PURPOSE: In this lab you will learn how to set up optimization constraints & directives with the Synopsys toolset for optimizing a design. You will synthesize a simple decoder for minimum area first, and then you will ask to synthesize the same design to meet a specified time constraint. You will study the results.

Background

1. Optimizing (compiling) is the step in the synthesis process that attempts to implement a combination of library cells that meets the functional, speed, and area requirements of your design. Optimization transforms a design into a technology-specific circuit based on the attributes and constraints you place on the design. The quality of optimization results depends on how the HDL description is written. In particular, the partitioning of the hierarchy in the HDL, if done well, can enhance optimization.

2. Exploring the Design Space: Experimenting with speed and area to get the smallest or fastest design is called exploring the design space. Using Design Compiler, you can examine different implementations of the same design in a relatively short time. Figure 1 shows a design space curve. The shape of the curve demonstrates the trade-off between area-efficient and speed-efficient circuits.

3. Optimization Phases: The optimization process modifies the logic in a netlist. Optimization uses cells from the technology library in an attempt to meet specified constraints. Using various commands and options, you can direct Design Compiler to perform all or some combination of the following major phases as it optimizes your design:

   3.a. Combinational optimization, including
      • Technology-independent
• Technology-specific (usually called “mapping”)

3.b. Sequential optimization: initial and final

3.c. Local optimizations

These are described in detail next:

4. Combinational Optimization: This phase transforms the logic-level description of the combinational logic to a gate-level netlist. The two major phases in combinational optimization are:

4.a. Technology-independent optimization, which operates at the logic level. Design Compiler represents the gates as a set of Boolean logic equations. The optimization method applies algebraic and Boolean techniques on this set of logic equations. This optimization re-implements the logic equations to meet your timing and area goals. Optimization techniques used are flattening and structuring.

4.b. Technology-specific optimization, which operates at the gate level. Both the logic and the gate structures are important to the overall quality of the design. Technology-specific optimization synthesizes a gate-level design that attempts to meet your timing and area constraints. During mapping, Design Compiler selects components from the technology library to implement the logic structure. Design Compiler tries different logic combinations, using only components that approach the defined speed and area goals.

5. Initial Sequential Optimization maps sequential cells to cells in the library. You can map to either standard sequential cells or scan-equivalent cells. Initial sequential optimization is in the first phase of gate-level optimization. At this point in the optimization process, information about the delay through the combinational logic is incomplete. Design Compiler does not have enough information to select the optimum sequential cell. The tool can correct this lack of information later, in the final sequential optimization phase. Design Compiler optimizes the sequential cells, defining the following information:

• Locations of the islands of combinational logic between sequential cells
• Timing constraints on the combinational islands required to meet the setup and hold constraints on the sequential cells

6. Final Sequential Optimization: Design Compiler has accurate values for all delays through the I/O pads and combinational logic before it enters the final sequential optimization phase. (Final sequential optimization is part of the “Mapping Optimization” phase.) In this phase, Design Compiler optimizes timing-critical sequential cells (cells on the critical path). The tool examines each sequential cell
and its surrounding combinational logic to determine if they might be replaced by more-complex sequential cells from the target library.

7. **Local Optimizations**: The final step in gate-level optimization involves making local changes. Design Compiler makes incremental modifications to the design to adjust for timing or area.

8. There are two compilation options: Full & Incremental.
   8.a. **Full Compilation**: The full compilation pass is used to optimize an RTL design. Depending on the compile options and attributes you set, it can include flattening, structuring, and mapping. During the full compilation process, Design Compiler removes the existing gate structure from a design, then rebuilds the design. A full compilation performs both technology-independent optimization (including flattening, if enabled, and structuring) as well as technology-specific optimization (mapping). This enables more design improvements than an incremental compilation, which focuses only on the portions of the design that do not meet constraints. The compile command invokes the full compilation process.

   8.b. **Incremental Compilation**: An incremental compilation improves the existing design cost by focusing on the areas of the design that do not meet constraints. The existing structure is preserved if all constraints are already met. Incremental mapping uses the existing gates from an earlier compilation as a starting point for the mapping process. Mapping optimizations are accepted only if they improve the circuit speed or area. Incremental mapping guarantees that a circuit can only be improved. Unlike a full compilation, the design does not go through an initial flattening or structuring phase in an incremental compilation. In addition, portions of the design are restructured if that improves the design costs with high effort compile. The compile command, with its `-incremental_mapping` option invokes the incremental compilation process.

