

CPE/EE 427, CPE 527, VLSI Design I: Tutorial #4, Standard cell design flow (from verilog to layout, 8-bit accumulator)

Joel Wilder, Aleksandar Milenkovic, ECE Dept., The University of Alabama in Huntsville

Adapted Illinois Institute of Technology, Dept. of Electrical and Computer Engineering
Author: Johannes Grad and James E. Stine

1. INTRODUCTION

In this tutorial exercise you will create an 8-bit accumulator from a verilog file. You will perform RTL simulation on this design. Next, you will synthesize this design to provide a gate-level netlist, and you will simulate this post-synthesized design. Subsequently, you will use Encounter to perform an automatic place-and-route of your 8-bit accumulator. Next, you will perform verifications on this design by importing it into the Cadence icfb tool as a schematic and a routed design. After ensuring that your core design is correct, you will add a padframe to your core design and then repeat the workflow as given for the core: synthesize the design, simulate, place-and-route, and verification steps. This tutorial repeats the workflow from Tutorial 3, but for a verilog design process (instead of schematic capture), straight through to creating an ASIC chip ready for fabrication.

You will base your design on the 0.5um AMI nwell process ($\lambda = 0.30\mu\text{m}$).

2. PREPARE THE CADENCE TOOLS

From your home directory, change directories into your cadence working directory:

```
% cd cadence
```

Make a directory for lab4 and change into that directory:

```
% mkdir lab4
```

```
% cd lab4
```

3. RTL SIMULATION

Typically you enter code in Verilog on the Register-Transfer level (RTL), that is, you model your design using clocked registers, datapath elements and control elements. Download the following files into your lab4 directory:

- `accu_nopads.v` - Verilog RTL code for an 8-bit accumulator, without pads
- `accu_pads.v` - Verilog RTL code for an 8-bit accumulator, with pads
- `accu_test.v` - Verilog testbench for `accu_nopads.v` and `accu_pads.v`

You're going to work with the core design first (no pads).

In order to simulate the Verilog code for this design, use this command at the Unix prompt (\$):

```
$verilog accu_nopads.v accu_test.v
```

See Figure 1 for the output of this command.

```

[sr4 9] ~/cadence/lab4 > verilog accu.v accu_test.v
Tool:  VERILOG-XL      05.40.002-p   Sep 21, 2005  11:07:03

Copyright (c) 1995-2004 Cadence Design Systems, Inc.  All Rights Reserved.
Unpublished -- rights reserved under the copyright laws of the United States.

Copyright (c) 1995-2004 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 "accu.v"
Compiling source file "accu_test.v"
Highest level modules:
stimulus

At Time:          5  Accumulator Output=  x
At Time:         10  Accumulator Output=  0
At Time:         15  Accumulator Output=  0
At Time:         20  Accumulator Output=  1
At Time:         25  Accumulator Output=  1
At Time:         30  Accumulator Output=  2
At Time:         35  Accumulator Output=  2
At Time:         40  Accumulator Output=  3
At Time:         45  Accumulator Output=  3
L21 "accu_test.v": $finish at simulation time 50
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.1 secs in simulation
End of Tool:  VERILOG-XL      05.40.002-p   Sep 21, 2005  11:07:04

```

Figure 1. RTL Verilog simulation output text.

This testbench provides results directly on the screen and also in a waveform database. From the screen you can see that the design behaves as expected. That is, every 10ns a “1” is added to the accumulator output. This is expected since in the testbench a clock of 10ns is specified and the input “in” is connected to a constant “1”.

Now, you will use the program Cadence Simvision to look at the waveform database that was created by the verilog simulation. Type the following command at the unix prompt (this tool was used in Tutorial 3):

\$ simvision&

(The “&” symbol tells the operating system to run the Simvision program in the background – relative to that terminal window.) In the Simvision Design Browser window, open the Waveform database by clicking on the “Open” symbol. Then double-click on “shm.db”, which is the folder where the file is located.

Inside the folder is only one file, shm.trn. Double-click on the file to open it. To see the contents of the waveform database, click on “stimulus” in the scope tree (as shown in Figure 2).

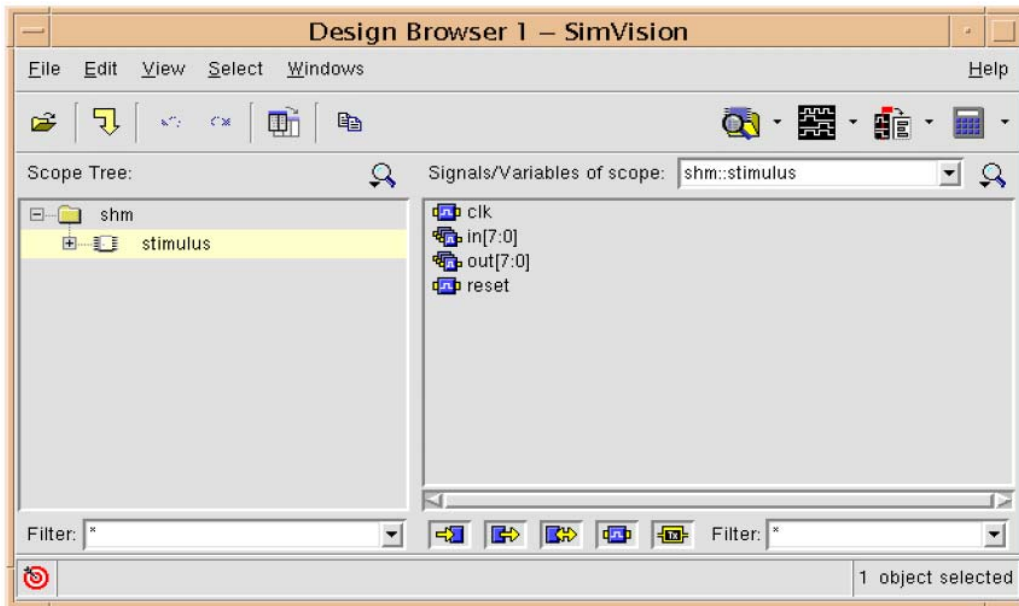


Figure 2. Design Browser window.

Now, plot the waveforms as you did in Tutorial 3 (select all signals and send them to the waveform browser). You will see your output similar to Figure 3 (and similar to the accumulator you constructed in Tutorial 3 via schematic capture!). Verify that the accumulator is operating as expected.

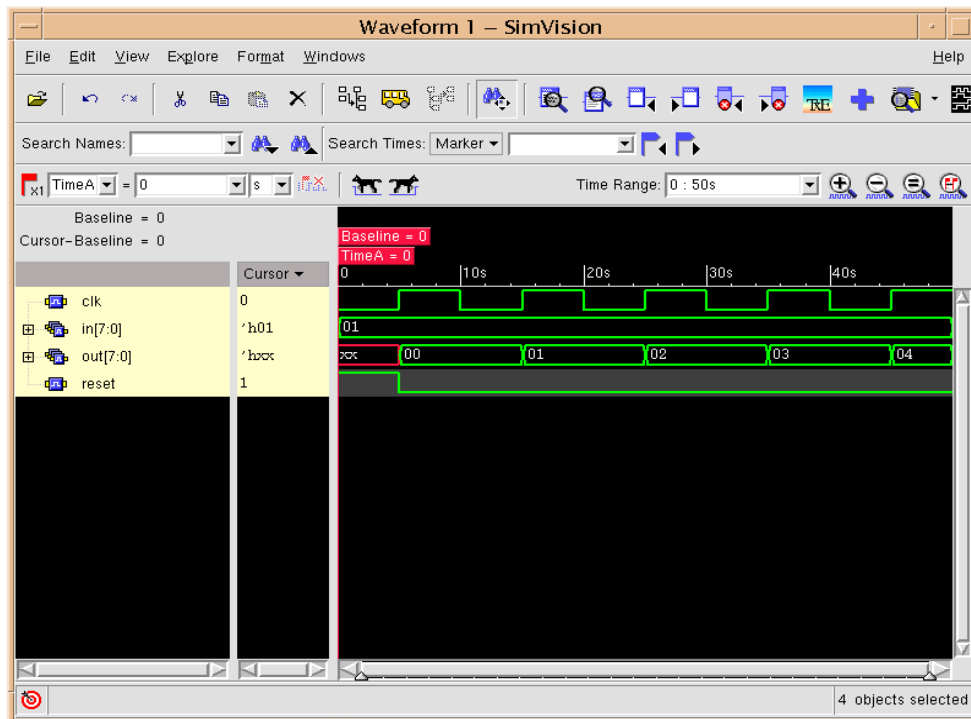


Figure 3. Simulation Results.

