Synthesize & Power Analyze

Modified: Yoon Seok Yang

Originally made by Nemat Allah Ahmadyan
Synthesis

• Process of converting verified HDL code to hardware
Synthesize

- The process of mapping **RTL netlist into Gate-level netlist**
- We recommend Synopsys Design Compiler.
- Environment setup for Design Compiler
  - `% setenv SYNOPSYS /opt/synopsys/Z-2007.05-sp3`
  - `% setenv LM_LICENSE_FILE /opt/licenses/license.dat`
  - `% set path = ($SYNOPSYS/linux/syn/bin $path)`
- Starting DC:
  - `dc_shell & dc_shell-t (TCL)`
  - `design_vision`
Synthesis Flow

1. Set string variables
2. Set min-max cell library
3. Read in design
4. Link and uniquify
5. Set environment and constraints
6. Set compile directives
7. 1st Compile
8. 2nd Compile to refine
9. 3rd Compile to optimize power
10. Save synthesized design, design information, and reports

Important Commands

- `set_min_library`
- `read_file`
- `link, uniquify,`
- `set_dont_touch`
- `set_fix_multiple_port_nets, set_multibit_options, set_structure, simplify_constants, transform_csa`
- `compile`
- `set_max_area, set Ultra_optimization, write, compile`
- `read_saif, set_max_dynamic_power, compile, write`
- `remove_unconnected_ports, write, write_sdf, write_sdc, write_parasitics, change_names, define_name_rule, report_area, report_timing, report_constraint, report_resource`

Triggers the use of Power Compiler
Defining Variables

• Variables includes:
  – Libraries (min/max)
  – Cache
  – Design
  – constraints
Reading libraries

• Libraries Usually will be provided in Liberty format (.lib)
• Read them using read_lib
• Then produce synopsys db file using write_lib command.

• ReRead the library db file to synopsys.
Reading Libraries

• For one process, we may have many timing libraries, usually, best, typical & worst.
• dc_shell> set_min_library worst.db –min_version best.db

```bash
if [array exist BEST_LIB] {
    echo "setting min/max libraries."
    foreach lib_w [array names BEST_LIB] {
        set lib_b $BEST_LIB($lib_w)
        set_min_library $lib_w -min_version $lib_b
    }
}
```

• For simplicity, we recommends:
• dc_shell> set link_library [set target_library [concat [list lib.db] [list dw_foundation.sldb]]]
• dc_shell> set target_library “lib.db“
• dc_shell> define_design_lib WORK -path ./WORK
Reading Design, link & uniq

- **Link**
  - Resolve the design reference based on reference names
  - Locate all design and library components, and connect them
- **Uniquify**
  - Removes multiply-instantiated hierarchy in the current design by creating a unique design for each cell instance

```
dc_shell> analyze -f verilog $my_verilog_files
dc_shell> elaborate $my_toplevel
dc_shell> current_design $my_toplevel
dc_shell> link
dc_shell> uniquify
```
Operating Condition

• Setting Min/Max operating condition (only if you’ve min/max libraries)

```plaintext
dc_shell> Set_operating_conditions --max "slow" --min "fast"
dc_shell> Set_operating_condition --max "slow"
```
Design Constraints

• Design Objectives
  – Speed
  – Area (default)
  – Power (requires Power Compiler license)

• When both area and delay constraints are set, design compiler will give speed priority.
Constraining the Design

• The synthesizer is "lazy", if you don’t set the proper constraints it will select constraints that will make him work less.

  Always set proper constraints

• Timing Constraint
  – Max delay combinational delay
  – Max area total circuit area
  – Max power for power limitation
  – Setting the constraint does not guarantee the result
Constraint for Area

• By default, timing constraints have higher priority over area constraint.
• 

“-ignore_tns” -> give area priority over timing.

