Synthesis and APR Flow for EECS 427

This document describes the standard cell synthesis and automatic place and route (APR) design flow for use in the design of your controller and peripheral blocks. Synopsys Design Compiler, accessed via the design_analyzer GUI and the dc_shell scripting language, will be used for synthesis and Cadence Silicon Ensemble will be used for APR. This document will take you through the synthesis and APR of an example design. Once you have completed this tutorial, you will be able to move on to your own design.

The Example Design

The example design is located at '$TSMC25/parts/cells/tutorial'. It contains the verilog, dc, schem, and layout directories as described below. See the README in the tutorial directory for a better description of many of the files in those directories. The example design is somewhat similar to the one you will be building for your control unit. You will find occasional comments on how your control unit (or other verilog designs) may be different from the example design.

The example design has been through this entire process and so all the files are there for you to see. We recommend you start with the verilog files and schematics and perform the whole flow yourself. If you get stuck or, to verify that you're proceeding correctly, please review the files in the tutorial directory.

Setup

First, create a link in your home directory to the directory $TSMC25/parts/cells. This is done because some of the tools (*sedsm*) we will be using to not recognize environment variables properly (i.e. $TSMC25). This link will make for less typing when trying to link to certain files.

% ln -s $TSMC25/parts/cells/ tsmc25cells

Next, copy the setup file '.synopsys_dc.setup' to your home directory. After you copy the file, feel free to change the 'designer' from "Wolverine" to your name or your group name.

% cp ~/tsmc25cells/synopsys/.synopsys_dc.setup ~

Now, go into whichever directory you want to work in, and create a directory *tutorial* and create the directory structure as shown below.

% cd <your PERSONAL (not your group's) class workspace>
% mkdir tutorial
% cd tutorial
% mkdir dc
% mkdir layout
% mkdir verilog
% mkdir schem

In the future, you will of course replace the 'tutorial' directory with the part you are synthesizing and put it in your group space.

Overview of Synthesis Design Flow

The following steps are involved in the synthesis of a block:

- Create a functional verilog description of the block and verify its functionality using Modelsim. For a design like the controller, you'll want to verify it together with the datapath. You can either use da_ic (and Export Verilog) to generate structural verilog or write structural verilog yourself (like you do for testbenches).
- Synthesize the functional verilog into a gate-level netlist composed of standard cells. This is done using *design_analyzer,* the GUI for Synopsys Design Compiler. You will provide timing constraints to drive the synthesis.
- Generate a Design Architect (da_ic) schematic using *dmgr_d1* (NOT dmgr!) and the output of *design_analyzer*. Verify the post-synthesis, pre-layout design in Modelsim.
- Once you're satisfied with the synthesis results, read the gate-level verilog netlist from *design_analyzer* into *Silicon Ensemble (sedsm)*. This tool will perform automatic place and route of a netlist composed of standard cells.
- Verify the post-layout design in Modelsim, this time including delay information from the .sdf file generated by *sedsm*.
- Export the layout from *sedsm* and use *ICStation* for any further edits (e.g. adding ports) and perform LVS and DRC.

Create and Verify the Verilog Description of your Block

Go into the 'tutorial' directory and copy the verilog files to your verilog directory.

% cp $TSMC25/parts/cells/tutorial/verilog/.v ./verilog/*

You'll also want to copy all the da schematics and symbols into your verilog using *dmgr*. Look at all the verilog files and make sure you understand what they're doing and how they work. Next, go ahead and create the *vsim* project similar to *minicalc_mix_test* in the tutorial to verify that everything works. Please note that we have provided the testbench for you also. It is very important that you build your vsim projects starting with the top-most module (testbench) and then moving downward. If you don't, when you have multiple files in your project (like synth.v, alu.v, decoder.v, etc.) the dummy alu module in your *synth.v* file will overwrite the module defined in the *alu.v* file! You could also just delete any dummy modules coming out of Export Verilog such that you don't have to be concerned about compile order in vsim.

For your control unit, you will of course be writing your own verilog, creating your own symbols and schematics, and performing the verification. We have already done all of that for you in the example design. When writing your verilog, be sure and pay attention to the guidelines and handouts given in class.

