

# Chapter 1

## Introduction

THE DESIGN process for digital integrated circuits is extremely complex. Unfortunately, the Electronic Design Automation (EDA) and Computed Aided Design (CAD) tools that are essential to this design process are also extremely complex. Finding a combination of tools and a way of using those tools that works for a particular design is known as finding a “tool path” for that project. This book will introduce one path through these complex tools that can be used to design digital integrated circuits. The tool path described in this book uses tools from Cadence ([www.cadence.com](http://www.cadence.com)) and Synopsys ([www.synopsys.com](http://www.synopsys.com)) that are available to university students through special arrangements that these companies make with universities. Tool bundles that would normally cost hundreds of thousands or even millions of dollars if purchased directly from the companies are made available through “university programs” at small fixed fees.

In order to justify these small fees, however, the EDA companies typically reduce their costs by offering very limited support for these tools to university customers. In an industrial setting there would likely be an entire CAD support department whose job it is to get the tools running and to develop tool flows for projects within the company. Few universities, however, can afford that type of support for their CAD tools. That leaves universities to sink or swim with these complex tools making it all the more important to find a usable tool path through the confusing labyrinth of the tool suites. This book is an attempt to codify at least one working tool path for a Cadence/Synopsys flow that students and researchers can use to design digital integrated circuits. It includes tutorials for specific tools, and an extended example of how these tools are used together to design a simple integrated circuit.

In addition to the CAD tools from Cadence and Synopsys, The tutorials assume that you have some sort of CMOS standard cell library avail-

*In this book I'll call the tools “CAD tools” and the companies “EDA companies”*

*Instructions for installing the CAD tools can be found in the appendices*

*Details about these  
libraries can also be  
found in the appendices*

able. The specific examples in this book will use a cell library developed at the University of Utah specifically for our VLSI classes known as the UofU\_Digital library. This library, and the technology information available through the NCSU CDK (North Carolina State University Cadence Design Kit), are freely available from the University of Utah and North Carolina State University respectively. If you don't have these libraries you should be able to follow most of the tutorials with your own library, but you must have a library of some sort.

## 1.1 Cad Tool Flows

*CAD tools typically do  
not include any  
technology or cell data.  
This data comes directly  
from the chip and cell  
vendors and contains  
information specific to  
their technologies.*

The general tool flow described here uses CMOS standard cells with automatic place and route to design the chip, but also includes details of how to design custom cells as layout and add those cells to a library. This custom portion of the flow could, of course, be used to design a fully custom chip. It can also be used to design your own cell library. Designing a cell library involved not only designing the individual cells, but characterizing those cells in terms of their delay in the face of different output loads and input slopes, and codifying that behavior in a way that the synthesis tools can use. The cells also must have their physical parameters characterized so that the place and route tools have enough information to assemble them into a chip. Finally, simulation at a variety of levels of detail and timing accuracy is essential if a functional chip is to result from this process.

This entire tool flow will use a large number of tools from both Cadence and Synopsys, a large number of different file formats and conversion programs, and a lot of different ways of thinking about circuits and systems. This is inevitable in a task as complex as designing a large integrated circuit, but it can be intimidating. One ramification of the type of complexity inherent in VLSI design is that the tools, designed as they are to handle very large collections of cells and transistors, aren't much simpler to use on just 4 transistors than they are on 4,000, 400,000, or 4,000,000 transistor designs. It is not easy to simply start small and add features. One must, in some ways, do it all right from the start which makes the learning curve quite steep. There are lots of pieces of the flow that must be available right from the start which can be overwhelming at first. Hopefully by breaking the tool flow into individual tool tutorials, and with detailed walk-through tutorials with lots of screen shots of what you should see on the screen, this can be made less intimidating.

Of course, as with any tool with a steep learning curve, once you've made it up the steep part of the curve, you may not want to refer back to the level of detail contained in the tutorial descriptions. For that stage of the process I've included slimmed down "highlights" versions of the tool

*File types for the  
complete flow are  
described as they are  
used in the flow and  
documented in the  
appendices. In addition  
to Cadence database  
files, they include .lib,  
.db, .lef, .gds, .sdf, .def,  
.v, .sdc, and .tcl files.*

instructions in the appendices of this volume. If you need to refer back to a tool that you've used before but haven't used for a while, you may just need to glance at the highlights in the appendices rather than walk through the entire tutorial again.

*I need at least one figure of the whole tool flow here, and perhaps individual pictures of flows for different purposes. For example, a flow for cell design, a flow for standard cell from Verilog descriptions, and a general flow.*

### **1.1.1 Custom VLSI and Cell Design Flow**

This is a tool flow for designing custom VLSI systems where the design goes down to the circuit fabrication layout. This flow starts with transistor schematics at the front end and uses custom layout to design portions of the system. It is used for designing cell librarys as well as for designing performance-critical portions of larger circuits where individual design of the transistor-level circuits is desired. The front end for this flow is transistor-level schematics in **Composer**. These schematics may be simulated at a functional level using Verilog simulators like **Verilog-XL** and **NC\_Verilog**, and with a detailed analog simulator like **Spectre**. The back end is composite layout using **Virtuoso** and more detailed simulation using analog simulators.

If the final target is a cell library then the cells can be characterized for performance using a simulator like **Spectre** or by using a library characterizer tool like **Signalstorm**. These chacterizations are required if you would like to use these cells with a synthesis system later on. Abstract view can also be generated so that the cells can be used with a place and route system.

### **1.1.2 Hierarchical Cell/Block ASIC Flow**

This is a tool flow for system level design using a CMOS standard cell library. The library may be a commercial library or it may be one that you design yourself, or a combination of the two. The front end can be schematics designed using cells from these libraries, or Verilog code. If the system description is in structural Verilog code which is set of instantiations of standard cells in Verilog, then this can be used directly as the front-end description. If the Verilog is behavioral Verilog, then a synthesis step using Synopsys **dc\_shell**, **design\_vision** or **module compiler**, or Cadence **BuildGates** can synthesize the behavioral description into a Verilog structural description.

These descriptions, whether structural, behavioral, or a combination of both, can be simulated for functionality using **Verilog-XL** or **NC\_Verilog**.

These simulations may use a zero-delay, unit-delay, or extracted delay model. The extracted delays come from the synthesis systems which extract timings based on the cell characterizations.

The back end to the ASIC flow uses SOC Encounter to place and route the structural file into a full chip. This description may be extracted again to get timings that include wiring delays, or the timing can be analyzed using a static timing analyzer like Synopsys PrimeTime. The system can also be simulated in mixed-timing mode where parts of the circuit are simulated at a switch level using a Verilog simulator and parts of the circuit are simulated at a detailed level using an analog simulator like Spectre. The final result is a gds (also known as stream) file that can be sent to a fabrication service such as MOSIS to have the chips built.

Of course, the tool flows described here only scratches the surface of what the tools can do! Please feel free to explore, press on likely looking buttons, and read the manuals to explore the tools further. If you discover new and wonderful things that the tools can do, document those additions to the flow and let me know and I'll include them in subsequent releases of this manual.

## 1.2 What this Manual is and isn't

This manual includes walk-through tutorials for a number of tools from Cadence and Synopsys, and description of how to combine those tools into a working tool flow for VLSI design. It is *not* a manual on the VLSI design process itself! There are many fine textbooks about VLSI design available. This is a “lab manual” that is meant to go along with those textbooks and describe the nuts and bolts of working with the CAD tools. I will assume that you either already understand general VLSI design, or are learning that as you proceed through the tutorials contained in this manual.

### Bugs in the Tools?

Before we dive into the tutorials, here's a quick word about tool bugs. These tools are complex, but so are the systems that you can design with them. They also feel very cumbersome and buggy at times, and at times they are! However, even with the inevitable bugs that creep into tools like this, I encourage you to follow the tutorials carefully and resist the temptation to blame a tool bug each time you run into a problem! I've found in teaching courses with these tools for years that it is almost 100% certainly that if you're having trouble with a tool in a class setting, that it's something that you've done or some quirk of your data rather than a bug in the tool. It's

amazing how subtle (or sometimes how obvious!) the differences can be in what you're doing and what the procedure specifies. Relax, take a deep breath, and think carefully about what's going on and what might cause it. Read the error messages carefully. Occasionally there is real information in the error message! Try explaining things to a fellow student. Often in the process of explaining what you're doing you'll see what's going on. Let someone else look at it. Let your first instinct be to try to figure out what's going on, not to blame the tool! If the tool turns out to be the problem, at least you will have exhausted the more likely causes of the problem first before you discover this.

## 1.3 Typographical Conventions

Finally, a word about typographical conventions. I will try to stick to these, but don't promise perfect adherence to these conventions! In general:

- I'll try to use `boxed, fixed width font` for any text that you should type in to the system. This, hopefully, will look a little like the fixed-width font you'll see on your screen while you're typing. So if you are supposed to type in a command like `cad-ncsu` it will look like that.
- I'll try to use **bold face** for things that you should see on the screen or in windows that the tools pop up. So if you should see **Create Library** in the title bar of the window, it will look like that in the text.
- I'll use *slanted text* in the marginal notes. These are little points of interest that are ancillary or parenthetical to the main text. *This is a margin note*
- I'll use a **non-serifed face** to give the names of the tools that we're working with. Note that the name of the tool, like **Composer**, is seldom the name of the executable program that you run to get to that program. For those, refer back to the typed commands like `cad-ncsu`.



# Chapter 2

## Cadence ICFB

**C**ADECNE is a large company that offers a dizzying array of software for Electronic Design Automation (EDA) or Computer Aided Design (CAD) applications. Cadence CAD software is generally targeted at the design of electrical circuits, both digital and analog, and extending from extremely low-level VLSI design to the design of circuit boards for large systems. This book is primarily interested in digital integrated circuit (IC) design so we'll look primarily at those tools from the Cadence suite.

*The custom design tutorials are a good starting point for custom analog IC design too*

### 2.1 Cadence Design Framework

Many of the digital IC design tools from Cadence are grouped under a framework called **Design Framework II** (**dfII**). The **dfII** environment integrates a variety of design activities including schematic capture (**Composer**), simulation (**Verilog-XL** or **NC\_Verilog**), layout design (**Virtuoso** and **Virtuoso-XL**), design rule checking (**DRC**) (**Diva** and **Assura**), layout versus schematic checking (**LVS**) (**Diva** and **Assura**), and abstract generation for standard cell generation (**Abstract**). These are all individual programs that perform a piece of the digital IC design process, but are all accessible (to a greater or lesser extent) through the **dfII** framework and the **dfII** user interface. Note that many of these programs were developed by separate companies that have been acquired by Cadence and folded into the **dfII** framework after that acquisition. Thus, some integrate better than others!

As we'll see, though, there are some pieces of the Cadence tool flow that are not linked into the **dfII** framework. Most notably place and route of standard cells with **SOC Encounter**, connection of large blocks with **ICC Chip Assembly Router** (**CAR**) and Verilog synthesis with **BuildGates** are done in separate programs with separate interfaces.

However, we'll start with the dfll tools in this tool flow, so we'll need to start up the dfll framework. The executable in the Cadence tool suite that starts up this framework is called icfb which stands for Integrated Circuit Front to Back design. If you were to set up your search path so that the Cadence tools were on your path, and execute the `icfb` command you would see the dfll framework start up.

Unfortunately, this wouldn't help you much! It turns out that having the tool framework is only half the battle. You also need detailed technology information about the devices you want to use for your design. This detailed design information includes technology information about the IC process that you are using and libraries of transistors, gates, or larger modules that you can use to build your circuits. This information includes many files of detailed (and somewhat inscrutable) information, and does not come from Cadence. Instead, it comes from the vendor of the IC process and from the vendor of the gate and module cells that you are using in your design. This collection design information is typically called a "Cadence Design Kit" or CDK.

We're using  
NCSU CDK v1.5

For this book we will use technology information for IC processes supported by the MOSIS chip fabrication service. This information has been assembled into a CDK by the good folks at North Carolina State University (NCSU). The NCSU CDK has detailed technology information for all the processes currently offered through MOSIS. These processes are available either in "vendor" rules which have the actual specifics of the technology as offered by the vendor, or through abstracted rules known as Scalable CMOS or SCOMS rules. The SCOMS are scalable in the sense that a design done in the SCOMS rules should, theoretically, be useable in any of the MOSIS processes simply by changing a scaling parameter. That means that the SCOMS rules are a little conservative compared to some of the vendor rules because they have to work for all the different vendors.

We're using the  
SCOMS V8.0 rules.

Of course, it's not quite that simple because as design features get smaller and smaller the IC structures don't scale at the same rate. But, it works pretty well. To handle the differences required by smaller geometry processes MOSIS has a number of modifiers to the SCOMS rules (SCOMS for "generic" SCOMS, SCOMS\_SUBM for submicron processes, and SC-MOS\_DEEP for even smaller processes). For this class we'll be using the SCOMS\_SUBM rules which will then be fabricated on the AMI C5N  $0.5\mu$  CMOS process.

But, that's getting ahead of ourselves a little bit. The important thing for now is that without the NCSU CDK, we won't have any technology information to work with. So, instead of starting up icfb directly, we'll start it up with the NCSU CDK already loaded. This will happen by calling Cadence from a script that we've written instead of calling the tool directly. This

script will start a new shell, set a bunch of required environment variables, and call the icfb tool with the right switches set. Other tools for the rest of this book will use similar scripts.

## 2.2 Starting Cadence

Before you start using cadence you need to complete the following steps:

First make a directory from which to run Cadence. This is important so that all of Cadence's files end up in a consistent location. It's also nice to have all of Cadence's setup and data files in a subdirectory and not clogging up your home directory. I recommend making an IC\_CAD directory and then under that making a cadence directory. Later on we'll add to that by making separate directories for the other IC tools like Synopsys dc\_shell, module complier, SOC Encounter and so on under that IC\_CAD directory.

```
cd
mkdir IC_CAD
mkdir IC_CAD/cadence
```

Now it's handy to set a few environment variables. In particular you want to set your UNIX search path to include the directory that has the startup scripts for the CAD tools. You also need to set an environment variable that points to a location for class-specific modifications to the general Cadence configuration files. I recommend that you put these commands in your .cshrc or .tshrc file so you won't have to retype them each time you start a shell. If you're using bash you'll have to adjust the syntax slightly. In general you want to set your path to point to where the tool startup scripts live, and set your **LOCAL\_CADSETUP** variable to point to the directory that holds the local information. These locations are site- and semester-specific so check with your instructor for details of your system's organization!

```
set path = ($path <path-to-tool-scripts>
setenv LOCAL_CADSETUP <path-to-local-setup-info>
```

As an example, these commands might be something like the following (again, check with your instructor or tool administrator for your local direcory information):

```
set path = ($path /uusoc/facility/cad_common/local/bin/F07)
setenv LOCAL_CADSETUP /uusoc/facility/cad_common/local/class/6710
```

Finally, you need to copy one Cadence init file from the NCSU CSK directory so that things get initialized correctly. The file is called **.cdsinit**

*CAD tools can generate a lot of temporary and auxiliary files!*

*All the Cadence and Synopsys CAD tools run on Solaris or Linux so if you don't have a good grasp of basic UNIX commands, now's the time to go learn them!*

(note the initial dot!). You can put it in your \$HOME directory so that you'll always get that init file, or you can put it in the directory from which you start Cadence if you think you might ever want to start Cadence from a different directory for different projects or classes with a different .cdsinit file.

I recommend making this a symbolic link so that if the system-wide .cdsinit file is updated you'll see the new version automatically. You only need to do this once so that a link to the bbb.cdsinit file is in place before you start **Cadence**. Again, the specific installation paths are site- and semester-specific so be sure to check for the correct path!

```
ln -s <path-to-NCSU-CDK>/cdsinit $HOME  
or  
cd $HOME/IC_CAD/cadence  
ln -s <path-to-NCSU-CDK>/cdsinit .
```

As an example, the <path-to-NCSU-CDK> might be /uusoc/facility/cad\_common/NCSU/CDK-F07.

Now that you have your own cadence directory (called \$HOME/IC\_CAD/cadence if you've followed the directions up to this point), set your path, and linked the NCSU .cdsinit file either to \$HOME or to \$HOME/IC\_CAD/cadence you're ready to start Cadence icfb with the NCSU CDK. Before starting the tool connect to your \$HOME/IC\_CAD/cadence directory (or where ever you wish to start Cadence from) first.

Start Cadence with the command:

```
cad-ncsu
```

We're using dfll from the IC v5.1.41 release

Of course, once you set this all up once, you should be able to jump right to the `cad-ncsu` step the next time you want to start Cadence.

You should see two windows once things get started up. The first is the **Command Interpreter Window** or CIW. It's shown in Figure 2.1. The other is the **Library Manager** shown in Figure 2.2. The CIW is the main command interface for all the dfll tools. In practice you will probably not type commands into this window. Instead you'll use interfaces in each of the tools themselves. However, because most of the tools put their diagnostic log information into the CIW, you will refer back to it often. Also, there are some things that just have to be done from this window. For now, just make sure that your CIW looks something like the one in Figure 2.1.

Cadence also keeps the log information in a CDS.log file which it puts in your \$HOME directory

The **Library Manager** is a general interface to all the libraries and cells views that you'll use in dfll. Cells in dfll are individual circuits that you want to design separately. In dfll there is a notion of a "cell view" which

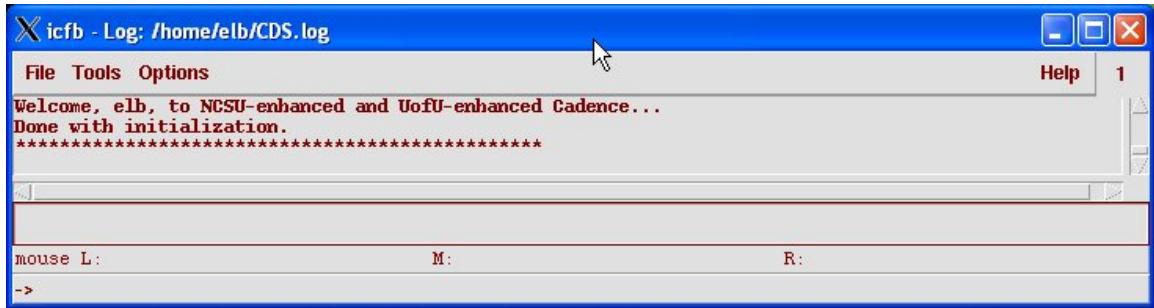


Figure 2.1: Command Interperter Window (CIW) for `cad-ncsu`

means that you can look at a cell in a number of different ways (in different views). For example, you might have a shematic view that shows the cell in terms of its components in a graphical schematic, or you might have a Verilog description of the cell as behavioral Verilog code. Both of these cell views can exist at the same time and are just alternate ways of looking at the same cell. The cell views that we'll eventually end up using in this tool flow are the following:

**schematic:** This view is a graphical schematic showing a cell as an interconnection of basic components, or as hierarchically defined components.

**symbol:** This view is a symbolic view of the cell that can be used to place an instance of this cell in another schematic. This is the primary mechanism for generaing hierarchy in a schematic.

**cmos\_sch:** This is a schematic that consists of CMOS transistors. A `cmos_sch` view corresponds to a cell that is completely contained in a single standard cell. That is, it is a leaf-cell in the standard cell hierarchy that corseponds to a cell in an existing library. It's important in some tool steps to differentiate the schematics that should be expanded by the netlisting process and the leaf cells where the netlisting should stop. That's the purpose of the `cmos_sch` view.

**extracted:** This view is generated by the circuit extraction process in the Cadence tools. It contains an extracted electrical netlist of the cell that the simulators can use to understand the electrical behavior of the cell.

**analog-extracted:** This view is generated from the extracted view and contains a little extra information for the analog simulator.

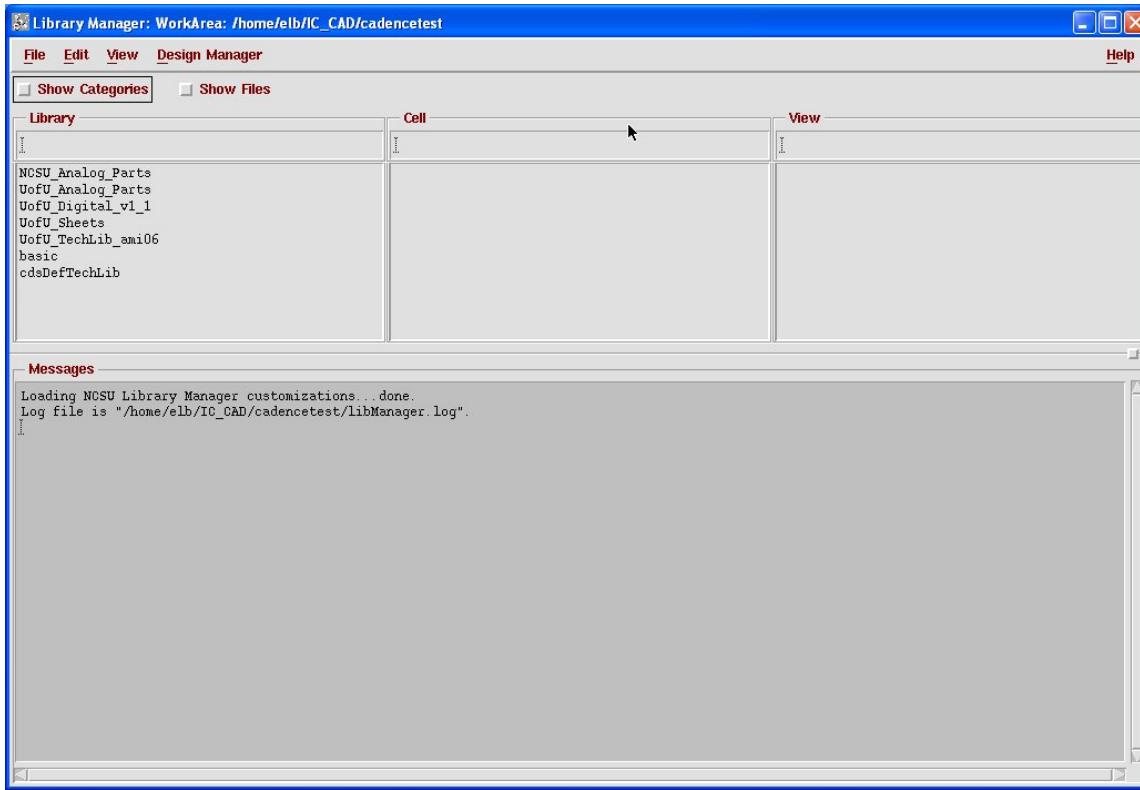


Figure 2.2: Library Manager window in `cad-ncsu`

**behavioral:** This view is Verilog code that describes the behavior of the cell.

**layout:** This view contains the composite layout information for a cmos\_sch cell. This is the graphical information that the IC fabrication service uses to fabricate the cell on the silicon.

**abstract:** This layer takes the layout and extracts only the information that the place and route software needs to do the placement and routing. That is, it needs to know the physical dimensions of the cell, the connection points and layers, and any obstructions for the metal routing layers, but it doesn't need to know anything about the transistor layers. This view will be generated by the **abstract** process.

**functional:** This view is reserved for behavioral descriptions of CMOS transistors. It's used for a similar reason to the cmos\_sch view: it lets the netlister know when it has hit a transistor. You won't need to create these views unless you're adding new transistor models to a library.

**spectre:** This view is used by the analog circuit netlister to generate an input file for the Spectre analog simulator. You won't need to create this view unless you're adding new transistor models to a library. There are a number of other similar views for other simulators that you also don't need to worry about.

A “library” is a collection of cells that are grouped together for some reason (being part of the same project, or part the same set of standard cells, for example). Libraries also have technology information attached to them so that the cells in the library refer to a consistent set of technology data. This technology information is linked rather than copied so that when updates are made on the technology, all libraries with that technology attached will see the updates. For example, all the standard gates cells that you'll be using (until you make your own!) are grouped into a library called **UofU\_Digital\_v1\_1**. You will create libraries for each of your designs so that you can keep the design data separate for different projects. Think of libraries as directories that collect design data together for a specific design. You could throw all your stuff into one directory, but it would be easier to find and use if you separate different designs into different libraries.

You should see a bunch of libraries already listed in the Library Manager. If you scroll around you should be able to see the following libraries if you are using the NCSU CDK:

**NCSU\_TechLib\_xxx:** These are technology libraries for each of the MOSIS processes. The “xxx” will be filled in with information about which MOSIS process is being described (**ami06** for the AMI C5N  $.6\mu$  process, for example). We won't use these directly, and depending on how Cadence is set up for your class you might not see these at all. If you're not using the UofU packages, then you'll probably see all of these.

**NCSU\_Analog\_Parts:** This library contains components (transistors, resistors, capacitors, etc.) that you'll use for transistor-level design, and also some components for circuit-level simulation using spectre (in the **Affirma** analog circuit design environment). The switch-level transistor models in this library have zero delay for simulation.

**NCSU\_Digital\_Parts:** This library contains a variety of Boolean logic gates that you can use for gate-level design. Note that these gates do **not** have layout or place and route views so they **can not** be used for actually building chips! They are typically used in classes just for the initial “learn about the schematic capture tool” assignments.

**basic:** This is the Cadence built-in library which you won't use directly. It has basic components from which other parts are built.

*The UofU\_Digital library uses the \_v1\_1 syntax to indicate version 1.1 of the library. Cadence doesn't like dots in cell names!*

*If you're using a different CDK or PDK, you'll see different libraries in the default list*

**cdsDefTechLib:** A generic Cadence technology that we won't use.

If you're using a different CDK (Cadence Design Kit) or PDK (Process Design Kit) you'll see different libraries. Also, you may see additional libraries for local additions or modifications to the default libraries. For example, at the University of Utah you'll see:

**UofU\_Analog\_Parts:** This is a library with copies of the transistor components from the **NCSU\_Analog\_Library**, but these transistors have 0.1ns of delay for switch level simulation.

**UofU\_TechLib\_ami06:** This is a technology library for AMI C5N  $0.5\mu$  library using the **SCMOS\_SUBM** rules that we'll use in the tutorials. It's based on the NCSU technology library for this process, but has some local tweaks that make it a little more friendly to this flow.

**UofU\_Sheets:** This library has graphics for schematic sheet borders that are specific to the University of Utah.

*If you look at the cells in UofU\_Digital\_v1\_1 you should see that each of them has a number of different cell views as defined previously*

**UofU\_Digital\_v1\_1:** This is a library of standard cells developed at the University of Utah for VLSI classes. It has the **UofU\_TechLib\_ami06** technology attached to it so it can be used with the AMI C5N  $0.6\mu$  CMOS process through the **SCMOS\_SUBM** design rules from MOSIS. This library will be enabled for viewing when it's needed.

**UofU\_Gates\_v1\_1:** This is a library with only the gate (cmos\_sch, behavioral, and symbol) views of the cells in the **UofU\_Digital\_v1\_1** library. It's used for initial assignments so that students can use the gates without seeing the other cell views.

**UofU\_Pads:** This library (enabled for viewing when it's needed) contains I/O pad cells to be used with the AMI C5N CMOS process.

Unfortunately, you'll have to keep very careful track of when to use components out of each of these libraries. Some have very specific uses. The only way to handle this is just to pay attention and keep track!

Now that you've started Cadence using the `cad-ncsu` script, we can move on to using the individual EDA tools in the **dfll** suite...

## Chapter 3

# Composer Schematic Capture

**C**OMPOSER is the schematic capture tool that is bundled with the dfII (Design Framework II) tool set. It is a full-featured schematic capture tool that we'll use for designing transistor level schematics for small cells, gate level schematics for larger circuits, and schematics containing a mix of gates and Verilog code for more complex circuits. In that case some of the components in the schematic will contain transistors at the lowest level, and some will contain Verilog code. Because the simulators that are used in conjunction with Composer are all Verilog simulators, these mixed schematics can be simulated using the same simulators used by schematics with only gates or transistors.

I find schematics extremely useful for all levels of design. Even for designs that are done completely in Verilog code I find that connecting the Verilog components in a schematic often makes things easier to understand than large pieces of code where connections are made with large argument lists and named wires. Your mileage may vary, of course.

Composer has connections to all sorts of other tools in the dfII tool suite, and to other tool suites. We'll look at all of them in future chapters.

- Composer is integrated with the Verilog-XL and NC\_Verilog simulators so that you can automatically export a schematic to a simulator. The Composer/Verilog integration will take your schematic and generate a Verilog netlist for simulation, and also build a simple testbench wrapper as a Verilog file that you can modify with your own testing commands. We'll see how that works in the chapter on Verilog simulation.
- There is also an interface that can take a schematic and convert that schematic to the Verilog structural file for input to a tool that uses that type of input. Synopsys dc\_shell for synthesis and Cadence SOC

*Although the schematic tool is called Composer in the documentation, it's called Virtuoso Schematic Editing in the window title. Virtuoso is also the specific name given to the layout editor in dfII. They're both part of the dfII Virtuoso tool suite I guess.*

*Composer is a part of the IC v5.1.41 tools*

*A file that captures the component and connection information for a circuit is called a “netlist,” and the process of generating that file is called “netlisting.”*

Encounter for place and route are just two of the possible tools that the structural Verilog file can be used with. This interface is known as the Cadence Synopsys Interface, or CSI, but it is a general way to convert a schematic to a structural netlist.

- Composer has a connection with the Cadence Virtuoso-XL layout tool so that the designer can see the connection between the layout and the schematic. This can be used as a guide for producing layout based on a schematic using Virtuoso-XL, and is also a mechanism for specifying the connectivity of a circuit when using the ICC Chip Assembly Router to assemble large chip pieces.
- There's also a connection to the Affirma Analog Environment which is an interface to the Cadence Spectre analog simulation tool. Using this interface you can, for example, pick circuit nodes from your schematic to be plotted in the analog simulation output file. This connection will also generate the required Spectre or Spice netlist from your schematic automatically.
- You can generate a schematic by hand, or you can generate a schematic automatically from a Verilog structural netlist. For example, you can take the output from Synopsys dc\_shell or Cadence BuildGates synthesis and generate a schematic from that structural Verilog file. The generated will be a mess to look at! But, it will be very useful for using the Cadence LVS (Layout Versus Schematic) checks using either Diva or Assura LVS. This will compare a layout to a schematic to make sure they are structurally the same.

You could easily use  
gates from the  
NCSU\_Digital\_Parts  
library for this tutorial if  
you are using the CDK  
without local  
enhancements.

For now, this chapter will introduce Composer by drawing schematics for simple circuits using standard cell gates and modules from the UofU\_Gates library, and transistors from the NCSU\_Analog\_Parts and UofU\_Analog\_Parts libraries. In order to follow these steps you must have started up Cadence icfb with the NCSU CDK extensions. This was explained in Chapter 2, and is done using the `cad-ncsu` script.

### 3.1 Starting Cadence and Making a new Working Library

Now that you have your own cadence directory (called `~/IC_CAD/cadence` if you've followed the suggestions up to this point), remember to connect to that directory before starting up Cadence. Also make sure you have your `LOCAL_CADSETUP` environment variable set so that you get the class extenions to the Cadence setup. Directions for setting this up were described in Chapter 2.

1. Start up Cadence dfII by running the command `cad-ncsu`, as described in Chapter 2. You may have used (and may still be using) a different setup for a different class, but please use `cad-ncsu` for this class. You should get a window (called the **Command Information Window** or CIW) similar to the one shown in Figure 2.1.
2. You should also get another window for the **Library Manager** as shown in Figure 2.2. The libraries that you'll see in the default **Library Manager** window are described in Chapter 2.
3. In order to build your own schematics, you'll need to define your own library for your own circuits. To create a new working library in the library manager, select **File** → **New** → **Library**. In the **Create Library** window that appears fill in the Name field as **tutorial**, or whatever you'd like to call your library. Leave the **Path** field blank so that it creates the new library under the directory in which you started Cadence. If you want to create a library somewhere other than in your `7IC_CAD/cadence` directory you can put the whole path in the path field. In the **Technology Library** box select **Attach to an existing tech library** and choose the **UofU AMI 0.60u C5N** technology library. The dialog box is shown in Figure 3.1 Now press OK and the new library will show up in your Library Manager window.

*Libraries are defined in a `cds.lib` file that is created in your cadence directory.*

*Don't make your library name start with a number, and don't use "-" or "." in the name! An underscore is all right.*

Now the working library has been created. All the project cells (components) that you generate should end up in this library. When you start up the **Library Manager** to begin working on your circuits, make sure you select your own library to work in.

## 3.2 Creating a New Cell

When you create a new cell (component in the library), you actually create a view of the cell. For now we'll be creating "schematic" views, but eventually you'll have lots of different views of the same cell. For example, a "layout" view of the same cell will have the composite layout information in it. It's a different file, but it should represent the same circuit. More about that later. For now, we're creating a schematic view. To create a cell view, carry out the following steps

*We'll eventually use "cmos\_sch" views for individual leaf cells and "schematic" views for cells with hierarchy. For now we'll just make "schematic" views to keep things simple.*

### Creating the Schematic View of a Full Adder

1. Select the **tutorial** library you just created in the Library Manager.

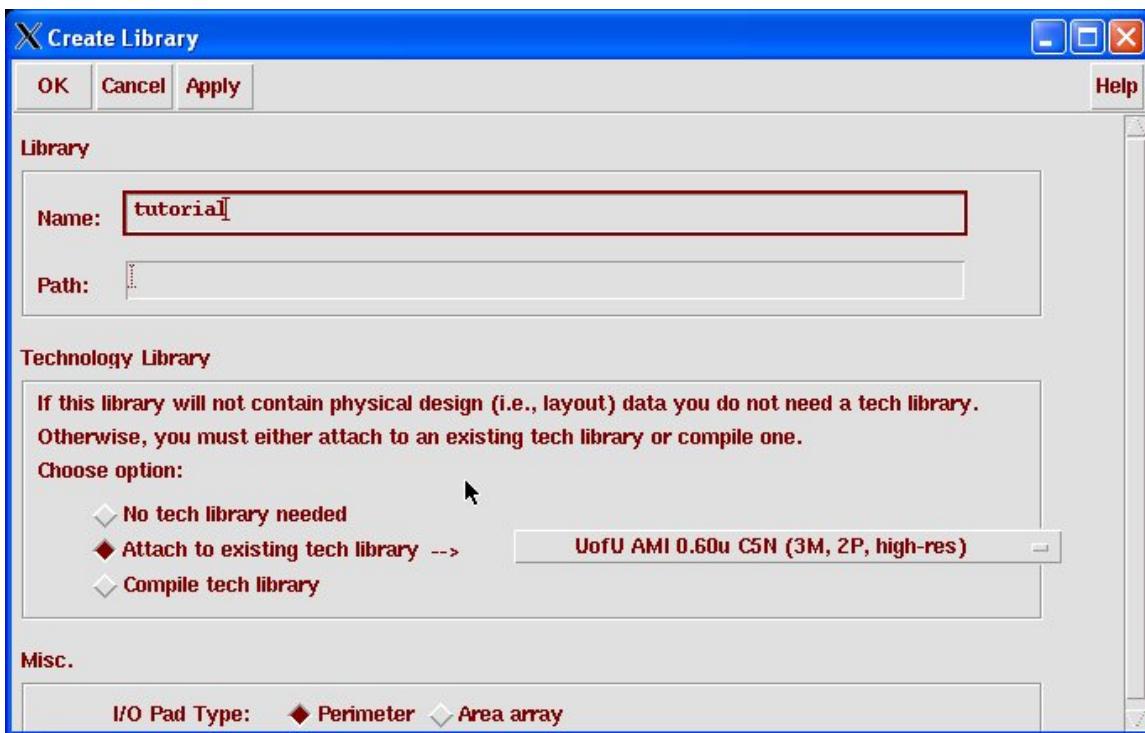


Figure 3.1: Library creation dialog box in Library Manager

2. Select **File → New → Cell View** from the **Library Manager** menu. The **Create New File** window appears. The **Library Name** field should be filled in as **tutorial** and the **Cell Name** field as **FullAdder**. Choose **Composer - Schematic** from the **Tool** list and the view name should be automatically filled as **Schematic**. This window is shown in Figure 3.2. Click **OK**.
3. A blank Composer schematic editing window appears which looks like that in Figure 3.3.
4. Adding Instances: An instance (either a gate from the standard cell library, or a cell that you've designed earlier) can be placed in the schematic by selecting **Add → Instance** or pressing **i** on the keyboard, and the **Component Browser** dialog appears (shown in Figure 3.4). Choose a library from the **Library** list and then choose the cell from the list box and the outline of the component's symbol will appear when your cursor is in the schematic window. Left clicks will add instances of that cell to your schematic.

*Many of the keyboard keys are bound to “hotkeys” which cause actions to happen. You can see these bindings in the menus.*

Additional options can be seen by popping up the **Add Instance** dialog box. In this box you can change, for example, change the orienta-



Figure 3.2: New cell dialog box in Library Manager

tion of the symbol by rotation or reflection which can come in handy for some schematics. This box is enabled with the **F3** function key. This box is shown in Figure 3.5.

The instances you should add are “symbol views” of the instances. There are lots of “views” of cells that all have different features, but for schematics, symbols are what you want.

For example to add a 2-input NAND gate use the the **Component Browser** window, chose the library in the Library list as **UofU\_Gates\_v1\_1** and cell in the list box as **nand2**. Alternately you can use the **Add Instance** window, fill the **Library** as **UofU\_Gates\_v1\_1**, the **Cell** field as **nand2**, and the **View** field as **Symbol**. Still another way to add an instance is through the **Add → Instance** menu in the composer window.

Other instances can be added in the similar fashion as above. To come out of the instance command mode, press **Esc**. This is a good command to know about in general. Whenever you want to exit an editing mode that you’re in, use **Esc**.

5. Connecting Instances with Wires: To connect the different instances with wires select **Add → Wire (narrow)**, or press **w** to activate the wire command, or select the “narrow wire” widget from the toolbar on the left (it looks like a narrow wire ...). Now go to a connection point of the gate instance and left-click on it to draw the wire and left-click on another connection point (another gate’s input or output, or another wire, for example) to make the connection. If you would like

*The **F3** function key often adds options to commands. Try it in different situations to see what it does.*

*I sometimes just hit a bunch of **Esc**’s whenever I’m not doing something else just to make sure I’m not still in a strange mode from the last command.*

*If you hover the mouse over the widgets on the left of the window you will get a little popup that tells you what that widget does.*

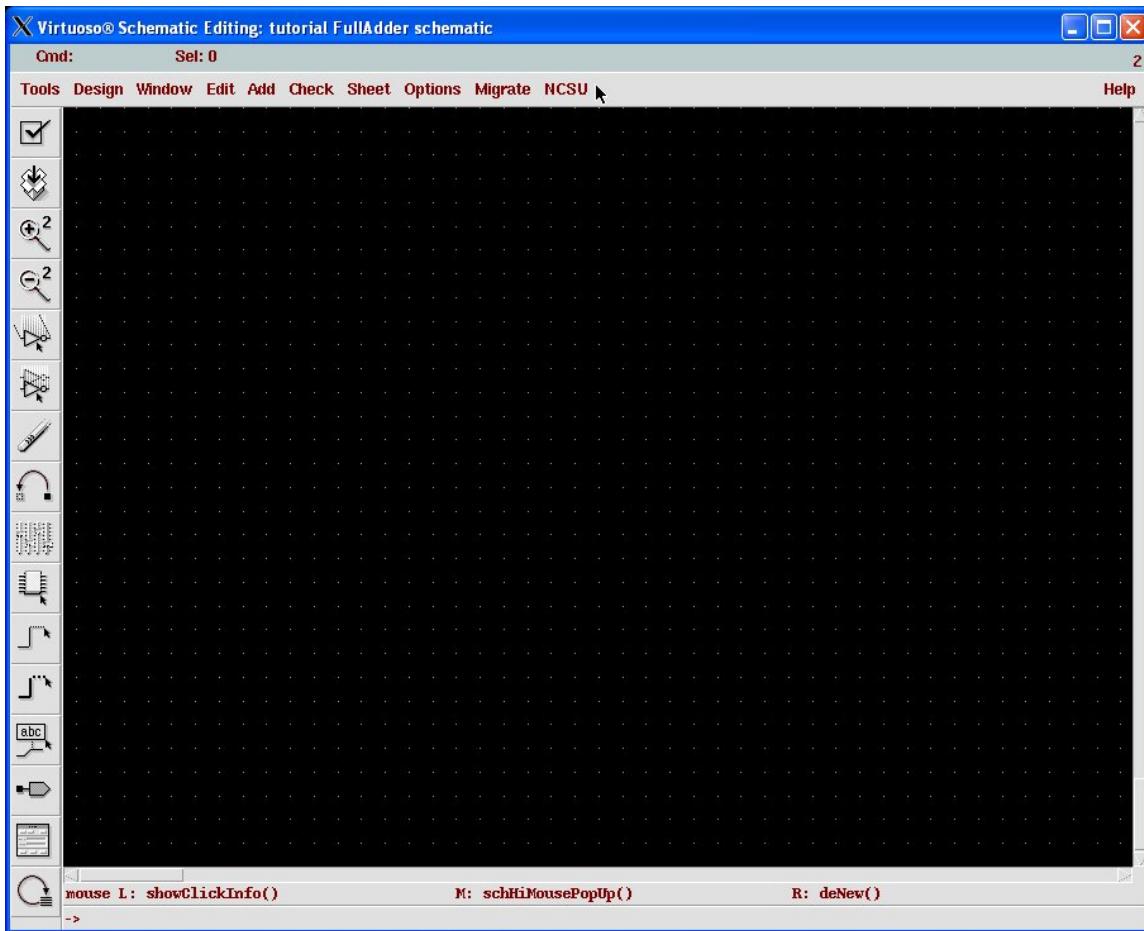


Figure 3.3: Blank Composer window - note command widgets on left of screen

to end the wire at any point other than a node (i.e. leave it hanging so that you can add a pin later on), double left-click at that point. To come out of the wire command mode, press **Esc**.

*Note that because we're eventually going to simulate this with a Verilog simulator, the names you pick for pins and wires must be legal Verilog names. Verilog, for example, is case sensitive!*

6. Adding Pins: Pins are connections that enter or leave this schematic sheet. They are called pins because they will correspond to pins on the symbol view of this schematic. Pins can be added by going to **Add → Pin** from the menu, or pressing **p**, or selecting the “Add Pin” widget from the left side of the Composer window to get the **Add Pin** dialog box (Figure 3.6). . For example, to put two input pins A and B, we can fill in the Pin Names field as A B (with a space) and the Direction list as input.

Now go to the wire where you need to place the pin and left-click on

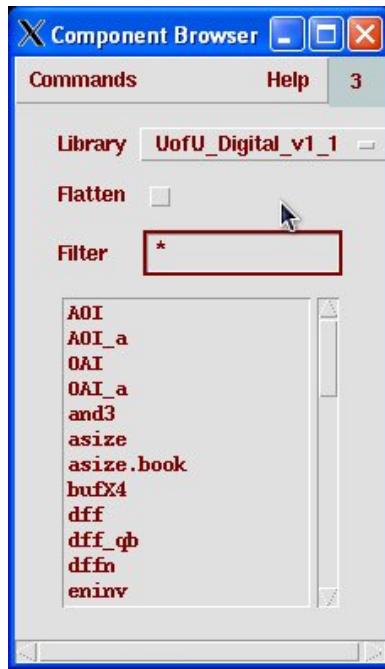


Figure 3.4: Component Browser dialog

it. The pin will be connected to the wire and look like a red pointer. If you need to rotate or flip the pins (i.e. to have an input coming in from the right instead of the left) use the buttons at the bottom of the **Add Pin** dialog box.

## 7. Other Command Functions

Some common command modes and functions available under the Add and Edit menus in cadence are (of course, there are many more!):

Under Add Menu:

**Add → Wire (wide)** or press **[W]** to add a bus

**Add → Wire name...** or press **[1]** to name wires

**Add → Note → Note Text...** or press **[L]** to add a note

Under Edit Menu:

**Edit → Undo** or press **[u]**

**Edit → Stretch** or press **[s]**

**Edit → Copy** or press **[c]**

**Edit → Move** or press **[m]**

**Edit → Delete** or press **[Delete]** key

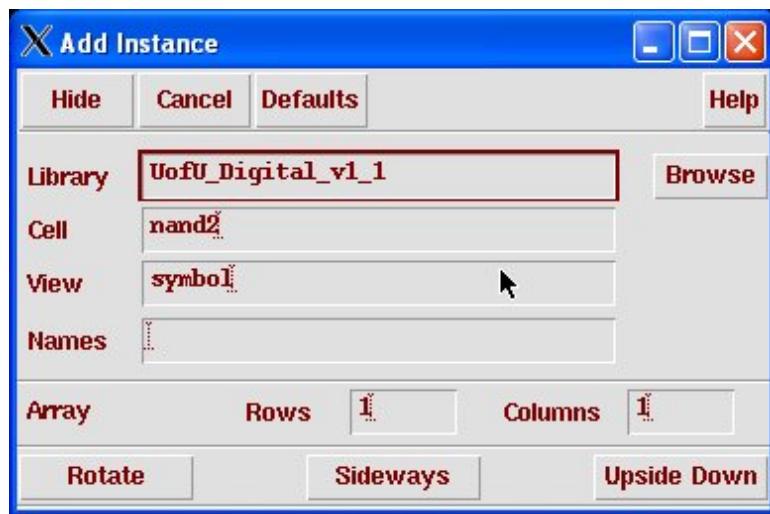


Figure 3.5: Add Instance dialog box

Edit → Rotate or press **[r]**

Edit → Properties → Object or press **[q]**

The **Edit Properties** command is a general command that can give you all sorts of information about whatever object you select. It's quite handy.

In general you can pick items in the schematic and move them using the left mouse button. You can usually select groups of objects by clicking and dragging with the left button, and you can zoom by clicking and dragging with the right button. All of these buttons are mode-dependent though. You can see the current bindings of the mouse buttons in the lower part of the Composer window.

8. It's good shematic practice to always put a border around your schematic. Borders can be found in the **UofU\_Sheets** library. The asize sheet is a good one for small circuits because when you print the schematic all the gates and labels will still be visible. Larger sheets like the bsize, csize, etc. will cause the gates and labels to be too small to see when printed on 8.5 x 11 paper. When you add a border you can add your name and other info in the title block using the **Sheet → Edit Title ...** dialog box.

*Libraries sometimes group their cells into “categories.” In this case you select the category first in the **Component Browser** before selecting the cell.*

Using the commands above to draw a full adder bit, and including an **Asize** sheet, and filling in the title block results in the schematic seen in Figure 3.7.

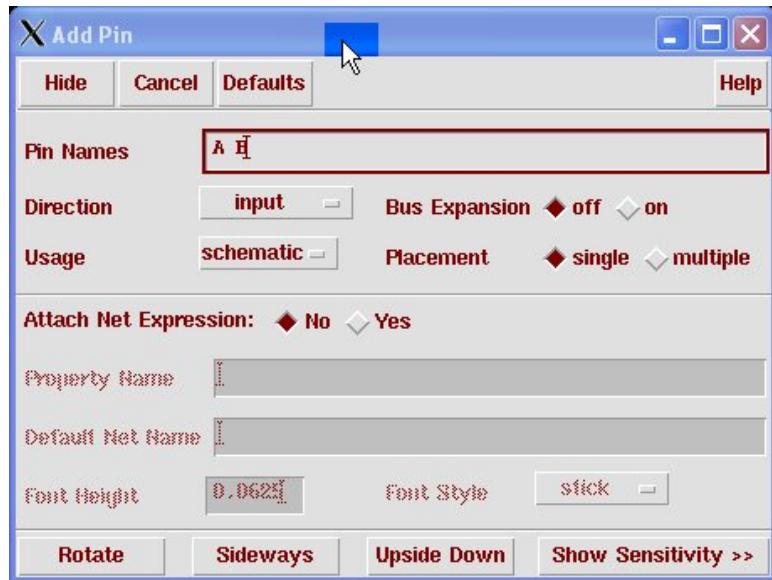


Figure 3.6: Add Pin dialog box

9. The design can be checked and saved by selecting **Design → Check and Save** or by pressing **[F8]**. For an error free schematic, you should get the following message in the CIW,

```
Extracting ''FullAdder schematic''
Schematic check completed with no errors.
''tutorial FullAdder schematic'' saved.
```

Note that there's a big difference between the **Save** and the **Check and Save** commands. The **Save** command doesn't do any checks on the schematic! If you know there are errors that you haven't fixed yet but want to save so you don't lose work, use **Save**, but eventually you need to do **Check and Save** so that Cadence checks the schematic for errors. The CIW should not show any warnings or errors when you check and save. If it does, you should read and understand all of them before moving on. Some warnings may be ignored, but only if you're sure you understand what they are and that they are safe to ignore. For example, you can ignore the warning about outputs connected together if those outputs are coming from gates with tri-state outputs, but not if they are coming from regular static gates.

After saving the design with no errors, you can close the window (and exit Composer if this was the last window) by selecting **Window → Close** or pressing **[ctrl+w]**. Or you can leave the window open to go on to the next step.

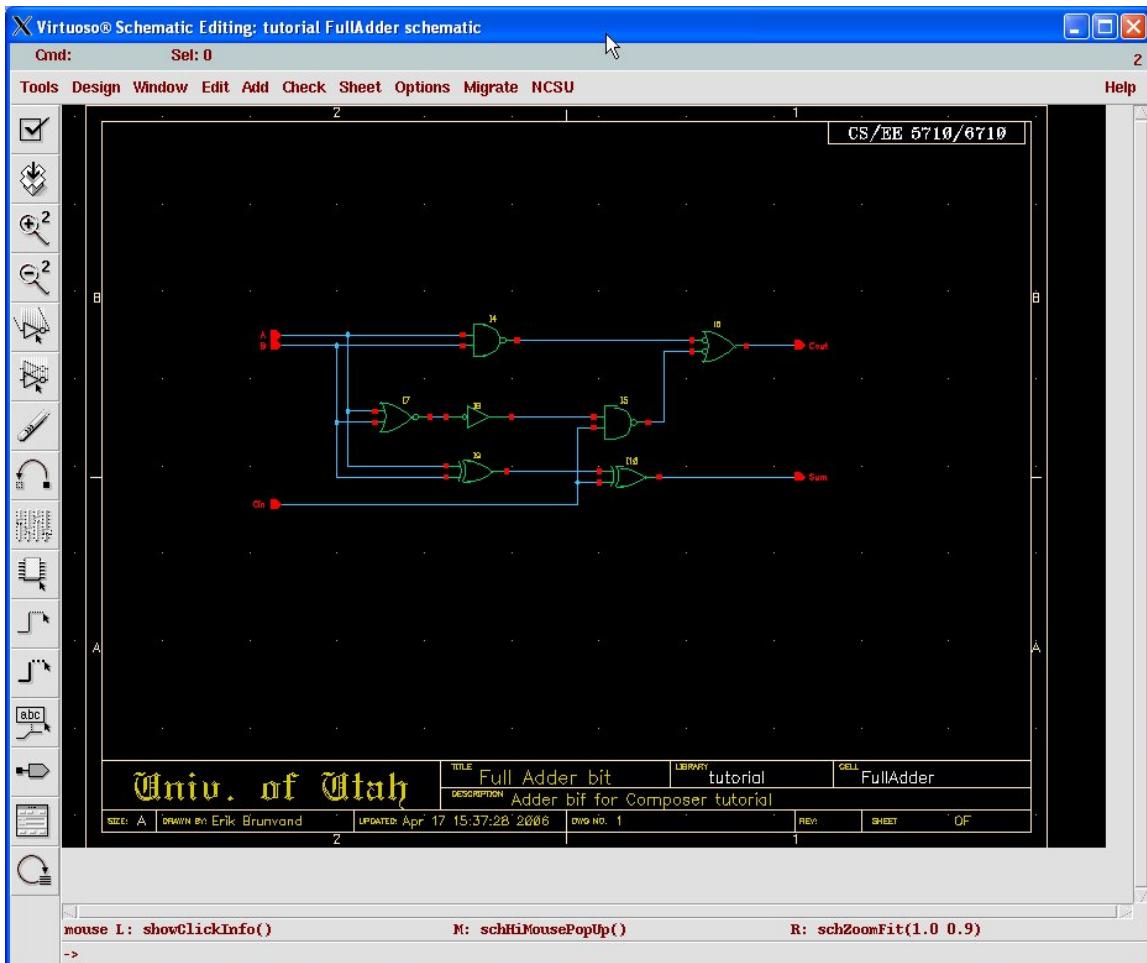


Figure 3.7: A Full Adder Bit Schematic

### Creating the Symbol View of a Full Adder

You have now created a schematic view of your full adder. Now you need to create a symbol view if you want to use that circuit in a different schematic. A symbol is a graphical wrapper for the schematic so that it can be used to place an instance of that circuit in another circuit. The pins of the symbol must match one to one with the pins inside the schematic. The name must also be the same, but that will happen automatically if the schematic and symbol are different views of the same cell.

1. In the Composer schematic window of the schematic you have created above, select **Design** → **Create Cell View** → **From Cell View**. A **Cell View from Cell View** window appears, press OK.



Figure 3.8: A simple symbol for the FullAdder circuit

2. In the **Symbol Editing** window which appears make modifications to make the symbol look as below. If you leave `[@partname]` in the symbol it will be filled in with the name of the cell when you instantiate this cell. If you want the symbol to say something different than the name of the cell you can replace `[@partname]` with some other text (**add→note→notetext**). The `[@instanceName]` placeholder will be filled in with an automatically generated instance number when you use the symbol. Note in the FullAdd schematic that there are yellow instance numbers above each gate. These are unique identifiers attached to every gate instance. An example of a very simple symbol for the FullAdd cell is shown in Figure 3.8.

I haven't made many modifications to the automatically generated symbol in this case. All I've done is reorder the input and output pins in the symbol so that they'll connect more efficiently in a ripple-carry adder connection. You can format the symbol to make it look like the one in Figure 3.8 or use the edit commands to make the symbol look like whatever you like (use **add→shape** in the symbol editor, for example). Save the symbol and exit using **Window → Close**.

Once you have a symbol view, when you **Check and Save** the schematic it will also check that the pins in the schematic match up with the pins in the symbol. Now the full adder is ready to be used in other schematics.

### Creating a 2-bit Adder using the FullAdder bit

Make sure that you have selected the **tutorial** library in the **Library Manager** and then select **File → New → Cell View** to make a new cell schematic. Fill in the **Cell Name** field as `twoBitAdder` and select the **Tool** from the list to be **Composer - Schematic**. This will make a new cell in the **tutorial** library for our new circuit.

In the Schematic Editing window which appears place two instances of the 1-bit full adder we created previously by selecting **Add → instance** or

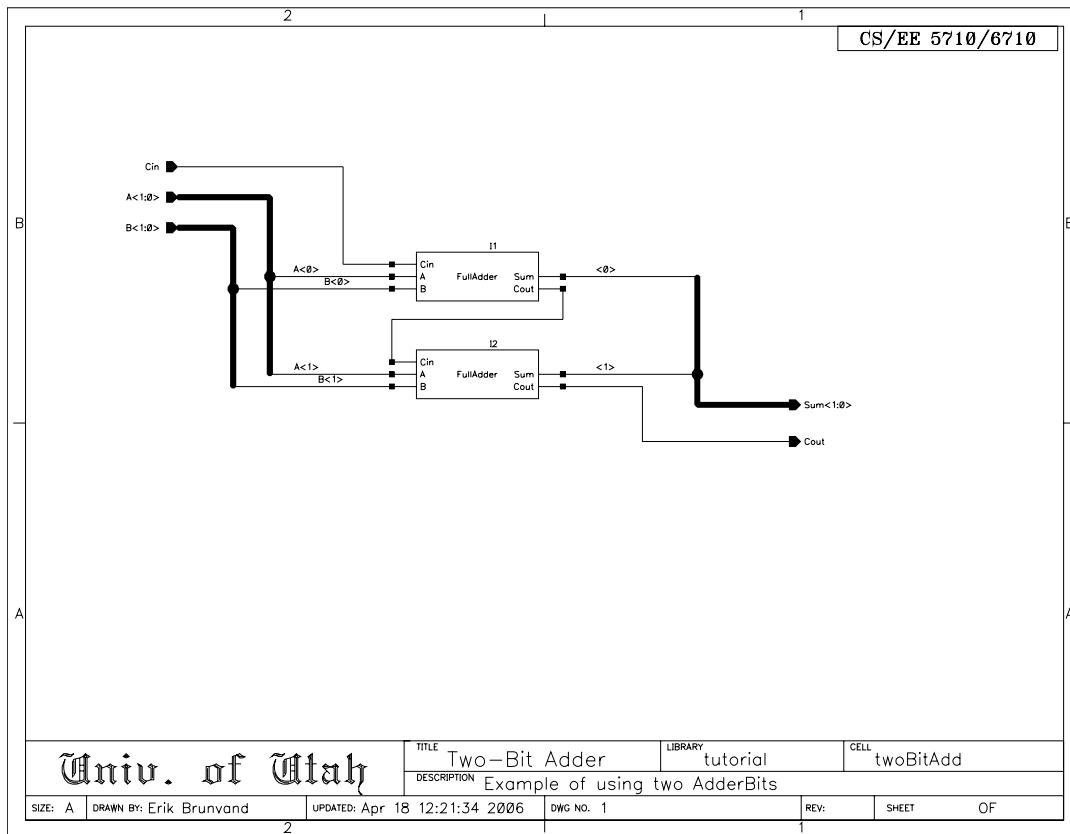


Figure 3.9: A two-bit adder using FullAdder bits

pressing **i** and attach them as in Figure 3.9. Remember to add an **A-size** sheet.

Note that the two-bit inputs and outputs have been bundled into two-bit buses in the schematic. Buses in Composer schematics are just wide wires, but they also have special naming conventions so that the tool knows how to assign names to each wire in the bus.

To add a bus in cadence, select **Add → Wire (wide)** or press **W** and then go about in a similar fashion as that of adding a wire to the schematic. The trickiest part of adding a bus is labeling. The pins connecting the buses are named in the following manner:  $A<1:0>$ ,  $B<1:0>$ , where the numbers in the angle brackets represent the number of bits in the bus. The first number is the starting bit and the next number is the ending bit. Each wire in the bus must be named according to the expansion of this bus label. For example, in the A bus there are two wires named  $A<1>$  and  $A<0>$ . In Verilog

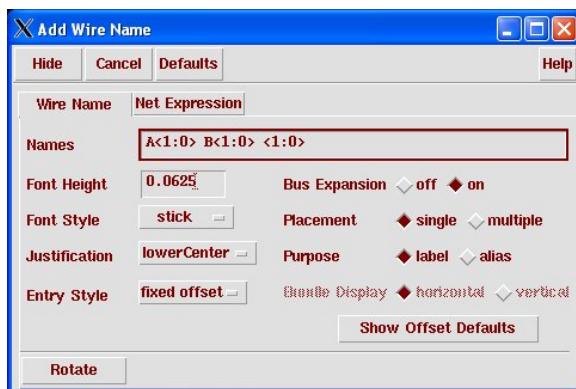


Figure 3.10: The Wire Name dialog box

buses are labeled in “little endian” mode which means that the high order bit should have the highest bit number and the low order bit the lowest. Thus, if you want the bus value to print out correctly later on your buses should be labeled A<1:0> and not A<0:1>.

In cadence the buses inherit the pin names and the nets tapping the buses inherit the bus name. This is why you can name the individual wires with the bit number only such as <0> and <1>. You can also name the individual wires A<1> and A<0> if you like. What is important is that every wire (thin single-bit wire or thick multi-bit wire) must be named so that the Composer knows which wires from the bus are being used. Wires (both thick and thin) can be named with the **Add → Wire Name ...**, or **[1]** (after first selecting the wire to name), or the wire naming widget. The dialog box is shown in Figure 3.10.

The wire naming dialog box has **expand bus names** and **don't expand bus names** options that are useful for bus naming. If you give a bus-name like A<1:0> in the wire naming dialog box with the **expand bus names** checked, then each left click will name a single wire with one of the bus wire names (A<1> then A<0> for example). If the **don't expand bus names** option is used, then the next left click will name a (thick) wire with the whole name (A<1:0> in this example). Figure 3.11 has a closer view of the circuit so that you can see examples of how the bus naming works.

### 3.3 Schematics that use Transistors

Transistors can be used in designs just like any other “gate” primitives. They will eventually be simulated using the built-in transistor switch models in Verilog. They can also be simulated in an analog simulator like Spectre or

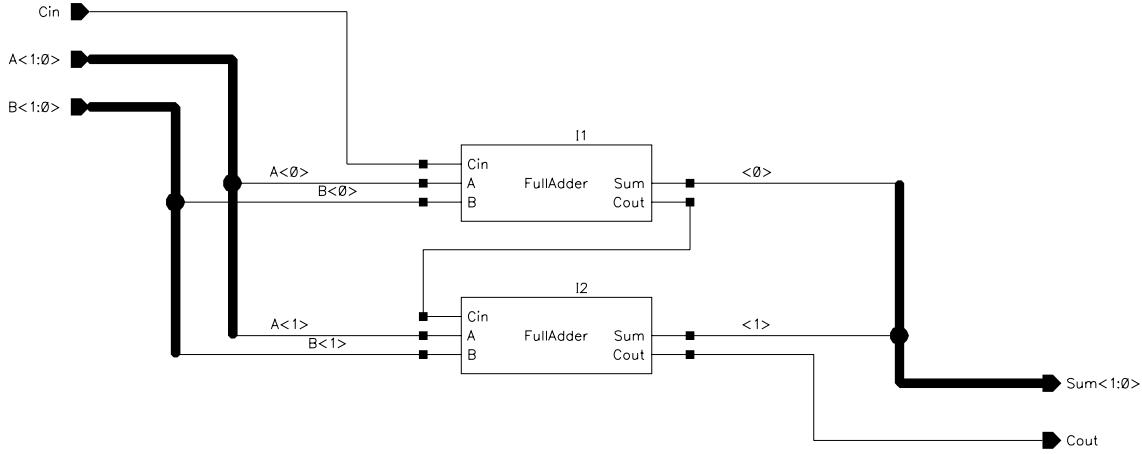


Figure 3.11: Close view of the twoBitAdd showing the bus naming

**Spice.** The generic transistor components are in the **NCSU\_Analog\_Parts** library. For now use the **nmos** and **pmos** transistors from those libraries. If you need four-terminal devices you can use **nmos4** and **pmos4** for devices with an explicit bulk node connection.

These will netlist into zero-delay switch models when they are simulated. If you'd rather have unit-delay simulation of your transistor switches you can use the same devices, but from the **UofU\_Analog\_Parts** library. The **UofU** library also has "weak" transistors that can be used for weak feedback circuits, and bidirectional transistors (the generic models must be used with their drain connections being the output connections - which is usually what you want and speeds up simulation speed).

The following is an example of a simple NAND gate designed using transistors:

1. Select **File** → **New** → **Cell View** from the Library Manager menu or from the CIW menu. The **Create New File** window appears. The **Library Name** field should be **tutorial**. Fill in the **Cell Name** field to **nand2**. Choose **Composer - Schematic** from the Tool list and the view name is automatically filled as **Schematic**. Click **OK**.

In the schematic window select **Add** → **Instance** or press **i**, and the **Add Instance** dialog appears. To add an NMOS transistor, in the **Component Browser** choose the library to be **NCSU\_Analog\_Parts**,

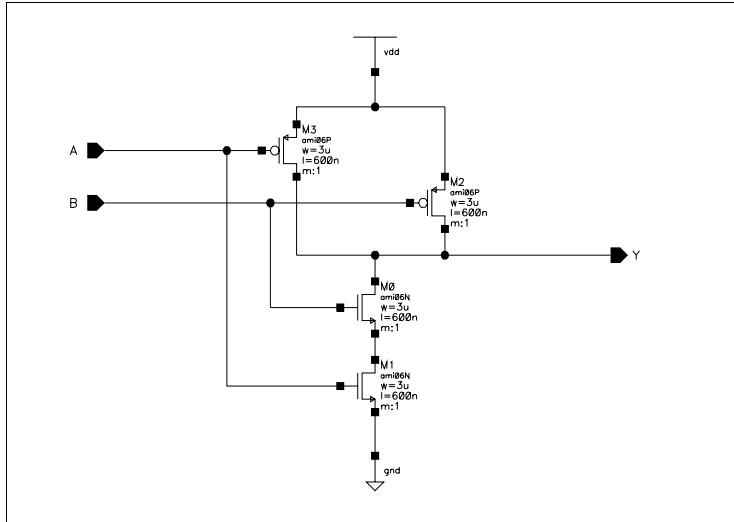


Figure 3.12: Transistor schematic for a two-input NAND gate

navigate through the **N\_Transistors** category to the **nmos** cell. Or, directly in the **Add Instance** window fill the **Library** as **NCSU\_Analog\_Parts**, the **Cell** field as **nmos** and the **View** field as **symbol**. Add the PMOS transistor symbols to the schematic using the same procedure through the **P\_Transistors** category to the **pmos** cell. Create the schematic of the NAND gate as in Figure 3.12 by adding wires and pins.

2. For transistor-based circuits you need to add the power supply connections explicitly. The vdd and gnd supply nets are in the **NCSU\_Analog\_Parts** library in the **Supply Nets** category. Connecting to the instances vdd and gnd in your schematic automatically connects these wires to logic 1 and logic 0 respectively in verilog simulations. We will connect those supply nets to particular voltages (+5v and 0v, for example) later in the analog simulations.
3. Transistors need a few more parameters than Boolean gates do. In particular, transistors need at least a length and width parameter to define their strength. You can do this in the **Add Instance** dialog box (you may have noticed that the **Add Instance** dialog was a lot bigger for the transistors than for the gates), or can do it after the fact by changing the properties of each transistor.

Do change the properties select one of the nmos transistors and select **Edit → Properties → Object**, or press **q**, or use the **Properties** widget to bring up the properties window. The full properties window

is shown in Figure 3.13.

Change the properties of the nmos transistor by changing the Width to  $[3\text{u M}]$  (3 microns). Leave the length as  $[600n\text{ M}]$  (0.6 microns). Similarly follow the steps for the pmos transistor with  $W/L = 3/0.6$  (i.e.  $W = [3\text{u M}]$  and  $L = [600n\text{ M}]$ ).

4. Create a symbol for the NAND gate by selecting **Design** → **Create CellView** → **From CellView**. The Cadence-generated symbol will be a simple rectangle. You can easily modify it to make it look like Figure 3.14 using arcs and circles.

Note that in order to get the circle for the nand2 bubble to be the right size in proportion to the rest of the gate you may have to use a finer grid while you're drawing that circle. You can change the grid size in the **Options** → **Display Options** dialog box, as the **Snap Spacing** value. But, when you're done make sure that you set the snap grid back to 0.0625 so that the pins you make will match up properly with the wires in the schematic!

## 3.4 Printing Schematics

To print schematics you need to have printers set up by your tools administrator. The printers available to you are defined in a **.cdsplotinit** file. This file lives in the Cadence installation tree, but may also exist in a site-specific location for local printer information. It contains printer descriptions that describe which plot drivers should be used (usually postscript), and how to spool the output to the printer. There is usually at least one postscript or eps (encapsulated postscript) option defined so that if you plot to a file you can get a postscript version of your schematic.

*Details of the .cdsplotinit file can be found in Chapter A*

To print (plot) a schematic, select the **Design** → **Plot** → **Submit...** menu choice. This will bring up the **Submit Plot** dialog box (seen in Figure 3.15). If all the choices are correct, you can select **OK** to plot the file. If you've selected a printer the schematic will print to that printer. If you've selected the **Plot To File** option you will get a file in the directory from which you started Cadence. Those are options that you have to select from the **Plot Options...** choice though.

When you click on the **Plot Options...** button you get another dialog box for the plot options as seen in Figure 3.16. This dialog box lets you set up all sorts of details about how you want your schematic plotted. The important options are:

**Top Section:** In this section you can choose which plotter (printer) you wish to send your hard copy to with the **Plotter Name** selection. This



Figure 3.13: Device Properties for an nmos device

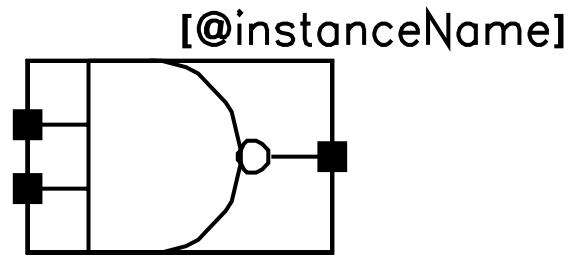


Figure 3.14: A symbol for the nand2 circuit



Figure 3.15: Dialog Box for Submit Plot

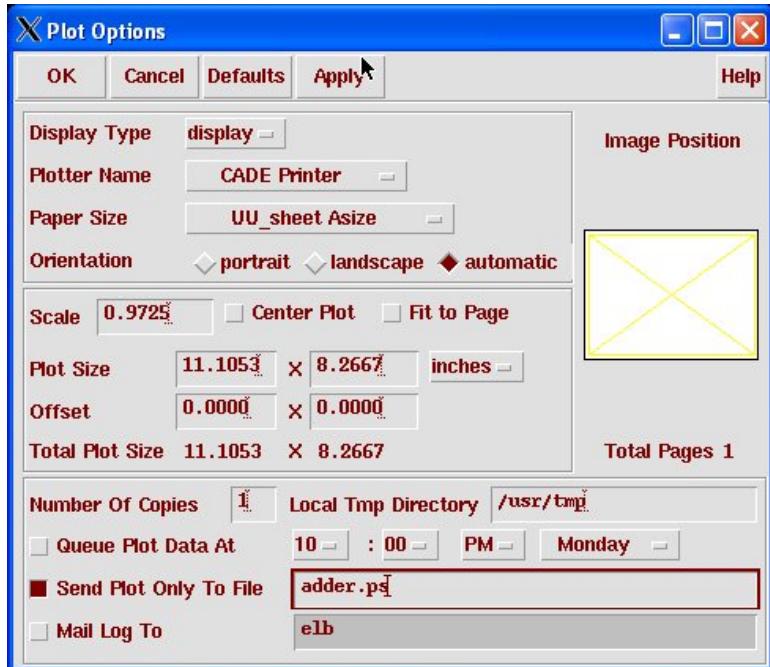


Figure 3.16: Dialog Box for Plot Options

also chooses a printer type for the **Plot to File** option. That is, if you want a postscript file, select a postscript plotter here. You can also choose a **Paper Size** if your printer has different sizes of paper loaded. Normally you use some sort of Asize sheet. This is defined in the .cdsplotinit file and, at least in the US, is likely to be 8.5x11 inches. **Orientation** chooses how the image is plated on the paper. You can see the image position on the right to see how things will be placed on the page.

**Middle Section:** The controls in the Plot Size section control how the plot is fit to the page. You may want to explore the **Center Plot** and **Fit to Page** options.

**Bottom Section:** Here you can select the number of copies to print and when to queue the plot (if you want to delay the printing until later). You can also select **Send Plot Only To File** to plot to a file instead of sending the file to a printer. The file name that you specify will appear in the directory in which Cadence was started. If you want email when the file has printed, select the **Mail Log To** option and give your email address. I seldom want extra email confirmation that the file has been printed, but you may like it.

Once you've selected your **Plot Options**, you can see them updated in the lower section of the **Submit Plot** dialog. In the example in Figure 3.15 you can see that although I'm selecting the **CADE Printer**, I've also selected to **Plot To File** with a filename of **adder.ps**.

In the **Submit Plot** dialog you can select which schematic to plot. This should already be filled in for the schematic you were viewing when you selected **Plot Submit** from the **Composer** menu. Note that you can select to plot the current **cellview** which is the entire schematic, or the **Viewing Area** which will plot the currently zoomed view in **Composer**. This is useful to zoom into sections of a schematic for printing. One final useful button is to unselect the **Plot With header** option so that the output is just a single page with the schematic and doesn't include an extra header sheet. If you're printing in an environment where header sheets are handy to keep hard copies organized, then leave the **header** option selected.

### 3.4.1 Modifying Postscript Plot Files

Although the Cadence **Composer** plotting interface produces good postscript files, there are some situations where I've found it useful to modify them. In particular, the postscript files produced by **Composer** are in color. This is fine if you're printing them on a color printer, or are including them in an on-line document that will be seen on a screen. However, if you're printing them on a black and white printer, the colors translate to grey values on the printer and the result is often a light grey schematic that is very hard to see. Also, the line size that is visible on the screen is sometimes not as bold on the printed page as you would like.

Unfortunately, I've found no easy way to modify the way that the postscript is produced by **Composer**. Instead, I've resorted to hand-editing the postscript files. Luckily postscript files are in plain ascii text and can be edited using any text editor.

To turn a color postscript output from Cadence into a black and white file (i.e. no grey scale), look for the following lines in your postscript file:

```
/slc {  
    /lineBlue exch def  
    /lineGreen exch def  
    /lineRed exch def  
    /color 1 def  
    lineBlue 1000 eq {lineGreen 1000 eq {lineRed 1000 eq {  
        /lineBlue 0 def  
        /lineGreen 0 def
```

```
/lineRed 0 def
} if} if} if
} def

/sfc {
    /fillBlue exch def
    /fillGreen exch def
    /fillRed exch def
    /color 1 def
    fillBlue 1000 eq {fillGreen 1000 eq {fillRed 1000 eq {
        /fillBlue 0 def
        /fillGreen 0 def
        /fillRed 0 def
    } if} if} if
} def
```

Change all the lines that read `/color 1 def` to be `/color 0 def` and the schematic will print in black and white with no color or grey scale approximation of color.

To change the thickness of the lines in the postscript file, look for the postscript function:

```
/ssls { [] 0 setdash
        1 setlinewidth
} def
```

This sets the default line width to 1 point. If you increase the value of the line width to 3 or 4 you will get a bolder line in your schematic. These hacks can make much better looking schematics in hardcopy or to include in another document. Of course, depending on which other document you are including the graphics into you may need to convert the postscript into some other format such as pdf (ps2pdf) or jpg (I use Illustrator on a PC for this trick. The xv program on linux also works.).

### 3.5 Variable, Pin, and Cell Naming Restrictions

Although it's not obvious at the moment, all the simulation in the Cadence dfll environment is through Verilog simulators. So, although Composer allows wire and pin names that aren't legal Verilog names, your life will be *much* easier if you only use Verilog-compatable names within Composer. This means that names must start with a letter, and can then consist of letters, numbers, and underscores (the `_` character) only. Do not use dashes or

periods in names. Also, you should avoid all Verilog reserved words. The reserved words that are most likely to bite you are “input” and “output.” The complete set of reserved words is:

and	for	output	strong1
always	force	parameter	supply0
assign	forever	pmos	supply1
begin	fork	posedge	table
buf	function	primitive	task
bufif0	highz0	pulldown	tran
bufif1	highz1	pullup	tranif0
case	if	pull0	tranif1
casex	ifnone	pull1	time
casez	initial	rcmos	tri
cmos	inout	real	triand
deassign	input	realtime	trior
default	integer	reg	trireg
defparam	join	release	tri0
disable	large	repeat	tril
edge	macromodule	rnmos	vectored
else	medium	rpmos	wait
end	module	rtran	wand
endcase	nand	rtranif0	weak0
endfunction	negedge	rtranif1	weak1
endprimitive	nor	scalared	while
endmodule	not	small	wire
endspecify	notif0	specify	wor
endtable	notif1	specparam	xnor
endtask	nmos	strength	xor
event	or	strong0	

# Chapter 4

## Verilog Simulation

**A**HARDWARE DESCRIPTION LANGUAGE (HDL) is a programming language designed specifically to describe digital hardware. Typical HDLs look somewhat like software programming languages in terms of syntax, but have very different semantics for interpreting the language statements. Digital hardware, when examined at a sufficient level of detail, is described as a set of Boolean operators executing concurrently. These Boolean operators may be complex Boolean functions, refined into sets of Boolean gates (NAND, NOR, etc.), or described in terms of the individual transistors that implement the functions, but fundamentally digital systems operate through the combined effect of these Boolean operators executing at the same time. There may be a few hundred gates or transistors, or there may be tens of millions, but because of the concurrency inherent in these systems an HDL used to describe these systems must be able to support this sort of concurrent behavior. Of course, they also support “software-like” sequential behavior for high-level modeling, but they must also support the very concurrent behavior of the fine-grained descriptions.

To enable this sort of behavior, HDLs are typically executed through event-driven simulators. An event-driven simulator uses a notion of simulation time and an event-queue to schedule events in the system being described. Each HDL construct (think of a construct as modeling a single gate, for example, but it could be much more complex than that) has inputs and outputs. The description of the construct includes not only the function of the construct, but how the construct reacts over time. If a signal changes then the event-queue looks up which constructs are affected by that change to know which code to run. That code may produce a new event at the construct’s output, and that output event will be sent to the event queue so that the output event will happen sometime in the future. When simulation time advances to the correct value the output event occurs which may cause other activity in the described system. The event-queue is the central controlling

structure that enables the HDL program to behave like the hardware that it is describing. So, although you may think of “running a program” written in a software programming language, it’s more correct to think of “running a simulation” when executing an HDL program.

*One reason to choose Verilog is that some of the tools in this CAD flow, place and route in particular, require Verilog as an input specification.*

Verilog is one of the two most widely used Hardware Description Languages with VHDL being the other main HDL in wide use today. Much of the simulation information described in this chapter will translate reasonably easily to VHDL simulators, but for this text I’ll stick with Verilog. The choice of using Verilog is somewhat arbitrary as the two languages are quite similar in their ability to describe digital systems. In very general terms, many designers find Verilog syntax simpler and easier to use, at the expense of VHDL’s richer type system, but both HDLs are used on “real” designs.

Verilog program execution (program simulation) requires a Verilog simulator that implements the event-driven semantics of the language. You will also need a method of sending inputs to your program, and a means of checking that the outputs of your Verilog program are correct. This is usually accomplished using a second Verilog program known as a *testbench* or *testfixture*. This is similar to how integrated circuits are tested. In that case the chip to be tested is called the Device Under Test (DUT), and the DUT is connected to a testbench that drives the DUT inputs and records and compares the DUT outputs to some expected values. If we use the same terminology for our Verilog simulations, then the Verilog program that you want to run would be the equivalent of the DUT, and you need to write a testbench program (also in Verilog) to drive the program inputs and look at the program outputs.

This general scheme is shown in Figures 4.1 and 4.2. These figures show the test environment that is created by the Composer system, but they are a good general testbench format. There is a top-level module named **test** that is simulated. This module includes one instance of the DUT. In this case the DUT is our twoBitAdd module from Chapter 3 and the instance name is **top**. It also includes testfixture code in a separate file named **testfixture.verilog**. In this file is an **initial** block that has the testfixture code. Example testfixture code will be seen in the following sections of this Chapter. Note that the module **test** defines all the inputs to the DUT as **reg** type, and outputs from the DUT as **wire** type. This is because the testfixture wants to set the value of the DUT inputs and look at the value of the DUT outputs.

This text does not include a tutorial on the Verilog language. There are lots of good Verilog overviews out there, including Appendix A of the class textbook *CMOS VLSI Design: A Circuits and Systems Perspective*, 3rd ed by Weste and Harris [1]. I’ll show some examples of Verilog code and testbench code, but for a basic introduction see Appendix A in that book, or any of the good Verilog introductions on the web.

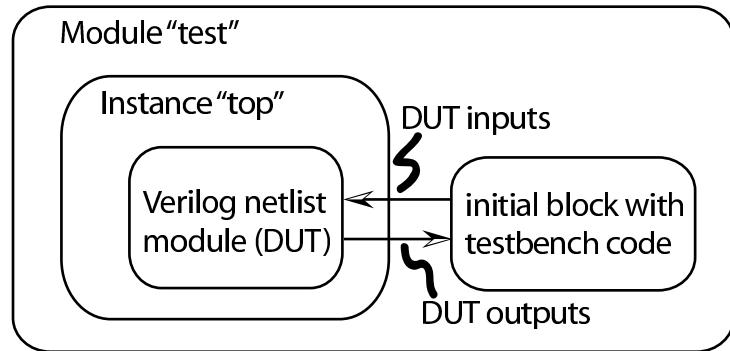


Figure 4.1: The simulation environment for a Verilog program (DUT) and testbench

```
'timescale 1ns / 100ps
module test;

wire Cout;
reg Cin;
wire [1:0] Sum;
reg [1:0] A;
reg [1:0] B;

twoBitAdd top(Cout, Sum, A, B, Cin);

`include "testfixture.verilog"
endmodule
```

Figure 4.2: Verilog code for a DUT/testbench simulation environment

There are three Verilog simulators of interest to this CAD flow. They are:

**Verilog-XL:** This is an interpreted simulator from Cadence. Interpreted means that there is a run-time interpreter executing each Verilog instruction and communicating with the event-queue. This is an older simulator and is the reference simulator for the Verilog-1995 standard. Because it is the reference simulator for that standard it has not been updated to use some of the more modern features of Verilog, and because it is interpreted it is not the fastest of the simulators. But, it is well integrated into the Cadence system and is the default Verilog simulator for many tools.

**NC\_Verilog:** This is a compiled simulator from Cadence. This simulator compiles the Verilog code into a custom simulator for that Verilog program. It converts the Verilog code to a C program and compiles that C program to make the simulator. The result is that it takes a little longer to start up (because it needs to translate and compile), but the resulting compiled simulator runs much faster than the interpreted Verilog-XL. It is also compatible with a large subset of the Verilog-2000 standard and is being actively updated by Cadence to include more and more of those advanced features.

**VCS:** This is a compiled simulator from Synopsys. It is not integrated into the Cadence tools, but is integrated to some extent with the Synopsys tools so it is useful if you spend more time in the Synopsys portion of the design flow before using the back-end tools from Cadence. It is also a very fast simulator like NC\_Verilog.

## 4.1 Verilog Simulation of Composer Schematics

The simulators from Cadence are integrated with the **Composer** schematic capture tool. This means that if there are Verilog models for the cells you use in your schematic, you can simulate your schematics without leaving the **dfl** environment. All the cell libraries that we will use have Verilog models so all schematic simulation will be done through a Verilog simulator.

In order to do this you need a Verilog version of your schematic. That is, you need walk the schematic hierarchy and generate Verilog code that captures the module connectivity of the schematic. Whenever a node is encountered whose behavior can be described with a piece of Verilog code, you need to insert that Verilog code. The result is a hierarchical Verilog program that captures the functionality of the schematic. This process is known as *netlisting*, and the result is a structural Verilog description that is

also sometimes called a *netlist*. According to Figure 4.1 this netlist could also be known as the DUT. Once you have the DUT code, you need to write testbench code to communicate with the DUT during simulation.

Fortunately the Composer-Verilog integration environment will generate a simulatable netlist for you from the schematic, and also generate a template for a testbench file. The netlisting process walks through your hierarchical schematic and generates a program that describes the hierarchy. If the netlister encounters a **behavioral** cell view, that view contains Verilog code that describes that module's function so the code in the **behavioral** view is added to the netlist. If a schematic uses transistors from the **NCSU\_Analog\_Parts** or **UofU\_Analog\_Parts** libraries, those transistors are replaced with Verilog transistor models. The Verilog transistor primitives are built into the Verilog language and simulate the transistors as switches. We use the **cmos\_sch** view to signal to the netlister that this is a leaf cell that contains only transistors.

These two different description techniques for leaf cells (**behavioral** and transistor **cmos\_sch**) can be mixed in a single schematic, and in fact if the leaf cells have both **behavioral** (Verilog) and **cmos\_sch** (transistor) views you can choose which low-level behavior is simulated by manipulating the netlisting procedure in Composer. This is useful because each type of simulation has its own advantages and disadvantages. Behavioral modeling can simulate more quickly, and allows back-annotation of timing from other tools like synthesis tools though a Standard Delay Format (sdf) file (described in more detail in Section 4.4 and in Chapter 8). Switch level modeling can be a more accurate simulation of the low level details of the circuit's operation and can expose problems that are not visible in the more high-level behavioral simulation.

#### 4.1.1 Verilog-XL: Simulating a Schematic

As an example of simulating a schematic using Verilog-XL we'll use the two-bit adder from Chapter 3. To start Verilog-XL simulation you can use the CIW window by going to **Tools** → **Verilog Integration** → **Verilog-XL**.... The **Setup Environment** window appears in which the Run Directory, Library, Cell and View fields need to be filled. Press OK.

Or (the much easier way) open up the Composer schematic of the two-bit adder using the library browser and in the Composer schematic editing window, select **Tools** → **Simulation** → **Verilog-XL**. The **Setup Environment** window appears with all the fields filled. The **Run Directory** can be changed or left as default <**designname**>.**run1**. A dialog box for simulation of the two-bit adder from Chapter 3 is shown in Figure 4.3. Press **OK**.

*It is very important to have a separate run directory for each different design, but you can keep the same run directory for different simulations of the same design.*

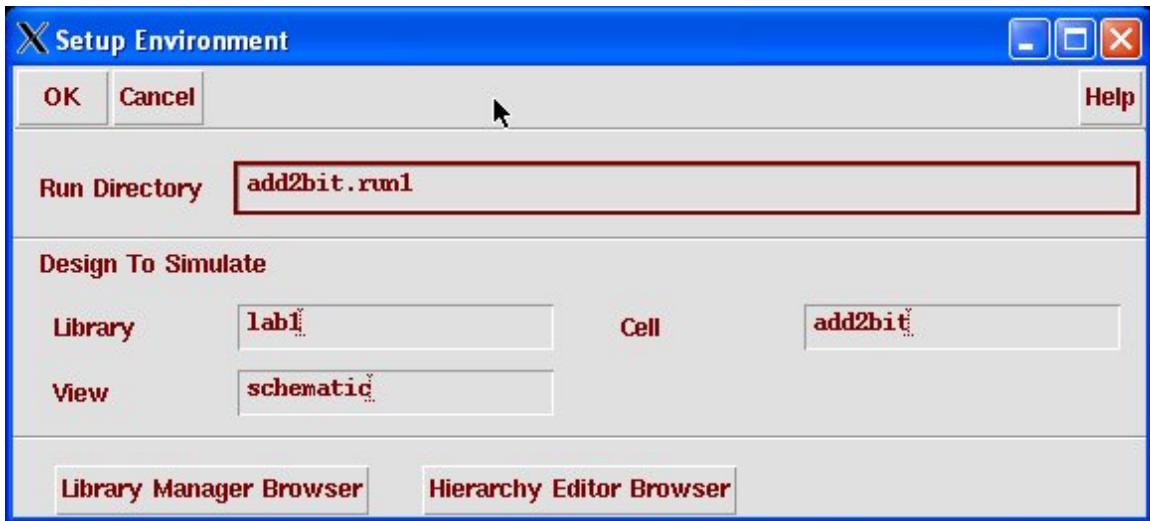


Figure 4.3: Dialog Box for Initializing a Simulation Run Directory

This will initialize the Verilog-XL simulator and bring up the Verilog-XL control window as shown in Figure 4.4. This is the window from which the simulation can be invoked. Most simulation activities can be controlled from the menus or from the widget icons on the left side of the panel. Hovering your mouse over those widgets will give you an idea of what they are. This manual won't go over all of them, but some playing around should reveal what most of them do.

Before starting simulation the environment needs to be set up and an input stimulus file for the circuit (a testbench, also called a *test fixture* by Verilog-XL) needs to be created.

### Setting up the Simulation Environment

Select **Setup → Record Signals....**. In the **Record Signal Options** window which appears you can change the signals that will be recorded during the simulation from **Top Level Primary I/O** to **All Signals** if you like. Saving only the top level I/O signals saves a lot of time and disk space, but only records the signals at the very top level of your circuit. Thus, you can't probe or observe any signals further down in the circuit hierarchy. If you want to be able to see circuit values inside the sub-circuits you should save **All Signals**.

Note: To make changes to the **Record Signal Options** later on, make sure that the interactive simulation is stopped. If it is not stopped then select **Simulation → Finish Interactive** or press the widget with the black square

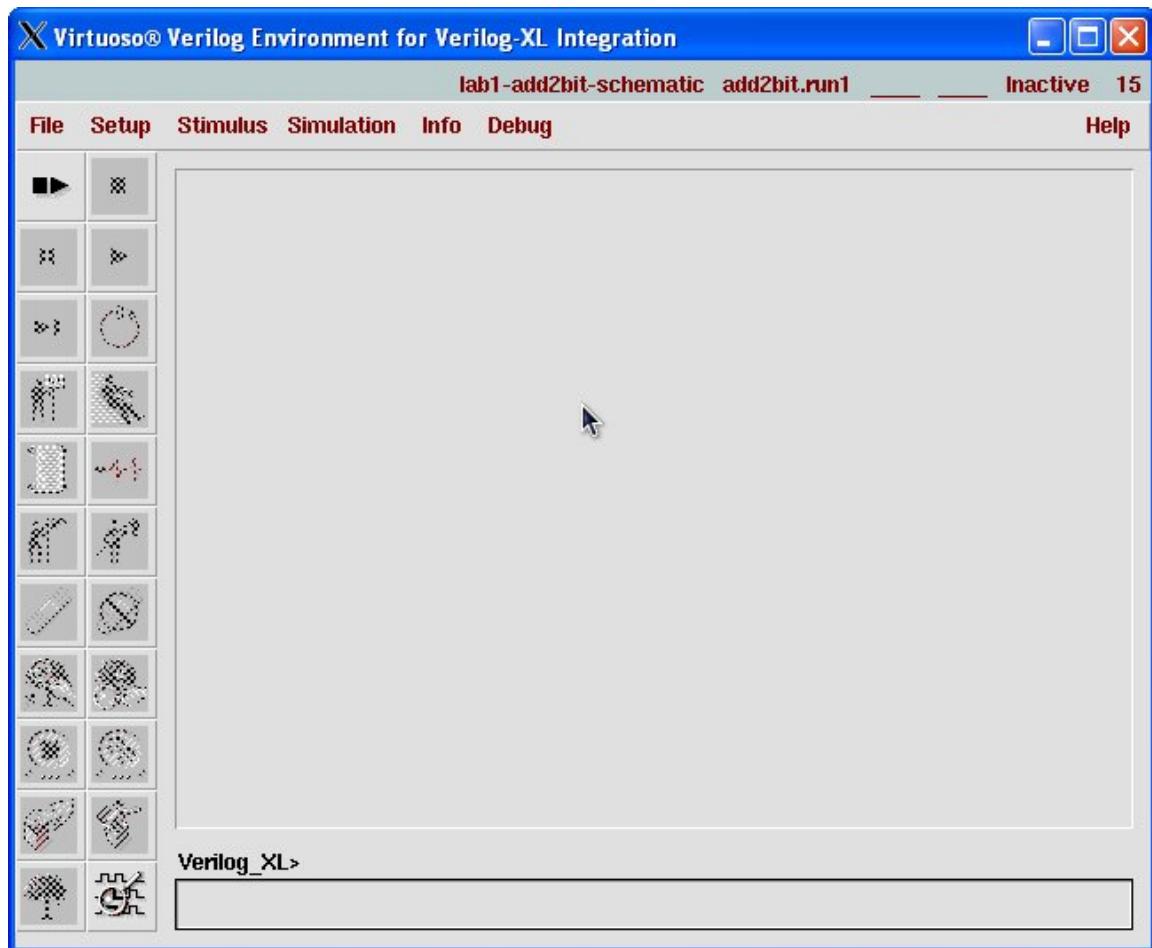


Figure 4.4: The Initial Verilog-XL Simulation Control Window

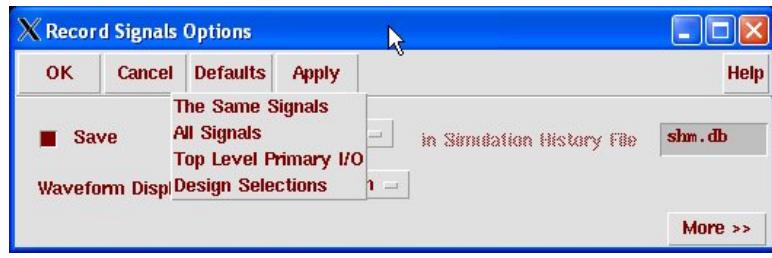


Figure 4.5: The Record Signals Dialog Box



Figure 4.6: Dialog to Create a New Test Fixture Template

which stops the current simulation.

### Defining the Test Fixture (testbench)

Select **Stimulus** → **Verilog...** and a prompt window appears asking if you wish to create a template (Figure 4.6). Select **Yes**.

All the steps in setting up the test fixture file must be completed before starting Interactive Simulation. If interactive simulation is already started, select **Simulation** → **Finish Interactive** or press the stop (black square) widget button.

In the Stimulus Options window which appears (See Figure 4.7) select **copy**. In the **Copy From:** frame, select the **File Name** from the list as **testfixture.Verilog**. The **File Name** in the **Copy To:** frame can be changed or left as the default **testfixture.new**. The **Design Instance Path** should not be changed from **test.top**. This is the DUT structure that is created by the Composer netlisting procedure. The uppermost cell is named **test**. This cell contains one instance of your top-level circuit which is given an instance name of **top**. The other component within the **test** module is your testbench code. The template for this testbench will be created when you press **Apply**.

*Verilog lets you access signals in a hierarchy using a “.” between levels of the hierarchy. A wire **foo** inside the instance **top** could be accessed using **test.top.foo**, for example.*

Now select **Edit** mode and choose **testfixture.new** (or the file name you have given to the test fixture) from the **Field Name**. Select **Make Current Test Fixture** and **Check Verilog Syntax** and press **Apply** or **OK**.

The default editor (most likely **emacs**) will open up. Use this editor to type in the Verilog code that you want to use as your test fixture. Then save the test fixture and close the editor. The original test fixture template should look something like that in Figure 4.8. The interface signals are found on the symbol and repeated here in the test fixture template. Your test code

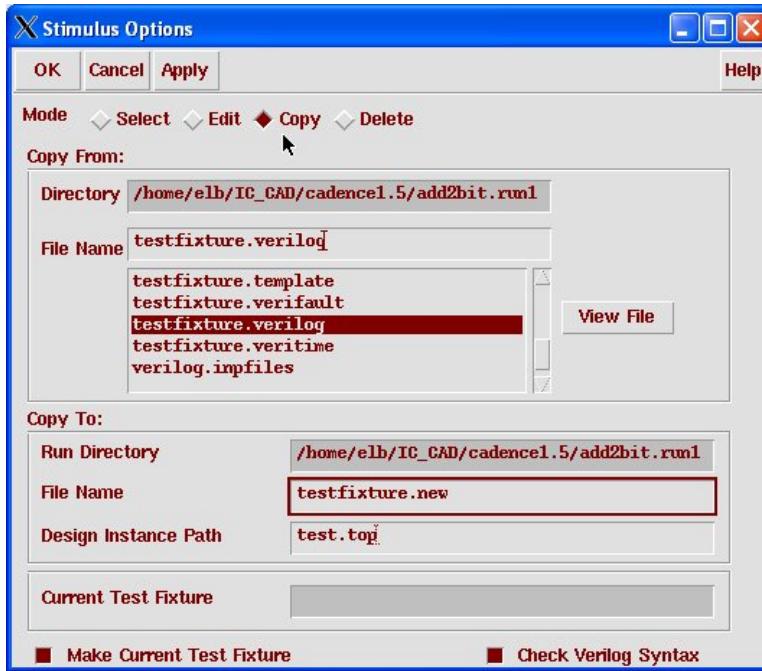


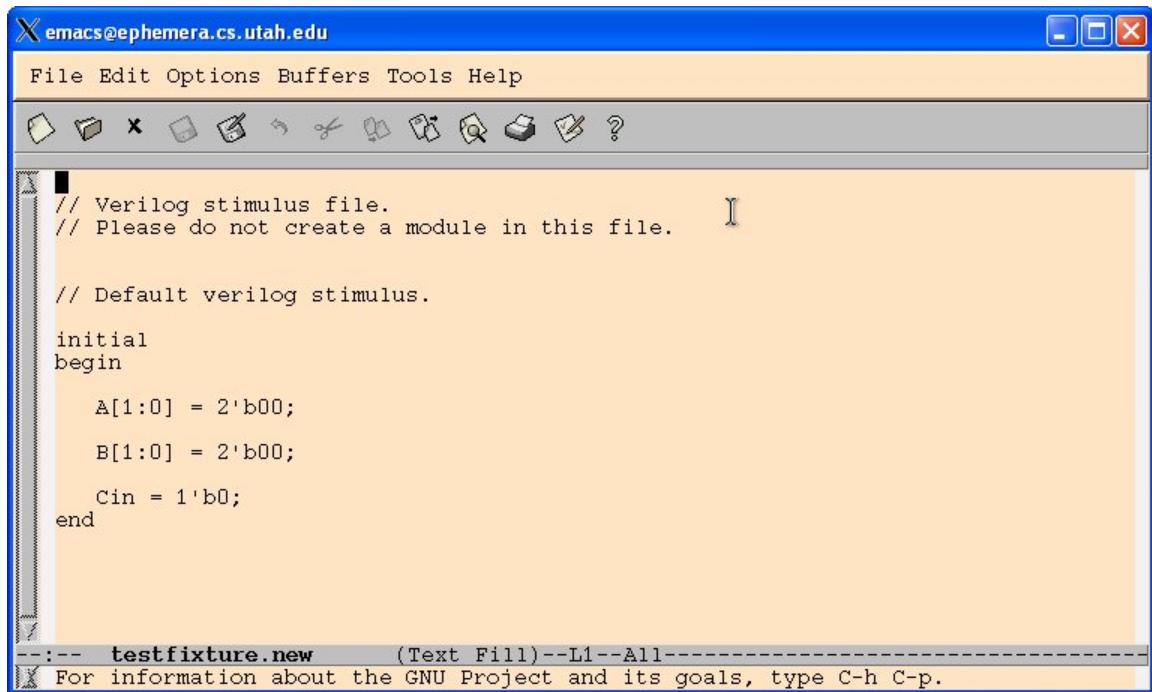
Figure 4.7: The Verilog-XL Stimulus Options Form

should go after the interface signal definitions and before the **end** statement. This piece of Verilog code is an **initial** block which is executed when you simulate the netlist assembled by Composer.

An example of a complete simple test fixture for the two bit adder is shown in Figure 4.9.

Some important things to notice about the test fixture in Figure 4.9 are:

- Verilog comments are either C-style with `/*` and `*/` bracketing the comment, or a pair of backslashes `\\"` denoting a comment to the end of the line. Please use comments in your test fixture!
- A **\$display** statement is the equivalent of a C `printf` statement in a Verilog program. These can be very helpful so that you can see how things are progressing during your simulation.
- A good Verilog testbench *always* checks for the correct value of the outputs in the testbench and prints something if the values are not correct. You can see this in the **if** statements in the testbench code. These statements check for a correct value and print an error message if the value is not correct. *All your testbenches should be self-checking like this!* Waveforms are a great way to debug some things and for the



```

// Verilog stimulus file.
// Please do not create a module in this file.

// Default verilog stimulus.

initial
begin

    A[1:0] = 2'b00;
    B[1:0] = 2'b00;
    Cin = 1'b0;
end

```

--:-- testfixture.new (Text Fill) --L1--A11-----  
 For information about the GNU Project and its goals, type C-h C-p.