9. **Area Optimization**: Assuming that you have placed area constraints on your design, Design Compiler attempts to minimize the number of gates in the design. You can direct Design Compiler to put a low, medium, or high effort into area optimization. (If you do not place area constraints on your design, Design Compiler performs a limited series of downsizing and area cleanup steps.)
   9.a. **Low effort**: Design Compiler does gate sizing and buffer and inverter cleanup. Design Compiler allocates limited CPU time to this effort level.
9.b. **Medium effort**: Design Compiler adds phase assignment to gate sizing and buffer and inverter cleanup. Design Compiler allocates more CPU time to this effort than to a low effort optimization.

9.c. **High effort**: Design Compiler tries still more gate minimization strategies. The tool adds gate composition to the process and allocates even more CPU time. 

*Note*: Whichever area optimization effort level you choose, the overall constraints cost vector (described in the Design Compiler Reference Manual: Constraints and Timing) prevails. Even during area optimization, if Design Compiler finds a new opportunity to improve delay cost, it makes the change even if it increases area cost. Area always has a lower priority than delay.

10. **Verifying Design Functionality**: Logic verification determines whether two designs are functionally equivalent. Verification ensures that the synthesis process or manual design changes do not introduce errors. Timing considerations are ignored. You can use logic verification to determine whether a process, such as compilation, preserves the functionality of the original design.

11. **Controlling Logic-Level Optimization**: The goal of logic-level optimization is to improve a design’s logic structure. Design Compiler strives to reduce the number of product terms. As a first-order approximation, decreasing the number of product terms relates to both area and delay reduction. The optimization of logic equations does not affect a particular part of a function. Rather, it has a global effect on the overall area or speed characteristics of a design. Therefore, the logic optimization strategy you choose affects the optimized design. Modern digital designs are hierarchical, where the main (top-level) design is composed of interconnected blocks of logic. Some blocks fall along the data path; others are part of the control logic. Some blocks are highly structured (perhaps a carry-lookahead adder), and other pieces of the hierarchy consist primarily of random logic (such as an instruction decoder). Because the logic in each block in the sample is different, each hierarchical piece requires different treatment by Design Compiler.

- For structured blocks, preserve and build on the existing structure.
- For unstructured blocks (random and control logic), remove redundant gates (logic terms) and have Design Compiler improve the structure.

Logic optimization has two components: Flattening and Structuring.

12. **Flattening**: Flattening is an optional logic optimization step that removes all intermediate variables and uses Boolean distributive laws to remove all parentheses. Thus, flattening removes all logic structure from a design. Removing poor intermediate variables enables Design Compiler to choose more-efficient
subfunctions. A flattened design can be fast, because it consists of just two levels of combinational logic.

*Note:* Flattening is often mistaken for flattening the hierarchy of a design. Flattening the hierarchy is called “ungrouping the design” and is independent of the flatten option of compile command. A flattened design implies a design optimized with the flatten option turned on; a flattened design does not mean a design devoid of hierarchy. Flattening is off by default because timing-driven structuring (on by default) can consider the critical paths. Some designs might benefit from flattening applied with or without structuring. You can also obtain a two-level (sum of products) equation representation of a design using flattening only, with no structuring or mapping. The result of flattening is a two-level, sum-of-products form.

### Table 1 Flattening Examples

<table>
<thead>
<tr>
<th>Before flattening</th>
<th>After flattening</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{out} = t_1 \ t_2 )</td>
<td>( \text{out} = a \ t d + b d f + b f c + a e + b c e' + b c' f )</td>
</tr>
</tbody>
</table>

In general, you can flatten random control logic, because automatic structuring usually improves on manual structuring. Do not flatten regular or highly structured designs, such as adders and ALUs designed with an explicit structure.

Not all designs can be flattened. Small designs of 10 inputs or less can always be flattened. Large designs that have 20 or more inputs are often impossible to flatten. Designs with many XOR and multiplexer gates, such as adders and ALUs, are generally difficult to flatten. If flattening does not complete, Design Compiler issues a message and proceeds to the next step.

Flattening does not always yield the fastest design. In the case of a design with few inputs and numerous outputs, flattening can mean that some inputs fan out to many outputs, placing a large load on the inputs. Figure 2 shows the trade-off between flattening and no structuring for a circuit with a large input fanout versus flattening and structuring for a circuit with many stages. Although Design Compiler has built-in buffering algorithms, heavily loaded inputs might still slow the critical path unacceptably.
By default, compilation does not flatten a design. To flatten a design or a subdesign during optimization, set the flatten attribute to true on that design using the set_flatten command. When set on a hierarchical design, the flatten attribute by default is set only on the current design and not set on any of its subdesigns.

12.a. Enabling and Controlling Minimization: The minimization step reduces the number and size of the product (AND) and sum (OR) terms of the logic equations. Karnaugh maps and the Quine-McCluskey method are two popular techniques for manually performing this optimization. After flattening, Design Compiler can minimize the resulting equations. The -minimize option is available only if set_flatten is set to true. You can think of the minimization that occurs at this stage as resembling Karnaugh map reduction.

