Design Compiler Tutorial

1. Introduction

This tutorial is intended to familiarize you with the logic synthesis tool called Design Compiler (DC) from Synopsys. This is a BRIEF tutorial. For further information on any command, you can type "help [command]" or "man [command]" in Synopsys. Wildcards are also allowed so "help get_*" will list all commands that start with "get_", for example. Then typing "man get_nets" will display the syntax and usage of a command.

Synopsys has a GUI, but it is rarely used. In fact, I have never used it. Instead, everyone uses a TCL shell that comes with Synopsys. This is run by typing "dc_shell-t" at the command prompt. (In prior versions, there was a different DCSH ("DC shell") version that was used. This can be invoked with just "dc_shell" but is not covered in this tutorial. If you find information on-line, it may use DCSH. Your path is automatically set up in the setup_cmpe222 script that I had you source.

The Synopsys On-Line Documentation (SOLD) is available by typing "sold" at the command prompt. For help with Design Compiler, select "Design Compiler", "next page", and what you want under "SYNTHESIS MAN PAGES" such as "Error Messages".

2. Reading Libraries

I wrote a script to set up the libraries for the Oklahoma State University TSMC 0.18um cell library. This is a very small library and does not have I/O pads, but we will use it for at least this tutorial. The script to set up this library is setup_osu180.tcl. This script sets up a few DC environment variables including: search_path, target_library, and link_library. search_path specifies directories to look for verilog files. All modules must be defined in verilog including the standard cells (as defined in osu_stdcells/lib/tsmc018/lib/osu018_stdcells.v). You can append your own paths to this if you like.

The script also defines a few variables that are used later in the scripts by prefixing them with a $. These include:

```
LIB_NAME The library name.
DFF_CELL The typical (minimum size) DFF cell name.
LIB_DFF_D The DFF input pin as library/cell/pin.
DFF_CKQ The assumed value of the Clock-to-Q delay of the DFF in technology units.
DFF_SETUP The setup time of the DFF in technology units.
```

$LIB_NAME will then return the value of the library name.

You can EXCLUDE certain cells in the library from being used with "set_dont_use". For example, you will typically not want clock buffers to be used for anything but clocks, so type:

```
set_dont_use [get_lib_cells "$LIB_NAME/CLK*"]
```

Other typical cells to exclude are: Tie high and tie low cells, delay cells, etc.

4. Configuration Variables

You will make many synthesis scripts. It is best, therefore, to make them generic by using per design variables. I typically use these variables:

```
# The top module name.
set TOPLEVEL  "lfsr"
# A subdirectory to put reports in.
set LOG_PATH  "log/"
# A subdirectory to put the synthesized (gate-level) verilog output in.
set GATE_PATH  "gate/"
# The subdirectory containing the unsynthesized (behavior) verilog input.
set RTL_PATH  "verilog/"
# A name to supplement reports and file names with.
set STAGE  "final"
# The clock input signal name.
set CLK  "clk"
# The reset input signal name.
set RST  "rst"
# The target clock period in library units.
set CLK_PERIOD 0.1;
# The clock uncertainty in library units.
set CLK_UNCERTAINTY  0.1;
```

3. Reading a Design

DC can read both Verilog and VHDL, but this tutorial uses Verilog. I first append the name of the directories containing my Verilog to the search path. Therefore, I don't need to have a path in front of every file name. This will read the Verilog for this tutorial:

```
set search_path  [concat $search_path $RTL_PATH $GATE_PATH]
# read the verilog in a "bottom up" manner to reduce the numbe of unresolved warning messages
read_verilog shifter.v
read_verilog lfsr.v
```

During this stage, it is also important to make sure that all of the registers (i.e. flip-flops or latches) in the design are inferred properly. If you do not know the design, this is tricky. The MOST COMMON ERROR in coding verilog is to inadvertantly make memory elements by coding something incorrectly. You will see a message for every single memory element that is inferred in your design which look something like this:

```
Inferred memory devices in process
        in routine shifter line 7 in file
                'dc_tutorial/verilog/shifter.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|       val_reg       | Flip-flop |  15   |  Y  | N  | Y  | N  | N  | N  | N  |
|       val_reg       | Flip-flop |   1   |  N  | N  | N  | Y  | N  | N  | N  |
===============================================================================
```

This information tells you the memory type (flip-flop or latch), the number of bits (15 and 1), whether it uses an asynchronous set/reset (AS, AR) or synchronous set/reset (SS, SR) and some other information. MAKE SURE THESE ARE RIGHT.

You can now make any module your "current module" with the current_design command. We want to ensure that we are optimizing the top level design:

```
current_design $TOPLEVEL
```

After everything is read in, do a final check to make sure that the design is able to find ("link") all modules. Typically, if you use memories, it will ensure that these are in the current library too. We then also "uniquify" the design. If a module is instantiated more than one time, this command will make separate modules for each instance. This allows each to be optimized independently for their particular usage. You may not want to do this if you DEFINITELY want two modules to be the same. Lastly, we run the "check_design" command and save the output in a log file. The final commands for reading a design are shown here:

```
link
uniquify
# check the design for errors such as missing module definisions
check_design > $LOG_PATH$TOPLEVEL-check_design.log
```

3. Constraining a Design

Besides the inferred memory elements, the other most important part of synthesis is properly constraining your circuit. The constraints tell the optimization tool how fast, how much power, or other essential information about your design. If you over-constraint, the tool may not meet your constraints.

The first step is to define the clock signal. This is done as follows:

```
create_clock $CLK -period $CLK_PERIOD
set_clock_uncertainty $CLK_UNCERTAINTY [all_clocks]
set_dont_touch_network [all_clocks]
```

The second statement defines a bound on clock skew that you will later try to meet during physical design. The last statement tells DC to not optimize the clock tree. This is best done during placement & routing when you actually know the physical locations of the design.

Similarly, the reset signal is also not optimized since we typically do not care about the performance of the reset signal:

```
remove_driving_cell $RST
set_drive 0 $RST
set_dont_touch_network $RST
```

Next, we define the input and output constraints of the design. For an output, it is assumed that it will drive four flip-flops unless we have more accurate information from the system level specification. We also require the outputs to arrive a setup time before the clock which is assumed to drive the flip-flops on the outputs.

```
set_output_delay $DFF_SETUP -clock $CLK [all_outputs]
set_load [expr [load_of $LIB_DFF_D] * 4] [all_outputs]
```

For inputs, we make a similar assumption. We first define the list of all inputs excluding the reset and clock pins. We then assume the signal comes from a flip-flop clocked by the system clock. We assume that the input is driven by the output pin of the DFF cell that we specified during our timing setup. These are SIMPLIFIED approximations. If you know more information about the system, USE IT. For example, if a signal is going off-chip, it's load will be much larger than four DFF cells. If it is going to another on-chip module with a different clock domain, specify that.

```
set all_inputs_wo_rst_clk [remove_from_collection [remove_from_collection [all_inputs] [get_port $CLK]] [get_port $RST]]
set_input_delay -clock $CLK $DFF_CKQ $all_inputs_wo_rst_clk
set_driving_cell -library $LIB_NAME -lib_cell $DFF_CELL -pin Q $all_inputs_wo_rst_clk
```

4. Optimization

To optimize your design, you can enable some new features by turning on "ultra" optimization. The design is then synthesized with the compile command:

```
set_ultra_optimization -f
compile -boundary_optimization -map_effort low
```

Only change the effort to "medium" or "high" after learning that "low" is not meeting your constraints. This will avoid a lot of extra CPU time.

5. Saving the Design

You can save your design in many formats, but the most common are as gate-level verilog ("structural verilog") or DC "db" format. Both commands are shown here:

```
write -hierarchy -format db -xg_force_db -output $GATE_PATH/$TOPLEVEL-$STAGE.db
write -hierarchy -format verilog -output $GATE_PATH/$TOPLEVEL-$STAGE.v
```

In addition to saving the design, we can also save the constraint setup in a file called a Synopsys Design Constraint (SDC) file. This will then be given to the placement & routing tool so that it can optimize the clock tree and placement of the design later. This command is shown here:

```
# this is needed to get Astro to work right...
set_propagated_clock [all_clocks]
write_sdc $GATE_PATH/$TOPLEVEL-$STAGE.sdc
```

6. Reports

Reports are the interface to the designer. In order to know what is going on with your design, you must learn to interpret the information in the various reporting formats. The most common report that you will use is the timing report. This tells you if some of your paths to not meet the cycle time constraint so that you can fix it by changing the logic or changing the constraints. Examples of all the reports are available by typing "man [report_command]". Here are some other report types that we will use:

```
report_area     > $LOG_PATH/$TOPLEVEL-$STAGE-area.log
report_timing -nworst 10        > $LOG_PATH/$TOPLEVEL-$STAGE-timing.log
report_hierarchy                > $LOG_PATH/$TOPLEVEL-$STAGE-hierarchy.log
report_resources                > $LOG_PATH/$TOPLEVEL-$STAGE-resources.log
#report_references               > $LOG_PATH/$TOPLEVEL-$STAGE-references.log
report_constraint               > $LOG_PATH/$TOPLEVEL-$STAGE-constraint.log
#report_ultra_optimizations      > $LOG_PATH/$TOPLEVEL-$STAGE-ultra_optimizations.log
```

7. Summary

The entire script that synthesizes our module is lfsr.tcl There are many other situations that you will run into with synthesis and learning to deal with them cannot be summarized in a single tutorial. Some other situations, for example, are:

- Back-annotating extracted parasitics from placement into timing analysis using read_parasitics.
- Writing Standard Delay Format (SDF) for timing-aware Verilog simulation using write_sdf
- Removing impossible paths ("false paths") from analysis using set_false_path
- Producing "glitch free" logic by telling synopsys to not change certain cells using set_dont_touch