

EE 330 Spring 2014

Laboratory 12: Design and Simulation of Digital Circuits using Hardware Description Languages

Purpose: The purpose of this experiment is to develop methods for using Hardware Description Languages for the design of digital circuits.

Part 1: Background

Hardware Description Languages (HDL) are widely used for representing digital systems at both the Behavioral and Structural levels. Full functionality of a digital system can ideally be captured in a HDL and the design of digital circuits is often accomplished within the confines of an HDL framework. At the appropriate level in the HDL, CAD tools can take a design down to the mask level for fabrication in silicon with minimal engineering intervention at the Physical Level in the design process. A system appropriately represented in a HDL can be simulated to predict not only the functionality of sequential and combinational logic circuits, but also timing information about how those circuits are predicted to perform when instantiated in silicon.

The two most widely used HDLs today are Verilog and VHDL. There is considerable similarity between these two languages and engineers are expected to be proficient in both. In this laboratory experiment we will limit our discussion to Verilog which has become more popular in US industry today. Specifically we will focus on how a HDL can be used for design and simulation of digital integrated circuits.

Appendix A of the Weste and Harris text has a brief discussion of Verilog and students should become familiar with the material in this appendix. There are also numerous books and WEB sites devoted to a discussion of Verilog. Beyond the basic introduction to Verilog discussed in this laboratory experiment, students will be expected to take the initiative to develop their own HDL skills to the level needed to support the digital design component of this course.

Part 1.1: Structure of Verilog Representations

A Verilog representation of a digital system, be it a small or a very large system, is characterized by a set of modules that describe the system. A large system will be comprised of a large number of nested modules whereas a small system may be comprised of only a single module. As such, the main building block in Verilog is the *module*.

The structure of a *module* includes statements about input/output variable mappings along with a description about how the input and output variables are related. An example of a module named “testcircuit” is shown below:

```

module testcircuit (vout, vin);
output vout;
input vin;
assign vout = ~vin;

endmodule

```

This module has one output, vout, one input, vin, and two bidirectional ports, vdd and vss. The relationship between the input and output for this simple module is defined by the assign statement. The module description is always terminated with an *endmodule* statement. In Cadence, we can create behavioral modules several ways, two of which will be discussed in this experiment.

Symbol-Based Module Creation

With the Symbol-based method, a symbol view (that may have been created earlier) is used to generate a behavioral view with the input/output pin and direction information. The user must then enter the remainder of the module to fully define the operation of the circuit.

Text-Based Module Creation

The user creates a module by creating a text file that includes the module name, pin names and input/output direction information. This text file defines the behavioral view of the module. The user must then enter the remainder of the module to fully define the operation of the circuit.

Modules that represent a digital system are simulated by using ModelSim software. ModelSim was looked at in an earlier lab and knowing how to use it is important to successful simulation of a module. To use ModelSim to simulate a module, a testbench for the design under test (DUT) must be created. This is another text file written in Verilog syntax which provides the test inputs to the DUT to check for correct functionality.

In this laboratory, you will be asked to create behavioral verilog of an inverter and the gates you used in Lab 4. You will also be asked to simulate each gate as well as the top level Boolean implementation of Lab 4 using individual test benches for each instance. Once the behavioral description of all the gates is complete, you will then simulate your top level design of Lab 4 in Verilog.

Part 2: Inverter

2.1 The Verilog File

Create a behavioral view of the Boolean Inverter that was created in Lab 2. The following steps create the skeleton of the module from which you can add the functional description. In the Library Manager, select the inverter that you designed in Lab 2 (or the updated version from Lab 4). Click on **File** → **New** → **Cell View** and in the View Name field, type **behavioral**. The Tool field should be set to **Verilog**. When you click OK, an *emacs* file editor window with the following contents will open:

```
//Verilog HDL for "your_library", "your_inverter" "behavioral"3
```

```

module your_inverter (vout, vdd, vss, vin);
    output vout;
    inout vdd;
    inout vss;
    input vin;
    (Add your code defining the operation of the inverter here)
endmodule

```



If you prefer to use another editor, type `editor="pico"` in CIW to choose pico. To use vi, use the command `editor="xterm -e vi"`. The change remains valid for the current Cadence session only. To make this choice permanent, add the command to your `~/ .cdsinit` file.

Since you've already designed, checked and saved this gate in the Cadence environment, the Verilog-Editor will have prior knowledge of how your module looks like, and we only have to add the line that describes the output. To save and exit in the *emacs* editor, click **Ctrl-X Ctrl-C**, then when prompted if you want to save, click y.

What we just did is write a description of the inverter in a behavioral way. In other words, how do the outputs relate to the inputs, without knowledge of how the operation would be implemented with transistors.

Often, an easier way to create the behavioral file for a given module is outside of the Cadence environment. Instead of creating a new cell view, create a .v file using the command *gedit* and type the module manually. Be sure to save the file as a .v file so later steps work correctly. This file will be simulated in ModelSim and later used in synthesis.

2.2 Simulating Behavioral Code

Now we are ready to simulate our file. Be sure to download the ModelSim source file. Create a directory in your EE330 folder to use for simulation. Change to this directory and source the ModelSim text file using the following command in the terminal.

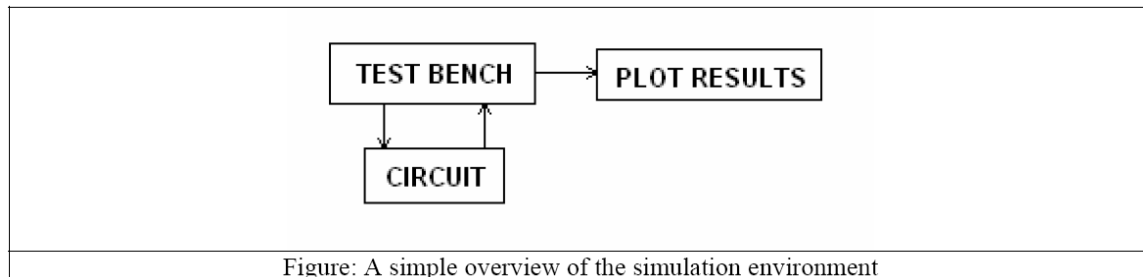
```
source <file_location>/ModelSim_env.txt
```

To open ModelSim type:

```
vsim &
```

Before simulating, a test bench for the design under test needs to be written. The test bench is needed to specify the time and value properties of inputs. For given inputs, certain outputs are expected which can be plotted during simulation and compared to expected results.