Figure 4.8: Test Fixture Template for the Two Bit Adder

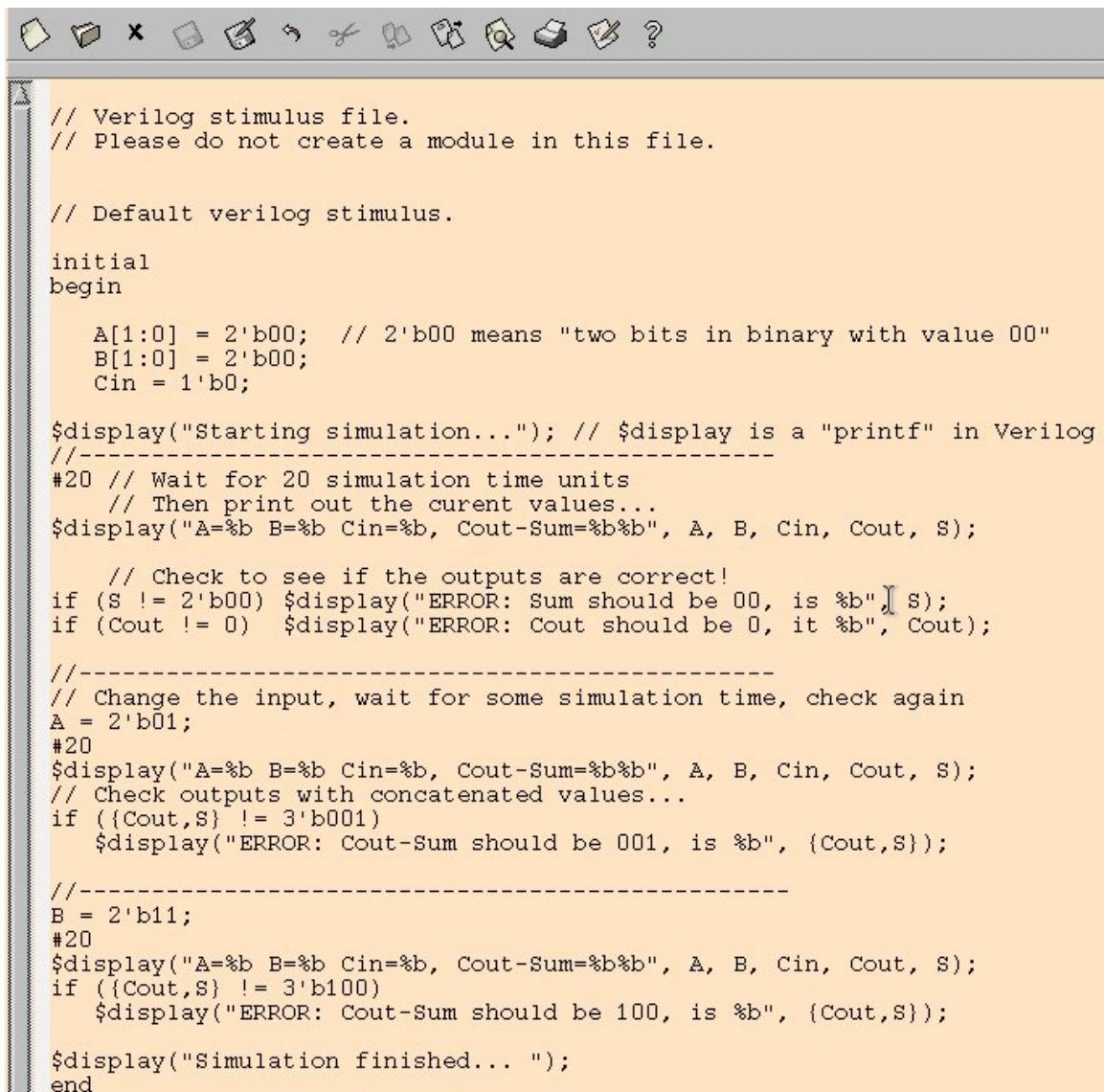
designer of the circuit to see what's going on, but they are really bad for checking whether a complete circuit is doing the right thing, or whether a change in the circuit has caused a change in behavior. Once you have a self-checking testbench you can make improvements and re-run the simulation to see if the behavior has changed.

*The syntax check is  
checking for  
Verilog-1995 syntax*

When you save the testbench code and exit the editor, the system will check the Verilog for correct syntax. If you don't pass the syntax check you'll need to reopen the testbench file and fix the errors. When you successfully dismiss the testfixture dialog box by selecting the new test fixture you are ready for simulation.

An example of a different type of testbench is shown in Figure 4.10. In this testbench loops are used to test the two bit adder exhaustively, and the checks are computed using Verilog to compute the answer instead of the testbench author writing down each result separately. Note that the integer variables required for the **for** loops are defined outside the **initial** block. Also note that there are a variety of syntax choices and shorthands available for referring to the input vectors.

Still another style of writing a testbench is shown in Figure 4.11. In this testbench the values of the inputs, and the values that should be checked



The screenshot shows a window with a toolbar at the top containing icons for file operations like open, save, and search. The main area contains Verilog code for a two-bit adder simulation. The code includes comments explaining the purpose of each section and how it checks the correctness of the adder's output.

```
// Verilog stimulus file.  
// Please do not create a module in this file.  
  
// Default verilog stimulus.  
  
initial  
begin  
  
A[1:0] = 2'b00; // 2'b00 means "two bits in binary with value 00"  
B[1:0] = 2'b00;  
Cin = 1'b0;  
  
$display("Starting simulation..."); // $display is a "printf" in Verilog  
//----------------  
#20 // Wait for 20 simulation time units  
// Then print out the current values...  
$display("A=%b B=%b Cin=%b, Cout-Sum=%b%b", A, B, Cin, Cout, S);  
  
// Check to see if the outputs are correct!  
if (S != 2'b00) $display("ERROR: Sum should be 00, is %b", S);  
if (Cout != 0) $display("ERROR: Cout should be 0, it %b", Cout);  
  
//----------------  
// Change the input, wait for some simulation time, check again  
A = 2'b01;  
#20  
$display("A=%b B=%b Cin=%b, Cout-Sum=%b%b", A, B, Cin, Cout, S);  
// Check outputs with concatenated values...  
if ({Cout,S} != 3'b001)  
$display("ERROR: Cout-Sum should be 001, is %b", {Cout,S});  
  
//----------------  
B = 2'b11;  
#20  
$display("A=%b B=%b Cin=%b, Cout-Sum=%b%b", A, B, Cin, Cout, S);  
if ({Cout,S} != 3'b100)  
$display("ERROR: Cout-Sum should be 100, is %b", {Cout,S});  
  
$display("Simulation finished... ");  
end
```

Figure 4.9: An Example Test Fixture for the Two Bit Adder

```
// Default Verilog stimulus.
integer i,j,k;
initial
begin

A[1:0] = 2'b00;
B[1:0] = 2'b00;
Cin = 1'b0;

$display("Starting simulation...");

for(i=0;i<=3;i=i+1)
begin
  for(j=0;j<=3;j=j+1)
  begin
    for(k=0;k<=1;k=k+1)
    begin
      #20
      $display("A=%b B=%b Cin=%b, Cout-Sum=%b%b", A, B, Cin, Cout, S);
      if ({Cout,S} != A + B + Cin)
        $display("ERROR: Cout-Sum should equal %b, is %b",
                 (A + B + Cin), {Cin,S});
      Cin=~Cin; // invert Cin
    end
    B[1:0] = B[1:0] + 2'b01; // add the bits
  end
  A = A+1; // shorthand notation for adding
end

$display("Simulation finished... ");
end
```

Figure 4.10: Another Test Fixture for the Two Bit Adder

for on the outputs, are held in external text files one value per line. These values might, for example, have been generated by some other application like Matlab or through some C, Java, python, or other program that you write to generate values to check for.

In this test fixture arrays are defined to hold the inputs for **A** and **B**, and for the results of the two bit addition. These Verilog arrays are initialized using the **\$readmemb** command to read memory values in binary format (actually it's ASCII format with 1 and 0 as the digits, as opposed to **\$readmemh** where the values in the test file are hex digits). The values in the data files are formatted with one array row of data on each line of the data file.

Once the data values are loaded into the arrays the simulation walks through all the values in the test arrays and check the answer against the value in the corresponding location in the **resultsarray**.

Of course, experienced Verilog programmers may have lots of additional ideas about how to write great test fixtures. These are just some ideas for how to think about your testbenches. Remember that all testbenches should be check for the correct answer in the testbench code!

### Running the Simulation

Once you have a testbench that passes the syntax check, and you have set up the signals that you want to record, you can run the simulation. Start the Verilog simulation by selecting **Simulation → Start Interactive** or pressing the widget button with the square and right-facing **play** button (upper left of the widgets). This netlists your design, check the netlist for errors, and prepares the netlist and test fixture for simulation.

After you have done this once in the current *run directory* you will get a dialog box like that in Figure 4.13 asking if you want to re-netlist the design or use the old netlist. Usually you want to re-netlist at this point so that any changes you've made since that last time you simulated are updated in the simulation netlist.

The results for the netlisting in my example looks like that in Figure 4.14. Note that for each of the basic gates I used in my schematic the netlistter chose **behavioral** views of those schematics. Later we'll see how to change the netlisting order so that the netlisting process will get the transistor switch-level views.

Note that once you've successfully netlisted the Verilog and initialized the simulator all the rest of the widgets that used to be grayed out become active. The Verilog window now looks like that in Figure 4.15.

Now that you're in "interactive" mode, you can run the simulation us-

```
// Default Verilog stimulus.
reg [1:0] ainarray [0:4]; // define memory arrays
reg [1:0] binarray [0:4]; // to hold input and result
reg [2:0] resultsarray [0:4];
integer i;

initial
begin

/* A simple Verilog test fixture for testing a 2-bit adder */

$readmemb("ain.txt", ainarray); // read values into
$readmemb("bin.txt", binarray); // arrays from files
$readmemb("results.txt", resultsarray);

A[1:0] = 2'b00; // initialize inputs
B[1:0] = 2'b00;
Cin = 1'b0;

$display("Starting...");
#10
$display("A = %b, B = %b, Cin = %b, Sum = %b, Cout = %b",
         A, B, Cin, Sum, Cout);

for(i=0; i<=4; i=i+1) // loop through all values in arrays
begin
    A = ainarray[i]; // set the inputs
    B = binarray[i]; // from the memory arrays
    #10
    $display("A = %b, B = %b, Cin = %b, Sum = %b, Cout = %b",
             A, B, Cin, Sum, Cout);
    // check against results array
    if ({Cout,Sum} != resultsarray[i])
        $display("Error: Sum should be %b, is %b instead",
                 resultsarray[i],Sum);
end
$display("...Done");
$finish;
end
```

Figure 4.11: A Test Fixture Using Values From External Files

01	01	010
10	10	100
11	11	110
00	11	011
01	11	100

(a) ain.txt                  (b) bin.txt                  (c)  
                                  results.txt

Figure 4.12: Data files used in Figure 4.11

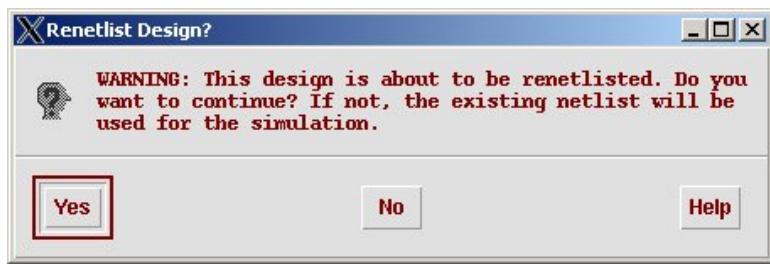
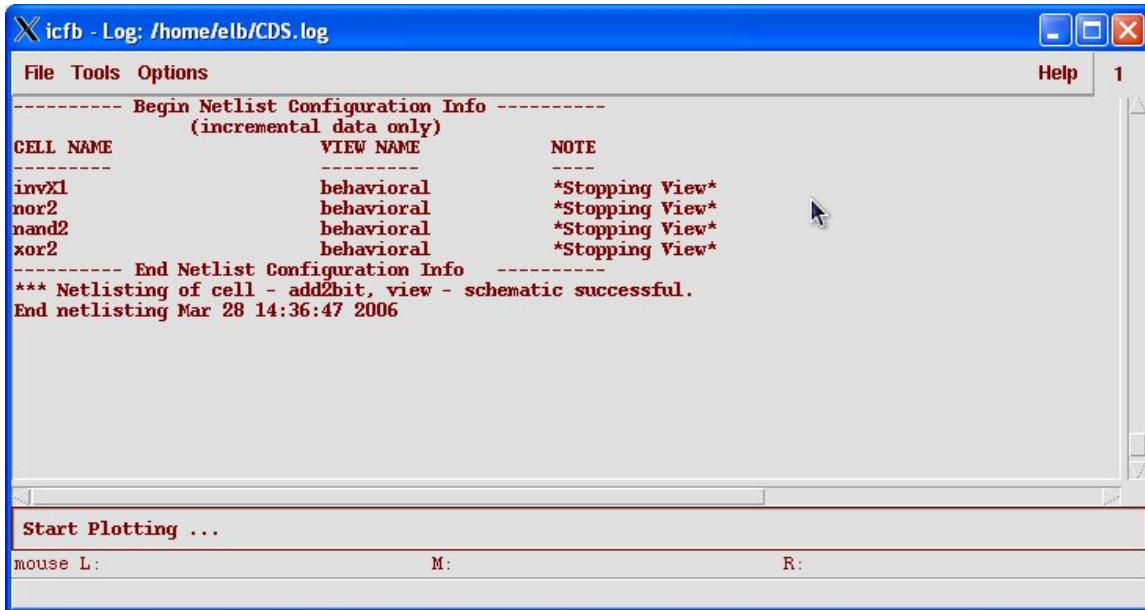


Figure 4.13: Dialog box for re-netlisting a previously netlisted design



The screenshot shows a window titled "icfb - Log: /home/elb/CDS.log". The window contains a table of netlist configuration information:

CELL NAME	VIEW NAME	NOTE
invX1	behavioral	*Stopping View*
nor2	behavioral	*Stopping View*
nand2	behavioral	*Stopping View*
xor2	behavioral	*Stopping View*

Below the table, the log message states: "End Netlist Configuration Info", "\*\*\* Netlisting of cell - add2bit, view - schematic successful.", and "End netlisting Mar 28 14:36:47 2006".

At the bottom of the window, there is a status bar with the text "Start Plotting ...", and mouse coordinates L:, M:, R:.

Figure 4.14: The Netlisting Log for the Two Bit Adder

ing the testbench you designed by selecting **Simulation** → **Continue** or by pressing the **play** widget (the right-facing triangle). This runs your testbench on the Composer-generated netlist. The result of the simulation on the testbench from Figure 4.9 is shown in Figure 4.16. You can see that the **\$display** statements have printed the simulation values, and none of the **ERROR** statements have printed, which means that the circuit passed this simulation with correct results.

### Printing Verilog-XL Output

The output in the Verilog-XL window is available through the **Edit** → **View Log File** → **Simulation** menu choice. This will bring up the contents of the Verilog-XL in a window where you can **Save-As** any file you like. This is just a text file that has the results of the **\$display** statements in your testbench.

### SimVision Waveform Viewer

Now that you have a successful simulation, you can, if you wish, look at the waveforms. Waveforms are a good way to get certain types of information from the simulation, but hopefully you've checked for enough values

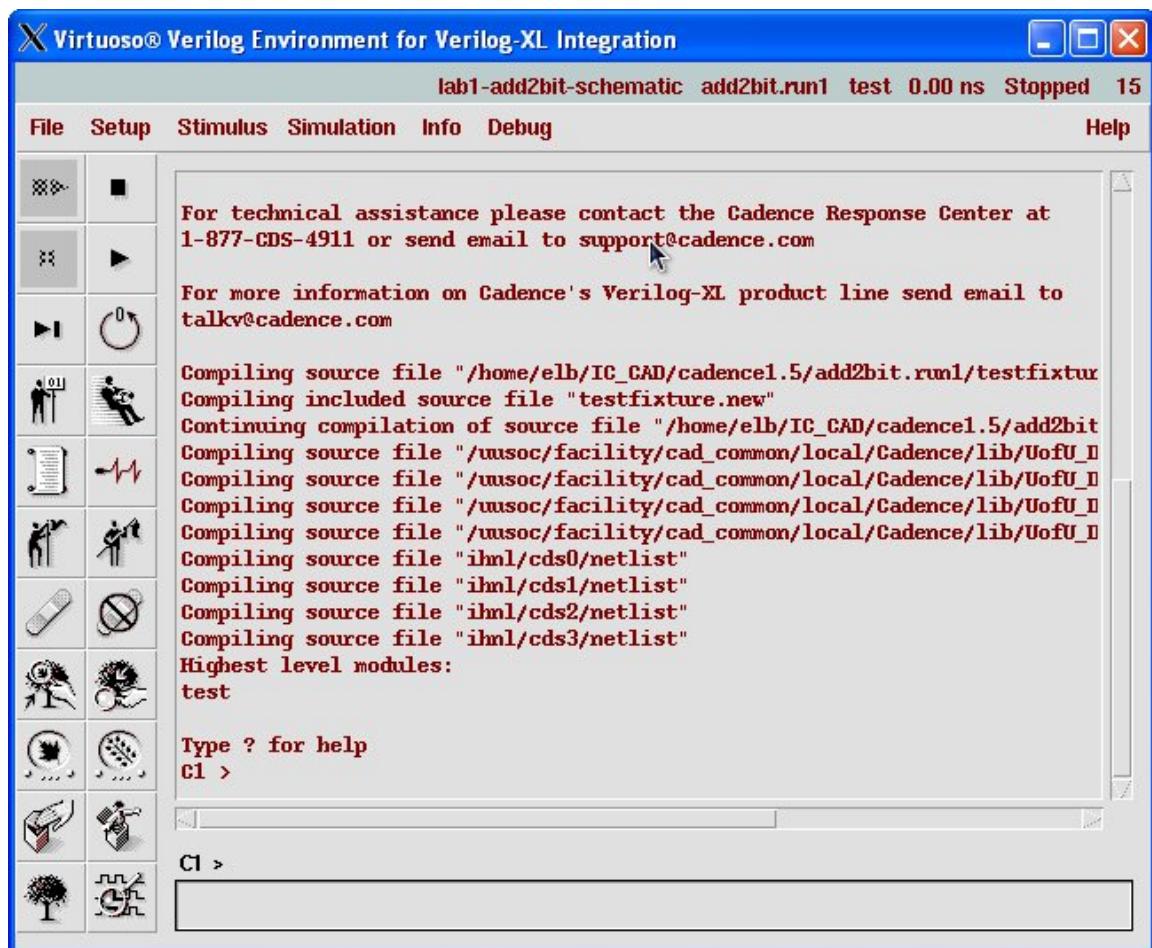


Figure 4.15: The Verilog-XL Window After Netlisting

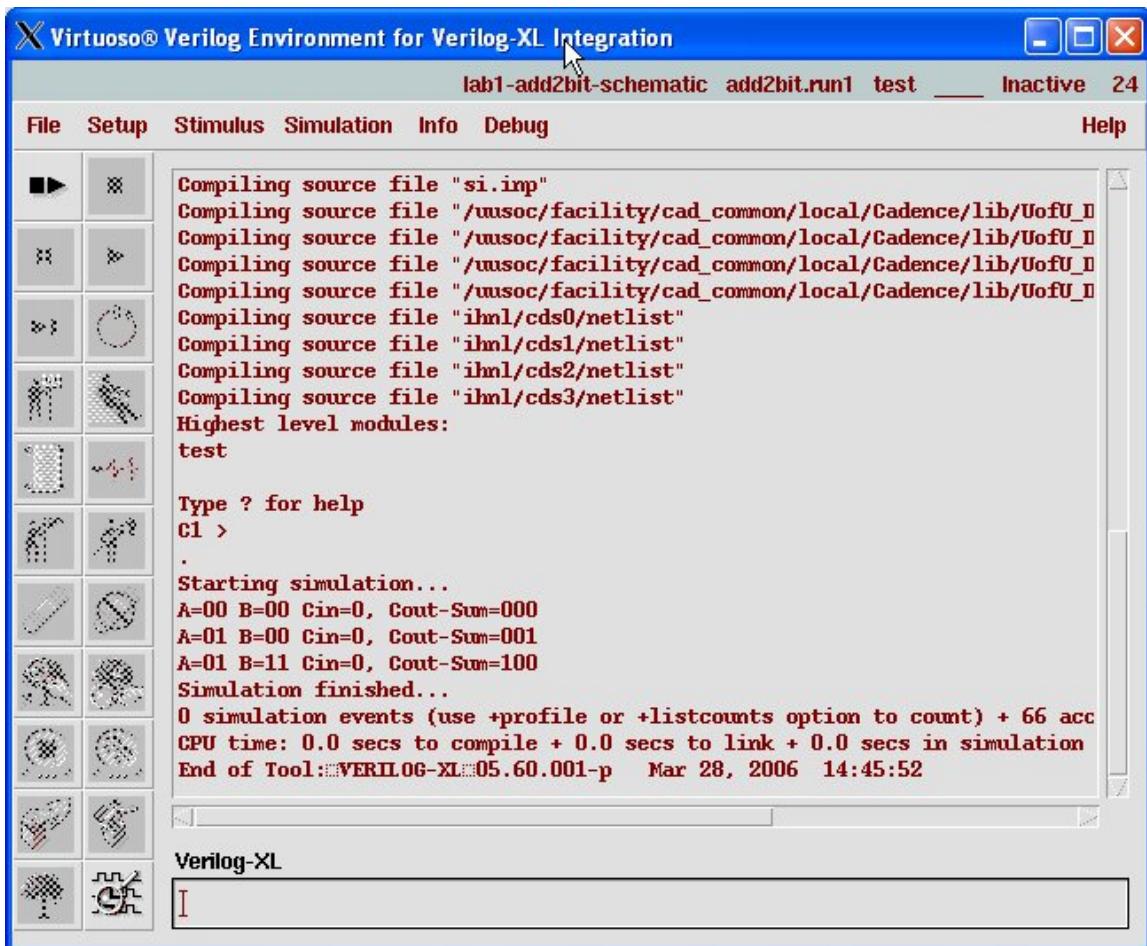


Figure 4.16: Result of Running with the Testbench from Figure 4.9

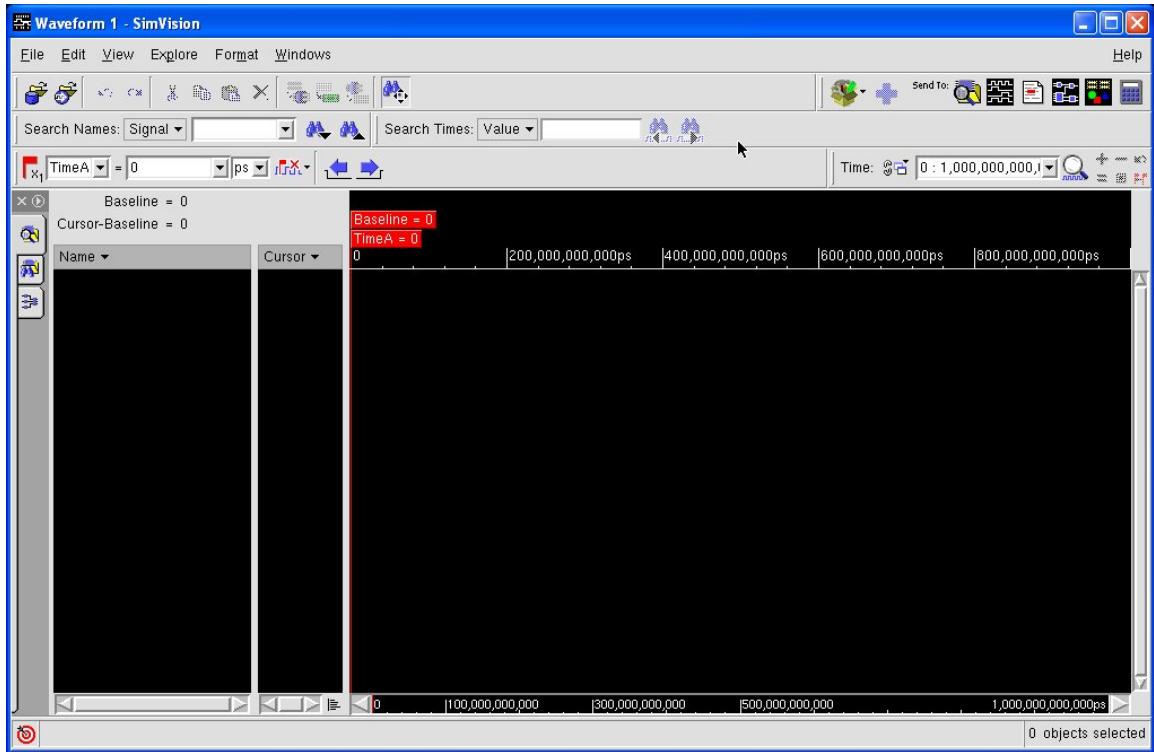


Figure 4.17: Waveform Window Without any Signals

in your testbench that you already know if your circuit is working or not. The waveform viewer is a good place for debugging if your circuit isn't completely correct though. Especially if you've selected the **Record All Signals** option, you can use the waveform viewer and navigation system to look at signals deep inside your circuit to see where things have started to fail. Standard debugging techniques apply here: starting with the incorrect output and working backwards in the circuit to figure out what caused that output to be incorrect is a great way to start. The waveform viewer attached to Verilog-XL by default is SimVision.

To start the SimVision waveform viewer after running a simulation, select **Debug** → **Utilities** → **View Waveform...** or pressing the waveform viewer widget (bottom right in the widget array). The **Waveform 1 - SimVision** window appears as shown in Figure 4.17.

No waveforms appear yet. You need to select which signals you want to appear in the waveform window. You do this through the **Design Browser**. Select **Windows** → **New** → **Design Browser** or press the **Design Browser** button (it looks like a folder with a magnifying glass in front of it), and the **Design Browser 1 SimVision** window appears as shown in Figure 4.18.

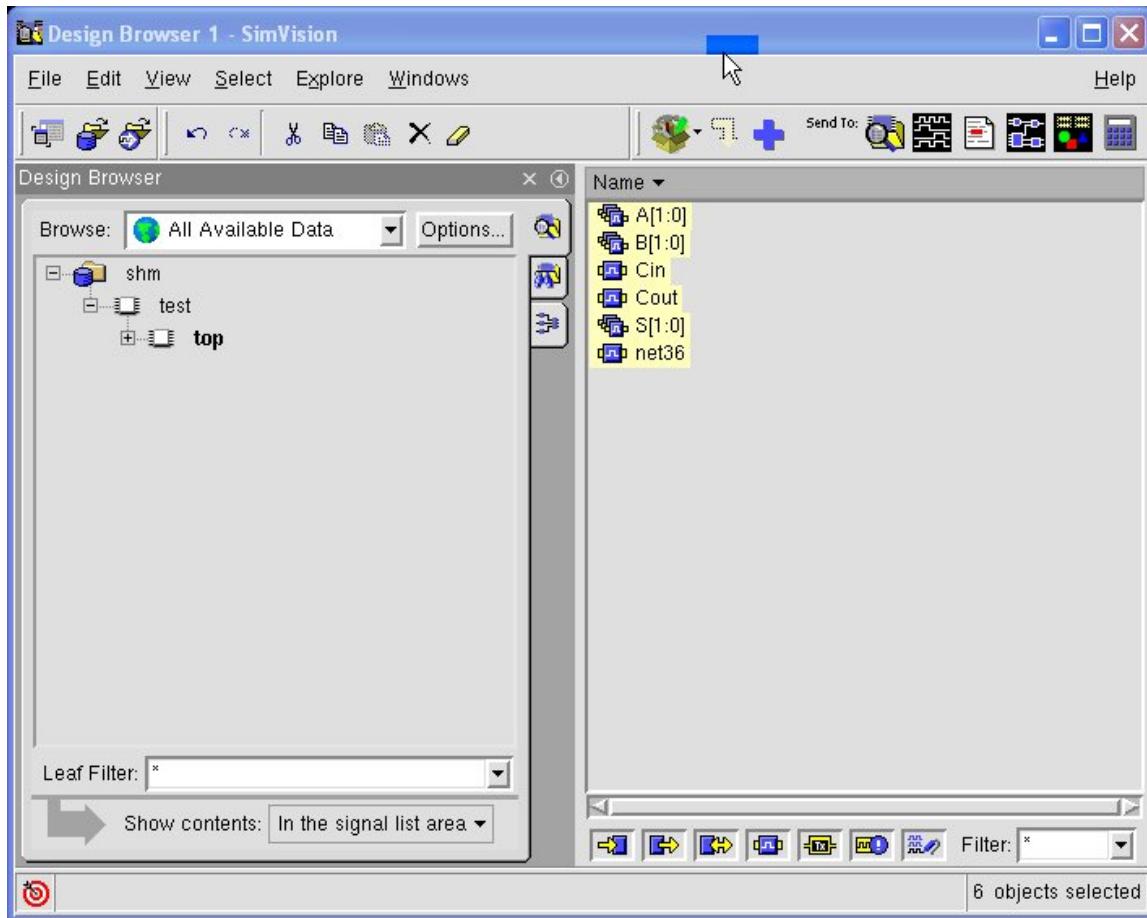


Figure 4.18: Design Browser with Signals Selected

Using this window you can navigate the hierarchy of your circuit to select the specific wires that you'd like to see in the waveform window. Once selected, you send the values to the waveform window using the widget that looks like waveforms in a black square. You can also right-click after selecting the signals you want to get a menu with the choice to **send to Waveform Window**. The **Design Browser** with all the top level I/O signals in the two bit adder selected is shown in Figure 4.18.

Once you send waveforms to the **Waveform** window, you'll see them as in Figure 4.19. This set of waveforms was generated using the exhaustive testbench from Figure 4.10. I've also zoomed out to see the entire test using the controls on the right next to the magnifying glass icon. The widget with the equal sign (=) will zoom out to fit the entire simulation on the X-axis. The + and - buttons will zoom in and out. You can also set cursors and measure time between cursors using these controls.

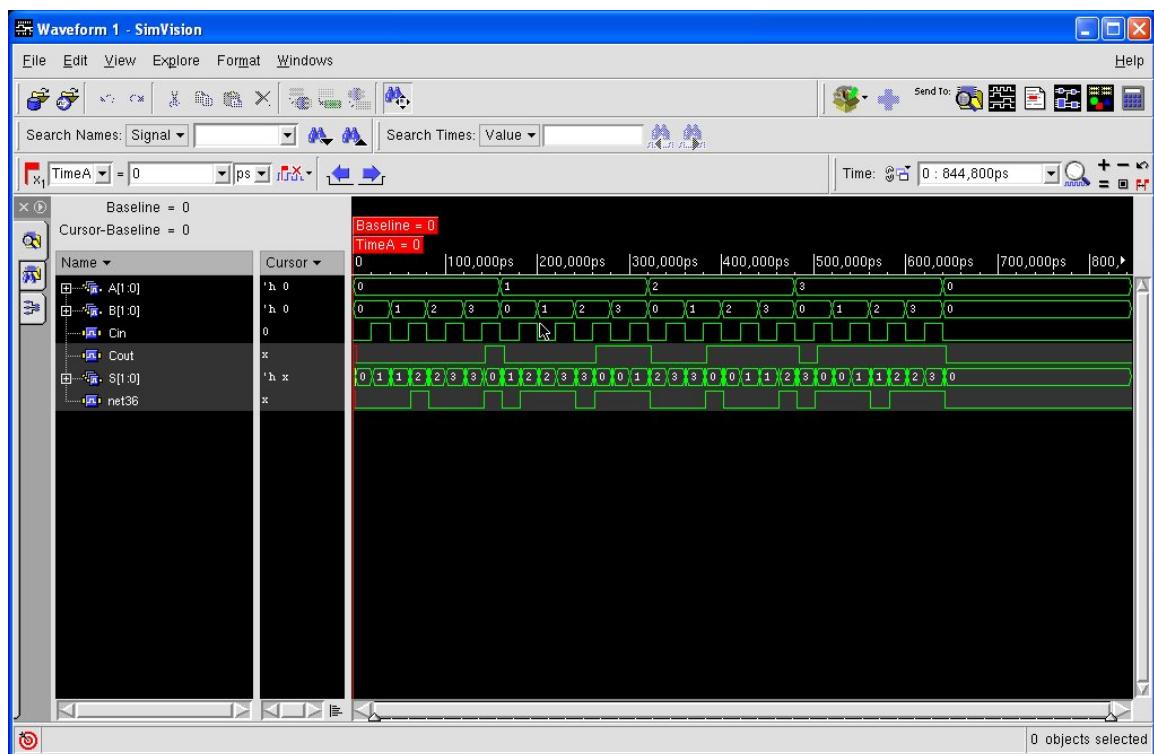


Figure 4.19: Waveform Window Showing Exhaustive Test Results

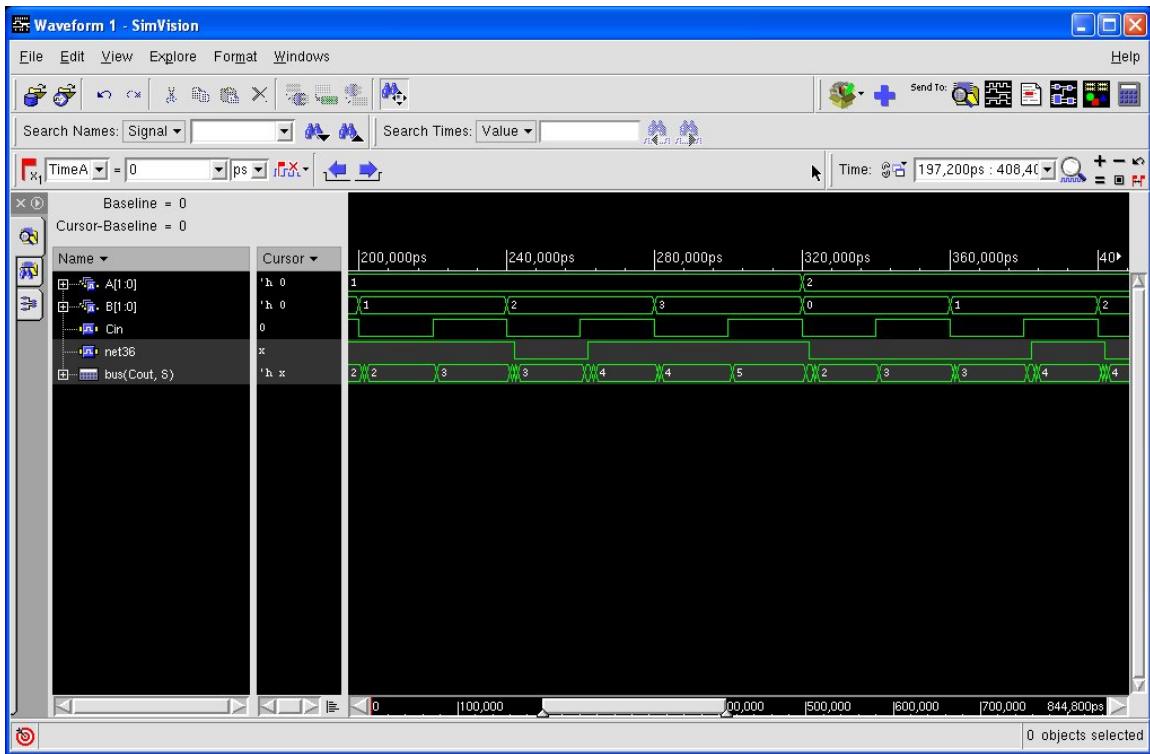


Figure 4.20: Waveform Window Showing Outputs as a Bus

However, the outputs are a little hard to read in this window because the carry-out (**Cout**) and sum (**Sum**) outputs are on different waveforms. It would be easier if these were combined into a single bus so that you could read the output value as a three-bit number. You can do this by selecting the **Cout** and **Sum** traces in the **Waveform** window and collecting them into a **bus** using **Edit** → **Create** → **Bus** or using the **Create Bus** option in the right-click menu once the traces are selected. The result, zoomed in to a closer view, is shown in Figure 4.20. In this waveform the output is collected into a bus and reads as a three-bit output value.

### Printing Waveform Outputs

Output from the SimVision waveform viewer can be printed using the **File** → **Print Window...** menu choice. This will bring up yet another different print dialog box, as shown in Figure 4.21. You can fill in your name and other information to have it printed along with the waveforms. At the top of the dialog box you can select a Unix/Linux print command (the default **lpr -l** works fine), or you can select **Print to file:** and give a file name. The

*With no -P argument, lpr uses the printer defined in your \$PRINTER environment variable*

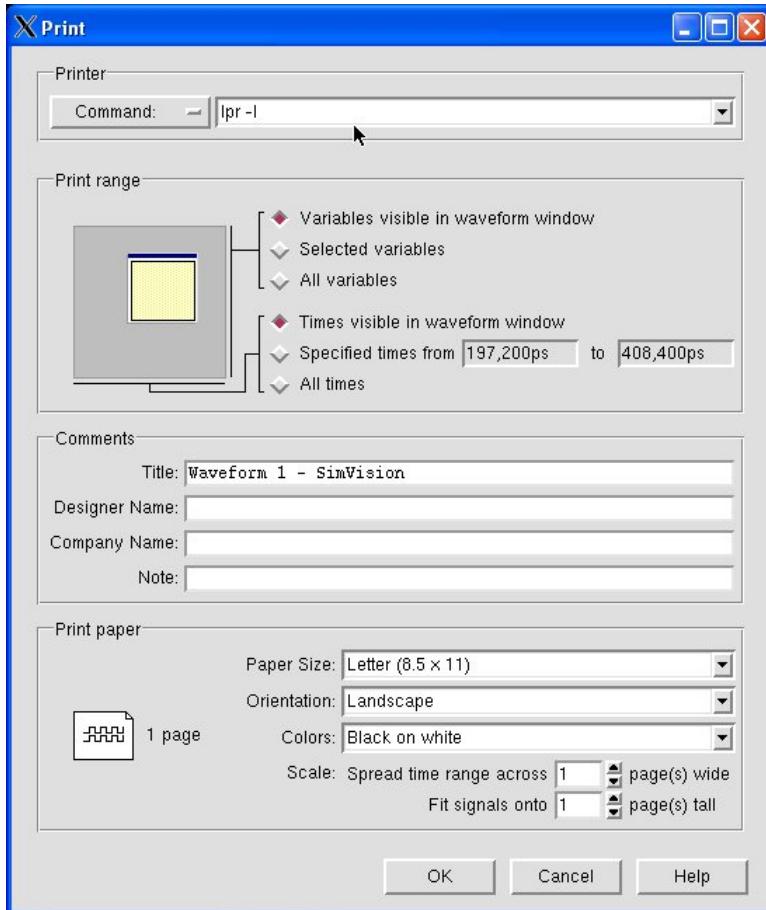


Figure 4.21: Printing Dialog Box for SimVision

result is a postscript file. I haven't found hand-modification required for the postscript produced by SimVision.

#### 4.1.2 NC\_Verilog: Simulating a Schematic

As an example of simulating a schematic we'll again use the two-bit adder from Chapter 3. To start NC\_Verilog simulation you can use the CIW window by going to **Tools** → **Verilog Integration** → **NC\_Verilog....**. The **Verilog Environment for NC\_Verilog Integration** window appears in which the Run Directory, Library, Cell and View fields need to be filled in.

Or (the much easier way) open up the Composer schematic of the two-bit adder using the library browser and in the Composer schematic editing window, select **Tools** → **Simulation** → **NC\_Verilog**. The **Verilog Environment**

*It is very important to have a separate run directory for each different design, but you can keep the same run directory for different simulations of the same design.*

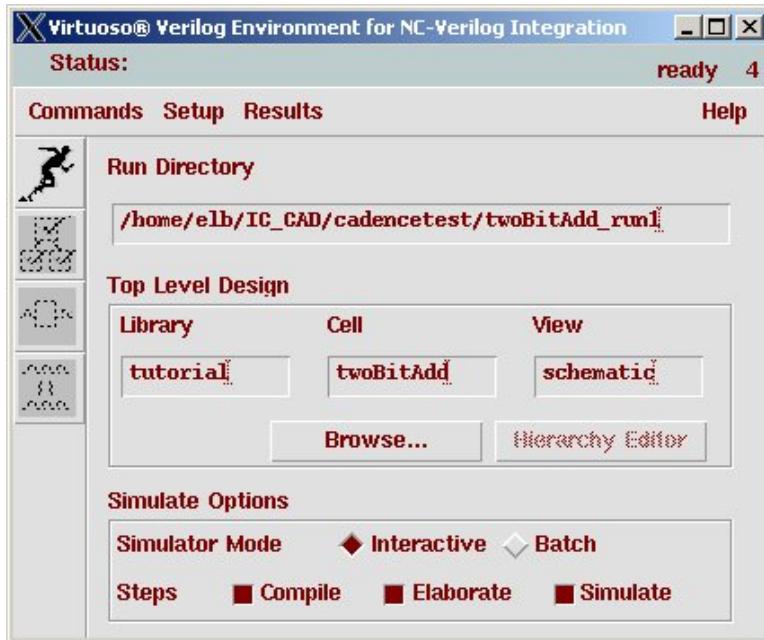


Figure 4.22: Dialog Box for Initializing a Simulation Run Directory for NC\_Verilog

ment for NC\_Verilog Integration window appears with all the fields filled in. If the fields are not filled in, you can use the **Browse** button to open a small **library manager** window to select the library, cell, and schematic view.

The **Run Directory** can be changed or left as default <designname>.run1. Note that the default run directory name for NC\_Verilog has an underscore rather than the dot used in the Verilog-XL environment. It's not a terribly important difference, but it can help you keep track of which run directory is which if you're using both simulators. A dialog box for simulation of the two-bit adder from Chapter 3 is shown in Figure 4.22.

Once the **Verilog Environment for NC\_Verilog Integration** dialog has been filled in, initialize the design by selecting **Commands** → **Initialize Design** or by selecting the **Initialize Design** widget that looks like a sprinter starting a race.

This will initialize the NC\_Verilog simulator and make some other widgets in the **Verilog Environment for NC\_Verilog Integration** dialog active, including the **Generate Netlist** widget. Selecting that widget (or issuing that command using the **Commands** menu) will generate a simulation netlist for the schematic.

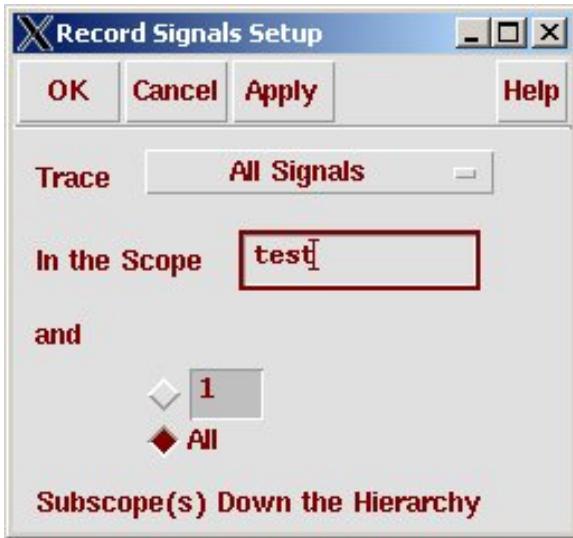


Figure 4.23: The Record Signals Dialog Box

Before starting simulation the environment needs to be set up and an input stimulus file for the circuit (a testbench, also called a *test fixture* by NC\_Verilog) needs to be created.

### Setting up the Simulation Environment

Select **Setup → Record Signals...**. In the **Record Signal Setup** window which appears you can change how many levels of hierarchy will have their internal signals saved. If you leave the default setting of 1 you will only get the top-level I/O signals from your schematic. The **test** scope in the netlist is the wrapper that the NC\_Verilog integration system builds with one instance of your circuit (with label **top**) and the test fixture. If you would like to be able to see the values of signals further down in your circuit hierarchy, change the number or select **All** to have all hierarchical signals saved. See Figure 4.23 for details.

### Generating the Simulation Netlist

Next generate the simulation netlist by selecting the **Generate Netlist** widget (the widget that looks like a tree of boxes with check marks in them), or by using the **Commands → Generate Netlist** menu choice. The result is that a netlist is generated with a top-level simulation macro named **test** that contains one instance of your circuit (the DUT) labeled **top** and a template for the test fixture code that you will fill in to drive your simulation.

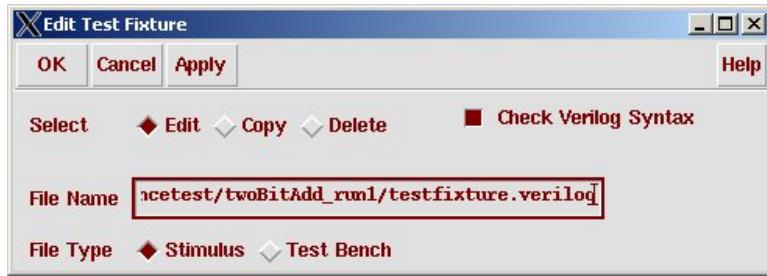


Figure 4.24: Dialog to Create a New Test Fixture Template

If you're re-simulating a design in the same run directory, you'll see a re-netlisting dialog box like that in Figure 4.13. As in the Verilog-XL case, it's almost always the case that you should re-netlist to make sure you're simulating the most recent changes to your circuit. The result of the netlisting command should be very similar to that shown in Figure 4.14 because the same netlister is used for both Verilog-XL and NC\_Verilog. This step also enables more of the widgets and commands in the **Verilog Environment for NC\_Verilog Integration** window.

### Defining the Test Fixture (testbench)

Select **Commands → Edit Test Fixture**. This brings up the **Edit Test Fixture** window with the default test fixture file name of **testfixture.Verilog** already selected as shown in Figure 4.24. The **File Type** should be **Stimulus** and it's a good idea to select the **Check Verilog Syntax** box. The **Test Bench File Type** is the top-level module named **test** that is created with the instance of your circuit instantiated as the DUT named **top**. You can view this file if you're curious. Your stimulus code is inserted with a Verilog **'include** statement in the **Test Bench**

You can edit the stimulus portion of the test bench by selecting **edit** in the **Edit Test Fixture** dialog and either selecting **Apply** or **OK**. The default editor (most likely **emacs**) will open up. Use this editor to type in the Verilog code that you want to use as your test fixture. Then save the test fixture and close the editor. The original test fixture template should look something like that in Figure 4.8 from Section 4.1.1. The interface signals are found on the symbol and repeated here in the test fixture template. Your test code should go after the interface signal definitions and before the **end** statement. This piece of Verilog code is an **initial** block which is executed when you simulate the netlist assembled by Composer.

For examples of test fixtures for the two bit adder, see Figures 4.9, 4.10, and 4.11 in Section 4.1.1. Once you have entered your test fixture code and

saved the result without Verilog errors you are ready to simulate.

Note that even though NC\_Verilog includes many Verilog-2000 features, the syntax checker for the test fixture appears to be checking for the Verilog-1995 syntax that Verilog-XL prefers. I believe, but have not tested completely, that if you want to use Verilog-2000 features in your test fixture you can remove the **Check Verilog Syntax** option when editing. However, this will open the door for Verilog syntax errors of all types to creep into your test fixture code so beware!

### Running the Simulation

Once you have a testbench that passes the syntax check, and you have set up the signals that you want to record, you can run the simulation. Start the Verilog simulation by selecting **Commands → Simulate** menu choice or pressing the **Simulate** widget button with the DUT rectangle with square waves on right and left denoting inputs and outputs.

This will analyze, elaborate, and compile the NC\_Verilog simulator for your design. The result is that two new windows pop up as shown in Figures 4.25 and 4.26: the **Console - SimVision** and the **Design Browser 1 - SimVision**. The **Design Browser** is the same as seen in the Verilog-XL example and is shown after the **test** and **top** nodes have been expanded. The **Console** is a control console for the NC\_Verilog simulation. If you look carefully you'll see the command line that initializes the simulation which includes the information we set previously to include (**probe**) signals at all levels of the circuit hierarchy.

Once these windows are open you can select the signals that you would like to see using the **Design Browser**. Click the **Send to Waveform** widget in the **Design Browser** window to open a **Waveform** window. Then click on the **Run Simulation** button in any of the windows. This button looks like a white right-pointing triangle, or a standard “play” button from applications like audio players. The simulation runs and the waveforms appear in the **Waveform** window. You can zoom in and out, add or delete waveforms, and group or expand buses as described in Section 4.1.1. Printing a waveform from SimVision is also described in that Section. The result of simulating using the exhaustive test fixture from Figure 4.11 is shown in Figure 4.27.

The SimVision application is a very complex front end to the NC\_Verilog simulator. It's not just a waveform viewer as it is with Verilog-XL. From SimVision you can set break points, single step simulations, explore the circuit being simulated using a **Signal Flow Browser**, reference the simulation to the Verilog source code, and many other features. Please explore these features as you are simulating your Verilog code with NC\_Verilog!

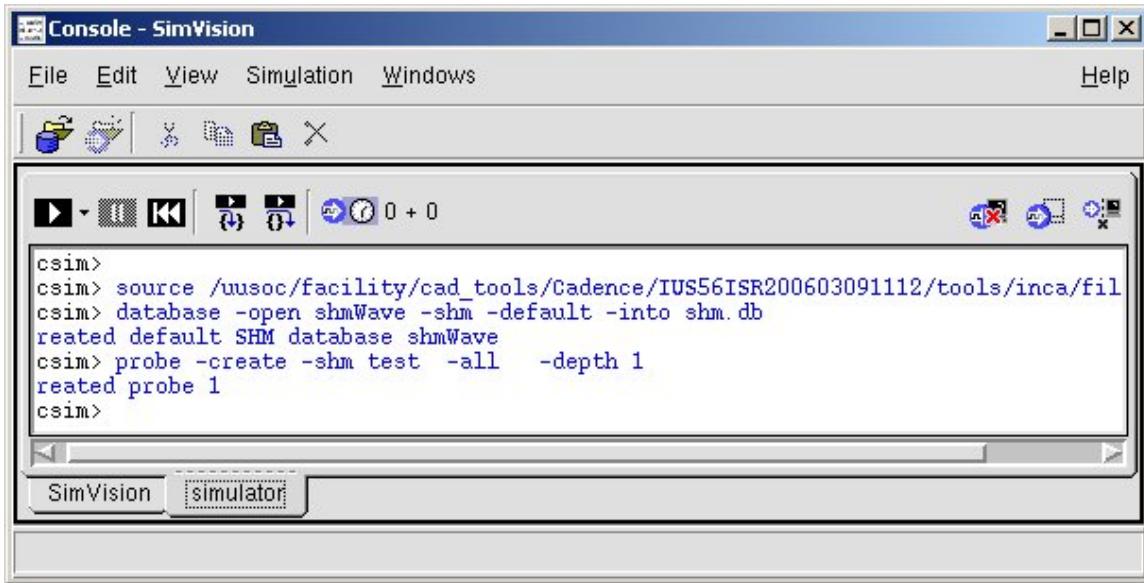


Figure 4.25: SimVision Console Window

### Printing NC\_Verilog Output

Printing waveforms from SimVision is the same whether you get to SimVision from Verilog-XL or from NC\_Verilog. The dialog for printing waveforms is shown in Figure 4.21 in Section 4.1.1.

The output of the **\$display** statements in your testbench code are stored in the log file of the simulation. There may be a way to access the log file from the SimVision menus, but I haven't found it. To see the output of the **\$display**, **\$monitor** and other output statements in your testbench look at the logfile in the simulation directory. Recall that the default simulation directory name is <**designname**>\_run1, and the default log file name is **simout.tmp**. This is simply a text file that includes the log information from the NC\_Verilog simulation including all the outputs from **\$display** and from other output commands in your test fixture.

## 4.2 Behavioral Verilog Code in Composer

One way to describe the functionality of your circuit is to use collections of Boolean gates from a standard cell library, as is shown in the previous sections. Another way is to describe the functionality of the circuit as Verilog code. The Verilog code might be purely structural meaning that it consists only of instantiations of gates from that same standard cell library. In that

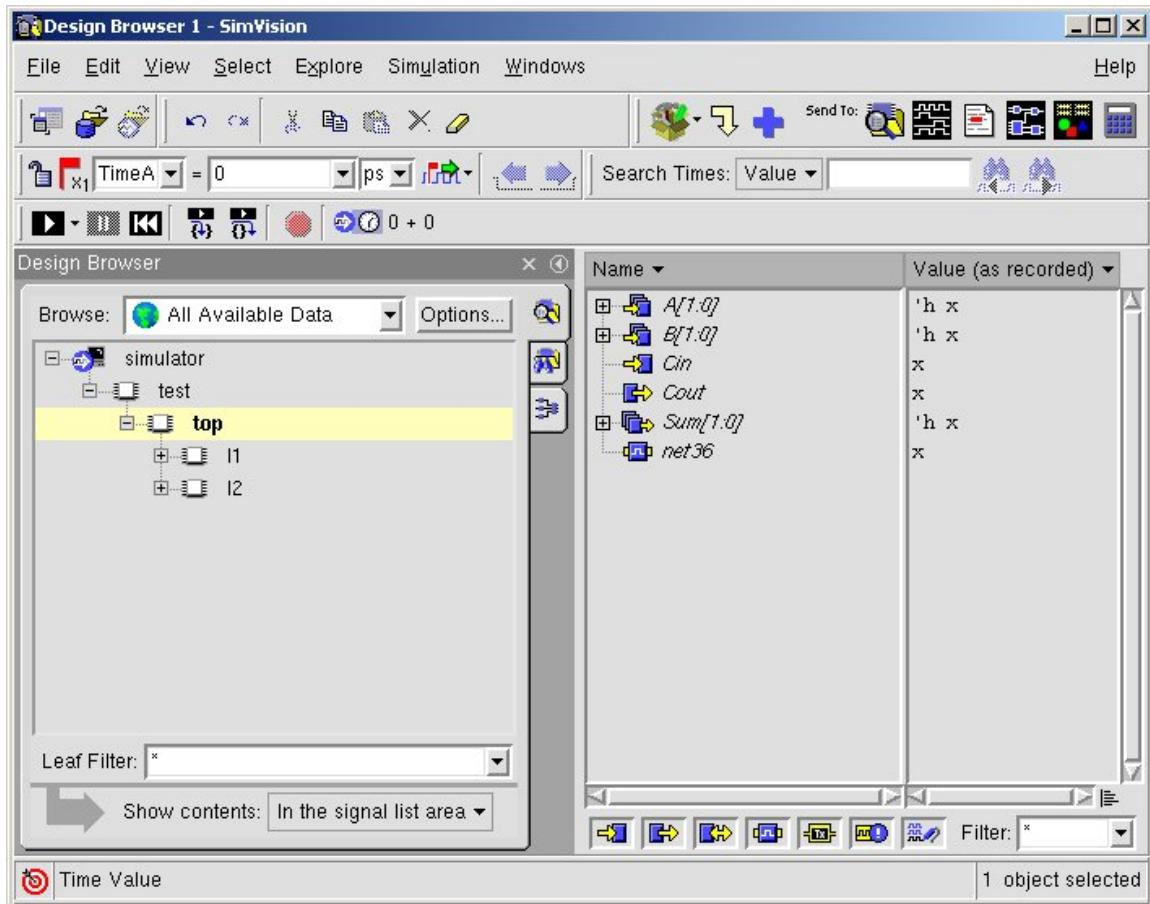


Figure 4.26: SimVision Design Browser Window

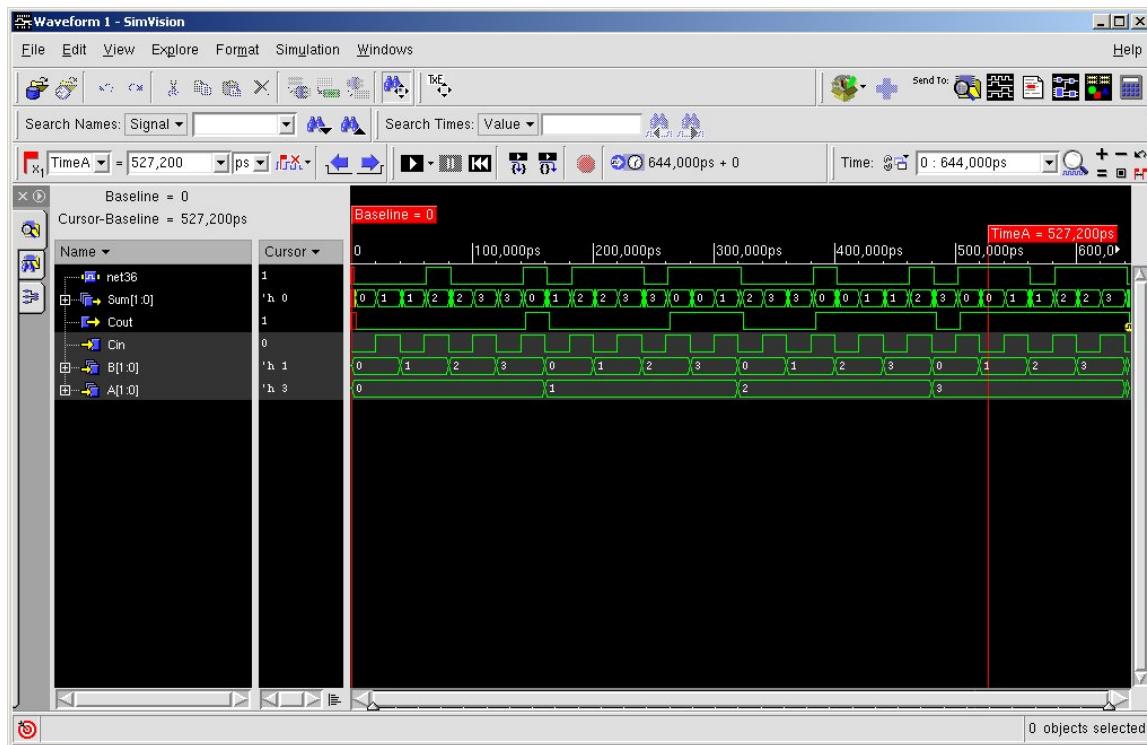


Figure 4.27: Waveform Window Showing the Output of Exhaustive Simulation

case the Verilog is simply a textual version of the schematic. Verilog can also describe a circuit as a behavior meaning that it uses statements in the Verilog language to describe the desired behavior. A behavioral Verilog description will look much more like a software program than like a circuit description, and will need to be converted (or *synthesized*) into a structural description at some point if you want to actually build the circuit, but as an initial description, a behavioral model can be much more compact, can use higher-level descriptions, and can simulate much more quickly.

A behavioral Verilog model can be developed as a Verilog program using a text editor, and simulated using a Verilog simulator without ever using the **Composer** schematic tool as will be seen in Section 4.3. However, symbols in **Composer** schematics can be an interface to behavioral Verilog code just as easily as they can contain gate schematics. These symbols which encapsulate Verilog behavioral code may be connected in schematics using wires just like other symbols, and can peacefully coexist with other symbols that encapsulate gate views. The cell view used for general schematics is **schematic** and the cell view we use for behavioral Verilog views is **behavioral**. In fact, a single cell may have both of these views and you can tell the netlister later which view to use in a given simulation. You can actually use any view name you like for the Verilog behavioral view. It might actually make sense in some situations to use a different name, like **verilog** for your Verilog view so you can keep track of which pieces of Verilog are your own behavioral code and which pieces of Verilog are the library cell descriptions. For now we won't make a distinction.

This allows great freedom in terms of how you want to describe your circuits. You can start with a Verilog **behavioral** view for initial functional simulation, and then incrementally refine pieces of the circuit into gate level **schematic** views, all while using the same testbench code for simulation.

*Synthesis of behavioral Verilog into structural (gate-level) Verilog is covered in Chapter 8.*

*Note that cell views that consist of transistors or that correspond to a single standard cell may use **cmos\_sch** as their schematic view*

### 4.2.1 Generating a Behavioral View

Making a behavioral view is very similar to making any other cell view. You use the Library Manager to make a new cell view, except that you choose **behavioral** as the view name instead of **schematic**.

Start in the Library Manager and select the library in which you want to make your new cell view. Of course, if you're starting a whole new library, you need to make the new library first. If you're adding a behavioral view to a cell that already exists, select the cell too. That will fill in both the library and the cell in the **new cell view** dialog box as shown in Figure 4.28. In this case I'm using the nand2 that was designed as a transistor schematic in Chapter 3 Section 3.3 and will add a behavioral view of that cell. Clicking OK to the dialog box of Figure 4.28 will open up a **Verilog Editor**. Actually,



Figure 4.28: Dialog Box for Creating a Behavioral View

this is just a text editor, in this case `emacs`.

In the editor you will get a very simple template for your new Verilog component. If you have already made a symbol for your cell, then the template will already have the inputs and outputs of the Verilog module defined from the pins on the symbol. If you haven't yet made a symbol, there will just be a module name. In this case because we already defined a symbol, the Verilog template looks like Figure 4.29. You now need to fill in this template with Verilog code that describes the behavior of your cell.

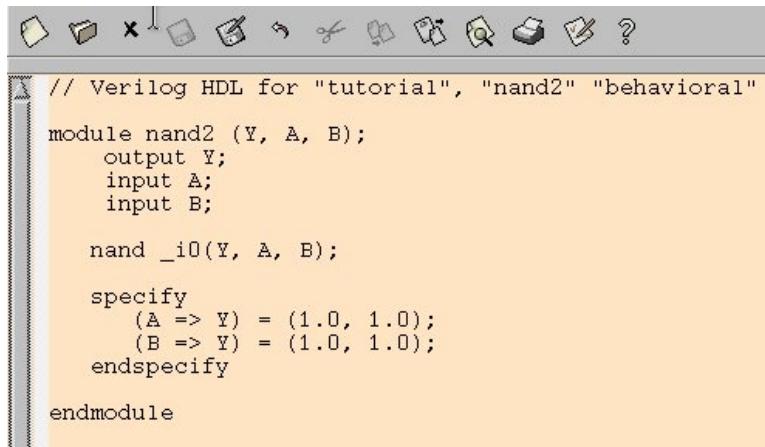
*Verilog built-in gate primitives include and, or, not, nand, nor, xor, and xnor.*

```

File Edit Options Buffers Tools Help
File Edit Options Buffers Tools Help
// Verilog HDL for "tutorial", "nand2" "behavioral"
module nand2 (Y, A, B);
    output Y;
    input A;
    input B;
endmodule

```

Figure 4.29: Behavioral View Template Based on the nand2 Symbol



```
// Verilog HDL for "tutorial", "nand2" "behavioral"
module nand2 (Y, A, B);
    output Y;
    input A;
    input B;

    nand _i0(Y, A, B);

    specify
        (A => Y) = (1.0, 1.0);
        (B => Y) = (1.0, 1.0);
    endspecify

endmodule
```

Figure 4.30: Complete Behavioral Description of a nand2 Cell

model. This isn't a gate from an external library, it's the built-in Verilog primitive. The name of the instance is `_i0`. Note that I've given the instance a name that starts in an underscore so that if I ever navigate to this instance in a simulation I can tell that it came from the behavioral model of the cell.

I've also used **specify** statements to describe the input to output delay of the cell. In this case both the rising and falling delays for both the A to Y and B to Y paths are set to 1.0 time units. These specify blocks are very important later on! They are the mechanism through which extracted timings from the synthesis procedures are annotated to your schematic. The timings in the Standard Delay Format (.sdf) file that comes from the synthesis program will be in terms of the input to output path delay, and the back-annotation procedure will look for specify statements in the cell descriptions to update to the new extracted timing values.

There are many different descriptions you could use in Verilog for a simple NAND gate like this. You could, for example, define the function as a continuous assignment

```
assign Y = ~(A & B);
```

or you could define the function as an assignment inside an **always** block

```
reg Y;
always @(A or B)
begin
    Y = ~(A & B);
end
```

or, if you had a standard cell library available with the appropriate gates, you could define it the NAND as a structural composition of gates from that library.

```
wire w;
AND2 _u1(w, A, B);
INV _u2(Y, w);
```

Each of these descriptions would work. The point is that any legal Verilog behavioral/structural syntax that will simulate in the Verilog simulator will work for the contents of a **behavioral** cell view.

*The syntax checker appears to be checking for Verilog 1995 (i.e. Verilog-XL) syntax in this check.*

Once you have entered your Verilog code into the template in the editor, save the file and quit the editor. This will cause the **Composer** Verilog integration system to check the syntax of your code and make sure that it is compatible with the Verilog editor. If it reports errors you need to re-edit and fix your Verilog code.

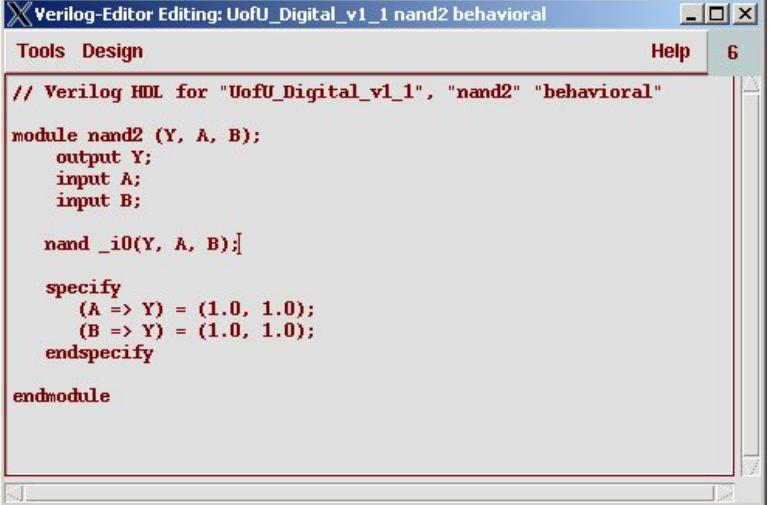
It will also check to make sure that the pin descriptions in your schematic view, the pins in your symbol, and the interface in your behavioral view are all consistent. This is critical if all the different views can be used in various combinations in later simulations.

Once you have a behavioral view of a cell, that view can be simulated from **Composer** using a **Cadence** Verilog simulator, and it can be included via the symbol in other schematics and those schematics can be simulated using either **Verilog-XL** or **NC\_Verilog**. All of the cells in the **UofU\_Digital.v1\_1** library have behavioral views. These views are the default choice when simulating schematics that include instances of these cells. If a transistor switch-level simulation is desired, follow the procedure in Section 4.4.1.

### 4.2.2 Simulating a Behavioral View

The simplest way to simulate a behavioral view is to make a new schematic and include an instance of the symbol that encapsulates the behavioral view. Then you can use the procedure described in Section 4.1.1 to simulate with **Verilog-XL**, or the procedure described in Section 4.1.2 to simulate with **NC\_Verilog**.

You can also simulate the Verilog code in the behavioral view directly without going through a schematic. If you double-click the **behavioral** view in the **library manager** you'll open an emacs editing window to edit the code. However, if you right-click on the behavioral view you'll get a menu where you can choose **Open (Read Only)**. If you select this choice you'll open a read-only window that looks like that in Figure 4.31. From here you



```
// Verilog HDL for "UofU_Digital_v1_1", "nand2" "behavioral"
module nand2 (Y, A, B);
    output Y;
    input A;
    input B;
    nand _i0(Y, A, B);

    specify
        (A => Y) = (1.0, 1.0);
        (B => Y) = (1.0, 1.0);
    endspecify
endmodule
```

Figure 4.31: Read Only Window for Simulation of Behavioral View

can select **Tools → Verilog-XL** to simulate that code using the Verilog-XL simulator as described in Section 4.1.1. There appears to be no option for direct simulation using NC\_Verilog.

### 4.3 Stand-Alone Verilog Simulation

The preceding sections have all used the Composer integration framework to simulate circuits and behavioral Verilog code through the Composer schematic capture tool. Of course, you may have Verilog code that you've developed outside of the schematic framework that you'd like to simulate. It's entirely reasonable and common to develop a system as a Verilog program using a text editor and then using that Verilog code as the starting point for a CAD flow. The CAD flow in this case goes from that Verilog code (through synthesis) directly to the back end place and route without ever using a schematic. Of course, the Verilog simulators can run (simulate) that Verilog code directly.

In order to simulate your Verilog code you will need both the code that describes your system, and testbench code to set the inputs of your system and check that the outputs are correct. As shown in Figure 4.1 this corresponds to DUT code (the code describing your system) and testbench or testfixture code. These should both be part of a single top-level module that is simulated. An example is shown in Figure 4.2 where the testfixture code is contained in a separate file named **testfixture.verilog**.

As another example, consider the Verilog code describing a simple state machine shown in Figure 4.32. This state machine looks at bits on the input **insig** and raises the output signal **saw4** whenever the last four bits have all been **1**. Of course, there are simpler ways of building a circuit with this functionality, but it makes a perfectly good example of a small finite state machine to simulate with each of the Verilog simulators.

In order to simulate this piece of Verilog code a testbench is required. Based on the technique described in Figures 4.1 and 4.2, a top-level Verilog file is required that includes an instance of the **see4** module and some testbench code. This top-level file is shown in Figure 4.33, and an example of a possible testbench in the **testfixture.v** included file is shown in Figure 4.34. These files will be used to demonstrate each of the Verilog simulators in stand-alone mode.

### 4.3.1 Verilog-XL

Verilog-XL is an interpreted Verilog simulator from Cadence that is the reference simulator for the Verilog-1995 standard. This means that Verilog constructs from later versions of the standard will not run with this simulator. As seen in Section 4.1.1, it is well-integrated with the Composer schematic capture system, but it is easily used on its own too. The inputs to the simulator are, at the simplest, just a list of the Verilog files to simulate, but can also include a dizzying array of additional arguments and switches to control the behavior of the simulator. If you're already set your path to include the standard CAD scripts (see Section 2.2), then you can invoke the Verilog-XL simulator using the script

```
sim-xl <verilogfilename>
```

I find it useful to put the files that you want to simulate in a separate file, and then invoke the simulator with the **-f** switch to point to that file. In the case of our example from Figures 4.32 to 4.34, the **files.txt** file simply lists the **see4.v** and **seetest.v** files and looks like:

```
see4.v  
seetest.v
```

In this case, I would invoke the Verilog-XL simulator using the command:

```
sim-xl -f files.txt
```

Anything that you put on the **sim-xl** command line will be passed through to the Verilog-XL simulator so you can include any other switches that you like this way. Try **sim-xl -help** to see a list of some of the switches.

```

// Verilog HDL for "Ax", "see4" "behavioral"
// Four in a row detector - written by Allen Tanner
module see4 (clk, clr, insig, saw4);
    input clk, clr, insig;
    output saw4;

    parameter s0 = 3'b000; // initial state, saw at least 1 zero
    parameter s1 = 3'b001; // saw 1 one
    parameter s2 = 3'b010; // saw 2 ones
    parameter s3 = 3'b011; // saw 3 ones
    parameter s4 = 3'b100; // saw at least, 4 ones

    reg [2:0] state, next_state;

    always @(posedge clk or posedge clr) // state register
    begin
        if (clr) state <= s0;
        else state <= next_state;
    end

    always @(insig or state) // next state logic
    begin
        case (state)
            s0: if (insig) next_state = s1;
                  else next_state = s0;
            s1: if (insig) next_state = s2;
                  else next_state = s0;
            s2: if (insig) next_state = s3;
                  else next_state = s0;
            s3: if (insig) next_state = s4;
                  else next_state = s0;
            s4: if (insig) next_state = s4;
                  else next_state = s0;
            default: next_state = s0;
        endcase
    end

    // output logic
    assign saw4 = state == s4;
endmodule //see4

```

Figure 4.32: A simple state machine described in Verilog: **see4.v**

```
\\" Top-level test file for the see4 Verilog code
module test;

\\ Remember that DUT outputs are wires, and inputs are reg
wire saw4;
reg clk, clr, insig;

\\ Include the testfixture code to drive the DUT inputs and
\\ check the DUT outputs
`include "testfixture.v"

\\ Instantiate a copy of the see4 function (named top)
see4 top(clk, clr, insig, saw4);

endmodule //test
```

Figure 4.33: Top-level Verilog code for simulating **see4** named **seetest.v**

If you look inside the **sim-xl** script you will see that it starts a new shell, sources the setup script for Cadence, and then calls Verilog-XL with whatever arguments you supplied. The command puts the log information into a file called **xl.log** in the same directory in which **sim-xl** is called.

If I run this command using the **see4** example, I get the output shown in Figure 4.35. The important parts of the output are the results from the **\$display** statements in the test bench. They indicate that the state machine is operating as expected because none of the **ERROR** statements has printed.

*Of course, if the  
testbench checking code  
is not correct, all bets  
are off!*

If there were errors in the original Verilog code, and the testbench was written to correctly check for faulty behavior then you would get **ERROR** statements printed. As an example, if I modify the **see4** code in Figure 4.32 so that it doesn't operate correctly, a simulation should signal errors. I'll change the next-state function in state **s4** so that instead of looping in that state on a **1** the state machine goes back to state **s0**. If I simulate that (faulty) state machine, the **xl.log** file of the simulation prints out a series of **ERROR** statements as shown in Figure 4.36.

### Stand-Alone Verilog-XL Simulation with simVision

It's also possible to run the Verilog-XL simulator in stand-alone mode and also invoke the gui, which includes the waveform viewer. To do this, use the **sim-xlg** script instead of **sim-xl**. The only difference is that the **sim-xlg** invokes the gui (**simVision**), and starts the simulator in interactive mode which means that you can select signals to see in the waveform viewer before starting the simulation. The command would be

```
sim-xlg -f files.txt
```

```

// Four ones in a row detector testbench (testfixture.v)
// Main tests are in an initial block
initial
begin
    clk = 1'b0; // initialize the clock low
    clr = 1'b1; // start with clr asserted
    insig = 1'b0; // insig starts low

    #500 clr = 1'b0; // deassert clear and start running

    // use the send_test task to test the state machine
    send_test(32'b0011_1000_1010_1111_0000_0111_1110_0000);
    send_test(32'b0000_0001_0010_0011_0100_0101_0110_0111);
    send_test(32'b1000_1001_1010_1011_1100_1101_1110_1111);
    send_test(32'b1011_1111_1101_1111_1111_1100_1011_1111);

    // Print something so we know we're done
    $display("\nSaw4 simulation is finished...");
    $display("If there were no 'ERROR' statements, then everything worked!\n");
    $finish;
end

// Generate the clock signal
always #50 clk = ~clk;

// this task will take the 32 bit input pattern and apply
// those bits one at a time to the state machine input.
// Bits are changed on negedge so that they'll be set up for
// the next active (posedge) of the clock.
task send_test;
    input [31:0]pat; // input bits in a 32-bit array
    integer i; // integer for looping in the for statement
begin
    for(i=0;i<32; i=i+1) // loop through each of the bits in the pat array
    begin
        // apply next input bit before next rising clk edge
        @(negedge clk)insig = pat[i];

        // remember to check your answers!
        // Look at last four bits to see if saw4 should be asserted
        if ((i > 4)
            && ({pat[i-4],pat[i-3],pat[i-2],pat[i-1]} == 4'b1111)
            && (saw4 != 1))
            $display("ERROR - didn't recognize 1111 at pat %d,", i);
        else if ((i > 4)
            && ({pat[i-4],pat[i-3],pat[i-2],pat[i-1]} != 4'b1111)
            && (saw4 == 1))
            $display("ERROR - signalled saw4 on %b inputs at step %d",
            {pat[i-3],pat[i-2],pat[i-1],pat[i]}, i);
    end // begin-for
end // begin-task
endtask // send_test

```

Figure 4.34: Testbench code for **see4.v** in a file named **testfixture.v**

```
--->sim-xl -f test.txt
Tool:    VERILOG-XL      05.10.004-s   Jul 29, 2006  20:50:01

Copyright (c) 1995-2003 Cadence Design Systems, Inc. All Rights Reserved.
Unpublished -- rights reserved under the copyright laws of the United States.

Copyright (c) 1995-2003 UNIX Systems Laboratories, Inc. Reproduced with Permission.

THIS SOFTWARE AND ON-LINE DOCUMENTATION CONTAIN CONFIDENTIAL INFORMATION
AND TRADE SECRETS OF CADENCE DESIGN SYSTEMS, INC. USE, DISCLOSURE, OR
REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF
CADENCE DESIGN SYSTEMS, INC. RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to
restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in
Technical Data and Computer Software clause at DFARS 252.227-7013 or
subparagraphs (c)(1) and (2) of Commercial Computer Software --
Restricted
Rights at 48 CFR 52.227-19, as applicable.

Cadence Design Systems, Inc.
555 River Oaks Parkway
San Jose, California 95134

For technical assistance please contact the Cadence Response Center at
1-877-CDS-4911 or send email to support@cadence.com

For more information on Cadence's Verilog-XL product line send email to
talkv@cadence.com

Compiling source file ''see4.v''
Compiling source file ''seetest.v''

Warning! Code following 'include command is ignored
[Verilog-CAICI]
      ''seetest.v'', 6:
Compiling included source file ''testfixture.v''
Continuing compilation of source file ''seetest.v''
Highest level modules:
test

Saw4 simulation is finished...
If there were no 'ERROR' statements, then everything worked!

L17 ''testfixture.v'': $finish at simulation time 13200
1 warning
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in
simulation
End of Tool:    VERILOG-XL      05.10.004-s   Jul 29, 2006  20:50:02
--->
```

Figure 4.35: Output of stand-alone Verilog-XL simulation of **seetest.v**

```
<previous text not included...>
Compiling included source file ``testfixture.v''
Continuing compilation of source file ``seetest.v''
Highest level modules:
test

ERROR - didn't recognize 1111 at pat      10,
ERROR - didn't recognize 1111 at pat      11,
ERROR - didn't recognize 1111 at pat      5,
ERROR - didn't recognize 1111 at pat      6,
ERROR - didn't recognize 1111 at pat      15,
ERROR - didn't recognize 1111 at pat      16,
ERROR - didn't recognize 1111 at pat      17,
ERROR - didn't recognize 1111 at pat      18,
ERROR - didn't recognize 1111 at pat      20,
ERROR - didn't recognize 1111 at pat      21,
ERROR - didn't recognize 1111 at pat      27,
ERROR - didn't recognize 1111 at pat      28,
ERROR - didn't recognize 1111 at pat      29,
ERROR - didn't recognize 1111 at pat      30,

Saw4 simulation is finished...
If there were no 'ERROR' statements, then everything worked!

L17 ``testfixture.v'': $finish at simulation time 13200
1 warning
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in
simulation
End of Tool:    VERILOG-XL      05.10.004-s Jul 29, 2006 21:21:49
```

Figure 4.36: Result of executing a faulty version of **see4.v**

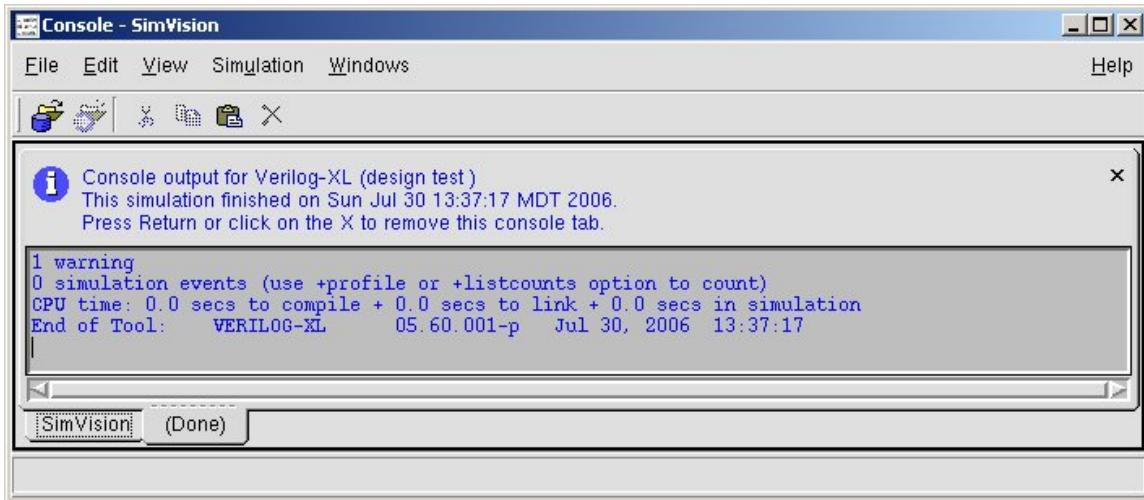


Figure 4.37: Control console for stand-alone Verilog-XL simulation using SimVision

which, if you look inside the script, does exactly the same thing that **sim-xl** does, but adds a couple switches to start up the simulation in the gui in interactive mode. The gui that is used is the same gui environment from Section 4.1.1. Select the signals that you would like to see in the waveform first before starting the simulation with the **play** button (the right-facing triangle). The control console is shown in Figure 4.37, the hierarchy browser in Figure 4.38, and the waveform window (after signals have been selected and the simulation run) in Figure 4.39. See Section 4.1.1 for more details of driving the simVision gui.

### 4.3.2 NC\_Verilog

**NC\_Verilog** is a compiled simulator from Cadence that implements many of the Verilog 2000 features. It compiles the simulator from the Verilog simulation by translating the Verilog to C code and compiling the C code. This results in a much faster simulation time at the expense of extra compilation time at the beginning. This simulator is also well-integrated with the Composer schematic capture tool as seen in Section 4.1.2. At its simplest, the inputs to the **NC\_Verilog** simulator are just a list of files to simulate, but like other Verilog simulators, there are many many switches that can be given at the command line to control the simulation. If you're already set your path to include the standard CAD scripts (see Section 2.2), then you can invoke the **NC\_Verilog** simulator using the script

```
sim-nc <verilogfilename>
```

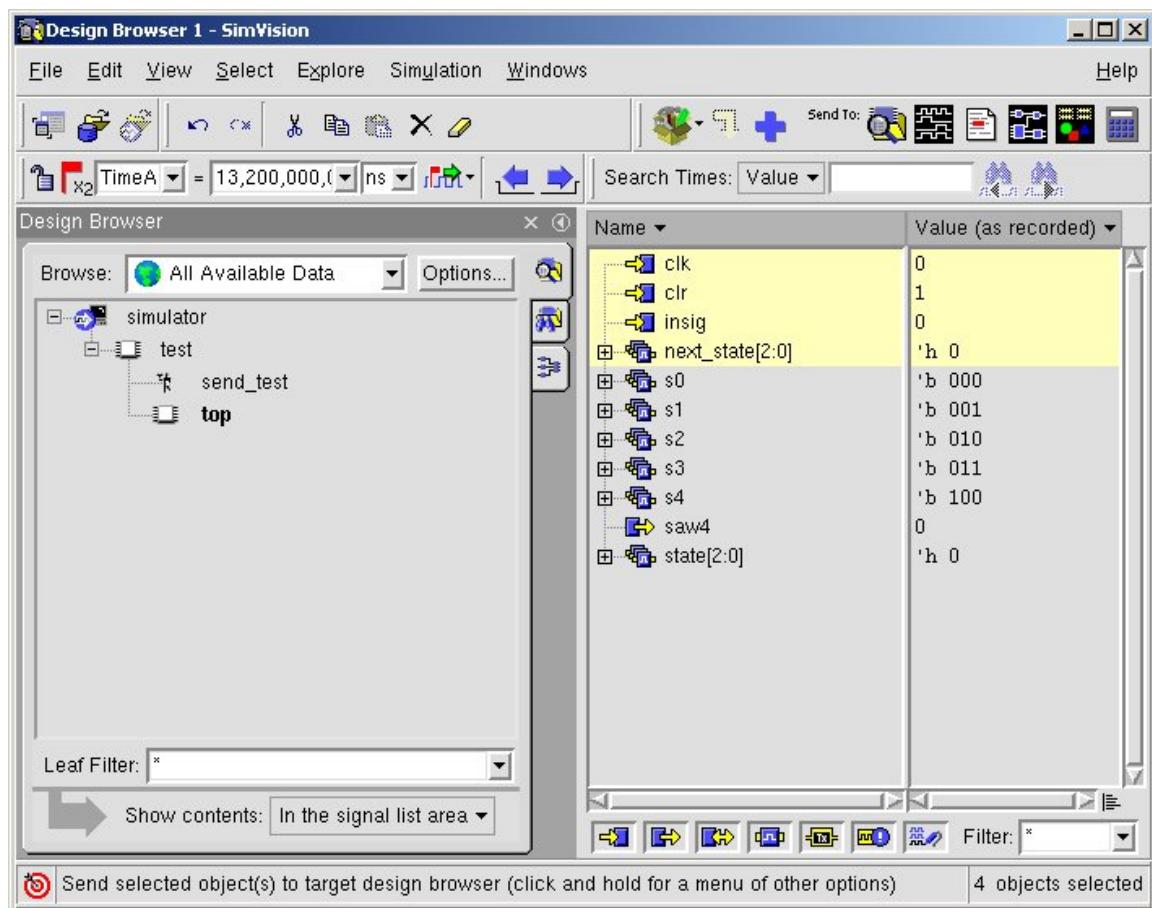
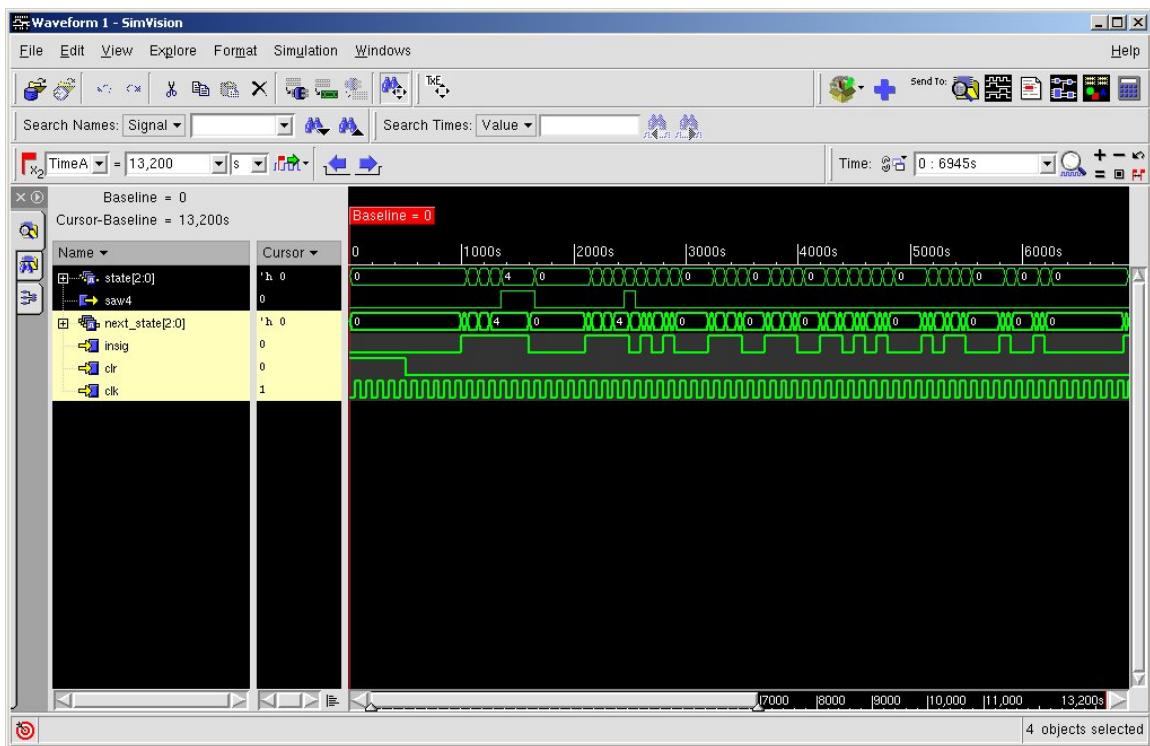


Figure 4.38: Hierarchy browser for the **see4** example

Figure 4.39: Waveform viewer after running the **see4** example

I find it useful to put the files that you want to simulate in a separate file, and then invoke the simulator with the `-f` switch to point to that file. In the case of our example from Figures 4.32 to 4.34, the `files.txt` file simply lists the `see4.v` and `seetest.v` files and looks like:

```
see4.v
seetest.v
```

In this case, I would invoke the NC\_Verilog simulator using the command:

```
sim-nc -f files.txt
```

Anything that you put on the `sim-nc` command line will be passed through to the NC\_Verilog simulator so you can include any other switches that you like this way. Try `sim-nc -help` to see a list of some of the switches. If you look inside the `sim-nc` script you will see that it starts a new shell, sources the setup script for Cadence, and then calls NC\_Verilog with the command line information that you have supplied. The log file is `nc.log`. If I run this command using the `see4` example, I get the output shown in Figure 4.40. The important parts of the output are the results from the `$display` statements in the test bench. They indicate that the state machine is operating as expected because none of the `ERROR` statements has printed.

If there were errors in the original Verilog code, and the testbench was written to correctly check for faulty behavior then you would get `ERROR` statements printed. These error statements would look similar to those in Figure 4.36.

*Of course, if the testbench checking code is not correct, all bets are off!*

### Stand-Alone NC\_Verilog Simulation with simVision

It's also possible to run the NC\_Verilog simulator in stand-alone mode and also invoke the gui, which includes the waveform viewer. To do this, use the `sim-nco` script instead of `sim-nc`. The only difference is that the `sim-nco` invokes the gui (simVision), and starts the simulator in *interactive* mode which means that you can select signals to see in the waveform viewer before starting the simulation. The command would be

```
sim-nco -f files.txt
```

which, if you look inside the script, invokes the same command as does `sim-nc`, but with a switch that starts up the simulation in the SimVision gui. This is the same gui environment from Section 4.1.2. Select the signals that you would like to see in the waveform first before starting the simulation with the `play` button (the right-facing triangle). The control console for the

```
---> sim-nc -f test.files
ncverilog: 05.10-s014: (c) Copyright 1995-2004 Cadence Design Systems,
Inc.
file: see4.v
    module worklib.see4:v
        errors: 0, warnings: 0
file: seetest.v
    module worklib.test:v
        errors: 0, warnings: 0
        Caching library 'worklib' .... Done
    Elaborating the design hierarchy:
    Building instance overlay tables: ..... Done
    Generating native compiled code:
        worklib.see4:v <0x3d4ece8f>
            streams: 5, words: 1771
        worklib.test:v <0x37381383>
            streams: 6, words: 4703
    Loading native compiled code: ..... Done
    Building instance specific data structures.
    Design hierarchy summary:
        Instances Unique
    Modules:      2      2
    Registers:   7      7
    Scalar wires: 4      -
    Always blocks: 3      3
    Initial blocks: 1      1
    Cont. assignments: 1      1
    Pseudo assignments: 3      4
    Writing initial simulation snapshot: worklib.test:v
    Loading snapshot worklib.test:v ..... Done
ncsim> source
/uusoc/facility/cad_tools/Cadence/LDV/tools/inca/files/ncsimrc
ncsim> run

Saw4 simulation is finished...
If there were no 'ERROR' statements, then everything worked!

Simulation complete via $finish(1) at time 13200 NS + 0
./testfixture.v:17      $finish;
ncsim> exit
--->
```

Figure 4.40: Output of stand-alone NC\_Verilog simulation of **seetest.v**

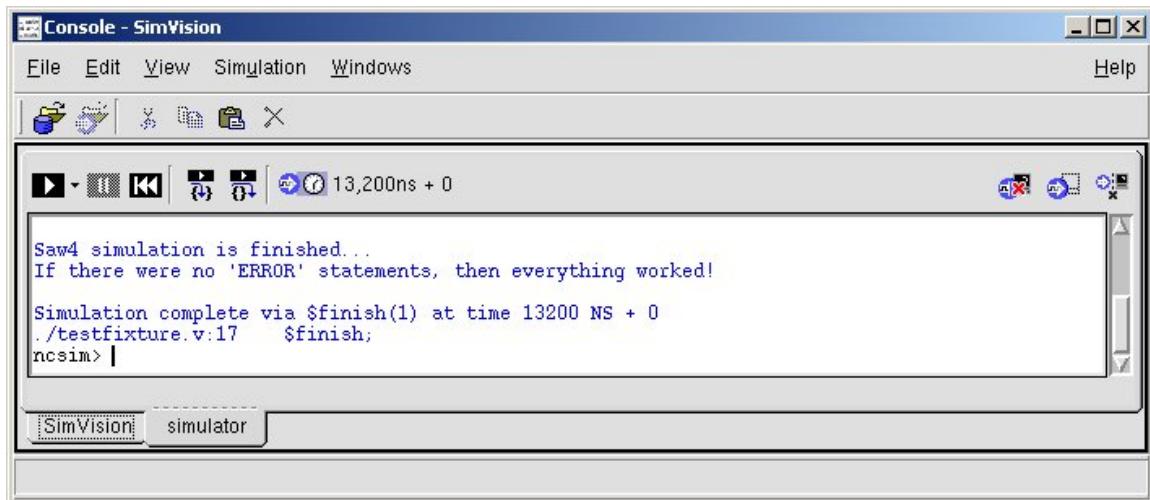


Figure 4.41: Control console for NC\_Verilog through SimVision

NC\_Verilog simulation is shown in Figure 4.41. The hierarchy and waveform windows will look the same as the Verilog-XL version in Figures 4.38 and 4.39. See Section 4.1.1 for more details of driving the simVision gui.

### 4.3.3 vcs

The Verilog simulator from Synopsys is **vcs**. This isn't integrated with the Cadence dfll tool suite, but is a very capable simulator in its own right so it's important to know about if you're using more of the **Synopsys** tools than **Cadence** tools. The **vcs** simulator is a compiled simulator so it runs very fast once the simulator is compiled, and is compatible with Verilog-2000 features. The inputs to the simulator are, at the simplest, just a list of the Verilog files to simulate, but can also include a dizzying array of additional arguments and switches to control the behavior of the simulator. If you're already set your path to include the standard CAD scripts (see Section 2.2), then you can invoke the **vcs** simulator using the script

```
sim-vcs <verilogfilename>
```

I find it useful to put the files that you want to simulate in a separate file, and then invoke the simulator with the **-f** switch to point to that file. In the case of our example from Figures 4.32 to 4.34, the **files.txt** file simply lists the **see4.v** and **seetest.v** files and looks like:

```
see4.v
seetest.v
```

*More correctly we are using vcs\_mx which is the “mixed mode” version that can support both Verilog and VHDL, but I’ll just call it vcs because we’re mostly interested in the Verilog version.*

In this case, I would invoke the **vcs** simulator using the command:

```
sim-vcs -f files.txt
```

Anything that you put on the **sim-vcs** command line will be passed through to the **vcs** simulator so you can include any other switches that you like this way. Try **sim-vcs -help** to see a list of some of the switches. If you look inside the **sim-ves** script you will see that it starts a new shell, sources the setup script for **Synopsys**, and then calls **vcs** with the command line information you have supplied. The log file is **vcs.log**. If I run this command using the **see4** example, I get the output shown in Figure 4.42.

For the **vcs** simulator, running the **sim-vcs** script doesn't actually run the simulation. Instead it compiles the Verilog code into an executable called **simv**. After compiling this simulator, you can run it by running **simv** to get the output seen in Figure 4.43, but the **simv** executable needs some setup information about **Synopsys** before it can run, so rather than run **simv** directly, you will run it through a wrapper called **sim-simv** as follows:

```
sim-simv <executable-name>
```

Because the executable will be named **simv** unless you've overridden that default name with a command-line switch, this will almost always be called as:

```
sim-simv simv
```

The important parts of the output are the results from the **\$display** statements in the test bench. They indicate that the state machine is operating as expected because none of the **ERROR** statements has printed.

If there were errors in the original Verilog code, and the testbench was written to correctly check for faulty behavior then you would get **ERROR** statements printed. These error statements would look similar to those in Figure 4.36.

Of course, if the testbench checking code is not correct, all bets are off!

### Stand-Alone **vcs** Simulation with **VirSim**

It's also possible to run the **vcs** simulator in stand-alone mode and also invoke the **gui**, which includes the waveform viewer. To do this, use the **sim-vcsg** script instead of **sim-vcs**. The only difference is that the **sim-vcsg** invokes the **gui** (**VirSim**), and starts the simulator in *interactive* mode which means that you can select signals to see in the waveform viewer before starting the simulation. The command would be

```
sim-vcsg -f files.txt
```

The only real difference in the **sim-vcsg** script from the **sim-ves** script

```
---> sim-vcs -f test.files
      Chronologic VCS (TM)
      Version X-2005.06-SP2 -- Sat Jul 29 21:49:35 2006
      Copyright (c) 1991-2005 by Synopsys Inc.
      ALL RIGHTS RESERVED

This program is proprietary and confidential information of Synopsys
Inc.
and may be used and disclosed only as authorized in a license
agreement
controlling such use and disclosure.

Parsing design file 'seed4.v'
Parsing design file 'seetest.v'
Parsing included file 'testfixture.v'.
Back to file 'seetest.v'.
Top Level Modules:
    test
No TimeScale specified
Starting vcs inline pass...
1 module and 0 UDP read.
recompiling module test
make: Warning: File 'filelist' has modification time 41 s in the
future
if [ -x ..../simv ]; then chmod -x ..../simv; fi
g++ -o ..../simv -melf_i386 -m32 5NrI_d.o 5NrIB_d.o gzYz_1_d.o SIM_l.o
/uusoc/facility/cad_tools/Synopsys/vcs/suse9/lib/libvirsim.a
/uusoc/facility/cad_tools/Synopsys/vcs/suse9/lib/libvcsnew.so
/uusoc/facility/cad_tools/Synopsys/vcs/suse9/lib/ctype-stubs_32.a -ldl
-lc -lm -ldl
/usr/lib64/gcc/x86_64-suse-linux/4.0.2/../../../../x86_64-suse-linux/bin/ld:
warning: libstdc++.so.5, needed by
/uusoc/facility/cad_tools/Synopsys/vcs/suse9/lib/libvcsnew.so, may
conflict with libstdc++.so.6
..../simv up to date
make: warning: Clock skew detected. Your build may be incomplete.
CPU time: .104 seconds to compile + .384 seconds to link
--->
```

Figure 4.42: Output of running **sim-vcs** on **files.txt**

```
--> sim-simv simv
Chronologic VCS simulator copyright 1991-2005
Contains Synopsys proprietary information.
Compiler version X-2005.06-SP2; Runtime version X-2005.06-SP2; Jul 29
21:49 2006

Saw4 simulation is finished...
If there were no 'ERROR' statements, then everything worked!

$finish at simulation time 13200
          V C S   S i m u l a t i o n   R e p o r t
Time: 13200
CPU Time:      0.090 seconds;      Data structure size:  0.0Mb
Sat Jul 29 21:49:54 2006
-->
```

Figure 4.43: Output of stand-alone vcs simulation of **seetest.v** using the compiled **simv** simulator

is that it adds a switch to “Run Interactive” after compilation through the VirSim gui environment. Using the **sim-vcs** script on the **see4** example you would see the window in Figure 4.44 pop up. This is the control console for the Synopsys interactive simulator VirSim. I would first open up a hierarchy window using the **Window → Hierarchy** menu choice, the **ctl-shft-H** hotkey, or the **New Hierarchy Browser** widget. You can use this window (shown in Figure 4.45) to select the signals that you would like to track in the simulation. You can also open a waveform window using the **Window → Waveform** menu, the **ctl-shft-W** hotkey, or the **New Waveform Window** widget. Signals selected in the hierarchy window can be added to the waveform window using the **add** button.

Once the signals that you want to track have been added to the waveform window you can run the simulation using the **continue** command (menu or widget). Figure 4.46 shows the waveform window after signals have been added from the hierarchy window to the waveform window and the simulation has been run using the **continue** button.

