Getting the Most Out of Synthesis

Dr. Paul D. Franzon

Outline
1. Timing Optimization Approaches
2. Area Optimization Approaches
3. Design Partitioning

References
1. Smith and Franzon, Chapter 11
2. D. Smith, Chapters 4, 6, 8.
3. Kurup, Chapters 2, 4, 5.
Optimizing For Speed

General HDL Coding Style Techniques:

- Specify don’t cares on outputs whenever possible. E.g. Consider the following Karnaugh Map with ‘x’ and with ‘0’ instead of ‘x’

<table>
<thead>
<tr>
<th>ab</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>cd</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Be careful: Does increase potential for mismatch between behavioral simulation and post-synthesis simulation.

- Specify multiplexors if possible (see earlier example).

- Using ‘+’ and ‘-’ automatically gives you a ripple carry adder unless you specify otherwise (hand design or use Design Ware)

- \( y = a/b; \) and \( z = a \% b; \) will **NOT** give you efficient circuits as Design Ware does not contain designs for these

- Design the circuits in detail instead
...Optimizing for Speed

Use Design Knowledge:

- Use **parentheses** to group items if you have special design knowledge. e.g. If you know inputs c and d will arrive after a and b:
  
  ```
  output = (((a + b) + c) + d);
  ```
  
  - Don’t use () unless you have extra knowledge

- Find **parallelism** whenever possible:
  
  - see `Writing Successful RTL Descriptions in Verilog`

- Use **bit-slicing** to make the Synthesis more optimal
  
  - Synopsys can optimize smaller designs easier than larger ones
  - see `Writing Successful RTL Descriptions in Verilog`

- Keep the **critical path** in one module
  
  - It is difficult to optimize across modules
  - Try to keep this module as small as possible

- Use structuring to improve the design
  
  - see `Writing Successful RTL Descriptions in Verilog`
Once within Synopsys:

- Make sure the **constraints** are correctly set:
  - `set_input_delay`
  - `set_output_delay`
  - `set_driving_cell`
  - `set_load`
  - `create_clock`

  - Initially, the first four of these are estimated. Later those estimates can be refined once the actual fan out, wire loading, etc. is known.
  - Other constraints might have to be set, depending on your library.
  - Use worst case library for set-up timing and best case for hold timing
  - Nets that don’t go through a clocked element will need a max delay constraint. (This is usually poor technique though)
    - e.g. `max_delay 8ns -from in1 -to out1`
  - Some inputs never change (e.g. configuration pins) and some outputs are asynchronous - use `set_false_path` on these.
...Optimizing For Speed

Within Synopsys:

- When you have multiple clocks, Synopsys needs to know what signals are going between the different clocks. e.g.
  ```
  create_clock period 20 clock20 waveform {0 10}
  create_clock period 10 clock10 waveform {0 5}
  set_multicycle_path 2 -setup -from ff1/cp -to ff2/D
  // ff1 has a 20 MHz clock while ff1 has a 10 MHz clock
  ```
- `set_multicycle_path` is also often required for Mealy outputs of FSMs
  - see Methodology Notes -> State Machine Design Techniques for details

How Synopsys Works:

- When the design is read in it is `mapped` onto a logic library but not optimized
- You then give Synopsys your design constraints and goals
- `compile` does logic minimization while trying to meet timing constraints
- `compile -incremental` can be performed (after changing various aspects of the constraints, etc.) to try to improve the design
Timing Optimization Within Synopsys

Scenario 1: You are trying to work out the fastest possible clock speed:

- Start with a clock speed that should be achievable but only just so and do a compile
- Find out the results with
  report_constraints -all_violators -verbose
  report_timing -max_paths 5    // report 5 worst case timings
- Tighten the clock with create_clock and execute a compile -incremental
- Never start with no clock specification or one that is way too tight or loose -- the initial optimization within Synopsys is significant in determining the final design
- Iterate on the create_clock/compile/report loop, saving the .db file whenever you get the best result
  - If only a few paths are creating setup violations, then you have a good chance of getting a better design.
Timing Optimization Within Synopsys

Scenario 2. You are having difficulty meeting required or desired timing

- After the first compile use
  
  ```
  report_constraints -all_violators -verbose
  report_timing -max_paths 5 // report 5 worst case timings
  ```
  to analyze your results

- Normally Synopsys concentrates on one critical path at a time, improving it until it is no longer critical, and then moving to another. If instead you tell it to concentrate on several paths simultaneously, it might produce a better results:
  
  ```
  group_path -name critical -critical_range 5 -to clock20
  // all paths within 5ns of the most critical path are given a higher weight of 1.0
  ```

- Modify the constraints to be more realistic
- Identify false paths and multicycle paths
- If you are close (usually 2ns) then throw the computer at it:
  
  ```
  compile incremental -map_effort high
  ```
- Re-assess your HDL code and design partitioning
- If using latches, consider cycle borrowing
Optimizing For Area

HDL Coding Style Techniques:

- **Automatic resource sharing**  e.g.
  