To create a test bench, start by considering the inputs and outputs of the DUT. The test bench should provide the inputs to the DUT so the test bench module will have outputs for the DUT's inputs. Also, outputs of the DUT are to be monitored, so inputs to the test bench are these signals. Thus, the DUT's inputs become the TB's outputs and the DUT's outputs are the TB's inputs. The figure below makes this relationship clear.



The TB for a DUT is also a Verilog file with certain syntax for applying inputs to the DUT at specific times. An example test bench for the inverter module is given below. Create the TB file in the same directory as the module it is testing, usually with a similar name such as *modulename_TB.v*. The '#' symbol has the simulation wait that many time units (default is nanoseconds) before the line is run. Remember that *initial* blocks run once at the beginning of the simulation whereas *always* blocks run continuously, over and over while the simulation is taking place.

```
// Verilog TB file.
module inverter_TB(vout, vin);
input vout;
output vin;
reg vin;

//Instantiation of DUT with generic name
inverter name(vout, vin); //Order of inputs must match original module

initial
begin
    vin = 1'b0;
end

always
    #20 vin = ~vin;
```

This TB will initially set *vin* at a (1 bit) value of 0, and after every 20 time units, will invert it. The whole process will last 80 time units.

Now we are ready to simulate. In ModelSim, create a new project using the menus and locate the new project in the simulation directory. Add files to the project as was done in Lab 2, including the Verilog files for the DUT and the TB. Now, compile the files using **Compile -> Compile All**. Once all errors are fixed and compilation completes successfully, run simulation by clicking **Simulate -> Start Simulation**. In the window that appears, **uncheck** the **Optimize Design** box and select the TB file as the simulation file under the **work** dropdown. Click **OK** to start simulation.

2.3 Accessing Simulation Results

Now we are ready to view the results. We want to view the waveform of the results to confirm the DUT is operating as expected. To do this, make sure simulation is completed without errors. Select **View -> Wave**. A waveform viewer will open. Click and drag signals from the object pane to the waveform viewer. Once all desired signals are selected, run simulation by changing simulation time and hitting the **Run** button next to the time field. Right-click on the waveform and select **Zoom Full** to view the waveform. Compare again what you expect for the output versus the inputs.

Part 3: Behavioral Views of Boolean Gates

Create Behavioral Views for all of the gates used in Lab4. Present the simulation results for all of the individual gates in ModelSim.

Part 4: Simulation of a Boolean System

We will next simulate a system made from gates designed earlier. Create a Verilog behavioral file for the function you created in Lab 4 and a corresponding test bench to test its functionality. Simulate and compare results with those found in Lab 4. They should agree, but one obviously took less time to create and simulate.

Part 5: Verilog Synthesis with RTL Compiler

Introduction

The purpose of this section is to develop investigate the concept of synthesis of digital systems from a hardware language description of the system. This is the intermediate step in a three-step design process which starts with the development of a HDL description, followed by synthesis into a set of standard cells. The third step is placing and routing the cells to create a layout of the circuit which can then be fabricated. The generalized digital circuit design flow, with the topics discussed in this tutorial highlighted, can be found below:

1. Design in HDL (Verilog file)
2. **RTL Compiler** (Verilog file --> Synthesized Verilog file)
3. Encounter (Synthesized Verilog file --> Layout)
4. Cadence (not looked at in this lab)
 1. Layout Import (Encounter --> CIW Import Stream)
 2. Netlist Import (Synthesized Verilog file --> Verilog Import)
5. LVS Verification

RTL Compiler

RTL Compiler is a tool that synthesizes and then optimizes behavioral circuit elements. Logic synthesis translates textual circuit descriptions like Verilog or VHDL into gatelevel representations. Optimization minimizes the area of the synthesized design and improves the design's performance. The HDL description can be synthesized into a gatelevel net-list composed of instances of standard cells.

Once the gate-level net-list is finished, it can be imported into Encounter and used to create a layout. Consistency between the layout and schematic can then be verified with an LVS within Cadence.

Preliminary Setup

To use RTL Compiler and Cadence Encounter tools on the lab computers, the correct location needs to be sourced. Open the ~/.software file from Lab 1 and add the following line to the end of the file.

```
EDI          *make sure this is the last line and is all caps
```

Then, the bash setup script needs to run so type 'bash' into the Linux terminal. This will source the correct location where the binaries for both pieces of software are located. If remote connecting or using certain computers in Coover, this step is not required as RTL Compiler may already be installed. If trying to source the location and an error occurs that the location does not exist, try to move on to the next step before seeking help.

Setup Standard Cell Library

First, the standard cells in the new OA version of Cadence are needed. They are found on my webpage and can be downloaded from the terminal with the following commands:

```
cd ~/ee330
wget http://home.eng.iastate.edu/~aafayed/ee330/Lab/OSU_stdcells_ami05.tar
tar -xvf OSU_stdcells_ami05.tar
```

This will download and extract the OSU standard cells for use. Now change the **cds.lib** file in your ee330 directory by adding the following line.

```
DEFINE OSU_stdcells_ami05 ~/ee330/OSU_stdcells_ami05
```

By opening up Virtuoso, the OSU05 standard cells should now appear. Click around to the different cells to make sure the layout and schematic views can be viewed. The standard cell library should now be available to use.

Standard cell files for RTL Compiler and Encounter

Now, the necessary files for use in RC and Encounter must be obtained to run the digital design flow. We will be using the OSU05 files in this lab. This cell library uses the same AMI05 process that we should be familiar with from the earlier labs.

Use the following commands to download the tar file which contains the LIB, LEF, and Verilog files for the OSU standard cells. Put the downloaded files in the folder where RC will be run.