12.b. Enabling Phase Inversion: In addition to the primary minimization technique, you can use phase assignment. Phase assignment compares possible implementations of each logic equation for both the original equation and its complemented (negated) form.

13. Structuring: Structuring is a logic optimization step that adds intermediate variables and logic structure to a design. During structuring, Design Compiler searches for subfunctions that can be factored out, then evaluates these factors based on the size of the factor and the number of times the factor appears in the design. The subfunctions that most reduce the logic are turned into intermediate variables and factored out of the design equations. Design Compiler structures a design during compilation by default. Structuring provides a powerful way to improve the logic structure of your designs. However, the optimization algorithms used to implement this step are not guaranteed to find the best logic structure for your designs. These algorithms perform well for random control logic but less well for complex designs, such as adders and ALUs. Structuring is incremental. When you structure a design that already has structure (intermediate variables), Design Compiler builds on the existing logic structure. Design Compiler provides two structuring options:
13.a. Timing-driven structuring only is the default. Timing-driven structuring with no flattening is the recommended first strategy to apply to any design, structured or unstructured. Timing-driven structuring can improve critical paths without changing the structures on the noncritical paths. When you use timing-driven structuring, you must define accurate timing and clock constraints. If you define a maximum delay of zero, the compiled circuit can be unacceptably large, because timing-driven structuring duplicates logic paths, as necessary, to minimize delay.

13.b. Boolean optimization structuring: This structuring uses Boolean algebra to capture don’t care information and reduce circuit area. For example, \( a \cdot !a = 0 \), \( a + a = a \), and \( a + !a = 1 \). You can specify don’t care information in the HDL source code. Table 2 shows the results of structuring a simple set of equations. In this example, the subfunction \( t_0 \) is isolated as an intermediate variable and then factored out of the remainder of the design.

<table>
<thead>
<tr>
<th>Before Structuring</th>
<th>After Structuring</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_0 = a \cdot b + a \cdot c )</td>
<td>( f_0 = a \cdot \bar{t_0} )</td>
</tr>
<tr>
<td>( f_1 = b + c + d )</td>
<td>( f_1 = d + \bar{t_0} )</td>
</tr>
<tr>
<td>( f_2 = b' \cdot c' \cdot e )</td>
<td>( f_2 = t_0' \cdot e )</td>
</tr>
<tr>
<td>( \bar{t_0} = b + c )</td>
<td></td>
</tr>
</tbody>
</table>

14. Structuring Considerations: Keep in mind these facts about structuring. The result of structuring is that terms are shared. This produces an area-efficient design but might have a negative effect on the delay if there are no delay constraints on the design or if timing-driven structuring is turned off (set_structure - timing false). In the absence of delay constraints or timing driven structuring, structuring might produce additional levels of logic on the critical paths.

14.a. Structured Circuits: Structured circuits contain data-path components, MUXs, XORs, and regular logic. ALUs, adders, multipliers, and comparators all belong to this class. Because this type of circuit usually contains carefully crafted and highly structured logic, do not flatten it unless you also structure it. If you flatten a design without also structuring it, you might remove all or part
of the existing structure, resulting in a less than optimal design. When optimizing a structured design for area or speed, first consider using default options, such as timing-driven structuring and mapping.

14.b. **Unstructured Circuits**: Unstructured circuits contain no data-path components, few XORs or MUXs, and no ALUs or adders. Examples of unstructured logic are random control logic and table-lookup circuits that can be implemented as ROMs. When optimizing an unstructured design for area or speed, consider using the default options first.

15. **Guidelines for Structured Circuits**:

15.a. **Optimizing for Area**: When optimizing for area, you can obtain good area results by using the default optimization settings. If you do not get good results, use mapping only. For further area optimization, use Boolean optimization.

15.b. **Optimizing for Speed**: When optimizing for speed, experiment first with the default settings, structuring, and mapping. The default Timing-driven structuring option optimizes the critical paths for delay while maintaining area efficiency on the noncritical paths. If this produces unacceptable results, try mapping only, no structuring. Then try flattening with structuring. In general, do not flatten structured designs unless you follow with structuring. The flattening might remove all or part of the existing structures. However, if flattening is followed by structuring, timing-driven structuring often builds good structures that both reduce the area and improve the critical path delays.

16. **Guidelines for Unstructured Circuits**:

16.a. **Optimizing for Area**: When optimizing for area, you can probably obtain good area results using the default optimization settings. If you are dissatisfied with the results, try flattening with structuring. For further area optimization, try Boolean optimization.

