Automatic Placement and Routing using Cadence Encounter

6.375 Tutorial 5
March 16, 2006

In this tutorial you will gain experience using Cadence Encounter to perform automatic placement and routing. A place+route tool takes a gate-level netlist as input and first determines how each gate should be placed on the chip. It uses several heuristic algorithms to group related gates together and thus hopefully minimize routing congestion and wire delay. Place+route tools will focus their effort on minimizing the delay through the critical path. To this end, these tools can resize gates, insert new buffers, and even perform local resynthesis. Place+route tools often have additional algorithms to help reduce area for non-critical paths. After placement, the place+route tool will attempt to route the design while minimizing wire delay. Place+route tools often include additional facilities for clock tree synthesis, power routing, and block level floorplanning. Figure 1 shows how Encounter fits into the 6.375 toolflow.

The following documentation is located in the course locker (`/mit/6.375/doc`) and provides additional information about Encounter and the Tower 0.18 μm Standard Cell Library.

- `tsl-180nm-sc-databook.pdf` - Databook for Tower 0.18 μm Standard Cell Library
- `encounter-user-guide.pdf` - Encounter user guide
- `encounter-command-line-ref.pdf` - Encounter text command reference
- `encounter-menu-ref.pdf` - Encounter GUI reference

Getting started

Before using the 6.375 toolflow you must add the course locker and run the course setup script with the following two commands.

```
% add 6.375
% source /mit/6.375/setup.csh
```

For this tutorial we will be using an unpipelined SMIPSv1 processor as our example RTL design. You should create a working directory and checkout the SMIPSv1 example project from the course CVS repository using the following commands.

```
% mkdir tut5
% cd tut5
% cvs checkout examples/smipsv1-1stage-v
% cd examples/smipsv1-1stage-v
```

Before starting, take a look at the subdirectories in the `smips1-1stage-v` project directory. Figure 2 shows the system diagram which is implemented by the example code. When pushing designs through the physical toolflow we will often refer to the *core*. The core module contains everything which will be on-chip, while blocks outside the core are assume to be off-chip. For this tutorial we are assuming that the processor and a *combinational memory* are located within the core. A combinational memory means that the read address is specified at the beginning of the cycle, and

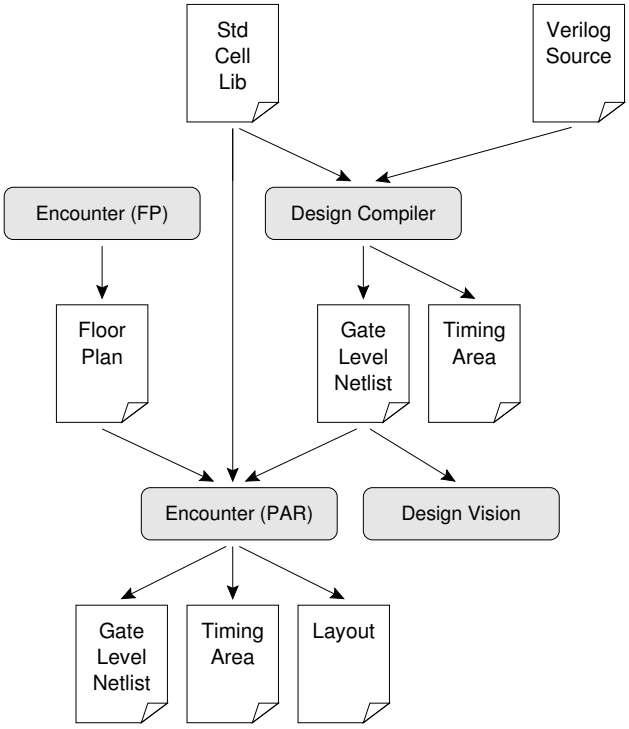

Figure 1: Encounter Toolflow

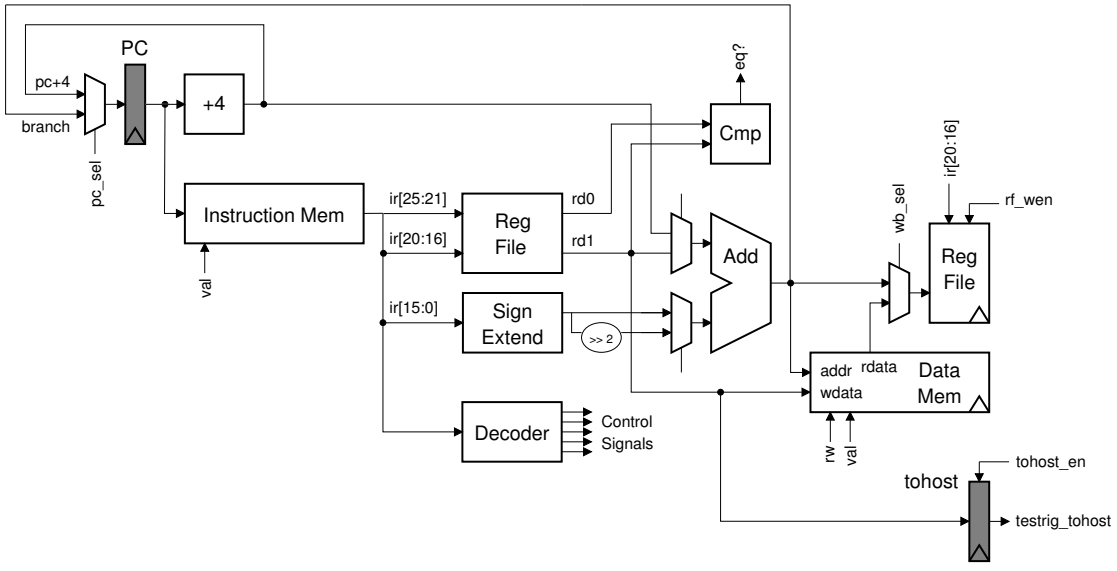

Figure 2: Block diagram for Unpipelined SMIPSv1 Processor

the read data returns during the same cycle. Building large combinational memories is relatively inefficient. It is much more common to use *synchronous memories*. A synchronous memory means that the read address is specified at the end of a cycle, and the read data returns during the next cycle. From Figure 2 it should be clear that the unpipelined SMIPSv1 processor requires combinational memories (or else it would turn into a four stage pipeline). For this tutorial we will not be using a real combinational memory, but instead we will use a dummy memory to emulate the combinational delay through the memory. Examine the source code in `src` and compare `smipsCore_rtl` with `smipsCore_synth`. The `smipsCore_rtl` module is used for simulating the RTL of the SMIPSv1 processor and it includes a functional model for a large on-chip combinational memory. The `smipsCore_synth` module is used for synthesizing the SMIPSv1 processor and it uses a dummy memory. The dummy memory combinationally connects the memory request bus to the memory response bus with a series of standard-cell buffers. Obviously, this is not functionally correct, but it will help us illustrate more reasonable critical paths in the design. In later tutorials, we will start using memory generators which will create synchronous on-chip SRAMs.

Now examine the `build` directory. This directory will contain all generated content including simulators, synthesized gate-level Verilog, and final layout. In this course we will always try to keep generated content separate from our source RTL. This keeps our project directories well organized, and helps prevent us from unintentionally modifying our source RTL. There are subdirectories in the `build` directory for each major step in the 6.375 toolflow. These subdirectories contain scripts and configuration files for running the tools required for that step in the toolflow. For this tutorial we will work in the `enc-par` directory for place+route and in the `enc-fp` directory for floorplanning.

Since Encounter takes a gate-level netlist as input, we need to run Synopsys Design Compiler to synthesize this netlist from the RTL. The following commands will run Design Compiler. Consult *Tutorial 4: RTL-to-Gates Synthesis using Synopsys Design Compiler* for more information.

```
% pwd
tut5/examples/smipsv1-1stage-v
% cd build/dc-synth
% make
```

Automatically Placing and Routing the Processor

We will begin by running several Encounter commands manually before learning how we can automate the tools with scripts. Encounter can generate a large number of output files, so we will be running Encounter within a build directory beneath `enc-par`. Before actually using Encounter to perform place+route, we need to *uniquify* our netlist. A unique netlist is one in which the module hierarchy is a true tree; in other words every module is instantiated once and only once. Use the following commands to create a build directory and to uniquify the synthesized netlist.

```
% pwd
tut5/examples/smipsv1-1stage-v/build
% cd enc-par
% mkdir build
% cd build
% uniquifyNetlist -top smipsCore_synth synthesized_unique.v \
                  ../../dc-synth/current/synthesized.v
```

When this is finished the uniquified netlist is called `synthesized_unique.v`, and it will be in your Encounter build directory. We can now start the Encounter GUI. Later we will see how to run encounter without the GUI for scripting purposes. The following command starts Encounter and leaves you at the Encounter command prompt. We can use `man <command>` at the Encounter command prompt to find out more information about any command. Our first step is to import our synthesized design into Encounter. Use the *Design > Design Import* menu option to display the *Design Import* dialog box. Fill in the following fields of the dialog box.

Field Name	**Value**
Verilog Files	`synthesized_unique.v`
Top Cell	`smipsCore_synth`
LEF Files	`/mit/6.375/libs/tsl180/tsl18fs120/lef/tsl18_6lm.lef` `/mit/6.375/libs/tsl180/tsl18fs120/lef/tsl18fs120.lef`
Max Timing Libraries	`/mit/6.375/libs/tsl180/tsl18fs120/lib/tsl18fs120_max.lib`
Min Timing Libraries	`/mit/6.375/libs/tsl180/tsl18fs120/lib/tsl18fs120_min.lib`
Common Timing Libraries	`/mit/6.375/libs/tsl180/tsl18fs120/lib/tsl18fs120_typ.lib`
Buffer Name/Footprint	`buffd1`
Delay Name/Footprint	`dl01d1`
Inverter Name/Footprint	`inv0d1`
Generate Footprint	This should be checked

The LEF files contain physical information about the standard cell library and the metal layers. This information includes capacitances, resistances, area, and the physical location of pins for each cell. The LIB files contain timing information about each cell; they are similar to the DB files used by Design Compiler. We must also specify various *footprints*. A footprint is a class of cells which are functionally interchangeable. Encounter needs to know which cells in the library it can use for buffer insertion.

We also need to specify a constraint file. As with Design Compiler, the constraint file specifies various input/output constraints on our design such as the target clock period, the drive strength of inputs, and the load capacitance on outputs. Encounter understands the same constraints we used for synthesis, so we can just point it to the `synth.sdc`. Go to the *Timing* tab of the *Design Import* dialog box and enter `../../dc-synth/current/synth.sdc` into the *Timing Constraint File* field.

After you have filled everything into the *Design Import* dialog box, click OK. You should see some output scroll by at the Encounter command prompt. Take a look at the Encounter GUI. Figure 3 shows several key areas of the Encounter GUI. The *Toolbar* contains various buttons; we will mostly use the zoom buttons, the redraw button, and the hierarchy buttons. The *View Panel* allows you to switch between the Floorplan View, the Amoeba View, and the Physical View. We will spend most of our time in the Physical View so change to that view now. You should see many empty rows where the standard cells will be eventually placed. The *Tools Panel* contains various tools for doing manual placement, wiring, etc. We will primarily use the Select Tool, the Move Tool, and the Rule Tool. For now leave the tool set to the Select Tool. The *Color Panel* allows us to show or hide various components in the system (the checkboxes in the V column). We can also decide which components are selectable (the S column). Click on the small color square to change the color of any component. The fifteen displayed components are really just a subset of the possible components; you can click on the *All Colors* button to change the visibility status and/or color of any component. Directly beneath the *All Colors* button are two very thin buttons. We will almost

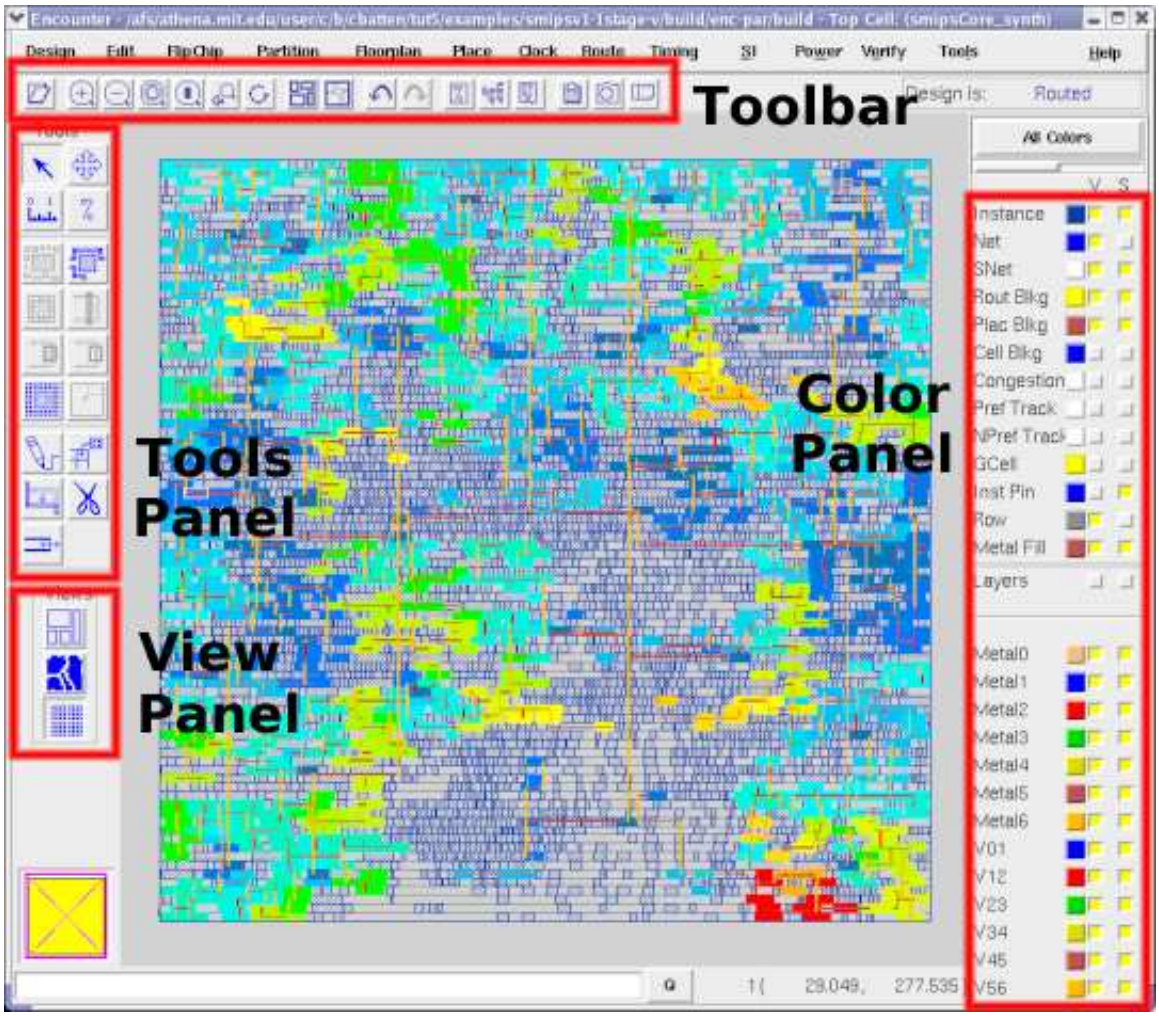

Figure 3: Encounter GUI showing clock skew

always want to choose the rightmost button. This will display many more layers. Try zooming around a bit to get a feel for the Encounter interface. You can zoom out so the whole design fits in the window with the `f` key. Click and drag the right mouse button to zoom in on a specific part of the design. The arrow keys allow you to pan the design.

Let's get started using Encounter to perform automatic placement and routing. The following command will do an initial placement of our design.

```
encounter> amoebaPlace
```

Skim over the output from the `amoebaPlace` command and verify that there are no errors. If Encounter reports any errors, then it was unable to fully place the design. You will need to increase the size of the chip. We will discuss how to do this later in the tutorial. After running `amoebaPlace`, refresh the GUI using `CTRL-R` so you can see the placement. Run `amoebaPlace` a couple of times. Since the tool uses various heuristics, it does not always result in the same placement. Notice that there are various holes in the placement. We can add *filler* cells later to

fill up these empty spaces. Filler cells are just empty standard cells which connect the power and ground rails.

After this initial placement, we can use the `optDesign` command to optimize our design. This command will rearrange cells, insert buffers, and even perform resynthesis as it tries to optimize timing and area. This is a very powerful command with many options. See the Encounter documentation for more information.

```
encounter> optDesign -preCTS
```

After the `optDesign` command is finished, refresh the Encounter GUI. You will see that Encounter has added many wires on the metal layers. These *trial routes* are not real routes since they are incomplete and may violate various process design rules. The trial route helps the `optDesign` command optimize placement. Now that we have finished our automatic placement, we will route the most important net in our design: the clock. Use the following commands to synthesize a clock tree. The tool will add clock buffers and route the clock in attempt to minimize skew between the various state elements.

```
encounter> createClockTreeSpec -bufFootprint {inv0d1} -invFootprint {buffd1} \
                               -output par.clk -routeClkNet
encounter> specifyClockTree -clkfile par.clk
encounter> ckSynthesis
```

Refresh the GUI to see the routed clock tree. To graphically display the clock skew, use the following command. Figure 3 shows an example. Colors at the red end of the spectrum indicate the greatest skew, while colors at the blue end of the spectrum indicate the least skew.

```
encounter> displayClockPhaseDelay
```

You can use the `clearClockDisplay` command to clear the skew coloring. We are now ready to perform the final routing of our design. The following command will attempt to route all the cells while minimizing the delay of the critical path.

```
encounter> globalDetailRoute
```

After the routing is finished, look over the final lines of output. The tool reports the number of warnings and failures. If there are any failures, then Encounter was unable to route your design. You will need to increase the size of the chip. We will discuss how to do this later in the tutorial.

We can now use the Encounter GUI to examine our final layout. Try hiding some of the metal layers by deselecting them in the `Color Panel` (use the V column and don't forget to refresh with `CTRL-R`). Notice that each metal layer is only used to route perpendicular to the layers below and above it. For example, metal 3 routes horizontally while metal 2 and metal 4 route vertically. Figure 4 shows a closeup of a few cells in the design.

The following commands use Encounter to perform static timing analysis on the design.

```
encounter> setAnalysisMode -setup -async -skew -clockTree
encounter> buildTimingGraph
encounter> reportSlacks -setup -outfile postroute_setup_slacks.rpt
```