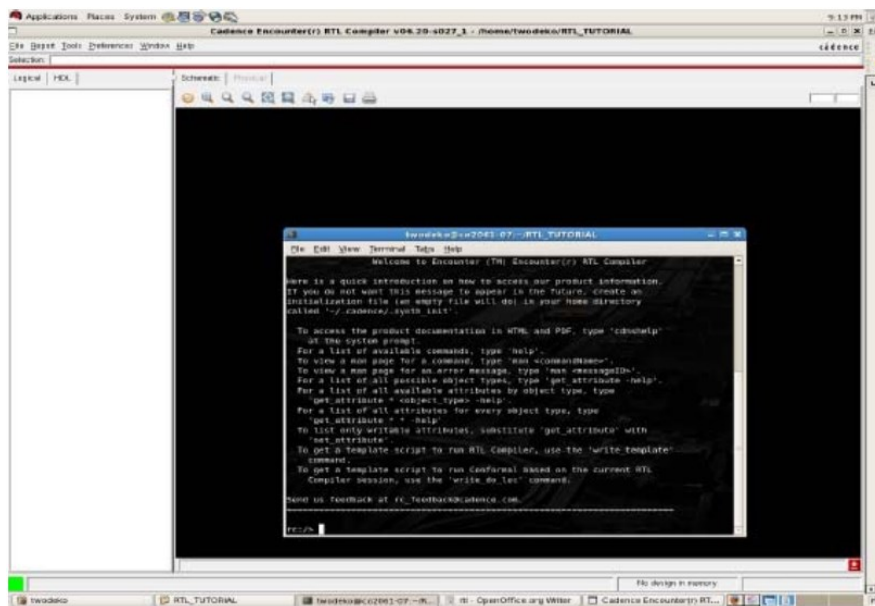
```
cd ~/ee330/ (or wherever you want to run RC)
wget http://home.eng.iastate.edu/~aafayed/ee330/Lab/rtl.tar
tar -xvf rtl.tar
```

Now the files are downloaded and extracted, which can be confirmed by looking for the *lab12_ee330* folder in the location used above. Throughout this lab, carefully watch for path names and make sure they are pointing to the correct directories and files.

You should now have a folder named *lab12_ee330* that contains folders and files needed for the rest of the tutorial. Change directories to *lab12_ee330* and type the following to begin RTL Compiler:

```
cd lab12_ee330/syn
rc -gui
```

A GUI will show as well as a command line window.



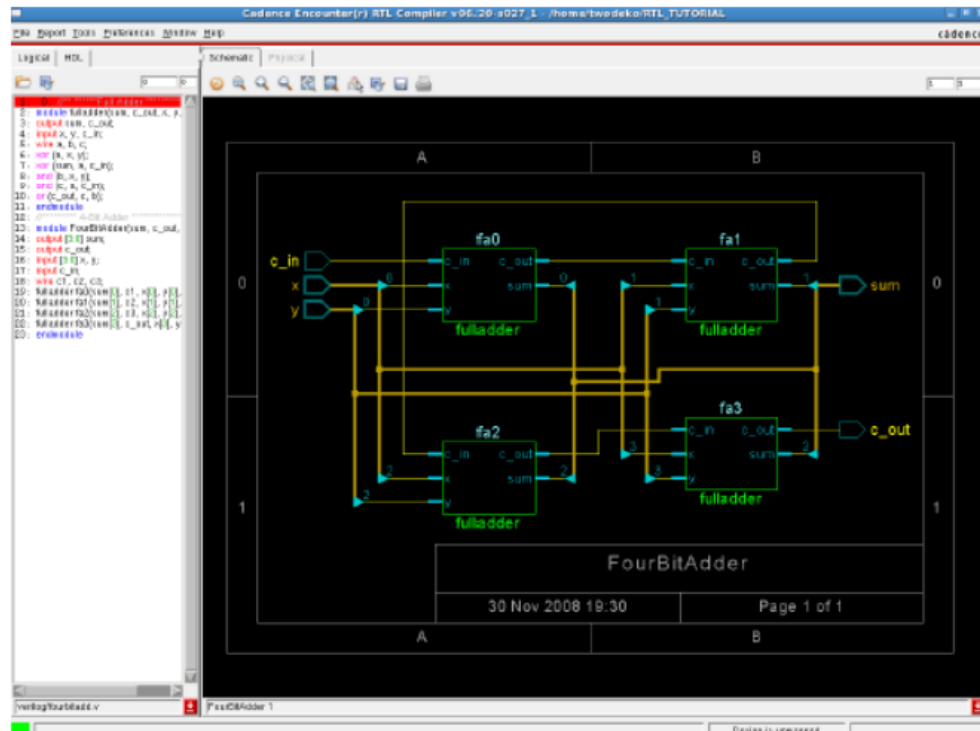
In the command window that has a “rc:/>” prompt, type the following to load the correct standard cell libraries and run the synthesis script:

```
set OUTPUT_DIR ./run_dir
set_attribute lib_search_path ../libdir
set_attribute library {osu05_stdcells.lib}
```

Once the library is set, you need to load and elaborate your Verilog file. In this case, it was placed in the rtl folder and can be processed with the following commands:

```
load -v2001 ../rtl/FourBitAdder.v
set DESIGN "FourBitAdder"
elaborate $DESIGN
```

When Verilog is elaborated, each line is evaluated and replaced with a combination of logic gates. The blank GUI screen should be replaced with circuit blocks that represent the input Verilog, which was the following for our four-bit adder:



At this point, RTL Compiler has generated an unoptimized version of the Verilog you provided. The final step is “synthesizing” the elaborated design, which optimizes the schematic already created and integrates your design with the TSMC standard cell library. While there are many optimization options available to you, we will skip them for this tutorial. The design can be synthesized with the following command:

```
synthesize -to_mapped -eff high -no_incr
```

The GUI window should be updated. You can check this by double-clicking on one of the fulladders to verify that the logic gates are attached to symbols and proper names (XOR2X1, AOI22X1, etc.) Exporting the synthesized Verilog file is done with these two commands:

```
write -mapped > ${DESIGN}_synth.v  
write_sdc > ${DESIGN}.sdc
```

Import Schematic to Cadence Virtuoso after Synthesis

After synthesis, we want to import the gate-level netlist created in RTL Compiler into Cadence Virtuoso. Open Virtuoso and in the CIW (where the errors and other messages appear) select *Import -> Verilog*.

