Chapter 11

Chip Assembly

THE PROCESS of wiring up pre-designed modules to make a complete chip core, or taking a finished core and routing it to the chip pads is known as *chip assembly*. Cadence has yet another tool that is designed specifically for this set of tasks known as the **Cadence Chip Assembly Router** (**ccar**) which is part of the **IC Craftsman (ICC)** tool set. This tool will route between large pre-designed blocks which are not placed on a regular grid. This makes it fundamentally different than **SOC Encounter** which wants to place small cells on a fixed grid. The blocks that are routed by **ccar** could be blocks that are placed and routed by **SOC Encounter**, or could also be blocks that are designed by hand or by other tools (i.e. custom datapath circuits, memories, or other dense regular arrays). Although there are certainly more features of **ccar** then are described here, we use **ccar** strictly for routing. That is, the user places the blocks by hand, connects the global power and ground nets, then uses **ccar** to connect the signal wires between those large blocks. The **ccar** tool is also used to route between finished cores and the pad ring.

11.1 Module Routing with ccar

The **ccar** tool, unlike other tools we've used so far, is not invoked directly. Instead it is used in conjunction with the Cadence **Composer** and **Virtuoso** tools. The overall process is:

1. Make a schematic that contains your modules connected together. Input and output pins should be used to indicate signals that enter and leave the collection of modules.

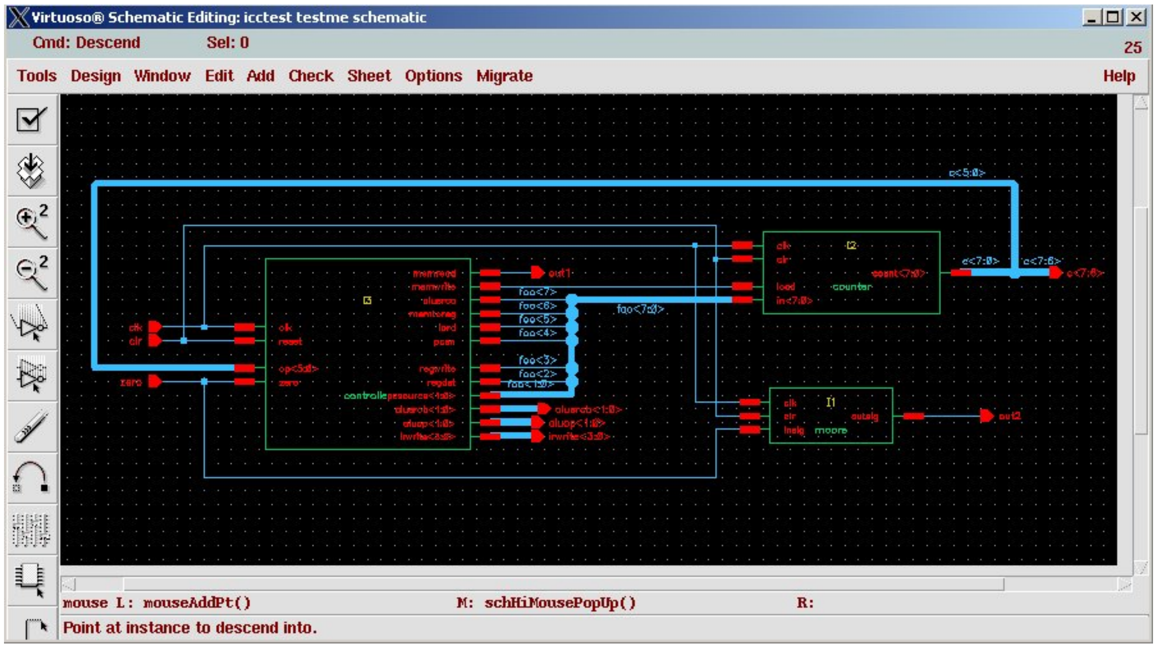
2. Use **Virtuoso-XL** to generate a new layout based on that schematic.

3. Place the modules by hand in the **layout** view.
4. Place the IO pins that were defined in the schematic (and generated in the layout) in the desired spots in the layout.
5. Connect **vdd!** and **gnd!** so that the modules are connected to a common power network.
6. Export the **layout** view to **ccar** for signal routing
7. Route the signal nets in **ccar**
8. Import the routed module back to **Virtuoso-XL**. Then DRC, extract, and LVS.

So, to start out you need:

1. The modules that you will be connecting need to have layout views that have all the connection points marked with shape pins of the right type (metal1, metal2, or metal3) and named the same things as on the symbol (blocks routed by **SOC Encounter** already have these).
2. For each cell you also need a symbol view of that cell that has the same pins on the interface of the symbol that are in the layout (cell blocks imported from the structural Verilog view generated by **SOC Encounter** have these symbols too).
3. You need a schematic showing the connection of parts that you want to route together. That is, you make a schematic that includes instances of the parts that you want to use, and all the connections between them. This is a good thing to have in general because you'll need it to simulate the functionality of the schematic and for LVS anyway. It's also what tells **ccar** which signals it should route and to where.
4. A rules file for the **IC Craftsman** (**icc**) **ccar** router called **icc.rul**. Copy this file from **/uusoc/facility/cad_common/local/class/6710/cadence/ICC** to the directory from which you start **cad-ncsu**.
5. Also copy the **do.do** file from that same class directory to the directory from which you start **cad-ncsu**. You'll need both of these later when you start up **ccar**.

What this process will do is generate a connected layout that corresponds to the schematic. You will do the placement by hand, but the router



```
mouse L: mouseAddPt()                M: schHiMousePopUp()                R:
Point at instance to descend into.
```

Figure 11.1: Starting **schematic** showing the three connected modules

will connect everything. For this example I'll take three modules that I've previously placed and routed with **SOC Encounter**: the **moore** example from Chapter 8, the **counter** example from Chapter 10, and the **controller** state machine from the MIPS example in the Harris/Weste CMOS textbook [1]. Connecting these cells together makes absolutely no functional sense, but it does show how three pre-assembled modules can be connected together with **ccar**. Each of these three examples has been imported into **icfb** using the procedure described in Chapter 10, Section 10.1.12. This means that I have (among other things) **layout**, **schematic**, and **symbol** views of each of these modules that I can use to make my new schematic as a starting point. The example starting schematic is shown in Figure 11.1. Note that this **schematic** can be in a whole new library if you like in order to keep things separated.

11.1.1 Preparing a Placement with Virtuoso-XL

In this part of the process you will generate a layout from the schematic and drag the components to where you want them placed. Start by opening the **schematic** view. From the schematic view select **Tools** → **Design Synthesis** → **Virtuoso-XL**. **Virtuoso-XL** is just **Virtuoso** with some extra features enabled. In this step you will be asked if you want to open an existing cellview or make a new one. It's talking about the layout view that it's about to generate. I'm assuming that you have no layout for the current schematic yet so you'll want to make a new one. If you have an existing layout but want to start over, this is where you can start over with a new layout view. This command will open a new **Virtuoso-X** window, and resize and replace the other windows on your screen. You may need to move things around after this process to see everything.

In the new **Virtuoso-XL** window select **Design** → **Gen From Source...** This will generate an initial layout based on the schematic as the source file. The dialog box is shown in Figure 11.2. The dialog lets you pick a layer for each external IO pin. This will determine what layer the end of the wire will be when the router creates it. If you have a lot of pins it's faster to choose one layer as default and apply that default to everything. You can then change individual pins to something else if you like. Make sure you choose a reasonable routing layer for your pins like one of the metal layers. The default layer before you change things is likely to be something unreasonable like **nwell**.

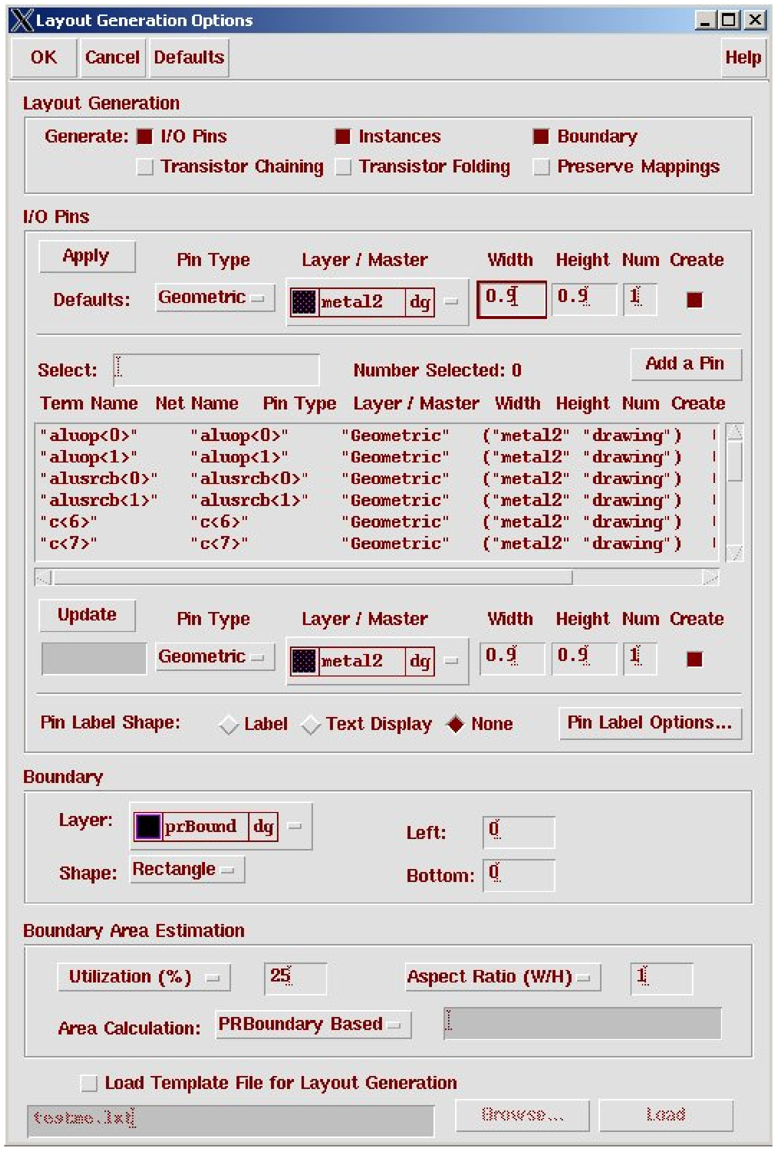
After you execute the **Gen From Source** you will see that the layout has a large purple box which defines the placement area (it's a box of type **prBoundary**), and your cells scattered outside that box. This view is shown in Figure 11.3. You can't see them until you zoom in, but all the IO pins defined in your schematic are just below the lower left corner of the **prBoundary** box.

You may need to make ***prBoundary*** *an active layer using the* ***Edit*** *→* ***Set Valid Layers*** *menu in the* ***LSW*** *(Layer Selection Window) in order to resize that layer.*

Now that you have in initial layout in the **Virtuoso-XL** window you can pick them up and move them to where you want them. Your job is to place both the modules and the IO pins inside the **prBoundary** square. You can also resize the **prBoundary** if you'd like a smaller or differently shaped final module. When you move the cells connecting lines show up that show you how this component is connected to the other. Also, when you select a cell in the layout window that same cell is highlighted in the schematic window. This is very handy for figuring out which cell in one window corresponds to which cell in the other.

Placing the IO pins is a little tricky just because they're likely to be very small compared with the modules. I find that it's easiest if I zoom in to the lower left corner of the **prBoundary** where the pins are and select a pin. Then use the **move** hotkey (`m`) to indicate that you're about to move the pin and click near the pin to define a starting point. Now you can use `f` and the zoom keys (`Z` and `ctl-z`) to change views while you're still moving the pin. You can also see the connecting line which shows where the pin will eventually be connected which can guide you to a good placement.