## 4.4 Timing in Verilog Simulations

Time is modeled in a Verilog simulation either by explicit delay statements or through implicit constructs that wait for an event or an edge of a signal before progressing. Explicit delays are denoted with the # character. You’ve seen the # character used in the testbench code examples in this chapter. The simplest syntax is **#10** which means to delay 10 time units before proceeding

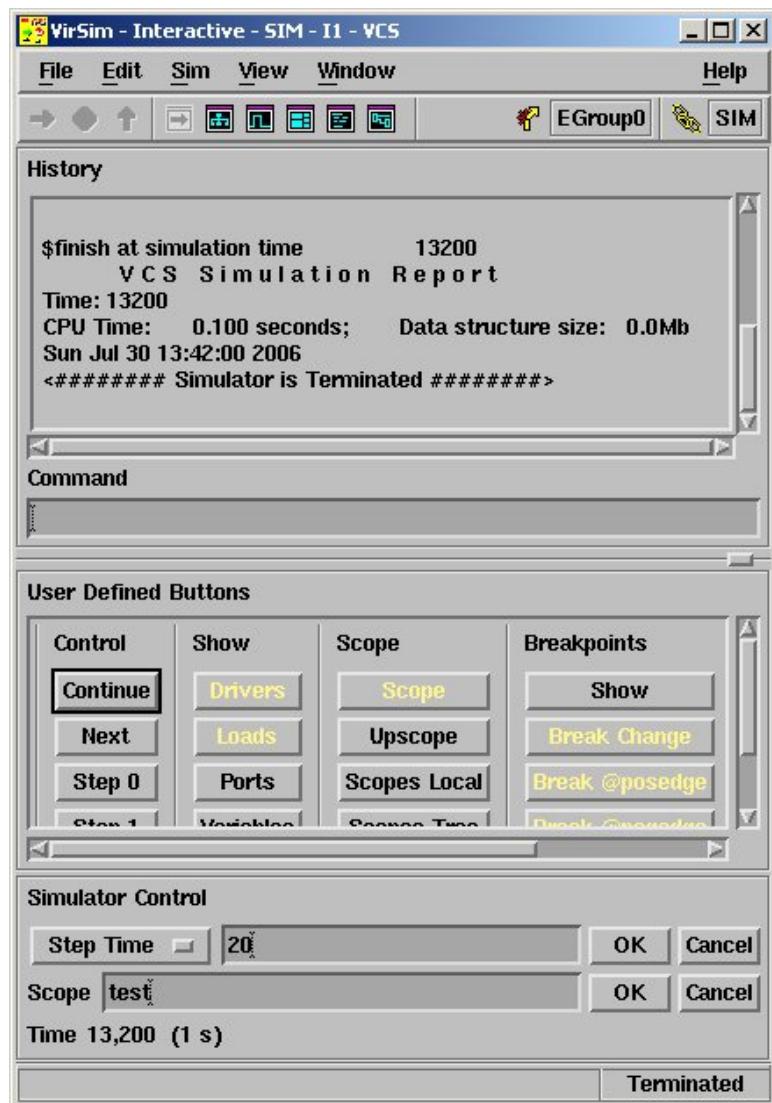


Figure 4.44: Console window for controlling a vcs simulation through VirSim

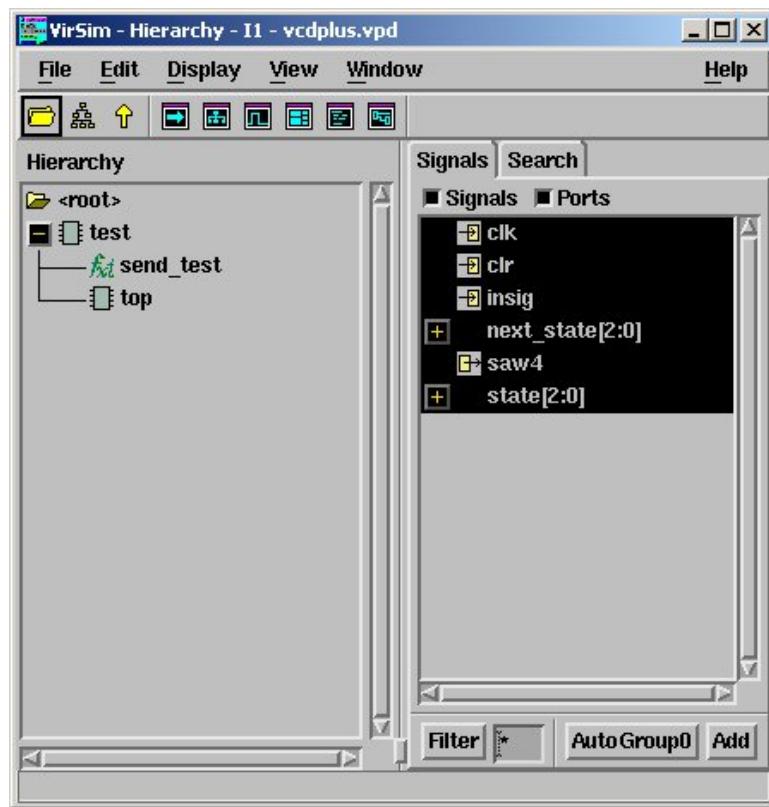


Figure 4.45: Hierarchy browser window from VirSim

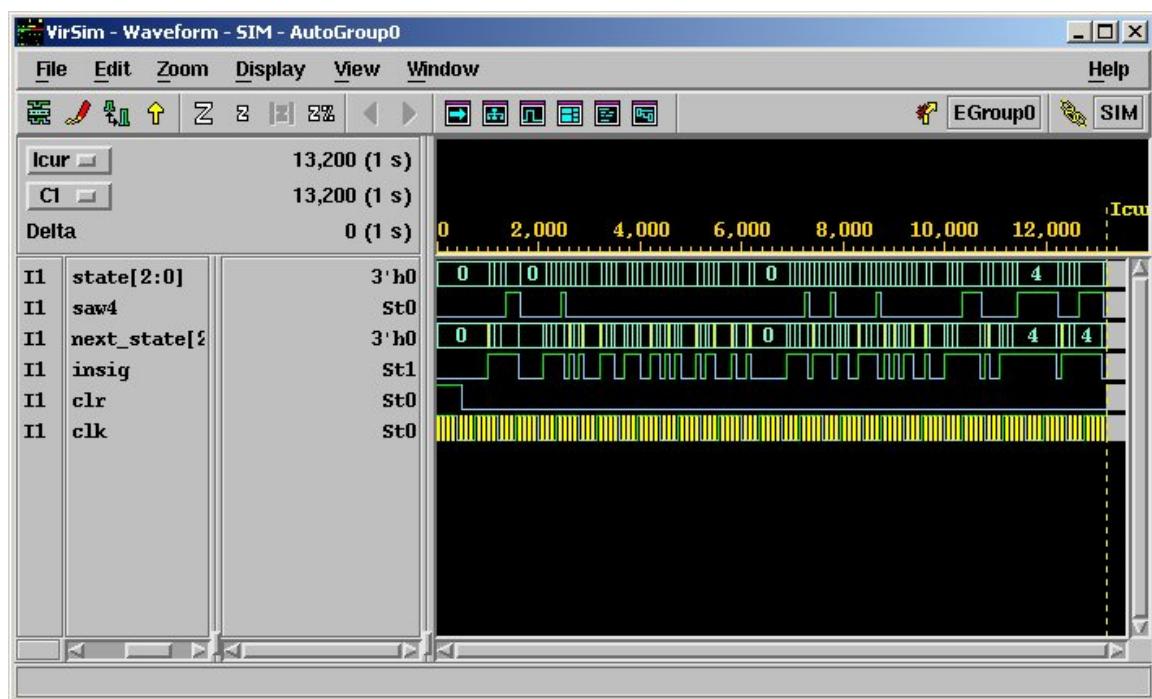


Figure 4.46: Waveform window for VirSim after running the **see4** simulation

with the simulation and executing the following statement. You can also use the **#8:10:12** syntax to indicate minimum, typical, and maximum delays. If delays are specified in this way you can choose which delay to use in a given simulation using command-line switches to the Verilog simulator. Typically (and for the three simulators we use) this is accomplished with a **+mindelays**, **+typdelays** or **+maxdelays** command-line argument.

Of course, if the **#10** construct delays by 10 time units, it is important to know if those time units correspond to any real time units or are simply *unit delay* timings that are not meant to model any specific real time values. Verilog uses the ‘**timescale**’ command to specify the connection between timing units and real timing numbers. The ‘**timescale**’ directive takes two arguments: a timing unit and a timing precision. For example,

**‘timescale 1 ns / 100 ps**

means that each “unit” corresponds to 1ns, and the precision at which timings are computed is 100ps. But be careful! For many simulations the delay numbers do **not** correspond to actual real times and are simply generic “unit timings” to model the fact that time passes without trying to model the exact amount of time the activity takes. Make sure you understand how timing is used in your simulation before you make assumptions!

Implicit delays in a Verilog simulation are signaled using the @ character which causes the following statement or procedural block to wait for some event before proceeding. The even might be a signal transition on a wire, an edge (**posedge** or **negedge**) of a signal change, or an abstract event through the Verilog **event** construct. These implicit delays are often used, for example, to wait for an active clock edge before proceeding.

#### 4.4.1 Behavioral versus Transistor Switch Simulation

In our CAD flow there are two main types of Verilog simulations:

**Behavioral:** The Verilog describes the desired behavior of the system in high-level terms. This description does not correspond directly to hardware, but can be synthesized into hardware using a synthesis tool (Chapter 8).

**Structural:** The Verilog consists of instantiations of primitive gates from a standard cell library. Each of the gates corresponds to a leaf cell from the library that can be placed and routed on the completed chip (Chapter 10). The structural Verilog can be strictly textual (Verilog code), or a hierarchical schematic that uses gate symbols from the standard cell library.

If you're simulating behavior only with high-level behavioral Verilog, then the timing in your simulation depends on the timing that you specify in your code using either explicit (#10) or implicit (@(posedge clk)) statements in your Verilog code. If you're simulating structural code then you (or, hopefully, the tools) need to generate a simulatable netlist of the structural code where each leaf cell is replaced with the Verilog code that defines the function of that cell.

A schematic may be strictly structural if all the leaf cells of the hierarchical schematic are standard cells, or it can contain symbols that encapsulate behavioral code using the behavioral modeling techniques in Section 4.2.

Given this way of thinking about Verilog descriptions of systems, it is easy to apply this to the standard cells themselves. Each of the standard cells in our standard cell library have two different descriptions which are instantiated in two different cell views in the Cadence library:

**behavioral:** In this cell view there is Verilog behavioral code that describes the behavior of the cell. An example of a behavioral view of a standard cell is shown in Figure 4.30. Note that this behavioral description includes unit delays for the **nand2** cell using a **specify** block to specify the input to output delays for that cell as **1.0** units in both the min and max cases.

**cmos\_sch:** In this schematic view the library cells are described in terms of their transistor networks using transistor models from the analog cells libraries. Timing in a simulation of this view would depend on the timing associated with each transistor model.

The CMOS transistors that are the primitives in the **cmos\_sch** leaf cell views have their behavior described in a separate cell view:

**functional:** In this view each transistor is described using a built in Verilog *switch-level* model of a transistor. Switch-level means that the transistor is modeled as a simple switch that is closed (conducting) from source to drain on one value of the gate input, and open (non-conducting) for the other value of the gate input. Using these models results in a *switch level simulation* of the hardware. This models the detailed behavior of the system is simulated with each transistor modeled as an ideal switch. This type of simulation is more accurate (in some sense) than the behavioral model, but not as accurate (or as time consuming) as a more detailed analog transistor simulation.

If each of the cells in the standard cell library has two different views that can be used to simulate the behavior of the cell, then you should be able

```

icfb - Log: /home/elb/CDS.log
File Tools Options Help 1
----- Begin Netlist Configuration Info -----
(incremental data only)
CELL NAME      VIEW NAME      NOTE
-----
xor2           behavioral     *Stopping View*
invX1          behavioral     *Stopping View*
nor2           behavioral     *Stopping View*
nand2          behavioral     *Stopping View*
twoBitAdd      schematic
FullAdder      schematic
nand2_a        schematic
invX1_a        schematic
----- End Netlist Configuration Info -----
*** Netlisting of cell - twoBitAdd, view - schematic successful.
End netlisting Jul 30 16:03:00 2006
mouse L: M: R:
->

```

Figure 4.47: A Netlisting Log for the Two Bit Adder that stops at behavioral views of the standard cell gates

to modify how the simulation proceeds by choosing a different view for each cell when you netlist the schematic. If you expand the schematic to the **behavioral** views of the standard cells you will get one flavor of simulation, and if you expand through the **cmos\_sch** views all the way to the **functional** views of each transistor you will get a different flavor of simulation of the same circuit. This is indeed possible.

Suppose you were simulating a schematic with either Verilog-XL or NC\_Verilog. The netlisting log might look like that in Figure 4.47. This is a repeat of the netlist for the two bit adder from Section 4.1.1 and shows that as the netlister traversed the hierarchy of the circuit it stopped when it found leaf cells with **behavioral** views. If this simulation were to run, it would use the behavioral code in those behavioral views for the simulation of the cells. How did the netlister decide to stop at the behavioral views? That is controlled by the **verilogSimViewList** and **verilogSimStopList** variables that control the netlister. Those values are set to default values in the **.simrc** file in the class directory. If you looked in that file you would see that they are set to:

```

verilogSimViewList = ('( "behavioral" "functional" "schematic" "cmos_sch" )
verilogSimStopList = ('( "behavioral" "functional" )

```

This means that the netlister will look at all the views in the View list, but stop when it finds a view in the Stop list. You can modify these lists before the netlisting phase of each of the Cadence Verilog simulators used

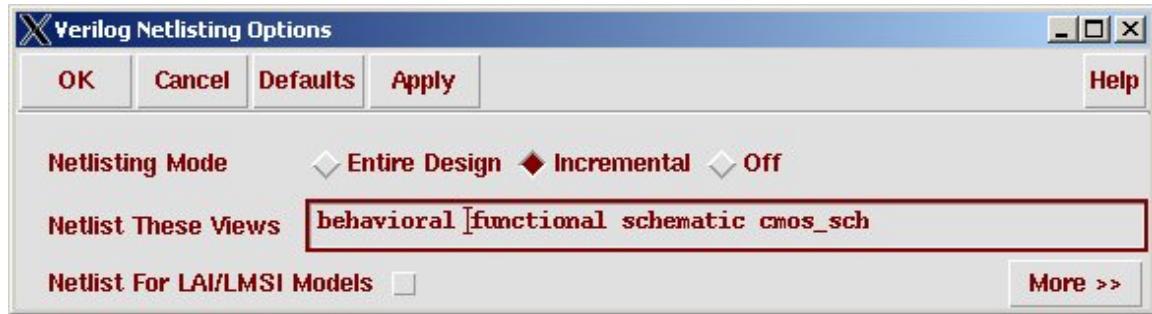


Figure 4.48: The **Setup Netlist** dialog from Verilog-XL

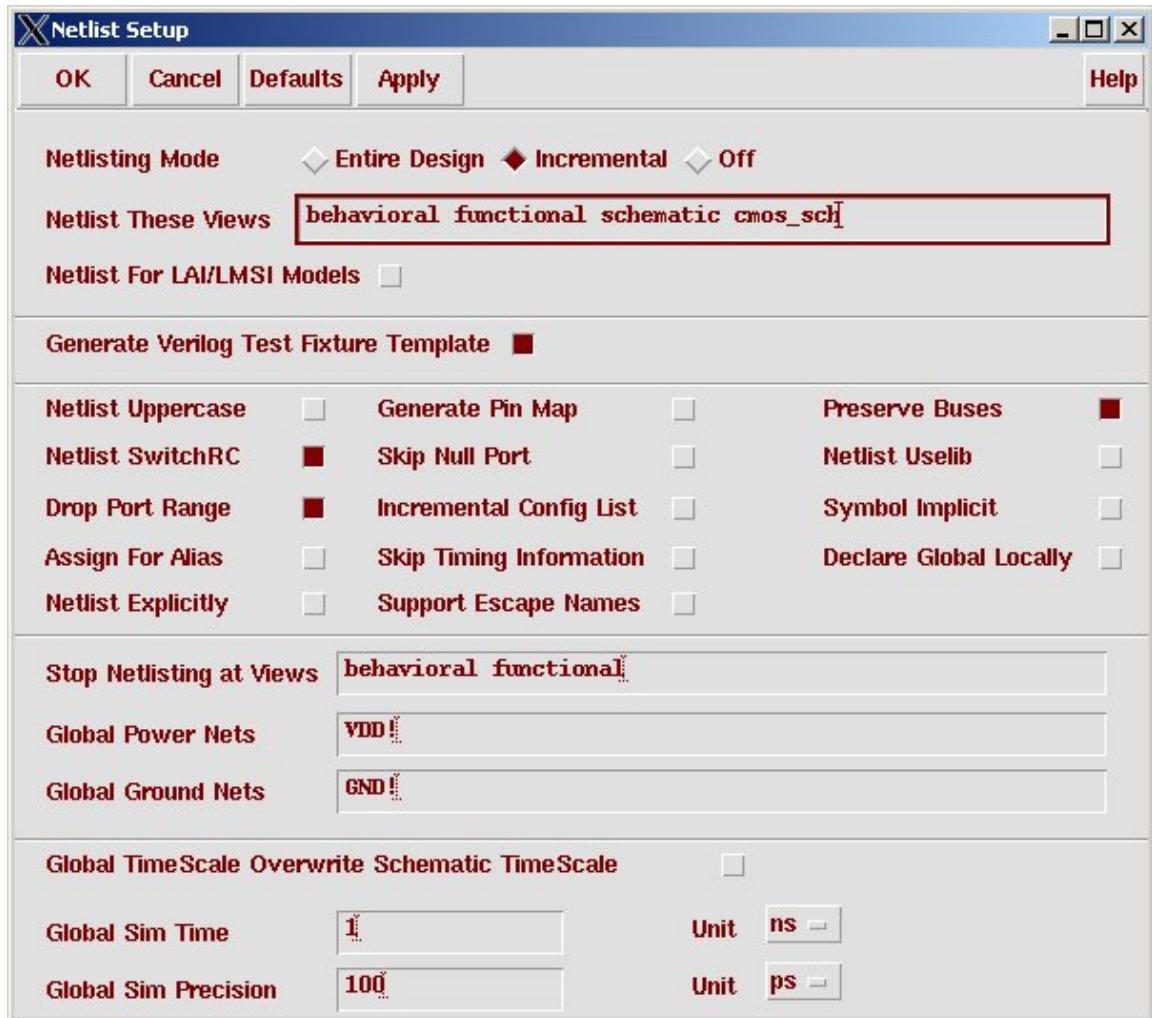
with the Composer schematic capture tool. Using either Verilog-XL or NC\_Verilog you can modify the **verilogSimViewList** before generating the netlist. Use the **Setup → Netlist** menu choice. In Verilog-XL you'll see the window in Figure 4.48, and in NC\_Verilog you'll see the window in Figure 4.49. In either case you can remove **behavioral** from the **Netlist These Views** list. The result of this change and then generating the netlist is seen in Figure 4.50 where the netlister has ignored the **behavioral** views, descended through the **cmos\_sch** views, and stopped at the **functional** views of the **nmos** and **pmos** transistors.

#### 4.4.2 Behavioral Gate Timing

If you're stopping the netlisting process at the **behavioral** views then any timing information in those behavioral views will be the timing used for the simulation. There are a number of ways to encode timing in the behavioral descriptions. One way to get timing information into a behavioral simulation is to include timing information explicitly into your behavioral descriptions. For example, using hash notation like **#10** will insert 10 units of delay into your behavioral simulation at that point in the Verilog code. This can enable a rough top-level estimate of system timing in a description that is a long ways from a real hardware implementation. An example of using explicit timing in a behavioral description is shown in Figure 4.51 for a two-input NAND gate. Another style of description with procedural assignment of the NAND function is shown in Figure 4.52.

Be aware, though, that **#10**-type timing is ignored for synthesis. Synthesis takes timing into account using the gate timing of the cells in the target library. It does not try to impose any timing that you specify in the behavioral description. This is covered in more detail in Chapter 8.

You can also put parameters in your descriptions so that you can override

Figure 4.49: The **Setup Netlist** dialog from NC\_Verilog

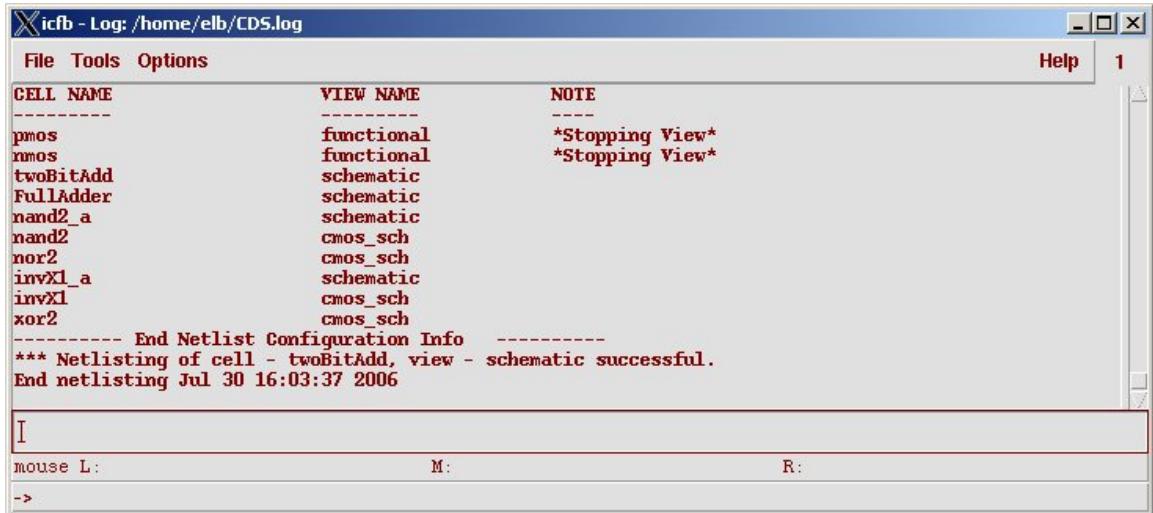


Figure 4.50: Netlisting result after removing **behavioral** from the **verilogSimViewList**

```
module NAND (out, in1, in2);
    output out;
    input in1, in2;

    assign #10 out = ~(in1 & in2);

endmodule
```

Figure 4.51: Verilog description of a NAND gate with explicit timing

```
module NAND (out, in1, in2);
    output out;
    reg out;
    input in1, in2;

    always @(in1 or in2)
        begin
            #10 out = ~(in1 & in2);
        end
endmodule
```

Figure 4.52: Another description of a NAND gate with explicit timing

```
module NAND (out, in1, in2);
    output out;
    reg out;
    input in1, in2;
    parameter delay = 10;

    always @(in1 or in2)
        begin
            #delay out = ~(in1 & in2);
        end
endmodule
```

Figure 4.53: NAND description with **delay** parameter

the default values when you instantiate the module. For example, the NAND in Figure 4.53 has a default delay of 10 defined as a **delay** parameter. This can be overridden when the NAND is instantiated using the syntax

```
NAND #(5) _io(a,b,c);
```

which would override the default and assign a delay of 5 units to the NAND function.

Behavioral Verilog code can also include timing in a **specify** block instead of in each assignment statement. A **specify** block allows you to define *path delays* from input pins to output pins of a module. Path delays are assigned in Verilog between **specify** and **endspecify** keywords. Statements between these keywords constitute a specify block. A specify block appears at the top level of the module, not within any other initial or always block. Path delays define timing between inputs and outputs of the module without saying anything about the actual circuit implementation of the module. They are simply overall timings applied at the I/O interface of the module. An example of our NAND with a **specify** block is shown in Figure 4.54. In this case the delays are specified with separate rising and falling delays. If only one number is given in a **specify** description it is used for both rising and falling transitions.

#### 4.4.3 Standard Delay Format (SDF) Timing

There are two main reasons to use a specify block to describe timing

1. So that you can describe path timing between inputs and outputs of complex modules without specifying detailed timing of the specific circuits used to implement the module

```

module nand2 (Y, A, B);
    output Y;
    input A;
    input B;

    nand _i0 (Y, A, B);

    specify
        (A => Y) = (1.5, 1.0);
        (B => Y) = (1.7, 1.2);
    endspecify

endmodule

```

Figure 4.54: NAND gate description with **specify** block

2. So that you can back-annotate circuits that use this cell with timing information from a synthesis tool.

Synthesis tools can produce a timing information file as output from the synthesis process in **sdf** format (standard delay format). An sdf file will have extracted timings for all the gates in the synthesized circuit. Synthesis programs from **Synopsys** and **Cadence** get these timings from the **.lib** file that describes each cell in the library and from an estimate of wiring delays so they're pretty accurate. They also assume that every cell in the library has a **specify** block that specifies the delay from each input to each output! It is those **specify** statements that are updated with new data from the **sdf** files.

Details on generating an sdf file are in Chapter 8 on Synthesis, but assuming that you have an sdf file, you can annotate your verilog file with this timing information. This will override all the default values in the specify blocks with the estimated values from the sdf file. For example, a snippet of an sdf file for a circuit that uses a NAND gate from the preceding figures would look like:

```

(CELL
(CELLTYPE "NAND")
(INSTANCE U21)
(DELAY
(ABSOLUTE
(IOPATH A Y (0.385:0.423:0.423) (0.240:0.251:0.251))
(IOPATH B Y (0.397:0.397:0.397) (0.243:0.243:0.243))
)
)
)

```

Notice the matching of the timing paths in the specify block to the IOPATH statements in the sdf file, and the the timings are specified in min:typ:max form for both rising and falling transitions. Each instance of a NAND in the circuit would have its own CELL block in the sdf file and thus get its own timing based on the extracted timing done by Synopsys during synthesis. Remember, though, that this is just estimated timing. The timing estimates come from the characterizations of each library cell in the .lib file. You can generate this sdf information after synthesis, or after synthesis and place and route. The sdf information after place and route will include timing information based on the wiring of the circuit and therefore be more accurate than the pre place and route timing.

Details on how to integrate the sdf timing in Verilog-XL and NC\_Verilog simulations will be postponed until Chapter 8 so that we can have structural Verilog simulations and sdf files to use as examples.

#### 4.4.4 Transistor Timing

When you use individual **nmos** and **pmos** transistors in your schematics (as described in Section 3.3), and you simulate those schematics using Verilog-XL or NC\_Verilog, what timing is associated with those primitive switch-level transistor models? That depends on how the transistor models are defined. In our case we have two choices:

1. We can use transistors from the **NCSU\_Analog\_Parts** library in which case the transistors are modeled as *zero delay* switches. There is no delay associated with switching of the transistors.
2. We can use transistors from the **UofU\_Analog\_Parts** library in which case there are 0.1 units of delay associated with each transistor. This is supposed to very roughly correspond to 0.1ns of delay for each transistor which is very roughly similar to the delay in transistors in the  $0.5\mu$  CMOS process that we can use through MOSIS for free class chip fabrication.

Fortunately (by design) each of these libraries has exactly the same names for each of the transistors (**nmos**, **pmos**, **r\_nmos**, **r\_pmos**, **bi\_nmos**, and **bi\_pmos** are the most commonly used devices for digital circuits). This means that you can control whether you are getting the versions with zero delay or the versions with 0.1 units of delay by changing which library you are using. There is a menu choice in the **Library Manager** which can be used to change the reference library name for a whole library. That is, using **Edit → Rename Reference Library ...** (see the dialog box in Figure 4.55) you can change all references to gates in library A to references of that same

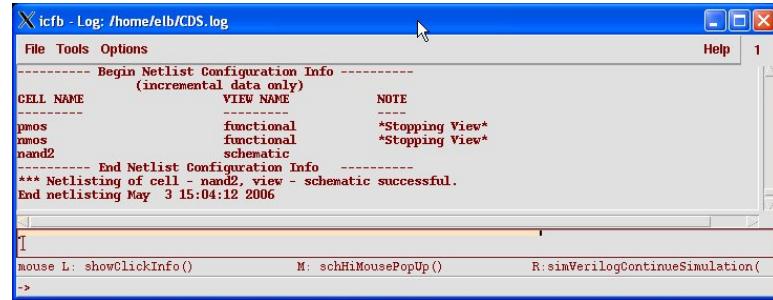


Figure 4.55: **Rename Reference Library** dialog box from Library Manager

gate in library B. The figure shows how you could change all transistor references from the **NCSU\_Analog\_Parts** library to the same transistors in the **UofU\_Analog\_Parts** library and therefore switch between zero delay switch level simulation and unit (0.1 units) delay switch level simulation.

Consider simulating the NAND gate designed as a schematic in Figure 3.12) using the 0.1ns delays on each transistor using the **UofU\_Analog\_Parts** library. In this case you will open the nand2 schematic and simulate using Verilog-XL. The netlist result in the CIW (The result is shown in Figure 4.56) shows that in this case the netlisting has proceeded through the nand2 and down to the **functional** views of the **nmos** and **pmos** devices. These **functional** views are the switch models of the transistor devices. A portion of the simulation waveform for this circuit is shown in Figure 4.57.

What's going on in this waveform? The **y** output signal should behave in a nice digital way but it looks like on one transition of the **b** input it's going to a high-impedance (**Z**) value (shown by the orange trace in the middle of the high and low ranges) for a while. On another edge of **b** the **y** output is going to an unknown (**X**) value (shown by the red box covering both high and low values). This is happening because of the delays at each transistor switch. If you are using the zero-delay transistors of the **NCSU\_Analog\_Parts** library you won't see this effect, but you will see a yellow circle warning and a red transition to show that things are glitching in zero time.



```

X icfb - Log: /home/elb/CDS.log
File Tools Options Help 1
----- Begin Netlist Configuration Info -----
(incremental data only)
CELL NAME      VIEW NAME      NOTE
pmos          functional      *Stopping View*
nmos          functional      *Stopping View*
nand2          schematic
----- End Netlist Configuration Info -----
*** Netlisting of cell - nand2, view - schematic successful.
End netlisting May 3 15:04:12 2006

```

mouse L: showClickInfo()      M: schHiMousePopUp()      R:simVerilogContinueSimulation()  
->

Figure 4.56: Netlist Log for the nand2 Cell

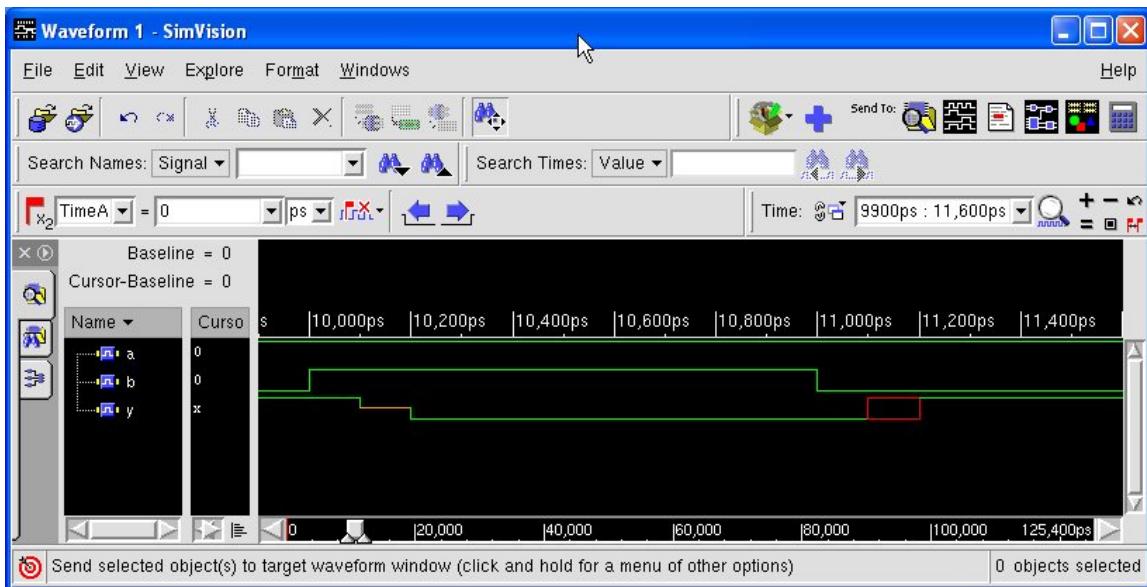


Figure 4.57: Waveform from Simulation of nand2 Cell

The reason that adding a small delay on each transistor causes this behavior is that there are serial pulldown **nmos** transistors pulling down the **y** output. When these transistors switch, it takes time for the result of the switching to be passed to the transistor's output. During the time that the transistor is switching, the output is unknown, or is unconnected. This is undesired behavior because the **Z** and **X** values can propagate to the rest of the circuit and cause all sorts of strange behavior that isn't a true reflection of the circuit's behavior. The issue is that in a real circuit it is true that the output might be unconnected for a brief amount of time during switching, but the parasitic capacitance on the output will hold the output at a fixed value during the switching.

To model this behavior in a switch-level simulation we need some way to tell Verilog to emulate this behavior. That is, we need to make the output node of the nand2 gate “sticky” so that when it is unconnected it sticks to the value it had before it was unconnected. This type of wire in Verilog is a **trireg** wire. It is still a wire, but it's a “sticky” wire whose value sticks when the driver is disconnected from the wire. It acts like a wire from the point of view of the Verilog code (i.e. it can be driven by a continuous assignment), but it acts like a register in terms of the simulation (the value sticks). To inform Composer that it should use a **trireg** wire on the output of the nand2 instead of the plain wire type we need to add an **attribute** to the wire.

Select the output **y** wire in the schematic and get its properties with the **q** hotkey (or the properties widget). At the bottom of this dialog box you can **Add** a new property to this wire. As shown in Figure 4.58 add a property named **netType** (make sure to get the spelling correct with a capital “T”), type **string**, and value **trireg**. The result is seen in the properties box shown in Figure 4.59. I've also changed the **Display** to **Value** so that the **trireg** value can be seen on the schematic to remind you that this additional property has been set.

In order for the Composer netlister to pay attention to this new property, you also need to set the **Switch RC** property in the netlisting options in Verilog-XL, but this should have been already set for you by the default **.simrc** file. Once you have the **trireg** property on the output node, the output will no longer go to a **Z** value. If the output is ever un-driven it will hold its previous value. Unfortunately, transitions that caused an **X** in the original simulation will still cause an **X** in the nand2 even with the **trireg**. This is because of the delay on the transistors again. There are situations where the pmos devices have turned on but the series nmos stack hasn't turned off yet because of the delay through each device. What you need now is a transistor at the top of the series stack (nearest the output node) that is weak enough to be over-driven by the pmos pullup if they're ever on at the same time.

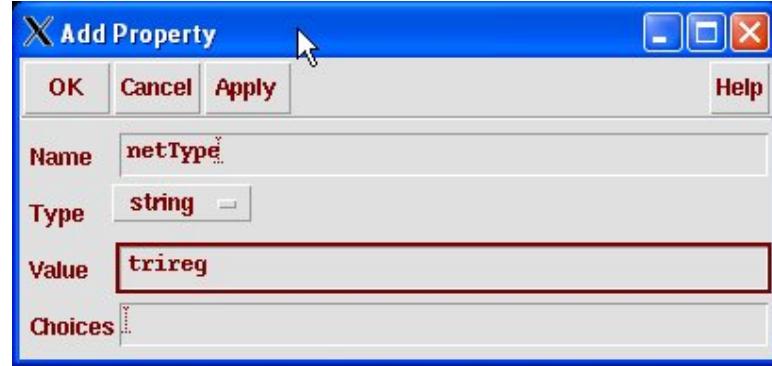


Figure 4.58: Adding a Property to a Net

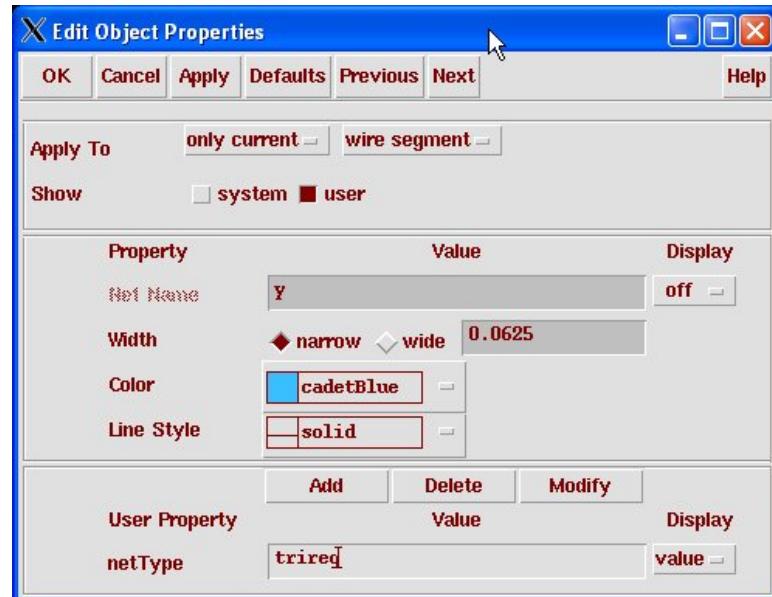


Figure 4.59: A Net With a **netType** Property

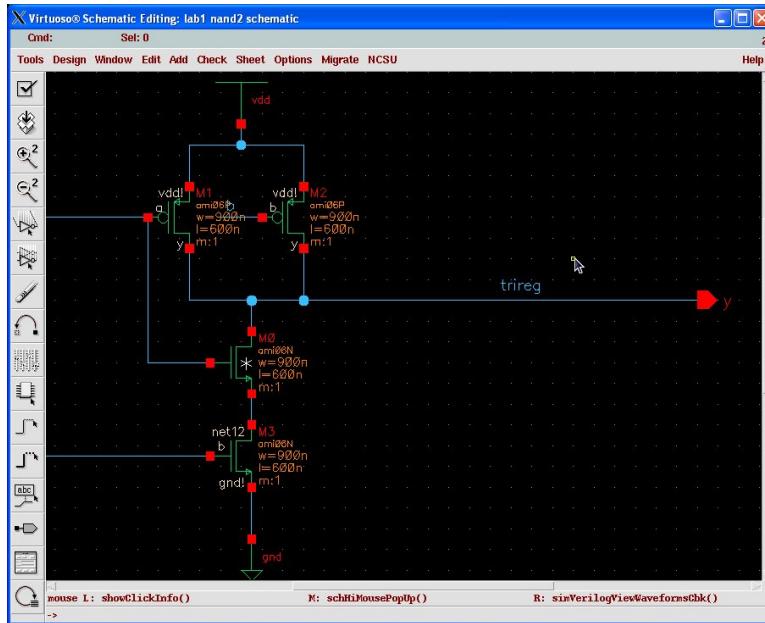


Figure 4.60: Nand2 Cell with **r\_nmos** and **trireg** Modifications

The weak transistors are called **r\_nmos** and **r\_pmos** (the “r\_” means that they are “resistive” switch models). Replacing the topmost nmos device with a weak device doesn’t change the function of the nand2 gate, but results in a cleaner waveform at the output. The schematic is seen in Figure 4.60. Figure 4.61 shows the same portion of the waveform but with both modifications: **trireg** on the output node and a **r\_nmos** device at the top of the pulldown stack.

Of course, you may not care about the glitches on the outputs of the nand2 gate. In a synchronous system where the gate outputs are being sent to a storage element like a latch or flip flop, the glitches should die out long



Figure 4.61: Waveform from Simulation of nand2 Cell with Modifications

before they reach the input of the storage element. In that case you may not care if you make these tweaks to your transistor circuits or not. Also, in general, it may not be possible to design a transistor circuit that had no glitches for all possible input changes using the Verilog transistor switch delay model.

I suggest that you at least add **netType trireg** attributes on the outputs of combinational circuits that you design. That's a simple thing to do and helps a lot in terms of confusing outputs.

# Chapter 5

## Virtuoso Layout Editor

**V**IRTUOSO is the layout editor from Cadence that you use to draw the composite layout for the fabrication of your circuit. It is part of the `dfll` tool suite. The process of drawing layout amounts to drawing lots of colored rectangles with a graphical editor where each color corresponds to a fabrication layer on the integrated circuit. There are a lot of design rules for how those layers interact that must be followed very carefully. The circuit described by these colored rectangles can be extracted from this graphic version and compared to the transistors in the schematic. The layout is eventually exported in standard format called **stream** or **gdsII** and sent to the foundry to be fabricated.

This chapter will show how to use the Virtuoso Layout Editor to draw the composite layout and how to check that the layout obeys design rules using DRC (Design Rule Checking). It's also critical that the layout correspond to the schematic that you wanted. A Layout Versus Schematic (LVS) process will check that the extracted circuit from your layout matches the transistor schematic that you wanted. What we're after is for all the different views of a cell to match in terms of the information about the cell that they share. The **schematic** or `cmos.sch` view should describe the same circuit that the **layout** view describes, the **symbol** view should have the same interface as both of those views, and the **behavioral** view should also share that interface. Eventually we'll add **extracted** and **abstract** views for other tools, which should also match the essential information. You may also use **config** views for analog simulation.

### 5.1 An Inverter Schematic

We'll demonstrate the layout process with a very simple inverter circuit. First you need a **schematic** view that describes the circuit you want. You

can then simulate this **schematic** view with a Verilog simulator like Verilog-XL or NC\_Verilog to verify that it's doing the right thing. This is important because once you design the layout in the **layout** view, you need something to compare it to to make sure the layout is correct. In our case we always refer back to the **schematic** view as the "golden" specification that describes the circuit we want.

### Starting Cadence icfb

Because Virtuoso is part of the Cadence dfll tool suite, the first step is starting Cadence icfb with the `cad-ncsu` script. The procedure for starting icfb, making a new library, and attaching the technology are described in Chapter 2. Make sure you have a library with the **UofU\_TechLib\_ami06** technology attached.

### Making an Inverter Schematic

The procedure for designing a schematic using the **Composer** schematic capture tool is described in Chapter 3. The procedure for using transistors in a schematic is specifically described in Section 3.3. If you use **nmos** and **pmos** transistors from the **NCSU\_Analog\_Parts** library you will get transistors that simulate with zero delay in the Verilog simulators. If you use transistors from the **UofU\_Analog\_Parts** library you will get transistors that simulate with 0.1 time units of delay in those same simulators. This is described in Section 4.4.

For this inverter, use **Composer** from the **Library Manager** to create a new cell view with a cell name of `inverter`. Make the view type `cmos_sch` because this will be an individual standard cell consisting of nothing but transistors. Make an inverter using **nmos** and **pmos** transistors from either of the **Analog\_Parts** libraries. Use the properties on the transistors to change the width of the transistors so that the **pmos** device is `6u M` wide and the **nmos** device is `3u M` wide. You can do this when you instantiate the devices, or later using the properties dialog box (use the `Q` hotkey or the **properties** widget). Your **inverter** schematic should look like Figure 5.1 when you're done.

*It's easy to change all cells (like **nmos**) from one library to be from another library later.*

*You can change the reference library from **NCSU\_Analog\_Parts** to **UofU\_Analog\_Parts** for example.*

### Making an Inverter Symbol

Now is a good time to make a symbol view of the inverter too. Follow the procedure from Chapter 3 to make a symbol view. If you like gates to look like the standard symbols for gates (always a good idea) your symbol might end up looking something like that in Figure 5.2.

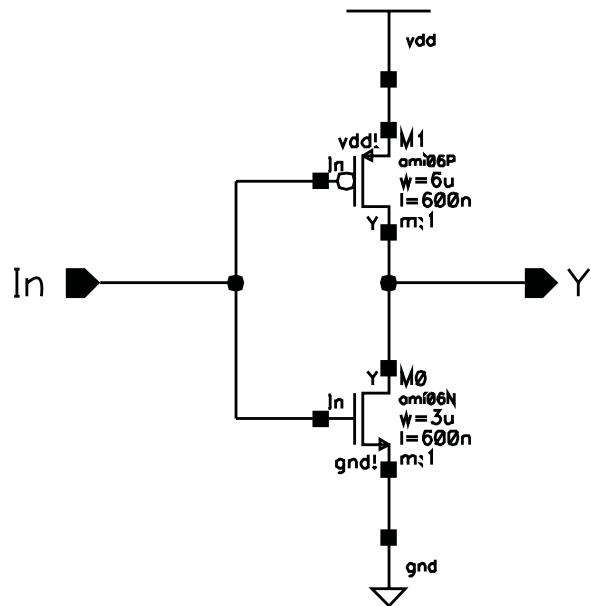


Figure 5.1: Inverter Schematic

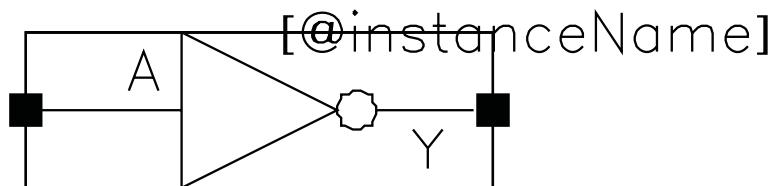


Figure 5.2: Inverter Symbol

Now **check and save** the **symbol** and **cmos.sch** views. They should both check and save without errors before you proceed to the layout!

## 5.2 Layout for an Inverter

*Note that there is a tutorial on Virtuoso that comes with the tools. Although it will not use our same technology information it may show you even more tricks of the tools.*

The process of drawing CMOS layout is to use the graphical editor to draw colored rectangles that represent the fabrication layers in the CMOS fabrication process.

### Creating a new layout view

In the **Library Manager**, in the same library as you used for the **inverter cmos.sch** view, create a new cell view. This cell view should have the **Cell Name** of **inverter** so that it will be a different view of the same inverter cell. The view type should be **layout**, and the tool should be **Virtuoso** (see Figure 5.3).

This will open up two windows; The **Virtuoso Editing** window where the layout is drawn and the **LSW** (Layer Select Window), where you select the layers (diffusion, metal1, metal2, polysilicon, etc) to draw.

If the layout is going to be a Standard Cell, the height of the cell as well as the width of the VDD and VSS power supply wires must be defined so that cells can eventually abut each other and have the power supply connections made automatically as the cells abut each other. To avoid DRC errors when abutting the cells, it is also important to keep the left and right borders of the cell free of any drawing that might cause an error when the cell is abutted to another cell. The n-tub is usually aligned with the cell borders so that it connects into a seamless rectangle when abutted. Usually cell templates of standard heights containing VDD and VSS contacts (wires) as well as default nwell dimensions are used. For this tutorial, you can just pick some reasonable dimensions, but keep in mind that later you'll want to plan for a particular power pitch.

### Drawing an nmos transistor

*There are two types of active defined in our technology file: nactive and pactive. They're different colors so you can tell them apart.*

Click on the nactive (light green) layer in the LSW window. In the Virtuoso Layout Editor window, press **[r]** to activate the Rectangle command. Now you can draw a rectangle by selecting the start and end points of the rectangle. This first **nactive** rectangle is shown in Figure 5.4.

Press **[k]** to activate the **Ruler** command. You can click on one of the corners of the green rectangle to place the ruler and measure the sides (typ-

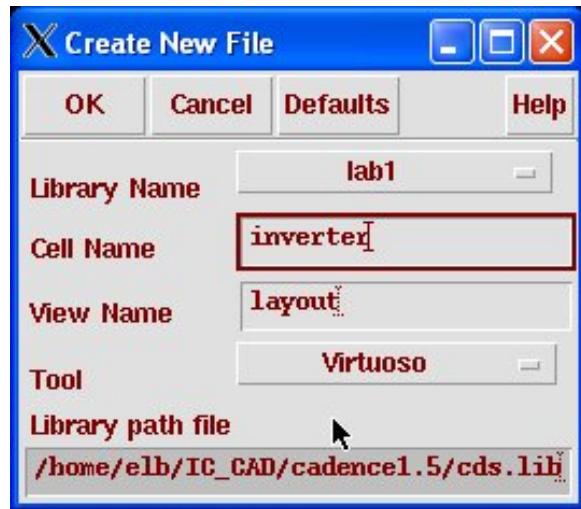


Figure 5.3: Dialog for Creating a **Layout View** of the **inverter** cell

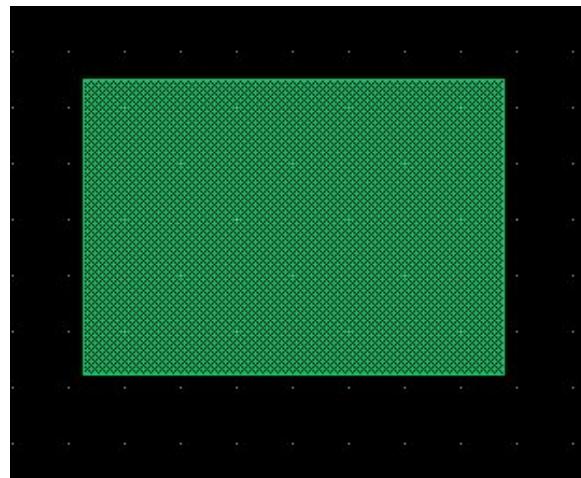


Figure 5.4: Initial **nactive** rectangle

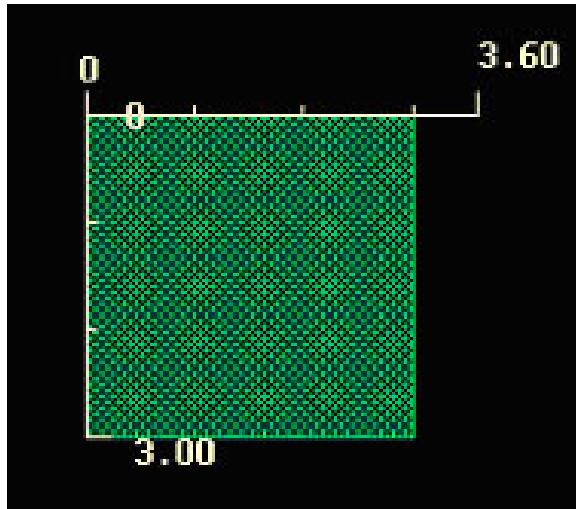


Figure 5.5: **nactive** Rectangle with Measurement Rulers

The terms “active” and “diffusion” are both used to describe the same layers in the layout.

ing [K] will clear all rulers). Remember that our N-transistor in the inverter **cmos.sch** view had a W/L of 3um/0.6um. It’s important to keep the width and length of a transistor in mind when you’re drawing active regions. The length is measured in the direction between the source and drain connections. The width is measured in the orthogonal direction. The width of the diffusion will be usually the same as the width of the transistors but the length of diffusion region has to account for the source/drain contacts and the actual gate length. In order to keep the parasitic capacitance low, the source and drain active regions should be kept as small as possible. This means don’t have extra unnecessary active area between the device and the contact. A general rule of thumb is to have the length of the diffusion region to be 3um + Gate length in the 0.5 micron technology that we’re using.

In Figure 5.5 you can see the rulers measuring a rectangle of **nactive**. A very useful command to know about is the [S] **stretch** command. You can use this to stretch the rectangle to be the right size once you have the rulers to guide you.

Now you need to add the source and drain contacts to the **nactive** regions. Contacts are “sandwiches” of active, contact (**cc**), and metal1 (**M1**) layers overlapped on top of each other. Contacts are special in the **MOSIS SC MOS** rules in that they have to be an exact size. Most other layers have a **mx** or **min** size, but contacts in the **UofU\_Techlib\_ami06** technology must be exactly 0.6um X 0.6um in size.

The easiest way to draw a contact is to use the pre-defined contacts in the technology. We have pre-defined in the technology file all the contacts

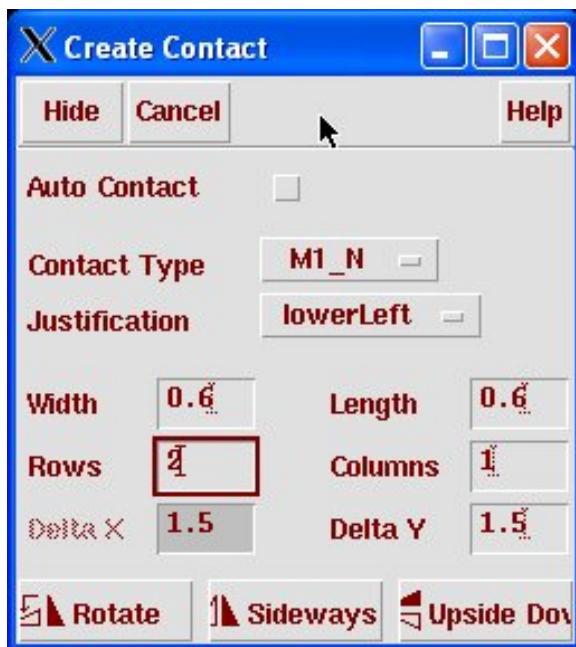


Figure 5.6: Create Contact Dialog Box

that you're likely to need. use the **Create → contact...** menu choice or press . This starts up a **create contact** window. In this window you can select any from the list of predefined contacts. You can also make an array of contacts if you have an area to contact that is larger than the area covered by just a single contact. In general you should use an array of as many contacts as will fit in the area that you're trying to contact. Contacts have non-negligible resistance compared to the pure metal connections so adding more contacts reduces the resistance of the connection by offering parallel resistors for the entire connection. The **create contact** dialog box looks like that in Figure 5.6. Note that we're making arrays of contacts with one column and two rows. You can change this later if desired by selecting the contact array and using the properties.

Pre-defined contacts are defined in the technology file. The pre-defined contacts that are available in this technology file include:

**M1.P:** Connection between Metal 1 and Pactive (source and drain of **pmos** devices) that includes the pselect layer

**M1.N:** Connection between Metal 1 and Nactive (source and drain of **nmos** devices) that includes the nselect layer

**NTAP:** Nwell connection used for tying the Nwell to Vdd

*Connections from Metal 1 “down” to active or Poly are known as “contacts” and connections between different metal layers are called “vias”*

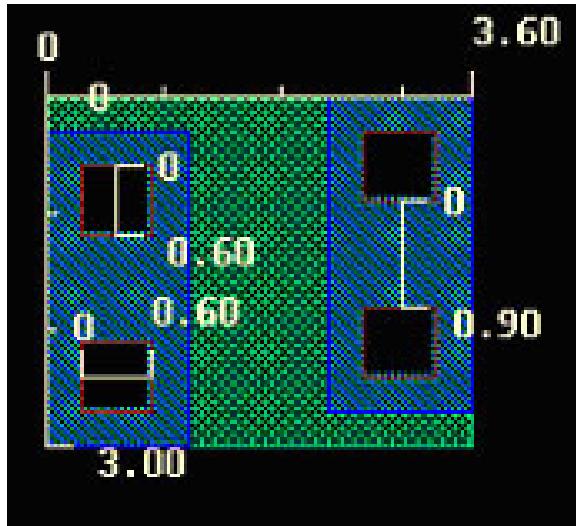


Figure 5.7: **nactive** showing source and drain connections

**SUBTAP:** Substrate connection used for tying the substrate to GND

**P\_CC:** Conenction between Metal 1 and Pactive without select layers (can be uses if it's already inside a pselect layer)

**N\_CC:** Conenction between Metal 1 and Nactive without select layers (can be used if it's already inside an nselect layer)

**M1\_POLY:** Connection between Metal 1 and Polysilicon

**M1\_ELEC:** Conenction between Metal 1 and the Electrode layer (Poly2).

**M2\_M1:** Connection between Metal 1 and Metal 2

**M3\_M2:** COnnection between Metal 2 and Metal 3

Our **nmos** transistor layout with arrays of two **M1\_CC** contacts on source and drain and rulers showing all the dimensions is shown in Figure 5.7. Of course, you can draw the contacts using rectangles of **nactive**, **cc**, and **M1** layers directly, but it's much easier to use the pre-defined contacts.

Now you can draw the gate of the transistor by putting a rectangle of polysilicon (**POLY**) over the active. It's important to note that the poly must overlap the active in order to make a transistor. When the transistor is fabricated there will be no diffusion implant in the channel region underneath the poly, but for the layout you must overlap active and poly to make transistors. The process that extracts the transistor information from the layout uses this overlap to identify the transistors.

It turns out that it's not enough to draw **nactive** to let the foundry know that you want n-type silicon for this device. You also need to put an **nselect** "selection" layer around the **nactive** region to signal that this active region should be made n-type. This is because of how the layers are represented in the fabrication data format. In that format there's only one layer for all active regions, and the N-type and P-type active is differentiated with the select layers. We use different layers for **nactive** and **pactive** and make them different colors so it's easy to tell them apart, but they both become generic "active" when they are output in **stream** format so the select layers are required.

You could have used the **M1\_N** contacts but because we drew a separate **nselect** rectangle around the whole transistor, it's not necessary to include the redundant **nselect** around the contacts. It wouldn't hurt anything, though. As long as the overlapped **nselect** rectangles cover the area that you want, it's just a matter of aesthetics whether you have one big rectangle or lots of overlapping smaller rectangles. Personally, I like the cleaner look of one large rectangle. It helps me see what's going on without the visual clutter of lots of overlapping edges.

The final **nmos** transistor layout with the (red) polysilicon gate, **nselect** region (green outline) is shown in Figure 5.8. Note that the *width* of the red **POLY** layer defines the transistor's **length** (0.6u M). The *width* of the transistor is the height of the **nactive** layer. The transistor gate must overlap the width of the transistor by 0.6 microns. Note also that the **nselect** layer must overlap the **nactive** by 0.6 microns. The source and drain regions of this transistor have connections to the M1 (first level of metal interconnect) through the **M1\_CC** contacts. You can connect to these regions using a rectangle of M1 later.

### Drawing a pmos transistor

Now complete the same set of steps to make a **pmos** transistor somewhere in your layout window. This is a very similar process to making the **nmos** device. The differences are:

- Use **pactive** instead of **nactive** for the active (diffusion) layer.
- Use **M1\_CC** (or **M1\_P**) contacts for the source and drain regions, or use a "sandwich" of **pactive**, **cc**, and **M1** layers with the correct dimensions. Remember that the **cc** contact must be exactly 0.6u M by 0.6 u M in this technology. The **pactive** and **M1** layers must overlap the **cc** layers by 0.3u M. As before, it's easier to use the pre-defined contacts using the **Create → contact...** menu choice or press .

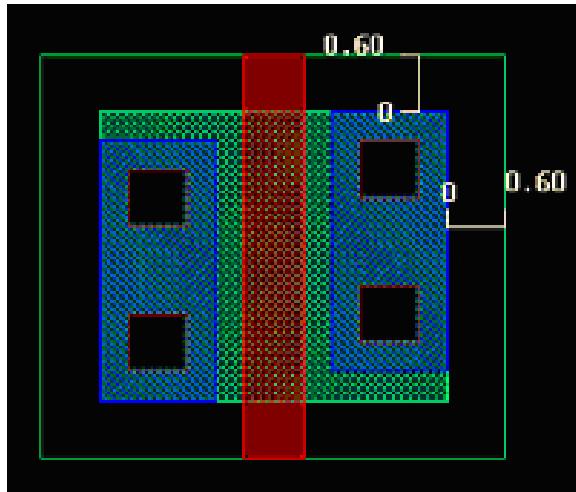


Figure 5.8: NMOS transistor layout

- Because of the mobility difference between **nactive** and **pactive**, make the **pmos** device twice as wide as the **nmos** device to roughly equalize drive capabilities. In other words, make the **pmos** transistor 6u M wide, and keep the length the default 0.6u M long.
- Surround the **pactive** rectangle of the **pmos** device with **pselect** so that the foundry knows to make this active region P-type.

The result of making this **pmos** transistor is shown in Figure 5.9. Note that the transistor is twice as wide as the **nmos** device, and has arrays of **M1\_P** contacts that are four rows high.

However, because of the processing technology used for this set of technology files, we're not done yet. The **pmos** device must live inside of an **NWELL**. This allows the **pactive** of the **pmos** device to be surrounded by N-type silicon, while the **nmos** device we drew earlier lives in the P-type silicon substrate. Draw a box of **NWELL** around the entire **pmos** device with overlaps around the **pactive** of at least 2.1u M. The completed **pmos** device with its surrounding **NWELL** is shown in Figure 5.10.

### Assembling the Inverter from the Transistor Layouts

Now that you have layout pieces of **nmos** and **pmos** devices, you can assemble them into an inverter by connecting the source and drain connections of the transistors to the appropriate places. Connections are made using rectangles of any conducting layers, but metal layers are the usual choice for making connections if that's possible.

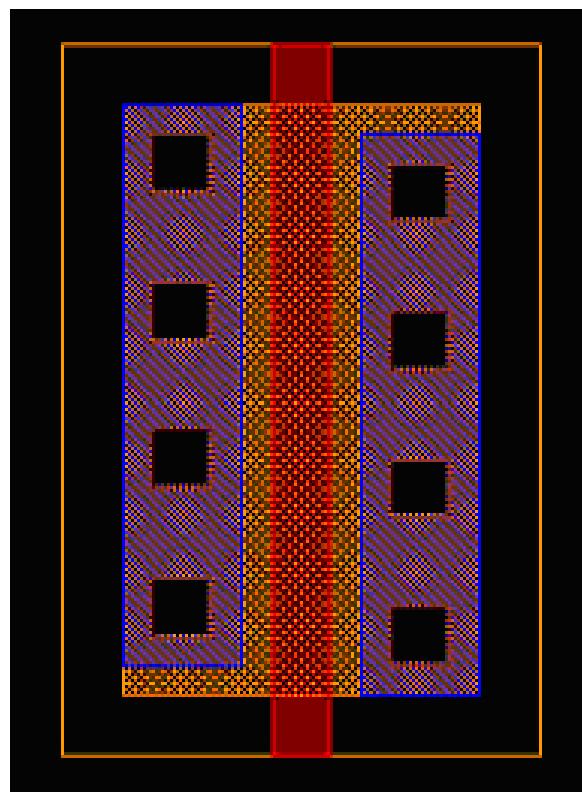


Figure 5.9: A **pmos** transistor 6 $\mu$  M wide and 0.6 $\mu$  M long

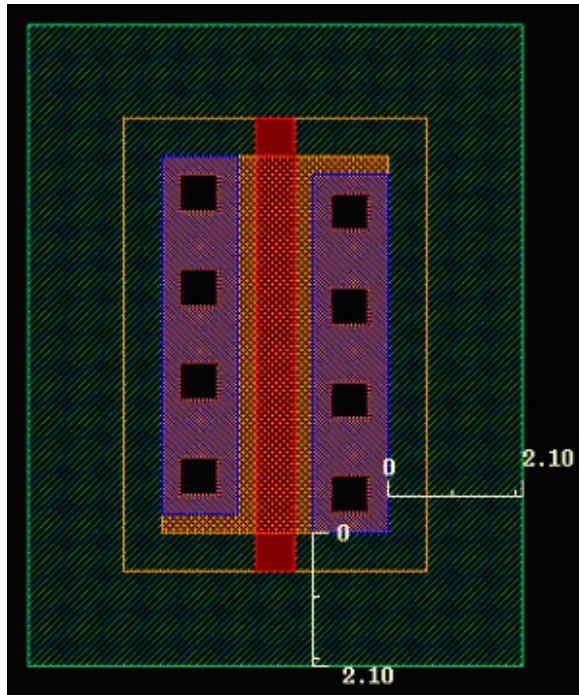


Figure 5.10: A **pmos** transistor inside of an **NWELL** region

Select the entire layout piece of one of your transistors using the left mouse button. Starting with the cursor in the arrow shape click and drag with the left button pressed. The selected rectangles are highlighted in white. Now hover your mouse over the selected pieces until the cursor changes to the “move” character (arrows pointing up, down, right, and left), and move the whole piece of layout using the left button again. This time everything that has been selected will move as you drag. Drag the layout so that the transistors are arranged vertically with a gap of at least 1.8u M between the **nwell** of the **pmos** and the **nselect** of the **nmos** transistors.

Once the transistors are place, you can connect the drains of the two transistors to make the output node of the inverter using a rectangle of **metal1**. You can also connect the gates of the transistors together with a rectangle of **poly**, and make the input connection from the **poly** to **metal1** using a **M1.POLY** contact.

You can also use “paths” for making these connections. A “path” is like a “wire” in Virtuoso. To make a path, select the layer you want, and then select **Create → Path** or use the **[P]** hotkey. This enters “path creating mode” in the editor. You start paths with a left click, make bends in the path with another left click, and finish a path with a double left click. You can

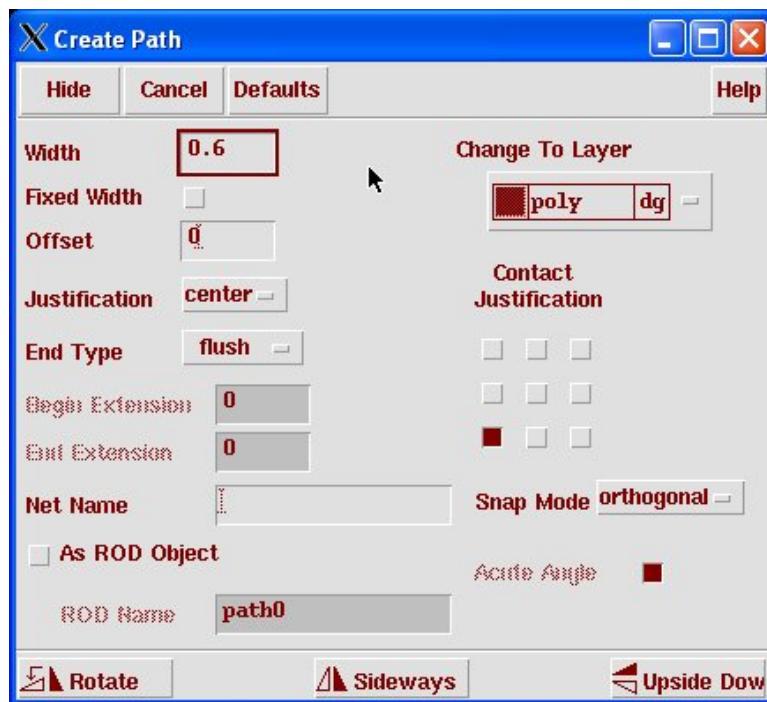


Figure 5.11: Dialog box for the **path** command

also get a dialog box that controls how the path is created using the **[F3]** function key. This dialog is shown in Figure 5.11 for a **poly** wire.

A few things to note about this **path** dialog are:

- You want to set the snap mode to **orthogonal** to keep your wires on vertical and horizontal axes
- You can change the **justification** of the wire from center to left or right depending on whether you want your mouse to define the center, or one of the edges of the path
- The width of the path defaults to the minimum allowable width for that layer in the technology. You can change this if you want a fatter wire, but don't make it narrower than the default.
- You can use the dialog box to change layers by selecting a different layer. The **path** mode will have you place an appropriate contact, and then switch to a path of that new layer.

The layout for the inverter with the input and output connections made between the two transistors is shown in Figure 5.12.

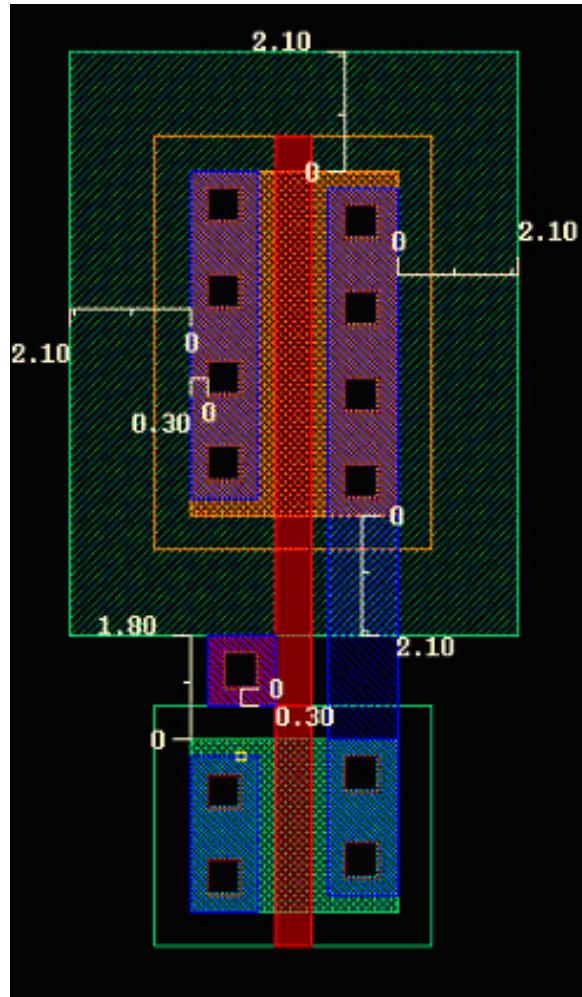


Figure 5.12: Inverter Layout with Input and Output Connections Made

Now make the source connections to the **Vdd** and **GND** power supplies at the top and bottom of the inverter. These are simply wide metal connections in **metal1**. Don't forget to make the connections from the wide wires to the source nodes of the transistors. Make the power supply wires at least 2.4u M wide to avoid electromigration problems in the future. The inverter with the power supply wires is shown in Figure 5.13.

Although this is a “functional” inverter layout at this point it's not complete. In CMOS you must make sure that the substrate is tied low with a connection to **GND** and that the **nwell** is tied high with a connection to **Vdd** to avoid latchup effects due to the parasitic devices formed by the wells. These contacts are most easily made using the **SUBTAP** and **NTAP** pre-defined contacts in the technology. These taps are contacts from the **nactive** and **pactive** active regions to **metal1**, but they are meant specifically for connections to the well and substrate. Any connection to **pactive** which is outside of an **nwell** is a substrate connection, and any connection to **nactive** inside of an **nwell** is a well connection. These contacts are a standard width so that they can be tiled in a consistent way. When you start fitting your cells to the library template you will see that cells in the **UofU\_Digital** library are sized in units of 2.4u M chunks that match with the well and substrate tap widths for convenience.

The inverter with the well and substrate connections is shown in Figure 5.14. The **nwell** layer in this example has been extended by 0.3u M beyond the **Vdd** metal layer to make it easier to abut these cells in a larger layout later on.

Finally, we need to add connection name information to our layout. Remember that all the different views of a cell must be consistent in their I/O interface (at least). The inverter **cmos\_sch** view and **symbol** view each have an input connection named **In** and an output connection named **Out**. In addition, our layout needs to identify the power supply connections explicitly. We'll specify the power supply connections with the names **vdd!** and **gnd!**. The “!” character is a flag to make these names global. If you don't use this flag you will have to export your power supply connections as pins to your cell. That's not incorrect, it's just a hassle.

Remember that Verilog  
names are case sensitive

In order to make connections in the layout view we need to add **pins** to the design. **Pins** are similar to the red-arrow pins in the **cmos\_sch** view, but in the layout we identify pieces of the layout with the pins. These are called **shape pins** because they identify a shape in the layout with the pin. There are other types of pins you can make in Virtuoso but please use only **shape pins** in this tool flow because other steps in the process depend on this.

A **shape pin** is a piece of layout (a rectangle) in a particular layer (i.e. **metal1**) that also has a pin name attached to it. Add a **shape pin** using the **Create → Pin** menu choice. The dialog box is shown in Figure 5.15. Note

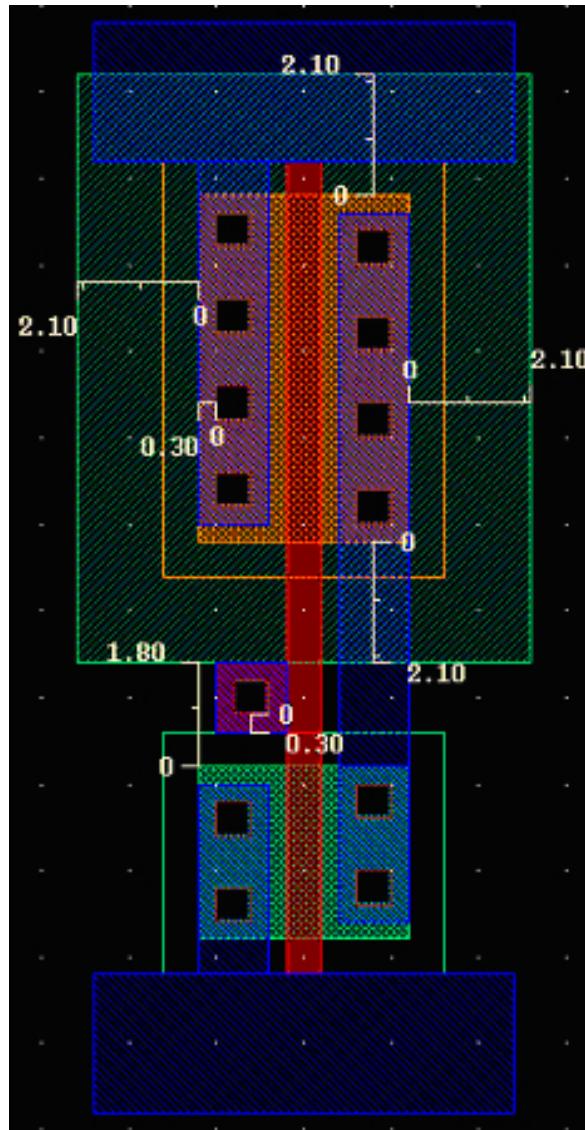


Figure 5.13: Inverter Layout with Power Supply Connections

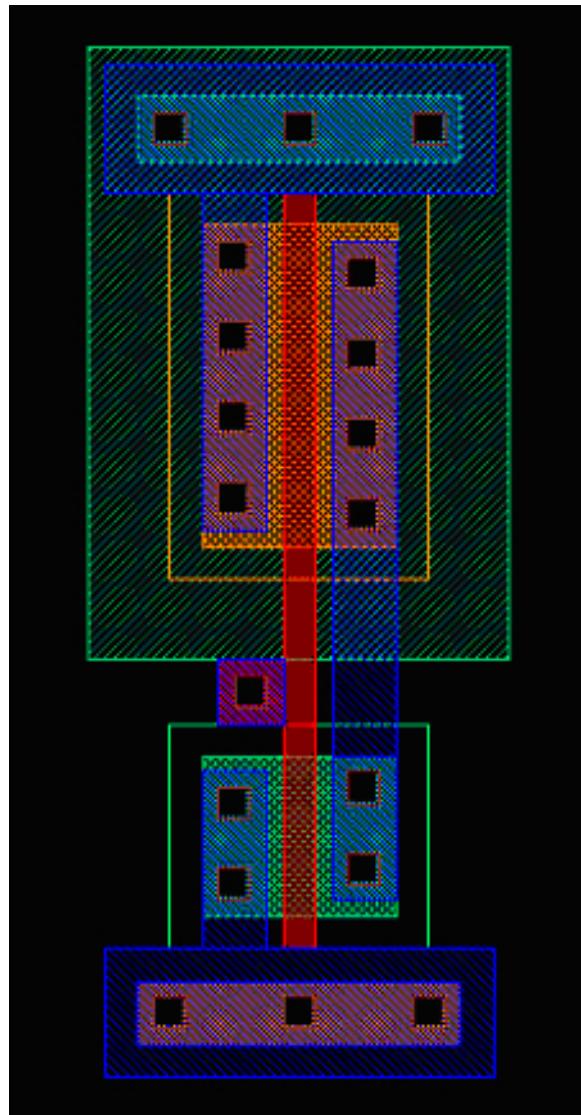


Figure 5.14: Inverter Layout with Well and Substrate Connections

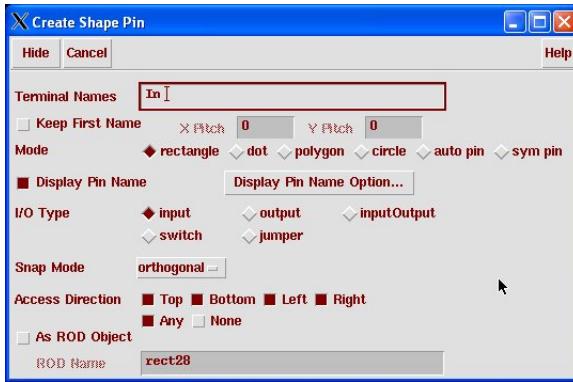


Figure 5.15: Shape Pin Dialog Box

that you won't see this exact dialog box until you select **shape pin** as the pin type. Once you are in the **shape pin** dialog box, make sure that you have selected the **Display Pin Name** option, that you're placing a rectangle, that you have selected the correct direction for the pin (i.e. **input**), and that you have entered the name (or names) you want to add. You select the layer that you want the pin to be made in (i.e. **metal1**) in the **LSW** (Layer Selection Window). When all this is correct, draw a rectangle of the layer in the spot that you want to have identified with the connection pin in the layout.

In our inverter place a **metal1 input shape pin** named **In** directly over the **M1\_POLY** contact for the inverter's input, and a **metal1 output shape pin** named **Out** somewhere overlapping the output node connection. For the power supply connections, make the shape pins as large as the power supply wire, and in **metal1**. This will define that entire wire as a possible connection point for the power supply. Power supply connection are made with the names **vdd!** and **gnd!**, and should be **input/output** type. The final layout for the inverter showing the pins with the pin names visible is shown in Figure 5.16. Note that it's important for a later stage in this process that the little "+" near the pin name that defines where the label is to be inside the shape of the **shape pin**.

Before proceeding to the next step, make sure to save your layout. In fact, you should probably save the layout often while you're working. There's nothing quite as frustrating as doing hours of layout only to have the machine or the network crash and you lose your work! Save often!

## Using Hierarchy in Layout

Once you have a piece of layout, you can use that layout as an instance in other layout. In fact, that's what you've already done with the pre-defined

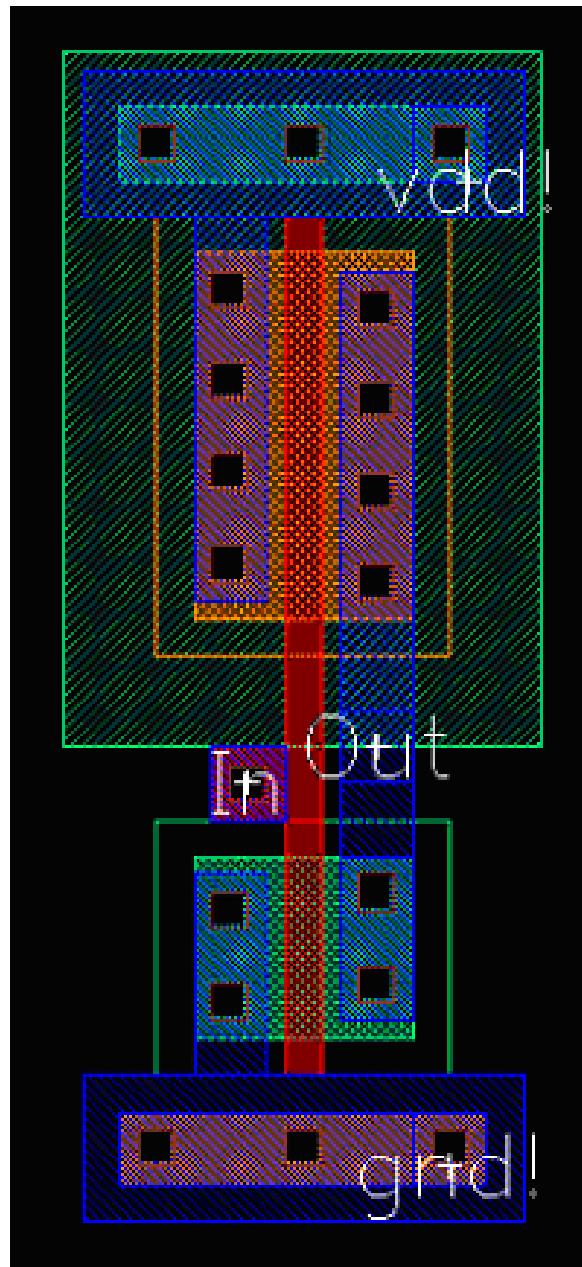


Figure 5.16: Final Inverter Layout

contacts in the **Add → Contact** step. The pre-defined contacts are actually pieces of layout in another layout view inside the **UofU\_TechLib\_ami06** technology library. When you put one in your layout you were really placing an instance of that cell in your layout.

You can see this by changing the scope of what's visible in the layout editor. That is, you can change things so that the only shapes you see are the shapes that are directly placed in your current cell view, and the hierarchical instances will be hidden with a red box around them. Try toggling the hierarchical display using **shift-f** and **ctrl-f** to see how this works.

As a very simple example of including other hierarchical cells in a layout, consider a simple circuit with four inverters in a row. Figure 5.17 shows a layout with four instances of an inverter where the output of each inverter is connected to the input of the inverter on its right. Figure 5.18 shows the same cell with the hierarchy expanded to look inside each of the inverter instances.

### 5.2.1 Virtuoso Command Overview

Virtuoso, like any full-featured layout editor, has hundreds of commands. This tutorial has only scratched the surface, but these commands should be all you need for most basic layout tasks. These tasks and commands are:

**Drawing Rectangles:** The basic components of a **layout** view are colored rectangles that represent fabrication layers. Layers are selected in the **LSW** Layer Selection Window. Rectangles are drawn with the **Create → Rectangle** menu choice, the rectangle widget, or the **r** hotkey.

**Connecting Rectangles:** Rectangles of the same material are electrically connected by abutment or by overlapping. Rectangles of different materials are electrically connected using contacts or vias. These can be drawn with a “sandwich” of layers with an appropriate contact shape between them (**cc**) for connecting **metal1** “down” to active or poly, and **via<n>** for connecting **metal<n>** to **metal<n+1>**. So, for connecting **metal2** to **metal3** a **via2** is used. Vias and contacts must be exactly 0.6u M by 0.6u M in this **UofU\_Techlib\_ami06** technology.

**Using Paths:** A convenient way to draw “wires” out of rectangles is to use the **path** feature through the **Create → Path...** menu choice or the **p** hotkey. Choose the layer from the **LSW**. Single-click the left button to change directions, double click the left button to end a wire.

**Moving and Stretching:** Select layout using the left button, or click and drag to select groups of layout objects. If the cursor changes to the

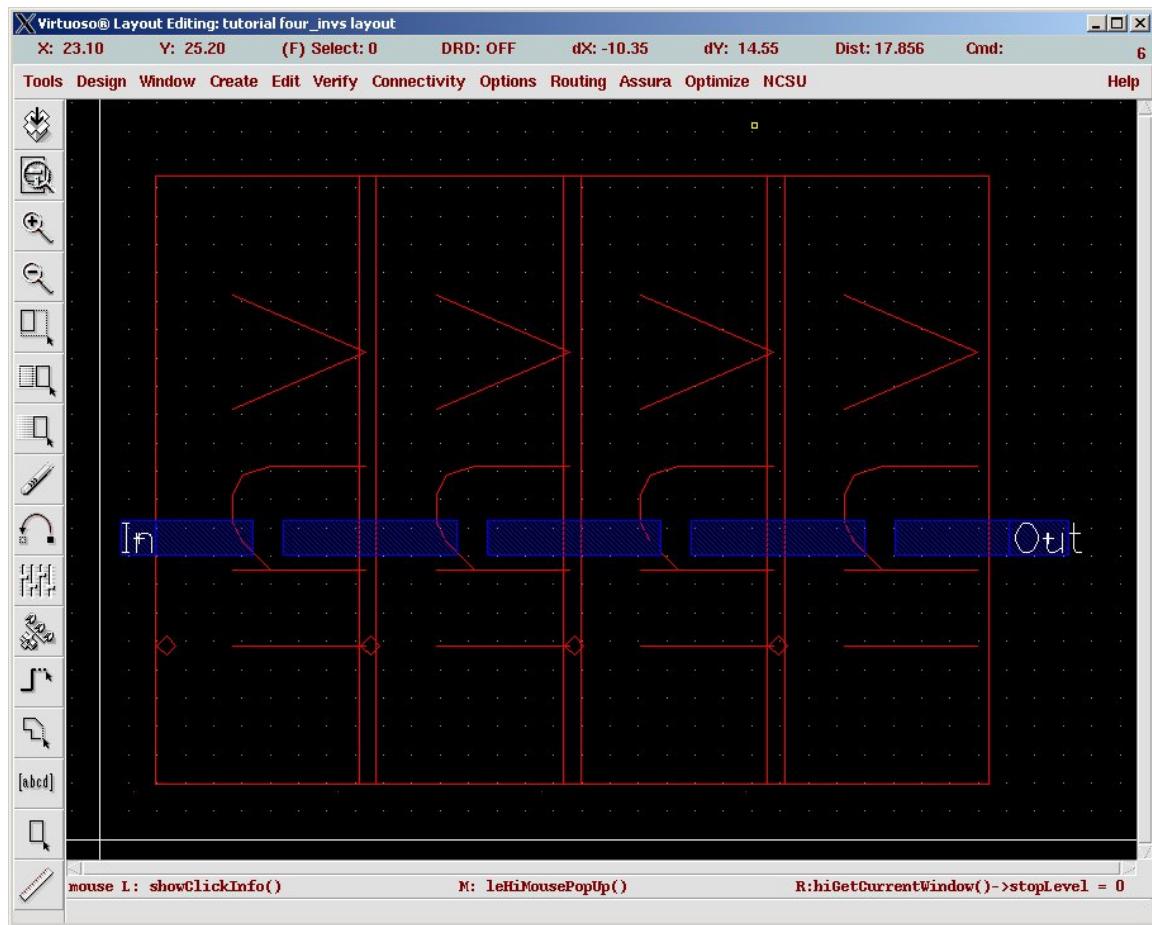


Figure 5.17: Layout with four inverter instances

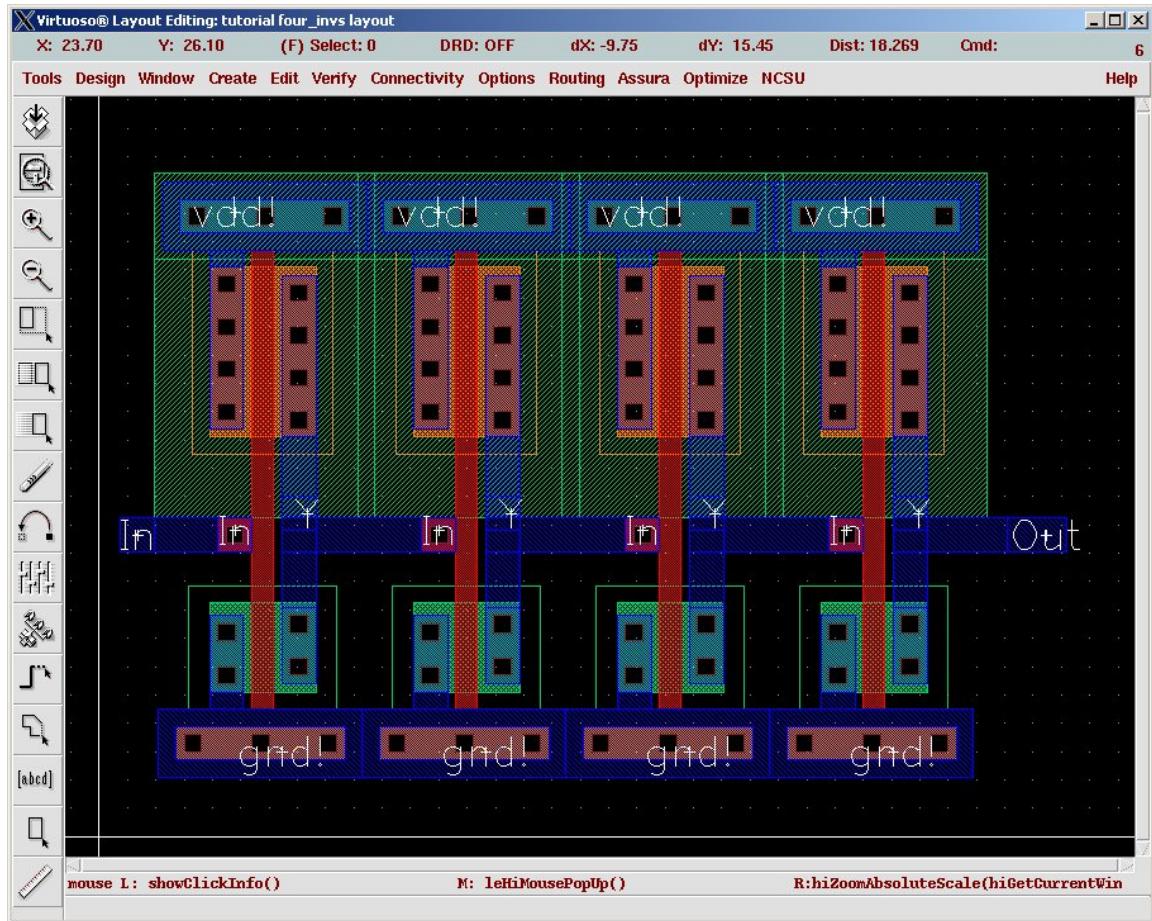


Figure 5.18: Layout with four inverter instances expanded to see all levels of layout

“movement arrows” you can move the selected shapes using the left button with click and drag. Reshapping can be done with the **Stretch** (hotkey **S**) command.

**Changing the View Hierarchy:** You can choose how many levels of hierarchy to view using the **Options** → **Display** menu choice and change the **Display Levels** numbers. You can also use the **ctrl-f** hotkey to hide all but the top level, and **F** (shift-f) hotkey to expand the display to all levels of the hierarchy. Note that the **f** hotkey fills and centers on the screen with the current cell.

**Viewing Connectivity:** You can use the **Connectivity** → **Mark Net** menu to mark an entire connected net. This will highlight the entire connected net through all conducting layers. Note that it does not *select* that net, it simply marks it so you can see what’s connected to what.

**Hierarchical Instantiation:** You can include instances of other layout cells in your current layout with the **Create** → **Instance** menu, the **i** hotkey, or the **instance** widget. These instances will show up as blank boxes with a red outline if you are not viewing deep enough in the hierarchy, and as filled in layouts if you have expanded the viewing hierarchy.

## 5.3 Printing Layouts

In order to print your layout view you can use the **Design** → **Plot** → **Submit...** menu. This will submit your design to be plotted (printed) either on a printer or to a file. The **Submit Plot** dialog box is shown in Figure 5.19. This Figure shows the dialog box after the plotter has been configured to send the output to an EPS (Encapsulated PostScript) file on A-sized paper to a file called **foo.ps**. These choices are set up in the **Plot Options...** dialog box as seen in Figure 5.20. In this dialog you can select the plotter (printer) that you would like to send the graphics to. You can either select a printer name (like the CADE printer), or save the file as EPS. You can choose to center the plot, fit it to the page, or scale the plot to whatever size you like. You can also choose to queue the plot for a later time, change the name of the file (if you’re plotting to a file), and choose to have email sent when the plotting completes (a somewhat silly feature, unless you’re really queuing the plot for sometime later). The EPS option will print to a color postscript file so that if you have access to a color printer you can print in color. I like to un-select the **Plot With Header** option so that I don’t get an extra header page included with the plot.

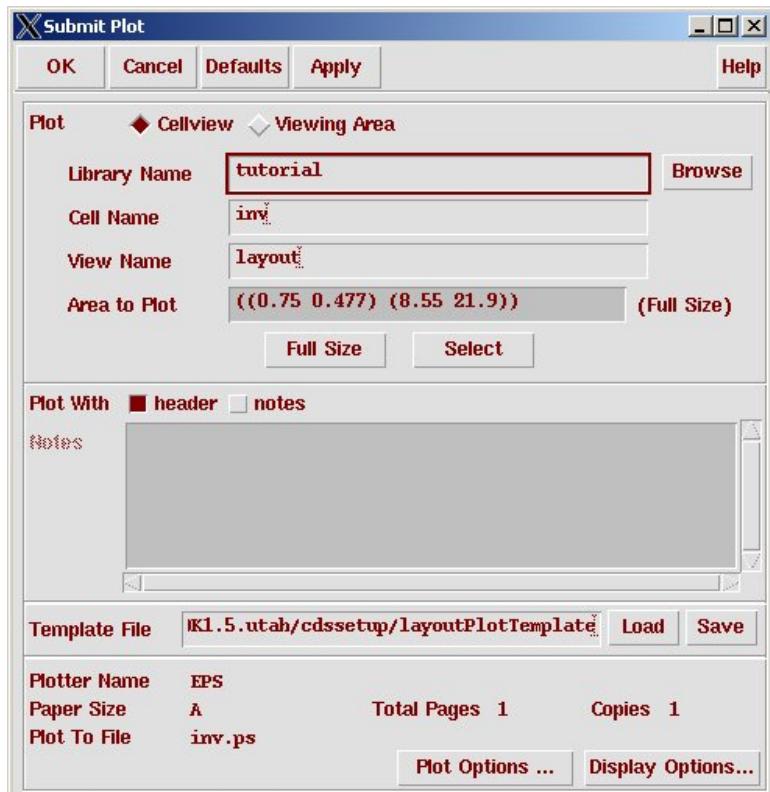
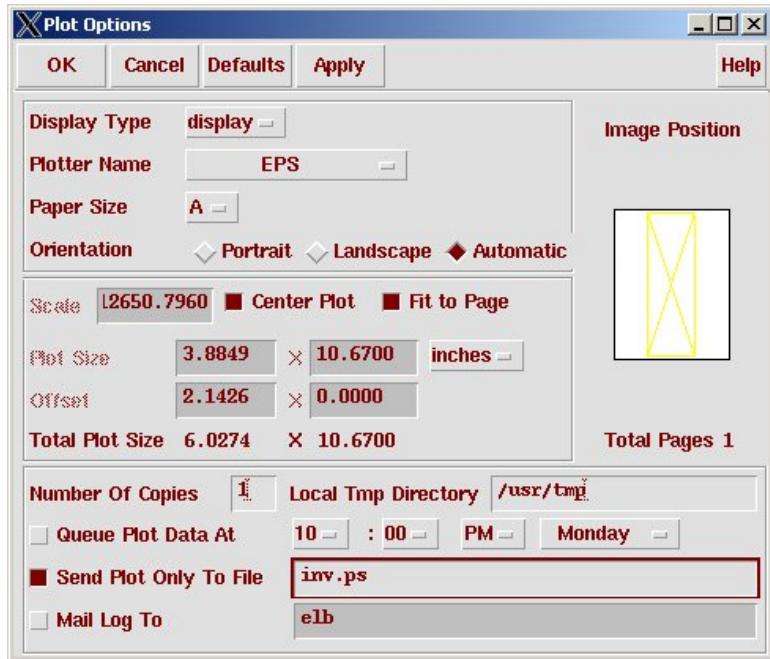


Figure 5.19: Submit Plot dialog box

Figure 5.20: **Plot Options** dialog box

## 5.4 Design Rule Checking

Because there are so many rules about how you can draw shapes in the different mask layers, it's very hard to tell just by looking if you have obeyed all the design rules (The MOSIS SCMOS rev8 design rules are shown in Appendix D). Letting the CAD tool check the design rules for you is known as *Design Rule Checking* or DRC. When you're drawing new layout using Virtuoso it's helpful to run the DRC check frequently as you're doing your layout. There are three DRC tools that we have access to in this CAD flow, but only DIVA is currently fully supported for our class technology. The tools are:

**DIVA:** This is the DRC engine that is integrated with Virtuoso and supported by our current class technology files. It will be described in detail in the next section.

**Assura:** This is a faster and more capable DRC engine that is also integrated with Virtuoso, but not yet fully supported by our technology information. It uses a different DRC rules file that is currently under development.

**Calibre:** This is the “industry standard” DRC engine from Mentor. It re-

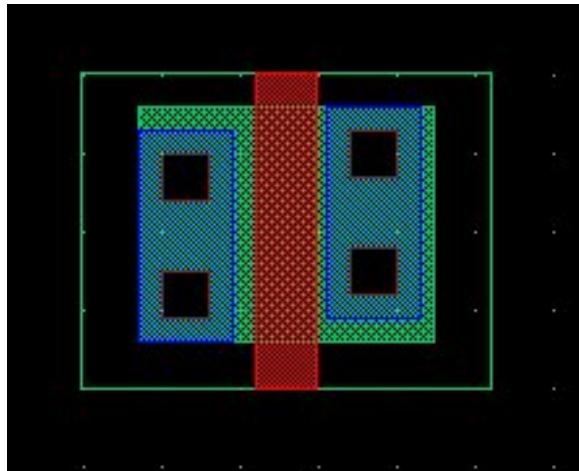


Figure 5.21: NMOS transistor layout (with DRC errors)

quires yet a different DRC rules file so it's not supported by our class flow at the time being.

#### 5.4.1 DIVA Design Rule Checking

If you have a **Virtuoso** layout window open you can check that layout for design rule violations using **DIVA** from that layout window. Consider a piece of layout as shown in Figure 5.21. This is a single nmos transistor similar, but not exactly the same as, Figure 5.8.

To run **DIVA** DRC choose the **Verify → DRC...** menu choice. A new window pops up as shown in Figure 5.22. Notice that the **Rules file** field is already filled in with **divaDRC.rul**. This is the rules file in the **UofU\_TechLib.ami06** library that checks for MOSIS SCOMS Rev8 design rules. Click **OK** to start the design rule check. The results of the DRC will be in the main **CIW** window, and if there are errors those errors will be highlighted in the layout with flashing white shapes.

In this case there are 4 total errors flagged by the DRC process. These rules are described in the **CIW** where the **SCMOS Rule** number tells you which of the rules from Appendix D has been violated. The violation geometry is also highlighted in the layout as shown in Figure 5.24. From these two sources we can see that the top of the **poly** gate needs to extend further over the **nactive**, the **nselect** also needs to be extended at the top of the transistor, and the **cc** contact layer is too close to the transistor gate.

In a larger layout it may be hard to spot all the error highlights. In this case you can use tools in **Virtuoso** to find the errors for you and change

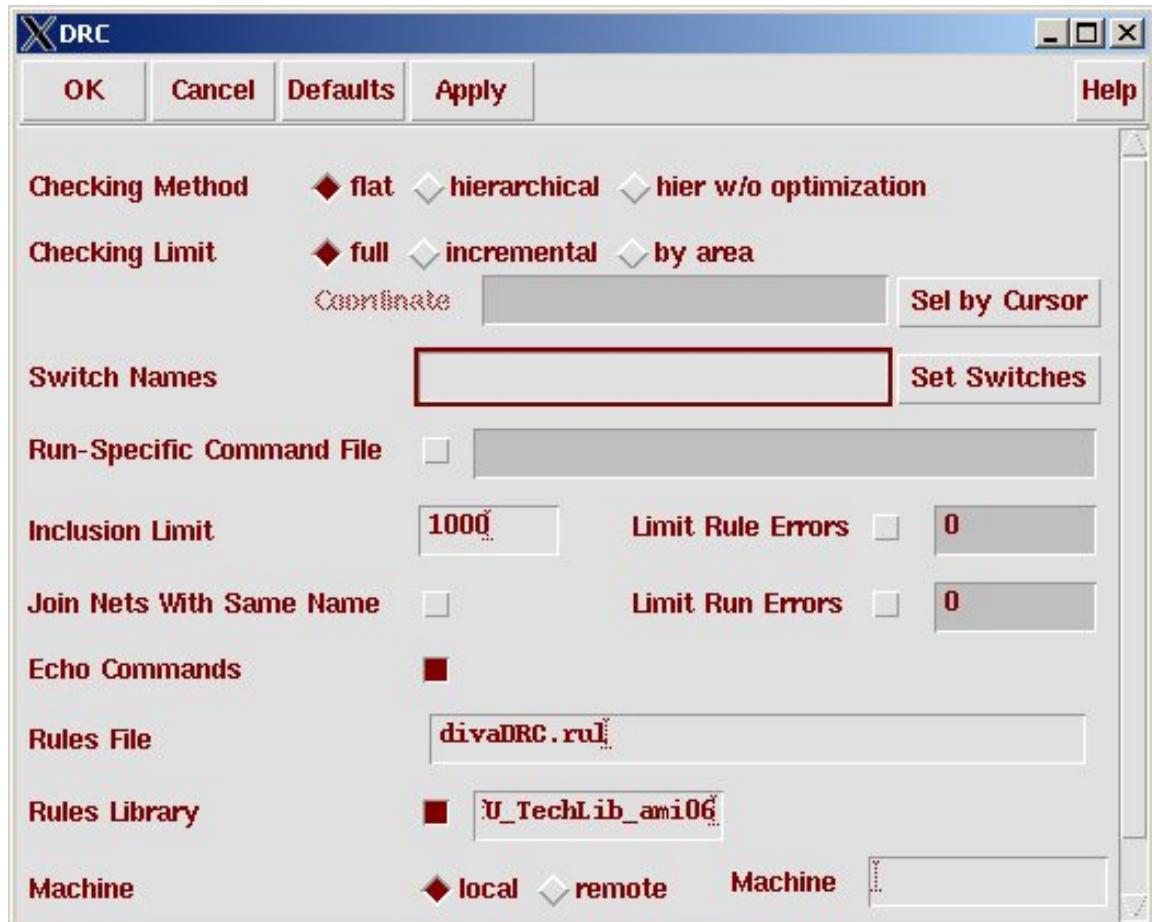


Figure 5.22: DIVA DRC control window

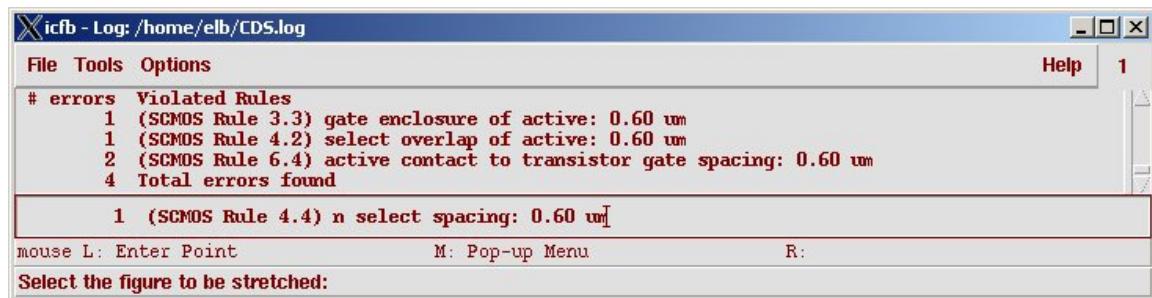


Figure 5.23: Results from the DRC in the CIW window

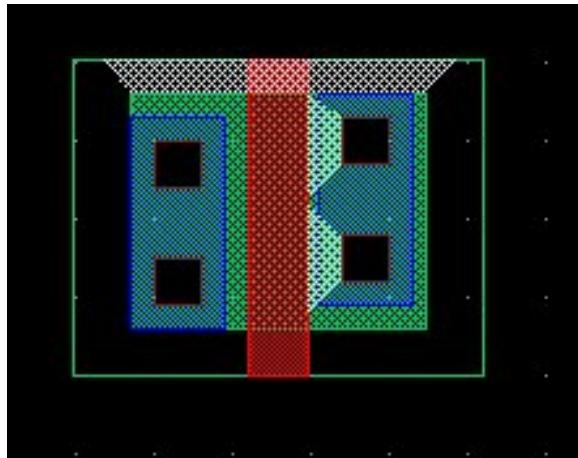


Figure 5.24: NMOS transistor layout (with DRC errors flagged)

the layout view so that you can see those errors. If you select **Verify** → **Markers** → **Explain** you can click on an individual error marker and have it explained. If you do this to the white marker at the top of the example in Figure 5.24 you get the explanation as shown in Figure 5.25. If you would like to step through all the errors in the current layout view, you can select **Verify** → **Markers** → **Find...** which pops up the window seen in Figure 5.26. Selecting the **Zoom To Markers** option and then clicking the **Next** button will zoom to each DRC violation in turn and put the explanation in a window like that in Figure 5.25. It's a great way to walk through and fix DRC violations one by one in a larger layout. When you are finished and would like to get rid of all the white DRC markers you can use **Verify** → **Markers** → **Delete** or **Verify** → **Markers** → **Delete All...** to erase them from the screen.