The following screen will appear. Fill it in as is shown in the screen shot except replace the Verilog file with your synthesized Verilog file created by RC and your own library name. This file will end with `_synth.v` if following the above steps.

(*Make sure to use the OSU library name that you defined above in the Reference Library field – for example “*OSU_stdcells_ami05*”)



The import process should complete and may have a few errors or warnings. Ignore for now. Go to the schematic cell view created by the import process and open in Virtuoso. The synthesized design should appear with the lowest level of hierarchy being the standard digital blocks made of transistors in the AMI06 process. If not something is wrong. Retry the import process or possibly synthesis before asking for help from the TAs.

Part 6: Layout of Digital Circuits with Encounter

Introduction

In a typical digital design flow, a hardware description language is used to model a design and verify the desired behavior. Once the desired functionality is verified, the HDL description is then taken to a synthesis tool such as RTL compiler. The output of RTL compiler is a gate-level description of the desired circuit. The next step is to take this gate-level description to a place-and-route tool that can convert it to a layout level representation. In this tutorial, you will be introduced to a place-and-route tool called SoC Encounter. For the purposes of this tutorial, it will be assumed that you have a synthesized Verilog file ready to be placed and routed.

The generalized digital circuit design flow, with the topics discussed in this tutorial highlighted, can be found below:

1. Design in HDL (Verilog file)
2. RTL Compiler (Verilog file --> Synthesized Verilog file)
3. **Encounter (Synthesized Verilog file --> Layout)**
4. Cadence
 1. Layout Import (Encounter --> CIW Import Stream)
 2. Netlist Import (Synthesized Verilog file --> Verilog Import)
5. LVS Verification

SoC Encounter

SoC Encounter, in its simplest form, allows a user to create a layout from a synthesized HDL file. Once the synthesized file is imported, Encounter provides a variety of tools to create a floorplan for your design, place standard cells, and automatically connect power rails and local routing. While this tutorial provides a brief overview of Encounter, there are many other optimization and layout options available.

Preliminary Setup

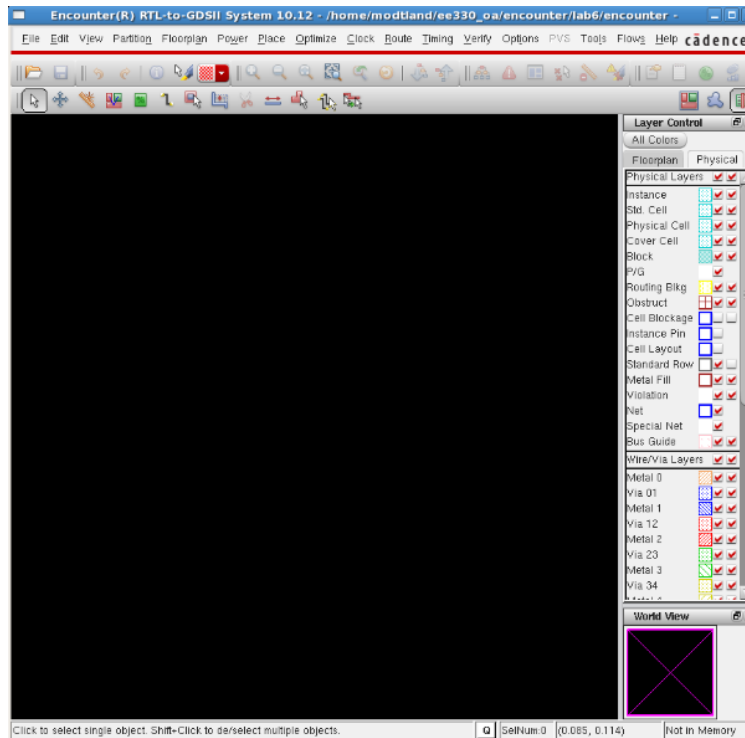
Before continuing with this tutorial, make sure that you have run through the preliminary setup instructions listed with the RTL Compiler tutorial. We need the LEF, LIB, and Verilog files for the OSU library. You should also confirm that two MAP files are in your *libdir* directory for the last step. If you haven't done so source the */etc/software/edi* location. You may have to do this more than one time if you change directories.

Running SoC Encounter

In the *lab12_ee330* folder, create a folder named *encounter* and then enter the *encounter* folder. Encounter can be started from here with the following command:

```
encounter
```

You will be presented with a GUI, which has been shown below.



Importing Synthesized Verilog

Once Encounter is initialized, you need to import your synthesized HDL description file. But first, set the power nets for a later step by typing the following commands in the encounter terminal.

```
set rda_Input(ui_pwrnet) {VDD}
set rda_Input(ui_gndnet) {GND}
```

With that complete, go to *File -> Import RTL*. Fill in the Verilog Files field with the Verilog files used in the design. Then give a top level name to the design. Under the max timing library field, load the *osu05_stdcells.lib*. Next, under Timing Constraints, load the SDC file created by RTL Compiler. Move to the Physical tab and load the LEF file for the OSU standard cells called *osu05_stdcells.lef* which was included in the *libdir* directory. Finally, in the OA Reference Libraries, add the *NCSU_TechLib_ami06* tech library. This will allow the tool to reference the ami06 library for layer information. Click OK and make sure the RTL is successfully imported by looking at the terminal for errors.

Go to *File -> RTL Synthesis*. Click OK to synthesize the RTL code again. This is performing the same task as RTL Compiler but for the use of the layout tool. When this completes, the floorplan step from Lab 12 can be completed.