  ```
  always@(a or b or c or d or i)
  begin
    z = 0; y = 0;
    if (i) z = a + b + c;
    else y = a + b + d;
  end
  
  Synopsys will build 2 adders and a mux (on c/d), rather than 4 adders and bigger output muxes as long as resource sharing is turned on (the default) and the sharable code is in the **same procedural block**.

  → If the performance constraint is too tight, you might get 4 adders

- Not all resources can be shared ... see `HDL compiler for Verilog manual’ if sharing does not seem to be working.

- Minimize the number of flip-flops
  
  Everything on the LHS after a `always@posedge(clock)` becomes a flip-flop.
... Optimizing for Area

Synthesis Strategies:

- After synthesis, execute `check_design` to make sure there is no unused logic.
- Use `report_resources` and refer to HDL to see if all possible resource sharing has taken place.
- Check timing constraints as you might be over-constraining the design.
- `set_structure -boolean true`, before a compile might lead to a smaller design by turning on certain boolean minimizations.
- A carefully tuned synthesis strategy leads to a smaller design
  - Separately optimizing control (esp. FSMs) and datapath.

For Area or Speed:

- Design by hand (using GTECH cell library directly) and place a `set_dont_touch` on it.
Partitioning a Design

ie. Deciding what to put in each module.

- **Keep the critical path within one module**
  - Critical path = slowest path between flip-flops
- **Keep related combinational logic and sharable resources within one module**
  - e.g. combinational logic blocks that relate to each other
  - e.g. adders and other datapath units that can be potentially shared
    - put sharable resources in same procedural block!
- **Separate designs with different compile strategies into separate modules**
  - e.g. Separate speed-critical modules from area-critical modules
  - e.g. Separate random logic that you plan to use `set_flatten` on from ALU resources, such as adders
- `Chiplets’ with different clocks or that use different edges should be in different modules
  - Removes complexity of having to deal with timing constraints that go between different clocks and edges
\textbf{... Design Partitioning}

- Register all outputs, especially when interfacing with someone else’s design
  - Makes `set_driving_cell’ and `set_input_delay’ easy for others to determine
  - Try to register all module outputs even within your own design, or at least keep the logic depths at the outputs small

- Separate FSMs into different modules if planning to optimize

- Have registers in each synthesized module or module group

- Synthesize the smallest module or group of modules of consistent with the above guidelines
  - Designs larger than 250 to 2000 gates are difficult for Synopsys to optimize

- Write modules that meet these guidelines OR use `current_design’ and `group’ and `ungroupup’ commands to create suitably sized synthesis targets
  - However, Synopsys can not break up modules for you
  - Best strategy is to `right-size’ the modules in design

- Use `characterize’ in a hierarchical synthesis strategy
Partitioning Example

Notes:
- circles = combinational logic
- bar instanced twice as U2 and U3
...Partitioning Example

Write top level Verilog module (ignoring details of inputs and outputs):

```verilog
module top ();
```

```verilog
endmodule;
```

Synthesis Script Extract:
(instead of current compile):

```plaintext
.....
// on worst_case cells/conditions:
characterize -constraints {U1}
current_design = foo
current_design = top
group {U4 U5} -design_name pets -cell_name U10
characterize -constraints {U10}
current_design = pets
current_design = top
```

```plaintext
uniquify -cell U3 -new_name bar2
characterize -constraints {U2}
current_design = bar
compile
current_design = top
characterize -constraints {U3}
current_design = bar2
compile
current_design = top
report_timing
```
...Partitioning Example

Comments:

- Function of `characterize`
  - Determines all constraints (set_input_delay, etc.) of cell being characterized based on current design mapping

- How are the critical paths handled?

- Would this strategy lead to the smallest design?
  - What could you do to reduce the design size?
  - How could you avoid having to do this step at the RTL coding stage?

- Why did we `uniquify` U3?
  - If you do not uniquify all instances of bar will have identical mappings
Partitioning

What is wrong with this partitioning?

Critical path is shared between two modules

Wont be timing or area optimized

Solution:

bar
**Design Ware**

Synopsys, and others, provide libraries of carefully optimized design blocks for you to use -- called `Design Ware`

- Libraries include: Arithmetic, Advanced Math, DSP, Control, Sequential, and Fault Tolerant
- For +,-,*,>=,<=,>, and <, design ware is automatically used
- More complex cells must be inferred via a procedure call. e.g. cosine:

  ```
  module trigger (angle, cos_out);
  parameter wordlength1 = 8, wordlength2 = 8;
  input [wordlength-1:0] angle;
  output [wordlength-1:0] cos_out;

  // passes the widths to the cos function
  parameter angle_width = wordlength1, cos_width = wordlength2;
  `include "$SYNOPSYS/dw/dw02/src_ver/DW02_cos_function.inc"
  wire [wordlength2-1:0] cos_out;

  // infer DW02_cos
  assign cos_out = cos(angle);
  endmodule
  ```
Summary

- Combinational Logic implied by `always@(x ...)` or `assign` statements
  - Try to build unprioritized logic in a case statement when possible
  - Use structural Verilog (`assign`) to build muxes for smallest fastest design
- Finite State Machines
  - Describe and optimize separately from datapath and random logic
  - Consider style of implementation for best design
- Fastest speed and density
  - `always` starts with good knowledge about the design
  - Design capture in the HDL has a major impact
  - Synthesis strategy can help improve a design but it IS NOT A MAGIC BULLET.
- Use DesignWare for reuse of carefully optimized designs done by others
  - You can add to your own design ware library
Partitioning

Remember:

Partition the design into the **smallest modules** that
- Entirely contain the critical paths
- Have FFs for all outputs (as much as practical)
- Contain sharable logic
- Make sense from a design perspective