ELEC-E3540 Digital Microelectronics II

Implementation instructions

Enrico Roverato

February 2018

Aalto University
School of Electrical
Engineering

Contents

1 General information

1.1 Introduction

During this tutorial, you will learn how to synthesize and generate a layout for a digital circuit, starting from its functional description in VHDL language. You will become familiar with the same software tools that are used in industry every day.

This document describes the digital flow for a simple Serial Peripheral Interface (SPI). The same flow applies to the PIC16F84A as well. However, because you are not familiar with synthesizable VHDL, your code will most likely require some modifications before it can be synthesized correctly. Hence, in order to familiarize yourself with the flow more smoothly, you are advised to first use the SPI interface. The necessary VHDL code is provided in the template directory.

The digital flow described in this tutorial is built around Cadence's Generic 45nm Process Design Kit (PDK).

1.2 Design flow

Figure 1.1 represents a typical design flow of a digital circuit, as well as the tools used in different phases.

The design process begins with theoretical design, where you have to define what the circuit is ought to do. This phase includes considering various solutions for the given problem. Numeric simulators, such as Matlab are usually used as help.

Once you are sure that the idea works and is worth realizing, you can start making a description of the circuit in a high-level hardware description language, usually VHDL. Any text editor, such as emacs, is good for this phase.

In order to make sure that the VHDL description really defines the wanted functionality correctly, the code needs to be simulated. You can use Mentor Graphics' Modelsim, for example. It can be used to simulate circuits described in VHDL, VERILOG, or the hybrid of these two.

When you are finally satisfied with the functional description, it is time to proceed to synthesis. Synopsys Design Compiler is used as a logic synthesis tool. The VHDL description of the circuit is entered into Design Compiler, together with design constraints for speed and operating conditions. Speed is determined by the desired clock frequency, while operating conditions (i.e. supply voltage and temperature) represent the extreme process corners at which you want your circuit to still operate correctly. As a result, Design Compiler will return a gate-level netlist

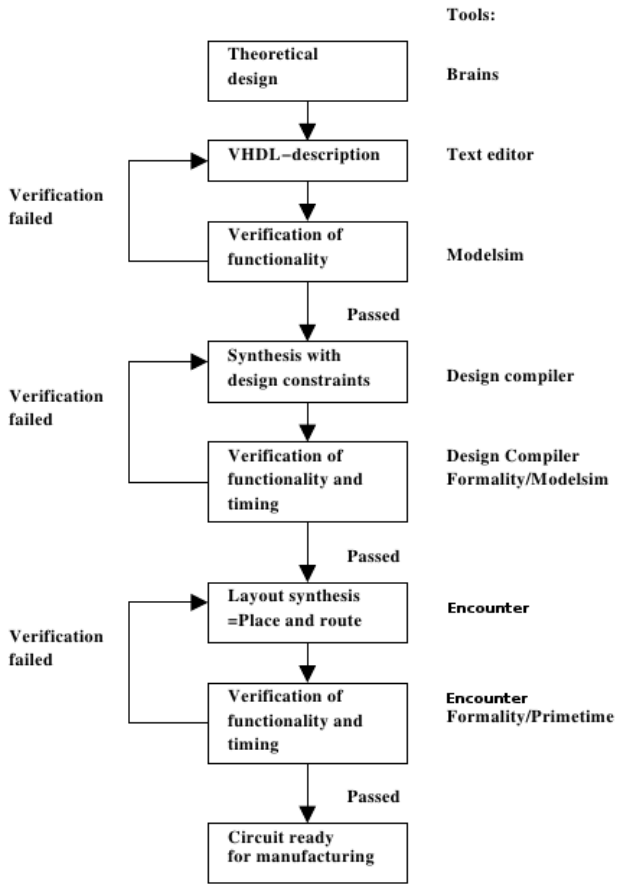

Figure 1.1: A typical digital design flow.

of the circuit, using the logic library of the process vendor (circuit manufacturer). The gate-level netlist just tells how to connect different standard cells to realize the desired functionality, such that design constraints are met.

After synthesis, you should make sure that the result matches with the original description. Synopsys Formality is used for static verification of functionality. It compares two different logic descriptions (the VHDL that you originally wrote, and the netlist produced by Design Compiler) and reports an error if their functionalities differ from each other. Static verification is considered better than comparison based on simulations, since the result of a simulation depends on the input, whereas static verification ensures that the two descriptions are logically equivalent. Static verification is also significantly faster than simulation.

After completing the verification test, it is time to move to layout generation. Cadence Encounter is used to "place and route" the standard cells of the synthesized netlist. This process consists of three main phases:

Place The standard cells found in the synthesized netlist are placed in the core layout area.

Clock tree synthesis The clock signal (which may have to drive thousands of flip-

flops) is buffered and routed in a tree-like fashion, in order to minimize the skew between different arrival points.

Route The remaining signal nets are routed, according to the connectivity defined in the synthesized netlist.

During the above process, timing is checked several times. Encounter can estimate path delays more accurately than Design Compiler, since cell placement and load of interconnecting wires are known. If timing is not met, Encounter optimizes your netlist by e.g. resizing standard cells and adding/removing buffers, in an effort to improve the overall performance of the circuit. After several iterations, this should lead to a (nearly) optimal implementation regarding speed, area, and power consumption.

Once layout generation is complete, you need to verify that the functionality of the final (optimized) netlist matches with that of the original netlist. Furthermore, you must make sure that all Design Rule Checks (DRCs) and timing constraints are met. Encounter, Formality and PrimeTime can be used to accomplish these tasks. In addition, the power consumption of your circuit can be estimated with Encounter or PrimeTime.

If the circuit passes all tests, it is ready to be sent to the circuit manufacturer. In mixed-mode designs, the generated digital layout can be taken into Cadence Virtuoso design environment, where it will be combined with the analog blocks.

1.3 Getting started

In order to access the software tools used in this tutorial, you need to connect to the VSPACE server. Therefore, make sure that you have a valid computer account. You are reminded that you are NOT allowed to transfer ANYTHING (files, software, technology information, etc.) from the VSPACE server, unless it is produced by yourself (e.g. own VHDL files).

A template directory tree can be downloaded from the MyCourses pages of the course. You just need to extract the archive to a location of your choice, and then you can start working. Please, use only the provided template to run the flow. Some of the sub-directories contain hidden setup files, that will automatically configure the software tools with the correct settings. For example, Design Compiler will not be configured correctly unless you run it from the **synthesis** sub-directory.

1.4 Technology libraries

Before you start the actual work, browse to the **techlibs** subfolder, and spend a few minutes to take a look to the technology libraries. Even though in this tutorial these files are provided ready for you, in a real design project you might have to dig them out of the design kit by yourself, as circuit manufacturers usually provide little or no instructions on how to set up the digital flow.

You will notice four different types of technology files.

.lib A Liberty library is an ASCII-readable file, that contains the complete characterization of all standard cells at one specific PVT (process, voltage, temperature) corner. The characterization includes logic function, pin capacitances, propagation delays, and leakage/dynamic power consumption. The process vendor always provides many *.lib* files, each characterized for a different PVT corner. In this tutorial, we are going to use only two corners, "slow" and "fast", representing the slowest and fastest possible operating conditions of the digital circuit.

.db These are just compiled versions of the corresponding *.lib* files, which are used by Synopsys tools (Design Compiler, Formality, PrimeTime).

.lef A Library Exchange Format file is an ASCII-readable file. There are two types of LEF files.

- The technology LEF file defines the DRC rules (width, spacing, metal density, etc.) of the given technology.
- The macro LEF contains the abstracts of the standard cells. An abstract is a simplified version of the cell layout, where only the layers that are relevant for place and route are retained.

qrcTechFile A QRC file contains the necessary technology information to extract the RC interconnect parasitics of a digital layout. This includes data such as metal resistivity, coupling capacitance between adjacent metal layers, and so on. The QRC file is binary, and thus not readable with a text editor. Usually the process vendor provides many QRC files, characterized for different extraction corners (for example, linewidth smaller than nominal will lead to increased wire resistance but also decreased coupling capacitance). In this tutorial, we are going to use only one extraction corner.

Feel free to open the *.lib* and *.lef* files with a text editor, and browse their contents, if you feel like. Can you find the supply voltage and temperature, that have been used to characterize the "slow" and "fast" PVT corners?

2 Synthesis

We are now ready to start with the actual digital flow. At this point, it is assumed that you have a ready VHDL code describing the functionality of your digital circuit, and you have verified its correct operation through extensive simulations. The first stage of the digital flow consists of *synthesizing* the VHDL description into a gate-level netlist, where the gates are selected from the logic library provided by the circuit manufacturer. In other words, we are converting a high-level description, written in a human-friendly programming language, into a low-level description, consisting of a circuit of interconnected logic gates. In order to generate the netlist correctly, the synthesis tool also needs some additional information, like the desired clock frequency, the capacitive load at output ports, and so on. This information is collectively referred to as *design constraints*.