Floorplanning

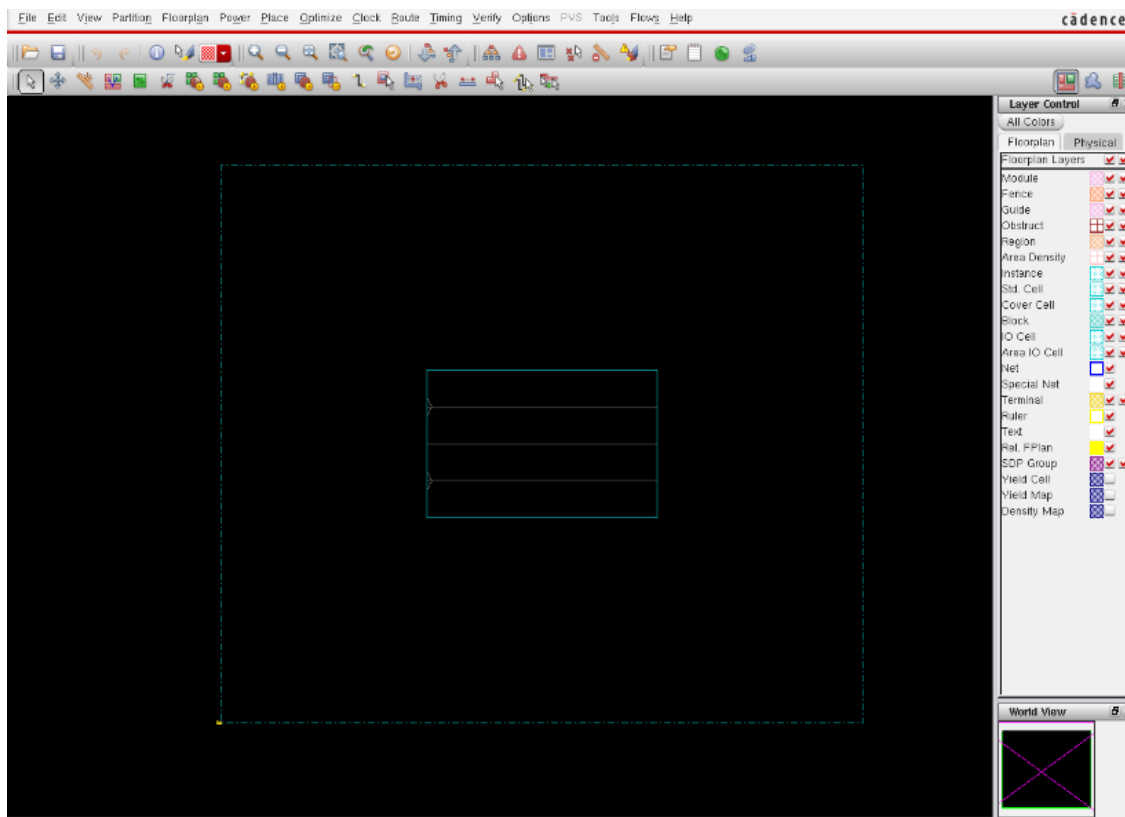
The Encounter GUI should now display an empty “die” and the information box in the bottom-right corner should read “Design is: In Memory.” The next step is to specify a

floorplan for your layout. Floorplanning is done to specify the dimensions of your layout and spacing between the core (or area where the standard cells are placed) and power/signal routing. Open up the floorplanning options by clicking on the *Floorplan* menu and selecting *Specify Floorplan*.

Options that you will need to pay attention to are:

- Aspect ratio – height/width ratio of the core.
 - Suggested: 1.0
- Core Utilization – the amount of core area used for placing standard cells. A larger value means that your design will be fairly compact but routing may be difficult, if not impossible. Smaller values will drastically increase the design area.
 - Suggested: 0.5
- Core Margins by – specifies the distance between the core and edge of the die. The margin area needed is proportional to the number of input/output (I/O) pins.
 - Suggested: 10 for core to left, right, top, and bottom

Once these values are entered, the layout die/core spacing and aspect ratio should be updated.



Power Planning

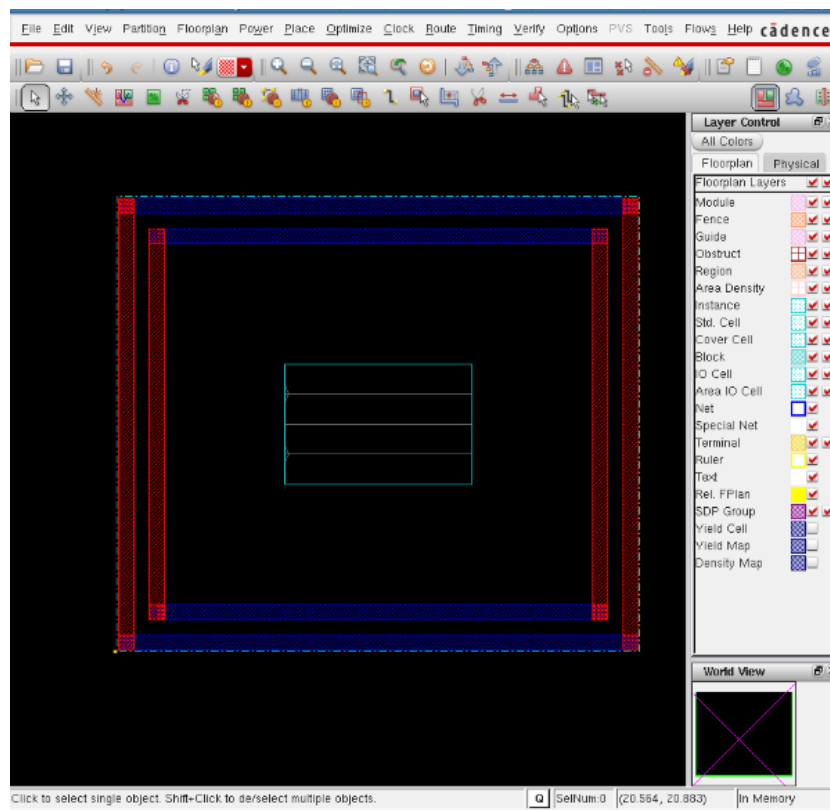
In order to connect your supply voltages to the standard cells, power and ground need to be available on all sides of the die. This is done by adding power rings and specifying global net connections.