When you have fixed all the violations and re-run the DIVA DRC process you should see **Total errors found: 0** in the CIW. Although the previous example of the inverter was completed using the rulers, in practice I would have used the DRC checker after each of the major steps to make sure that things were legal as I was drawing the layout.

#### 5.4.2 Assura Design Rule Checking

*This is to be added. The Assura tool can read a DIVA DRC file, but it needs some modification. This has been done, but not documented yet (10/2006).*

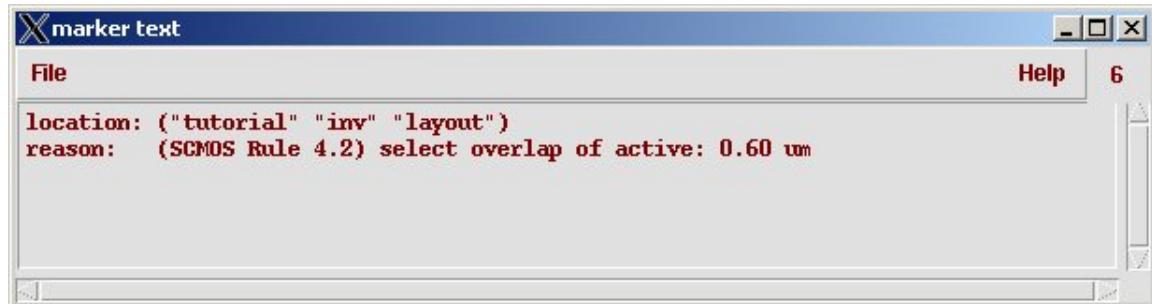


Figure 5.25: Explaination of DRC violation

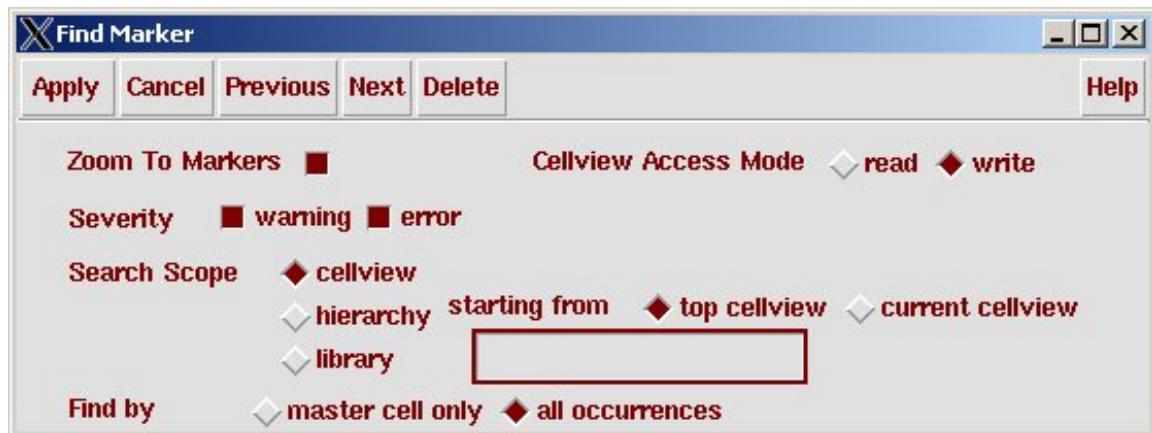


Figure 5.26: Finding all DRC violations

## 5.5 Generating an Extracted View

Another view that is essential to the design process is an *extracted* view of a cell. An extracted view is one that takes the layout view of the cell and extracts the transistor schematic from that layout. The extraction tool does this by knowing the electrical properties of the layers and the overlap of the layers. For example, one of the extraction rules is that when a **polysilicon** rectangle overlaps a rectangle of **nactive**, the area of the overlap is an nmos transistor. The connection between the transistors is extracted from the conducting layers and the connection between those layers.

To run the extraction process using DIVA start by opening the **layout** view of a cell (for example, the inverter from Figure 5.16). From the layout window choose the **Verify → Extract...** menu choice. A new window pops up as shown in Figure 5.27. Notice that the **Rules file** field is already filled in with **divaEXT.rul**. This is the rules file in the **UofU\_TechLib\_ami06** library that extracts the transistors using the technology defined according to the MOSIS SCMOS Rev8 design rules. Another thing to consider about extraction is whether you want to extract with or without parasitic capacitances. Usually you want to include parasitics because it will give a more accurate (although slightly slower) analog simulation of the cell later on. To include the parasitic capacitors use the **Set Switches** button in the extraction dialog box. This will bring up another box (Figure 5.28) that has a number of switches. The most important is **Extract\_Parasitic\_Caps**. If you select this and click **OK** you will see this switch show up in the main extraction dialog box.

Once things are set the way you want them, click **OK** to start the extraction. The results will show up in the **icfb** Command Interpreter Window (CIW). Note that you should run the extractor only after passing the DRC phase. A correct extraction should show 0 errors as shown in Figure 5.29.

*All layers in the virtuoso layout editor are actually “layer/purpose pairs.” The “purpose” of the layers is **drawing**. The layers in the extracted view are “net” purpose and have outline views to indicate this different purpose.*

After extraction you will see that for your cell (the **inv** in this example) will now have a new view named **extracted**. You can open this view in **icfb** and look at the result. For the inverter you should see outline views of the mask layers and transistor symbols for each of the extracted transistors. The extracted transistors will be annotated with their width and length as defined by the rectangles that are drawn. See Figure 5.30 for an example. The extracted view is used to compare the layout with a schematic to make sure that the layout accurately captures the desired schematic. This is a critical step in the design process!

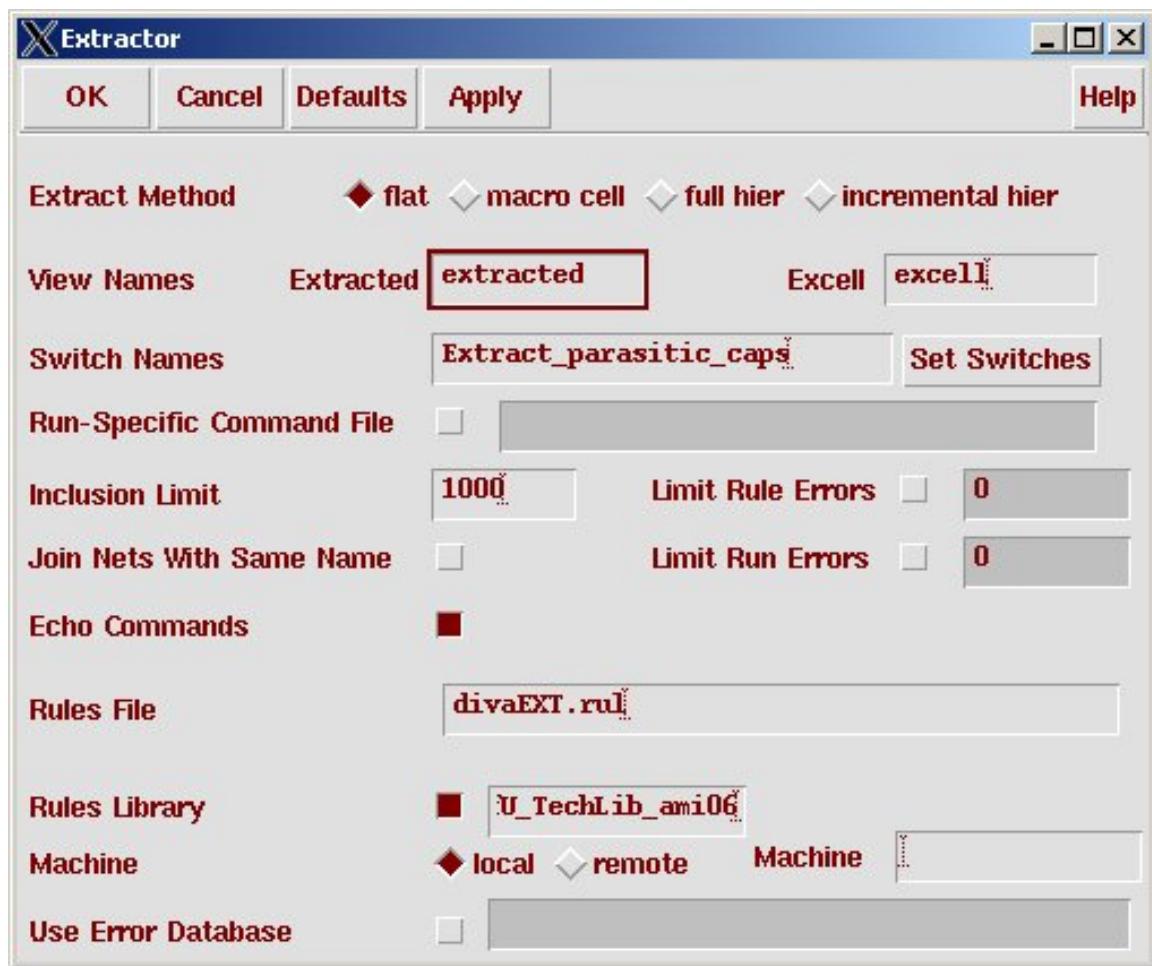


Figure 5.27: DIVA extraction control window

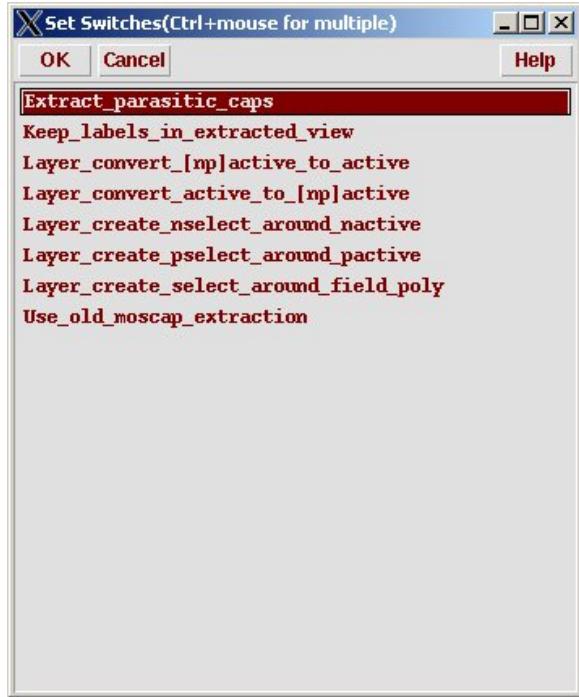


Figure 5.28: DIVA extraction special switches

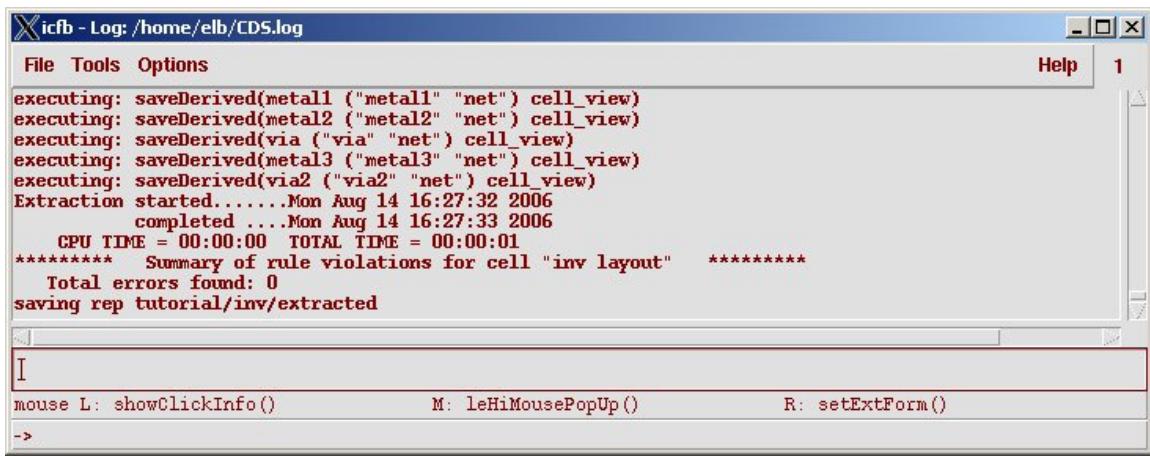


Figure 5.29: DIVA extraction result in the CIW

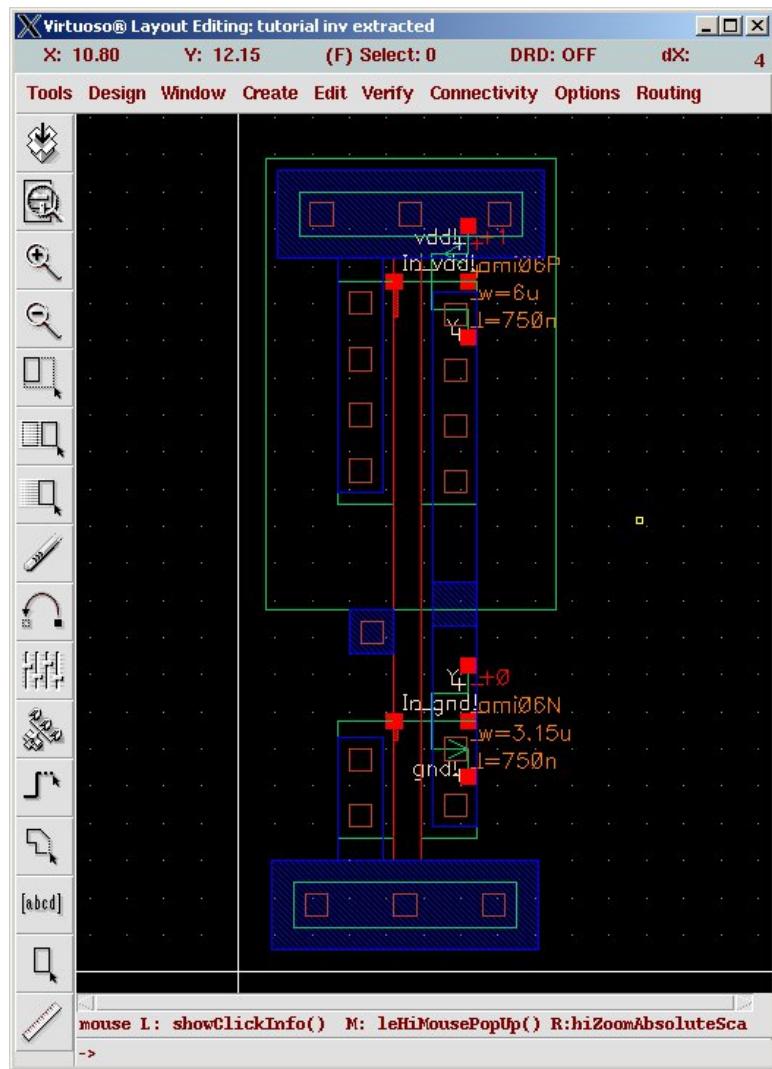


Figure 5.30: Extracted view of the inverter

## 5.6 Layout Versus Schematic Checking (LVS)

The process of verifying that an extracted layout captures the same transistor netlist as a schematic is known as *Layout versus Schematic* or (LVS) checking. In order to do this check you need both a **schematic** view and an **extracted** view of the cell. Note that this LVS process will not tell you if your schematic is correct. It will only tell you if the schematic and layout views describe the same circuit!

To run the LVS process using DIVA start by opening the **layout** view of a cell (for example, the inverter from Figure 5.16). From the layout window choose the **Verify → LVS...** menu choice. A new window pops up as shown in Figure 5.31. Notice that the **Rules file** field is already filled in with **divaLVS.rul**. This is the rules file in the **UofU\_TechLib\_ami06** library that compares the transistor netlist defined by the schematic (extracted from the schematic using the **netlister**) with the netlist defined by the **extracted** view. When this window pops up you may see a window as shown in Figure 5.32. This window is asking if you want to use old information for the LVS, or re-do the LVS using the information in the LVS form. You almost always want to select the **use form contents** option here before moving to the LVS window.

Once you get to the LVS window, you need to fill in the **Library**, **Cell**, and **View** for each of the **schematic** and **extracted** sections in the window. You can either type these in by hand, or use the **Browse** button to pop up a little version of the **Library Manager** where you can select the cells and views by clicking. Note that the LVS window in Figure 5.31 has the **Rewiring** and **Terminals** buttons selected. **Rewiring** means that the netlists can be changed so that the process will continue after the first error. **Terminals** means that the terminal labels in both the schematic and extracted views will be used as starting points for the matching. To have this be useful the terminals must be named identically in the layout and schematic.

You can also change some of the LVS options in the **NCSU → Modify LVS Rules ...** menu choice in the layout editor. This window, shown in Figure 5.33, sets various switches that modify how the LVS rules file interprets the two netlists. The default switches are shown in the figure and the names should be reasonably self-explanatory. Mostly they involve issues of whether structures in the netlist should be combined such that their effect is the same regardless of their physical connection. For example, the **Allow FET series Permutation** switch controls whether two netlists should be considered the same if series FETs are in a different order in the schematic and the layout. Often you don't care about the order in a series connection. If you do, unselect this switch before running LVS. **Combine Parallel FETs** controls whether two parallel FETs should be considered the same as

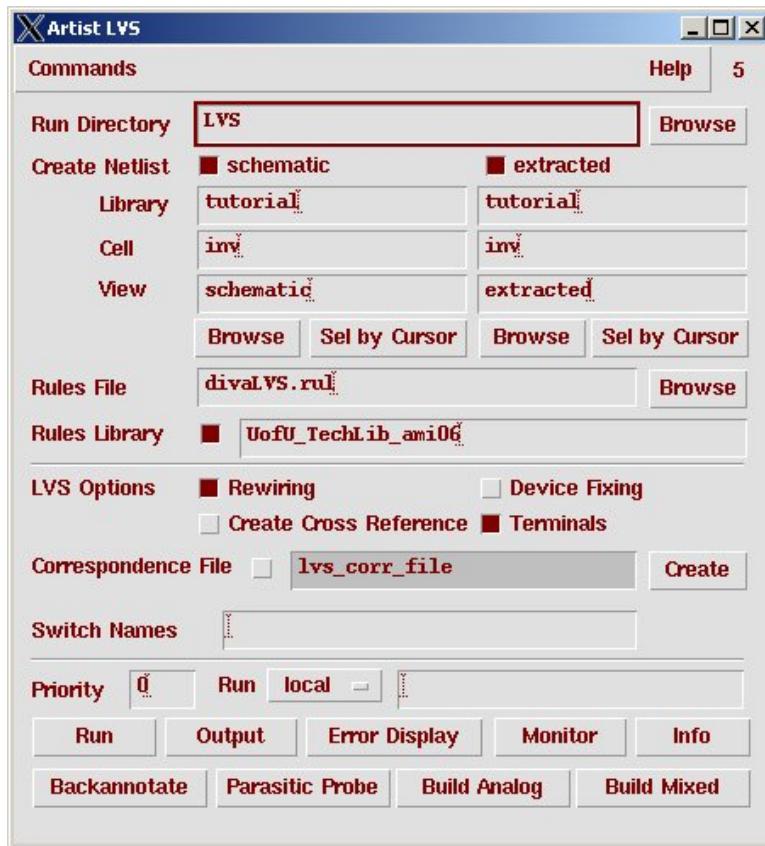


Figure 5.31: DIVA LVS control window

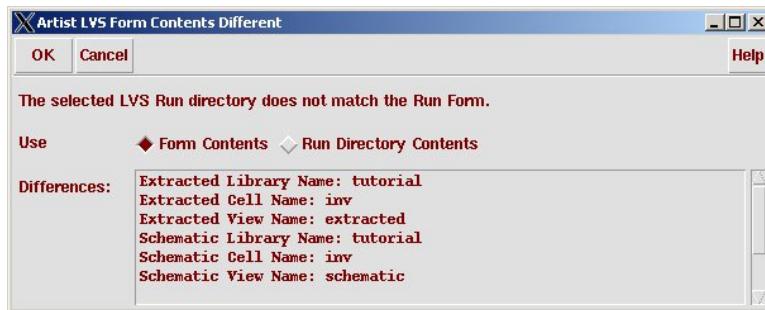


Figure 5.32: DIVA LVS Control Form

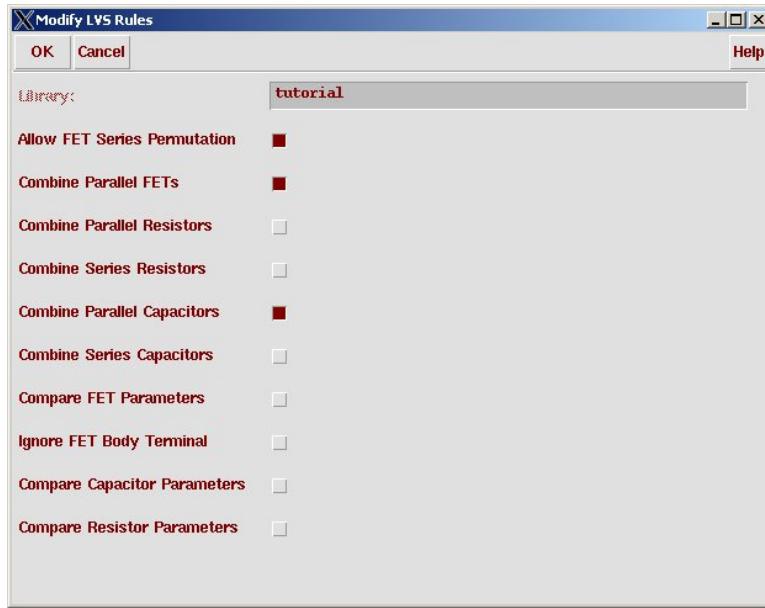


Figure 5.33: NCSU form to modify LVS rules

one FET. The logical effect is the same, so usually this is allowed. However, by default the LVS process does not check for matching FET parameters between the schematic and layout. If you would like this extra level of checking, you can select the **Compare FET Parameters** switch and the result is that two FETs in the same position in the netlist will not match if their width and length parameters do not match.

Once the window is filled in as in Figure 5.31 you can start the LVS process using the **Run** button. This will sometimes ask if you can save the schematic or extracted view before continuing. You almost always want to agree since you want to LVS against the most current views. Clicking on the **Run** button will have little visible effect. Look in the CIW and you'll see **LVS job is now started...** to indicate that things are running. If you think about LVS in general algorithmic terms, it's really performing exact graph matching on two circuit netlist graphs to see if the graphs are the same. This is a very hard problem in general (many graph matching problems are NP-complete, although it's not known if exact graph matching falls into this category). Hints such as using named terminals can make the problem much easier, but for large circuit this can still take quite a bit of time.

When the process finishes you will see a window like that in Figure 5.34. In this case the report is that the process **has succeeded**. If you get this result that's a great start, but all it means is that the LVS algorithm has finished. It does *not* tell you whether the graphs have matched (i.e. if the schematic



Figure 5.34: DIVA LVS completion indication

and extracted view describe the same circuit). For that you need to click the **Output** button in the LVS window. In this case, the Output window (Figure 5.35 tells us that although the numbers of **nets**, **terminals**, **pmos** and **nmos** transistors are the same in the two views, that the netlists failed to match. Scrolling down in this window (Figure 5.36) you can see that the problem is with the naming of the output terminal. The output terminal in this example was named **Y** in the layout (extracted) view and **Out** in the schematic view. This can be corrected by changing the name of one of the terminals and re-running LVS. After doing this the output of LVS shows that the netlists do indeed match.

*Remember that if you change anything in the layout view you need to re-run the extraction process to generate a new **extracted** view before re-running LVS.*

If instead of the **has succeeded** message after running LVS you get a **has failed** message you need to figure out what caused the LVS process to not run to completion. To see the process error messages to help figure this out select the **Info** button in the LVS control window (Figure 5.31), and in the **Display Run Information** window that pops up (Figure 5.37) select the **Log File** button to see why the LVS process failed. Usually this is because one of the two netlists could not be generated. The most common problems are either you didn't have the correct permissions for the files, or you may have forgotten to generate the extracted view altogether. You can also use the **Display Run Information** window to help track down issues where LVS reports that the netlists fail to match.

Tracking down LVS issues can be quite frustrating, but it must be done. If LVS says that the schematic and layout don't match, you must figure out what's going on. Experience shows that even though you can see no issues with your layout, that the LVS process is almost never wrong. Here are some thoughts about debugging LVS issues:

- - If LVS fails to run, then you have a problem right up front. Usually it's because you have specified the files incorrectly in the LVS dialog box. You can see what caused the problem by clicking on the **Info** button in the LVS dialog box, and then looking at the **RunInfoLogFile**.

```

X/home/elb/IC_CAD/cadencetest/LVS/si.out
File Help 6

e(#)$CDS: LVS.exe version 5.1.0 07/23/2006 23:38 (cicln01) $

Command line: /uusoc/facility/cad_tools/Cadence/IC5141ISR200607280110/tools/dfII/bin/
Like matching is enabled.
Net swapping is enabled.
Using terminal names as correspondence points.
Compiling Diva LVS rules...

Net-list summary for /home/elb/IC_CAD/cadencetest/LVS/layout/netlist
count
 4      nets
 4      terminals
 1      pmos
 1      nmos

Net-list summary for /home/elb/IC_CAD/cadencetest/LVS/schematic/netlist
count
 4      nets
 4      terminals
 1      pmos
 1      nmos

Terminal correspondence points
N2      N2      In
N3      N1      gnd!
N1      N0      vdd!

Devices in the rules but not in the netlist:
  cap nfet pfet nmos4 pmos4

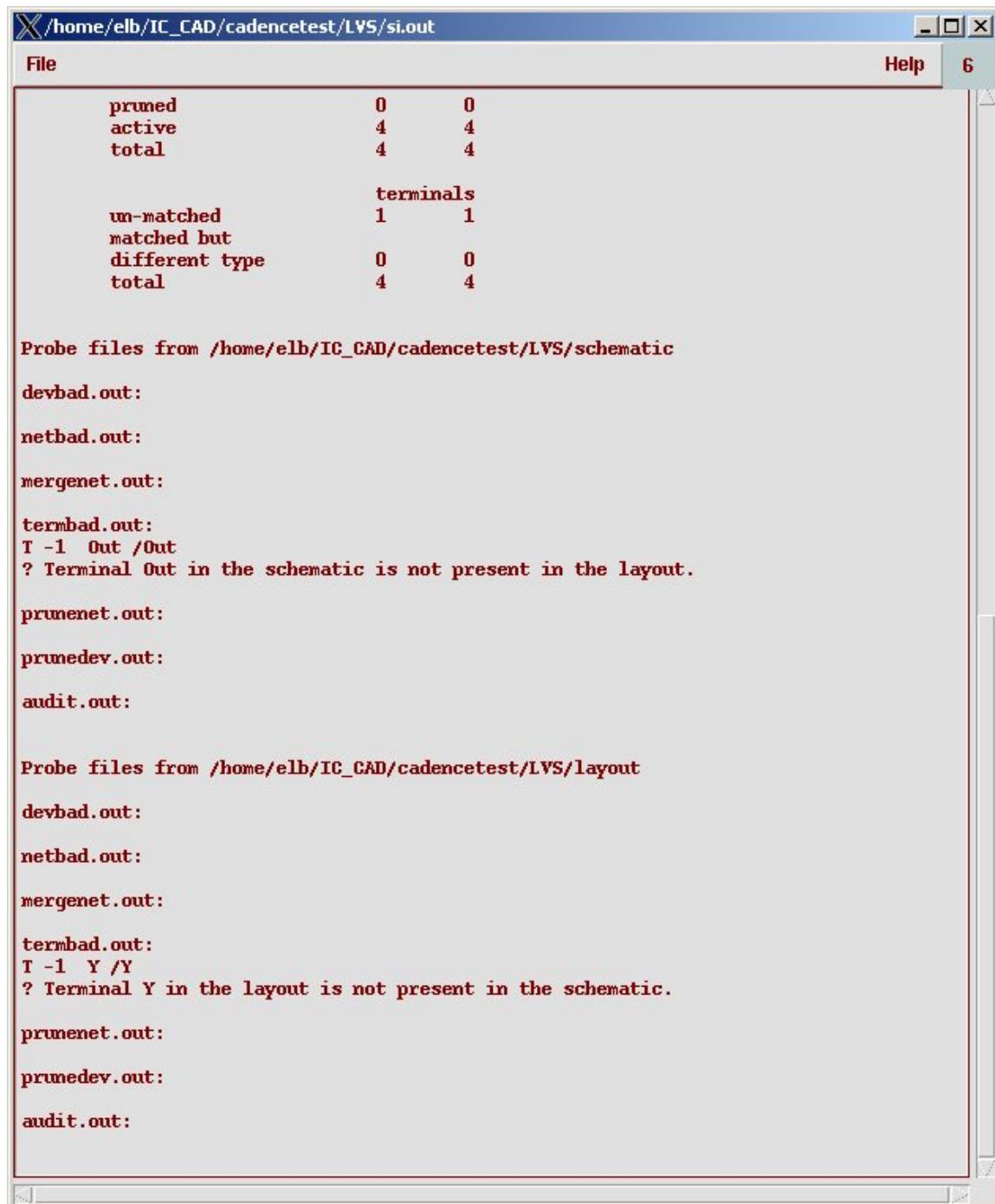
The net-lists failed to match.

          layout   schematic
                      instances
un-matched          0      0
rewired             0      0
size errors         0      0
pruned              0      0
active              2      2
total               2      2

          nets
un-matched          0      0
merged              0      0
pruned              0      0
active              4      4
total               4      4

```

Figure 5.35: DIVA LVS ouptut



The screenshot shows a terminal window titled '/home/elb/IC\_CAD/cadencetest/LVS/si.out'. The window contains the following text:

```
File Help 6

pruned      0      0
active       4      4
total        4      4

terminals
un-matched  1      1
matched but
different type  0      0
total        4      4

Probe files from /home/elb/IC_CAD/cadencetest/LVS/schematic
devbad.out:
netbad.out:
mergenet.out:
termbad.out:
T -1 Out /Out
? Terminal Out in the schematic is not present in the layout.

prunenet.out:
prunedev.out:
audit.out:

Probe files from /home/elb/IC_CAD/cadencetest/LVS/layout
devbad.out:
netbad.out:
mergenet.out:
termbad.out:
T -1 Y /Y
? Terminal Y in the layout is not present in the schematic.

prunenet.out:
prunedev.out:
audit.out:
```

Figure 5.36: DIVA LVS output (scrolled)

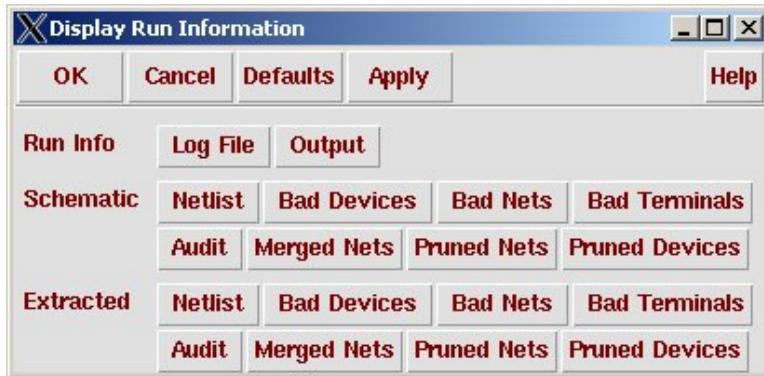


Figure 5.37: DIVA LVS Run Information Window

- If the LVS is successful, but the netlists fail to match, now you have a detective problem of tracking down the cause of the mismatch. Now you need to look at the Output Report to see what's going on.
- The first thing to look at is the output report and check the netlist summary for both **schematic** and **layout (extracted)** views. If the number of nets is different, that can give you a place to start looking. Let's assume that you've already simulated the schematic using a **Verilog** simulator so you believe that's working. (This should definitely be true!).
  - If you have fewer nets in the layout than you do in the schematic, suspect that there are things connected together in the layout that weren't supposed to be. This is easy to do by overlapping some metal that you didn't intend to overlap (for example).
  - If there are more nets in the layout than in the schematic, then there are nodes that were supposed to be connected, but weren't. This too is easy to do, by leaving out a contact or via, for example.
  - Now you go into detective mode to figure out what's different. For small cells you just poke around in the layout with **Connectivity** → **Mark Net** to see the connected layout and see if there's anything connected that shouldn't be.
  - You can also open the extracted layout and look at things there. The advantage of this is that the extracted layout has the netlist already constructed so when you select something you're selecting the whole connected layout net by default. You can also use **[q]** with the selected net to see things like net number of the selected net.

- You can also use the error reporting mechanism in LVS to see which nets are different. After running LVS, open a schematic window and an extracted view window for the cell in question. Now click on the **Error Display** button in the LVS dialog box. This will let you walk through the errors in the LVS output file one at a time and will highlight in the schematic or the extracted view which node that error is talking about. This can be very helpful if you have only a few errors in the output file, but can also be more confusing than helpful if you have a lot of errors. This is because once you have one error, it can cause lots of things to look like errors. This can, however, be a good way to see that the output file is talking about, and you might walk through the errors and see a big connected net in the layout that you thought should be two separate nodes (for example).
- If you'd like to look at the extracted view and search for a particular net, you can do it through a slightly non-obvious procedure. Of course, you can use the **Error Display** to step through the errors until you get to the one with the net you're interested in. Or you can open the extracted view, select **Verify** → **Shorts**. Dismiss the informational dialog box to get the **Shorts Locator** box. Type the net name or number in the **Net Name** box and click on **Run**. This will highlight that node in orange. Click **Finish** to start over with another net. Clearly this dialog box does something more interesting than just highlight a net. It has something to do with tracking down shorts between nets, but I have to admit I don't know exactly what it does. I do know that it highlights nets by name in extracted views!
- Remember that a single LVS error can cause lots of things to look like errors. Once you find one error, rerun LVS and see what's changed. A single fix can cause lots of reported violations to go away.
- If all else fails and you're convinced that things are correct, try removing the `~/IC_CAD/cadence/LVS` directory and re-running LVS. It's possible (but rare) that you mess LVS up so badly that you need to delete all its generated files and start over.

If your cell passes LVS then it's a pretty good indication that your layout is a correctly functioning circuit that matches the behavior of the transistor schematic.

### 5.6.1 Generating an analog-extracted view

The LVS process compares the **schematic** view with the **extracted** view to see if the netlists are the same. Once LVS says that the netlists are the same,

there is one further step that you can take. Click on the **Build Analog** button in the **LVS** control window (Figure 5.31). Make sure to **Include All** the **extracted paracitics** so that if you extracted parasitic capacitors they will be included, and click **OK**. This will generate yet another **Cell View** called **analog-extracted**. This view is very similar to the **extracted** view, but has additional power supply information so that the view can be simulated with the analog simulator Spectre. This will be seen in Chapter 6.

## 5.7 Overall Cell Design Flow (so far...)

The complete cell design flow involves each of the steps defined previously. They are:

1. Start with a schematic of your desired circuit. The schematic can be made using components from the **UofU\_Digital\_v1\_1** library and the transistors from the **NCSU\_Analog\_Parts** and **UofU\_Analog\_Parts** libraries. Use **Composer** as described in Chapter 3.
2. Simulate that schematic using one of the Verilog simulators defined in Chapter 4 to make sure the functionality is what you want. Make sure to use a self-checking testbench.
3. Now draw a layout of the cell using the **Virtuoso** as described in this Chapter.
4. Make sure that the **layout** view passes DRC.
5. Generate an **extracted** view.
6. Make sure that the **schematic** and **extracted** view match by using the **LVS** checker.
7. Generate the **analog-extracted** cell view for later analog simulation.

## 5.8 Standard Cell Template

Designing a standard cell library involves creating a set of cells (gates, flip flops, etc.) that work together throughout the CAD tool flow. In general this means that the cells should have all the views necessary for designing with the cells, and should be compatible in terms of their attributes so that they can all work together. At this point in our understanding of the CAD flow this means that each cell in the cell library needs the following views:

**schematic:** This view defines the gate or transistor level definition of the cell. It can be simulated using any of the Verilog simulators in Chapter 4 as a switch-level or a behavioral-level simulation.

**behavioral:** This view is the Verilog description of the cell. It should include both behavior and **specify** blocks for timing so that the timing can be back-annotated with better estimates as the design progresses through the flow.

**layout:** This view describes the mask layout of the cell. It should pass all DRC checks. It should also follow strict physical and geometrical standards so that the cells will fit with each other and work together (the subject of this Section).

**extracted:** This view extracts the circuit netlist from the layout and is generated by the extraction process. It should be used with the LVS checker to verify that the **layout** and **schematic** views represent the same circuit.

**analog-extracted:** This view is an augmented version of the **extracted** view that includes information so it can be used by the analog simulator Spectre. It is generated from the extracted view through the LVS process.

In addition to these views, subsequent Chapters in this text will introduce a number of additional views that are required for the final, complete cell library. They include:

**abstract:** This is a view that is derived from the layout. It tells the place and route tool where the ports of the cell are, and where the “keep-outs” or obstructions of the cell are that it should not try to route over. Generating this view from the layout is described in Chapter 9.

**LEF:** This is a Library Exchange Format file that is derived from the abstract view of the cell and is read by the place and route tool (SOC Encounter) so that it can get information about the technology that it is routing in, and about the abstract views of the cells in the library. This view is also generated by the Abstract program.

**Verilog Interface:** The system you design will eventually be input to the place and route tool as a structural verilog file that describes the standard cell gates used in your design and the connections between them. In addition to this file you need a simple I/O interface of each cell (separate from the LEF file) so that the place and route tool can parse the structural verilog file. This will also be described in Chapter 9.

**Lib:** This is a standard format, called **Liberty** format (usually with a **.lib** filename extension), for describing cells so that the synthesis tools can understand the cells. The file describes the I/O interface, the logic function, the capacitance on the cell pins, and extracted timing for the cell. This file can be generated by hand, using analog simulation using **Spectre** or **Spice**, or using the **SignalStorm** tool from **Cadence** and will be described in Chapter 7.

### 5.8.1 Standard Cell Geometry Specification

In standard cell based design flow, the automatic place and route tool (**SOC Encounter** in this flow) places the standard cells selected by the synthesis engine (**Synopsys design compiler** or **Cadence BuildGates**) in a set of rows. To make this packing as efficient as possible we can plan our cells so that they can be placed adjacent to each other. With some simple rules about how the internal layout of the cells is designed, we can guarantee that any two cells can be placed next to each other without causing any DRC violations. Some overall design considerations are:

- Since the cells are going to be placed in rows adjacent to each other we will make all our cells the same height so that they fit nicely into rows.
- All our CMOS based standard cells have a set of **pmos** transistors in an **nwell** for the pull-up network and **nmos** transistors for the pull-down network. We will always place the **nwell** with **pmos** transistors on top of the layout and **nmos** transistors at the bottom. By doing this and having the same height in the **nwell** layer for all cells, we can have a continuous well throughout the row of cells.
- All standard cells have a power and ground bus. To avoid shorting of power and ground while placing cells adjacent to each other, we place the **vdd!** bus on top and **gnd!** bus at the bottom of the cell (which makes sense because we have the pull up network on top and pull down at the bottom). Having equal heights for these buses for all cells and running these buses throughout the width of the cells allow for automatic connection of adjacent cell power and ground buses by abutment.
- All the standard cell layouts will have a fixed origin in the lower left corner of the layout, but pay attention, it is not the absolute lower left of the layout!

Hence we define the following standard cell dimensions for class cells that will be implemented in the AMI C5N  $0.5\mu$  CMOS process.

**Cell height = 27 microns.** This height is measured from the center of the **gnd!** bus to the center of the **vdd!** bus.

**Cell width = multiple of 2.4 microns.** This matches the vertical **metal2** routing pitch and also the pitch of the **NTAP** and **SUBTAP** well and substrate contacts.

**nwell height from top of cell = 15.9 microns.** This is 2/3 of the cell height because **pmos** transistors are usually made wider than **nmos** transistors.

**vdd! and gnd! bus height = 2.4 microns.** This height is centered on the supply lines so that 1.2 microns is above the cell boundary and 1.2 microns is below the cell boundary. This way if the cells are abutted vertically with the supply lines overlapping those supply lines remain 2.4 microns wide.

**vdd! and gnd! bus width = 1.2 microns overhang beyond the cell boundary.** This is for convenience so that the **metal1** layer of the well and substrate contacts can overlap the cells safely.

Based on the **nwell** size and placement, and the MOSIS SCOMS rules, we can now standardize the **nselect** and **pselect** sizes as well. The closest that select edges can be to the well edges is 1.2 microns (SCOMS rules 2.3 and 4.2). So, the select restrictions are:

**pselect height from inner edge of vdd! bus = 13.5 microns**

**nselect height from inner edge of gnd! bus = 8.7 microns**

**nselect and pselect width = cell width**

**nwell overhang outside cell width = 1.5 microns** (to top, left and right of the cell)

To avoid *latch-up* in the circuit we place substrate and well contacts to make sure that the **nwell** is tied to vdd and the substrate is tied to gnd. As seen in Section 5.2, we use predefined contacts for these connections in the form of **NTAP** and **SUBTAP** contacts. Each of these contacts is 2.4 microns wide so you can think of the overall cell width as a multiple of 2.4 microns, or as an integer number of **NTAP** and **SUBTAP** cells. These contacts are placed centered on the **vdd!** and **gnd!** lines. Note that placing this many well and substrate contacts is overkill, but does not have negative effects.

*The usual rule of thumb for the minimum number of well contacts is to place one well contact for every square of well material.*

It's a good idea to build a standard cell layout *template* incorporating these basic design specifications. This can serve as a starting point for the

design of new cells. It's also a good idea to keep your library cells in a separate Cadence library from your designs. This keeps the distinction clear between the library cells and the designs that use those cells. You could, for example, make a template with the **gnd!** bus on the bottom, including the **SUBTAP** contacts, the **vdd!** bus on the top, including the **NTAP** contacts, and the **nwell**, **nselect**, and **pselect** layers as defined earlier. An example of such a template with rulers showing the dimensions, and example **pmos** and **nmos** transistors is shown in Figure 5.38. This cell, even without actual connections between the transistors, should pass DRC with no violations.

*The cell boundary will be generated as a rectangle of type **prBound** by the **Abstract** program. This layer is not editable by default, but can be enabled if needed through the **LSW** window.*

The notion of a *cell boundary* was used without specific definition in the preceding text. The cell boundary is the rectangle that defines the outline of the cell from the place and route tool's point of view. We have defined our library so that the cell boundary is actually inside of the cell layout geometry so that when the cells are abutted with respect to the cell boundary, they actually overlap by a small amount. The cell boundary is shown in Figure 5.39, but you don't have to draw it by hand. It will be derived during the process of extracting an **abstract** view as described in Chapter 9.

*In order to make sure that there will not be any DRC violations when cells are abutted, it is very important that no geometry on any layers (other than the layers already in the template) falls within 0.6 microns from the left and right of the cell boundary or within the **vdd!** or **gnd!** buses. This rule is in addition to other SC莫斯 Design Rules within the cell and applies even if the DRC of the individual standard cell does not complain. For example, in the layout shown in Figure 5.39, the **nactive** and **pactive** layers are placed  $0.6\mu$  away from the cell boundaries on left and right side. They cannot come any closer to the boundary. Also, in the **layout** view shown, the **pmos** and **nmos** transistor gates (**poly** layer) are placed just touching the **vdd!** and **gnd!** bus inner edges. The gates could go farther away from the bus inner edges but cannot extend into the bus structure. If you need more space than these rules allow, you will have to make your whole cell wider. Each time you make a cell wider, it has to be made wider in increments of  $2.4\mu$ .*

### 5.8.2 Standard Cell I/O Pin Placement

The design rules for a specific VLSI process define the minimum dimensions and spacing for all layers, including the metal layers that are used for routing signals. The minimum spacing between two metal routing layers is called the *pitch* of that layer. It's also more efficient for overall wiring density if the metal wiring on each layer of metal is restricted to a single direction. This is well known by people who make multi-layer printed circuit

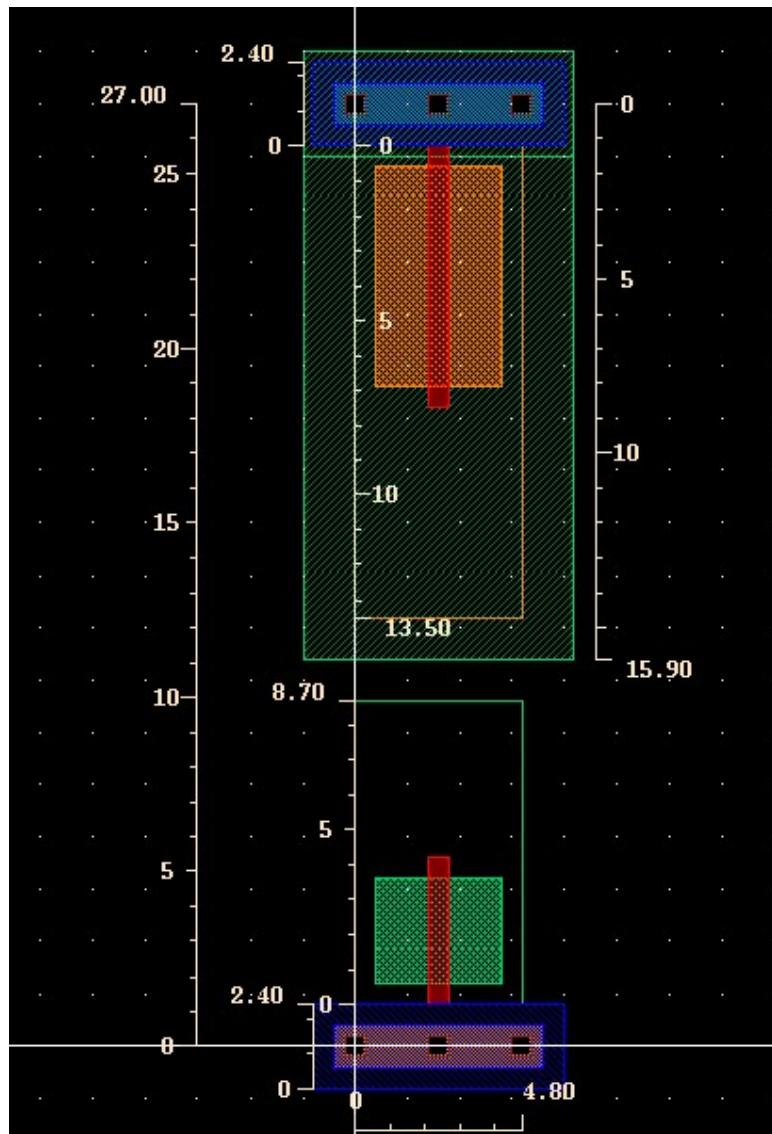


Figure 5.38: A **layout** template showing standard cell dimensions

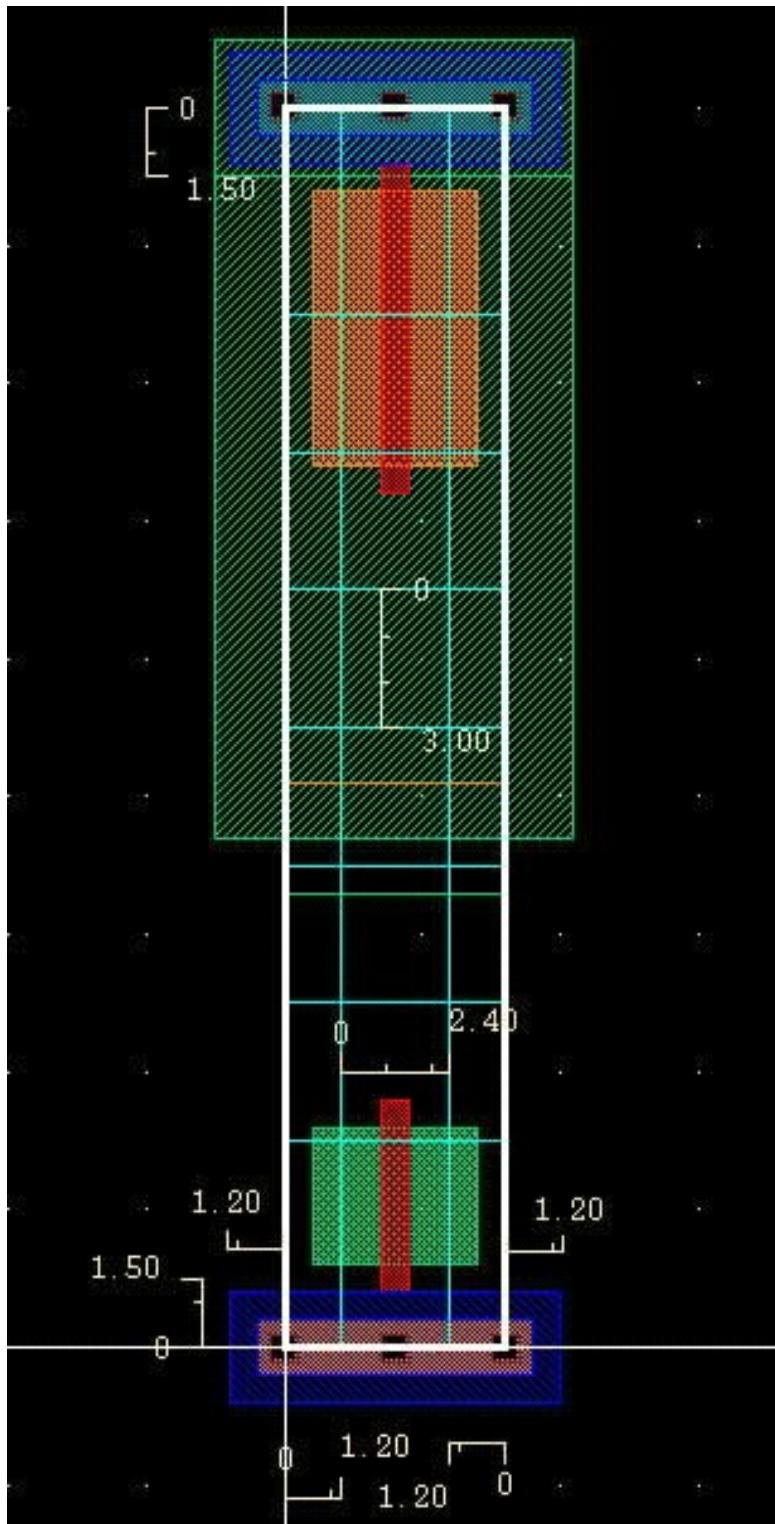


Figure 5.39: A **layout** template showing cell boundary (**prBound** layer)

boards and the principle is the same for VLSI chips. In our case we will eventually tell the place and router to prefer horizontal direction routing for layers **metal1** and **metal3**, and vertical routing for layer **metal2**. The pitch for **metal2** vertical routing according to the MOSIS SCMOS design rules is  $2.4\mu$  which, conveniently (but not coincidentally), matches our standard cell dimension increment and well and substrate contact width. The **metal3** routing pitch is  $3.0\mu$  and so sets the routing pitch for all horizontal routing by the place and route tool even though **metal1** can actually be routed on a finer pitch when routed by hand.

It turns out to make the job of the place and route tool much easier if the connection points to the cell are centered on horizontal and vertical routing pitch dimensions. The grid lines in Figure 5.39 (actually 0-width rectangles of type **wire** in the layout) show the horizontal and vertical routing channels in the cell. We follow offset grid line placement rules where the grid lines are offset by half of the vertical and horizontal grid line pitch from the origin. Thus the vertical grid lines start at  $1.2\mu$  from the y-axis and are spaced  $2.4\mu$  thereafter. Similarly, the horizontal grid lines start at  $1.5\mu$  from the x-axis and are spaced  $3.0\mu$  thereafter. However it should be noted that the first and last horizontal grid lines cannot be used, as any geometry placed on them extends into the **vdd!** or **gnd!** bus violating the rules defined above.

*Remember to use **shape pins** for all connection pins in a cell.*

Input and ouput pins for a cell should be placed at the intersection of the horizontal and vertical routing channels if at all possible. Also, if possible, do not place two pins on the same vertical or horizontal routing channel. For our cells it is most efficient if all cell I/O pins are made in **metal2**. This makes it easy for the router to jump over the **vdd!** and **gnd!** buses to make contact with the pin using a **metal2** vertical wire.

As far as possible use only **metal1** for signal routing within the standard cell layout. Routing on **metal2** layer can be used if necessary, but if it is used it should always run centered on a vertical routing channel line and never run horizontally except for very short distances (this is to reduce metal2 blockage layers enabling easier place and route). In any case, make sure that there are no horizontal metal2 runs around I/O pins. You should try to avoid any use of **metal3** in your library cells. That routing layer should be reserved for use by the place and route tool. I know of at least one commercial standard cell library for a three-metal process like ours that contains over 400 cells and uses only **metal1** in the cells (other than the  $1.2\mu$  by  $1.2\mu$  **metal2** connection pins) so it is possible to define lots of cells without using more than one metal layer.

### 5.8.3 Standard Cell Transistor Sizing

For critical designs MOSIS recommends the minimum transistor width to be  $1.5\mu$  for the AMI C5N process. The minimum transistor length for AMI C5N process is  $0.6\mu$ . Hence the minimum dimension for an **nmos** transistor pulling the output directly to **gnd!** is  $W/L = 1.5\mu/0.6\mu$ . To roughly equalize the strength of the pullup and pulldowns the minimum dimension of a **pmos** transistor pulling the output directly to **vdd!** should be  $W/L = 3.0\mu/0.6\mu$ . For series stack of transistors between output and **vdd!** or **gnd!**, the transistor width should be the number of transistors in the stack multiplied by the minimum width for a single transistor stack. If you can't make the series transistors quite wide enough, at least do your best. Also, a stack greater than four should not be used. For example, if there are three **pmos** transistors in a series stack between output node and **vdd!**, then these transistors will be sized  $W/L = (3 \times 3.0)\mu/0.6\mu = 9.0\mu/0.6\mu$  if possible.

*The 2:1 width ratio roughly compensates for the mobility of majority carriers in **pmos** transistors (holes) which is roughly half of that of **nmos** transistor carriers (electrons).*

You can also create cells with different drive strengths by adjusting the width of the output transistors. For example, an inverter with a  $1.5\mu$  pull-down and  $3.0\mu$  pullup could be considered a 1x inverter, and one with a  $3.0\mu$  pulldown and  $6.0\mu$  pullup would then be a 2x inverter denoting that its output drive is two times as strong as the 1x inverter.

A 2x inverter has extra output drive, but also extra input capacitance, so the resulting logical effort is the same, but the electrical effort is different.

A standard cell template for a cell that is four contacts wide is shown in Figure 5.40. The standard 1x inverter from the **UofU\_Digital\_v1\_1** cell library is shown in Figurefig:inv1x, and the 1x two-input nand gate from that library is shown in Figure 5.42.

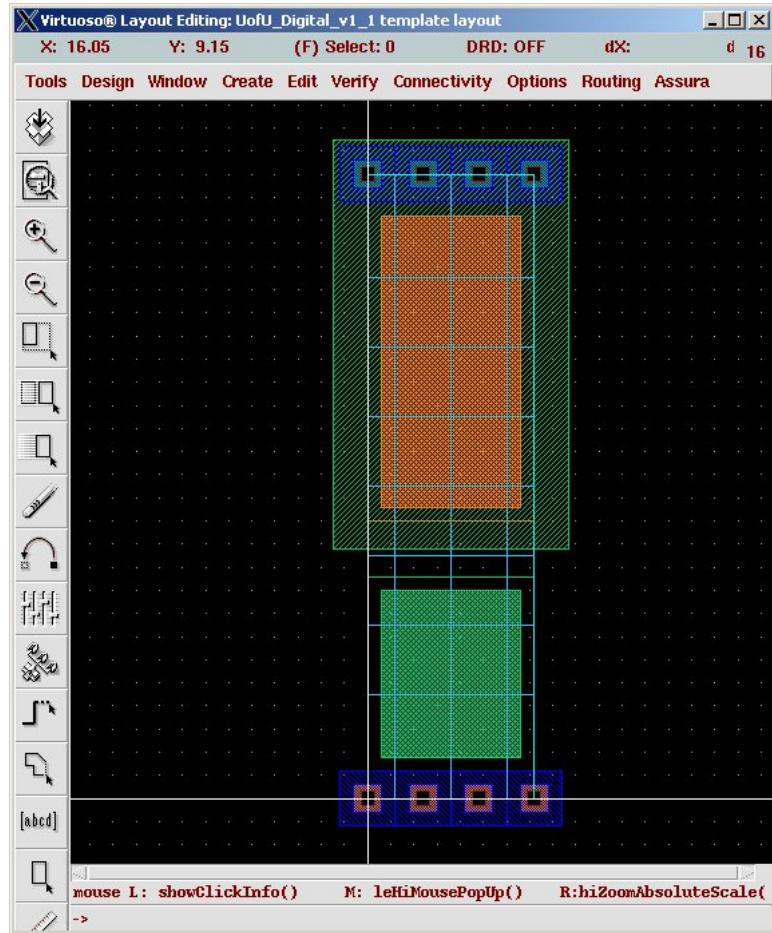


Figure 5.40: A **layout** template for a four-wide cell

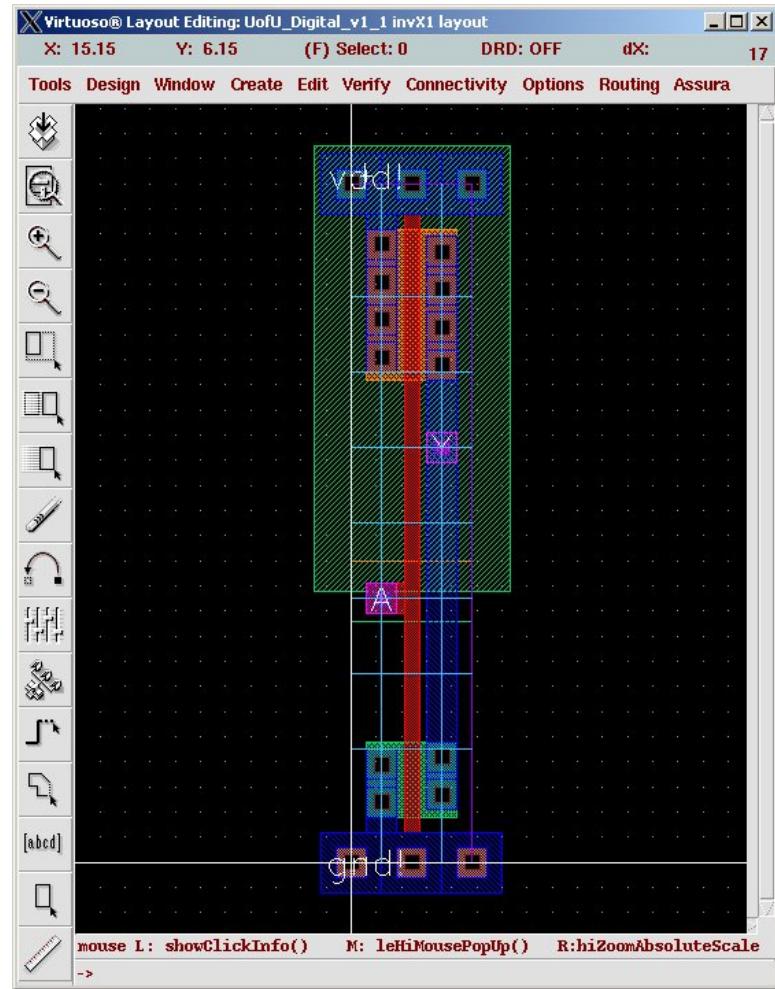


Figure 5.41: Standard cell layout for a 1x inverter

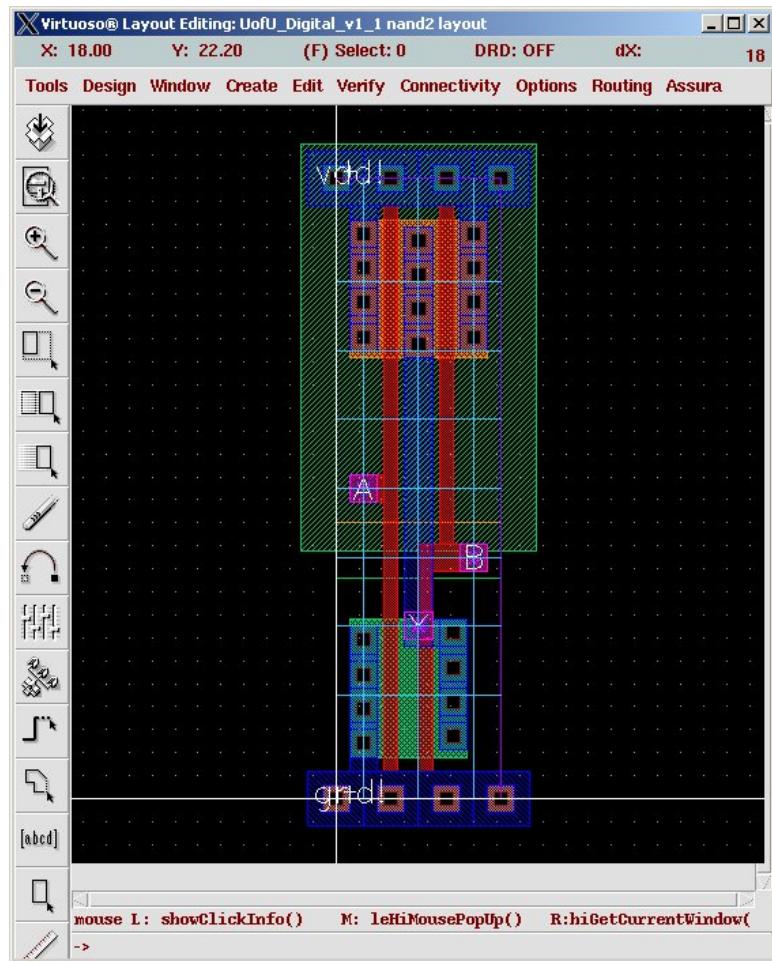


Figure 5.42: Standard cell layout for a 1x NAND gate



# Chapter 6

## Spectre Analog Simulator

THE SIMULATION described in Chapter 4 is either *behavioral simulation* where the Verilog describes high-level behaviors in a software-like style, or *switch level simulation* where the circuit is expanded all the way to transistors that are modeled as perfect switches using built-in Verilog transistor-switch models. A more accurate simulation would try to model the transistors as analog devices and use as much detail as possible in that analog model to as accurately as possible reflect the real electrical behavior of the transistor network. Analog simulators of this sort can generally trace their background to a simulator called **Spice** that was originally developed at Berkeley in the early 1970's. Through the early 1980's Spice was still written in FORTRAN, and you can actually still obtain the Spice2G6 FORTRAN code from Berkeley if you look around carefully enough. Spice3 was the first C-language version in 1985. Since escaping from Berkeley there have been many commercial versions of Spice and Spice-like programs including Hspice, Pspice, IS\_Spice, and Microcap.

The analog simulator integrated into the Cadence framework is called **Spectre**. It is similar to Spice in terms of simulating the analog behavior of the transistors, and it even accepts “spice decks” as input in addition to its own **Spectre** format. It operates slightly differently internally and is claimed to be a little faster than Spice. From your point of view they are essentially identical simulators though. They take the same input decks and produce the same waveform outputs that show the analog behavior of the circuit network.

An important thing to keep in mind is that while the Verilog simulations used abstract logical 0 and 1 signals for inputs and outputs, analog simulations with **Spectre** will use analog voltages for inputs and outputs. Thus, you now need to be concerned with things like what voltage the power supply is set to, what the voltages should be to be considered a logic 1 or logic

The FORTRAN history of Spice is why even today the input files for analog simulators are called “spice decks.” That term goes back to the punched card decks that were used for input to these FORTRAN programs.

Note that the simulator described in this chapter is **Spectre** and not **SpectreS** which is an older version with a socket interface. Depending on which version of the NCSU CDK you are using, **SpectreS** may be the default, and you may have to modify the **NCSU\_Analog\_Parts** library for **Spectre** as described in the appendices.

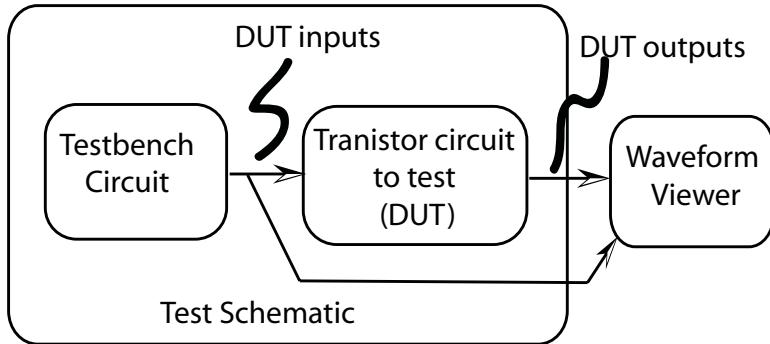


Figure 6.1: The analog simulation environment for a circuit (DUT)

0, what the slope of changing inputs should be, and other analog electronic considerations.

The overall simulation environment for analog simulation of a circuit is very similar to the DUT/testbench model from Chapter 4. The circuit you would like to simulate is the *Device Under Test* or DUT, and you need to construct some sort of testbench circuit around the DUT to provide inputs and sense outputs. The general form of this DUT/testbench approach is shown in Figure 6.1 (compare to Figure 4.1). For analog simulation the inputs need to be described in terms of analog voltages, and the outputs will be returned in terms of analog voltages.

In this chapter we'll see four different ways to use the **Spectre** simulator as it is integrated with the other **Cadence** tools. The first three techniques are *transient* simulations of how the circuit behaves over time, and the fourth is a *static* simulation of DC operating points of the circuit. One main difference between these two types of simulation is that in the transient case the inputs and outputs are described as voltages over time, and in the static case the inputs and outputs are defined as static single voltages. In practice these static operating points are swept through a range of voltages to generate curves of, for example,  $V_{in}$  versus  $V_{out}$  or  $V_{in}$  versus  $I_{out}$ . The four examples of analog simulation in this chapter are:

1. Transient simulation of a transistor-level schematic described in **Composer**.
2. Transient simulation of a transistor-level schematic where some of the cells have extracted views from the layout and include more information about the physical layout of the transistors and parasitic capacitance from the layout for more accuracy.
3. Transient mixed-mode analog/digital simulation where part of the cir-

cuit is simulated at the switch or behavioral level with a Verilog simulator and part is simulated at the analog level with **Spectre**. This can be very useful for simulating a critical component in the framework of a larger system without simulating the entire system at the analog level.

4. Static DC operating point simulations of individual circuits.

## 6.1 Simulating a Schematic (Transient Simulation)

To demonstrate the simplest mode of Spectre simulation we'll use the NAND gate from Chapter 3, Section 3.3, Figure 3.12. The symbol for this NAND gate was shown in Figure 3.14. To simulate this gate we'll need to open a new schematic and add some analog components so that the simulator will know about the analog environment in which to simulate the NAND. What we're aiming for is to generate analog input waveforms that will stimulate the NAND through all possible input combinations. In Chapter 4 this was done with a testbench Verilog program. For analog simulation we will construct a testbench circuit in **Composer** that includes *voltage sources* to define the power supply and the input signals. Voltage sources are statements in **Spectre** input decks that define voltages in the circuit to be simulated. We will instantiate them as schematic components and they will be extracted into the simulator input deck by the netlister. Commonly used voltage sources in the **NCSU\_Analog\_Parts** library include:

**vdc:** A static DC voltage source at a user-defined voltage.

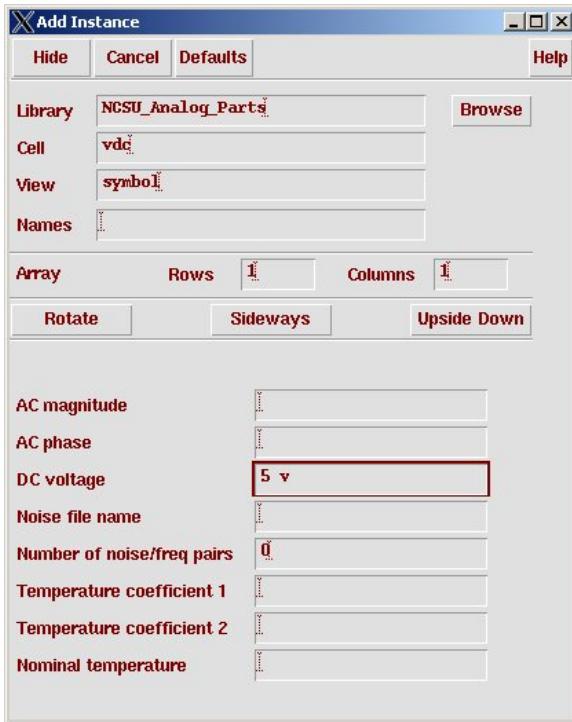
**vpulse:** A voltage source that generates voltage pulse streams. The user can control aspects of the pulse including delay, pulse width, period, rise time, fall time, and other parameters.

**vpwl:** A piecewise linear voltage source. The voltage is defined as a set of linear segments that define the output voltage waveform. The syntax is a series of time/voltage pairs that define the vertices of the piecewise linear waveform.

**vpwlf:** A piecewise linear voltage source that takes its waveform definition (time/voltage pairs) from a file.

**vsin:** A sine wave voltage generator. The user can set the frequency, amplitude, and other parameters of the output.

**vexp:** A voltage source that generates a single pulse consisting of an exponentially defined rise and an exponentially defined fall.

Figure 6.2: Component parameters for the **vdc** voltage source

*Remember to include an **ASIZE** frame from **UofU\_Sheets** around your schematic!  
Note that the **vdd** and **gnd** symbols actually connect the nets to the global **vdd!** and **gnd!** labels so that power and ground can be referenced globally in the hierarchy through this one **vdc** connection.*

For the NAND example test circuit we'll start with a new schematic, include a copy of the NAND gate (the DUT), and then use **vdc** and **vpulse** voltage sources for the testbench circuit. Open a new schematic, I'll call mine **nand-test**, and add an instance of the NAND gate from your library (or from the **UofU\_Digital\_v1\_1** library if you haven't designed one yet).

Now along with this NAND gate you'll need some analog components from the **NCSU\_Analog\_Parts** library. To let Spectre know about the power supply you need to include a **vdd** and a **gnd** component. These are in the **Supply\_Nets** category in the **NCSU\_Analog\_Parts** library. Go to the **Voltage Sources** category in **NCSU\_Analog\_Parts** and add a **vdc** DC voltage source. When you instantiate this part, or by using the **Q** hot key later, you should set the DC voltage on this component to be **5 v** as seen in Figure 6.2. The top connection of the **vdc** The top terminal of the **vdc** component should be connected to the **vdd** symbol and the bottom terminal should be connected to the **gnd** symbol. This indicates that for this simulation there should be a five volt power supply (5 v DC between **vdd** and **gnd**).

To generate the input waveforms I'll use two **vpulse** voltage sources. By defining them to have the same pulse width and period, but delaying one of

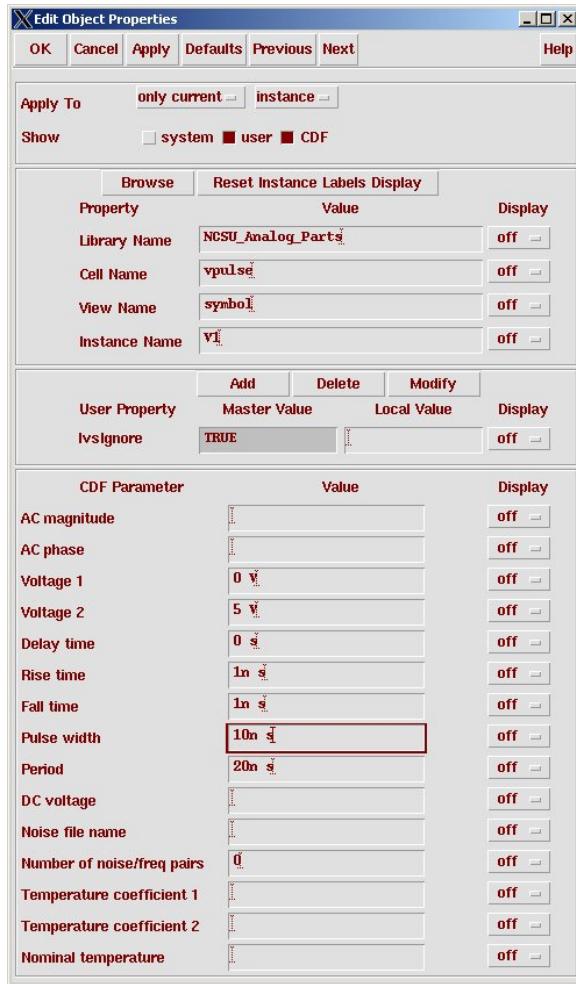


Figure 6.3: Component parameters for the **vpulse** voltage source

the pulses, I can generate all input combinations of high and low voltages to the inputs of the NAND gate. The parameter dialog box for the **vpulse** voltage source is seen in Figure 6.3 where you can see the settings for one of these components. The other **vpulse** component is the same except that it has 5ns of delay before the pulses start.

Finally, I'll connect the NAND gate (DUT) to a capacitor to simulate the effect of driving into a capacitive output load. Another approach to this would be to add other gates to the output of the DUT to simulate driving into specific other gates. I'll add a **cap** component from the **R\_L\_C** category in the **NCSU\_Analog\_Parts** library and give it a relatively huge capacitance of 1pf. The final **nand-test** schematic looks like Figure 6.4.

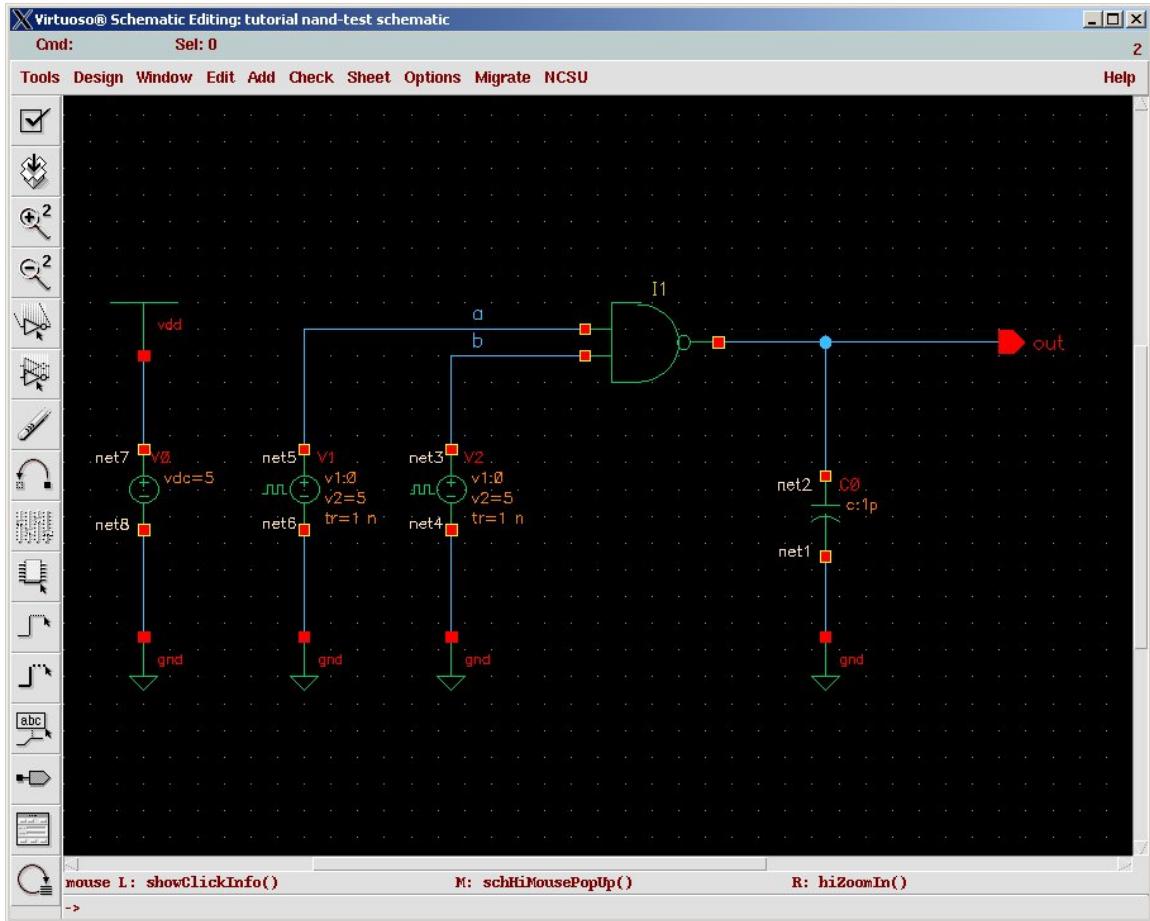


Figure 6.4: Schematic for the nand-test DUT/testbench circuit

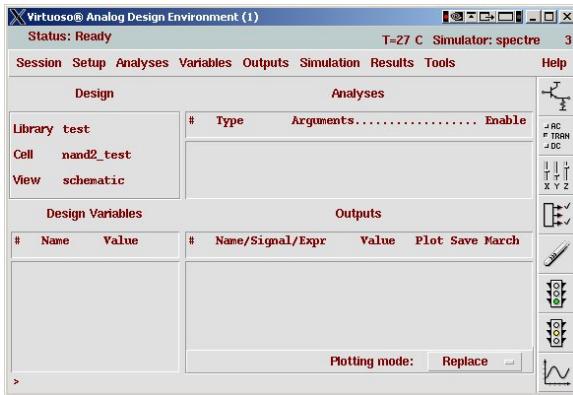


Figure 6.5: Virtuoso Analog Environment control window

## 6.2 Simulation with the Spectre Analog Environment

Starting with your **nand-test** schematic open in Composer, select the **Tools → Analog Environment** menu choice. This will bring up a **Virtuoso Analog Design Environment** dialog box as shown in Figure 6.5. You should see the **Library**, **Cell**, and **View** already filled in. If not, or if you'd actually like to simulate a different schematic, you can change this with the **Setup → Design** menu in the **Analog Environment**. The other **setup** options should be set for you. We'll be using the **Spectre** simulator, and the **Model Path** should be set to point to the generic nominal transistor models for the class (**ami06.scs** in the class directory). At some point you may wish to point to a different set of models to get models for worst-case (slow) processes corners, or to use models from a specific MOSIS fabrication run. You can set the path to other models that you want to use using the **Setup → Model Libraries** menu.

Now select **Analysis → Choose...** in the **Analog Environment** or click on the **.** Select **tran** for a transient analysis, and make sure to fill in the **Stop Time** to let the simulator know how long the simulation should run. I'll fill this in to **300n** for 300 nanoseconds of simulation. This dialog box is shown in Figure 6.6.

Now you need to select which circuit nodes you would like to see plotted in the output waveform. The easiest way to do this is to select the nodes that you'd like to have plotted by clicking on them in the schematic. Select the **Outputs → To Be Plotted → Select on Schematic** menu choice. The **Status** shown in the top of your **Composer** window should now say **Selecting outputs to be plotted...** Select the wires (nodes) that you'd like to see in your output waveform by clicking. I'll select the **a** and **b** inputs to the NAND, and the **out** output node. Note that the wires change color in the

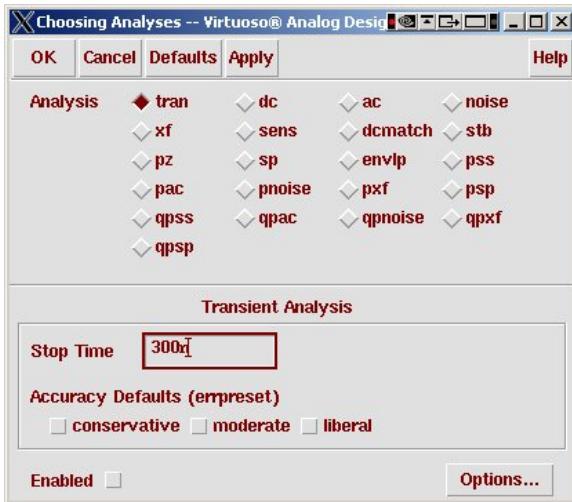


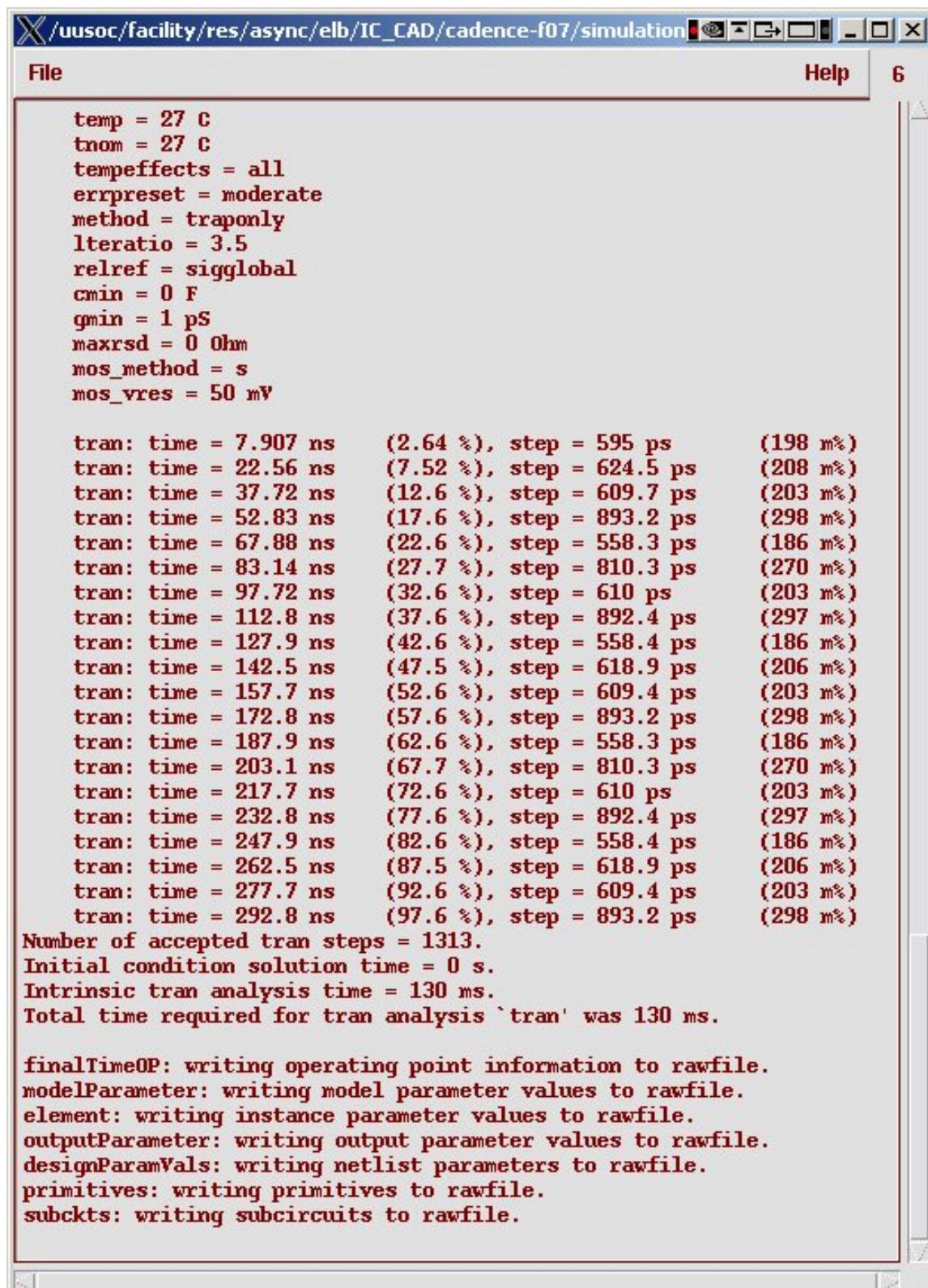
Figure 6.6: Choosing Analysis dialog box

schematic to indicate that they've been selected, and they also appear in the **outputs** pane of the **Analog Design Environment** control window.

*One way to get a runaway simulation is to forget the **n** in the description of the time to simulate. If you're simulating for 300s instead of 300ns the simulation may take a long time!*

Now that you've chosen the design to simulate, the type of simulation (transient, 300ns), and the outputs to be plotted, you can start the simulation by selecting **Simulation → Netlist and Run** from the menu, or the **Netlist and Run** widget which looks like a traffic light with a green light. When you run the simulation you'll see the simulation log in the **CIW**. If there are any issues with the simulation the warning and error messages will show up here. If the simulation completes you'll see the elapsed time in the **CIW**, a **Spectre** window that shows the log results of running the simulation, and a waveform window will open up with the results that you specified plotted. The **Spectre** log window looks like that in Figure 6.7.

For our NAND example the waveform window initially looks like that in Figure 6.8. It's a little confusing because all the waveforms are layered on top of each other. This is great if you want to see the details of one waveform in relation to another, but mostly it's just confusing. To fix this you can move to *strip chart mode* by using the **Axis → Strips** menu or using the **strip chart mode** widget which looks like four white bars. This will separate the waveforms so that each one is in its own strip. Now the waveform viewer looks like that in Figure 6.9. You can see the inputs in the top two strips (pink and purple in this example). These inputs are generated by the two **vpulse** voltage sources. The output is shown in the bottom strip in yellow. Because of the huge capacitance I put in the **test-nand** schematic the output has a shape characterized by the exponential behavior of an output transistor charging a large capacitance. Using the **Zoom** menu option I can



```

X/uusoc/facility/res/async/elb/IC_CAD/cadence-f07/simulation Help 6

File

temp = 27 C
tnom = 27 C
tempeffects = all
errpreset = moderate
method = traponly
lteratio = 3.5
relref = sigglobal
cmin = 0 F
gmin = 1 pS
maxrstd = 0 Ohm
mos_method = s
mos_vres = 50 mV

tran: time = 7.907 ns      (2.64 %), step = 595 ps      (198 m%)
tran: time = 22.56 ns      (7.52 %), step = 624.5 ps     (208 m%)
tran: time = 37.72 ns      (12.6 %), step = 609.7 ps     (203 m%)
tran: time = 52.83 ns      (17.6 %), step = 893.2 ps     (298 m%)
tran: time = 67.88 ns      (22.6 %), step = 558.3 ps     (186 m%)
tran: time = 83.14 ns      (27.7 %), step = 810.3 ps     (270 m%)
tran: time = 97.72 ns      (32.6 %), step = 610 ps        (203 m%)
tran: time = 112.8 ns      (37.6 %), step = 892.4 ps     (297 m%)
tran: time = 127.9 ns      (42.6 %), step = 558.4 ps     (186 m%)
tran: time = 142.5 ns      (47.5 %), step = 618.9 ps     (206 m%)
tran: time = 157.7 ns      (52.6 %), step = 609.4 ps     (203 m%)
tran: time = 172.8 ns      (57.6 %), step = 893.2 ps     (298 m%)
tran: time = 187.9 ns      (62.6 %), step = 558.3 ps     (186 m%)
tran: time = 203.1 ns      (67.7 %), step = 810.3 ps     (270 m%)
tran: time = 217.7 ns      (72.6 %), step = 610 ps        (203 m%)
tran: time = 232.8 ns      (77.6 %), step = 892.4 ps     (297 m%)
tran: time = 247.9 ns      (82.6 %), step = 558.4 ps     (186 m%)
tran: time = 262.5 ns      (87.5 %), step = 618.9 ps     (206 m%)
tran: time = 277.7 ns      (92.6 %), step = 609.4 ps     (203 m%)
tran: time = 292.8 ns      (97.6 %), step = 893.2 ps     (298 m%)

Number of accepted tran steps = 1313.
Initial condition solution time = 0 s.
Intrinsic tran analysis time = 130 ms.
Total time required for tran analysis `tran' was 130 ms.

finalTimeOP: writing operating point information to rawfile.
modelParameter: writing model parameter values to rawfile.
element: writing instance parameter values to rawfile.
outputParameter: writing output parameter values to rawfile.
designParamVals: writing netlist parameters to rawfile.
primitives: writing primitives to rawfile.
subckts: writing subcircuits to rawfile.

```

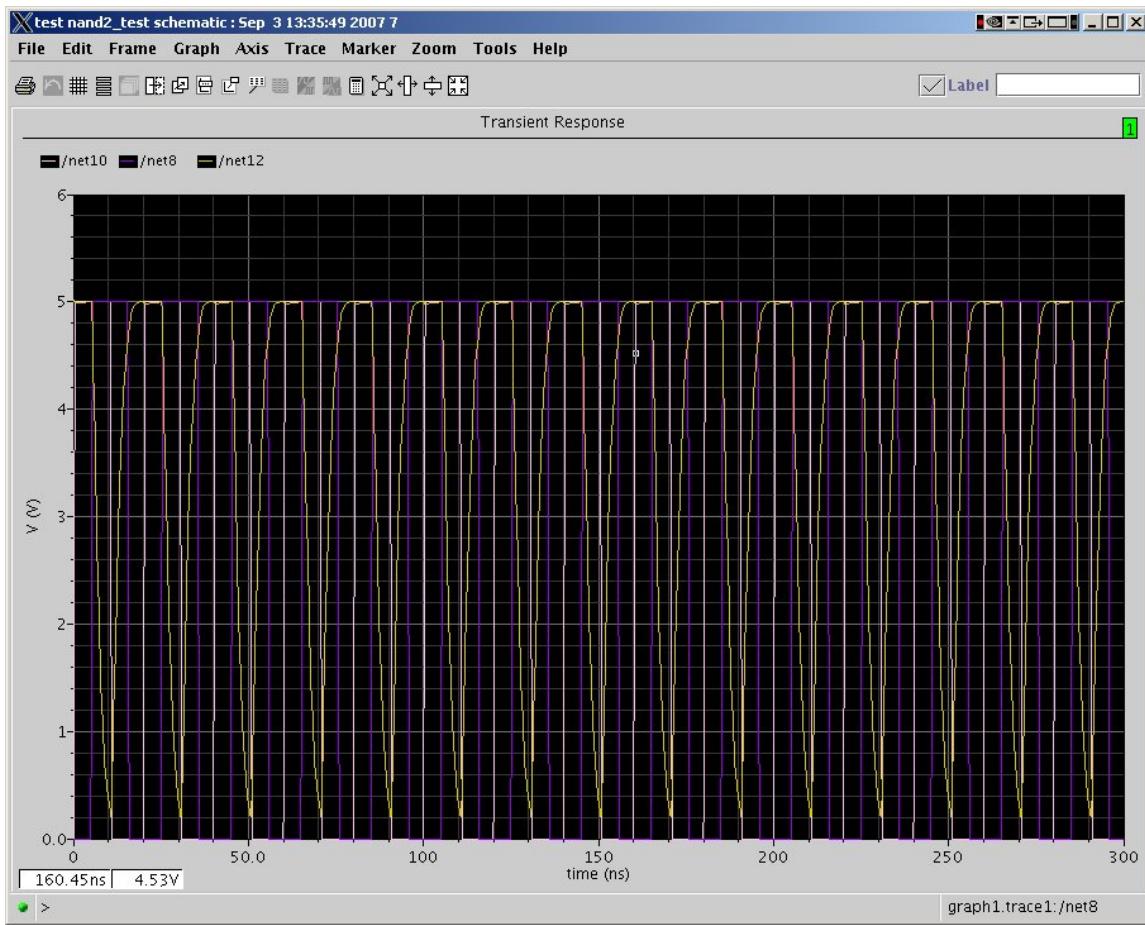


Figure 6.8: Initial Waveform Output Window

zoom in to look at the waveforms more carefully as in Figure 6.10.

You should experiment with the waveform window to see how you can use it to measure and compare waveforms. Some helpful tools are **markers** which are vertical or horizontal cursors that can be moved around in the waveforms and used to measure the curves, and **zoom** to change what part of the waveforms you are looking at.

You can place markers in the schematic using the **Marker → Place → Vert Marker** or **Horiz Marker** menus. Markers are vertical or horizontal lines that you can place in the schematic which will measure points on the waveforms as you move them around. As an example I'll place two vertical markers in the waveform. These markers will be labeled **M0** and **M1**. You can move them around with the left mouse button (the cursor changes to  $<>$  when you hover near them). Make sure that when you select a location

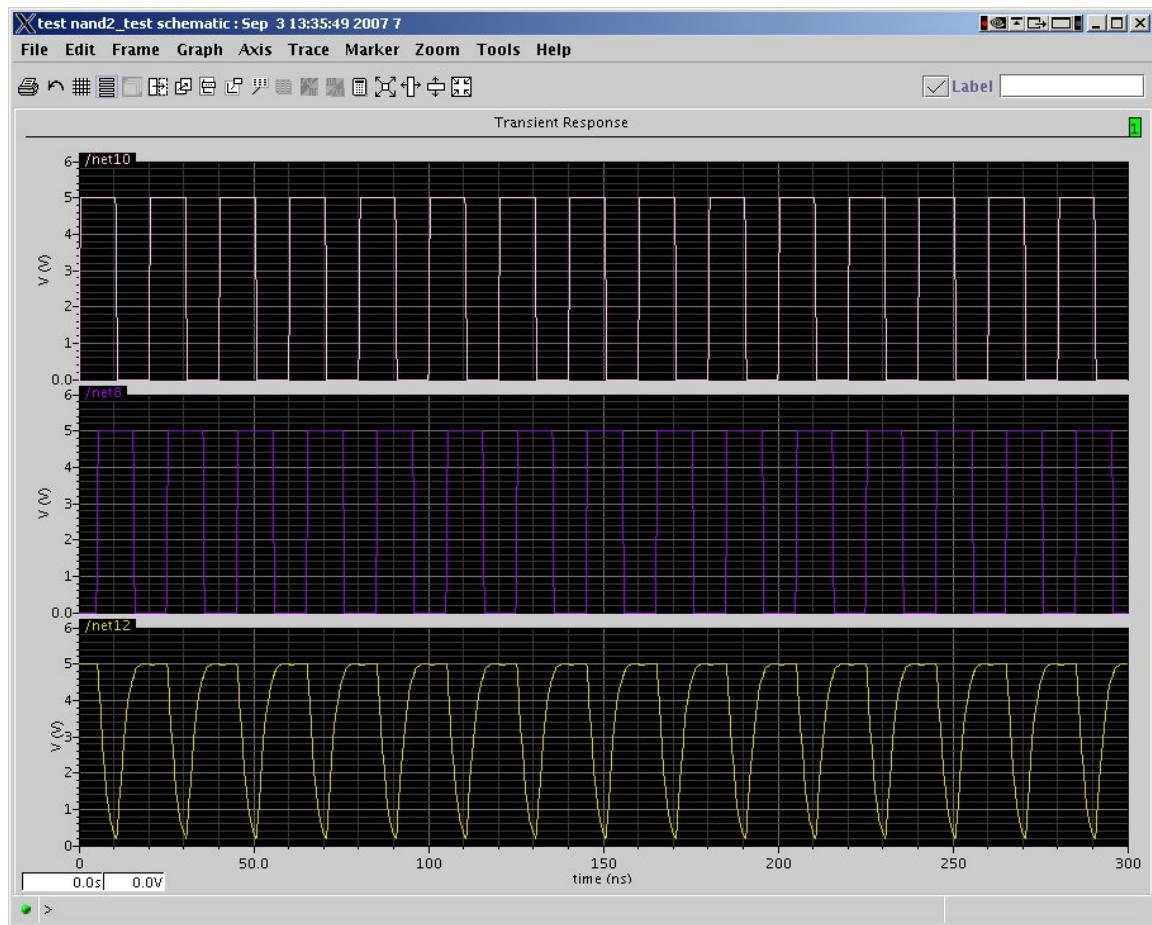


Figure 6.9: Waveform Output Window in Strip Mode

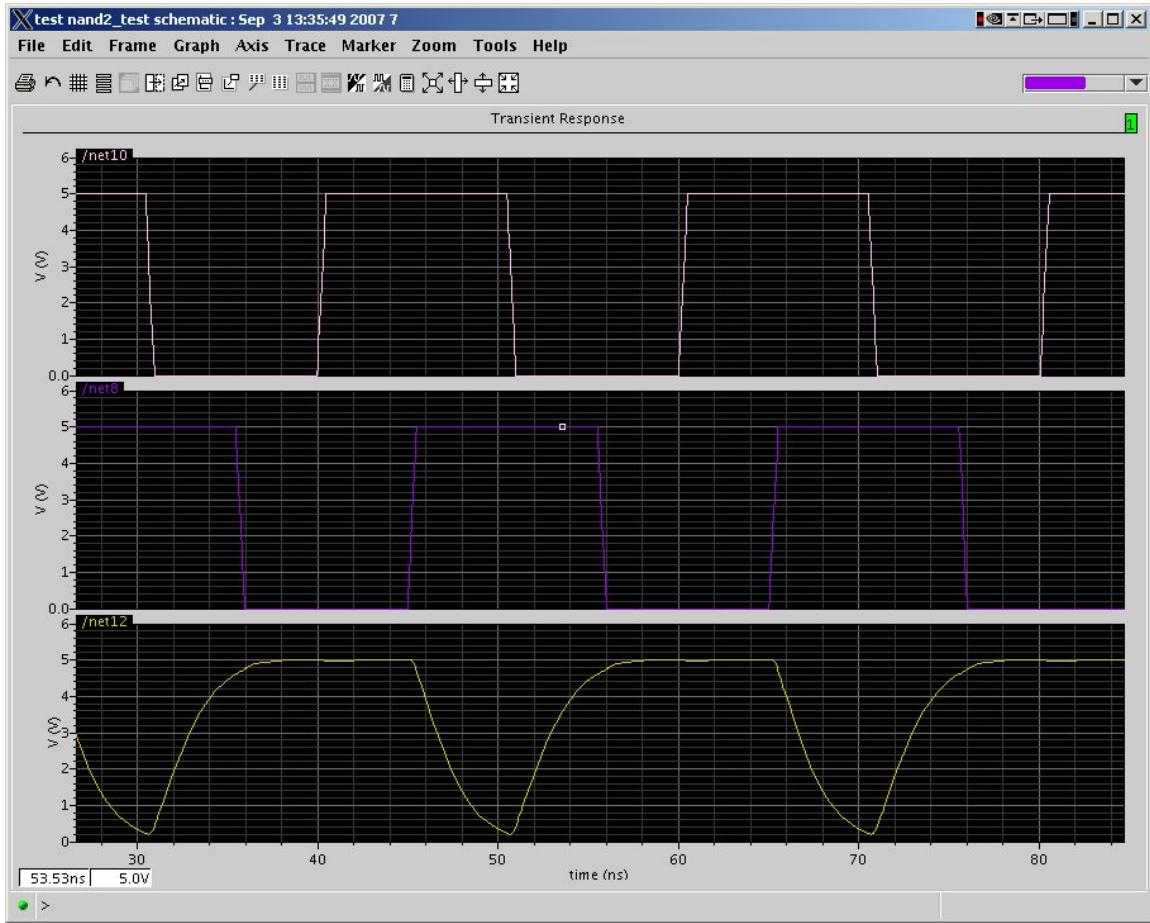


Figure 6.10: Waveform Output Window: Zoomed View

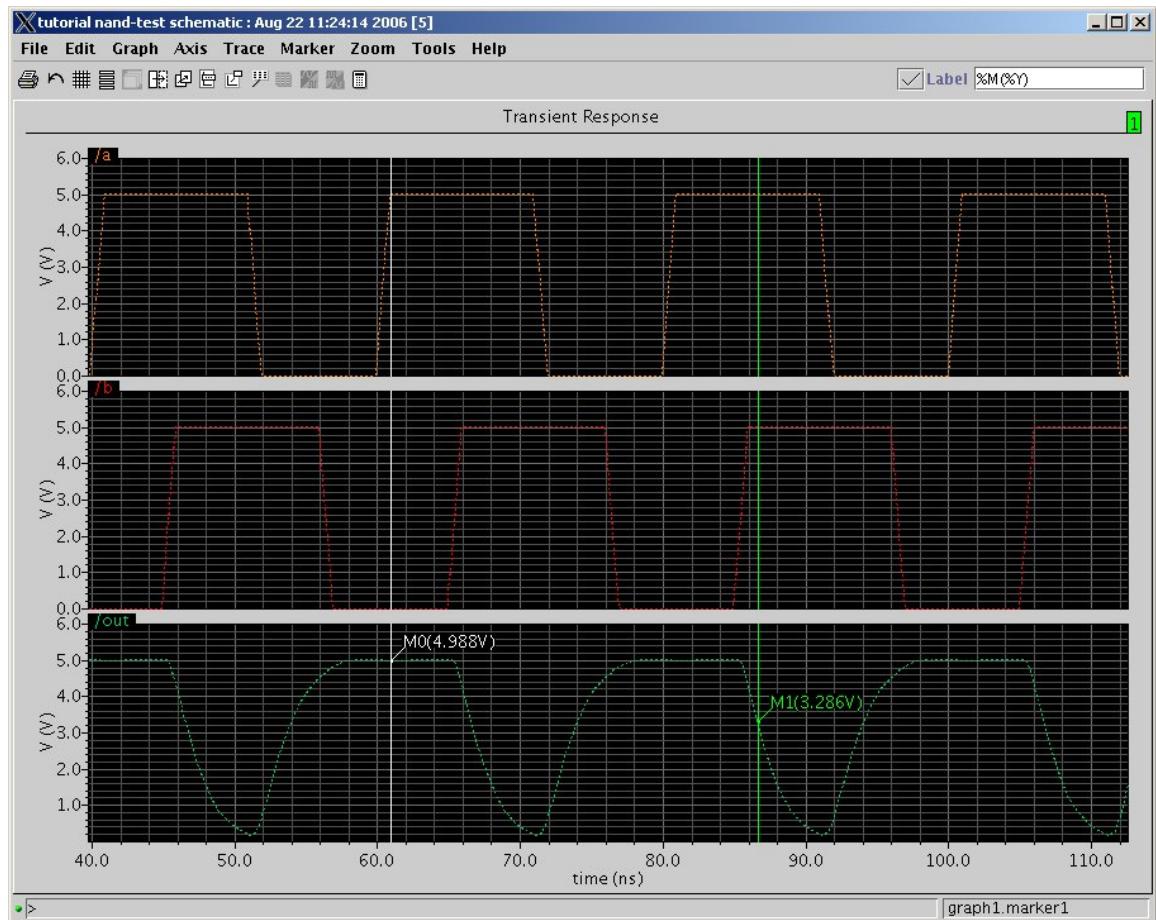


Figure 6.11: Waveform Output with Markers

for the marker that you select inside the trace that you want to see. This will attach the vertical voltage marker on that trace. If you want to change the trace that a maker is associated with you can pick up the trace, move it off the right side of the screen, and then move back into the trace you want to see. Figure 6.11 shows the trace with two output markers inserted. You can use both horizontal and vertical markers to measure the waveforms.