4. LOGIC SYNTHESIS

Once you have verified that your Verilog RTL code is working correctly you can synthesize it into standard cells. The result will be a gate-level netlist that only contains interconnected standard cells. (In Tutorial 3 you created the gate-level netlist through schematic capture – you specified how the gates would be connected. Here, the RTL code specifies the functionality of the circuit, and then you use logic synthesis in order to transfer that functionality into a gate-level netlist. Using this Verilog workflow, you will observe that there are two levels of circuit optimization, meaning that you specify a target frequency to the logic synthesizer, and it works to achieve that target frequency by generating a gate-level netlist and then checking slack time to see if the data signal will arrive before the rising edge of the clock. The second level of optimization is through Encounter, which you dealt with in Tutorial 3 through the .sdc file, where you set the timing constraint for the clock and Encounter routes the design in order to achieve that frequency. The .sdc file is generated automatically through the logic synthesizer.)

There are template files for all the following steps already prepared for you. You will now copy those templates into your project. To keep things organized you will run synthesis in a separate folder. That way it will be separate from the original RTL code. Use the mkdir command to create a folder. Then copy the templates, as shown below:

```
$ mkdir encounter
$ cd encounter
$ cp /apps/iit_lib/osu/osu_stdcells/flow/ami05/* .
$ cp /home/grad/wilderj/public_html/synopsys/* . (retrieves a script file and a library file)
```

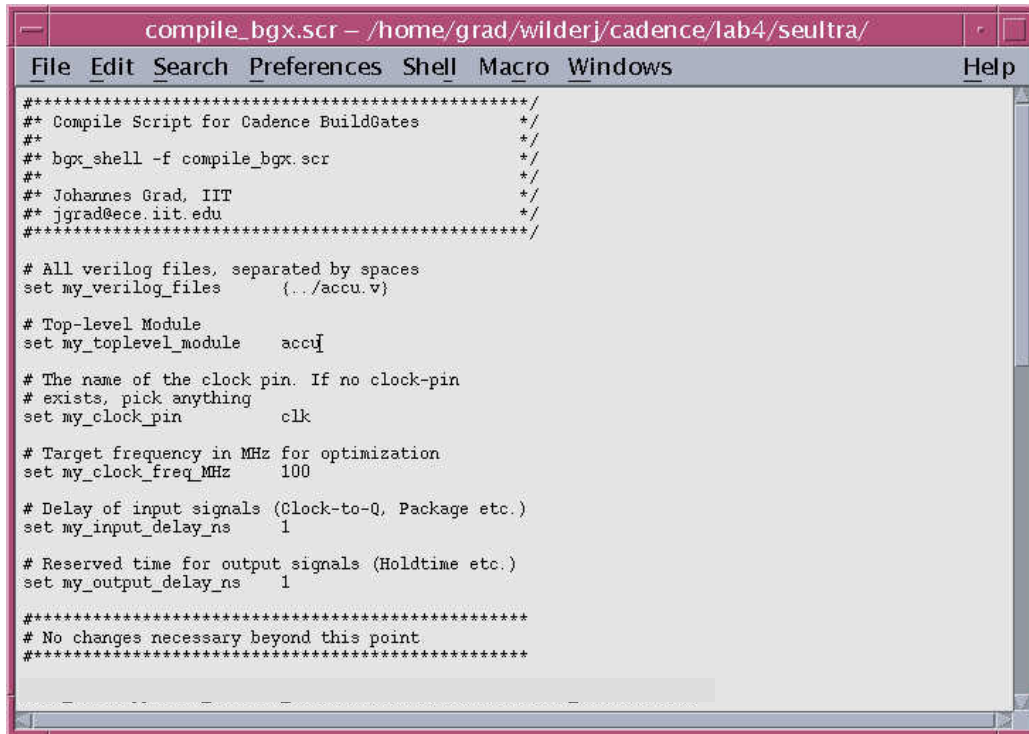
Now you have created the “encounter” folder and filled it with the template files for the AMI 0.5um technology. (identical to what was done in Tutorial 3)

You will use the tool Synopsys Design Compiler for logic synthesis. The script file you’ll use with Synopsys is called “compile_dc.tcl”. This file is retrieved as shown above. Now you will open “compile_dc.tcl” in a text editor and modify it according to your accumulator design. This file is a script file for Synopsys and will be executed line by line. To make it easier to modify, all key values are defined in the beginning of the file.

So the only modifications will be done in the header of the file. Specifically, you need to change the following four values:

my_verilog_files	../accu.v	The RTL input file that we want to synthesize.
my_toplevel_module	accu	The name of the top-level module in the RTL code.
my_clock_pin	clk	The name of the clock pin in the RTL code.
my_clock_freq_MHz	100	Tells PKS to optimize the circuit so that it is capable to operate at least at 100 MHz.

Later, once you have more experience, you can take a look at “compile_dc.tcl” and look at the different commands that are used to read in RTL code, optimize it and output gate-level code. But for now it is enough to just plug in your desired values in the header of the file. You can use any text editor (xemacs, nedit, etc.), with your modified file similar to the one shown in Figure 4.



```

compile_bgx.scr - /home/grad/wilderj/cadence/lab4/seultra/
File Edit Search Preferences Shell Macro Windows Help
#*****/
## Compile Script for Cadence BuildGates */
## */
## bgx_shell -f compile_bgx.scr */
## */
## Johannes Grad, IIT */
## jgrad@ece.iit.edu */
#*****/

# All verilog files, separated by spaces
set my_verilog_files {../accu.v}

# Top-level Module
set my_toplevel_module accu

# The name of the clock pin. If no clock-pin
# exists, pick anything
set my_clock_pin clk

# Target frequency in MHz for optimization
set my_clock_freq_MHz 100

# Delay of input signals (Clock-to-Q, Package etc.)
set my_input_delay_ns 1

# Reserved time for output signals (Holdtime etc.)
set my_output_delay_ns 1

#*****
# No changes necessary beyond this point
#*****

```

Figure 4. Synthesis script file.

Since the Synopsys tool needs some initialization files that are installed in a different directory from your current working directory, you need to temporarily re-assign the OSUcells environment variable. This is done in the following way (at the unix prompt):

```
$ export OSUcells=/apps/cadence2007/osu_soc_v27/synopsys
```

Now everything should be set up to run the Synopsys synthesizer:

```
$ /apps/synopsys/Z-2007.03-SP3/bin/dc_shell-t -f compile_dc.tcl
```

Synopsys will run for a short time. When it is finished, it will return to the command line. Any errors that are generated can most likely be ignored. What you are interested in is the netlist file which Synopsys produces, called accu.vh.

The most important number in digital logic design is the “Slack”, or the safety margin when comparing data arrival time with respect to the clock frequency (as shown in Figure 6).