Select the *Power* menu and click on *Power Planning > Add Rings*.

A dialog box will pop up and allow you to specify the ring type and configuration. You can plan power in various modes to facilitate various design styles. Since it is preferable to have the routing done inside the power rings, it is a better idea to have them around the I/O boundary so choose this option. If it is not already entered, add “GND VDD” (without quotes) to the Net(s) field.

In the Ring Configuration section, verify that the width and spacing comply with the process design rules and are a multiple of lambda.

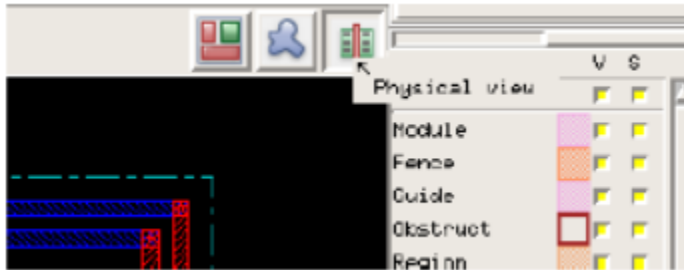
Once finished, you should see two rings that run inside the I/O boundary. It should look like the following:



Connection to the global nets is done automatically if the nets are given the same names as was done in the setting of the power nets above. If all was done correctly, power planning should be finished.

Placement

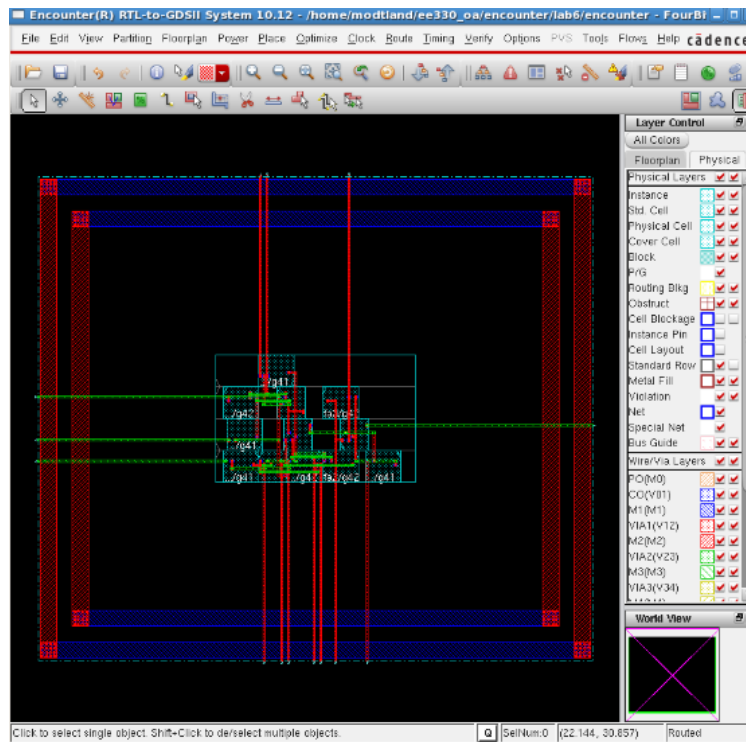
Now we are ready to perform the placement. At the end the cells used in the design will be placed so that they don't overlap and there is enough space for routing in between them. Select the *Place* menu and then *Standard Cells*. In the dialog box that appears, leave the default settings and hit OK. In the main GUI window, you can verify that the standard cells were placed by selecting the Physical View Icon in the upper-right:



You should see the cells placed in the middle of the core. The IO pins have shifted position accordingly as well.

If placement fails, it is most likely because Encounter cannot place the cells in the given area without causing gate overlaps. To solve this problem, you must start over by reinitializing the floorplan with relaxed constraints. The best constraint to change is the row utilization factor. By lowering this factor, you can increase the likelihood of placement without overlaps.

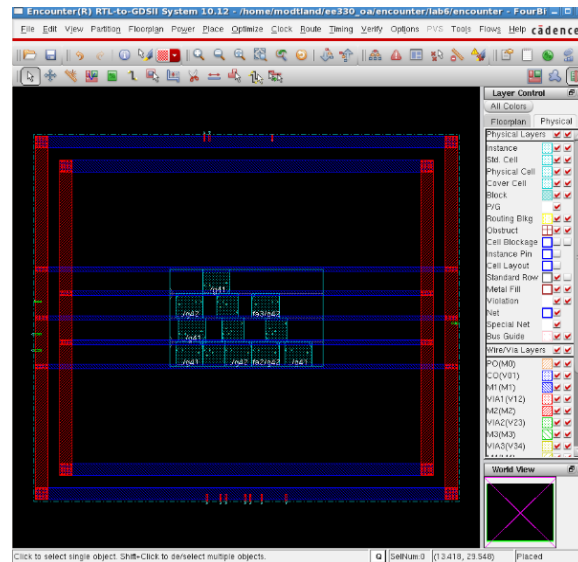
The final placement should look similar to the following:



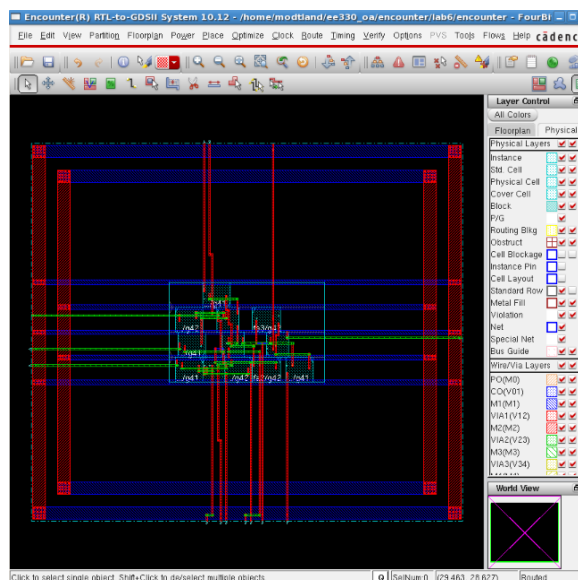
Routing

The built-in routing options within Encounter can now be used to connect the supply nets and create interconnects between the standard cells.

Power and ground routing is performed with the “Special Route” tool. This can be found by selecting the *Route* menu and then *Special Route*. In the dialog box that appears, unselect *Block pins*, *Pad pins*, and *Pad rings*. After clicking OK, you should see horizontal and/or vertical metal power routing added to your design.



Once power routing is complete, the remaining connections between standard cells and the die I/O can be routed using the “NanoRoute” tool. Select *Route* and then *NanoRoute* and finally *Route...* to enter the dialog box. You do not have to make any changes and can click OK. Your design should now resemble the following:



Filler Cells

If you import the complete layout from Encounter into Cadence at this point, you may have DRC errors due to nwell layers not being spaced far enough. You can fix these errors by hand in Cadence or you can add pre-designed filler cells with Encounter. These cells provide continuity for nwell layers and power rails between placed cells.

Filler cells are placed by selecting the *Place* menu and then *Physical Cell -> Add Filler*. In the dialog box that appears, click “Select” in the Cell Name(s) field and select the appropriate fill layers. FILLER CELLS IN THE OSU05 LIBRARY ARE JUST CALLED ‘FILL’ AND THERE IS ONLY ONE OF THEM. You can now click OK to fill in the blank regions in the core of your layout.

Verification

Before exporting your layout, you need to verify that there are not any geometric errors that occurred during placement or connectivity errors from routing.

This can be done by selecting *Verify Geometry* and *Verify Connectivity* in the *Verify* menu. Reports will be generated and placed in the directory that you started Encounter from. You can also look at the terminal used to start Encounter and see the results of the verification.

If no errors exist in either geometry or connectivity, the layout is good to fabricate. Save the design by selecting *File -> Save Design*. Give a name and save.

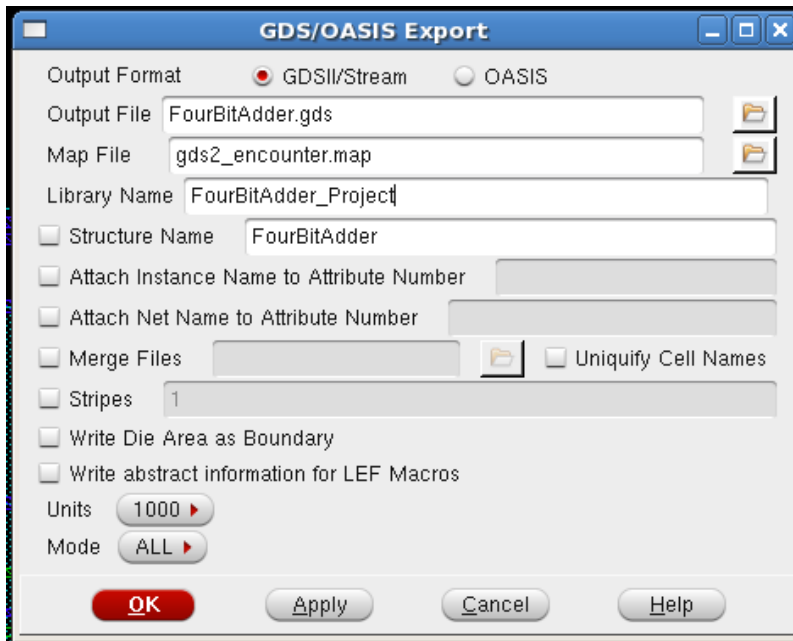
GDS Export of Layout from Encounter

Once the layout is complete in Encounter (post-routing and post-filler cells), then you need to perform an export of the layout. This is done in Encounter by going to *File -> Save -> GDS/OASIS...*

In the window that appears, select the GDSII radio button. Then type an output file name in ‘Output File’ field such as *design.gds*. In the map field select the ‘gds2_encounter.map’ file located in the *libdir* directory.

In the Library Name, type the Library in Cadence you would like to add the design to. I suggest making up a new library so nothing gets overwritten.

Leave all other fields as default making sure the units are ‘1000’ are mode is ‘ALL’. Click OK. Look at the terminal and make sure Stream out has finished successfully.



Now let's import the design to Cadence.

GDS Import of Layout into Virtuoso

After creating a layout in Encounter and creating the GDS file, open Virtuoso. Click on the CIW (Virtuoso window where errors and messages appear) and click *File -> Import -> Stream*. This may produce an error the first time. Try again and it should work and bring up an import window.

On the stream in window choose the GDS file created in Encounter as the 'Stream In' file.

Under library, type the name of the library you would like to save the module in. Then give a name to the top level cell once it is imported.

Attach the 'NCSU_TechLib_ami06' technology library.

Select *Show Options* and click the following Tabs with the following options:

General Tab:

- View Name should be layout.
- Check the box that says 'Replace [] with <>'

Virtuoso (R) XStream In

Stream File: /home/modtland/ee330_oa/encounter/lab6/encounter/FourBitAdder.gds

Library: FourBitAdder_Project

Top Level Cell: FourBitAdder

Attach Technology Library: NCSU_TechLib_ami06

Load ASCII Tech File:

☐ Generate Technology Information From Stream File

Stream Tech File:

Hide Options Save Template Load Template

StreamIn Options

General Cells Fonts Geometry Layers Libraries Properties

Log File: strmin.log

Summary File:

Run Directory: .