• area constraint can be set using the “set_max_area” command:
  
dc_shell> set_max_area 100
Sequential Timing

• Timing Paths
  – Register to register
Sequential Timing

- Timing Paths
  - Register to register
  - Input to register
Sequential Timing

- **Timing Paths**
  - Register to register
  - Input to register
  - Register to output
Sequential Timing

• Timing Paths
  – Register to register
  – Input to register
  – Register to output
  – Input to output

One of these paths will limit the performance of the system.
Sequential Timing

- Timing Paths
  - Register to register
  - Input to register
  - Register to output
  - Input to output

One of these paths will limit the performance of the system.
Constrain for Speed

• Always have a “Time Budget”
• With the simplified timing assumption:
  – dc_shell> create_clock “CLK” –period T –waveform { T/2 T }
  –name cn
  – Delay of input signals (Clock-to-Q, Package etc.)
    dc_shell> set_input_delay 0 –clock cn all_outputs() – CLK
  – Don’t forget! Remove_input_delay [get_ports CLK]
  – Reserved time for output signals (Holdtime etc.)
    dc_shell> set_output_delay 0 –clock cn all_outputs()
  – SDC file (write_sdc)
  – Later STA & P&R tools need these constraints

• Virtual Clock (for combinational circuit)
Constraint for speed

- **Set_max_delay**
  - Specifies the desired maximum delay for paths in the current design.

  ```
  dc_shell> set_max_delay 15.0 -from {ff1a ff1b} -through {u1} -to {ff2e}
  dc_shell> set_max_delay 8.0 -from {ff1/CP} -rise_through {U1/Z U2/Z} -fall_through {U3/Z U4/C} -to {ff2/D}
  ```

- **set_min_delay**
  - sets the minimum delay target for paths in the current design

  ```
  dc_shell> set_min_delay 3.0 -from ff1/CP -rise_through {U1/Z U2/Z} -fall_through {U3/Z U4/C} -to ff2/D
  ```
Different constraints, different circuits
Don’t trust the synthesizer too much
Don’t trust the synthesizer too much
Don’t trust the synthesizer too much
Don’t trust the synthesizer too much
Timing Exceptions

• Static timing analysis assumes all data transfer within one clock cycle.
• By default, all timing paths are measured using the same rule.
• Any exception to the above are referred to as timing exception. The following are commands to set timing exceptions:
  – `set_false_path`
  – `set_multicycle_path`
  – `set_max_delay`
  – `set_min_delay`
• Timing exceptions are identified by designers only. It is not possible to identify timing exceptions automatically using tools.
Clock

- Create_clock
- Set_clock_skew
- Set_clock_uncertainty
- Set_clock_transition

```plaintext
set CLOCK_PIN_NAME clk_p
set CLOCK_NAME clock
set RESET_PIN_NAME rst_p
set CLOCK_PERIOD 20
set CLOCK_CLK_TRANSITION 1.0
set CLOCK_UNCERTAINTY 0.2

set WIRE_LOAD_NAME "UMC18_Conservative"
set WIRE_LOAD_LIB "slow"

set MIN_BUFFER "slow/BUFX1/A"
```
Time Budget

• You’re not alone in the design!
• For a 100 MHz Clock, block N used 40% of clock period.
• Better to budget conservatively than to compile with paths unconstrained.
Gated Clock

- Gated clocks can be specified at the root of the clock port.
- By default, design compiler will assume ideal clock and take the gating logic as zero delay elements.
- Derived clocks must be specified at the outputs of sequential elements:

```
dc_shell> create_clock {ClkRoot} -p 8 -name "croot"
dc_shell> create_clock {clkgen/Q1 clkgen/Q2}-p 16 -name "croot_by_2"
```
Compiling

• Usually, we have to perform 2 or 3 compile

1st compilation  
Rough compilation (timing only)

dc_shell> compile -map_effort medium

2nd compilation  
Refine circuit area and timing

dc_shell> add some constraints
dc_shell> set_ultra_optimization true
dc_shell> set_ultra_optimization -force
dc_shell> compile -map_effort high -incremental_map

3rd compilation  
Optimize power
Optimize for Power with SYNOPSYS POWER COMPILER
Power Compiler

• Power Compiler always works within the Design Compiler shell and is transparent to Design Compiler users.
• Synopsys Power Optimizations “tricks”
  – gating clocks of register banks
  – operand isolation.
Power Components

- Leakage
- Dynamic
  - Switching
  - Internal

[Diagram with labels: Rising signal at IN, Leakage current, Short-circuit current, Switching current, Falling signal at OUT, Voltage, Time, Leaks, Switching current]
Power Compiler flow
Switching activity

• Back annotation file:
  – contains the resultant switching activity of the elements monitored during RTL simulation.
  – Annotate the switching activity on some or all design objects by using the read_saif, annotate_activity or set_switching_activity commands

• Forward annotation file:
  – Containing directives that determine which design elements to trace during simulation.
  – The gate-level forward-annotation file is created by using the lib2saif command.
  – RTL forward annotation file is generated using rtl2saif command.
    • using information from the GTECH design created by HDL Compiler.