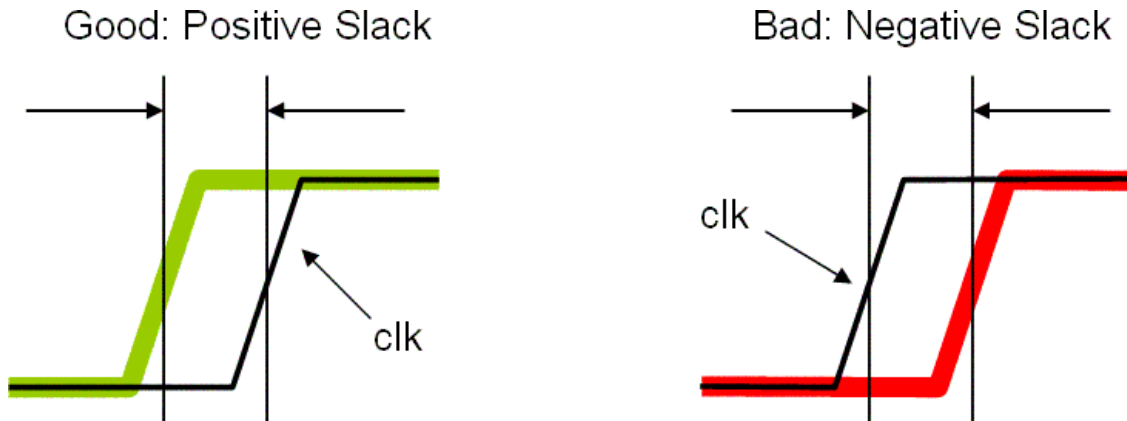


Figure 6. Slack Time.

To get more detail regarding slack and timing, look in the file “timing.rep” that was created by Synopsys. It contains very detailed timing information about your circuit. To see the file use “cat”, which displays text files:

\$ cat timing.rep

The output of a timing file which is similar to that produced by Synopsys is shown in Figure 7.

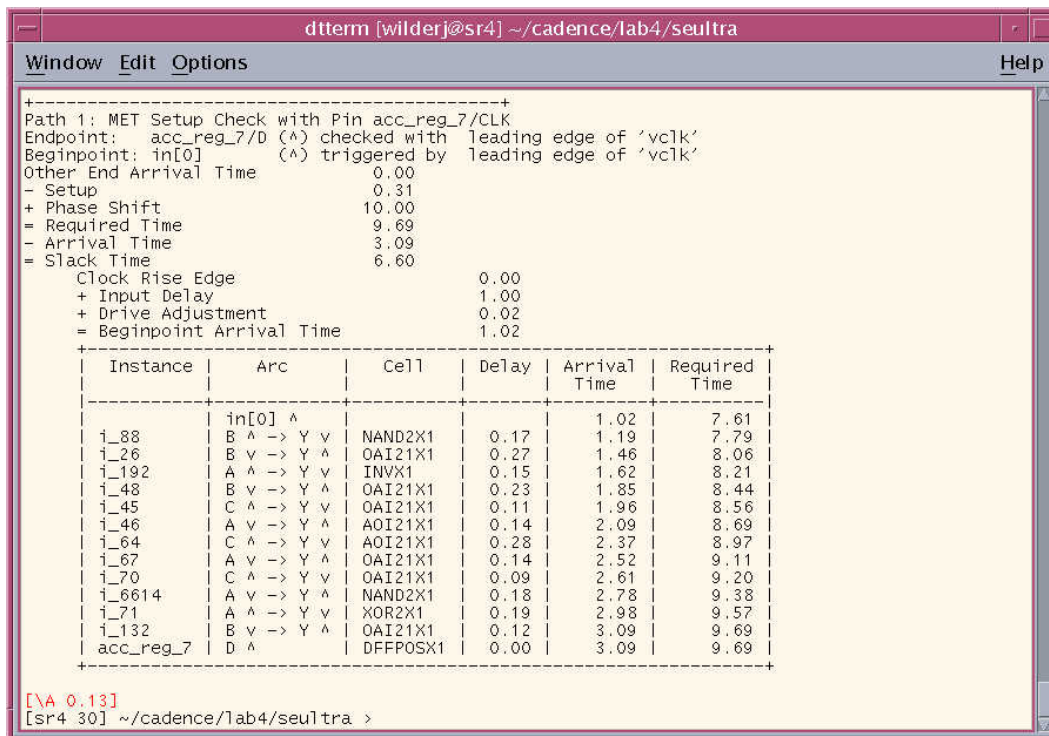


Figure 7. timing.rep output.

The first thing you notice is the “critical path” of your circuit. This is the path of logic that has the longest delay and, therefore, is the upper limit on the clock frequency. The clock frequency cannot be faster than the inverse of the delay of the critical path. When you look closer you see that the critical path starts at the input pin “in[0]” and ends at the “D” pin of a D-Flip-Flop. A

critical path can be anywhere in the circuit: from an input to a register, from a register to the output or between two registers. If the circuit is combinational, that is it has no registers, then the critical path will always be between an input and an output.

The last line of “timing.rep” tells you that the delay of the critical path is 3.09ns. And in order to be able to operate at 100MHz the required time of the critical path has to be below 9.69ns. Hence the “Slack”, is the difference between the two: $9.69ns - 3.09ns = 6.60ns$.

In this case you are well within the limit. The critical path could be up to 6.6ns longer and you would still meet your goal of 100MHz. This means you could compute the theoretical maximum operating frequency of your existing design as only accounting for the worst delay in the circuit (the “critical path”):

$$\frac{1}{3.09ns} = 323.6MHz$$

In this theoretical maximum limit, the slack time is reduced to zero by setting the data arrival time (the critical delay) equal to the arrival time of the rising clock edge, making the new clock frequency 323.6 MHz.

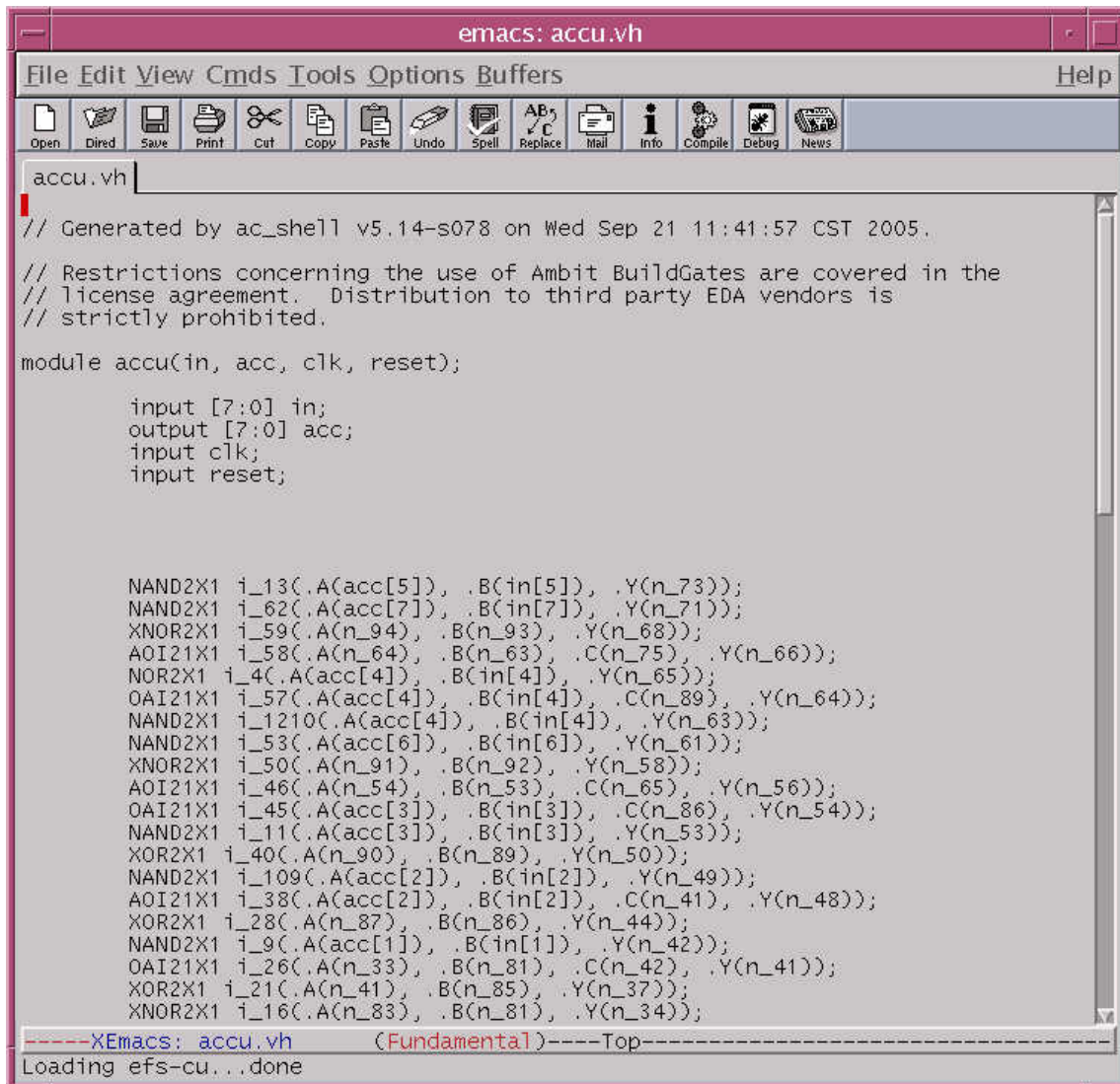
This means if you were to fabricate the current circuit you could operate it at a speed of up to 323.6MHz. But you could also go back to “compile_dc.tcl” and change the target clock frequency from 100MHz to 350MHz. That would force Synopsys to optimize your circuit more than previously. That is, it has to try to shorten the critical path, using Boolean arithmetic, in order to get its delay below 2.86ns, which is the inverse of 350MHz.