When you've placed all the modules and pins you can look at the con-



Layout Generation

Generate: [✓ I/O Pins] [✓ Instances] [✓ Boundary] Transistor Chaining, Transistor Folding, Preserve Mappings

I/O Pins

	Pin Type	Layer / Master	Width	Height	Num	Create
Defaults:	Geometric	metal2 dg	0.9	0.9	1	🟥

Select: Number Selected: 0

```
Term Name   Net Name   Pin Type   Layer / Master   Width  Height  Num  Create
"aluop<0>"      "aluop<0>"      "Geometric"    ("metal2" "drawing")
"aluop<1>"      "aluop<1>"      "Geometric"    ("metal2" "drawing")
"alusrcb<0>"    "alusrcb<0>"    "Geometric"    ("metal2" "drawing")
"alusrcb<1>"    "alusrcb<1>"    "Geometric"    ("metal2" "drawing")
"c<6>"          "c<6>"          "Geometric"    ("metal2" "drawing")
"c<7>"          "c<7>"          "Geometric"    ("metal2" "drawing")
```

	Pin Type	Layer / Master	Width	Height	Num	Create
	Geometric	metal2 dg	0.9	0.9	1	🟥

Pin Label Shape: Label, Text Display, [✓ None]

Boundary

Layer: prBound dg Left: 0

Shape: Rectangle Bottom: 0

Boundary Area Estimation

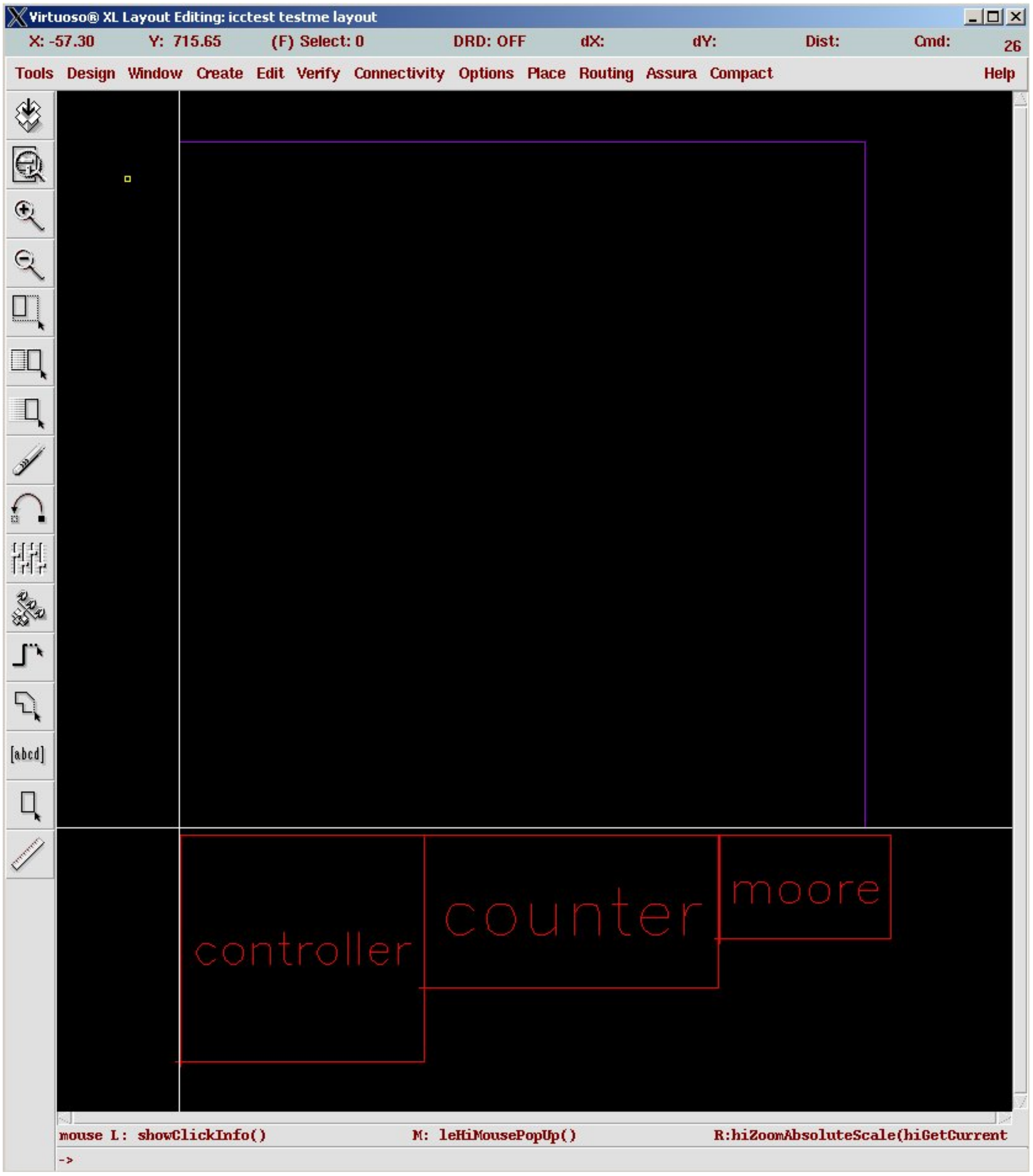
Utilization (%) 25 Aspect Ratio (W/H) 1

Area Calculation: PRBoundary Based

Load Template File for Layout Generation

testme.lxt

Figure 11.2: The **Gen From Source** dialog box



```
mouse L: showClickInfo()          M: leHiMousePopUp()          R:hiZoomAbsoluteScale(hiGetCurrent
->
```

Figure 11.3: Initial layout before module and IO placement

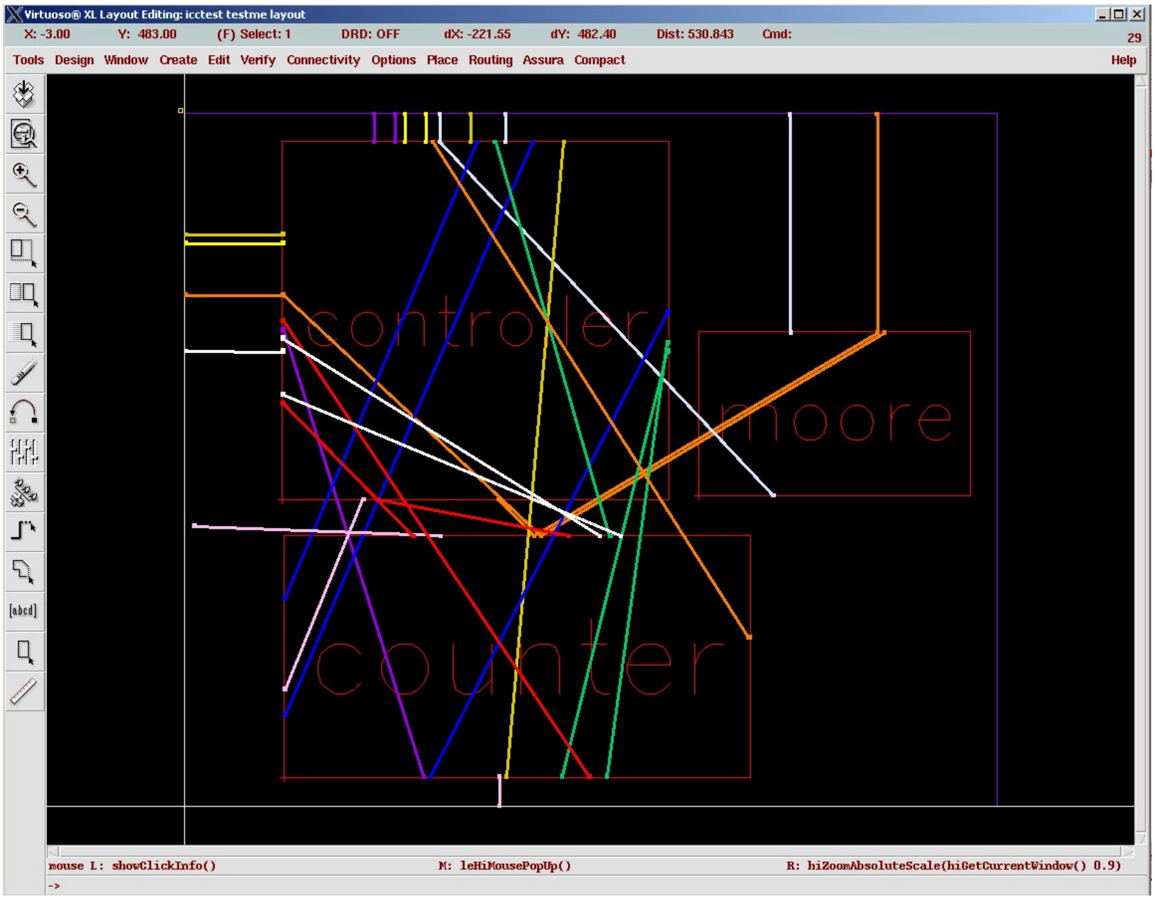

Virtuoso® XL Layout Editing: icctest testme layout
X: -3.00 Y: 483.00 (F) Select: 1 DRD: OFF dX: -221.55 dY: 482.40 Dist: 530.843 Cmd:
29
Tools Design Window Create Edit Verify Connectivity Options Place Routing Assura Compact
Help
controller
moore
counter
mouse L: showClickInfo()
M: leHiMousePopUp()
R: hiZoomAbsoluteScale(hiGetCurrentWindow() 0.9)
->

Figure 11.4: A placement of modules and IO pins with unrouted nets turned on

nectivity using the **Connectivity** → **Show Incomplete Nets** option in **Virtuoso-XL**. Because nothing is routed yet, this shows all the connections that will eventually be made by **ccar**. A placement of the modules and pins is shown in Figure 11.4.

Now you should connect the power and ground nets of the various modules in your layout. If you don't do this now you might run out of space to do it later once the signals are routed. In Figure 11.5 I've connected the rings of the modules together with fat 9.9 micron wires to match the pitch of the power and ground rings that were placed by **SOC Encounter**. Notice that I've kept the routing conventions intact. That is, **metal1** is used horizontally and **metal2** is used as a vertical routing layer. Because **ccar** will also use these conventions it's important to keep that in mind so as not to restrict the routing channels. At this point you should save the layout and run DRC to make sure that everything is all right before you send the layout to the **router** tool.

Make sure that your module placement, IO pin placement, **prBoundary** size and shape, and power routing is how you want it. You can play around with this a lot to get things looking just right before you send it to the router. The router will fill in the rest of the signal routing for you. Of course, if you've made your **prBoundary** too small and left too little room for signal routing you might have to redo the floorplan later!

11.1.2 Invoking the ccar router

Remember to copy ***icc.rul*** *and* ***do.do*** *from the class ICC directory before you run this step.*

Now that you have a placed and power-routed layout you can send this to the router. Use the **Routing** → **Export to Router** command. The dialog box is shown in Figure 11.6. Everything should be filled in correctly, but make sure that the **Use Rules File** box is checked and that the **icc.rul** file is specified. Also make sure that the **Router** that is specified is **Cadence chip assembly router**. Clicking **OK** will start up **ccar** on your cell. A new **ccar** window will pop up that shows your layout with the not-yet-connected nets shown as in Figure 11.7.

Look at the log information that show up in the shell you used to start **cad-ncsu** and make sure that there are no issues with **ccar** as it starts up. Unless you have specified your pins strangely back in the **Gen From Source** step, it's unlikely that there will be issues here, but you should always check.

This may not actually be strictly necessary, but it has seemed to ease some problems in the past so for historical reasons I'm leaving this instruction in.

The first thing you should do is execute a *do file*. This will set up a few things in the icc tool so you won't have to do them by hand. Select **File** → **Execute Do File** and tell it to execute **do.do** (which you have already copied into your current directory).