The synthesis tool that we are going to learn is Synopsys Design Compiler (DC).

2.1 Starting the software

1. With the console, go to the **synthesis** subfolder of the template directory tree.
2. Take the latest version of DC into use by writing to the console:
   `use syn_2015.06-SP4`.
3. Now start DC's GUI (called Design Vision) by typing: `design_vision`.
4. Take a minute to familiarize yourself with the interface (Figure 2.1). The "Hier.1" window allows you to browse the design hierarchy (now empty). The `Log` tab is located at the bottom of the main window (you might need to stretch it upwards, to make it visible). Under the `Log` tab, there is a textbox where you can enter commands to the software.

2.2 Analysis

The first step consists of importing all VHDL files related to the design to be synthesized into DC.

1. Select **File** –> **Analyze...**

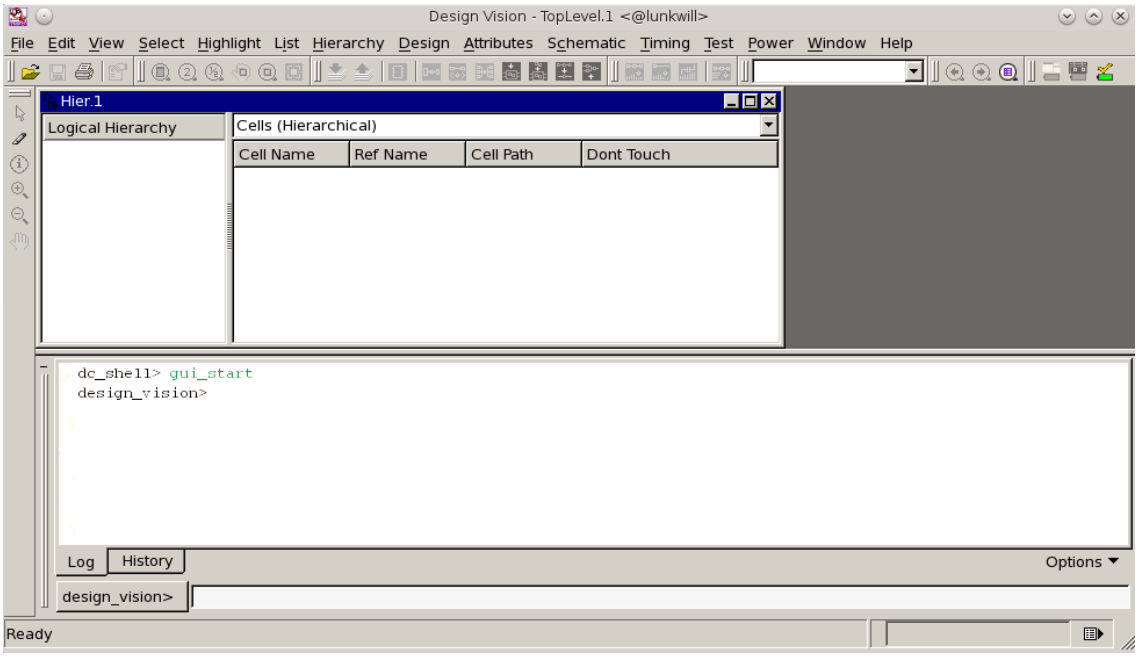

Figure 2.1: The main window of Design Vision GUI.

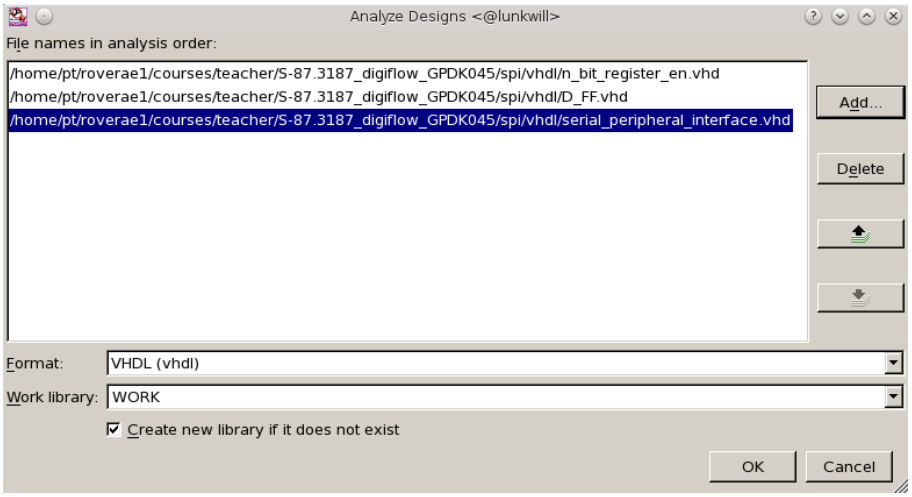

Figure 2.2: Analyze Designs window.

2. From the appearing window choose Add... to add VHDL files. You can change the file order by using the arrow buttons. The file with the top-level entity should be the last one to be analyzed.

3. Change Format to VHDL, and check the "Create new library if it does not exist" box.

4. Your window should now look like in Figure 2.2.

5. Press OK to analyze the files. Check from the log that no error messages occur.

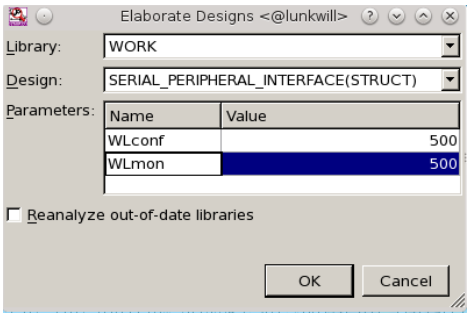

Figure 2.3: Elaborate Designs window.

2.3 Elaboration

Next in line is elaboration, where each VHDL construct found in your code is translated into its equivalent logic circuit. The logic gates used during elaboration are chosen from DC's generic library, *not* from the process vendor's standard cell library. Moreover, elaboration does not perform any optimization on the resulting logic circuit. In other words, it is just a 1:1 mapping between VHDL and logic gates.

1. Select **File** –> **Elaborate...** The window shown in Figure 2.3 appears.
2. Check that Library is the same where you analyzed your design (WORK is the preset). Also check that Design is the same as your top-level block (SERIAL_PERIPHERAL_INTERFACE). In the Parameters box, set values for the generics of your top-level entity, if any (WLconf = WLmon = 500 for the SPI).
3. Click OK. If elaboration fails, and error window will pop up.
4. Check anyway the log and search if there are any error or warning messages. For the best synthesis result, elaboration should not produce any warnings. Therefore, you are strongly advised to modify your VHDL code accordingly.

After elaboration completes successfully, the logical hierarchy of your design should appear in the "Hier.1" window. If you feel like, you may also browse the schematic of the elaborated logic circuit, by selecting the top-level design from the list and then **Schematic** –> **New Schematic View**.

2.4 Constraint definitions

After elaboration, you need to specify the environment in which your design will operate, the extreme operating conditions, and some constraints that will drive the optimization later on. This is perhaps the most important phase of the synthesis process.

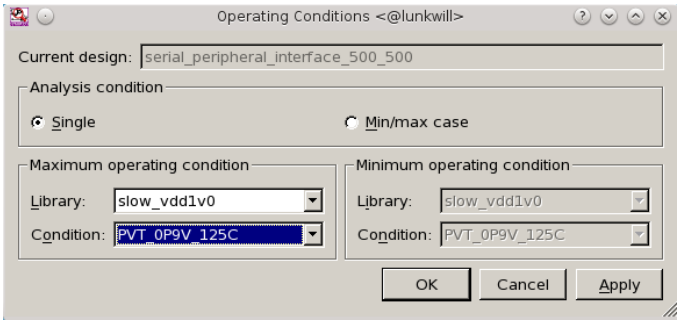

Figure 2.4: Operating Conditions window.

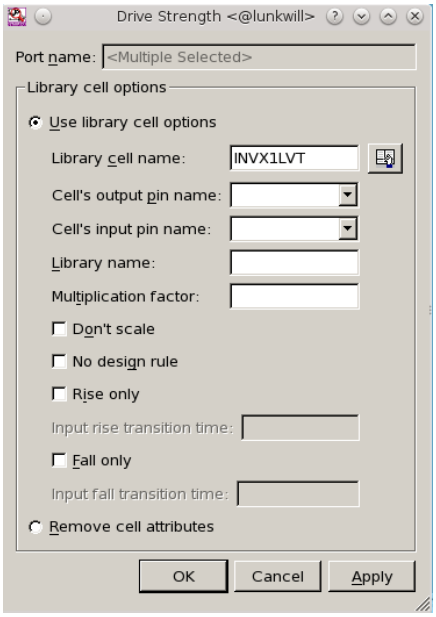

Figure 2.5: Drive Strength window.

1. Go to **Attributes** –> **Operating Environment** –> **Operating Conditions...** The window of Figure 2.4 appears.

2. Ensure that Analysis condition is "Single", meaning that you will check the operation of your design only at a single PVT corner. This is usually enough for synthesis. Later on, during layout generation, you typically check your design at multiple corners.

3. Library should be "slow_vdd1v0", and Condition should be "PVT_0P9V_125C". If your design works properly at the slowest possible PVT corner, then it is also fast enough to operate at any other corner. Press OK.

4. Now choose the logic gate that will drive each input port of your design. First select all input ports with **Select** –> **Ports/Pins** –> **Input Ports** (you can check that the ports have been selected, by changing the view to Pins/Ports in the "Hier.1" window). Then go to **Attributes** –> **Operating Environment**

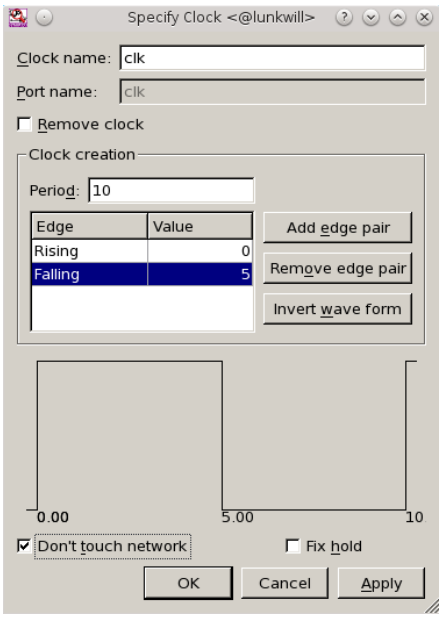

Figure 2.6: Specify Clock window.

–> **Drive Strength**. Select "Use library cell options" and type INVX1LVT in the Library cell name field, as shown in Figure 2.5. This way, you are specifying that your input ports will be driven by the smallest inverter provided by the process vendor. Click OK.

5. Next, set the capacitive load that has to be driven by each output port of your design. This has to be done in command mode. Enter the following command to the console: `set_load 0.001 [all_outputs]`. This means that you are setting a load of 0.001 pF (i.e. 1 fF) to each output port.

6. Now it is time to specify the clock frequency at which your design has to operate. Select the clk signal from the "Hier.1" window, and then **Attributes –> SpecifyClock...**

7. Specify a name for your clock (can be the same as the port name). Specify a period of 10 (unit is always ns) for a clock frequency of 100 MHz. Specify the rising and falling edges to occur at 0 and 5 ns respectively. In the end, check "Don't touch network". This means that the clock network will be kept as ideal during synthesis. The clock tree will be created during layout generation.

8. The window should now look like in Figure 2.6. Press OK.

9. Now specify some properties of your clock. Enter the following commands:

   - `set_clock_uncertainty -setup 1 clk`
   - `set_clock_transition 0.1 clk`

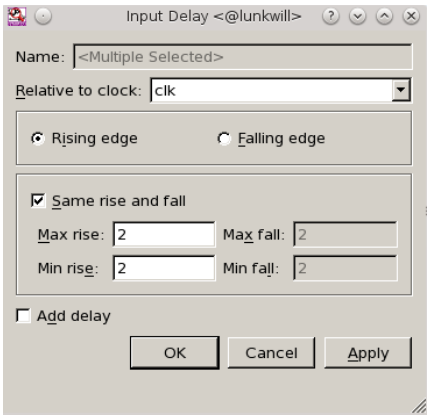

Figure 2.7: Input Delay window.

With the first command, you are specifying that all setup checks relative to "clk" must take into account 1 ns of clock skew (10% of the clock period is usually good). This gives you some safety margin, which will be accounted for throughout the digital flow. With the second command, you are specifying that the transition time of "clk" at all nodes of the clock network shall be at most 0.1 ns. This is quite meaningless at this point (since the clock network is ideal), but it will drive the generation of the clock tree during the layout phase.

10. After specifying the clock signal, you must state how much time it will take for input data to be ready at each synchronous input port, after a clock rising edge. Select all input ports (like in point 4) and then deselect all clock and asynchronous ports of your design ("clk" and "rst" for the SPI). Then go to **Attributes** –> **Operating Environment** –> **Input Delay...**

11. Specify a delay of e.g. 2 ns for both Max rise and Min rise, as shown in Figure 2.7. Press OK.

12. Similarly, you must also state how much *before the next clock rising edge* data has to be ready at each synchronous output port. Change the selection with **Select** –> **Ports/Pins** –> **Output Ports**, then go to **Attributes** –> **Operating Environment** –> **Output Delay...** and set a margin of e.g. 2 ns, like you did for the input delay (the form looks exactly the same).

13. In the end, direct DC to optimize your design netlist with respect to area consumption. Enter the following command: `set_max_area 0`. This way, you target a total area consumption of 0 (i.e. the smallest possible area).

2.5 Compilation

After all design constraints are defined, we can finally move to compilation. During this phase, the generic logic components selected during elaboration are mapped

to standard cells from the process vendor's library. Moreover, the resulting gate-level netlist will be optimized with respect to all the constraints defined in Section 2.4. Just as an example, if your VHDL code defines an addition "+", then one particular adder architecture (e.g. ripple carry, carry select, etc.) will be chosen for implementation, such that the addition can be computed in the given clock period.

1. Before starting the compilation, flatten your design hierarchy by selecting **Hierarchy −> Ungroup...** From the "Ungroup" window, choose "All" and "Ungroup all levels". Press OK.

2. You will notice that the design hierarchy has disappeared from the "Hier.1" window, except the top-level design. Removing the hierarchy boundaries makes netlist optimization more efficient.

3. Start compilation/optimization by going to **Design −> Compile Ultra...** and pressing OK. Completing this task will take some time, depending on your design size and the feasibility of the constraints defined earlier. You can follow the progress in the Log tab.

2.6 Saving the results

We are almost done with synthesis. In order to proceed with the digital flow, we first need to export the data produced by DC.

1. Before exporting the Verilog netlist, you need to make sure that all instance and net names in your design conform to Verilog naming rules. Hence, type to the console `change_names -rules verilog -hierarchy`.

2. Now export the Verilog netlist by selecting **File −> Save As...** Choose Format to be VERILOG (v) and save the netlist as **serial_peripheral_interface.v** to the **netlist** subfolder.

3. You should also export the design constraints entered in Section 2.4. You can easily do that with the command `write_sdc ../timing/prelayout.sdc`. This will save a Synopsys Design Constraints (SDC) script in the **timing** subdirectory.

Check the two files that you have just exported with a text editor. In the Verilog netlist, you should see the list of standard cells that build up your top-level design, as well as their connections. In the SDC script, you should be able to find the commands corresponding to all design constraints defined earlier.

2.7 Reports

Design reports provide some essential information on the synthesis outcome. You can, for example, get first estimates of area and power consumption of your circuit, or check whether the design constraints were met.

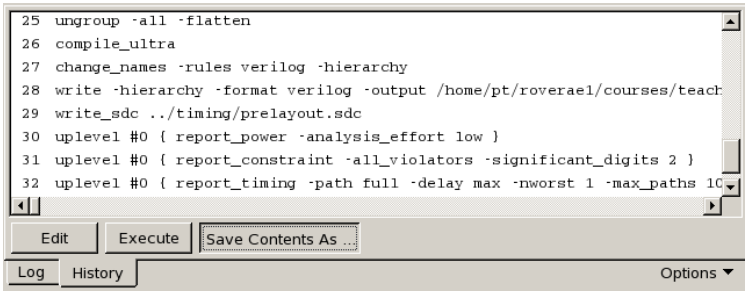

Figure 2.8: History tab.

1. Check the area estimate with **Design −> Report Area...** and press OK. Note that the report can be written to a file if needed.

2. Similarly, check the power consumption estimate with **Design −> Report Power...** and press OK.

3. Check if any design constraint is violated. Go to **Design −> Report Constraints...**, check "Show all violators" and click OK. You should see no violators, except the power targets (which will obviously be higher than 0).

4. Check the critical paths in your design with **Timing −> Report Timing Path...** Change Max paths per group to e.g. 10 and press OK. The timing report will then show the detailed setup check of the 10 worst paths of your design. You will see information such as path startpoint and endpoint, propagation delays of the cells along the path, setup time of the receiving flip-flop, clock skew, and so on.