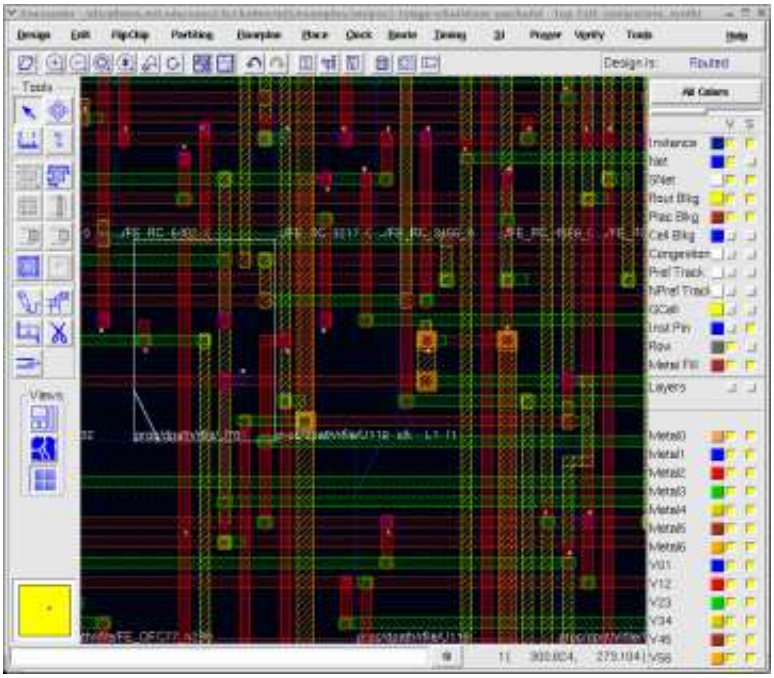

Figure 4: Encounter GUI showing closeup of standard cells with routing

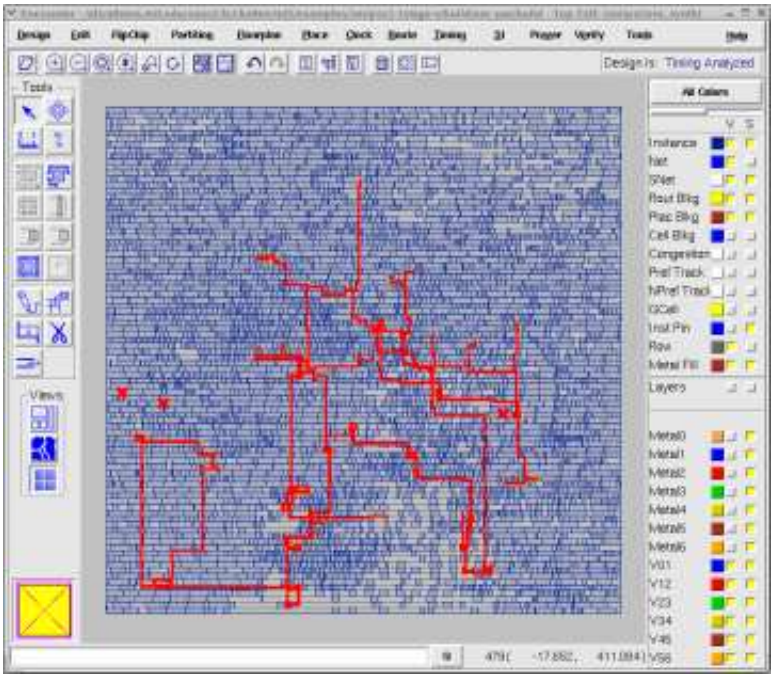

Figure 5: Encounter GUI showing critical path

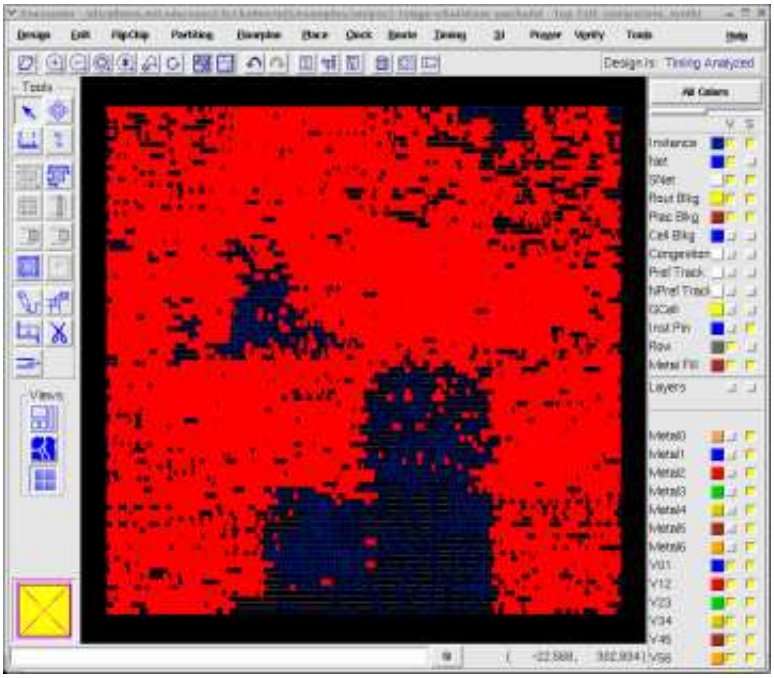

Figure 6: Encounter GUI with the register file highlighted

Now use the *Timing > Timing Debug > Slack Browser* menu option to load the `postroute_setup_slacks.rpt` slack file. This will display the *Timing Slack Browser*. If you double click on a path, Encounter will display the delay of all the cells on the path. The path will also be highlighted graphically in the Encounter GUI. The slacks are ordered starting with the worst path, so the very first path is the critical path in your design. Figure 5 illustrates the critical path in the SMIPSv1 processor. The path starts at the PC and then goes through the instruction memory, through the register file read, through the adder, through the data memory, and into the register file write port.

It is often useful to see a histogram of all the path slacks in your design. You can do this with the *Timing > Timing Debug > End Point Slack Histogram* menu option. Decrease the stepsize and check *Report Non Violating* to see all of the paths in your design. If there are just a few paths with large negative slacks, then you may be able to use a local approach to meeting timing. If there are many paths with large negative slacks then a more global approach is probably needed.

Encounter includes a *Design Browser* which can help you understand how your design has mapped physically onto the chip. Select *Tools > Design Browser* to open the *Design Browser*. Browse through the module hierarchy and find the register file. If you click on a module in the *Design Browser* it will be highlighted in the GUI (see Figure 6). This can help you gain some intuition on how Encounter is doing the placement of your modules. It is also should be quite clear how large the register file is! You can also use the *Design Browser* to highlight specific standard cells and nets. When you are browsing you will probably see some standard cells with names which begin with `FE`. This indicates that these cells were inserted by Encounter. The name also contains information on why they were inserted. `FE_OCPC` means that the cell was added when optimizing the critical path, while `FE_RC` means that the cell was added during local resynthesis. Consult page

599 of the *Encounter User Guide* (`encounter-user-guide.pdf`) for more information. Finally, it is sometimes useful to know the capacitance of a specific net. Select the net in the *Design Browser* and then choose *Tool > Attribute Editor* from the menu to display various parameters about the net.

Entering in these commands by hand can be tedious and error prone, plus doing so makes it difficult to reproduce a result. Thus we will mostly use TCL scripts to control the tool. Even so, using the GUI directly is useful for finding out more information about a specific command or playing with various options. It is also very useful to use the GUI to examine your design after the automatic scripts have finished.

Before continuing, exit Encounter and delete your build directory with the following commands.

```
encounter> exit
% pwd
tut5/examples/smipsv1-1stage-v/build/enc-par/build
% cd ..
% rm -rf build
```

Automating Place+Route with TCL Scripts and Makefiles

In this section we will examine how to use various TCL scripts and makefiles to automate the place+route process. There are four files in the `build/enc-par` directory.