Now, if you want to, you can select the layers used by **ccar** to do the routing. Everything is set up so that the router will use metal1 as a horizontal layer, metal2 as a vertical layer, and metal3 as a horizontal layer. If you want to change this, or restrict the router to not use a certain layer, you can do that now. Select the layer icon, which is the icon with the three rectangles overlapping on the left side of the **ccar** window. The pop-up will look like Figure 11.8. The icons next to the metal1, metal2, and metal3 layers show what direction they will be routed in. If you want to change the direction or restrict the router from using that layer, you can change that icon. The circle with the slash means not to use that layer as a routing layer. Note that the vias are marked as **don't use** This doesn't mean that the router won't put in vias (it will), it just means it won't make a wire out of the via layer. There's really nothing you need to change here unless you want to.

The next thing you might want to do is change the costs of various routing features. This is a mechanism to control how the router does the routing.

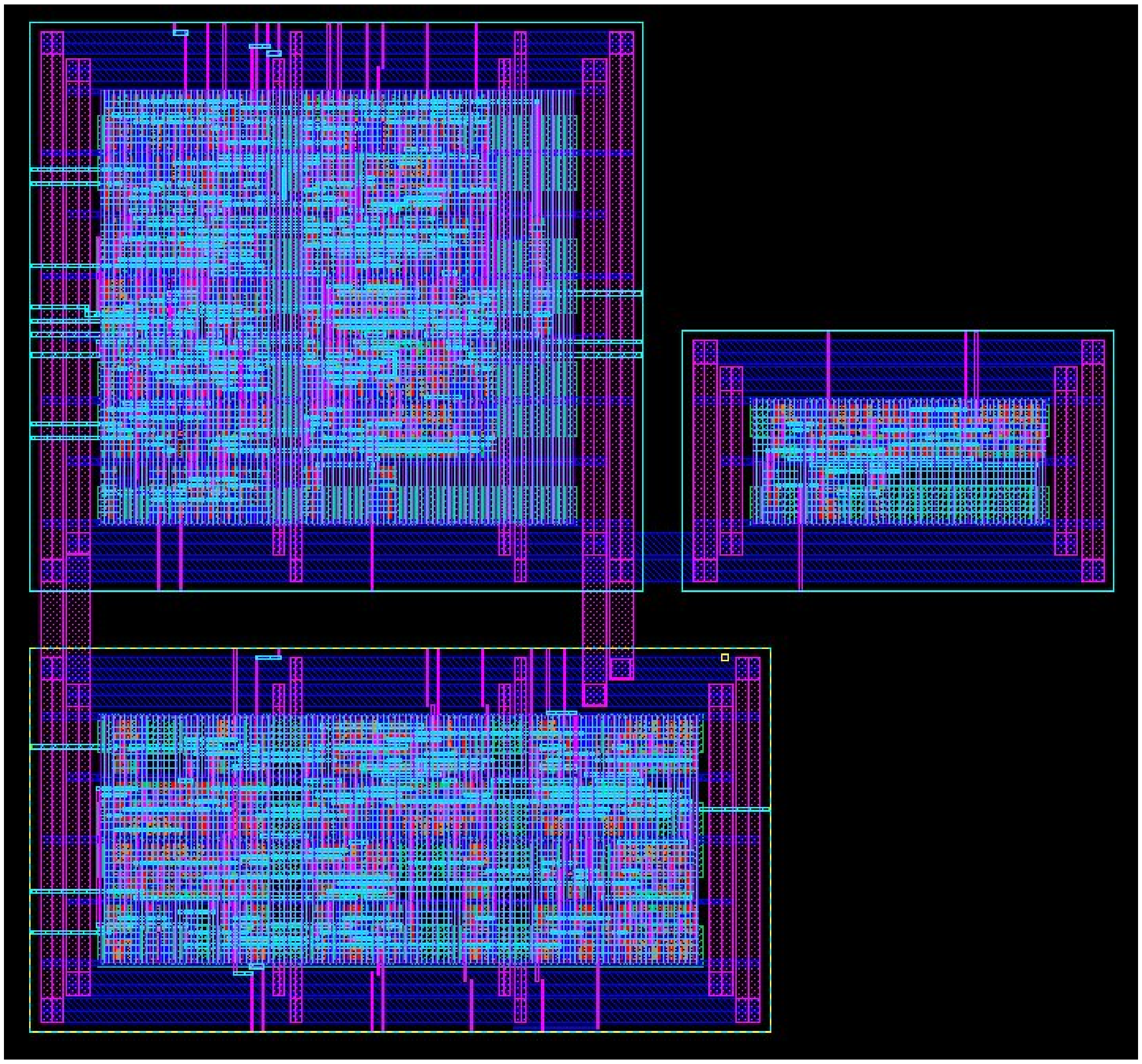
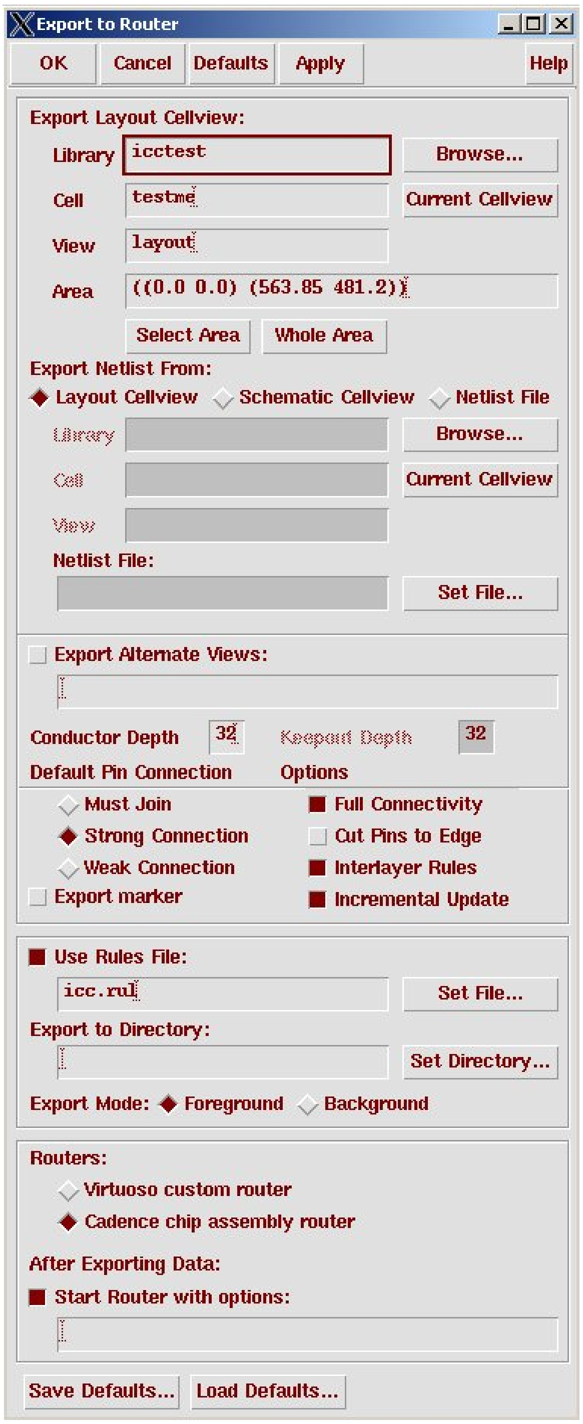


Figure 11.5: Layout showing placement and power routing before routing



Export Layout Cellview:
Library icctest
Cell testme
View layout
Area ((0.0 0.0) (563.85 481.2))

Export Netlist From:
[✓ Layout Cellview] Schematic Cellview Netlist File
Library
Cell
View
Netlist File:

Export Alternate Views:
Conductor Depth 32 Keepout Depth 32

Default Pin Connection
Must Join
[✓ Strong Connection]
Weak Connection
Export marker

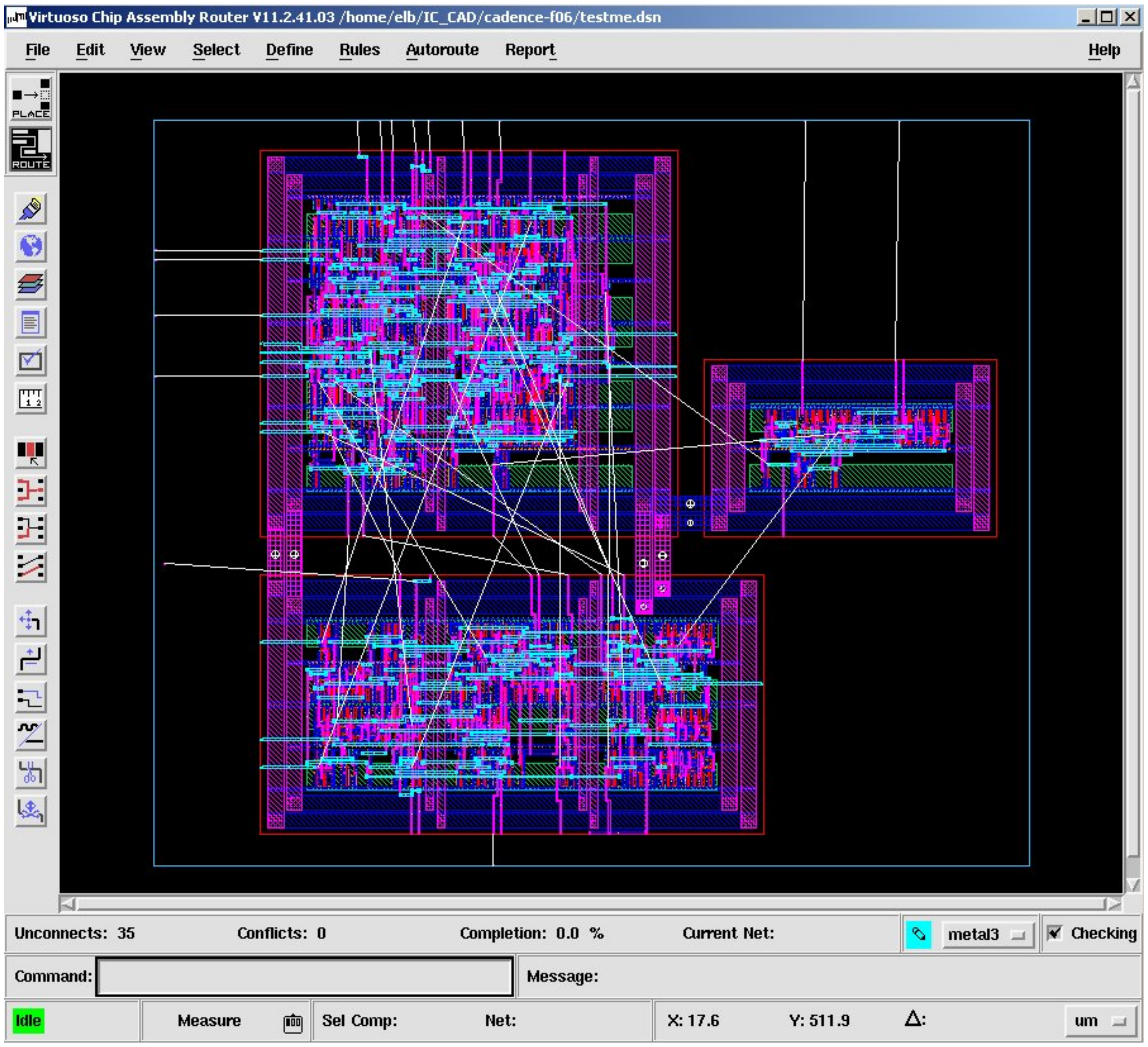
Options
[✓ Full Connectivity]
Cut Pins to Edge
[✓ Interlayer Rules]
[✓ Incremental Update]

[✓ Use Rules File:]
icc.rul
Export to Directory:
Export Mode: [✓ Foreground] Background

Routers:
Virtuoso custom router
[✓ Cadence chip assembly router]

After Exporting Data:
[✓ Start Router with options:]

Figure 11.6: **Export to Router** dialog box


Virtuoso Chip Assembly Router V11.2.41.03 /home/elb/IC_CAD/cadence-f06/testme.dsn
File
Edit
View
Select
Define
Rules
Autoroute
Report
Help
PLACE
ROUTE
metal3
Checking
Measure
um

Unconnects: 35 Conflicts: 0 Completion: 0.0 % Current Net:

Command: Message:

Idle Sel Comp: Net: X: 17.6 Y: 511.9 Δ:

Figure 11.7: Initial **ccar** window

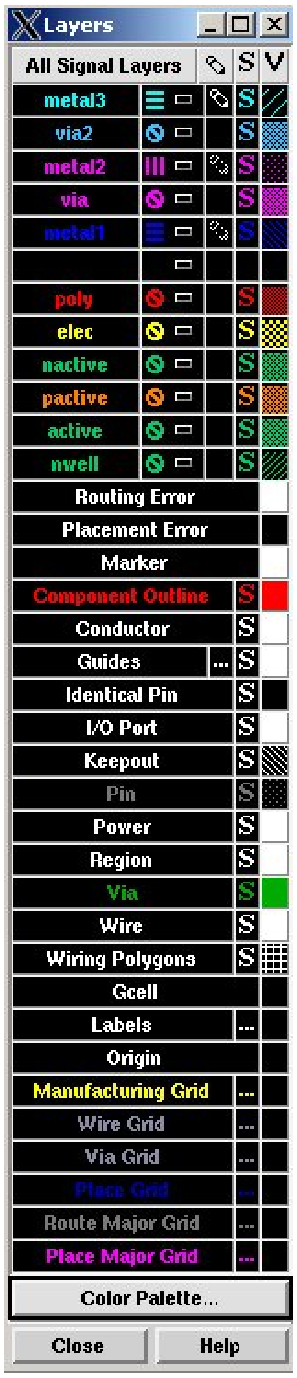
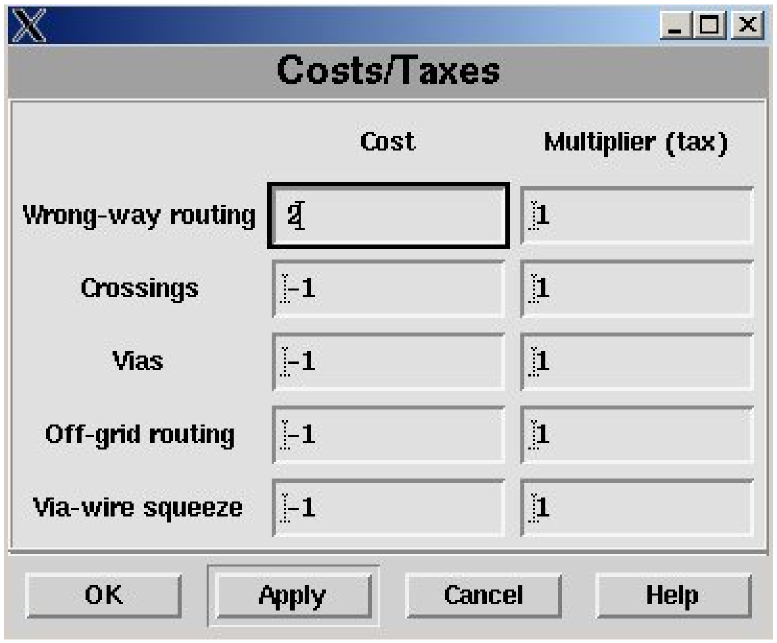


Figure 11.8: Layer configuration dialog box



	Cost	Multiplier (tax)
Wrong-way routing	2	1
Crossings	-1	1
Vias	-1	1
Off-grid routing	-1	1
Via-wire squeeze	-1	1

Figure 11.9: Routing cost factor dialog box

Choose **Rules** → **Cost/taxes** and you can modify the relative costs of various routing features. The dialog box is shown in Figure 11.9. The -1 means that there is no penalty. Putting in any number raises the cost penalty in the routing algorithm and makes it less likely that the router will behave in that way. For example, if you feel strongly that the routing layers should stick with the h-v-h routing plan, then add some penalty for wrong-way routing. I haven't played with this enough to know how changing this really affects the circuit. Feel free to play around here and see what happens. Leaving it alone will result in good, generic results.

Normally you want **ccar** to route all the signal pins. However, if you want it to leave some wires unrouted (because you want to route them by hand for some reason) you can use the **Edit** → **[UN]Fix Nets** to *fix* the nets that you don't want **ccar** to route. There are clearly many many more options that you can play with, but these are the basics. Feel free to explore the others.

Once you've finished setting things up you can tell **ccar** to route the nets. This is a multi-step process:

1. Select **Autoroute** → **Global Route** → **Local Layer Direction** and tell it to get its layer directions from the **Layer Panel**.
2. Now select **Autoroute** → **Global Route** → **Global Route** and tell

it how many routing passes you'd like it to try before giving up (the default of 3 is probably fine unless you have a very congested circuit). This may not look like it's doing anything if the layout is small. It's making some notes for the global routing of signals. If you get warnings about non-optimal results, go ahead and click OK.

3. Now select **Autoroute** → **Detail Route** → **Detail Router** to get all the detailed wires. Again you tell it how many passes to take. The tighter the layout and the smaller the area you've specified, the more passes it's likely to take to get a successful route. If you have lots of room then it will probably only take 1-5 passes. A tighter routing situation may require 25 or more passes. This is a fun step because you get to watch as the router tries to connect everything.

4. If you don't get a successful route, you'll have go to the costs and reduce the costs of some of the routing features, or go back to the layers window and give it more layers to use. You may even need to go back to the layout and give the router some more room by increasing the space around the modules or increasing the size of the **prBoundary**. This example has (relatively speaking) acres of routing room so there's no problem. In fact, it routes correctly in the second pass.

5. If you do get a successful route you need to clean up after the route. The router may have introduced errors in the circuit and the routes may be a little crooked with unnecessary jogs. Select **Autoroute** → **Clean** to clean up things. This will take a post-pass on the routing and clean up messy bits.

6. You also need to remove notches. These are little gaps in the routing that got left in because of corners being turned, or other features of the routing. Select **Autoroute** → **Postroute** → **Remove Notches**. The final routed circuit is shown in Figure 11.10.

Now you're done. Save the routed circuit by selecting **File** → **Write** → **Session**. You want to save your work as a session so that you can import it back into Virtuoso and use it as layout. Once you've saved your session you can quit **ccar**. You can see that this isn't the best looking routing in the world, but it is connected according to the connections shown in the schematic, and it was done automatically.

When you saved the session in **ccar** you should have seen the routing updated in the **Virtuoso-XL** layout window. If it didn't you can import it with the **Routing** → **Import from Router** menu. You should save this view in **Virtuoso-XL**. This is now just like any other **layout** view. You can use it in your chip, or generate a symbol and use it at another level of your hierarchy, and even use it with **ccar** at another level of the hierarchy. The

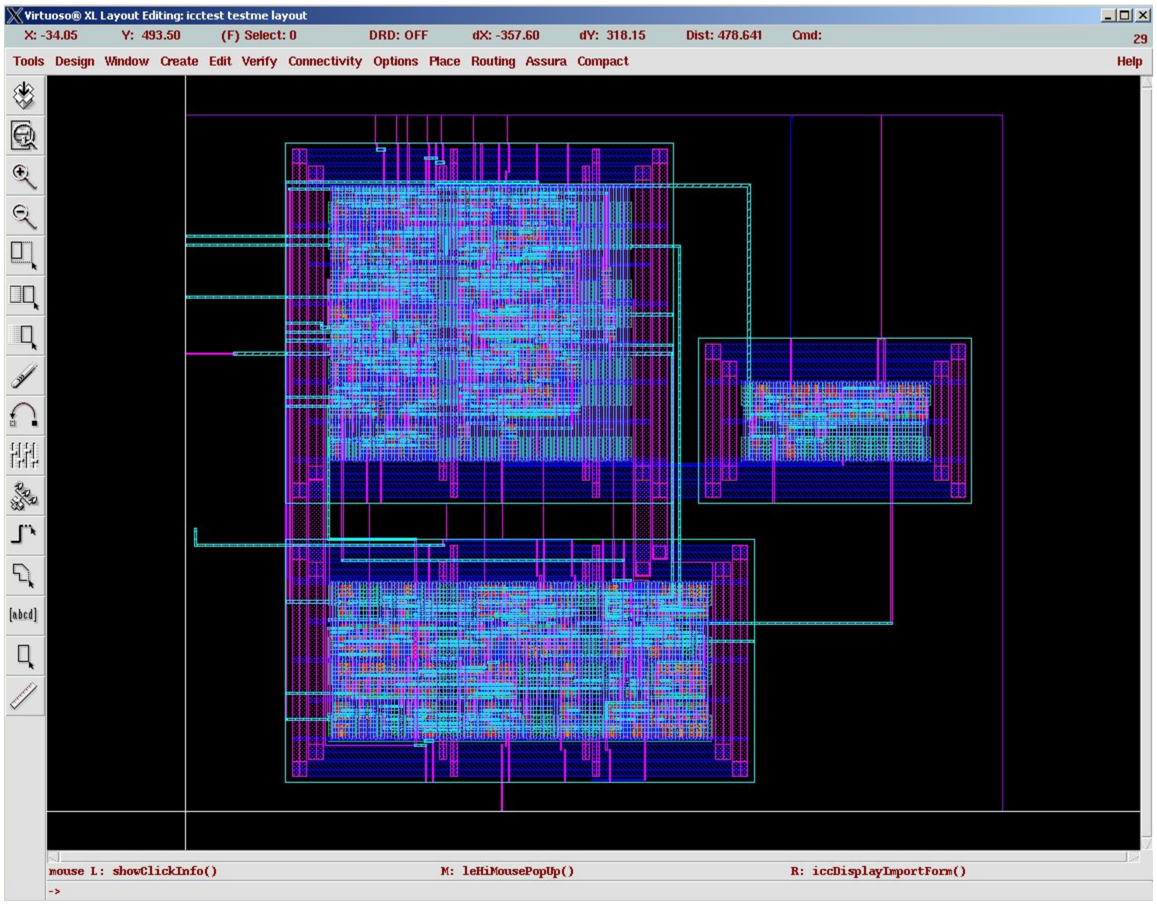


Figure 11.10: Final routed circuit (shown in **Virtuoso** window)

only difference between this and a **layout** view that you did by hand is that after you did a placement of the cells by hand in this example, the Chip Assembly Router did the interconnect for you.

Once you have the layout back in Virtuoso, you will want to do the usual things to it: DRC and LVS for example. The first thing you should do is run DRC to make sure that the autorouted circuit does not have any new DRC errors in it. Although the autorouter is good, it's not perfect and it may leave a few small errors around. In particular, you may get metal spacing errors at the point where the autorouted wires connect to the **SOC**-routed block. This is because the connections that **SOC** uses are shape pins that use **pin** type material instead of **drawing** type material. You can see this if you open the **SOC**-routed layout and zoom in close to one of the IO pins. The IO pin that **SOC** put in looks different than real metal. Each of the layers in Cadence is really a *layer-purpose pair*. That is, there is a set of layers, and a set of purposes, and each layer can be paired with a purpose. Normally you're using **drawing** purpose. The layers in your LSW have a little **dg** after them to show that they're drawing-purpose layers.

__SOC__ uses __pin__ layers for the IO pins that it generates, and an __extracted__ view uses __net__ purpose layers for the extracted nets.

If you do have metal spacing errors, you can fix them by hand by putting a small piece of metal over the mistake, or you can fix them by editing your **SOC** layout. To do this, open the **SOC** layout. Then use the **Edit** → **Search** dialog to find all rectangles on layer **metal1 pn** (or **metal2 pn** or **metal3 pn** depending on where you put your pins), and change them to layer their **dg** versions. This will convert them to **drawing** purpose layers and the DRC process should stop complaining.

If you have any other DRC errors, you need to check them out and fix them.

Now you can LVS against the schematic that you used to define the connectivity in the first place. This is done in the usual way generate an **extracted** view, LVS the **extracted** against the **schematic**. The only thing you need to be careful of here is to make sure **vdd!** and **gnd!** were connected before you try to LVS. You will need to connect these supply nets by hand (which you should have done before you routed the module).

