

Chapter 10

SOC Encounter Place and Route

PLACE AND ROUTE is the process of taking a structural file (Verilog in our case) and making a physical chip from that description. It involves *placing* the cells on the chip, and *routing* the wiring connections between those cells. The structural Verilog file gives all the information about the circuit that should be assembled. It is simply a list of standard cells and the connections between those cells. The cell layouts are used to place the cells physically on the chip. More accurately, the **abstract** views of the cells are used. The abstracts have only information about the connection points of the cells (the **pins**), and *routing obstructions* in those cells. Obstructions are simply layers of material in the cell that conflict with the layers that the routing tool wants to use for making the connections. Essentially it is a layout view of the cell that is restricted to pins and metal routing layers. This reduces the amount of information that's required by the place and route tool, and it also lets the vendor of the cells to keep other details of their cells (like transistor information) private. It's not need for place and route, so it's not included in the **abstract** view.

The files required before starting place and route are:

Cell characterization data: This should be in a **liberty** (or **<filename>.lib**) formatted file. It is the detailed timing, power, and functionality information that you derived through the characterization process (Chapter 7). It's possible that you might have up to three different **.lib** files with typ, worst, and best timing, but you can complete the process with only a single **.lib** file. It is very important that your **.lib** file include footprints for all cells. In particular you will need to know the footprint of inverter, buffer, and delay cells (delay cells can just be buffers or inverters). If you have special buffers/inverters for building

clock trees, those should use a different footprint than the “regular” buffers and inverters. If you have multiple drive strengths of any cells with the same functionality, those cells should have the same footprint. This enables certain timing optimizations in the place and route tool.

You might have multiple **.lib** files if your structural Verilog uses cells from multiple libraries.

Cell abstract information: This is information that you generated through the **abstract** process (Chapter 9), and is contained in a **LEF** (or **<filename>.lef**) file. The LEF file should include technology information and macro information about all the cells in your library.

You might have multiple **.lef** files if your structural Verilog uses cells from multiple different libraries.

Structural Verilog: This file defines the circuit that you want to have assembled by the place and route tool. It should be a purely structural description that contains nothing but instantiations of cells from your library or libraries.

If your design is hierarchical you might have multiple Verilog files that describe the complete design. That is, some Verilog modules might include instantiations of other modules in addition to just cells. In any case you should know the name of the top-level module that is the circuit that you want to place and route.

Delay constraint information: This is used by the place and router during timing optimization and clock tree generation. It describes the timing and loading of primary inputs and outputs. It also defines the clock signal and the required timing of the clock signal. This file will have been generated by the Synopsys synthesis process, and is **<filename>.sdc**. You can also generate this by hand since it’s just a text file, but it’s much easier to let Synopsys generate this file based on the timing constraints you specified as part of the synthesis procedure (Chapter 8).

If you have all these files you can proceed to use the place and route tool to assemble that circuit on a chip. In our case we’ll be using the **SOC Encounter** tool from Cadence. My recommendation is to make a new directory from which to run the tool. I’ll make an `IC.CAD/soc` directory, and in fact, under that I’ll usually make distinct directories for each project I’m running through the **soc** tool. In this example I’ll be using a simple counter that is synthesized from the behavioral Verilog code in Figure 10.1 so I’ll make an `IC.CAD/soc/count` directory to run this example. Inside this directory I’ll make copies or links to the **.lib** and **.lef**

```

module counter (clk, clr, load, in, count);
parameter width=8;
input clk, clr, load;
input [width-1 : 0] in;
output [width-1 : 0] count;
reg [width-1 : 0] tmp;

always @(posedge clk or negedge clr)
begin
    if (!clr)
        tmp = 0;
    else if (load)
        tmp = in;
    else
        tmp = tmp + 1;
end
assign count = tmp;
endmodule

```

Figure 10.1: Simple counter behavioral Verilog code

files I'll be using. In this case I'll use **example.lib** and **example.lef** from the small library example from Chapters 7 and 9. The structural Verilog file (**count_struct.v**) generated from Synopsys (Chapter 8) is shown in Figure 10.2, and the timing constraints file, **count_struct.sdc** is shown in Figure 10.3. This is generated from the synthesis process and encodes the timing constraints used in synthesis. Once I have all these files in place I can begin.

10.1 Encounter GUI

As a first tutorial example of using the **SOC Encounter** tool, I'll describe how to use the tool from the GUI. Most things that you do in the GUI can also be done in a script, but I think it's important to use the tool interactively so that you know what the different steps are. Also, even if you script the optimization phases of the process, it's probably vital that you do the floor planning by hand in the GUI for complex designs before you set the tool loose on the optimization phases.

First make sure that you have all the files you need in the directory you will use to run **SOC Encounter**. I'm using the counter from the previous Figures so I have:

count_struct.v: The structural file generated by Synopsys

count_struct.sdc: The timing constraints file generated by Synopsys

```

module counter ( clk, clr, load, in, count );
    input [7:0] in;
    output [7:0] count;
    input clk, clr, load;
    wire  n39, n40, n41, n42, n43, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14,
        N15, N16, N17, N18, N19, N20, n2, n3, n4, n5, n6, n7, n8, n9, n10,
        n11, n12, n13, n14, n15, n16, n17, n18, n19, n20, n21, n22, n23, n24,
        n25, n26, n27, n28, n30, n31, n32, n33, n34, n36, n37;

    DFF tmp_reg_0_ ( .D(N12), .G(clk), .CLR(clr), .Q(n43) );
    DFF tmp_reg_1_ ( .D(N13), .G(clk), .CLR(clr), .Q(n42) );
    DFF tmp_reg_2_ ( .D(N14), .G(clk), .CLR(clr), .Q(n41) );
    DFF tmp_reg_3_ ( .D(N15), .G(clk), .CLR(clr), .Q(n40) );
    DFF tmp_reg_4_ ( .D(N16), .G(clk), .CLR(clr), .Q(n39) );
    DFF tmp_reg_5_ ( .D(N17), .G(clk), .CLR(clr), .Q(count[5]) );
    DFF tmp_reg_6_ ( .D(N18), .G(clk), .CLR(clr), .Q(count[6]) );
    DFF tmp_reg_7_ ( .D(N19), .G(clk), .CLR(clr), .Q(count[7]) );
    MUX2_INV U13 ( .A(N20), .B(in[5]), .S(load), .Y(n4) );
    MUX2_INV U14 ( .A(N9), .B(in[4]), .S(load), .Y(n5) );
    MUX2_INV U15 ( .A(N8), .B(in[3]), .S(load), .Y(n6) );
    MUX2_INV U16 ( .A(N7), .B(in[2]), .S(load), .Y(n7) );
    MUX2_INV U17 ( .A(N6), .B(in[1]), .S(load), .Y(n8) );
    MUX2_INV U18 ( .A(N5), .B(in[0]), .S(load), .Y(n9) );
    INVX1 U5 ( .A(n4), .Y(N17) );
    INVX1 U9 ( .A(n8), .Y(N13) );
    INVX4 U19 ( .A(n27), .Y(n15) );
    NAND2 U20 ( .A(n23), .B(n20), .Y(n10) );
    INVX1 U21 ( .A(n10), .Y(n21) );
    INVX1 U22 ( .A(n30), .Y(n12) );
    MUX2_INV U23 ( .A(count[0]), .B(N5), .S(n11), .Y(N6) );
    XOR2 U24 ( .A(n25), .B(count[2]), .Y(N7) );
    NAND2 U25 ( .A(count[2]), .B(n25), .Y(n14) );
    MUX2_INV U26 ( .A(n40), .B(n13), .S(n14), .Y(N8) );
    XOR2 U27 ( .A(count[4]), .B(n15), .Y(N9) );
    MUX2_INV U28 ( .A(count[5]), .B(n16), .S(n28), .Y(N20) );
    NOR2 U29 ( .A(n27), .B(n32), .Y(n17) );
    XOR2 U30 ( .A(count[6]), .B(n17), .Y(N10) );
    INVX1 U31 ( .A(count[7]), .Y(n18) );
    INVX1 U32 ( .A(count[6]), .Y(n19) );
    NOR2 U33 ( .A(n19), .B(n16), .Y(n20) );
    NOR2 U34 ( .A(n31), .B(n22), .Y(n23) );
    NAND2 U35 ( .A(n12), .B(count[2]), .Y(n22) );
    INVX1 U36 ( .A(n26), .Y(n24) );
    NAND2 U37 ( .A(n40), .B(count[2]), .Y(n26) );
    NAND2 U38 ( .A(n12), .B(n24), .Y(n27) );
    NOR2 U39 ( .A(N5), .B(n11), .Y(n25) );
    NAND2 U40 ( .A(count[4]), .B(n15), .Y(n28) );
    MUX2_INV U41 ( .A(n18), .B(count[7]), .S(n21), .Y(N11) );
    INVX1 U42 ( .A(N5), .Y(count[0]) );
    INVX1 U43 ( .A(n42), .Y(n11) );
    NAND2 U44 ( .A(n43), .B(n42), .Y(n30) );
    INVX1 U45 ( .A(n40), .Y(n13) );
    NAND2 U46 ( .A(count[4]), .B(n40), .Y(n31) );
    INVX1 U47 ( .A(n33), .Y(n32) );
    NOR2 U48 ( .A(n37), .B(n16), .Y(n33) );
    INVX1 U49 ( .A(n43), .Y(N5) );
    INVX1 U50 ( .A(count[5]), .Y(n16) );
    INVX1 U51 ( .A(n41), .Y(n34) );
    INVX4 U52 ( .A(n34), .Y(count[2]) );
    INVX1 U53 ( .A(n11), .Y(count[1]) );
    INVX1 U54 ( .A(n13), .Y(count[3]) );
    INVX1 U55 ( .A(n39), .Y(n37) );
    INVX1 U56 ( .A(load), .Y(n36) );
    INVX1 U57 ( .A(n7), .Y(N14) );
    MUX2_INV U58 ( .A(in[7]), .B(N11), .S(n36), .Y(n2) );
    INVX4 U59 ( .A(n37), .Y(count[4]) );
    INVX1 U60 ( .A(n5), .Y(N16) );
    INVX1 U61 ( .A(n6), .Y(N15) );
    MUX2_INV U62 ( .A(in[6]), .B(N10), .S(n36), .Y(n3) );
    INVX1 U63 ( .A(n9), .Y(N12) );
    INVX1 U64 ( .A(n3), .Y(N18) );
    INVX1 U65 ( .A(n2), .Y(N19) );
endmodule

```

Figure 10.2: Simple counter structural Verilog code using the **example.lib** cell library

```
#####

# Created by write_sdc on Sun Oct  8 17:14:10 2006

#####
set sdc_version 1.6

set_driving_cell -lib_cell INVX4 -library example [get_ports clr]
set_driving_cell -lib_cell INVX4 -library example [get_ports load]
set_driving_cell -lib_cell INVX4 -library example [get_ports {in[7]}]
set_driving_cell -lib_cell INVX4 -library example [get_ports {in[6]}]
set_driving_cell -lib_cell INVX4 -library example [get_ports {in[5]}]
set_driving_cell -lib_cell INVX4 -library example [get_ports {in[4]}]
set_driving_cell -lib_cell INVX4 -library example [get_ports {in[3]}]
set_driving_cell -lib_cell INVX4 -library example [get_ports {in[2]}]
set_driving_cell -lib_cell INVX4 -library example [get_ports {in[1]}]
set_driving_cell -lib_cell INVX4 -library example [get_ports {in[0]}]
set_load -pin_load 0.0659802 [get_ports {count[7]}]
set_load -pin_load 0.0659802 [get_ports {count[6]}]
set_load -pin_load 0.0659802 [get_ports {count[5]}]
set_load -pin_load 0.0659802 [get_ports {count[4]}]
set_load -pin_load 0.0659802 [get_ports {count[3]}]
set_load -pin_load 0.0659802 [get_ports {count[2]}]
set_load -pin_load 0.0659802 [get_ports {count[1]}]
set_load -pin_load 0.0659802 [get_ports {count[0]}]
create_clock [get_ports clk] -period 3 -waveform {0 1.5}
set_input_delay -clock clk 0.25 [get_ports clr]
set_input_delay -clock clk 0.25 [get_ports load]
set_input_delay -clock clk 0.25 [get_ports {in[7]}]
set_input_delay -clock clk 0.25 [get_ports {in[6]}]
set_input_delay -clock clk 0.25 [get_ports {in[5]}]
set_input_delay -clock clk 0.25 [get_ports {in[4]}]
set_input_delay -clock clk 0.25 [get_ports {in[3]}]
set_input_delay -clock clk 0.25 [get_ports {in[2]}]
set_input_delay -clock clk 0.25 [get_ports {in[1]}]
set_input_delay -clock clk 0.25 [get_ports {in[0]}]
set_output_delay -clock clk 0.25 [get_ports {count[7]}]
set_output_delay -clock clk 0.25 [get_ports {count[6]}]
set_output_delay -clock clk 0.25 [get_ports {count[5]}]
set_output_delay -clock clk 0.25 [get_ports {count[4]}]
set_output_delay -clock clk 0.25 [get_ports {count[3]}]
set_output_delay -clock clk 0.25 [get_ports {count[2]}]
set_output_delay -clock clk 0.25 [get_ports {count[1]}]
set_output_delay -clock clk 0.25 [get_ports {count[0]}]
```

Figure 10.3: Timing information (.sdc file) for the counter example

example.lib: A link to my cell library's characterized data in **.lib** format. Make sure this file has footprint information for all cells.

example.lef: A link to my cell library's abstract data in **.lef** form. Make sure that you have correctly appended the **TechHeader.lef** information in front of the **MACRO** definitions.

After connecting to your directory (I'm using **IC_CAD/soc/counter**) you can start the **SOC Encounter** tool using the `cad-soc` script. You'll see the main **encounter** window as seen in Figure 10.4. This Figure is annotated to describe the different areas of the screen. The palette on the right lets you choose what is currently visible in the design display area. The **Design Views** change how you see that design. From left to right the **Design Views** are:

Floorplan View: This view shows the overall floorplan of your chip. It lets you see the area that is generated for the standard cells, and how the different pieces of your design hierarchy fit into that standard cell area. For this first example there is no hierarchy in the design so the entire counter will be placed inside the cell area. For a more complex design you can manually place the different pieces of the design in the cell area if you wish.

Amoeba View: This view shows information related to the **Amoeba** placement and routing of the cells. It gives feedback on cell placement, density, and congestion.

Physical View: This view shows the actual cells as they are placed, and the actual wires as they are routed by the tool.

All three views are useful, but I generally start out with the **floorplan** view during, as you might guess, floorplanning, then toggle between the that view and the **physical** view once the place and route gets under way.

10.1.1 Reading in the Design

Once the tool is started you need to read all your design files into the tool. Select the **Design** → **Design Import...** menu choice to get the **Design Import** dialog box. This box has multiple fields in multiple tabs that you need to fill in. First fill in the **Basic** fields with the following (see Figure 10.5):

Verilog Netlist: Your structural Verilog file or files. You can either let **SOC Encounter** pick the top cell, or you can provide the name of the top level module.