• Synopsys HDL Compiler converts the design to a technology-independent format called a GTECH design
SAIF file

- The forward-and back-annotation files are in Switching Activity Interchange Format (SAIF).
- many simulators (including ModelSim) support the Value Change Dump (VCD) format.
  - Synopsys offers an interface between VCD and SAIF.
    - vcd2saif command
- ModelSim VCD Command:
  - vsim> vcd file test.vcd
  - vsim> vcd add –r testbench/core/*
Activity Generation

- Activity of the synthesis invariant nodes is captured during RTL simulation
  - primary inputs, sequential elements, black boxes, three-state devices, and hierarchical ports.
- For more accurate power estimation, dumping activity of all node is required.
- Manually annotating activity
  - `dc_shell> annotate_activity -static_probability 0.5 -toggle_rate 0.2 -period 20`
  - `dc_shell> annotate_activity -static_probability 0.5 -toggle_rate 2.0 -period 20 -objects clock`
Switching Activity in ModelSim

- We recommends USING VCD with ModelSim
  - `vsim> vcd file test.vcd`
  - `vsim> vcd add –r testbench/core/*`
- However, it’s possible to generate SAIF file in modelsim
  `vsim –foreign “dpfli_init dpfli.so” test (or Use PLI )`
  `Read_rtl_saif fwd.saif test/DUT`
  `Set_toggle_region test/DUT`
  `Toggle_start`
  `Run -all`
  `Toggle_stop`
  `Toggle_report back.saif 1e-9 test/DUT`
Constraints for Power

- Triggers Power Compiler
- Usually it’s like this:
  - First compile
  - read saif (backward)
  - set_max_dynamic_power
  - set_max_leakage_power
  - Compile, write
Power Compiler - Analyze

• First, generate the forward saif & simulate the design in ModelSim. Then run the design compiler, after initial commands, loading libraries etc, use:

  dc_shell> create_power_model -format vhdl -hdl_files {sm_seq.vhd sm.vhd} -top_design sm_seq
  dc_shell> reset_switching_activity -all

• Read the backward-saif

  dc_shell> read_saif -input sm_back.saif -instance test_sm/dut -rtl_direct
  dc_shell> report_activity > reports/report_activity_5.rpt
  dc_shell> report_rtl_power > reports/report_rtl_power_5.rpt
Power Compiler - Compile

• Must specify switching activity
• Invokes Power Compiler

```bash
dc_shell> reset_switching_activity -all
dc_shell> read_saif -input test.saif -instance testbench/core -rtl_direct
dc_shell> report_power
```

• Setting Constraints & Compile
```bash
dc_shell> set_max_dynamic_power 450 uW
dc_shell> set_max_leakage_power 200 nW
dc_shell> compile -map_effort high -incremental_map -verify_effort medium
```

• Final reports
```bash
dc_shell> report_saif -hier -missing -rtl > reports/report_saif_6_1.rpt
dc_shell> report_power -hier -verbose -analysis_effort medium -net -cell -sort_mode name > reports/report_power_6_1.rpt
```
Power Compiler – Clock Gating

- Example: Latch-based clock gating
Clock Gating user control

- Integrated or non-integrated gating cell
- Latch based or latch–free
- Logic to increase testability
- Minimum nr of bits to trigger clock gating
- Explicitly include/exclude signals
- Max fanout for each gating element
- Rewire clock-gated register to another clock gating cell
- Resize clock-gating element
Clock Gating Command

```
set_clock_gating_style
    [-sequential_cellllatch | none]
    [-minimum_bitwidthminimum_bitwidth_value]
    [-setupsetup_value]
    [-holdhold_value]
    [-positive_edge_logic{ gate_list | integrated}]
    [-negative_edge_logic{ gate_list | integrated}]
    [-control_pointnone | before | after]
    [-control_signalscan_enable | test_mode]
    [-observation_pointtrue | false]
    [-observation_logic_depthdepth_value]
    [-max_fanoutmax_fanout_count]
    [-no_sharing]
```
Power Compiler – Clock Gating