We strongly recommend that you partition your synthesizable verilog to some extent. This makes it simpler to assess the synthesis results by viewing the generated schematics. If you instead create one large verilog module describing your controller, you will get a huge, automatically generated schematic that will give you little insight into the quality of the synthesis. As you become more experienced with writing synthesizable verilog, you will tend to make your partitions larger and larger, allowing the synthesis tool to do better optimization.

When you partition the synthesizable design, you will also need to create structural verilog in order to wire the submodules together. You can do this one of two ways: 1) create symbols and "empty" schematics for the submodules and a schematic containing instantiations of the submodules, then Export Verilog to get the structural verilog, or 2) write structural verilog describing the submodule instantiations and wiring. If you choose 1), do so as follows. First, run *da_ic* as usual. Click *Open Symbol* in the palette and enter the name you want (the module you have created) and click OK. Then go ahead and draw a rectangle for your symbol, add pins (specifying the orientation of the pins and the name of each). Note that it's not easy to create bus pins with this method and it's not required. Feel free to try though. Once you have completed drawing your symbol, create the "empty" schematic (pins only) by selecting *Miscellaneous->Create Sheet*. Then take these symbols, instantiate them into another schematic (i.e. like the *synth* block in the example design). The symbol for this block can then be instantiated with your datapath symbol (or the cdff4x16 in the example) and you can Export Verilog for Modelsim verification. Delete the empty module definitions for the low-level synthesizable blocks and replace them with the actual synthesizable verilog code.

Once you have verified that your verilog is correct, go ahead and copy all the verilog you want to synthesize into the *dc* directory. In the tutorial, this will be the following commands:

% cp synth.v ../dc

% cp alu.v ../dc

% cp decoder.v ../dc

%cp sequencer.v ../dc

After copying all the verilog files, be sure and comment out the 'include lines in your verilog, replacing them instead with only the 'timescale directive. design_analyzer will give you errors if it sees the 'uselib line in the included file.

Design_Analyzer

Go into the *dc* (design compiler) directory. Invoke design_analyzer by typing.

% design_analyzer &

For your convenience, a script has been written to automate the synthesis. It is located at $TSMC25/parts/cells/tutorial/dc/dc_script. Please copy this into your own *dc* directory. For your own designs, you will most likely want to use this script as a starting point. If you take a moment to open this script, we'll go through what it does for you.

First, it creates the WORK directory and defines the directory as a design. You will want to create the directory the first time only and always define the design. Next, it reads in all your verilog and elaborates it. The elaborate step maps the verilog into dc's own generic library (gtech). The next big section defines your timing constraints. This is an important part as it tells synthesis what the various delay requirements are (i.e. output1 must be valid 2ns prior to the next clock edge and the clock frequency is 100MHz).

Variables may be defined and used throughout the script. Clocks are declared and used later as references for input and output delays. Note that you can create virtual clocks (i.e. not a real signal in the circuit) for the purposes of referencing. Real clocks can be used for referencing too but are also used in setup/hold checks. The arguments for the -waveform option in create_clock are just transition times in the period, starting with

a rise transition. Input delays are referenced to the right of the clock edge while output delays are referenced to the left. A convenient way of remembering this is to consider inputs to be changing sometime after the clock edge. Outputs, on the other hand, must arrive in advance of the next clock edge to satisfy some setup time. Not all outputs go to flip-flop inputs, but you can still use this method to express their delay requirements.

The next part of the script sets realistic output load capacitances (which you will have to estimate in your design based on what the control unit is driving). Capacitances are in fF. After this, an area constraint is set and a wireload model is selected. In this example, the max area is set to 0. This constraint will never be met but it will drive the synthesis to generate a design with minimal area. Use this setting for your own designs as well. The wireload model provides synthesis with a means to estimate routing capacitance per fanout. Each cell in the standard cell library has timing characterization data based in part on output capacitance. This, along with the wireload model, gives synthesis an idea of gate delays. Static timing analysis is performed during the synthesis to check that timing constraints are being met. We provide only one wireload model (typical_427_controller) but typically more are available based on expected block size.