Please take some time to get acquainted with these reports. They are really important. Especially the timing report: you must be able to identify the critical path of your own design, with the information provided by the report. In digital design, you are not going to run transient simulations, as they would take way too long time. Therefore, you must be able to dig out of the reports everything that you want to know about your design. For example, in case you see some timing violations in the constraints report, you can use the timing report to identify why and where they occur, and then take corrective measures.

2.8 Creating a TCL script

In order to ease the synthesis procedure for the future, it is a good idea to create a script with the commands run so far. Using the GUI is way too slow to re-synthesize the design after small changes. When the script is created, it can be invoked by starting DC with `design_vision -f synthesis_script.tcl`, or from **File −> Execute Script...** This way, the whole synthesis procedure described in this Chapter can be run automatically.

1. Switch from Log tab to History tab in the lower part of the main window, as shown in Figure 2.8. You will see the complete list of commands that have been executed since the software was started.

2. Save the commands to a TCL file called **synthesis_script.tcl**.

When you are finished, make sure you exit DC by selecting **File** –> **Exit** or by typing `exit` to the console.

3 Static verification (synthesis)

Synopsys Formality checks that the synthesized netlist is logically equivalent to the original VHDL code. Formality itself is a very simple program to use, as it only requires the user to specify the reference code, the synthesized code, and the libraries used to process the design.

Sometimes, DC optimizes the netlist so heavily, that static verification will fail, even though logic functionality is in fact correct. When this happens, the only solution is to verify the functionality by simulation (e.g. simulating the Verilog netlist in Modelsim). Nevertheless, this should not be your case, as your design is simple enough, and you have not used any special optimization features during synthesis. If you get errors during this stage, they most likely indicate that your VHDL coding style is poor. You should modify your original VHDL code, and synthesize it again. The bad news here, is that Formality errors might be cumbersome to interpret.

1. Take Formality into use by moving to the **formality** subfolder, and writing `use formality_2015.06-SP4`.

2. Start the software by typing `formality`. The GUI appears.

3. You will now start loading data to Formality. During each of the following steps, after picking up the appropriate files, you need to press the Load Files button at the center of the main window, in order to complete the loading process.

4. First, press the Guidance... button, browse to the **synthesis** subfolder, and select a file called **default.svf**. The SVF file keeps a record of the changes (e.g. optimizations, etc.) that DC makes during synthesis, with the purpose of guiding Formality during verification. Please note that this file is reset every time you start DC. Therefore, make sure that you do not open DC again, unless you want to re-run the whole synthesis flow.

5. Now load the reference code. Move to 1. Reference tab, choose the VHDL subtab, and press the VHDL... button. Find the VHDL files of your design, and load them to Formality.

6. After loading, move to 3. Set Top Design subtab, select the top-level design from the list (e.g. serial_peripheral_interface), and press Set Top.

7. Now move to 2. Implementation tab, and press the Verilog... button to select the synthesized netlist.

8. Load the reference library by moving to the Read DB Libraries subtab. Press DB... and choose the file **slow.db**, located in the **techlibs** sub-directory. This is the same library that has been used by DC. However, there you did not need to load the library explicitly, since it was configured automatically by a hidden setup file located in the **synthesis** subfolder.

9. Then move to 3. Set Top Design subtab, and set again the top-level design.

10. Move to 4. Match tab, and click Run Matching. If all points are matched, everything is fine. Otherwise, it is possible that you will not pass the static verification. Keep your fingers crossed, and move to next phase.

11. Last, switch to 5. Verify, and click Verify to run static verification. A popup window should appear soon, stating whether verification succeeded or failed.

Next, you can collect the commands just executed to a TCL script, like with DC. Select the History tab at the bottom of the main window, copy-paste the commands to a text editor, and save the file as **verify_synthesis.tcl**. In the future, you can re-run static verification by simply invoking this script, rather than using the GUI.

Close Formality and move to layout generation.

4 Place and route

After we have a verified Verilog netlist, we can use Cadence Encounter to generate the physical layout of our circuit. Encounter is a rich and complex software tool, so in this course we are going to learn only a small subset of its functionality. Like Synopsys tools, Encounter also uses TCL as a control language, thus every function accessible through the GUI corresponds to a text command. However, in contrast to Synopsys tools, the GUI of Encounter does not itself embed a console. Instead, the console is located in the same terminal from where you start the software.

Encounter produces a huge amount of information in the console, including warning and error messages. While errors are usually not acceptable, many warnings can be safely ignored. Therefore, you need to learn which warnings are relevant and which are not. When running the flow for the SPI, try to memorize the warnings seen in the console. You can expect to see the same warnings when processing your own design. If you see additional warnings, you should then try to understand why they occur, and what can you do to get rid of them.

The place and route flow presented in this Chapter is based on Cadence's recommended Foundation Flow for flat designs.

4.1 Starting the software

1. Move to the **layout** subfolder, and type `use encounter_14.2` to take Encounter into use.

2. Start the software by writing `encounter`. The GUI appears. For now, it looks quite empty.

4.2 Importing the design

1. Choose **File** −> **Import Design...** The "Design Import" form appears.

2. Browse to the synthesized Verilog netlist in the topmost textbox. For the Top Cell option, select "Auto Assign".

3. As Technology/Physical Libraries, select "LEF Files", browse to the **techlibs** subfolder, and pick the two LEF files. Make sure that **gsclib045_tech.lef** is first in the list, otherwise Encounter will return an error!

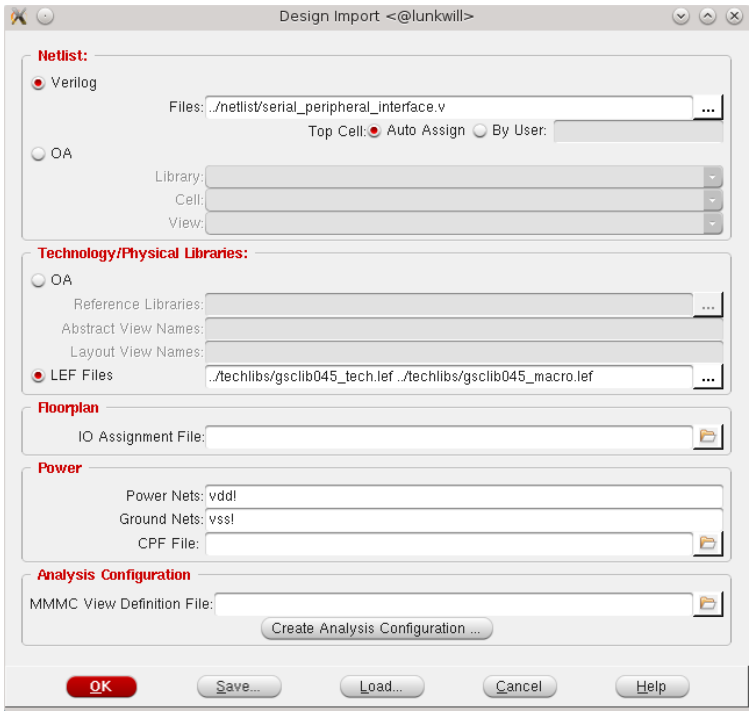

Figure 4.1: Design Import window.

4. In the Power Nets and Ground Nets fields, type "vdd!" and "vss!" respectively.

5. Your window should now look like in Figure 4.1.

6. Next, you need to create a view definition file for the Multi-Mode Multi-Corner (MMMC) analysis. Click the Create Analysis Configuration... button to open the "MMMC Browser" form. The procedure is explained in the next Subsection.

4.2.1 Creating a MMMC view

A MMMC view specifies all necessary information to configure Encounter for RC extraction, delay calculation, and timing analysis. The view needs to be created only once for each design. After that, it can be reused every time you run the layout generation with Encounter for the same design.

1. Start with specifying the *.lib* files used for timing analysis. For a basic flow, you will normally configure at least two library sets: one for the best-case PVT corner, and one for the worst-case PVT corner.

   - Right click on Library Sets, and select **New...** The "Add Library Set" window pops up.
   - Type a meaningful identifier in the Name field, like "slow_libs". Then add the **slow.lib** library (**techlibs** subfolder) to the Timing Library Files list. Press OK.

- Repeat for the **fast.lib** file, in a "fast_libs" library set.

2. Next, you need to define at least one RC corner. This provides the necessary data to drive the RC extractors used to compute parasitic resistances and capacitances.

   - Right click on RC Corners, and select **New...**
   - Give a name (e.g. "rc_basic"), then go to the QRC Technology File field, and browse to **qrcTechFile** in the **techlibs** subfolder.
   - You don't need to enter any other information, so just press OK.

3. Now that you have specified all the necessary technology files, let's move one level up in the MMMC hierarchy. Here, you will define the delay corners. Each delay corner specifies the complete set of library data to perform extraction, delay calculation, and timing analysis. The delay corner is composed of linkages to previously configured library sets and RC corners.

   - Right click on Delay Corners, and select **New...**
   - Give a name to the slow delay corner, e.g. "slow_corner".
   - Make sure that Type is set to "Single/BcWc". This way, the software will perform setup and hold checks on the paths in your design.
   - Select the only RC corner available, and "slow_libs" as the library set. Click OK.
   - Repeat the procedure for the "fast_libs" library set, in a "fast_corner" delay corner.
   - At this point, defining delay corners might seem quite useless to you. However, in case the circuit manufacturer provides many RC corners (QRC files), it might be interesting to create multiple delay corners, by associating different combinations of library sets and RC corners. Be aware that extraction corners are completely independent from PVT corners, because they model the variations of interconnect parameters (resistance and capacitance) during the manufacturing process.

4. So far, you have only entered information about the manufacturing process. With constraint modes, you can specify design specific constraints. Defining multiple constraint modes is useful, for example, for designs featuring different functional and test modes. Because this is not your case, you are going to define only one constraint mode.

   - Right click on Constraint Modes, and select **New...**
   - Give a name, for example "constraint_basic".
   - Under SDC Constraint Files, add the SDC file that you exported from DC. Press OK.

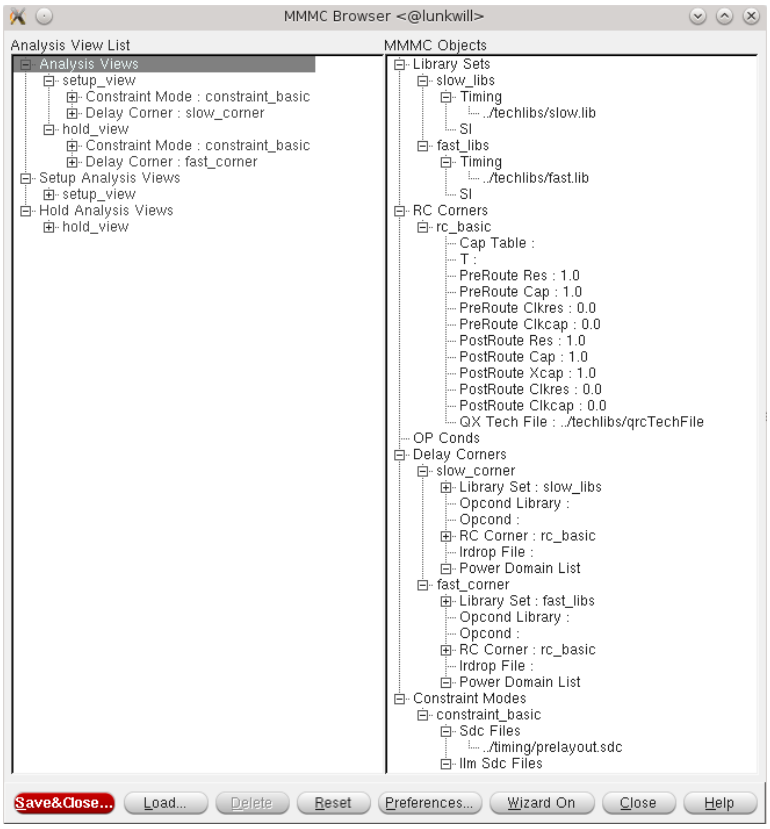

Figure 4.2: MMMC Browser window.

5. We are almost done. The last step is to combine delay and library information from delay corners with the functional information from constraint modes, in order to create the top-level analysis views. Each view is an independent timing analysis that is analyzed concurrently. In addition, you will also specify which analysis views should be considered for setup checks (max path analysis), and which for hold checks (min path analysis).

   - Right click on Analysis Views, and select **New...**
   - Give a name to the max path analysis view (e.g. "setup_view"), select the only constraint mode available, and "slow_corner" as the delay corner. Press OK.
   - Repeat for the "fast_corner" delay corner, in a "hold_view" view for min path analysis.
   - Now right click on Setup Analysis Views, select **New...**, and in the form that appears choose "setup_view".
   - Similarly, choose "hold_view" for Hold Analysis Views.

6. At this point, if you expand the tree items in the "MMMC Browser" window, the configuration should look like in Figure 4.2.

7. Press Save&Close... to save the view definition as a TCL script.

8. After that, the path to the view file just created should appear in the MMMC View Definition File field of the "Design Import" window.

9. Finally, click OK to start the design import process. You can follow the progress from the console. After a few seconds, the initial floorplan should appear in the main window (press key "f" to zoom full).

If you open the MMMC view definition script with a text editor, you should be able to recognize the commands corresponding to your selections in the "MMMC Browser" GUI. The syntax is quite self-explanatory. In the future, if you need to do small modifications to the MMMC view (for example, to adapt it to a different design), you may simply modify the script, instead of going through the tedious GUI procedure again.

4.3 Floorplan

Floorplanning includes a variety of tasks, such as defining the layout area and shape, placing the netlist modules in the core area, fixing the I/O pins positions, defining placement/routing blockages, and so on. Because your netlist contains only the top-level module (recall that you have flattened the hierarchy in DC), floorplanning is going to be really basic.

1. Select **Floorplan –> Specify Floorplan**.

2. Change all four core margins to 15, in order to leave some room for the power rings.

3. Note that Core Utilization is about 0.7 by default. This means that the core area is automatically calculated, such that only 70% of it will be occupied by standard cells. This is a good starting value.

4. Press OK. Now you should see some margins between "die" and "core" boundaries.

5. Now start placing the I/O pins. Open the "Pin Editor" window from **Edit –> Pin Editor...**

6. Spread all 500 pins of "config_port" to the center of the top die side, on the M4 metal layer, at regular 0.2 μm intervals. Select "config_port [ ]" from the list on the left, and enter the settings as shown in Figure 4.3. Click Apply. A message `Successfully spread [500] pins` should appear in the console.

7. Similarly, spread the remaining pins with the settings shown in Table 4.1.

8. Close the Pin Editor. In the layout window, you can now check that the pins have been placed correctly. If necessary, you can use the "ruler" tool as help.

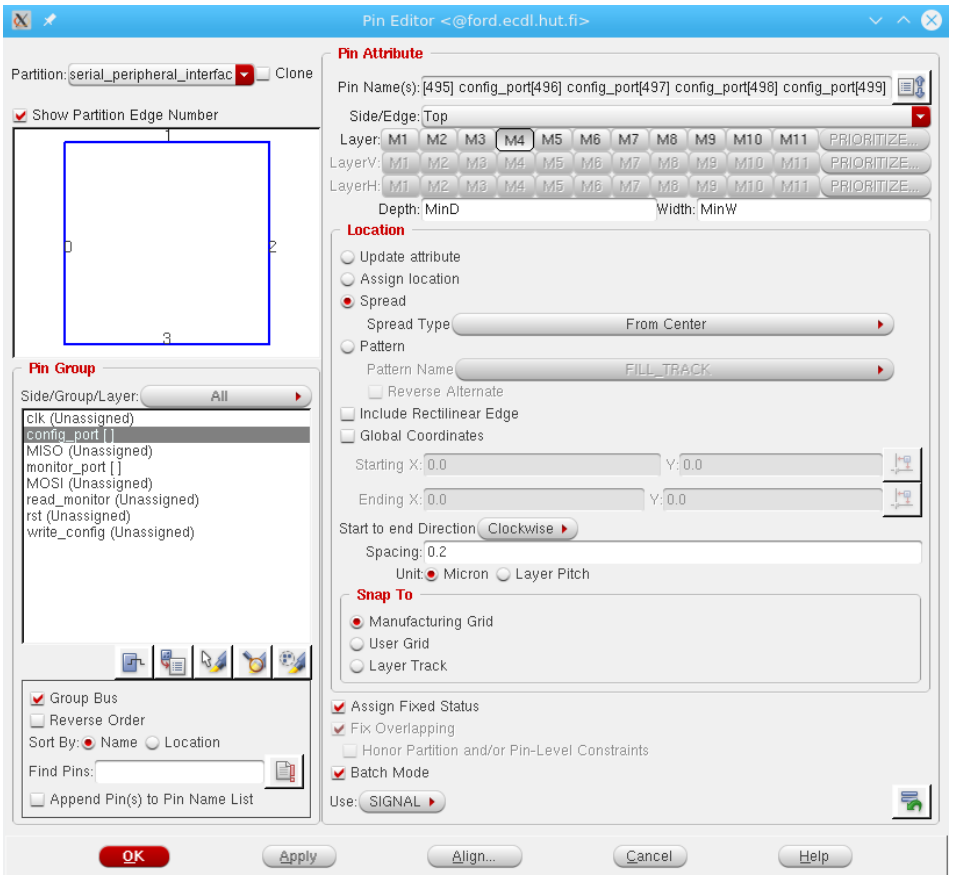

Figure 4.3: Pin Editor window.

Port names	Side	Layer	Direction	Spacing
config_port	Top	M4	Clockwise	0.2
monitor_port	Bottom	M4	Counterclockwise	0.2
clk, read_monitor, rst, write_config	Left	M3	Clockwise	10
MISO, MOSI	Right	M3	Clockwise	10

Table 4.1: Parameters to place the I/O pins of the SPI interface.

Note that we have explicitly placed the pins on different metal layers. We did so just to ensure that the router will be able to reach the pins by using the *preferred routing direction*. This is a general property of digital design kits. In our 45nm kit, all odd metal layers (M1, M3, M5...) are preferably routed with horizontal direction, whereas all even layers (M2, M4, M6...) with vertical direction.

4.4 Power planning

Power planning involves creating metal rings and stripes on the floorplan, as well as building a power grid to bring the supply connections to all standard cells. This is a critical phase of the layout generation process, which deserves special care. If the

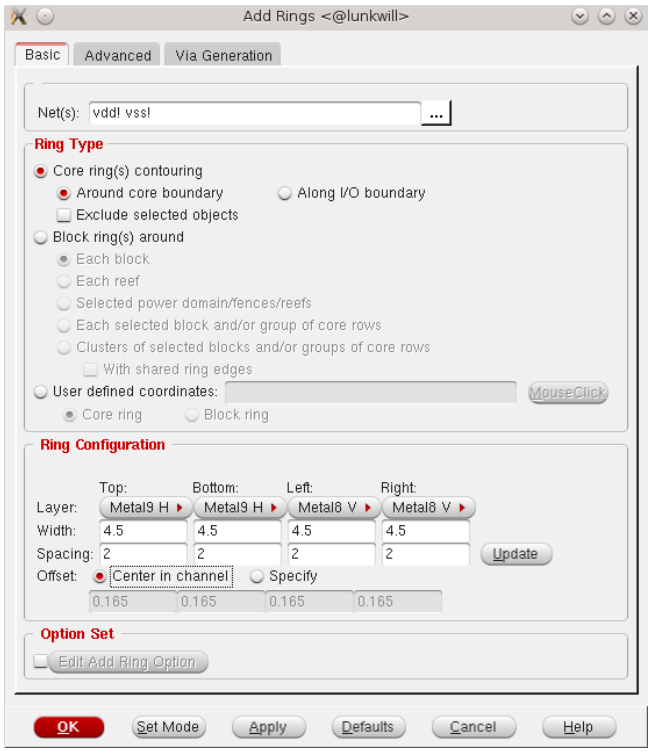

Figure 4.4: Add Rings window.

power connections are too weak, the effective supply voltage over the standard cells will drop, thus causing decreased performance or sometimes even functional failure.

1. Go to **Power** –> **Power Planning** –> **Add Ring...** to add power rings around the core.

2. Enter the settings shown in Figure 4.4 and press OK.

3. Now go to **Power** –> **Power Planning** –> **Add Stripe...** to add power stripes over the core area.

4. Add three sets of vertical power stripes on layer M8, by entering settings like in Figure 4.5. Press OK.

5. Now define the connectivity between VDD and VSS pins of the standard cells and global power nets, from **Power** –> **Connect Global Nets...**

6. Pin Name(s) should be VDD or VSS (capital case) and To Global Net should be "vdd!" or "vss!" respectively. Check Figure 4.6 for an example. Click Add to List for both supplies, then press Apply and close the window.

7. Finally, route the power grid by selecting **Route** –> **Special Route...** The "SRoute" form appears.

8. Enter "vdd! vss!" in the Net(s) field, and make sure that only "Follow Pins" is checked in the SRoute frame. Press OK.

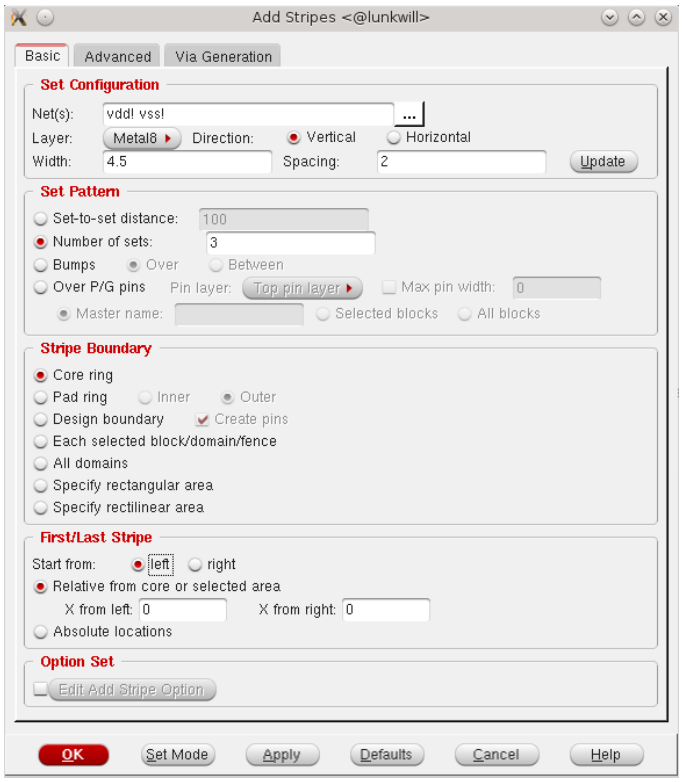

Figure 4.5: Add Stripes window.

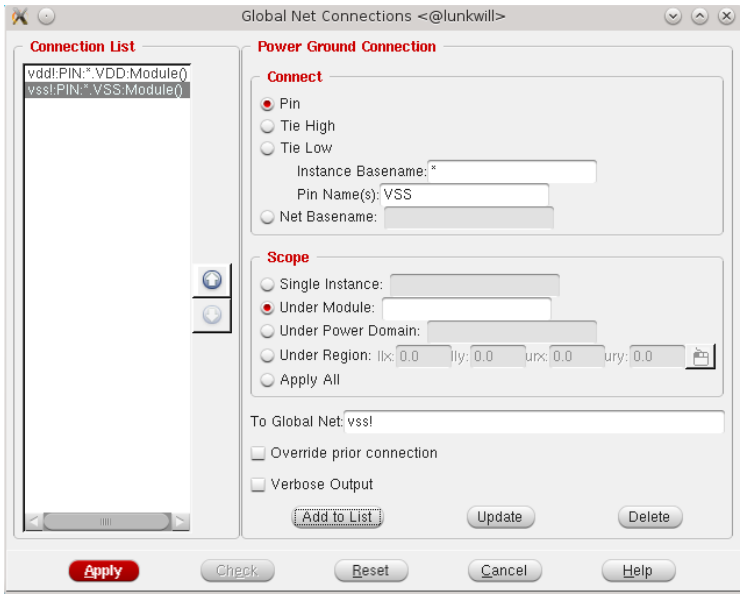

Figure 4.6: Global Net Connections window.

4.5 Place

Now that the floorplan is completely defined, we can move to the first "main" phase of the layout generation flow: placing the standard cells in the core layout area.

Figure 4.7: View buttons: Floorplan, Amoeba, Physical.

Thanks to the automated placer integrated in Encounter, this task should be simple and quick to perform.

1. Before placing the standard cells, enter the following text commands to the console:

   - `setDesignMode -process 45`
   - `setMaxRouteLayer 6`

   The first command tells Encounter to use the default global settings for 45nm process node throughout the rest of the flow. The second command states that only metal layers from M1 to M6 (i.e. those with the smallest pitch) can be used for signal routing.

2. Now place the standard cells, by selecting **Place –> Place Standard Cell...** Check both "Pre-Place" and "In-Place" optimization, and press OK.

3. Placement might take some time. You can follow the progress from the console. During this process, Encounter also performs a task known as *trial route*. This consists of a temporary and quick routing of the signal nets, with the goal of identifying possible routing congestion areas, as well as providing a first estimation of the capacitance values.

4. After the process is complete, you should see the standard cells as small blue boxes with different width, placed along the core rows (you might need to switch to Physical view by pressing the corresponding button on the top-right corner of the main window, see Figure 4.7). You should also see the signal nets routed by the *trial route* engine.

4.6 Clock tree synthesis

So far, we have considered the clock signal to be ideal. However, in practice, the clock needs extensive buffering, because it has to drive a huge number of flip-flops that are spread all around the chip. Furthermore, the clock skew between adjacent flip-flops should be generally minimized. These requirements lead to a tree-like clock distribution network. The automated synthesis and routing of this network is referred to as Clock Tree Synthesis (CTS).

4.6.1 Pre-CTS optimization

Before moving to the actual CTS, a first optimization round can be performed on the placed layout. Optimization improves the timing of your circuit and corrects

Figure 4.8: Optimization window.

design rule violations, by using techniques such as adding/removing buffers on the signal nets, restructuring the netlist, resizing gates, moving instances, and so on.

1. Go to **Optimize –> Optimize Design...**
2. Apply the settings shown in Figure 4.8, and press OK.
3. When optimization completes, you should see (in the console) a summary table, from which you can get the following information:
   - Worst Negative Slack (WNS), i.e. the setup check margin of the slowest path in your design. A negative value here means that the setup check is violated.
   - Total Negative Slack (TNS), i.e. the sum of the negative slacks from all paths that violate the setup check. A value of 0 means that there are no setup check violations.
   - The total number of timing paths checked, and the number of violating paths.
   - The number of design rule violations.

4.6.2 CTS

1. Before routing any nets, go to **Options –> Set Mode –> Mode Setup...** Select "NanoRoute" from List of Modes, switch to DFM subtab (Design For Manufacturing), and set Concurrent Via Optimization to be "Medium". This settings specifies that, during routing, multi-cut vias (i.e. via arrays with at least two cuts) have to be used instead of simple single-cut vias, whenever possible. Using multi-cut vias improves the manufacturing yield, especially in deep sub-micron process nodes.
2. Starting from Encounter 14.2, the default engine used to synthesize clock trees is "CCOpt" (Clock Concurrent Optimization). Since this engine is quite advanced and can be configured only from command line, let's switch back to the older setting with the command `setCTSMode -engine ck`.

3. Now start CTS with **Clock −> Synthesize Clock Tree...**

4. Press Gen Spec... to generate a clock tree specification file from the SDC constraints that you entered earlier.

5. In the "Generate Clock Spec" window, you only need to specify manually which buffers are to be used in the clock tree. Select all cells whose names start with CLK, click Add, and then OK to generate the clock tree specification file.

6. Press OK also in the "Synthesize Clock Tree" window to start CTS.

7. You should see quite much progress in the console, as CTS proceeds. Some of the information can be really cryptic to understand. Anyway, try to extract the following data (you should find them around the end of the log, after CTS has terminated): number of levels in the clock tree, buffers/inverters used at each level, total number of sinks (i.e. flip-flops to be driven), maximum phase delay (i.e. clock latency), maximum skew and transition time of the clock signal.

8. Encounter provides also more sophisticated tools for clock tree analysis. You can find them from **Clock −> Browse/Debug/Display Clock Tree...** Feel free to check them out, if you want.

4.6.3 Post-CTS optimization

Now we are ready for two more optimization rounds. The first one is for setup checks, like in pre-CTS optimization. Moreover, now that the clock tree has been built, we must ensure that the hold check is met as well. The second optimization round takes care of that.

1. Perform post-CTS setup optimization, by following the same procedure described in Subsection 4.6.1, except that now Design Stage should be set to "Post-CTS".

2. Similarly, perform post-CTS hold optimization. You just need to run optimization again, but this time uncheck "Setup" and check "Hold" in the "Optimization" window.

3. If you scroll the log, you should be able to find what cells have been added to fix/reduce hold violations (if any).

4.7 Route

After placement and CTS, the last "main" task of the layout generation flow is to route the remaining signal nets.

1. Go to **Route −> NanoRoute −> Route...** The "NanoRoute" window appears.

2. Set the antenna diode standard cell, that can be used to prevent antenna violations (you may google "antenna effect" to find out more about this topic). You just need to check "Insert Diodes", and type ANTENNALVT (capital case) under Diode Cell Name.

3. Press OK to start the *NanoRoute* engine. Follow the progress from the console.

4.8 Post-route

Our digital layout now looks pretty much ready. However, we still need to do some final refinements.

4.8.1 Tiling

Circuit manufacturers usually require that, for each metal layer, the density be within a specified range over the whole chip area. This increases manufacturing yield, as the metal distribution is more uniform. Because metal density is usually too low after routing, we need to add to our layout some "dummy" metal shapes, a process known as *tiling* or *metal filling*. While tiling is usually a nightmare for analog designers, in Encounter this task can be performed with a simple click.

1. Go to **Route** –> **Metal Fill** –> **Add...** Deselect the "Connection" checkbox under the Model Selection frame, so that only floating metal shapes are added to the layout.

2. The rest of the settings should be already configured correctly by the LEF file, so just click OK. You should soon see dummy metal rectangles appearing all over your layout.

3. After metal filling terminates, check if there are any density violations left, by selecting **Verify** –> **Verify Metal Density...** and pressing OK.

4. You will most likely see some markers appearing on the layout. Check on which layer the violations occur, by selecting **Tools** –> **Violation Browser...**

5. The reason for these remaining violations is as follows. During fabrication, the metal shapes added as "normal" tiling do not undergo Optical Proximity Correction (OPC), which is used for regular wires. Because of this, metal fill shapes must observe larger minimum width/spacing DRC rules, and therefore cannot fit to tiny spaces between regular wires. This might lead to the situation, where wires are not dense enough to meet the density rules, but at the same time they are too dense for metal fill shapes to fit between them.

6. Because of the above reason, Encounter also allows to insert OPC metal fill shapes, which observe same or similar DRC rules for width and spacing as regular wires. To perform OPC tiling, go to **Route** –> **Metal Fill** –> **Setup...**, change Fill Mode to "Fill Wire OPC" and press OK. Then go again

to **Route** –> **Metal Fill** –> **Add...**, make sure that only the metal layers with violations remain selected in the Layer Selection frame, and click OK.

7. Now if you check the metal density like in point 3, you should see no more violations.

4.8.2 Post-route optimization

The timing of your circuit has most likely been affected by routing and tiling. Therefore, one last optimization round is needed at this point.

1. Before running optimization, change the parasitic extraction settings. Go to **Options** –> **Set Mode** –> **Specify Analysis Mode...**, change Timing Mode to "On-Chip Variation", and click OK. Then go to **Options** –> **Set Mode** –> **Specify RC Extraction Mode...**, change the mode to "Post-Route", Effort Level to "High", and Extraction Type to "Coupled RC". Press OK.

2. Now run both setup and hold optimization, by following the same procedure explained in Subsections 4.6.1 and 4.6.3. The only difference is that Design Stage should be set to "Post-Route".

3. Remember to always check from the console, that all slacks are positive and no DRVs are left.

4. Because optimization might re-route a number of signal nets, some shorts or DRC violations can easily occur with the existing metal fill patterns. In order to get rid of these errors, you just need to *trim* the metal filling by selecting **Route** –> **Metal Fill** –> **Trim...** and pressing OK.

NOTE: After post-route optimization, you will most likely start to see some DRVs in the reports. This is a known problem with the current version of the flow, for which we have no solution. Therefore, you may just ignore those DRVs.

4.8.3 Fillers

While tiling can effectively take care of metals, it does nothing for the density of other physical layers, such as polysilicon and diffusion areas. Nevertheless, these layers also have their own density requirements. Therefore, we need to add *fillers* to eliminate all empty spaces in the core rows. Fillers are standard cells containing arrays of dummy transistors, provided by the circuit manufacturer specifically for this purpose.

1. Go to **Place** –> **Physical Cell** –> **Add Filler...**

2. Press the Select button next to the Cell Name(s) field, and add all the cells from the list.

3. Press OK in the "Add Filler" window. The number of filler instances added for each cell type should be reported in the console.

4.9 Signoff

Signoff consists of a series of verification steps that must pass before the design can be taped out. While the actual signoff is typically performed in Cadence Virtuoso environment (with the aid of tools such as Calibre), it is still mandatory to check the design in Encounter before moving on.

1. To check that the design has no DRC errors, go to **Verify** –> **Verify Geometry...** and press OK. Possible violations can be seen in the console.
2. Similarly, check for wiring errors by going to **Verify** –> **Verify Connectivity...**
3. Check for antenna violations, by selecting **Verify** –> **Verify Process Antenna...**
4. Finally, check once more for metal density violations with **Verify** –> **Verify Metal Density...** Just to make sure that post-route optimization and trimming did not create new violations.

4.10 Saving the results

At the end of the layout generation process, your window should look like in Figure 4.9. Now you need to export some data to be used during final design verification. Note that, since in this course we are not going to further process the layout, you don't need to export the layout itself.

1. Export the final Verilog netlist from **File** –> **Save** –> **Netlist...** Save the file as **serial_peripheral_interface_layout.v** to the **netlist** subfolder.
2. Export the post-layout design constraints, by writing the following text command to the console: `writeTimingCon -sdc ../timing/postlayout.sdc`. Check the file with a text editor. The only remarkable difference compared to **prelayout.sdc** should be one added command, `set_propagated_clock`. When performing static timing analysis with PrimeTime, this command specifies that delays must be propagated through the clock network to determine latency at register clock pins (in other words, the clock signal is not ideal anymore).
3. Finally, extract the RC parasitics of the nets by selecting **Timing** –> **Extract RC...** Check "Save SPEF to" and save the parasitic file to the **timing** subfolder as **serial_peripheral_interface.spef**.
4. You may now exit Encounter.

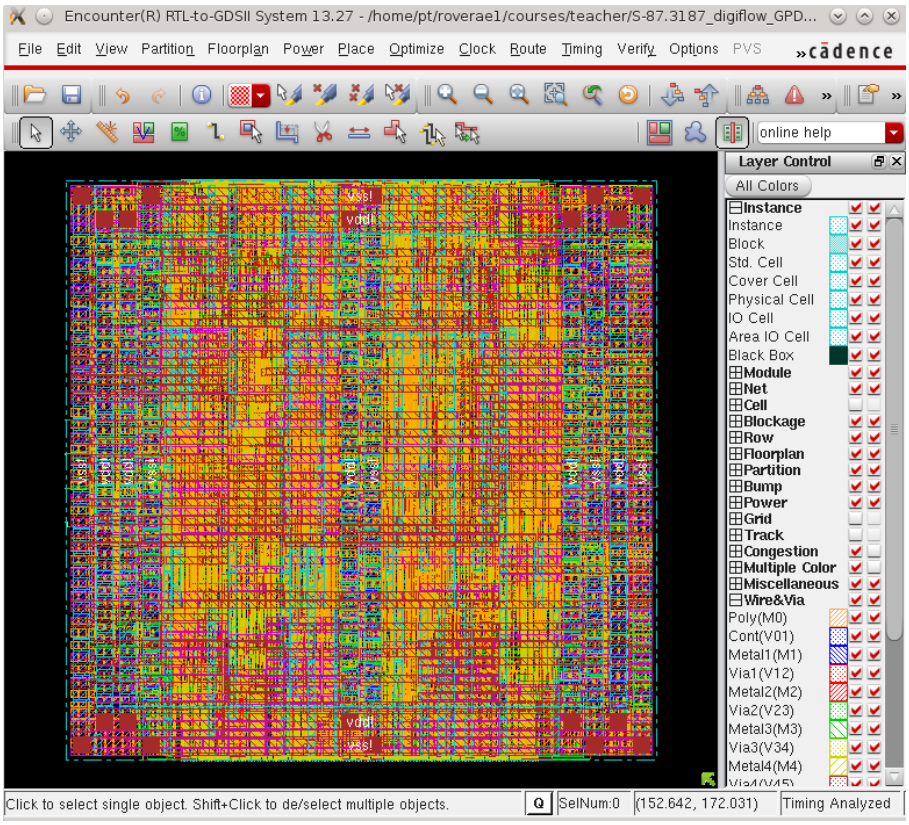

Figure 4.9: Final layout of the SPI.

4.11 Creating a TCL script

As with DC and Formality, it is convenient to create a script with the commands to run automatically the layout generation flow in Encounter. After starting the software, the script can be invoked by typing `source layout_script.tcl` to the console.

1. Go to the **layout** subfolder with the console or a file browser.

2. You should see one or more files matching the pattern **encounter.cmd***. Each of these files contains the list of commands that have been executed during one Encounter session. Some commands correspond to the flow tasks described in this Chapter, whereas other commands are GUI related (even just selecting a wire from the layout window corresponds to a text command).

3. Gather only the flow related commands to a TCL file called **layout_script.tcl**.

5 Static verification (layout)

It is a good idea to verify the logic equivalence between the netlists exported by DC and Encounter. This task is performed again with Formality. Because the netlist changes introduced by Encounter are usually minimal, this time verification should succeed without problems.

1. Start the software as in Chapter 3.
2. Move to 1. Reference tab, and load the Verilog netlist exported by DC.
3. Load the reference library by moving to the Read DB Libraries subtab. Choose the file **slow.db**, located in the **techlibs** sub-directory.
4. Then move to 3. Set Top Design subtab, and set the top-level design.
5. Now move to 2. Implementation tab, load the netlist exported by Encounter, and set the top-level design again.
6. Run matching and verification as explained in Chapter 3.
7. Before closing Formality, you may again create a TCL script to run the process automatically.

6 Static timing analysis

Static Timing Analysis (STA) is used to check the overall timing of a digital circuit. As with static functionality verification, STA is more complete than timing analysis based on simulations, since it does not depend on the input. With STA, you can easily identify the critical paths of your circuit, and take corrective measures if necessary.

We have already used STA several times during the digital flow. The timing report generated by DC after synthesis is based on STA. Encounter also uses STA, for example during optimization, in order to identify the critical paths that need fixing. Now, we are going to use Synopsys PrimeTime to run STA. The timing report will look very similar to that generated by DC, but the results will be more accurate, because they also take into account the RC parasitics extracted from the layout.

Like DC and Formality, PrimeTime also has a GUI. However, PrimeTime's GUI is so poor in its functionality, that we are not going to use it at all, and we will work with command line instead. You can start the GUI at any time from the shell by typing `start_gui`.

6.1 Starting the software

1. Go to the **primetime** sub-directory with the console.
2. Take PrimeTime into use by typing: `use primetime_2015.12`.
3. Start PrimeTime's shell by entering: `pt_shell`.

6.2 Importing the design

1. Load the Verilog netlist exported from Encounter:
   `read_verilog "../netlist/serial_peripheral_interface_layout.v"`.
2. Set the path where PrimeTime will search for library files:
   `set search_path "../techlibs/"`.
3. Set the standard cell library which models the worst-case PVT corner, to be used for max path analysis (setup checks):
   `set link_library "slow.db"`.

4. Now link the Verilog netlist to the standard cells from the logic library: `link_design`.

6.3 Design constraints

Now we need to specify the same design constraints (clock frequency, driving strength, etc.) used earlier in the digital flow. Since we have exported a SDC file from Encounter at the end of the layout generation process, it is sufficient to import the file into PrimeTime with the following command:
`source "../timing/postlayout.sdc".`

6.4 Back-annotation of RC parasitics

The Verilog netlist is only a list of standard cell instances and their logical connections. It does not contain any information about the actual metal wires that route the signals between the cells. Nevertheless, the RC parasitics of these wires can give an important contribution to the propagation delays. Therefore, it is mandatory to *back-annotate* the design in PrimeTime with the parasitics exported from Encounter. This can be done with the command:
`read_parasitics "../timing/serial_peripheral_interface.spef".`

6.5 Reports

We have now specified all the necessary input data. Therefore, it is time to run the actual STA and generate some design reports. All reports will be saved as text files under the **reports** subfolder. The reports look the same as those produced by DC.

1. Check for design constraint violations with:
   `report_constraint -all_violators > "./reports/setup_viol.rpt".`

2. Now report the detailed setup check of the 10 worst paths in your design:
   ```
   report_timing -delay_type max -max_paths 10 \
        -slack_lesser_than 10 > "./reports/setup_check.rpt".
   ```

3. You can also generate a power consumption report. Even though this has nothing to do with STA, it is convenient to analyze power consumption with PrimeTime. Enter the following commands:

   - `set power_enable_analysis true`
   - `report_power -verbose > "./reports/setup_power.rpt"`

Take some time to analyze the reports. For example, you can try to compare the timing and power reports with those produced by DC, and see how the results differ. Also, you can check what fractions of the total power are taken by clock network, combinational cells, and registers.

6.6 Hold timing analysis

With the flow described so far, we have used the worst-case PVT library to perform setup checks only (as well as power analysis). The same flow can be also used to perform hold checks. You just need to restart PrimeTime, and re-execute the flow with the following modifications.

- When importing the design, select **fast.db** as the standard cell library. This file models the best-case PVT corner.
- Change delay_type to "min" when reporting timing.
- Change the names of all reports from **setup_*.rpt** to **hold_*.rpt**, so that the previous files are not overwritten.

After running the flow, compare the power reports produced with **slow.db** and **fast.db**. Why the power consumption is different?

6.7 Creating TCL scripts

Once again, you can gather the commands described in this Chapter to a TCL script, that can be invoked by starting PrimeTime with the "-f" option (like DC and Formality). In this case, you will actually create two scripts: one for setup STA, and one for hold STA.