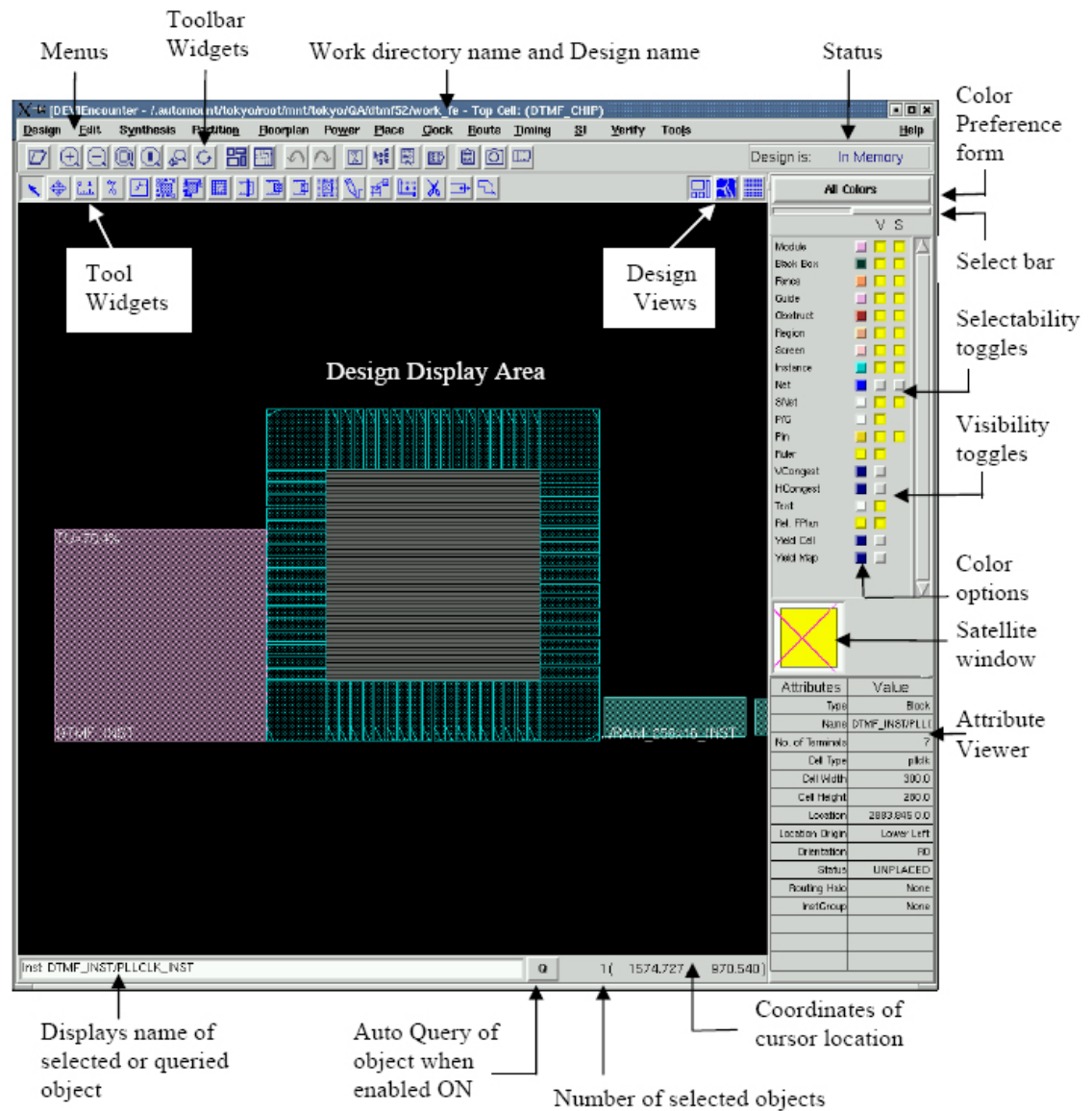


Figure 10.4: Main SOC Encounter gui

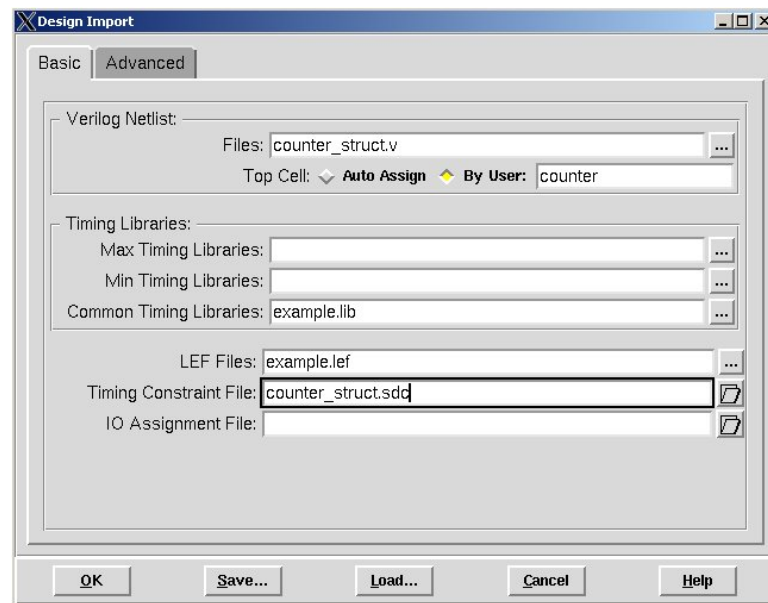


Figure 10.5: **Design Import** dialog box - basic tab

Timing Libraries: Your **.lib** file or files. If you have only one file it should be entered into the **Common Timing Libraries** line. If you have **best, typ, worst** timing libraries, they should be entered into the other fields with the **worst case** entered into the **max** field, the **best case** into the **min** field, and the **typical case** in the **common** field. This is optional and the process works just fine with only one library in the **common** field.

LEF Files: Enter your **.lef** file or files.

Timing Constraint File: Enter your **.sdc** file.

Now, move to the **Advanced** tab and make the following entries:

IPO/CTS: This tab provides information for the **IPO** (In Place Optimization) and **CTS** (Clock Tree Synthesis) procedures by letting **SOC Encounter** know which buffer and inverter cells it can use when optimizing things. Enter the name of the footprints for buffer, delay, inverter, and CTS cells. Leave any blank that you don't have. I'm entering **inv** as the footprint for delay, inverter, and CTS, and leaving buffer blank as shown in Figure 10.6. Your library may be different.

Power: Enter the names of your power and ground nets. If you're following the class design requirements this will be **vdd!** and **gnd!** (Figure 10.7).

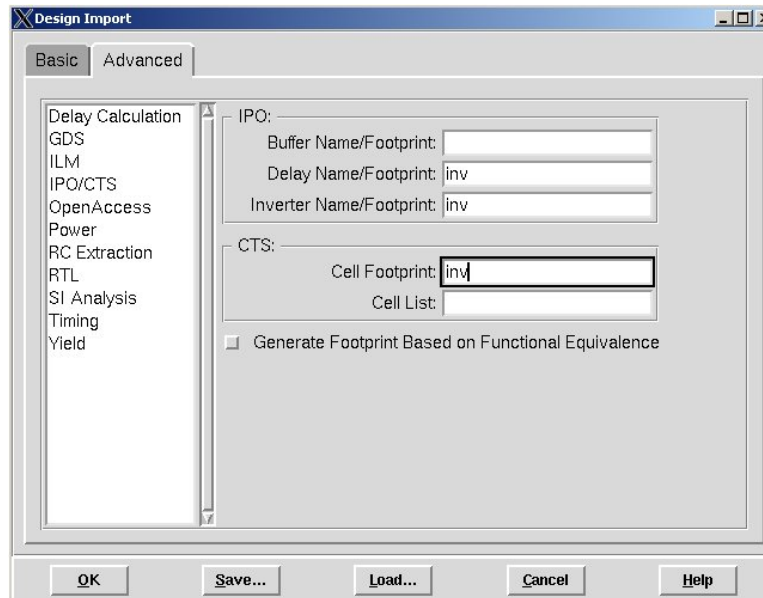


Figure 10.6: Design Import IPO/CTS tab

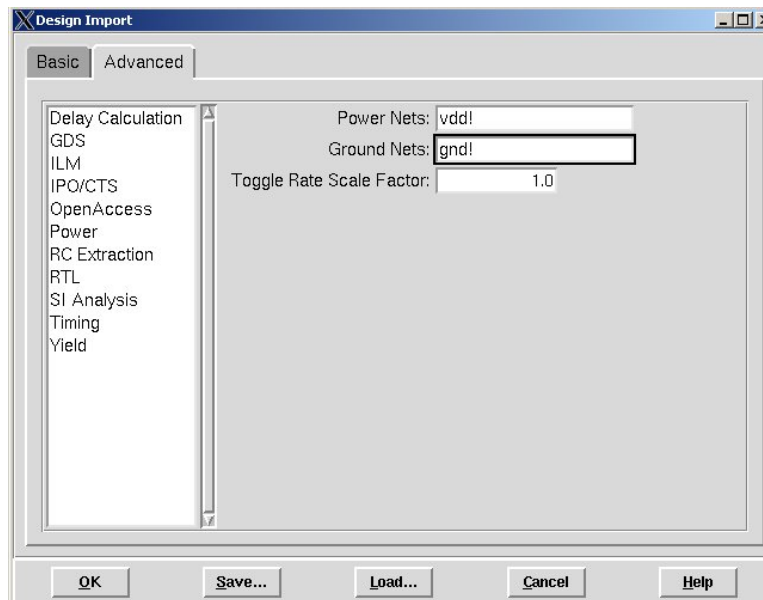


Figure 10.7: Design Import Power tab

Now you can press **OK** and read all this information into **SOC Encounter**. The comments (and potential warnings and errors) will show up in the shell window in which you invoked `cad-soc`. You should look at them carefully to make sure that things have imported correctly. If they did you will see the **SOC Encounter** window has been updated to show a set of rows in which standard cells will be placed.

10.1.2 Floorplanning

Floorplanning is the step where you make decisions about how densely packed the standard cells will be, and how the large pieces of your design will be placed relative to each other. Because there is only one top-level module in the **counter** example, this is automatically assumed to cover the entire standard cell area. If your design had more structure in terms of hierarchical modules, those modules would be placed to the side of the cell placement area so that you could place them as desired inside the cell area. The default is just to let the entire top-level design fill the standard cell area without further structuring. In practice this spreads out the entire design across the entire area which, for large systems with significant structure, may result in lower performance. For a system with significant structure a careful placement of the major blocks can have a dramatic impact on system performance.

But, for this example, what we really care about is cell density and other area parameters related to the design. Select **Floorplan** → **Specify Floorplan...** to get the floorplanning dialog box (Figure 10.8). In this dialog box you can change various parameters related to the floorplan:

Aspect Ratio: This sets the (rectangular) shape of the cell. An aspect of close to 1 is close to square. An aspect of .5 is a rectangle with the vertical edge half as long as the horizontal, and 1.5 is a rectangle with the vertical edge twice the horizontal. This is handy if you're trying to make a subsystem to fit in a larger project. For now, just for fun, I'll change the **aspect ratio** to **0.5**. Note that the tool will adjust this number a little based on the anticipated cell sizes.

Core Utilization: This lets the tool know how densely packed the core should be with standard cells. The default is around 70% which leaves room for **in place optimization** and **clock tree synthesis**, both of which may add extra cells during their operation. For a large complex design you may even have to reduce the utilization percentage below this.

*All measurements are
assumed to be in
microns.*

Core Margins: These should be set by **Core to IO Boundary** and are

to leave room for the power and ground rings that will be generated around your cell. All the **Core to ...** values should be set to **30**. Note that even though you specify **30**, when you **apply** those values they may change slightly according to **SOC Encounter's** measurements.

Others: Other spots in the **Specify Floorplan** dialog can be left as default. In particular you want the standard cell rows to be **Double-back Rows**, and you can leave the **Row Spacing** as zero to leave no space between the rows. If your design proves hard to route you can start again and leave extra space between the rows for routing.

After adjusting the floorplan, the main **SOC Encounter** window looks like Figure 10.9. The rows in which cells will be placed are in the center with the little corner diagonals showing how the cells in those rows will be flipped. The dotted line is the outer dimension of the final cell. The power and ground rings will go in the space between the cells and the outer boundary.

Saving the Design

This is a good spot in which to save the current design. There are lots of steps in the process that are not “undo-able.” It’s nice to save the design at various points so that if you want to try something different you can reload the design and try other things. Save the design with **Design → Save Design...** and name the saved file **<filename>.enc**. In my case I’ll name it **floorplan.enc** so that I can restore to the point where I have a floorplan if I want to start over from this point. Saved designs are restored into the tool using the **Design → Restore Design...** menu.

I like to save the design at each major step so that I can go back if I need to try something different at that step. Be aware that there’s no general “undo” function in Encounter.

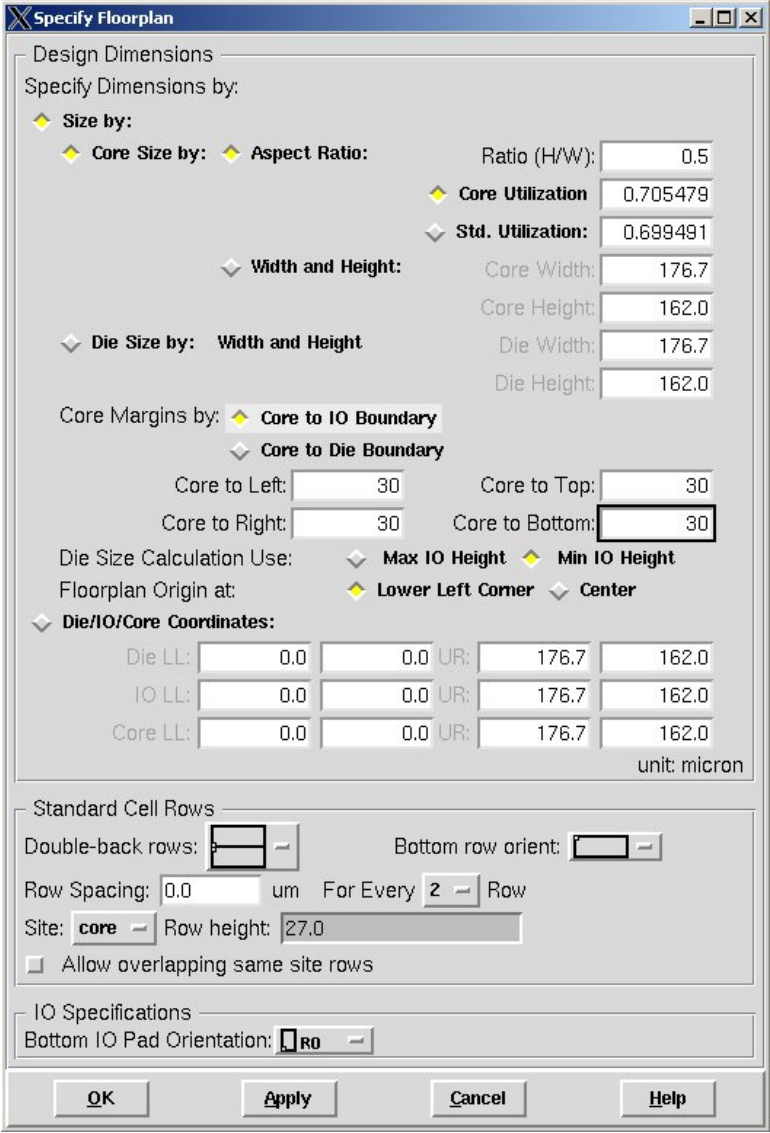
10.1.3 Power Planning

Now it’s time to put the power and ground ring around your circuit, and connect that ring to the rows so that your cells will be connected to power and ground when they’re placed in the row. Start with **Power → Power Planning → Add Rings**. From this dialog box (Figure 10.10) you can control how the power rings are generated. The fields of interest are:

Ring Type: The defaults are good here. You should have the **Core ring(s) contouring:** set to **Around core boundary**.

Ring Configuration: You can select the metal layers you want to use for the ring, their width, and their spacing. I’m making the top and bottom of the ring horizontal **metal1**, and the right and left vertical

Remember that all the sizes and spacings you specify must be divisible by the basic lambda unit of our underlying technology. That is, everything is measured in units of 0.3 microns, so values should be divisible by 0.3.