Note that these calculations are somewhat simplistic. Since you originally specified 100MHz, you might expect the limit on the critical path to be simply the inverse, or 10ns. In reality you see that the requirement is 9.69ns. For example, the setup time of the D-Flip-Flop was neglected. The signal must reach the input of the Flip-Flop some time before the clock edge, in order to be reliably stored. In addition, some initial delay on the input signal was also assumed, which could be the delay of the IC package, or the delay of the gate that produces the signal on “in[0]”.

If you want to know immediately how fast your circuit is, set the target frequency to an impossible high value, e.g. 3000MHz, which is 3GHz. That will force Synopsys to produce the fastest possible circuit. Then you can see from “timing.rep” what the fastest possible clock frequency is and plug it back into “compile_dc.tcl”. But you have to realize that Synopsys will try very hard to meet your impossible target, resulting in a long run time and a very big circuit with many extra gates.

As a final exercise, you can look at the output of Synopsys. As was said above, it is a gate-level Verilog netlist that only contains interconnected standard cells. The netlist will be called accu.vh, and can be viewed, as shown in Figure 8, by typing:

\$ more accu.vh



```

emacs: accu.vh
File Edit View Cmps Tools Options Buffers Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News
accu.vh
// Generated by ac_shell v5.14-s078 on Wed Sep 21 11:41:57 CST 2005.
// Restrictions concerning the use of Ambit BuildGates are covered in the
// license agreement. Distribution to third party EDA vendors is
// strictly prohibited.
module accu(in, acc, clk, reset);
    input [7:0] in;
    output [7:0] acc;
    input clk;
    input reset;

    NAND2X1 i_13(.A(acc[5]), .B(in[5]), .Y(n_73));
    NAND2X1 i_62(.A(acc[7]), .B(in[7]), .Y(n_71));
    XNOR2X1 i_59(.A(n_94), .B(n_93), .Y(n_68));
    AOI21X1 i_58(.A(n_64), .B(n_63), .C(n_75), .Y(n_66));
    NOR2X1 i_4(.A(acc[4]), .B(in[4]), .Y(n_65));
    OAI21X1 i_57(.A(acc[4]), .B(in[4]), .C(n_89), .Y(n_64));
    NAND2X1 i_1210(.A(acc[4]), .B(in[4]), .Y(n_63));
    NAND2X1 i_53(.A(acc[6]), .B(in[6]), .Y(n_61));
    XNOR2X1 i_50(.A(n_91), .B(n_92), .Y(n_58));
    AOI21X1 i_46(.A(n_54), .B(n_53), .C(n_65), .Y(n_56));
    OAI21X1 i_45(.A(acc[3]), .B(in[3]), .C(n_86), .Y(n_54));
    NAND2X1 i_11(.A(acc[3]), .B(in[3]), .Y(n_53));
    XOR2X1 i_40(.A(n_90), .B(n_89), .Y(n_50));
    NAND2X1 i_109(.A(acc[2]), .B(in[2]), .Y(n_49));
    AOI21X1 i_38(.A(acc[2]), .B(in[2]), .C(n_41), .Y(n_48));
    XOR2X1 i_28(.A(n_87), .B(n_86), .Y(n_44));
    NAND2X1 i_9(.A(acc[1]), .B(in[1]), .Y(n_42));
    OAI21X1 i_26(.A(n_33), .B(n_81), .C(n_42), .Y(n_41));
    XOR2X1 i_21(.A(n_41), .B(n_85), .Y(n_37));
    XNOR2X1 i_16(.A(n_83), .B(n_81), .Y(n_34));
-----XEmacs: accu.vh (Fundamental)-----Top-----
Loading efs-cu...done

```

Figure 8. Synthesized Verilog Output.

Note that the top-level module still has the name “accu” and the names of the inputs and outputs have not changed. From the outside it is exactly the same circuit as you coded on the RTL level. But on the inside all functionality is now expressed only in terms of standard cells. (Note: accu.vh is equivalent to the 8-bit accumulator schematic you did in Tutorial 3. The difference here is that the synthesizer creates the “schematic”, or netlist, for you from the simple accu.v file. You can see how much faster it is to create a few lines of verilog code rather than creating a picture in the schematic view.)

Since the layouts for all standard cells are available through the OSU library, you can now easily place and route them for the final layout. But before you do that you should perform another round of verification to make sure your circuit behavior has been preserved during the transition from the RTL level to the gate level.

Now, before moving on, we must reset the OSUcells environment variable:

```
$ export OSUcells=/apps/iit_lib/osu/osu_stdcells
```

5. POST-SYNTHESIS VERIFICATION

Since the gate-level netlist is also in Verilog format you can use the same Verilog simulation command that was used for RTL level simulation. In addition, since you are now using standard cells, you need to include an extra file "osu05_stdcells.v" that includes definitions for all standard cells. Use the following command:

```
$ verilog osu05_stdcells.v accu.vh ../accu_test.v
```

Note how the original testbench from the RTL level simulation was reused. That is an excellent way to ensure that the gate-level representation matches the RTL level. Since you are now working in the "encounter" folder the "../" notation was used to tell the operating system that "accu_test.v" is located one folder above the current folder. Verify that the post-synthesis verilog simulation results, as shown in Figure 9, are similar to the pre-synthesis simulation results (ignore the osu05_stdcells.v warnings.)

```
At Time:          5 Accumulator Output= x
At Time:          10 Accumulator Output= 0
At Time:          15 Accumulator Output= 0
At Time:          20 Accumulator Output= 1
At Time:          25 Accumulator Output= 1
At Time:          30 Accumulator Output= 2
At Time:          35 Accumulator Output= 2
At Time:          40 Accumulator Output= 3
At Time:          45 Accumulator Output= 3
L21 "../accu_test.v": $finish at simulation time 5000
7 warnings
0 simulation events (use +profile or +listcounts option to count) + 538 accelerated events + 830
timing check events
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool:  VERILOG-XL   05.40.002-p   Sep 22, 2005  11:11:09
```

Figure 9. Post-synthesis simulation results.

6. AUTO PLACE AND ROUTE

At this point you have synthesized your original Verilog design, which was implemented on the RTL level into a gate-level design. You have also verified that the synthesis result is still correct by using the original testbench.

Now you can use Cadence Encounter to place the standard cells and route them.