11.2 Core to Pad Frame Routing with ccar

Actually, it does matter for a grade in the class. Everyone should connect their core to the pads for the final report even if the core isn't working completely yet!

In this section I will walk through the procedure for using icc to connect your chip core to a pad frame. Before you start the process of connecting your core, you need a complete core! A complete core is the complete guts of your chip with all the components connected, all the **vdd!** and **gnd!** lines connected, and simulated for functionality. If you don't have a functional, simulated, complete core, then it doesn't matter whether it has pads around

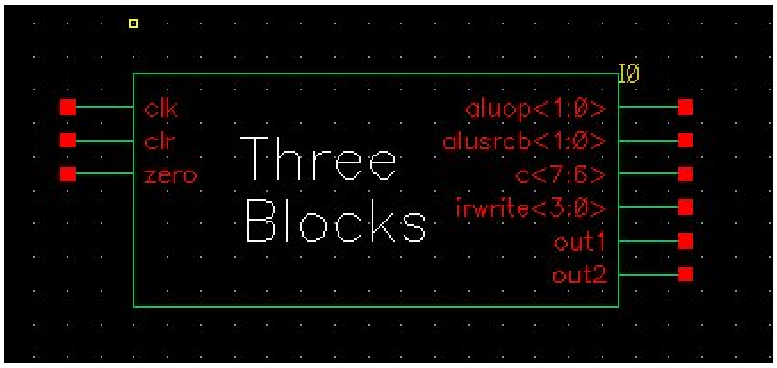


Figure 11.11: Symbol for the **Three Blocks** example core

it!

For this example, I'll start with a core made up of the three blocks from the previous Section which were placed by hand in **Virtuoso-XL** and routed by **ccar**. Your core may be completely placed and routed by **SOC**, or it might include blocks routed by **ccar** or it might be completely custom. The point is to start with a completed core. A complete core should have (at least) **layout**, **schematic**, and **symbol** views, should have **vdd!** and **gnd!** connected in the core, and should pass DRC and LVS. The three-block example from Section 11.1 has all of these characteristics. The layout was seen in Figure 11.10 and the schematic in Figure 11.1. The symbol was created using **Design** → **Create Cellview** → **From Cellview** and is seen in Figure 11.11.

11.2.1 Copy the Pad Frame

The first step is to copy the pad frame that you want to use from the **UofU_Pads** library into your own library. You need to copy the frame because you're going to modify the frame to contain the pads you want to use. The frame defines the placement of the pads, but you can adjust which type of pad you want in each pad position. Using the pre-designed frames is very important. The frames are designed to be the exact outside dimensions allowed by **MOSIS** for class fabrication. Making the frames one micron bigger would double the cost. Our "cost" is measured in *Tiny Chip Units* or TCUs. Each TCU is 1500 X 1500 microns in outside dimensions. It's possible to use multiple TCUs together to make larger chips. But, because we have a limited TCU budget, you should definitely try to fit your core in the smallest frame you can. The available frames are:

Frame1_38: A single tiny chip (1 TCU) frame with 38 signal pins (plus

1-vdd and 1-gnd). Usable core area is approximately 900 X 900 microns.

Frame2h_70: A two-tiny-chip (2 TCU) frame with 70 signal pins (plus 1-vdd and 1-gnd). The core a horizontal rectangle (long edges on top and bottom). Usable core area is approximately 2300 X 900 microns. If you aren't using all the signal pad locations and want to add extra power and ground pads you can add them as follows: vdd on pads 16 and 33, gnd on pads 48 and 74.

Frame2v_70: A two-tiny-chip (2 TCU) frame with 70 signal pins (plus 1-vdd and 1-gnd). The core a vertical rectangle (long edges on right and left). Usable core area is approximately 900 X 2300 microns. If you aren't using all the signal pad locations and want to add extra power and ground pads you can add them as follows: vdd on pads 16 and 33, gnd on pads 48 and 74.

Frame4_78: A four-tiny-chip frame with 78 signal pins (plus 3-vdd and 3-gnd). Usable core area is approximately 2300 X 2300 microns.

For this example I'll use the **Frame1_38**. I'll copy that cell from the **UofU_Pads** library to my own library so that I can modify for the needs of the core.

11.2.2 Modify the Frame schematic view

Once the frame of your choice is copied into the library you are using for chip assembly, you need to replace pads that are in the frame with the pads you want. The frames have **vdd** and **gnd** pads in the correct places for the tester so **DO NOT** change the location of the vdd and gnd pads! All the other pads in the frame are **pad_nc** for no-connect. You should replace the **pad_nc** cells with the pads you want. The cells available are:

pad_bidirhe: A bidirectional (tri-state) pad with high-enable. From the core to the pad the signals are **DataOut** and **EN**. These are outputs from your core, and inputs to the pad cell. From the pad to the core the signals are **DataIn** and **DataInB**. These are outputs from the pad and inputs to your core. The pad itself is connected to an inout pin called **pad**.

pad_in: An input pad meaning from the outside world to your core. The signals are **DataIn** and **DataInB** going from the pad to your core. The pad itself is on an input pin called **pad**.

pad_out: An output pad, meaning going from your core out to the outside world. The signal that comes from your core is called **DataOut**, and the pad itself is on an output pin called **pad**.

pad_nc: This is a pad that does not connect to your core. If you are not using all the pads in the pad ring make sure that the ones you're not using are pad_nc so that the vdd and gnd are connected in the pad ring, and so that **MOSIS** doesn't get confused by the number of bonding areas that it expects to see.

pad_vdd: A **vdd** pad. These are in the right spots for our test board so you should not move them! There are no pins because the global **vdd!** Label is used inside the pad schematic. There is a **vdd** connection in the layout view.

pad_gnd: Same thing, but for **gnd**.

pad_corner: You shouldn't have to mess with these. They are layout-only and provide **vdd** and **gnd** connectivity for the pad ring. They are placed in the correct locations in the pad frame layout views.

pad_io: An analog input/output pad with a series resistor in the io signal path. You shouldn't have to use this! Don't confuse this for the **pad_bidirhe**!

pad_io_nores: An analog pad with no series resistor. You shouldn't have to use this either!

At this point I should choose which pad locations get which signals and which pad type. This depends somewhat on the physical placement of the pins on the core layout, and somewhat on how you want your external pins to map to the package. Anything will work, but you may want to be more deterministic about where each pin goes. In this example I'm picking pin locations somewhat randomly. I'll map the signals to pins and pad types as follows:

Pin Name	Pad Number	
clk	15	pad_in
clr	27	pad_in
zero	37	pad_in
aluop<1:0>	79, 78	pad_out
alusrcb<1:0>	73, 72	pad_out
c<7:6>	70, 69	pad_out
irwrite<3:0>	11, 10, 9, 8	pad_out
out1	55	pad_out
out2	54	pad_out

This involves changing the appropriate **pad_nc** cells into **pad_in** and **pad_out** cells in the **schematic** view. You can use the $\boxed{\text{q}}$ Object Properties menu to change which cell is instantiated in that spot without moving the cell.

It's possible that the origins of the cells may be in different spots and you may have to move things around to keep things looking neat.

Once I've made the cell type changes, I will add wires and pins to the appropriate cell inputs and outputs. What we're aiming for is to define the input and output signals of the frame, collect this into a **symbol** view, and then make another schematic that has our core and frame connected together. This will be the starting point for the **ccar** router to route the signal pins to the pads.

Now add input, output, and inout (if you have bidirectional pads) pins (and wires) to your frame schematic. There are two types of pins that you need to add: signals coming from the outside world to your pad ring (these connect to the **pad** pins on the pad cells as if you were wiring the external signals to the pads), and signals going between your pad ring and your core (connecting to the **DataIn**, **DataInB**, and **DataOut** pins on the pad cells). The pad frame for this example is shown in Figure 11.12.

Make sure to get the directions right! Remember that going from your core to the pad frame is an output as far as the overall chip is concerned, but that will be an input to the frame cell because it's coming in to the frame from your core. Likewise, a signal going from your pad frame to the core is an input to your core because it's coming into your core from the outside world, but it's an output with respect to your frame cell. Think of the frame as a doughnut with the core in the hole. Outputs from your core go into the inside edge of the doughnut and emerge at the pads on the outside edge of the doughnut.

I like to name the pins that go between the core and the pad frame to be the names of the signals with a **_i** suffix for "internal," and use the existing "real" signal names on the signals that are on the pads. That way I can keep track of which signals are internal and which are on the pads and connect to the outside world. It also lets me use the same testbench on the version with pads as with just the core itself. Of course you can come up with your own naming scheme. An example of my naming scheme is seen in Figure 11.13. An automatically generated frame symbol is seen in Figure 11.14 where it is connected to the core macro in a core-with-pads **schematic** view that I'll call **wholechip**. The frame symbol was generated automatically using the **Design** → **Create Cellview** → **From Cellview** menu. Note that I've collected some of the buses into bus pins in the frame schematic, and left some expanded as individual wires just to show that both are possible.

*I modified the frame **symbol** slightly to position the pins of the **symbol** in a "nice" way with respect to the core symbol.*

When I save the **wholechip schematic** I get some warnings about the **DataInB** signals being floating outputs. I didn't connect them because I wasn't using them, so I'll ignore those warnings. Unconnected outputs are