Specify Floorplan

Design Dimensions

Specify Dimensions by:

Size by:

Core Size by: Aspect Ratio: Ratio (H/W): 0.5

Core Utilization: 0.705479

Std. Utilization: 0.699491

Width and Height: Core Width: 176.7

Core Height: 162.0

Die Size by: Width and Height

Die Width: 176.7

Die Height: 162.0

Core Margins by: Core to IO Boundary

Core to Die Boundary

Core to Left: 30 Core to Top: 30

Core to Right: 30 Core to Bottom: 30

Die Size Calculation Use: Max IO Height Min IO Height

Floorplan Origin at: Lower Left Corner Center

Die/IO/Core Coordinates:

Die LL: 0.0 0.0 UR: 176.7 162.0

IO LL: 0.0 0.0 UR: 176.7 162.0

Core LL: 0.0 0.0 UR: 176.7 162.0

unit: micron

Standard Cell Rows

Double-back rows: Bottom row orient:

Row Spacing: 0.0 um For Every 2 Row

Site: core Row height: 27.0

Allow overlapping same site rows

IO Specifications

Bottom IO Pad Orientation: RO

OK Apply Cancel Help

Figure 10.8: The **Specify Floorplan** dialog box

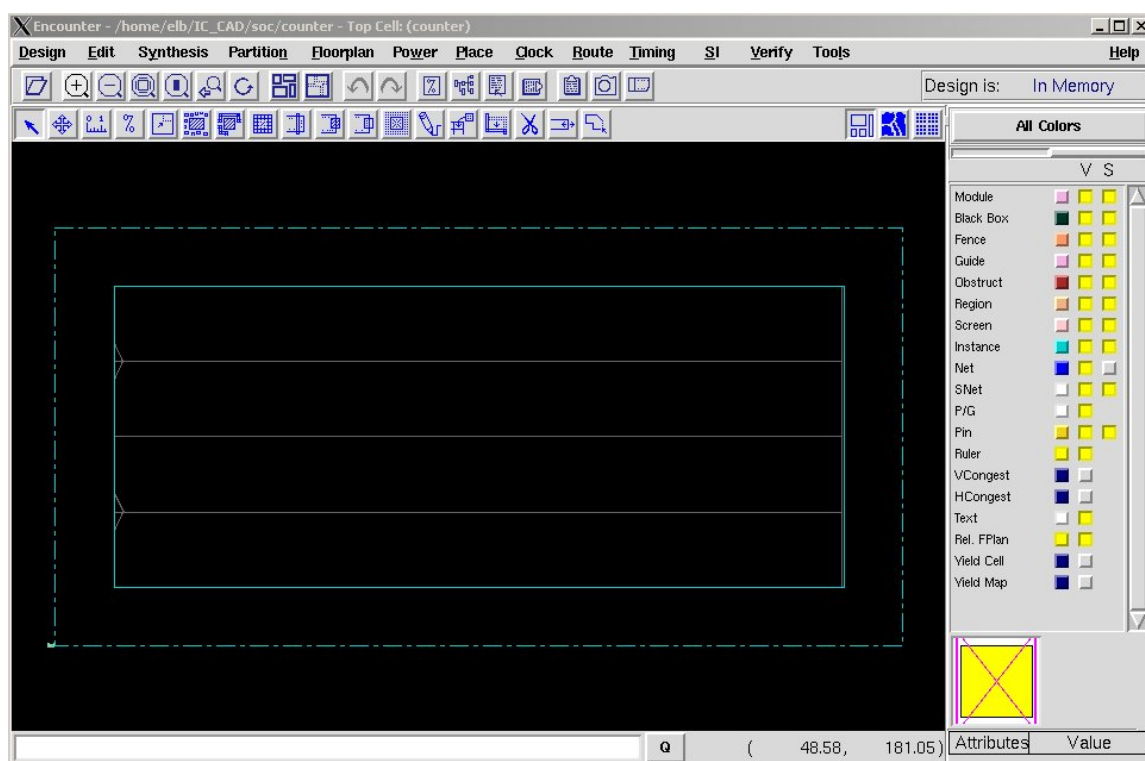


Figure 10.9: Main design window after floorplanning

metal2 to match our routing protocol. Change the **width** of each side of the ring to **9.9** and the spacing should be set to **1.8** because of the extra spacing required for wide metal. Finally, the offset can be left alone or changed to **center in channel**. If it's left alone it should probably be changed to **1.8** to match the wide metal spacing.

When you click **OK** you will see the power and ground rings generated around your cell. You can also zoom in and see that the tool has generated arrays of vias where the wide horizontal and vertical wires meet.

Now, for this simple small design, this would be enough, but for a larger design you would want to add *power stripes* in addition to the *power rings*. Stripes are additional vertical power and ground connections that turn the power routing into more of a mesh. Add stripes using the **Power** → **Power Planning** → **Add Stripes...** menu (Figure 10.11). The fields of interest are:

Set Configuration: Make sure that all your power and ground signals are in the **Net(s)** field (**vdd!** and **gnd!** in our case). Choose the layer you want to the stripes to use. In our case the stripes are vertical so it makes sense to have them in the normal vertical routing layer of **metal2**. Change the width and spacing as desired (I'm choosing **4.8** for the width and **1.8** for the spacing - remember that they need to be multiples of 0.3).

Set Pattern: This section determines how much distance there is between the sets of stripes, how many different sets there are, and other things. You can leave the **Set-to-set distance** to the default of **100**.

Stripe Boundary: Unless you're doing something different, leave the default to have the stripes generated for your **Core ring**.

First/Last Stripe: Choose how far from the left (or right) you want your first stripe. I'm using **75** from the left in this example so that the stripes are roughly spaced equally in the cell. Note that this is probably overkill from a power distribution point of view. For a larger cell 250 micron spacing might be a more reasonable choice.

Advanced Tab - Snap Wire to Routing Grid: Change this from **None** to **Grid**. The issue here is that our cells are designed so that if two cells are placed right next to each other, no geometry in one cell will cause a design rule violation in the other cell. That's the reason that no cell geometry (other than the well) is allowed within 0.6μ from the **prBoundary**. That way layers, such as metal layers, are at least 1.2μ from each other when cells are abutted. However, the power stripes don't have that restriction and if you don't center the power stripes on the grid, a cell could be placed right next to a power grid and cause a

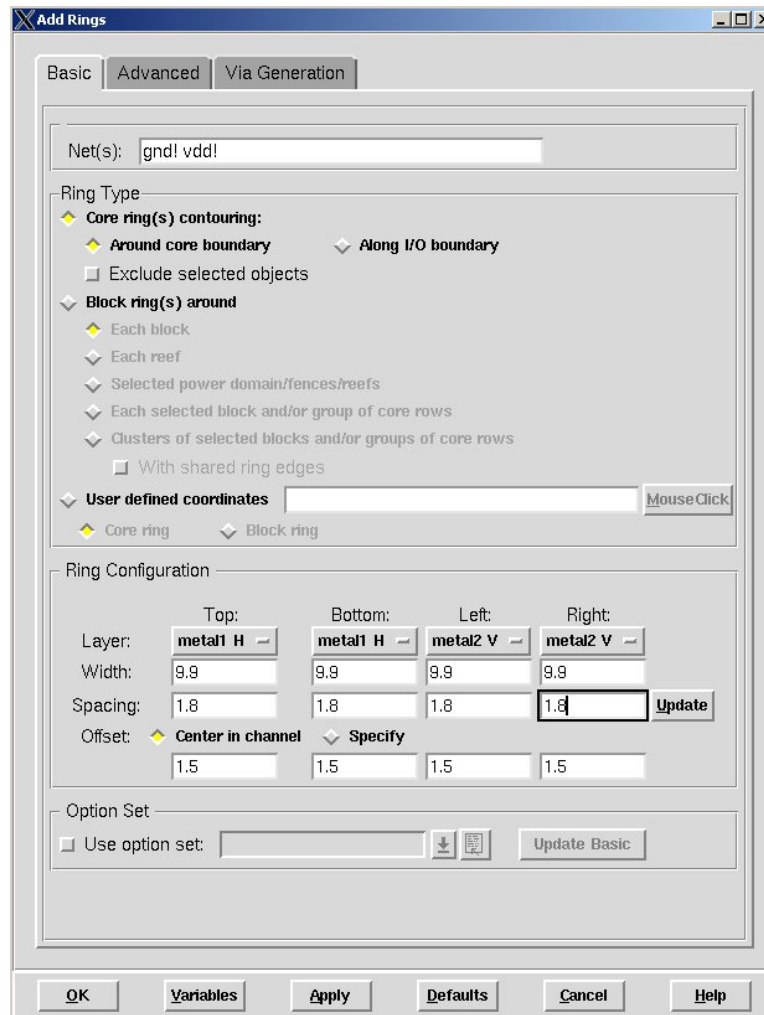


Figure 10.10: Dialog box for adding power and ground rings around your cell

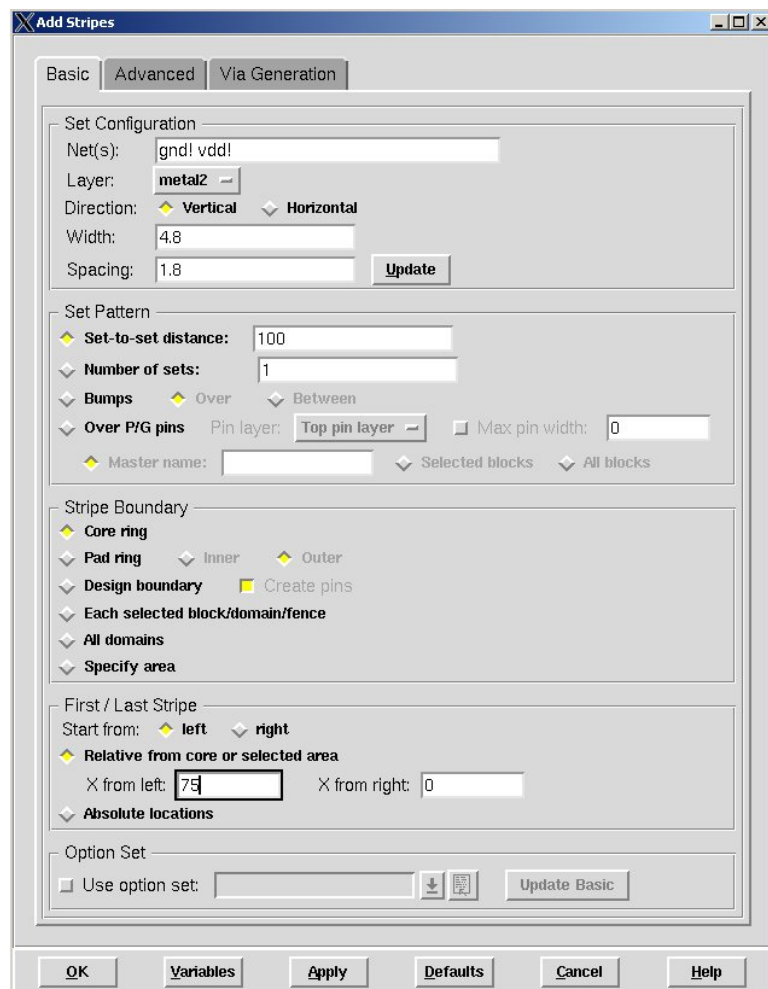


Figure 10.11: Dialog box for planning power stripes

metal spacing DRC violation when the metal of the stripe is now only 0.6μ from the metal in the cell. Centering the stripe on a grid keeps this from happening by placing the metal of the stripe in a position so that the next legal cell position isn't directly abutting with the power stripe.

Clicking **Apply** will apply the stripes to your design. If you don't like the looks of them you can select and delete them and try again with different parameters. Or you can select **OK** and be finished.

*Now is another good time to save the cell again. This time I'll save it as **powerplan.enc**.*

Once you have the stripes placed you can connect power to the rows where the cells will be placed. Select **Route** → **Special Route** to route the power wires. Make sure that all your power supplies are listed in the **Net(s)**

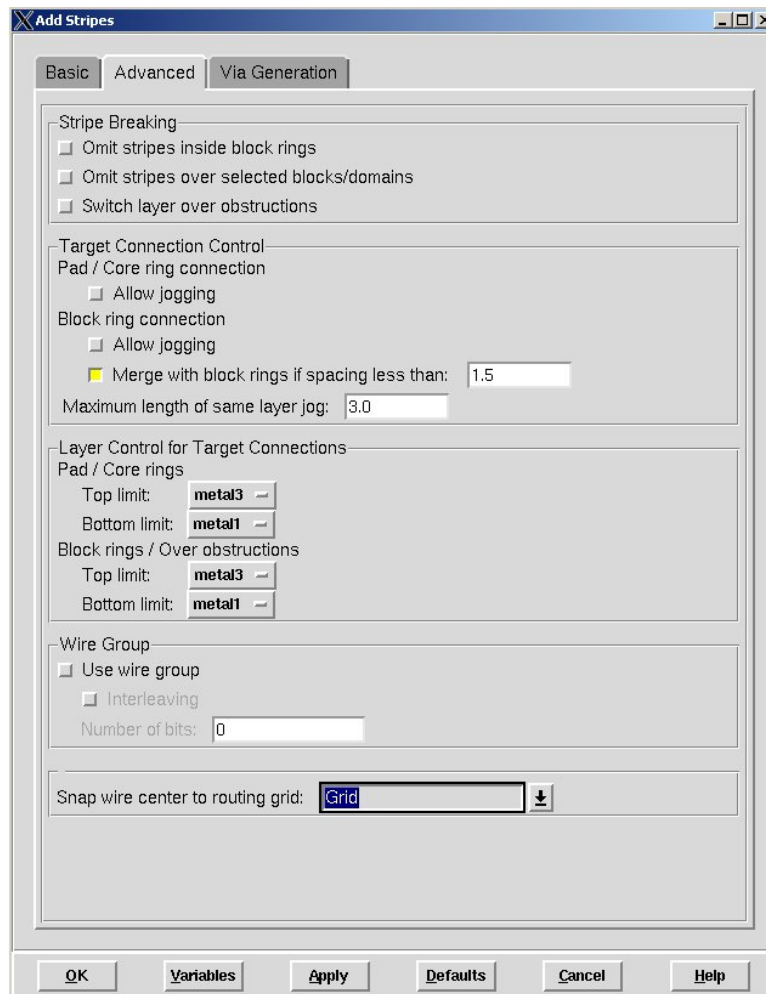


Figure 10.12: Advanced Tab of Dialog box for planning power stripes

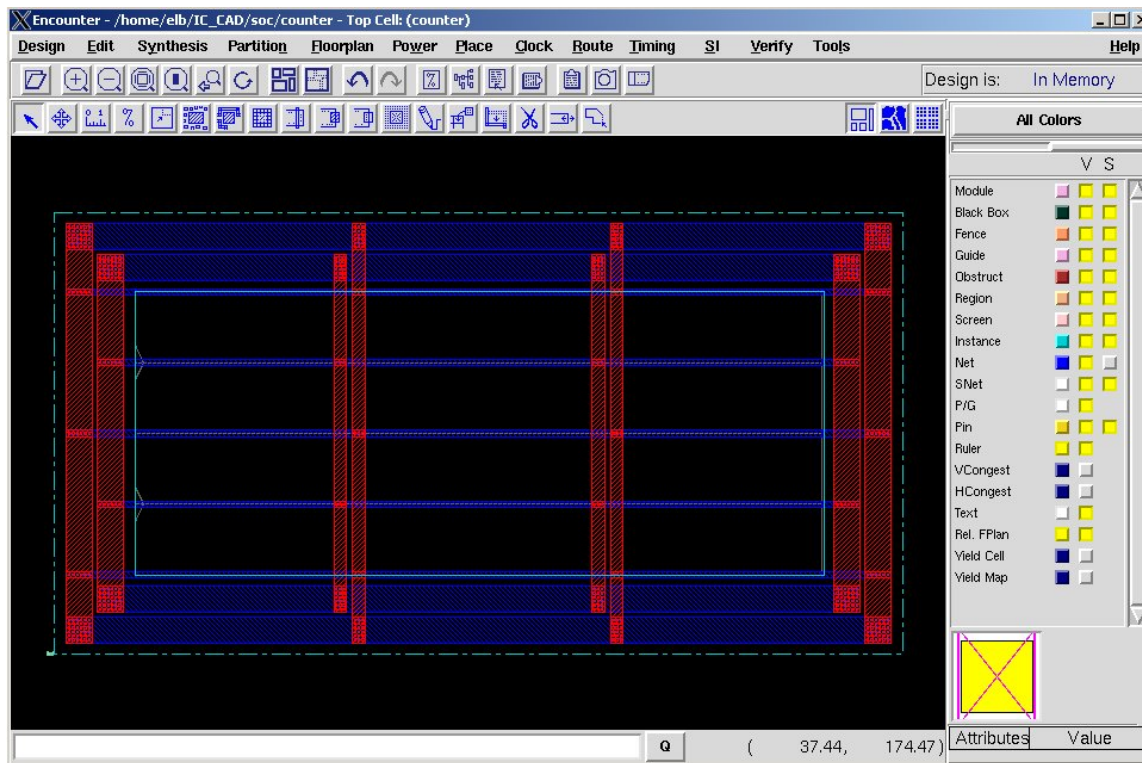


Figure 10.13: Floorplan after power rings and stripes have been generated and connected to the cell rows

field (**vdd!** and **gnd!** in our case). Other fields don't need to be changed unless you have specific reasons to change them. Click **OK** and you will see the rows have their power connections made to the ring/stripe grid as seen in Figure reffig:soc-pp4. Zoom in to any of the connections and you'll see that an array of vias has been generated to fill the area of the connection.