The result will be the final mask layout that could be shipped to the AMI foundry for fabrication. Cadence Encounter has a very intuitive graphical user interface, however, it is faster and more convenient to execute it with a command file. This file is called "encounter.conf". Open this file for editing by typing:

\$ xemacs encounter.conf&

In the encounter.conf file, edit the following line as shown:

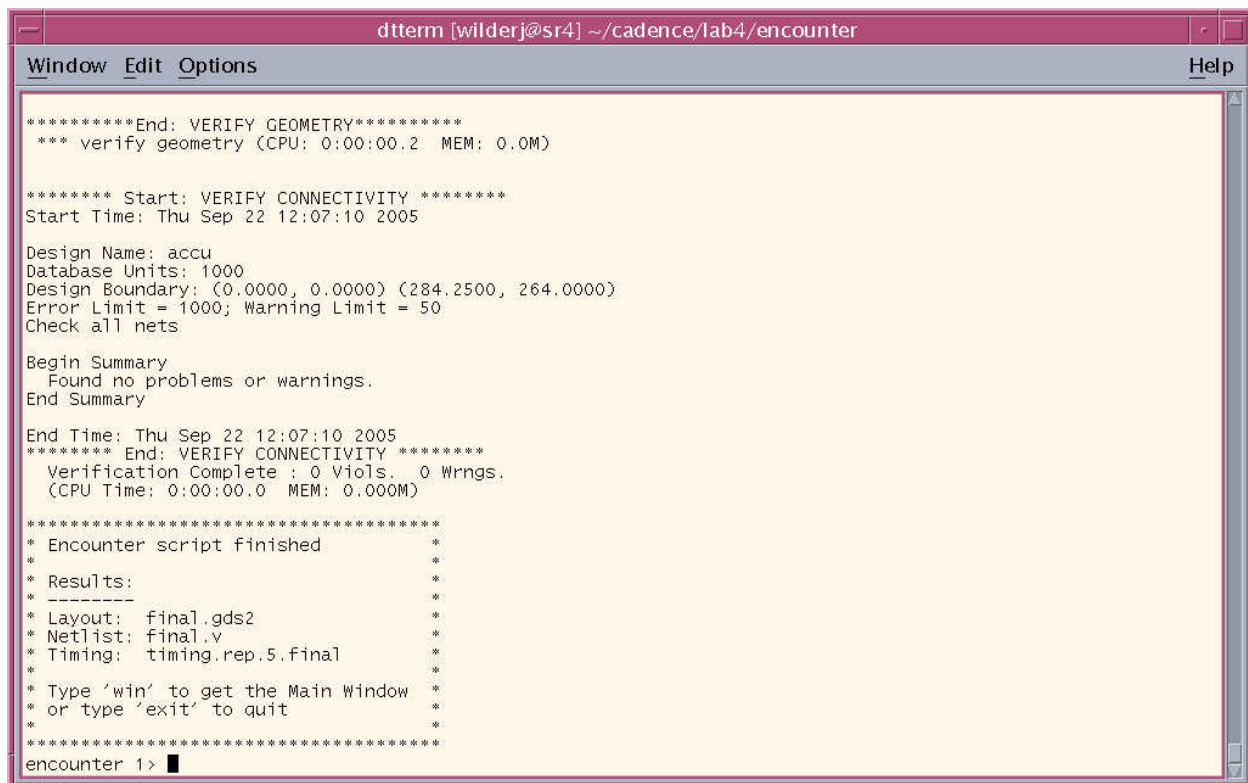
From:	To:
set my_toplevel MY_TOPLEVEL	set my_toplevel accu

Editing this file in this way points the tool to your accu design. Overall, encounter.conf is a command file that is executed line by line. It includes commands to input the gate-level netlist, floorplan the chip, place the cells, route the cells and to verify the layout. In the end, the final layout is written in GDS-II format, which is the most popular format for IC layouts. It can be imported into Cadence Virtuoso and many other layout tools.

You are now ready to perform automatic place and route by typing the following command:

\$ encounter -init encounter.tcl

The results of this execution are shown in Figure 10. (If you have set everything up to this point, this should be a seamless, carefree process.)



```

dtterm [wilderj@sr4] ~/cadence/lab4/encounter
Window Edit Options Help
*****End: VERIFY GEOMETRY*****
*** verify geometry (CPU: 0:00:00.2 MEM: 0.0M)

***** Start: VERIFY CONNECTIVITY *****
Start Time: Thu Sep 22 12:07:10 2005

Design Name: accu
Database Units: 1000
Design Boundary: (0.0000, 0.0000) (284.2500, 264.0000)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Thu Sep 22 12:07:10 2005
***** End: VERIFY CONNECTIVITY *****
Verification Complete : 0 Viols. 0 Wrngs.
(CPU Time: 0:00:00.0 MEM: 0.000M)

*****
* Encounter script finished *
* *
* Results: *
* ----- *
* Layout: final.gds2 *
* Netlist: final.v *
* Timing: timing.rep.5.final *
* *
* Type 'win' to get the Main Window *
* or type 'exit' to quit *
* *
*****
encounter 1> █

```

Figure 10. Auto place and route processing.

At the encounter 1 prompt in the terminal, type 'win' to take a look at the resulting layout. Your layout should look similar to Figure 11.

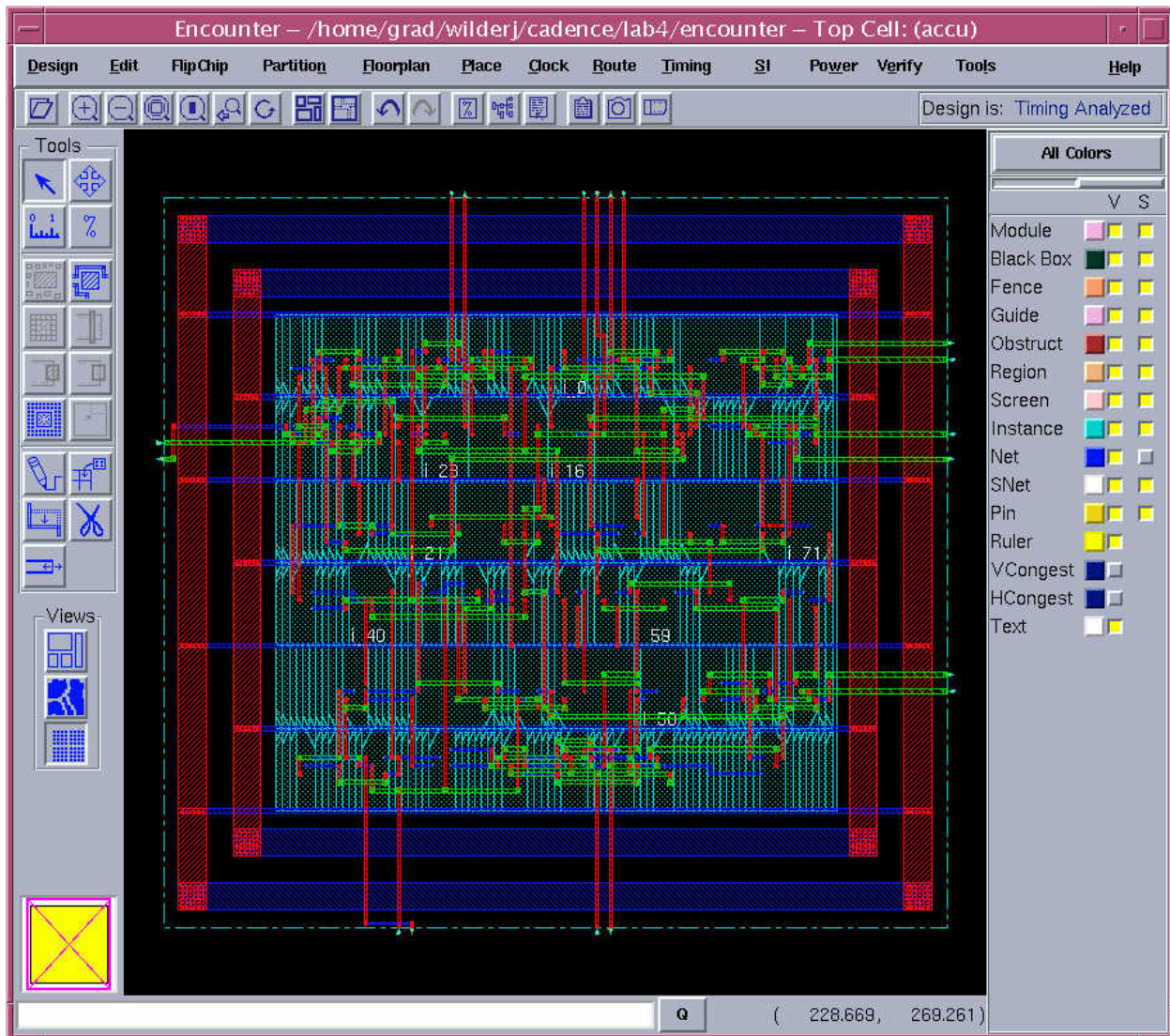


Figure 11. Encounter Layout.

Observe that the layout has been done for you by way of placing standardized cells from a pre-existing library. This is the way to go, since you don't have to do custom layouts from the transistor level! In Encounter, go to **Tools->Summary Report**. **Record the core size and chip size of this design** (you will be asked to hand this in). Note that while Encounter has a GUI interface for doing different step-by-step procedures to perform the auto place and route, most of these things have been done in the encounter.conf and encounter.tcl files. Further, Encounter can be used to analyze your layout via its GUI interface. Spend some time going through the tool menus, and then you can close Encounter and exit the program in the terminal.

You looked at the slack time in the post-synthesized design. You'll do that again for the post-layout design. The post-synthesized design only accounted for delays through the gates themselves, but not through the interconnections. You should expect a slight increase in the

delays looking at the post-layout design due to these interconnection delays. Encounter performs a timing analysis for you, and uses this internally as it's optimizing the layout. The final timing report before actual layout can be found in file **timing.rep.5.final**. Have a look by typing

\$ more timing.rep.5.final

A snapshot of these results is shown in Figure 12. This file contains the 10 worst paths (relative to slack), with the worst offender shown first. (note that the picture shows the 10th worst path)

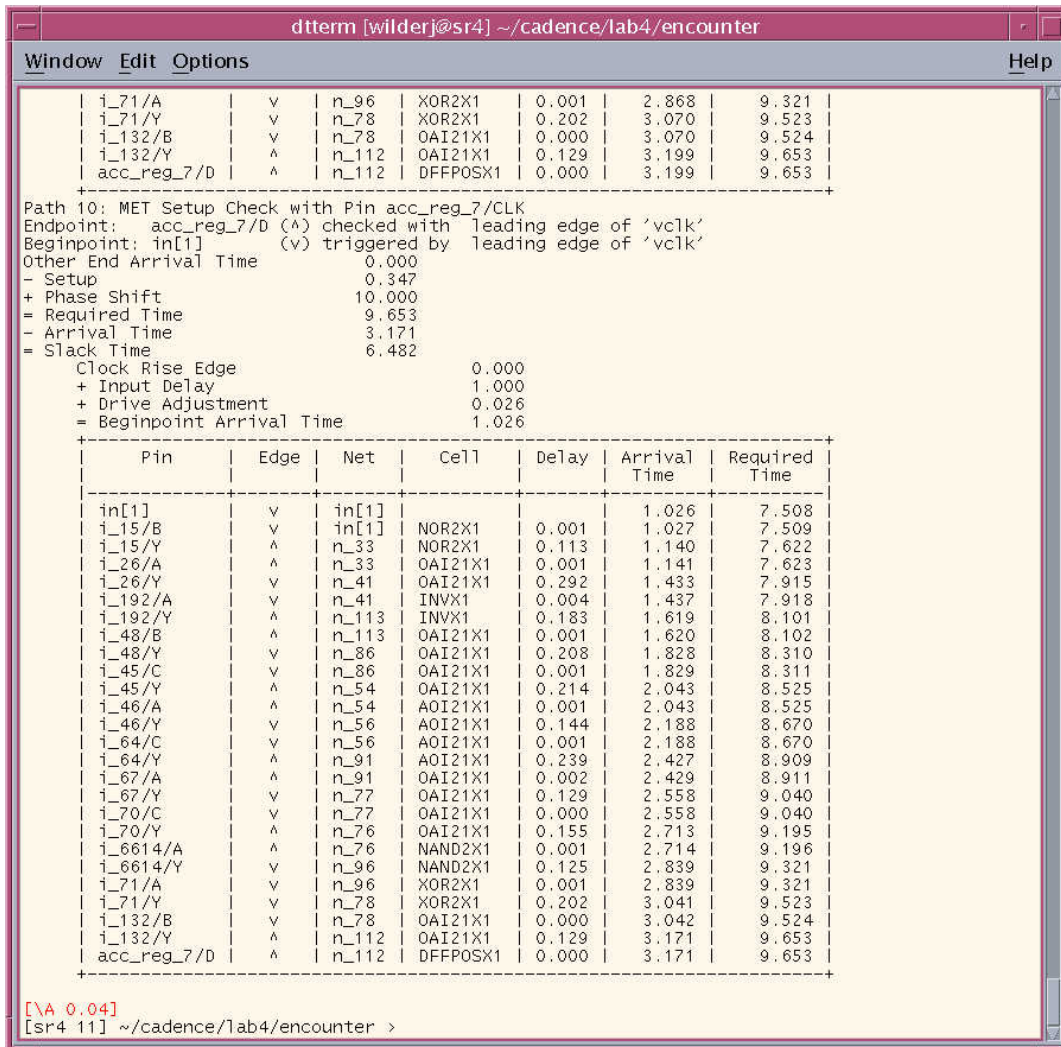


Figure 12. Post-layout slack time.

Locate the worst offender in the file and record the slack time (you will be asked to turn this in). Observe that this path is from in[0] to the input of the register for bit 7. Verify that this slack time is *less* than the slack time from the post-synthesized design. This is what you should expect since the delay for the data arrival to the clock edge has increased due to the interconnections of the gates (and thus there is less time between the moment when the data arrives at the register input and time when the clock edge will then latch it in).

7. IMPORT LAYOUT DESIGN INTO CADENCE VIRTUOSO AND SCHEMATIC TOOLS

Let's generate a schematic and a layout design from the automated Encounter layout design to perform some final checks on the design. Encounter exports the routed design into a GDS2 stream, which is the standard for describing mask geometry. You will use this file to perform the conversion. First, let's prepare by copying needed Cadence library files into your existing directory (should be \$HOME/cadence/lab4/encounter):

```
$ cp /apps/cadence2005/local/cdssetup/* .lib .
```

Next, perform the conversion by typing:

```
$ osucells_enc2icfb
```

The results of this conversion should be similar to Figure 13.

```
dtterm [wilderj@sr4] ~/cadence/test
Window Edit Options Help
[VA 0.09]
[sr4 91] ~/cadence/lab4/encounter > osucells_enc2icfb
/apps/cadence2005/local/lib
/apps/cadence2005/local/lib
/apps/iit_lib/osu/osu_stdcells/lib
Checking if final.gds2 exists.....OK
Determining top-level name.....OK (accu)
Creating temporary cds.lib.....OK
Determining Technology.....OK (AMI 0.5um)
Removing old library.....OK (accu)
Creating new DFII library.....OK (accu)
Creating PIP0 script file.....OK
Running PIP0 (GDS Stream-In).....
*****
* CADENCE Design Systems, Inc. *
* Virtuoso(R) STREAM Interface 5.1.0 *
* EXEC TIME : 22-Sep-2005 13:46:19 *
* @(#) $CDS: pipo.exe version 5.1.0 06/11/2004 20:28 (cds125839) $ *
*****
Reading Stream File ...
*** There were 0 error and 1 warning messages ***
Statistic and more information, please check /home/grad/wilderj/cadence/lab4/encounter/PIP0.LOG file.
NORMAL EXIT ...

Creating IHDL script file.....OK
Running IHDL (Verilog In).....
@(#) $CDS: ihdl version 5.1.0 06/11/2004 20:23 (cds125839) $: (c) Copyright 1994-1995, Cadence Design Syst
*WARNING* techOpenTechFile: unable to open file techfile.cds in library accu in r mode
Analyzing design file(s) ...
Done
Creating Schematic ...
Generated sheet 1 for schematic accu

Cleaning up.....OK
Good by.
```

Figure 13. GDS2 conversion to Cadence layout/schematic.

You will need the same library files that you used from Tutorial 3: cds.lib, osu.lib, and ncsu.lib. Copy those files from your Tutorial 3 work area into your Tutorial 4 work area (or re-download them from the website).

Start icfb at the prompt:

```
$ icfb&
```

First, open up the schematic in your accu library. This schematic was automatically generated from the post-routed netlist. Take a good look at this design and think about how it compares to the schematic that you generated for the 8-bit accumulator in Tutorial 3. Perform a DRC on this schematic (check and save) and verify that there are no errors. **Print out the results of the DRC from the CIW window** (you will hand in this print out to the instructor). Close the schematic view.

Next, open up the layout view in your accu library (should be identical to what was generated in Encounter). You will need to go to **Options->Display** and change the setup from 0 to 32 (remember that you have had to do this in the past). You should now be able to see the routed design. Have a look around the design. Perform a DRC on the layout (**Verify->DRC->OK**). You will see 20 or so errors having to do with improperly formed shapes. Fix this the way you did in Tutorial 3: Go to **Verify->Markers->Find** and select **Zoom to Markers**, and then click **Next**. The marker will point to some text label. Select the text and type 'q' to bring up the properties. Notice that this text label has been assigned to a metal layer and will need to be changed to the text-dg layer. You can do this with the menu selection. Change all of these (go back and select **Next** in the find markers window to find the next one), and then re-run DRC and verify that there are no errors in the CIW window. **Print out the results of the DRC from the CIW window** (you will hand in this print out to the instructor).

Next, you will extract parasitic capacitances from your layout by going to **Verify->Extract**. Click on **Set Switches** and select **Extract_parasitic_caps** in the resulting window and click OK. Your Extractor window should now look like Figure 14.

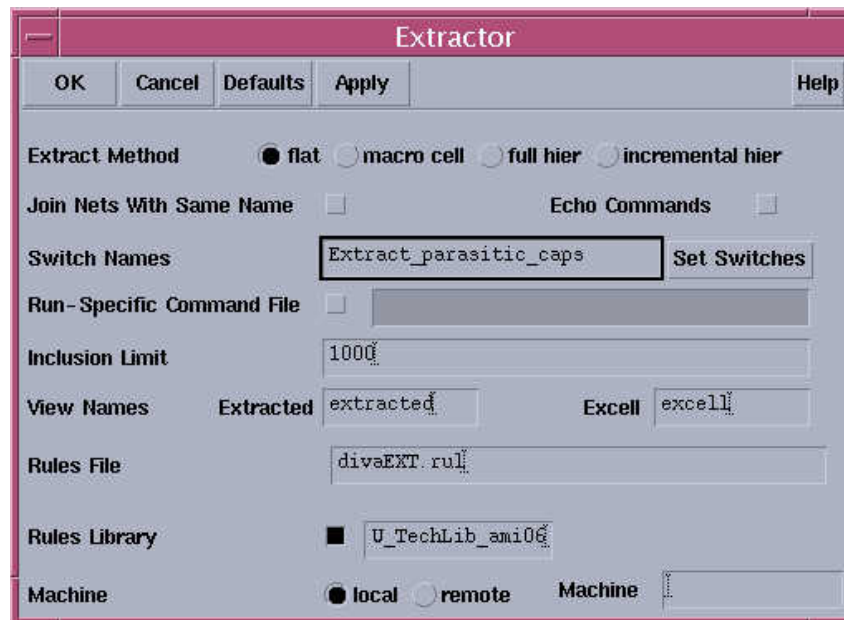


Figure 14. Extractor Window.

Click on OK to perform this operation. Now close the layout view of your design. Bring up the extracted view of your layout by double-clicking it from the library manager (Figure 15).

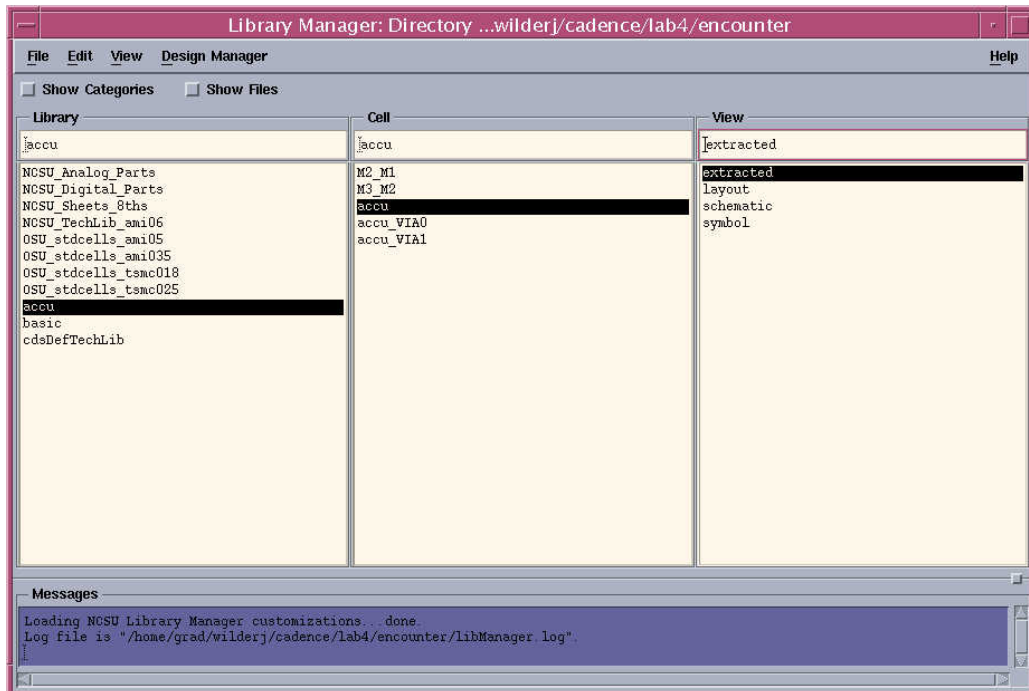


Figure 15. Find accu extracted view.

Zoom in on the extracted layout and find capacitors that have been placed in the design to represent the parasitic capacitances.

Next, you will perform a layout-versus-schematic comparison as another design check. Go to **Verify->LVS** and you will see the window as shown in Figure 16. Fill it out according to this window.

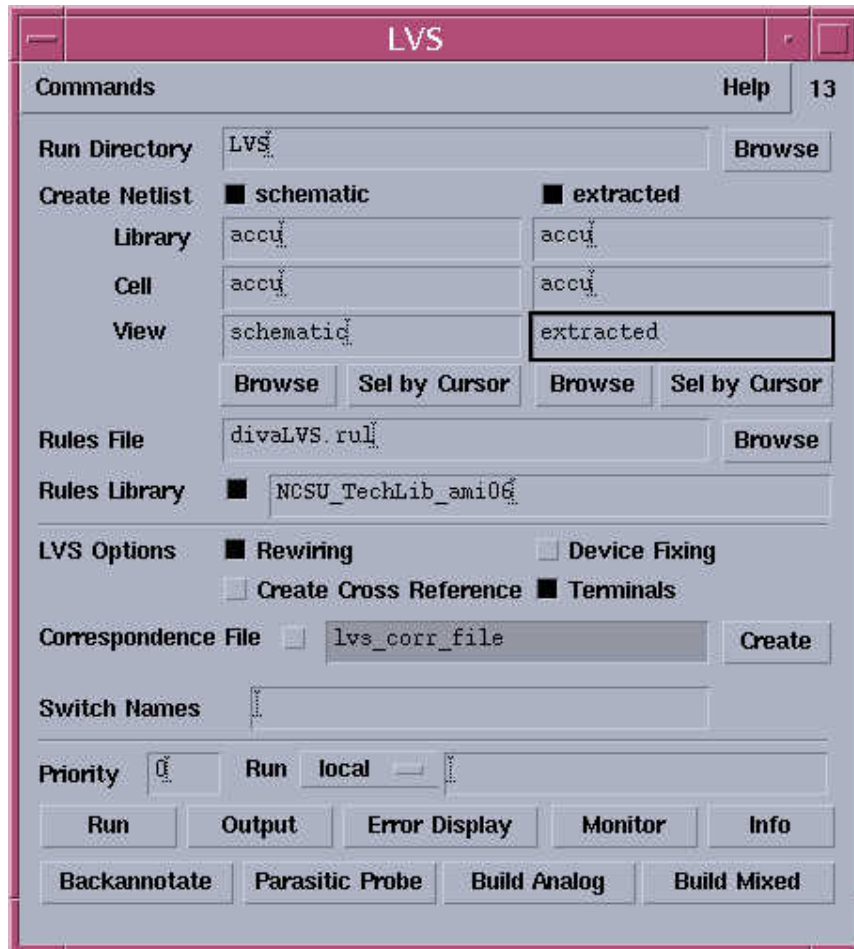


Figure 16. LVS Window.

Click **Run** and wait *patiently* for a popup window to tell you that the LVS process has completed successfully. Note that this is not telling you that the layout and schematic netlists are identical. To find out the results from the LVS, click on **Output**. Determine if the netlists match from the resulting window. **Print out these results** (you will hand this in to the instructor).

8. ADD PADFRAME TO CORE

Repeat the above Verilog workflow for the accumulator design with padframe added (use `accu_pads.v` – Note that this file instantiates the core of the accumulator design and adds pads plus internal wiring. Observe the use of the `<signal_name>_i` to specify the wiring from the core to the pad. Also, observe the special naming of the pads that was detailed in Tutorial 3, which must be in agreement with the `encounter.io` file that's used during place-and-route to arrange the orientation of the pads in the padframe.)

Do the following (you may wish to create a separate workspace, such as `cadence/lab4/pads`):

- Simulate your verilog code
 - In order to do this with the pad instances added to your code, do the following:
 - `$ cp /apps/iit_lib/osu/osu_stdcells/flow/ami05/osu05_stdcells.v .`
 - `verilog osu05_stdcells.v accu_pads.v accu_test.v`
 - (Note that “`osu05_stdcells.v`” was added to the Verilog simulation command line. This is necessary since it contains the definitions of the pad cells you

instantiated. The Verilog simulator will issue warnings related to negative hold times in osu05_stdcells.v. You can safely ignore those warnings. They will appear anytime you include "osu05_stdcells.v" on the Verilog simulation command line.)

- Run Simvision to ensure your design is working as expected
- Synthesize your design based on AMI 0.5um technology (you can keep 100MHz as the target CLK frequency)
 - **Record the worst slack time** (*you will hand this in*).
 - Answer the question: **What's the theoretical maximum operating frequency limit of this present synthesized design?**
- Run a post-synthesis verification of your design (simulate it) to ensure it's working
- Perform auto place-and-route on your synthesized design
 - In order to do this, you will need to do the following (same as was done in latter part of Tutorial 3):
 - Modify encounter.conf as shown in Figure 17 (tell Encounter to use the I/O file)
 - Modify encounter.tcl as shown in Figure 18 (specify the proper spacing from the pad frame to the core and remove the "-center 1" option from the "supply rings around core" section)
 - **Record the core size and chip size of this design** (*you will hand this in*)
 - **Record the worst slack time in your post-routed design** (*you will hand this in*)
- Import the design into Cadence Virtuoso and Schematic editors
 - Check for any DRC errors in the schematic and layout views (Note: Don't worry about fixing errors in the layout view. The pad designs don't meet the conservative DRC requirements of your core, and since they came from a standard library, you can't do anything about them. The schematic view should **not** have any errors.)
 - Extract the layout and perform LVS. Note that the netlists do not match. This discrepancy is due to the pads, once again, and details the importance of first doing these checks **without** the padframe, on the core only!
 - **Make a printout of your synthesized schematic** (*you will hand this in*)
 - **Make a printout of your extracted layout** (*you will hand this in*)

```

emacs: encounter.conf
File Edit View Cmds Tools Options Buffers Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News
encounter.conf
#####
#
# FirstEncounter Input configuration file      #
#                                             #
#####
#
# Specify the name of your toplevel module
set my_toplevel accu

# Only if your design as pads:
# - make sure the names in 'encounter.io' match your pad names
# - uncomment the following line
set rda_input(ui_io_file) "encounter.io"
#####
# No changes required below
#####

global env
set OSUcells $env(OSUcells)

global rda_input
set rda_input(ui_netlist) $my_toplevel.vh
set rda_input(ui_timingcon_file) $my_toplevel.sdc
set rda_input(ui_topcell) $my_toplevel

set rda_input(ui_netlisttype) {Verilog}
set rda_input(ui_11m1list) {}
set rda_input(ui_settop) {1}
set rda_input(ui_cell111b) {}
set rda_input(ui_iolib) {}
set rda_input(ui_areaolib) {}
set rda_input(ui_blklib) {}
set rda_input(ui_kboxlib) ""
set rda_input(ui_timelib) "$OSUcells/11b/ami05/11b/osu05_stdcells.t1f"
-----XEmacs: encounter.conf (Fundamental)-----Top-----
Loading efs-cu...done

```

Figure 17. Modify encounter.conf as shown.

```

emacs: encounter.tcl
File Edit View Cmds Tools Options Buffers Tcl Help
#####
# Run the design through Encounter
#####
# Setup design and create floorplan
loadConfig ./encounter.conf
commitConfig
# Create Floorplan
floorplan -r 1.0 0.6 450 450 450 450
# Add supply rings around core
addRing -spacing_bottom 9.9 -width_left 9.9 -width_bottom 9.9 -width_top 9.9 -spacing_top 9.9
.9 -layer_bottom metal1 -width_right 9.9 -around core -layer_top metal1 -spacing_right 9.9
-spacing_left 9.9 -layer_right metal2 -layer_left metal2 -offset_top 9.9 -offset_bottom 9.9
-offset_left 9.9 -offset_right 9.9 -nets { gnd vdd }
# Place standard cells
amoebaPlace
# Route power nets
sroute -noBlockPins -noPadRings
# Perform trial route and get initial timing results
trialroute
buildTimingGraph
setCteReport
reportTA -nworst 10 -net > timing.rep.1.placed
# Run in-place optimization
# to fix setup problems
setIPOMode -mediumEffort -fixDRC -addPortAsNeeded
initECO ./ipo1.txt
fixSetupViolation
endECO
buildTimingGraph
setCteReport
-----XEmacs: encounter.tcl (Tcl)-----Top-----
Loading tcl...done

```

Figure 18. Modify encounter.tcl as shown.

9. ASSIGNMENT

Based on your efforts of following the verilog to synthesis to post layout design flow, turn in the following to the lab instructor:

- Core design only (no pads):
 - The core size and the chip size of your routed 8-bit accumulator design (8%)
 - The worst slack time in your **post-layout** design (8%)
 - Printout of the DRC from the Cadence schematic (8%)
 - Printout of the DRC from the Cadence layout (8%)
 - NOTE: Any DRC errors will result in –5 points each
 - Printout of the LVS results (8%)
- For integrated circuit design (core plus pads):
 - The worst slack time in your **post-synthesized** design (8%)
 - What's the theoretical maximum operating frequency limit of this present synthesized design? (8%)
 - The core size and the chip size of your routed 8-bit accumulator design with padframe(8%)
 - The worst slack time in your post-layout design (8%)
 - Printout of your extracted layout (8%)
 - Printout of your synthesized schematic (8%)
 - Compare the post-layout slack time of your design with and without pads. Which slack time is better and why? (12%)