Then the design is actually compiled and results are saved. Using an effort of medium is recommended. High won't gain much performance, but will take a lot longer to run in *dc*. The files that are saved are the .db file which is basically the entire design and the sdf (standard delay format) which is a rough estimate of the delay of your circuit (without actual layout parasitics, only estimates of these parasitics) and the gate-level verilog netlist you'll need for APR in Silicon Ensemble.

In order to run this script, do 'Setup -> Command Window...'. This will pop up a new window. In the box at the bottom, you can type in commands (or copy and paste) from a script to see what they do and help debug a script. Or you can type *include dc_script* at the *design_analyzer>* prompt to run an entire script. You can see the results in the window. Also, if you need help with commands, you can type *help <command>* at the *design_analyzer>* prompt and you will get help with commands.

It is very important that you look at the results in the window. It will tell you when latches are inferred, if timing is not met, and many other valuable things. Read through all of it. Eventually, you will see what is important and what isn't. Synopsys can also generate many type of reports via the *Analysis->Report...* pulldown menu. Of particular importance is the timing report. Synthesis will produce a design even if it cannot meet the timing requirements. If you don't look at the timing reports, your synthesized result may work only at 10MHz rather than 50MHz. There are no flashing yellow warnings - you must look at the report. If timing is violated, be sure to review your constraints. Very often new users will write constraints incorrectly such that it's difficult or impossible to meet the constraint.

Please note that you can change back and forth from the module view to symbol view to schematic view by using the mouse to double click, the buttons in the upper left to change levels and views, or the 'View -> Change View' and 'View -> Change Level' pulldowns. This will give you a quick view of your schematic before converting to mentor format.

Now you may exit design_analyzer. If you've saved your design in db format, you can easily get back in and write out anything you may have forgotten.

Note that you can also run *dc_shell* instead of *design_analyzer*. The difference is that *dc_shell* is purely a scripting language. There is no GUI interface and no way to see your results. Also, not as much information is returned. But it is there for you to use if you wish. As you get more experienced with synthesis, you will be relying less and less on viewing schematics to assess the synthesis result and therefore *dc_shell* will become a more natural choice.

Schematic Generation

Synopsys provides an interface to Mentor Graphics EDDM database format. However, this interface doesn't currently work with *dmgr_ic*. You will have to use the D1 version of Design Manager, called dmgr_d1 on the CAEN network.

Run the Mentor D1 Design Manager with:

% dmgr_d1 &

Find the *synth.db* file you saved and select it. *RMB > Open > db2eddm*. In the pop up window, click NO for the COMP property. Add $TSMC25/parts/cells/synopsys to the search path. Enter $TSMC25/parts/cells/synopsys/tsmc25.map for the map file. Enter the *schem* directory for the output directory. Click OK. Another window will open. When this says *Finish DB to EDDM Translation .... Terminal Window Closed,* you can go ahead and close and your schematic will be in the *schem* directory.

Your *schem* directory will contain the same hierarchy you saw in the *design_analyzer* schematic. If you use the same symbol for the top-level synthesis module (i.e *synth* in the example design), you can easily rerun your Modelsim verification with the datapath (or cdff4x16). You will typically have to do some debugging at this step. Your synthesizable verilog description can have ambiguities such that the synthesized result doesn't functionally match the synthesizable code (often called "RTL"). So don't be surprised if you run into

discrepancies at this step.

The easiest way to reuse your top-level symbol is by deleting the pins on its empty schematic and copy/pasting the new schematic in its place. Now when you Export Verilog, you will get a complete structural verilog netlist of your design. The full-custom portion will resolve to primitive switches and the synthesized portion will resolve to standard cells. Once you've run this verilog in Modelsim and you're happy with the results, you can move on to place and route in Silicon Ensemble.

It is possible to include delay information for the synthesized design in this simulation. However, we don't recommend that you do this in the EECS 427 design flow. The sdf (standard delay format) exported from design_analyzer will be based on estimated capacitances from the wireload model. You can get much better sdf info after place and route is complete. Our place and route cycle will be very fast since our synthesized modules are relatively small, so you might as well wait until you have layout to run Modelsim with the sdf. If you wish to try this, though, add the following line:

```
initial $sdf_annotate("<path_to_sdf>/synth.sdf");
```

to your *minicalc_mix.v* file (the result of Export Verilog), inside the *synth* module definition. You need to add this line inside the module definition, after all the input/output declarations, but before any instance names.