16.b. **Optimizing for Speed**: When optimizing for speed, experiment first with the default settings, structuring, and mapping. If this produces unacceptable results, try flattening with structuring. If area is not an issue, try flattening without structuring. As discussed previously, flattening a design and then mapping it produces a tall, narrow design. If the inputs are not excessively loaded, this strategy might produce the best speed results, although it might take much longer to process large blocks. If the results from the previous strategies are unacceptable and Design Compiler was unable to flatten your
design using its low-effort, try the previous strategies with a greater flatten effort.

**Now to actual work**

17. We will synthesize a simple decoder. Create a directory lab4 under your labs directory and change into that. Type in the following Verilog code in a text editor and save it as `decode.v` under lab4. This is a straight forward Boolean description of a decoder.

```verilog
module dec5x32 (op, opsig);
input [4:0] op;
output [32:1] opsig;

assign opsig[1] = ((~op[4]) & (~op[3]) & (~op[2]) & (~op[1]) & (~op[0]));
assign opsig[2] = ((~op[4]) & (~op[3]) & (~op[2]) & (~op[1]) & op[0]);
assign opsig[17] = (op[4] & (~op[3]) & (~op[2]) & (~op[1]) & (~op[0]));
assign opsig[18] = (op[4] & (~op[3]) & (~op[2]) & (~op[1]) & op[0]);
```

Univ. of Minnesota          February 16, 2009          9 of 15
endmodule

18. Copy the file “.synopsys_dc.setup” from lab3, modify the line in that file and save it under lab4:
   define_design_lib LIB4 -path home/class/ee5327xx/labs/lab4/lib4 (use your userid)
   Delete “class.db” in set target_library and set link_library lines.
20. From within your lab4 directory, type the following command in the xterm:
   design_vision. The Design Vision window will open up.
21. Choose File->Setup from the menu of the Design Vision window. Your details, the link,
   target, symbol libraries and the search path should be setup properly. Close the window
   by choosing Cancel.
22. Open the Command Window (CW) by choosing Setup -> Command Window and resize it to
   fit below the Design Analyzer window.
23. To analyze the Verilog design, choose File -> Analyze. The Analyze File window
   appears. Select decoder.v Set format as Auto. Under library, choose LIB4. Click OK. The
   Analyze window appears and displays the activities of the analyze command. Resize this
   window and position it beside the other two windows.
24. Elaborate the design. To elaborate the top Verilog design,
   24.b. Scroll the Library list, and select LIB4. The Elaborate Design window displays the
         contents of the lib4 directory.
   24.c. Click Re-Analyze Out-Of-Date Libraries to select it.
24.d. From the Design list select dec5x32. Click OK. The Elaborate window appears and displays the activities of the elaborate command.

24.e. The Design View of the Design analyzer should now contain a gate icon with the following label: dec5x32.

25. Right click on the dec5x32 design in Hierarchy window to bring up its schematic view and use the Symbol View button on shortcut menu to generate the Symbol View. Consider this design simply as a black box with a set of inputs and a set of outputs. We are not concerned with its function. We simply want to study the effects of constraints on the final result.

26. Set the following attributes to the design.

Attributes:
- Drive strength (rise/fall) values op[4:0] = 0.05 (design_vision>set_drive 0.05 op)
- Load values opsig[32:1] = 5 (design_vision>set_load 5 opsig)
- Wire load model of dec5x32, estimated_max (cx4551_lib_max) (design_vision>set_wire_load_model-name estimated_max-library cx4551_lib_max)
- Operating conditions, quick_max(cx4551_lib_max) (design_vision>set_operating_conditions-library cx4551_lib_max quick_max)

27. Save the design as dec5x32.ddc to preserve your attribute settings. Ensure you save it to /labs/lab4 and Save All Designs in Hierarchy option is set to on. You will use this ddc and run various optimization schemes on it.

28. Removing Existing Constraints: In the lab, you have not yet defined constraints on dec5x32. Before you do, it is good practice to remove any existing constraints. To remove existing constraints, enter the following command in the Command Window:

remove_constraint -all.

The remove_constraint command does not remove the attributes that we set as part of the design environment.

29. Setting Clock & Delay Constraints: We are not going to do these in this lab, as our aim is only to compare and not arrive at absolute values. However in your project, you should not miss any of these steps.

30. Checking the Design for Errors: The Check Design and Check Timing commands are optional for synthesis. We will not use these in this lab. They will return errors as we have not set delay constraints.

31. Choose Attributes -> Optimization Directives -> Design. Check Structure logic and check Applying Timing Driven structuring, and choose OK.

32. To open the Design Optimization window, i) Select dec5x32 in the Designs view. ii) Choose Design -> Compile Design. The Design Optimization window appears. Note the default settings. Map Design and Medium Map Effort are selected. Use these default
settings. Click OK. After compilation finishes, Design Vision updates the Command Window.


34. To generate area and timing reports, i) Report area ii) Under Timing, click *Report Timing Path*. iii) Click OK.

35. