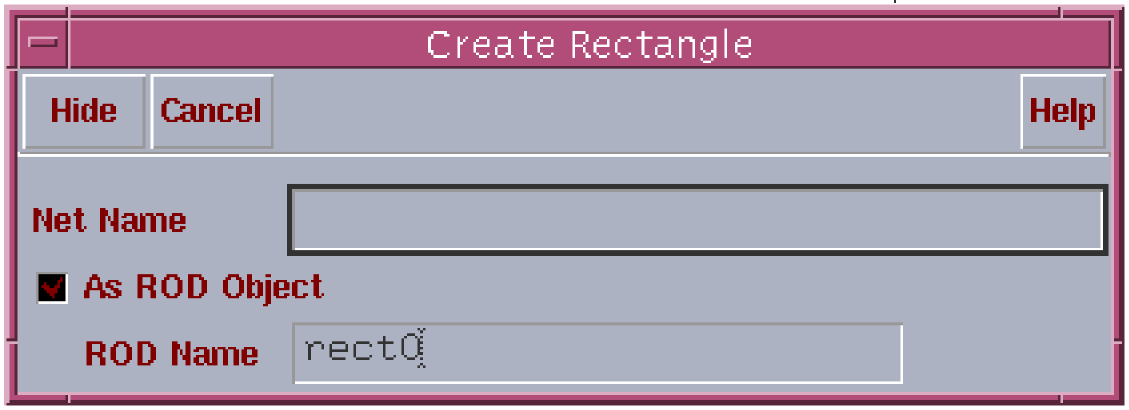
As ROD Object

ROD Name rect0

Notice the controls for specifying the rectangle as a ROD object and for naming the object in addition to the **Net Name** field.

3. Click on the **As ROD Object** option.

   Notice that the **ROD Name** field becomes editable as shown below:



Net Name

✓ As ROD Object

ROD Name rect0

4. If you desire, you may specify a name for the new rectangle other than the default shown.

5. Click and drag in the layout editor window to specify the extent of the new rectangle. Choose dimensions that suit you - the actual size of the rectangle does not matter. Release the mouse button.

   A rectangle is created in the layout editor window.

6. Press the **Escape** (or **Esc**) key.

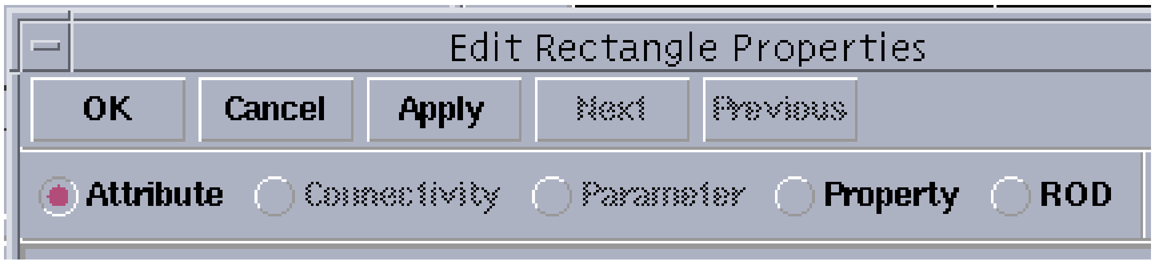
   This cancels the rectangle entry mode. The rectangle options form disappears.

7. Click on the new rectangle to select it.

   The rectangle appears highlighted in the layout editor window.

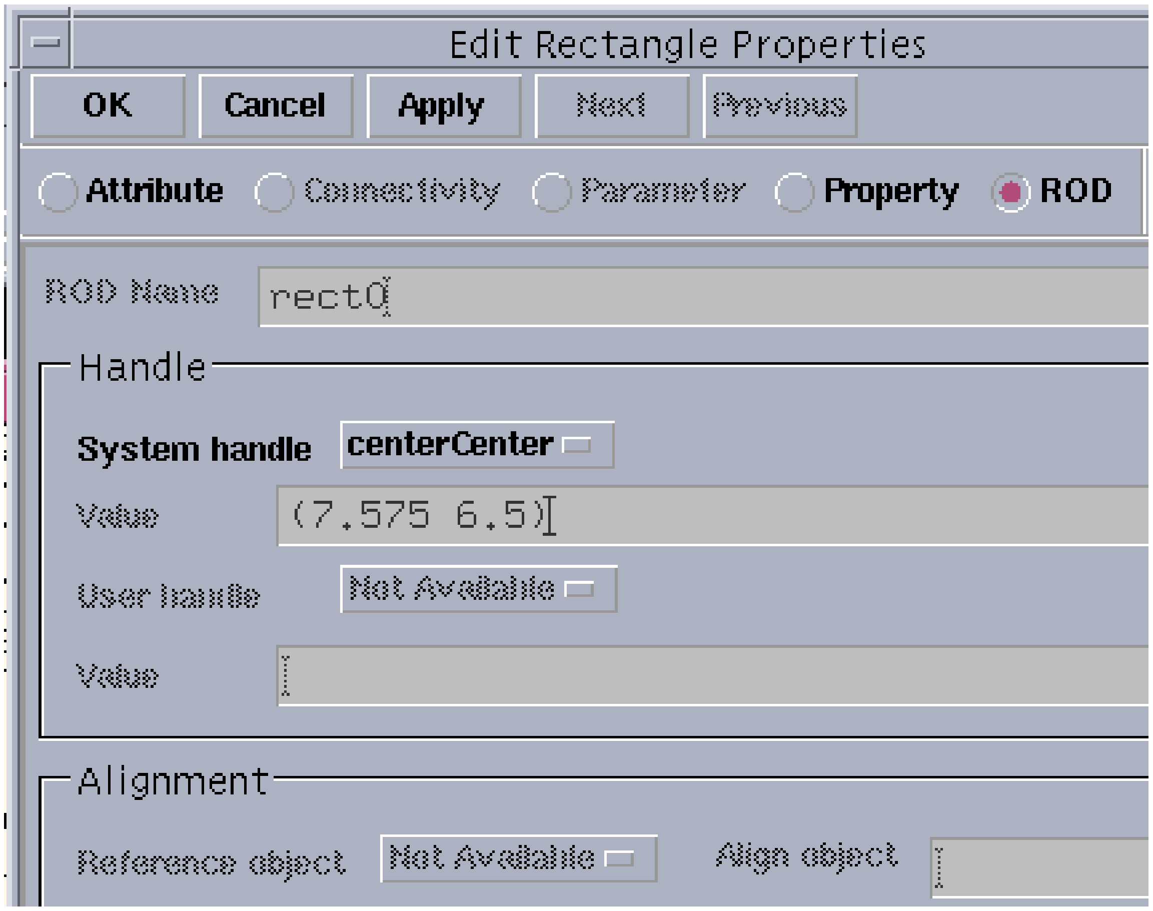
8. Select **Edit—Properties** from the layout editor window menus.

   A property editor form appears containing information about the new rectangle. Notice that the property editor category called **ROD** is selectable:



9. Click on the ROD category in the property editor form.

   ROD information now appears in the property editor form:



Attribute Connectivity Parameter Property [✓ ROD]

ROD Name rect0

System handle centerCenter

Value (7.575 6.5)

User handle Not Available

Value

Reference object Not Available

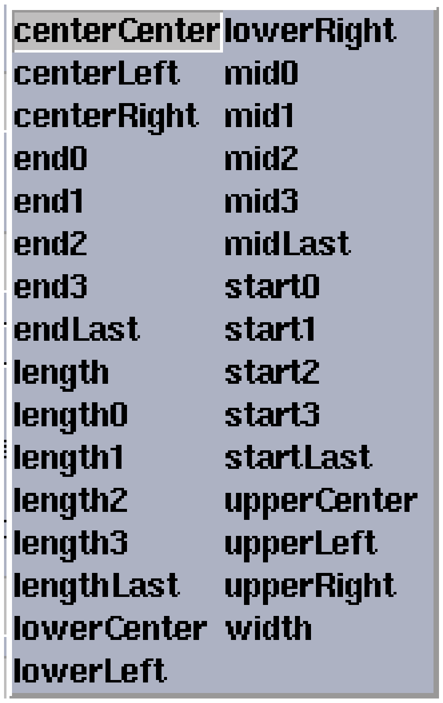
Align object

   Notice the wealth of information accessible from this category of the property editor form. In addition to seeing the object's ROD, you can examine the names and values of the object's handles and any alignment information it may have.

10. Verify that the name specified in the **ROD Name** field matches the name you specified in the rectangle options form.

11. Click on the **System handle** cyclic field.

    A list of system handle names appears:



    When you choose one of these handle names, its value appears in the **Value** field below the **System handle** field.

12. Select several handles from the **System handle** field and examine their values.

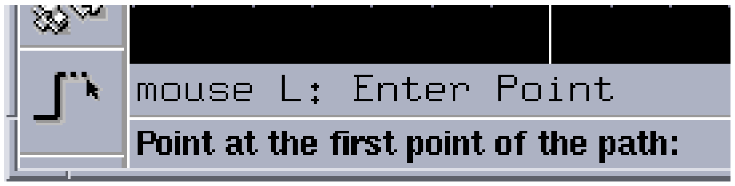
    You may use the mouse-pointer or the **Attribute** category of the property editor to verify these handle values.

Create a ROD Path Interactively

You use the path options form to create a single-layer path that is a ROD object.

1. Select **Create—Path** from the menus in the layout editor window.

   Look at the lower-left region of the layout editor window. Notice that the system is prompting you to enter the first point of the new path:

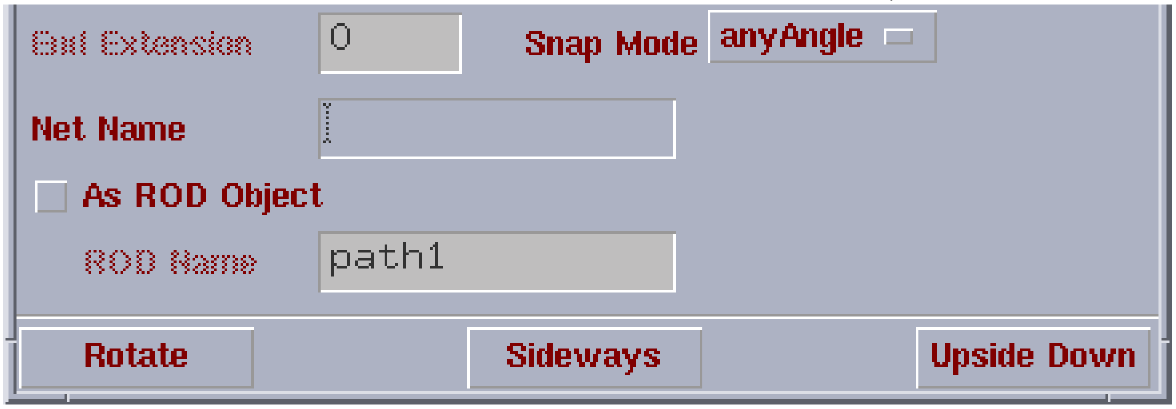
   

   mouse L: Enter Point

   Point at the first point of the path:

2. Press the **F3** key to display the object creation options form.

   The lower portion of the options form appears as shown below:

   Exit Extension 0

   Snap Mode anyAngle

   Net Name

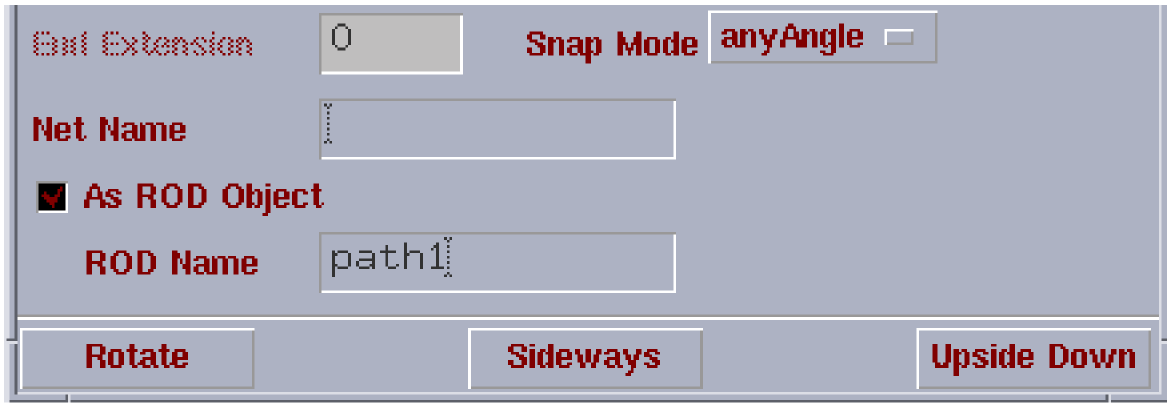
   As ROD Object

   ROD Name path1

   Notice the controls for specifying the path as a ROD object and for naming the object.

3. Click on the **As ROD Object** option.

   Notice that the **ROD Name** field becomes editable as shown below:



Ext Extension: 0

Snap Mode: anyAngle

Net Name:

[✓] As ROD Object

ROD Name: path1

4. If you desire, you may specify a name for the new path other than the default shown.

5. Click on a series of points in the layout editor window to specify the extent of the new path. Choose points that suit you—the actual shape of the path does not matter. Use the **Back Space** key if you wish to undo the most recent point. Click twice at the same point or press the **Return** key to complete the path.

   A path is created in the layout editor window along the points you specified.

6. Press the **Escape** (or **Esc**) key.

   This cancels the path entry mode, and the path options form disappears.

7. Click on the new path to select it.

   The path appears highlighted in the layout editor window.

8. As with the rectangle before, use the **ROD** category of the property editor to inspect the name and various handles of the new path.

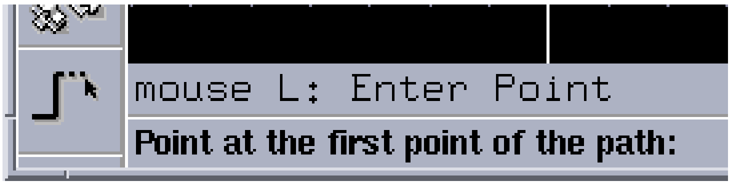
Create a ROD Multipart Path Interactively (Virtuoso XL Only)

In this section, you use a series of forms for interactively creating ROD multipart paths. This differs from the path created at the beginning of this activity in that you need not write SKILL code to specify the multipart path. Furthermore, you can save the architecture of the path you specify as a *template* for paths you create within the current session and within future sessions.

Remember that this interface is only available with the Virtuoso XL option.

1. Select **Create—Multipart Path** from the menus in the layout editor window.

   Look at the lower-left region of the layout editor window. Notice that the system is prompting you to enter the first point of the new path:

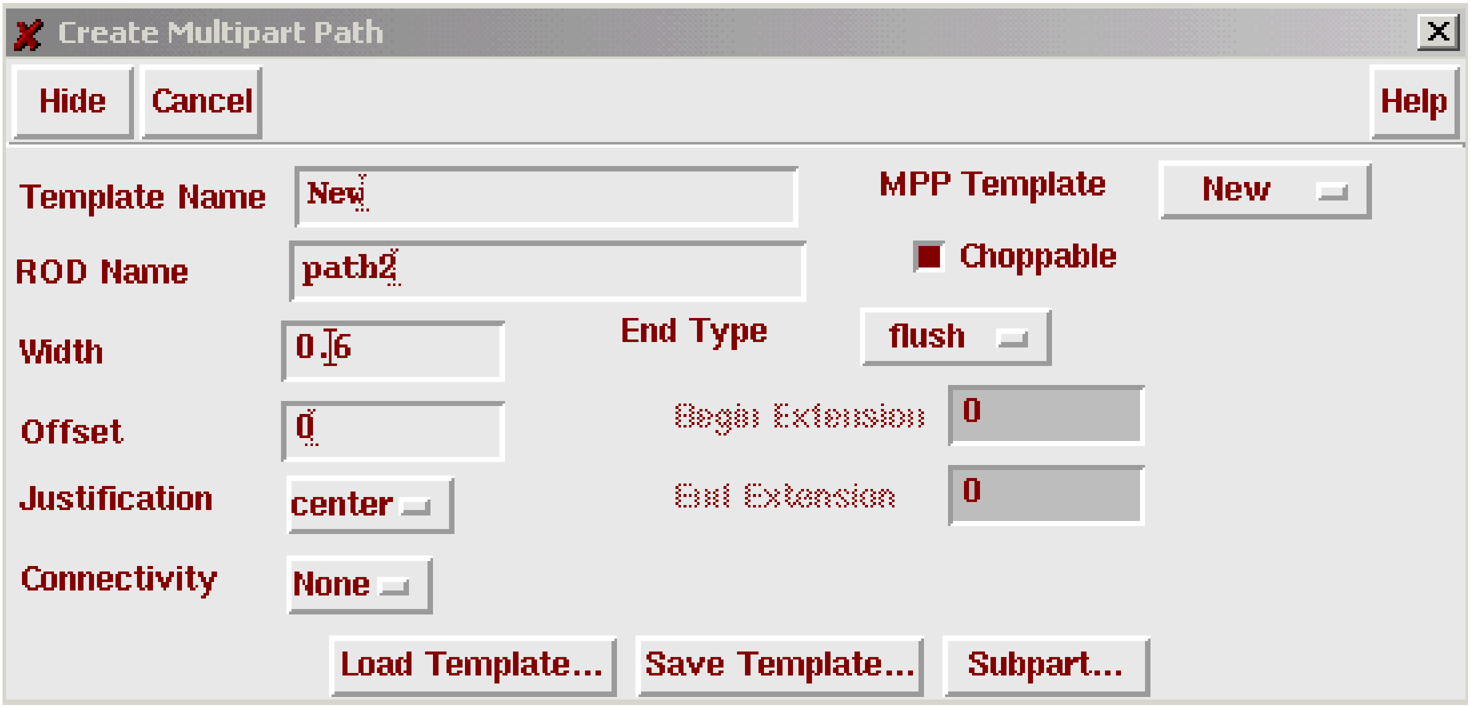


mouse L: Enter Point

Point at the first point of the path:

2. Press the **F3** key to display the object creation options form.

   The option form appears as shown below:



Create Multipart Path

Hide | Cancel | Help

Template Name	New	MPP Template	New
ROD Name	path2	■ Choppable	
Width	0.6	End Type	flush
Offset	0	Begin Extension	0
Justification	center	End Extension	0
Connectivity	None		

Load Template... | Save Template... | Subpart...

   This form contains options for the *master path* which defines the overall shape of the multipart path.

   Notice the first field called **MPP Template**. You use this field to select from existing multipart path templates. Because no templates exist for the current technology, only the entry called **New** is present for this cyclical field.

   Notice also the **Save Template** button near the lower right corner of the form. You use this button to store your multipart path templates to the current technology information.

   You use the **Subpart** button just to the right of the **Save Template** button to specify paths subordinate to the master path.

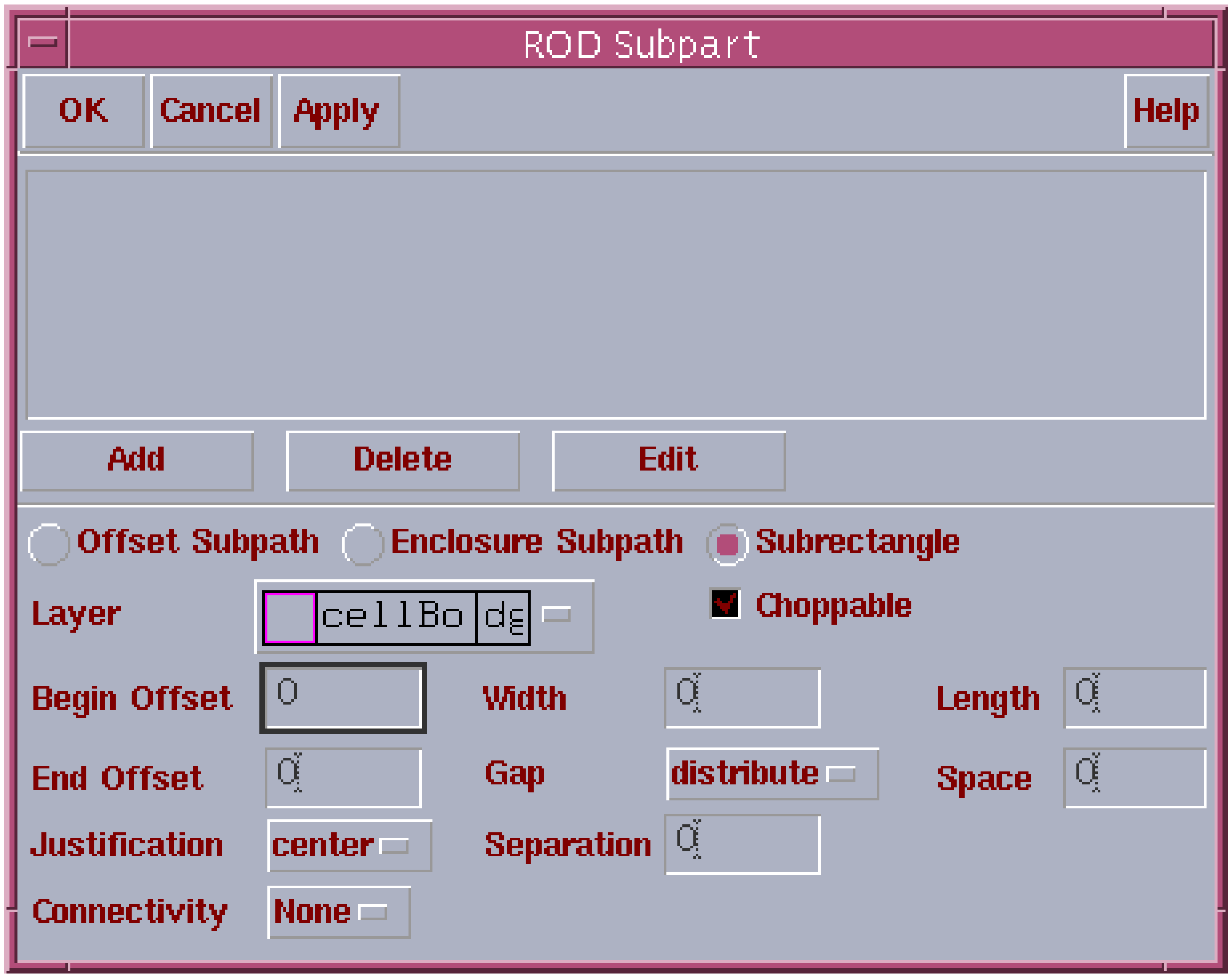
3. Change the following field to reflect the value listed below:

```
Width          1.4
```

   This sets the width of the master path to 1.4 units.

4. Click on the **Subpart** button.

   A form appears allowing you to create subpaths for this multipart path:



Offset Subpath [✓ Subrectangle] Enclosure Subpath

Layer: cellBo dg

Choppable ✓

Begin Offset	0	Width	0	Length	0
End Offset	0	Gap	distribute	Space	0
Justification	center	Separation	0		
Connectivity	None				

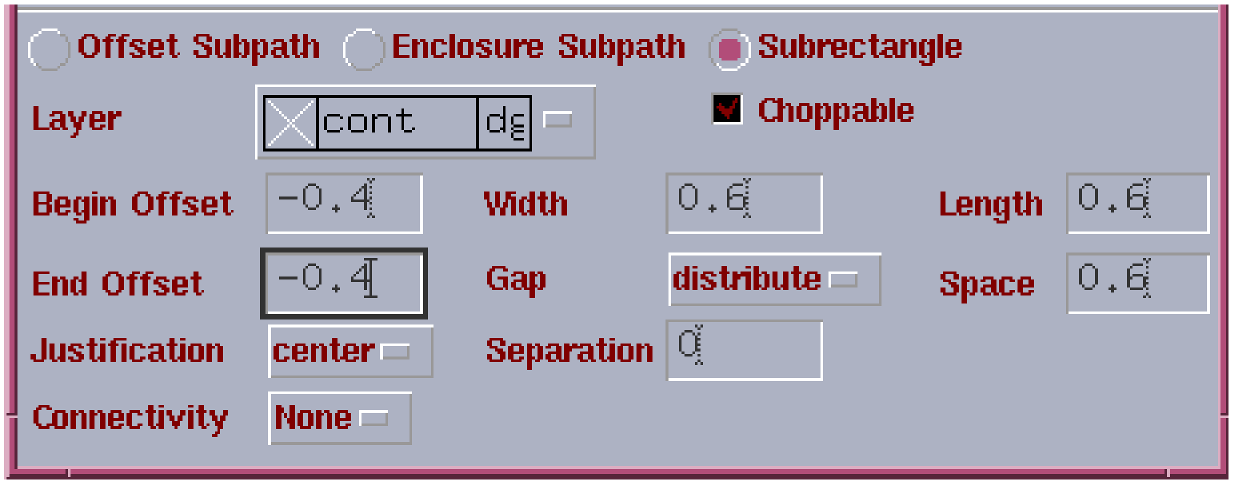
You use this single form to specify any number of the three subpath types: *offset*, *enclosure*, and *subrectangle*. The type of subpath is controlled by the radio buttons near the center of the form. At this point in the exercise, you will specify a subrectangle path (currently selected).

5. Enter the following values in the specified fields:

Layer	`cont dg`
Begin Offset	`0.4`
End Offset	`0.4`

With these values, the multipart path will contain a series of square contact structures that begin and end 0.4 units inside the extent of the metal master path.

Notice that as you make the change to the layer specification, the values of the **Width**, **Length**, and **Space** fields are updated with minimum dimension and spacing information from the current technology, as shown below:



( ) Offset Subpath ( ) Enclosure Subpath [● Subrectangle]

Layer: cont dg

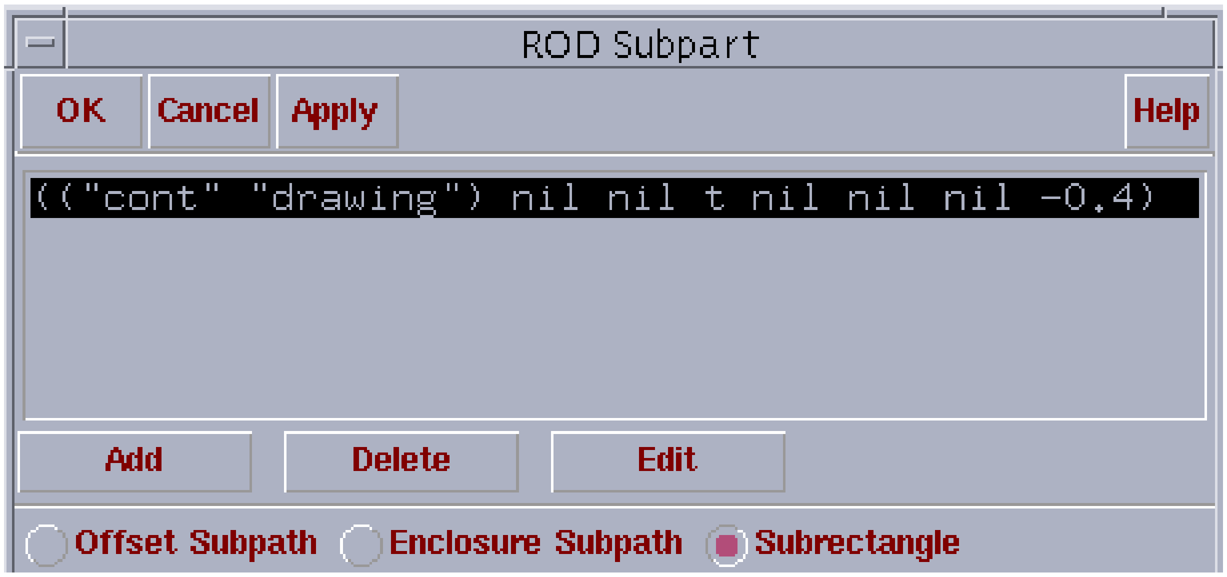
[✓] Choppable

Begin Offset	-0.4	Width	0.6	Length	0.6
End Offset	-0.4	Gap	distribute	Space	0.6
Justification	center	Separation	0		
Connectivity	None				

Also notice that as you specify a new value for **Begin Offset** the value for **End Offset** is automatically updated.

6. Click the **Add** button near the center-left portion of the form.

   This adds a subrectangle subpath specification with the options you entered. A description of the new subpath appears in the upper pane of the form as shown below:

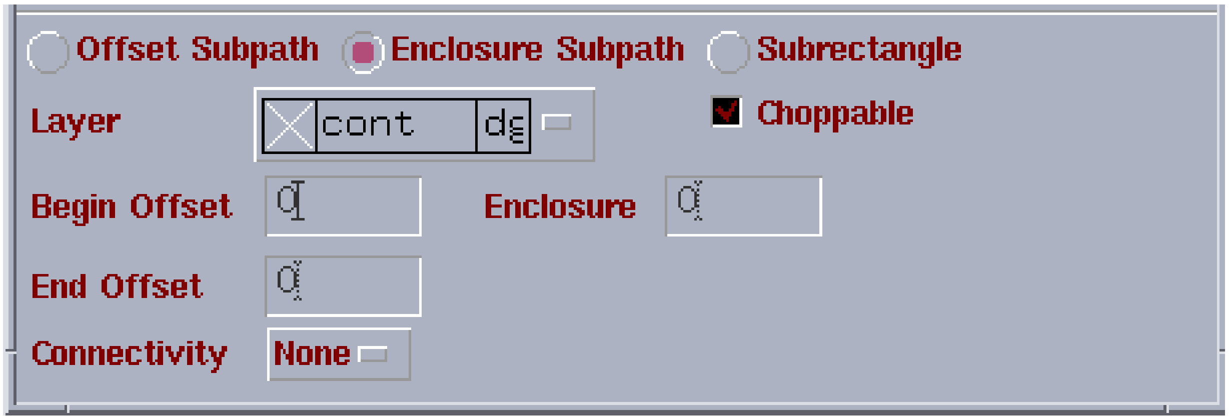


```
(("cont" "drawing") nil nil t nil nil nil -0.4)
```

Offset Subpath Enclosure Subpath [✓ Subrectangle]

7. Click the **Enclosure Subpath** option near the center of the form.

   A new collection of options appears as shown below:



Offset Subpath [✓ Enclosure Subpath] Subrectangle

Layer	cont dg	✓ Choppable	
Begin Offset	0	Enclosure	0
End Offset	0		
Connectivity	None		

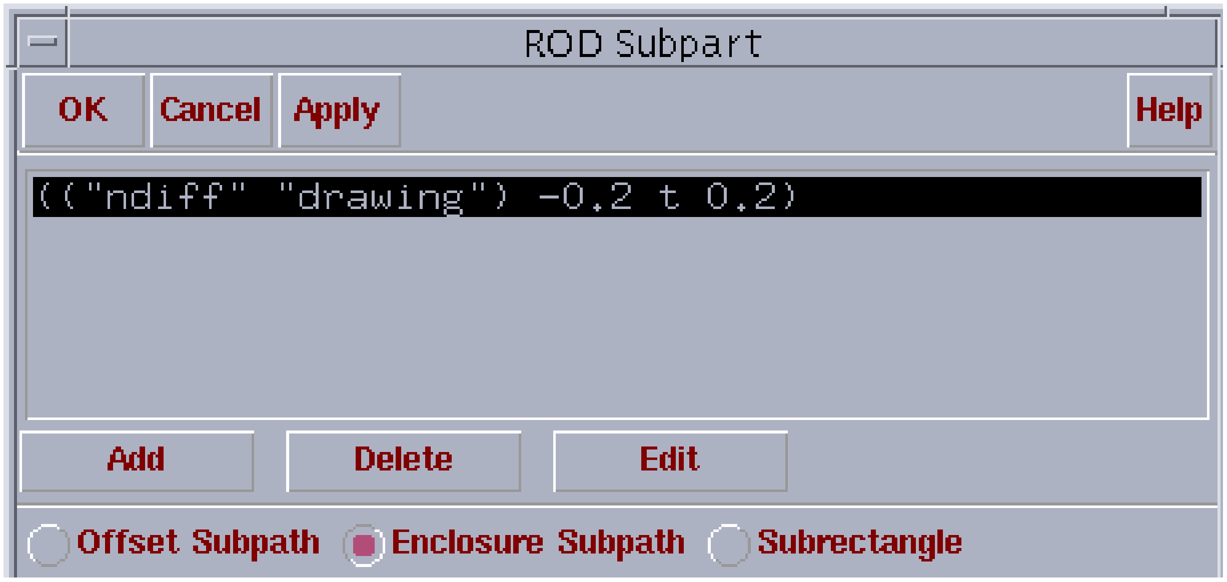
8. Enter the following values in the specified fields:

```
Layer          ndiff dg
Begin Offset    0.2
End Offset      0.2
Enclosure      -0.2
```

   These values specify that a subpath of diffusion material will surround the master path by 0.2 units on all sides.

9. Click the **Add** button near the center-left portion of the form.

   This creates an enclosure subpath with the options you entered. A description of the new subpath appears in the upper pane of the form as shown below:



```
(("ndiff" "drawing") -0.2 t 0.2)
```

Offset Subpath [✓ Enclosure Subpath] Subrectangle

10. Click the **OK** button at the top of the form.

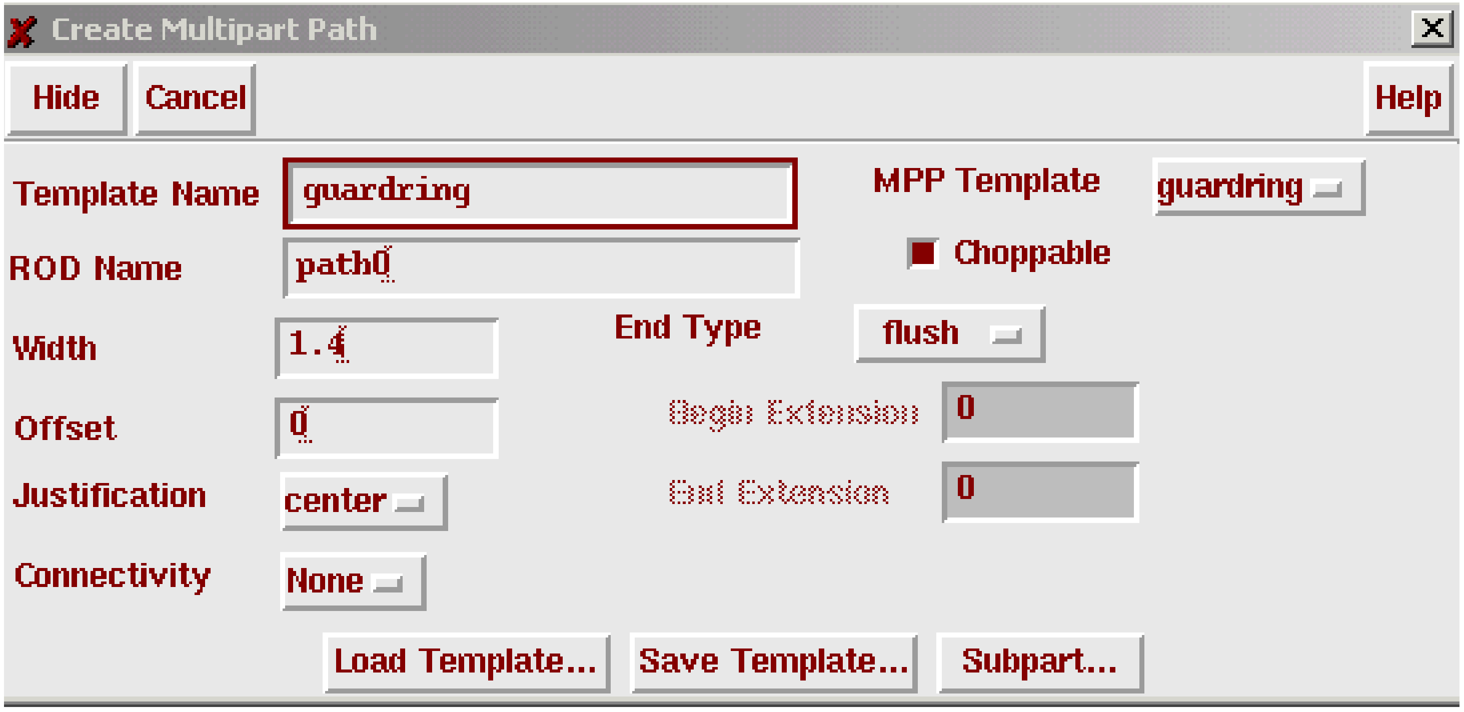
    This action adds these two subpaths to the definition of the multipart path.

11. In the layout editor window, click in various locations to enter the points of a new multipart path which uses the master and subpath definitions you entered. Click twice at the same point or press the **Return** key to end the path.

    The architecture of the resulting multipart path should resemble the path created using the **F10** key at the beginning of this activity.

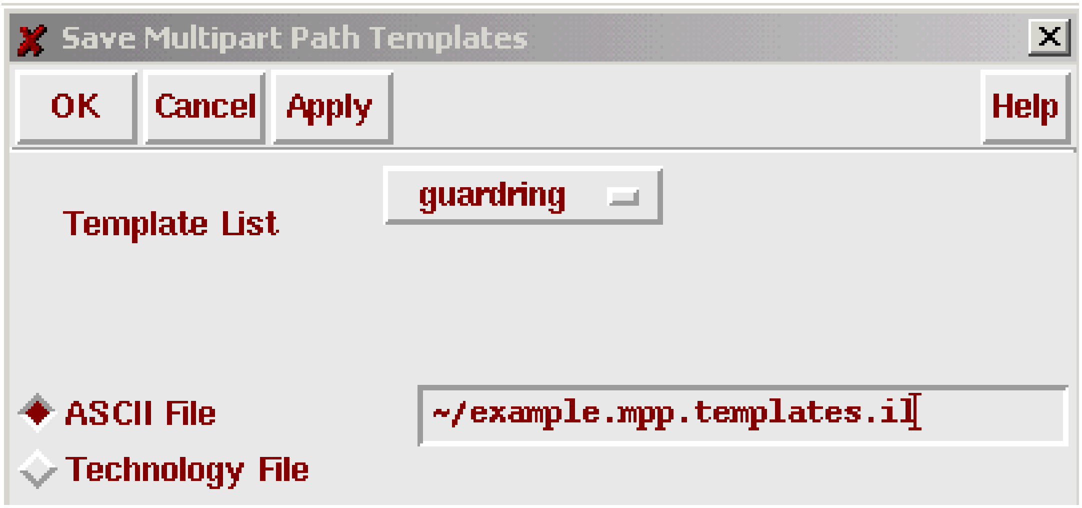
Saving the Guard-ring Template

1. In the *ROD Create Multipart Path* form (the form that appeared when you pressed the **F3** key), enter *guardring* in the Template Name field and then click the **Save Template** button near the lower-right corner of the form.



Template Name	guardring	MPP Template	guardring
ROD Name	path0	Choppable	■
Width	1.4	End Type	flush
Offset	0	Begin Extension	0
Justification	center	End Extension	0
Connectivity	None		

2. You may now save it to a file or the technology file. Select *ASCII file* and use the default file name.



Template List: guardring

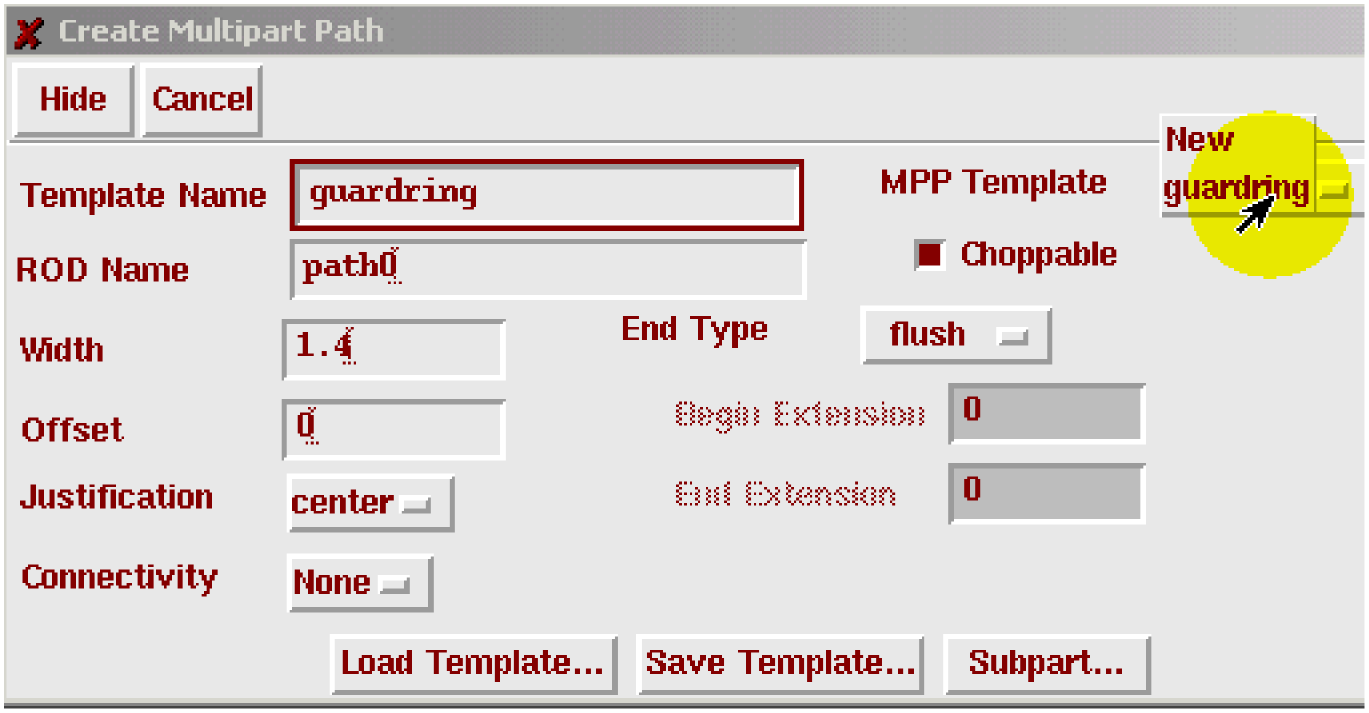
[✓ ASCII File] ~/example.mpp.templates.il
Technology File

3. Click the **OK** button in this form.

   This action saves the architecture (or *template*) of the multipart path into the ASCII file.

4. In the *ROD Create Multipart Path* form, click on the **MPP Template** cyclical field.

   Notice that the *guardring* template appears as shown below:



Template Name: guardring
ROD Name: path0
Width: 1.4
Offset: 0
Justification: center
Connectivity: None
MPP Template: guardring
Choppable
End Type: flush
Begin Extension: 0
End Extension: 0

The ROD Create Multipart Path form will allow you to create paths with the *guardring* template in future design sessions.

You can add any number of other templates as necessary for your design flow.

Prepare for the Next Lab

Execute the following steps to prepare for the next lab exercises:

1. In the CIW, select **File—Exit** and click on **Yes** in the dialog box that appears.

   This command closes the Design Framework II session and all of the windows associated with it.

2. In the shell, enter the following command:

   ```
   cd ~
   ```

   This command places you in your home directory.

Summary

In these exercises, you have seen how ROD shapes can be created and aligned. You have seen how ROD can be used to create a low-level design entity (a transistor), a complete cell (an inverter), and a multi-layer structure (a guard-ring). You also have used the interactive feature of the layout editor to create different ROD objects including a multipart path.

You now have an insight to the capabilities provided by relative object design.

End of Lab

Labs for Module 3

Exploring Relative Object Design

Before You Begin Module 3 Labs

In this exercise, you investigate the structure of ROD shapes. Your investigation will provide insight into:

- ROD handles
- ROD shapes with connectivity
- Behavior of ROD objects when their underlying shapes are modified
- Creating ROD shapes from existing ROD shapes

To perform this investigation, you are required to type SKILL commands into the Command Interpreter Window (CIW) and observe the effects.

Entering SKILL Commands

For your convenience, a file named *Commands* has been provided in the exercise directory. This file contains the text that you type into the CIW during the course of the exercise. If you wish, you may open a text editor (such as *vi*) on this file and copy and paste each entry into the CIW as required by the lab instructions. For example, once you have completed step 1 of *Start a Design Framework Session*, you can enter this command into your shell:

```
vi Commands
```

Cadence Online Documentation

The exercises in this lab explore ROD objects in significant detail, and many calls to ROD functions are made. To gain further insight into the structure of ROD object and options available in the ROD functions, it is recommended that you open the Cadence® online documentation, *Virtuoso Relative Object Design User Guide* prior to beginning the exercises. To do this, enter the following command into the shell:

```
cdsdoc
```

When the documentation window appears, click on the Layout Editor folder, then click on Virtuoso Relative Object Design User Guide.

Lab 3-1 Investigating ROD Object Structure

Objective: This exercise introduces the structure of ROD objects.

Start a Design Framework II Session

In this section, you open a Cadence Design Framework IIsession. A Virtuoso® Layout Editor window for the cellview *explore rodobj layout* opens automatically.

1. Enter these commands into a terminal window:

```
cd ~/Skill_PCells_5_1_41/exploreROD
layout &
```

   After a few moments the Command Interpreter Window (CIW) appears followed by a new (empty) layout editor window for the cellview *explore rodobj layout*.

2. Obtain the ID for the current cellview by entering the following command into the CIW:

```
cv = geGetEditCellView()
```

Create and Examine a ROD Rectangle

In this section, you create a ROD rectangle in the layout editor window. You then examine various aspects of its structure including its bounding-box point handles.

1. Enter the following command into the CIW:

```
rect = rodCreateRect(
   ?cvId  cv
   ?layer "poly"
   ?bBox  list(1:1 3:8)
)
```

   Look carefully at the returned value. *What type of object was created?*

2. Examine the structure of the ROD object by entering the following command into the CIW:

   ```
   rect~>??
   ```

   From the information printed in the CIW, answer the following questions:

   a. What is the *name* of this ROD object?

   b. Which attribute identifies the cellview containing this object?

   c. How many user-defined handles are associated with this object?

3. Inspect the rectangle database object associated with this ROD object by entering this command into the CIW:

   ```
   rect~>dbId~>??
   ```

   From the information printed in the CIW, answer the following questions:

   a. What type of object is this?

   b. Is it a different type from the ROD object itself?

   c. Does the bounding box (*bBox*) of this rectangle match the specifications given in the *rodCreateRect* call?

4. Enter the following commands into the CIW to examine useful handles of the new ROD object:

   ```
   rect~>lowerLeft
   rect~>upperRight
   ```

   Do these values correspond to the coordinates given in the *rodCreateRect* call?

5. Try accessing some of these bounding box point handles by their abbreviations. Enter the following commands into the CIW:

   ```
   rect~>lL
   rect~>cC
   rect~>uR
   ```

   Can you identify to which point handles these abbreviations correspond?

Create and Examine a Rectangle Array

In this section, you use *rodCreateRect* to create an array of rectangles. You then examine the resulting list of ROD objects.

1. Create an array of rectangle objects by entering the following command into the CIW:

   ```
   array = rodCreateRect(
      ?cvId      cv
      ?name      "contacts"
      ?layer     "cont"
      ?width     1
      ?length    1
      ?origin    1:10
      ?elementsX 3
      ?elementsY 2
      ?spaceX    1
      ?spaceY    2
   )
   ```

   Examine the results printed in the CIW and answer these questions:

   a. What type of data is returned from this call?

   b. How many elements does it contain?

   c. What type of data are each of the elements?

2. Enter the following command into the CIW to examine the names of the objects in the array:

   ```
   array~>name
   ```

   Examine the results in the CIW and answer these questions:

   a. What are the names of each object?

   b. How were the names formed? That is, what convention was used to create each name?

3. Attempt to discern which of the rectangles is the fifth element of the array. Enter the following command into the CIW to verify your answer:

   ```
   dbMoveFig(nth(4 array)~>dbId
             cv list(0:1 "R0"))
   ```

   Note: Remember that *nth* numbers the first item in a list as *0*.

   Was your answer correct?

Create and Examine a ROD Polygon

A ROD polygon differs considerably in its structure from a simple rectangle. In this section, you create a ROD polygon and explore its structure and its behavior as you manipulate it with the **Stretch** editing command. You also use a SKILL utility provided in the current design framework session to display locations of various point handles. The source code for this utility may be found in the following file:

```
source/SPCannotatePointHandles.il
```

1. To create a ROD polygon, enter the following command into the CIW:

   ```
   polygon = rodCreatePolygon(
      ?cvId  cv
      ?layer "metal1"
      ?pts   list(9:4 11:4 11:1 5:1 5:2 9:2)
   )
   ```

2. To compare this object with the rectangle created in the last section, enter the following commands into the CIW:

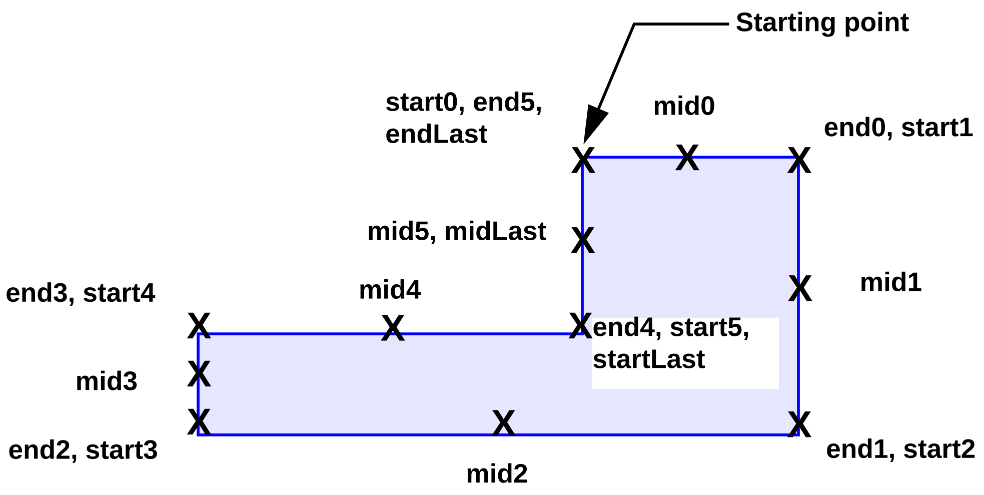
   ```
   rect~>??
   polygon~>??
   ```

   Does the polygon possess any attributes not found on the rectangle?

3. Examine the system handle names of the two objects and determine which system handles appear in the polygon but are absent in the rectangle. Enter the following command into the CIW to confirm your findings:

```
setof(handle polygon~>systemHandleNames
      !member(handle
              rect~>systemHandleNames))
```

4. Consider the following diagram of ROD polygon segment handles:



a. What are the values of the *start0*, *end5*, and *endLast* handles? **Hint**: refer to the point-list used in the call to *rodCreatePolygon*. Enter any one of these commands into the CIW to verify your answer:

```
polygon~>start0
polygon~>end5
polygon~>endLast
```

b. What is the value of the *startLast* handle? Enter this command into the CIW to verify your answer:

```
polygon~>startLast
```

5. A SKILL function has been provided in the current design framework session that displays the coordinates of ROD shape handles in a layout editor window. You may use the **h** bindkey to toggle these coordinate displays in a layout editor window. Enable point handle coordinate display in the layout editor window and record the locations of these points:

   start0

   mid0

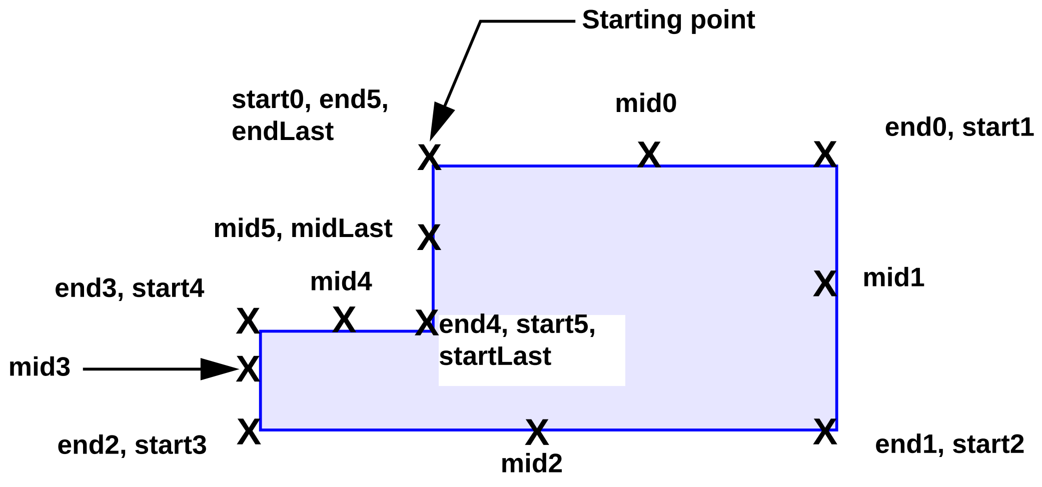
   mid4

   startLast

   midLast

6. Disable point handle coordinate display by pressing the **h** bindkey again.

7. To observe the effect on the ROD object's handles, change the shape of the polygon by using the **Edit—Stretch** command on the vertical edge close to the center of the polygon. Move the edge 2.0 units to the left. The modified polygon should look like this:



8. Enable point handle coordinate display in the layout window by pressing the **h** bindkey once more. Compare the values of the handles listed in step 5 with those now displayed in the layout editor window. Notice that the values are automatically updated after the stretch operation.

9. Disable point handle coordinate display in the layout editor window by pressing the **h** bindkey again.

Create and Examine a Simple Path

In this section you create a simple path and examine its structure and behavior.

1. To create a simple, single layer path, enter the following command into the CIW:

```
path = rodCreatePath(
   ?cvId  cv
   ?layer "metal2"
   ?width 2.0
   ?pts   list(13:2 18:2 18:5 23:5)
)
```

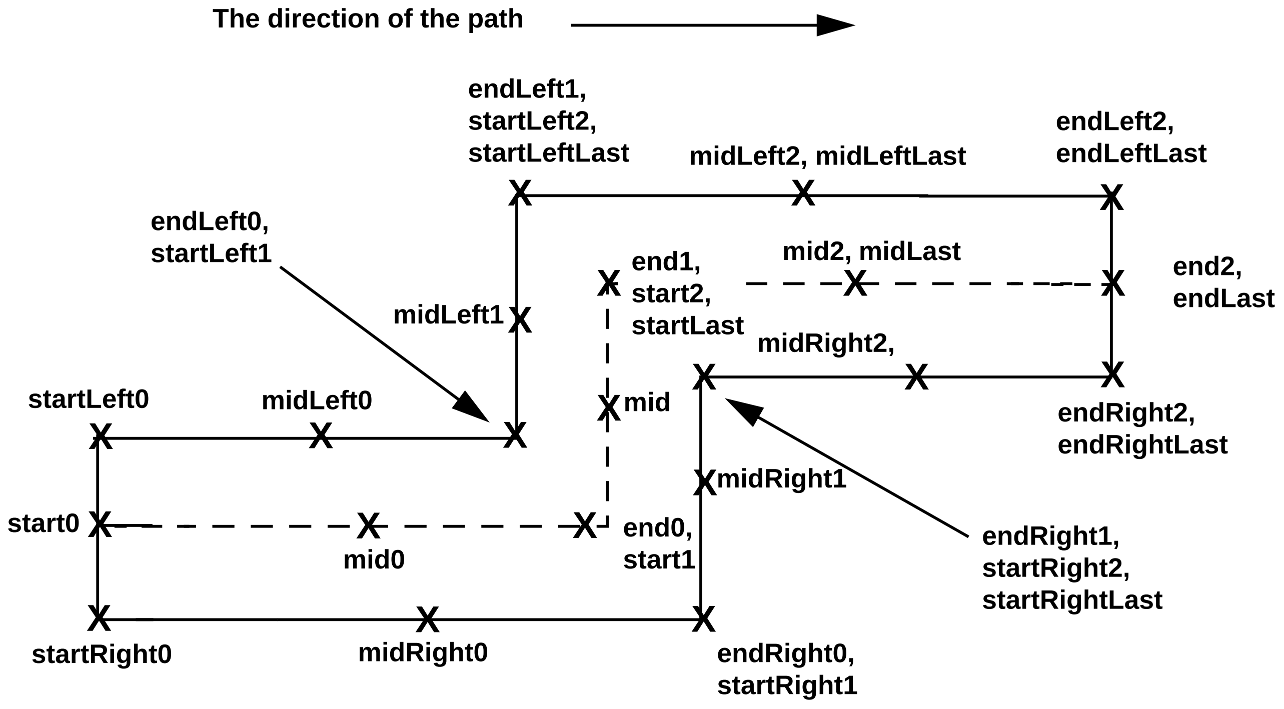
2. Enter the following command into the CIW to examine the structure of the simple path object:

```
path~>??
```

Which point handles appear that were not seen in the polygon object? To confirm your answer, enter the following command into the CIW:

```
setof(handle path~>systemHandleNames
      !member(handle
              polygon~>systemHandleNames))
```

3. Consider the following diagram of point handles for this path:



Enable point handle coordinate display in the layout window by pressing the **h** bindkey. Examine the points displayed, and answer the following questions:

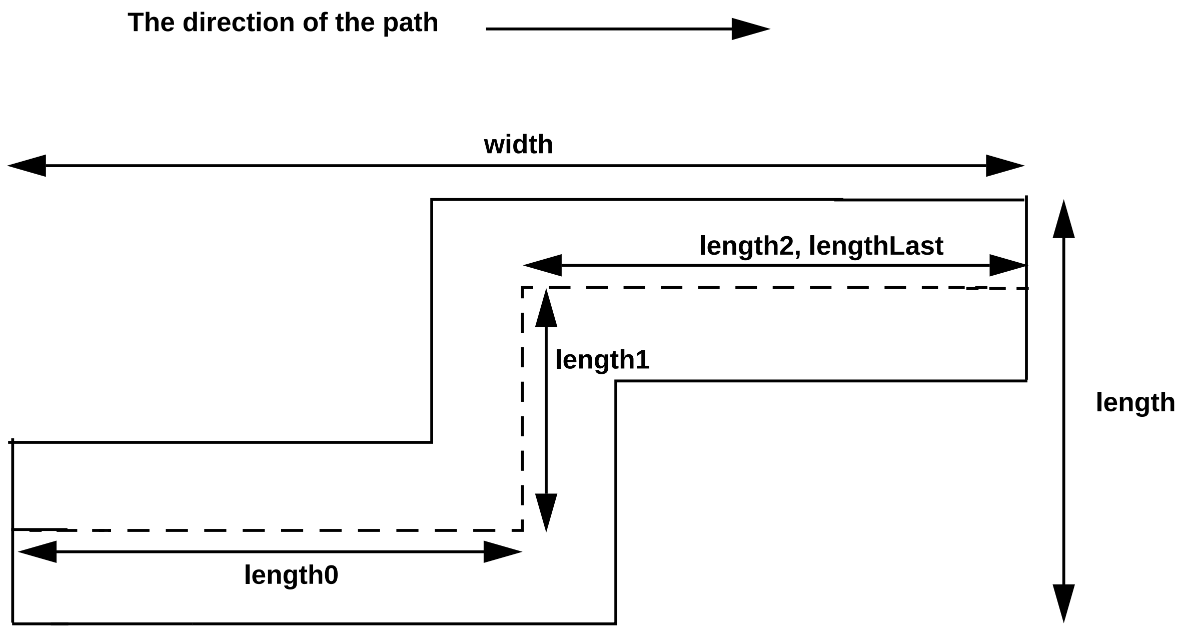
a. What are the values of the *start0* and *endLast* point handles? Enter these commands to confirm your answers:

```
path~>start0
path~>endLast
```

b. What is the value of the *mid2* point handle? Enter this command to confirm your answer:

```
path~>mid2
```

4. Consider the following diagram of length handles for this path:



What are the values of the *length* and *width* handles for this path? Confirm your answer by entering the appropriate commands into the CIW. *Are these values directly applicable to calculating, for example, interconnect length?*

5. Determine the values of length0, length1, and length2 by entering the appropriate commands into the CIW. *How would you calculate the total length of a path with an arbitrary number of segments?* **Hint**: You can find an example SKILL function for such a calculation in the appendices of the *Relative Object Design User Guide*.

6. Disable point handle coordinate display in the layout editor window by pressing the **h** bindkey again.

Create and Examine a Multipart Path

In this section, you create a multipart path and examine its structure compared to the simple path created in the previous section. You will use a SKILL utility provided in the current design framework session to create the multipart path. The source for this utility may be found in the following file:

```
source/SPCguardRing.il
```

1. Enter the following command into the CIW to create the multipart path:

   ```
   mpp = SPCguardRing(list(10:9 17:9 17:13))
   ```

2. To observe the structure of the multipart path object, enter this command into the CIW:

   ```
   mpp~>??
   ```

   Examine the results printed in the CIW and answer these questions:

   a. Does this ROD object contain any sub-shapes?

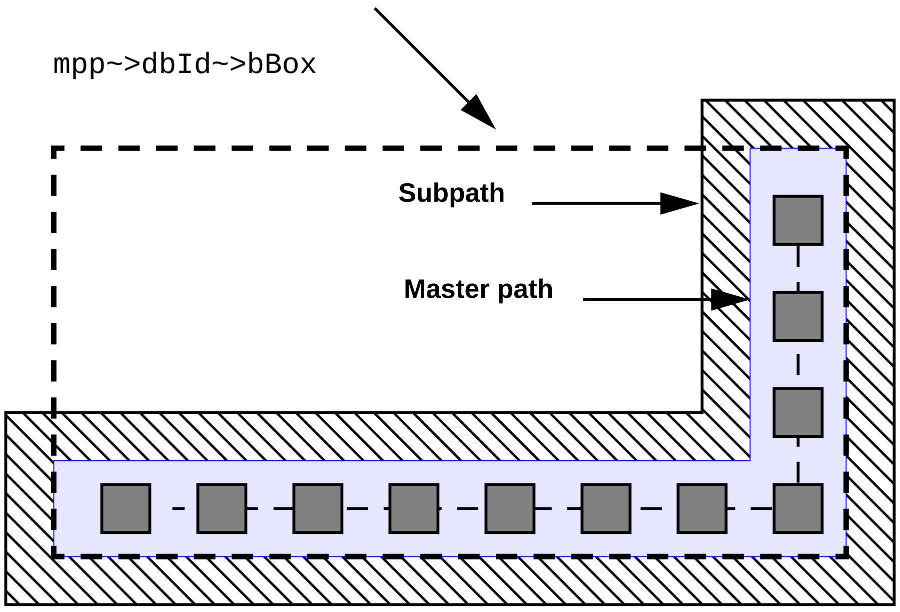
   b. Did the simple path created in the previous section contain any sub-shapes?

   c. Are these sub-shapes ROD objects, database objects, or some other type of object?

3. It is important to understand that the bounding-box of a multipart path can be different from the bounding-box of the master path itself. Consider the master path bounding-box diagram below:

Bounding box around master path:

```
mpp~>dbId~>bBox
```



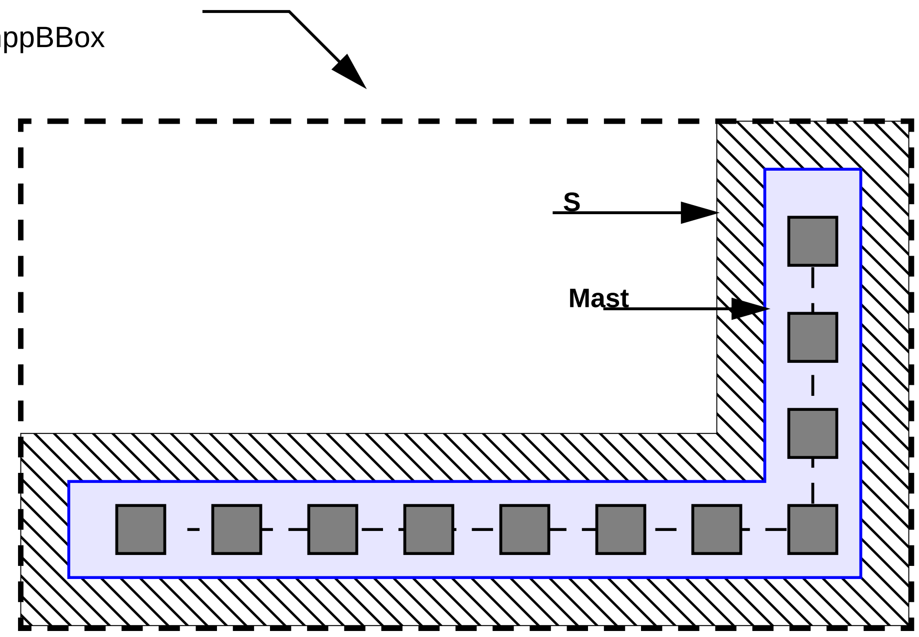
4. To observe the master path bounding-box, enter this command into the CIW:

```
rodCreateRect(
   ?cvId  cv
   ?layer list("cellBoundary" "drawing")
   ?bBox  mpp~>dbId~>bBox
)
```

5. Consider the following diagram of the whole multipart path bounding-box:

Bounding box around the whole multipart path:

mpp~>mppBBox



6. To observe the whole multipart path bounding-box, enter this command into the CIW:

```
rodCreateRect(
   ?cvId  cv
   ?layer list("cellBoundary" "drawing")
   ?bBox  mpp~>mppBBox
)
```

Under what circumstances would these two bounding boxes be equal?

Create and Examine Shapes with Connectivity

In this section you create another simple rectangle and assign connectivity to it in the same ROD call. You then investigate the resulting ROD object and its associated connectivity structures. Next, you create a rectangle and make it a pin with a single ROD call. You then examine the resulting ROD object and its associated connectivity structures.

1. Investigate the presence of nets and terminals in the current cellview by entering the following commands into the CIW:

   ```
   cv~>nets
   cv~>signals
   cv~>terminals
   ```

2. Enter the following command into the CIW to create a rectangle with associated connectivity:

   ```
   gnd = rodCreateRect(
      ?cvId     cv
      ?layer    list("metal1" "drawing")
      ?bBox     list(20:8 22:10)
      ?termName "gnd"
   )
   ```

3. Inspect the database object associated with the new ROD rectangle by entering this command into the CIW:

   ```
   gnd~>dbId~>??
   ```

   In what way can you determine that this rectangle has connectivity associated with it? **Hint**: Does this rectangle have a *net* associated with it?

4. Use the mouse to select this rectangle.

5. Choose **Edit—Properties** from the layout editor menus. A property editor form appears displaying the object's attributes.

6. Choose the button marked **Connectivity** in the property editor. Notice the signal associated with the rectangle. *Is this the signal name you expected?*

7. Click on **Cancel** in the property editor form.

8. Enter any or all of the following commands into the CIW to investigate the changes to the electrical information contained in the current cellview:

   ```
   cv~>nets~>name
   cv~>signals~>name
   cv~>terminals~>name
   ```

9. Create a ROD pin in the current cellview by entering the following command into the CIW:

   ```
   pin = rodCreateRect(
      ?cvId     cv
      ?layer    list("metal2" "drawing")
      ?bBox     list(20:11 22:13)
      ?termName "data"
      ?pin      t
      ?pinLabel t
   )
   ```

   Some of the connectivity arguments supplied to *rodCreateRect* enabled creation of a label for the pin. *Is the label visible?*

10. Choose the **Options—Display** menu in the layout editor window. A form appears with a number of display options. Enable the **Pin Names** option and click **OK** on the form. The pin label is now visible.

11. Use the mouse to select this rectangle.

12. Choose **Edit—Properties** from the layout editor menus. A property editor form appears. If not already displayed, click on the **Connectivity** option in the property editor form. Examine the form and answer these questions:

    a. What is the signal name associated with this rectangle?

    b. What is the I/O type for this pin?

    c. How did the pin attain this I/O type?

    d. How would you specify a different I/O type in the call to *rodCreateRect*? **Hint**: Refer to the section entitled *ROD Connectivity Arguments for Rectangles* in the Relative Object User Guide.

13. Click on **Cancel** in the property editor form.

Create a User Handle

In this section, you create a user handle for one of the ROD objects created in a previous section. You then examine the change to the target object. Another SKILL utility called *SPCprintHandleValues* is demonstrated in this section. You may find the source to this utility in the following file:

```
source/SPCprintHandleValues.il
```

1. Verify that the rectangle created in the previous section has no user handles by entering the following command into the CIW:

   ```
   pin~>userHandleNames
   ```

   Are there any user handles associated with this object?

2. Enter the following command into the CIW to add a user handle to this ROD object:

   ```
   rodCreateHandle(
      ?rodObj pin
      ?name   "maxCap"
      ?type   "float"
      ?value  0.015
   )
   ```

   What value is returned by *rodCreateHandle*? What other data types are allowed for user handles? **Hint**: See the section entitled *User-Defined Handles* in the *Relative Object Design User Guide*.

3. Examine the change to the ROD object by entering the command entered in step 1:

   ```
   pin~>userHandleNames
   ```

   Is the value returned different from the value in step 1?

4. To observe the value of the handle, enter the following command into the CIW:

   ```
   pin~>maxCap
   ```

   Is the value returned equal to the value provided in the call to *rodCreateHandle*?

5. Use the mouse to select this rectangle.

6. Choose **Edit—Properties** from the layout editor menus. A property editor form appears. If not already displayed, click on the **Property** option in the property editor form. *Are there any properties on this object?*

   It is important to note that adding a user handle for a ROD object does not add a property or any other information to the associated shape.

7. Click **Cancel** in the property editor form.

8. Use the *SPCprintHandleValues* function to examine all handles of this ROD object:

   ```
   SPCprintHandleValues(pin)
   ```

9. If you wish, you may open a text editor (or viewer) on the source file for this function to determine how it functions. For example, enter this command into the CIW:

   ```
   view “source/SPCprintHandleValues.il”
   ```

Lab 3-2 Creating ROD Objects from Other ROD Objects

Objective: This exercise introduces the concept of creating ROD objects based upon the structure of other ROD objects.

Create Rectangles from Other Objects

In this section, you will experiment with options to *rodCreateRect* to make several new ROD rectangles based upon existing ROD objects. To do this, you use the *?fromObj* and *?size* keyword arguments available with *rodCreateRect*.

Note: If you did not complete the previous lab successfully you need to set up this lab. Enter:

```
load “lab3-1.commands”
```

1. Enter the following command into the CIW:

   ```
   rodCreateRect(?cvId cv ?layer "metal1"
                 ?fromObj last(array))
   ```

   Examine the array of contacts created earlier in this exercise. The lower left-most contact now appears with a rectangle of first layer metal superimposed upon it.

   In this case, *rodCreateRect* uses the dimensions and location of the source object to create a generated object of the same shape and position in the cellview.

2. Enter the following command into the CIW:

   ```
   rodCreateRect(?cvId cv ?layer "cont"
                 ?fromObj gnd ?size -0.6)
   ```

   Examine the **metal1** rectangle at the right side of the window. This is the ground pin object created earlier in this exercise. Notice that a contact now appears within this rectangle.

   The *?size* keyword argument instructs *rodCreateRect* to create a new rectangle 0.6 units smaller than the source rectangle.

3. Enter the following command into the CIW:

```
rodCreateRect(?cvId cv ?layer "poly"
              ?fromObj gnd ?size 0.4)
```

   Examine the **metal1** rectangle at the right side of the window once again. Notice that the **metal1** rectangle and contact now appear within a rectangle of polysilicon.

   In this case, the *?size* keyword argument instructs *rodCreateRect* to create a new rectangle 0.4 units *larger* than the source rectangle.

Create Polygons from Other Objects

In this section, you will experiment with options to *rodCreatePolygon* to make several new ROD polygons based upon existing ROD objects. To do this, you use the *?fromObj* and *?size* keyword arguments available with *rodCreatePolygon*.

1. Enter the following command into the CIW:

```
rodCreatePolygon(?cvId cv ?layer "cellBoundary"
                 ?fromObj polygon ?size 0.5)
```

   Examine the **metal1** polygon created earlier in this exercise. An outlining polygon now appears around the **metal1** polygon.

   Here the *?size* keyword argument instructs *rodCreatePolygon* to create a polygon 0.5 units larger than the dimensions of the source object.

2. Enter the following command into the CIW:

```
rodCreatePolygon(?cvId cv ?layer "ndiff"
                  ?fromObj polygon ?size -0.5)
```

   Examine the **metal1** polygon again. A new polygon appears in diffusion enclosed by the **metal1** polygon.

   Notice that when instructed to create a polygon 0.5 units *smaller* than the source polygon, *rodCreatePolygon* created a rectangular shape. This was done because the narrow extension on the left side of the source object is only 1.0 units tall. Hence, no part of the generated object would fit there and still be 0.5 units smaller than the source object.

Create Paths from Other Objects

In this section, you will experiment with options to *rodCreatePath* to make several new ROD paths based upon existing ROD objects. To do this, you use the *?fromObj* and *?size* keyword arguments available with *rodCreatePath*.

1. Enter the following command into the CIW:

   ```
   rodCreatePath(?cvId cv ?layer "metal1"
                 ?fromObj path ?size 0.6)
   ```

   Examine the **metal2** path created earlier in this exercise. An outlining **metal1** path now appears around the **metal2** path.

   Notice that the generated path is based upon the *perimeter* of the source object.

2. Enter the following command into the CIW:

   ```
   rodCreatePath(?cvId cv ?layer "metal1" ?width 2
                 ?fromObj array ?size -0.5)
   ```

   Examine the array of contacts created earlier in this exercise. A **metal1** path now surrounds the contacts.

   In the call to *rodCreatePath*, you specify a list of objects (the collection of contacts stored in the SKILL variable *array*) as the source objects to the *?fromObj* keyword. Notice that the generated path is based upon the *bounding box* of the source objects.

3. Enter the following command into the CIW:

   ```
   rodCreatePath(?cvId cv ?layer "metal2"
                 ?fromObj rect ?size 0.6
                 ?startHandle "mid0"
                 ?endHandle "end2")
   ```

   Examine the polysilicon rectangle created earlier in this exercise in the lower-left corner of the cellview. A **metal2** path now surrounds a portion of this rectangle.

   In this case, you use the *?startHandle* and *?endHandle* keyword arguments to *rodCreatePath* so that the resulting path begins at the midpoint of the rectangle’s first segment and ends at the endpoint of the rectangle’s third segment.

Prepare for the Next Lab

Execute the following steps to prepare for the next lab exercises:

1. In the CIW, choose **File—Exit** and click on **Yes** in the dialog box that appears.

   This command closes the Design Framework II session and all of the windows associated with it.

2. In the shell, enter the following command:

   ```
   cd ~
   ```

   This command places you in your home directory.

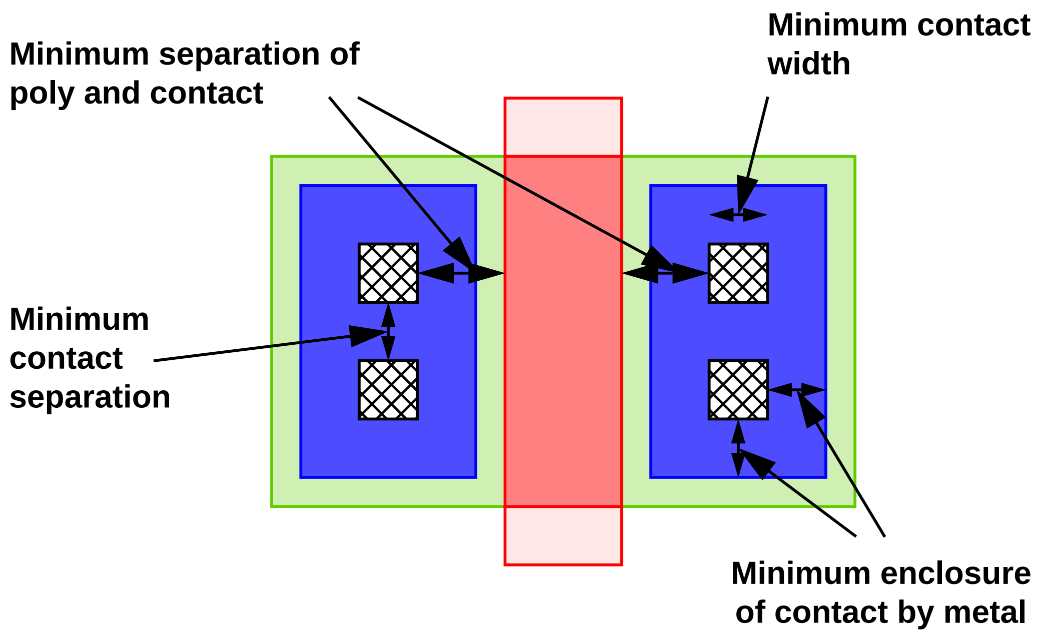
Summary

In the course of these exercises, you have seen the underlying structure of ROD rectangles, rectangle arrays, polygons, simple paths, multipart paths, and shapes with associated connectivity. You have observed the effects of editing ROD objects within the layout editor. You created new ROD shapes based on existing shapes. You should now be familiar enough with ROD objects to use them effectively in your layout design.

End of Lab

Labs for Module 4

Creating and Using SKILL Parameterized Cells


Minimum separation of poly and contact
Minimum contact width
Minimum contact separation
Minimum enclosure of contact by metal

Before You Begin Module 4 Labs

In these exercises, you begin each section with a SKILL source code file containing nearly all the information necessary to create a pcell. Your task is to determine the information missing in the body of the pcell.

Each missing piece of information has been replaced by a descriptive token surrounded by ‘#’ characters. For example, suppose the name of a layer is required at one point in the code, and your task is to specify this name as *metal1*. You might find a line in the file like this:

```
lpp = list(“#layerName#” “drawing”)
```

To accomplish your task, you would change the line to the following:

```
lpp = list(“metal1” “drawing”)
```

Hint: You can locate all of the missing information tokens by repeatedly searching for the ‘#’ character.

The pcell may not compile or behave properly if any missing information tokens remain in the source file. Only after you have supplied all missing information will the pcell compile and behave correctly.

Choose a Text Editor

You will need to edit source code files in this lab. You may choose from the following editors:

- vi
- textedit

If you are unfamiliar with these editors, choose *textedit* as it implements simple point-and-click editing.

Cadence Online Documentation

The exercises in this lab deal with relative object design in significant detail, and you will be required to refer to the *Relative Object Design User Guide* in certain sections of this lab. Therefore, open this documentation prior to beginning the exercises.

Enter the following command into the shell:

```
cdsdoc
```

When the documentation window appears, click on the Layout Editor folder, then click on Virtuoso Relative Object Design User Guide.

Lab 4-1 Creating a Simple SKILL Pcell

Objective: Learn fundamental pcell concepts by creating and experimenting with a basic pcell.

You begin by creating a simple pcell. You test this pcell and determine that it behaves properly. You then extend the pcell's functionality by adding a new parameter.

Start a Design Framework II Session

1. Enter these commands into a terminal window:

   ```
   cd ~/Skill_PCells_5_1_41/pcellIntro
   layout &
   ```

   After a few moments, the Command Interpreter Window (CIW) appears followed by a new (empty) Virtuoso® Layout Editor window for the cellview *pcells intro layout*.

2. Choose **Tools—Technology File Manager** from the menus in the CIW.

3. A form appears. Select **Dump** from this form.

4. In the form, select the following and click **OK**
   :

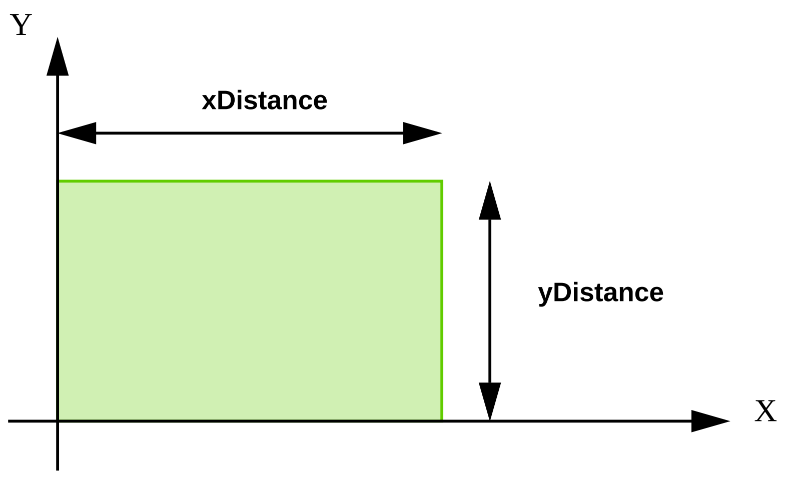
Technology Library	`pcells`
Classes	`Select All`
ASCII Technology File	`techfile`

   A readable form of the technology information will now reside within the file called *techfile*.

   Many of the exercises in this lab deal with technology information stored in the *pcells* library. You will find it helpful to have this information easily accessible in a text file.

Create a Simple Pcell

In this section, you create a pcell containing a single ROD rectangle. The incomplete definition for the pcell is in a file called *step1.il* in the current directory. The pcell accepts two parameters, *xDistance* and *yDistance*, that control the dimensions of the rectangle as depicted below:



As mentioned previously, you add missing information to complete this pcell definition. When you have specified all the missing information, you may use the pcell to create instances of a rectangle that can be sized by changing the instance parameters.

1. Open a text editor on the file *step1.il*.

2. Carefully read the comments at the top of the file.

   The intended function of the pcell is described here. You will find all the information needed to complete the pcell definition within these comments.

3. Examine the remainder of the file. Answer these questions:

   a. Can you identify the places in the file with missing information?

   b. In which portion of the file is the actual rectangle created?

   c. In the call to *rodCreateRect*, which variable is used to specify the target cellview?

   d. How is this variable set?

e. Are there any user-defined local variables in this pcell?

4. Recall the structure of *pcDefinePCell*:

   ```
   pcDefinePcell(
       l_cellIdentifier
       l_formalArgs
       body of code
   )
   ```

   The first section of the pcell definition is a list of items identifying the master cellview. Find this line within the source code file and replace the descriptive tokens with the appropriate information.

   Hint: Replace *#libraryName#* with *pcells*.

5. The next section of the call declares the parameters for this pcell. Locate the parameter declarations in the source code file and replace the descriptive tokens with the appropriate information.

6. Save your changes to the file.

7. In the CIW, enter the following command:

   ```
   load “step1.il”
   ```

   Did you see any errors or warnings? If so, try to rectify the problem, save your changes to the file, and repeat this step.

Instantiate and Test the Pcell

1. Once your pcell definition compiles without errors or warnings, open the resulting master cellview (*pcells step1 layout*) in a layout editor window.

2. Select the rectangle.

3. Choose **Edit—Properties** from the layout editor menus.

   Does the rectangle have the expected dimensions? Is it on the layer specified by the pcell definition?

4. Close this window and bring the window containing *pcells intro layout* forward.

5. Create an instance (**Create—Instance** in the layout editor menus) of *pcells step1 layout*.

6. Select this instance and invoke the property editor (**Edit—Properties** from the layout editor menus). In the property editor form, select the **Parameters** option.

7. Change the **xDistance** parameter to 2.0 and the **yDistance** parameter to 5.5. Click on the **Apply** button. Observe the change to the instance.

   Has the rectangle resized to the specified dimensions?

8. Change the **xDistance** parameter to 0.0 and click on **Apply**.

   What happened?

   Zoom out if necessary to see what has happened to the pcell instance. Look at the output pane of the CIW.

   Do you see any warning messages?

   Can you tell what went wrong by the content of these messages?

9. Change the **xDistance** parameter back to 2.0 and click on **OK**. Zoom in if necessary to verify that the instance is now intact.

Add a Parameter to the Pcell

You created a pcell that accepts parameters to define the dimensions of a rectangle. The rectangle is created on a layer hard-coded into the pcell definition. You now modify this pcell to accept a layer parameter.

1. In the text editor, add comments to the top portion of the *step1.il* file, that describes the new parameter.

   The new parameter is called *layer* which will be a string with the default value of *pdiff*. Use a format similar to the comments for the original two parameters.

2. Add a line to the formal parameters section declaring the new parameter. Insert this line immediately after the declaration for the *yDistance* parameter.

3. Now you must replace the hard-coded layer specification with the new *layer* parameter. Look at the body of the *rodCreateRect* call and make this substitution.

4. Save your changes to the file.

5. In the CIW, enter the following command:

   ```
   load “step1.il”
   ```

   Did you see any errors or warnings?

   If so, try to rectify the problem, save your changes to the file, and repeat this step.

Test the Layer Parameter

1. Once your pcell definition compiles without errors or warnings, select the instance you made in *pcells intro layout*.

2. **Choose Edit—Properties** from the layout editor menus and select the **Parameter** choice.

   Do you see the new parameter?

3. Change the **layer** parameter to *metal2* and click on the **OK** button. Observe the change to the instance.

   Did the instance exhibit the behavior you expected?

Important

If you had difficulty with this section or the previous section, you may refer to this file to see a complete solution:

```
source/step1.il
```

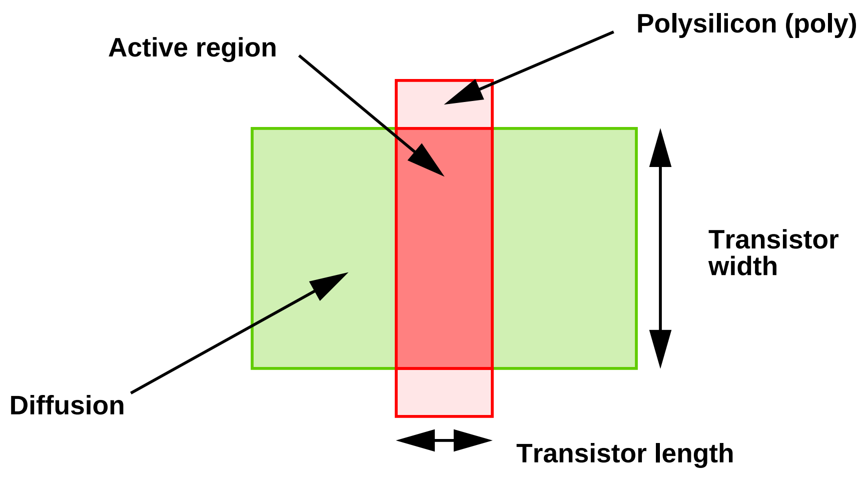
Prepare for the Next Section

Close the text editor for the file *step1.il*.

Lab 4-2 Creating an Elementary Transistor Structure

Objective: Learn how to access technology information and how to align rectangles.

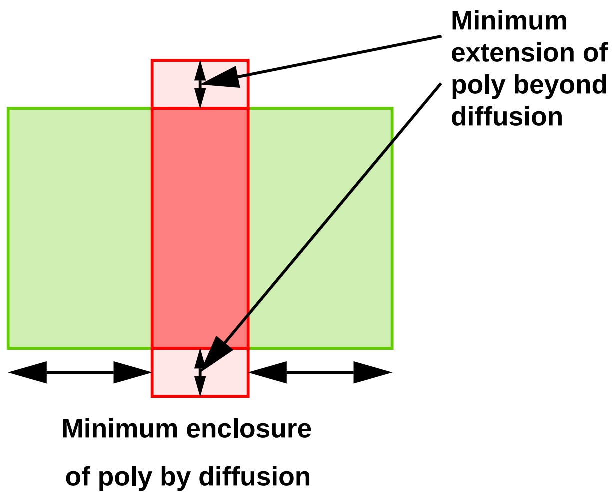
In this section, you define a pcell containing the fundamental shapes for a transistor. This pcell accepts parameters for the width and length of the transistor as well as for the layers of the rectangles used to form the gate and diffusion regions, as depicted below:



Important

Notice that the width and length parameters refer to the dimensions of the active region (transistor width and transistor length). This is contrary to the standard notion of width (X-dimension) and length (Y-dimension) of a shape such as a rectangle.

You will access various technology information to ensure that the device meets requirements for minimum size, spacing, and enclosure. Specifically, the rectangles must meet the design constraints depicted here:


Minimum extension of poly beyond diffusion
Minimum enclosure of poly by diffusion

1. Open a text editor on the pcell source code file *step2.il*.

2. Carefully read the comments at the top of the file. The intended function of the pcell is described here along with a description of the cell's parameters.

3. Now examine the cell identification section of this call to *pcDefinePCell*.

   Does the cellview here match the comments above?

4. Examine the formal parameter section below the cell identification section.

   Are the parameters specified in accordance with the comments?

5. Locate the call to *techGetTechFile* and supply the appropriate library name.

6. In the next statement in the source file, a spacing rule is retrieved from the library technology. Supply the name of the rule needed for this call to *techGetSpacingRule*.

   Use the comment line above the statement to determine the rule required. You can refer to the text file called *techfile* to determine valid rule names; look for the section titled "PHYSICAL RULES."

7. The next statement calls *techGetOrderedSpacingRule*. Supply the required layers in the proper order for this statement.

   Hint: Use cell parameters for the layer identifiers.

8. In the first call to *rodCreateRect*, the rectangle is created for the diffusion region. Supply the proper variable for the *width* keyword argument in this statement.

   Note: The width of the diffusion region depends upon the length of the transistor plus a minimum extension on both sides of the gate. If necessary, refer to the diagram provided at the beginning of this section.

9. The gate is created by the second call to *rodCreateRect*. Supply the missing parameter for the width of the rectangle.

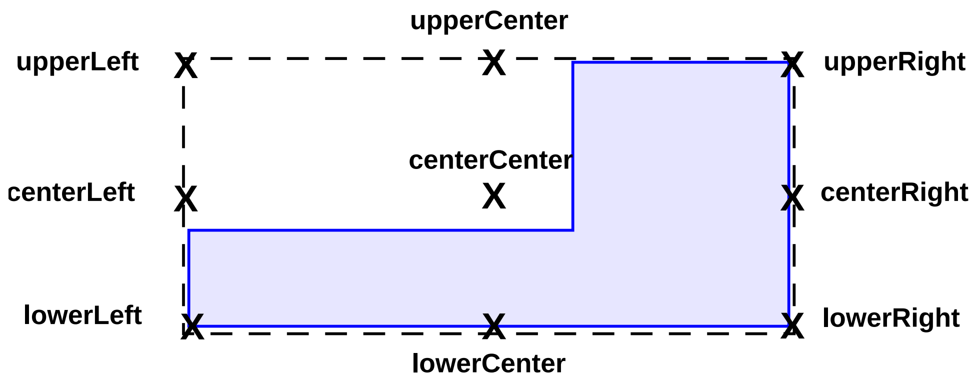
   Note: This distance is the length of the transistor.

10. Also in this statement, you must complete the expression for the length of the rectangle.

    Note: The length is the width of the transistor plus minimum overlap of diffusion on both ends.

11. The final statement in the body of the pcell establishes alignment between the two rectangles using *rodAlign*. You must supply the name of the point handle on the diffusion rectangle that will be used in this alignment.

    Refer to the diagram of the structure at the beginning of this section, and to the diagram of ROD point handles below:



12. For the *refObj* keyword argument, you must supply the variable associated with the gate rectangle.

13. Save your changes to the file.

14. In the CIW, enter the following command:

    ```
    load “step2.il”
    ```

    Did you see any errors or warnings?

    If so, try to rectify the problem, save your changes to the file, and repeat this step.

Test the Pcell

1. Create an instance of *pcells step2 layout* in the layout editor window.

   Does the instance closely resemble the diagram at the beginning of this section?

2. Measure the overlaps and compare them to the values in the technology file.

3. Select this instance and invoke the property editor.

4. Change the *width* and *length* parameters for this instance to various values and observe the results.

   Does the instance behave as expected?

Important

If you had difficulty with this section, you may refer to this file to see a complete solution:

```
source/step2.il
```

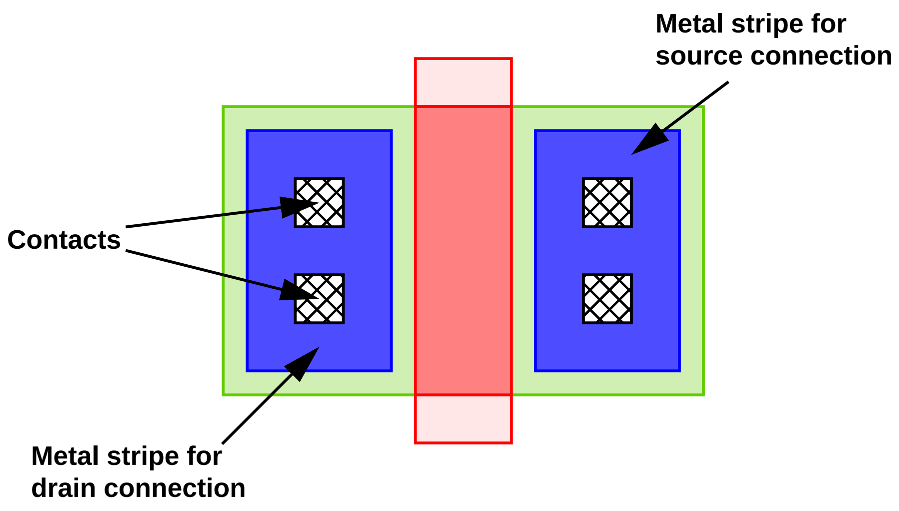
Prepare for the Next Lab

Close the text editor for the file *step2.il*.

Lab 4-3 Adding Source and Drain Connections Using Multipart Paths

Objective: Understand the use of multipart paths to create source and drain connections for the transistor pcell. Use ROD connectivity arguments to add electrical information to these connections.

You will now expand on the transistor pcell by adding source and drain connections. These are implemented by metal stripes with embedded contact arrays. The length of the metal stripes size proportionally with the width (vertical dimension) of the transistor. The contact arrays are properly spaced sets that fit within the length of each metal stripe. An example instance is depicted in the diagram below:

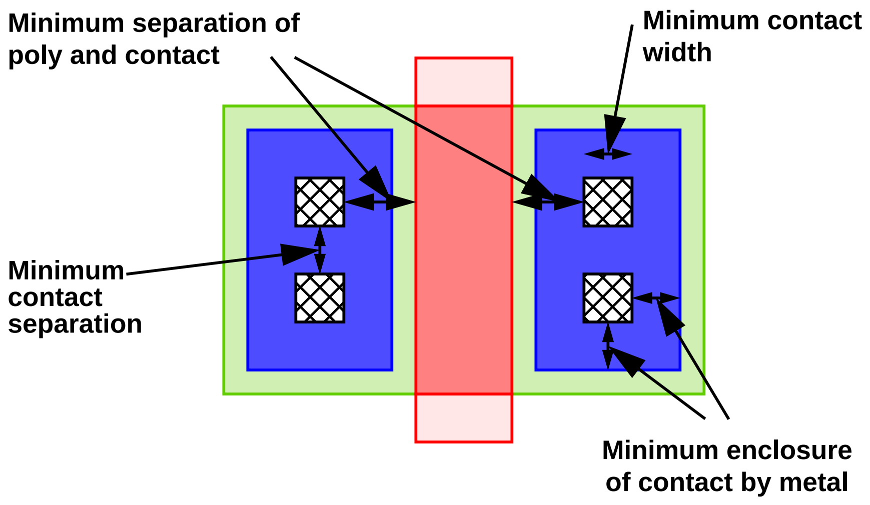

Metal stripe for source connection
Contacts
Metal stripe for drain connection

While it is still possible to use a number of non-ROD approaches for constructing the metal stripes and contact arrays, you will simplify the task by using a ROD multipart path to implement the source and drain connections.

You will define two identical multipart paths, one for each connection. The master path will be the metal stripe, and you will form the contact array by a single *subRect* declaration within the *rodCreatePath* call.

Each multipart path will be offset from the gate region by the minimum poly-to-contact spacing. The dimensional constraints for the diffusion and gate rectangles remain as they were in the previous section.

A diagram of the pertinent constraints is given below:


Minimum separation of poly and contact
Minimum contact width
Minimum contact separation
Minimum enclosure of contact by metal

1. Open a text editor on the pcell source code file *step3.il*.

2. Carefully read the comments at the top of the file. The intended function of the pcell is described here along with a description of the cell's parameters.

   Which parameter or parameters are new in comparison to the previous pcell?

Add Connectivity to the Gate Stripe

1. Search for the first *rodCreateRect* call. This is the specification for the polysilicon rectangle comprising the transistor's gate. You must provide the proper signal parameter for this structure.

   Hint: A cell parameter is required here rather than a hard-coded string.

Complete the Drain Connection

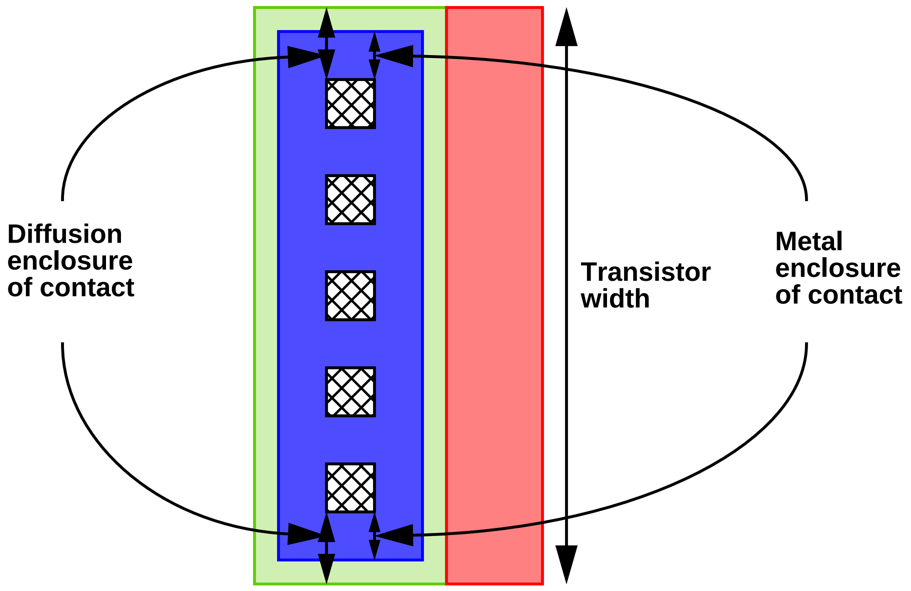
1. Search for the first *rodCreatePath* call. This is the specification for the drain connection. You must provide the proper signal parameter for the drain connection in the *termName* keyword argument.

2. In this same call to *rodCreatePath*, you must also specify the width of the master path.

   This distance is governed by the minimum contact width and the minimum enclosure of contact by metal. Refer to the previous diagram to determine the width of the master path, and put your answer in place.

3. Next, you must specify two points for the single-segment path. To simplify the path specification, the beginning point is chosen as (0,0). The end point, therefore, has an X-ordinate of 0.0 and a Y-ordinate equal to the required length for the path. Once the path has been created at the origin with the correct dimensions, a call to *rodAlign* moves it to its proper place within the cellview.

   Consider the following diagram of the multipart path to determine the calculation for the master path length:



   Hint: The length of the path will depend upon the width (vertical dimension) of the transistor and the two design rules shown in the diagram above.

   Enter your calculation in the appropriate place within the *pts* keyword argument.

4. When you are satisfied with your answers, locate the next call to *rodCreatePath*. This is the specification for the source connection. As with the drain connection, you must provide the proper signal parameter in the *termName* keyword argument.

5. In this same call to *rodCreatePath*, you must supply the values for the *width* and *pts* keyword arguments. Since the source and drain connections are of exactly the same dimensions, you can use the values determined for the first *rodCreatePath* call.

Examine the Connection Alignment

1. In the Virtuoso Relative Object Design User Guide document window, page down in the table of contents until you find About Sets of Subrectangles in the *rodCreatePath* section of Chapter 3. Click on this topic.

2. Locate the *subRect* keyword argument of the *rodCreatePath* statement. This portion of the *rodCreatePath* call specifies the contact array. Referring to the Virtuoso Relative Object Design User Guide, answer these questions:

   a. Are the width and length of the contact explicitly stated in this construct?

   b. How are the dimensions of the contacts determined?

   c. Is the separation between contacts explicitly specified here?

   d. How is the contact spacing determined?

3. Locate the *rodAlign* statement below the current *rodCreatePath* statement. Refer to the previous diagrams to answer these questions:

   a. Which two ROD objects are being aligned here?

   b. Why was *centerRight* chosen for the alignment point on the gate object?

      Hint: What governs the X-location of this point?

   c. Why does the *xSep* expression contain no reference to metal spacing rules?

      Hint: For which two layers is minimum spacing critical here?

4. Save your changes to the file.

5. In the CIW, enter the following command:

   ```
   load “step3.il”
   ```

   Did you see any errors or warnings?

   If so, try to rectify the problem, save your changes to the file, and repeat this step.

Test the Pcell

1. Create an instance of *pcells step3 layout* in the layout editor window.

   Does the instance closely resemble the diagram at the beginning of this section?

2. Select this instance and invoke the property editor.

3. Change the *width* and *length* parameters for this instance to various values and observe the results. For instance, change the *width* parameter to a relatively large number such as 10.0.

   Does the instance behave as expected?

Important

If you had difficulty with this section, you may refer to this file to see a complete solution:

```
source/step3.il
```

Prepare for the Next Section

Close the text editor for the file *step3.il*.

Lab 4-4 Multipart Path Transistor

Objective: Learn how to reduce the number of separate ROD objects by means of a multipart path.

The previous pcell transistor implementations have used separate objects for the various shapes in the cell. You now reduce the number of ROD function calls to a single call to *rodCreatePath*.

The poly rectangle is the central shape for the transistor in many ways. It follows that this shape becomes the master path for the multipart path you define. The diffusion region is implemented as a subpath along with the source and drain metal stripes and their associated contacts.

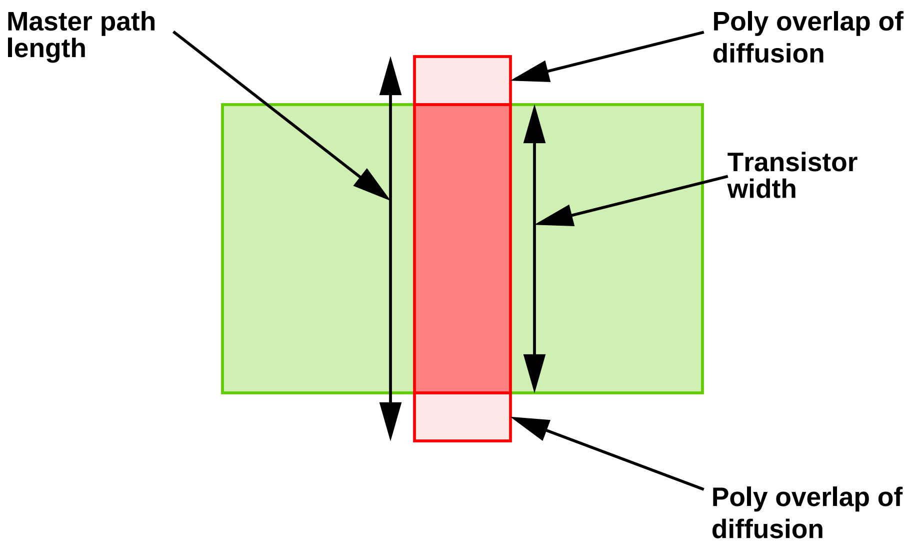
1. Open a text editor on the pcell source code file *step4.il*.

2. Carefully read the comments at the top of the file. The intended function of the pcell is described here along with a description of the cell's parameters.

Complete the Master Path Specification

1. Locate the call to *rodCreatePath*.

   You must supply the length of the master path in the *pts* keyword argument. To do this, refer to the following diagram.

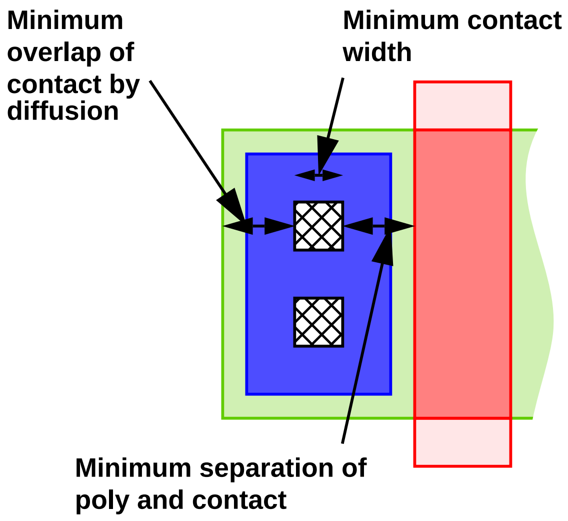


Hint: The location of the second point will be a function of the *width* parameter and the poly overlap of diffusion design rule.

Complete the Definition of the Diffusion Subpath

1. Locate the *encSubPath* keyword argument. This enclosing subpath forms the transistor's diffusion region. You must specify the distance this subpath overlaps the master path on each side using the *enclosure* keyword argument.

   Refer to the following diagram to form the required expression:


Minimum overlap of contact by diffusion
Minimum contact width
Minimum separation of poly and contact

   Notice that a *negative* value is specified for this argument. This instructs *rodCreatePath* to create a subpath that is larger than the master path by the specified amount.

2. Locate the *beginOffset* keyword argument in the enclosing subpath definition.

   You must supply the distance by which the diffusion region is shorter than the poly stripe on either end.

   Notice that you do not need to specify the *endOffset* keyword argument because its value, by default, is the same as the value specified to the *beginOffset* keyword argument.

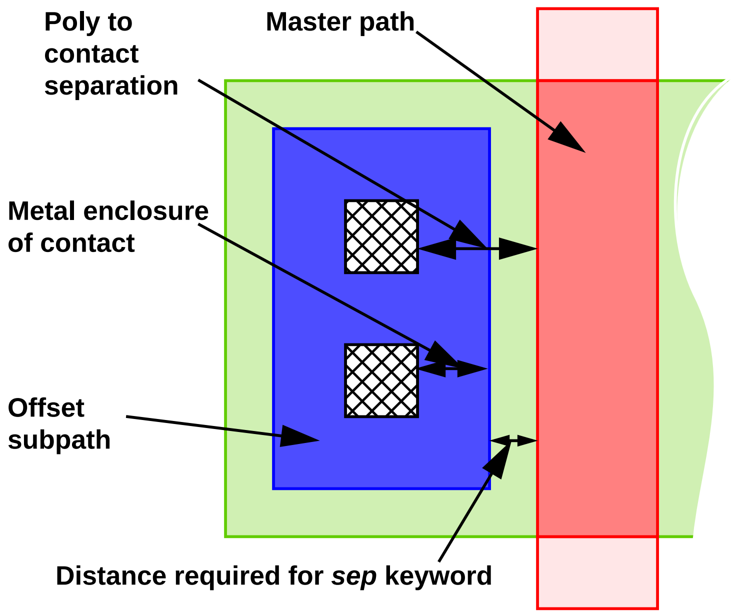
Complete the Definition of the Source and Drain Connection Subpaths

1. Locate the *offsetSubPath* keyword argument. The subpath definitions here form the metal stripes for the source and drain connections.

   Notice that the first offset subpath is left-justified, this subpath forms the drain metal. Similarly, the second offset subpath is right-justified, this subpath forms the source metal.

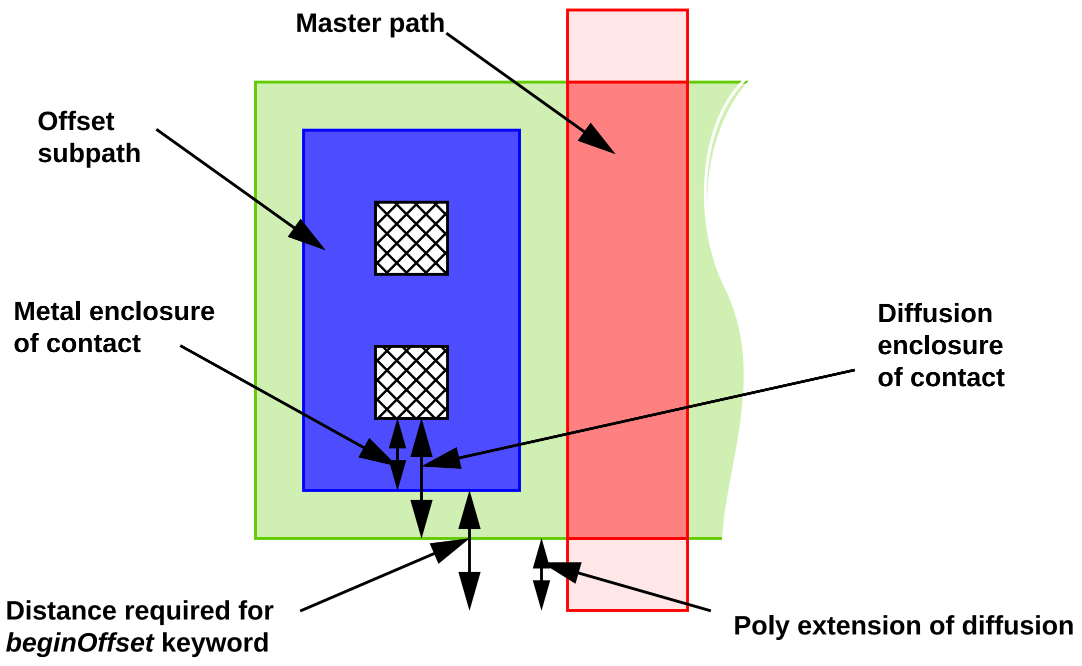
2. Locate the *sep* keyword argument in the first offset subpath specification. You must specify the distance between the left edge of the poly stripe and the right edge of the drain metal.

   Refer to the diagram below to determine the expression for this distance:


Poly to contact separation
Master path
Metal enclosure of contact
Offset subpath
Distance required for *sep* keyword

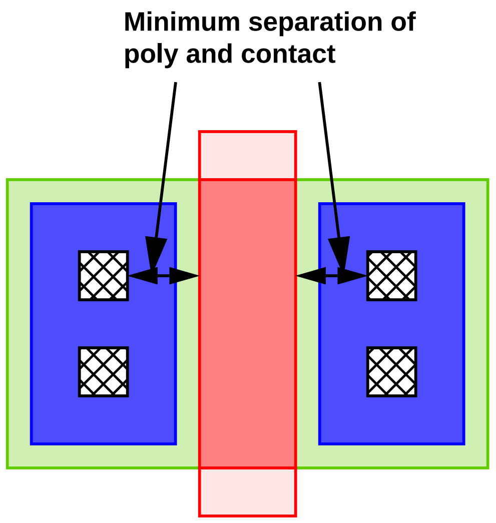
3. In this subpath definition, you must also specify the beginning offset. As with the diffusion region specification, you need only specify the beginning offset here as the ending offset defaults to the same value.

   Refer to the diagram below to determine the expression for this distance:


Master path
Offset subpath
Metal enclosure of contact
Diffusion enclosure of contact
Distance required for *beginOffset* keyword
Poly extension of diffusion

4. Locate the second offset subpath definition. Enter the values for the *sep* and *beginOffset* keyword arguments that you determined for the first offset subpath definition.

5. Locate the subRect keyword argument. This section defines the source and drain contact arrays. In each sub-rectangle definition, you must supply the separation between the contacts and the poly stripe as shown below:



Notice that each separation is a positive value. Also notice that the *justification* keyword argument places each rectangle array on the proper side of the poly stripe.

6. Save your changes to the file.

7. In the CIW, enter the following command:

```
load “step4.il”
```

Did you see any errors or warnings? If so, try to rectify the problem, save your changes to the file, and repeat this step.

Test and Examine the Pcell

1. Create an instance of *pcells step4 layout* in the layout editor window.

   Does the instance exactly resemble the instance from the previous section? If not, examine the changes you made to the source code, make appropriate edits and repeat the two previous steps.

2. Select this instance and invoke the property editor.

3. Change the *width* and *length* parameters for this instance to various values and observe the results. Does the instance behave the same as the previous instance?

Important

If you had difficulty with this section, you may refer to this file to see a complete solution:

```
source/step4.il
```

Prepare for the Next Section

Close the text editor for the file *step4.il*.

Lab 4-5 Experimenting with Process Independence

Objective: Observe the process independence built into the parameterized cell.

The parameterized cells implemented in the previous sections have used technology file access so as to make the resulting cell adaptable to process changes. However, this process independence has not been demonstrated in the course of the previous sections. You now make changes to the technology information in the *pcells* library and observe the effects to the pcell from the last section.

1. In the layout editor window, select the instance of *pcells step4 layout* and invoke the property editor.

2. Set the *width* parameter to 7.0 and the *length* parameter to 1.0.

3. Zoom in on this instance so that it fills the window.

4. Open a text editor on the technology file called *techfile* you created earlier. Make these changes:

   - Set *minEnclosure* of *cont by pdiff* to *0.9*.
   - Set *minWidth* for the *cont* layer to *1.0*.
   - Set *minSpacing* for the *cont* layer to *0.8*.

5. Save your changes to *techfile*.

6. Load your changes into the Design Framework II session (in the CIW, choose **Technology File—Load**, enter *techfile* in the **ASCII Technology File** field, enable the **Select All** option under the **Classes** field, and click **OK**).

7. The pcell from the last section must be recompiled to obtain these changes in the library's technology. Enter this command into the CIW:

   ```
   load “step4.il”
   ```

8. While observing the instance of *pcells step4 layout*, select **Window—Redraw**. What happened?

9. Measure these separations and dimensions to verify that the changes made to the library technology are now reflected in the pcell instance:

   - Contact-to-contact spacing
   - Contact dimensions
   - Diffusion pin location and dimensions
   - Diffusion extension beyond contact.

10. Revert to the original technology information by selecting **Technology File—Discard** in the CIW.

11. When the form appears, select *pcells* in the **Technology Library** field and click **OK**.

12. A dialog box appears asking you to confirm your actions. Click **OK** in the form.

13. Recompile the pcell by entering this command into the CIW:

    ```
    load “step4.il”
    ```

14. While observing the instance of *pcells step4 layout*, select **Window—Redraw**. What happened? Does the transistor appear as it did before the changes to the technology?

Prepare for the Next Lab

Execute the following steps to prepare for the next lab exercises:

1. In the CIW, choose **File—Exit** and click on **Yes** in the dialog box that appears.

   This command closes the Design Framework II session and all of the windows associated with it.

2. In the shell, enter the following command:

   ```
   cd ~
   ```

   This command places you in your home directory.

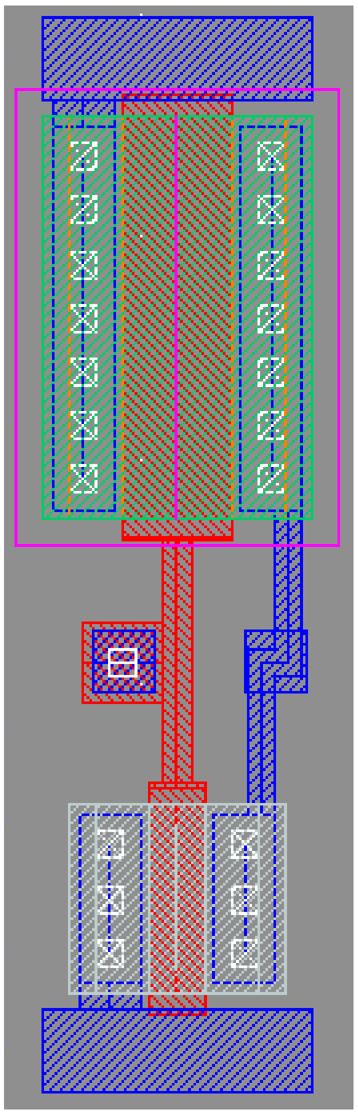
Summary

You began these exercises by compiling your first SKILL parameterized cell. Although this first cell was merely a parameterized rectangle, it demonstrated the steps necessary to create a pcell from a source code file. You then expanded upon this simple example to form a rudimentary, parameterized transistor. In the final sections of the lab, you added more functionality, and utilized more and more sophisticated features of relative object design.

End of Lab

Labs for Module 5

Going Further with SKILL Pcells



Before You Begin Module 5 Labs

In these exercises, you begin each section with a SKILL source code file containing nearly all the information necessary to create a pcell. Your task is to determine the information missing in the body of the pcell.

Each missing piece of information has been replaced by a descriptive token surrounded by ‘#’ characters. For example, suppose the name of a layer is required at one point in the code, and your task is to specify this name as *metal1*. You might find a line in the file like this:

```
lpp = list(“#layerName#” “drawing”)
```

To accomplish your task, you would change the line to the following:

```
lpp = list(“metal1” “drawing”)
```

Hint: You can locate all of the missing information tokens by repeatedly searching for the ‘#’ character.

The pcell may not compile or behave properly if any missing information tokens remain in the source file. Only after you have supplied all missing information will the pcell compile and behave correctly.

Choose a Text Editor

You will need to edit source code files in this lab. You may choose from the following editors:

- vi
- textedit

If you are unfamiliar with these editors, choose *textedit* as it implements simple point-and-click editing.

Cadence Online Documentation

The exercises in this lab deal with relative object design in significant detail, and you will be required to refer to the *Virtuoso Relative Object Design User Guide* in many sections of this lab. Therefore, open this documentation prior to beginning the exercises.

Enter the following command into the shell:

```
cdsdoc
```

When the documentation window appears, click on the Layout Editor folder, then click on Virtuoso Relative Object Design User Guide.

Lab 5-1 Pcell Hierarchy

Objective: Learn how to create hierarchy in a pcell and observe ROD point transformation through hierarchy.

In this exercise you create instances of the *tran* pcell within a new pcell called *inv*. You then learn how to access points on objects throughout the hierarchy and observe how ROD performs hierarchical coordinate transformation.

Start a Design Framework II Session

1. Enter these commands into a terminal window:

   ```
   cd ~/Skill_PCells_5_1_41/morePcells
   layout &
   ```

   After a few moments, the Command Interpreter Window (CIW) appears followed by a new (empty) layout editor window for the cellview *stdcells test layout*.

2. Choose **Tools—Technology File Manager** from the menus in the CIW.

3. When the Technology File Tool Box appears, select **Dump**.

4. When the Dump Technology File form appears, select the following and click **OK**:

Technology Library	`stdcells`
Classes	`Select All`
ASCII Technology File	`techfile`

   A readable form of the technology information will now reside within the file called *techfile*.

   Many of the exercises in this lab deal with technology information stored in the *stdcells* library. You will find it helpful to have this information easily accessible in a text file.

   Note: The technology for this library differs from the technology of the libraries in the previous exercises.

Define and Examine the Transistor Cell

The cell you define requires two transistors. In these exercises, you use a variation of the transistor pcell created in the previous lab. You must compile the definition for this cell in the *stdcells* library.

1. Open a text editor for this file:

   ```
   tran.il
   ```

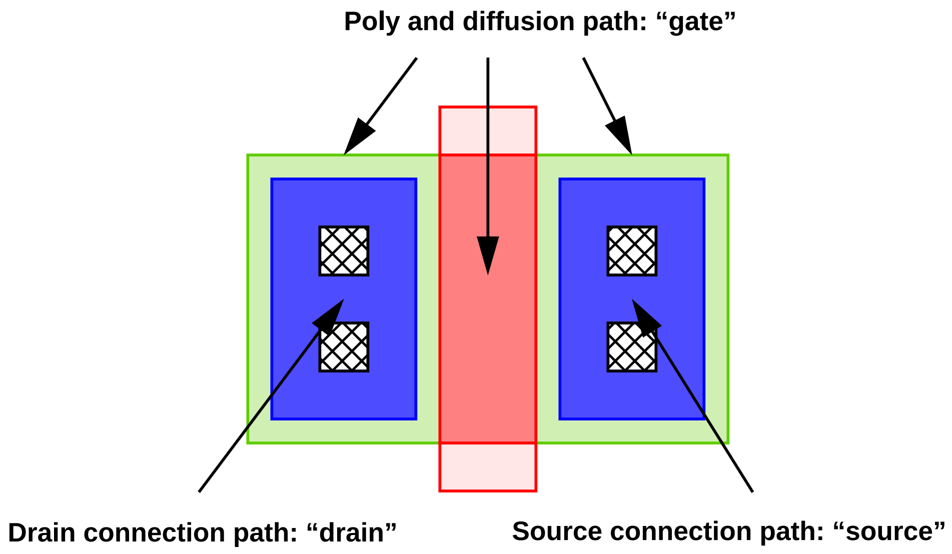
   The definition of the transistor pcell resides in this file.

2. Examine the comments at the beginning of this file. Make sure you understand the purpose of this cell.

   For the sake of demonstrating the capabilities of ROD in a hierarchical pcell, the transistor cell you use in this lab is implemented as three separate multipart paths:

   - The poly and diffusion regions
   - The drain connection, with contact array
   - The source connection, with contact array

   Each path is assigned a unique name that will be used in the inverter cell. The paths and their names can be seen in the diagram below:


Poly and diffusion path: "gate"
Drain connection path: "drain"
Source connection path: "source"

3. Compile the transistor cell by entering the following command into the CIW:

   ```
   load “tran.il”
   ```

   This command creates a new cellview called *stdcells tran layout.*

4. Open this cellview in a layout editor window.

   To do this, choose **File—Open** in the CIW. When the **Open File** form appears, select these options and click **OK** in the form:

Library Name	`stdcells`
Cell Name	`tran`
View Name	`layout`
Mode	`read`

   The cellview appears in a layout editor window.

5. Enter this command into the CIW:

   ```
   cv = geGetWindowCellView()
   ```

   This assigns the database ID for the current cellview to the variable called *cv*.

6. Enter this command into the CIW:

   ```
   shapes = rodGetNamedShapes(cv)
   ```

   This assigns a list of ROD objects for the current cellview to the variable called *shapes*.

7. Enter this command into the CIW:

   ```
   shapes~>name
   ```

   This returns a list of the names assigned to these shapes. *What are these shapes? Are they actually shapes, or another type of object?*

8. Search in the source file for this cell and discover where each of these names is assigned.

9. Enter this command into the CIW:

   ```
   gateObj = rodGetObj(“gate” cv)
   ```

   This assigns the ROD object ID for the multipart path called *gate* to the variable called *gateObj*.

10. Verify that the ROD object ID returned is one of those returned by *rodGetNamedShapes* by entering this command into the CIW:

    ```
    member(gateObj shapes)
    ```

    A non-nil return value indicates that the ROD object ID assigned to *gateObj* appears in the list called *shapes*.

    IMPORTANT: Notice that you can use *rodGetNamedShapes* to determine the names of the objects in a given cellview and that you can use *rodGetObj* to obtain a specific named object.

11. Enter this command into the CIW:

    ```
    gateCenter = gateObj~>centerCenter
    ```

    You use this value later to observe the hierarchical point transformation performed by ROD.

12. Close the layout editor window for *stdcells tran layout*.

    To do this, choose **Window—Close** in the layout editor window.

13. Close the text editor for *tran.il*.

Create and Examine Instances of the *tran* Cell

Now that the *tran* cell has been defined, you create two instances of the cell within the *inv* cell definition.

1. Open a text editor for the file *inv1.il*.

2. Examine the opening comments and the parameter list of this cell.

3. Locate the first call to *dbCreateParamInst*. Study this call and answer these questions:

   a. Which cellview is instantiated here?

   b. In which cellview is it instantiated?

c. What is the name of the instance?

d. What are the coordinates of instance?

e. Which parameters of the master cellview are specified?

f. Which other parameters will assume default values?

Use this call as an example to complete the P-transistor instance.

4. Locate the second call to *dbCreateParamInst*. This call creates the P-transistor instance.

5. Supply these items as prescribed by the missing information token:

   - Y-ordinate of the instance's location
   - Cell parameter for the P-transistor's width
   - Cell parameter for the P-transistor's length
   - Cell parameter for the P-transistor's diffusion layer

6. Locate the call to *dbClose* immediately below the second call to *dbCreateParamInst*.

   IMPORTANT: Always close any cellviews opened with *dbOpenCellViewByType* before the end of the call to *pcDefinePCell*.

   In the example of the inverter pcell, the *stdcells tran layout* cellview is closed as soon as it is no longer needed.

7. Locate the first call to *rodGetObj*.

   This statement assigns the ROD object ID (*not* the database ID) for the N-transistor instance to a local variable.

   Below this line in the source file are several similar statements collecting ROD object IDs for subcomponents of the two transistor instances. These ROD object IDs are crucial in the construction of the inverter pcell for aligning objects and performing various calculations.

8. Using the first two calls to *rodGetObj* as examples, supply the missing pathnames for the remaining statements.

9. Save your changes to the file.

10. Compile the inverter cell by entering the following command into the CIW:

    ```
    load “inv1.il”
    ```

    Did you see any errors or warnings? If so, try to rectify the problem, save your changes to the file, and repeat this step.

Important

If you had difficulty with this section, you may refer to this file to see a complete solution:

```
source/inv1.il
```

Investigate ROD Points in a Hierarchy

The inverter pcell contains two instances of the *tran* cell. You now enter commands into the CIW to observe how ROD calculates points on objects through hierarchy.

1. Open this cellview in a layout editor window.

   To do this, choose **File—Open** in the CIW. When the **Open File** form appears, select these options and click **OK** in the form:

Library Name	`stdcells`
Cell Name	`inv`
View Name	`layout`
Mode	`read`

   The cellview appears in a layout editor window.

2. Enter this command into the CIW to get the database ID of the cell:

   ```
   cv = geGetWindowCellView()
   ```

3. Enter this command into the CIW:

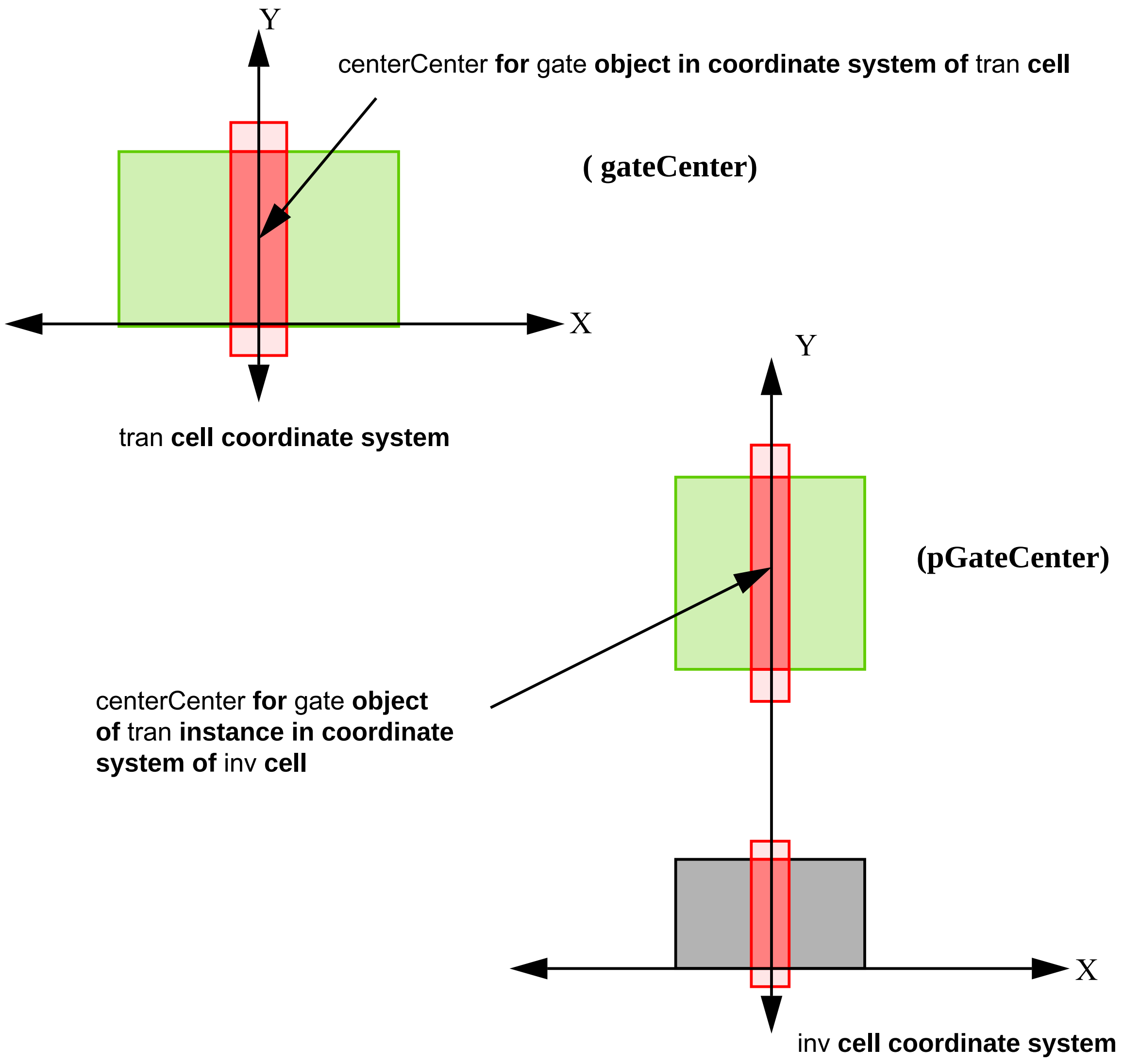
   ```
   pGateObj = rodGetObj("ptran/gate" cv)
   ```

   This assigns the ROD object ID for the multipart path called *gate* to the variable called *pGateObj*.

4. Enter these commands into the CIW:

```
pGateCenter = pGateObj~>centerCenter
gateCenter
```

Notice that the coordinates for the center of the *tran* instance's gate path are automatically transformed to the coordinate system of the *inv* cellview. Consider the diagram below:


Y
centerCenter for gate object in coordinate system of tran cell
(gateCenter)
X
tran cell coordinate system
Y
(pGateCenter)
centerCenter for gate object of tran instance in coordinate system of inv cell
X
inv cell coordinate system

The point transformation feature of ROD is used often in the inverter pcell constructed in this lab.

5. Close the layout editor window for *stdcells inv layout*.

6. Close the text editor for *inv1.il*.

7. Bring forward the layout editor window for *stdcells test layout.*

8. Create an instance of *stdcells inv layout* in this window. Supply the name *I1* for this instance.

   To do this, choose **Create—Instance** in the layout editor window. In the form that appears, fill in this information:

Library	`stdcells`
Cell	`invy`
View	`layout`
Names	`I1`

   Click in a location of your choice in the layout editor window to place the instance, then click **Cancel** on the form.

9. Enter this command into the CIW:

   ```
   cv = geGetWindowCellView()
   ```

10. Enter this command into the CIW:

    ```
    invPgateObj = rodGetObj("I1/ptran/gate" cv)
    ```

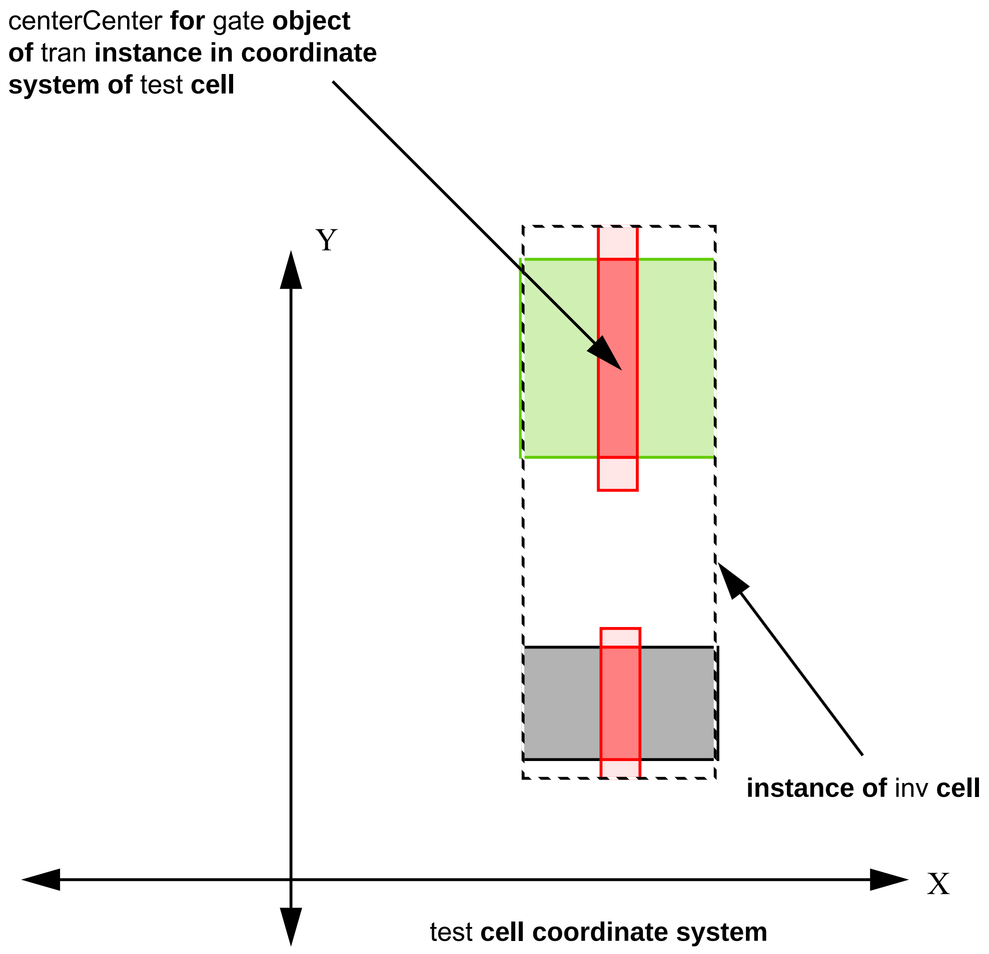
    This assigns the gate object of the P-transistor contained in the inverter instance to the variable called *invPgateObj*.

    Notice the ease of accessing the desired object by means of a simple pathname rather than descending through the hierarchy manually using database calls.

11. Enter these commands into the CIW:

```
invPgateObj~>centerCenter
pGateCenter
gateCenter
```

Notice that the coordinates for the center of the *tran* instance's gate path are automatically transformed to the coordinate system of the *test* cellview. This is illustrated in the diagram below:


centerCenter **for** gate **object of** tran **instance in coordinate system of** test **cell**
Y
X
instance of inv **cell**
test **cell coordinate system**

12. Enter this command into the CIW:

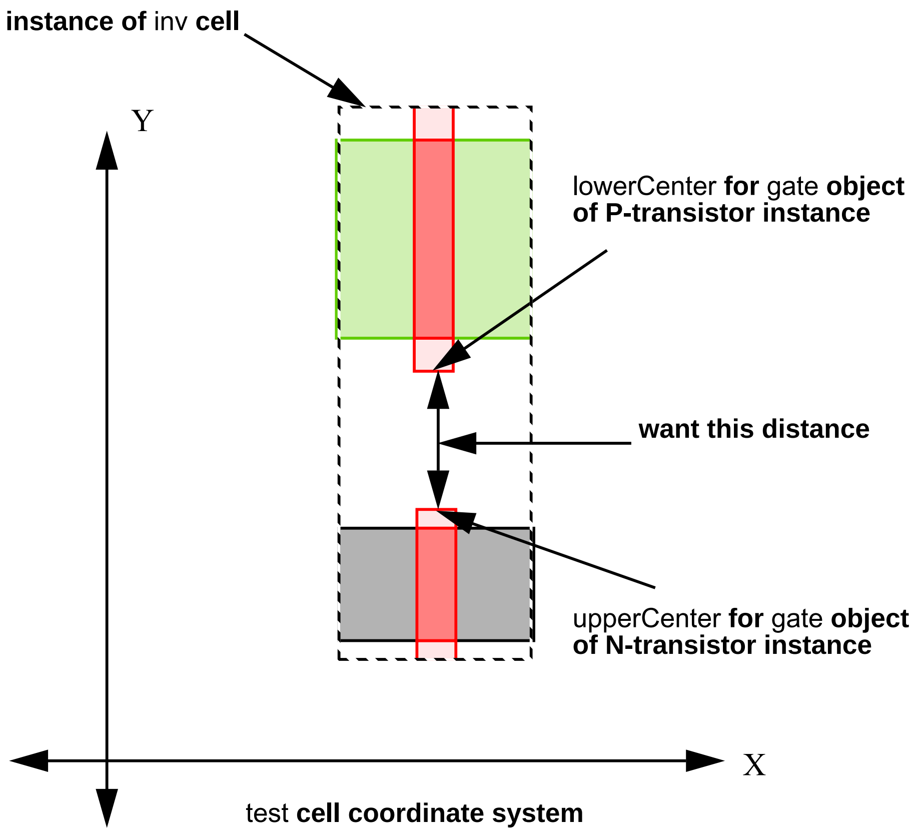
```
invNgateObj = rodGetObj("I1/ntran/gate" cv)
```

This assigns the ROD object ID for the N-transistor's *gate* object to the variable called *invNgateObj*.

13. Assume you wanted the X-ordinate of the N-transistor's *gate* object center point. Enter this command into the CIW to observe the utility of *rodPointX*:

```
rodPointX(invNgateObj~>centerCenter)
```

14. Assume you want to obtain the vertical distance between the top of the N-transistor's *gate* object and the bottom of the P-transistor's *gate* object.


instance of inv cell
Y
lowerCenter for gate object of P-transistor instance
want this distance
upperCenter for gate object of N-transistor instance
X
test cell coordinate system

Enter this command into the CIW to observe the utility of *rodSubPoints* and of *rodPointY*:

```
rodPointY(
   rodSubPoints(invPgateObj~>lowerCenter
                invNgateObj~>upperCenter)
)
```

15. Use the ruler feature (**Window—Create Ruler**) to verify the distance returned by the previous step.

Lab 5-2 Using ROD Points in Hierarchy

Objective: Use ROD object points throughout the inverter hierarchy to define and align structures within the inverter.

In the previous section, you learned how to access ROD objects and points on the objects throughout a hierarchy. Now you use this ability to create and align structures within the inverter cell.

Examine the Source Code

1. Open a text editor on this file:

   ```
   inv2.il
   ```

2. Read the comments at the beginning of the file.

   Notice that there are many new parameters for this version of the inverter.

3. Find the initial *let* statement in the file.

   Notice the large number of local variables. This was done to ensure no variables were accidentally made global.

4. Look below the *let* statement in the technology access section.

   Notice that many items from the technology file are required for this cell.

5. Search forward for the line that begins with the following:

   ```
   minHeight =
   ```

   Study the calculation and answer these questions:

   a. What distance is calculated here?

   b. Why is it necessary?

   c. What object dimensions drive the minimum height for the inverter cell?

   d. What design rules are involved in this calculation?

6. Locate the *when* statement below this calculation.

   What is the purpose of this statement?

7. Examine the *if* statement below the *when* statement.

   Answer these questions:

   a. What distance is calculated here?

   b. Why is it necessary?

   c. What object dimensions drive the minimum width for the inverter cell?

   d. Are any design rules involved in this calculation?

Align the Transistor to the Supply Rail

With the supply rails properly aligned, you now align the N- and P-transistors to their respective supply rails.

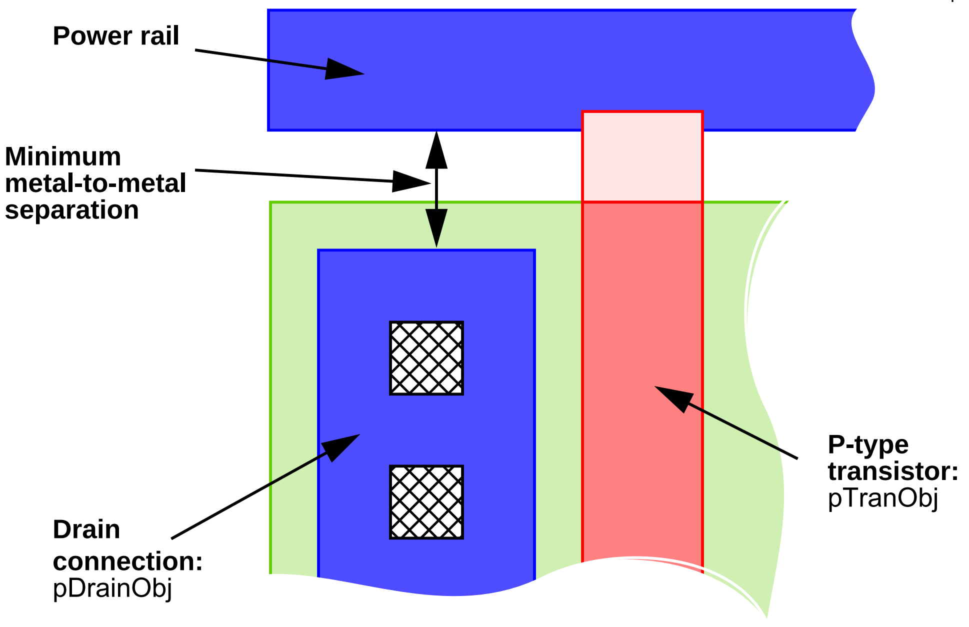
1. Open a text editor on this file:

   ```
   inv2.il
   ```

2. Page forward to the end of the file.

3. Locate the *rodAlign* call for the P-transistor.

   This statement centers the P-transistor under the power rail with just enough separation between the power rail and the transistor's source and drain connections, as depicted below:


Power rail
Minimum metal-to-metal separation
P-type transistor: pTranObj
Drain connection: pDrainObj

4. Examine the keyword arguments in this statement.

   Answer these questions:

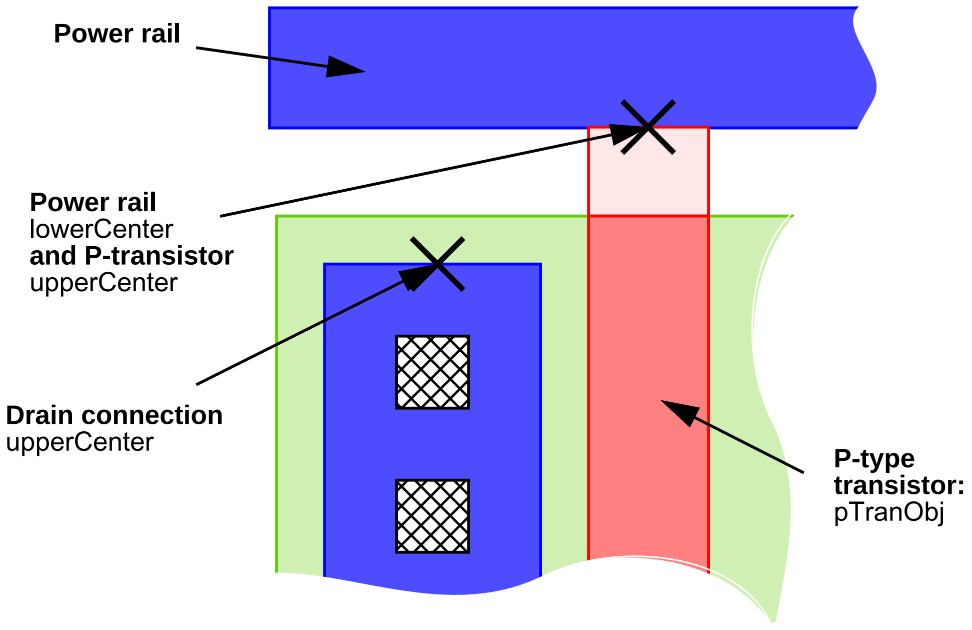
   a. Which two objects are being aligned?

   b. Which two points are used for this alignment?

   c. Is it necessary to supply a vertical separation to this alignment?

5. Supply the missing information for the *ySep* keyword argument.

   With no vertical separation specified, this statement would abut the upper-center of the P-transistor to the lower-center of the power rail, as depicted below:


Power rail
Power rail lowerCenter and P-transistor upperCenter
Drain connection upperCenter
P-type transistor: pTranObj

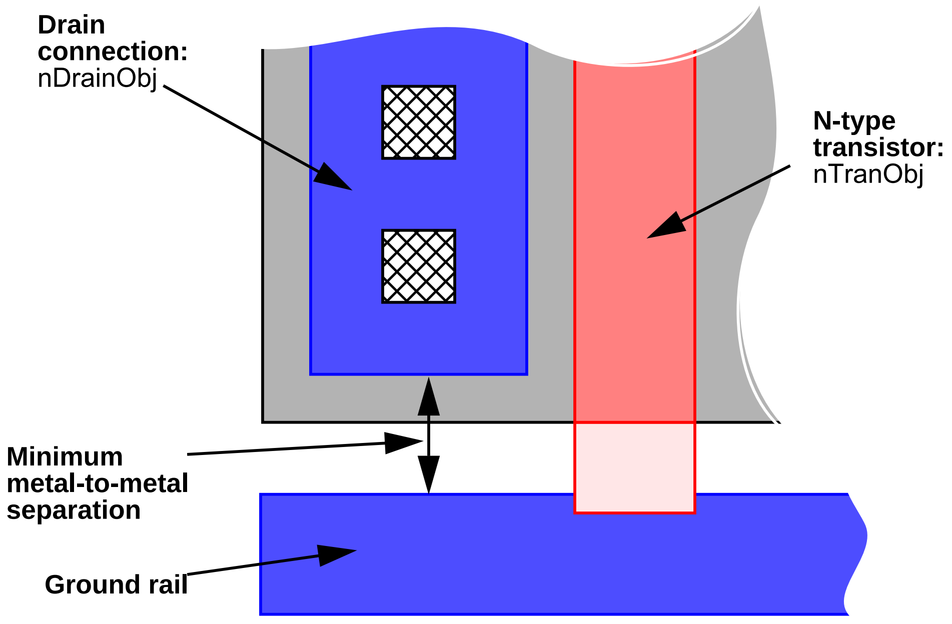
   The critical separation is the metal-to-metal spacing between the drain connection and the power rail.

   Hence, you need to subtract the metal-to-metal spacing rule from the vertical distance between the upper-center of the P-transistor and the upper-center of its drain connection.

   The essence of this calculation is already provided. You need to supply the appropriate spacing rule variable, and the two points mentioned in this discussion in proper order.

6. Supply similar information for the N-transistor alignment statement.

   Be aware that you are now aligning the lower-center of the N-transistor to the upper-center of the ground rail, as depicted below:


Drain connection: nDrainObj
N-type transistor: nTranObj
Minimum metal-to-metal separation
Ground rail

   Therefore, you need to use the lower-center points of the N-transistor and its drain connection in this calculation.

7. Save your changes to the file.

8. Compile the inverter cell by entering the following command into the CIW:

   ```
   load “inv2.il”
   ```

   Did you see any errors or warnings? If so, try to rectify the problem, save your changes to the file, and repeat this step.

9. In the layout editor window for *stdcells test layout*, redraw the window (**Window—Redraw**) to see the changes to the *inv* cell.

10. Look for the *minSpacing* rule for *metal1* in the technology file created at the beginning of these exercises.

11. Zoom in to the area between the P-transistor and the power rail. Verify that the distance between the transistor's drain connection and the power rail is exactly this minimum spacing distance.

12. If the distance is not correct, attempt to rectify the problem and repeat the previous five steps.

13. Perform the same distance verification on the N-transistor.

Important

If you had difficulty with this section, you may refer to this file to see a complete solution:

```
source/inv2.il
```

Close the text editor for *inv2.il*.

Add Gate and Drain Connections

With the transistors properly aligned and the gate connection completed, you now add connections from the drains of each transistor to its associated supply rail.

1. Open a text editor on this file:

   ```
   inv3.il
   ```

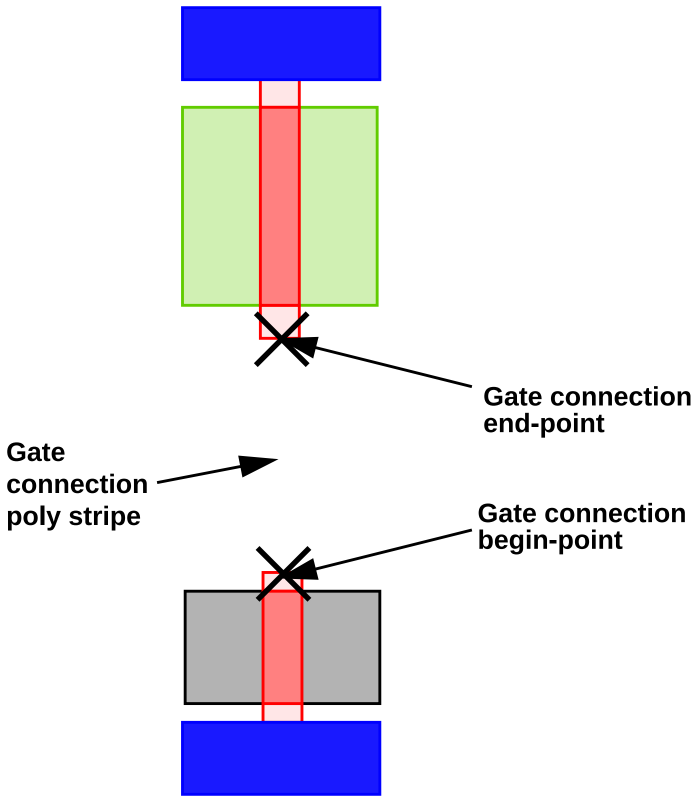
2. Locate the first call to *rodCreatePath*.

   This statement defines the poly connection between the gates of the two transistors.

   The gate connection is implemented as a simple path. The beginning and ending points of this path are obtained through hierarchy from the transistor instances.

3. Supply the beginning and ending point for this path.

   Refer to the following diagram to determine the proper points:


Gate connection end-point
Gate connection poly stripe
Gate connection begin-point

   Hint: The necessary points lie on the bounding boxes of *nGateObj* and *pGateObj*.

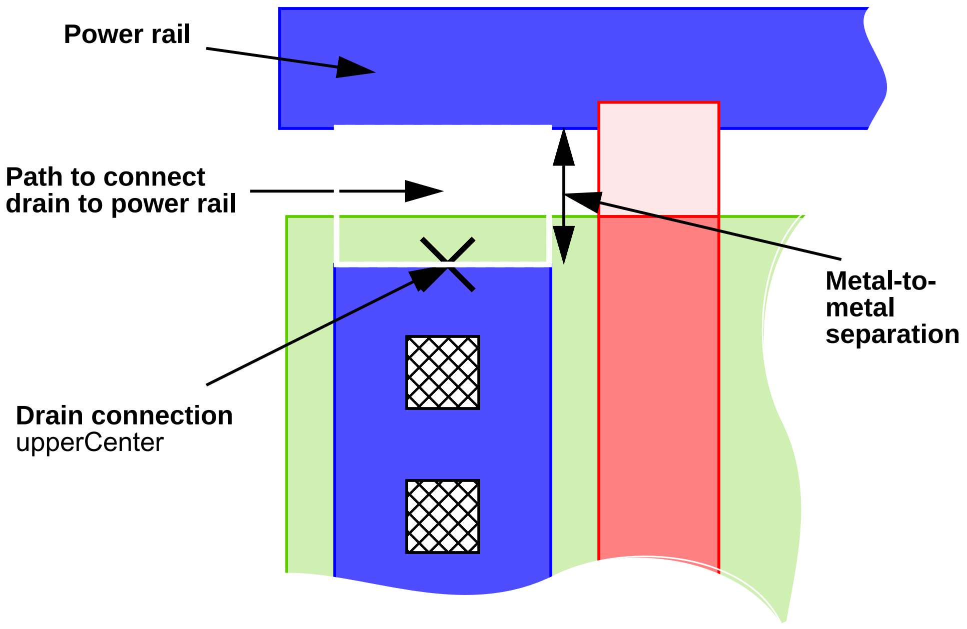
4. Locate the next call to *rodCreatePath*.

   This statement defines the connection between the power rail and the P-transistor drain.

5. Supply the starting and ending point for this path.

   You need to abut this path to the upper-center point of the P-transistor's drain connection. The length of this connection will be equal to the metal-to-metal spacing (recall the calculation to offset the P-transistor as it was aligned to the power connection).

   Refer to the diagram below to determine the proper point:


Power rail
Path to connect drain to power rail
Metal-to-metal separation
Drain connection
upperCenter

   Notice that the end-point for the path is calculated by adding the metal-to-metal spacing to the Y-ordinate of the point chosen for the start of the path.

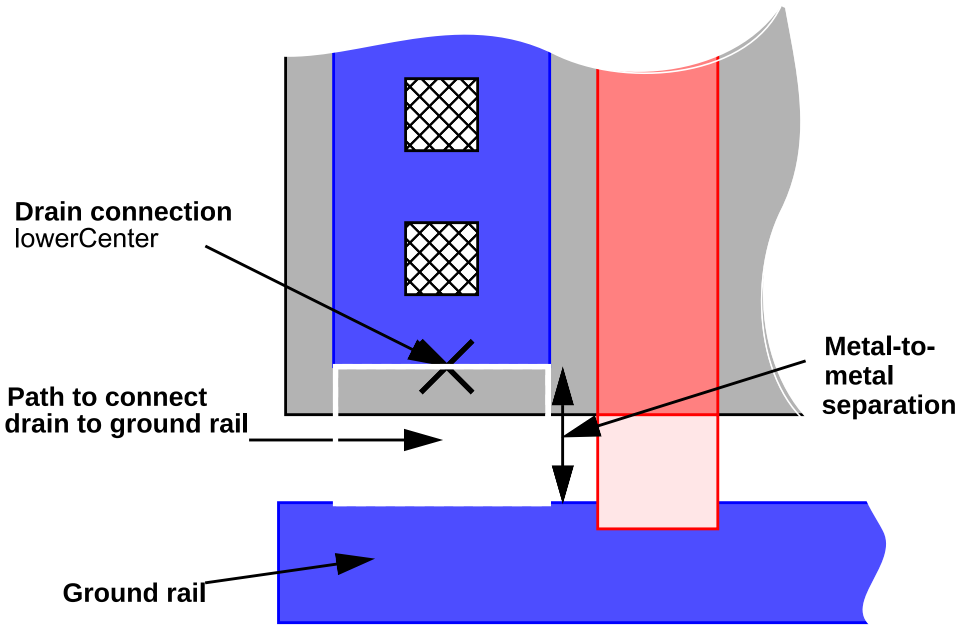
6. Locate the next call to *rodCreatePath*.

   This statement defines the connection between the ground rail and the N-transistor drain.

7. Supply the starting and ending point for this path.

   These points are chosen in a similar fashion to the drain/power connection.

   Refer to the diagram below to determine the proper point:


Drain connection
lowerCenter
Path to connect drain to ground rail
Metal-to-metal separation
Ground rail

8. Save your changes to the file.

Examine Other Structures Comprising the Inverter

Once you have completed the gate and drain connections, a number of other structures need to be defined to complete the inverter. These include:

- N-well for the P-transistor
- Input pin
- Source connection
- Output pin

You now examine the code used to create these structures to understand the approach used.

1. Locate the next call to *rodCreatePath*.

   This statement creates an N-well around the diffusion region of the P-transistor. Notice that this structure is implemented as an path with extended ends.

   Examine this call to *rodCreatePath* and answer the following questions:

   a. For this extended end-type path, what design rule is used to determine the extensions?

   b. What information is used to determine the width of the path?

   c. How is the location and vertical dimension of this path linked to the location and dimensions of the P-transistor?

2. Locate the next call to *rodCreatePath*.

   This statement creates an input pin structure complete with a first-layer metal pad and contact. Notice that this is a multipart path which defines objects on three layers.

   Examine this call to *rodCreatePath* and answer the following questions:

   a. Does this path employ extended ends?

   b. Which three layers are used?

   c. What other object is used to specify the vertical location of this structure?

3. Locate the next call to *rodCreatePath*.

   This statement creates a connection between the sources of the transistors.

   Examine this call to *rodCreatePath* and answer the following questions:

   a. How many points are used to define this path? Why?

   b. Where are the end-points for this path anchored?

4. Locate the call to *rodCreateRect*.

   This statement creates an output pin structure. Notice that this is a simple rectangle on first-layer metal. The rectangle is moved to the proper place in the cellview by the subsequent call to *rodAlign*.

   Examine the call to *rodAlign* statement. What is the purpose of the *if* statement specified as the value of the *refHandle* keyword argument?

Compile and Test the Inverter Cell

1. Compile the inverter cell by entering the following command into the CIW:

   ```
   load “inv3.il”
   ```

   Did you see any errors or warnings? If so, try to rectify the problem, save your changes to the file, and repeat this step.

2. In the layout editor window for *stdcells test layout*, redraw the window (**Window—Redraw**) to see the changes to the *inv* cell.

3. Zoom in to the area between the P-transistor’s drain and the power rail. Verify that the connection between these structures is correct.

4. Zoom in to the area between the N-transistor’s drain and the ground rail. Verify that the connection between these structures is correct.

5. If you found problems with the structures defined in this section, attempt to rectify them and repeat the previous four steps.

Important

If you had difficulty with this section, you may refer to this file to see a complete solution:

```
source/inv3.il
```

Close the text editor for *inv3.il*.

Lab 5-3 Adding a CDF Parameter to the Inverter Cell

Objective: Investigate the use of Component Description Format (CDF) with parameterized cells.

You now use CDF functions to add a new parameter called *size* to the inverter pcell. This parameter allows you to choose from a finite list of drive strengths for the cell. Hence, rather than entering discrete values for the length and width of the transistors within the inverter, you simply select a specific size, such as *A* or *D*.

Examine the Source Code for the Inverter CDF

1. Open a text editor on this file:

   ```
   invCDF.il
   ```

   This file contains code to add CDF parameters to the inverter cell and to define a SKILL function called *SPCsetInvSize*. The function serves as the call-back for the new parameter called *size*.

2. Examine the *let* statement near the top of the file and answer the following questions:

   a. Which cell is affected by this code?

   b. How is the base-CDF created for the cell?

   c. What check is made before the base-CDF is created and why?

   d. Which function call creates the *size* parameter?

   e. Which keyword argument in the call specifies the label used for the parameter when it is displayed?

   f. Which function is called when the value of this parameter changes?

   g. What is accomplished by the *foreach* loop near the end of the function definition?

   h. Which functions are called to perform clean-up prior to the end of this function?

3. Examine the definition of the procedure called *SPCsetInvSize* and answer the following questions:

   a. How is *cdfgData* defined?

   Hint: Supply the name of this variable to **Search—All** in the Cadence® online documentation to determine its purpose.

   b. Which section of the function handles the size selection?

   c. How is a size selection affected in the pcell?

Create and Experiment with the Inverter CDF

1. ***If*** your version of *stdcells inv layout* did **not** function properly at the end of the previous lab section, enter this command into the CIW:

```
load “source/inv3.il”
```

2. Enter this command into the CIW:

```
load “invCDF.il”
```

   This command creates CDF parameters for the inverter cell and defines the callback function *SPCsetInvSize*.

3. In the layout editor window for *stdcells test layout*, select the inverter instance.

4. Invoke the property editor (**Edit—Properties**) and choose the **Parameter** option on the property editor form.

   Notice the new **Size** field on the form. Is its current value as you expected from the code in the *let* statement?

5. Change the **Size** field to *A* and click **Apply** on the property editor form.

   Observe the change to the inverter instance in the layout editor window. If you desire, you can measure the active areas of the N- and P-type transistors in the inverter cell to verify that they match the values specified in the code for *SPCsetInvSize.*

6. Change the **Size** field to other values as in the previous step and observe the behavior of the inverter. Does it behave as you expected?

7. **Optional activity**: modify the definition of *SPCsetInvSize* so that the supply rail width is increased by 0.4 units for the *D*, *E*, and *F* sizes. Load the changed code and test the cell.

8. Close the text editor for *invCDF.il*.

Lab 5-4 Pcell Stretch Handles

Objective: Investigate pcell stretch handles and learn various stretch handle options.

With the introduction of relative object design to the layout editor, you now have the option to make certain aspects of your pcells modifiable in a graphical fashion. This capability is known as *pcell stretch handles*. In this section, you add stretch handles to a simple pcell and experiment with various stretch handle options.

Complete the Stretch Handle Example

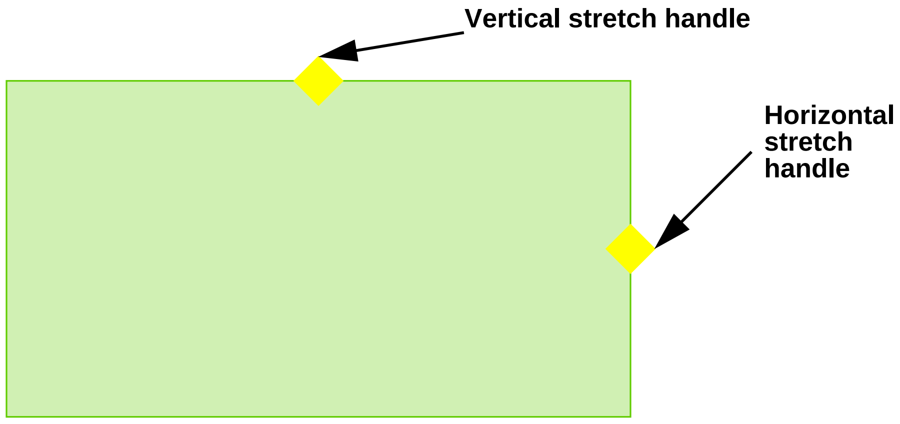
In this exercise, you specify the parameter and handle names to implement rudimentary stretch handles.

1. Open a text editor on this file:

   `rect1.il`

   This file contains code to define a simple pcell (a programmable rectangle). The *rodAssignHandleToParameter* function is used within the pcell definition to add stretch handles for two of the pcell's parameters.

   The goal is to create stretch handles as depicted below:

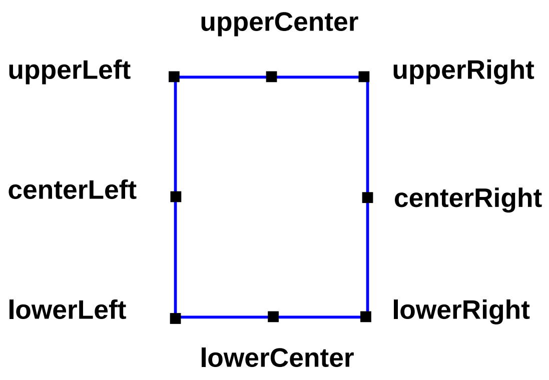


2. Page forward to the bottom of the file.

   Examine the first *rodAssignHandleToParameter* statement. You must supply the parameter name controlling the X-dimension of the rectangle.

3. In this same call to *rodAssignHandleToParameter*, you must also supply the name of the handle on the rectangle to which the stretchability will be assigned.

   Refer to the diagram below to determine the appropriate point on the bounding-box for this handle:



ROD bounding box point handles

4. Locate the second *rodAssignHandleToParameter* statement. Supply appropriate values for the *parameter* and *handleName* keyword arguments similar to those you specified in the first *rodAssignHandleToParameter* statement.

5. Save your changes to the file.

Compile and Test the Rectangle Cell

1. Enter this command into the CIW:

   ```
   load “rect1.il”
   ```

   This command creates the rectangle pcell.

   Did you see any errors or warnings? If so, try to rectify the problem, save your changes to the file, and repeat this step.

2. Create an instance of the cell *stdcells rect layout*.

   To do this, choose **Create—Instance** in the layout editor window. In the form that appears, fill in this information:

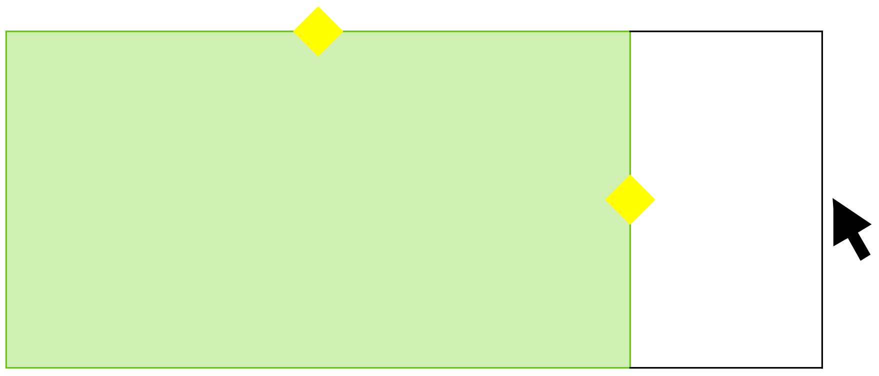
Library	`stdcells`
Cell	`rect`
View	`layout`

   Click in a location of your choice in the layout editor window to place the instance, then click **Cancel** on the form.

3. Zoom in on the instance if necessary (**Window—Zoom—In**).

4. Choose **Edit—Stretch** from the menus in the layout editor window.

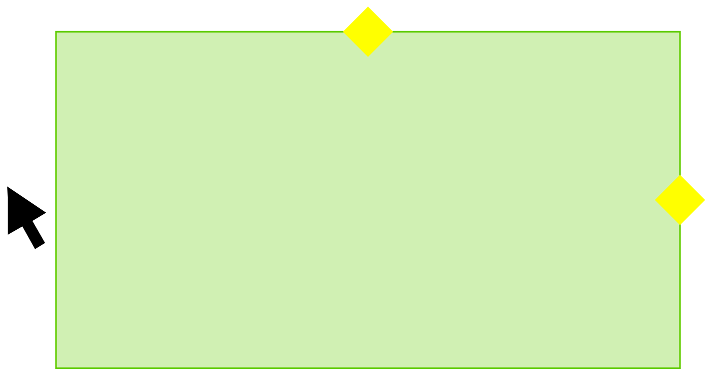
5. Place the pointer over the center of the horizontal stretch target and click the left-mouse button. Move the pointer in the positive X direction (towards your right). As you move the pointer, you will see an outline of the rectangle follow the pointer motion, as depicted below:



Note: If the outline of the rectangle does not appear or the entire instance begins to follow the motion of the pointer, you may have missed the stretch target. In this case, hit the Escape key and start with step 4 again.

6. When the rectangle is the size you desire, click the left mouse button again. The rectangle takes on the new horizontal dimension you specified with the stretch handle.

7. Repeat the previous three steps. For this experiment, drag the stretch handle in the *negative* X direction (towards your left) so that the pointer is beyond the left-most edge of the rectangle as depicted below:



8. Examine the contents of the CIW.

   You will see one or more groups of error messages similar to this:

   ```
   *WARNING* Pcell evaluation for
   stdcells/rect/layout has the following error(s):
   *WARNING* ("error" 0 t nil ("*Error*
   rodCreateRect: command failed\n"))
   *WARNING* Error kept in "errorDesc" property of
   the label "pcellEvalFailed" on layer/purpose
   "marker/error" in the submaster.
   ```

   Can you determine what happened as you stretched the pcell beyond its left-most edge?

   As you dragged the pointer beyond the left-most edge of the rectangle instance, the system attempted to create a rectangle with a negative width. This results in an error, signified by the messages in the CIW.

9. Press the **Escape** key to cancel the current stretch operation.

10. Experiment with the vertical stretch handle to ensure that it functions properly. If it does not, attempt to fix the code, save and load your changes, and test the instance once again.

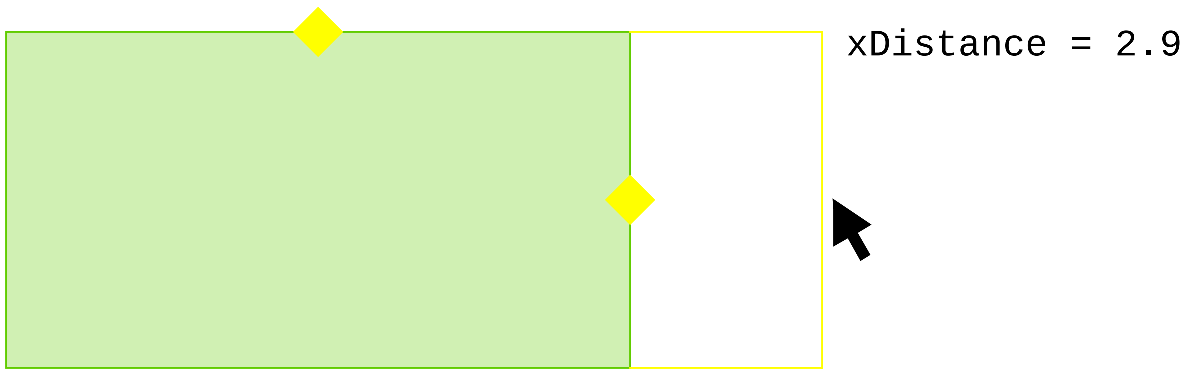
Add Visual Feedback to the Stretch Handles

In the previous section, the stretch handles allowed you to interactively resize an instance of the rectangle pcell. However, it was difficult to tell the exact value entered for the stretched parameter. You now improve the stretch handles by adding code for a visual cue as the parameter is stretched.

1. Add the following code to the first *rodAssignHandleToParameter* statement:

   ```
   ?displayName "xDistance”
   ```

   This statement instructs the system to place a label with the text *xDistance* = *value* near the instance as the parameter is stretched, as shown below:



   Notice that you may choose to place text other than the exact parameter name as the value of the *displayName* keyword argument. For example, you could use *width* in place of *xDistance* and *length* in place of *yDistance*.

2. Add the following code to the last *rodAssignHandleToParameter* statement:

   ```
   ?displayName "yDistance”
   ```

   This adds a label to the vertical stretch handle similar to the one in the previous step.

3. Save your changes to the file.

Compile and Test the Feedback Display

1. Enter this command into the CIW:

   ```
   load “rect1.il”
   ```

   This command creates the rectangle pcell with the new feedback display code.

   Did you see any errors or warnings? If so, try to rectify the problem, save your changes to the file, and repeat this step.

2. As in the previous section, choose **Edit—Stretch** from the menus in the layout editor window and test the behavior of the horizontal and vertical stretch handles.

   Notice the benefit of this feedback! You can tell the new value of the stretch handle before you click.

Important

If you had difficulty with this section, you may refer to this file to see a complete solution:

```
source/rect1.il
```

3. Close the text editor for *rect1.il*.

Controlling Parameter Stretches

In the previous stretch handle examples, you were able to stretch either handle in such a way to create errors in the system. You now employ *user functions* and *user data* to solve this problem.

1. Open a text editor on this file:

   ```
   rect2.il
   ```

   Examine the code in this file. Notice that technology information is now obtained and used in this pcell.

   What technology information is obtained? Why?

2. Page forward to the bottom of the file.

   Locate the first *rodAssignHandleToParameter* statement and examine the *userData* and *userFunction* keyword arguments. Notice that the value of the *userFunction* keyword argument is a nameless function known as a *lambda function*.

   ```
   ?userFunction
     lambda((SPCinfo)
       if(SPCinfo->paramVal < SPCinfo->userData
         SPCinfo->userData + SPCinfo->increment
         SPCinfo->paramVal + SPCinfo->increment))
   ```

   This function is called repeatedly as you stretch the parameter referenced in this *rodAssignHandleToParameter* statement.

   Recall that the value of the *userData* keyword argument is passed to the user function within a data structure (a *defstruct*). You can reference this value within the user function through the *userData* field in the data structure. This can be seen in the code above.

   Also recall that the value of the stretch parameter can be obtained from the data structure through the *paramVal* field, and the current amount the target handle has been stretched can be obtained from the *increment* field.

   Can you determine the purpose of this user function?

   Note: Remember that an ***if*** statement without ***then*** and ***else*** implies that the first statement in its body is the ***then*** clause and the remaining statements form the ***else*** clause. The value retuned by the if statement is the value of the last statement executed in its body.

3. Supply the value for the *userData* keyword argument in both *rodAssignHandleToParameter* statements.

4. Save your changes to the file.

Compile and Test the Parameter Control

1. Enter this command into the CIW:

   ```
   load “rect2.il”
   ```

2. As in the previous section, choose **Edit—Stretch** from the menus in the layout editor window and test the behavior of the horizontal and vertical stretch handles. In particular, try to stretch each handle in the negative direction beyond the minimum width for the chosen layer.

   What happens? Does the instance behave as expected?

Important

If you had difficulty with this section, you may refer to this file to see a complete solution:

```
source/rect2.il
```

3. Close the text editor for *rect2.il*.

Prepare for the Next Lab

Execute the following steps to prepare for the next lab exercises:

1. In the CIW, choose **File—Exit** and click on **Yes** in the dialog box that appears.

   This command closes the Design Framework II session and all of the windows associated with it.

2. In the shell, enter the following command:

   ```
   cd ~
   ```

   This command places you in your home directory.

Lab 5-5 Auto-Abutment for Virtuoso XL

Objective: Understand the pcell code necessary to perform auto-abutment in Virtuoso XL.

The Virtuoso® XL Layout Editor provides features for automatically connecting structures with electrically equivalent pins. This allows you to rapidly construct transistor- or cell-level layouts while minimizing area and design rule violations. However, to be compatible with these features, the cells you use must process abutment and un-abutment events as they occur within the system.

A sample pcell library, complete with source code, is available with each installation of the Virtuoso tools. Some cells demonstrate the code necessary to handle abutment and un-abutment events. Furthermore, you can install these cells into a library containing your own technology and experiment with them. This provides an excellent learning opportunity which you will explore in this exercise.

Start a Design Framework II Session

1. Enter these commands into a terminal window:

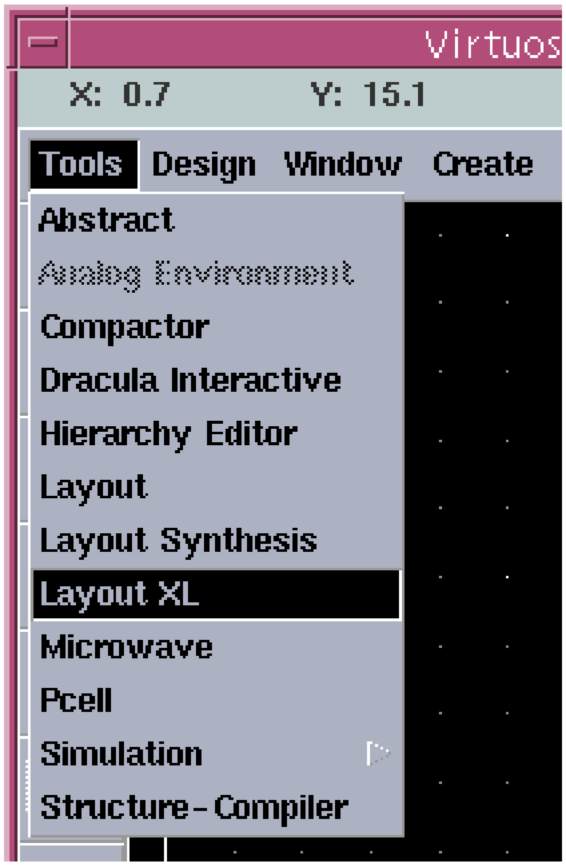
   ```
   cd ~/Skill_PCells_5_1_41/samplePcells
   layoutPlus &
   ```

   Note: You must use the layoutPlus command or you will not have access to the Virtuoso XL features, including auto-abutment.

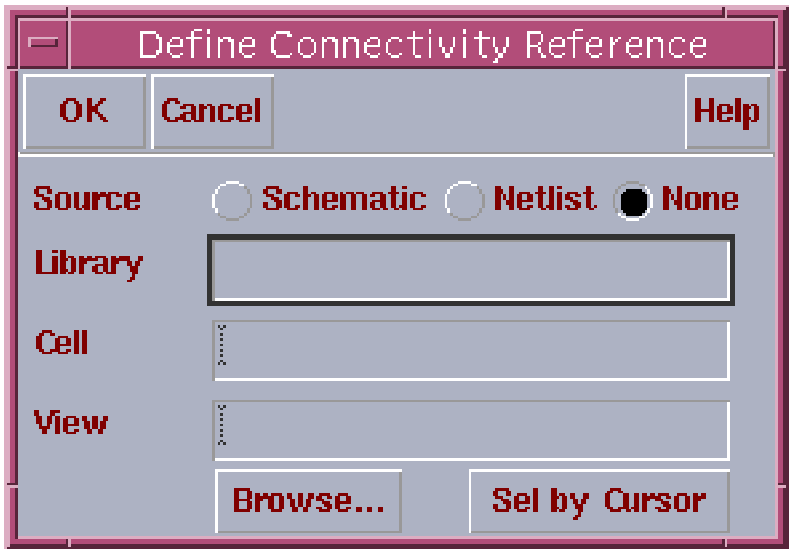
   After a few moments, the Command Interpreter Window (CIW) appears followed by a new (empty) layout editor window for the cellview *abut experiment layout*.

   You now invoke the Virtuoso XL features in this window.

2. Select **Tools—Layout XL** in the layout editor window, as shown below:



A form appears allowing you to select the connectivity source for the open cellview, as shown below:



Source ○ Schematic ○ Netlist [✓ ● None]
Library
Cell
View

3. Choose **None** in the **Source** field and click **OK** on the form.

   These actions enable the Virtuoso XL features for this layout editor window.

Experiment with Automatic Abutment

To expedite this exercise, the necessary sample pcells have been installed in the *abut* library. If you want to know about the installation process, you can refer to the Cadence online documentation *Sample Parameterized Cells Installation and Reference.*

In this section, you create instances of simple MOS devices from the sample pcells library and experiment with their behavior.

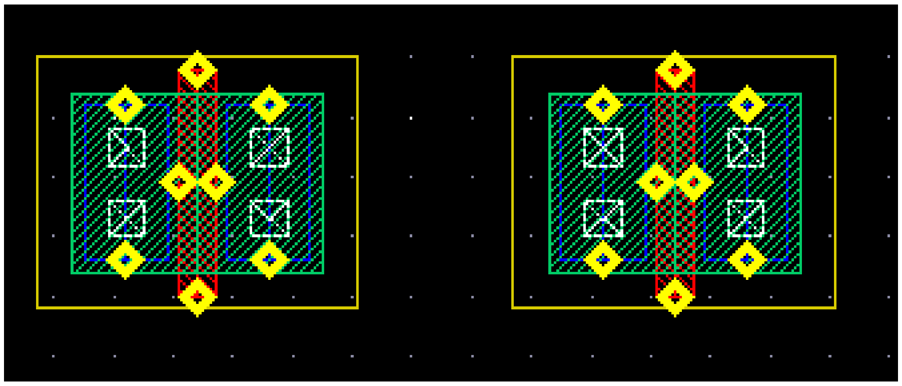
1. Create two instances of *abut spcpmos layout* at locations of your choosing in the layout editor window.

   To do this, select **Create—Instance** and enter the following values in the form:

Library	`abut`
Cell Name	`spcpmos`
View Name	`layout`

   Move the outline of the cell in the layout editor to the desired location and click the left mouse button. Move the cell outline and left-click once again in a different location to create the second instance. Click **Cancel** on the options form when you are done.

   The instances might appear like this in the editor window:

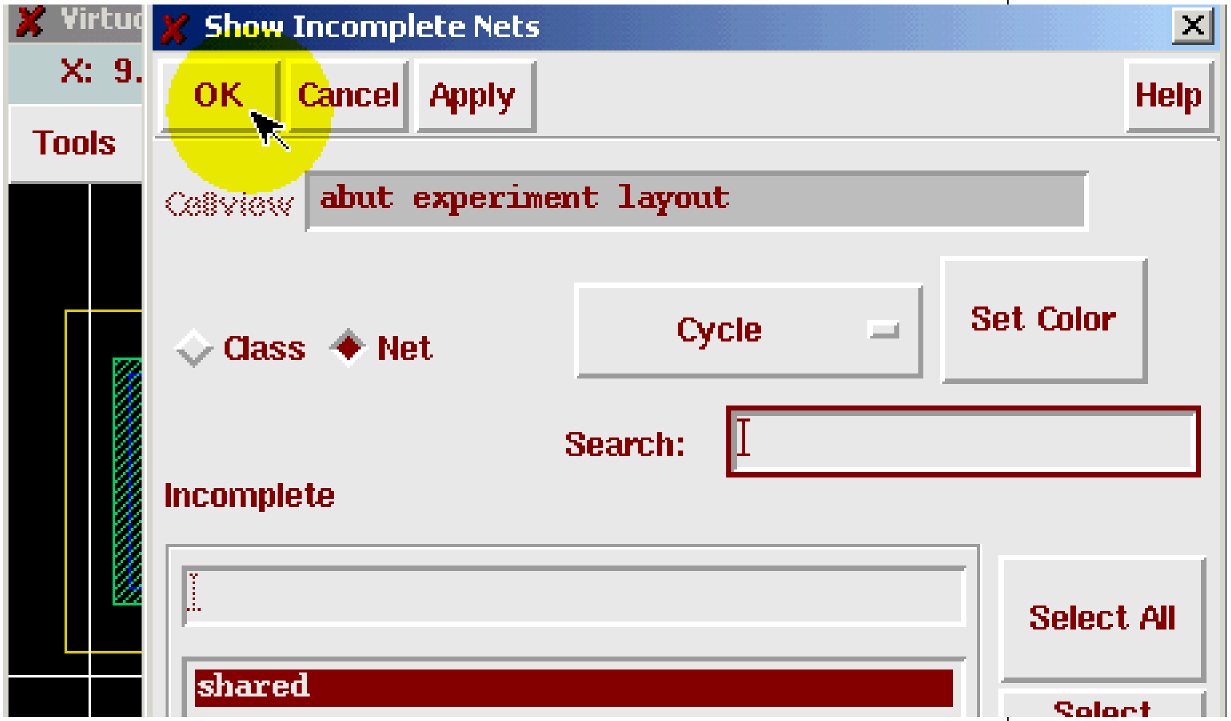


2. Establish connectivity between the Source of one instance to the Drain of the other instance. Run a custom SKILL procedure to assign a common net to each of the instance terminals.

   ```
   ExConnect()
   ```

3. Check the connectivity by highlighting incomplete nets. In the layout window select **Connectivity—Show Incomplete Nets**,

   When the form appears select *shared* from the Incomplete nets field and click **OK** on the form.



Cellview abut experiment layout

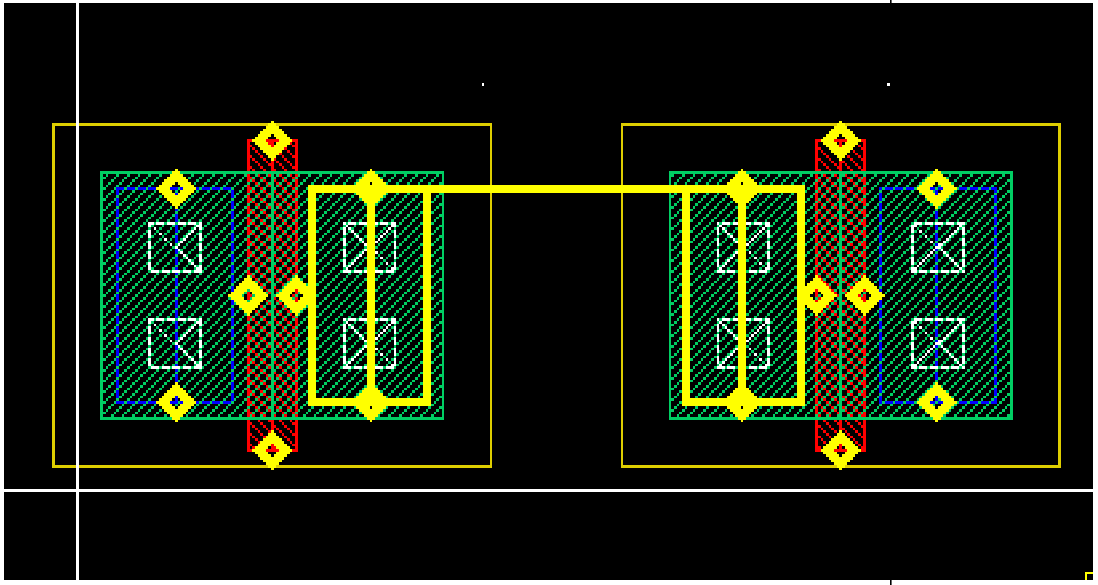
Class [✓ Net]

Search:

Incomplete

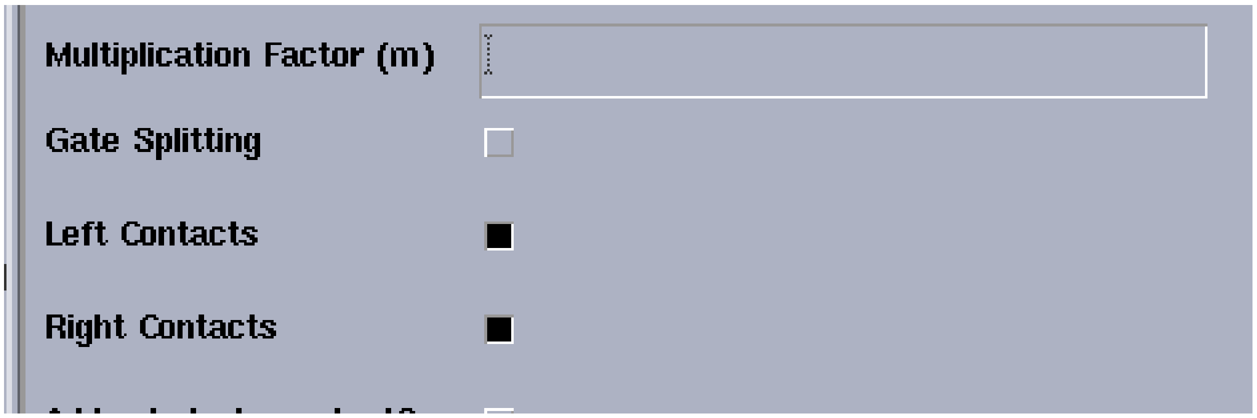
shared

4. The Source terminal of one device is connected to the Drain terminal of the other with a flight line to show they share the common net.



5. Select one of the *spcpmos* instances and examine its parameters (**Edit—Properties**, then choose the **Parameter** category).

   Notice the parameters controlling the source and drain metal connections (**Left Contacts** and **Right Contacts**):



Multiplication Factor (m)	
Gate Splitting	☐
Left Contacts	■
Right Contacts	■

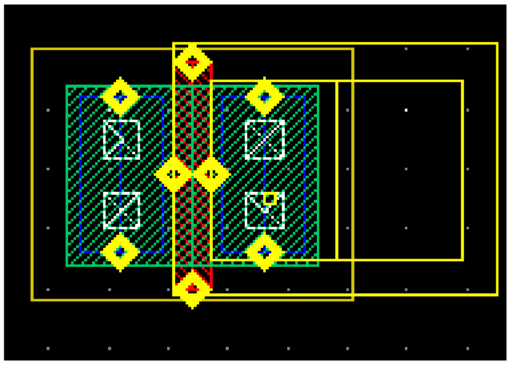
   When enabled, these parameters instruct the code for this cell to create a metal connection with contacts on either the left or right side of the transistor.

6. Obtain the database ID for this instance by entering the following code into the CIW:

   ```
   i1 = car(geGetSelSet())
   ```

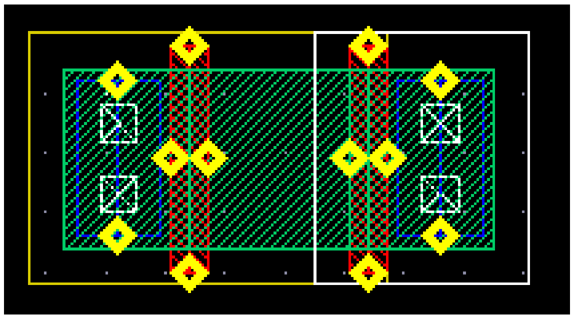
   Note: You need the database ID to identify this instance later in the exercise.

7. Drag the selected instance over the other *spcpmos* instance such that their source and drain regions overlap as shown below:



8. Release the mouse button and observe the results.

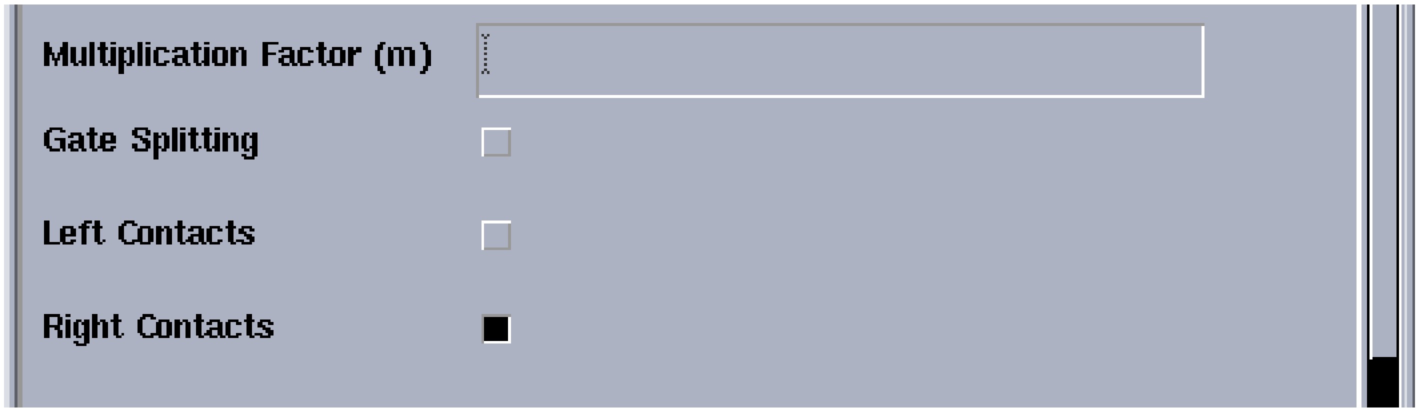
   The system determined the intended connectivity between the two instances, and performed an abutment operation upon them:



   In connecting these transistors in series, various cell parameters were set so that the metal connection on the shared source/drain region does not appear for either transistor.

9. Examine the property editor form.

   Notice that the **Left Contacts** parameter is disabled:



Multiplication Factor (m)	
Gate Splitting	☐
Left Contacts	☐
Right Contacts	■

10. Click on the other instance to select it.

    The property editor form displays information about the currently selected instance.

11. Examine the properties on this instance.

    Notice that the **Right Contacts** parameter is disabled.

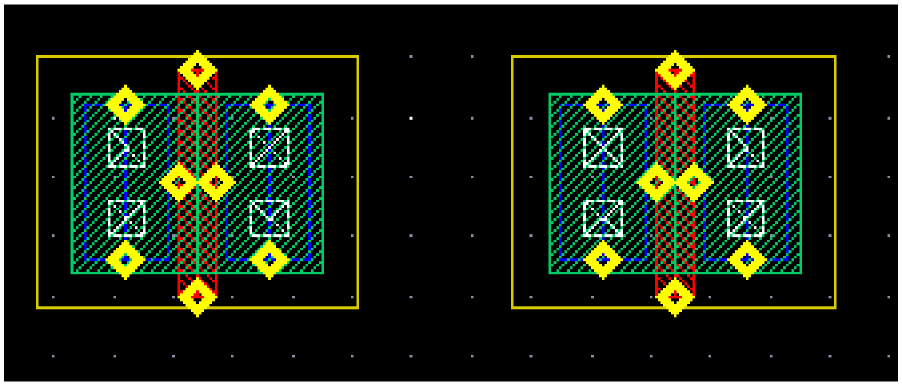
12. Obtain the database ID for this instance by entering the following code into the CIW:

    ```
    i2 = car(geGetSelSet())
    ```

    Note: You need the database ID to identify this instance later in the exercise.

13. Move one of the instances to a location such that the drain/source regions no longer overlap.

    Notice that the metal connections reappear for the source and drain regions of the two transistor instances, as shown below:



14. Examine the **Parameter** category of the property editor form.

    The **Left Contacts** and **Right Contacts** parameters have been updated to enable the source and drain metal connections.

Investigating the Abutment Function

In the previous section, you witnessed behavior of transistor instances in the Virtuoso XL layout editor as the instances processed abutment events. Now you investigate the SKILL code that implements this behavior.

1. Open a text editor on the file called *mos.il* in the current directory.

   This code defines the functions used to implement the *spcpmos* and *spcnmos* transistors.

2. Repeatedly search for this string in the file:

   ```
   spcMosAbutFunction
   ```

   You will find two occurrences of this string used in code like this:

   ```
   dbReplaceProp(objId "abutFunction" "string"
                 "spcMosAbutFunction")
   ```

   This call to *dbReplaceProp* places a property called *abutFunction* on the database object stored in the local variable *objId*. The value of the property is the name of a function to be called when the cell processes an abutment event. The object annotated with this property is a ROD rectangle forming one diffusion connection for the transistor. Both diffusion connections are annotated in this manner, so there are two similar calls to *dbReplaceProp*.

   The final occurrence of this string appears in the definition of the *spcMosAbutFunction* itself.

3. Examine the body of *spcMosAbutFunction*.

   In particular, notice the debugging statements near the beginning of the function definition. Also, notice that this comment line appears prior to the printing statements used for debugging:

   ```
   ;        debugging = t
   ```

   By removing the comment character from the beginning of this statement, the debugging code will be executed as this function is called.

4. Remove the comment character (the semicolon) from the beginning of the line, as shown below:

   ```
            debugging = t
   ```

5. Save your changes to the file.

6. Load the file into your Design Framework II session by entering this command into the CIW:

   ```
   load “mos.il”
   ```

   As you cause abutment events to occur between instances of the *spcpmos* cell, you will now see debugging information printed to the CIW.

7. Drag one of the *spcpmos* instances over the other instance such that their source and drain regions overlap. Release the instance.

8. Examine the contents of the CIW and answer these questions:

   a. How many times was *spcMosAbutFunction* called when you dropped the *spcpmos* instance? Why?

   b. Which instance is designated *instA* in the debugging information?

   Hint: Use the values stored in the variables *i1* and *i2* to make this determination.

   c. Which instance is designated *instB* in the debugging information?

   d. Which event occurred in each call to *spcMosAbutFunction*?

   Hint: Remember the meaning of the *event* value:

   ```
   1 = compute and return abutment offset
   2 = adjust pcell parameters for abutment
   3 = adjust pcell parameters for unabutment
   ```

   e. What is the return value for each call to *spcMosAbutFunction* and why are they different?

   Hint: The type of event determines the type of return data.

9. Locate this line of code in the source file:

   ```
           case(event
   ```

   This *case* statement defines the behavior of *spcMosAbutFunction* for each of the possible abutment events. Examine the clauses in the statement and answer the following questions:

   a. Which of these clauses was executed in each call to *spcMosAbutFunction*?

   b. What output statements in the CIW confirm your answer to the previous question?

   c. For an offset event (that is, when the value of *event* is 1), does *spcMosAbutFunction* ever return a non-zero value? Why?

10. Examine the abutment event clause (that is, when the value of *event* is 2).

    Notice that the bulk of this clause is itself a *case* statement. For this statement, the controlling expression is the value of the *conn* argument passed to *spcMosAbutFunction*. The *conn* argument can take on these values and meanings:

    ```
    1 = pins are connected to the same net and do not
        connect to other pins
    2 = pins are connected to the same net and the net
        connects to other pins
    ```

    The body of each clause of this subordinate *case* statement is an *if* statement or a series of nested *if* statements. These *if* statements control the actions for the abutment event based upon the type of connection and the relative sizes of the abutting pins.

    Referring to the diagram below, answer these questions:

    a. Which clause handles the type of abutment shown in diagram **A**?

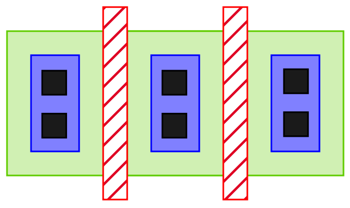
    b. Which section of code in the body of this clause is actually executed for the situation depicted in diagram **A**?

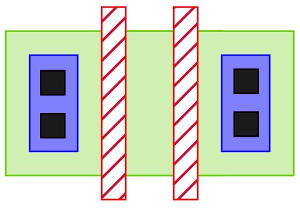
    c. Which section of code is executed for the situation depicted in diagram **B**?

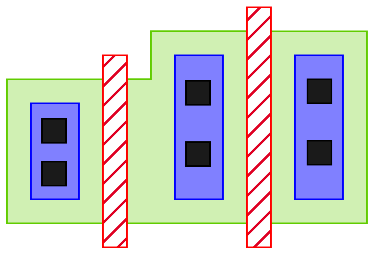
    d. Which section of code is executed for diagram **C**?

    e. In each *if* statement, what actual changes are made to the instances involved in the abutment?

MOS transistors sharing a diffusion


A. Same size, terminals with external connection on same net


B. Same size, no other connections on the same net


C. Different size, terminals with external connection on the same net

11. Select one of the transistor instances and move it to a location such that the source and drain regions of the instances no longer overlap.

12. Examine the messages written to the CIW and answer these questions:

    a. How many events occurred?

    b. Which type of event(s) occurred?

    c. What value was returned?

13. Examine the code for *spcMosAbutFunction*. Can you determine the flow of execution through the function for this event?

    Notice in the *case* statement for the event type, the clause that handles unabut events (*event* = 3) does little more than update properties on the instances involved:

    ```
    foreach(propList group~>oldValues
      cond(
        (car(propList) == instA~>name
          dbReplaceProp(instA cadr(propList)
            caddr(propList) cadddr(propList))
        )
        (car(propList) == instB~>name
          dbReplaceProp(instB cadr(propList)
            caddr(propList) cadddr(propList))
        )
      )
    )
    ```

    Notice also that the property values involved in the update come from the *group* argument to *spcMosAbutFunction*. This block of code simply restores each instance to its state prior to abutment. For example, if source and drain connections were enabled prior to abutment, this condition would be restored after moving the instances apart from one another.

Abutment with Varied Transistor Widths and Additional Connectivity

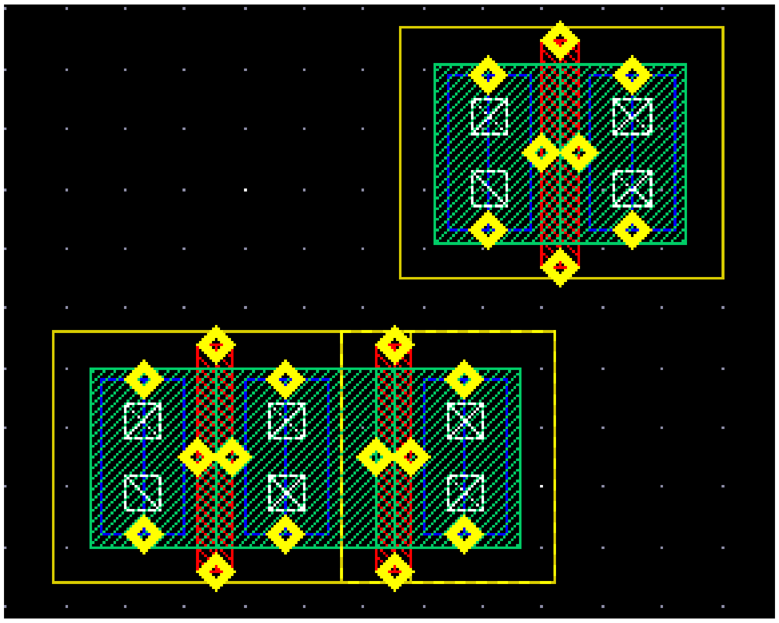
You now investigate the behavior of the *spcpmos* cell when multiple connections are possible and when transistors of differing widths are abutted to one another.

1. In the same manner as with the previous instances, create a third instance of *abut spcpmos layout* at a location of your choosing in the layout editor window.

2. Abut this new instance to an existing transistor.

   What happens?

   The transistors appear similar to the depiction below:



   Notice that the source/drain connection remains on the abutted instance. Why?

3. Examine the messages in the CIW to determine the flow of execution through *spcMosAbutFunction*. In particular, locate the code that produced this line in the messages:

   ```
   *Contact needed
   ```

   Why did *spcMosAbutFunction* choose to execute this code?

4. Enter the following into the CIW:

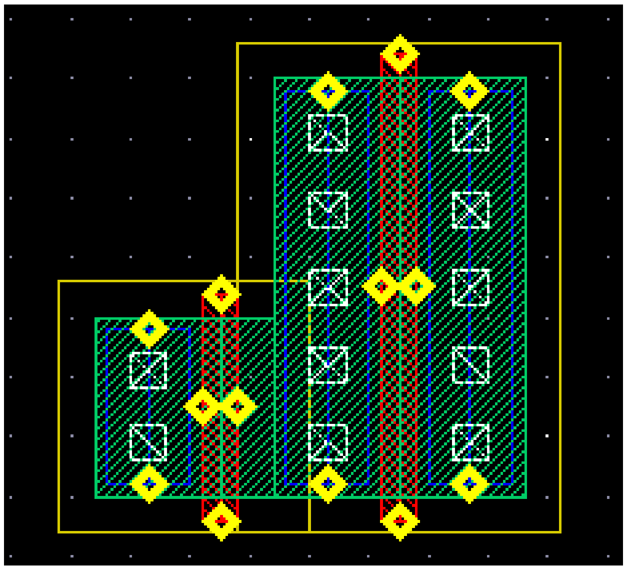
   ```
   “STRETCH”
   ```

   This acts as a visual cue to separate the previous abutment event reports from those to follow.

5. Stretch the gate width (vertical dimension) of the new transistor instance.

   To do this, choose **Edit—Stretch** from the menus in the layout editor window, then place the pointer over the appropriate stretch target. Click and drag the stretch target to its new location.

   The abutted transistors will now look something like this:



6. Using the “`STRETCH`” cue to identify the most recent abutment event reports, examine the messages in the CIW and answer these questions:

   a. How many events were processed by *spcMosAbutFunction*?

   b. Of what type was each event?

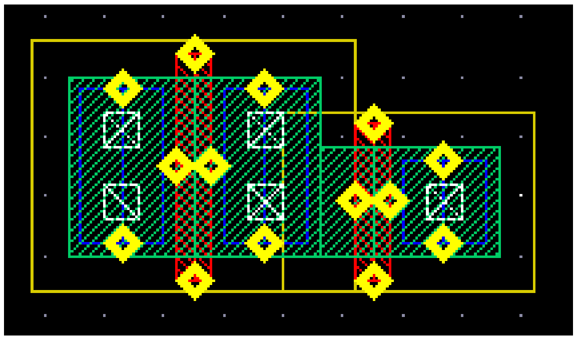
   c. In what order did the events occur?

7. Look in the messages for this line:

   ```
   pinAw is larger than pinBw.
   ```

   Trace through the code for *spcMosAbutFunction* and determine how this line was generated.

8. Stretch this instance once again such that it’s gate length is shorter than the abutted transistor. The instances will look similar to this:



9. Look in the messages for this line:

```
pinAw is smaller than pinBw.
```

Notice that *spcMosAbutFunction* recognized that the stretched instance now has a smaller source/drain connection region than the abutted transistor.

Trace through the code for *spcMosAbutFunction* and determine how this line was generated.

10. If you wish, you may perform additional experimentation with the existing instances or with additional instances. As you experiment, examine the messages reported in the CIW and relate them to the code in *mos.il*.

Prepare for the Next Lab

Execute the following steps to prepare for the next lab exercises:

1. Close the text editor for *mos.il*.

2. In the CIW, choose **File—Exit** and click on **Yes** in the dialog box that appears.

   This command closes the Design Framework II session and all of the windows associated with it.

3.In the shell, enter the following command:

```
cd ~
```

This command places you in your home directory.

Summary

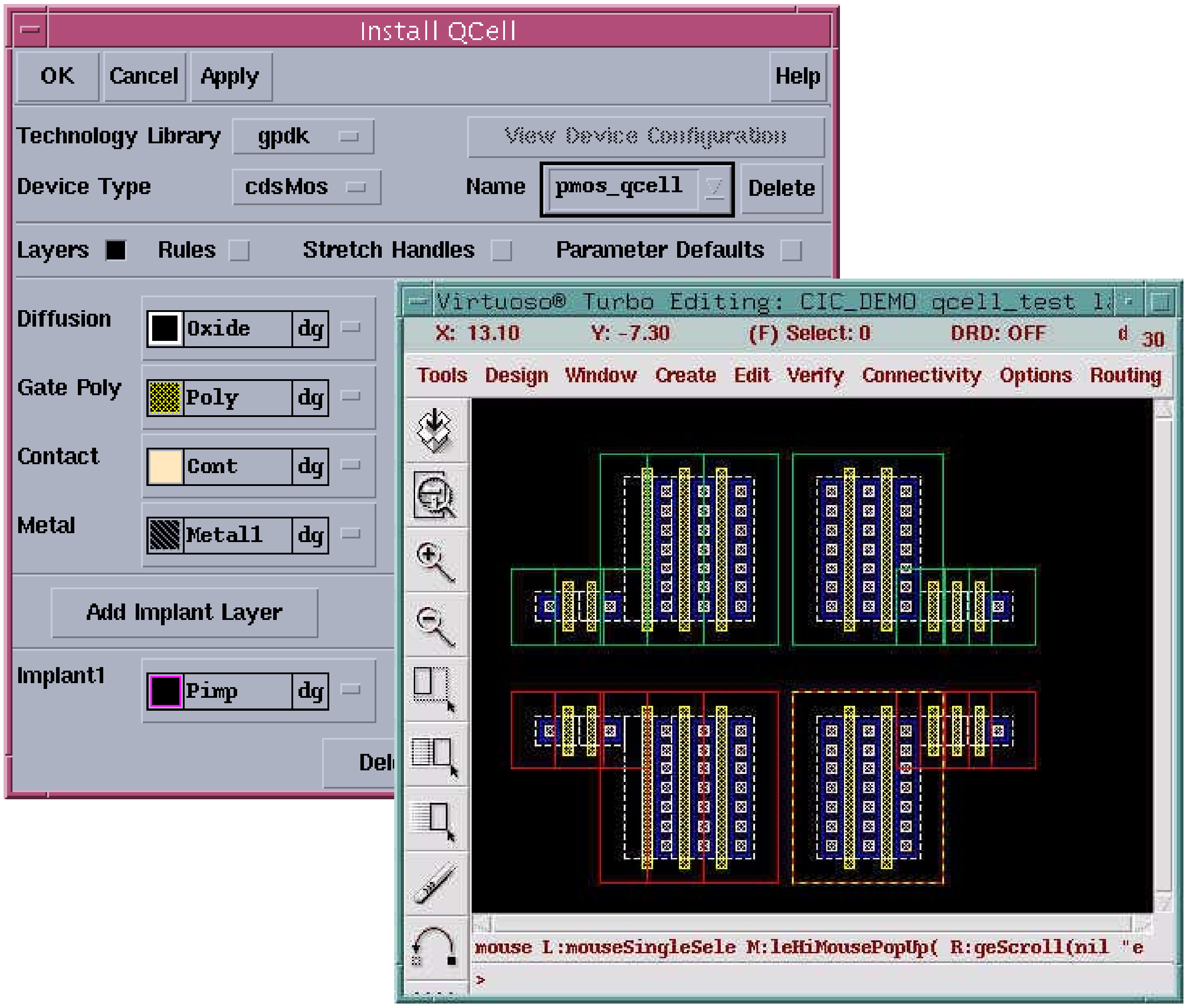
During these exercises, you have:

- Created pcells with hierarchy
- Taken advantage of ROD hierarchical point transformation
- Created and used CDF parameters on a complete pcell
- Created and used pcell stretch handles
- Experimented with auto-abutment in Virtuoso XL
- Examined source code to an example transistor that implements auto-abutment

End of Lab

Labs for Module 6

Creating and Using Qcells (Optional)



Lab 6-1 Installing cdsMos Qcells

Objective: Learn how to define and install cdsMos qcells.

In this lab, you will define and install the PMOS and NMOS qcells based on given specifications.

Note: This optional module requires the use of Virtuoso® Layout Editor Turbo or Virtuoso XL. Qcells are not available in the standard Virtuoso Layout Editor tool.

Starting the Cadence Software

1. Change to the working directory in an xterm window. Enter:

```
cd ~/Skill_PCells_5_1_41/Qcells
```

2. In the same xterm window, enter:

```
layoutPlus &
```

   The Virtuoso Layout Editor Turbo software starts in background mode. The Command Interpreter Window (CIW) appears.

Installing the PMOS cdsMos Qcell

1. From the CIW, select **File—Technology File Manager**.

   The Technology File Toolbox window appears.

2. From the Technology File Toolbox window, select **Qcell**.

   The Install Qcell window appears.

3. Select **File—Close** in the Technology File Toolbox window.

4. Within the Install Qcell window, set the following fields:

Parameter	**Value**
Technology Library	`gpdk`
Device Type	`cdsMos`
Name	`pmos_qcell`

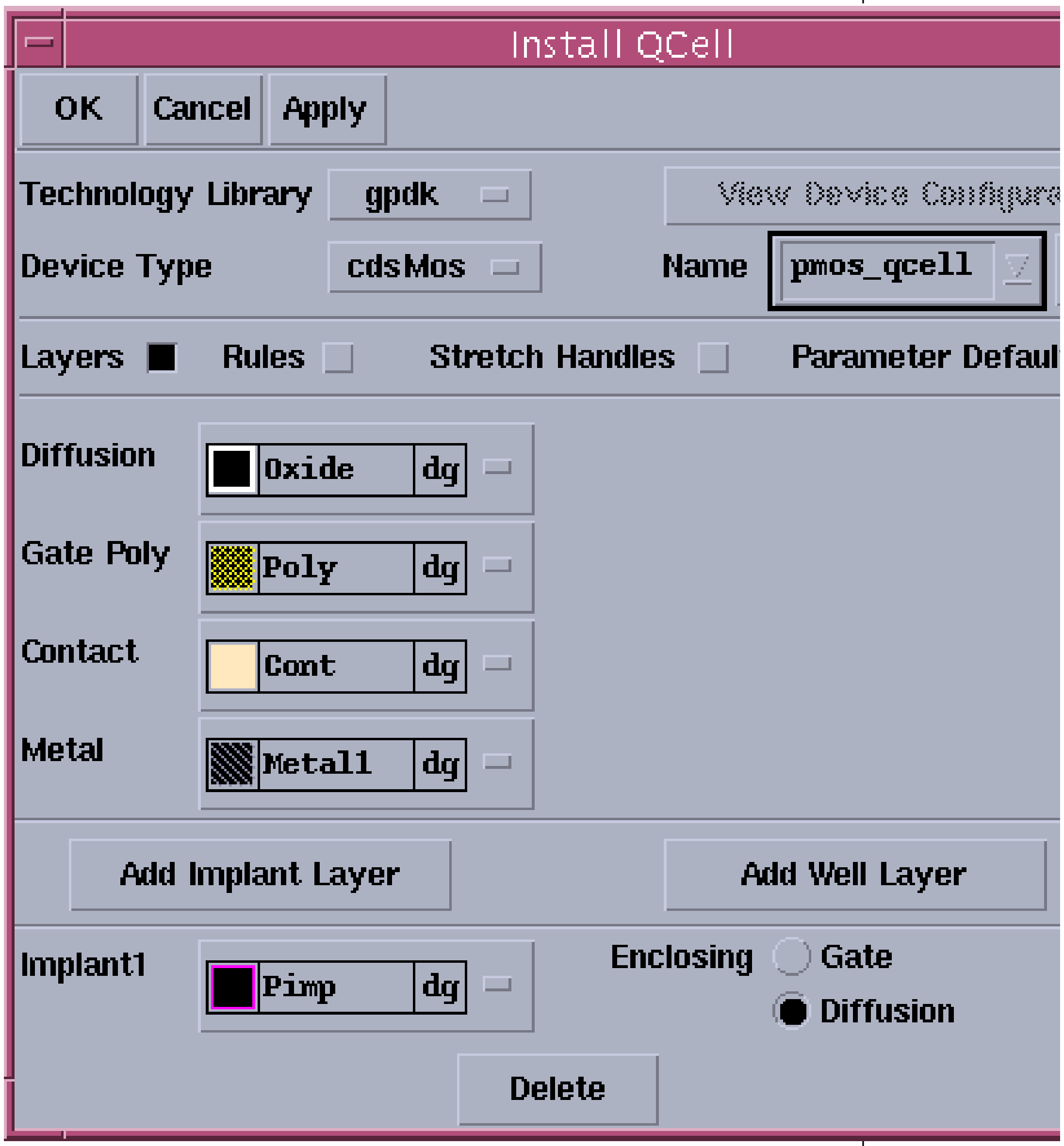
5. With the **Layers** cyclic button clicked, set the following layers for each of these parameters:

Parameter	**Value**
Diffusion Layer	`Oxide dg`
Gate Poly	`Poly dg`
Contact	`Cont dg`
Metal	`Metal1 dg`

6. Click on the **Add Implant** button and set the Implant1 field to be
   `Pimp dg`

7. Within the Implant1 field, also set the **Enclosing** cyclic field to be:
   `Diffusion`

8. Review the Install Qcell form. It should look like the example:



Install QCell

OK Cancel Apply

Technology Library	gpdk	View Device Configur	
Device Type	cdsMos	Name	pmos_qcell

Layers ■ Rules □ Stretch Handles □ Parameter Defaul

Diffusion	■ Oxide	dg
Gate Poly	▨ Poly	dg
Contact	□ Cont	dg
Metal	▨ Metal1	dg

Add Implant Layer Add Well Layer

Implant1	■ Pimp	dg	Enclosing	○ Gate
				● Diffusion

Delete

9. Click on the **Rules** radio button toward the top of the Install Qcell form.

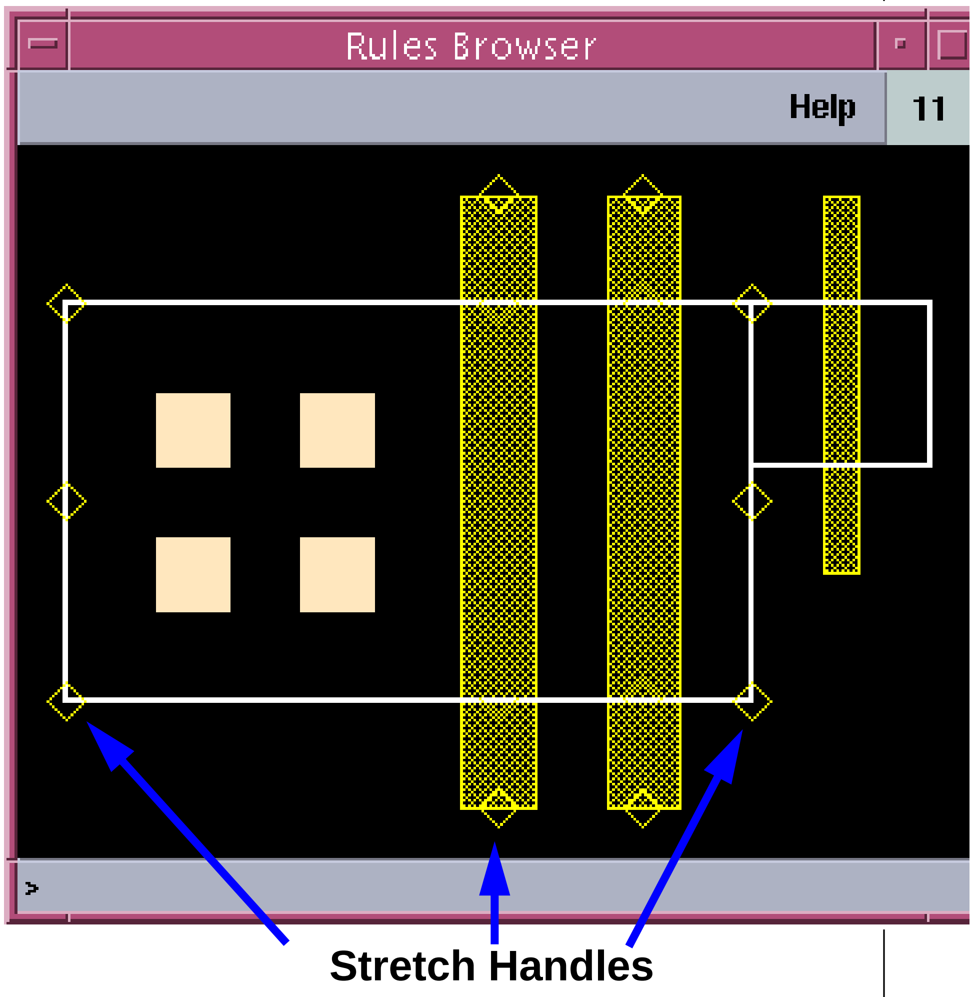
   The Install Qcell form refreshes and reflects the rules defaulted from the technology file. The Rules Browser window also appears.

10. Move the Rules Browser window to the side of the Install Qcell form.

11. Review the rules set from the technology file in the Rules section of the Install Qcell form. You do not need to make any changes.

12. Click on the **Stretch Handles** radio button toward the top of the Install Qcell form.

    The Install Qcell form reflects the stretch handles defaulted from the technology file. The Rules Browser window reflects which stretch handles are chosen from the form with a diamond-shaped marker as illustrated:



13. Choose the **Select All** button within the Install Qcell form.

14. Click on the **Parameter Defaults** radio button toward the top of the Install Qcell form.

    The Install Qcell form reflects the parameter defaults for the parameters defaulted from the technology file. The Rules Browser window closes automatically.

15. Set the following parameters for each of these fields:

Parameter	**Value**
Abutment Class	`pmos`
Component Class	`PMOS`
MOS Type	`pmos`
Fold Threshold	`10u`

16. In the Contact Cut Spacing Method field at the bottom of the Install Qcell form, ensure that **Distribute** is selected.

17. Click **OK** within the Install Qcell form.

18. The system prompts with an Install Qcell window denoting:

```
Save the technology file to disk?
```

19. Click **Yes** to save the technology file to disk.

    Note: You must have write access to the technology file in order to install the qcells. If you do not have access you will get an error message at this point.

Installing the NMOS cdsMos Qcell

1. From the CIW, select **File—Technology File Manager**.

   The Technology File Toolbox window appears.

2. From the Technology File Toolbox window, select **Qcell**.

   The Install Qcell window appears.

3. Select **File—Close** on the Technology File Toolbox window.

4. Within the Install Qcell window, set the following fields:

Parameter	**Value**
Technology Library	`gpdk`
Device Type	`cdsMos`
Name	`nmos_qcell`

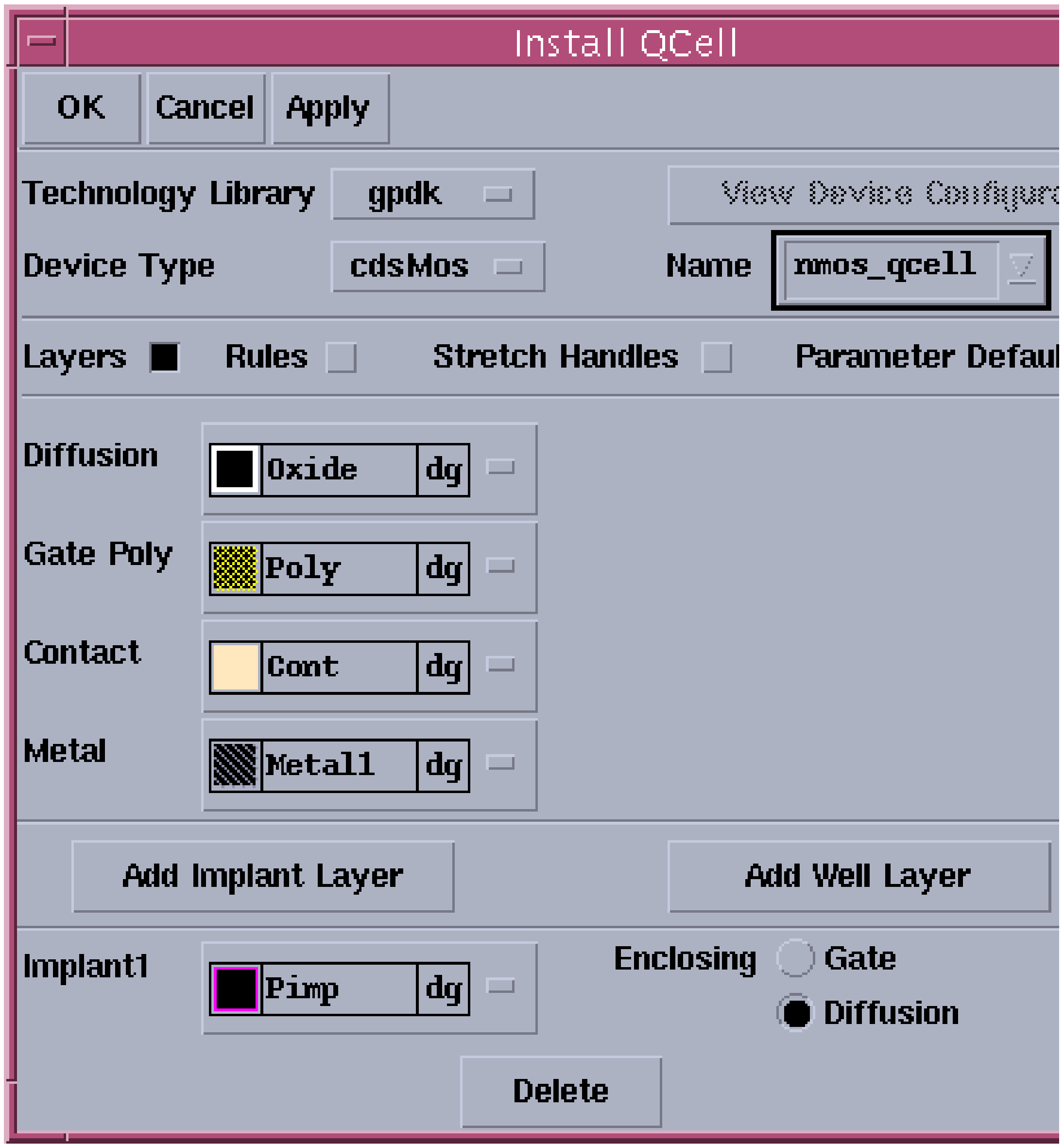
5. With the **Layers** cyclic button clicked, set the following layers for each of these parameters:

Parameter	**Value**
Diffusion Layer	`Oxide dg`
Gate Poly	`Poly dg`
Contact	`Cont dg`
Metal	`Metal1 dg`

6. Click on the **Add Implant** button and set the Implant1 field to be
   `Nimp dg`

7. Within the Implant1 field, also set **Enclosing** cyclic field to be:
   `Diffusion`

8. Review the Install Qcell form. It should look the example:



Technology Library gpdk
Device Type cdsMos
Name nmos_qcell

[✓ Layers] Rules Stretch Handles Parameter Defaul

Diffusion	Oxide	dg
Gate Poly	Poly	dg
Contact	Cont	dg
Metal	Metal1	dg
Implant1	Pimp	dg

Enclosing Gate [● Diffusion]

9. Click on the **Rules** radio button toward the top of the Install Qcell form.

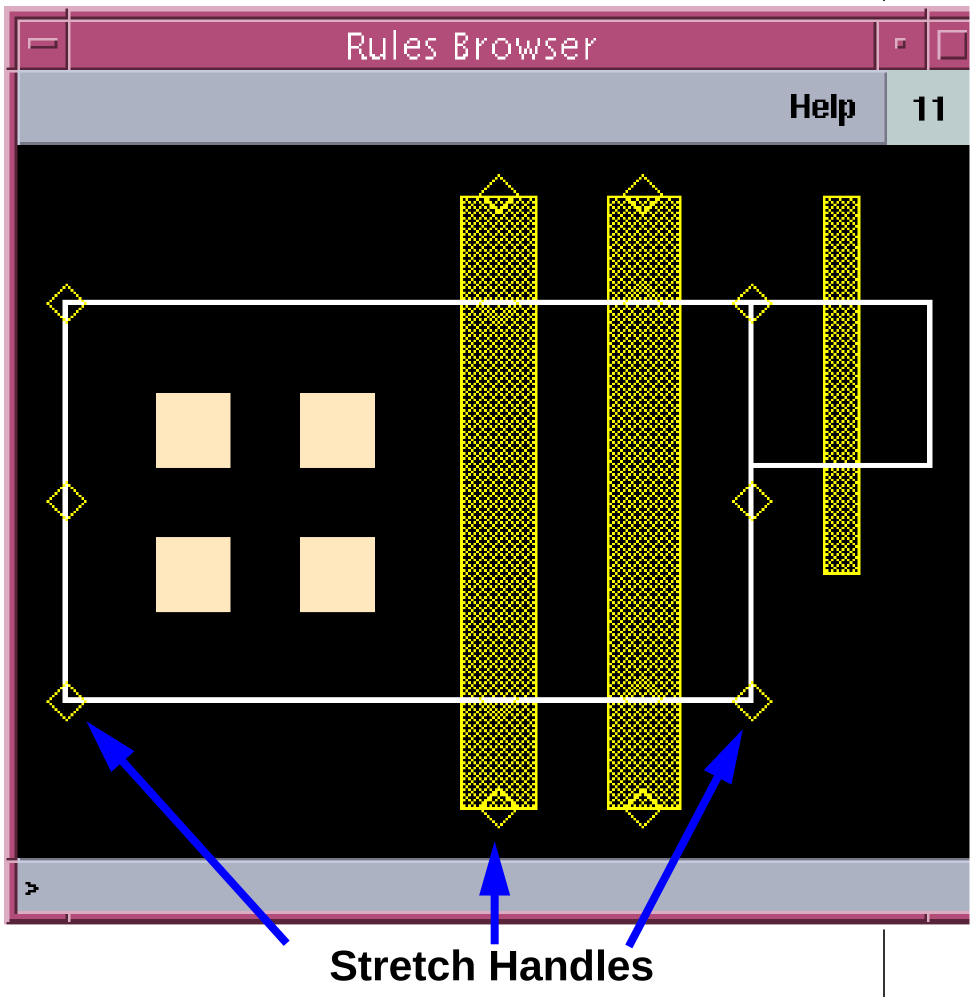
   The Install Qcell form reflects the rules defaulted from the technology file. The Rules Browser window also appears.

10. Move the Rules Browser window to the side of the Install Qcell form.

11. Review the rules set from the technology file in the Rules section of the Install Qcell form. You do not need to make any changes.

12. Click on the **Stretch Handles** radio button toward the top of the Install Qcell form.

    The Install Qcell form refreshes and reflects the stretch handles defaulted from the technology file. The Rules Browser window reflects which stretch handles are chosen from the form with a diamond-shaped marker as illustrated:



13. Click on the **Select All** button within the Install Qcell form.

14. Click on the **Parameter Defaults** radio button toward the top of the Install Qcell form.

    The Install Qcell form refreshes and reflects the parameter defaults for the parameters defaulted from the technology file. The Rules Browser window closes automatically.

15. Set the following parameters for each of these fields:

Parameter	**Value**
Abutment Class	`nmos`
Component Class	`NMOS`
MOS Type	`nmos`
Fold Threshold	`10u`

16. In the Contact Cut Spacing Method field at the bottom of the Install Qcell form, ensure that **Distribute** is selected.

17. Click **OK** within the Install Qcell form.

18. The system prompts with an Install Qcell window denoting:

```
Save the technology file to disk?
```

19. Click **Yes** to save the technology file to disk.

    Note: You must have write access to the technology file in order to install the qcells. If you do not have access you will get an error message at this point.

Verifying the *pmos_qcell* and *nmos_qcell* views

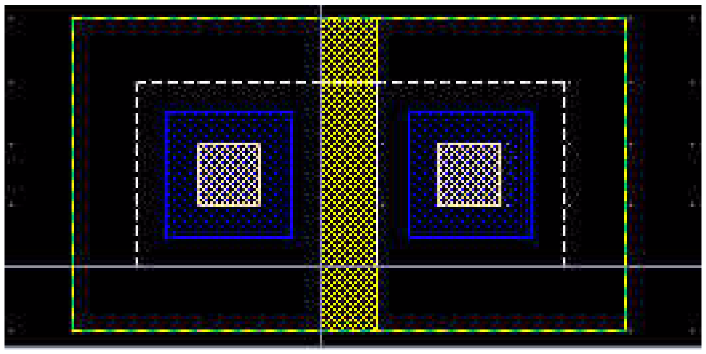
1. From the CIW, select **File—Open**, then enter the following values in the form:

Library Name	`gpdk`
Cell Name	`pmos_qcell`
View Name	`layout`

2. Click **OK**.

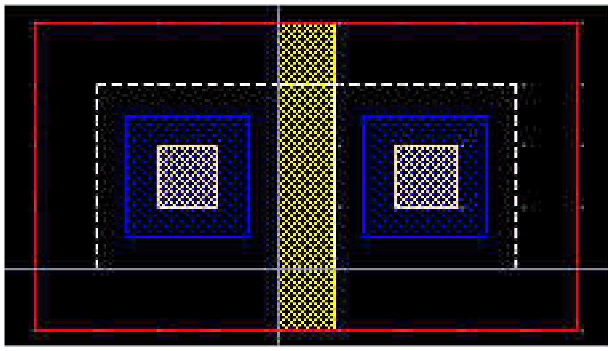
   The *pmos_qcell layout* cellview opens for editing.

3. Ensure that your *pmos_qcell layout* view looks like the example.



4. Select **Window—Close** in the pmos_qcell layout window.

5. Ensure your *nmos_qcell layout* view looks like the following example.



6. Select **Window—Close** in the nmos_qcell layout window.

7. Leave the Cadence® session open for the next lab.

Summary

In this lab, you learned the following:

- Installing the PMOS cdsMos qcell
- Installing the NMOS cdsMos qcell

End of Lab

Lab 6-2 Installing cdsVia Qcells

Objective: Learn how to define and install cdsVia qcells.

In this lab, you will define and install a substrate tie qcell based on given specifications.

Installing the Substrate Tie Qcell

1. From the CIW, select **File—Technology File Manager**.

   The Technology File Toolbox window appears.

2. From the Technology File Toolbox window, select **Qcell**.

   The Install Qcell window appears.

3. Select **File—Close** on the Technology File Toolbox window.

4. Within the Install Qcell window, set the following fields:

Parameter	Value
Technology Library	`gpdk`
Device Type	`cdsVia`
Name	`psub`

5. Ensure that **Substrate/Well Tie** is enabled.

6. With the **Layers** cyclic button clicked, set the following layers for each of these parameters:

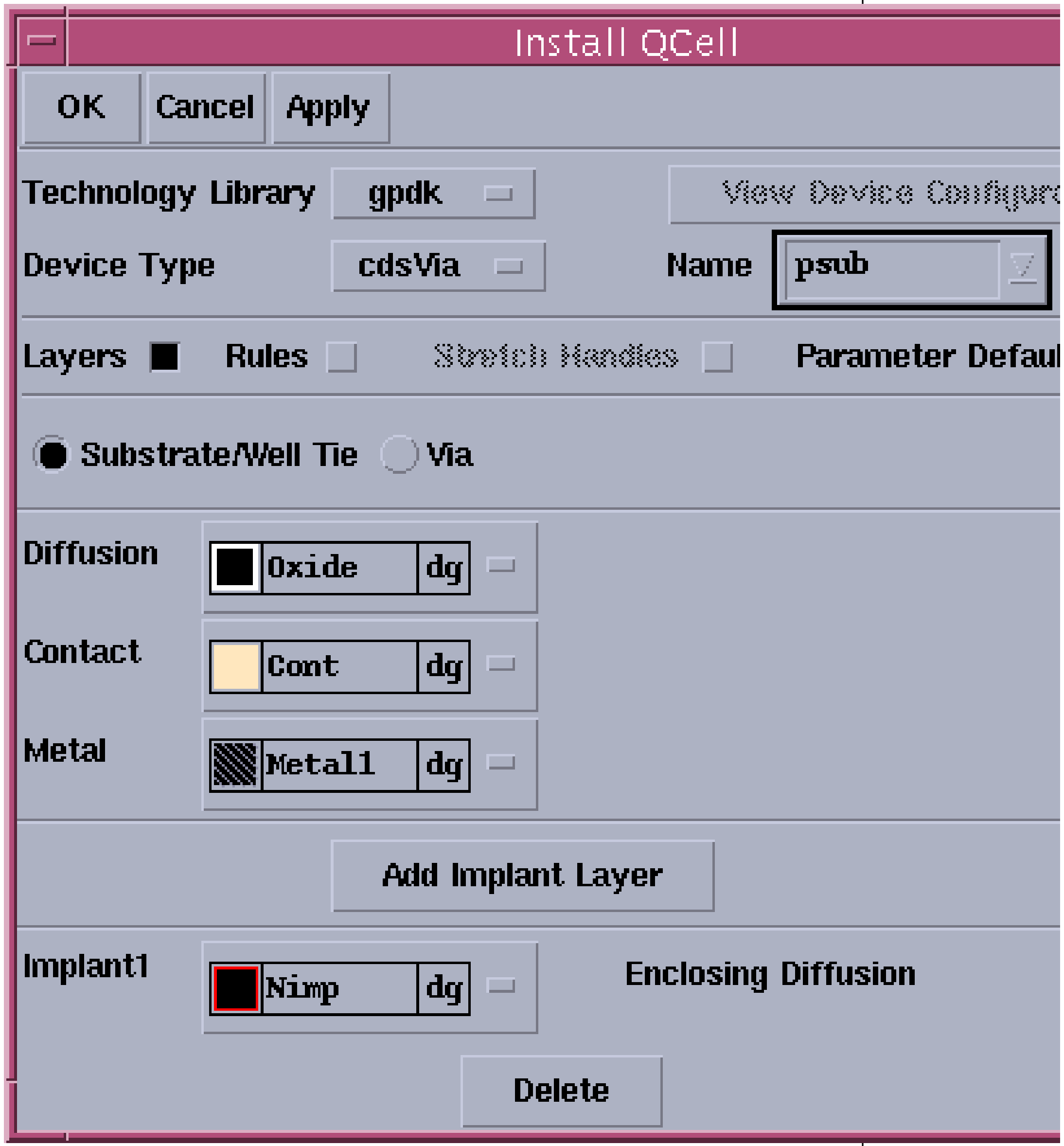
Parameter	Value
Diffusion Layer	`Oxide dg`
Contact	`Cont dg`
Metal	`Metal1 dg`

7. Click on the **Add Implant** button and set the Implant1 field to be `Nimp dg`

8. Within the Implant1 field, also set the **Enclosing** cyclic field to be:

   `Diffusion`

9. Review the Install Qcell form. It should look the example:



Technology Library: gpdk
Device Type: cdsVia
Name: psub
Layers ■ Rules □ Stretch Handles □ Parameter Defaul
● Substrate/Well Tie ○ Via
Diffusion: Oxide dg
Contact: Cont dg
Metal: Metal1 dg

Implant1: Nimp dg
Enclosing Diffusion

10. Click on the **Rules** radio button toward the top of the Install Qcell form.

    The Install Qcell form reflects the rules defaulted from the technology file. The Rules Browser window also appears.

11. Move the Rules Browser window to the side of the Install Qcell form.

12. Review the rules set from the technology file in the Rules section of the Install Qcell form. You do not need to make any changes.

13. Click on the **Parameter Defaults** radio button toward the top of the Install Qcell form.

    The Install Qcell form refreshes and reflects the parameter defaults for the parameters defaulted from the technology file. The Rules Browser window closes automatically.

14. Set the following parameter for the following field:

Parameter	Value
Abutment Class	`pmos`

15. Click **OK** within the Install Qcell form.

16. The system prompts with an Install Qcell window denoting:

```
Save the technology file to disk?
```

17. Click **Yes** to save the technology file to disk.

Note: You must have write access to the technology file in order to install the qcells. If you do not have access you will get an error message at this point.

Verifying the psub Views

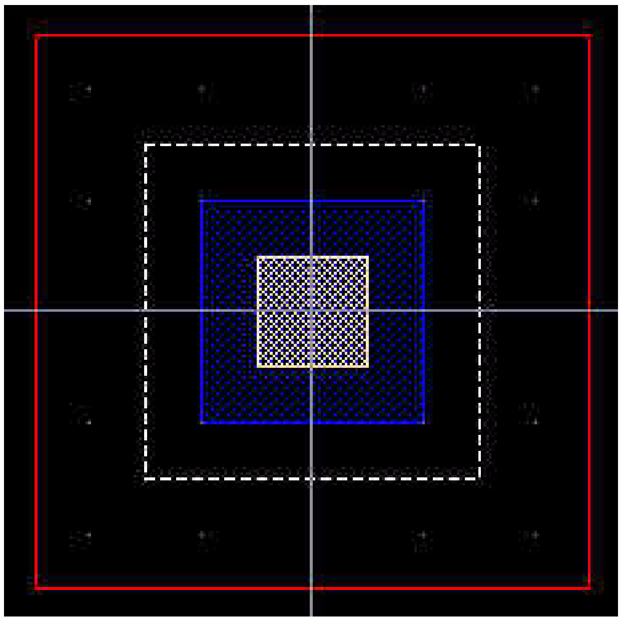
1. From the CIW, select **File—Open**, then enter the following values in the form:

Library Name	`gpdk`
Cell Name	`psub`
View Name	`layout`

2. Click **OK**.

   The *psub layout* cellview opens for editing.

3. Ensure your *psub layout* view looks like the following example.



Summary

In this lab, you learned the following:

- Installing the psub cdsVia qcell.

End of Lab

Lab 6-3 Installing Guard-Ring Qcells

Objective: Learn how to define and install a guard-ring qcell.

In this lab, you will define and install a guard-ring qcell based on given specifications.

Evaluating the Substrate Tie Qcell

1. From the CIW, select **File—Technology File Manager**.

   The Technology File Toolbox window appears.

2. From the Technology File Toolbox window, select **Qcell**.

   The Install Qcell window appears.

3. Within the Install Qcell window, set the following fields:

Parameter	**Value**
Technology Library	`gpdk`
Device Type	`guardRing`

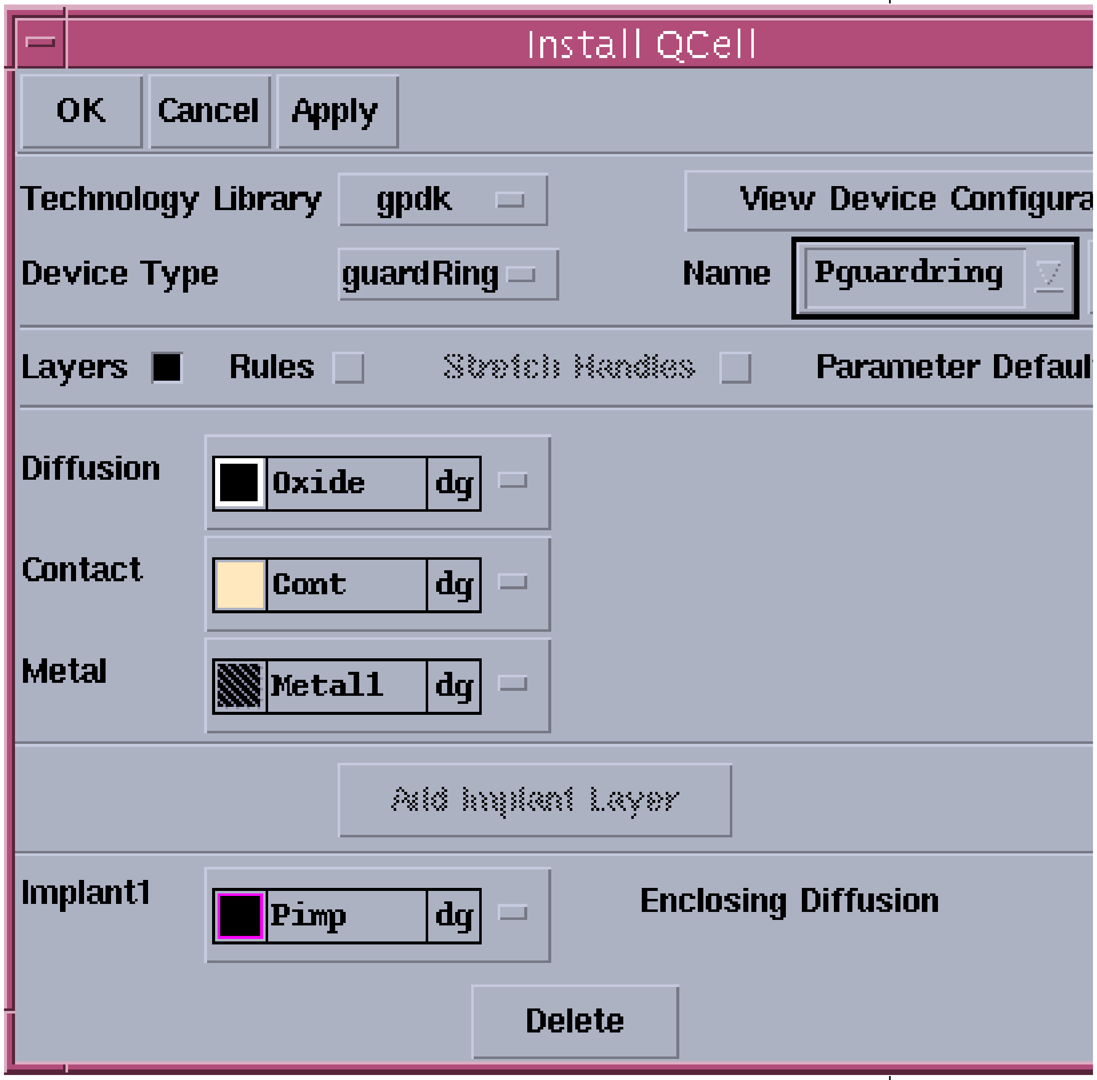
4. Notice that a guard ring called *Pguardring* has already been installed.

   Where does "Pguardring" come from?

   Answer: Use the **Dump** option from the Technology File Toolbox window and examine the content.

5. Select **File—Close** in the Technology File Toolbox window.

6. Review the Install Qcell form. It should look the example:



Technology Library gpdk
Device Type guardRing
Name Pguardring
Layers ■ Rules ☐ Stretch Handles ☐ Parameter Defaul

Diffusion	Oxide	dg
Contact	Cont	dg
Metal	Metal1	dg
Implant1	Pimp	dg

Enclosing Diffusion

7. Click on the **Rules** radio button toward the top of the Install Qcell form.

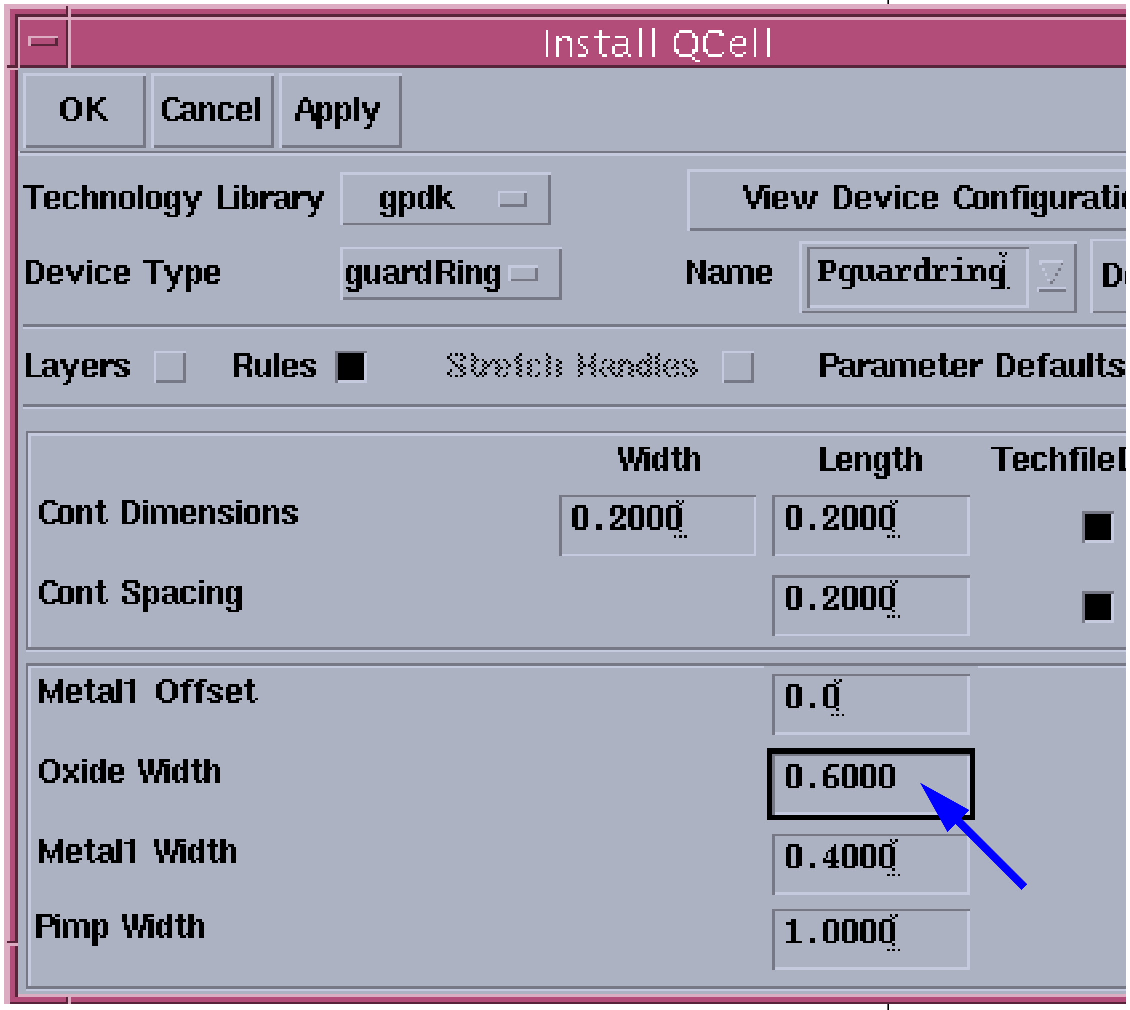
   The Install Qcell form reflects the rules defaulted from the technology file. The Rules Browser window also appears.

8. Move the Rules Browser window to the side of the Install Qcell form.

9. Click on the different rule values and notice that a yellow arrow appears in the Rules Browser window and illustrates the rule currently selected.

   Note: When you click on the Cont Dimensions **Width** or **Length** fields, the arrow appears on the actual contact.

10. Click on the field for the parameter **Oxide Width**.



Install QCell

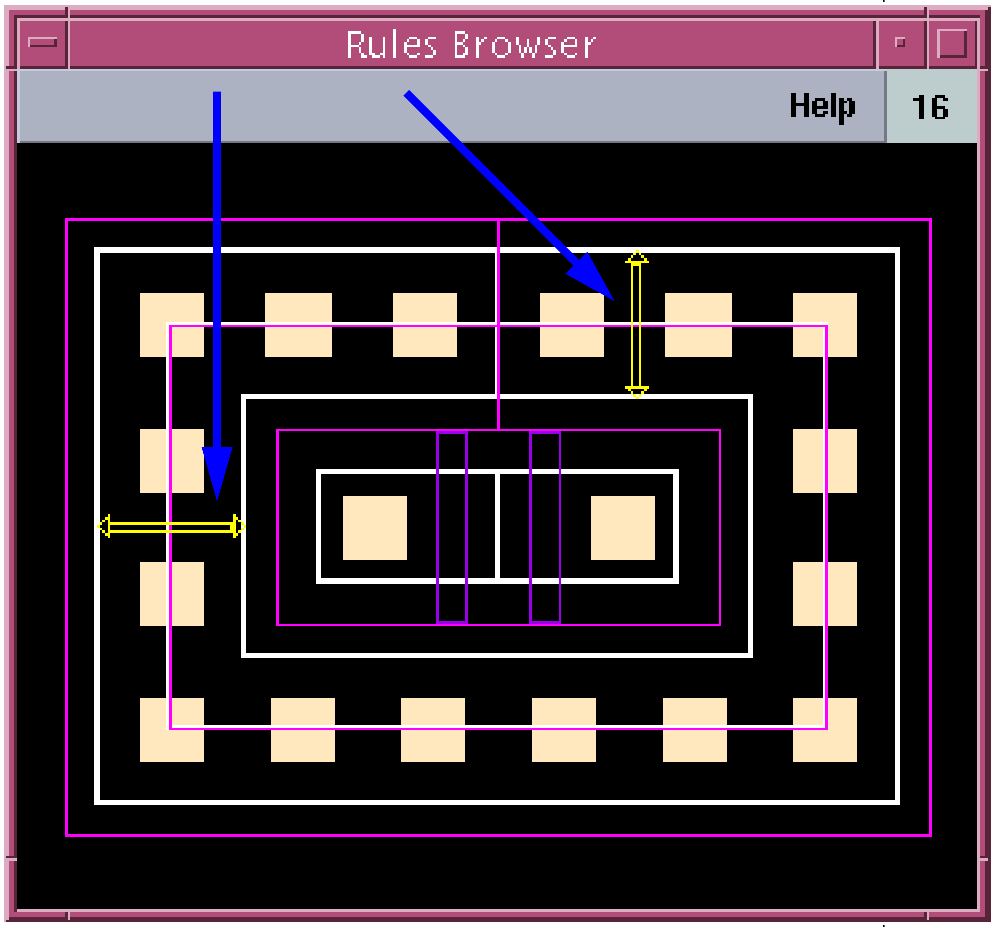
OK | Cancel | Apply

Technology Library: gpdk
View Device Configurati
Device Type: guardRing
Name: Pguardring
Layers ☐ Rules ■ Stretch Handles ☐ Parameter Defaults

	Width	Length	Techfile
Cont Dimensions	0.2000	0.2000	■
Cont Spacing		0.2000	■

Metal1 Offset	0.0
Oxide Width	0.6000
Metal1 Width	0.4000
Pimp Width	1.0000

11. Notice in the Rules Browser window that the same rule is illustrated with a yellow arrow denoting the oxide width.



What is the master path of Pguardring?

Note: This information is useful when you later specify a value for *Enclose By* when the guard ring is used.

12. Click **OK** within the Install Qcell form.

13. The system prompts with an Install Qcell window denoting:

```
Save the technology file to disk?
```

14. Click **Yes** to save the technology file to disk.

Note: You must have write access to the technology file in order to install the qcells. If you do not have access you will get an error message at this point.

Summary

In this lab, you learned the following:

- Evaluating the installed *Pguardring* qcell.

End of Lab

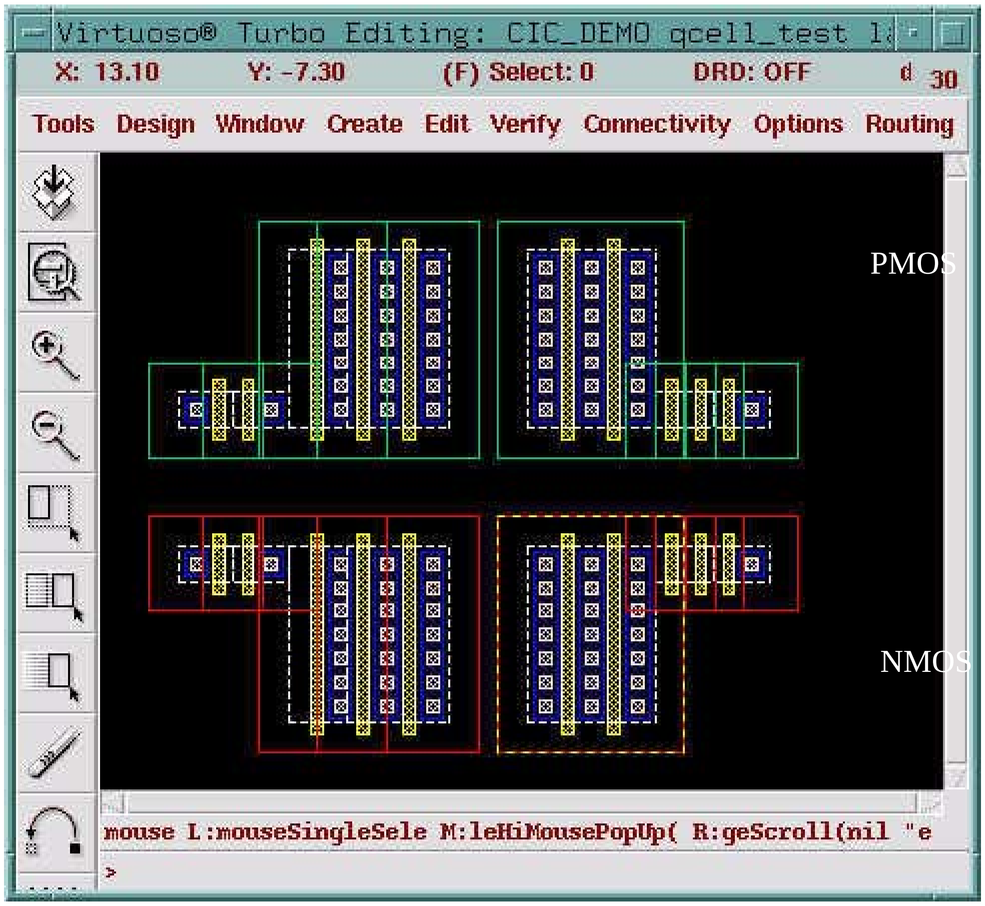
Lab 6-4 Creating Devices

Objective: Learn how to create qcell devices in the Virtuoso Layout Editor Turbo environment.

In this lab, you will experiment with creating qcell device chains that look like the one in the illustration. **Refer to the lecture material and CDSDoc anytime throughout the installation process**.

- CDSDoc: Virtuoso Layout Editor Turbo User Guide
- SourceLink:

  `http://sourcelink.cadence.com/docs/files/Release_Info/icoa5141/turbohelp/turbohelpTOC.html`



1. Open the following design:

Library	`Qcells`
Cell	`qcell_test`
View	`layout`

2. Invoke Virtuoso LE Turbo by selecting **Tools—Layout Turbo**.

   Notice that the layout window banner changes to Virtuoso Turbo Editing.

3. Select **Create—Device**.

   Press **F3** if the command options form does not appear.

   What is the shortcut to invoke Create—Device?

4. Create the NMOS device chains first by setting the **Name** field to *nmos_qcell*.

Important

Leave the Create Device form up at all times during device creation process. The form is set back to the defaults once you close it, so you lose all previous settings.

5. Take a close look at the NMOS device chains in the illustration at the beginning of the lab. Practice with different *Mos Patterns* with different w and finger number. Remember you have to move your cursor to the layout window to see the ghost image of device chain.

6. Create the NMOS device chains based on the illustration. Place them anywhere in the design. Do not close the Create Device form yet.

   Note: The solution is at the end of the lab. Try to go through the exercise without looking at the answer.

7. With the Create Device form still open, create the PMOS device chains by changing **Name** to *pmos_qcell* and **Alignment** to *bottom*.

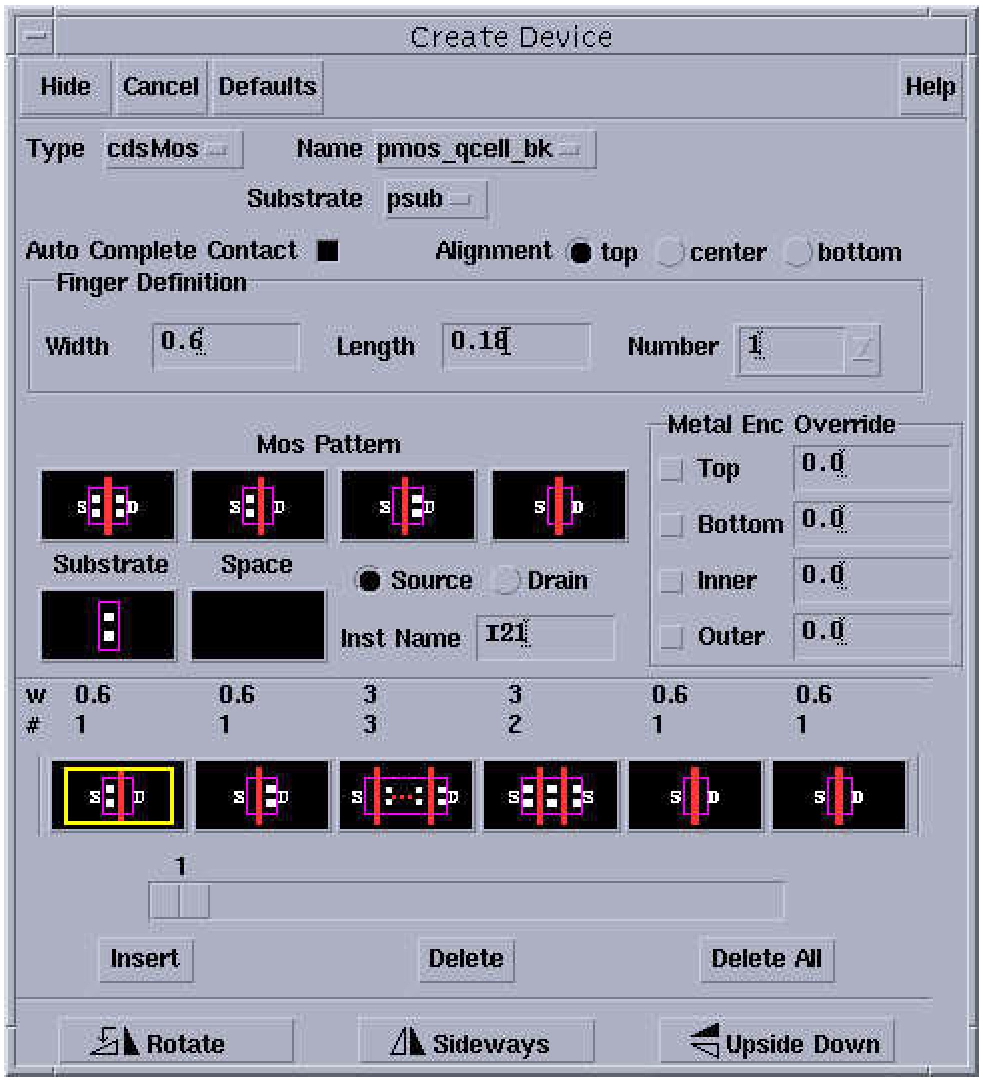
8. Place the PMOS chains above the NMOS chains so the gates align in the design.

   You should have both PMOS and NMOS chains placed in the design by now.

9. Select **Design—Save** and do not close the layout window. Make sure you have accomplished this before moving on to the next lab.

Solution to the NMOS Device Chain Problem: Left Side

The next two illustrations show the solution to the NMOS device chain problem. This one shows the left side of the row of devices. The horizontal scroll bar below the row of devices to create is scrolled to the left.



Create Device

Type cdsMos   Name pmos_qcell_bk
Substrate psub
Auto Complete Contact ■   Alignment [✓ top] center bottom

Finger Definition
Width 0.6   Length 0.18   Number 1

Metal Enc Override
Top 0.0
Bottom 0.0
Inner 0.0
Outer 0.0

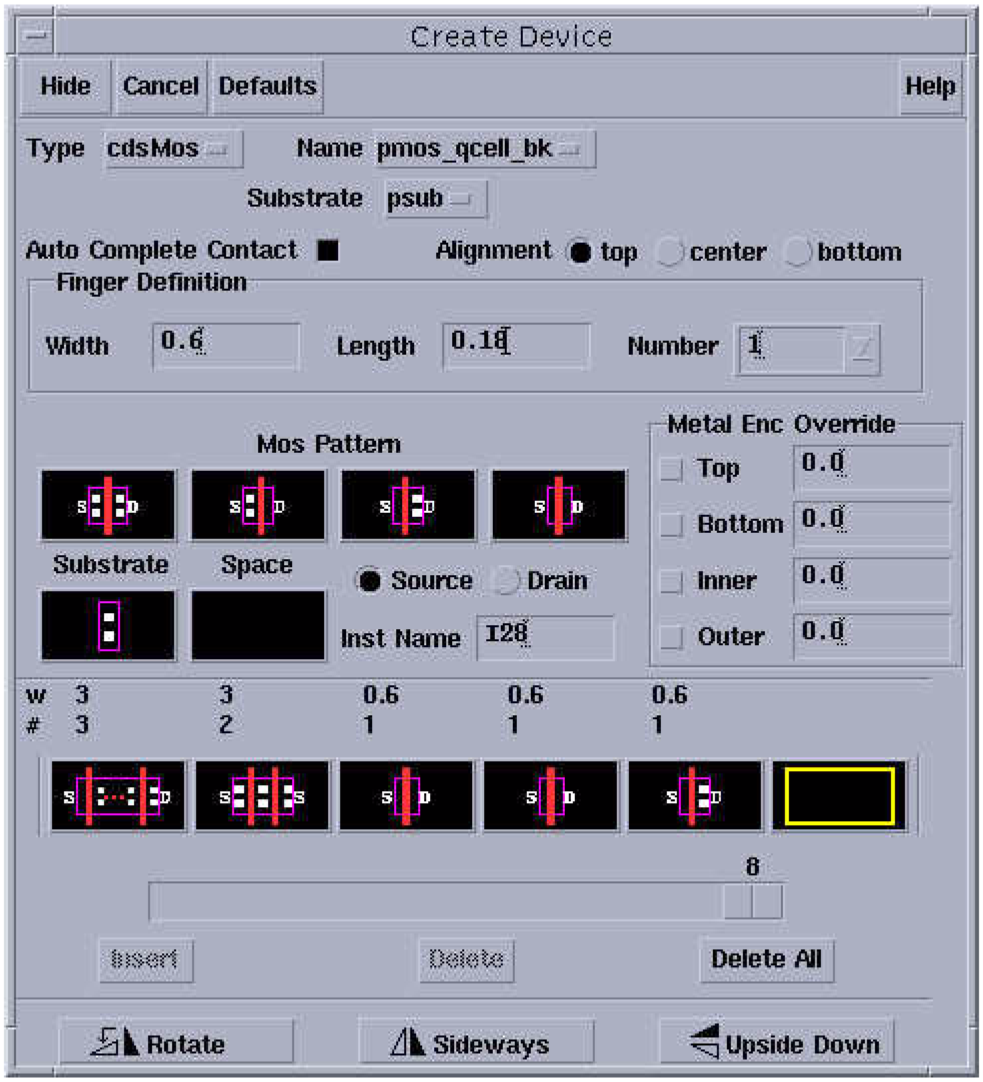
Mos Pattern
[✓ Source] Drain
Inst Name I21

w	0.6	0.6	3	3	0.6	0.6
#	1	1	3	2	1	1

Additional illustration on next page

Solution to the NMOS Device Chain Problem: Right Side

This illustration shows the same form as the previous one, but the horizontal scroll bar below the row of devices to create is scrolled to the right.



Create Device

Hide Cancel Defaults Help

Type cdsMos Name pmos_qcell_bk

Substrate psub

Auto Complete Contact ■ Alignment [✓ top] center bottom

Finger Definition

Width 0.6 Length 0.18 Number 1

Mos Pattern

Metal Enc Override

Top 0.0

Bottom 0.0

Inner 0.0

Outer 0.0

Substrate Space

[✓ Source] Drain

Inst Name I28

w	3	3	0.6	0.6	0.6
#	3	2	1	1	1

8

Insert Delete Delete All

Rotate Sideways Upside Down

Lab 6-5 Editing Devices

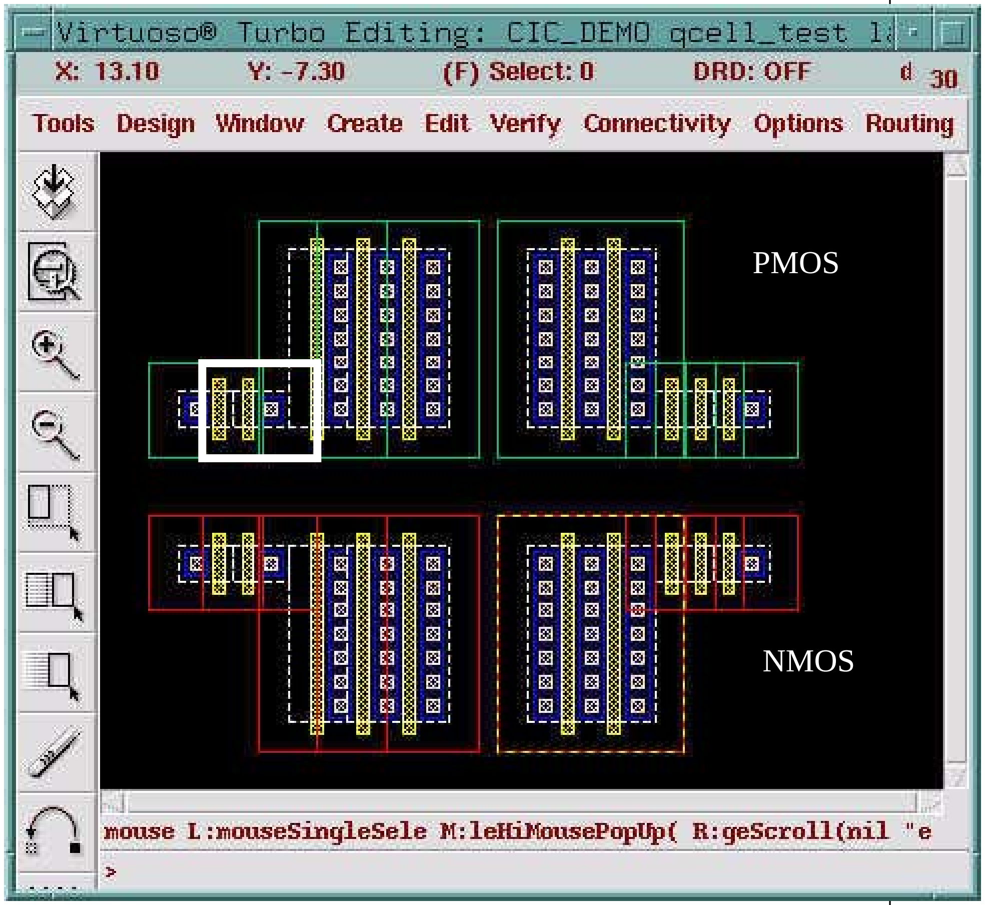
Objective: Learn how to edit qcell devices in the Virtuoso Layout Editor Turbo environment.

In this lab, you will experiment editing qcell device chains. You will add and remove a contact in an existing chain, move a partial chain, and create a guard ring. **Refer to the lecture material and CDSDoc anytime throughout the installation process**.

1. You should already have this design open on your desktop:

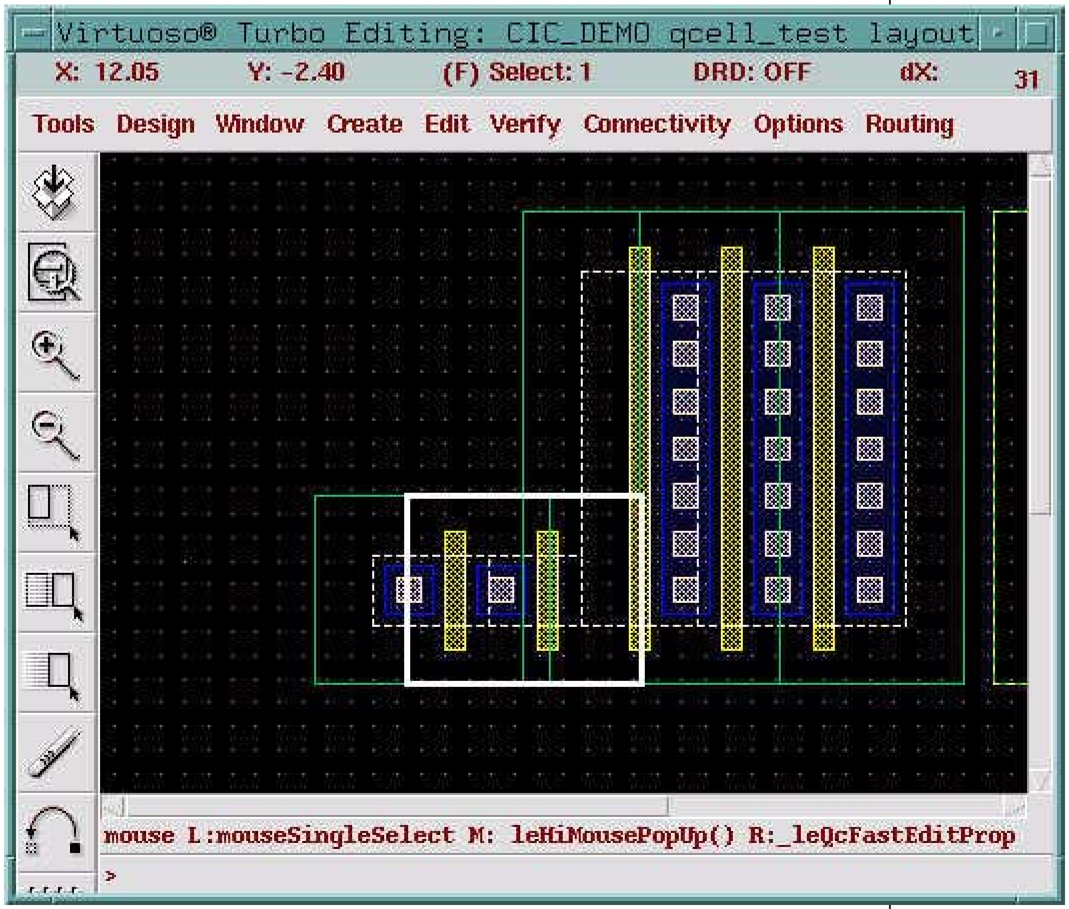
Library	`Qcells`
Cell	`qcell_test`
View	`layout`

The design looks like this:


Virtuoso® Turbo Editing: CIC_DEMO qcell_test l
X: 13.10 Y: -7.30 (F) Select: 0 DRD: OFF d 30
Tools Design Window Create Edit Verify Connectivity Options Routing
PMOS
NMOS
mouse L:mouseSingleSele M:leHiMousePopUp(R:geScroll(nil "e
>

The device you need to modify is the second one from the left in PMOS row in the previous illustration. The first goal is to add a contact on the left and remove the contact on the right.