Silicon Ensemble (layout generation)

Now we are ready to create the layout for the synthesized result. To get started, set up your *layout* directory and launch Silicon Ensemble as follows:

```
%cd ../layout
%cp ../dc/synth_gate.v .
%cat ~/tsmc25cells/cells.v >> synth_gate.v
%cp $TSMC25/misc/se.ini .
% sedsm
```

Be sure to use the gate-level netlist from design_analyzer and not the one from Export Verilog. Export Verilog doesn't declare wires properly and Silicon Ensemble will not load the netlist. In addition, Silicon Ensemble wants module definitions for all instances. Appending the standard cell verilog (cells.v) is the easiest way to do this. Finally, the *se.ini* file contains settings that will make all the routing layers visible for you. If you don't do this, you won't see some of the higher level metal routes.

One comment about sedsm. When doing the file selections as described below, sometimes it has problems navigating through directory structures so just be patient and keep playing with it to find the files you need. Sometimes you may need to enter the complete path to some files (note that you can paste the path into the selection boxes instead of typing the entire thing).

In the window that pops up, choose 'File -> Import -> LEF...'. The lef file contains *abstracts* of each standard cell. An *abstract* provides the APR tool with cell footprint and port location/layer info for each cell as well as general technology information (design rules, number of routing layers and routing grid info). It doesn't contain complete layout, though. For example, there is no diffusion, nwell, etc. layer info in the lef file.The lef file is located at "~/tsmc25cells/cadence/cells.lef". Click 'OK' after specifying this path.

Once we have given the layout tool a description of our library, we need to give it our verilog files. To do this, choose 'File -> Import -> Verilog'. Click 'Browse' in the 'Verilog Selection' box. Click your *synth_gate.v* file and click 'Add' to add it to the window. Click 'Ok'. Next, type in your top level verilog module name (*synth*) in the 'Verilog Top Module' box. Also, change all the vdd! and gnd! to vdd and gnd (no !) and click 'OK'.

Next, load the timing information for the standard cell library. This information is found in the '~/tsmc25cells/cadence/tlf/tsmc25.gcf' (which is also linked to the tsmc25.tlf file in the same directory). To import this file, choose 'File -> Import -> Timing Library' and select the tsmc25.gcf file (you don't need to select the tlf file ever, it is automatically linked). This file contains all of the timing characterization information for the standard cells and allows Silicon Ensemble to issue a static timing verification run based on real routing parasitics. The results is an sdf file you can use in Modelsim later. Note that in a more complete flow, you would also feed in your timing constraints to Silicon Ensemble and have the static timing analysis check to see if the routed result meets timing.

Now you're ready to floorplan your design, place i/o pins and cells and do the route. Choose 'Floorplan -> Initialize floorplan...'. Click on 'Flip Every Other Row' and 'Abut Rows' which will share power rails between cells. You can change the aspect ratio (width to height ratio of the completed layout) to give different rectangular shapes to match your higher-level floorplan. You can also change the row utilization to vary the amount of space in rows that is to be filled by cells. You might reduce this value if sedsm has trouble completing routes.

Click 'OK'. You should now see rows in the layout window. There are no cells yet since they haven't been placed.

As an optional (but recommended) step, we can tell *sedsm* where we want our pins placed. To do this, choose 'Place -> IOs...'. In the pop up window, click on 'I/O Constraint File' and then click on 'Write'. This will write a sample pin file with the filename given in the box. If you open this file, you will see a list of all your pins in the 'IGNORE' section. To place your pins, simply edit this file and move the pins into the sections you want them (TOP, BOTTOM, LEFT, or RIGHT). You must include the entire line that exists in the IGNORE section into whichever section you wan the pin. It is very important that you leave NO pins in your IGNORE section! An example file is in '$TSMC25/parts/cells/tutorial/layout/ioplace.ioc'.

Now, we're ready to finish the layout. Choose 'Place -> IOs...'. Click 'I/O Constraint File' and specify the metal layers for different pins and the spacing. When you're done, click 'OK' and your pins will show up in your design. If you don't specify what layers to use for pins, metal1 will be used by default and you may end up with signal pins laying on top of vdd/gnd routes (sedsm will fail on power routing if this occurs). The example design APR did not use the ioplace file. Pins were placed randomly instead with metal5 used for pins on the left and right edges and metal4 used for pins on top and bottom edges. Again, you will typically want to use the ioplace file to control pin placement based on your knowledge of the routing plan at the next level up in the hierarchy.

Now, place the standard cells choosing 'Place -> Cells...' and click 'OK'. Note the small layer palette to the left of the layout window. "Sl" is for select and "Vs" is for visible. Cell, Row and Pin are self-explanatory. Net is any regular, non-power, non-special signal route. "Swire" is special wire, typically just vdd/gnd for us, though clocks can often be special wires. Next, we route the power grid by choosing 'Route -> Route power -> Follow pins'. Set the width to 9.5 (width of the power line routes) and choose 'OK'. Now, route the rest of the signals by choosing 'Route -> Wroute...'. Leave the defaults and select 'OK'.

You can typically go back and generate different layout quickly by selecting Floorplan->Reset Floorplan... then going through the previous steps starting from Floorplan->Initialize Floorplan... Note that you can't view the full layout in sedsm since it doesn't have any layout information for poly, diffusion, etc. inside the cells. Sedsm only needs to know about metal layers, pins and cell boundaries.

To view the whole layout, we need to export it in a format that ICStation can understand. A standard format for this is gdsii (aka gds2). This is a common binary format for exchanging layout data between tools. To export this file, choose 'File -> Export -> GDSII'. Click the box by 'GDS-II' and 'Map File'. The map file just describes the layers that are allowed for conversion from the *sedsm* layout to the gdsii description that will later become the ICStation layout. After clicking the boxes, type in a filename for the gdsii export, most likely 'synth.gds2'. Also, use the browse button to select the map file '~/tsmc25cells/cadence/gds2.map'. Also, click on the 'Units' box and change to 'Thousands'. Click 'OK'.

We also want to export more accurate timing information about our design. To export the SDF file, choose 'File -> Export -> SDF...' and enter your 'synth_sedsm.sdf'. Leave the defaults and choose 'OK'. As a final step, you will need to add the *$sdf_annotate* line in your Export Verilog output to point to this sdf file (see end of Schematic Generation section for syntax). You definitely should use this SDF file in your Modelsim simulation.

At some point you should save your layout database in sedsm's own format. You can do this with File->Save, though "LBRARY" is the default name to save to. To choose your own name, select File->Save As. SYNTH has been saved in the tutorial.

ICStation

Run ic as normal and when it opens, scroll down in the 'Session' palette and choose 'ICLink'. In the pop up window, choose source to be 'gdsii' and destination to be 'icgraph' and click 'OK'. In the new window, use the navigator to choose the s*ynth.gds2* file you created for the GDSII box, choose the 'layout' directory for your 'ICGraph Directory' and browse to '$TMSC25/process/import.gds.options' for the options file. The options file is similar to the map file we used when creating the gdsii file. When all three have been selected, click 'Ok' and then click 'Yes' in the box that may pop up asking to continue. When the process is complete, you can open the cell that ICLink created named <design_name> in your layout directory. You may see some other cells that ICLink created also, but these are used in the hierarchy so you do not need to open those.

DRC and LVS need to be run. You will typically run into a handful or DRC errors due to nwell spacing/width errors and metal1 wide-metal spacing errors. These are typically simple to fix. LVS should always be clean. Before running DRC and LVS, you'll have to add ports to your design. You can add ports automatically for all i/o's other than vdd/gnd by running an ample script as follows (first click on the layout window to make this the active scope):

```
dofile $TSMC25/misc/text2ports.ample
```

This will make ports out of the text objects that are present after the conversion. At this time, there is no equivalent code to add ports for vdd and gnd. You will have to do that manually for now.