### 6.3 Simulating with a Config View

In the previous section, the NAND gate was simulated by making a separate schematic that included the NAND (the DUT) and a testbench circuit around that NAND instance, then simulating that schematic with Spectre. This works very well if what you want to simulate is the **schematic** (or

Recall that the **analog\_extracted** view is generated only after LVS succeeds! See Chapter 5, Section 5.6.1.

*The **analog\_extracted** view is essentially the same as the **extracted** view, but with additional information about power supply connections for simulation.*

**cmos.sch**) view of the NAND. That is, the schematic that contains the transistor level netlist. But, as you've seen in Chapter 5, you can also have a **layout** view, and from the layout view you can generate an **extracted** and an **analog\_extracted** view of the same cell. The **analog\_extracted** view has additional information about the circuit that is useful for simulation such as parasitic capacitance of the wires, exact layout dimensions of the transistors, etc. In order to simulate the **analog\_extracted** view you need a way to tell Spectre (through the **Analog Environment**) which view you really want to simulate.

Imagine that you have a **nand-test** schematic like that shown in Figure 6.4. If you make that schematic into yet another view, called a **config** view, then you can use that **config** view to select which underlying view of the NAND you'd like to simulate: the **cmos.sch** or the **analog\_extracted**.

One way to do this is by making yet another view called a **config** view of the testbench cell. To make a **config** view you need a **schematic** view to start with. I'll use the existing **nand-test** schematic. Now, in the **Library Manager** select **File** → **New** → **Cell View**. Fill in the **Cell Name** as **nand-test** and the **View Name** as **config**. This should automatically select **Hierarchy-Editor** as the **Tool**. See Figure 6.12. In the **New Configuration** window that pops up select **Use Template**. In the **Use Template** dialog select **Spectre** as the template name and click **OK**. Back in the **New Configuration** dialog you will see that the **Global Bindings** and other fields have been filled in. Make sure that the **Library** and **Cell** are filled in with your schematic name, and change the **View** from **myView** to **schematic**. Figure 6.13 shows this dialog after this has been done. Click **OK** to create the **config** view.

What you see for the **config** view is the cell described in the **Hierarchy Editor**. The initial window is seen in Figure 6.14. You can see that the **nand-test** cell is defined by five components from the **NCSU\_Analog\_Parts** library (**cap**, **nmos**, **pmos**, **vdc**, and **vpulse**) that have **SpectreS** views), a **nand2** component from my **UofU\_Example** library (with a **cmos.sch** view), and the top-level **nand-test** schematic (a **schematic** view). This is just a list of all the cells in the hierarchy. A better view is to change the view to the **tree** view using **View** → **tree**. The **Hierarchy Editor** window with the **tree** view is shown in Figure 6.15.

In the **tree** view you can see each of the components at the top level, and by expanding the **nand2** component you can see the transistors that are in that component's level of the hierarchy. Now select the **nand2** component. Right click on **nand2** and in the pop-up menu select **Set Instance View** → **analog\_extracted**. In the **Hierarchy Editor** window you now see that the **View to Use** field of the **nand2** component now says **analog\_extracted** in blue. Update the **config** view with **View** → **Update**, **File** → **Save** and you

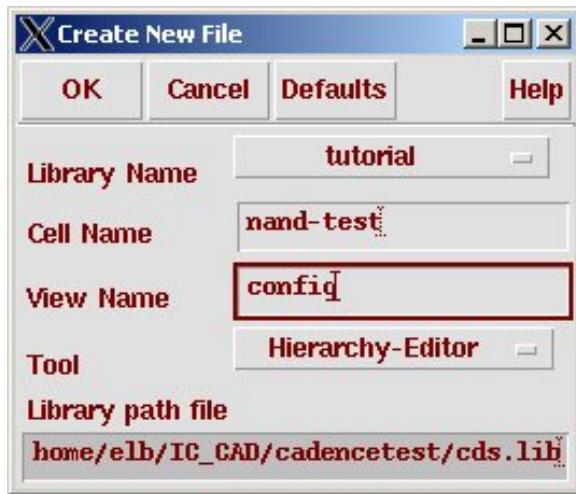


Figure 6.12: Create New File dialog for the config view

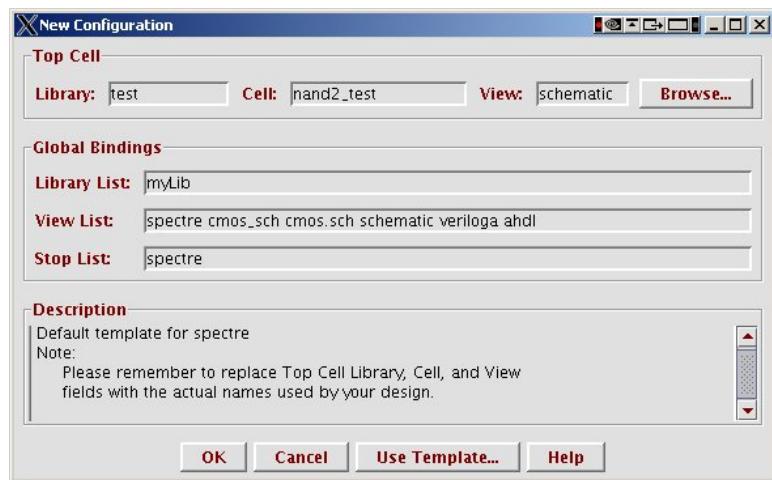
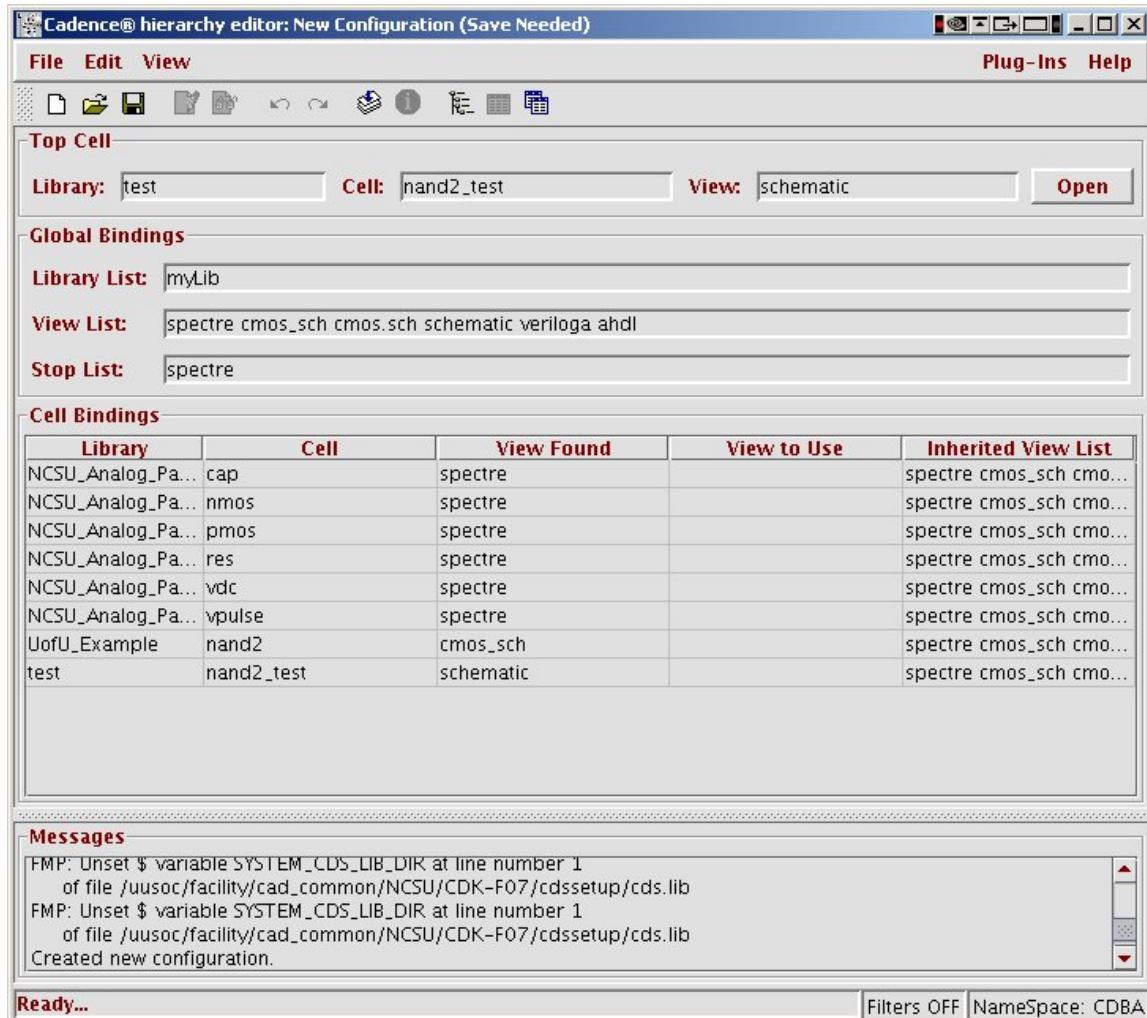
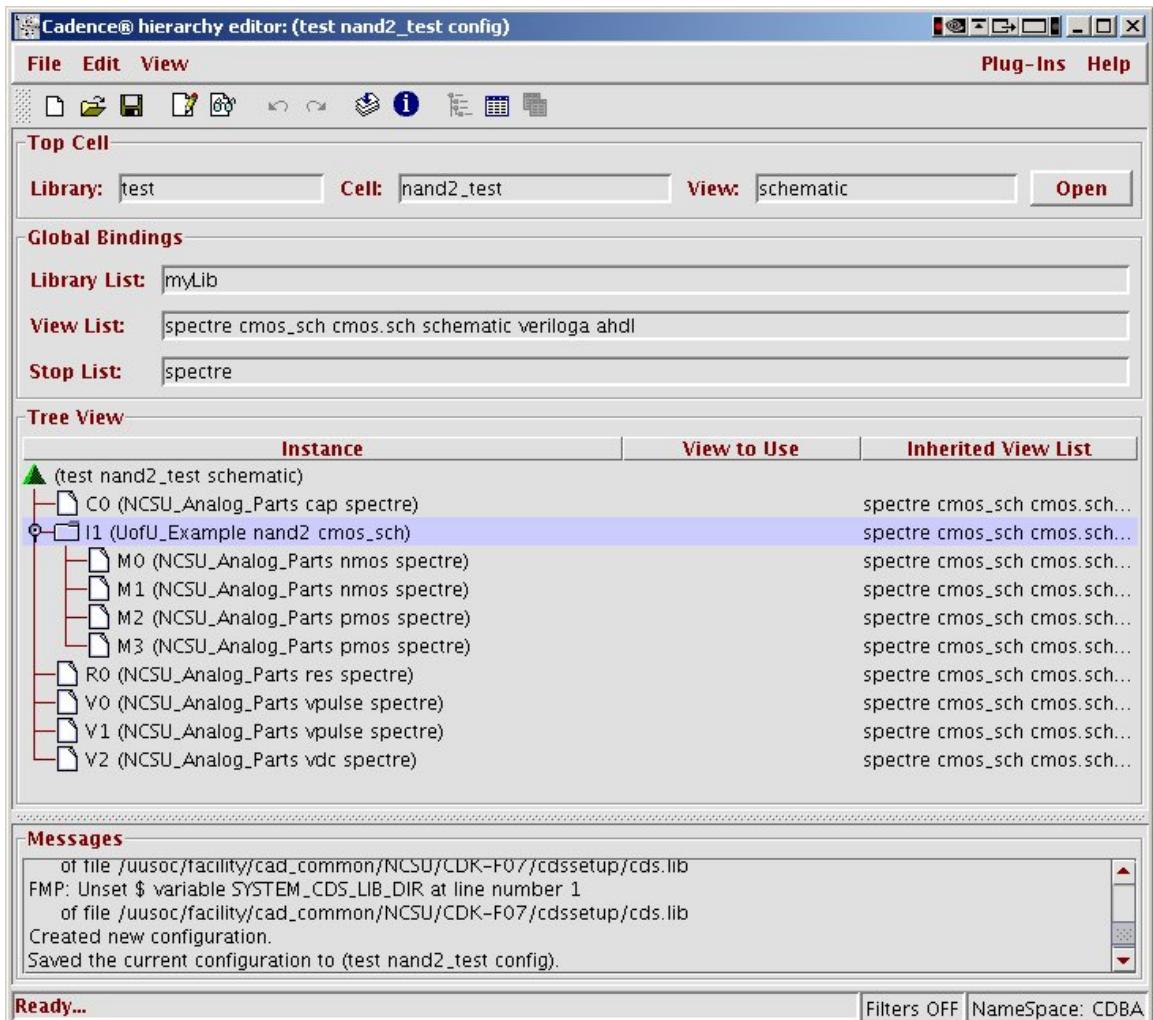


Figure 6.13: New Configuration dialog box

Figure 6.14: Hierarchy Editor view for **nand-test** (table view)

Figure 6.15: Hierarchy Editor view for **nand-test** (tree view)

have saved a new view that will use the **analog\_extracted** view instead of the **cmos.sch** view when you use **Spectre**.

After closing the **Hierarchy editor** I now have two views of my **nand-test** cell: **schematic** and **config**. If I double click on the **config** view to open it, I get a choice of whether to open the **schematic** along with it. If I do this, and select **Tools** → **Analog Environment** as I did in the previous section, you can see that the **Analog Design Environment** dialog opens up just like in the last section (Figure 6.5), but this time the **View** is set to be **config**. If you follow the same steps for simulation you will get the same waveform output, but this time it will have been the **analog\_extracted** view that was simulated because that's the view you specified in the **config** view through the **Hierarchy editor**. You can use this technique in a large schematic to select any combination of (simulatable) views that you want for each of the components. For this example it makes almost no difference in the simulation results, but for a larger circuit the difference can be significant because of the additional information in the **analog\_extracted** view that is not in the **cmos.sch** view.

## 6.4 Mixed Analog/Digital Simulation

The most detailed and accurate simulation in this flow is analog simulation using **Spectre**. This very detailed simulation of every transistor in your design gives you timing results that are within a few percent of the fabricated chip. Of course, it's also very slow, especially for large chips. It's also difficult to use the **vpulse** and **vpwl** components to generate complex data streams for digital circuits. However, if you really want good timing information about your chip, there's no substitute for analog simulation of the whole chip!

*Even more detailed simulation is possible that includes 3-d transistor and interconnect models, field-solvers for understanding the effects of changing signals, localized heating, and many other effects, but they're beyond the scope of this flow.*

Luckily, there is a compromise between full analog simulation and purely functional simulation. **Cadence** is designed to do mixed mode simulation where part of your design is simulated using Verilog-XL or NC\_Verilog and part is simulated using **Spectre**. You can use this capability for a variety of simulation tasks.

1. You can simulate circuits that are actually mixed mode circuits. That is, systems that have both analog and digital components like a successive approximation ADC.
2. You can simulate large digital systems by simulating most of the circuit using a Verilog simulator, but specify that certain critical parts are simulated using the analog simulator for more accuracy.

3. You can simulate the entire system with the analog simulator, but have a small set of digital components in your testbench file so that you can write the testbench in Verilog instead of using **vpulse** and **vpwl** components.

Cadence does this sort of simulation through the **config** view that was described in the previous section. It also automatically installs *interface elements* between the digital and analog portions of the design (they're called **a2d** and **d2a**), and automatically interfaces the two simulators.

To set up for mixed mode simulation you need a top-level schematic that includes your DUT, and also components that drive the input and deliver the outputs. In the top-level schematic for the pure Spectre case the inputs were driven by analog voltage sources. Because we want to drive the inputs from Verilog in this case, the inputs should be driven by digital sources. Because the mixed-mode simulator wants to include interface elements between the digital and analog parts of the circuit, I'm going to include two inverters in a row driving the inputs, and two inverters in a row for the output signals. This will let me make one of these inverters digital (so that it can be driven from Verilog) and the second inverter can be analog (so that it will provide an analog signal to the NAND gate DUT). If there are inputs to your DUT that you want to be driven from analog voltage sources you can also include those in this test schematic. Because part of the simulation is analog, you also need to include the **vdc** for the power supply connection in any case. The **mixed-test** schematic is shown in Figure 6.16.

Create a config view of your testbench schematic as if you were doing an analog simulation. When you get to the **New Configuration** dialog box, use a template to set things up. To set up for mixed mode simulation use the **SpectreVerilog** template (remember to change the view to schematic). The main difference from this **config** view and the **config** view used for pure Spectre is that the **View List** and **Stop List** include some Verilog views because part of the circuit will be simulated as Verilog.

Now you have a config view that describes each instance in the schematic and what view to use to simulate it. The trick to mixed mode simulation is to specify a view for some cells that results in analog simulation, and a view for other cells that results in digital (Verilog) simulation for those cells. Note that you're making this choice for a tree of cells. If you make a choice for a cell, all cells under that cell inherit that choice unless you descend into the hierarchy and override that decision. From the initial **Hierarchy Editor** view (Figure 6.17) you can see that every component has the **behavioral** view selected initially so the entire simulation will be digital Verilog simulation (this is the **tree view** of the **config** view).

Before we change the partitioning of the circuit to make some of the

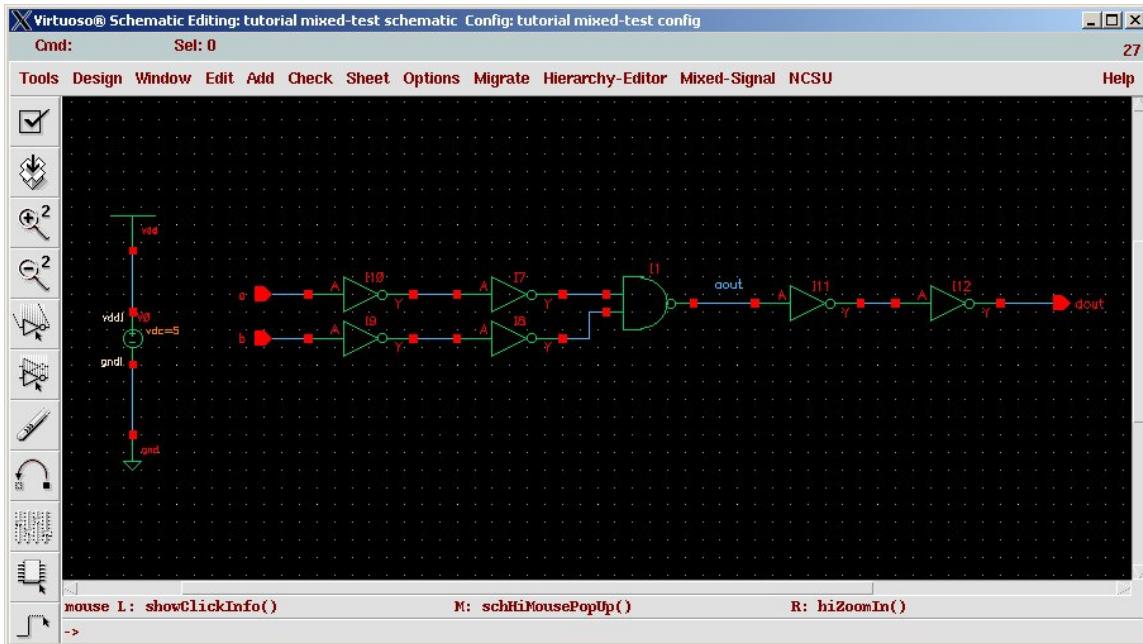
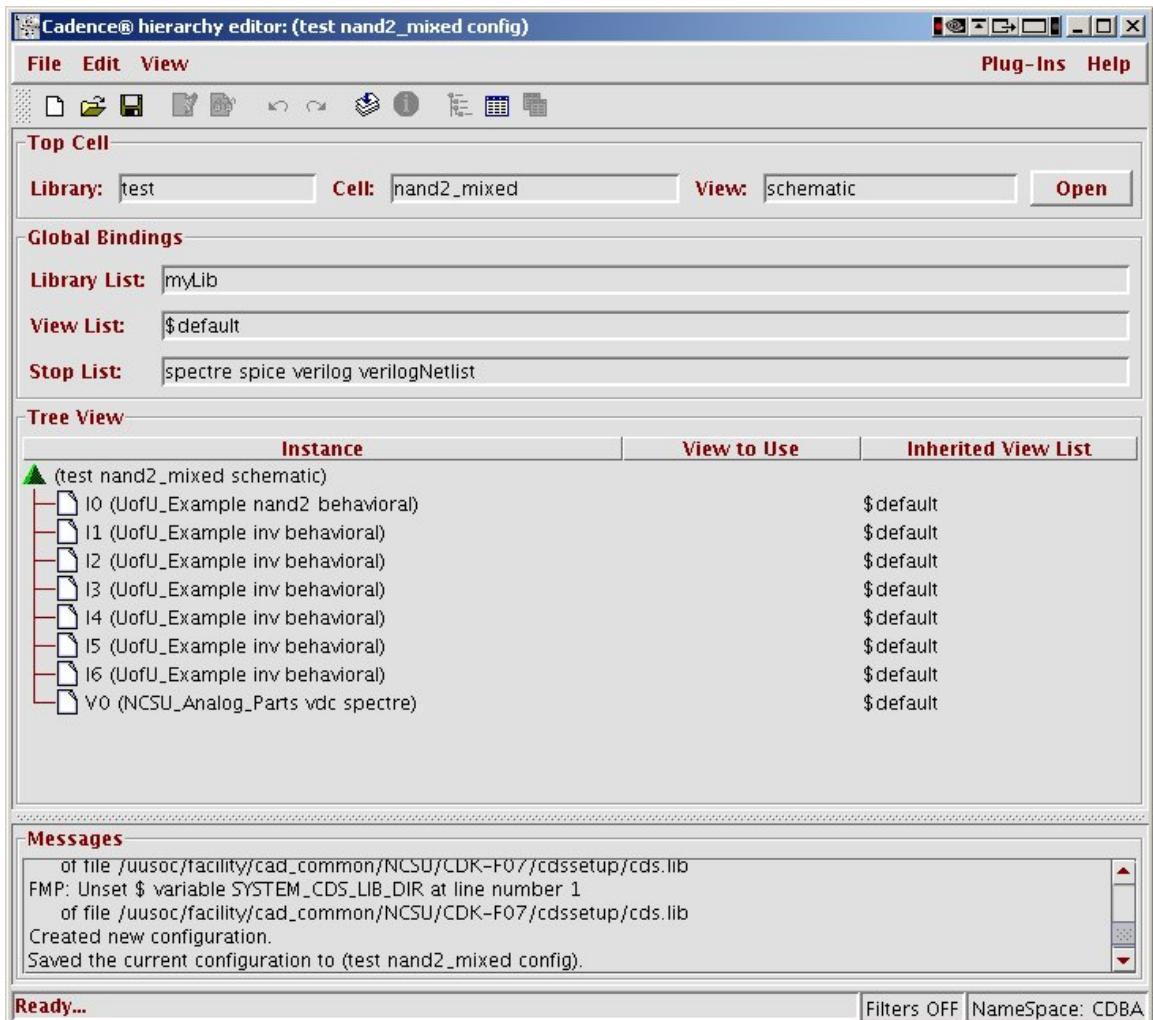


Figure 6.16: Test schematic for the mixed-mode NAND (DUT) simulation

components simulated by **Spectre** we have to make sure that the mixed-mode simulator can find the interface elements that it wants to put between the digital and analog portions of the circuit. If you have just created the **config** view you'll need to close the view, and re-open the same **config** view from the library manager. This time if you select **yes** for both the **config** and **schematic** views you'll get both windows at the same time. Switch the **Hierarchy Editor** to **tree** view, and in the schematic window select **Tools** → **Mixed Signal Ops** to get a few new menu choices in the Composer window.

In the Composer schematic window select the **Mixed-Signal** → **Interface Elements** → **Default Options** menu choice to get the dialog box. In this box, update the **Default IE Library Name** to be **NCSU\_Analog\_Parts** (see Figure 6.18). Now you can select **Mixed-Signal** → **Interface Elements** → **Library** to change the default characteristics of the interface elements. For example, the **output d2a** devices have rising and falling slopes, and high and low voltages defined so that they know how to take digital signals and generate analog versions. For the **input** devices you can set at what analog voltage the **a2d** thinks the value is a logic 0 or a logic 1, and the max amount of time an **a2d** can remain between a logic 1 and logic 0 before it reports an X to the digital simulator. The defaults will probably work fine, but this is where you change things if you want to. Figures 6.19

Figure 6.17: Mixed mode **config** view for **mixed-nand**

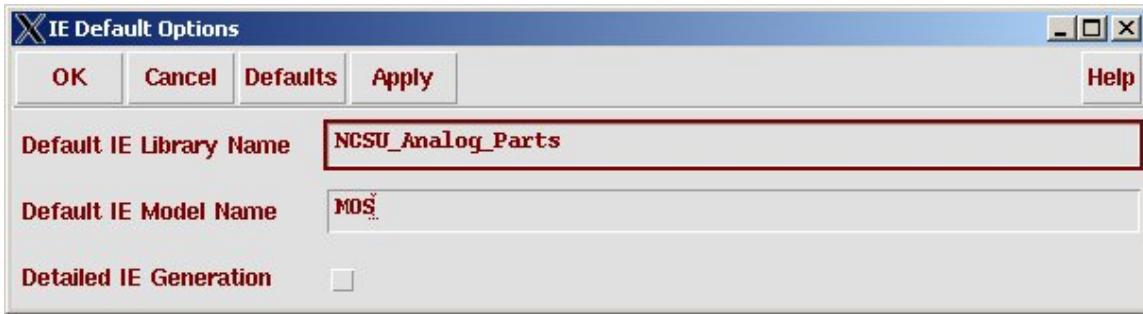


Figure 6.18: Interface Library Dialog Box

and Figure 6.20 show the default values.

Once you have things set up, you can change the partitioning by changing how the **config** view looks at each circuit. In this case I'll change the inverters closest to the DUT (the NAND gate) to use analog simulation, and choose analog simulation for DUT itself machine. I'll be able to apply inputs to the test circuit using Verilog to drive the **a** and **b** signals. I can see the analog output on **aout** and the digital output as the output of the second output inverter (which is simulated with Verilog) **dout**.

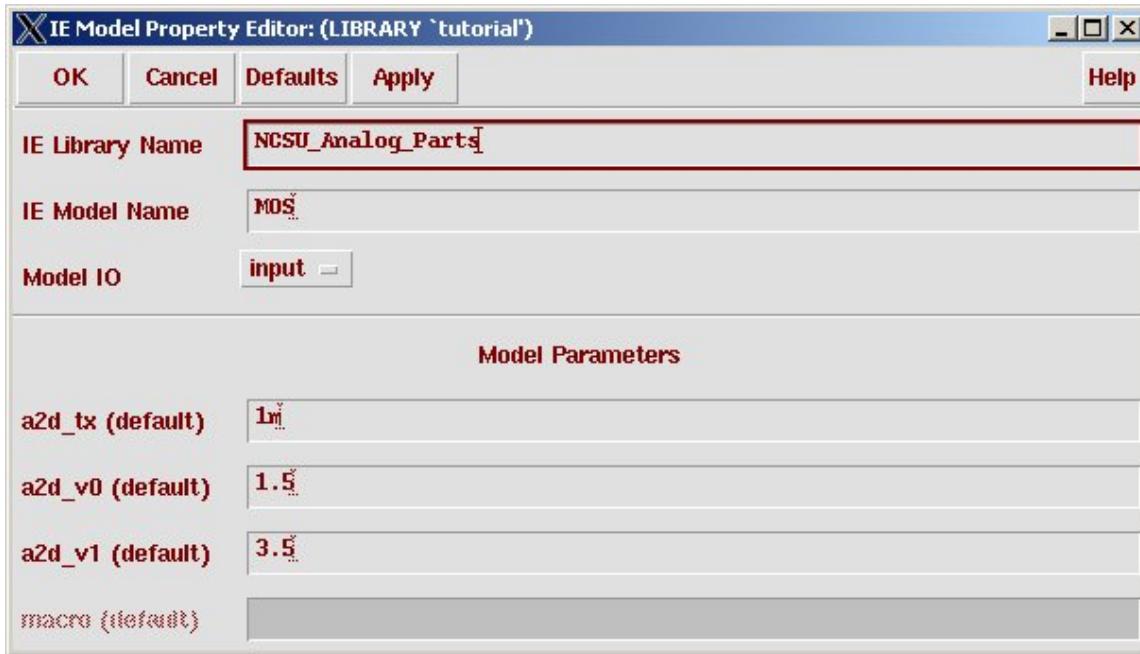
*Note that if you select the inverters in the config view they are also highlighted in the schematic view.*

In my schematic, inverters I1, I4 and I5 are the ones closest to the DUT (NAND). I'll change their **View to Use** to **cmos.sch** (using a right-mouse and **Set Instance View**) so that the netlister will expand them, and change their inherited view list to be **spectre** so that the netlister will stop only when it finds that view (which is an analog view according to the Analog Stop List). The **View to Use** is updated by right-clicking the component and using the menu. The **Inherited View List** is updated by clicking on the field and then typing **spectre** by hand.

I'll leave the other inverters as is (using the **behavioral** view) so that they'll be simulated by the Verilog simulator. For the **nand2** DUT block, I'll change it to a **analog-extracted** view, and **spectre** for the **Inherited View List** so that it's also simulated with **spectre**, but with the extracted parasitics. Note that if the DUT had more structure, that is it was composed of a hierarchy of **schematic** and **cmos.sch** views I would need **schematic**, **cmos.sch** and **spectre** in the **Inherited View List** so that the netlister will continue to expand the schematic views until it finally gets to a **spectre** stopping view. The config now look like Figure 6.21. Update the view using **View → Update**, and save the **config** view before moving on.

You can now go back to the schematic view and click on **mixed-Signal → Display Partition → All Active** to see the partitioning that you've spec-

Figure 6.19: **d2a** interface element parameters

Figure 6.20: **a2d** interface element parameters

ified. As you can see in the schematic (Figure 6.22), the components highlighted in orange will be simulated with Verilog and the components highlighted in red will be simulated with Spectre. The mixed-signal support process has added **d2a** and **a2d** components between the simulation domains for you (shown in blue in the schematic view).

Now in the Composer schematic you can select **Tools** → **Analog Environment** to start the mixed mode simulation. This will look just like the dialog box in Figure 6.5 but will have **config** as the **View** to simulate. Before you start the rest of the analog simulation process, you need to change which simulator will be used. Select **Setup** → **Simulator/Directory/Host** and change the **Simulator** to **spectreVerilog** so that the analog environment knows that you're going to use both analog and Verilog simulators. If you're using different transistor models than the class defaults make sure you update the **model path** with the **Setup** → **Model Libraries** menu. Also, you may want to check in the **Setup** → **Environment** menu, in the **Verilog Netlist Option** sub-form, make sure you're using **Verimix** as the **Generate Test Fixture Template** type and that **Drop Port Range**, **Netlist SwitchRC** and **Preserve Buses** are all selected. These should be selected by default but it's worth a minute to check.

Now you need to edit your digital testbench code (as you did in Chap-

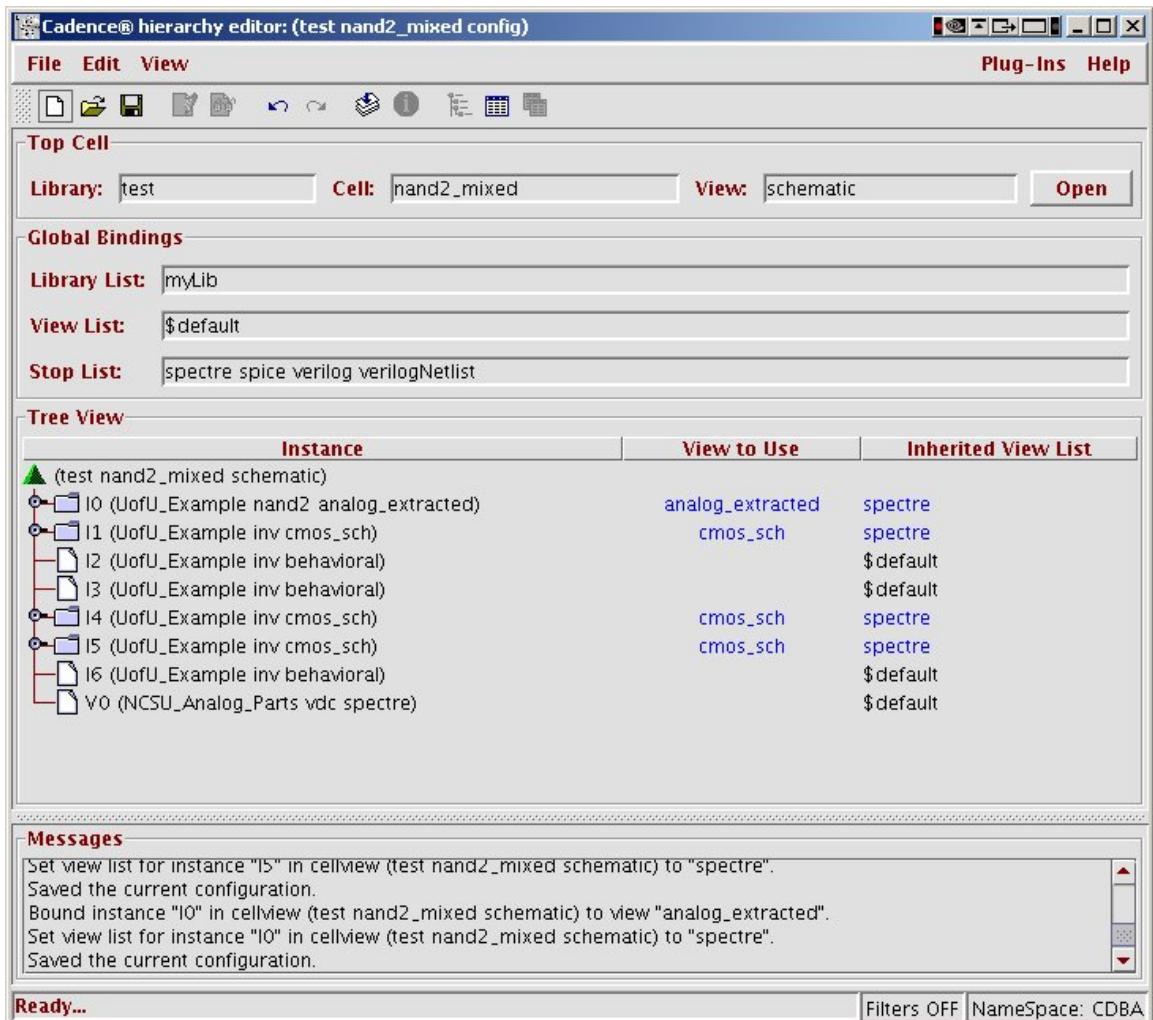


Figure 6.21: Mixed-mode config view with analog/Verilog partitioning

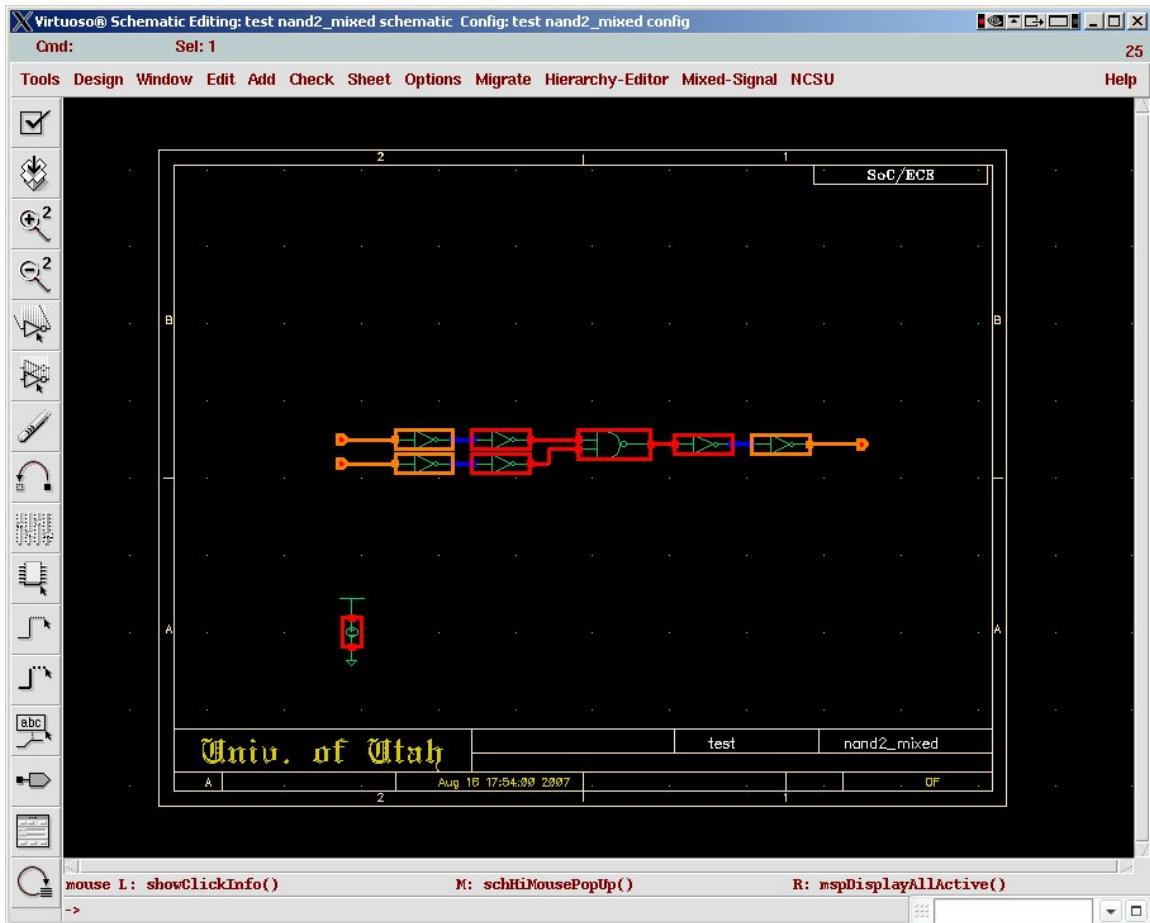


Figure 6.22: The **mixed-test** schematic showing analog/digital partitioning

```

// Vermix stimulus file.
// Default verimix stimulus.

initial
begin

    a = 1'b0;
    b = 1'b0;

#10 $display("ab = %b%b, out = %b", a, b, dout);
if (dout != 1) $display("Error - that's wrong!");

a=1;
#10 $display("ab = %b%b, out = %b", a, b, dout);
if (dout != 1) $display("Error - that's wrong!");

b=1;
#10 $display("ab = %b%b, out = %b", a, b, dout);
if (dout != 0) $display("Error - that's wrong!");

a=0;
#10 $display("ab = %b%b, out = %b", a, b, dout);
if (dout != 1) $display("Error - that's wrong!");

end

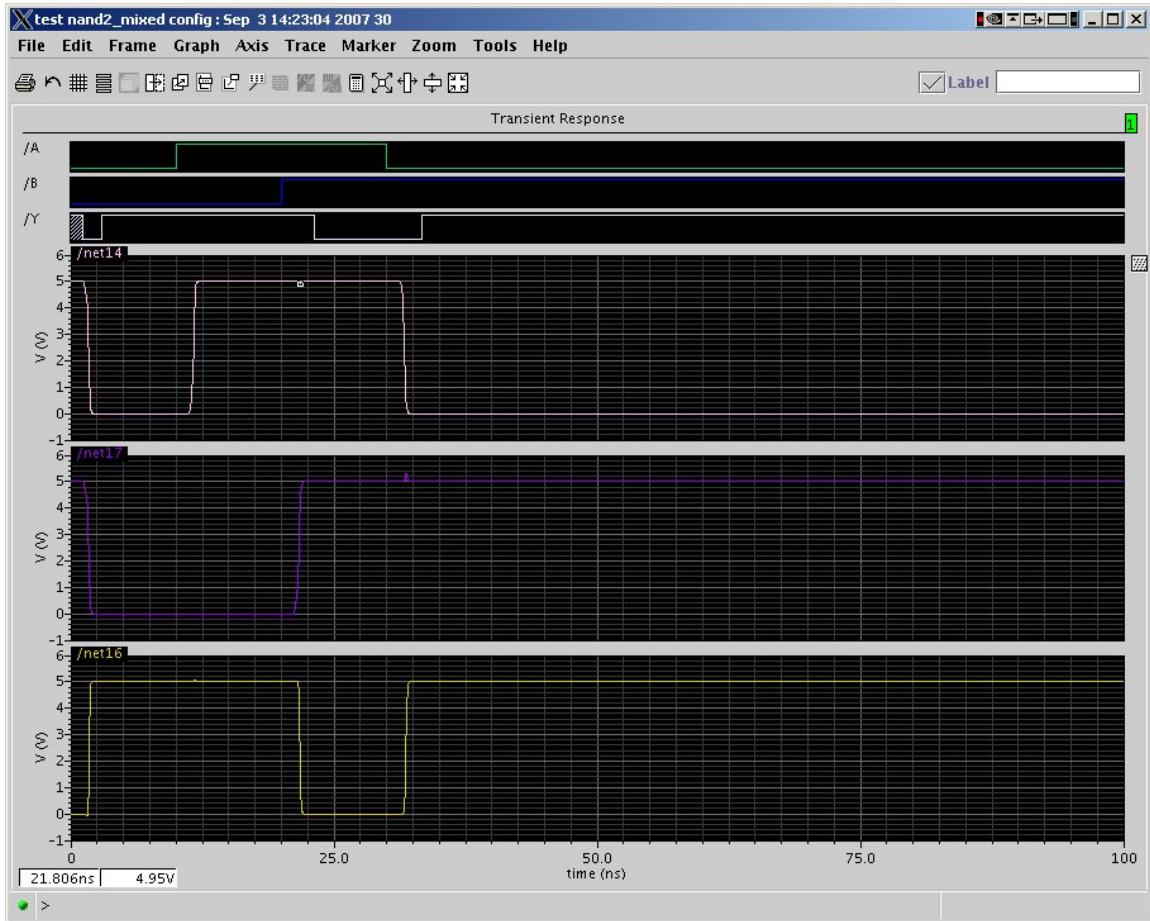
```

Figure 6.23: The digital testbench for the mixed-nand simulation

ter 4). Select **Setup** → **Stimului** → **Digital** to make your Verilog Testbench. My example testbench is shown in Figure 6.23. It looks just like a pure Verilog testbench, but the “guts” of the simulation of the DUT (the NAND gate) will have been done in the **Spectre** analog simulator.

This testbench includes Verilog delays that in total last for 40ns so the transient analysis in **Spectre** had better go for at least that long. I’ll choose a transient analysis of 100ns for this example just to be safe. I’ll choose analog signals to be plotted by selecting them on the schematic just like in the previous cases, but for fun I’ll also select the digital inputs and outputs. Then I can start the simulation. After switching the display to **Strip Chart Mode** you can see in Figure 6.24 that the digital waveforms are output at the top, and the analog waveforms are plotted in the bottom three strips. By clicking and dragging the digital waveforms over the top of the analog waveforms I can also generate a waveform like that in Figure 6.25 where the digital waveforms are superimposed on the analog waveforms. In this Figure you can see that the analog inputs to the DUT are delayed because they have to go through two inverters, and the analog output is delayed through a pair of inverters before it becomes the digital output.

The next question you might have is: Where did the **\$display** outputs

Figure 6.24: Results of the mixed-mode simulation of **mixed-test**

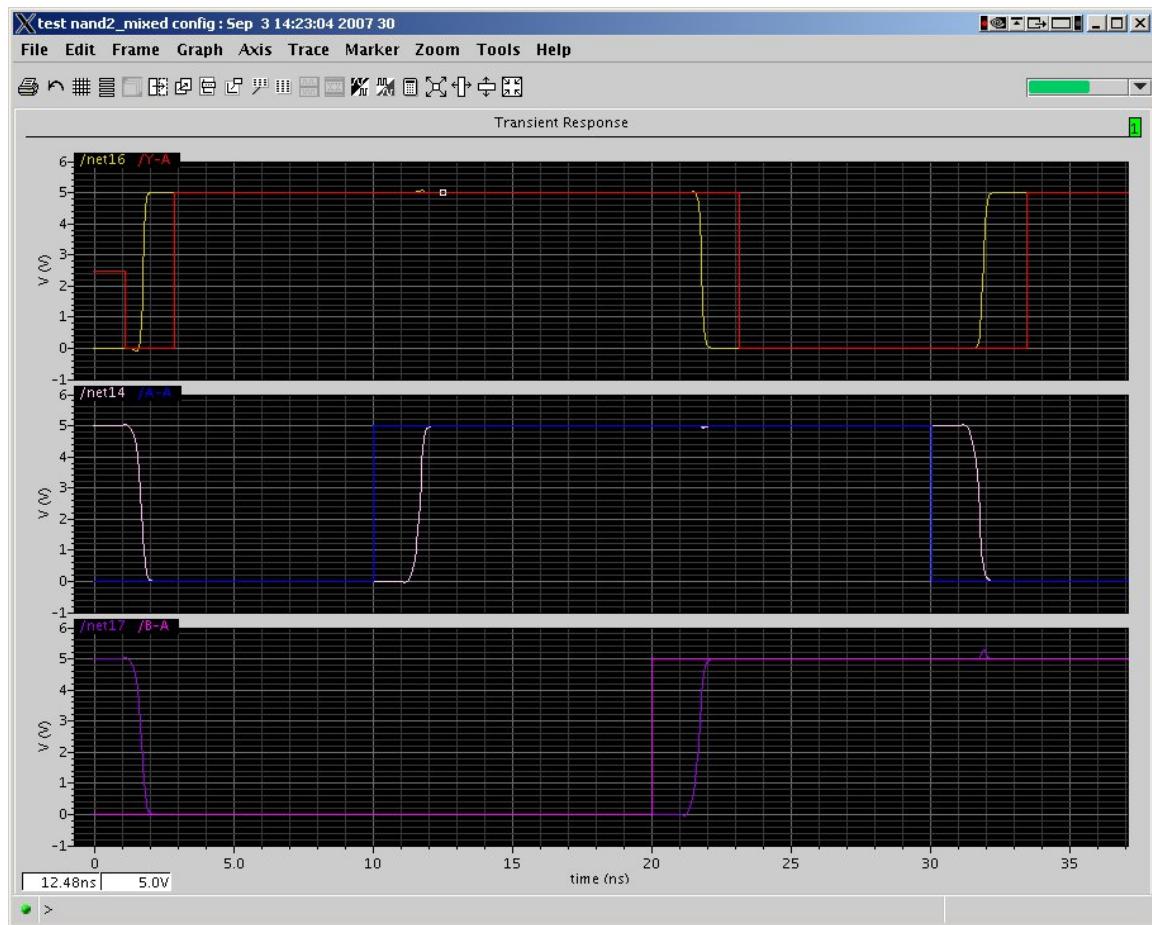


Figure 6.25: Rearranged results of the mixed-mode simulation

```

Switching from DC to transient. VERILOG time 0 (units of 100ps) corresponds to spectre time 0.

Message! At the end of DC initialization the logic values
of the following ports are X (unknown):

    net16
    net18
                                [Mixed_Sig]
    "IE.verimix", 4: ...
ab = 00, out = 1
ab = 10, out = 1
ab = 11, out = 0
ab = 01, out = 1
Verilog/spectre Interface: 165 messages sent, 167 messages received.
0 simulation events (use +profile or +listcounts option to count) + 29 accelerated events
CPU time: 0.0 secs to compile + 0.0 secs to link + 3.6 secs in simulation
End of Tool:    VERILOG-XL      05.81.001-p   Aug 23, 2006 10:58:39

```

Figure 6.26: **\$display** output from the **mixed-test** simulation

go? They went into a very distant file. Specifically, they went into:

<your-cadence-dir>/simulation/<schName>/spectreVerilog/config/netlist/digital/verilog.log.

In this directory path <your-cadence-dir> is the name of the directory from which you started Cadence, and <schName> is the name of the schematic that you were simulating (**mixed-test** in this case). If you look into that file you'll see your **\$display** output near the bottom. For this example the output is seen in Figure 6.26.

That's not very convenient, but at least it is there if you need to look for it. You can use this to trap errors using **if** statements the same way you would in a normal Verilog-XL simulation, just make sure that you're testing digital values against digital values. If you try to look at an analog value in the Verilog testbench it's likely to be **Z** or **X**.

A more convenient solution is to specify a file for the outputs of the Verilog testbench. For example, a testbench for the **mixed-test** simulation could be written as in Figure 6.27. In this example I've opened a file for output using the **\$fopen** Verilog command and then written my **\$display** outputs to that file. The output is shown in Figure 6.28

### Final Words about Mixed Mode Simulation

Note that mixed-mode simulation is a little touchy and delicate. There are lots of ways to break it. But, when it works it's a great way to simulate larger circuits with **Spectre** and still provide digital input streams using a Verilog testbench. It is the recommended way to drive complex circuits so that you can write testbenches in Verilog, and you can also write self-checking analog simulations by including **if** statements in your testbench that check

```

// Vermix stimulus file.
// Default verimix stimulus.

integer file; // declare the file descriptor first
initial
begin
    file = $fopen( "/home/elb/IC_CAD/cadencetest/testout.txt" );
    a = 1'b0;
    b = 1'b0;

$fdisplay(file, "Starting mixed-test simulation of NAND");
$fdisplay(file, "using digital inputs to an analog simulation");

#10 $fdisplay(file, "ab = %b%b, out = %b", a, b, dout);
if (dout != 1) $fdisplay(file, "Error - that's wrong!");

a=1;
#10 $fdisplay(file, "ab = %b%b, out = %b", a, b, dout);
if (dout != 1) $fdisplay(file, "Error - that's wrong!");

b=1;
#10 $fdisplay(file, "ab = %b%b, out = %b", a, b, dout);
if (dout != 0) $fdisplay(file, "Error - that's wrong!");

a=0;
#10 $fdisplay(file, "ab = %b%b, out = %b", a, b, dout);
if (dout != 1) $fdisplay(file, "Error - that's wrong!");

end

```

Figure 6.27: **mixed-test** testbench file with file I/O

```

Starting mixed-test simulation of NAND
using digital inputs to an analog simulation
ab = 00, out = 1
ab = 10, out = 1
ab = 11, out = 0
ab = 01, out = 1

```

Figure 6.28: **mixed-test** testbench file with file I/O

the digital versions of your outputs.

## 6.5 DC Simulation

The **Spectre** simulations that have been described in this Chapter until now have been transient analysis. That is, you are applying a time-varying signal to the circuit and you're getting a simulation of the time-varying output of the circuit in response to that input. This is valuable stuff, and is generally what digital designers are mostly concerned about since this relates to the operating behavior of the digital circuit over time. However, when we're looking at the behavior of the devices themselves, we may want to do a *DC analysis* that looks at the steady state instead of the time-varying state. For example, Figure 2.7 in your text [1] is a plot of steady state behavior of an nmos transistor mapping the  $I_{ds}$  current as  $V_{ds}$  is changed. In fact, there are five different DC analyses in this figure. Each of the five curves is a DC analysis that holds the  $V_{gs}$  at a fixed level, then sweeps the  $V_{ds}$  from 0v to 5v and plots the resulting  $I_{ds}$ .

To run the same DC analysis in **Spectre**, start by creating a schematic with a single **nmos** device and two **vdc** supply nodes from either the **NCSU\_Analog\_Parts** or the **UofU\_Analog\_Parts** libraries. The Composer version looks like Figure 6.29. To perform a DC analysis on this circuit you would fix the gate voltage (it's fixed at 3v in this Figure 6.29), and then do a DC sweep of the  $V_{ds}$  by sweeping the voltage of the other **vdc** component from 0 to 5 volts. To see the  $I_{ds}$  value you would plot the current at the transistor's drain instead of the voltage of a node.

Now you can follow the following steps to complete the DC analysis of this circuit:

1. Open the schematic in Composer. Set the voltages of the **vdc** components as they are in Figure 6.29.
2. Select **Tools → Analog Environment**. When you choose the **analysis type** select **DC** instead of **tran**. This changes the **Choosing Analysis** dialog box to **DC**.
3. Choose your sweep variable for the DC analysis. In this case we're going to sweep the voltage of the **vdc** component that's connected to the **nmos** drain so choose **Component Parameter** so that we can select the component that we want to use for the sweep.
4. This changes the dialog box so that you can tell it which component you're going to select, which parameter of the component you want to vary, and what the sweep range is of that parameter. Click on **Select**

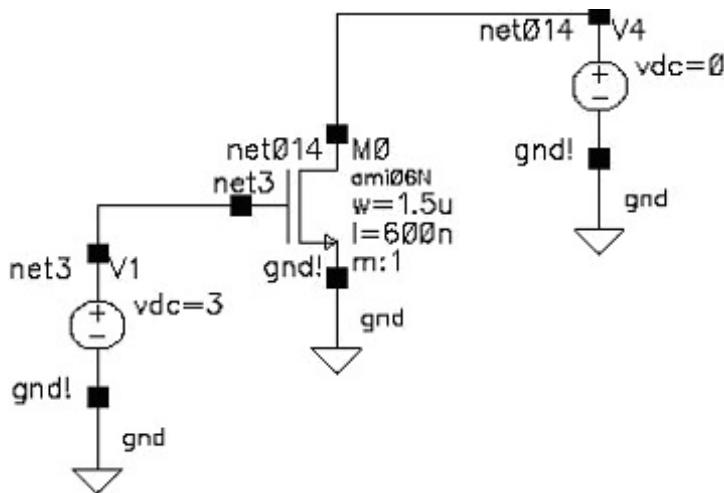


Figure 6.29: Simple circuit for DC analysis (**schematic** view)

**Component** so that you can select the **vdc** connected to the transistor drain by clicking on it in the schematic.

5. When you select (click on) the **vdc** connected to the transistor drain, you get another box (Figure 6.30) that asks you to select the component parameter that you want to sweep. We want to sweep the **dc voltage**. Click on that and then OK. You'll see that the component name (**/V1** in my circuit) and the parameter name (**dc**) have been added to the **Choosing Analysis** dialog box which now looks like that in Figure 6.31.
6. Now enter the Sweep Range. Start from 0 and stop at 5. This will set up the DC analysis to sweep  $V_{ds}$  from 0v to 5v during the analysis. Click OK.
7. Back in the Analog Environment you need to select the outputs to be plotted. Use the same **Select on Schematic** choice that you've used before, but instead of selecting a wire, select the red square on the drain of the **nmos** transistor (the top leg of transistor **M0** in Figure 6.29 - the one without the arrow). This will cause a circle to be put around the drain, and the drain current will be plotted. This shows up as **M0/D** for the output in my circuit (**M0** is the transistor identifier, and **D** is the drain current). The **Analog Environment** dialog box should look like Figure 6.32 when you're done.
8. Now run the simulation. You should get a waveform output like Figure 6.33 that has a single  $I_{ds}$  curve for a 3v gate voltage ( $V_{gs}$ ) and a DC sweep from 0v to 5v on  $V_{ds}$ .

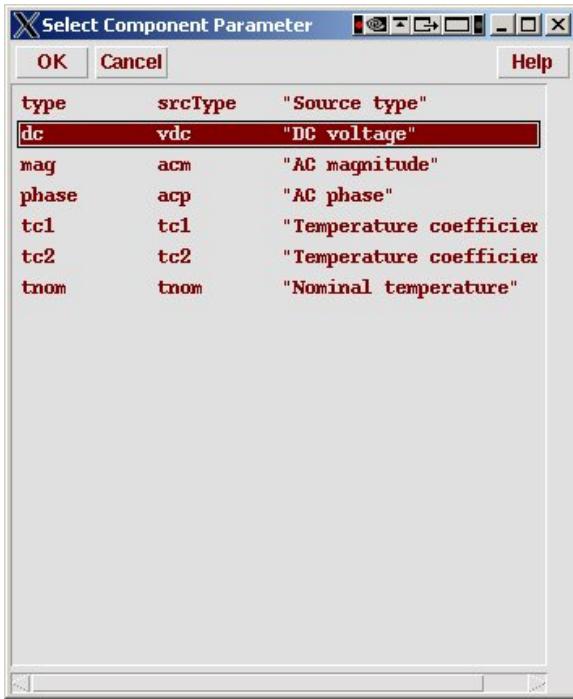


Figure 6.30: Component parameter selection dialog box for DC analysis

### 6.5.1 Parametric Simulation

To generate a graph like Figure 2.7 in the book, you need to run five DC analyses and plot them all together. Fortunately, there is an easy way to run all five in Spectre. It's called *parametric simulation*. To get a parametric simulation of all five curves in Figure 2.7, you need to run five different simulations, each one with a different  $V_{gs}$ . In the book they are simulating a 180nm process with a 1.8v power supply. We'll be using our  $0.6\mu$  (600nm) process with a 5v supply, so we'll sweep from 1v to 5v ( $V_{gs} = 1\text{v}, 2\text{v}, 3\text{v}, 4\text{v}$  and  $5\text{v}$ ). Here's how to set up that parametric simulation:

1. Quit the Analog Environment and go back to editing your schematic.
2. Edit the properties of the **vdc** component connected to the gate of the transistor (use **q** to get the properties dialog box). Change the DC Voltage value from **3 V** to **foo V**. What you've done is change the fixed value of 3 into a variable named **foo**.
3. Start up the Analog Environment and set up the same DC analysis as before (DC analysis with **Component Parameter** as the Sweep

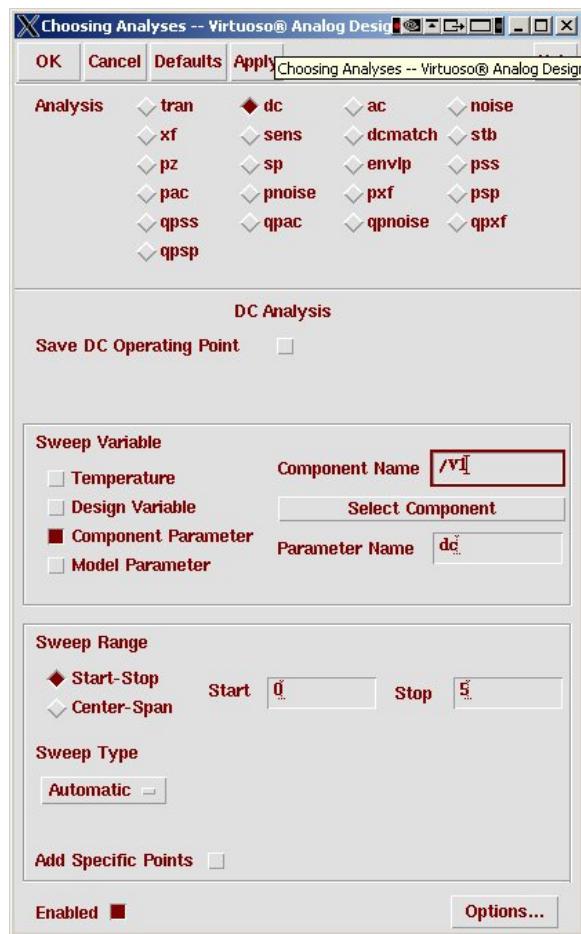


Figure 6.31: DC analysis dialog box

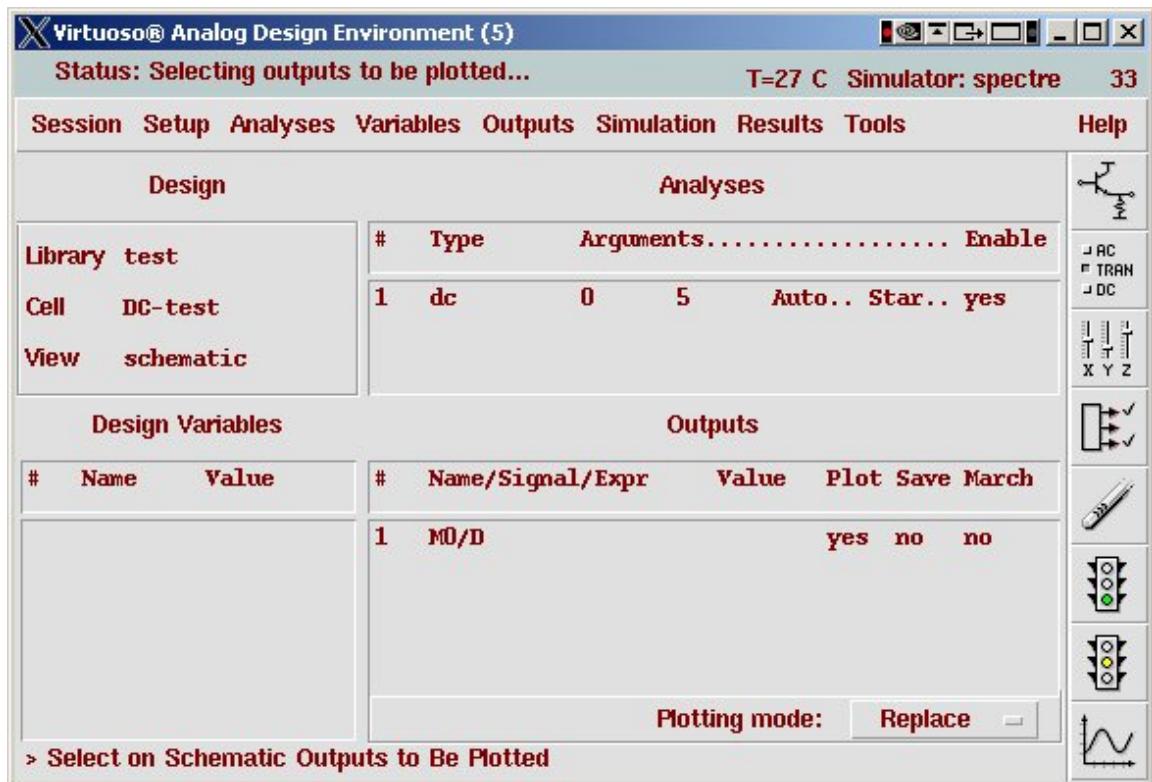


Figure 6.32: Analog Environment dialog box for DC analysis

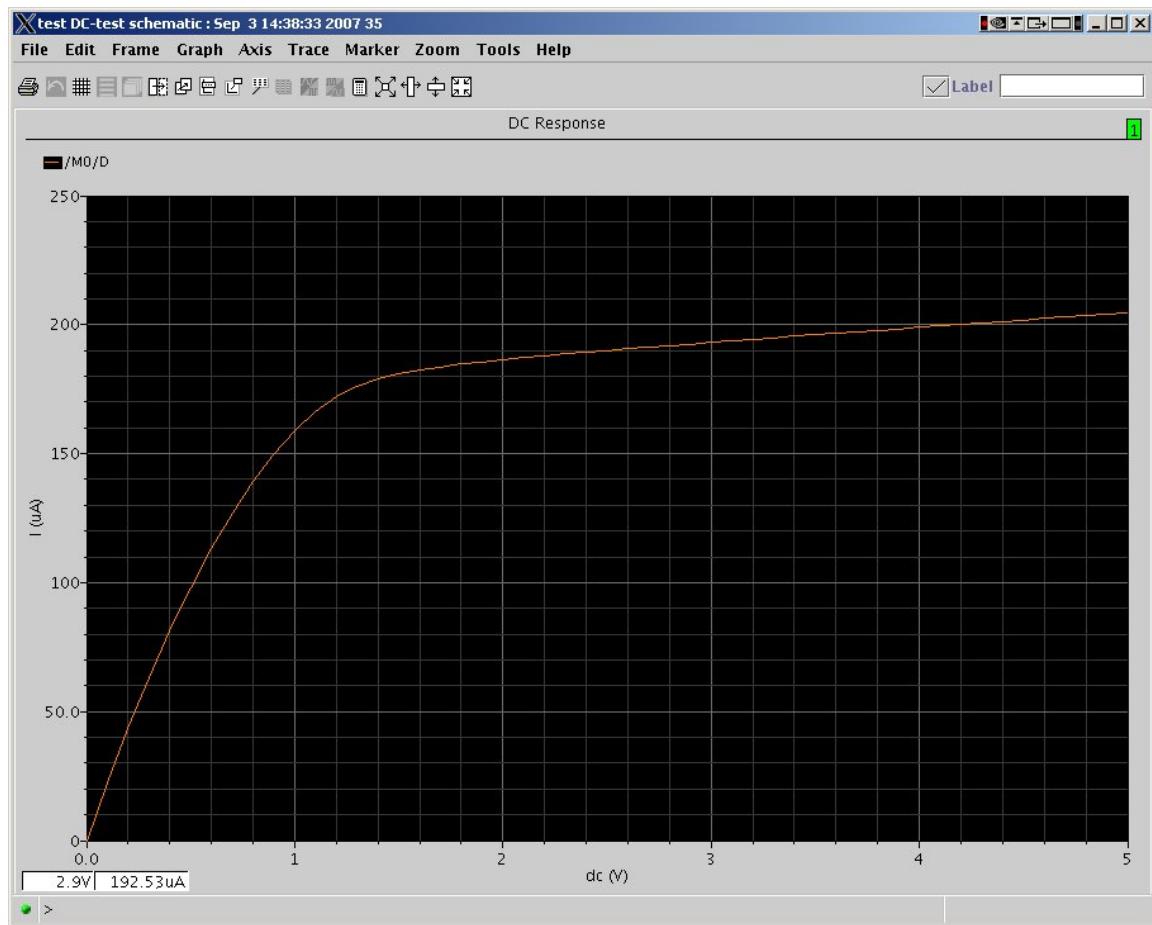


Figure 6.33: DC analysis output waveform for a single set of parameters

**Variable**, the **vdc** connected to the drain as the component, and **dc voltage** as the parameter name. Sweep the voltage from 0 to 5 volts).

4. Select the drain current as the output to be plotted, the same as before. For parametric simulation it's important that the selected outputs to be plotted show up in the **Analog Environment** control window as both **plot** and **save** so that the waveform viewer can see the results from all the runs. If **save** is not set you can set it by double-clicking the output name in the **Outputs** pane of the window.
5. Now comes the tricky part. You have the  $V_{gs}$  voltage defined as a variable **foo**. You need to set that variable to some value. You could set it to a fixed value by selecting **Variables** → **Copy From Cellview** to get **foo** into the **Design Variables** pane, and then click on that design variable to set the value to whatever you want it to be (3 for example). Or, you can set things up so that all five DC analyses are run with **foo** set to a different voltage on each run.
6. First select **Variables** → **Copy From Cellview** to get the **foo** variable into the **Design Variables** pane in the **Analog Environment** dialog box.
7. Now select **Tools** → **Parametric Analysis**. This pops up a **Parametric Analysis** window.
8. In this window enter **foo** in the **Variable Name** slot, and range the value of the variable from 1 to 5 with 5 total steps. The dialog box looks like that in Figure 6.34.
9. Now choose **Analysis** → **Start** and all five simulations will run, and the results will be plotted in a single waveform window. The result looks like Figure 6.35. Note that the parameters that were used on each run (the value of **foo** in this case) are shown in the legend at the top of the graph.

This parametric analysis feature is very slick. You can name all sorts of parameters as variables in your schematics, and use the parametric analysis feature to simulate over a range of values. In this case we did five DC analyses: each one had a different fixed  $V_{gs}$  and swept the  $V_{ds}$  from 0v to 5v and plotted the resulting drain current of the nmos device. But, you can imaging using this technique to run all sorts of simulations where you want to vary parameters over multiple simulation runs. The basic technique of setting parameters in the components using variables and then using parametric analysis to do multiple simulation runs while varying the values of those variables is the same.

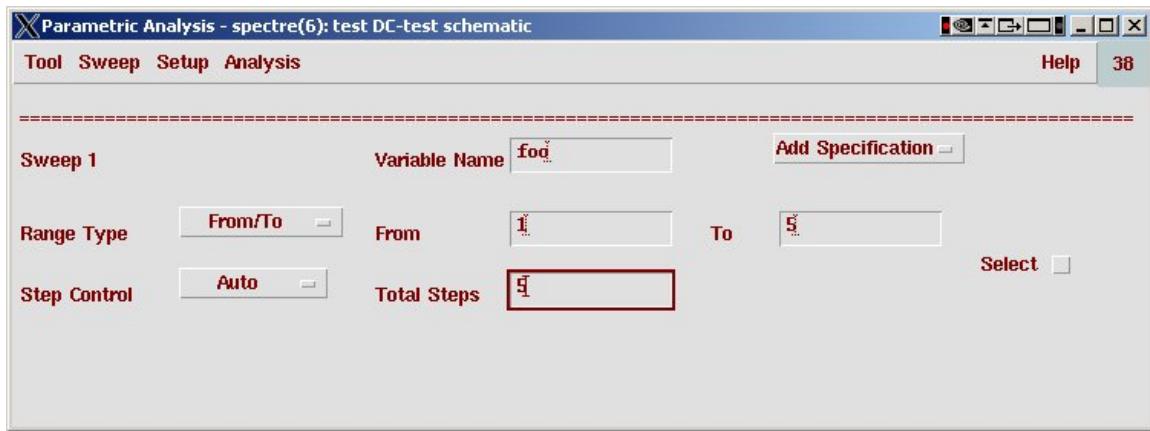


Figure 6.34: Dialog to set variable parameters for parametric simulation

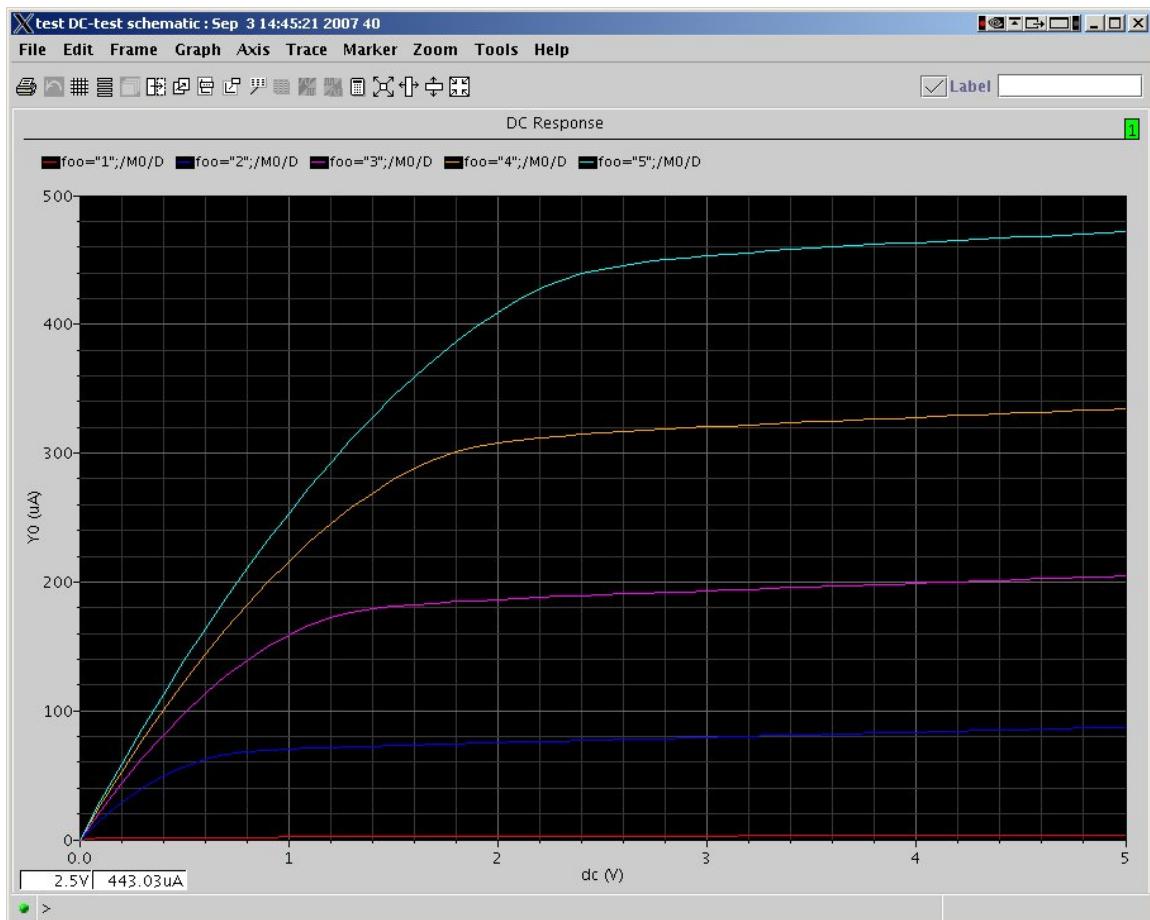


Figure 6.35: Output of parametric DC simulation with five curves

In particular, you can use the parametric simulation for **transient** simulation as well as for **DC** simulation. You could, for example, put a variable in the **capacitance** value of a load capacitor and do multiple transient simulations with varying amounts of output load capacitance. Or vary the width of transistors and run multiple transient simulations with different transistor sizes. You can vary just about any parameter that can be set in the **properties** box. It's a very powerful technique.

# Chapter 7

## Cell Characterization

**S**TANDARD CELLS must be *characterized* so that they can be understood and used by the various tools in the CAD flow. So far if you've designed schematic and layout views of your cells the tools understand the transistor netlist and the layout of the cells, but they don't understand the function of the cells in a way that the tools can use, and other information such as the input load, speed, and power of the cell has not been extracted in a way that the tools can understand. The synthesis tools coming up in Chapter 8, for example, need to know the logic function of the cell, the load that the cell input will present to a signal connecting to it, the speed of the cell under different input slope and output loading conditions, the power that the cell will consume, and the area of the cell in order to do a good job of synthesizing a behavioral description to a collection of standard cells. *Cell characterization* is a process of simulating a standard cell with an analog simulator to extract this information in a way that the other tools can understand. This can be done through specific analog simulation (using SpectreS) whose output you look at to generate the characterization data, or by using a library characterization tool. In this case we'll use SignalStorm from Cadence.

### 7.1 Liberty file format

We need to encode the cell characterization data in a standard format called **liberty** format which usually uses a **.lib** file extension. **Liberty** format is an ASCII file that describes a cell's characterized data in a standard way. This file is used both by the synthesis tools described in Chapter 8 and by the place and route tools in Chapter 10. The general form of a **liberty** file is shown in Figure 7.1. At this level of detail it simply describes the overall structure of the file. Each field has lots of detail that you can add by hand, or

```
/* General Syntax of a Technology Library */
library (nameoflibrary) {
... /* Library level simple and complex attributes */
... /* Library level group statements */
... /* Default attributes */
... /* Scaling Factors for delay calculation */

/* Cell definitions */

cell (cell_name) {
... /* cell level simple attributes */

/* pin groups within the cell */
pin(pin_name) {
... /* pin level simple attributes */

/* timing group within the pin level */
timing(){
... /* timing level simple attributes */
} /* end of timing */

... /* additional timing groups */

} /* end of pin */

... /* more pin descriptions */

} /* end of cell */

... /* more cells */

} /* end of library */
```

Figure 7.1: General format of a **liberty (.lib)** file

you can have **SignalStorm** generate for you. In general the **SignalStorm** approach is much easier if everything works. For some out of the ordinary cells, however, you may have to resort to your own simulations.

The **Liberty** file format is, as you might guess, very complex with huge numbers of special statements that can describe all sorts of parameters that would be relevant to the different CAD tools that use this format to get information about the standard cells in the library. This chapter will touch on the most important from our flow's point of view, and give a number of examples, but in this case the full file format is too large and complex to include. This information should be enough to show you how to generate working **.lib** files for our flow. The format is described in full detail in documentation from **Synopsys**.

An example of the header information generated from a **SignalStorm**

characterization run is shown in Figure 7.2. This is all data above the *cell definitions* in the **.lib** file. Later we will see how to add additional information to this technology header to generate, for example, a simple model of wire loads that estimates the delay that will be caused by the RC delays in the wiring of the circuit once it is physically placed and routed. For now we can look at the information in the header and see what's going on. The **delay\_model** says that the characterized delays will be described in **table\_lookup** format. This means that the delays will be described as a table of delays simulated with different input and output conditions. In our characterizations this will be the input slope on one dimension of the table and output capacitive load on the other axis. The synthesis tool can use this information, and information about the connection of the cells that it is synthesizing, to estimate delays with reasonable accuracy (assuming that the tables have enough data, and that the data is accurate of course). There are other delay models possible, but **table\_lookup** is by far the most widely used.

The **in\_place\_swap\_mode** tells the tool that uses this file that cells whose **footprints** match can be used as replacements for each other. This allows us to have, for example, inverters with the same logic function but with different output drives swapped for each other as required. To prevent this swapping you would set this mode to be **no\_swapping**.

The **unit\_attributes** should be fairly self-explanatory. They define the default units that various electrical parameters will be described in so that simple numbers can be used in the cell descriptions with the units assumed to be these defaults. The **thresholds** define where in the waveform **slews** and **delays** are computed. In this case the **rise** and **fall** times are computed at the 30-70% points in the input or output waveform. That is, delays are defined as being measured from 30% of the total rise or fall on the input to 70% of the rise or fall of the output. The percentages are based on vdd. The default operating conditions are set which are actually only really useful if multiple operating conditions are characterized (usually typical, max, and min) but, of course, that increases characterization time. Finally the format of the **lu\_table\_template** is defined for both delay and power computations. These are the *look up tables* that will contain the actual timing and power data. In this case we define 5x5 tables with input slope (**input\_net\_transition**) on the first axis and output capacitance (**total\_output\_net\_capacitance**) on the second axis. The actual value of the input slope (in ns) and the output loading (in pF) will be defined for each look up table defined in the cell definition section.

*Delays are measured with these percentages because measuring from the 50% point can result in a negative delay. This can happen if a gate has high gain so that a small change on the input causes the output to change rapidly. By the time the input has risen to 50%, the output has already changed.*

```
library(foo) {

    delay_model : table_lookup;
    in_place_swap_mode : match_footprint;

    /* unit attributes */
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1uA";
    pulling_resistance_unit : "1kohm";
    leakage_power_unit : "1nW";
    capacitive_load_unit (1,PF);

    slew_upper_threshold_pct_rise : 80;
    slew_lower_threshold_pct_rise : 20;
    slew_upper_threshold_pct_fall : 80;
    slew_lower_threshold_pct_fall : 20;
    input_threshold_pct_rise : 30;
    input_threshold_pct_fall : 70;
    output_threshold_pct_rise : 70;
    output_threshold_pct_fall : 30;
    nom_process : 1;
    nom_voltage : 5;
    nom_temperature : 25;
    operating_conditions ( typical ) {
        process : 1;
        voltage : 5;
        temperature : 25;
    }
    default_operating_conditions : typical;

    lu_table_template(delay_template_5x5) {
        variable_1 : input_net_transition;
        variable_2 : total_output_net_capacitance;
        index_1 ("1000.0, 1001.0, 1002.0, 1003.0, 1004.0");
        index_2 ("1000.0, 1001.0, 1002.0, 1003.0, 1004.0");
    }
    power_lut_template(energy_template_5x5) {
        variable_1 : input_transition_time;
        variable_2 : total_output_net_capacitance;
        index_1 ("1000.0, 1001.0, 1002.0, 1003.0, 1004.0");
        index_2 ("1000.0, 1001.0, 1002.0, 1003.0, 1004.0");
    }
}
```

Figure 7.2: Example technology header for a **liberty (.lib)** file

## Wire Load Models

An increasingly important contributor to the overall delay in a circuit is the delay in the wires. For a simple model of wire delay it's not too hard to compute the wire's contribution to delay once the wiring is actually known (i.e. after place and route). But, if you're using information in the **.lib** file to estimate performance before you have physical placement and wiring, how do you estimate the wire delays? One way is to use an educated guess at the average length of a wire for a broad class of circuits of different sizes. It turns out that, not surprisingly, average wires tend to get longer the larger the circuit is. When a program like **Synopsys** generates a circuit from a behavioral description, it has information from the **.lib** file about the size of the cells, so it has a good estimate of what the final size of the circuit might be. Using that circuit size, a *wire load model* can be used to estimate (roughly) what the wire delay will be for the final circuit.

A good wire load model might come from a company where many designs have been done using a particular cell library, and statistics have been kept on the average wire length for circuits build using those libraries. We don't have such good statistics for our library so we'll have to guess. The wire load model in Figures 7.3 and 7.3 define a wire load model derived from some rough guesses about wire length and some information about the resistance and capacitance of general wiring in our  $0.6\mu$  CMOS process. These models would be placed in the **.lib** file right after the technology information in Figure 7.2.

### 7.1.1 Combinational Cell Definition

The cell definition (with some details deleted so it fits on a page) that was derived from the **SignalStorm** characterization for an inverter is shown in Figure 7.5. You can see that the name of the cell is **INVX1** and the overall attributes are defined. Then each pin is described. Pin **A** is an input pin so its attributes are all related to the capacitance that this pin presents to any signal that connects to it. Remember the default units earlier in the file so these numbers are in units of pF.

The output pin **Y** is much more interesting. Delays occur at the output pins with respect to changes on the input pins and hence only output pins have timing information for combinational cells. Because it doesn't feed back into the cell the capacitance that output pin **Y** sees because of this cell is 0. The **function** of the output defines the logical behavior of the cell. In this case the function is **!A** or the inverse of the **A** input. The **timing()** section defines how the output timing relates to changes on each input pin. There's only one input pin in this case but for a gate with multiple inputs the

```
/* Because the cell area is in units of square microns, all the      */
/* distance units will be assumed to be in microns or square microns. */

/* fudge = correction factor, routing, placement, etc. */
fudge = 1.0;

/* cap = fudge * cap per micron          */
/* I assume cap is in capacitance units per micron */
/* (remember that our capacitance unit is 1.0pf)   */
cap = fudge * 0.000030; /* .03ff/micron for avg metal      */
res = fudge * 0.00008 /* 80 m-ohm/square, in kohm units */

/* length_top = the length of one side of a square die (in our case, */
/* a 4 TCU die of 2500u on a side of core area) length_10k = the      */
/* length of one side of a block containing 10k gates (I'll assume    */
/* this is a core of a single TCU which is 900u on a side)           */
length_10k = 900;
length_top = 2500.0;

/* sqrt(5000/10000) = .71          */
/* sqrt(2000/10000) = .45 etc     */
length_5k = length_10k * 0.71;
length_2k = length_10k * 0.45;
length_1k = length_10k * 0.32;
length_500 = length_10k * 0.22;
```

Figure 7.3: Define some variables in the **.lib** file for wire load calculations

timing should be characterized based on each **related\_pin** (i.e. each input pin). This means that there would be multiple **timing** sections for each output pin in that case. Within each timing group the **related\_pin** attribute identifies the input pin to which the timing is related.

The pin timing is defined for **cell\_rise**, **rise\_transition**, **cell\_fall** and **fall\_transition**. Recall that the look up tables were defined in the header to have input slope in ns on **index\_1**, and output loading in pF on **index\_2** and you can see the actual values for these parameters on each look up table. These values are chosen by you at the time you do the characterization and are based on the estimated values that you expect the cells to see in an actual circuit. The values in some of the lookup tables have been deleted so that the figure will fit on a page. The power is also calculated for each output pin and for each related input pin.

Cell function is described (as seen in Figure 7.5) as a **function** attribute on each output pin of the cell. The Boolean function of a combinational cell can be described using the syntax in Figure 7.6. This is a standard format called **EQN** format and is used by a number of tools, not just **liberty** files. Description of sequential cells like flip-flops is more complex.

```

wire_load("top") {
    resistance : res ;
    capacitance : cap ;
    area : 1 ; /* i.e. 1 sq micron */
    slope : length_top * .5 ;
    fanout_length(1,2500); /* length */ *
    fanout_length(2,3750); /* length * 1.5 */ *
    fanout_length(3,5000); /* length * 2 */ *
    fanout_length(4,5625); /* length * 2.5 */ *
    fanout_length(5,6250); /* length * 2.5 */ *
    fanout_length(6,6875); /* length * 2.75 */ *
    fanout_length(7,7500); /* length * 3 */ *
}

wire_load("10k") {
    resistance : res ;
    capacitance : cap ;
    area : 1 ;
    slope : length_10k * .5 ;
    fanout_length(1,900); /* length */ *
    fanout_length(2,1350); /* length * 1.5 */ *
    fanout_length(3,1800); /* length * 2 */ *
    fanout_length(4,2025); /* length * 2.5 */ *
    fanout_length(5,2250); /* length * 2.5 */ *
    fanout_length(6,2475); /* length * 2.75 */ *
    fanout_length(7,2700); /* length * 3 */ *
}

wire_load("5k") {
    resistance : res ;
    capacitance : cap ;
    area : 1 ;
    slope : length_5k * .5 ;
    fanout_length(1,639); /* length */ *
    fanout_length(2,959); /* length * 1.5 */ *
    fanout_length(3,1278); /* length * 2 */ *
    fanout_length(4,1439); /* length * 2.5 */ *
    fanout_length(5,1598); /* length * 2.5 */ *
    fanout_length(6,1757); /* length * 2.75 */ *
    fanout_length(7,1917); /* length * 3 */ *
}

/* define how the wire loads are selected based on total circuit area */
wire_load_selection (foo) {
    wire_load_from_area ( 0, 3000000, "5k");
    wire_load_from_area (3000000, 7000000, "10k");
}

default_wire_load_mode : enclosed ;
default_wire_load : "top" ;
default_wire_load_selection : "foo" ;
/* end of wire_load calculation */

```

Figure 7.4: Use the wire load variables to compute wire load models based on wire RC

```

/* -----
 * Design : INVX1 *
 * -----
 */
cell (INVX1) {
    cell_footprint : inv;
    area : 129.6;
    cell_leakage_power : 0.0310651;
    pin(A) {
        direction : input;
        capacitance : 0.0159685;
        rise_capacitance : 0.0159573;
        fall_capacitance : 0.0159685; }
    pin(Y) {
        direction : output;
        capacitance : 0;
        rise_capacitance : 0;
        fall_capacitance : 0;
        max_capacitance : 0.394734;
        function : "(!A)";
        timing() {
            related_pin : "A";
            timing_sense : negative_unate;
            cell_rise(delay_template_5x5) {
                index_1 ("0.06, 0.18, 0.42, 0.6, 1.2");
                index_2 ("0.025, 0.05, 0.1, 0.3, 0.6");
                values ( \
                    "0.147955, 0.218038, 0.359898, 0.922746, 1.76604", \
                    "0.224384, 0.292903, 0.430394, 0.991288, 1.83116", \
                    "0.365378, 0.448722, 0.584275, 1.13597, 1.97017", \
                    "0.462096, 0.551586, 0.70164, 1.24437, 2.08131", \
                    "0.756459, 0.874246, 1.05713, 1.62898, 2.44989"); }
            rise_transition(delay_template_5x5) {
                index_1 ("0.06, 0.18, 0.42, 0.6, 1.2");
                index_2 ("0.025, 0.05, 0.1, 0.3, 0.6");
                values ( ... ); }
            cell_fall(delay_template_5x5) {
                index_1 ("0.06, 0.18, 0.42, 0.6, 1.2");
                index_2 ("0.025, 0.05, 0.1, 0.3, 0.6");
                values ( ... ); }
            fall_transition(delay_template_5x5) {
                index_1 ("0.06, 0.18, 0.42, 0.6, 1.2");
                index_2 ("0.025, 0.05, 0.1, 0.3, 0.6");
                values ( ... ); }
        } /* end timing */
        internal_power() {
            related_pin : "A";
            rise_power(energy_template_5x5) {
                index_1 ("0.06, 0.18, 0.42, 0.6, 1.2");
                index_2 ("0.025, 0.05, 0.1, 0.3, 0.6");
                values ( ... ); }
            fall_power(energy_template_5x5) {
                index_1 ("0.06, 0.18, 0.42, 0.6, 1.2");
                index_2 ("0.025, 0.05, 0.1, 0.3, 0.6");
                values ( ... ); }
        } /* end internal_power */
    } /* end Pin Y */
} /* end INVX1 */

```

Figure 7.5: Example **liberty** description of an inverter (with lookup table data mostly not included)

Operator	Description
'	invert previous expression
!	invert following expression
^	logical XOR
*	logical AND
&	logical AND
space	logical AND
+	logical OR
	logical OR
1	signal tied to logic 1
0	signal tied to logic 0

Figure 7.6: Function description syntax for **Liberty** files (EQN format)

### 7.1.2 Sequential Cell Definition

Sequential cells in general have events driven by the clock (or gate) signals. Thus these cells are characterized by *setup* and *hold* times as well as the normal propagation delay and rise/fall times. Because of this the timing measurements required for a sequential standard cell are:

- Setup time with respect to the clock when the input data is rising ( $T_{slh}$ )
- Setup time with respect to the clock when the input data is falling ( $T_{shl}$ )
- Hold time with respect to the clock when the input data is rising ( $T_{hlh}$ )
- Hold time with respect to the clock when the input data is falling ( $T_{hhf}$ )
- Propagation delay with respect to the input when output is falling ( $T_{phl}$ )
- Propagation delay with respect to the input when output is rising ( $T_{plh}$ )
- Rise time (of output) per unit load ( $T_{rise}$ )
- Fall time (of output) per unit load ( $T_{fall}$ )

Figure 7.7 shows the measurement of setup and hold times for a positive edge triggered flip flop or for a negative level gated latch (for both of these

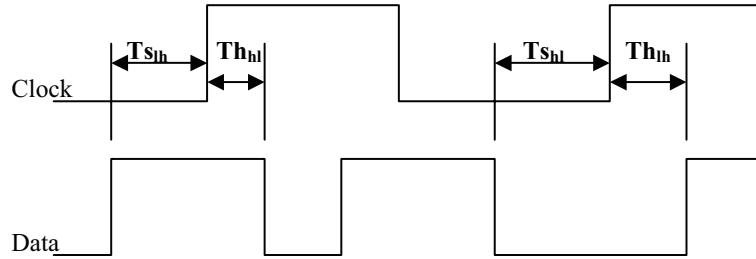


Figure 7.7: Setup and hold timing relative to the clock

sequential elements we make setup and hold time measurements with respect to the positive edge of the clock). The setup and hold times should be measured for all the input signals with respect to the clock. The propagation times and rise/fall times should be measured for all the outputs separately with respect to each of the input signals including clock.

Sequential cell functionality is identified by an **ff** group, **latch** group or **statetable** group statement inside the cell group.

### **ff** group syntax

The **ff** group describes a flip flop (an edge triggered memory element) in a cell. General syntax of an **ff** group is

```
ff ( variable1_name , variable2_name ) {
  clocked_on : "Boolean expression" ;
  next_state : "Boolean expression" ;
  clear : "Boolean expression" ;
  preset : "Boolean expression" ;
  clear_preset_var1 : L | H | N | T | X ;
  clear_preset_var2 : L | H | N | T | X ;
}
```

**Variable1\_name** is the name of the variable whose value is the state of the non-inverting output of the flip flop. This can be considered as the 1 bit storage of the flip flop. This is the internal Q (for example). **Variable2\_name** is the name of the variable whose value is the state of the inverting output of the flip flop. This is internal value of Qbar. It is these variables that are associated with the function attribute of the output pin group to describe the flip flop cell functionality. Both of the above variables should be specified even if either of the two is not connected to a primary

output pin. Valid variable names are anything except the pin names of the cell.

The **clocked\_on** attribute identifies the active edge of the clock signal and is required in all **ff** groups. The **next\_state** attribute is a Boolean expression that determines the value of **variable1** at the next active transition of the **clocked\_on** attribute. The Boolean expression is a function of the cell inputs and the **variable1** attribute (but never of **variable2**). The next state attribute is required in all **ff** groups.

The **clear** attribute gives the active value of the clear input and is optional. The **preset** attribute gives the active value of the preset input and is also optional.

The **clear\_preset\_var1** attribute determines the value of **variable1** when both **clear** and **preset** are active at the same time. This attribute can be present only if, and is required if, both **clear** and **preset** attributes are present in the **ff** group.

The **clear\_preset\_var2** attribute determines the value of **variable2** when both **clear** and **preset** are active at the same time. Similar to **clear\_preset\_var1**, this attribute also can be present only if, and is required if, both **clear** and **preset** attributes are present in the **ff** group. Valid values for **clear\_preset\_var1** and **clear\_preset\_var2** are:

L for low or '0'  
 H for high or '1'  
 N for no change  
 T for toggle  
 X for unknown

The flip-flop cell is activated whenever **clear**, **preset**, or **clocked\_on** changes. To sum up the above, Figure 7.8 shows the **ff** declaration for a JK flip-flop with asynchronous active low clear and preset and a D flip flop with synchronous active low clear.

### **latch group syntax**

The **latch** group describes a latch (a level sensitive memory element) cell. General syntax of latch group is

```
latch ( variable1_name , variable2_name ) {  

enable : "Boolean expression" ;  

data_in : "Boolean expression" ;  

clear : "Boolean expression" ;
```

```

ff(IQ,IQN) {
  clocked_on : "CLK" ;
  next_state : "(J K IQ') + (J K') + (J' K' IQ)" ;
  clear : "CLR'" ;
  preset : "SET'" ;
  clear_preset_var1 : X ;
  clear_preset_var2 : X ;
}

ff (IQ, IQN) {
  next_state : "D * CLR'" ;
  clocked_on : "CLK" ;
}

```

Figure 7.8: **ff** descriptions for a JK and a D flip flop

```

  preset : "Boolean expression" ;
  clear_preset_var1 : L | H | N | T | X ;
  clear_preset_var2 : L | H | N | T | X ;
}

```

The **variable1\_name**, **variable2\_name**, **clear**, **preset**, **clear\_preset\_var1**, and **clear\_preset\_var2** have the same meanings as that in the **ff** group discussed above.

The **enable** attribute identifies the active level of the clock signal, and the **data\_in** value is the name of the data signal if it is used. The **data\_in** and **enable** attributes are optional in a **latch** group, but if one of them is used, the other also must be used. The **latch** cell is activated when either of **clear**, **preset**, **enable**, or **data\_in** changes. Examples of two latch cells are shown in Figure 7.9. The first is a D-latch with active-high enable and active-low clear. The second is a set-reset (SR) latch with active-low set and reset signals.

### statetable group syntax

The statetable group describes functionality of more complex sequential cells. A state table is a sequential lookup table that specifies new values to internal nodes based on the current internal node values and inputs. The general syntax of a statetable group is

```

statetable( "input node names", "internal node names" ) {
  table : "input node values : current internal values : next internal values,\n"
         input node values : current internal values : next internal values,\n
         :
         :
}

```

```

latch(IQ, IQN) {
    enable : "CLK" ;
    data_in : "D" ;
    clear : "CLR'" ;
}

latch(IQ, IQN) {
    clear : "S'" ;
    preset : "R'" ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}

```

Figure 7.9: **latch** descriptions for a D latch and SR latch

```

statetable ("J      K      CLK      CLR", "IQ" ) {
    table: "-      -      -      L : -      : L, \
              -      -      ~F     H : -      : N, \
              L      L      F      H : L/H   : L/H, \
              H      L      F      H : -      : H, \
              L      H      F      H : -      : L, \
              H      H      F      H : L/H   : H/L" ;
}

```

Figure 7.10: **statetable** description of a JK flop flop

```

input node values : current internal values : next internal values";
}

```

The valid values for inputs and internal variables are

- L for low or '0'
- H for high or '1'
- F for falling or negative edge
- R for rising or positive edge
- N for no change
- T for toggle
- X for unknown

It's possible in a sequential library cell to have a **ff** or **latch** group along with a **statetable** group. But no sequential library cell can have more than one **statetable**. Also one needs to be careful not to have conflicting **ff** or **latch** groups along with a statetable group. An example of a **statetable** group for a JK flip flop with active-low asynchronous clear and negative edge clock is shown in Figure 7.10.

The timing attributes of a sequential cell are more complex than for a combinational cell. They are:

**timing\_type** This attribute distinguishes a combinational and sequential cell.

If this attribute is not assigned, the cell is considered combinational. The general format of this attribute is as shown below and is included in the input and/or output pin timing group of the cell. The syntax is **timing\_type : value ;**

Some of the valid values for sequential timing arcs and their meaning are as follows:

**rising\_edge** Identifies a timing arc whose output pin is sensitive to a rising signal at the input pin.

**falling\_edge** Identifies a timing arc whose output pin is sensitive to a falling signal at the input pin.

**preset** A preset arc implies that you are asserting a logic 1 on the output pin when the designated **related\_pin** is asserted.

**clear** A clear arc implies that you are asserting a logic 0 on the output pin when the designated **related\_pin** is asserted.

**hold\_rising** Designates the rising edge of the **related\_pin** for the hold check.

**hold\_falling** Designates the falling edge of the **related\_pin** for the hold check.

**setup\_rising** Designates the rising edge of the **related\_pin** for the setup check on clocked elements.

**setup\_falling** Designates the falling edge of the **related\_pin** for the setup check on clocked elements.

**timing\_sense** This attribute describes the way an input pin logically affects an output pin. Usually the **timing\_sense** is included in the output pin timing group of a cell and is derived automatically from the logic function of a pin. This attribute is assigned to override the derived value or while defining a non combinational element and the general format is **timing\_sense : value;**

The valid values for **timing\_sense** attribute and their meanings are as follows:

**positive\_unate** A function is said to be positive unate if a rising change on an input variable causes the output function variable to rise or not change and a falling change on an input variable causes the output function variable to fall or not change. For example, the output function of an AND gate is positive unate.

**negative\_unate** A function is said to be negative unate if a rising change on an input variable causes the output function variable to fall or not change and a falling change on an input variable causes the output function variable to rise or not change. For example, the output function of a NAND gate is negative unate.

**non\_unate** The **non\_unate** timing sense represents a function whose output value change cannot be determined from the direction of the change in the input value. For example, the output function of an XOR gate is **non\_unate**.

Also note that if a **related\_pin** is an output pin, the **timing\_sense** attribute for that pin must be defined.

**clock** This attribute is optional and indicates whether an input pin is a clock pin. The syntax is **clock : true;** or **clock : false;**

**min\_period** This is an optional attribute on the clock pin of a flip flop or latch specifies the minimum clock period required for the input pin. The syntax is **min\_period : value;**

A D-type edge triggered flip flop can be described in terms of all these attributes. The **.lib** description (with the data in the lookup tables missing so that it will fit on a page) is shown in Figure 7.11. Note that in this case the setup and hold timing are defined as scalar types, but they could just as easily (well, not really just as easily because there would be extra simulation involved) be described in a lookup table too. The first part of the flip flop description is an **ff** block describing the functionality. Then the pin descriptions follow. The **D** input pin had setup and hold timing. The **clk** pin is defined as a clock, and the **Q** pin has rise and fall timing described as lookup tables and has timing described with both **clk** and **clr** as related pins. The timing values are defined as 5 x 5 lookup tables as seen in Figure 7.5.