- `Makefile` - Makefile for driving place+route with the TCL scripts
- `par.tcl` - Primary TCL script which contains the Encounter commands
- `par.conf` - Additional configuration information for Encounter
- `par.sdc` - User specified constraints

First take a look at the `par.tcl` script. You will see many familiar commands which we executed manually in the previous section. You will also see some new commands. Take a closer look at the bottom of this TCL script where we write out several text reports. Remember that you can get more information on any command by using `man <command>` at the Encounter prompt. The very first line of the `par.tcl` script loads the `make_generated_vars.tcl` script. This script is generated by the makefile and it contains variables which are defined by the makefile and used by the TCL scripts. Encounter has a constraint file which is very similar to the constraint file used by Design Compiler. If you change a constraint for Design Compiler you will probably want to change it for Encounter as well. Since Design Compiler uses very rough wire load models, you might want to use a slightly smaller clock period constraint for synthesis than what you use for place+route. This will force Design Compiler to work harder and hopefully make it more likely you will meet the target clock frequency.

Now take a look at the `par.conf` script. This is where we set all the parameters which we would interactively set in the `Design Import` dialog box such as the input Verilog, the LEF files, and the toplevel module. Most of these parameters are actually TCL variables which are defined in `make_generated_vars.tcl`.

Now that we are more familiar with the various TCL scripts, we will see how to use the makefile to drive synthesis. Look inside the makefile and identify where the toplevel module is defined. Also notice that the `floorplan` make variable is set. Initially we will not be using floorplanning, so comment out the `floorplan` make variable. The build rules in the makefile will create new build directories, copy the TCL scripts into these build directories, and then run Encounter. Use the following make target to create a new build directory.

```
% pwd
tut5/examples/smipsv1-1stage-v/build/enc-par
% make new-build-dir
```

You should now see a new build directory named `build-<date>` where `<date>` represents the time and date. The `current` symlink always points to the most recent build directory. If you look inside the build directory, you will see the `par.tcl`, `par.conf`, and `par.sdc` scripts but you will also see an additional `make_generated_vars.tcl` script. Various variables inside `make_generated_vars.tcl` are used to specify the search path, which Verilog files to read in, the toplevel Verilog name, etc. After using `make new-build-dir` you can `cd` into the `current` directory, start the Encounter GUI, and run Encounter commands by hand. For example, the following sequence will perform the same steps as in the previous section.

```
% pwd
tut5/examples/smipsv1-1stage-v/build/enc-par
% cd current
% uniquifyNetlist -top smipsCore_synth synthesized_unique.v \
                   ../../dc-synth/current/synthesized.v
% encounter
encounter> source make_generated_vars.tcl
encounter> source par.conf
encounter> commitConfig
encounter> amoebaPlace
encounter> optDesign -preCTS
encounter> createClockTreeSpec -bufFootprint {inv0d1} -invFootprint {buffd1} \
                               -output par.clk -routeClkNet
encounter> specifyClockTree -clkfile par.clk
encounter> ckSynthesis
encounter> globalDetailRoute
encounter> exit
% cd ..
```

The `new-build-dir` make target is useful when you want to conveniently run through some Encounter commands by hand to try them out. To completely automate our synthesis we can use the `par` make target (which is also the default make target). For example, the following commands will automatically place+route the design and save several text reports to the build directory.

```
% pwd
tut5/examples/smipsv1-1stage-v/build/enc-par
% make par
```

You should see Encounter start and then execute the commands located in the `par.tcl` script. Once place+route is finished try running `make par` again. The makefile will detect that nothing

has changed (i.e. the gate-level Verilog source file and Encounter scripts are the same) and so it does nothing. Take a look at the current contents of `enc-par`.

```
% pwd
tut5/examples/smipsv1-1stage-v/build/enc-par
% ls -l
build-2006-02-26_16-15
build-2006-02-26_16-31
current -> build-2006-02-26_16-31
CVS
Makefile
par.conf
par.tcl
par.sdc
```

Notice that the makefile does *not* overwrite build directories. It always creates new build directories. This makes it easy to change your place+route scripts or source Verilog, resynthesize and place+route your design, and compare your results to previous designs. We can use symlinks to keep track of what various build directories correspond to. Every so often you should delete old build directories to save space. The `make clean` command will delete *all* build directories so use it carefully. Sometimes you want to really force the makefile to place+route the design but for some reason it may not work properly. To force a place+route without doing a `make clean` simply remove the `current` symlink. For example, the following commands will force encounter to place+route the design again without actually changing any of the source TCL scripts or Verilog.

```
% pwd
tut5/examples/smipsv1-1stage-v/build/enc-par
% rm -rf current
% make par
```

As we start adding more tools to the toolflow, it is useful to have a toplevel makefile. Take a look at the makefile in `examples/smipsv1-1stage-v/build`. You can use this makefile to run simulations, perform synthesis, and run place+route. The makefile tracks dependencies, so for example, if you run place+route before synthesis the makefile knows to run synthesis first. The following commands will run the assembly tests, run synthesis, and and run place+route.

```
% pwd
tut5/examples/smipsv1-1stage-v/build/enc-par
% cd ..
% make run-asm-tests
% make enc-par
```

Interpreting the Final Layout and Text Reports

The `par.tcl` script saves the entire design three times during place+route: after clock synthesis (`postclksynth`), before routing (`preroute`), and after routing (`postroute`). You can examine each of these designs using the Encounter GUI. For example, to look at the final layout you would move into the current build directory, start the Encounter GUI, and source the `postroute` file. The following commands illustrate this process.

```
% pwd
tut5/examples/smipsv1-1stage-v/build
% cd enc-par/current
% encounter
encounter> source postroute
```

Now you can use all of the techniques discussed earlier in the tutorial to examine the routing, the clock tree, the cell placement, and the critical paths. When you first load the design, press the `f` key to zoom out so that you can see the entire chip. Note that you will need to rerun the `extractRC` command before you can observe any net capacitances, and you will need to rerun timing analysis with the *Timing > Timing Analysis* menu option before using displaying a *End Point Slack Histogram*. You might also want to delete the filler cells in the design to better visualize your chip utilization. You can do this with the following command.

```
encounter> deleteFiller -prefix feedth
```

One key use of the Encounter GUI is to measure the final area of your chip including filler cells. Various text reports will only report the area of the standard cells, but it is important to include the area of filler cells as well. To determine the total area of your chip, simple use the ruler tool and measure the height and width of your chip in microns.