10.1.4 Placing the Standard Cells

Now you want the tool to place the standard cells in your design into that floorplan. Select **Place** → **Standard Cells and Blocks...** Leave the defaults in the dialog box as seen in Figure 10.14. You definitely want to use **timing driven placement** and **pre-place optimization**.

After pressing **OK** your cells will be placed in the rows of the floorplan. This might take a while for a large design. When it's finished the screen won't look any different. Change to the **physical view** (the rightmost design view widget - see Figure 10.15) and you'll see where each of your cells has been placed. The placed counter looks like that in Figure 10.16.

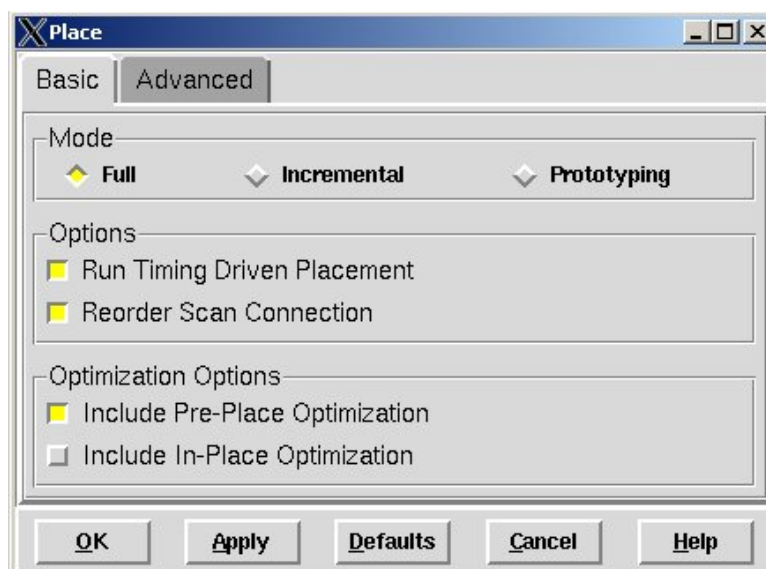


Figure 10.14: Placement dialog box

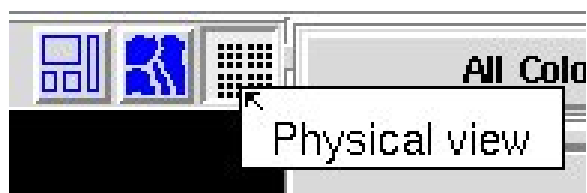


Figure 10.15: Widget for changing to the physical view of the cell

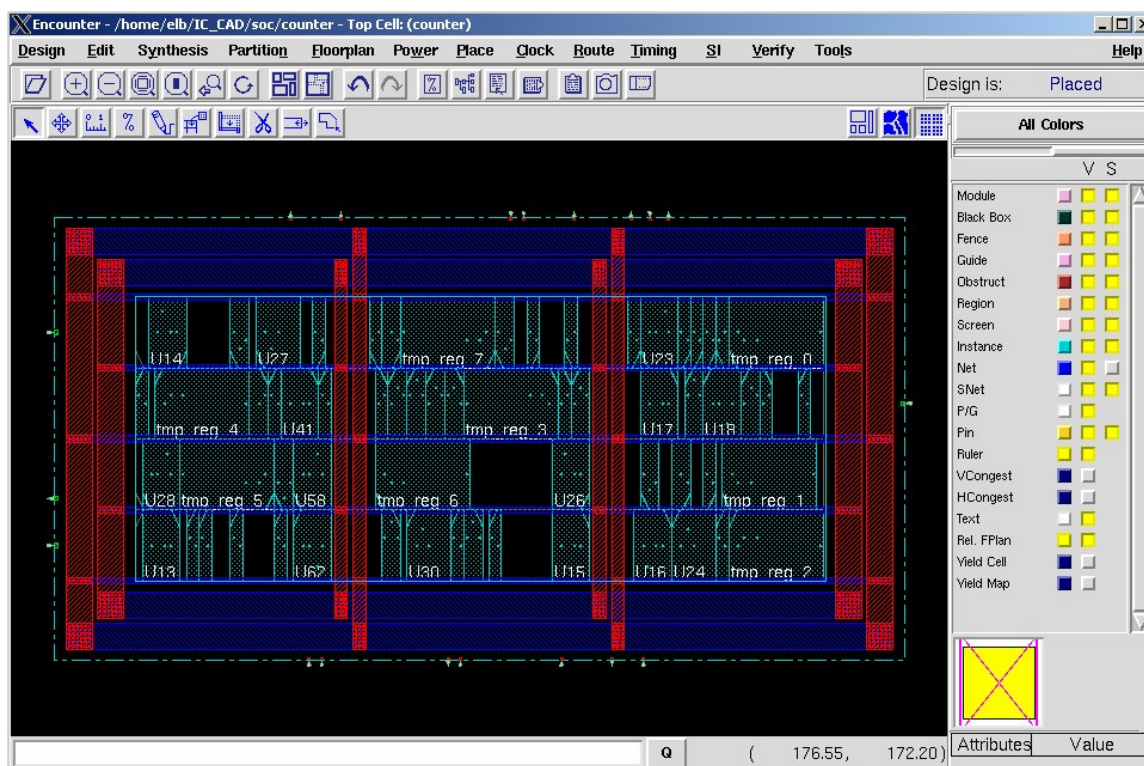


Figure 10.16: View after placement of the cells

If your design had more than one floorplan elements you could go back to the floorplan view and select one of the elements. Then moving to the physical view you would see where all the cells from that element had ended up. This is an interesting way of seeing how the placement has partitioned things.

10.1.5 First Optimization Phase

Now you can perform the first timing and optimization step. At this stage of the process there are no wires so the timing step will do a *trial route* to estimate the wiring. This is a low-effort not-necessarily-correct routing of the circuit just for estimation. Select **Timing** → **Optimization**. Notice that under **Design Stage** you should select **pre-CTS** to indicate that this analysis is before any clock tree has been generated (Figure 10.17). You're also doing analysis only on **setup** time of the circuit. Click **OK** and you'll see the result of the timing optimization and analysis in the shell window. If you refresh the screen you'll also see that it looks like the circuit has been routed! But, this is just a **trial route**. These wires will be replaced with real routing wires later.

In this case the timing report (shown in Figure 10.18) shows that we are not meeting timing. In particular, there are 7 violating paths and the **worst negative slack** is **-1.757ns**. The timing was specified in the **.sdc** file and came from the timing requirements at synthesis time. In this case the desired timing is an (overly aggressive) 3ns clock period just to demonstrate how things work.

Note that you can always re-run the timing analysis after trying things by selecting **Timing** → **Timing Analysis** from the menu. Make sure to select the correct **Design Stage** to reflect where you are in the design process. At this point, for example, I am in **Pre-CTS** stage.

10.1.6 Clock Tree Synthesis

Now we can synthesize a clock tree which will (hopefully) help our timing situation. In a large design this will have a huge impact on the timing. In this small example design it will be less dramatic. Select **Clock** → **Create Clock Tree Spec** to start. You should fill in the footprint information that the CTS process can use to construct a clock tree. This should have been filled in with the information from our original design import but it's not for some reason. I'm filling in **inv** as the inverter footprint and another for buffers because my library doesn't have non-inverting buffers (see Figure 10.19). Your library may be different. Clicking **OK** will generate a clock tree specification in the (default) **counter.ctstch** file.

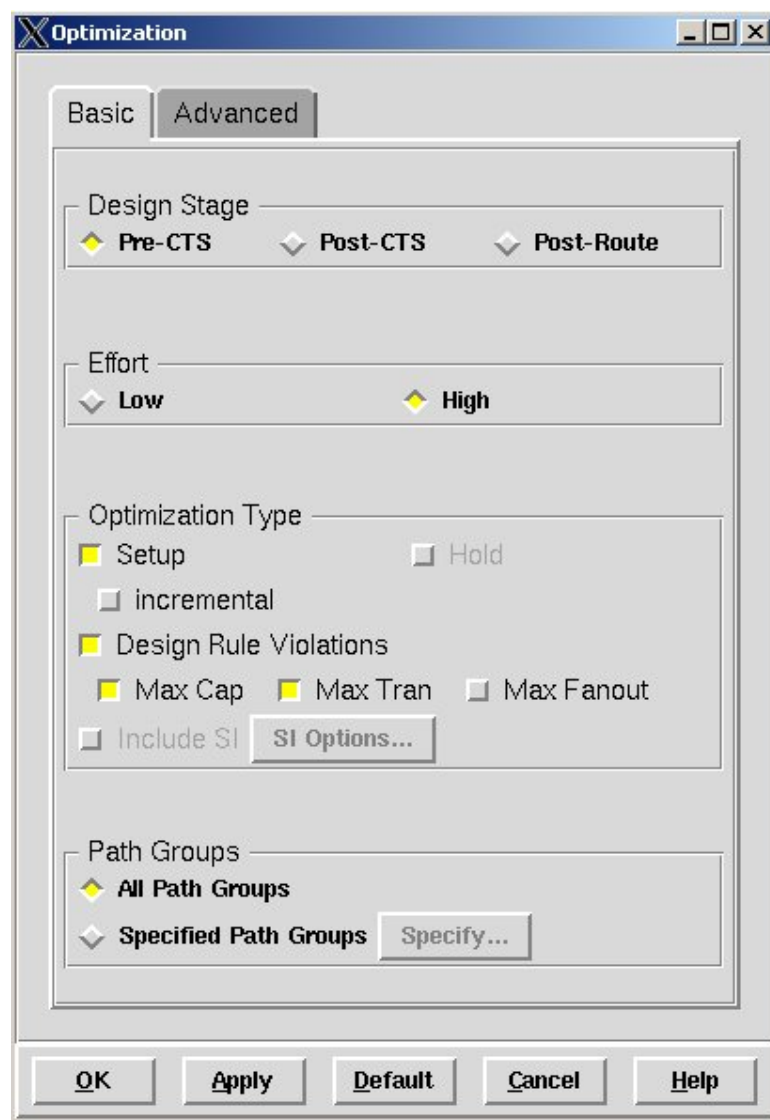


Figure 10.17: Dialog box for timing optimization

timeDesign Summary						
Setup mode	all	reg2reg	in2reg	reg2out	in2out	clkgate
WNS (ns):	-1.757	-1.757	0.700	0.891	N/A	N/A
TNS (ns):	-7.589	-7.589	0.000	0.000	N/A	N/A
Violating Paths:	6	6	0	0	N/A	N/A
All Paths:	24	8	16	8	N/A	N/A

Figure 10.18: Initial pre-CTS timing results

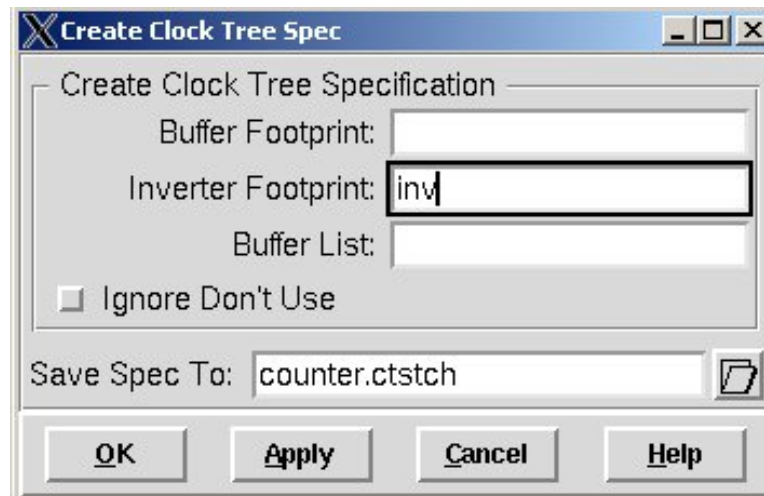


Figure 10.19: Generating a clock tree specification

Now you need to specify the clock tree based on this spec (yes, this seems a little redundant...). Select **Clock** → **Specify Clock Tree** and use the clock tree specification file (**counter.ctstch**) that you just generated.

Now you can actually generate the clock tree with **Clock** → **Synthesize Clock Tree**. You can leave the defaults in this dialog box and just click **OK** to generate the clock tree. This will use the cells you told it about to generate a clock tree in your circuit. If you watched carefully you would have seen some cells added and other cells moved around during this process.

If you'd like to see the clock tree that was generated you can select **Clock** → **Display** → **Display Clock Tree** to see the tree. This will annotate the display to show the tree. You should select **Clock Route Only** in the dialog box (Figure 10.20). If you select **Display Clock Phase Delay** you'll see the clock tree (in blue) and the flip flops connected to that tree colored by their differing phase delays. If you select **Display Min/Max Paths** you'll see the min and max path from the clock pin to the flip flop displayed. See Figure 10.21 for an example. You can clear the clock tree display with **Clock** → **Display** → **Clear Clock Tree Display**.

10.1.7 Post-CTS Optimization

After you have generated the clock tree you can do another phase of timing optimization. Again select **Timing** → **Optimization** but this time select **post-CTS** (refer back to Figure 10.17). This optimization phase shows that the addition of the clock tree and the subsequent optimization helped a little, but not much. In a larger design this would have a much more dramatic impact. See Figure 10.22.