The result should look like this:



```
mouse L:mouseSingleSelect M: leHiMousePopUp() R:_leQcFastEditProp
>
```

2. Select the device, then select **Edit—Properties** (or press **q**).

3. Update the Edit QCell Properties form to accomplish the goal.

   Is there a shortcut to this operation?

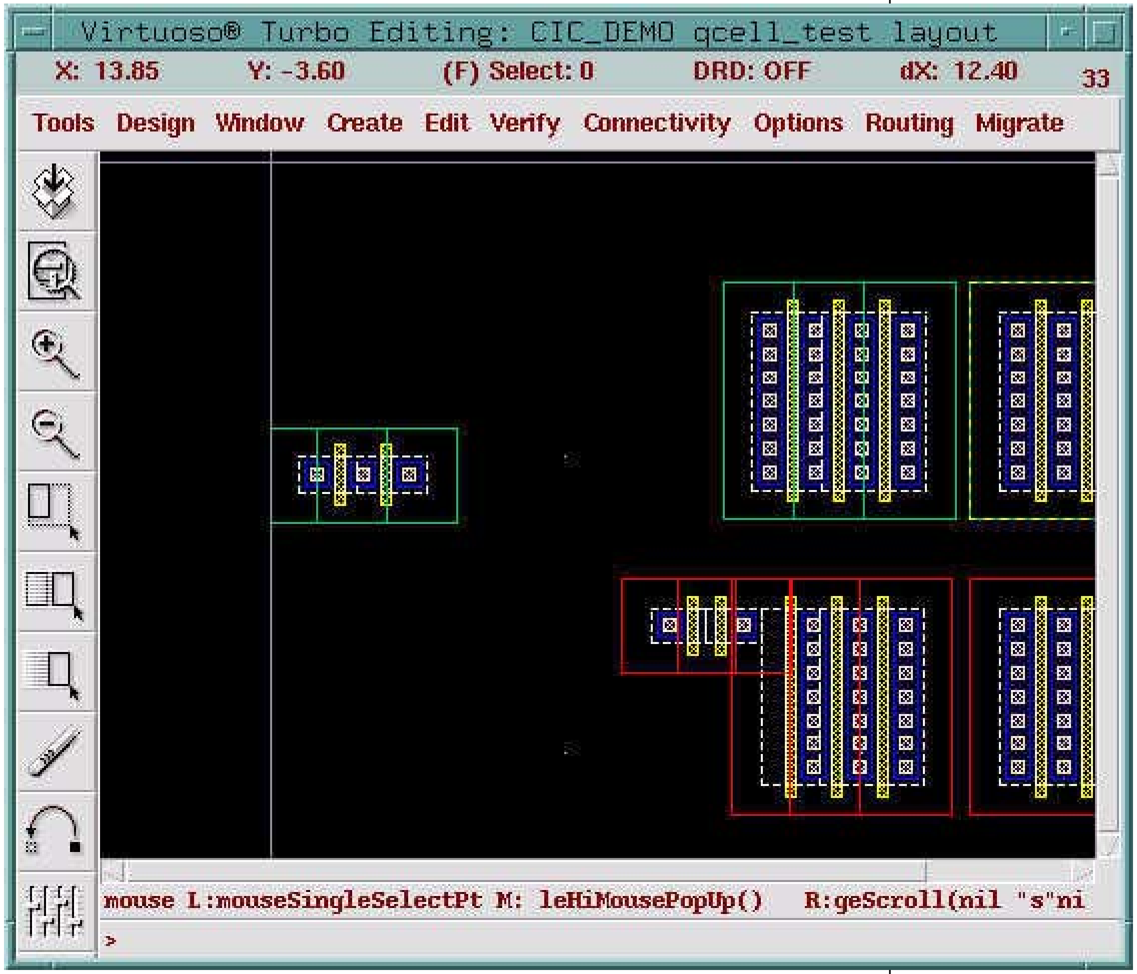
   Answer: Press the middle mouse button.

The next step is to move the selected device and everything to its left together to another location.

4. With the device still selected, select **Edit—Move** (or press **m**).

5. Press the middle mouse button and select **Chain Mode—Selected Plus Left**.

6. Move the selected partial chain to another location.


Virtuoso® Turbo Editing: CIC_DEMO qcell_test layout
X: 13.85 Y: -3.60 (F) Select: 0 DRD: OFF dX: 12.40 33
Tools Design Window Create Edit Verify Connectivity Options Routing Migrate

mouse L:mouseSingleSelectPt M: leHiMousePopUp() R:geScroll(nil "s"ni

>

7. Select **Create—Guard Ring** (or press **G**).

   The Guard Ring form appears.

8. "Area select" the devices that you just moved in the last step.

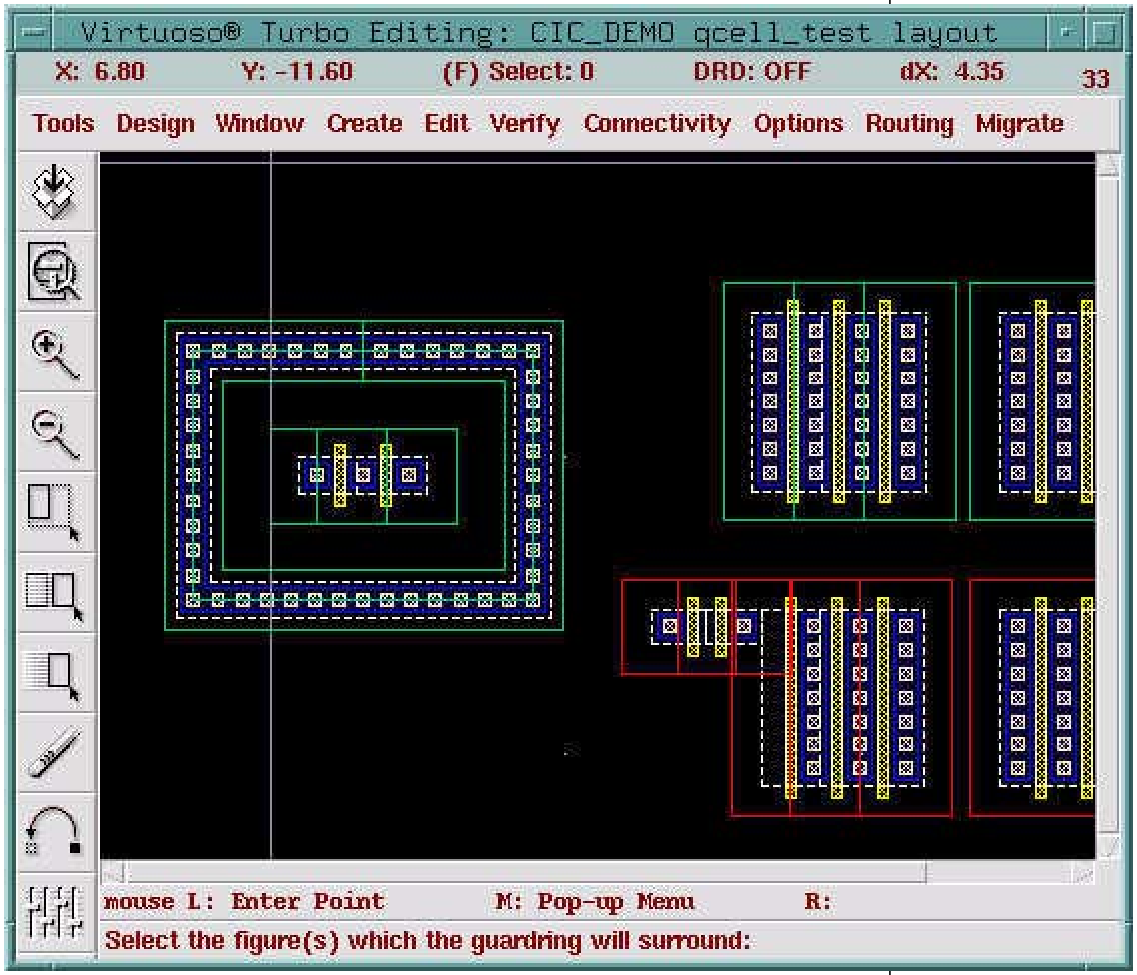
   Notice the ghost image of a guard ring that displays around the devices.

9. Change **Enclose by** to *1*.

   The guard ring size changes.

10. Double-click anywhere in the layout window to place the guard ring.

    The result should look like this:



mouse L: Enter Point M: Pop-up Menu R:

Select the figure(s) which the guardring will surround:

11. Select **Window—Create Ruler** (or press **k**) to measure the enclosure value.

    Remember that the distance is calculated from the master path of the guard ring.

12. Practice creating guard rings at other parts of the layout.

    The guard rings can be polygonal rather than rectangular, depending on the shape outline of the selected objects.

Summary

In this lab you:

- Added and moved a contact
- Selected and moved several devices at once

End of Lab

Appendix A

Selected Source Code

step1.il: Creating a Simple SKILL Pcell

```
;
;  This section of code uses ‘pcDefinePCell’ to create a simple pcell
;  containing one rectangle.  The name of the master view is:
;
;     pcells step1 layout
;
;  This pcell accepts these parameters:
;
;     xDistance  X-dimension of the rectangle.
;                (float, default = 2.4)
;
;     yDistance  Y-dimension of the rectangle.
;                (float, default = 1.2)
;
;     layer      The name of the rectangle’s drawing layer.
;                (string, default = “pdiff”)
;
pcDefinePCell(

   ;  Identify the target cellview.
   list(ddGetObj(“pcells”) “step1” “layout”)

   ;  Declare formal parameter name-value pairs.
   (
      (xDistance  2.4)
      (yDistance 1.2)
      (layer  “pdiff”)
   )
```

```
    ;  Define the contents of the cellview.
    let(()
       ;  Create the rectangle.
       rodCreateRect(
          ?cvId   pcCellView
          ?layer  list(layer “drawing”)
          ?width  xDistance
          ?length yDistance
       )
    )
 )
```

step4.il: Creating a Multipart Path Transistor

```
;
;  This pcell demonstrates the implementation of a transistor as a
;  multipart path.  All shapes and electrical information in the
;  transistor are created with a single call to ‘rodCreatePath’.  The
;  name of the master view is:
;
;     pcells step4 layout
;
;  This pcell accepts these parameters:
;
;     width       Width of the transistor’s active area (note: this is
;                 a distance measured in the Y-direction).
;                 (float, default = 3.0)
;
;     length      Length of the transistor’s active area (note: this is
;                 a distance measured in the X-direction).
;                 (float, default = 0.6)
;
;     polyLayer   Name of the poly layer.
;                 (string, default = “poly”)
;
;     diffLayer   Name of the diffusion layer.
;                 (string, default = “pdiff”)
;
;     contLayer   Name of the contact layer.
;                 (string, default = “cont”)
;
;     metalLayer  Name of the metal layer.
;                 (string, default = “metal1”)
;
;     drainName   Name of the drain connection.
;                 (string, default = “D”)
;
;     gateName    Name of the gate connection.
;                 (string, default = “G”)
;
```

```
;     sourceName  Name of the source connection.
;                 (string, default = “S”)
;
pcDefinePCell(
   ;  Identify the target cellview.
   list(ddGetObj(“pcells”) “step4” “layout”)

   ;  Define formal parameter name-value pairs.
   (
      (width      3.0)
      (length     0.6)
      (polyLayer  “poly”)
      (diffLayer  “pdiff”)
      (contLayer  “cont”)
      (metalLayer “metal1”)
      (drainName  “D”)
      (gateName   “G”)
      (sourceName “S”)
   )

   ;  Define the contents of this cellview.
   let((tfId polyExtend contWidth polyContSep diffContEnclose
        metalContEnclose)

      ;  Get the technology information for this cell.
      tfId = techGetTechFile(ddGetObj(“pcells”))

      ;  Get the minimum extension of poly beyond diffusion.
      polyExtend = techGetSpacingRule(tfId “minExtension” polyLayer)

      ;  Get the minimum contact width.
      contWidth = techGetSpacingRule(tfId “minWidth” contLayer)

      ;  Get the minimum poly to contact spacing.
      polyContSep = techGetSpacingRule(tfId “minSpacing” polyLayer contLayer)
```

```
    ;  Get the minimum diffusion enclosure of contact.
    diffContEnclose =
    techGetOrderedSpacingRule(tfId “minEnclosure” diffLayer contLayer)

    ;  Get the minimum metal enclosure of contact.
    metalContEnclose =
    techGetOrderedSpacingRule(tfId “minEnclosure” metalLayer contLayer)

    ;  Create the gate and diffusion regions.
    rodCreatePath(
       ?layer    list(polyLayer “drawing”)
 ?width    length
       ?pts      list(0.0:0.0 0.0:width + 2.0*polyExtend)
       ?encSubPath
       list(
          ;  Define the diffusion region.
          list(
             ?layer       list(diffLayer “drawing”)
             ?enclosure   -(contWidth + polyContSep + diffContEnclose)
             ?beginOffset -polyExtend
          )
       )
       ?offsetSubPath
       list(
          ;  Define the drain metal stripe.
          list(
             ?layer         list(metalLayer “drawing”)
             ?pin           t
             ?termName      drainName
             ?justification “left”
             ?width         contWidth + 2.0*metalContEnclose
             ?sep           polyContSep - metalContEnclose
             ?beginOffset   metalContEnclose - polyExtend - diffContEnclose
          )
          ;  Define the source metal stripe.
          list(
             ?layer         list(metalLayer “drawing”)
             ?pin           t
```

```
                ?termName      sourceName
                ?justification “right”
                ?width         contWidth + 2.0*metalContEnclose
                ?sep           polyContSep - metalContEnclose
                ?beginOffset   metalContEnclose - polyExtend - diffContEnclose
             )
          )
          ?subRect
          list(
             ;  Define the drain contact array.
             list(
                ?layer         list(contLayer “drawing”)
                ?justification “left”
                ?sep           polyContSep
                ?beginOffset   -(polyExtend + diffContEnclose)
             )
             ;  Define the source contact array.
             list(
                ?layer         list(contLayer “drawing”)
                ?justification “right”
                ?sep           polyContSep
                ?beginOffset   -(polyExtend + diffContEnclose)
             )
          )
       )
    )
 )
```

tran.il: Defining and Examining the Transistor Cell

```
;
;  This pcell implements a simple transistor with various ROD objects.  Each
;  transistor instance created with this pcell contains a single gate
;  structure plus drain and source connections complete with contact arrays.
;  This pcell uses technology information so that it can adapt to process
;  changes.
;
;  The transistor’s width and length can be modified by means of parameters
;  of the same name.  Other parameters involve layer names and signal names.
;  See below for more information.
;
;  The name of the master view is:
;
;     stdcells tran layout
;
;  This pcell accepts these parameters:
;
;     width       Width of the transistor’s active area (note: this is a
;                 distance measured in the Y-direction).
;                 (float, default = 3.0)
;
;     length      Length of the transistor’s active area (note: this is a
;                 distance measured in the X-direction).
;                 (float, default = 0.6)
;
;     polyLayer   Name of the poly layer.
;                 (string, default = “poly”)
;
;     diffLayer   Name of the diffusion layer.
;                 (string, default = “pdiff”)
;
;     contLayer   Name of the contact layer.
;                 (string, default = “cont”)
;
;     metalLayer  Name of the metal layer.
;                 (string, default = “metal1”)
;
```

```
;     drainName   Name of the drain connection.
;                 (string, default = “D”)
;
;     gateName    Name of the gate connection.
;                 (string, default = “G”)
;
;     sourceName  Name of the source connection.
;                 (string, default = “S”)
;
pcDefinePCell(
   ;  Identify the target cellview.
   list(ddGetObj(“stdcells”) “tran” “layout”)

   ;  Define formal parameter name-value pairs.
   (
      (width      3.0)
      (length     0.6)
      (polyLayer  “poly”)
      (diffLayer  “pdiff”)
      (contLayer  “cont”)
      (metalLayer “metal1”)
      (drainName  “D”)
      (gateName   “G”)
      (sourceName “S”)
   )

   ;  Define the contents of this cellview.
   let((tfId polyExtend contWidth polyContSep diffContEnclose
        metalContEnclose transObj)

      ;  Get the technology information for this cell.
      tfId = techGetTechFile(ddGetObj(“stdcells”))

      ;  Get the minimum extension of poly beyond diffusion.
      polyExtend = techGetSpacingRule(tfId “minExtension” polyLayer)

      ;  Get the minimum contact width.
      contWidth = techGetSpacingRule(tfId “minWidth” contLayer)
```

```
        ;  Get the minimum poly to contact spacing.
        polyContSep = techGetSpacingRule(tfId “minSpacing” polyLayer contLayer)

        ;  Get the minimum diffusion enclosure of contact.
        diffContEnclose =
        techGetOrderedSpacingRule(tfId “minEnclosure” diffLayer contLayer)

        ;  Get the minimum metal enclosure of contact.
        metalContEnclose =
        techGetOrderedSpacingRule(tfId “minEnclosure” metalLayer contLayer)

        ;  Create the gate and diffusion regions.
        transObj = rodCreatePath(
           ?name  “gate”
           ?endType “variable”
           ?beginExt polyExtend
           ?layer list(polyLayer “drawing”)
     ?width length
           ?pts   list(0.0:0.0 0.0:width)
           ;  Define the diffusion region.
           ?encSubPath
           list(
              list(
                 ?layer       list(diffLayer “drawing”)
                 ?enclosure   -(contWidth + polyContSep + diffContEnclose)
                 ?beginOffset -polyExtend
              )
           )
        )

        ;  Create the drain connection.
        rodCreatePath(
           ?name     “drain”
           ?termName drainName
           ?pin      t
           ?layer    list(metalLayer “drawing”)
           ?width    contWidth + 2.0*metalContEnclose
```

```
      ?pts
      list(
         rodAddPoints(transObj~>startLeft0
                      -polyContSep - contWidth/2.0:
                      polyExtend + diffContEnclose - metalContEnclose)
         rodSubPoints(transObj~>endLeft0
                      polyContSep + contWidth/2.0:
                      polyExtend + diffContEnclose - metalContEnclose)
      )
      ?subRect
      list(
         list(
            ?layer       contLayer
            ?beginOffset -metalContEnclose
         )
      )
   )

   ;  Create the source connection.
   rodCreatePath(
      ?name     “source”
      ?termName sourceName
      ?pin      t
      ?layer    list(metalLayer “drawing”)
      ?width    contWidth + 2.0*metalContEnclose
      ?pts
      list(
         rodAddPoints(transObj~>startRight0
                      polyContSep + contWidth/2.0:
                      polyExtend + diffContEnclose - metalContEnclose)
         rodAddPoints(transObj~>endRight0
                      polyContSep + contWidth/2.0:
                      metalContEnclose - polyExtend - diffContEnclose)
      )
      ?subRect
```

```
          list(
             list(
                ?layer       contLayer
                ?beginOffset -metalContEnclose
             )
          )
       )
    )
 )
```

inv3.il: Using ROD Points in Hierarchy

```
;
;  This pcell implements a simple inverter.  The cell contains 2 instances
;  of the ‘stdcells tran layout’ pcell.  All necessary pins and signals are
;  defined. This pcell makes use of technology information so that it can
;  adapt to process changes.
;
;  The name of the master view is:
;
;     stdcells inv layout
;
;  This pcell depends upon:
;
;     stdcells tran layout
;
;  This pcell accepts these parameters:
;
;     cellWidth   Width of the resulting cell.  The resulting cell width
;                 will be at least the same as the width of the widest
;                 transistor.
;                 (float, default = 0.0)
;
;     cellHeight  Height of the resulting cell.  The resulting cell height
;                 will be adjusted to accommodate design rules, if necessary.
;                 (float, default = 20.0)
;
;     pWidth      Width of the P-transistor.
;                 (float, default = 6.0)
;
;     pLength     Length of the P-transistor.
;                 (float, default = 1.2)
;
;     nWidth      Width of the N-transistor.
;                 (float, default = 3.0)
;
;     nLength     Length of the N-transistor.
;                 (float, default = 0.6)
;
```

```
;     pwrWidth    Width of the power rail.
;                 (float, default = 1.8)
;
;     gndWidth    Width of the ground rail.
;                 (float, default = 1.8)
;
;     inputName   Name of the input connection.
;                 (string, default = “in”)
;
;     outputName  Name of the output connection.
;                 (string, default = “out”)
;
;     powerName   Name of the power connection.
;                 (string, default = “vdd”)
;
;     groundName  Name of the ground connection.
;                 (string, default = “gnd”)
;
;     polyLayer   Name of the poly layer.
;                 (string, default = “poly”)
;
;     contLayer   Name of the contact layer.
;                 (string, default = “poly”)
;
;     metalLayer  Name of the metal layer.
;                 (string, default = “metal1”)
;
;     nwellLayer  Name of the N-type well layer.
;                 (string, default = “nwell”)
;
;     pdiffLayer  Name of the P-type diffusion layer.
;                 (string, default = “pdiff”)
;
;     ndiffLayer  Name of the N-type diffusion layer.
;                 (string, default = “ndiff”)
;
pcDefinePCell(
   ;  Identify the target cellview.
```

```
list(ddGetObj(“stdcells”) “inv” “layout”)

;  Define formal parameter name-value pairs.
(
   (cellWidth  0.0)
   (cellHeight 20.0)
   (pWidth     6.0)
   (pLength    1.2)
   (nWidth     3.0)
   (nLength    0.6)
   (pwrWidth   1.8)
   (gndWidth   1.8)
   (inputName  “in”)
   (outputName “out”)
   (powerName  “vdd”)
   (groundName “gnd”)
   (polyLayer  “poly”)
   (contLayer  “cont”)
   (metalLayer “metal1”)
   (nwellLayer “nwell”)
   (pdiffLayer “pdiff”)
   (ndiffLayer “ndiff”)
)

;  Define the contents of this cellview.
let((tfId contWidth nwellPdiffEnclose metalContEnclose
     polyWidth polyPdiffSep polyNdiffSep metalWidth polyContEnclose
     metalMetalSep nDrainObj pDrainObj tranId nTranObj pTranObj
     nWellObj nGateObj pGateObj pwrObj gndObj inPinObj outPinObj
     nGroundTieObj pPowerTieObj gateConnObj nSourceObj pSourceObj
     outputConnObj minHeight)

   ;  Get the technology information for this cell.
   tfId = techGetTechFile(pcCellView)

   ;  Get the minimum contact width.
   contWidth = techGetSpacingRule(tfId “minWidth” contLayer)
```

```
 ;  Get the minimum poly width.
 polyWidth = techGetSpacingRule(tfId “minWidth” polyLayer)

 ;  Get the minimum metal width.
 metalWidth = techGetSpacingRule(tfId “minWidth” metalLayer)

 ;  Get the minimum poly to P-diffusion spacing.
polyPdiffSep = techGetSpacingRule(tfId “minSpacing” polyLayer pdiffLayer)

 ;  Get the minimum poly to N-diffusion spacing.
polyNdiffSep = techGetSpacingRule(tfId “minSpacing” polyLayer ndiffLayer)

 ;  Get the minimum N-well enclosure of P-diffusion.
 nwellPdiffEnclose =
 techGetOrderedSpacingRule(tfId “minEnclosure” nwellLayer pdiffLayer)

 ;  Get the minimum metal enclosure of contact.
 metalContEnclose =
 techGetOrderedSpacingRule(tfId “minEnclosure” metalLayer contLayer)

 ;  Get the minimum poly enclosure of contact.
 polyContEnclose =
 techGetOrderedSpacingRule(tfId “minEnclosure” polyLayer contLayer)

 ;  Get the minimum metal-to-metal spacing.
 metalMetalSep = techGetSpacingRule(tfId “minSpacing” metalLayer)

 ;  Open the ‘tran’ pcell.
 tranId = dbOpenCellViewByType(“stdcells” “tran” “layout”)

 ;  Create the N-type transistor instance.
 dbCreateParamInst(
    pcCellView tranId “ntran” (0:0) “R0” 1
    list(
       list(“width” “float” nWidth)
       list(“length” “float” nLength)
       list(“diffLayer” “string” ndiffLayer)
    )
)

;  Create the P-type transistor instance.
dbCreateParamInst(
   pcCellView tranId “ptran” (0:10) “R0” 1
   list(
      list(“width” “float” pWidth)
      list(“length” “float” pLength)
      list(“diffLayer” “string” pdiffLayer)
   )
)

;  Close the ‘tran’ pcell.
dbClose(tranId)

;  Get the ROD ID of the N-transistor.
nTranObj = rodGetObj(“ntran”)

;  Get the ROD ID of the N-transistor’s gate path.
nGateObj = rodGetObj(“ntran/gate”)

;  Get the N-transistor drain connection ROD ID.
nDrainObj = rodGetObj(“ntran/drain”)

;  Get the N-transistor source connection ROD ID.
nSourceObj = rodGetObj(“ntran/source”)

;  Get the ROD ID of the new P-transistor.
pTranObj = rodGetObj(“ptran”)

;  Get the ROD ID of the P-transistor’s gate path.
pGateObj = rodGetObj(“ptran/gate”)

;  Get the P-transistor drain connection ROD ID.
pDrainObj = rodGetObj(“ptran/drain”)

;  Get the P-transistor source connection ROD ID.
pSourceObj = rodGetObj(“ptran/source”)
```

```
;  Calculate the minimum height (based on design rules) for this cell.
minHeight =
2.0*(metalMetalSep + polyContEnclose) + contWidth + pwrWidth +
gndWidth + nGateObj~>length0 + pGateObj~>length0 + polyPdiffSep +
polyNdiffSep

;  Adjust the minimum height of the cell, if necessary.
when(cellHeight < minHeight
   cellHeight = minHeight
)

;  Calculate the width of the power and ground connections based on
;  the widest transistor.
if(pTranObj~>width > nTranObj~>width
then
   cellWidth = if(pTranObj~>width > cellWidth pTranObj~>width cellWidth)
else
   cellWidth = if(nTranObj~>width > cellWidth nTranObj~>width cellWidth)
)

; Create the power rail.
pwrObj = rodCreateRect(
   ?name “powerRail”
   ?pin t
   ?termName powerName
   ?layer list(metalLayer “drawing”)
   ?width cellWidth
   ?length pwrWidth
)

; Create the ground rail.
gndObj = rodCreateRect(
   ?name “groundRail”
   ?pin t
   ?termName groundName
   ?layer list(metalLayer “drawing”)
   ?width cellWidth
   ?length gndWidth
)

;  Align the power and ground rails.
rodAlign(
   ?alignObj pwrObj
   ?alignHandle “upperCenter”
   ?refObj gndObj
   ?refHandle “lowerCenter”
   ?ySep cellHeight
   ?maintain nil
)

;  Align the P-transistor with the power rail.
rodAlign(
   ?alignObj pTranObj
   ?alignHandle “upperCenter”
   ?refObj pwrObj
   ?refHandle “lowerCenter”
   ?ySep rodPointY(rodSubPoints(pTranObj~>upperCenter
                               pDrainObj~>upperCenter)) - metalMetalSep
   ?maintain nil
)

;  Align the N-transistor with the ground rail.
rodAlign(
   ?alignObj nTranObj
   ?alignHandle “lowerCenter”
   ?refObj gndObj
   ?refHandle “upperCenter”
   ?ySep rodPointY(rodSubPoints(nTranObj~>lowerCenter
                               nDrainObj~>lowerCenter)) + metalMetalSep
   ?maintain nil
)

; Create the gate connection between the transistors.
gateConnObj = rodCreatePath(
   ?name “inputConn”
```

```
    ?layer list(polyLayer “drawing”)
    ?width polyWidth
    ?pts list(nGateObj~>upperCenter pGateObj~>lowerCenter)
)

;  Create the connection from power to the P-transistor drain.
pPowerTieObj = rodCreatePath(
    ?termName powerName
    ?layer list(metalLayer “drawing”)
    ?width pDrainObj~>width
    ?pts
    list(
        pDrainObj~>upperCenter
        rodAddToY(pDrainObj~>upperCenter metalMetalSep)
    )
)

;  Create the connection from ground to the N-transistor drain.
nGroundTieObj = rodCreatePath(
    ?termName groundName
    ?layer list(metalLayer “drawing”)
    ?width nDrainObj~>width
    ?pts
    list(
        nDrainObj~>lowerCenter
        rodAddToY(nDrainObj~>lowerCenter -metalMetalSep)
    )
)

;  Place a well around the P-transistor.
rodCreatePath(
    ?layer list(nwellLayer “drawing”)
    ?width pTranObj~>width + 2.0*nwellPdiffEnclose
    ?endType “variable”
    ?beginExt nwellPdiffEnclose
    ?endExt nwellPdiffEnclose
    ?pts list(pGateObj~>start0 pGateObj~>endLast)
)
```

```
;  Create the input pin structure.
inPinObj = rodCreatePath(
   ?name “inputPin”
   ?termName inputName
   ?pin t
   ?termIOType “input”
   ?pinLabel t
   ?layer list(polyLayer “drawing”)
   ?width contWidth + 2.0*polyContEnclose
   ?pts
   list(
      gateConnObj~>centerLeft
      rodPointX(gateConnObj~>centerLeft) -
     contWidth - 2.0*polyContEnclose:rodPointY(gateConnObj~>centerLeft)
   )
   ?encSubPath
   list(
      ;  Create the central contact.
      list(
         ?layer list(contLayer “drawing”)
         ?enclosure polyContEnclose
         ?beginOffset -polyContEnclose
         ?endOffset -polyContEnclose
      )
      ;  Create the metal pad.
      list(
         ?layer list(metalLayer “drawing”)
         ?enclosure polyContEnclose - metalContEnclose
         ?beginOffset metalContEnclose - polyContEnclose
         ?endOffset metalContEnclose - polyContEnclose
      )
   )
)

;  Create the connection between the sources of the transistors.
outputConnObj = rodCreatePath(
   ?name “outputConn”
```

```
         ?layer list(metalLayer “drawing”)
         ?width metalWidth
         ?pts
         list(
            rodAddToX(nSourceObj~>upperRight -metalWidth/2.0)
            rodPointX(nSourceObj~>upperRight) - metalWidth/2.0:
            rodPointY(inPinObj~>centerCenter)
            rodPointX(pSourceObj~>lowerRight) - metalWidth/2.0:
            rodPointY(inPinObj~>centerCenter)
            rodAddToX(pSourceObj~>lowerRight -metalWidth/2.0)
         )
      )

      ;  Create the metal pad for the output connection.
      outPinObj = rodCreateRect(
         ?termName outputName
         ?pin t
         ?termIOType “output”
         ?pinLabel t
         ?layer list(metalLayer “drawing”)
         ?width contWidth + 2.0*metalContEnclose
         ?length contWidth + 2.0*metalContEnclose
      )

      ;  Align the output metal pad to the jog in the source connection.
      ;  If the jog is not present, place the pad in the middle of the
      ;  metal stripe.
      rodAlign(
         ?alignObj outPinObj
         ?alignHandle “centerCenter”
         ?refObj outputConnObj
         ?refHandle if(outputConnObj~>numSegments == 1 “centerCenter” “mid1”)
         ?maintain nil
      )
   )
)
```

invCDF.il: Adding a CDF Parameter to the Inverter Cell

```
;
;  This code adds base CDF information to the pcell ‘stdcells inv’.  A
;  CDF parameter is added for each of the pcell parameters as well as a
;  CDF parameter called “size”.  The “size” parameter is implemented as
;  a cyclic field with choices of “A”, “B”, “C”, “D”, “E”, and “F”
;  corresponding to standard sizes for the inverter.
;
let((cellId cdfId)

   ;  Get the database ID for this cellview.
   cellId = ddGetObj(“stdcells” “inv”)

   ;  Delete the old CDF if it exists.
   when( cdfId = cdfGetBaseCellCDF(cellId)
      cdfDeleteCDF(cdfId)
   )

   ;  Create new base CDF information for this cell.
   cdfId = cdfCreateBaseCellCDF( cellId )

   ;  Add a CDF parameter for standard sizes.
   cdfCreateParam( cdfId
     ?name     “size”
     ?type     “cyclic”
     ?prompt   “Size”
     ?choices  list(“A” “B” “C” “D” “E” “F”)
     ?defValue “C”
     ?callback “SPCsetInvSize()”
   )

   ;  Open the ‘stdcells inv layout’ cellview.
   unless(cv = dbOpenCellViewByType(“stdcells” “inv” “layout”)
      error(“couldn’t read ‘stdcells inv layout’.”)
   )

   ;  Add a CDF parameter for each of the pcell parameters on the
   ;  cellview.
```

```
      foreach(param reverse(cv~>parameters~>value)
         cdfCreateParam( cdfId
           ?name     param~>name
           ?type     param~>valueType
           ?prompt   param~>name
           ?defValue param~>value
         )
      )

      ;  Close the ‘stdcells inv layout’ cellview.
      dbClose(cv)

      ;  Save the new CDF for the cell.
      cdfSaveCDF(cdfId)
      )
   )



   ;
   ;  This function sets the parameters for an inverter instance to appropriate
   ;  values when the user selects a size from the CDF parameter called “size”.
   ;
   procedure(SPCsetInvSize()
      let(()

         ;  Set the cell height to a standard value.
         cdfgData->cellHeight->value = 28.0

         ;  Set other parameters based on the “size” selection.
         case(cdfgData->size->value
            (“A”
               cdfgData->pWidth->value = 4.0
               cdfgData->pLength->value = 1.2
               cdfgData->nWidth->value = 2.0
               cdfgData->nLength->value = 0.6
            )
            (“B”
               cdfgData->pWidth->value = 6.0
```

```
                 cdfgData->pLength->value = 1.2
                 cdfgData->nWidth->value = 3.0
                 cdfgData->nLength->value = 0.6
              )
              (“C”
                 cdfgData->pWidth->value = 8.0
                 cdfgData->pLength->value = 2.0
                 cdfgData->nWidth->value = 4.0
                 cdfgData->nLength->value = 1.2
              )
              (“D”
                 cdfgData->pWidth->value = 9.0
                 cdfgData->pLength->value = 2.4
                 cdfgData->nWidth->value = 4.2
                 cdfgData->nLength->value = 1.2
              )
              (“E”
                 cdfgData->pWidth->value = 12.0
                 cdfgData->pLength->value = 2.4
                 cdfgData->nWidth->value = 4.8
                 cdfgData->nLength->value = 1.2
              )
              (“F”
                 cdfgData->pWidth->value = 14.0
                 cdfgData->pLength->value = 4.0
                 cdfgData->nWidth->value = 5.2
                 cdfgData->nLength->value = 1.8
              )
           )
        )
  )
```

ExConnect.il: Establishing Connectivity for PCell Testing

```
procedure( ExConnect()
  let( ( i1 i2 instlist net cv )
    cv = geGetEditCellView()
    instlist = cv~>instances
    i1 = car(instlist)
    i2 = cadr(instlist)
    net = dbMakeNet( cv "shared" )
    dbCreateConnByName( net i1 "S")
    dbCreateConnByName( net i2 "D")
  ) ; let
) ; procedure ExConnect
```

Appendix B

Lab: Using a SKILL GUI to View All Pcells in a Library

Lab B-1 Using a SKILL GUI to View All Pcells in a Library

Objective: Load and run a SKILL program with a custom GUI to view all pcells in a library.

In this exercise you load and run a custom SKILL program to view all pcells in a library. You can use this program to quickly locate all pcells and view the parameters of those pcells. The parameters are identified as local or CDF. You can change the parameters and place an instance of the pcell also.

You will be running this SKILL program on the Cadence® Generic Process Design Kit (GPDK). The latest version of the GPDK is available online at *http://www.cadencepdk.com*.

Starting a Design Framework II Session

1. Enter these commands into a terminal window:
   ```
   cd ~/Skill_PCells_5_1_41/GPDK
   layout &
   ```
   After a few moments, the Command Interpreter Window (CIW) appears.

2. Open a new cellview.

Library	`gpdk`
Cell	`pcelltest`
View	`layout`

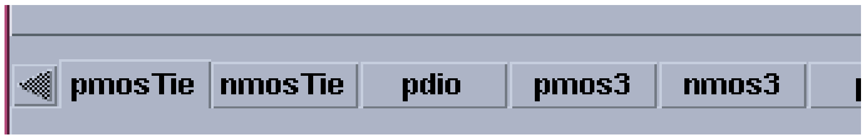
Loading and Running the SKILL GUI

1. Load the SKILL program. In the CIW, enter:
   ```
   load(“./TrPcellGuiEnh.il”)
   ```

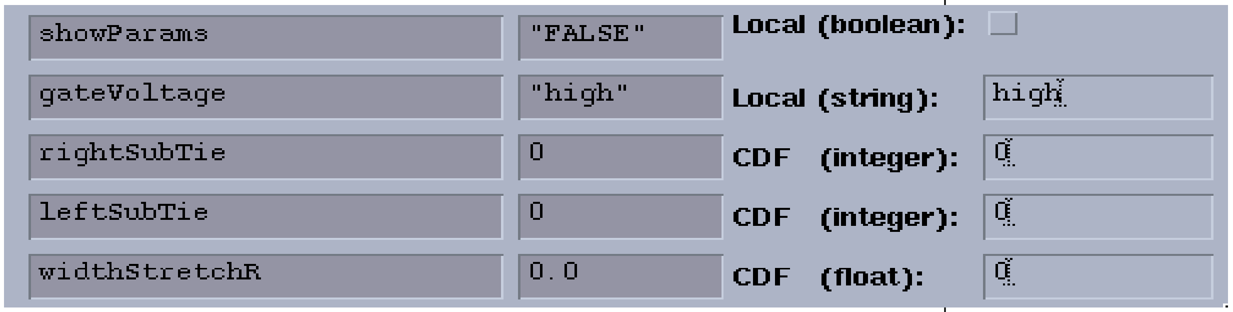
2. Run the SKILL program. In the CIW, enter:
   ```
     TrPcellGui( “gpdk” )
   ```
   After a few moments the pcell GUI form appears.

3. In the pcell GUI form, click the tabs containing the names of the pcells.

   As each tab is selected the parameters for that pcell appear. The grayed-out fields are the default values.



4. Select a pcell and change the parameters of the pcell. Be careful to enter the correct type of data for each parameter you change.



showParams	"FALSE"	Local (boolean):	☐
gateVoltage	"high"	Local (string):	high
rightSubTie	0	CDF (integer):	0
leftSubTie	0	CDF (integer):	0
widthStretchR	0.0	CDF (float):	0

5. Click **Apply** to place an instance of the modified pcell in the *gpdk pcelltest layout* cellview. It will be placed at 0,0.

6. Move or remove the instance before placing additional pcell instances as the new ones will all be placed at 0,0 also.

End of Lab