User Skill File:

View Name: layout Reference Tech Libraries:

☐ Automatically Detect Vias Via Map File:

Exclude MapToVia File:

Translation Depth: 32767

Text Namespace: cdba

Maximum Cells in Target Library: 20000

☒ [Replace [] With <>]

☒ Translate Entire Hierarchy

☒ Overwrite Existing Cells

☐ Append Existing Database

☐ Merge Purposes to Drawing

☐ Disable File Locking

Label Case Sensitivity

☐ Lower

☐ Upper

☒ Preserve

Number of Threads: 1

☐ Stream In to Virtual Memory

☒ Show Completion Message Box

Translate Cancel Apply Reset All Fields Help

Geometry Tab:

- Check 'Snap to Grid'

Virtuoso (R) XStream In

Stream File: /home/modtland/ee330_oa/encounter/lab6/encounter/FourBitAdder.gds

Library: FourBitAdder_Project

Top Level Cell: FourBitAdder

Attach Technology Library: NCSU_TechLib_ami06

Load ASCII Tech File:

☐ Generate Technology Information From Stream File

Stream Tech File:

Hide Options Save Template Load Template

StreamIn Options

General Cells Fonts Geometry Layers Libraries Properties

☐ Ignore Box Records

☒ Snap to Grid

☐ Keep Stream Cells

☐ Convert paths to pathsegs

☐ Translate Nodes

☐ Convert Arrays to Scalar Instances

☐ Report Bad Polygons and Paths

☐ Skip Undefined LPPs

☐ Ignore Zero Width Paths

☐ Report Precision Loss

Pin Attribute Number: 0

Scale Text Height: 1.00000

DB Units per User Units: 0

Number of Threads: 1

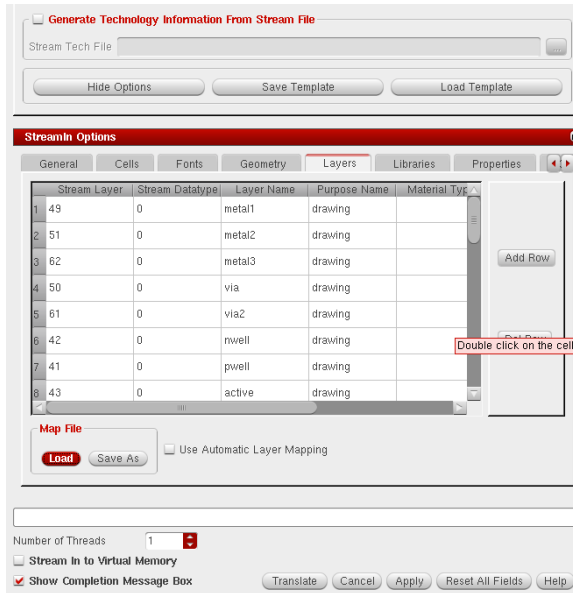
☐ Stream In to Virtual Memory

☒ Show Completion Message Box

Translate Cancel Apply Reset All Fields Help

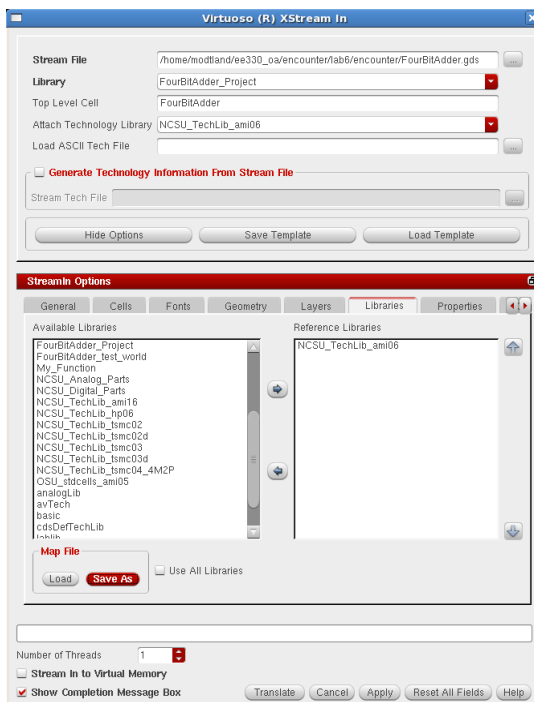
Layers Tab:

- Select 'Load' under "Map File" - Choose "gds2_icfb.map" and click ok to load the layers map. The screen should now appear as below.



Libraries Tab:

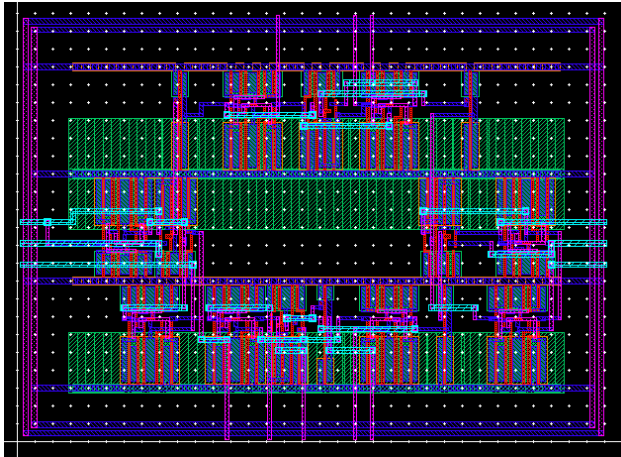
- Select the two libraries NCSU_TechLib_ami06 and the OSU_stdcells_ami05 library to obtain the reference cells.
- This setting must be saved by hitting Save As and then typing a name such as "import.map"



*****Make sure to add the OSU cells. This image doesn't have it added yet.**

Now we are ready to translate the file and import the design. Click *Translate*. If a box to save the libraries comes up make sure to save the libraries as something. The process will take a little bit. When complete there should be a log file and a message box that appears. This should have 0 errors and only a few warnings if not 0.

Go to the library created in the Library Manager and try to open the layout of the design. The standard cells should appear. Hit *Ctrl+F* to see the layout view.



At this point, the synthesized schematic and layout from Encounter have been imported into Virtuoso. Opening the layout allows DRC to be run. Some errors may exist due to the pins but they can be fixed manually by changing the layer information from text to the correct level of metal. This is done by changing the properties of the TEXT itself from a layer of metal to one of text. Once DRC returns zero errors, run LVS to compare to the schematic. Zero errors should exist if the lab has been followed closely.

Congrats on completing this tutorial. Please provide a summary of the steps required for synthesizing and performing layout with Cadence RTL Compiler and SoC Encounter. Explain what the main steps do and provide screen shots of major steps as well as the two verification steps at the end.

References

Verilog HDL Quick Reference Guide:

http://www.sutherland-hdl.com/on-line_ref_guide/vlog_ref_top.htm