### 7.1.3 Tristate Cell Definition

The major difference of a tristate cell to other cells is that a tristate cell output can take a value of **Z** as well as **1** and **0**. The output goes to the **Z** value when the cell enable pin is de-asserted or when the tristate condition (generally a Boolean expression) is true. Electrically this **Z** value is a high impedance value on the output. It is as if the output drivers are disconnected from the output wire. It is also worthwhile to note that both combinational tristate cells as well as sequential tristate cells exist.

The type of measurements required for tristate cells are the same as any other combinational or sequential cells. However when it comes to

```

/* positive edge triggered D-flip flop with active low reset */
cell(dff) {
    area : 972;
    cell_footprint : "dff";
    ff("IQ", "IQN") {
        next_state : " D ";
        clocked_on : " G ";
        clear : " CLR' ";
    }
    pin(D) {
        direction : input;
        capacitance : 0.0225;

        timing() { /* hold time constraint for a rising transition on G */
            timing_type : hold_rising;
            rise_constraint(scalar) { values("-0.298"); }
            fall_constraint(scalar) { values("-0.298"); }
            related_pin : "G";
        }
        timing() { /* setup time constraint for a rising transition on G */
            timing_type : setup_rising;
            rise_constraint(scalar) { values("0.018"); }
            fall_constraint(scalar) { values("0.018"); }
            related_pin : "G";
        }
    } /* end of pin D */
    pin ( CLK ) {
        direction : input;
        capacitance : 0.0585;
        clock : true;
    } /* end of pin CLK */
    pin ( CLR ) {
        direction : input;
        capacitance : 0.0135;
    } /* end of pin CLR */
    pin ( Q ) {
        direction : output;
        function : "IQ";

        timing () { /* propagation delay from rising edge of CLK to Q */
            timing_type : rising_edge;
            cell_rise(lu5x5) { values( "..."); }
            rise_transition(lu5x5) { values( "..."); }
            cell_fall(lu5x5) { values( "..."); }
            fall_transition(lu5x5) { values( "..."); }
            related_pin : "CLK";
        } /* end of Q timing related to CLK */

        timing () { /* propagation delay from falling edge of clear to Q=0 */
            timing_type : clear;
            timing_sense : positive_unate;
            cell_fall(lu5x5) { values( "..."); }
            fall_transition(lu5x5) { values( "..."); }
            related_pin : "CLR";
        } /* end of Q timing related to CLR */
    } /* end of pin Q */
} /* end of cell dff */

```

Figure 7.11: Example D-type flip flop description in .lib format

measurement of the tristate output pin parameters with respect to the cell enable pin (or pin related to the tristate condition), one needs to remember that either the initial output value or the final output value will be **Z**. Also two sets of measurements are required with respect to the cell enable pin: one when the cell is being enabled (output going to **1** or **1**) and one when the cell is being disabled (output going to **Z**). This makes measurement of time-to-reach-tristate and time-to-start-driving a little tricky. If the outputs are not connected to capacitors the analog simulator generally simulator puts the output of a tri-stated device at around 2.5v for a 5v process, so we can measure from this point when measuring the time to or from a tristate output state (or some percentage of that point if you're using a 10% to 90% measurement, for example).

Special attributes for a tristate gate include

**three\_state** This attribute defines a tristate output pin in a cell. The syntax is **three\_state : <Boolean-expression> ;**

The Boolean expression is the condition to make the corresponding output pin to go to tristate or high impedance condition.

**three\_state\_enable** This is the value the **timing\_type** attribute should take for the tristatable output pin in the “cell being enabled” timing group. The **cell\_rise** and **rise\_transition** will correspond to the **Z to 1** transition, and the **cell\_fall** and **fall\_transition** will correspond to the **Z to 0** transition.

**three\_state\_disable** This is the value then **timing\_type** attribute should take for the tristatable output pin in the “cell being disabled” timing group. The **cell\_rise** and **rise\_transition** will correspond to the **0 to Z** transition, and the **cell\_fall** and **fall\_transition** will correspond to the **1 to Z** transition.

An example of a tristate inverter is shown in Figure 7.12. Note that the enable and disable timing matrix templates are different than the “regular” delay value matrix template. The indices of these templates should have been defined in the technology header of the **.lib** file (see Figure 7.2), or the actual index values can be defined before each **values** statement in the **timing** block (as they were for the inverter in Figure 7.5).

## 7.2 Cell Characterization with SignalStorm

SignalStorm is a tool from Cadence that uses Spectre to characterize cells and outputs the results in a format that can be converted into **liberty**

```
/* Enabled inverter or tristate inverter */
cell(eninv) {
    area : 324;
    cell_footprint : "eninv";
    pin(A) {
        direction : input;
        capacitance : 0.027;
    } /* end of pin A */
    pin(En) {
        direction : input;
        capacitance : 0.0135;
    } /*end of pin En */
    pin(Y) {
        direction : output;
        function : "A'";
        three_state : "En'";
        timing () {
            timing_sense : negative_unate;
            related_pin : "A";
            cell_rise(lu5x5) { values( " ... " );}
            rise_transition(lu5x5) { values( " ... " );}
            cell_fall(lu5x5) { values( " ... " );}
            fall_transition(lu5x5) { values( " ... " );}
        } /* end of enabled timing */
        timing() {
            timing_sense : positive_unate;
            timing_type : three_state_enable;
            related_pin : "En";
            cell_rise(delay_template_5x5) { values( " ... " );}
            rise_transition(delay_template_5x5) { values( " ... " );}
            cell_fall(delay_template_5x5) { values( " ... " );}
            fall_transition(delay_template_5x5) { values( " ... " );}
        } /* end of enable timing */
        timing() {
            timing_sense : negative_unate;
            timing_type : three_state_disable;
            related_pin : "En";
            cell_rise(delay_template_5x1) { values( " ... " );}
            rise_transition(delay_template_5x1) { values( " ... " );}
            cell_fall(delay_template_5x1) { values( " ... " );}
            fall_transition(delay_template_5x1) { values( " ... " );}
        } /* end of disable timing */
    } /* end of pin Y */
} /* end of eninv */
```

Figure 7.12: Example tristate inverter .lib description

format. It takes a bit of fiddling to get the inputs just right for this program, mostly because **SignalStorm** uses pure **Spectre** format as input whereas the NCSU tech libraries that are the basis for our cells uses the **Spice**-style input format that **SpectreS** can read. However, once the input is massaged to the right form this is a very slick program. When you consider that even for the inverter, using  $5 \times 5$  timing matrices (25 values each), there are  $6 * 25 = 150$  **Spectre** runs required to characterize the cell. Each run changes the input slope or the output loading or both and then measures the input and output waveform timing to get a number for one of the timing matrices. Cells with larger numbers of inputs and outputs, and sequential cells will require even more **Spectre** runs to characterize. Even a two-input NAND takes  $12 * 25 = 300$  **Spectre** runs for full characterization using  $5 \times 5$  matrices.

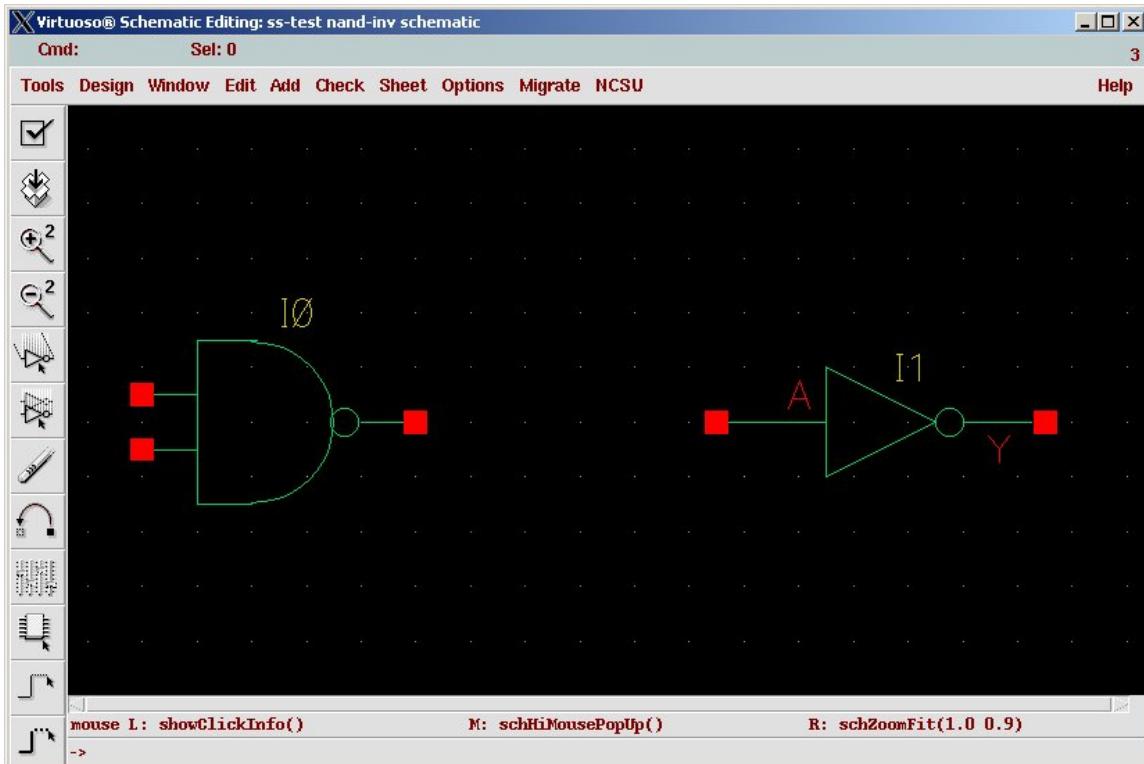
The other very nice thing about **SignalStorm** is that it even figures out for you what the functionality of your cells is. You don't have to specify in advance that a cell is, for example, a two-input and-or-invert gate. **SignalStorm** will figure this out for you based on analyzing the transistor network. Of course, you need to check to make sure that it has determined the functionality correctly, but it's quite good at this.

### 7.2.1 Generating the **SignalStorm** netlist

**SignalStorm** requires a netlist of all the cells that you would like it to characterize. This netlist should be *sub-circuit* definitions in **Spectre** format (which is very similar, but not exactly like **Spice** format). You could write these by hand, but it's much more convenient to have **Cadence** generate them for you from your schematics. The process will be very similar to the first steps of the analog simulation process from Chapter 6, but you'll have to specify the **Spectre** simulator instead of **SpectreS**, make a few more tweaks, and then stop at netlisting without actually simulating. From there (unfortunately) some hand-modification of the netlist is required.

Before you can characterize a cell you need a **cmos\_sch** transistor level schematic view, and a **layout** view of each cell you want to characterize. The layout should have passed both **DRC** and **LVS**, and you should have created an **analog\_extracted** view after the **LVS** was complete. It is the **analog\_extracted** view that we'll use to generate the transistor netlist for **SignalStorm** characterization. This is because if you have extracted with parasitic capacitors then the **analog\_extracted** view will include not only extra information on the transistor sizes, but also those parasitics.

As an example I'll start with a library that includes the inverter and nand2 cells that you've seen from Chapters 3, 4, and 5. The library includes (at least) **cmos\_sch**, **layout**, and **analog\_extracted** views of both of these

Figure 7.13: Schematic with one instance of **inv** and **nand2**

cells named **inv** and **nand2**.

Start by creating a schematic that contains one instance of each cell you would like to characterize. Don't connect them to anything, just place an instance in the schematic. When you save this schematic Cadence will complain that there are unconnected pins, but you can ignore that. As an example, my schematic (named **SignalStorm**) is shown in Figure 7.13.

*Chapter 6 uses SpectreS because that's what is most easily supported for user-driven analog simulation by the NCSU technology files we're using.*

After saving this schematic (and ignoring the warnings), open the **Analog Environment** with **Tools → Analog Environment**. This is just like Chapter 6. However, we need to change a couple things because SignalStorm requires pure Spectre format and not the SpectreS format that is used in Chapter 6.

1. Use the **Setup → Simulator / Directory / host** menu to open a dialog box. From there change the **Simulator** to be **Spectre** (with no “S” on the end). This will force the analog netlister to use Spectre format when it generates the netlist. See Figure 7.14.
2. Use the **Setup → Environment** menu to open a dialog box and add

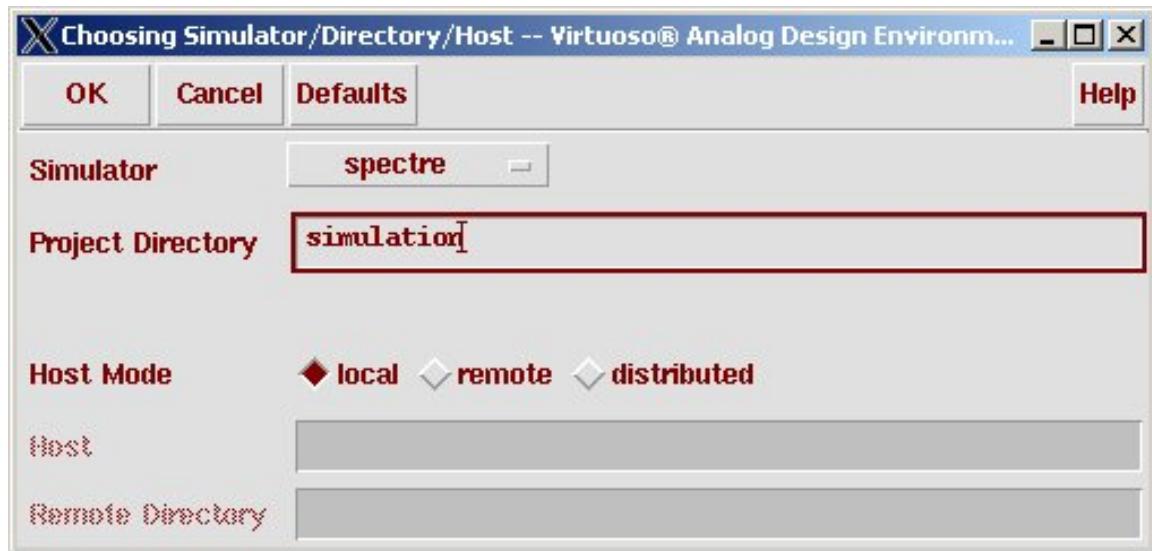


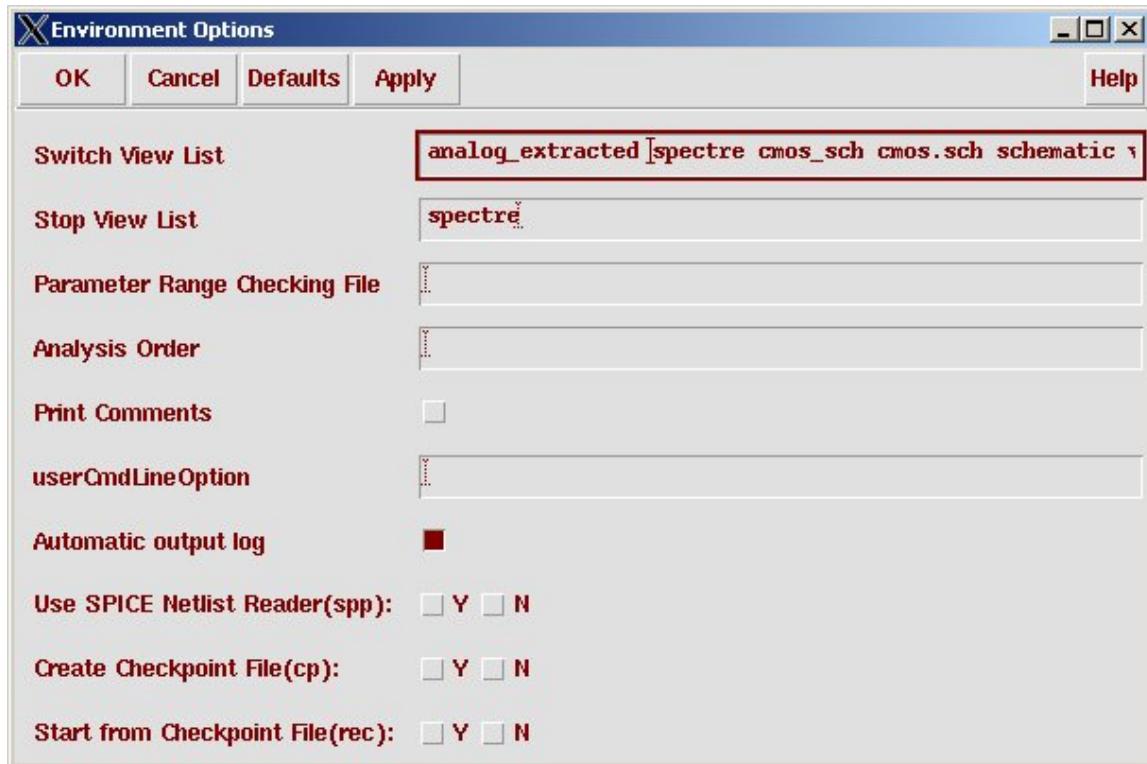
Figure 7.14: Choosing Spectre as the simulator

**analog\_extracted** to the front of the **Switch View List** list. See Figure 7.15. This will tell the netlister to look for the **analog\_extracted** view first to find the transistor netlists for each cell.

Now you can generate the netlist with **Simulation** → **Netlist** → **Create**. This will pop up a new window with the Spectre netlist of your schematic (the one that has instances of each cell). You should **Save As** this file to a file named **dut.scs** which will appear in the directory in which you started **cad-ncsu**. Now, unfortunately, you need to hand-edit this file to make some changes! This is because the netlister from NCSU wasn't designed to make perfect SignalStorm files. But, they're close. Once you've created and saved this **dut.scs** file you can close the Analog Environment and exit Cadence.

What SignalStorm wants is a file that contains nothing but **subckt** definitions. These are the descriptions of each cell as a sub-circuit. The initial **dut.scs** file is shown in Figure 7.16. You can see that it starts with some control statements, then defines each cell as a **subckt**, then includes instances of each of those sub-circuits (the **I1** and **I0** statements are instances), and finishes with some additional control statements for the simulator. You need to take this file and modify it in the following ways:

1. First remove all the lines of text before the first **subckt** definition (leaving the comments is all right), and after the last **subckt** is finished. All SignalStorm wants is the **subckt** definitions, nothing else.

Figure 7.15: Adding **analog\_extracted** to the **Switch View List**

Note that **Spectre** (at least as used by **SignalStorm**) has no concept of global signals so removing the **global** statement won't cause problems.

The exception to this rule is that you must leave the following line in the file before the first **subckt** so that **Spectre** knows what language to expect (i.e. not to use Spice syntax):

```
simulator lang=Spectre
```

2. **SignalStorm** doesn't like non-alpha characters in signal names. Change every instance of **vdd!** to be **vdd**.
3. **SignalStorm** doesn't like to use **0** as the default ground node. This is quite odd because this is a standard convention, but it appears to be true for the specific way that **SignalStorm** uses **Spectre**. Unfortunately, this is a trickier thing to change with search-and-replace because there are also 0's in other parameters. Make the following replacements:
  - (a) Replace “**0**” with “**gnd**” (note that there are spaces on both sides of each of those strings!).
  - (b) Replace “(**0**)” with “(**gnd**)”.
  - (c) Replace “**0)**” with “**gnd)**”.
4. Because **SignalStorm** doesn't use global signal names, you need to add **vdd gnd** to the argument list of each **subckt** definition.

A **dut.scs** file that has had all the required modifications is shown in Figure 7.17. Note that it consists only of **subckt** definitions, has **vdd gnd** added to all **subckt** argument lists, and uses **vdd** and **gnd** to define the power and ground connections.

Now that you have a modified **dut.scs** file with **subckt** definitions for the cells you want to characterize, you can start using **SignalStorm**. I recommend that you make a separate directory in which to run **SignalStorm** so that you can compartmentalize things and keep track of where the log files and result files are. I'm using a **\$HOME/IC\_CAD/slC** directory for this (**slC** stands for **SignalStorm Library Characterizer**). Move your **dut.scs** file to that directory.

In your **slC** directory you need to copy some script files from the class directory, specifically the **cadence/SLC** subdirectory. The directory path is:

```
/uusoc/facility/cad_common/local/class/6710/cadence/SLC
```

From that directory, copy the **step1**, **step2** and **step3** files to your **slC** directory. These are command files you can use to drive the **SignalStorm**

```

// Generated for: Spectre
// Generated on: Aug 31 18:05:16 2006
// Design library name: tutorial
// Design cell name: signalstorm
// Design view name: schematic
simulator lang=Spectre
global 0 vdd!

// Library name: tutorial
// Cell name: inv
// View name: analog_extracted
subckt inv A Y
  \+1 (Y A vdd! vdd!) ami06P w=6e-06 l=6e-07 as=9e-12 ad=9e-12 ps=9e-06 \
    pd=9e-06 m=1 region=sat
  \+0 (Y A 0 0) ami06N w=3e-06 l=6e-07 as=4.5e-12 ad=4.5e-12 ps=6e-06 \
    pd=6e-06 m=1 region=sat
ends inv
// End of subcircuit definition.

// Library name: tutorial
// Cell name: nand2
// View name: analog_extracted
subckt nand2 A B Y
  \+3 (vdd! B Y vdd!) ami06P w=6e-06 l=6e-07 as=5.4e-12 ad=9e-12 \
    ps=1.8e-06 pd=9e-06 m=1 region=sat
  \+2 (Y A vdd! vdd!) ami06P w=6e-06 l=6e-07 as=9e-12 ad=5.4e-12 \
    ps=9e-06 pd=1.8e-06 m=1 region=sat
  \+1 (Y B _6 0) ami06N w=6e-06 l=6e-07 as=2.7e-12 ad=9e-12 ps=9e-07 \
    pd=9e-06 m=1 region=sat
  \+0 (_6 A 0 0) ami06N w=6e-06 l=6e-07 as=9e-12 ad=2.7e-12 ps=9e-06 \
    pd=9e-07 m=1 region=sat
ends nand2
// End of subcircuit definition.

// Library name: tutorial
// Cell name: signalstorm
// View name: schematic
I1 (net1 net2) inv
I0 (net5 net4 net3) nand2
simulatorOptions options reltol=1e-3 vabstol=1e-6 iabstol=1e-12 temp=27 \
  tnom=27 scalem=1.0 scale=1.0 gmin=1e-12 rforce=1 maxnotes=5 maxwarns=5 \
  digits=5 cols=80 pivrel=1e-3 ckptclock=1800 \
  sensfile="../psf/sens.output" checklimitdest=psf
modelParameter info what=models where=rawfile
element info what=inst where=rawfile
outputParameter info what=output where=rawfile
designParamVals info what=parameters where=rawfile
primitives info what=primitives where=rawfile
subckts info what=subckts where=rawfile
saveOptions options save=allpub

```

Figure 7.16: **dut.scs** file as generated by Analog Environment

```

simulator lang=Spectre

// Library name: tutorial
// Cell name: inv
// View name: analog_extracted
subckt inv A Y vdd gnd
    \+1 (Y A vdd vdd) ami06P w=6e-06 l=6e-07 as=9e-12 ad=9e-12 ps=9e-06 \
        pd=9e-06 m=1 region=sat
    \+0 (Y A gnd gnd) ami06N w=3e-06 l=6e-07 as=4.5e-12 ad=4.5e-12 ps=6e-06 \
        pd=6e-06 m=1 region=sat
ends inv
// End of subcircuit definition.

// Library name: tutorial
// Cell name: nand2
// View name: analog_extracted
subckt nand2 A B Y vdd gnd
    \+3 (vdd B Y vdd) ami06P w=6e-06 l=6e-07 as=5.4e-12 ad=9e-12 \
        ps=1.8e-06 pd=9e-06 m=1 region=sat
    \+2 (Y A vdd vdd) ami06P w=6e-06 l=6e-07 as=9e-12 ad=5.4e-12 \
        ps=9e-06 pd=1.8e-06 m=1 region=sat
    \+1 (Y B _6 gnd) ami06N w=6e-06 l=6e-07 as=2.7e-12 ad=9e-12 ps=9e-07 \
        pd=9e-06 m=1 region=sat
    \+0 (_6 A gnd gnd) ami06N w=6e-06 l=6e-07 as=9e-12 ad=2.7e-12 ps=9e-06 \
        pd=9e-07 m=1 region=sat
ends nand2
// End of subcircuit definition.

```

Figure 7.17: **dut.scs** after required SignalStorm modifications

characterization process. These scripts are used one at a time in sequence, but they represent three important steps in the process where you may need to stop and fix errors which is why they're separate. You can use the scripts as-is, or edit them to do different things once you are more familiar with the process. To run **SignalStorm** use the `cad-slc` script in the same location as the `cad-ncsu` script. With these files in place you can run the characterization with the following steps. The syntax for running with the script is

```
cad-slc -S <scriptname>
```

1. `cad-slc -S step1` This script will open a database called **foo** to hold your data (with the **SignalStorm** command `db_open`). The script (shown in Figure 7.18) then sets some flags to control how **SignalStorm** works, and uses `db_install` to install a set of transistor models (typical case models for the AMI C5N  $0.6\mu$  process that we're using) and your `dut.scs` file with the subckt descriptions into that database. The `db_gsim -force` command will evaluate the transistor networks to understand what they are, and generate test vectors for characterization. In this phase **SignalStorm** actually applies all possible input vectors to your circuit to try to figure out what it is. Brute force, but effective. The `db_gate` command will print out what gate it thinks your cell is so that you can verify that **SignalStorm** figured it out right. The `db_setup` command reads a setup file that contains lots of details about how the simulation should happen. Finally `db_close` closes the database.

After running **cad-slc** with the **step1** script you should look carefully at the output to make sure that things are proceeding correctly. If you see messages such as **no supply0** or **no simulation** anywhere in the output of this process, you have a problem! Usually these are editing errors in `dut.scs` from the hand-edit phase of this process. Look for misspelled `vdd` and `gnd` names, or for any extra lines in the file that are not related to the `subckt` definitions. The `db_gate` command will print out what gate **SignalStorm** thinks it has extracted from your transistors. Make sure that the gate matches with what you think the gates are!

The output of the `db_gate` command for my `dut.scs` file that contains one instance of an inverter and one instance of a nand2 is shown in Figure 7.19. There is a lot more output from the **step1** phase of the process. This is just a piece of a much longer piece of output. Make sure that you are getting no errors or warnings before proceeding to **step2**!

2. `cad-slc -S step2` This script (shown in Figure 7.20) starts by

```

db_open foo

set_var SG_SPICE_SIMPLIFY true
set_var SG_HALF_WIDTH_HOLD_FLAG true
set_var SG_SIM_NAME "Spectre"
set_var SG_SIM_TYPE "Spectre"
set_var SG_SPICE_SUPPLY1_NAMES "vdd"
set_var SG_SPICE_SUPPLY0_NAMES "gnd"

db_install -model /uusoc/facility/.../6710/cadence/SLC/ami_c5n_typ.scs -subckt dut.scs
db_gsim -force
db_gate
db_setup -s /uusoc/facility/.../6710/cadence/SLC/setup.ss -process typical
db_close
exit

```

Figure 7.18: SignalStorm **step1** script (path names are shortened...)

```

=====
DESIGN : INV
=====
DESIGN ( INV );
// =====
// PORT DEFINITION
// =====
INPUT A ( A );
OUTPUT Y ( Y );
SUPPLY0 GND ( GND );
SUPPLY1 VDD ( VDD );
// =====
// INSTANCES
// =====
NOT ( Y, A );
END_OF_DESIGN;

=====
DESIGN : NAND2
=====
DESIGN ( NAND2 );
// =====
// PORT DEFINITION
// =====
INPUT A ( A );
INPUT B ( B );
OUTPUT Y ( Y );
SUPPLY0 GND ( GND );
SUPPLY1 VDD ( VDD );
// =====
// INSTANCES
// =====
NAND ( Y, A, B );
END_OF_DESIGN;

```

Figure 7.19: SignalStorm **step1** output (portion)

```
db_open foo

# Remove the next 3 lines to use the ipsd/ipsc
# deamons for load balancing on multiple machines
set_var SG_SIM_USE_LSF 1
set_var SG_SIM_LSF_CMD ""
set_var SG_SIM_LSF_PARALLEL 10

set_var SG_SIM_NAME "Spectre"
set_var SG_SIM_TYPE "Spectre"
set_var SG_SPICE_SUPPLY1_NAMES "vdd"
set_var SG_SPICE_SUPPLY0_NAMES "gnd"
set_var SG_HALF_WIDTH_HOLD_FLAG true

db_spice -s spectre -p typical -keep_log
db_close
exit
```

Figure 7.20: SignalStorm **step2** script

opening the database named **foo** that was created in **step1**. The comment about the next three lines relates to a feature of SignalStorm that lets you start servers on a bunch of machines and spawn simulation jobs on those remote machines during characterization. We won't use that option, but you can easily imagine that this would be a good idea for a very large library and a very large room of computers chugging away at characterization. In our case we'll restrict ourselves to one machine, but spawn up to 10 Spectre jobs at the same time.

The **db\_spice** command is where all the simulation action happens. It uses the **subckt** definitions you supplied in **dut.scs** and the test vectors that were derived in **step1** to simulate and extract timing and power numbers using Spectre. This step can take a very long time depending on the number of cells and the complexity of the cells that you're characterizing. The key here is to make sure that the results of all the simulations is **PASS**. An example of a portion of the output from this step on my example is shown in Figure 7.21. If you get **FAIL** on any of the simulations you'll have to look at the output in the **signalstorm.log** and **signalstorm.work** directories to try to figure out what's going on. Also look back at the output from **step1** to make sure you weren't starting with problems.

Once you complete **step2** correctly, the characterization data is in the **foo** database and you can move to **step3**.

3. **cad-slc -S step3** This script (shown in Figure 7.22) simply outputs to the results of **step2** into file named **foo.alf** and also outputs a Verilog file with simple behavioral descriptions of each cell in the

```

slc> db_spice -s spectre -p typical -keep_log

        DESIGN      PROCESS      #ID     STATUS
-----+-----+-----+-----+
INV      typical    D0000    SIMULATE
INV      typical    D0001    SIMULATE
===== | ===== | ===== | =====
INV      typical    2       2
-----+-----+-----+-----+
NAND2     typical   D0000    SIMULATE
NAND2     typical   D0001    SIMULATE
NAND2     typical   D0002    SIMULATE
NAND2     typical   D0003    SIMULATE
NAND2     typical   D0004    SIMULATE
===== | ===== | ===== | =====
NAND2     typical   5       5
-----+-----+-----+-----+
-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
Simulation Summary
-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
-----+-----+-----+-----+-----+
        DESIGN | PROCESS | #ID   | STAGE  | STATUS
-----+-----+-----+-----+-----+
INV      typical   D0000  VERIFICATE  PASS
INV      typical   D0001  VERIFICATE  PASS
NAND2     typical   D0000  VERIFICATE  PASS
NAND2     typical   D0001  VERIFICATE  PASS
NAND2     typical   D0002  VERIFICATE  PASS
NAND2     typical   D0003  VERIFICATE  PASS
NAND2     typical   D0004  VERIFICATE  PASS
-----+-----+-----+-----+-----+
-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
- Total Simulation : 7
- Total Passed     : 7(100%)
- Total Failed     : 0(0%)
-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
slc> db_close

Database : foo is closed
slc> exit

```

Figure 7.21: SignalStorm step2 output (portion)

```
db_open foo
db_output -r foo.rep -alf foo.alf -p typical
db_verilog -r foo.v
db_close
exit
```

Figure 7.22: SignalStorm **step3** script

library. The **.alf** file contains all the characterization data that was generated by SignalStorm. Unfortunately, **.alf** format is not quite what we want. We want the **liberty** format described in Section 7.1. Fortunately, there's an automatic way to do this.

4. Use yet another script to run the conversion program from **alf** to **lib** format. This script is

`cad-alf2lib <basename>` where `<basename>` is the base-name of the **alf** file. In our case this, unless you've edited the scripts, this will be **foo**. So, with `cad-alf2lib foo` you will generate a **foo.lib** file which contains the **liberty** version of your newly characterized library.

*It seems odd with a name like “footprint,” but cells with the same I/O and function, but with different sizes, can share the same **footprint**. For example, all inverters, regardless of physical area, should be defined to use a common **footprint** so that they can be used later for driving different loads.*

This script uses a **footprints.def** file to get some additional information about your cells. The most important missing pieces of information are the area of the cells (in square microns) and the **footprint** of the cell. The **footprint** is a way of grouping cells that have the same function but perhaps different drive strengths, for example. These cells are grouped into a common **footprint** so that later in synthesis they may be used interchangeably depending on loads being driven or speed required. An example of a **footprints.def** file is shown in Figure 7.23. This file is also in the following directory:

`/uusoc/facility/cad_common/local/class/6710/cadence/SLC`

You can copy it from there and modify as needed. This file says that all cells whose name starts with **INV** should have the same **inv** footprint, and should have area **129.6** (in square microns). Clearly you will need to edit this file to reflect the sizes and names of your cells. If you don't have this **footprints.def** file in your directory, the **cad-alf2lib** process will still generate a **foo.lib** file, but without areas or footprints. This information is very useful to the synthesis process, though, so you should include it!

You can, of course, run through this process with small numbers of cells and then edit them into a larger **.lib** file. There's no reason to wait until all cells are ready before running characterization, or to re-characterize cells

```

cell INV* {
footprint inv ;
area 129.6 ;
};

cell NAND2* {
footprint nand2 ;
area 194.4 ;
};

cell NOR2* {
footprint nor2 ;
area 194.4 ;
};

```

Figure 7.23: Example **footprints.def** file

that haven't changed. Remember to keep the technology header of the **.lib** file intact once, and add new cell descriptions to the end.

### 7.2.2 Cell Naming and SignalStorm

In **step1** of the SignalStorm procedure a setup file named **setup.ss** is loaded from the class directory. This file sets up a number details about how the simulation should proceed. Most importantly it sets up what the indices should be for the characterization matrices. These matrices have **input\_transition\_time** in ns on one axis, and **total\_output\_net\_capacitance** in pF on the other axis. The default values for these are:

```

Index DEFAULT_INDEX{
    Slew = 0.100n 0.30n 0.7n 1.0n 2.0n;
    Load = 0.025p 0.05p 0.1p 0.3p 0.6p;
};

```

However, the **setup.ss** file also defines some different loads and slews for cells that are designed to drive larger loads. Any cell whose name ends in **X1** will also have this standard load. A cell whose name ends in **X2** to indicate twice as much drive capability on the outputs will have a different set of values:

```

Index X2{
    Slew = 0.100n 0.30n 0.7n 1.0n 2.0n;
    Load = 0.050p 0.10p 0.2p 0.6p 1.2p;
};

```

Furthermore, still different loads and slews are defined for cells whose name ends in **X4** and **X8**. So, for example, if you have several different

inverters in your library with different drive strengths (different output transistor sizes) named **INVX1**, **INVX2**, **INVX4** and **INVX8**, then they will each be characterized with a load and slew that best matches where they are intended to be used. Of course, you can copy the **setup.ss** file and modify things if you'd rather follow a different naming convention, or want to add or modify things. The full **setup.ss** file can be seen in Appendix C.

### 7.2.3 Best, Typical, and Worst Case Characterization

Commercial libraries are almost always characterized for three different versions of cell timings: best case, typical case, and worst case. These are supposed to represent three different variations on how the circuit is likely to work. Best case timings are usually feasible, but quite optimistic. They assume transistors that operate in the fast region measured by the foundry, with a vdd that's typically 10% higher than nominal, and at a nice cool temperature. Typical case timings are supposed to model what you'll usually see in the fabricated circuit. The transistor models are taken from the middle of the measured performance distribution, vdd is nominal (5v in our  $6\mu$  process), and the temperature is assumed to be 25C. Worst case timings are taken from the slow end of the transistor distribution, with vdd at 10% below nominal, and a nice toasty temperature of 125C. If you're trying to see what speed your circuit might run at, use the **typical** timings. If you're trying to see whether your circuit will work in more general environments, use the **worst** timings. Only a very optimistic person would ever use **best** timings!

The **step1**, **step2** and **step3** scripts are designed to measure **typical** case timings for you cells. However, if you would like to generate **best** and **worst** case libraries as well, there are versions of the scripts to do this. For example, **step[1,2,3]-worst** will measure wort case timings using worst-case transistor models from **MOSIS**, vdd assumed to be 4.5v, and a temperature of 125C. These scripts are in the class library.

## 7.3 Cell Characterization with Spectre

If you want to characterize cells by hand, you can do this using **SpectreS** by making your own test fixture circuits and doing analog simulation as described in Chapter 6. This is clearly a little more hand-work than letting **SignalStorm** run everything, but you have much more control over every aspect of the characterizations. My recommendation is to use **SignalStorm**, but here are some things to think about if you'd like to do things directly with **SpectreS**.

The process of characterizing cells involves running lots of simulation

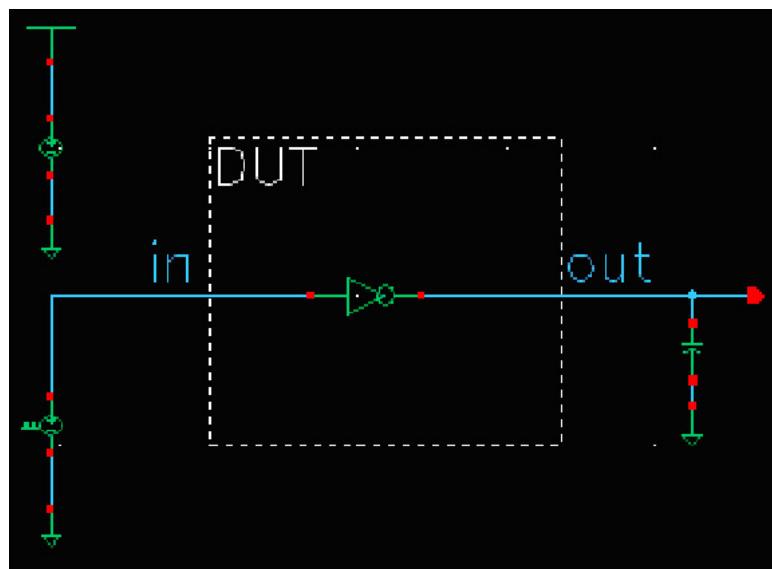


Figure 7.24: Test fixture for hand-simulation and characterization

while varying the input slope and the output load so that you can fill out the timing tables. This argues strongly for using parametric simulation where you vary the parameters of the voltage sources and the output loads automatically. For example, you could set up a test fixture schematic as in Figure 7.24. In this test fixture the DUT is driven by a **vpulse** voltage source, and is driving load capacitor. The slope of the pulse will be controlled by a variable named **slope** and the size of the load will be controlled by a variable named **load**. To assign these component parameters a variable name, simply give the variable name in the **q** properties of the device instead of a numeric value.

Now you can use the same parametric simulation techniques as described in Chapter 6 Section 6.5.1. In this case you could vary the input slope to be a series of times in ns, and the output load to be a series of values in pF and generate all 25 values needed to populate the tables we've been using. You could, of course, also use different numbers of **slope** and **load** values to generate differently sized tables. This is easily set up in the **Analog Environment** GUI, but it can be tedious to use the menu and dialog box interfaces to set this up time after time. Another answer is to write a script to automate the process of doing the parametric simulation.

In the Cadence tool suite the separate tools under the Design Framework umbrella are controlled by a scripting language called *Skill*. Skill is a language that looks a lot like Lisp. In fact, it looks almost exactly like Lisp. But, if you aren't as fond of Lisp syntax conventions as I am, there is

a C-like syntax available for Skill programs. If you know the right function calls, you can write a Skill script to do almost anything in Cadence that you can do through the GUI. Of course, it's sometimes a little tricky to figure out what the function calls are, and how the script gets all the information it needs, so usually Skill scripting is reserved for the hard-core user or for the CAD support folks. But, for some relatively simple stuff, it can be very useful. Skill is, of course, documented in the Cadence documentation. If you're curious I recommend that you start with the Skill users manual (accessible from the Cadence help menu).

An extension to Skill that is specifically for writing scripts to control analog simulation is called **ocean**. An **ocean** script looks just like Skill, but has some additional functions that are specific to the task of interacting with the **Analog Environment**. For example, **ocean** scripts can set which design you're looking at, which set of waveform results you're evaluating, and which signals in those waveforms you're measuring.

I've written an ocean script that you can use or modify to perform parametric simulation of a cell and which will generate the output in a set of lookup tables formatted for the **.lib** file. It turns out that formatting the output to look like **liberty** format wants it is practically the hardest part! The other hard part is that I can't figure out is how to generate the netlist for simulation in the first place in the script, so you still need to use the **Analog Environment** GUI for part of the process, the same as for **SignalStorm**. But, after you get things set up, it's as easy as typing a function name in the CIW and you get the parametric simulation results generated for you. Of course, it's not remotely as automated as **SignalStorm** so I still recommend that you stop reading now and go back to the **SignalStorm** section!

If you're still reading, in order to use the script, you'll need a schematic that describes the DUT simulation setup. I called this schematic **test\_setup** for lack of a more imaginative name. Because we're aiming to simulate over a range of output loads I put a capacitor on the output with the variable **load** as the capacitance value. Because we're also aiming to simulate over a range of input slopes, I put a **Vpulse** on the input of the DUT with the variable **slope** as the rise and fall time values. I can then use the parametric simulation setup to vary both of those values. The schematic is shown in Figure 7.24.

Notice that the DUT is being driven by the **Vpulse** and is driving into the load capacitor. If you were simulating a multi-input gate, you'd need to tie the non-driven inputs of that gate to an appropriate value that would turn the output of the gate to either an inverter or a buffer. For example, if you were simulating a 2-input NAND gate, you'd want to tie the input that you're not driving to logic 1 (vdd) so that the pulse on the other input would come through the output as an inverted signal. Remember that you need to

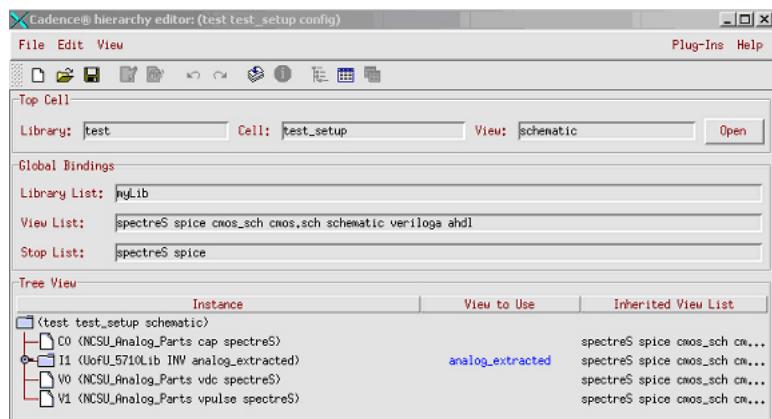


Figure 7.25: **config** view for hand-simulation and characterization

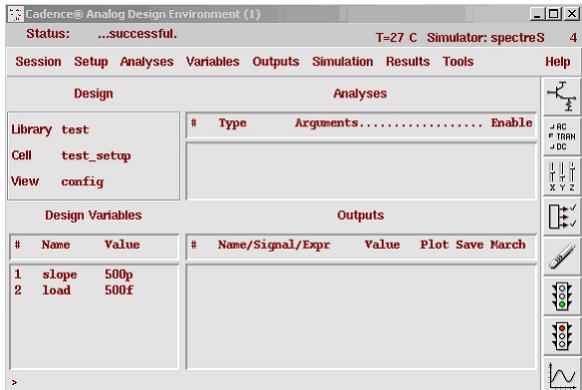
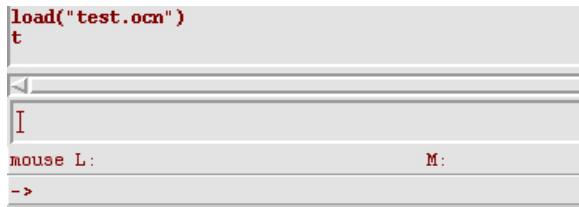
generate timing for all outputs with respect to all inputs so in the case of a 2-input NAND you'd need timing from **A** → **Y** and from **B** → **Y**. Each time you would modify the schematic and generate a new netlist to simulate with the script. You also need to keep track of whether the gate is acting like an inverter (**negative\_unate** timing), or as a buffer (**positive\_unate** timing) for this simulation.

In order to make sure that I'm simulating the right thing, I'll need a **config** view of the **test\_setup** schematic, just like you've done before. In the **config** view you will want to specify the **analog\_extracted** view of whatever cell you're using as the DUT in this simulation run. In Figure 7.25 you can see the **config** view of my **test\_setup** schematic with the **analog\_extracted** view of my inverter selected.

To generate the analog netlist open the **test\_setup** schematic and then open the **Analog Environment**. Make sure that you have the **config** view selected (use **Setup** → **Design** if it's not correct). Here's an annoying part of the process: you can't generate the netlist unless all the variables in your schematic have initial values even though you're going to override those defaults when you run the parametric simulation.

To set the variables to a starting value use the **Variables** → **Copy From Cellview** option. This will fill in the design variables pane of the **Analog Environment**. Double click on the variable names and set them to some value. In Figure 7.26 I've set slope to **500ps** and load to **500ff**. Once things look like Figure 7.26 you can use **Simulation** → **netlist** → **Create Final** to create the final netlist. All this does is create the netlist. The **ocean** script will actually run the simulation for you. You can dismiss the text window that pops up to show you the netlist that was just created.

*Note that because we're going to simulate this ourselves, we'll go ahead and use SpectreS as described in Chapter 6. That is, we don't need to reset the simulator as we did for SignalStorm.*

Figure 7.26: Setting variables in the **Analog Environment**Figure 7.27: Loading the **test.ocn** script

Once you have the netlist generated you need to load and run the ocean script. The script is named **test.ocn** and is in the class directory

```
/uusoc/facility/cad_common/local/class/6710/cadence
```

You should copy this script to your own directory. You might want to take a look at the file to see what it does. You might also want to look at the first section to see if there are any things you would like to change about how the script does its stuff. You might have different names for the parametric variables, for example. Or you might want to use a different set of values for the range of the parametric variables, or to use more or fewer values in those lists depending on the table templates you've defined.

Once you have things set up in the script the way you want them you need to load the script so that you can execute the new functions. You do this by typing the load command into the CIW. The command is

<code>load("test.ocn")</code>	or	<code>(load "test.ocn")</code>
-------------------------------	----	--------------------------------

depending on whether you like C-syntax or Lisp syntax. See Figure 7.27 for an example of what you should see in the CIW. The **t** just means that you've loaded the script successfully.

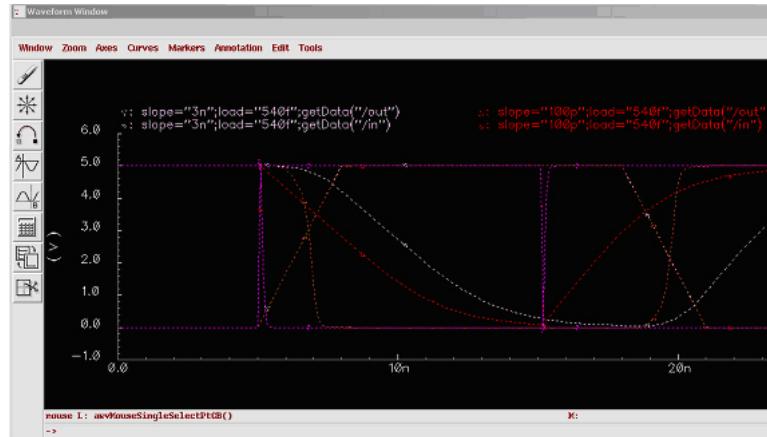


Figure 7.28: Plot output from a **test.ocn** simulation

Once you've loaded the script you can type the command into the CIW to perform the parametric simulation. The command is:

`run_test( ) or (run_test)`

This function actually has an optional argument of the stop time for the transient analysis. This defaults to the value in the script, but if you want to change it you can include it in the call to **run\_test**:

`run_test( "80n" ) or (run_test "80n")`

What you'll see in response is a bunch of text in the CIW output pane for each of the parametric transient simulations. You might want to try this with only a couple values for load and for slope so that it doesn't take so long to run the simulations just to make sure that everything is working. You can easily restore the **loadlist** and **slopelist** to their full values and run it again once you get things working. If you do make changes to the **test.ocn** file, remember to load it again so that the changes will take effect!

Once the parametric simulation is complete, you'll get a plot window that shows all the parametric in and out waveforms. This is just for you to look at (and if you're annoyed by it you can comment out the **plot** line in the script.) Without the **ocean** script you could use the A and B measurement bars to measure the various timings, but the script can do this for you. An example plot is shown in Figure 7.28.

Once you've run the tests, you can print the results in **.lib** format into a file of your own choosing. The directory that this file goes into is set in the script (in this example the output file is in **IC\_CAD/cadence/simulation/test\_setup/<filename>**). The **fprint\_results** function takes two arguments: the first is a string which is the name of the file to put the results into, and the second is either '**neg**' for

```

cell_rise(lu5x5) {
    values( \
    ".145649, .461407, .821977, 1.54147, 2.95759" ,\
    ".384865, .72295, 1.07269, 1.79433, 3.21885" ,\
    ".60943, 1.02507, 1.38113, 2.08612, 3.51231" ,\
    "1.0132, 1.55007, 1.98756, 2.70889, 4.09905" ,\
    "1.65337, 2.35774, 2.89704, 3.77152, 5.1905" );}

rise_transition(lu5x5) {
    values( \
    ".148306, .610152, 1.14291, 2.2072, 4.34568" ,\
    ".305532, .700173, 1.17722, 2.21651, 4.35869" ,\
    ".469617, .894771, 1.35714, 2.32296, 4.38333" ,\
    ".743882, 1.26289, 1.70498, 2.63657, 4.52904" ,\
    "1.20801, 1.84408, 2.32966, 3.29812, 5.05465" );}

cell_fall(lu5x5) {
    values( \
    ".185292, .602238, 1.07644, 2.02688, 3.91734" ,\
    ".432254, .852127, 1.3175, 2.2643, 4.14154" ,\
    ".683672, 1.15366, 1.61688, 2.5389, 4.4227" ,\
    "1.14297, 1.72898, 2.2261, 3.14691, 4.99237" ,\
    "1.89231, 2.63401, 3.23074, 4.2367, 6.03552" );}

fall_transition(lu5x5) {
    values( \
    ".167624, .72171, 1.355, 2.62284, 5.13434" ,\
    ".309461, .78558, 1.37218, 2.62191, 5.10869" ,\
    ".474777, .961437, 1.49416, 2.65833, 5.13505" ,\
    ".780225, 1.2901, 1.82791, 2.9399, 5.23852" ,\
    "1.24947, 1.86456, 2.44295, 3.52342, 5.67453" );}

```

Figure 7.29: Results of running **test.ocn** on an inverter and printing as **.lib** tables

a **negative\_unate** timing (i.e. an inverting gate) or ‘**pos**’ for a **positive\_unate** measurement.

This outputs the data into tables that are in the right format for pasting into your **.lib** file. An example of running this script on an **X1** inverter is shown in Figure 7.29. You can run this script for each of the input/output combinations and paste the results into the timing section of the **.lib** file for each of those pins if you have more general circuits.

This is a home-grown method for deriving lookup table timings for library cells. With SignalStorm it may be less useful than it was, but it may still be useful for odd cells, or for understanding how to script and parameterize analog simulations in general.

## 7.4 Converting Liberty to Synopsys Database (db) Format

Once you have generated a **liberty** formatted file that describes your cells you should be done because this is the most common industry standard format for describing cell timing and power data. But, even though this is a format that originally came from **Synopsys**, the **Synopsys** tools actually require a compiled format that is compiled from this **liberty** text description. The **Synopsys library compiler** tool can easily compile the **.lib** file into this **.db** format, but unfortunately, and for reasons I don't understand, the **.lib** file that is generated by **SignalStorm** actually needs a little hand-modification to be completely compatible with **Synopsys**! Annoying but true.

Start with the **.lib** file that you got from **SignalStorm** and make the following changes. If you've written your own **.lib** file then you may have other issues of course, but you also need to pay attention to these.

- There are errors in the compilation process if a pin has a **fall\_capacitance\_range** parameter, but not a **rise\_capacitance\_range**. One quick fix is just to take out all **[rise/fall].capacitance\_range** lines. This won't cause a huge difference. These lines describe the range of capacitance a pin will see during signal transitions. Alternatively, you could add the missing **[rise/fall].capacitance\_range** statement for each pin that has one but not the other. For the missing range you can use the single value for the pin capacitance for the top and bottom of the range.
- If you remove all those attributes, you may get warnings about not having them, but you can ignore them. Look at the warnings. They tell you that the compiler will assume that the range is just equal to the one measured value, which should be fine.
- If you have negative delays in any of your timing tables (I had them for **disable timings** for the **eninv** when the input rise times were slow, for example) you should "round" those negative timings to 0.
- If you want to avoid warinings about not having default values set you should include the following lines in the technology header section of your **.lib** file:

*Instructions for  
compiling .lib into .db  
are coming up in this  
section*

```
/* Default attributes */  
  
/* Fanout (in terms of capacitive load units) */  
default_fanout_load : 0.3 ;
```

```
default_max_fanout : 10.0 ;  
  
/* Pin Capacitance */  
default inout pin cap : 0.00675 ;  
default input pin cap : 0.00675 ;  
default output pin cap : 0.0 ;  
  
/* leakage power */  
default cell leakage power : 0.0 ;  
default leakage power density : 0.0 ;
```

- If you want to include the wire load models, include the text from Figures 7.3 and 7.4 in the tech header section of your **.lib** file.

Once you have a correct (and correctly modified) **.lib** file you can compile it into a **Synopsys** synthesis database (**.db**) file. To do this you need to run the **Synopsys design compiler** synthesis tool, but only in a very simple way. The command to start **design compiler** or, more precisely the command-line shell version called **dc\_shell** is with the command

**syn-dc**

This will start up **design compiler** in a mode where you can type in commands to the **design compiler** shell. What you want to do is read in the library in **.lib** format, and write out that library in **.db** format. The first command that you type to the shell is the **read\_lib** command to read the **.lib** file. It looks like (dc\_shell-xg-t> is the **design compiler** shell prompt:

**dc\_shell-xg-t> read\_lib <filename>.lib**

where, of course, **<filename>.lib** is the name of your **.lib** file. This will issue a bunch of warnings about defining new variables and perhaps about missing range attributes. You should look carefully at this stage to make sure there are no errors! If you are satisfied that there are only warnings and that the warnings are not important, you can then write out that library in **.db** format with the **write\_lib** command. The name of the library is the name you defined in the **library(<name>)** command at the top of your **.lib** file.

**dc\_shell-xg-t> write\_lib <libname> -o <libname>.db**

The **-o** switch lets you specify whatever file name you like for the output file, but your life will be easier if you name it with a **.db** extension.

If this finishes without error, you can exit with the **exit** command to the **design compiler** shell. You now (hopefully) have a correctly formated binary database for your cell library that **Synopsys** can use in its synthesis procedure.

# Chapter 8

## Verilog Synthesis

**S**YNTHESIS is the process of taking a behavioral Verilog file and converting it to a structural file using cells from a standard cell library. That is, the behavior that is captured by the Verilog program is *synthesized* into a circuit that behaves in the same way. The synthesized circuit is described as a collection of cells from the *target cell library*. This Verilog file is known as **structural** because it is strictly structural instantiations of cells. It is the Verilog text equivalent of a schematic. This structural file can be used as the starting point for the backend tools which will place those cells on the chip and route the wire connections between them.

There are three different behavioral synthesis tools that are usable in our flow, and one schematic-based helper application. They are:

**Synopsys Design Compiler:** This is the flagship synthesis tool from Synopsys. It comes in two flavors: **dc\_shell** which has a TCL shell-command interface and **design\_vision** which is a gui window/menu driven version. The **dc\_shell** version is often driven by writing scripts and executing those scripts on new designs.

*TCL is Tool Command Language and is a standard syntax for providing input commands to tools.*

**Synopsys Module Compiler:** This is a specialty synthesis tool from Synopsys that is specifically for synthesis of arithmetic circuits. It has more knowledge of different arithmetic circuit variants than **design compiler**, including many fast adder variants and even floating point models. It also has its own circuit specification language that you can use to describe complex arithmetic operations. It uses the same target library database at **design compiler**.

**Cadence Build Gates:** This is the primary synthesis tool from Cadence. It uses the **.lib** file directly as the cell database and it is also usually driven from scripts.

**Cadence to Synopsys Interface (CSI):** This is a tool integrated with the Composer schematic capture tool that lets you take schematics that use a combination of standard cells and behavioral Verilog (wrapped in Composer symbols) and output that schematic as a structural Verilog file.

## 8.1 Synopsys dc\_shell Synthesis

*Add some intro stuff here*

### 8.1.1 Basic Synthesis

In order to make use of Synopsys synthesis you need (at least) the following files:

**.synopsis\_dc.setup:** This is the setup file for **design compiler** and **module compiler**. It can be copied from

/uusoc/facility/cad\_common/local/class/6710/synopsis

Note that it has a dot as the first character in the file name. You should copy this file into the directory from which you plan to run the Synopsys tools.

**a cell database file:** This is the binary **.db** file that contains the cell information for the target library that you would like your behavior compiled to. Details of cell characterization and file formats for characterized libraries are in Chapter 7. It's helpful if you either have this file in the directory from which you are running Synopsys, or make a link to it in that directory if it really lives elsewhere.

**A behavioral Verilog file:** This is the file whose behavior you would like to synthesize into a circuit. It can contain purely behavioral code, or a mixture of behavioral and structural (cell instantiations) code. This should also be either in or linked to the directory you use for running Synopsys

If you have these three files you can run very basic synthesis using a class script. This script, named `beh2str`, converts a behavioral Verilog file into a structural Verilog file in a very simplified way. It does no fancy optimization, and it only works for a single behavioral Verilog file as input (no multiple-file designs). It's not really designed for final synthesis, it's just designed as a quick and dirty script that you can use to check and see how things are working, and for initial investigations into small designs. To

```
/* Behavioral Model of an Inverter */
module INV_Test(INV_in,INV_out);
    input INV_in;
    output INV_out;
    assign INV_out = ~INV_in;
endmodule
```

Figure 8.1: Verilog behavioral description of an inverter

use the script you don't need to know anything about what **design compiler** is actually doing. For more advanced synthesis you'll want much more direct control. But, this is a good introduction to the process of synthesis, and a good way to see what the synthesis engine does to different ways of expressing things in Verilog.

### Tiny Example: An Inverter

As an example, consider the extremely basic piece of Verilog code in Figure 8.1 which describes the behavior of a single inverter. I'll start by making a new directory in which to run the synthesis tools. I'll call it **\$HOME/IC\_CAD/synth** and I'll put the file from Figure 8.1 in that directory. I'll also put a copy (or a link) of a **.db** file of a target library in that directory. (I'll use **UofU\_Digital\_v1\_1.db** but you can use a file which describes your own cell library). Finally I'll put a copy of the **.synopsys\_dc.setup** file in this directory. If you make a directory called **WORK** in your **synth** directory, then some of the files that get generated in the synthesis process will go there instead of messing up your **synth** directory. I recommend doing this.

Once I have all these files in my **synth** directory I can connect to that directory and fire up the basic synthesis script. The usage information is shown in Figure 8.2. The command I'll use is:

```
beh2str inv-behv.v inv-struct.v UofU_Digital_v1_1.db
```

This will fire up **design compiler** with the right arguments and produce a file called **inv-struct.v** as output. This output file is seen in Figure 8.3. As you might hope, the tool has synthesized the behavior of the inverter into a single inverter cell from the target library.

### Small Example: A Finite State Machine

As a (very) slightly larger example, consider the Verilog description of a simple four-state finite state machine in Figure 8.4. This FSM description

```
> behv2str

CORRECT>beh2str (y|n|e|a)? yes
beh2str - Synthesizes a verilog RTL code to a structural code
           based on the synopsys technology library specified.
Usage   : beh2str f1 f2 f3
           f1 is the input verilog RTL file
           f2 is the output verilog structural file
           f3 is the compiled synopsys technology library file
```

Figure 8.2: Usage information for the **beh2str** script

```
module INV_Test ( INV_in, INV_out );
    input INV_in;
    output INV_out;

    invX1 U2 ( .A(INV_in), .Y(INV_out) );
endmodule
```

Figure 8.3: Synthesized inverter using a cell from the target library

uses parameters to define state encodings, and defines a state register in an **always** statement. The register will have an active-high clock and an active-low asynchronous clear. The state transition logic is defined with a case statement, and the output is defined with a continuous assignment.

If this state machine is synthesized using the **beh2str** script with the command

```
beh2str moore.v moore-struct.v UofU_Digital.v1_1.db
```

this results in the structural file shown in Figure 8.5. Note that internal wires have been defined by the synthesis procedure. Also note that for unknown reasons the synthesis procedure choose to use a **dff.qb** cell for state register bit 1 (**state\_reg\_1\_**) even though it didn't use the **QB** output. I have no explanation for this, other than that the **beh2str** script is extremely basic and doesn't apply many optimizations.

### 8.1.2 Scripted Synthesis

If you look “under the hood” of the **beh2str** script you find that it is a wrapper that calls the much more general **Synopsys dc shell** interface of **design compiler** with a very simple script. The script, shown in Figure 8.2, shows a very basic version of a general synthesis flow. All **Synopsys** scripts are

```

module moore (clk, clr, insig, outsig);
    input clk, clr, insig;
    output outsig;

    // define state encodings as parameters
    parameter [1:0] s0 = 2'b00, s1 = 2'b01, s2 = 2'b10, s3 = 2'b11;

    // define reg vars for state register and next_state logic
    reg [1:0] state, next_state;

    //define state register (with asynchronous active-low clear)
    always @(posedge clk or negedge clr)
    begin
        if (clr == 0) state = s0;
        else state = next_state;
    end

    // define combinational logic for next_state
    always @(insig or state)
    begin
        case (state)
            s0: if (insig) next_state = s1;
                 else next_state = s0;
            s1: if (insig) next_state = s2;
                 else next_state = s1;
            s2: if (insig) next_state = s3;
                 else next_state = s2;
            s3: if (insig) next_state = s1;
                 else next_state = s0;
        endcase
    end

    // now set the outsig. This could also be done in an always
    // block... but in that case, outsig would have to be
    // defined as a reg.
    assign outsig = ((state == s1) || (state == s3));

endmodule

```

Figure 8.4: Simple State Machine

```

module moore ( clk, clr, insig, outsig );
    input clk, clr, insig;
    output outsig;
    wire n6, n7, n8, n9;
    wire [1:0] next_state;

    dff state_reg_0_ (.D(next_state[0]), .G(clk), .CLR(clr), .Q(outsig) );
    dff_qb state_reg_1_ (.D(next_state[1]), .G(clk), .CLR(clr), .Q(n6) );
    mux2_inv U7 ( .A(n7), .B(outsig), .S(n6), .Y(next_state[1]) );
    nand2 U8 ( .A(outsig), .B(insig), .Y(n7) );
    xor2 U9 ( .A(insig), .B(n8), .Y(next_state[0]) );
    nor2 U10 ( .A(n6), .B(n9), .Y(n8) );
    invX1 U11 ( .A(outsig), .Y(n9) );
endmodule

```

Figure 8.5: Result of running **beh2str** on **moore.v**

written in a scripting language called **Tool Command Language** or **TCL**. **TCL** is a standard language syntax for writing tool scripts that is used by most CAD tools, and certainly by most tools from **Synopsys** and **Cadence**. There is a basic **TCL** tutorial linked to the class web site, and eventually there will be a written version in an appendix to this text.

By looking at the **beh2str** script you can see the basic steps in a synthesis script. You can also see some basic **TCL** syntax for **dc\_shell**. Variables are set using the **set** keyword. Lists are generated with a **list** command, or built by concatenating values with the **concat** command. **UNIX** environment variables are accessed using the **getenv** command. Other commands are specific to **dc\_shell**. There are hundreds of variables that control different aspects of the synthesis process, and about as many commands. The **beh2str** script uses just about the bare minimum of these commands. The more advanced script shown later in this chapter uses a few more commands, but even that only scratches the surface of the opportunities for controlling the synthesis process.

The basic steps in **beh2str** are:

1. Inform the tools which cell libraries you would like to use. The **target\_library** is the cell library, or a list of libraries, whose cells you would like to use in your final circuit. The **link\_library** is a list of libraries that the synthesis tool can use during the linking phase of synthesis. This phase resolves design references by linking the design to library components. As such it includes libraries that contain more complex primitives than are in the **target\_library**. The **synthetic\_library** is set in the **.synopsys\_dc.setup** file and points to a set of high level macros provided by **Synopsys** in their **DesignWare** package.
2. Read in the behavioral Verilog.
3. Set constraints and provide other information that will guide the synthesis. In general this section will have many more commands in a more advanced script. In this basic script the only constraint is to set a variable that forces the structural circuit to have buffer circuits for nets that might otherwise be implemented with a simple **assign** statement (i.e. nets with no logic that just pass through a module). Some downstream tools don't consider **assign** statements, even those with only a single variable on the right hand side, to be structural.
4. Compile the behavioral Verilog into structural Verilog using the constraints and conditions, and using the libraries specified earlier in the script. In this example hierarchy is flattened with the **ungroup\_all** command.

```

# beh2str script
set target_library [list [getenv "LIBFILE"]]
set link_library [concat [concat "*" $target_library] $synthetic_library]

read_file -f verilog [getenv "INFILE"]

/* This command will fix the problem of having          */
/* assign statements left in your structural file.      */
set_fix_multiple_port_nets -all -buffer_constants           */

compile -ungroup_all
check_design

/* always do change_names before write...             */
redirect change_names { change_names -rules verilog -hierarchy -verbose }

write -f verilog -output [getenv "OUTFILE"]
quit

```

Figure 8.6: **beh2str.tcl** basic synthesis command script

5. Apply a **dc\_shell** rewriting rule that makes sure that the output is correct Verilog syntax
6. Write the synthesized result to a structural Verilog file.

These steps are the basic synthesis flow that is used for this simple synthesis script, and for a more complex script. Of course, you could start up the **dc\_shell** command line interface and type each of these commands to the shell one at a time, but it's almost always easier to put the commands in a script and execute from that script. The command to start the **dc\_shell** tool with the command line interface is

**syn-dc**

All arguments that you give to that command will be passed through to the **dc\_shell** program. So, if you wrote a script of your own synthesis commands you could execute that with the following command.

**syn-dc -f <scriptname>**

You can use **syn-dc -help** for a usage message, and if you start the tool with **syn-dc** you can type **help** at the shell prompt for a long list of all available commands. This shell accepts commands in **TCL** syntax, and each command generally has a help message of its own. So, for example, typing **write\_file -help** to **dc\_shell** will give detailed documentation on the **write\_file** command.

The basic sequence of a generic **Synopsys** synthesis flow, along with commands to consider at each step, is as follows (this is a more elaborate version of what you saw in Figure 8.6). Note that you don't have to write

these scripts from scratch. There is a class example of a relatively full-featured synthesis script that you can use.

**Write behavioral Verilog:** First, of course, you need to develop your design as a set of Verilog files. The description can use a combination of behavioral and structural Verilog, but remember that you can only use the “synthesis subset” of Verilog, and any structural references must come from your eventual **target library** or from a Synopsys-provided library such as the **DesignWare** cells (more about them later).

**Specify Paths and Libraries:** These are paths and libraries that Synopsys uses.

**search\_path:** This is the search path that **design compiler** uses to search for libraries and other files and it is set in the **.synopsys\_dc.setup** file. You can override that in your script by setting the **search\_path** variable there. At a minimum you probably want this path to contain `.` (your current directory) and the `/libraries/syn` directory in the **Synopsys** installation directory for the generic libraries. You should also include any other directories that hold database files of interest to your synthesis.

**target\_library:** This is the library, or list of libraries, that you want **design compiler** to target as the result of the synthesis. It’s your cell library in **.db** format.

**synthetic\_library:** This is the list of libraries that contain information about pre-defined structures. The most common example is the **DesignWare** libraries from **Synopsys** that contain information about a host of datapath structures that **design compiler** can use. Include at least **dw.foundation.sldb** on this list.

**link\_library:** A list of libraries that you want your design linked to. This is typically a list of `*` (meaning your own module descriptions in your Verilog code), your **target\_library** list and your **synthetic\_library** list.

**symbol\_library:** This is a list of libraries that have graphical symbol information for showing the gates of the design in the graphical design tool **design vision**. This is typically just the built-in **Synopsys** generic library **generic.sdb**.

**Read in the Verilog code:** If you have a single Verilog file you can read it with the one-step **read\_file** command, but in general (and required if you have more than one Verilog file in your input) you should use the sequence of **analyze** and **elaborate**.

**analyze -format verilog -lib WORK <files>** This command parses all the input files and puts the semi-digested versions into a directory called **WORK**.

**elaborate <top-module-name> -lib WORK -update** This command compiles the input files into an intermediate technology independent internal form. As part of this step all the inferred memory devices are discovered and reported.

**Set the Operating Environment:** In this phase you tell **design compiler** which operating conditions, wire load models, etc to use from the **.db** file, and also define the input and output drive expected to and from the module. Typically the operating conditions (worst, typ, best) are set in the **.db** file so you don't have to change it, as is the wire load model. If your **.lib** (and thus your **.db**) file has multiple operating conditions defined in it, here's your chance to pick one.

Other commands to consider at this point are: **set\_drive**, **set\_driving\_cell**, **set\_load**, **set\_load\_cell**, and **set\_fanout\_load**. You can get details of the syntax of these commands from the **design compiler** shell. There are also examples in the class generic script. If you don't set the driving cell or the drive **design compiler** will assume there is infinite drive available on the inputs. This may or may not be what you want to assume. I typically set the input driving cell to be a 4X inverter, and the output load to be driving the input of a 4x inverter. The D and Q signals from a DFF are another good choice.

**Set the Design Constraints:** This is where you tell **design compiler** how fast you want the synthesized circuit to run, how big it should be, and other constraints. This is a critical section. It's where you set speed and area goals for synthesis and it determines how hard **design compiler** tries to optimize things. Commands include: **create\_clock**, **set\_clock\_latency**, **set\_propagated\_clock**, **set\_clock\_uncertainty**, **set\_clock\_transition**, **set\_input\_delay**, **set\_output\_delay**, and **set\_max\_area**.

The most important of these commands are:

**create\_clock** This is the command to use to set your speed goal for the synthesis. If you have a clock signal in your design, use that signal as the clock signal (which should be the obvious thing to do). The period you set is the speed goal that **design compiler** tries to hit. Think about this carefully! Too aggressive a speed goal will cause **design compiler** to spend a *long* time trying to meet the goal and then failing. Too conservative a goal will be easily met, but with a very conservative design. You can also set a **virtual** clock if your design is combinational. This is a name you use for a fake clock that is used just to set the speed goal for synthesis. See the example of this command in the class script.

**set\_max\_area** This sets the area goal. Speed is always the primary goal for synthesis. But, after speed is achieved, **design compiler** will try to optimize for a smaller circuit. This is one situation where it's common to set the max area goal to 0 to force **design compiler** to try to make as small a design as possible.

**Compile (synthesize/optimize) your design:** In this step you call the **compile** command to synthesize your circuit, subject to your constraints. The newest version of **design compiler** has a *mega command* called **compile\_ultra** which runs through a Synopsys-approved set of compilation procedures. You should probably use this one unless you have a good reason not to. The other choice is the plain **compile** command which has lots of switches you can read about in **design compiler** documentation.

**Analyze and report results:** The **check\_design** command will check the result to make sure nothing funny has happened. You can also report the area, timing, power, and other results as analyzed by **design compiler**. The commands you might use here are:

**write -format verilog** This command will write out the synthesized structural Verilog. Before you issue this command you should always issue the **change\_names -rules verilog** command which will make sure that correct Verilog syntax is used. Why this isn't an automatic part of the **write** command I don't know.

**write.rep** This generates a synthesis report that describes (among other things) the critical path of the design and whether the synthesis has achieved the speed target.

**write\_ddc** This writes a Synopsys formatted binary database file that you can read in to either **design compiler** or **design vision** for further processing.

**write\_sdf** This writes a *standard delay format* file that you can use to back-annotate your simulations with extracted timings from the synthesized circuit.

**write\_sdc** This writes a constraints file that is used to pass the constraints that you set in your synthesis script on to other tools like the place and route tool. It's especially important for the clock tree synthesis phase of the place and route tool.

**write\_pow** This writes a report file that describes the power your design will dissipate as best as **design compiler** can tell.

A much more advanced script that demonstrates this flow is available in the **/uusoc/facility/cad\_common/local/class/6710/synopsis** directory. It's called **syn-script.tcl**. This is a much more general script for synthesis that

you can use for many of your final synthesis tasks, and as a starting point if you'd like to use other features. The script is shown in three separate Figures, but should be kept in a single file for execution. The first part of the script, shown in Figure 8.7, is where you set values specific to your synthesis task. Note that this script assumes that the following variables are set in your **.synopsis\_dc.setup** file (which they will be if you make sure that you've linked the class version to your synthesis directory):

**SynopsysInstall:** The path to the main synopsys installation directory

**synthetic\_library:** The path to the **DesignWare** files

**symbol\_library:** The path to a library of generic logic symbols for making schematics

In this first part of the **syn-script.tcl** file you need to modify things for your specific synthesis task. You should look at each line carefully, and in particular you should change everything that has “!!” in front and back to the correct values for your synthesis task. Some comments about the things to set follow:

- You need to set the name of your **.db** file as the **target\_library**, or make this a list if you have multiple libraries with cell descriptions.
- You also need to list all of your Verilog behavioral files. The examples have all been with a single Verilog file, but in general a larger design will most likely use multiple files.
- The **basename** is the basename that will be used for the output files. An extra descriptor will be appended to each output file to identify them.
- **myclk** is the name of your clock signal. If your design has no clock (i.e. it's combinational not sequential) you can use a **virtual clock** for purposes of defining a speed target. Synopsys uses the timing of the clock signal to define a speed goal for the synthesis. A **virtual clock** is a name not attached to any wire in your circuit that can be used for this speed goal if you don't actually have a clock.
- The **useUltra** switch defines whether to use “ultra mode” for compiling or not. Unless you have very specific reasons to drive the synthesis directly, “ultra mode” will probably give you the best results.
- The timing section is where you set speed goals for the synthesis. The numbers are in ns. A period of **10** would set a speed goal of 100MHz, for example.

```


/*
 * search path should include directories with memory .db files */
/* as well as the standard cells */
set search_path [list . \
[format "%s%s" SynopsysInstall /libraries/syn] \
[format "%s%s" SynopsysInstall /dw/sim_ver] \
!!your-library-path-goes-here!!]

/*
 * target library list should include all target .db files */
set target_library [list !!your-library-name!!]

/*
 * synthetic_library is set in .synopsys_dc.setup to be */
/* the dw_foundation library. */
set link_library [concat [concat "*" $target_library] $synthetic_library]

/*
 * below are parameters that you will want to set for each design */

/*
 * list of all HDL files in the design */
set myfiles [list !!all-your-files!!]
set fileFormat verilog ;# verilog or VHDL
set basename !!basename!! ;# choose a basename for the output files
set myclk !!clk!! ;# The name of your clock
set virtual 0 ;# 1 if virtual clock, 0 if real clock

/*
 * compiler switches...
 * set useUltra 1 ;# 1 for compile_ultra, 0 for compile
 * #mapEffort, useUngroup are for
 * #non-ultra compile...
 * set mapEffort1 medium ;# First pass - low, medium, or high
 * set mapEffort2 medium ;# second pass - low, medium, or high
 * set useUngroup 1 ;# 0 if no flatten, 1 if flatten

/*
 * Timing and loading information */
set myperiod_ns !!10!! ;# desired clock period (sets speed goal)
set myindelay_ns !!0.5!! ;# delay from clock to inputs valid
set myoutdelay_ns !!0.5!! ;# delay from clock to output valid
set myinputbuf !!invx4!! ;# name of cell driving the inputs
set myloadcell !!UofU_Digital/invX4/A!! ;# name of pin that outputs drive
set mylibrary !!UofU_Digital!! ;# name of library the cell comes from

/*
 * Control the writing of result files */
set runname struct ;# Name appended to output files

/*
 * the following control which output files you want. They */
/* should be set to 1 if you want the file, 0 if not */
set write_v 1 ;# compiled structural Verilog file
set write_db 0 ;# compiled file in db format (obsolete)
set write_ddc 0 ;# compiled file in ddc format (XG-mode)
set write_sdf 0 ;# sdf file for back-annotated timing sim
set write_sdc 0 ;# sdc constraint file for place and route
set write_rep 1 ;# report file from compilation
set write_pow 0 ;# report file for power estimate


```

Figure 8.7: Part 1 of the **syn-script.tcl** synthesis script

- The other delays define how inputs from other circuits and outputs to other circuits will behave. That is, how will this synthesized circuit connect to other circuits.
- The **myinputbuf** variable should be set to the name of the cell in your library that is an example of what would be driving the external inputs to your circuit, and the **myloadcell** variable should be the name of the pin on a cell in your library that represents the output load of an external output. The example in Figure 8.7 references the **A** input of the **invX4** inverter in the **UofU\_Digital** library (also defined as the **target\_library**). These must be names of cells and cell inputs in one of your target libraries.
- The **write** flags define which outputs should be generated by the synthesis process. You almost certainly want to generate a structural Verilog file. You will usually also want at least a report file for timing and area reports. The other output files are used for other phases of the total flow. The **ddc** file is the Synopsys binary database format. You can save the synthesized circuit in **.ddc** format for ease of reading it back in to Synopsys for further processing. The **.sdf** and **.sdc** files are timing and constraint files that are used later in the flow. The power report uses power information in the **.db** file to generate a very rough estimate of power usage by your design.

The second part of the **syn-script.tcl** is seen in Figure 8.8. It contains the synthesis commands that use the variables you set in the first part of the file. You shouldn't have to modify anything in this part of the file unless you'd like to change how the synthesis proceeds. Note that the **read** command from **beh2str** has been replaced with a two-step process of **analyze** and **elaborate**. This is because if you have multiple Verilog files to synthesize, you need to analyze them all first before combining them together and synthesizing them. The other commands are documented in the script. You can see that constraints are set according to your information in part1. Finally the design is compiled, checked, and violations are checked.

The third part of the **syn-script.tcl** file is where the outputs are written out. It's pretty straightforward. Note that when you write the structural Verilog output you also **change\_names** to make sure that the output is in correct Verilog syntax. You'd think that this would be part of the **write** command, but it's not. The output file names are constructed from **basename** and **run-name** which are set in the first part of the file.

```
# analyze and elaborate the files
analyze -format $fileFormat -lib WORK $myfiles
elaborate $basename -lib WORK -update
current_design $basename

# The link command makes sure that all the required design
# parts are linked together.
# The uniquify command makes unique copies of replicated modules.
link
uniquify

# now you can create clocks for the design
if { $virtual == 0 } {
    create_clock -period $myperiod_ns $myclk
} else {
    create_clock -period $myperiod_ns -name $myclk
}

# Set the driving cell for all inputs except the clock
# The clock has infinite drive by default. This is usually
# what you want for synthesis because you will use other
# tools (like SOC Encounter) to build the clock tree
# (or define it by hand).
set_driving_cell -library $mylibrary -lib_cell $myinputbuf \
    [remove_from_collection [all_inputs] $myclk]

# set the input and output delay relative to myclk
set_input_delay $myindelay_ns -clock $myclk \
    [remove_from_collection [all_inputs] $myclk]
set_output_delay $myoutdelay_ns -clock $myclk [all_outputs]

# set the load of the circuit outputs in terms of the load
# of the next cell that they will drive, also try to fix
# hold time issues
set_load [load_of $myloadcell] [all_outputs]
set_fix_hold $myclk

# This command will fix the problem of having
# assign statements left in your structural file.
# But, it will insert pairs of inverters for feedthroughs!
set_fix_multiple_port_nets -all -buffer_constants

# now compile the design with given mapping effort
# and do a second compile with incremental mapping
# or use the compile_ultra meta-command
if { $useUltra == 1 } {
    compile_ultra
} else {
    if { $useUngroup == 1 } {
        compile -ungroup_all -map_effort $mapEffort1
        compile -incremental_mapping -map_effort $mapEffort2
    } else {
        compile -map_effort $mapEffort1
        compile -incremental_mapping -map_effort $mapEffort2
    }
}

# Check things for errors
check_design
report_constraint -all_violators
```

Figure 8.8: Part 2 of the **syn-script.tcl** synthesis script

```

set filebase [format "%s%s" [format "%s%s" $basename "_"] $runname]

# structural (synthesized) file as verilog
if { $write_v == 1 } {
    set filename [format "%s%s" $filebase ".v"]
    redirect change_names \
        { change_names -rules verilog -hierarchy -verbose }
    write -format verilog -hierarchy -output $filename
}

# write out the sdf file for back-annotated verilog sim
# This file can be large!
if { $write_sdf == 1 } {
    set filename [format "%s%s" $filebase ".sdf"]
    write_sdf -version 1.0 $filename
}

# this is the timing constraints file generated from the
# conditions above - used in the place and route program
if { $write_sdc == 1 } {
    set filename [format "%s%s" $filebase ".sdc"]
    write_sdc $filename
}

# synopsys database format in case you want to read this
# synthesized result back in to synopsys later (Obsolete db format)
if { $write_db == 1 } {
    set filename [format "%s%s" $filebase ".db"]
    write -format db -hierarchy -o $filename
}

# synopsys database format in case you want to read this
# synthesized result back in to synopsys later in XG mode (ddc format)
if { $write_ddc == 1 } {
    set filename [format "%s%s" $filebase ".ddc"]
    write -format ddc -hierarchy -o $filename
}

# report on the results from synthesis
# note that > makes a new file and >> appends to a file
if { $write_rep == 1 } {
    set filename [format "%s%s" $filebase ".rep"]
    redirect $filename { report_timing }
    redirect -append $filename { report_area }
}

# report the power estimate from synthesis.
if { $write_pow == 1 } {
    set filename [format "%s%s" $filebase ".pow"]
    redirect $filename { report_power }
}

quit

```

Figure 8.9: Part 3 of the **syn-script.tcl** synthesis script

```
module moore ( clk, clr, insig, outsig );
    input clk, clr, insig;
    output outsig;
    wire n2, n3, n4;
    wire [1:1] state;
    wire [1:0] next_state;

    dff state_reg_0_ (.D(next_state[0]), .G(clk), .CLR(clr), .Q(outsig) );
    dff state_reg_1_ (.D(next_state[1]), .G(clk), .CLR(clr), .Q(state[1]) );
    mux2_inv U3 ( .A(n2), .B(outsig), .S(state[1]), .Y(next_state[1]) );
    nand2 U4 ( .A(outsig), .B(insig), .Y(n2) );
    xor2 U5 ( .A(insig), .B(n3), .Y(next_state[0]) );
    nor2 U6 ( .A(state[1]), .B(n4), .Y(n3) );
    invX4 U7 ( .A(outsig), .Y(n4) );
endmodule
```

Figure 8.10: Result of running **syn-dc** with the **syn-script.tcl** on **moore.v**

### Small Example: A Finite State Machine

As an example, if the **moore.v** Verilog file in Figure 8.4 was compiled with this script, I would set the **target\_library** to **UofU\_Digital.db**, include **moore.v** in the **myfiles** list, use a **basename** of **moore**, a **myclk** of **clk**, and use the ultra mode with **useUltra**. With a speed goal of **10ns** period and the input buffer and output load set as in the example, the result is seen in Figure 8.10. Note that it's almost the same as the simple result in Figure 8.5 (which would expect for such a simple Verilog behavioral description), but there are differences. This version of the synthesis uses the same **dff** cells for both state variables, and the inverter producing the **outsig** output has been sized up to an **invX4**.

During the synthesis procedure a lot of output is produced by **dc\_shell**. You should not ignore this output! You really need to look at it to make sure that the synthesis procedure isn't complaining about something in your behavioral Verilog, or in your libraries!

One place that you really need to pay attention is in the **Inferred memory devices** section. This is in the elaboration phase of the synthesis where all the memory (register and flip flop) devices are inferred from the behavioral code. In the case of this simple finite state machine the inferred memory is described as seen in Figure 8.11. You can see that the synthesis process inferred a **flip-flop** memory (as opposed to a gate latch) with a width of **2**. The other features with Y/N switches define the features of the memory. In this case the **flip-flop** has an **AR** switch which means Asynchronous Reset. The other possibilities are Asynchronous Set, Synchronous Reset, Synchronous Set, and Synchronous Toggle. One reason it's critical to pay attention to the inferred memories is that it is easy to write Verilog code that will result in an inferred memory when you meant the construct to be combinational.

```
Inferred memory devices in process
  in routine moore line 14 in file
    './moore.v'.
=====
| Register Name | Type      | Width | Bus | MB | AR | AS | SR | SS | ST |
=====
| state_reg     | Flip-flop | 2     | Y   | N  | Y  | N  | N  | N  | N  |
=====
```

Figure 8.11: Inferred memory from the **moore.v** example

```
link

Linking design 'moore'
Using the following designs and libraries:
-----
moore                      /home/elb/IC_CAD/syn-f06/moore.db
UofU_Digital_v1_1 (library) /home/elb/IC_CAD/syn-f06/UofU_Digital.db
dw_foundation.sldb (library) /uusoc/.../SYN-F06/libraries/syn/dw_foundation.sldb
```

Figure 8.12: Link information from the **dc\_shell** synthesis process (with **dw\_foundation** path shortened)

Another section of the compilation log that you might want to pay attention to is the **link** information. This tells you which designs and libraries your design has been linked against and lets you make sure that you're using the libraries you want to be using. The **link** information for the **moore.v** example is shown in Figure 8.12. The **dw\_foundation** link library is the **Synopsys DesignWare** library that was defined in the **.synopsys\_dc.setup** file.

The final timing report and area report are both contained in the **moore\_struct.rep** file and are shown in Figures 8.13 and 8.14. The timing report (shown in Figure 8.13) tells you how well **dc\_shell** did in compiling your design for speed. It lists the worst-case timing path in the circuit in terms of the delay at each step in the path. Then it compares that with the speed target you set in the script to see if the resulting design is fast enough. All the timing information is from your cell library in the **.db** file. In the case of the **moore.v** example you can see that the worst case path takes **2.30ns** in this synthesized circuit. The required time set in the script is **10ns**, minus the **0.2ns** setup time defined in the library for the flip flops. So, in order to meet timing, the synthesized circuit must have a worst case critical path of less than **9.98ns**. The actual worst case path according to **dc\_shell** is **2.30ns** so the timing is **met** with **7.69ns** of slack.

If the timing was not met, the slack would tell you by how much you need to improve the speed of the worst case path to meet the timing goal. If I reset the speed goal to a **1ns** period with **0.1ns** input and output delays

to try to make a very fast (1GHz) circuit rerun the synthesis, the critical path timing will be reduced to **1.90ns** because **dc\_shell** works harder in the optimization phase, but that doesn't meet the required arrival time of **0.98ns** so the slack is violated by **-0.98ns**. That is, the circuit will not run at the target speed.

The area report is shown in Figure 8.14. It tells us that the design has four ports: three inputs (insig, clk, and clr), and one output (outsig). There are seven cells used, and 10 total nets. It also estimates the area of the final circuit using the area estimates in the **.db** file. This area does not take placement and routing into account so it's just approximate.

### 8.1.3 Design Vision GUI

A “tall thin” designer is someone who knows something about the entire flow from top to bottom. This is as oppose to a “short fat” designer who knows everything about one layer of the design flow, but not much about the other layers.

Running **dc\_shell** with a script (like **syn-script.tcl**) is the most common way to use the synthesis tool in industry. What usually happens is that a CAD support group designs scripts for particular designs and the Verilog design group will develop the behavioral model of the design and then just run the scripts that are developed by the CAD team. But, there are times when you as a “tall thin” designer want to run the tool interactively and graphically and see what’s happening at each step. For this, you would use the graphical GUI interface to **design compiler** called **design\_vision**.

To start **design compiler** with the **design\_vision** GUI use the **syn-dv** script. This will read your **.synopsys\_dc.setup** file and then open a GUI interface where you can perform the synthesis steps. Each step in the **syn-script.tcl** can be performed separately either by typing the command into the command line interface of **design\_vision** or by using the menus. The main **design\_vision** window looks like that in Figure 8.15.

#### Small Example: A Finite State Machine

Using the same small Moore-style finite state machine as before (shown in Figure 8.4) I can read this design into **design vision** using the **File → Analyze** and **File → Elaborate** menu commands. After the elaboration step the behavioral Verilog has been elaborated into an initial circuit. This circuit, shown in Figure 8.16, is mapped to a generic set of internally defined gates that are not related to any library in particular. It is this step where memory devices are inferred from the behavioral Verilog code. You can see the inferred memory information in the **Log** window.

Now you can set constraints on the design using the menu choices or by typing the constraint-setting commands into the shell. One of the most important is the definition of the clock which sets the speed goal for synthesis.

```
*****
Report : timing
  -path full
  -delay max
  -max_paths 1
Design : moore
Version: Y-2006.06
Date   : Mon Sep 25 15:52:13 2006
*****
```

Operating Conditions: typical Library: UofU\_Digital\_v1\_1  
 Wire Load Model Mode: enclosed

Startpoint: state\_reg\_0\_  
 (rising edge-triggered flip-flop clocked by clk)  
 Endpoint: state\_reg\_0\_  
 (rising edge-triggered flip-flop clocked by clk)  
 Path Group: clk  
 Path Type: max

Des/Clust/Port	Wire Load Model	Library
moore	5k	UofU_Digital_v1_1

Point	Incr	Path
clock clk (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
state_reg_0_/G (dff)	0.00	0.00 r
state_reg_0_/Q (dff)	1.31	1.31 f
U7/Y (invX2)	0.28	1.59 r
U6/Y (nor2)	0.30	1.89 f
U5/Y (xor2)	0.40	2.29 r
state_reg_0_/D (dff)	0.00	2.30 r
data arrival time		2.30
clock clk (rise edge)	10.00	10.00
clock network delay (ideal)	0.00	10.00
state_reg_0_/G (dff)	0.00	10.00 r
library setup time	-0.02	9.98
data required time		9.98
data required time		9.98
data arrival time		-2.30
slack (MET)		7.69

Figure 8.13: Timing report for the **moore.v** synthesis using **syn-script.tcl**

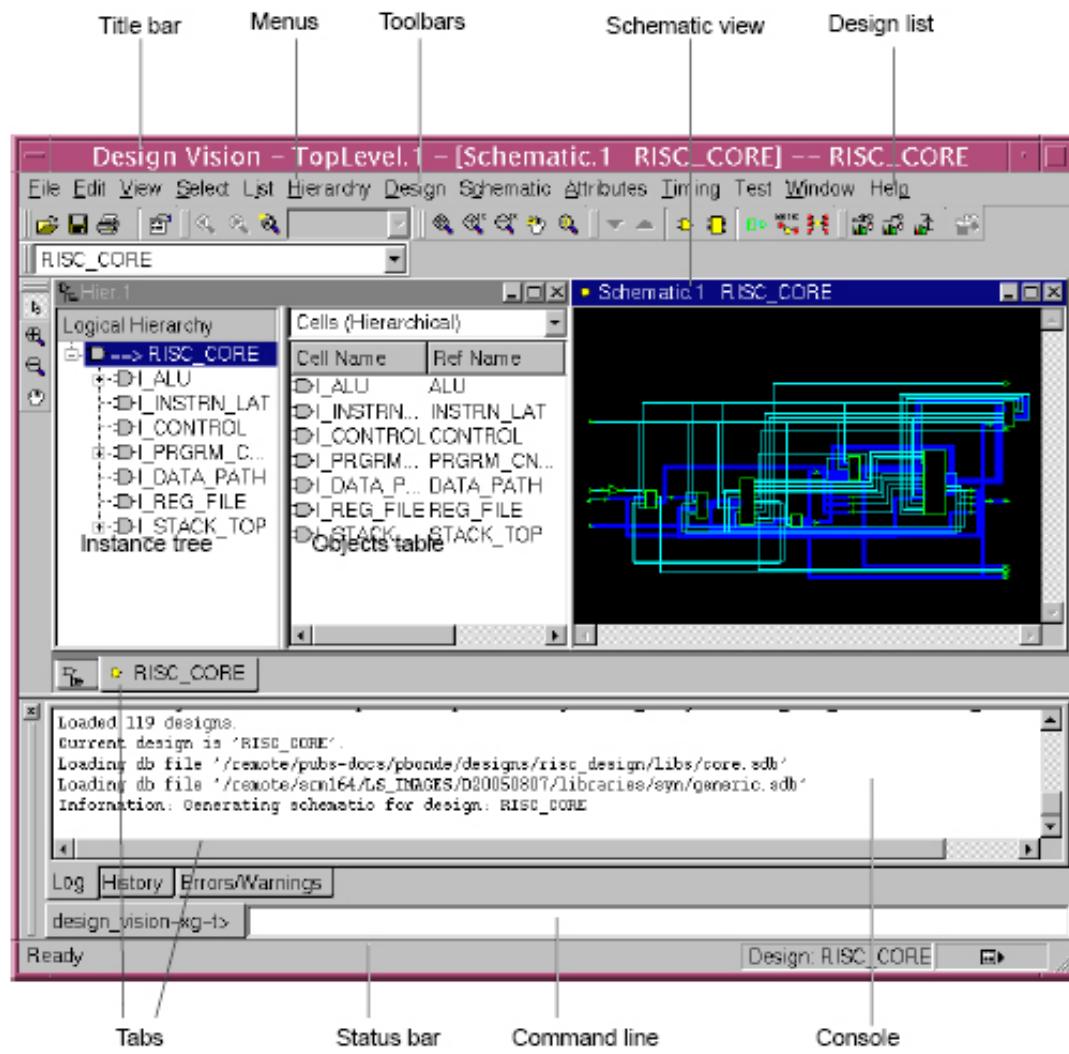
```
*****
Report : area
Design : moore
Version: Y-2006.06
Date   : Mon Sep 25 15:24:12 2006
*****  
Library(s) Used:  
UofU_Digital_v1_1 (File: /home/elb/IC_CAD/syn-f06/UofU_Digital.db)  
  
Number of ports: 4  
Number of nets: 10  
Number of cells: 7  
Number of references: 6  
  
Combinational area: 1231.199951  
Noncombinational area: 1944.000000  
Net Interconnect area: 8470.000000  
  
Total cell area: 3175.199951  
Total area: 11645.200195
```

Figure 8.14: Area report for the **moore.v** synthesis using **syn-script.tcl**

You can set the clock by selecting the **clk** signal in the schematic view, and then selecting **Attributes** → **Specify Clock** from the menu. An example of a clock definition with a period of **10ns** and a symmetric waveform with the rising edge of the clock at **5ns** and the falling edge at **10ns** is shown in Figure 8.17.

After your desired constraints are set, you can compile the design with the **Design** → **Compile Ultra** menu. After the design is compiled and mapped to your target library the schematic is updated to reflect the new synthesized and mapped circuit as seen in Figure 8.18. This file can now be written using the **Edit** → **Save As** menu. If you choose a filename with a **.v** extension you will get the structural Verilog view. If you choose a file name with a **.ddc** extension you will get a Synopsys database file. You can also write out report files with the **Design** → **Report ...** menus.

Perhaps the most interesting thing you can do with **design vision** is use the graphical display to highlight critical paths in your design. You can use the **Timing** → **Report Timing Paths** menu command to generate a timing report for the worst case path (if you leave the paths blank in the dialog box), or for a specific path in your circuit (if you fill them in). You can also obtain timing slack information for all path endpoints in the design using endpoint slack histograms. These histograms show a distribution of the timing slack values for all endpoints in the design giving an overall picture of how the design is meeting timing requirements. Use the **Timing** → **Endpoint Slack** command to generate this window. You can choose how many histogram bins to use. The **endpoint slack histogram** for the **moore** example isn't all

Figure 8.15: General view of the **Design Vision** GUI

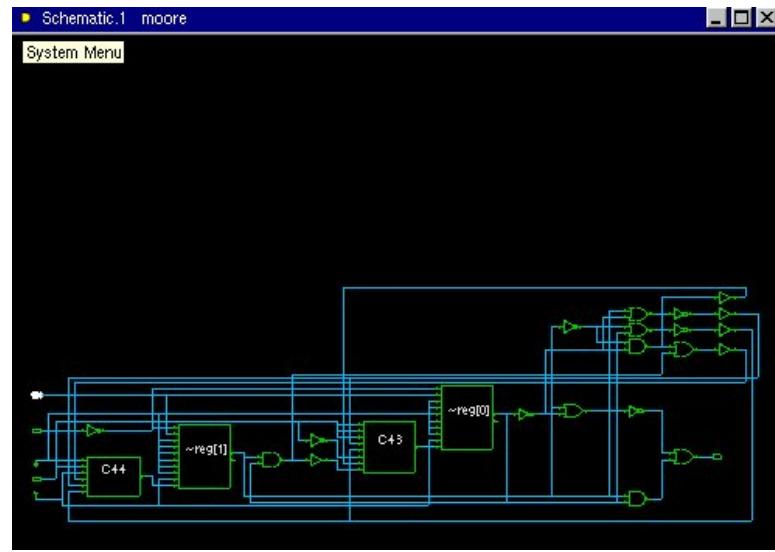


Figure 8.16: Initial mapping of the **moore.v** example

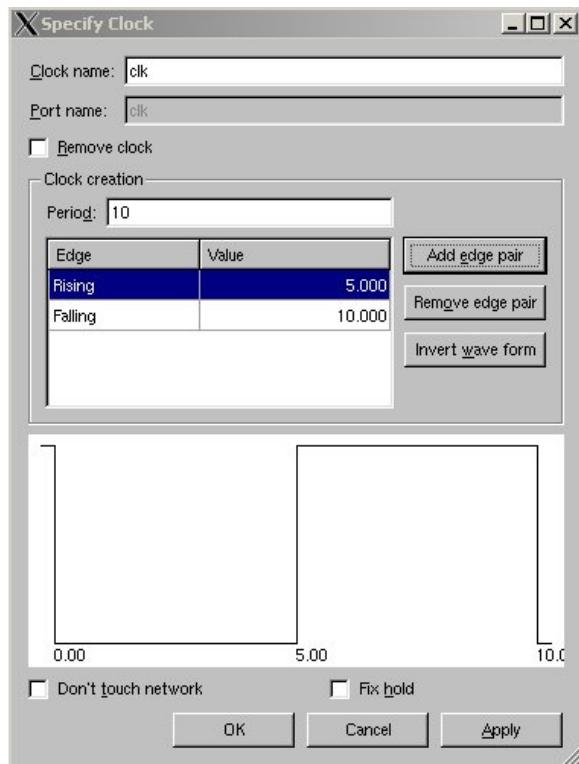


Figure 8.17: Clock definition in **Design Vision**

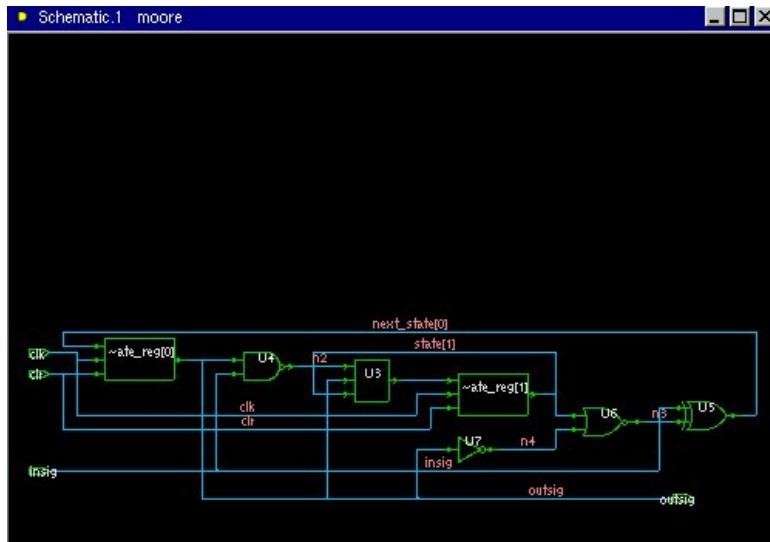


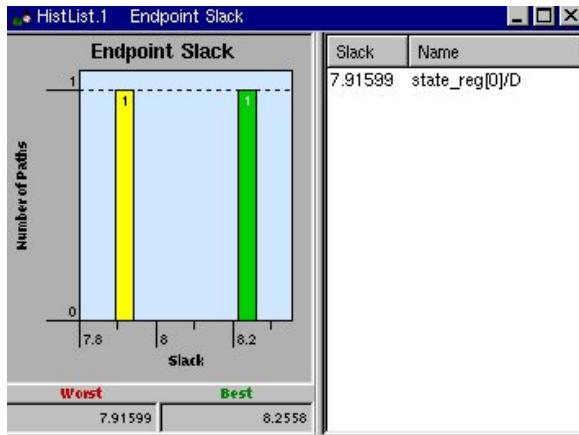
Figure 8.18: Final mapping of the **moore.v** example

that interesting since there are only two paths of interest in the circuit (one ending at each state bit). The result is shown in Figure 8.19. You can see which path corresponds to each bar in the histogram by clicking on it. You can also look at net capacitance, and general path slack (i.e. not ending at points in the circuit. See Figures 8.20 and 8.21 for examples. Clicking on a path in a slack window will highlight that path in the schematic as seen in the Figures. You can also use the **Highlight** menu to highlight all sorts of things about the circuit including the critical path. It's a pretty slick interface. You should play around with it to discover things it can do (especially on more complex circuits than the **moore** machine).