10.1.8 Final Routing

Now that the design has a clock tree you can perform final routing of the design. Select **Route** → **NanoRoute** → **Route** to invoke the router. There are lots of controls, but **Timing Driven** is probably the only one you need to change from the default (see Figure 10.23). Of course, you are welcome to play around with these controls to see what happens. Once you select **Timing Driven** you can also adjust the **Effort** slider to tell the tool how much effort to spend on meeting timing and how much on reducing congestion. On a large, aggressive design you may need to try things are various settings to get something you like. Because this is not un-doable, you should save the circuit in a state just before routing so that you can restore to that state before trying different routing options.

This can take a long time on a large design. Check the shell window

*For a large complex design you may need to go all the way back to the floorplanning stage and add more space between rows for routing channels. Because this technology has only three routing layers, and you have likely used a lot of **metal1** in your cells, the chance for routing congestion in large designs is high.*

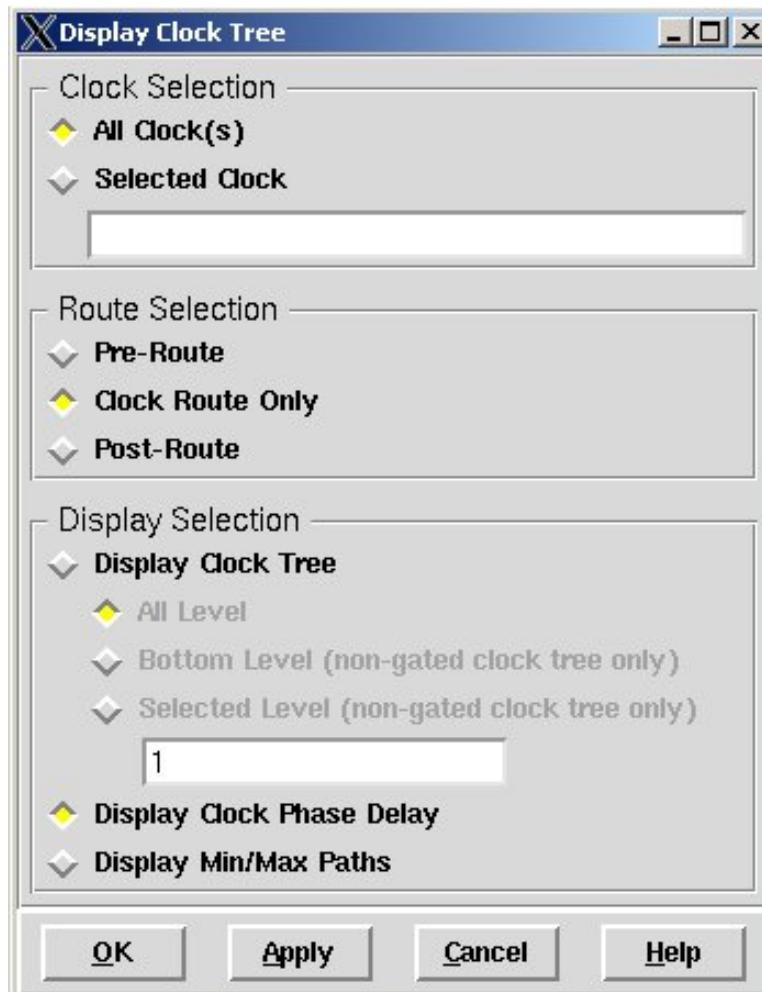


Figure 10.20: Dialog box to display the clock tree

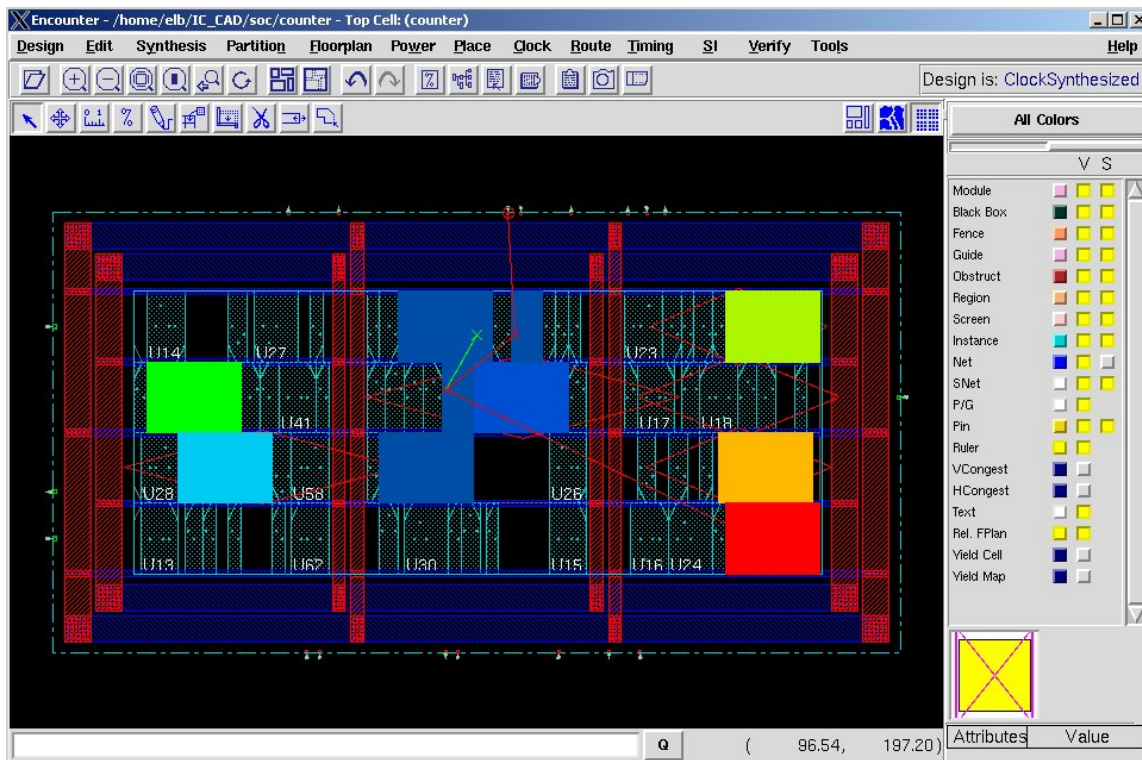


Figure 10.21: Design display showing the clock tree

optDesign Final Summary							
Setup mode	all	reg2reg	in2reg	reg2out	in2out	clkgate	
WNS (ns):	-1.649	-1.649	1.061	0.442	N/A	N/A	
TNS (ns):	-7.542	-7.542	0.000	0.000	N/A	N/A	
Violating Paths:	6	6	0	0	N/A	N/A	
All Paths:	24	8	16	8	N/A	N/A	

Figure 10.22: Timing results after the second (**Post-CTS**) optimization

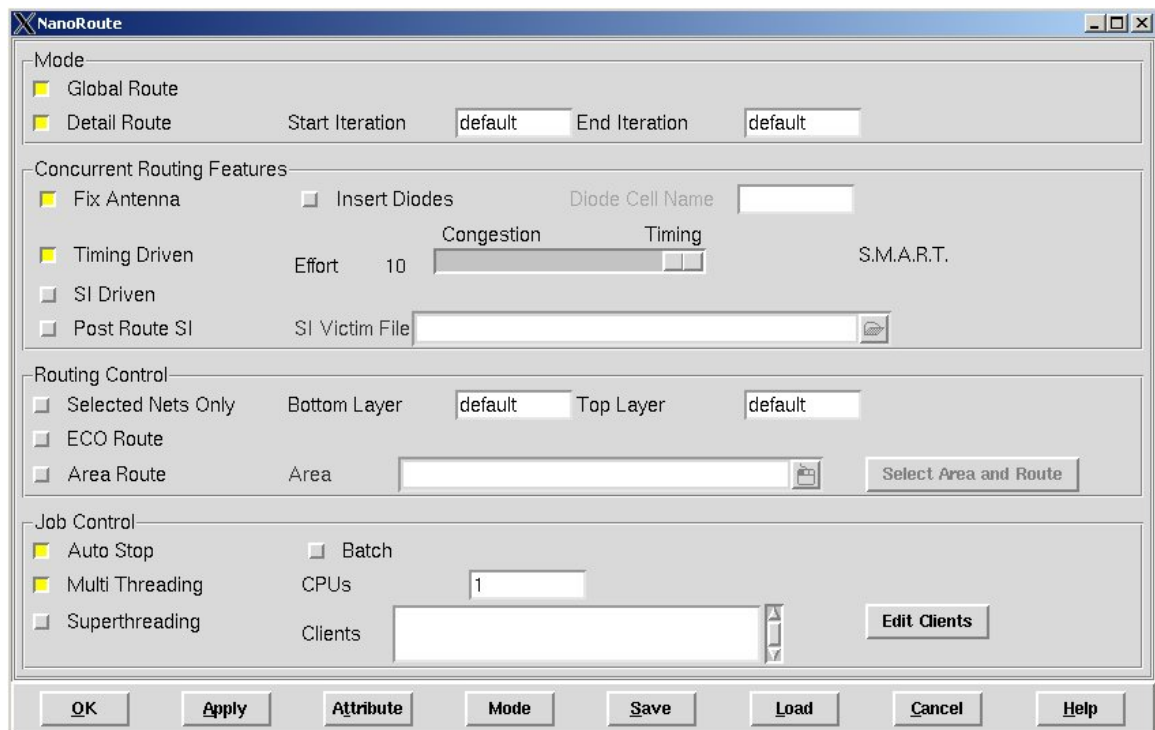


Figure 10.23: **NanoRoute** dialog box

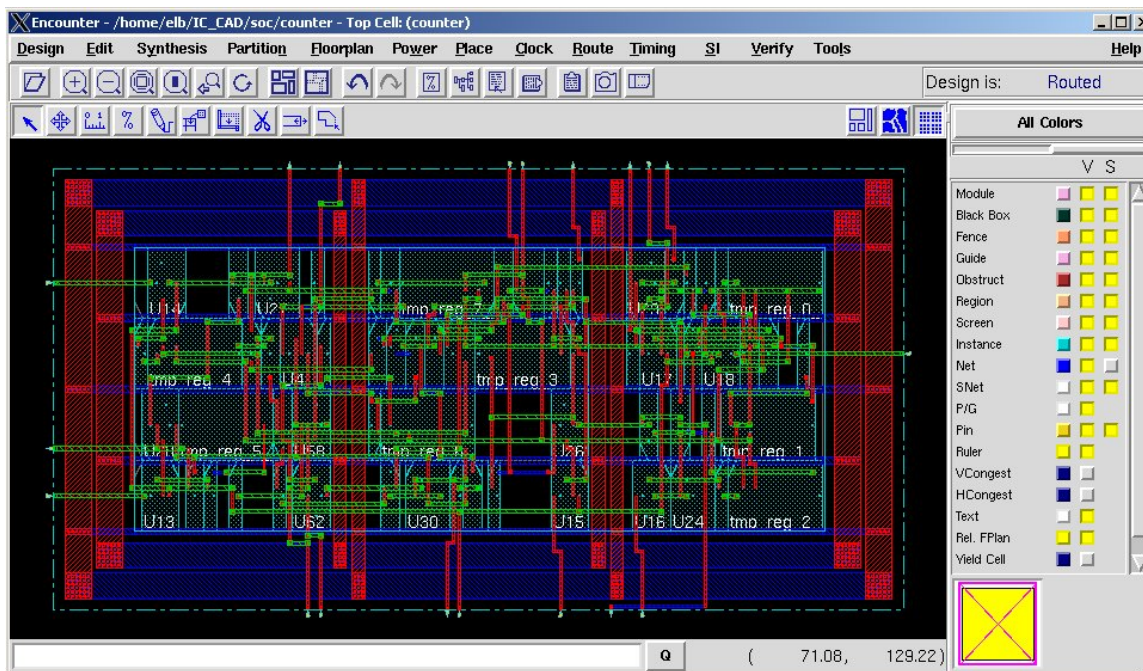


Figure 10.24: The counter circuit after using **NanoRoute** to do final routing

for updates on how it is progressing. When it finishes you should see that there are 0 **violations** and 0 **fails**. If there are routing failures left then you will either have to go in and fix them by hand or start over from a whole new floorplan! An example of a completely routed circuit is shown in Figure 10.24.

It's possible that you might have an error associated with a pin after routing. It seems to happen every once in a while, and I don't know exactly what the cause is. You can fix it by starting over and routing again to see if you still have the error, or you can zoom in to the error and move things around to fix the error. I generally just move the wires a little bit and they snap to the grid and everything's fine. If you want to do this you can zoom into the correct part of the layout using the right mouse button. Once you're there you can use the **Tool Widgets** in the upper left of the screen (in blue) to move, add, etc. the wires on the screen. The **Edit** → **Edit Route** and **Edit** → **Edit Pin** tools can help with this. Usually you don't need to mess with this though. Or, you can wait to fix things back in **Virtuoso** after you've finished with **SOC Encounter**.

Figure 10.25 shows one of these errors. In this case the **metal2** of the pin is too close to the **metal2** of the power stripe. I'll fix it by moving the whole pin over to the left as seen in Figure 10.26.