The `par.tcl` script also saves the final gate-level netlist as `par.v` in the build directory. This post-place+route netlist can be different than the post-synthesis netlist because Encounter resizes gates, adds buffers, and does local resynthesis. In addition to the final gate-level netlist, the `par.tcl` script also generates several text reports. Reports usually have the `rpt` filename suffix. The following is a list of the place+route reports.

- `postroute_area.rpt` - Contains area information for each module instance
- `postroute_critpath.rpt` - Contains the combinational critical path of your design
- `postroute_setup_timing.rpt` - Contains paths which violate setup timing
- `postroute_setup_slacks.rpt` - Contains the slack values for many paths in your design
- `postroute_hold_timing.rpt` - Contains paths which violate hold timing
- `postroute_hold_slacks.rpt` - Contains the slack values for many paths in your design
- `postroute_wire.rpt` - Contains information about many of the wires in your design
- `par.cal` - Capacitance values of all nets in the design
- `enc.log` - Log file of all output during Encounter run

There are also `preroute` versions of these files which can be used to quickly look at the progress of your design while the lengthy routing process is still running. The `postroute_area.rpt` report

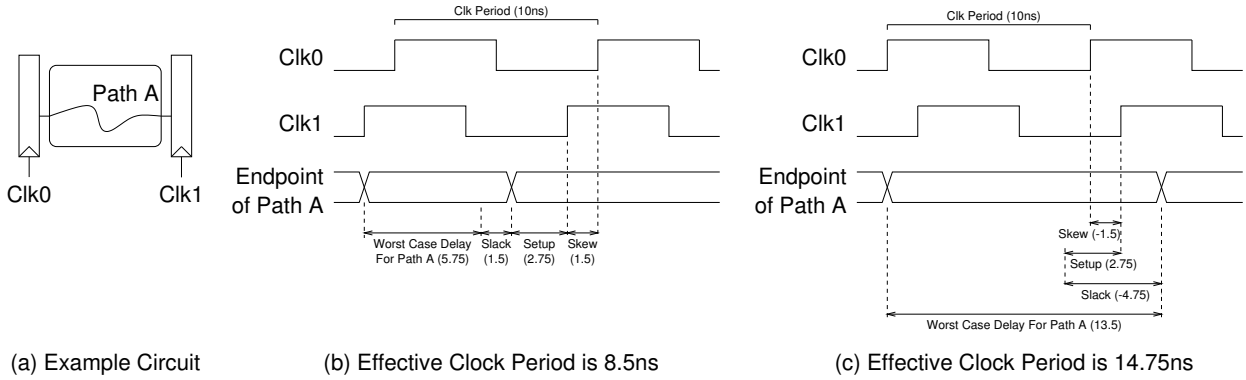

Figure 7: Determining your hardware's effective clock period

shows the area in micron for each module instance in the design. Remember that this ignores filler cells, so to measure the area of your entire chip use the GUI. The `par.cal` and `postroute_wire.rpt` reports can be useful for learning more about the interconnect in your design. The `par.cal` is particularly useful since it breaks down the capacitance for every net in your design into the wire capacitance and the gate capacitance.

Although the `postroute_critpath.rpt` report will show you the longest combinational critical path in your design, the setup timing information in `postroute_setup_timing.rpt` and `postroute_setup_slacks.rpt` are much more important for determining your final effective clock period. Figure 8 shows a fragment from `postroute_setup_timing.rpt`. The paths are sorted such that the first path in the file is the worst case path in your design (and thus sets the effective clock period). It is important to note that the post-place+route critical path can be different than the post-synthesis critical path. Furthermore, the effective clock period after place+route can be much worse than after synthesis. This is because the synthesis tool uses a relatively primitive wireload model, while the Encounter is able to estimate wire delay much more accurately. As shown in Figure 8, the `postroute_setup_timing.rpt` lists the delay of each cell on the critical path, the slew for each cell input, and the capacitive load of each net. Figure 9 shows a fragment from `postroute_setup_slacks.rpt`. This report provides a very brief summary of hundreds of paths in your design. The first column is the start of the path (i.e. the rising clock edge) and the final column is the end of the path. The second column is the constraint; in this case the only constraint is the clock period. The third column shows the slack for that path for both the rising and falling edge. A positive number means that the path made timing and a negative number means that the corresponding path did *not* make timing by the listed amount in nanoseconds.

In Figure 9 you can see that the design did not make timing by 1.595 ns on a path which ends at bit 19 of register 25 in the register file. Even though this design did not make the 5 ns clock period constraint it is still a valid piece of hardware which will operate correctly with some clock period (it is just slower than 5 ns). Similarly, a design which makes the timing constraint but does so with a positive slack can run *faster* than the constrained clock period. We are more concerned about the *effective* clock period of your design as opposed to the clock constraint you set before synthesis. The effective clock period is simply the clock period constraint *minus the worst slack* ($T_{clk} - T_{slack}$). `postroute_setup_timing.rpt` and `postroute_setup_slacks.rpt` are both sorted by slack so that the path with the worst slack is listed first. To determine the effective clock period

```
Path #: 1
Startpoint: proc/dpath/pc_pf/q_np_reg[21]/Q (latency: 2.657)
Endpoint: proc/dpath/rfile/registers_reg[25][19]/D (Setup time: 0.307, latency: 2.660)
Data required time: 5.003 (skew: 0.003 adjusted 1 cycle)
Data arrival time: 6.598
Slack: -1.595 (SETUP VIOLATION)

Object name                              Delta r/f (ns)  Sum r/f (ns)        Slew (ns)     Load
-------------------------------------------------------------------------------------------------
proc/dpath/pc_pf/q_np_reg[21] CP->Q (dfnrq4)    0.466f/0.418r 0.466f/0.418r 0.150f/0.170r 0.045
imemreq_bits_addr[21]                           0.008f/0.007r 0.473f/0.426r
...
imemresp_bits_data[21]                          0.004f/0.004r 1.385f/1.459r
proc/dpath/rfile/U78 I->Z (buffda)              0.218f/0.197r 1.603f/1.656r 0.118f/0.125r 0.260
proc/dpath/rfile/n548                           0.026f/0.027r 1.629f/1.683r
proc/dpath/rfile/FE_RC_0 I->Z (buffda)
...
proc/dpath/rf_rdata0[5]                         0.002f/0.002r 3.107f/3.084r
proc/dpath/op1_mux/FE_RC_1 A2->ZN (nd12d2)      0.056r/0.055f 3.163r/3.139f 0.160f/0.167r 0.010
proc/dpath/op1_mux/FE_RN_3                      0.001r/0.001f 3.164r/3.139f
proc/dpath/op1_mux/FE_RC_4 A2->ZN (nd12d2)      0.072f/0.070r 3.236f/3.210r 0.107r/0.104f 0.020
proc/dpath/op1_mux_out[5]                       0.000f/0.000r 3.236f/3.210r
proc/dpath/adder/add_29/FE_RC_5 I->ZN (inv0d2)
...
proc/dpath/adder/add_29/FE_RC_6 A1->ZN (nd02d2) 0.105f/0.102r 4.847f/4.837r 0.098r/0.096f 0.030
dmemreq_bits_addr[19]                           0.002f/0.002r 4.849f/4.838r
...
dmemresp_bits_data[19]                          0.001f/0.001r 5.933f/6.046r
proc/dpath/wb_mux/FE_RC_7 A2->ZN (nd12d2)       0.050r/0.050f 5.983r/6.096f 0.103f/0.111r 0.010
proc/dpath/wb_mux/FE_RN_8                       0.001r/0.001f 5.984r/6.097f
proc/dpath/wb_mux/FE_RC_9 A2->ZN (nd12d2)       0.064f/0.064r 6.048f/6.160r 0.110r/0.108f 0.017
proc/dpath/rf_wdata[19]                         0.001f/0.001r 6.049f/6.162r
proc/dpath/FE_0_rf_wdata_19_ I->Z (buffda)      0.196f/0.172r 6.245f/6.333r 0.127f/0.131r 0.208
proc/dpath/FE_1_rf_wdata_19_                    0.046f/0.047r 6.291f/6.380r
proc/dpath/rfile/registers_reg[25][19] CP^^D (denrq1)
                                                0.307f/0.158r 6.598f/6.538r 0.203f/0.225r
```

Figure 8: Fragment from `postroute_setup_timing.rpt`

```
# Analysis mode: -setup -skew -caseAnalysis -async -noClkSrcPath
# reportSlacks -setup -outfile postroute_setup_slacks.rpt
# Format: clock  timeReq  slackR/slackF  setupR/setupF  instName/pinName
#
ideal_clock1(R) 5.000 -1.549/-1.595 0.158/0.307 proc/dpath/rfile/registers_reg[25][19]/D
ideal_clock1(R) 5.000 -1.548/-1.594 0.158/0.307 proc/dpath/rfile/registers_reg[27][19]/D
ideal_clock1(R) 5.000 -1.546/-1.592 0.158/0.307 proc/dpath/rfile/registers_reg[18][19]/D
ideal_clock1(R) 5.000 -1.546/-1.591 0.158/0.307 proc/dpath/rfile/registers_reg[23][19]/D
ideal_clock1(R) 5.000 -1.545/-1.591 0.158/0.307 proc/dpath/rfile/registers_reg[21][19]/D
```

Figure 9: Fragment from `postroute_setup_slacks.rpt`

for your design simply choose the worst of the rising and falling edge slacks. Figure 7 illustrates two examples: one with positive slack and one with negative slack. In this example, our clock period constraint is 10 ns. In Figure 7(b), the post-place+route reports indicate a positive slack value of 1.5 ns and thus the effective clock period is 8.5 ns. In Figure 7(c), the post-place+route reports indicate a negative slack value of 4.75 ns and thus the effective clock period is 14.75 ns. Notice that the effective clock period in Figure 7(c) is *not* equal to the worst case combinational critical path (i.e. 13.5 ns). This is because we must also factor in setup time and clock skew. Note that just because the design did not make timing at 5 ns, this does not mean it cannot go faster. If we set the clock period constraint to 4 ns it might result in a design with an effective clock period of 4.2 ns. Lower clock period constraints force the tools to work harder and they may or may not do better. So from Figure 9 we can see that the effective clock period of our unpipelined SMIPSv1 processor is 6.595 ns which is equivalent to a clock frequency of about 150 MHz.

Using Encounter for Floorplanning

In the previous sections we let Encounter handle all of the placement for our design. With larger designs it is critical that we have more input into the *floorplanning* of our design. Floorplanning is the positioning of various large blocks on the chip. Power distribution is often closely coupled with floorplanning, since it is important to carefully manage how power gets to the various macroblocks. In this section we will learn how to use Encounter to perform floorplanning and power routing.

In the previous sections we disabled floorplanning by commenting out the `floorplan` make variable in the place+route makefile. Before continuing, uncomment this variable as follows.

```
floorplan = ../enc-fp/current/floorplan.fp
```

We perform floorplanning in a different build directory than place+route. Move into the floorplanning build directory and take a look at the following three files.

- `Makefile` - Makefile for driving floorplanning with the TCL scripts
- `fp.tcl` - Primary TCL script which contains the Encounter commands
- `fp.conf` - Additional configuration information for Encounter

The floorplanning scripts are setup similar to the synthesis scripts and the place+route scripts. The `fp.tcl` script has three main parts. In the first part we define the dimensions of the chip. The chip includes the core area where the standard cell rows are located as well as the margins around the core for pads and the power ring. We have two options when specifying the dimensions of the chip. The first approach uses the gate-level netlist and a user defined utilization target to estimate the size of the chip. For example, the following Encounter command will create a core which has a square aspect ratio, a target utilization of 70%, and 20 μm margins. The aspect ratio is specified as the height divided by the width.

```
encounter> floorPlan -r 1 0.7 20 20 20 20
```

Essentially, Encounter calculates the total area of the synthesized gate-level netlist, divides this by the target utilization, and uses the desired aspect ratio to determine the chip dimensions. We usually use a relatively low target utilization since Encounter needs extra area to perform buffer insertion, gate resizing, and local netlist resynthesis. This approach is useful since there is no need

to change anything as you add or remove elements from your design; the chip size will grow or shrink accordingly.

The second approach for specifying the dimensions of the chip is to use absolute measurements for the height and width. This gives the designer much more control and enables more precise module floorplanning. For example, the following Encounter command will create a core which has a width of 500 μm, a height of 700 μm, and 20 μm margins.

```
encounter> floorPlan -d 500 700 20 20 20 20
```

Since we are using absolute measurements, these need to be updated whenever we make significant changes to the design.

After executing the `floorPlan` command, the script positions two modules on the chip using the `setObjFPlanBox` command. The TCL code positions the register file in the lower portion of the chip, and positions the dummy memory in the upper right hand corner. Notice that we use several TCL variables to create a very flexible relative floorplan; we can change the chip size and the modules will be automatically repositioned. A module floorplan box is really just a suggestion to help Encounter produce better placements. Encounter is free to place some cells which are in the module outsize of the floorplan box, or place some cells which are not in the module inside the floorplan box. We can also use `relativeFPlan` commands to position RAM macroblocks.

Finally, we use the `addRing` and `addStripe` commands to create a power grid on the metal 5 and metal 6 layers. The following commands use the makefile to run the commands in `fp.tcl` with Encounter.

```
% pwd
tut5/examples/smipsv1-1stage-v/build
% cd enc-fp
% make fp
```

The floorplan is contained in the `current/floorplan.fp` file. During place+route, Encounter will read in this file. To view the floorplan, move into the current build directory, start encounter, and source the `floorplan` script.

```
% pwd
tut5/examples/smipsv1-1stage-v/build/enc-fp
% cd current
% encounter
encounter> source floorplan
```

You should be able to see the power ring, metal 5 horizontal stripes, and metal 6 vertical stripes. Figure 10 shows a closeup of the upper left corner of the chip. To see the module floorplanning, choose the floorplan view on the *View Panel*. Although you can adjust the floorplan interactively using the *Move* tool and the hierarchy buttons on the toolbar, it is best to script your floorplans in TCL. Once you have finished floorplanning, you can now rerun place+route. Figure 12 shows the final layout after using floorplanning. The dummy memory is highlighted in red and the register file is highlighted in green.

Review

The following sequence of commands will setup the 6.375 toolflow, checkout the SMIPSv1 processor example, synthesize the design, perform floorplanning, and place+route the gate-level netlist.

```
% add 6.375
% source /mit/6.375/setup.csh
% mkdir tut5
% cd tut5
% cvs checkout examples/smipsv1-1stage-v
% cd examples/smipsv1-1stage-v/build
% make enc-par
```

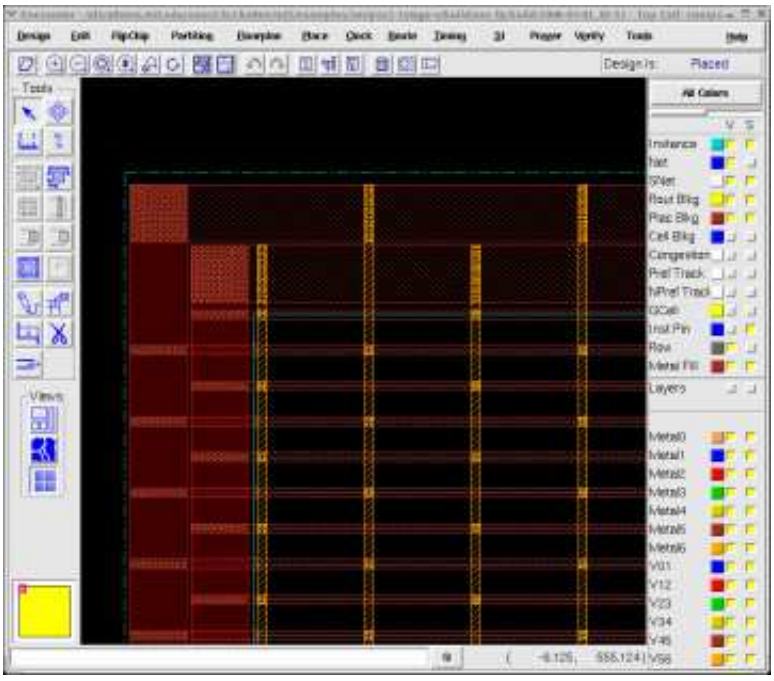

Figure 10: Encounter GUI showing the power grid after floorplanning

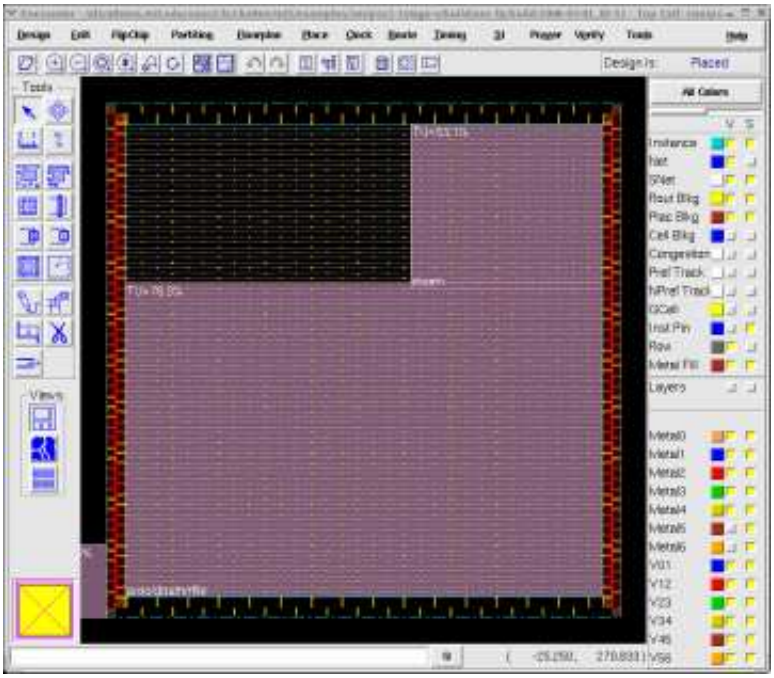

Figure 11: Encounter GUI showing the floorplan

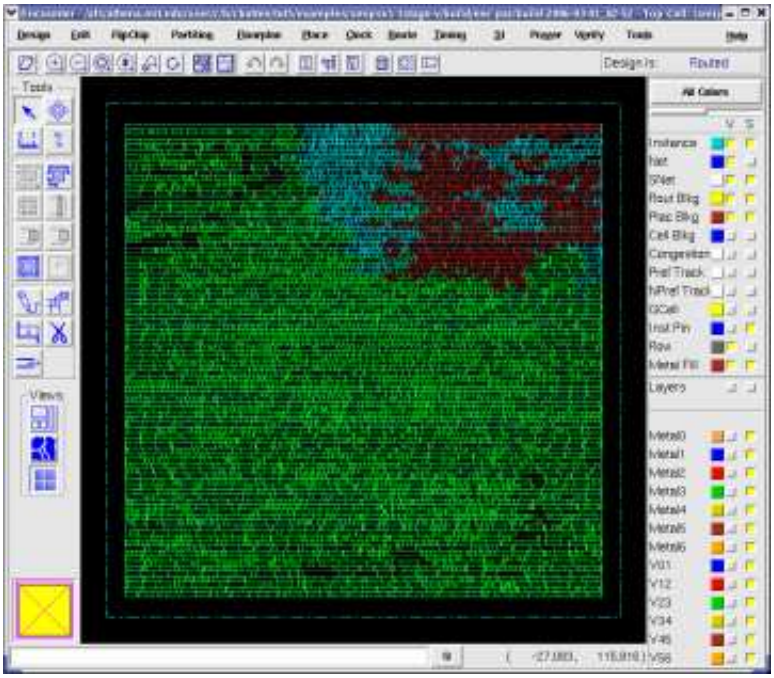

Figure 12: Encounter GUI showing the final placement with floorplanning