• Enabled by
  • dc_shell> set_clock_gating_style -pos {inv nor buf} -neg {inv and inv}
  • dc_shell> elaborate sm_seq -gate_clock

• Reports:
  • dc_shell> report_clock_gating > reports/report_clock_gating_11.rpt
  • dc_shell> set_clock_skew ideal CLK
  • dc_shell> propagate_constraints -gate_clock

• Then compile
Power Compiler – Operand Isolation

**Problem**
Operands change inducing switching even when the output is being ignored

**Solution**
Isolate operands using the control signal
Operand Isolation

• Pragma Isolation Method (in HDL code)
  
  if ( c1='1') then
    o <= temp + b ; -- synopsys isolate_operands
  else
    o <= g ;
  end if ;

• Based on Synopsys Gtech Isolation Method
  – DC Script:
    • set_operand_isolation_cell {FSM/DW02_MULT}
Power Compiler – Operand Isolation

• Enable it by:
  • `dc_shell> do_operand_isolation = true`
  • `dc_shell> set_operand_isolation_style -logic AND`
  • `dc_shell> set_operand_isolation_cell {FSM/DW02_MULT}`
  • `dc_shell> set_operand_isolation_slack 2`

• Then Compile

• Reports
  • `dc_shell> report_operand_isolation > reports/operand_isolation_12.rpt`
Synthesize with StYLe!

- **Use scripts**
  - **Automatic**
    - Press and run
    - No user interaction required
  - **Less error prone**
    - Avoids user’s mistake during operating GUI interface
  - **Reusable**
    - Synthesis script can be easily modified for different projects
  - **Be procedural**
  - **Suggestion:** build your scripts with make
  - **Suggestion:** organize your scripts
    - Compile.tcl
    - Constraints.tcl
    - Util.tcl ...
Save your work!

- Remove unconnected ports before saving the synthesis design
- Save synthesized design and info
  - XXX_syn.db  SynopsysDB file
  - XXX_syn.v  Verilog gate-level netlist
  - XXX_syn.sdf  back annotated time info for gate-level netlist
  - XXX_syn.spef  parasitic info (RC) of the gate-level netlist

remove_unconnected_ports -blast_buses [find -hierarchy cell "*"]
write -hierarchy -format db -output [format "%s/s/s" "DB/" $TOP "_syn.db"]
write -hierarchy -format verilog -output [format "%s/s/s" "GATE_SYN/" $TOP "_syn.v"]
write_sdf -context verilog -version 1.0 [format "%s/s/s" "GATE_SYN/" $TOP "_syn.sdf"]
write_parasitics -output [format "%s/s/s" "GATE_SYN/" $TOP "_syn.spef"] -format reduced
write_sdc [format "%s/s/s" "GATE_SYN/" $TOP "_syn.sdc"]
Important Notes

- Analyze package files (if any exists) before elaboration
- Current design is one of the elaborated ones.
- Note files’ order when using `analyze command`
- Use `reset_switching_activity command` before `read_saif command`
- Use `check_design–post_layout to understand current design errors and warnings`
- Annotate switching activity before and after each compile
Important Notes

- You are not allowed to use `--rtl_direct` option for `read_saif` command in `dc_shell`.
- Do not use generate loops during back SAIF file generation using file DPFLI.
- Different reports generated by Synopsys Design Compiler:
  - `report_clock`
  - `report_bus`
  - `report_references`
  - `report_net`
  - `report_cell`
  - `report_timing --delay min/max --max_path`
  - `report_constraint --all_violators`
  - `report_resources`

...
Synthesis Results

• Synthesis is just a tool
  – Synthesis tools do not magically generate circuits
  – They are supposed to generate exactly the circuit that you want
  – You must have a good idea of what the synthesis result will be

If the result is not as you expect, you should convince the synthesizer to produce the correct result.