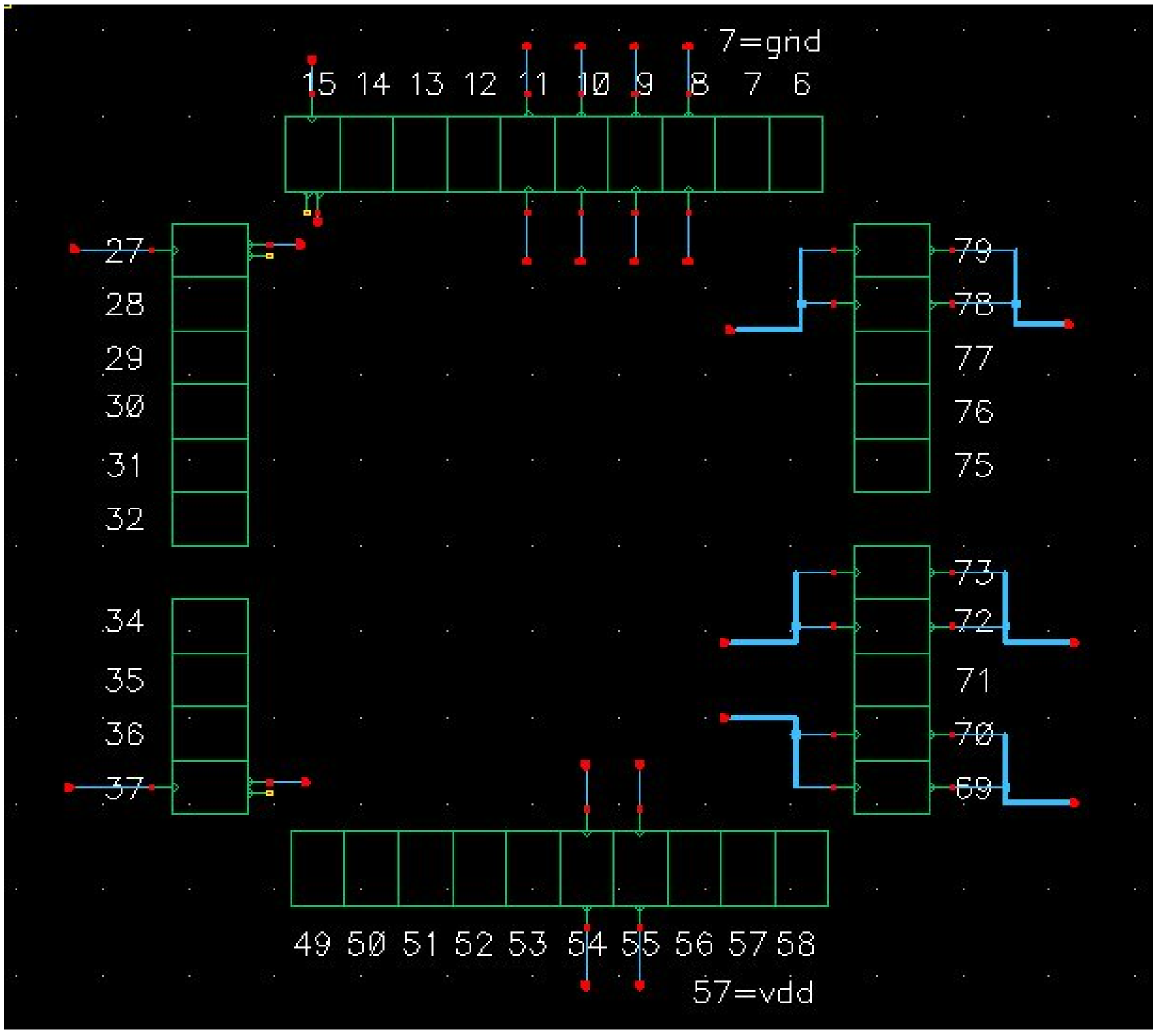


Figure 11.12: Pad frame with signal wires

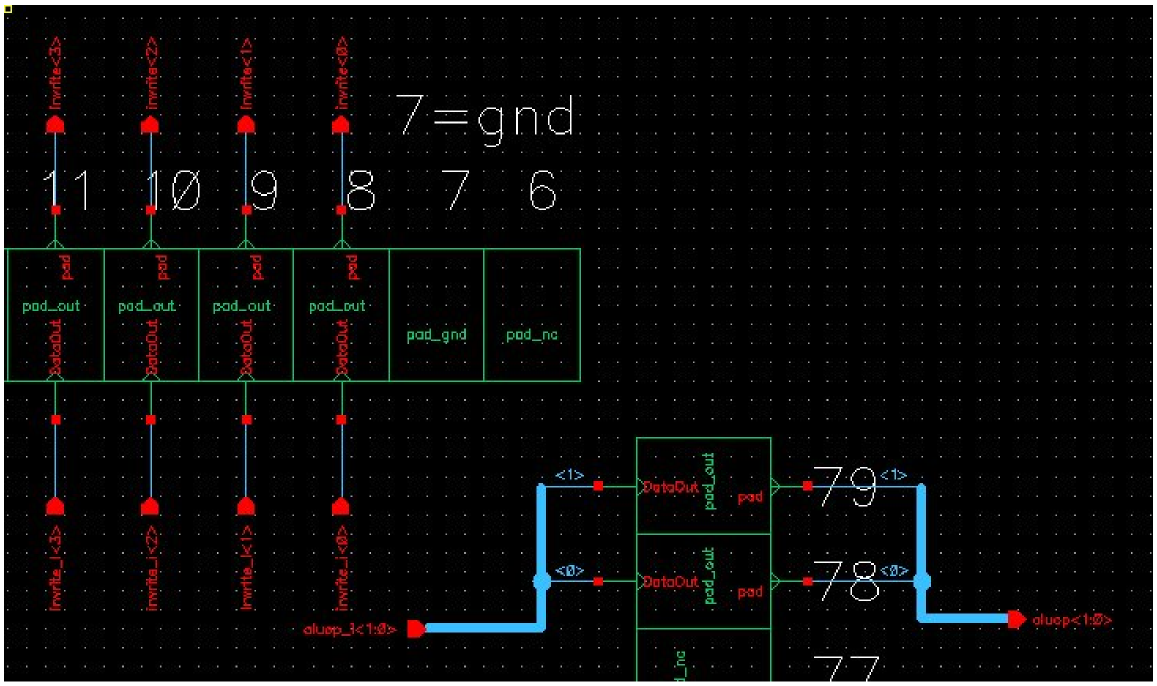


Figure 11.13: Pad frame with signal wires - zoomed view

generally all right (assuming that you really didn't want to connect them). Unconnected inputs, on the other hand, are usually a big problem. Remember that the **pad_in** input pads have both **DataIn** and **DataInBar** so you can count on getting both polarities of signals from the input pads.

Once you have a **wholechip schematic** that shows your core connected to your frame, and assuming that you've used the same naming conventions that I have, you can now simulate the whole chip at the Verilog switch level using the same test fixture that you used for the core. The only difference will be that the signals now go through pads on their way to and from your core, but the functionality should be the same as the core itself. This is a very good check that you have things connected properly!

If you've used bidirectional data paths in your design you should be using **pad_bidirhe** pads to drive those signals to and from the outside world, and you should have put **inout** pins on the **pad** connections of those pads. The connection between your core and the **pad_bidirhe** pad should have separate input and output pins for each signal, and an enable signal that determines whether the pad is driving to the output or not. If your core wants to drive a signal to the outside world then you need to make sure that the enable **EN** is high, and that the outside world isn't also trying to drive in. If you're trying to get a signal from the outside world then you need to make

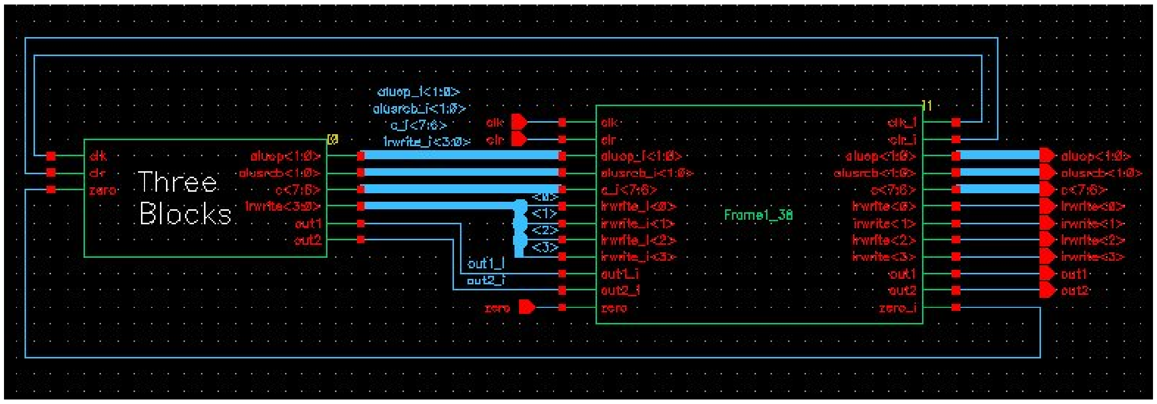


Figure 11.14: Frame and core components connected together

sure that the enable **EN**signal is low and that the outside world is driving a signal value.

In Verilog simulation there is one strange thing that happens when you simulation with **inout** pins. Because the testbench is in an **initial** block, the **inout** pins need to be reg type. So, the netlister makes a fake reg signal for you. If your signal is named **foo**, then the fake reg-type signal that shows up in the testfixture is called **io_foo**. The way to use this signal is as follows:

- If you're trying to drive a value into your chip from the outside world then set the value of **io_foo** in your testbench.
- If you're trying to let the chip drive a value out through the pad then set the value of **io_foo** to **z**. This lets the chip drive the signal through the pad, and the **io_pad** isn't trying to overdrive what your chip is sending.
- If you want to see (**$display**) the value on the pad, then look at**foo** (NOT **io_foo**).

11.2.3 Modify the Frame layout view

Before you fire up **ccar**, you need to modify your frame's **layout** to have the same pins as in your frame schematic. Once you do this you can LVS the frame layout against the schematic, and you can also use **ccar** to route the signals. First you need to update the layout of the frame to match the changes made to the schematic. That is, you need to edit the layout view of the frame and replace the **pad_nc** cells with the **pad_in**, **pad_out**, and

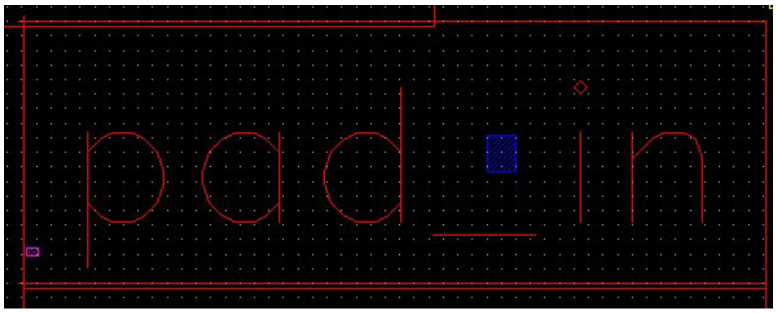


Figure 11.15: **pad_in** cell with **clk** and **clk_i** connections

pad_bidirhe cells to match the schematic. Again, you can do this by selecting the cell, getting the properties with $\boxed{\text{q}}$ and changing the cell name to the cell you want. This will maintain the orientation of the cell. Cell orientation is important in the layout because they need to be placed correctly with the pad itself on the outside of the ring. *It's critical that you NOT change the placement of the cells in the layout, just which cell they are!*

Add pins in the frame **layout** (shape pins on **metal2** for interior signal pins, shape pins on **metal1** for the **pad** pins) to each of the pads that correspond to the pins in the schematic. That is, you'll put shape pins overlapping pad cells for **clk**, **clk_i**, **clr**, **clr_i**, etc. Remember to pay attention to the direction. The **clk** pin is an input because it's coming into the pad. The **clk_i** is an output because it's going out of the frame and into the core. A good place to put the **pad** pin is on the **metal1** between the pad itself and the pad driver circuits. It's easier to see here, and you won't have the tool routing anything to that pin anyway. The signal pins need to go on the connectors around the inside edge of the frame. Note that all signal pins are in **metal2**. Note also that the pins are placed in the frame **layout** not inside the pad layouts, but placed so that they overlap the **metal1** or **metal2** in the underlying pad cell. You can also put a **vdd!** and **gnd!** **metal1** shape pin on the **pad** connection of the **pad_vdd** and **pad_gnd** cells if you like. This will make the LVS process a little faster because those nodes will be matched in that process.

These figures are rotated clockwise so they fit on the page better...

You can see a close up of the **pad_in** pad used for the **clk** and **clk_i** signals in Figures 11.15 and 11.16. In the first figure you can see the **metal1** pad shape pin for the **clk** connection on the **pad**, and the **metal2** shape pin for the **clk_i** connection on the **DataIn** connection of the pad. The second figure has the same view, but with the pad cell expanded so that you can see where the shape pins are overlapped with the pad layout. Figures 11.17 and 11.18 show extreme close ups of the **clk_i** connection to the **DataIn** port of the **pad_in** cell.

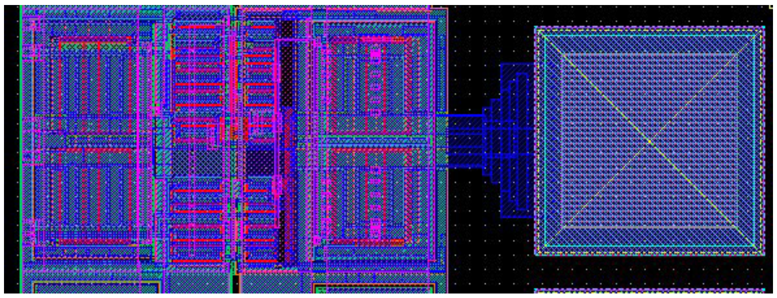


Figure 11.16: Expanded **pad_in** cell with **clk** and **clk_i** connections

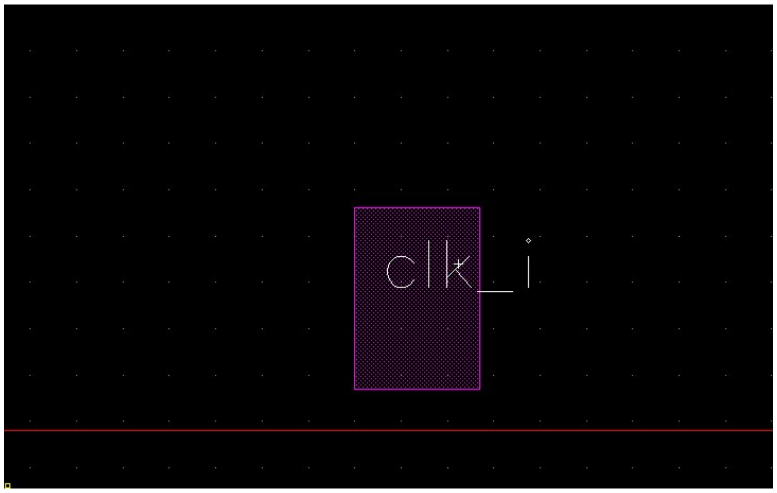


Figure 11.17: Detail of **clk_i** connection

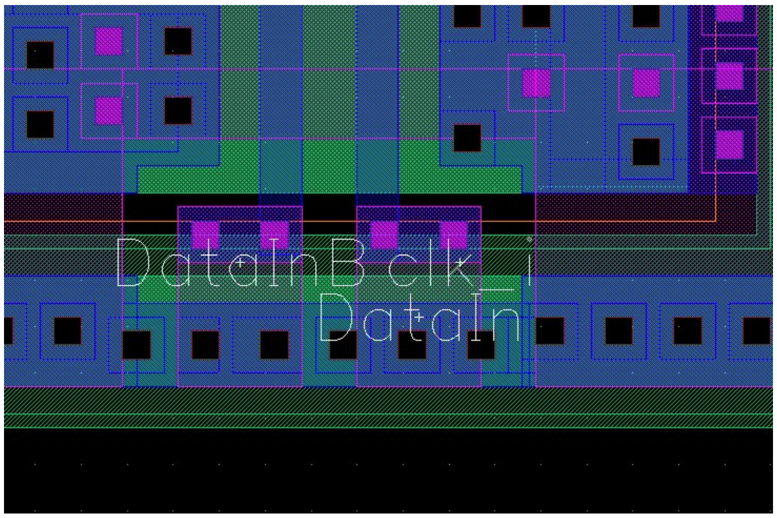


Figure 11.18: Expanded detail of **clk_i** connection

11.2.4 Routing the Core to Frame with ccar

Now that you have the following, you can use **ccar** to route the core to the frame:

1. A core with your complete chip
2. A frame schematic with the pads replaced to be the correct type and all the signals included. I put the signals with the names from your core on the external **pad** connections, and append a **_i** to the internal signals that go from core to frame.
3. A symbol for that frame.
4. A layout with the pads replaced to be the correct types and with shape pins added to match the pins of the schematic/symbol
5. A **wholething** schematic view that includes the core and the frame connected together. This is the schematic that **ccar** uses to know how the core should be routed to the frame.

The process of routing the core to the frame is essentially the same one as described in the general **ccar** discussion in Section 11.1. Start by opening the **wholechip** schematic in **Composer** and use **Tools** → **Design Synthesis** → **Layout XL** to launch the **Virtuoso-XL** tool. Now use **Design** → **Gen from Source** in the **Virtuoso-XL** window to generate a new layout view of the **wholething** schematic.

In the **Gen from Source** dialog box you have a chance to select which pins you want to create. Pins are created for signals that are routed from this **layout** view to an external pin so that the resulting routed **layout** can be used hierarchically. In this case, the external connections are all to pads. In other words, there are *no* pins that you want created because there are no pins that are exported outside this cell. All the connections are made internal to the **wholechip** schematic. All the connections are made to connect the core to the frame. In the **Gen from Source** dialog box select all the pins (which should be only external pads, not the internal **_i** signals) and turn off the **create** box. In the dialog box select all pins, turn off the **create** button, and press **Update.** Also un-select the **Generate I/O Pins** button in the **Layout Generation** section. This way none of those pins will be created.

Now you can place the frame and core layouts in **Virtuoso-XL**. Grab the frame and put it inside the purple **prBoundary**. You'll probably want to expand the view so that you can see inside the cells. Now move the core to be inside the pad frame. When you do the placement in **Virtuoso-XL** you should see the lines that connect the core to the frame. See Figure 11.19 for an example. You can see that I didn't do a particularly good job of placing the pad frame pins in convenient locations for the core. For this example that's fine because there's lots of routing room. For your chip you might want to think about this a little more carefully.

Check to make sure that there aren't any stray pads (like **vdd!** and **gnd!**) down in the lower left just below the **prBoundary**. If there are, delete them! You don't want icc to route anything but the signal pads. If you didn't manage to get all the signal pins turned off in the **Gen from-Source** step, here's another chance to get rid of them.

Before you send this to the router, you need to connect **vdd!** and **gnd!**. Look at the **pad_vdd** and **pad_gnd** pads. There are big **metal1** connection points for **vdd** and **gnd** in the middle of the edge of the pad cell. The power tabs should be 28.8u wide. These are what you use to make a connection between **vdd** and **gnd** from the pads to your core. Use the largest wires that make sense, and remember to put arrays of contacts if you have to switch layers to connect large wires. DRC is a good idea at this point before you send things to the router! there's a view of the core placed inside the pad frame just before I send it to the **ccar** router in Figure 11.20.

Now that I have a placed and power-routed core and frame layout, I can send it to the **ccar** router with the **Routing → Export to Router** command as in Section 11.1. The initial view in the **ccar** router is shown in Figure 11.21. You can now perform the routing (set the costs/taxes, set local layer direction, global route, detail route, cleanup, and remove notches). The result (seen after importing back into **Virtuoso** is seen in Figure 11.22. This layout should then be processed for DRC, Extract, and LVS to make

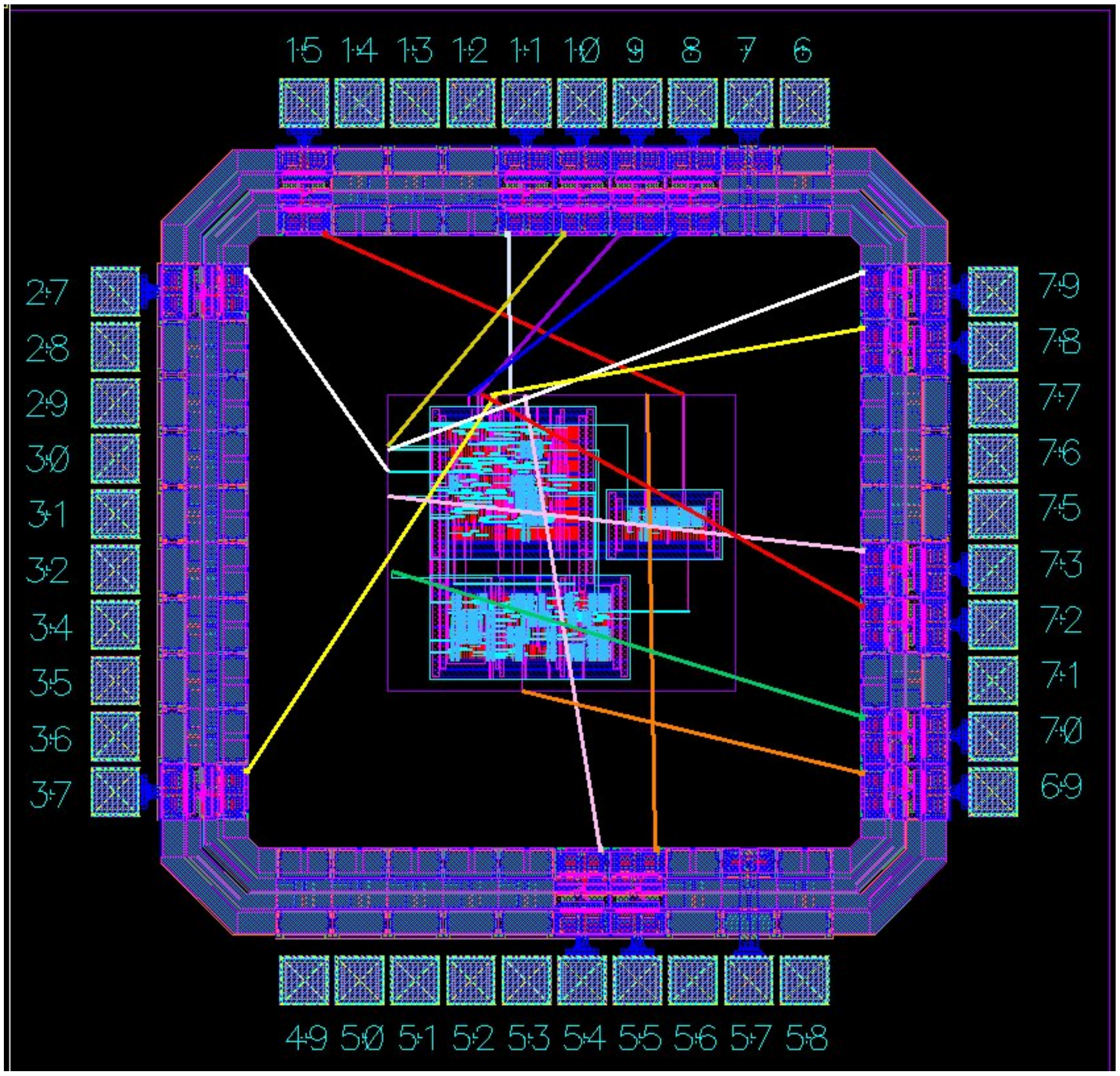


Figure 11.19: Frame and core placed in **Virtuoso-XL**

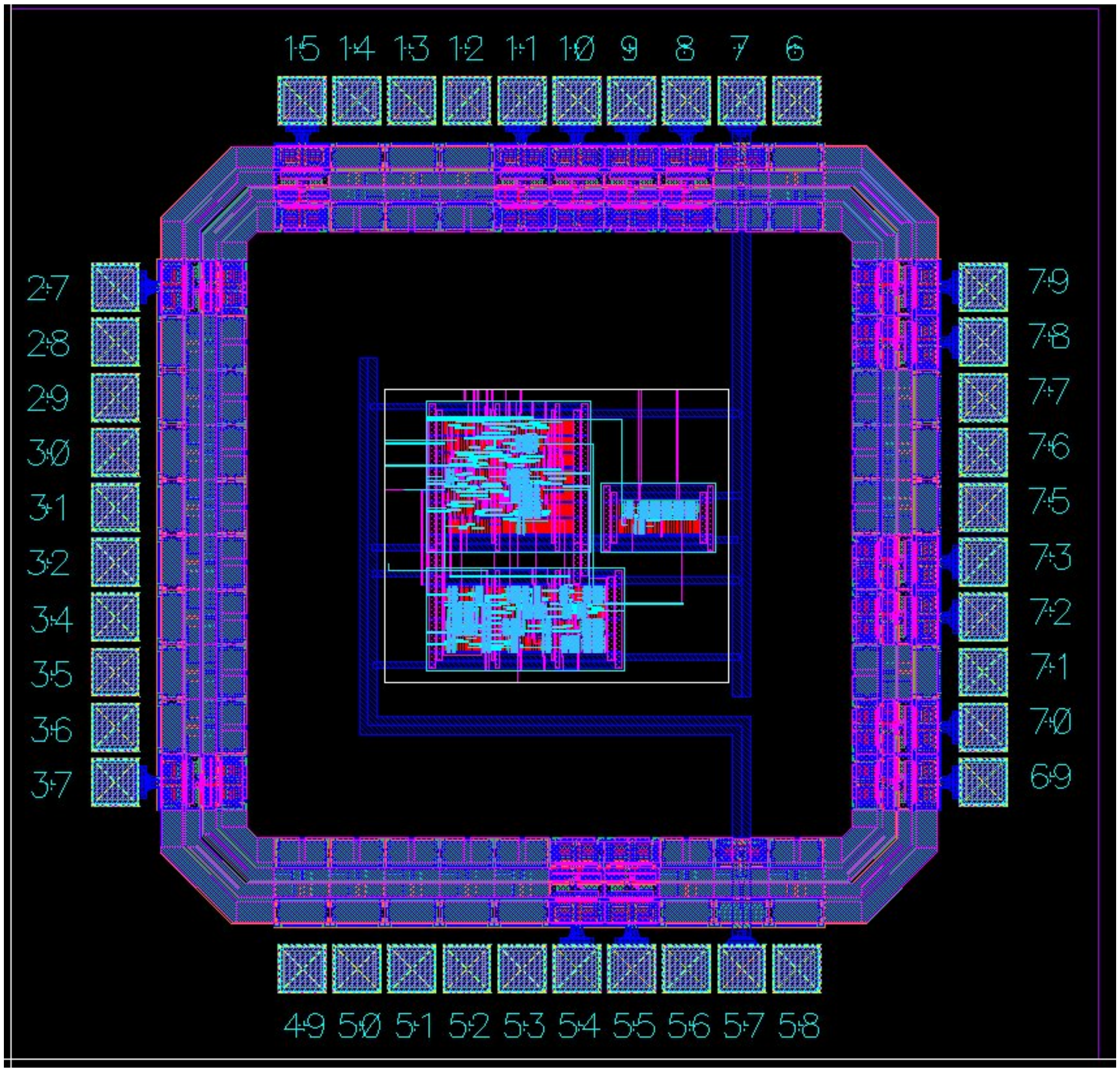


Figure 11.20: Frame and core placed in **Virtuoso-XL** and with **vdd** and **gnd** routing completed

sure that everything is still correct. If it is, you have a complete chip! You can also (if you want to try something that will really bring large computers to their knees) generate an **analog_extracted** view of the complete chip and use **Spectre** to run analog simulations of the entire chip including the pad frame.