Note that you can also use **design vision** to explore a circuit graphically that has been compiled with a script. Make sure that your script writes a **Synopsys** database file (a **.ddc** file), and then you can fire up **design vision** with **syn-dv** and read in the compiled file as a **.ddc** file. You can then explore the timing and other features of the compiled circuit with **design vision**.

#### 8.1.4 DesignWare Building Blocks

Although you can write Verilog code to describe almost any behavior you want to describe, there are some behaviors and structures that are so common you might ask “Isn’t there a pre-designed version out there somewhere so I don’t have to do it from scratch?” In fact, there are. The **Synopsys DesignWare** package is a large set of library building blocks that are pre-

Figure 8.19: Endpoint slack for the two endpoints in **moore**

designed by **Synopsys** for use with **design compiler**. They are generally quite parameterizable so you can tailor them to your particular application. They are instantiated either by structural instantiation of a particular component from the **DesignWare** library in your Verilog code, or by writing your code in such a way that **design compiler** can easily figure out that it should use a **DesignWare** component for that circuit.

The full set of **DesignWare** building block IP components grows with each release of the **design compiler** tool. The current set includes arbiters, datapath components (adders, subtractors, shifters, incrementers, decrementers, multipliers, dividers, etc.), floating point arithmetic operations, parity and CRC generators, FIR filters, clock-domain crossing circuits, encoders, decoders, counters, FIFOs, and even flip-flop based RAMs. The full set of components can be seen on the **Synopsys** web site at <http://synopsys.com/dw/buildingblock.php>. On this site you'll find datasheets for each component that describe how to use the component in Verilog code. For example, if you want a decrementer in your design, you can use the code shown in Figure 8.22. The **width** parameter controls how wide the synthesized decrementer is, and the reference to **DW01\_dec** pulls that model from the **DesignWare** library. Now when you use **decrementer** in other Verilog modules, it will be implemented with the **DesignWare** version. Check the **Synopsys DesignWare** web site for a full list of modules and datasheets that describe how to use them. If I synthesize the 8-bit decrementer with

```
syn-dc -f dec-script.tcl
```

(using the **Uofu\_Digital\_v1\_1** library as a target), I get the structural code shown in Figure 8.23.

Even if you don't directly instantiate a **DesignWare** model, if **design**

*I modified the class  
syn-script.tcl with the  
information needed to  
synthesize the  
decrementer.*

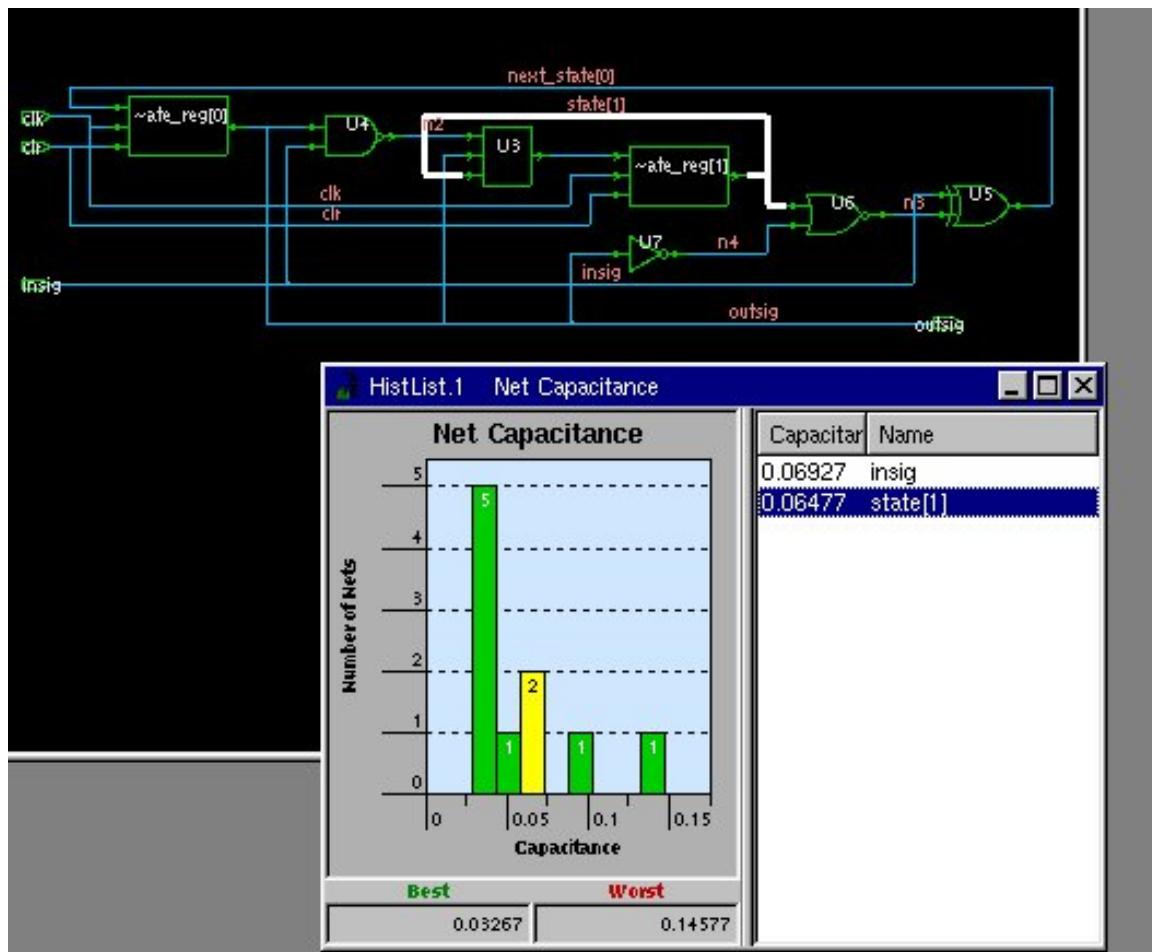


Figure 8.20: Wiring capacitance histogram with highlighted path

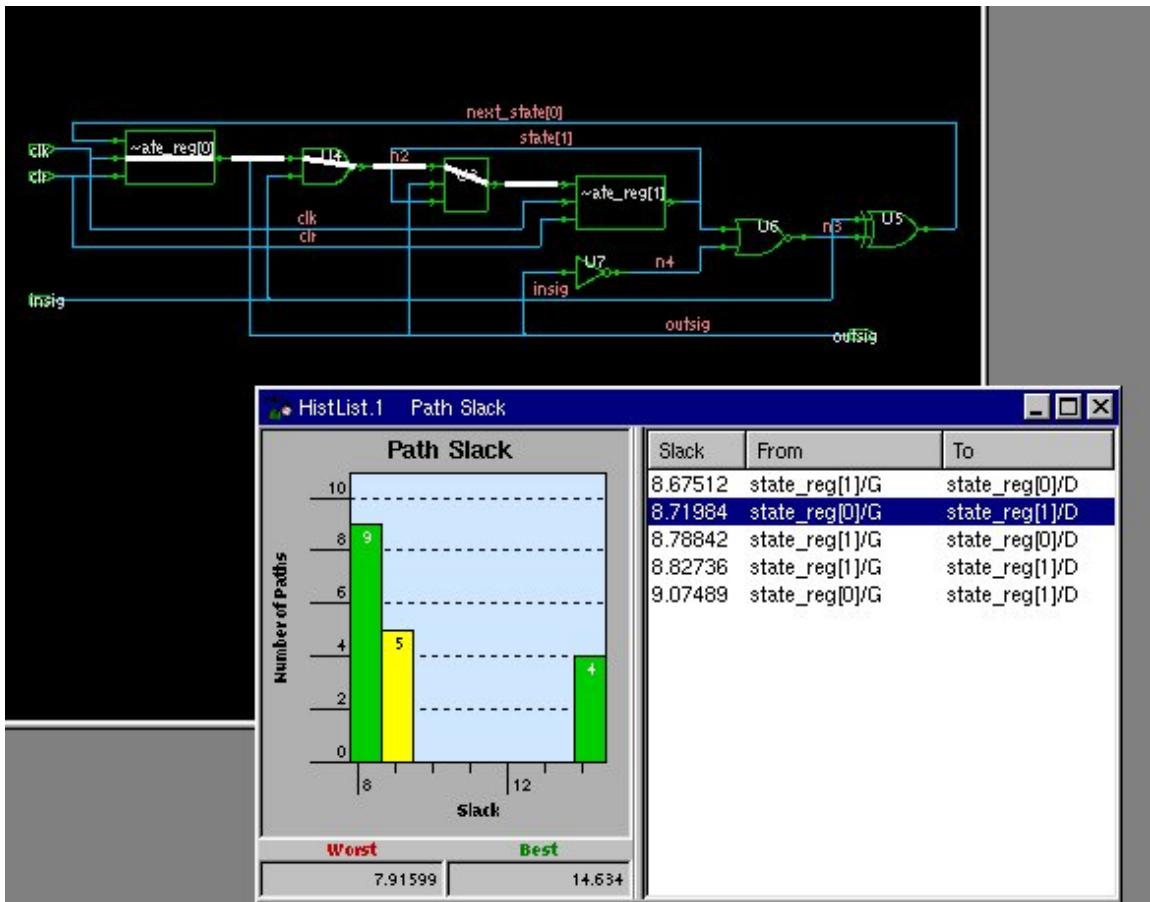


Figure 8.21: Timing path histogram with highlighted path

```

module decrementer( inst_A, SUM_inst );
  parameter width = 8;
  input [width-1 : 0] inst_A;
  output [width-1 : 0] SUM_inst;
  // Instance of DW01_dec
  DW01_dec #(width)
  U1 ( .A(inst_A), .SUM(SUM_inst) );
endmodule

```

Figure 8.22: DesignWare 9 bit decrementer instantiated in Verilog code

```

module decrementer ( inst_A, SUM_inst );
    input [7:0] inst_A;
    output [7:0] SUM_inst;
    wire    n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13;

    nand2 U1 ( .A(n1), .B(SUM_inst[0]), .Y(n3) );
    OAI U2 ( .A(SUM_inst[0]), .B(n1), .C(n3), .Y(SUM_inst[1]) );
    nor2 U3 ( .A(inst_A[2]), .B(n3), .Y(n4) );
    AOI U4 ( .A(n3), .B(inst_A[2]), .C(n4), .Y(n2) );
    nand2 U5 ( .A(n4), .B(n5), .Y(n6) );
    OAI U6 ( .A(n4), .B(n5), .C(n13), .Y(SUM_inst[3]) );
    xnor2 U7 ( .A(inst_A[4]), .B(n13), .Y(SUM_inst[4]) );
    nor2 U8 ( .A(inst_A[4]), .B(n6), .Y(n10) );
    nor2 U9 ( .A(inst_A[4]), .B(inst_A[3]), .Y(n7) );
    nand3 U10 ( .A(n7), .B(n4), .C(n9), .Y(n8) );
    OAI U11 ( .A(n10), .B(n9), .C(n8), .Y(SUM_inst[5]) );
    xnor2 U12 ( .A(inst_A[6]), .B(n8), .Y(SUM_inst[6]) );
    nand2 U13 ( .A(n10), .B(n9), .Y(n11) );
    nor2 U14 ( .A(inst_A[6]), .B(n11), .Y(n12) );
    xor2 U15 ( .A(inst_A[7]), .B(n12), .Y(SUM_inst[7]) );
    bufX4 U16 ( .A(n6), .Y(n13) );
    invX1 U17 ( .A(inst_A[1]), .Y(n1) );
    invX1 U18 ( .A(inst_A[0]), .Y(SUM_inst[0]) );
    invX1 U19 ( .A(inst_A[3]), .Y(n5) );
    invX1 U20 ( .A(inst_A[5]), .Y(n9) );
    invX1 U21 ( .A(n2), .Y(SUM_inst[2]) );
endmodule

```

Figure 8.23: Structural code for the 8-bit **DesignWare** decrementer

**compiler** runs across a piece of behavioral Verilog code that it thinks it can best implement with a **DesignWare** module, that's what it will use. There's a good chance you would get a **DesignWare** circuit if you had an assignment of the form

**assign sum = in - 1;**

in your code (or the equivalent inside of an **always** block). That's what is happening if you get sub-modules in your synthesized circuit with names that mysteriously start with **DW**. Of course, you have to have **dw.foundation.sldb** in your **synthetic\_library** list for this to happen. The flip side of this, of course, is that if you don't want **design compiler** to have these libraries available, you should remove that synthetic database file from that list.

### 8.1.5 Module Compiler

Module compiler is a separate tool specifically for synthesizing arithmetic circuits. It uses the same cell library database as **dc\_shell** but has more information about building efficient arithmetic structures, including floating point units. More details are coming!

There's some evidence that the **Module Compiler** tool is no longer rel-

event. It looks like the **DesignWare** macros that used to be restricted to **Module Compiler** are now available for general **design compiler**. I'll document **module compiler** procedures here, but you may want to see if you can do what you need to do directly in **design compiler** with the same (or even better) results.

## 8.2 Cadence BuildGates Synthesis

### 8.2.1 Basic Synthesis

*This is Cadence's version of the generic synthesis tool. Some people report better results for this tool than for dc\_shell. I'm sure it depends on your circuit and your familiarity with the tools. More details are coming!*

### 8.2.2 Scripted Synthesis

*More advanced scripted synthesis*

## 8.3 Importing Structural Verilog into Cadence

Once you have generated structural Verilog from Synopsys or from **Cadence**, one thing you might want to do is import that structural Verilog into Cadence **Composer** as a schematic and as a symbol so that you can use it in other schematics. This is known as *importing Verilog into Composer*. To import Verilog first decide which Cadence library you want to import the structural Verilog into. You may want to make a new library (make sure to attach it to the **UofU AMI C5N** technology library). Once you know which library you want to import the circuit into, Use the **CIW** menu **File** → **Import** → **Verilog....** I'll use the decremener from Figure 8.23 for this example. Fill in the fields:

**Target Library Name:** The library you want to read the Verilog description into. In this case I'll use a new library that I created named **decremener**.

**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 your synthesis process. In this case it's **dec\_structr.v** from my use of **Synopsys design compiler**.

**-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 8.24. 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 8.25) and symbol (Figure 8.26) of the decrementer. 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).

## 8.4 Cadence to Synopsys (CSI) Schematic/Netlist Interface

*Generating structural netlists from schematics - Although this is billed as an interface between Cadence and Synopsys, it is really a way to generate a structural netlist from a schematic. If you have a schematic with standard cells gates, this will generate a netlist that only goes down to those gates, and not descend all the way to the transistors as would happen for simulation. More details are coming!*

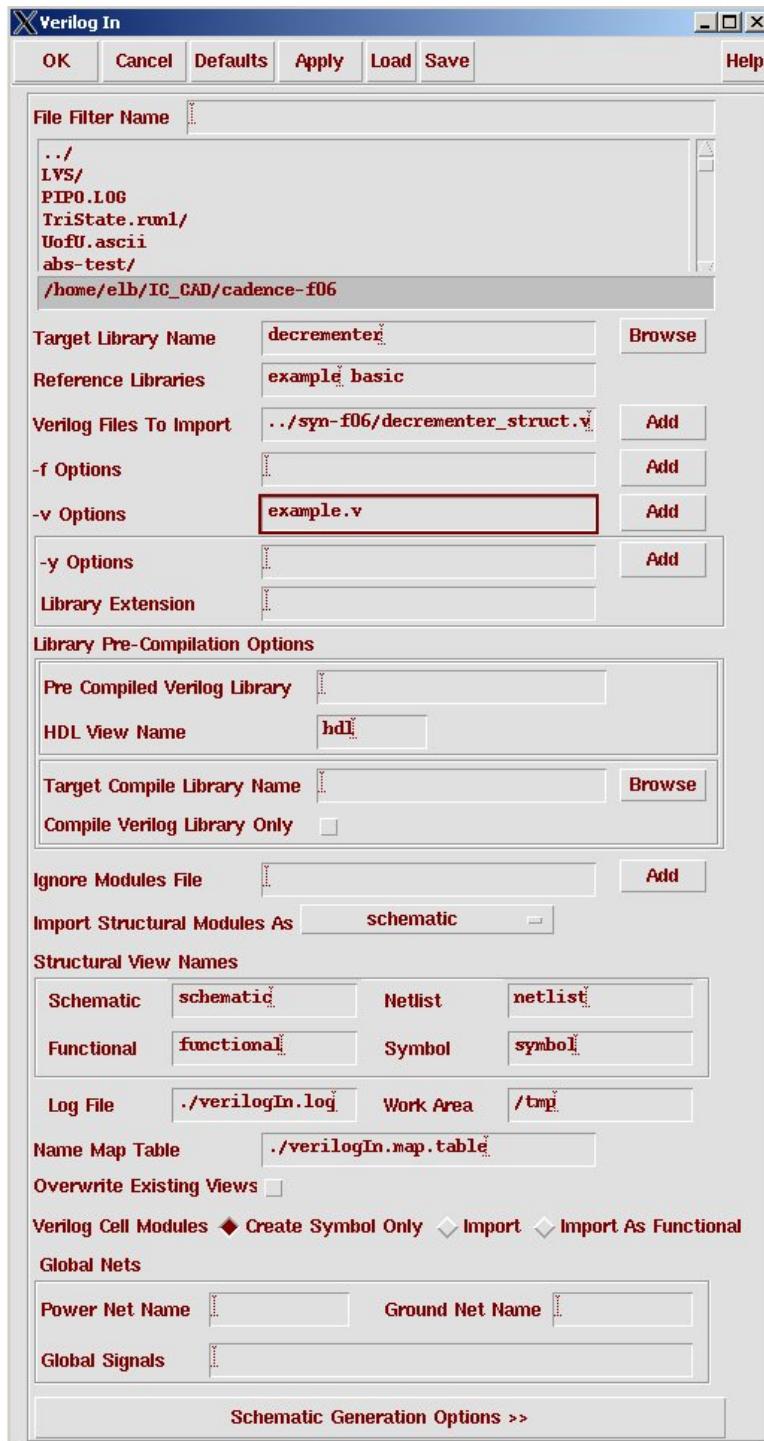


Figure 8.24: Dialog box for importing structural Verilog into a new schematic view

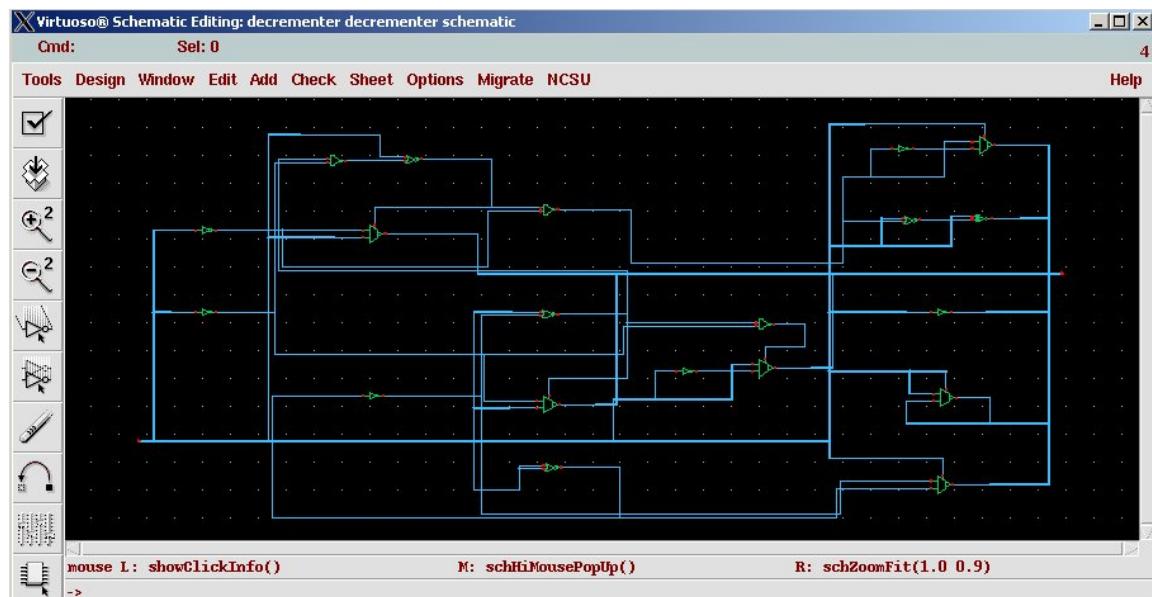


Figure 8.25: Schematic that results from importing the decremter into Composer

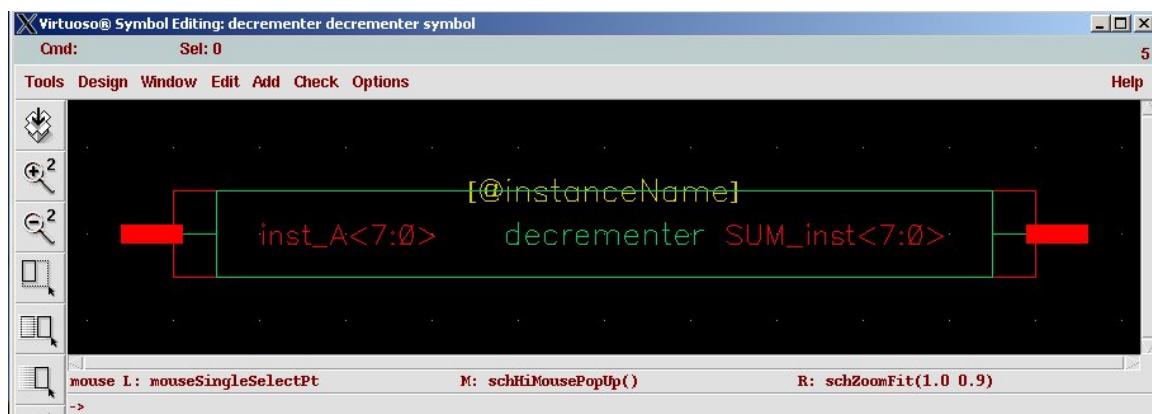


Figure 8.26: Symbol that is created for the decremter



# Chapter 9

## Abstract Generation

**I**N ORDER for the place and route tool to do its job, it needs to know certain physical information about the cells in the library it is using. For example, it needs to know the bounding box of each cell so it can use that bounding box when it places cells next to each other. Note that this bounding box may be described by the furthest pieces of geometry in the layout view of the cell, or it may be offset from the layout so that the cell geometry overlaps or has a gap when it is placed next to another cell. The place and route tool doesn't need to know anything about the layout. It just places the cells according to their bounding box.

The other critical information for the place and route tool is the signal connection points on the cells. It needs to know where the power and ground connections are, and the signal connections so that it can wire the cells together. It also needs to know just enough about the layout geometry of the cells so that it knows where it can and can't place wires over the top of the cell.

A view of the cell that has only this information about the cell (bounding box, signal connections, and wiring blockages) is called an **abstract** view. An **abstract** view has much less information than the full layout view which is useful for two reasons: it's smaller and therefore less trouble to read, and it doesn't have full layout information so cell library vendors can easily release their abstract views while keeping the customers from seeing the full layout views and thus retain their proprietary layout.

As an example, consider a simple inverter. Figure 9.1 shows the **layout** view side by side with the **abstract** view. You can see that the abstract is very simple compared to the layout. The abstract view doesn't need any information about any layers that aren't directly used by the place and route program. Because the place and route program uses only the metal layers for routing, only the metal layers show up in the abstract. This example

actually has four connection points: A (input), Y (output), and vdd (the top metal piece) and gnd (the bottom metal piece). Note that the power and ground connections are designed to abut (actually overlap) when the cells are placed next to each other with their bounding boxes touching.

## 9.1 Abstract Tool

We, of course, have full layout views because that's what we've been designing. The tool that extracts the **abstract** information from the full layout view is called, appropriately enough, **abstract** and it can be run from your cadence directory using the `cad-abstract` script.

Before you run **abstract** you need to have **layout** and **extracted** views (at least) of the cells that you wish to generate the abstracts of. Connect to the directory from which you run Cadence, and fire up the tool with `cad-abstract`. The **abstract** tool will now have access to your design libraries that you've been using in Cadence.

### 9.1.1 Reading your Library into Abstract

*You can also get short help on many things by hovering your mouse over the object.*

When you start up **abstract** you'll get a command window. All the major operations in generating abstracts can be initiated with the widgets near the top of the window or by using the menus. The far-left widget that looks like a book is used to load a Cadence library into **abstract**. The first step in generating **abstract** views is to load a library that contains your cells. Figure 9.2 shows the **Open Library** box where I'm choosing my **abs-test** library.

Once you've loaded your library you should see all the cell names in your library sorted into **bins**. the **Core** bin will contain the cells that **abstract** thinks will be general core cells. The **Ignore** bin are the cells that it will ignore. The other bins (**IO**, **Corner**, **Block**) are related to pad I/O and pre-designed blocks. We won't use them for our cell library. If you are only interested in generating abstracts for some of your cells you can use the **Cells → Move...** menu choice to move cells to different **bins**. By moving cells to the **Ignore bin**, for example, you can exclude them from further processing. My example library has 11 cells that I would like to process into **abstract** views, and 2 that I want to ignore. The window looks like that in Figure 9.3. Note that you may have warnings that you don't already have **abstract** views. You can ignore them. You should, however, see green check marks by each cell in the **layout** column to indicate that the layout has been successfully read. If you see an orange exclamation point instead that means that there was some issue when the cell was read into **abstract**.

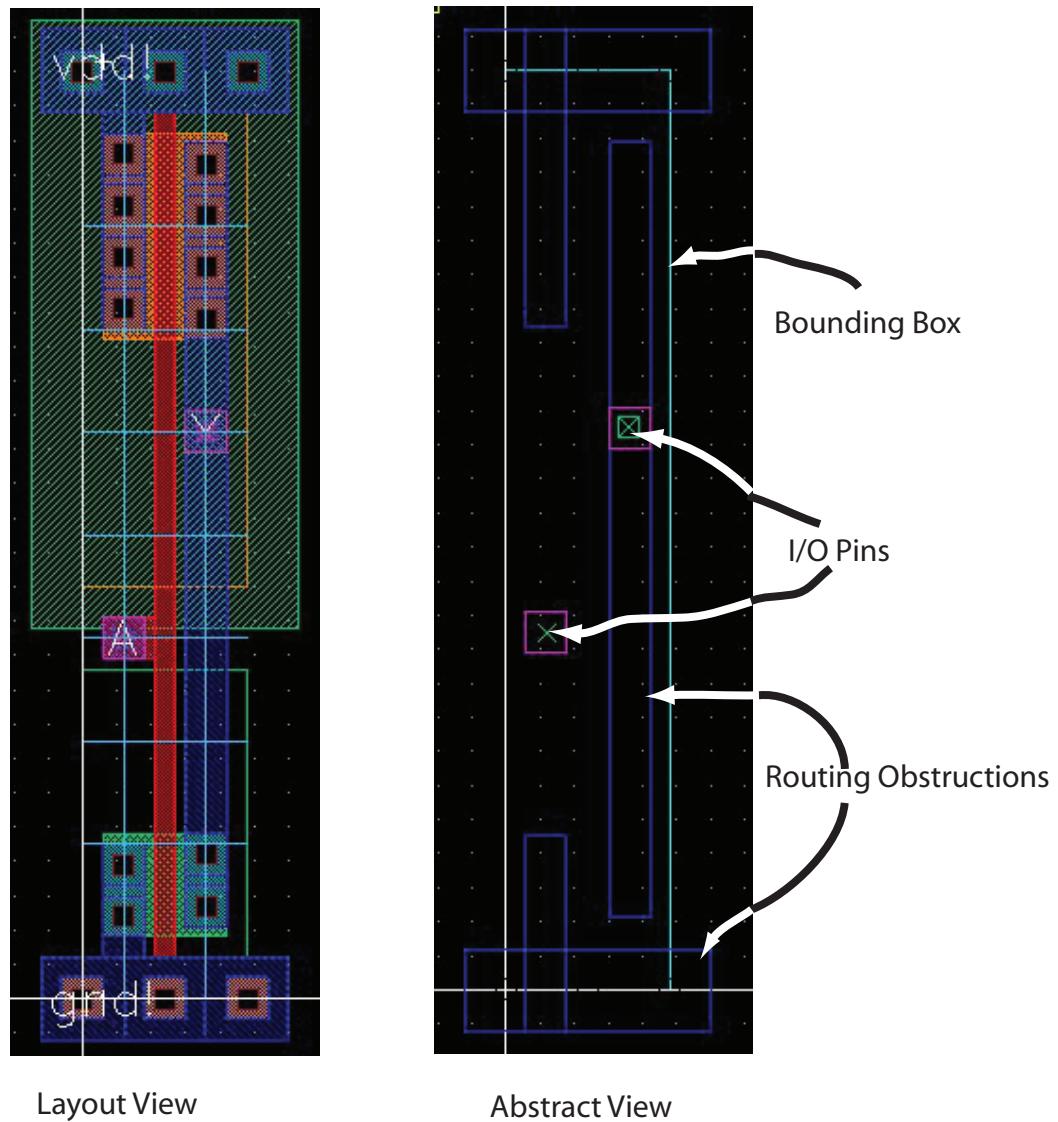


Figure 9.1: Side by side **layout** and **abstract** views of an inverter

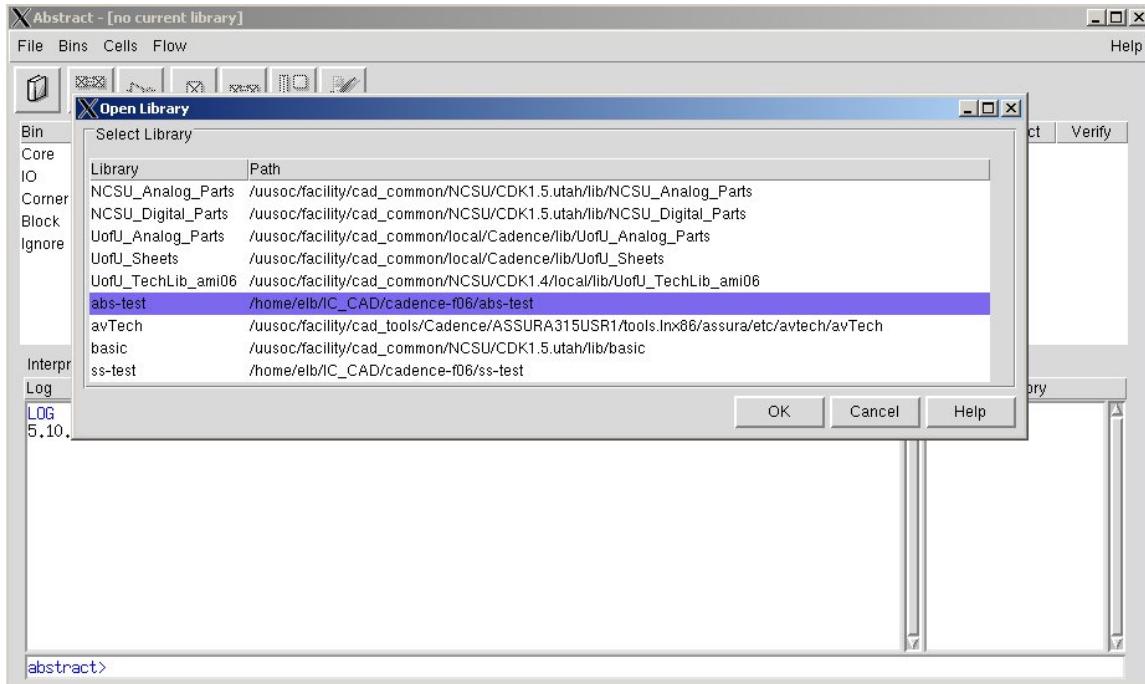


Figure 9.2: Opening a library in abstract

You need to look at the log to figure out what's going on. Remember that warnings may often be ignored, but you need to know why you can ignore them! Errors are something you need to fix.

### 9.1.2 Finding Pins in your cells

Once you've read in your library and you have green checks by all the cells that you want to convert, select the cells you're interested in by clicking and dragging to highlight them, and move to the next step, which in our case is the **Pins** step. We skip the **Logical** step because our place and route tool don't care about the logical behavior of the cells. We put that information in the **.lib** file. The **Pins** step can be initiated by the menu (**Flow → Pins...**) or by the **Pins** widget which is the single square with the cross in it (looks like a contact). Initiating the **Pins** phase will bring up the **Pins** dialog box as seen in Figure 9.4. In the **Pins** dialog box you need to add some information.

1. If you have made *all* your pins as **shape pins** as described in Chapter 5, Section 5.2, then you can leave the **Map text labels to pins:** section blank. If you have made pins by just putting a text label on top of some routing material then you need to add something here.

*The notation (text drawing) is a layer-purpose pair which means the rectangle is text type with drawing purpose.*

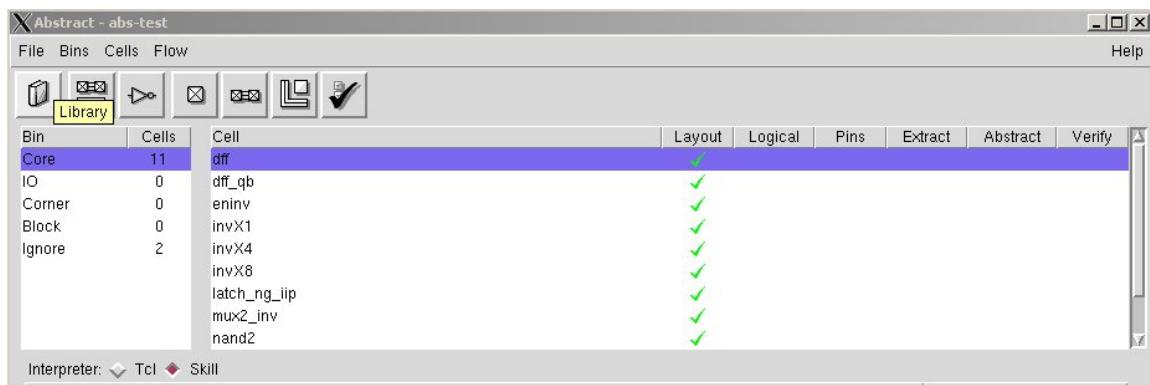


Figure 9.3: Cells in **abstract** after the library has been read

In particular, you should add a string that defines which text layer on top of which routing layer should be extracted as a signal pin. For example:

**((text drawing)(metal1 drawing)(metal2 drawing))**

This text says that any **(text drawing)** layer that overlaps either **metal1** or **metal2** should be extracted as a pin.

2. You need to add any names you used for clock pins in your design.
3. You need to add any names you used for output pins in your design.
4. The regular expressions for power and ground should default to something that recognizes the **vdd!** and **gnd!** labels that we use. If you've used something else, you need to modify these settings.

In the **Boundary** tab of this dialog box, you need to make sure that the geometry layers that you want **abstract** to use to generate the bounding box are set correctly, and that the **Adjust Boundary By** fields are set to **-1.2**. Our standard cell format requires that the bounding box is smaller than the geometry by  $1.2\mu$  on each side. This way the cells will overlap slightly when they are abutted according to the bounding box. This tab is shown in Figure 9.5.

When both of these modifications are made, you can click **Run** to run the **Pins** step of the process.

When I run the **Pins** step I always get a lot of warnings. In particular you'll probably see **ABS-515** warnings because the generated bounding box doesn't enclose all the layout geometry. Of course, we told **abstract** to do it that way in the **Boundary** tab of the **Pins** dialog! You'll also probably get **ABS-502** warnings that no terminals were created in your cell. If you used

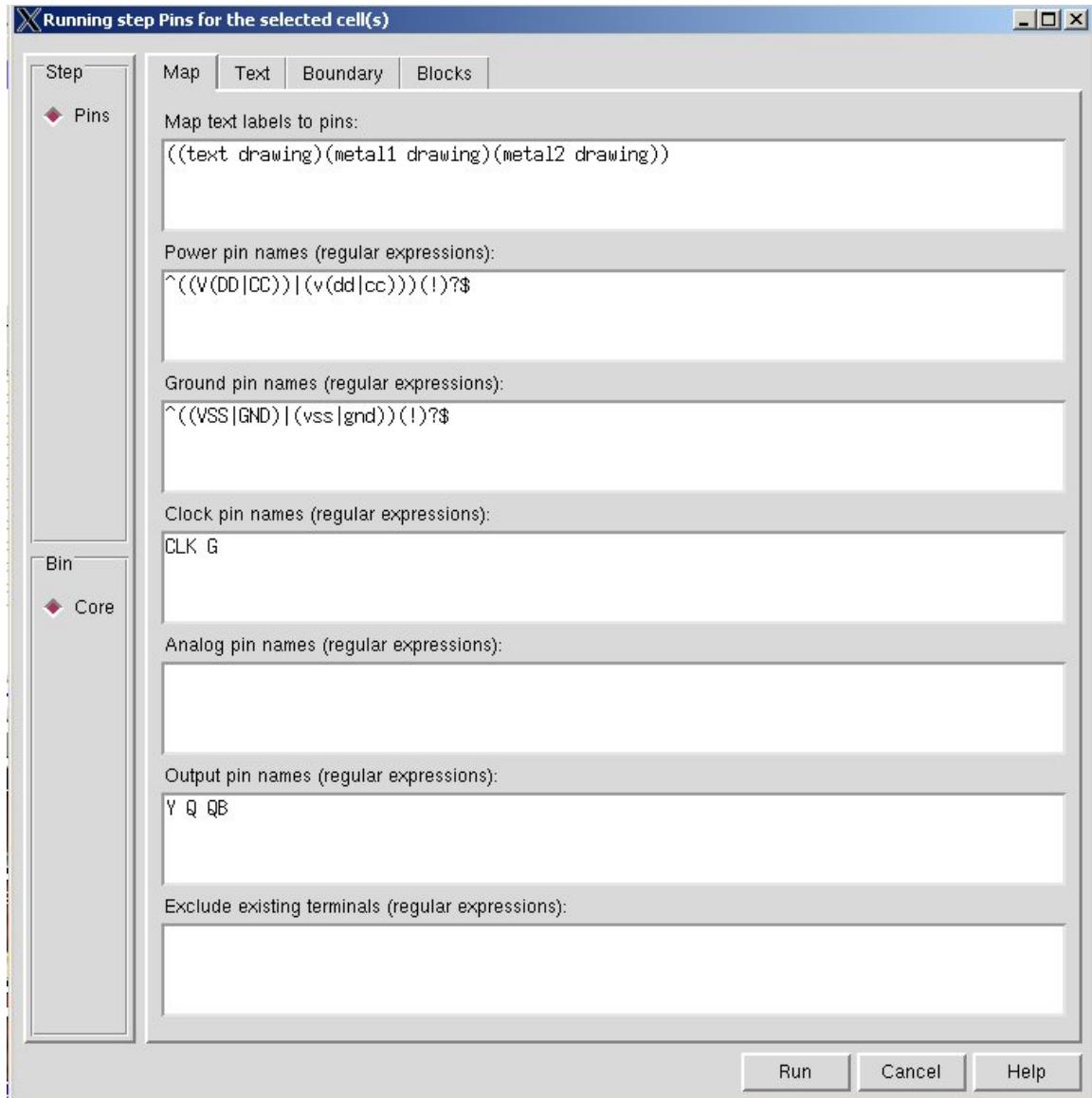


Figure 9.4: The **Map** tab in the **Pins** step dialog box

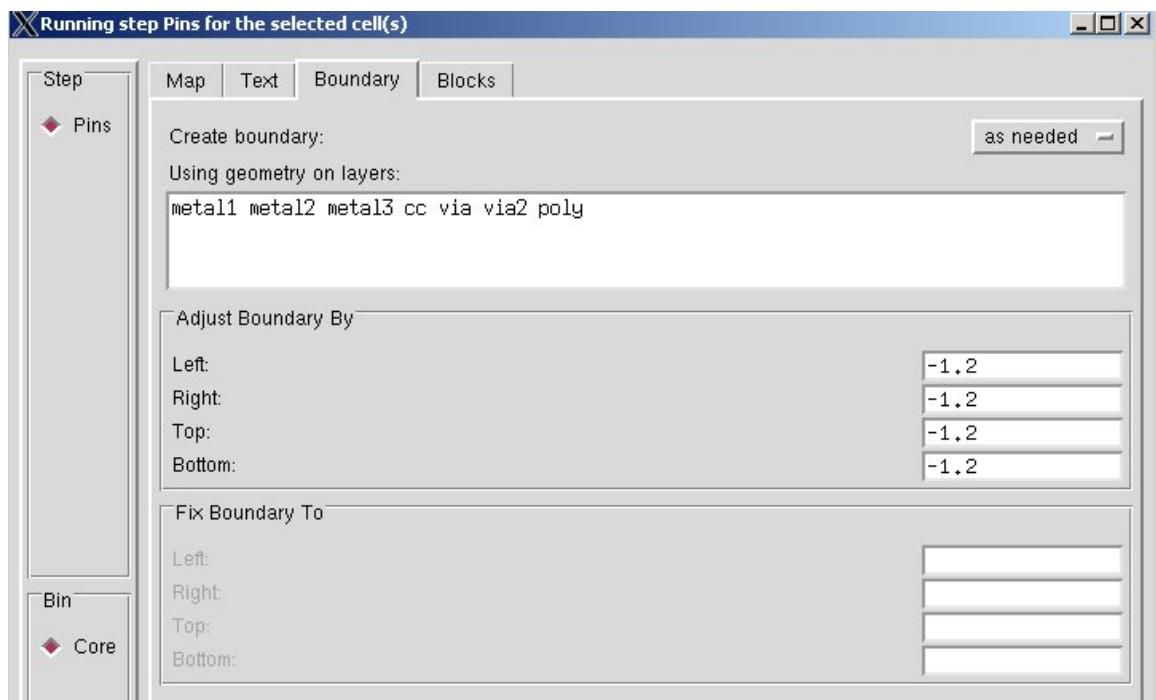


Figure 9.5: The **Boundary** tab in the **Pins** step dialog box

**shape pins** correctly then you can ignore this. You should see a previous line that says that the terminals already exist. All this warning is telling you is that the terminals already exist as **shape pins** and nothing had to be extracted through the text label technique. I also sometimes see warnings about **rcbScan** failures. These can also be ignored. They are minor issues with the database that don't effect behavior.

However, the result of all these warnings is that you're likely to see the pins step have all orange exclamation points!

### 9.1.3 The Extract step

The next step is to **Extract** the connection and obstruction information about your cells. The extract step widget looks like two contacts connected by a little wire. The dialog box is shown in Figure 9.6, and you shouldn't have to modify it from the defaults. The result of running this step should be green check marks in all rows of the **Extract** column.

### 9.1.4 The Abstract step

Finally you can generate the actual **abstract** views of your cells with the **Abstract** step. In the dialog box you can leave most things as defaults. In the **Site** tab you should change the **Site** to be **core** (see Figure 9.7). You can check the other tabs if you like. The **Blockage** tab should have all the routing layers specified as **Detailed Blockages**. The **Grids** tab will have **metal1** and **metal2** routing grid information that was derived from the technology file. In particular the **metal1** pitch will be  $3\mu$  because it is helpful if it matches the wide pitch of the other horizontal routing layer **metal3**. The **metal2** vertical pitch is smaller at  $2.4\mu$ .

If each of these steps has been run correctly you should see something similar to Figure 9.8. We skip the **Verify** step because it involves setting up test scripts in the place and route program which we haven't done. Instead we now generate the output file.

### 9.1.5 LEF File Generation

The result of this process is that the **abstract** views of the cells can now be output in a **Library Exchange Format** or **.lef** file. This is an ASCII-formatted file that contains a technology header and then a description of the abstract views in a format that the place and route tools can understand. You can create this **.lef** file using the **File → Export → LEF** menu. In the dialog box (Figure 9.9) you should export only the **Geometry LEF Data**

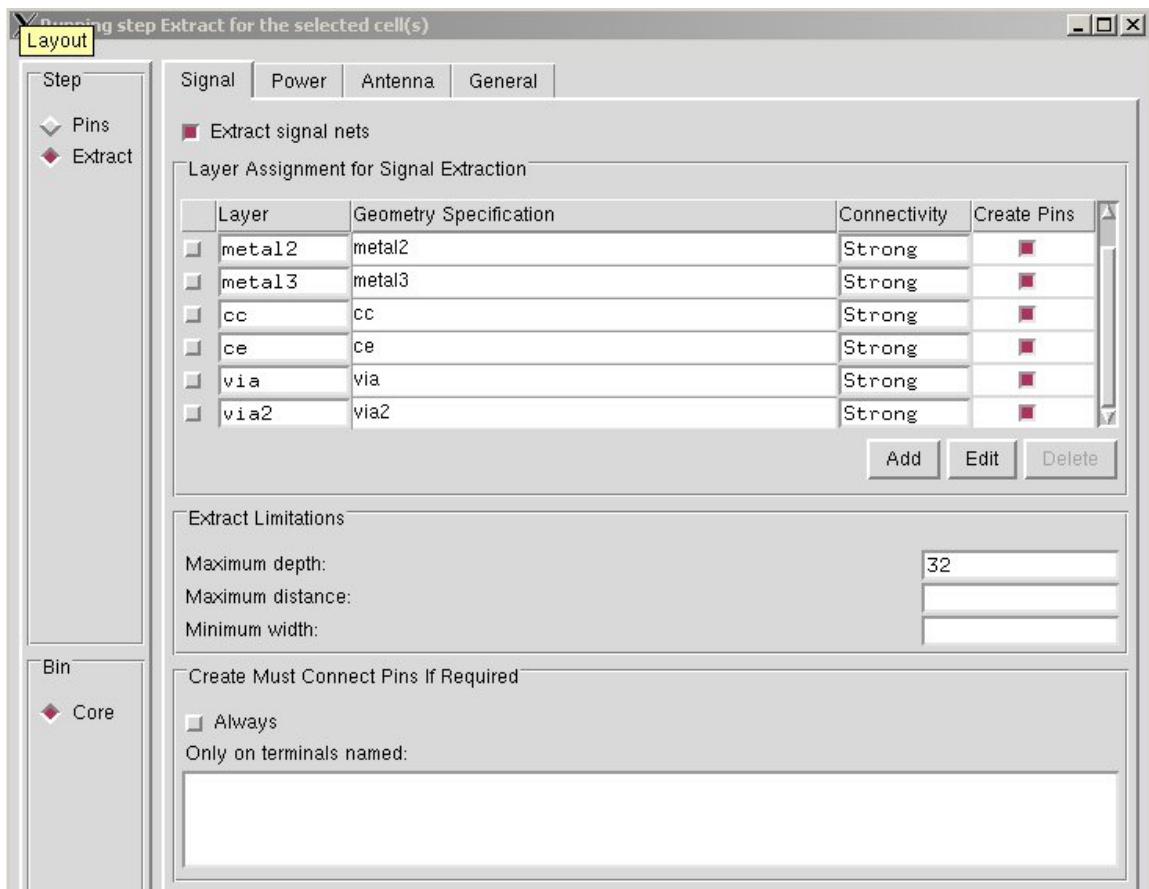


Figure 9.6: Extract step dialog box

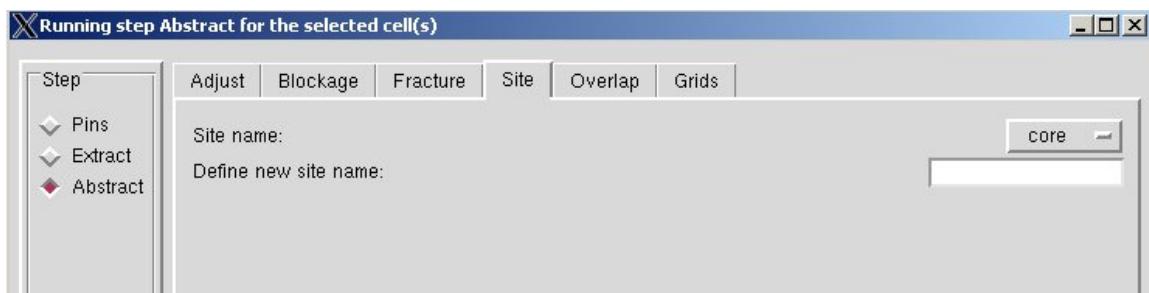


Figure 9.7: Site tab in the Abstract step dialog box

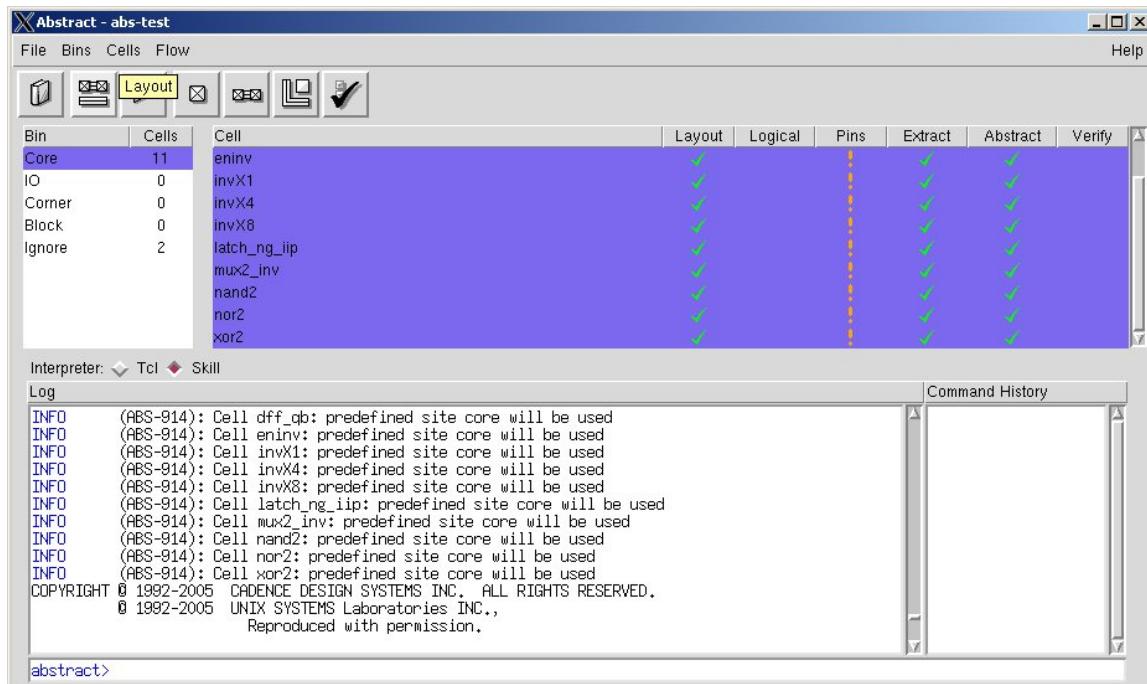


Figure 9.8: **Abstract** window will the **Layout**, **Pins**, **Extract** and **Abstract** steps completed

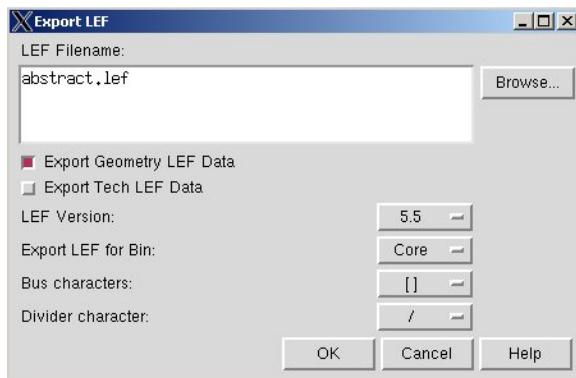


Figure 9.9: Dialog box for exporting **LEF** information

(We'll add our own customized tech data later), make sure that the **Version** is **5.5**, and the the **Bin** to be exported is **Core**. You can name the exported file anything you wish.

Once you have successfully exported the **.lef** file you can exit the **Abstract** program.

### 9.1.6 Modifying the LEF file

When you export the **Geometry LEF Data** you are exporting only the information about each of your cells. You want to collect these macro descriptions into one big **LEF** file that describes your entire library, but with only one copy of the **tech** header information. Also, we've added some extra information into the **tech** header beyond what gets generated by **Abstract**. You can get the **tech** header from **/uusoc/facility/cad\_common/local/Cadence/lef\_files/TechHeader.lef**. This file is also listed in Appendix E.

When you're ready to assemble your complete **LEF** file for your full library you should start with one copy of the **TechHeader.lef** file, and then insert only the **MACRO** definitions from your use of the **Abstract** program. That is, do not repeat the **VERSION**, **SITE** and other info in your final **LEF** file.



# 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 N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, N15, N16, N17, N18, N19,
          N20, n6, n7, n8, n9, n10, n11, n12, n13, n14;

    MUX2X2 U3 ( .A(N11), .B(in[7]), .S(load), .Y(N19) );
    MUX2X2 U4 ( .A(N20), .B(in[6]), .S(load), .Y(N18) );
    MUX2X2 U5 ( .A(N10), .B(in[5]), .S(load), .Y(N17) );
    MUX2X2 U6 ( .A(N9), .B(in[4]), .S(load), .Y(N16) );
    MUX2X2 U7 ( .A(N8), .B(in[3]), .S(load), .Y(N15) );
    MUX2X2 U8 ( .A(N7), .B(in[2]), .S(load), .Y(N14) );
    MUX2X2 U9 ( .A(N6), .B(in[1]), .S(load), .Y(N13) );
    MUX2X2 U10 ( .A(N5), .B(in[0]), .S(load), .Y(N12) );
    DCBX1 tmp_reg_0_ ( .D(N12), .CLK(clk), .CLR(clr), .Q(count[0]), .QB(N5) );
    DCBX1 tmp_reg_1_ ( .D(N13), .CLK(clk), .CLR(clr), .Q(count[1]), .QB(n6) );
    DCBX1 tmp_reg_2_ ( .D(N14), .CLK(clk), .CLR(clr), .Q(count[2]) );
    DCBX1 tmp_reg_3_ ( .D(N15), .CLK(clk), .CLR(clr), .Q(count[3]), .QB(n8) );
    DCBX1 tmp_reg_4_ ( .D(N16), .CLK(clk), .CLR(clr), .Q(count[4]) );
    DCBX1 tmp_reg_5_ ( .D(N17), .CLK(clk), .CLR(clr), .Q(count[5]), .QB(n11) );
    DCBX1 tmp_reg_6_ ( .D(N18), .CLK(clk), .CLR(clr), .Q(count[6]) );
    DCBX1 tmp_reg_7_ ( .D(N19), .CLK(clk), .CLR(clr), .Q(count[7]) );
    AOI22X1 U11 ( .A(count[0]), .B(count[1]), .C(n6), .D(N5), .Y(N6) );
    NOR2X1 U12 ( .A(N5), .B(n6), .Y(n7) );
    XOR2X1 U13 ( .A(count[2]), .B(n7), .Y(N7) );
    NAND2X1 U14 ( .A(count[2]), .B(n7), .Y(n9) );
    MUX2NX1 U15 ( .A(count[3]), .B(n8), .S(n9), .Y(N8) );
    NOR2X1 U16 ( .A(n9), .B(n8), .Y(n10) );
    XOR2X1 U17 ( .A(count[4]), .B(n10), .Y(N9) );
    NAND2X1 U18 ( .A(count[4]), .B(n10), .Y(n12) );
    MUX2NX1 U19 ( .A(count[5]), .B(n11), .S(n12), .Y(N10) );
    NOR2X1 U20 ( .A(n12), .B(n11), .Y(n13) );
    XOR2X1 U21 ( .A(n13), .B(count[6]), .Y(N20) );
    NAND2X1 U22 ( .A(count[6]), .B(n13), .Y(n14) );
    XNOR2X1 U23 ( .A(count[7]), .B(n14), .Y(N11) );
endmodule

```

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

```
#####
# Created by write_sdc on Thu Oct  4 16:44:27 2007
#####
set sdc_version 1.7

set_units -time ns -resistance kOhm -capacitance pF -voltage V -current uA
set_driving_cell -lib_cell INVX4 [get_ports clr]
set_driving_cell -lib_cell INVX4 [get_ports load]
set_driving_cell -lib_cell INVX4 [get_ports {in[7]}]
set_driving_cell -lib_cell INVX4 [get_ports {in[6]}]
set_driving_cell -lib_cell INVX4 [get_ports {in[5]}]
set_driving_cell -lib_cell INVX4 [get_ports {in[4]}]
set_driving_cell -lib_cell INVX4 [get_ports {in[3]}]
set_driving_cell -lib_cell INVX4 [get_ports {in[2]}]
set_driving_cell -lib_cell INVX4 [get_ports {in[1]}]
set_driving_cell -lib_cell INVX4 [get_ports {in[0]}]
set_load -pin_load 0.0659726 [get_ports {count[7]}]
set_load -pin_load 0.0659726 [get_ports {count[6]}]
set_load -pin_load 0.0659726 [get_ports {count[5]}]
set_load -pin_load 0.0659726 [get_ports {count[4]}]
set_load -pin_load 0.0659726 [get_ports {count[3]}]
set_load -pin_load 0.0659726 [get_ports {count[2]}]
set_load -pin_load 0.0659726 [get_ports {count[1]}]
set_load -pin_load 0.0659726 [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 **floorplan** 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 → Import Design...** 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.

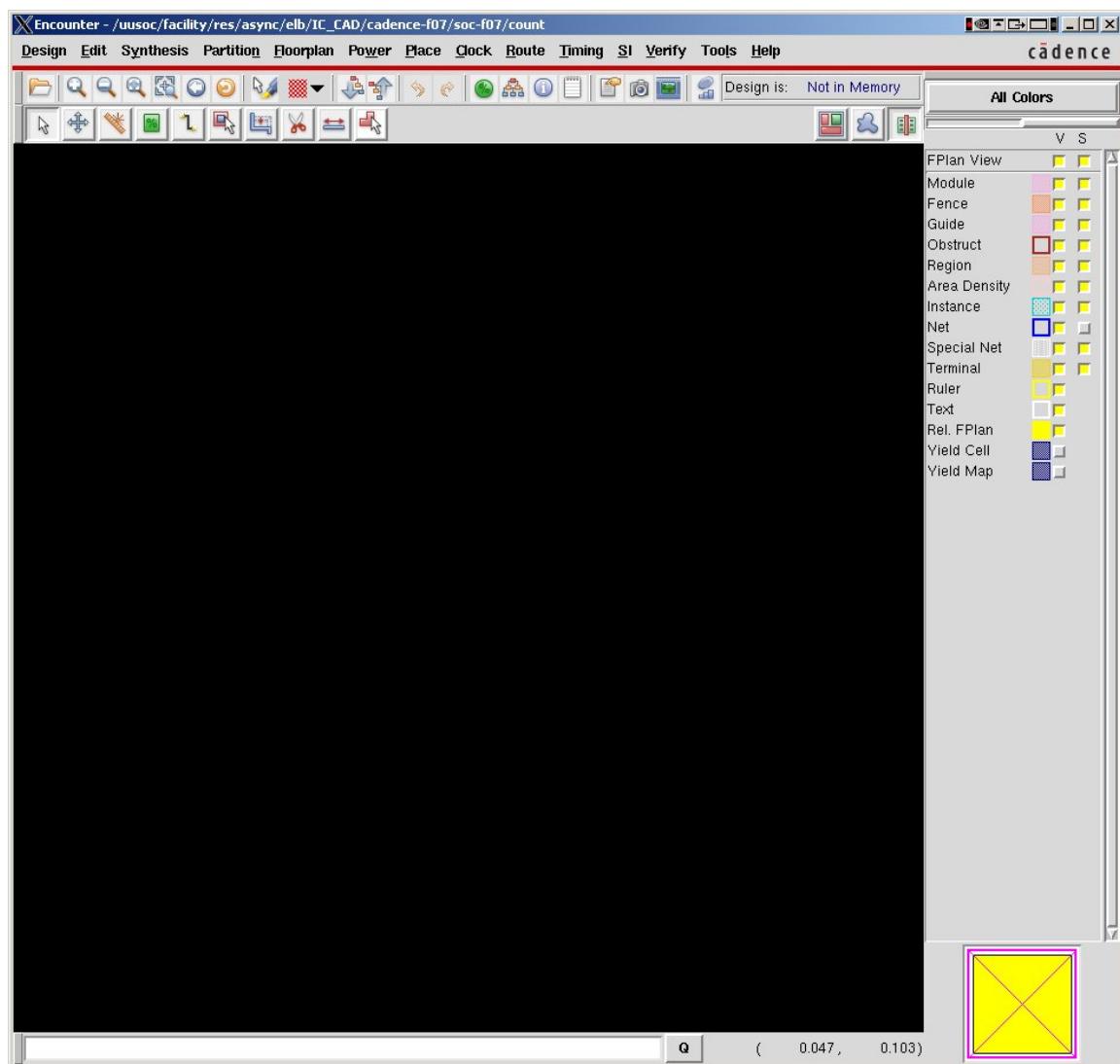


Figure 10.4: Main SOC Encounter gui

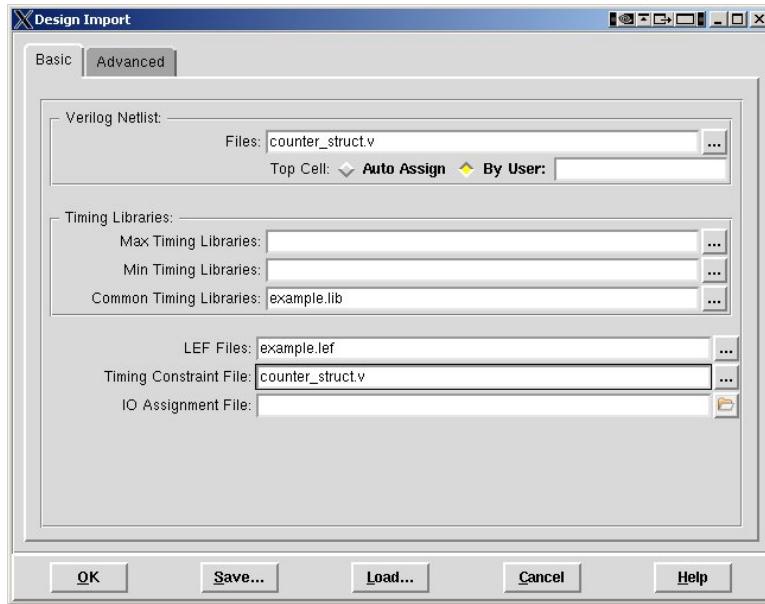


Figure 10.5: **Design Import** dialog box - basic tab

**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).

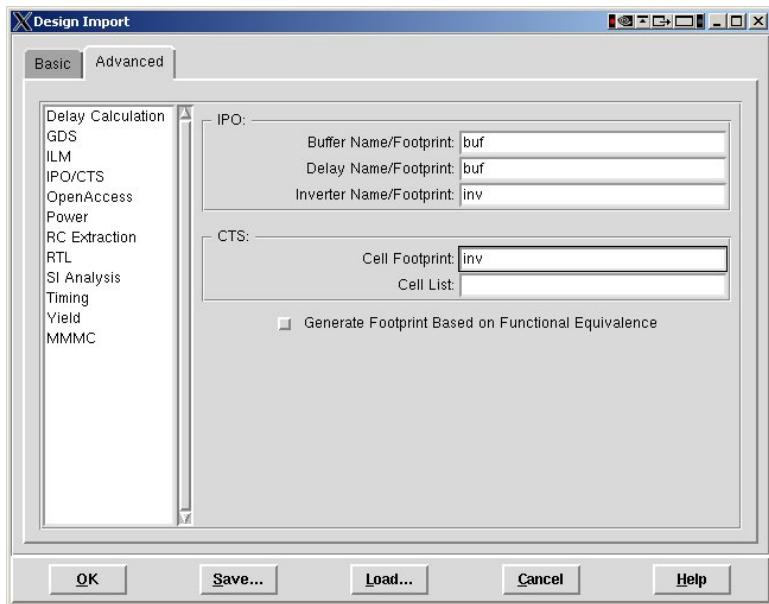


Figure 10.6: **Design Import IPO/CTS** tab

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 mode 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.

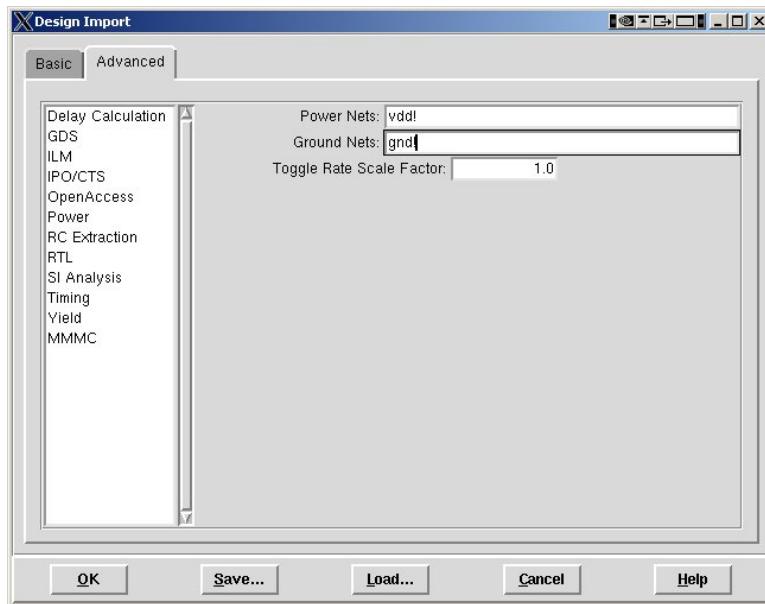


Figure 10.7: Design Import Power tab

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 (Figures 10.8 and 10.9). In this dialog box you can change various parameters related to the floorplan (basic and advanced tabs):

**Aspect Ratio** (Basic tab): 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** (Basic tab): 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.

**Core Margins** (Basic tab): This 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

All measurements are assumed to be in microns.

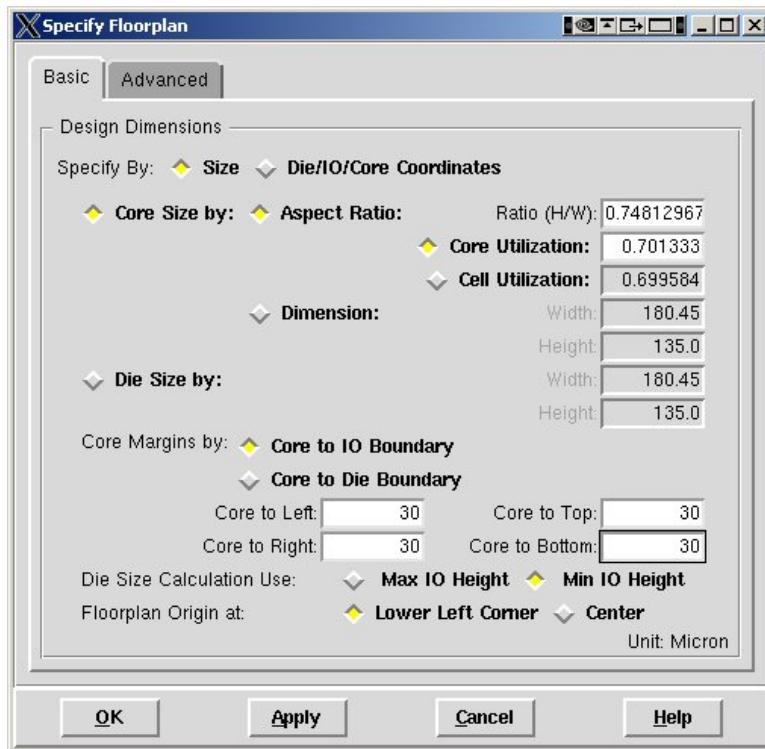


Figure 10.8: The **Specify Floorplan** dialog box, Basic tab

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** (Advanced tab): Other spots in the **Specify Floorplan** dialog can be left as default. In particular, and in the Advanced tab) 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. However, because we have only three routing layers in our process, and you will likely use more than one of them in your cells, you may want to increase the spacing between each pair of rows so that **SOC** has more room for routing. As designs become larger, **SOC** will need that space because of routing congestion. If we had a process with more routing layers, you could likely leave the Row Spacing at 0.

*IBM's 90nm process, for example, has eight metal routing layers.*

After adjusting the floorplan, the main **SOC Encounter** window looks like Figure 10.10. 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

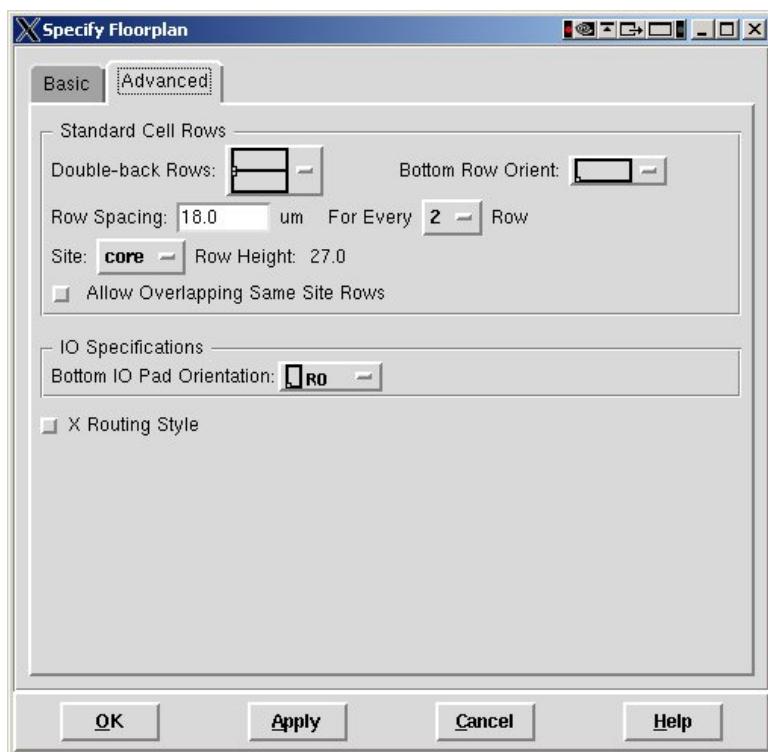


Figure 10.9: The **Specify Floorplan** dialog box, Advanced tab

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 As...** menu. Choose the **SoCE...** option for the save format (the other choice is the Open Access (**OA**) database that is not supported by our tech files yet) and name the saved file <filename>.enc. In my case I’ll name it **fplan.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...→ SoCE...** 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 10.11) you can control how the power rings are generated. The fields of interest are:

**Net(s):** The default values should be set to **gnd!** **vdd!** from when you imported the design.

**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 **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.

*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.*

**Other tabs:** You should be able to leave the defaults as they are in the other tabs.

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.

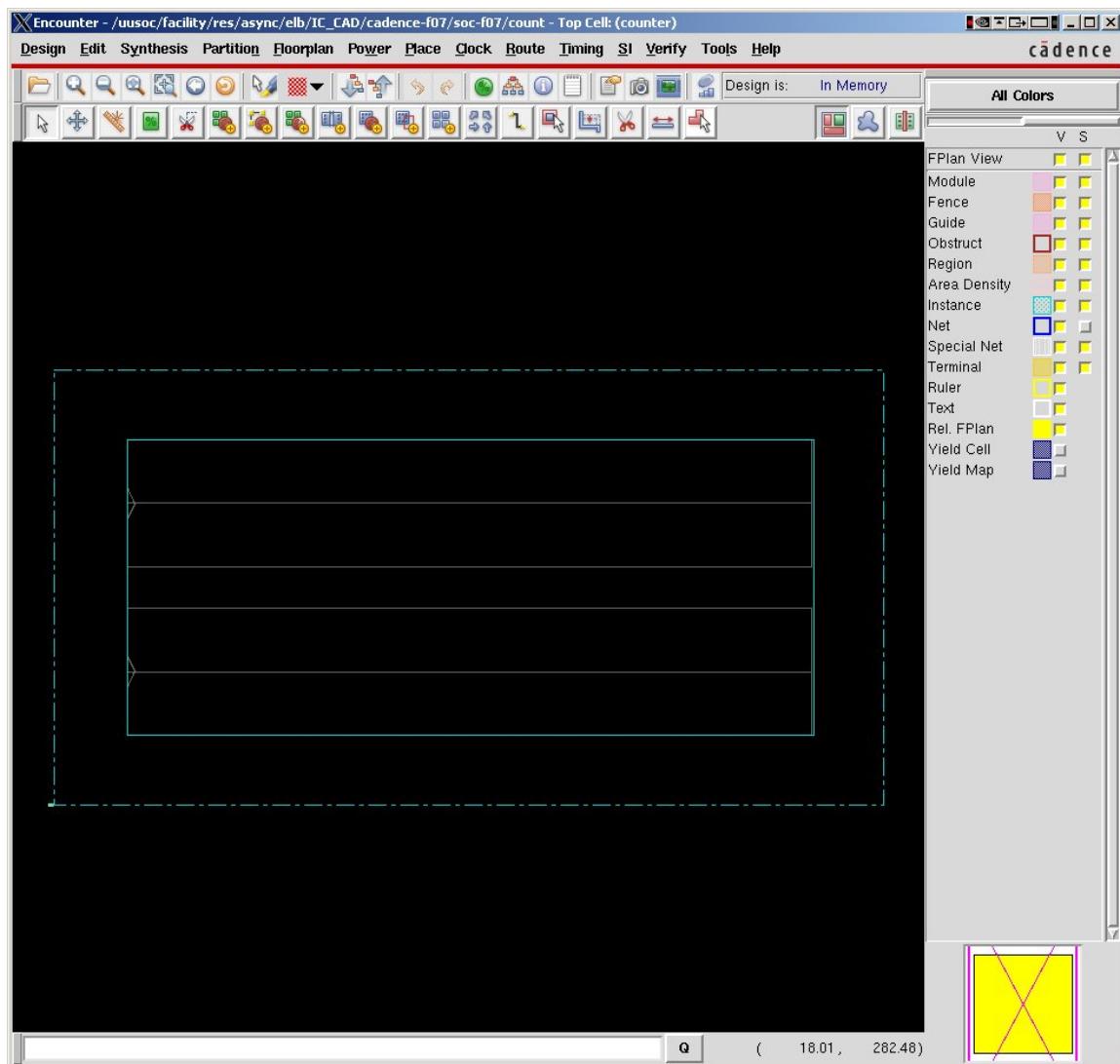


Figure 10.10: Main design window after floorplanning

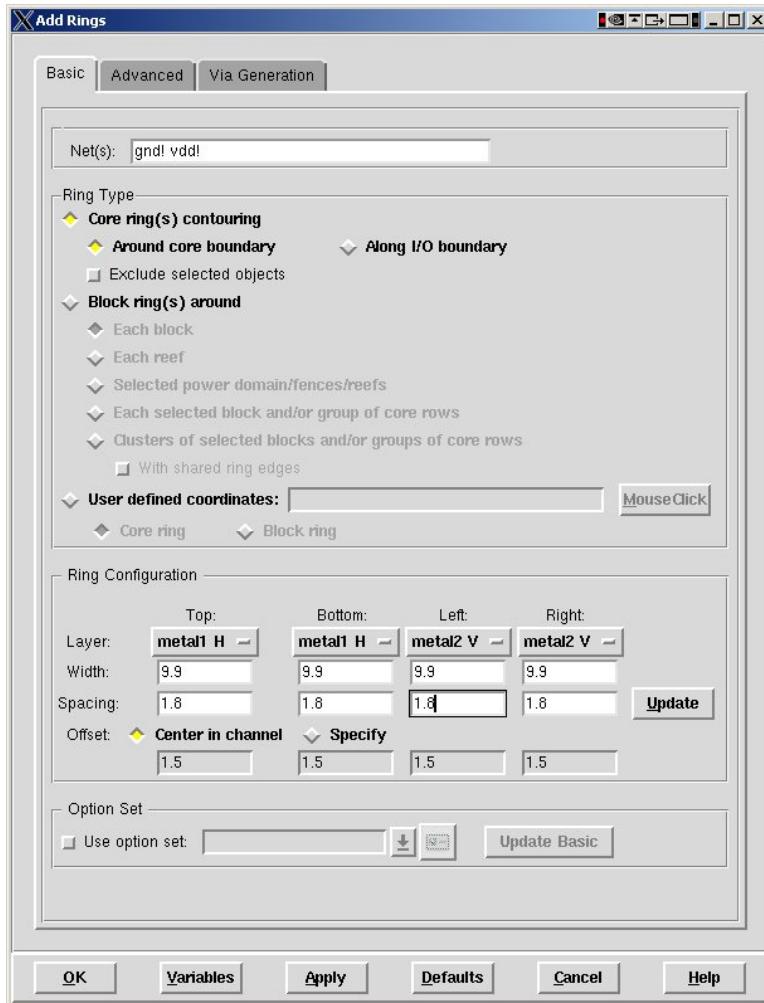


Figure 10.11: Dialog box for adding power and ground rings around your cell

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 10.12. 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 should make sure the **Set-to-set distance** is a multiple of 0.3 like **99**.

**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 **90** 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 150 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  $0.6\mu$  from the **prBoundary**. That way layers, such as metal layers, are at least  $1.2\mu$  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 metal spacing DRC violation when the metal of the stripe is now only  $0.6\mu$  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.

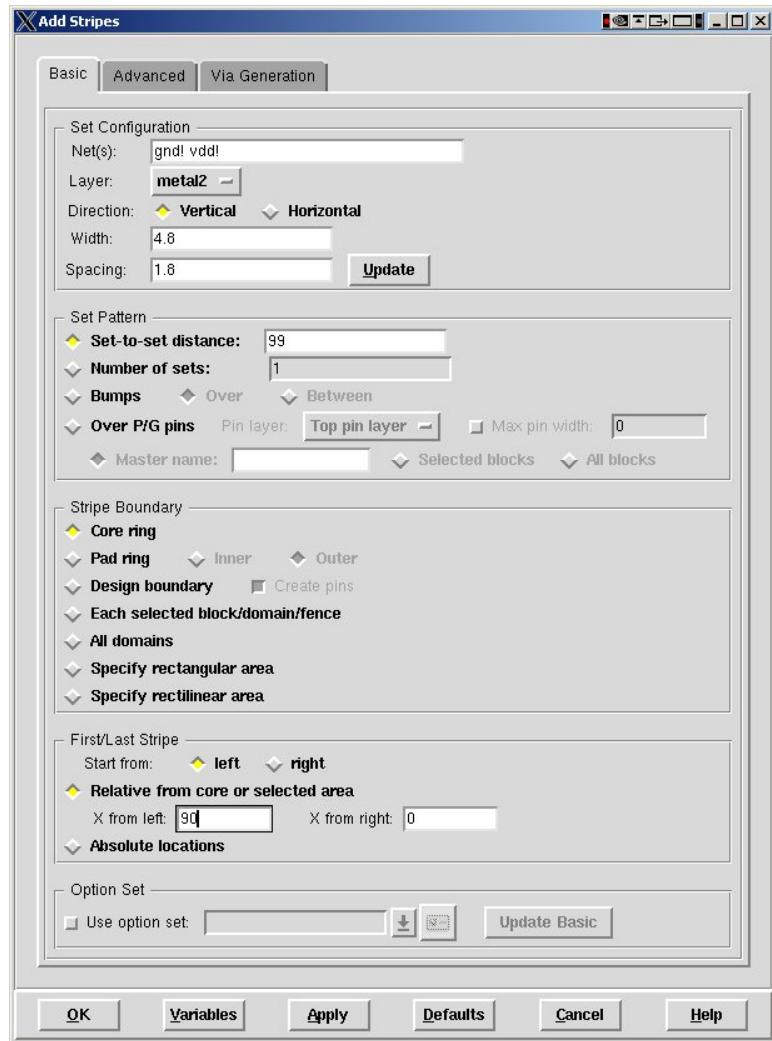


Figure 10.12: Dialog box for planning power stripes

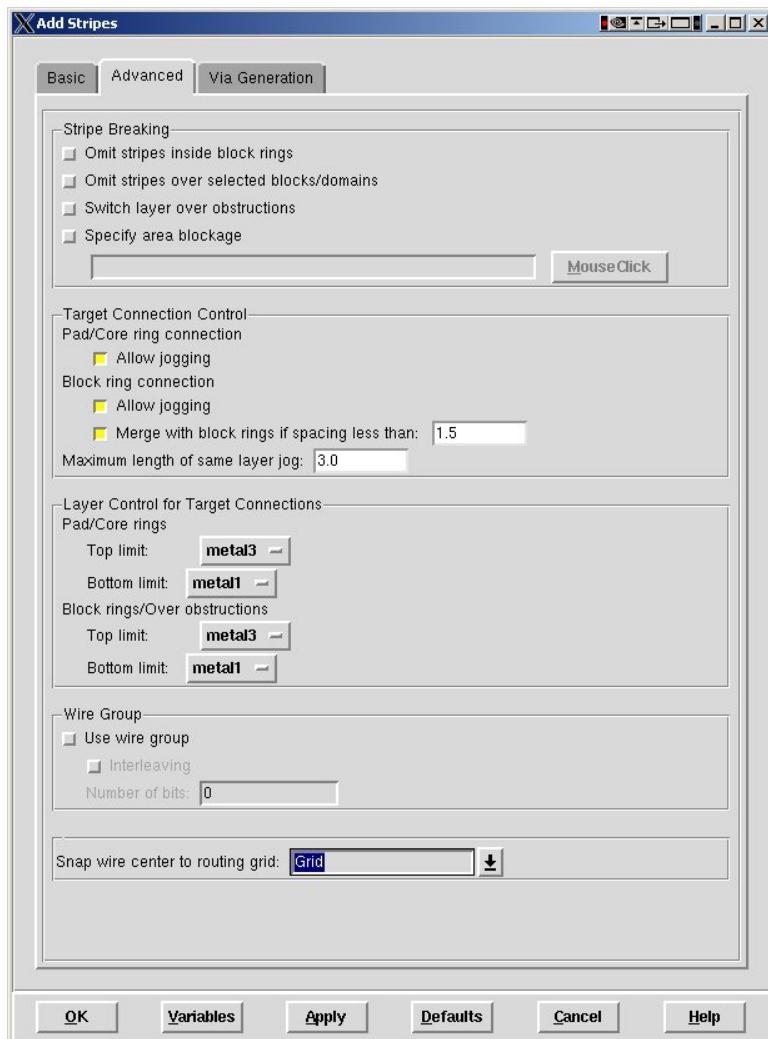


Figure 10.13: Advanced Tab of Dialog box for planning power stripes

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)** 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.

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

Note that you'll get a number of warnings at this step complaining that there are no cells and no pins to be routed. That's fine. We haven't placed any cells yet so the tool is correct - there are no cells or pins yet. But, our cells will be placed in the rows such that they'll use the power and ground wires so everything will be fine.

#### 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 10.15. You want to **Run Full Placement** and **Include 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 that looks like a little transistor) and you'll see where each of your cells has been placed. The placed counter looks like that in Figure 10.16.

If your design had more than one floorplan element 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. A floorplan element is basically a piece of your design hierarchy as represented by the hierarchy of modules in your original structural Verilog code. In this example there was only one module with no instantiated modules, just instances of standard cells.

#### 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** → **Optimize**. Notice that

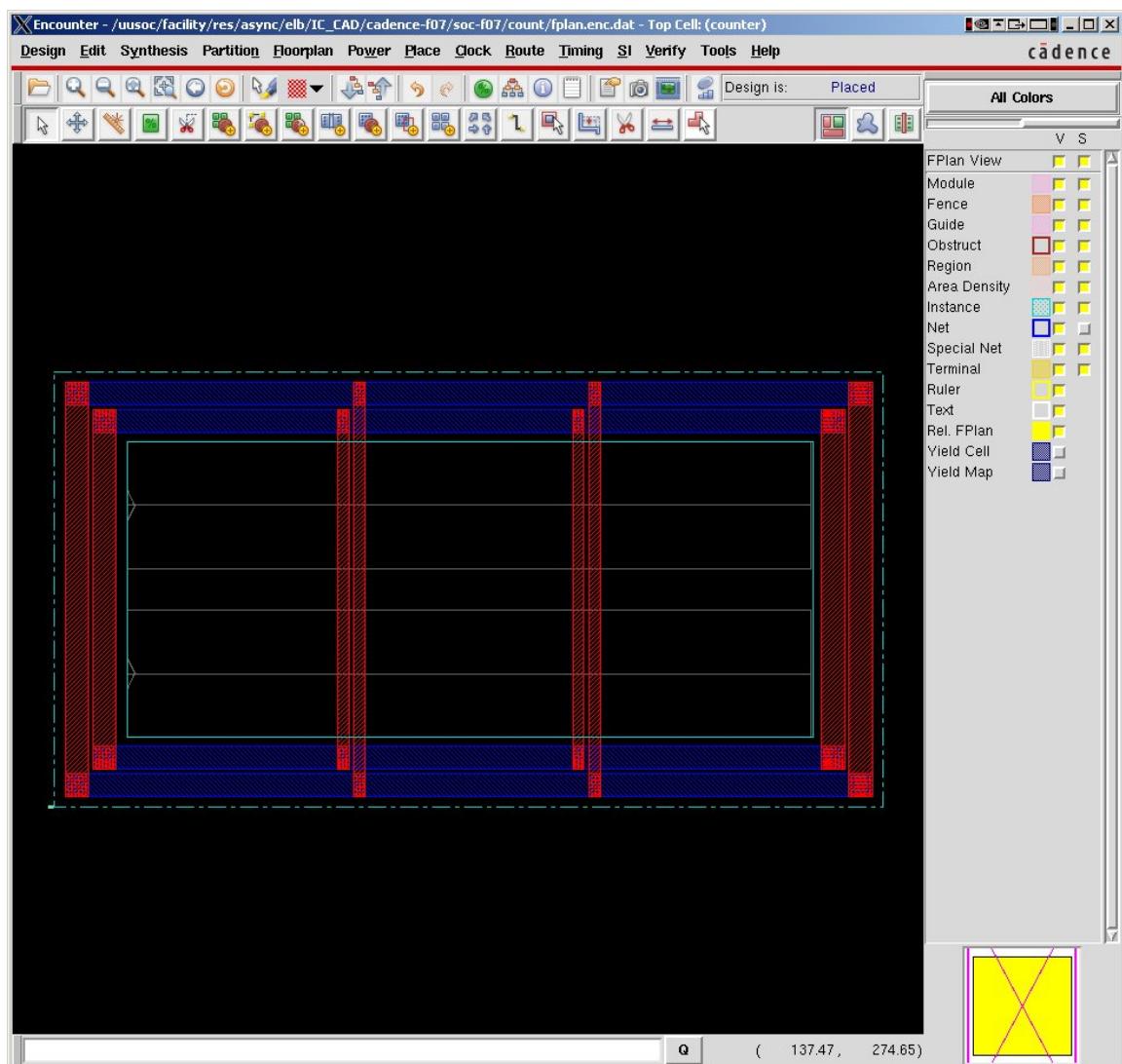


Figure 10.14: Floorplan after power rings and stripes have been generated and connected to the cell rows



Figure 10.15: Placement dialog box

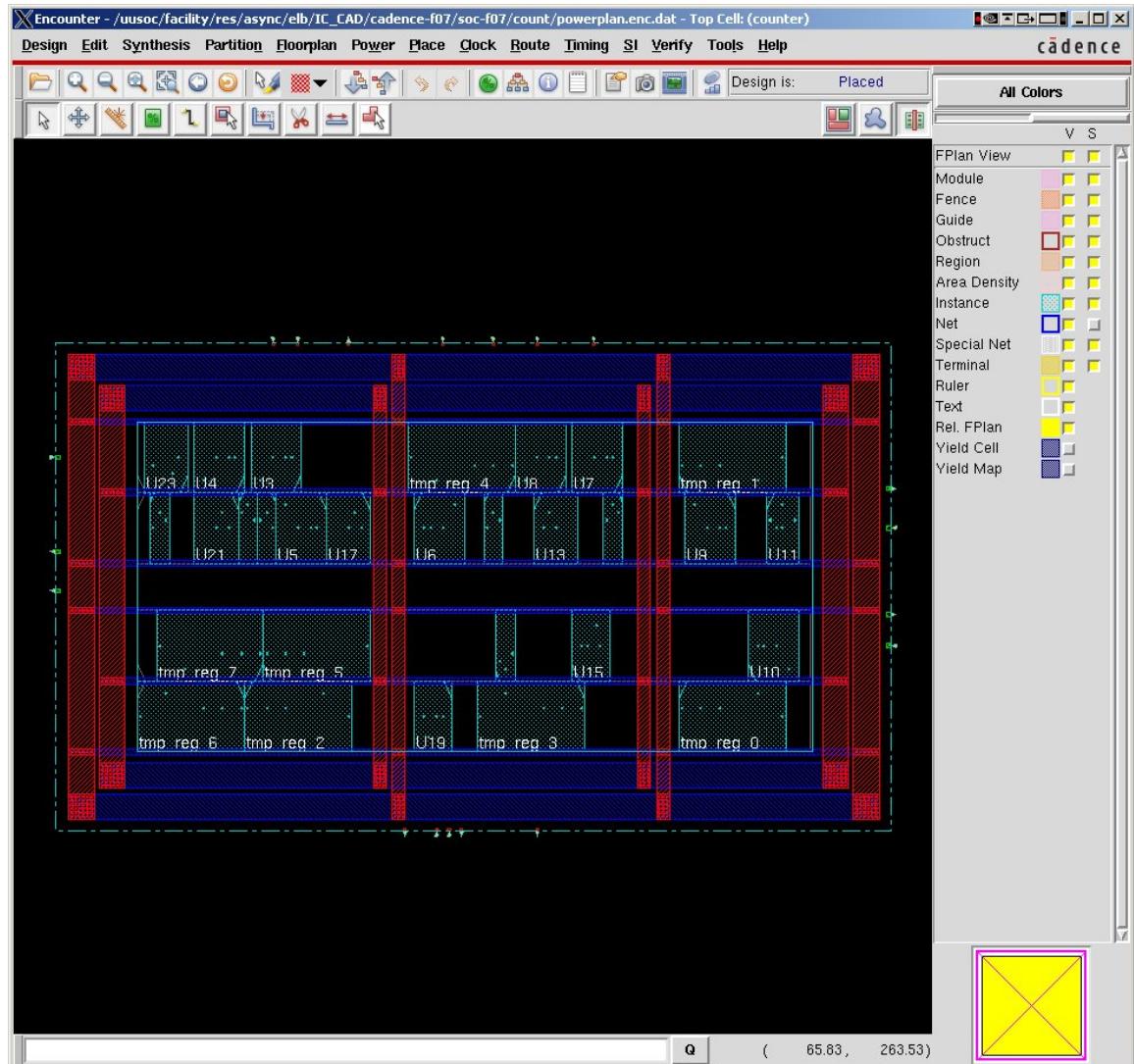


Figure 10.16: View after placement of the cells

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.

This would be a good time to save the design as **pre-cts.enc**.

### 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** → **Design Clock** to start (Figure 10.19). Because we not have a **Clock Specification File** yet, you can use the **Gen Spec...** button to create one.

Because we filled in the footprint information when we imported the design, you should see the inverters (cells with footprint **inv**) already selected. Clicking **OK** will generate a clock tree specification in the (default) **counter.ctstch** file.

Now you can synthesize the clock tree by clicking **OK** in the **Synthesize Clock Tree** dialog box (Figure 10.20). This will use the cells you told it to use 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 10.21). 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

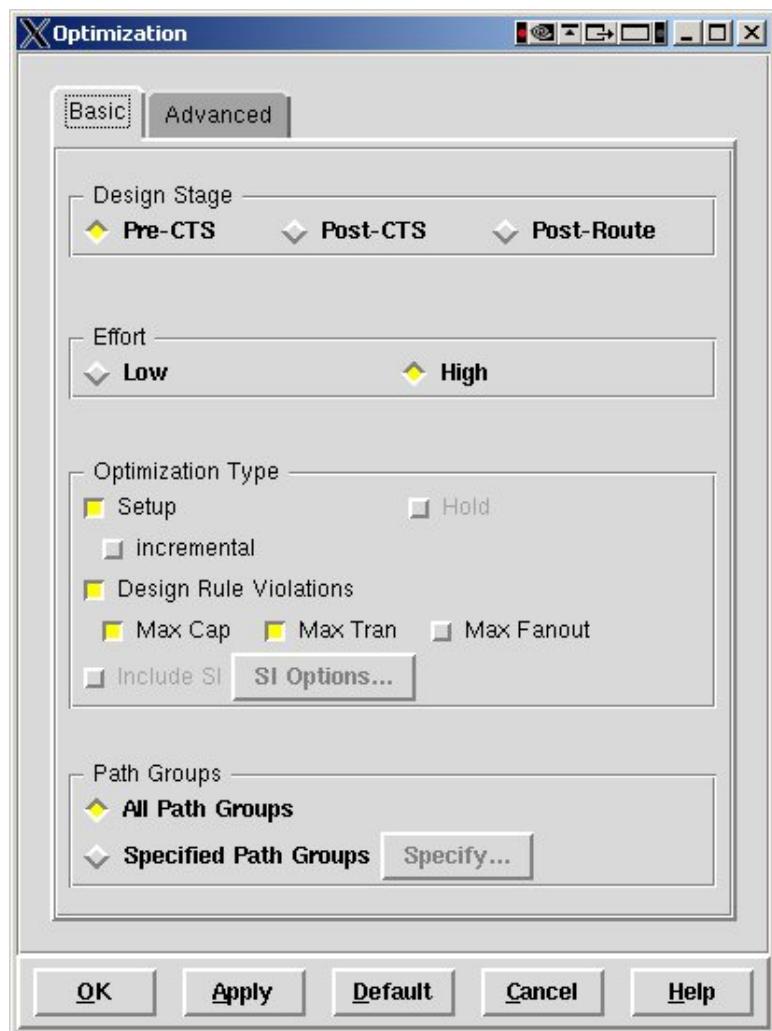


Figure 10.17: Dialog box for timing optimization

-----  
optDesign Final Summary  
-----

Setup mode	all	reg2reg	in2reg	reg2out	in2out	clkgate
WNS (ns):	-0.602	-0.602	1.197	1.177	N/A	N/A
TNS (ns):	-2.209	-2.209	0.000	0.000	N/A	N/A
Violating Paths:	5	5	0	0	N/A	N/A
All Paths:	24	8	16	8	N/A	N/A

Figure 10.18: Initial pre-CTS timing results

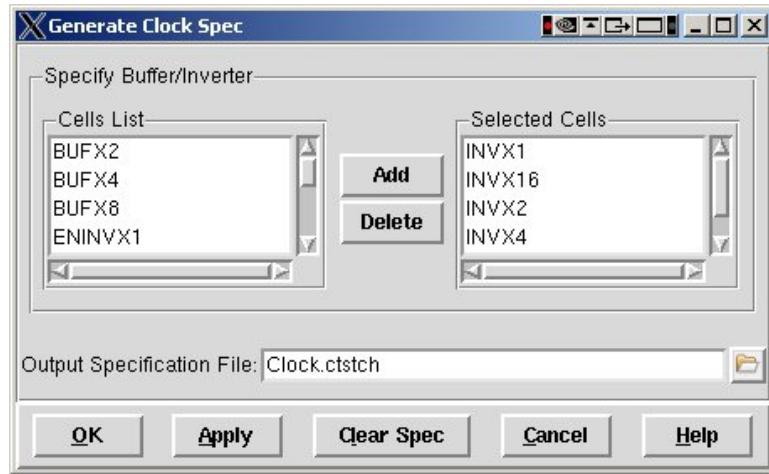


Figure 10.19: Generating a clock tree specification

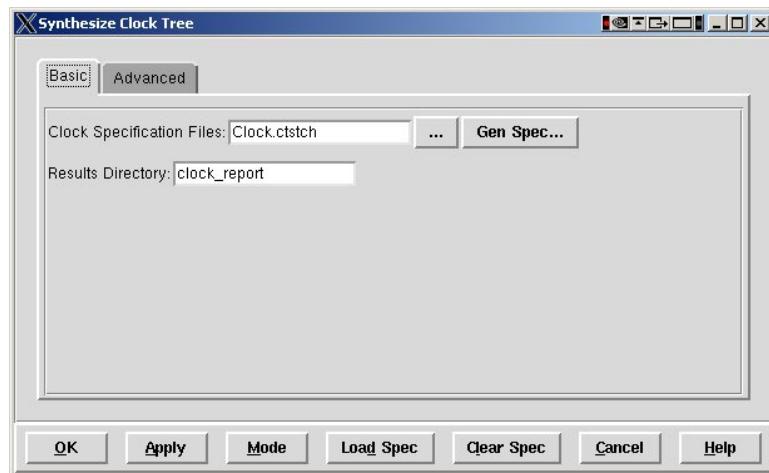


Figure 10.20: Synthesize Clock Tree dialog box

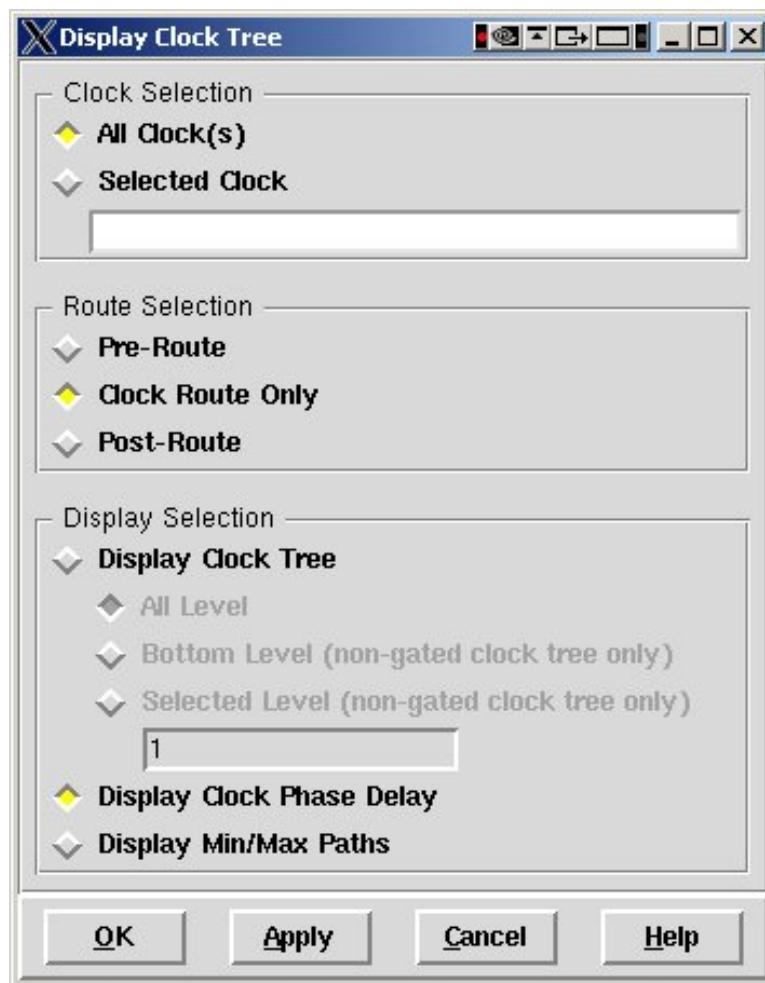


Figure 10.21: Dialog box to display the clock tree

see the min and max path from the clock pin to the flip flop displayed. See Figure 10.22 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 → Optimize** 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 tiny bit, but not much. In a larger design this would have a much more dramatic impact. See Figure 10.23.

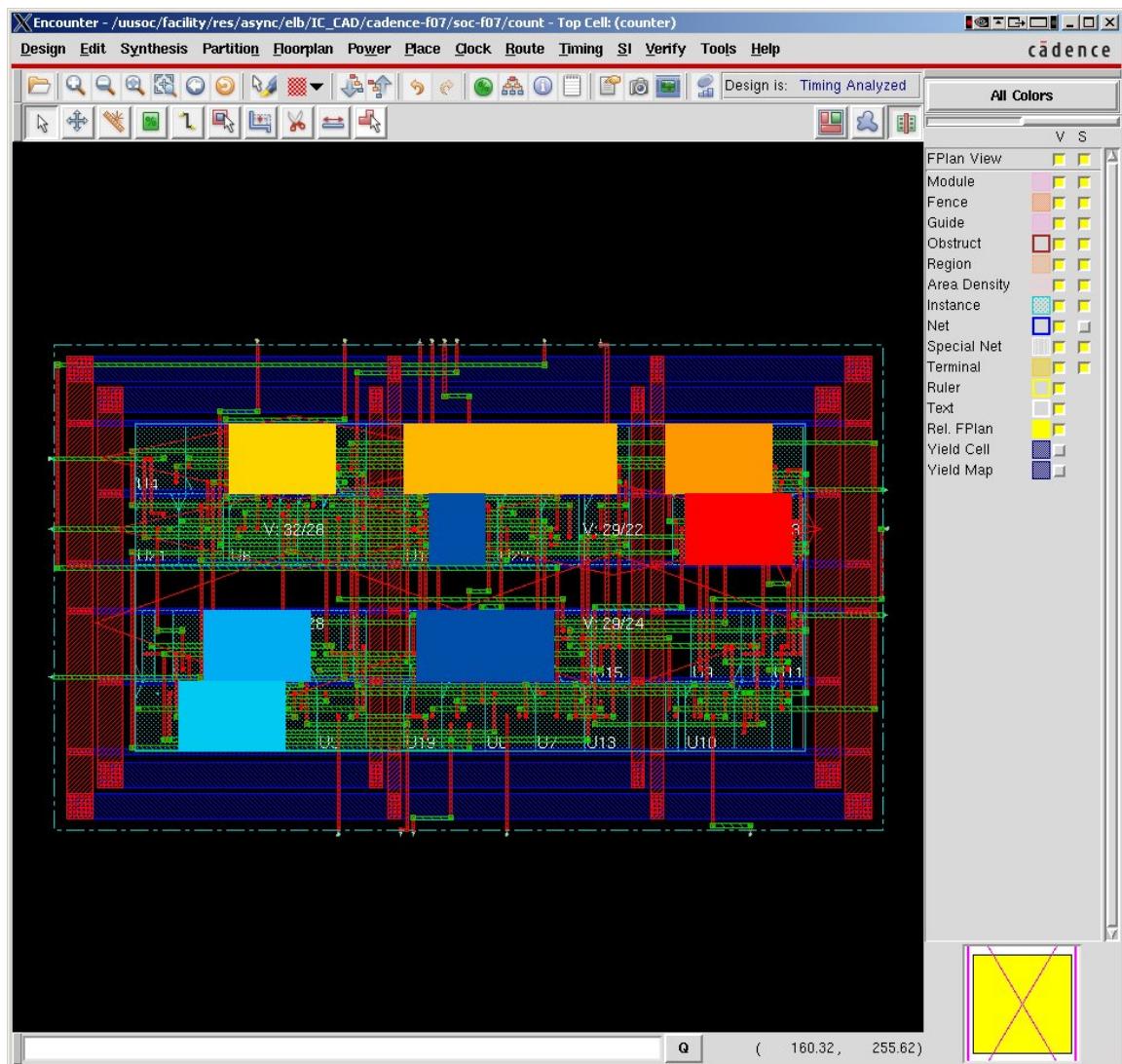


Figure 10.22: Design display showing the clock tree

optDesign Final Summary						
Setup mode	all	reg2reg	in2reg	reg2out	in2out	clkgate
WNS (ns):	-0.602	-0.602	1.507	1.072	N/A	N/A
TNS (ns):	-2.272	-2.272	0.000	0.000	N/A	N/A
Violating Paths:	5	5	0	0	N/A	N/A
All Paths:	24	8	16	8	N/A	N/A

Figure 10.23: Timing results after the second (**Post-CTS**) optimization

### 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 10.24). 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 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! For example, you may need to adjust the cell density or the space between pairs of rows to give the routed more breathing room. An example of a completely routed circuit is shown in Figure 10.25.

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

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.

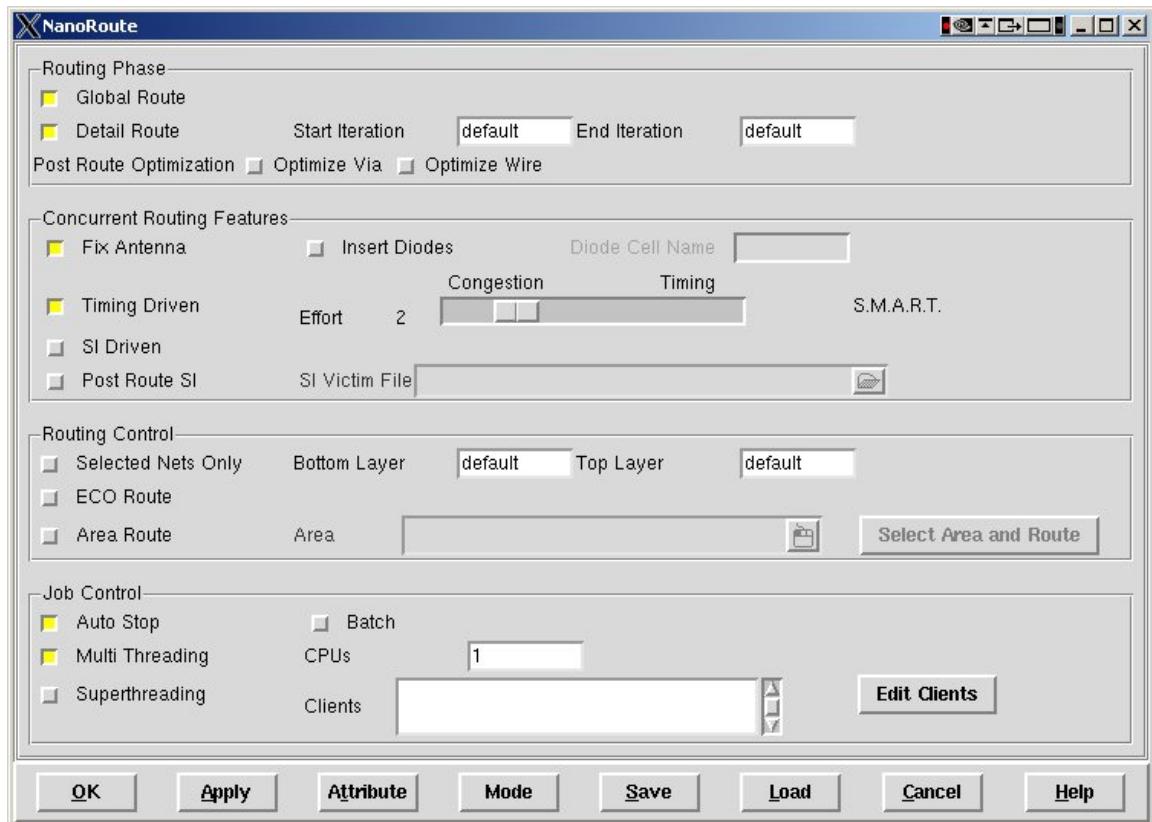


Figure 10.24: **NanoRoute** dialog box

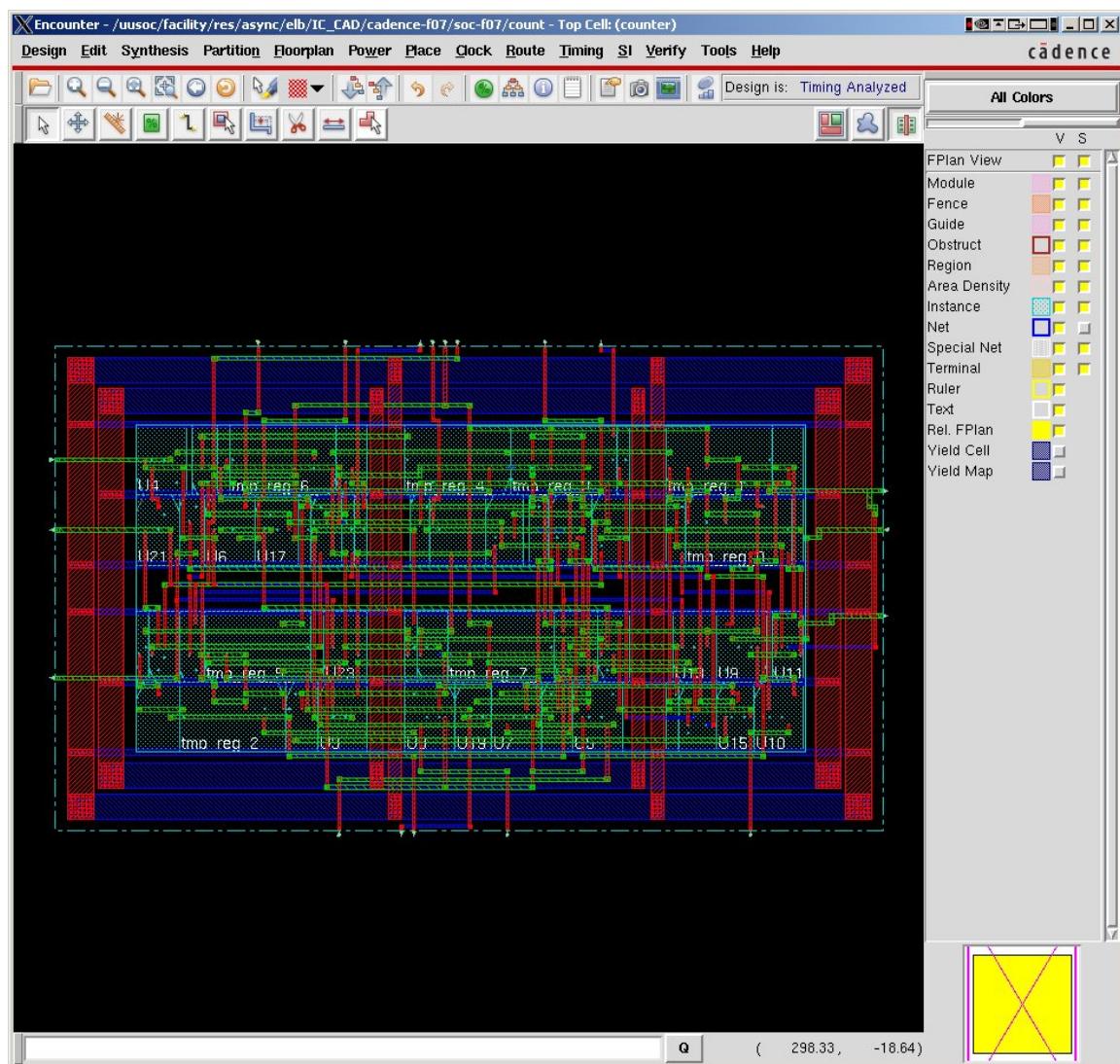


Figure 10.25: The counter circuit after using **NanoRoute** to do final routing

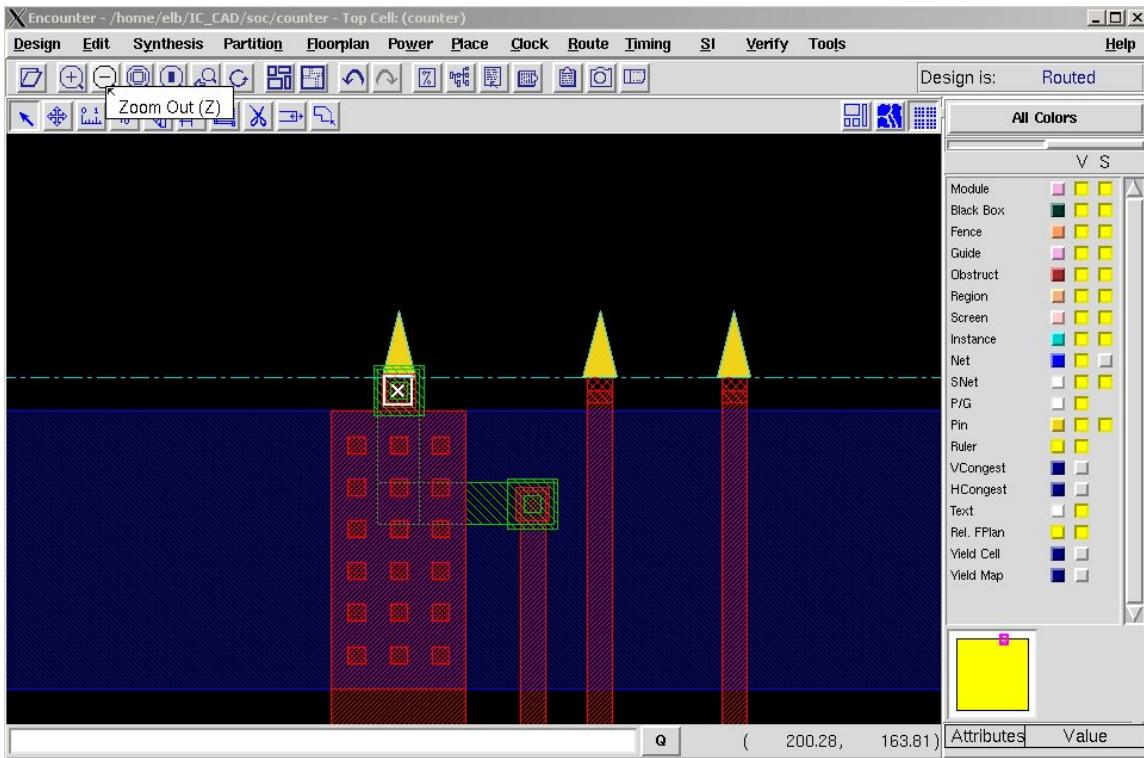


Figure 10.26: An error in pin placement after final routing

finished with **SOC Encounter**.

Figure 10.26 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 10.27.

#### 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. In fact, it's worse than the pre-CTS predictions. Apparently the real routing wasn't as nice as the trial routing and predicted delays. See Figure 10.28.

#### 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

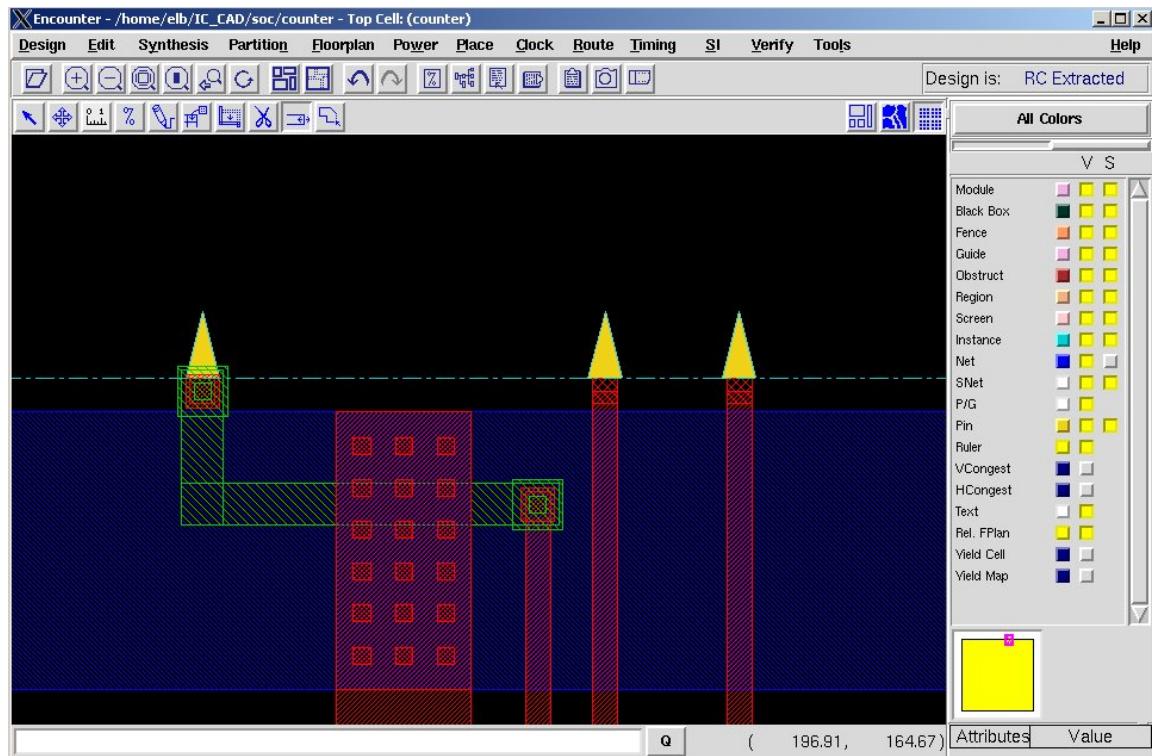


Figure 10.27: Design display after fixing the pin routing error

```
-----  
optDesign Final Summary  
-----  
  
+-----+-----+-----+-----+-----+-----+-----+  
| Setup mode | all | reg2reg | in2reg | reg2out | in2out | clkgate |  
+-----+-----+-----+-----+-----+-----+-----+  
| WNS (ns): | -0.834 | -0.834 | 1.568 | 1.016 | N/A | N/A |  
| TNS (ns): | -3.631 | -3.631 | 0.000 | 0.000 | N/A | N/A |  
| Violating Paths: | 5 | 5 | 0 | 0 | N/A | N/A |  
| All Paths: | 24 | 8 | 16 | 8 | N/A | N/A |  
+-----+-----+-----+-----+-----+-----+
```

Figure 10.28: Final post-route timing optimization results

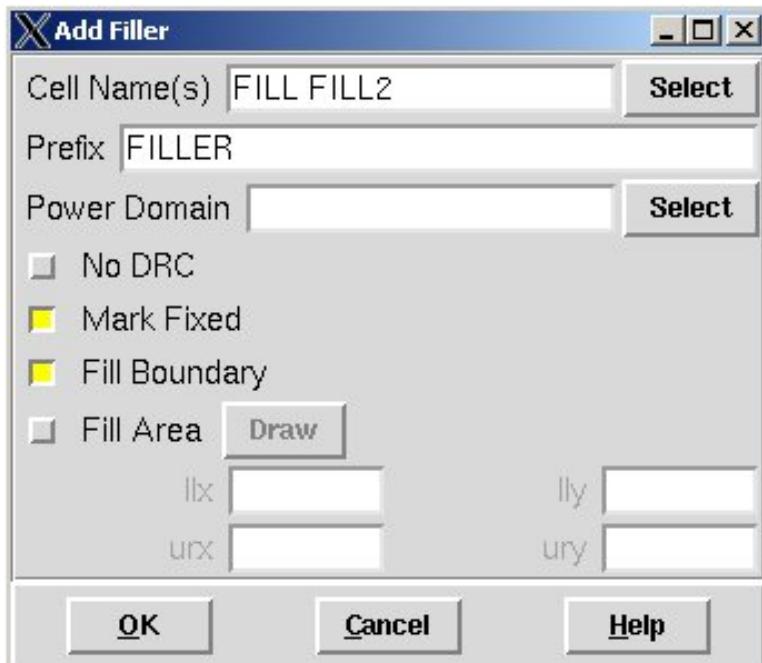


Figure 10.29: Filler cell dialog box

**Place** → **Filler** → **Add Filler** 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 10.29). Clicking **OK** will fill all the gaps in the rows with filler cells for a final cell layout as seen in Figure 10.30. You can also zoom around in the layout to see how things look and how they're connected as in Figure 10.31.

### 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 10.32 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 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.

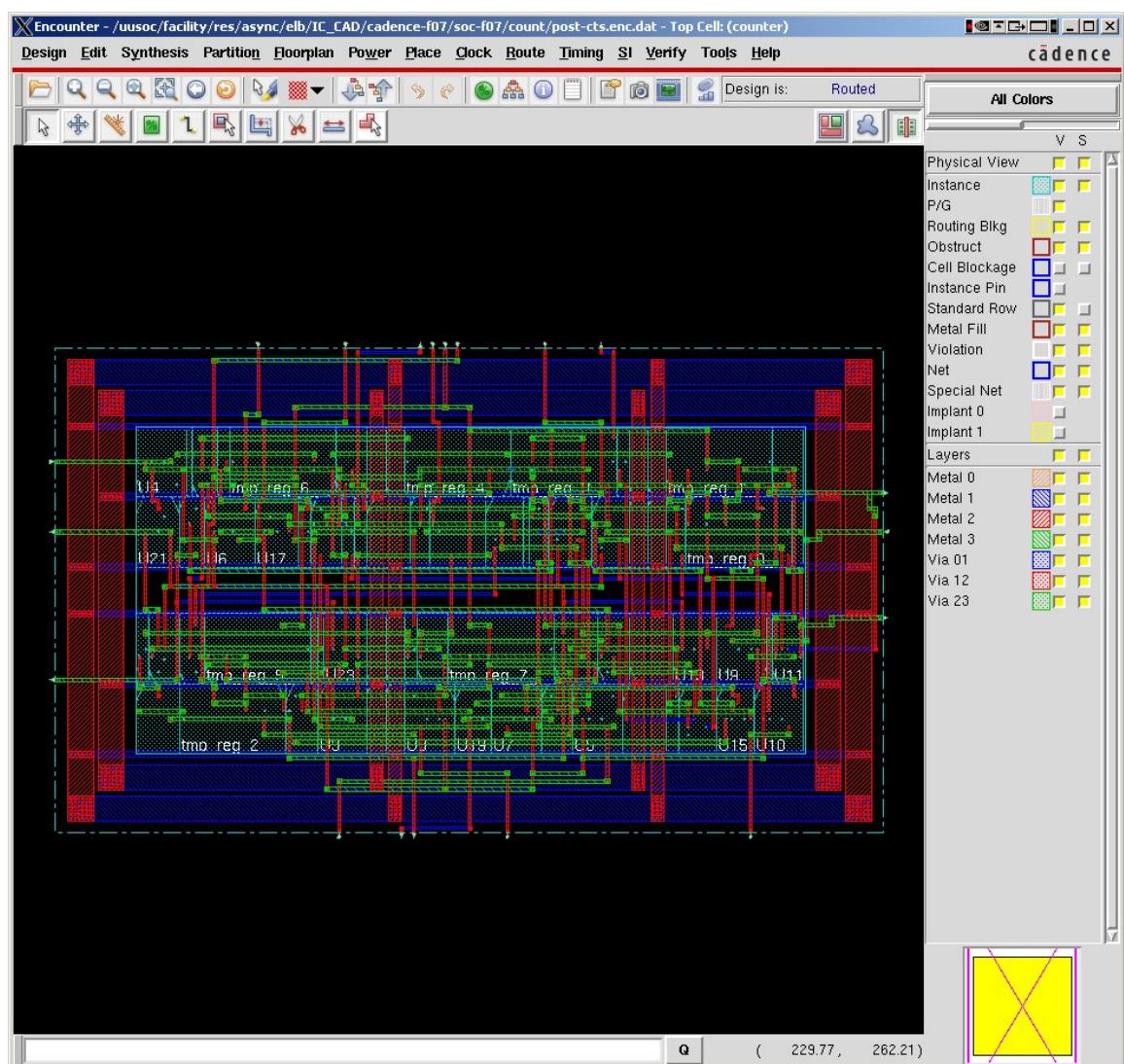


Figure 10.30: Final cell layout after filler cells are added

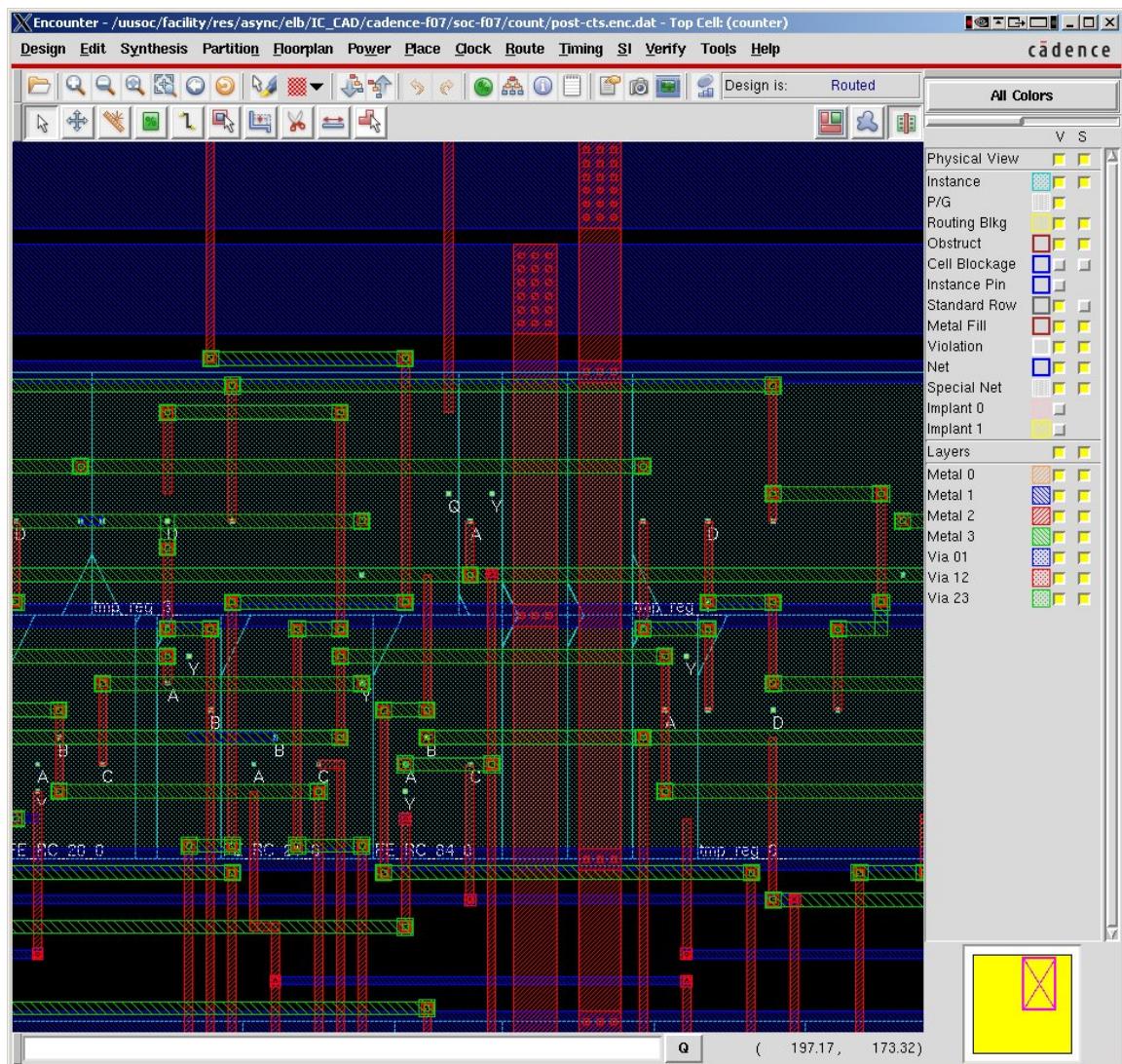


Figure 10.31: A zoomed view of a final routed cells

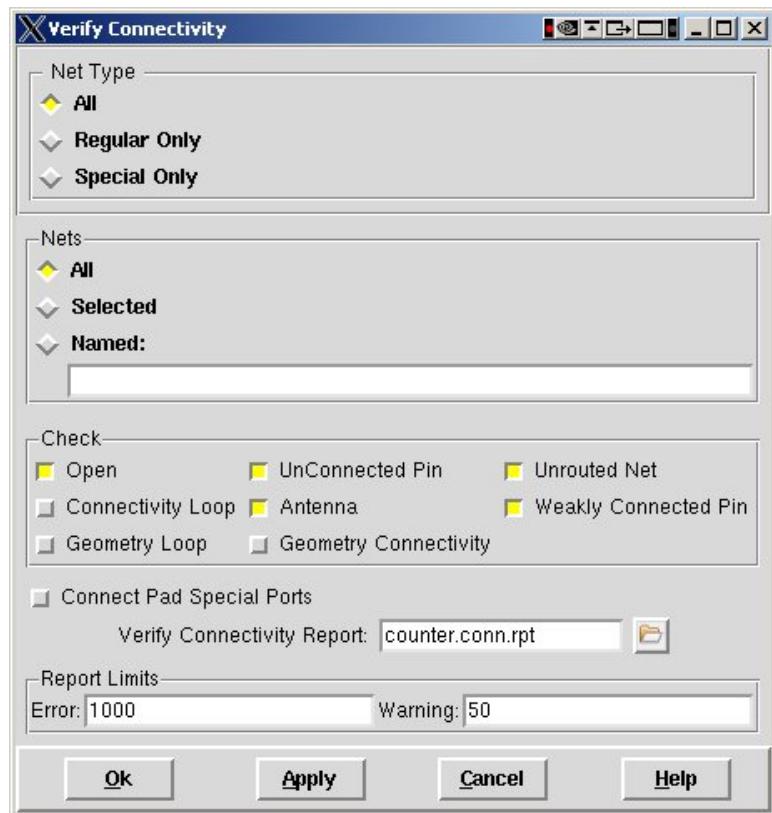


Figure 10.32: Dialog box for verifying connectivity

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 10.33. 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.

**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 10.34) you can leave the default selections, but change the **Output DEF Version** 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. Give it a new name to make sure you know that this is the cell with the **SOC** modifications.

**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

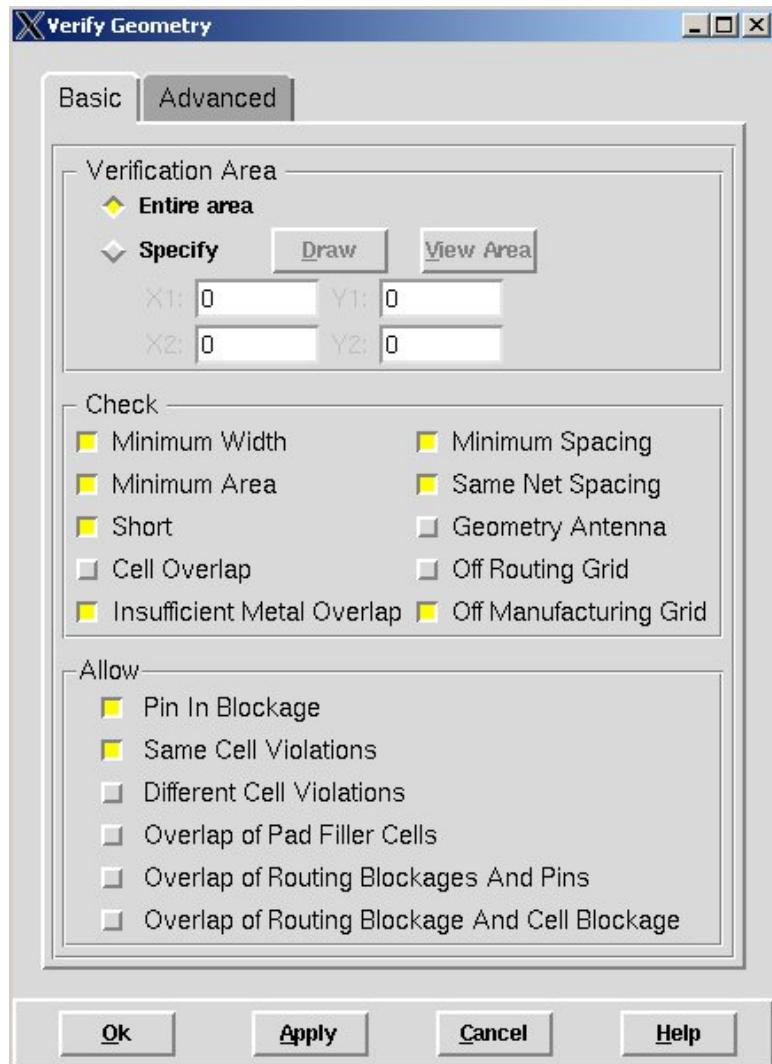


Figure 10.33: Dialog box for checking geometry



Figure 10.34: Dialog box for exporting **DEF**

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 depending on what library you're using.

**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 10.35. 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**

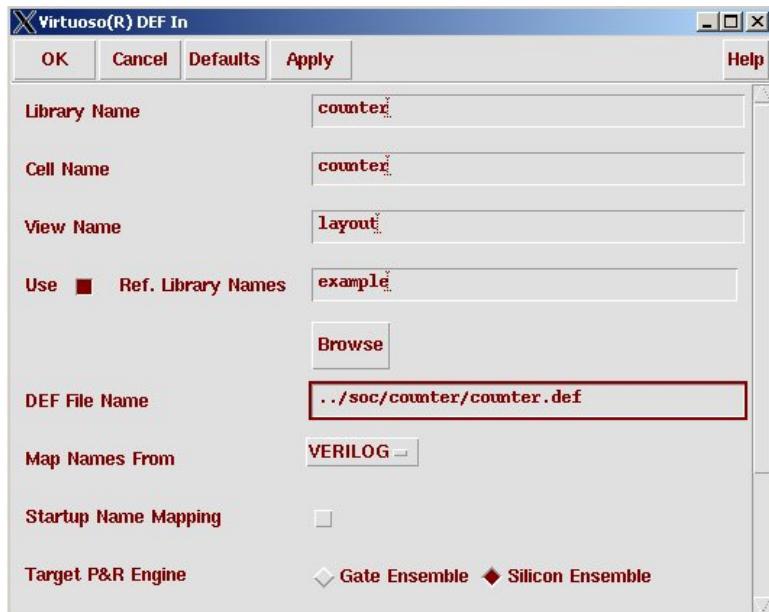


Figure 10.35: Dialog box for importing DEF files to **icfb**

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 10.36). 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 10.37). 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.

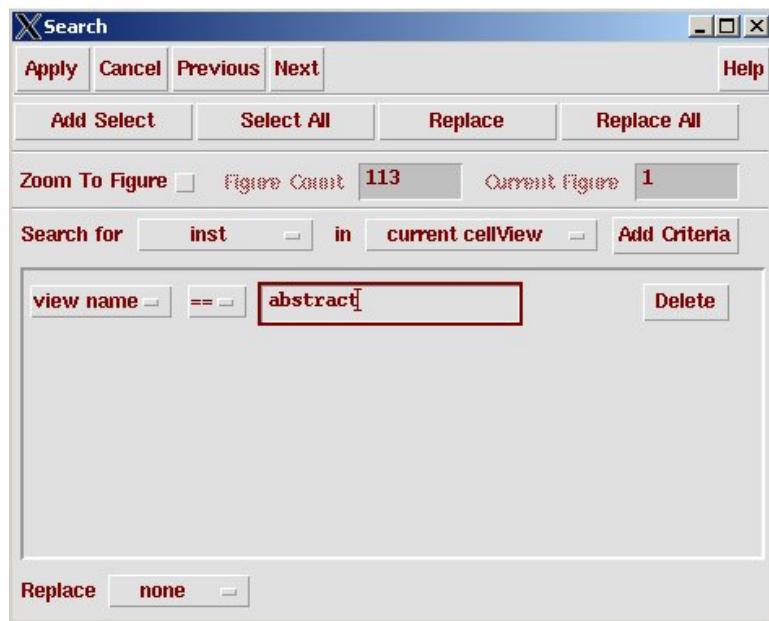


Figure 10.36: **Search** dialog box for finding all the cell abstracts

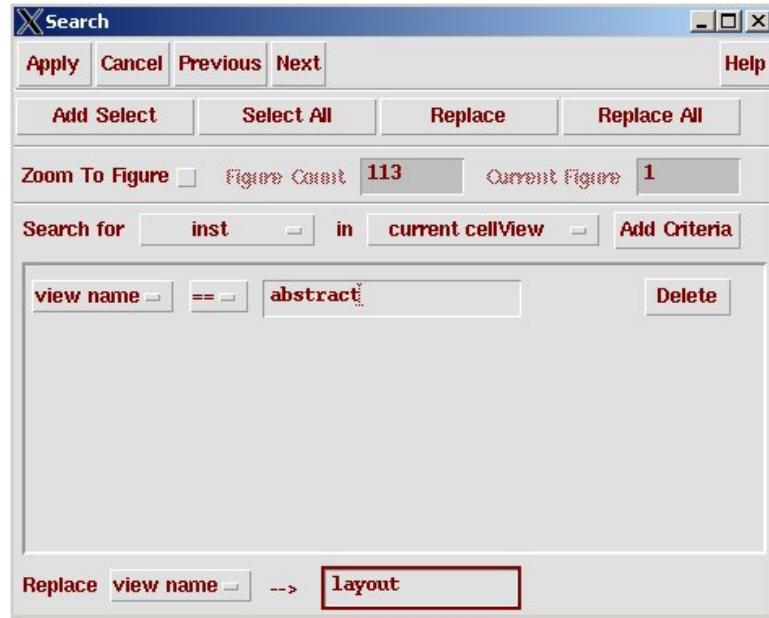


Figure 10.37: **Search** dialog box for replacing abstracts with layouts

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 10.38. 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 10.39) and symbol (Figure 10.40) 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**.

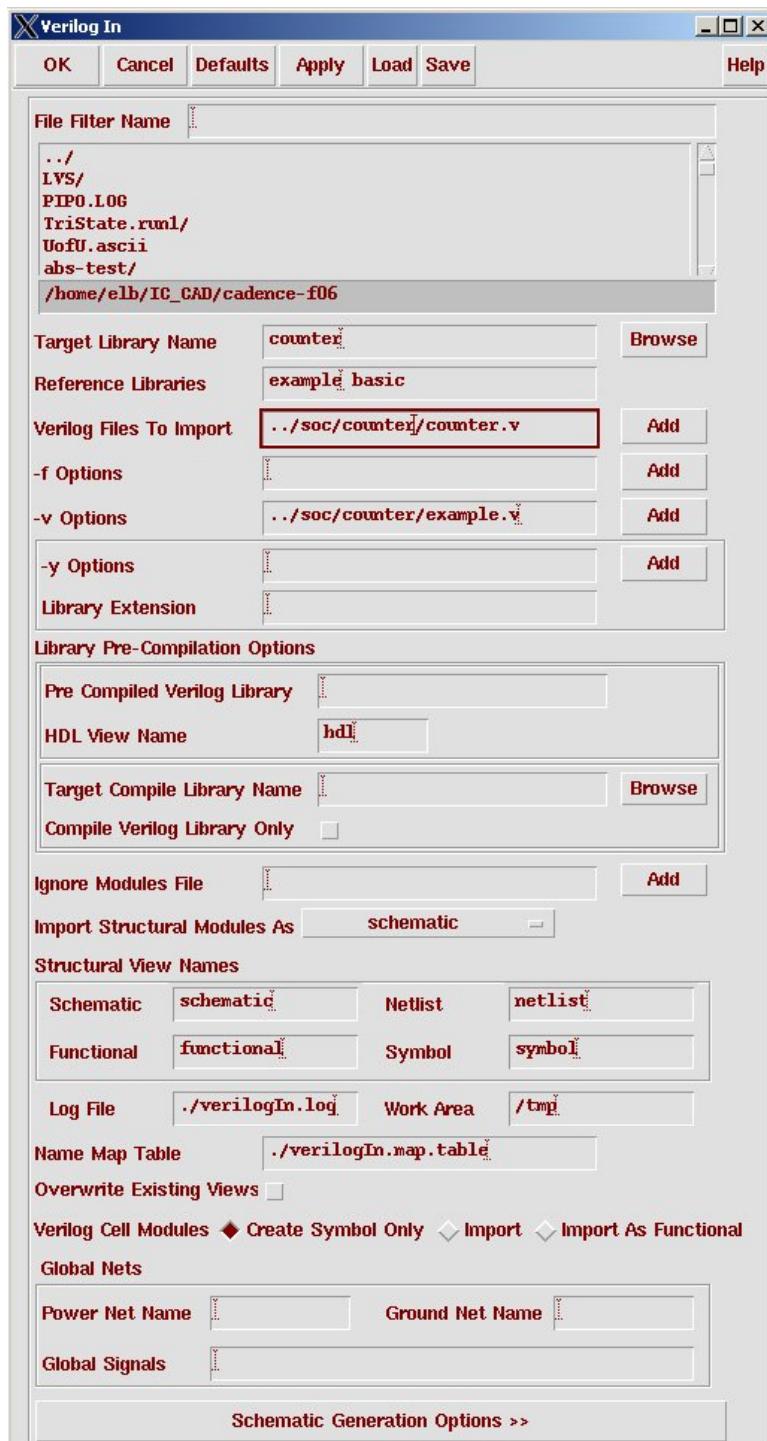


Figure 10.38: Dialog box for importing structural Verilog into a new schematic view

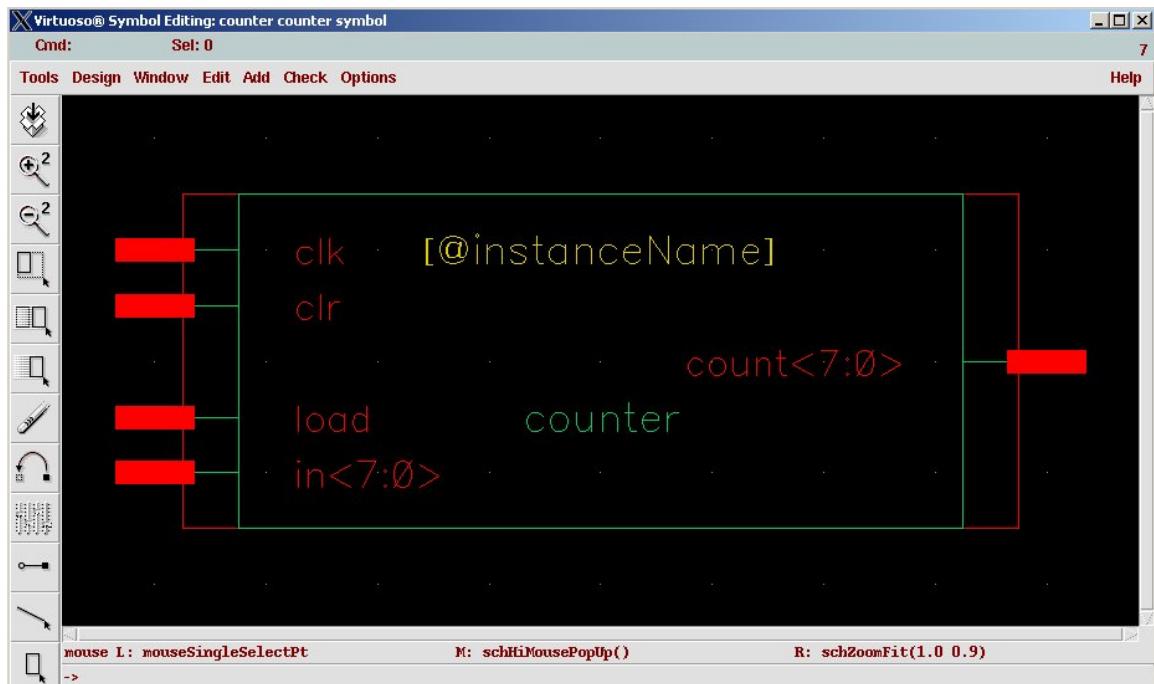


Figure 10.39: Schematic that results from importing the structural counter from **SOC Encounter** into **icfb**

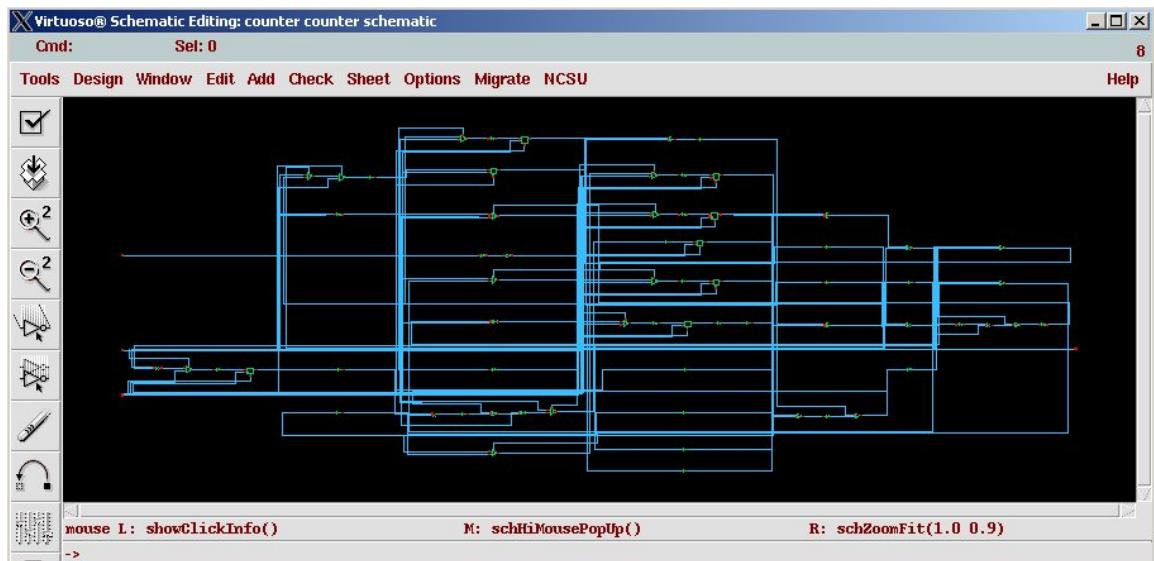


Figure 10.40: Symbol that is created for the counter

## 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 10.41. The full configuration file template is shown in Appendix C, and, of course, you can see it on-line.

*Warning - the following text was tested for version 5.2 of SOC Encounter, but has not yet been tested for version 6.2. I believe it should work, but there may be tweaks needed. You can see the syntax of the command-line commands from the “encounter.log” file generated when you ran things with the GUI.*

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.

```
#####
#
#   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)       {counter}
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 10.41: Configuration file for reading in the **counter** example

### 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 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 10.42 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 99
set soffset 90

# Import design and floorplan
# If the config file is not named $basename.conf, edit this line.
loadConfig $basename.conf 0
commitConfig

...
```

Figure 10.42: The first part of the **UofU\_opt.tcl** script for **SOC Encounter**



# 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** than 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

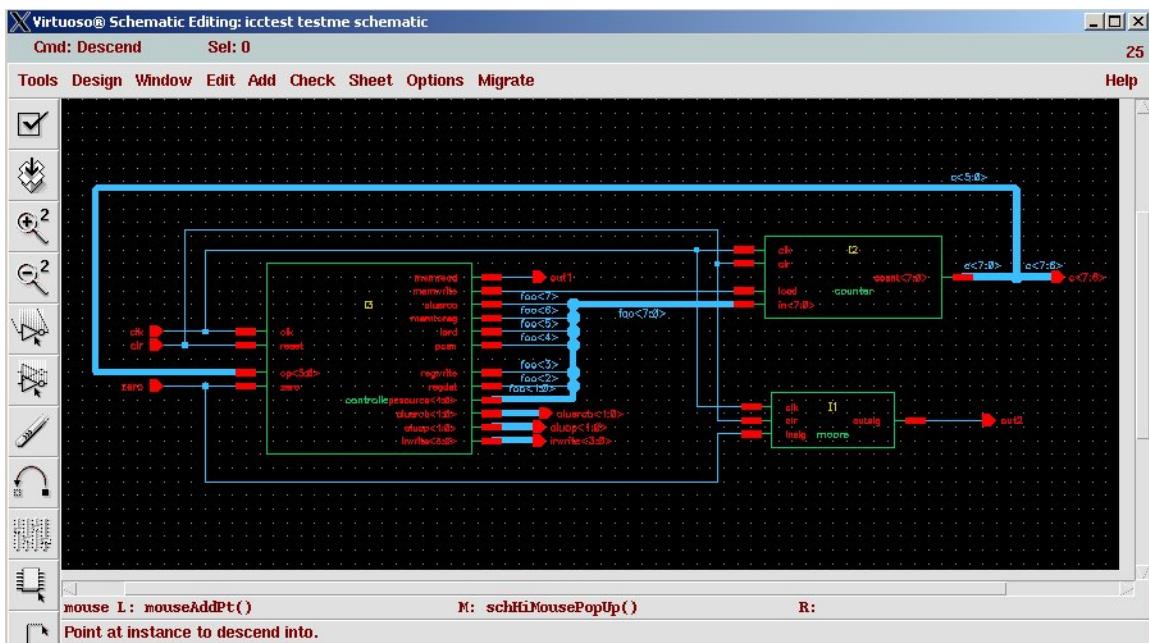


Figure 11.1: Starting **schematic** showing the three connected modules

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 an 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 **ctrl-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-

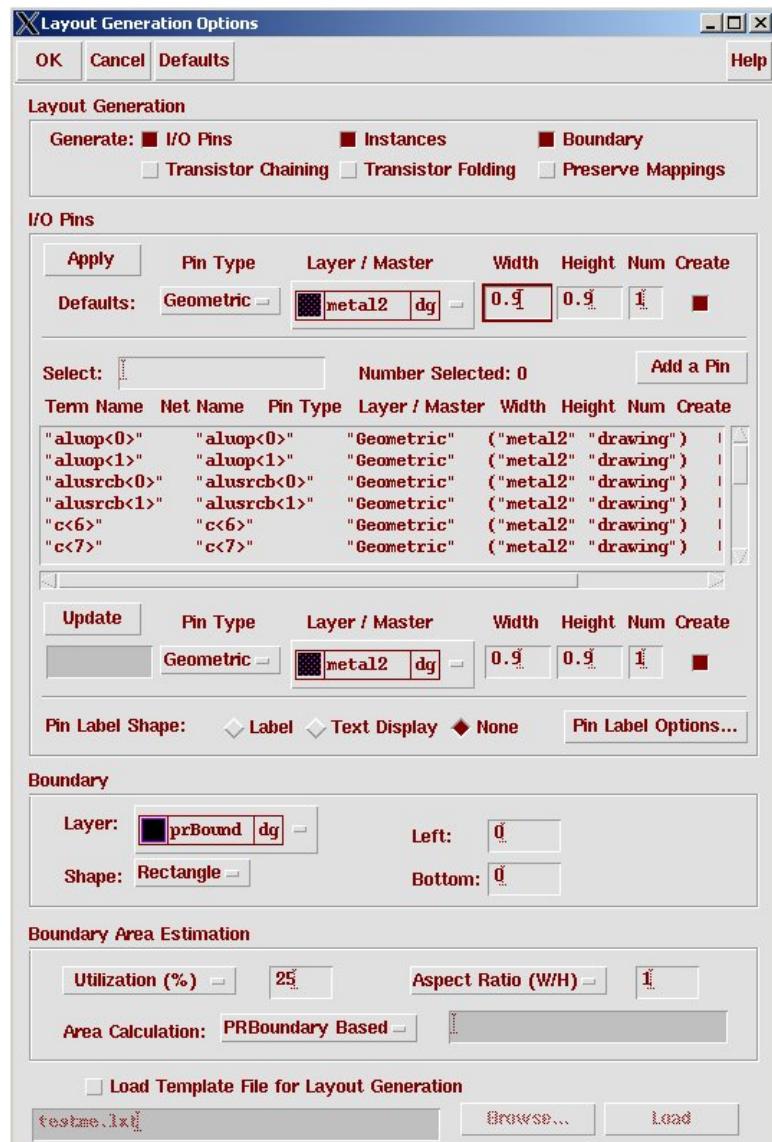


Figure 11.2: The **Gen From Source** dialog box

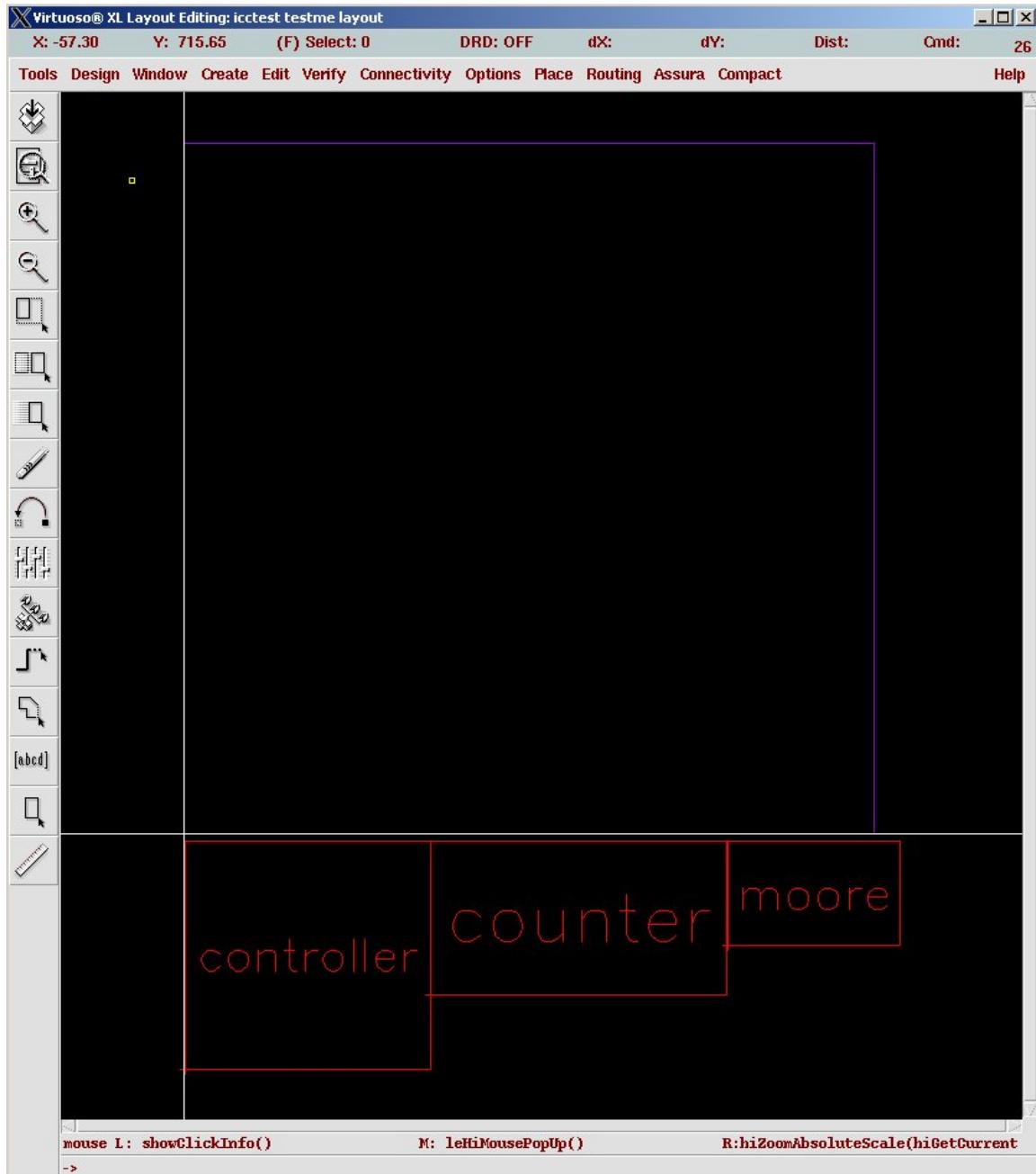


Figure 11.3: Initial layout before module and IO placement

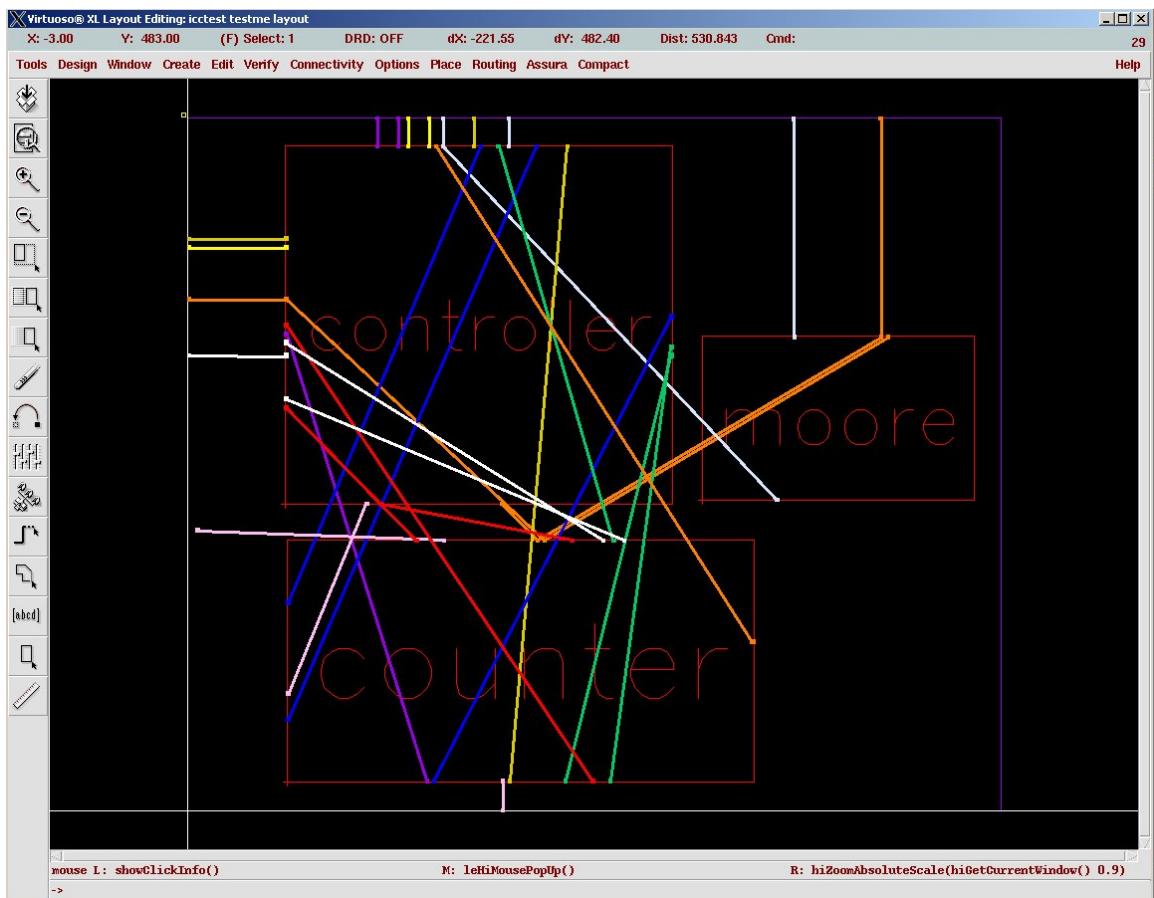


Figure 11.4: A placement of modules and IO pins with unrouted nets turned on

nnectivity 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.

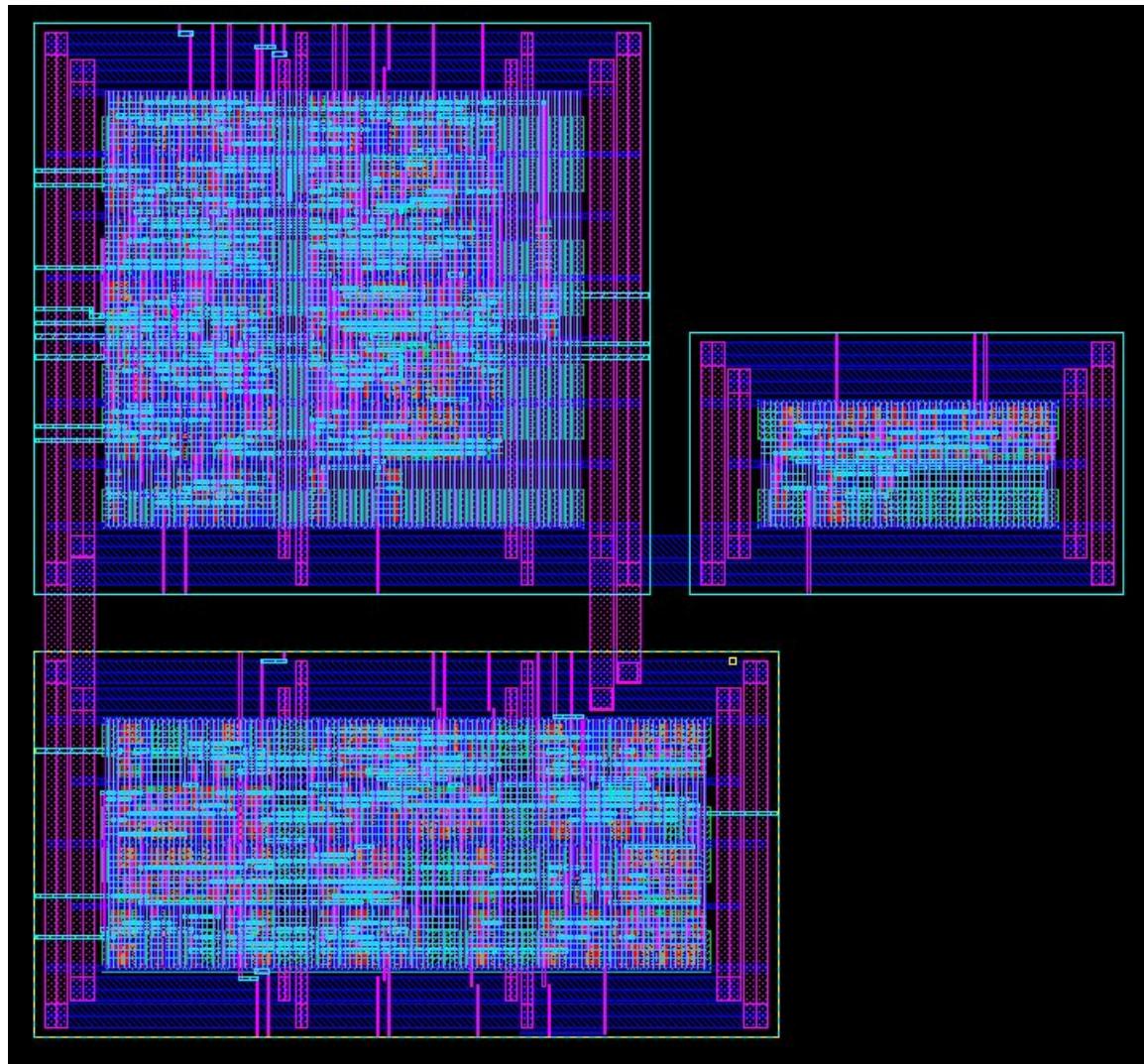


Figure 11.5: Layout showing placement and power routing before routing

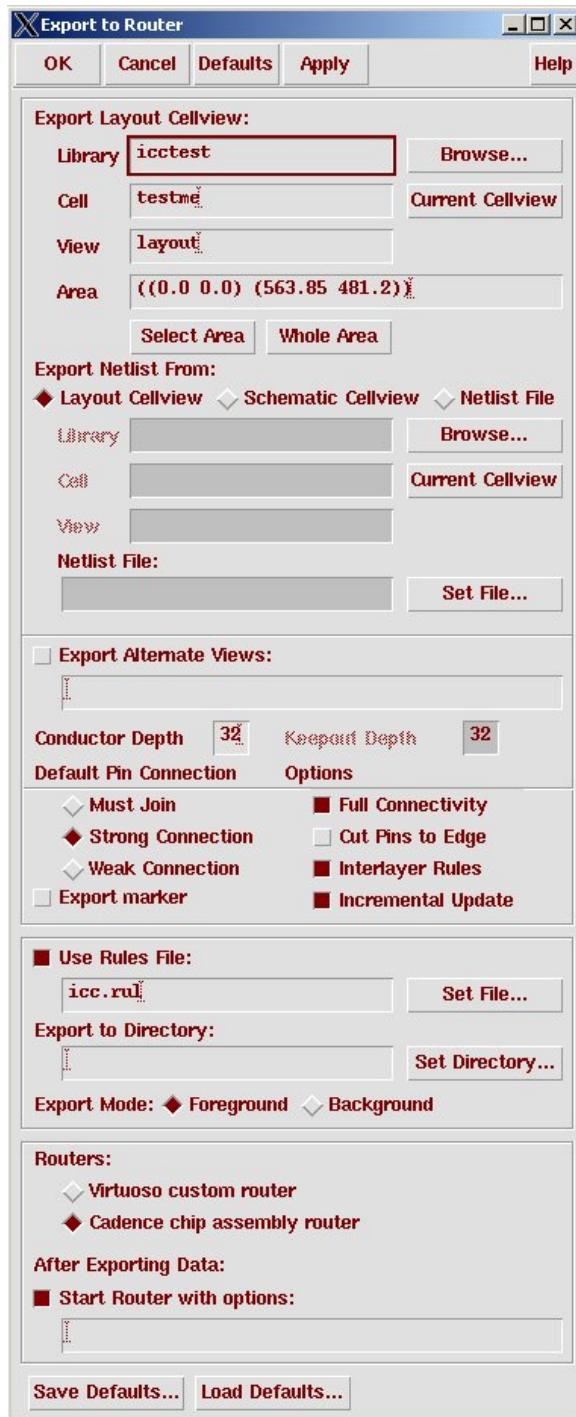


Figure 11.6: Export to Router dialog box

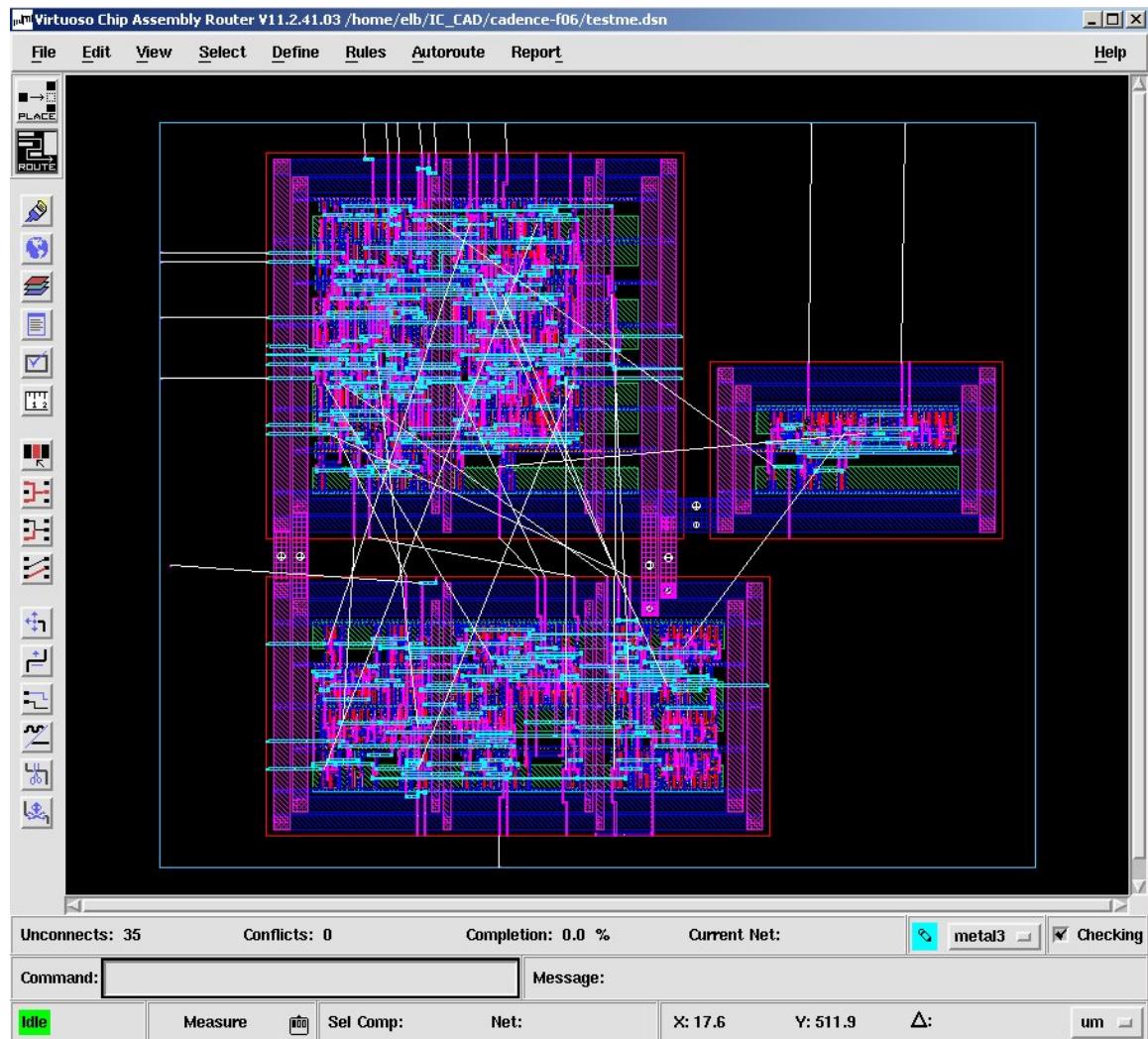


Figure 11.7: Initial **ccar** window



Figure 11.8: Layer configuration dialog box

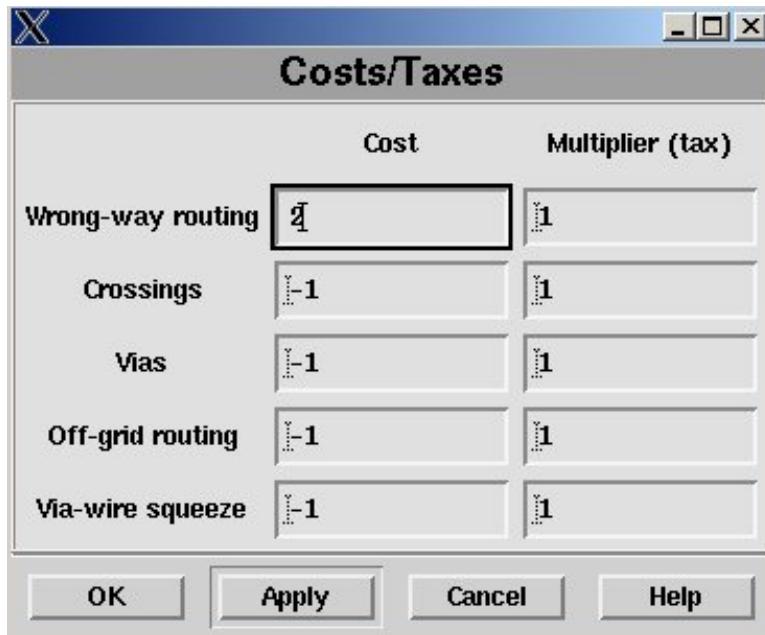


Figure 11.9: Routing cost factor dialog box

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 unroute (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 to 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 **cbar**. 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 **cbar** 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 **cbar** at another level of the hierarchy. The

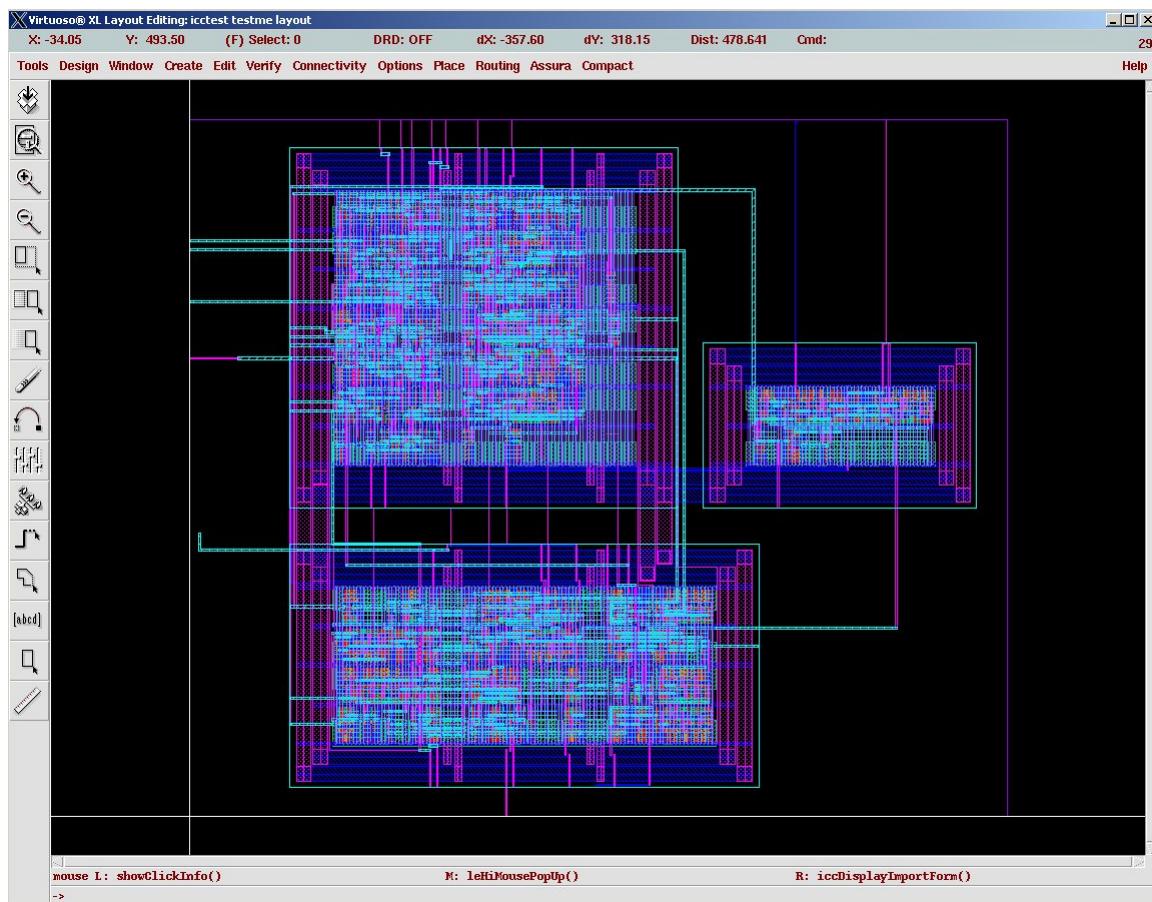


Figure 11.10: Final routed circuit (shown in **Virtuoso** window)

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.

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

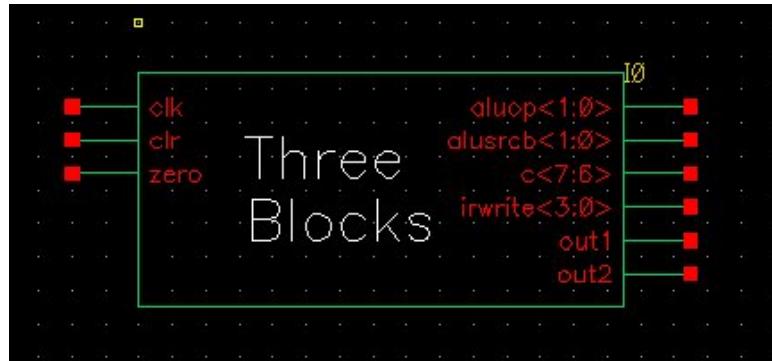


Figure 11.11: Symbol for the **Three Blocks** example core

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\_bidihe!**

**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

*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.*

This involves changing the appropriate **pad\_nc** cells into **pad\_in** and **pad\_out** cells in the **schematic** view. You can use the  Object Properties menu to change which cell is instantiated in that spot without moving the cell.

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

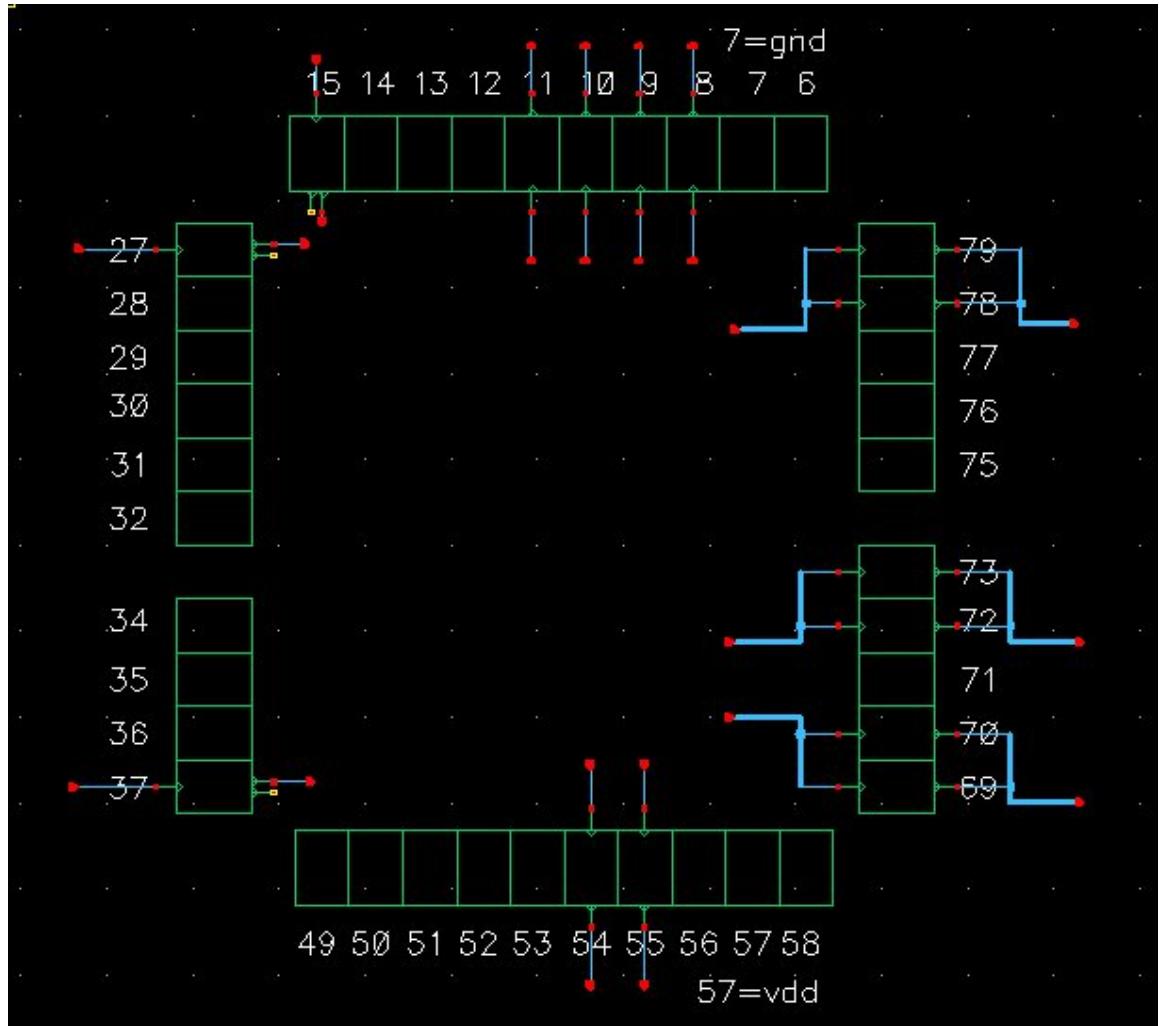


Figure 11.12: Pad frame with signal wires

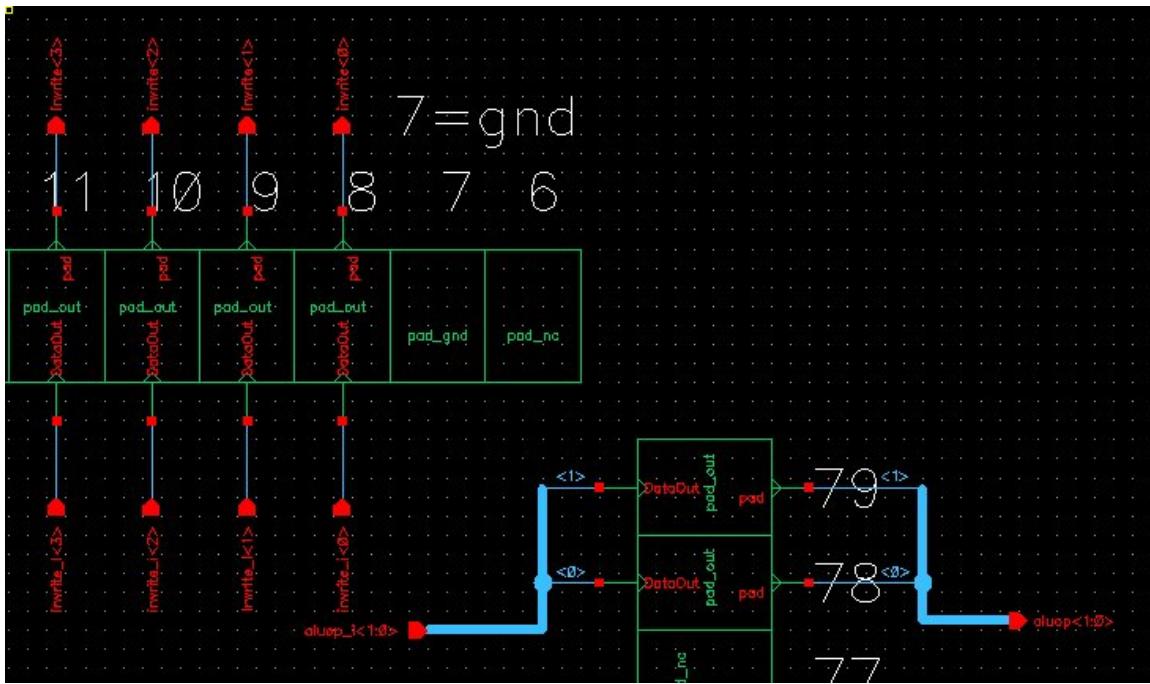


Figure 11.13: Pad frame with signal wires - zoomed view

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

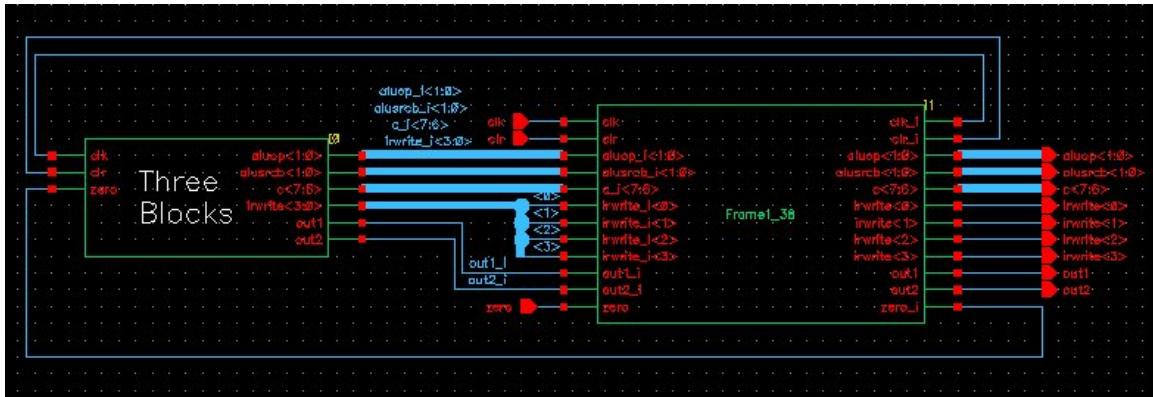


Figure 11.14: Frame and core components connected together

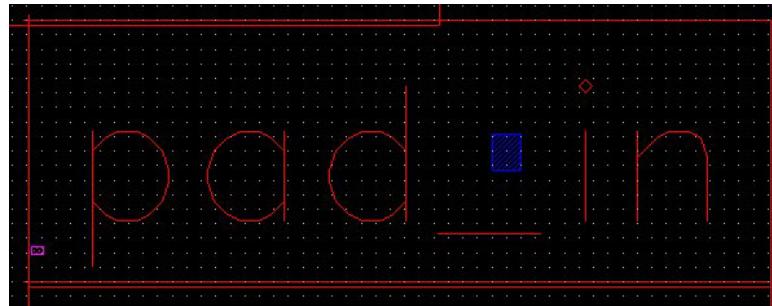
sure that the enable **EN** signal is low and that the outside world is driving a signal value.

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\_bidihr** cells to match the schematic. Again, you can do this by selecting the cell, getting the properties with **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.

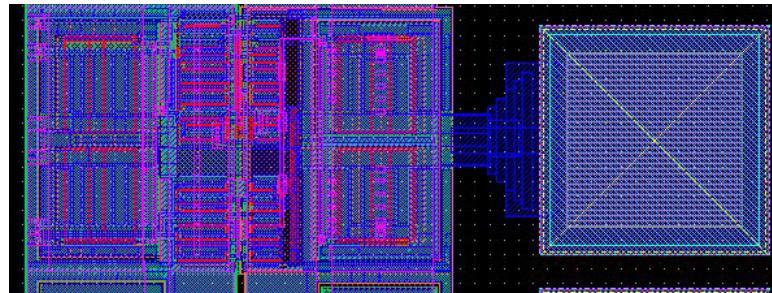


Figure 11.16: Expanded **pad.in** cell with **clk** and **clk.i** connections

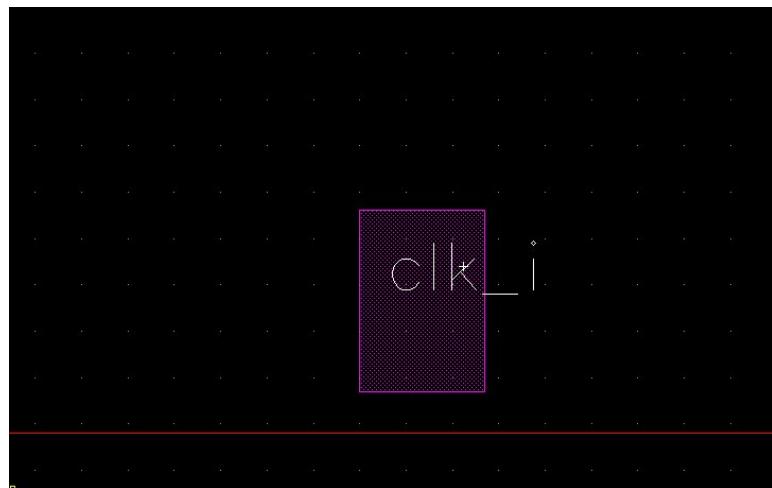


Figure 11.17: Detail of **clk.i** connection



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

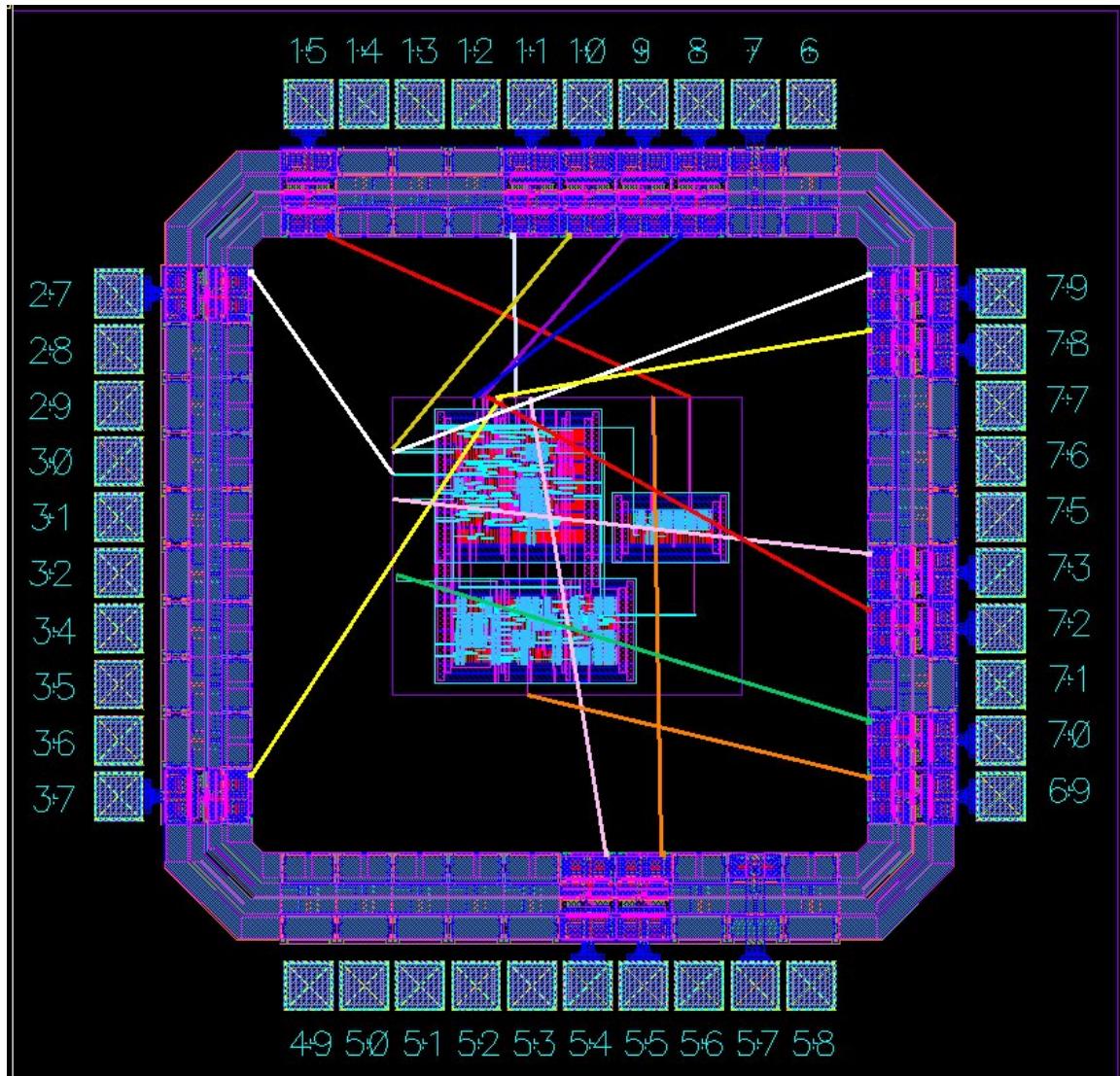


Figure 11.19: Frame and core placed in **Virtuoso-XL**

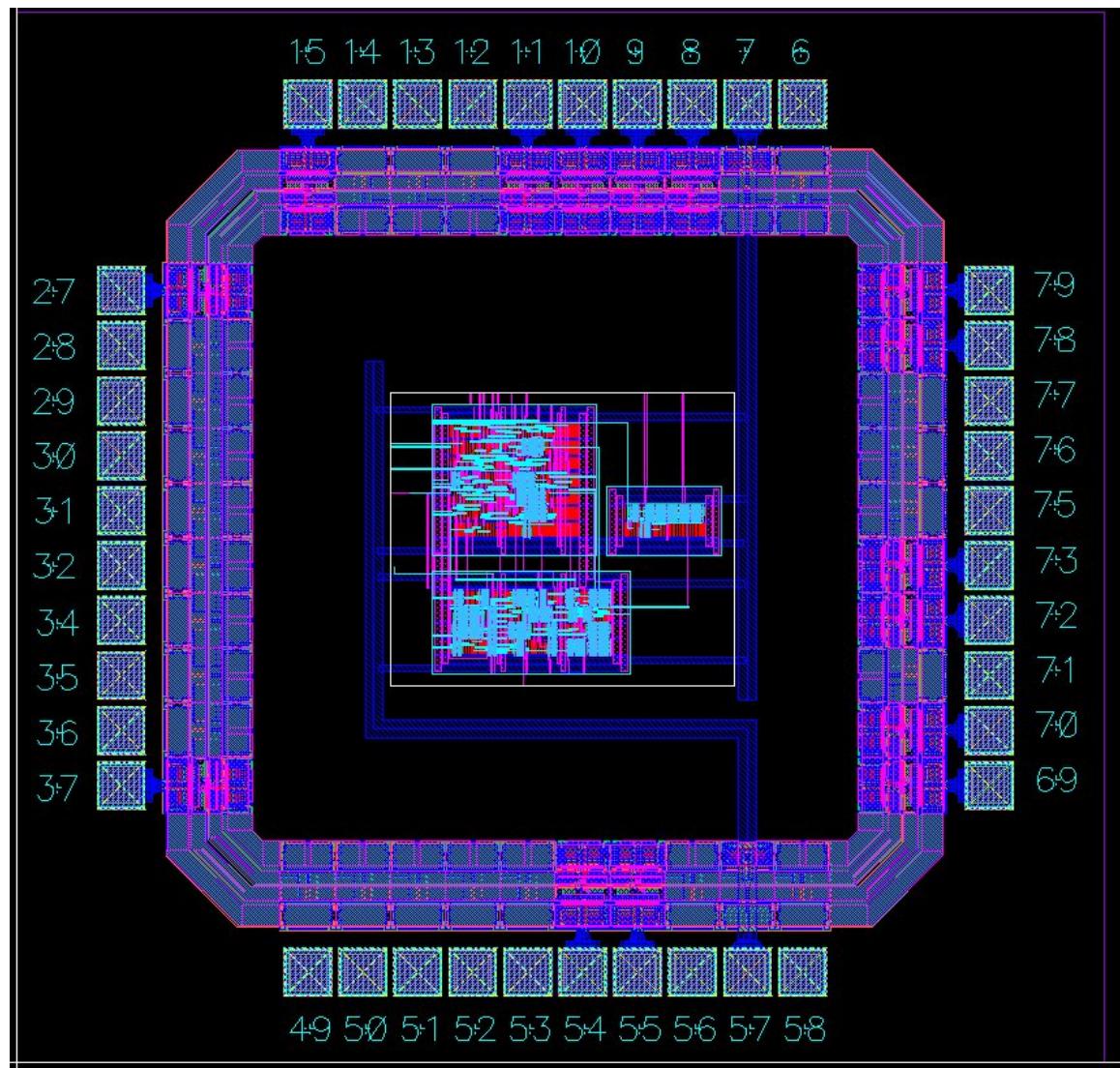


Figure 11.20: Frame and core placed in **Virtuoso-XL** and with **vdd** and **gnd** routing completed

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 piece 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.

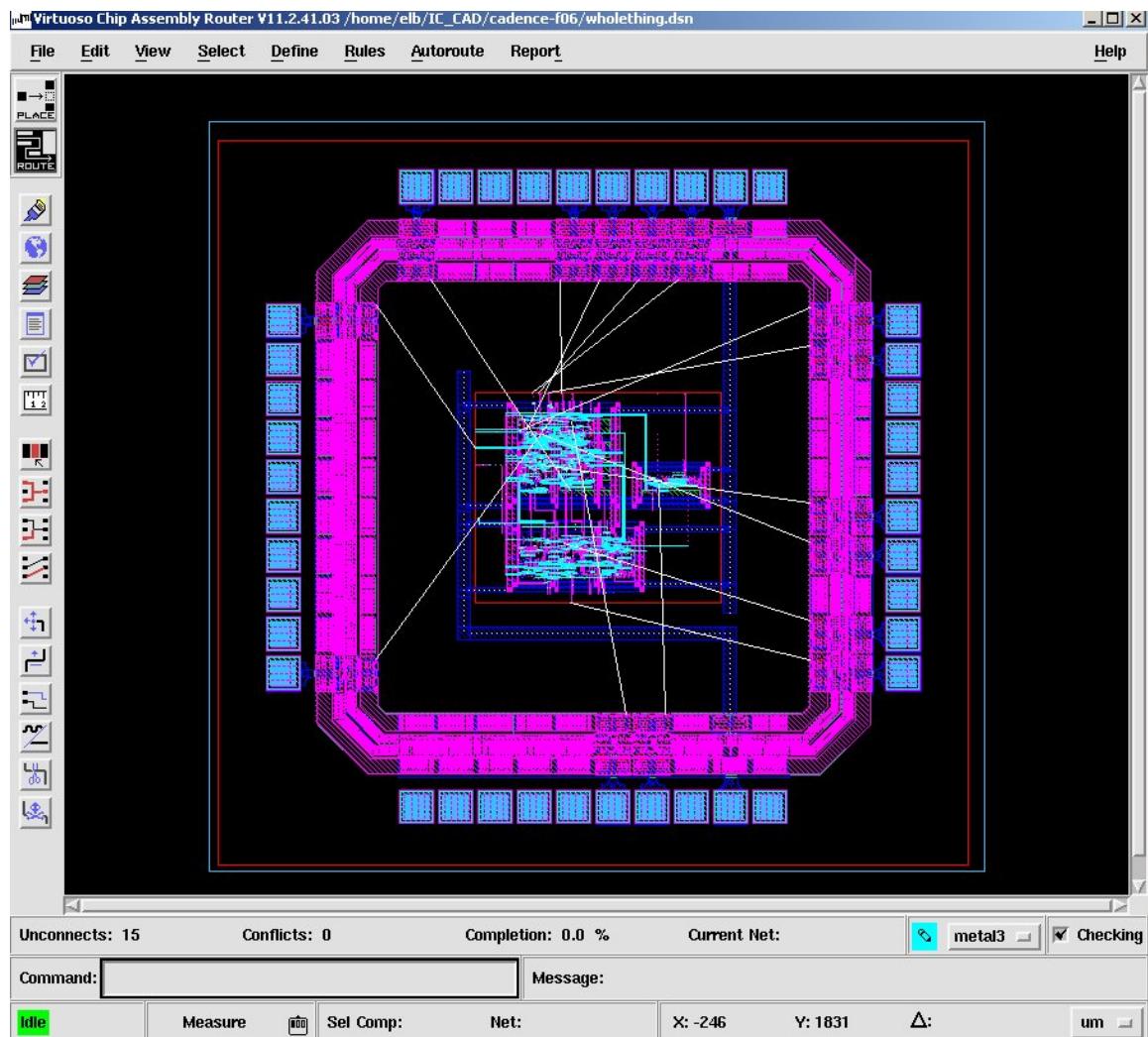


Figure 11.21: Frame and core before routing in ccar

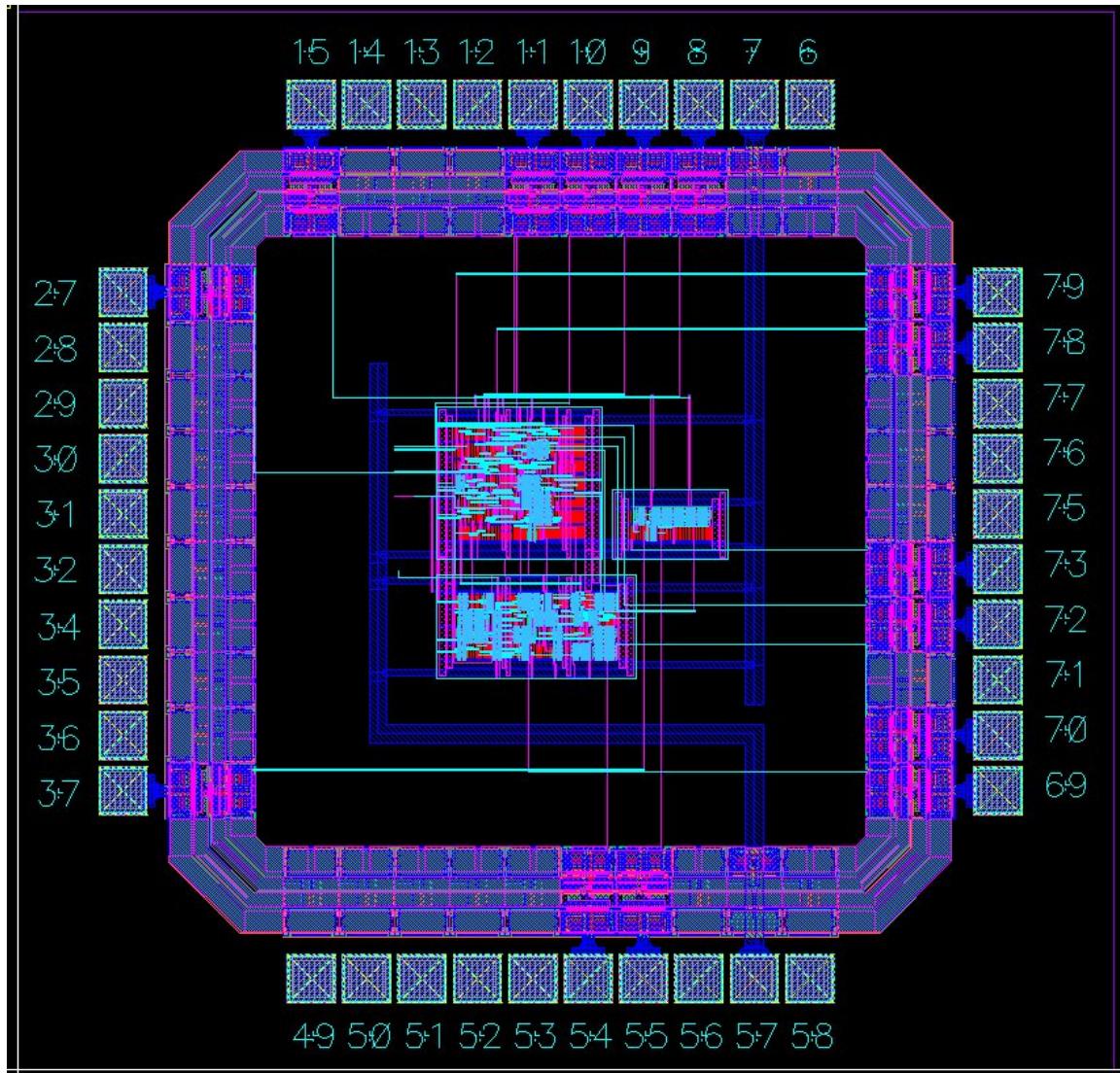


Figure 11.22: Frame and core after routing in **Virtuoso**

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*