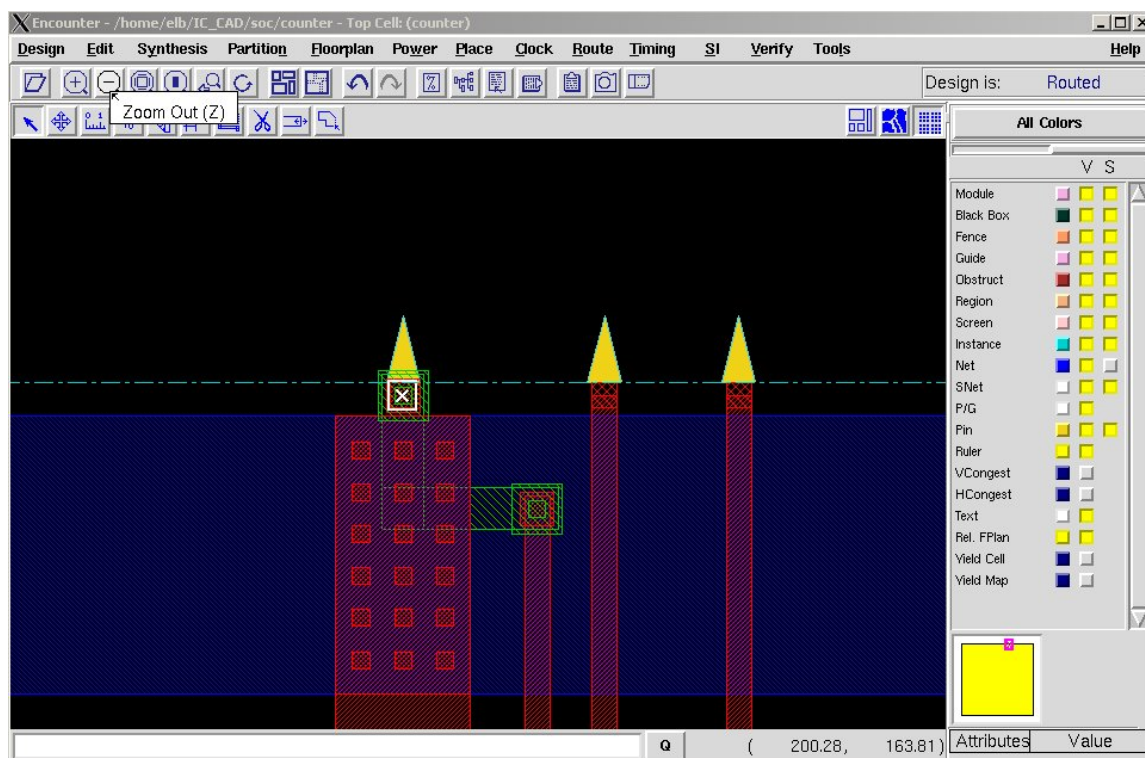


Figure 10.25: An error in pin placement after final routing

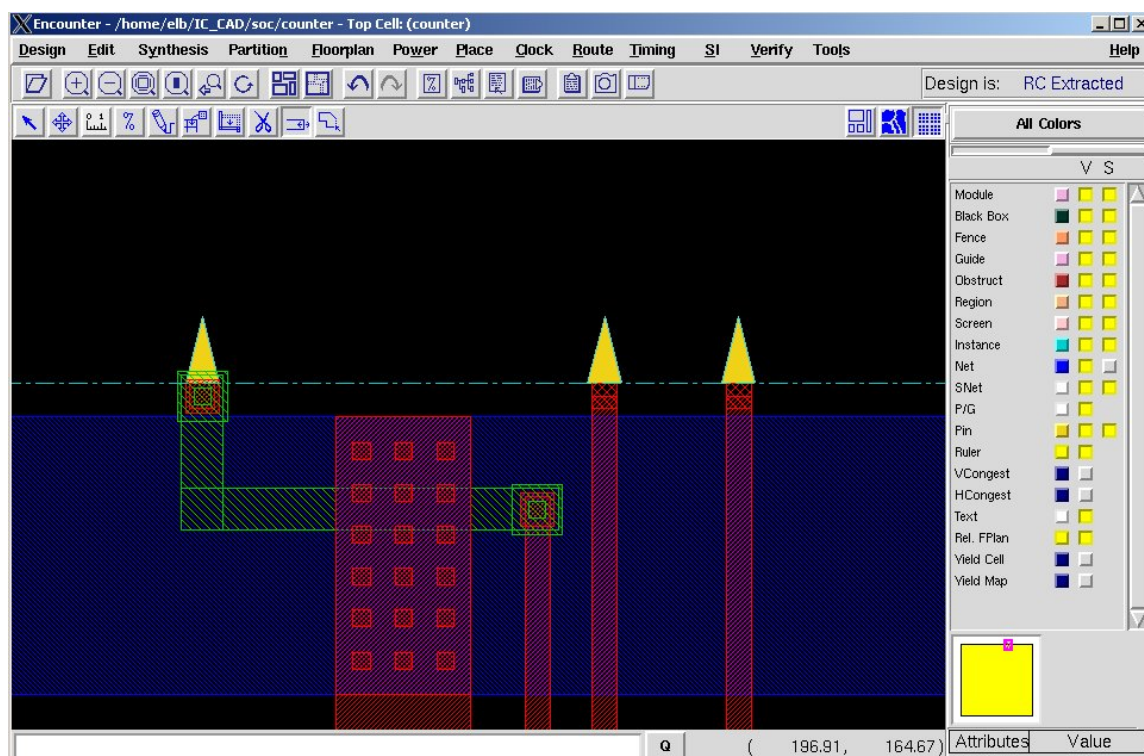


Figure 10.26: Design display after fixing the pin routing error

----- optDesign Final Summary -----							
Setup mode	all	reg2reg	in2reg	reg2out	in2out	clkgate	
WNS (ns):	-1.524	-1.524	1.051	0.363	N/A	N/A	
TNS (ns):	-8.050	-8.050	0.000	0.000	N/A	N/A	
Violating Paths:	7	7	0	0	N/A	N/A	
All Paths:	24	8	16	8	N/A	N/A	

Figure 10.27: Final post-route timing optimization results

Post-Route Optimization

You can now run one more optimization step. Like the others, you use **Timing** → **Optimization** but this time you choose **Post-Route** for the **Design Stage**. In this case the timing was improved slightly again, but still doesn't meet the (overly aggressive) timing that was specified. See Figure 10.27.

10.1.9 Adding Filler Cells

After the final optimization is complete, you need to fill the gaps in the cell row with *filler cells*. These are cells in your library that have no active circuits in them, just power and ground wires and **NWELL** layers. Select **Place** → **Filler** → **Add** and either add the cell names of your filler cells or use the **browse** button to find them. In my library I have two different filler cells: a one-wide **FILL** and a two-wide **FILL2** (see Figure 10.28). Clicking **OK** will fill all the gaps in the rows with filler cells for a final cell layout as seen in Figure 10.29. You can also zoom around in the layout to see how things look and how they're connected as in Figure 10.30.

10.1.10 Checking the Result

There are a number of checks you can run on the design in **SOC Encounter**. The first thing you should check is that all the connections specified in your original structural netlist have been made successfully. To check this use the **Verify** → **Verify Connectivity** menu choice. The defaults as seen in Figure 10.31 are fine. This should return a result in the shell that says that there are no problems or warnings related to connectivity. If there are, you need to figure out what they are and start over from an appropriate stage of the place and route process to fix them. If there are problems it's likely that routing congestion caused the problem. You could try a new route, a

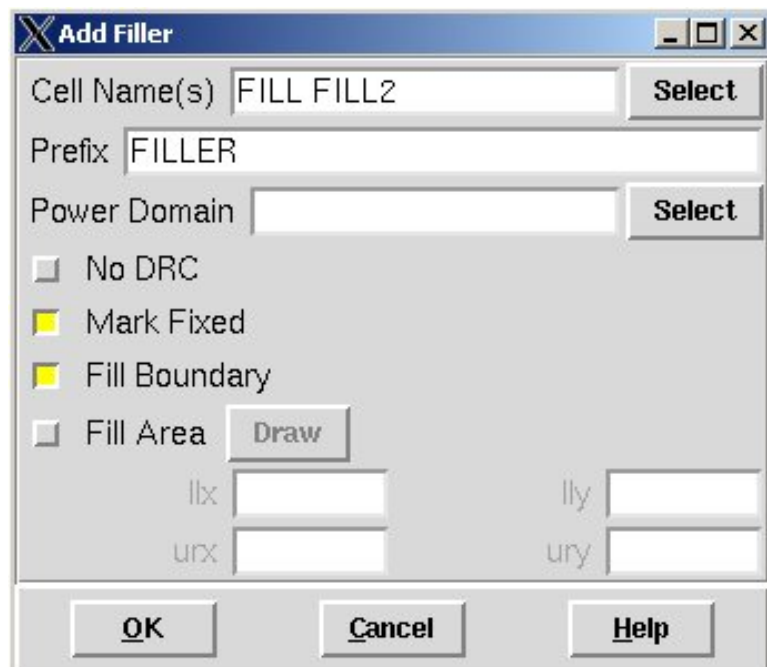


Figure 10.28: Filler cell dialog box

new placement, or go all the way back to a new floorplan with a lower core utilization percentage or more space between rows for routing channels.

Another check you can make is to run a DRC check on the routing and abstract views. To do this select **Verify** → **Verify Geometry**. You should probably un-check the **Cell Overlap** button in this dialog box as seen in Figure 10.32. I've found that with our cells and our technology information (in the LEF file) that if you leave **Cell Overlap** checked the tool flags some perfectly legal contact overlaps as errors. This is *not* a substitute for a full DRC check in Virtuoso. This check only runs on the routing and abstract views, and has only a subset of the full rules that are checked in **Virtuoso**. But, if you have errors here, you should try to correct them here before moving on to the next step. You can view the errors with the **violation browser** from the **Verify** menu. You can use this tool to find each error, get information about the error, and zoom in to the error in the design window.

10.1.11 Saving and Exporting the Placed and Routed Cell

Now that you have a completely placed, routed, optimized, and filled cell you need to save it and export it. There are a number of options depending on how you want to use the cell in your larger design.

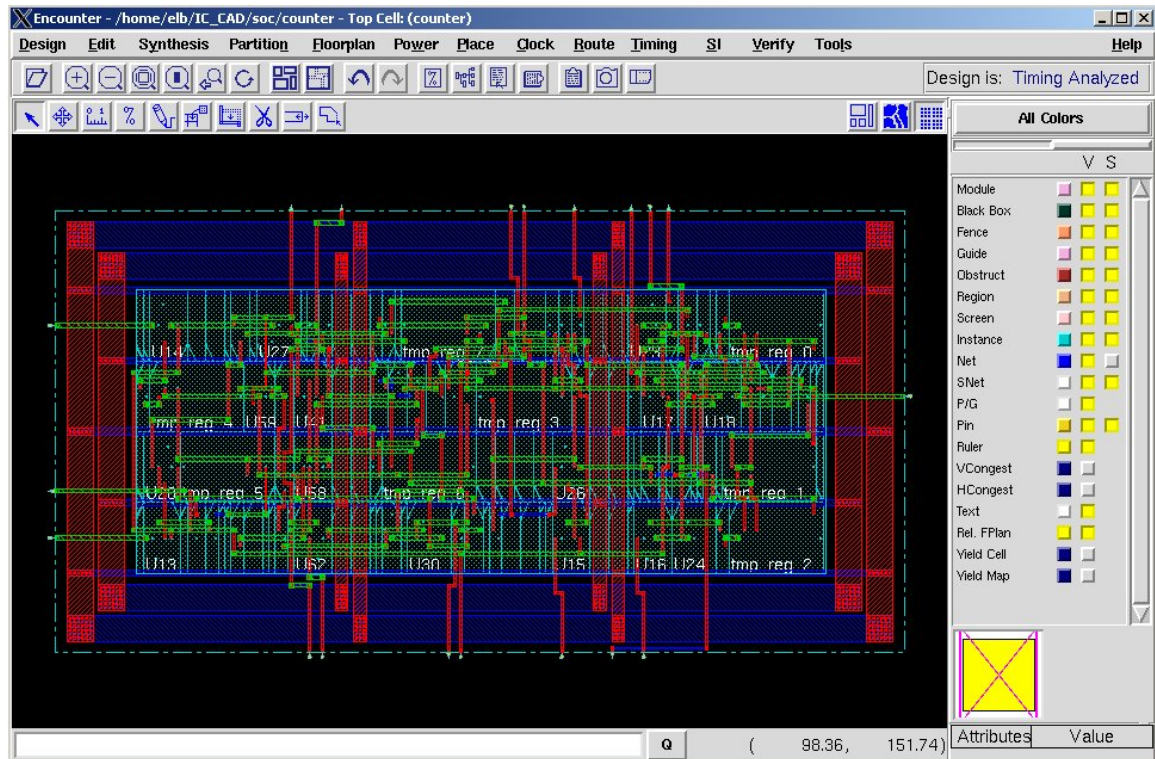


Figure 10.29: Final cell layout after filler cells are added

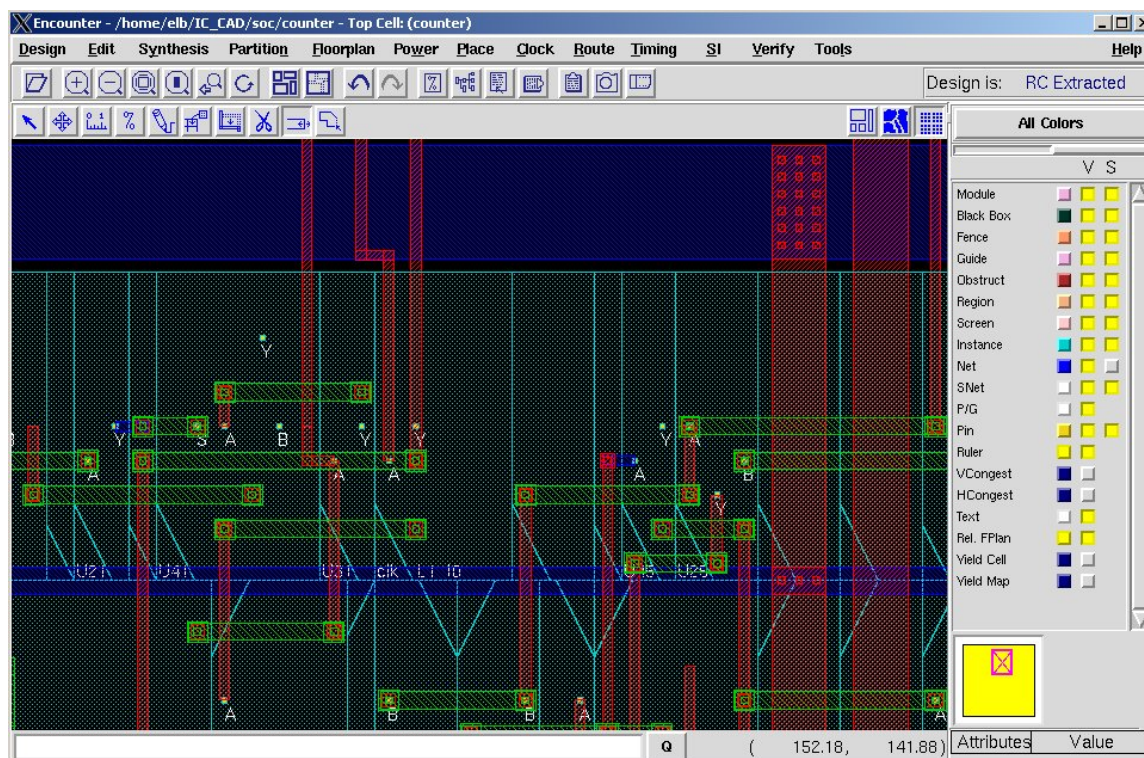


Figure 10.30: A zoomed view of a final routed cells

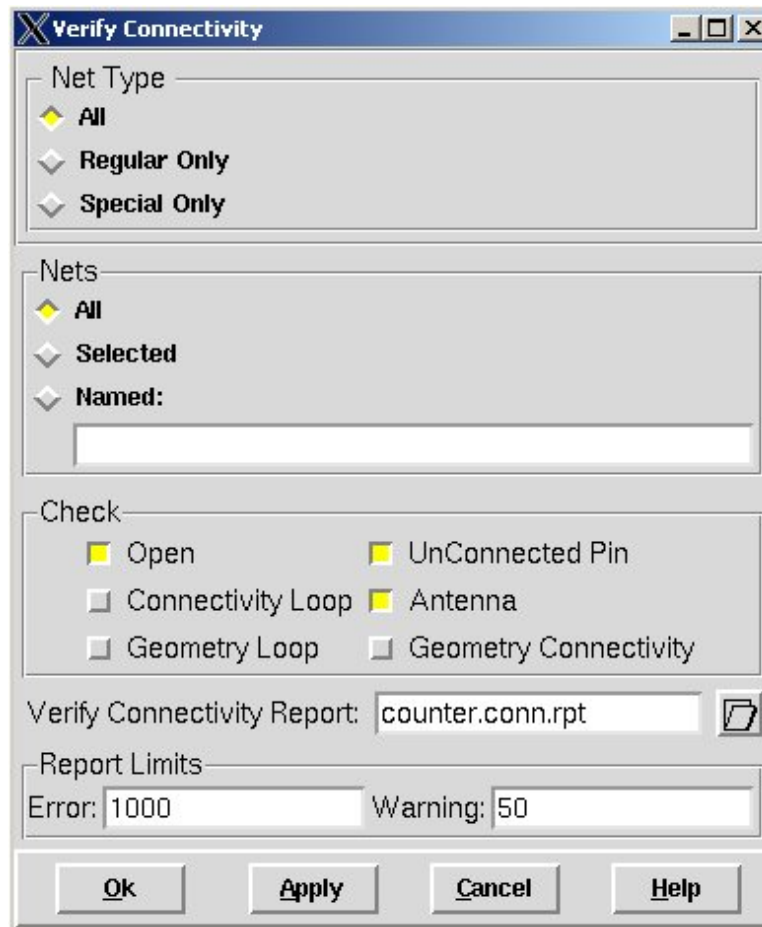


Figure 10.31: Dialog box for verifying connectivity

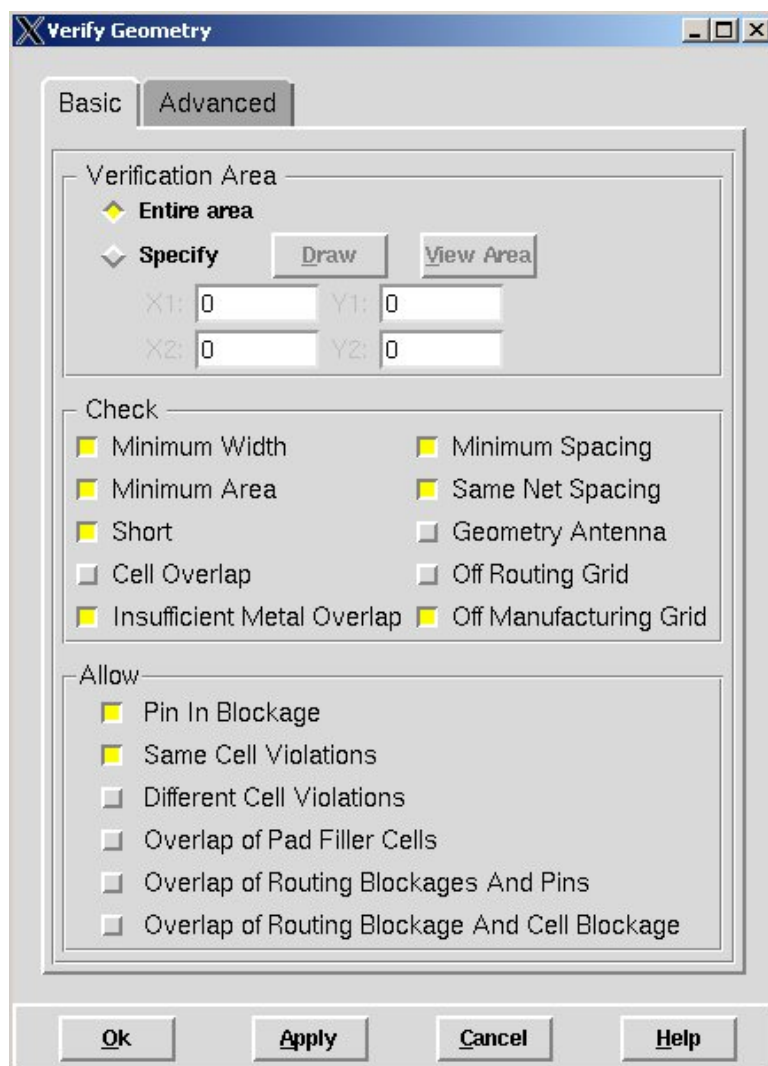


Figure 10.32: Dialog box for checking geometry

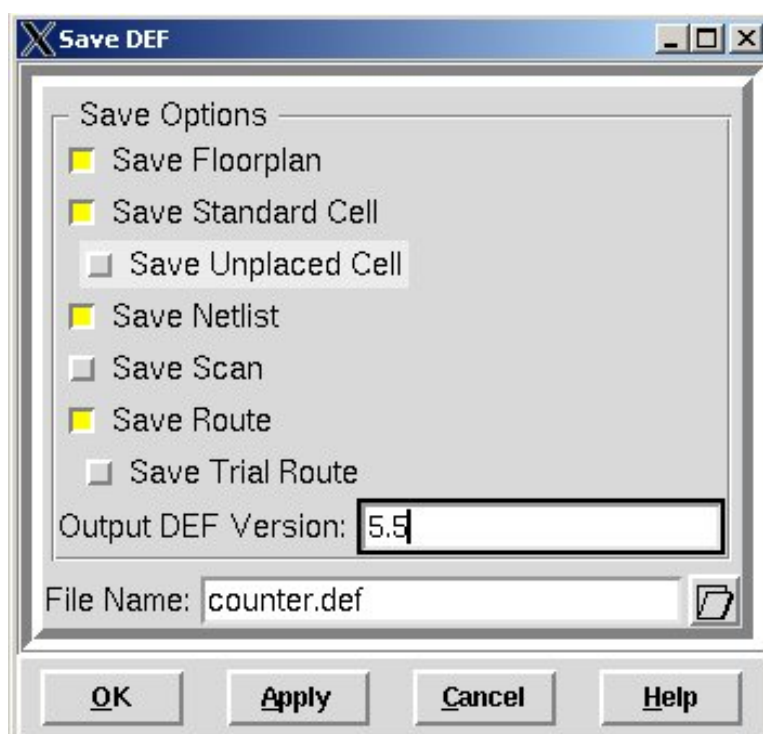


Figure 10.33: Dialog box for exporting DEF

Exporting a DEF file: In order to read the layout back in to **Virtuoso** so that you can run DRC and other tests on the data. The format that is used to pass the layout of the cell back to **Virtuoso** is called **DEF** (Design Exchange Format). You can export a **DEF** file by selecting the **Design** → **Save** → **DEF...** menu choice. In the dialog box (Figure 10.33) you can leave the default selections, but change the **Output DEF Version** from **NULL** to **5.5** so that you get the correct version of **DEF**. You can also change the file name to whatever you like.

*Without the version specification change you will get DEF version 5.6 which is not readable by our version of **Virtuoso** (IC 5.1.41)*

Exporting a structural Verilog file: Because we have had **SOC Encounter** generate a clock tree and run a number of optimization steps, the circuit that we've ended up with is *not* the same circuit that we started with. Cells may have been exchanged during optimization, and cells were added to make the clock tree. So, if you want to compare the layout to a schematic, you need the new structural file. Export this with **Design** → **Save** → **Netlist**. This will generate a new structural Verilog file that contains the cells that are in the final placed and routed circuit.

Exporting an SDF file: Now that the circuit has been placed and routed,

you can export an **SDF** (Standard Delay Format) file that has not only timings for the cells but timing extracted from the interconnect. You can use this file with your Verilog simulation if all your cells have behavioral views with **specify** blocks for timing to get quite accurate timing in your behavioral simulation. Generate this file by first selecting **Timing** → **Extract RC** to extract the RC timings of the interconnect. You can leave the default of just saving the capacitance to the **.cap** file if you just want an **SDF** file, or you can select other types of RC output files. The **SPEF** file in particular can be used by the Synopsys tool **PrimeTime** later for static timing analysis if you want. Once you've extracted the RC information you can generate an **SDF** file with the **Timing** → **Calculate Delay** menu choice.

Exporting a LEF file: If you want to use this placed and routed cell as a **block** in a larger circuit you need to export the cell as a **LEF** file and as a **liberty** file. That is, you could take this entire cell and instantiate it as a block in another larger circuit. It could be placed as a block in the floorplanning stage and other standard cells could be placed and routed around the block. The **LEF** file would describe this entire cell as one **MACRO**, and you would include that **LEF** file in the list of **LEF** files in the design import phase. The **.lib** file would define the timing of this cell. For some reason, exporting **LEF** and **Lib** information is not in the menus. To generate these files you need to type the commands directly to the **SOC Encounter** prompt in the shell window. Type the commands:

```
do_extract_model -blackbox_2d -force counter.characterize.lib  
lefOut counter.lef -stripePin -PGpinLayers 1 2
```

This will generate both **.lib** and **.lef** files that you can use later to include this cell as a block in another circuit. The options to the **lefOut** command make sure that the power and ground rings and stripes are extracted in the cell macro. If you look at the **.lef** file you'll notice that the cell macro is defined to be of **CLASS BLOCK**. This is different from the lower level standard cells which are **CLASS CORE**. This indicates that this is a large block that should be placed separately from the standard cells in **SOC Encounter**.

10.1.12 Reading the Cell Back Into Virtuoso

Now your cell is complete so you can read it back in to **Virtuoso** for further processing (DRC, extract, and LVS) or for further use as a macro cell in other designs.

Importing Layout Information

To import the cell layout, first go back to your cadence directory and start **icfb** with the `cad-ncsu` script. Then create a new library to hold this cell. Make sure that you attach the **UofU_TechLib_ami05** technology library to the new library. For this example I've made a new library called **counter**. Now that you have a new library to put it in, you can import the cell. From the **CIW** (Command Interpreter Window) select the **File** → **Import** → **DEF** menu. In the **DEF In** dialog box fill in the fields:

Library Name: This is the library (**counter** in this example) where you would like the cell to go.

Cell Name: This is the name of the top-level cell (**counter** in this case).

View Name: What view would you like the layout to go to? The best choice is probably **layout**.

Use Ref Library Names: Make sure to check this option and enter the name of the library that contains all the standard cells in your library. In this example the library is **example**, but your library will be different (probably **6710.Lib**).

DEF File Name: This is the name of the **DEF** file produced by **SOC Encounter**.

The other fields can be left at their defaults. This example is seen in Figure 10.34. When you click **OK** you'll get some warnings in the **CIW**. You can ignore warnings about not being able to open the **techfile.cds**, about being unable to find **master core** and failing to open the cell views for **viagen** cells. I don't know the exact cause for all these warnings, but they don't seem to cause problems. You should be able to see a new layout view in your new library. Zoom in to the cell and make sure that the connections between the large power rings have arrays of vias. You may need to expand the view to see the vias. If you have arrays of vias then the import has most likely worked correctly. If you don't have any vias in the connection then something has failed. You may have used the wrong version of **DEF** from **SOC Encounter** for example.

Now open the **layout** view. You'll see the circuit that consists of cells and routing. But, you need to adjust a few things before everything will work correctly.

- The first thing is that the layout has most likely opened in **Virtuoso Preview** mode. You need to get back to **Layout** mode which is the layout editor mode that you're familiar with. Select **Tools** → **Layout**

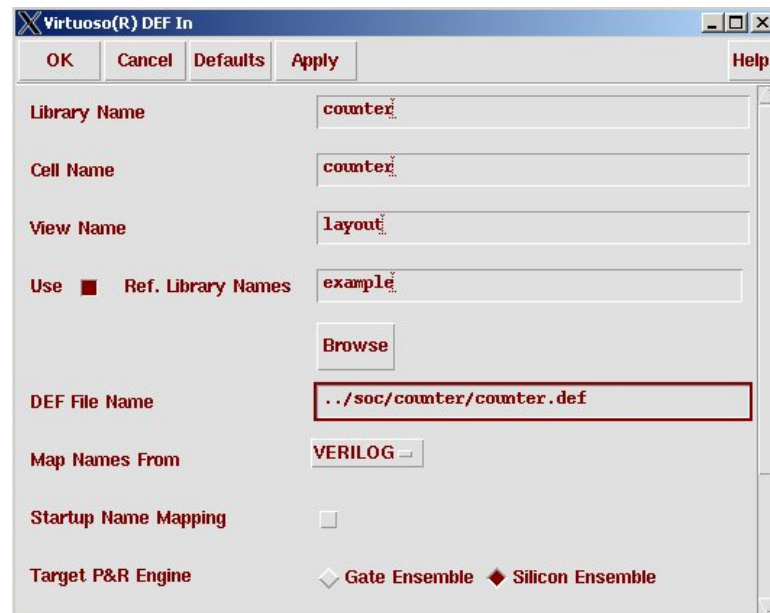


Figure 10.34: Dialog box for importing DEF files to **icfb**

to put the editor back into **Layout** mode. You'll see the familiar set of menu choices come back to the top of the window and the **LSW** Layer Selection Window will return.

- All the cells in the design are currently **abstract** views. You need to change them all to be **layout** views to be able to run the rest of the procedures. This can be done with the **Edit** menu as follows:
 1. Select **Edit** → **Search** to bring up the **search** dialog box. You will use this search box to find all the cells in the design that are currently using the **abstract** view. Click on **Add Criteria**. In the new **criteria** change the leftmost selection to **View Name**, and change the name of the view to **abstract** (see Figure 10.35). Now clicking on **Apply** will find all the cells in the design whose **View Name** is equal to **abstract**. You should see every cell in the design highlighted. The dialog box will also update to tell you how many cells it found. In this example it found 113 cells.
 2. Now you need to replace the view of those cells with **layout**. Click on the **Replace** field to change it to **View Name** and change the name to **layout** (See Figure 10.36). Now click on **Replace All** and all the cells whose view used to be **abstract** will be updated to use **layout** views.
 3. You can now cancel this dialog box and save the view.

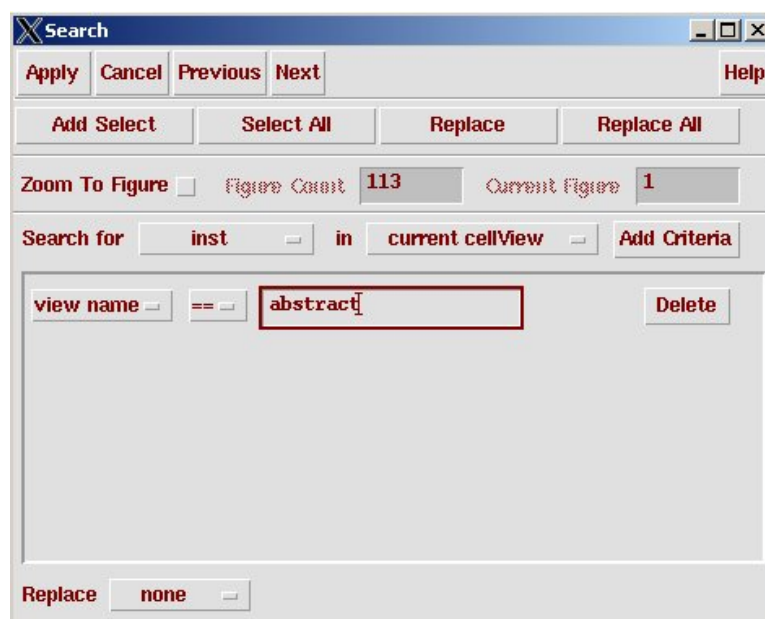


Figure 10.35: **Search** dialog box for finding all the cell abstracts

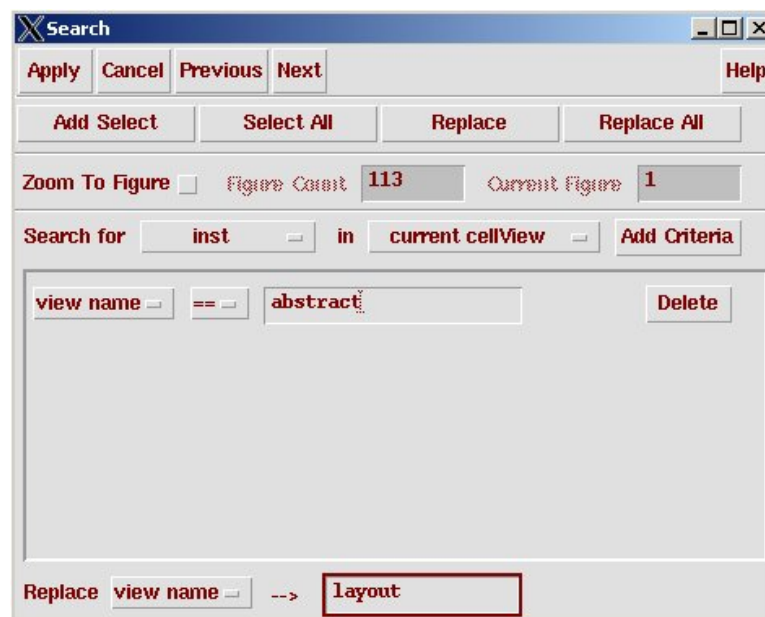


Figure 10.36: **Search** dialog box for replacing abstracts with layouts

Now you can run **DRC** on the cell with **Verify** → **DRC** the same way you've done for other cells. You should have zero errors, but if you do have errors you'll need to fix them. You can also extract the circuit using **Verify** → **Extract** for **LVS**. Again you should have zero errors.

Importing the structural Verilog

Recall that the structural Verilog from **SOC Encounter** is different than the structural Verilog that came directly from Synopsys **design compiler** because things were optimized and a clock tree was added. To import the new Verilog you can use the same procedure as described in Chapter 8. Use the **CIW** menu **File** → **Import** → **Verilog...** Fill in the fields:

Target Library Name: The library you want to read the Verilog description into. In this case it will be **counter**.

Reference Libraries: These are the libraries that have the cells from the cell libraries in them. In this case they will be **example** and **basic**. You will use your own library in place of **example**.

Verilog Files to Import: The structural Verilog from **SOC Encounter**. In this case it's **counter.v** from my **soc/counter** directory.

-v Options: This is the Verilog file that has Verilog descriptions of all the library cells. In this case I'm using **example.v**. You'll use the file from your own library.

The dialog box looks like that in Figure 10.37. You can click on **OK** to generate a new schematic view based on the structural Verilog. Strangely this will result in some warnings in the **CIW** related to bin files deep inside the **Cadence IC 5.1.41** directory, but it doesn't seem to cause problems. You now have a schematic (Figure 10.38) and symbol (Figure 10.39) of the counter. The log file of the Verilog import process should show that all the cell instances are taken from the cell library (**example** in this case).

Once you have a schematic view you can run **LVS** to compare the **extracted** view of the cell to the **schematic**. They should match! This cell may now be used in the rest of the flow. If this is the final circuit you can use the chip assembly tools (Chapter 11) to connect it to pads. If it's part of a larger circuit you can use it in subsequent uses of with the chip assembly router or back in **SOC Encounter**.

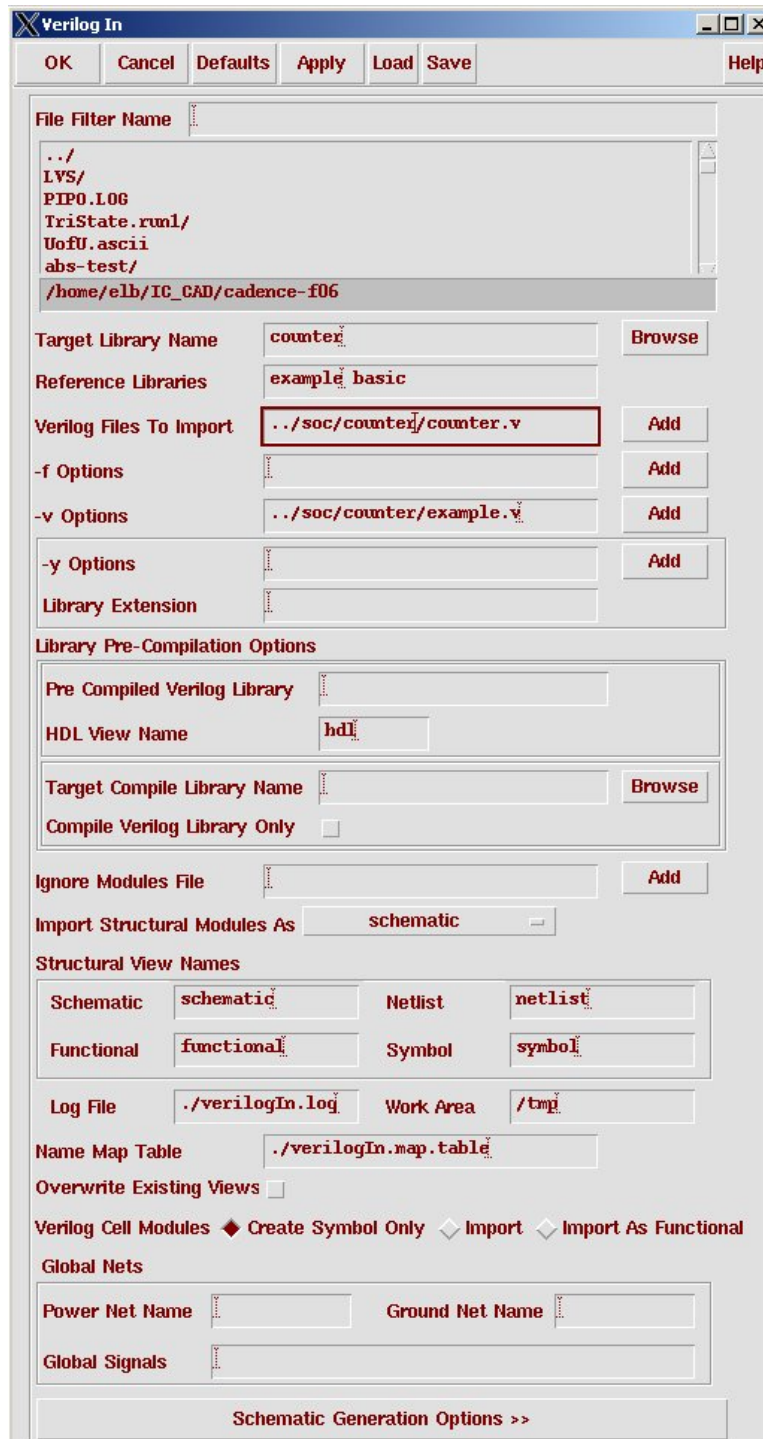


Figure 10.37: Dialog box for importing structural Verilog into a new schematic view

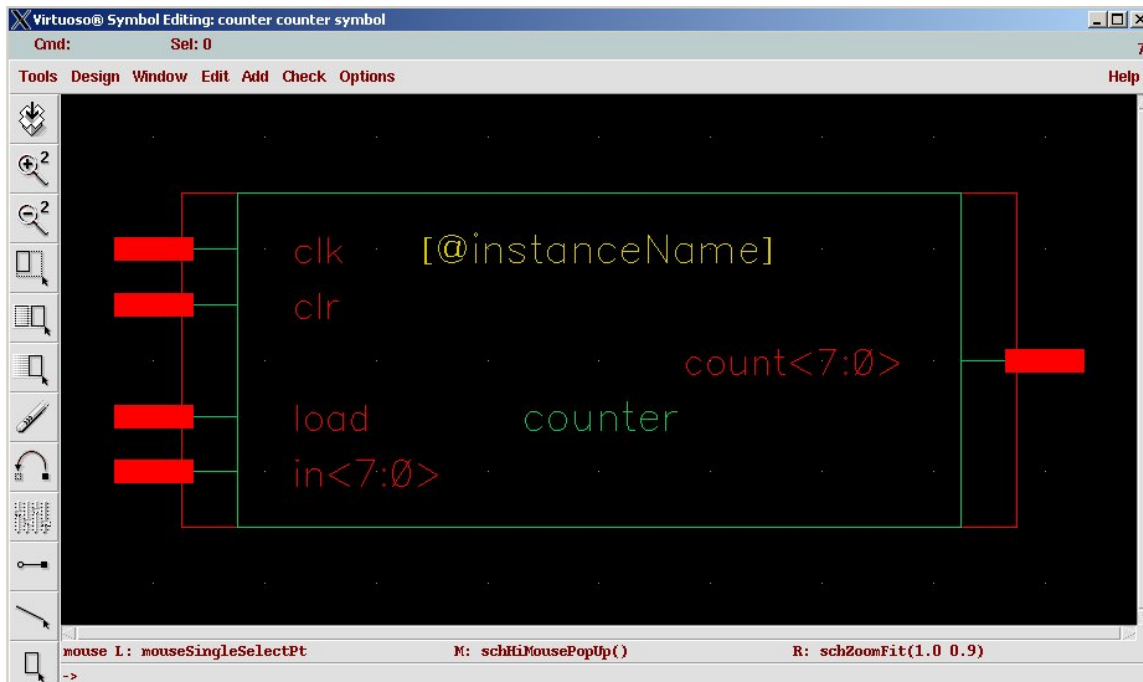


Figure 10.38: Schematic that results from importing the structural counter from **SOC Encounter** into **icfb**

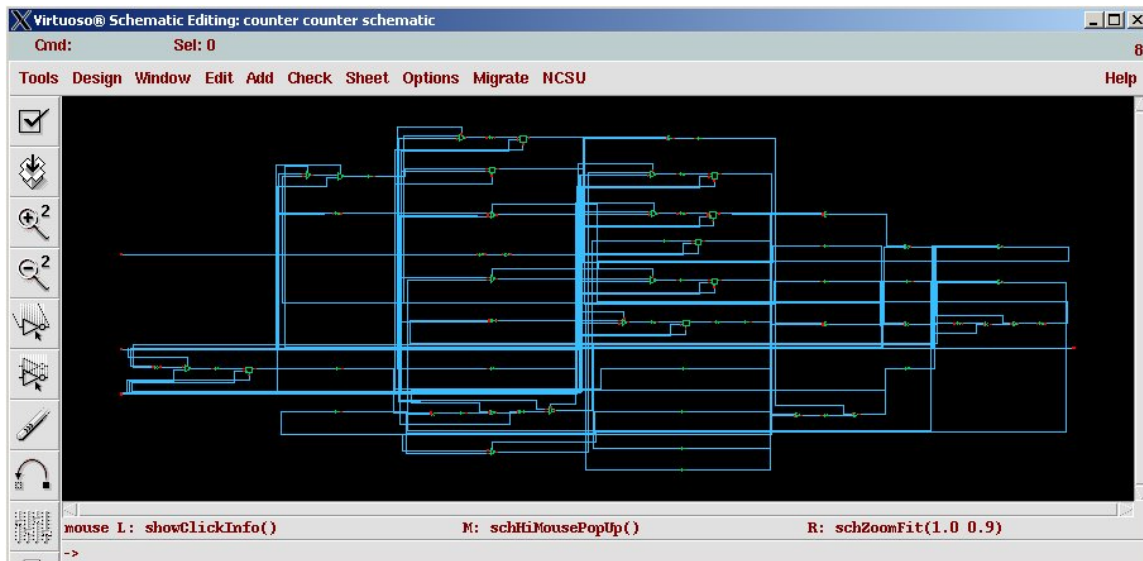


Figure 10.39: Symbol that is created for the counter

10.2 Encounter Scripting

SOC Encounter, like most CAD tools, can be driven either from the window-based GUI or from scripts. Once you've completed the process of place and route a few times from the GUI, you may want to automate parts of the process using scripts. In general, you probably want to do the floor planning portion of the flow by hand, and then use a script for the rest of the flow (placement, optimization, clock tree synthesis, optimization, routing, optimization, exporting data).

You can always peruse the **SOC Encounter Text Command Reference Guide** in the documentation to see how to write the script commands, or you can look in the output logs from your use of the GUI. **SOC Encounter** produces two logs while you are using the GUI: **encounter.log** which logs the messages in the shell window, and **encounter.cmd** which logs the commands that you execute while using the menus. You can use the **encounter.cmd** file to replicate commands in scripts that you previously used in the GUI.

10.2.1 Design Import with Configuration Files

The first step in using **SOC Encounter** is to use the **Design Import** menu to read in your initial structural Verilog file. You can streamline this process by using a **configuration** file. In the class directory related to **SOC Encounter** (`/uusoc/facility/cad_common/local/class/6710/cadence/SOC`) you will find an input configuration file called **UofU_soc.conf** that you can customize for your use. Change the file names and library names in the top portion of the file for your application before using it. The first part of this file, as configured for the **counter** example is shown in Figure 10.40. The full configuration file template is shown in Appendix C, and, of course, you can see it on-line.

To use this configuration file, after filling in your information, use the **Design** → **Design Import** menu, but choose **Load** and then load the **.conf** file that you've customized for your circuit. This will read in the circuit with all the extra information in the other tabs already filled in.

10.2.2 Floor Planning

You can now proceed to the floor planning stage of the process using the **Floorplan** → **Specify Floorplan** menu. You'll see that the **Aspect Ratio**, **Core Utilization**, and **Core to IO Boundary** fields will be already filled in, although may not be exactly **1**, **0.7**, and **30** (but they'll be close!). You can modify things, or accept the defaults. From this point you can proceed as

```
#####  
#                                     #  
#   Encounter Input configuration file   #  
#   University of Utah - 6710           #  
#                                     #  
#####  
# Created by First Encounter v04.10-s415_1 on Fri Oct 28 16:15:04 2005  
global rda_Input  
#  
#####  
# Here are the parts you need to update for your design  
#####  
#  
# Your input is structural verilog. Set the top module name  
# and also give the .sdc file you used in synthesis for the  
# clock timing constraints.  
set rda_Input(ui_netlist)           {counter_struct.v}  
set rda_Input(ui_topcell)           {sounter}  
set rda_Input(ui_timingcon_file)    {counter_struct.sdc}  
#  
# Leave min and max empty if you have only one timing library  
# (space-separated if you have more than one)  
set rda_Input(ui_timelib)           {example.lib}  
set rda_Input(ui_timelib,min)       {}  
set rda_Input(ui_timelib,max)       {}  
#  
# Set the name of your lef file or files  
# (space-separated if you have more than one).  
set rda_Input(ui_leffile) {example.lef}  
#  
# Include the footprints of your cells that fit these uses. Delay  
# can be an inverter or a buffer. Leave buf blank if you don't  
# have a non-inverting buffer. These are the "footprints" in  
# the .lib file, not the cell names.  
set rda_Input(ui_buf_footprint)     {}  
set rda_Input(ui_delay_footprint)   {inv}  
set rda_Input(ui_inv_footprint)     {inv}  
set rda_Input(ui_cts_cell_footprint) {inv}
```

Figure 10.40: Configuration file for reading in the **counter** example

in the previous sections. The `<filename>.conf` configuration file will have made the **Design Import** process much easier.

10.2.3 Complete Scripting

You could run through the rest of the flow by hand at this point, or you could script the rest of the flow. In the class directory you'll find a complete script that runs through the entire flow. This script is called **UofU_opt.tcl**. You can modify this script for your own use. You may, for example, want to run through the floor planning by hand, and run the script for everything else. Or you may want to adjust your **configuration** file for the floorplan you want, and run the script for everything. Or you might want to extract portions of the script to run separately. You can also use the script as a starting point and add new commands based on the commands that you've run in the GUI. It's all up to you! This script is **not** meant to be the end-all be-all of scripts. It's just a starting point.

I haven't yet found a way to execute scripts from the GUI or the shell once **SOC Encounter** is running. Instead, you can run the script from the initial program execution with `cad-soc -init <scriptfile>` which will run the script when the program starts up. The first part of the **UofU_opt.tcl** script is shown in Figure 10.41 as I configured it for the **counter** example. The full script is in the class directory and in Appendix C. Remember that if you try something that works well in the GUI, you can look in the **encounter.cmd** log file to find the text version of that command to add to your own script.

```
#####  
#                                                                    #  
#   Encounter Command script                                          #  
#                                                                    #  
#####  
  
# set the basename for the config and floorplan files. This  
# will also be used for the .lib, .lef, .v, and .spef files...  
set basename "counter"  
  
# set the name of the footprint of the clock buffers  
# from your .lib file  
set clockBufName inv  
  
# set the name of the filler cells in your library - you don't  
# need a list if you only have one...  
set fillerCells [list FILL FILL2]  
  
#####  
# You may not have to change things below this line - but check!  
#  
# You may want to do floorplanning by hand in which case you  
# have some modification to do!  
#####  
  
# Set some of the power and stripe parameters - you can change  
# these if you like - in particular check the stripe space (sspace)  
# and stripe offset (soffset)!  
set pwidth 9.9  
set pspace 1.8  
set swidth 4.8  
set sspace 100  
set soffset 75  
  
# Import design and floorplan  
# If the config file is not named $basename.conf, edit this line.  
loadConfig $basename.conf 0  
commitConfig  
  
...
```

Figure 10.41: The first part of the **UofU_opt.tcl** script for **SOC Encounter**