11.3 Module Routing with SOC Encounter

Examples of hierarchical design using SOC - take a module that was placed and routed with SOC. Generate the .lib file from SOC for timing. Then read .def back to icfb, and use abstract to generate a .lef file. This .lef file can be used to make this placed and routed pice a hard macro or fixed block in a hierarchical route in Encounter. There are a number of steps to make this all work...

11.3.1 Liberty File Generation with SOC

How to get the timing information exported from SOC

11.3.2 Abstract and LEF Generation for Hard Macros

How to get the hard macros in the right form for hierarchical place and route

11.3.3 Block Placement and Routing in the SOC Flow

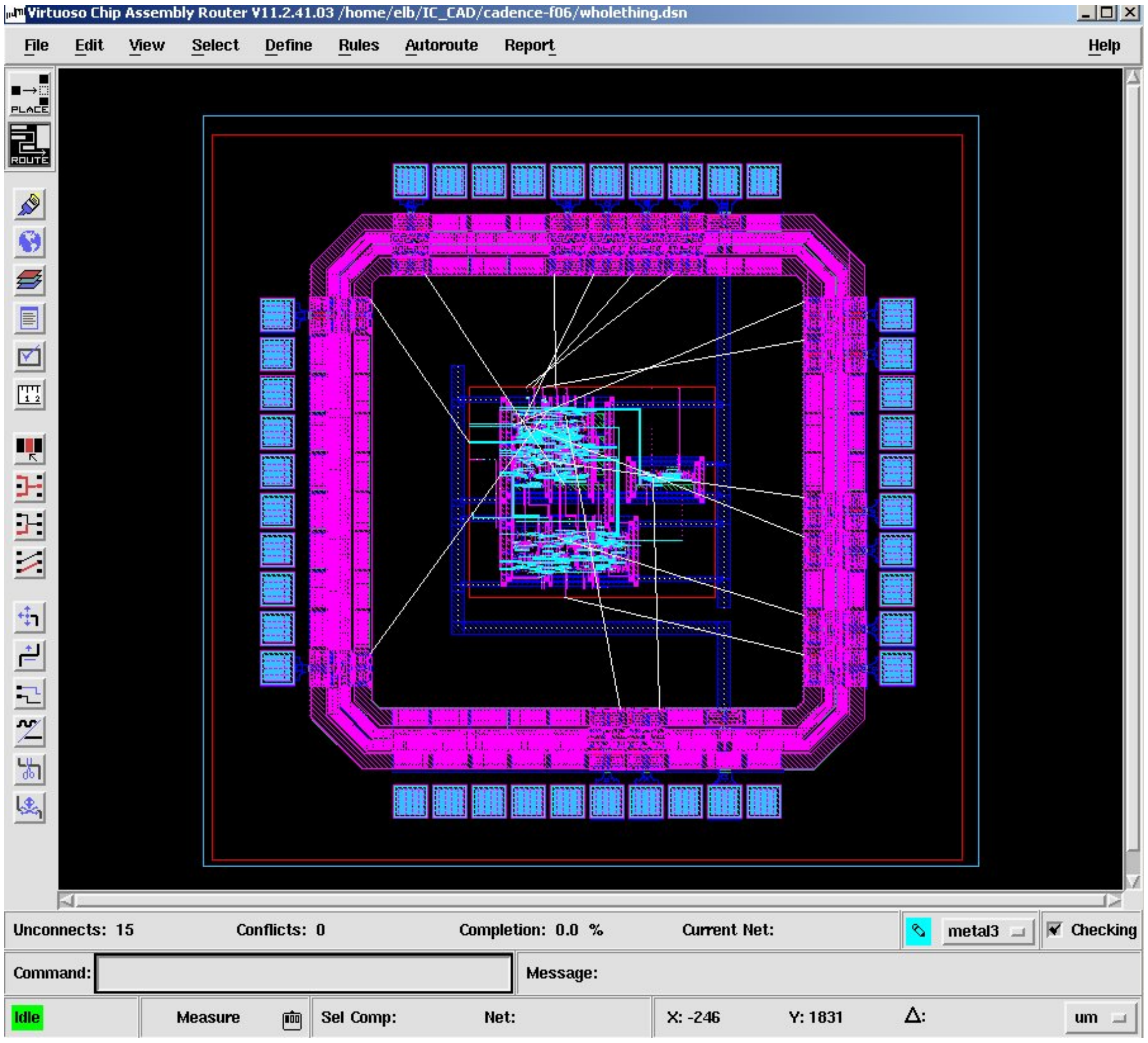
How to get those blocks placed with standard cells flowing around them in SOC

11.4 Core to Pad Frame Routing with SOC Encounter

Example of using SOC with a pad frame. The pad frame is specified using pad descriptions in the structural Verilog file, and the locations defined in the .io file.

The sequence should be:

1. Use **Abstract** to generate abstract views of the pad cells making sure to place them in the **IO** bin. Generate a **.lef** file with these cells.


Virtuoso Chip Assembly Router V11.2.41.03 /home/elb/IC_CAD/cadence-f06/wholething.dsn
File
Edit
View
Select
Define
Rules
Autoroute
Report
Help
PLACE
ROUTE
metal3
Checking
Idle
Measure
um

Unconnects: 15 Conflicts: 0 Completion: 0.0 % Current Net:

Command: Message:

Sel Comp: Net: X: -246 Y: 1831 Δ:

Figure 11.21: Frame and core before routing in **ccar**

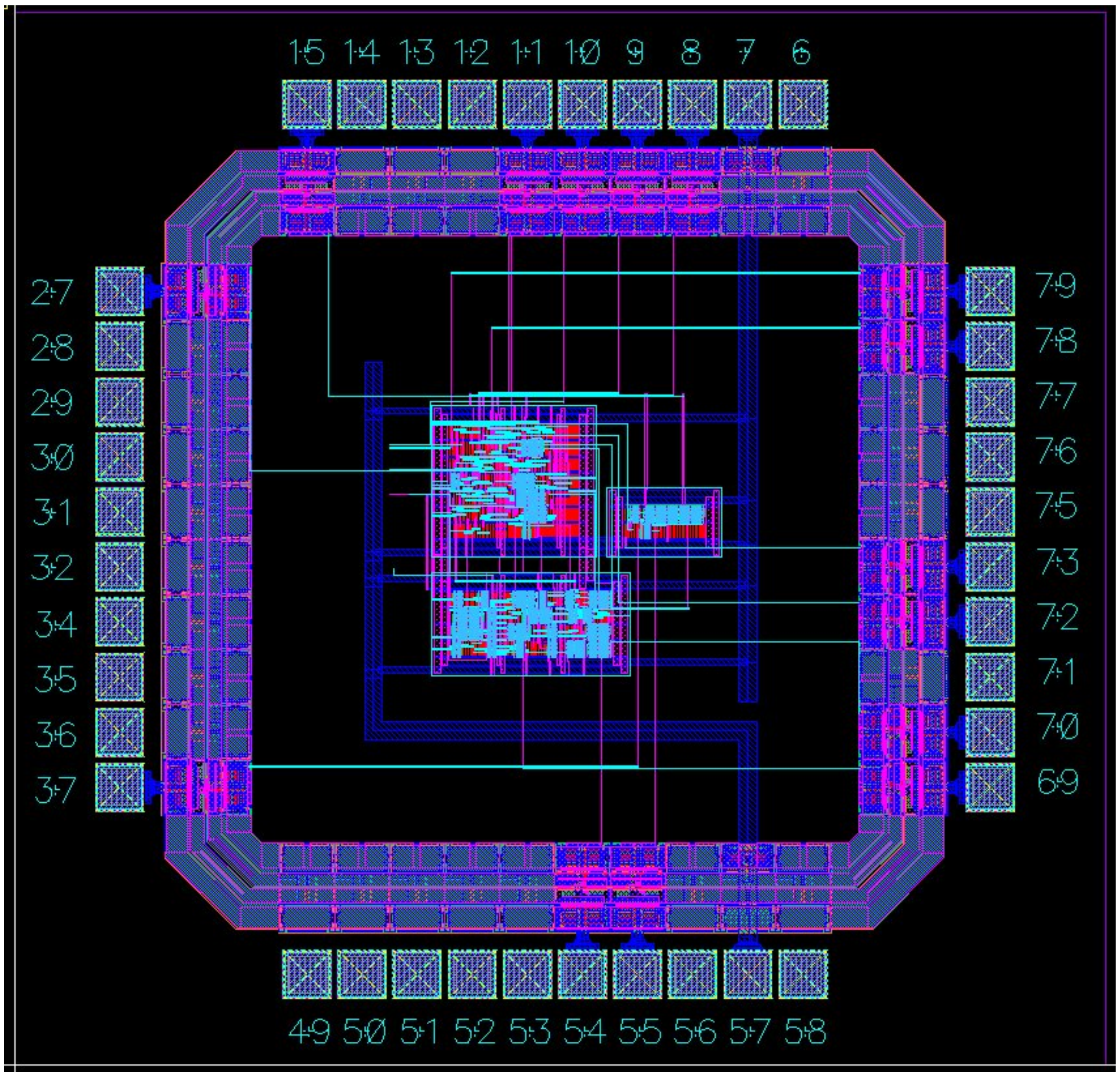


Figure 11.22: Frame and core after routing in **Virtuoso**

2. Use **SignalStorm** to characterize the pad cells and generate **.lib** timing information for the pads. This will also generate **.v** behavioral information about the pads.

3. Use a placement file <**filename**>**.io** to describe the placement of pads in the pad rings. This file describes which pads go on which sides of the pad ring, and the spacing between the pad cells.

4. Make a new structural Verilog file that contains an instance of your finished core cell and instances of all the pad cells. The names of the pad cells in the structural Verilog must match the names of the pad cells in the <**filename**>**.io** IO placement file. Connect the core to the pads in this structural file by using internal wires that connect between the core module and the pad cells.

5. Read this file into **SOC Encounter**. The pad cells will be placed in the *IO site* and the core will be placed in the *core site* (as described in the **LEF** technology header) because that's how the macros will be defined in the **.lef** files.

6. Use the floorplanning process in **SOC Encounter** to make sure that the pad cells are in the right places, the right orientation, and that the outside dimensions of the chip are what you want. Also make sure that the core is the right size and in the right place.

7. Continue with the regular **SOC Encounter** flow as described in Chapter 10.

11.4.1 Pad Frame Definition

Define the pad frame with a combination of structural Verilog and .io files

11.5 Final GDS Generation

Final Stream (GDS) output - talk about map files, also about metal/poly fill (which can be done in SOC) and final DRC/LVS on the GDS file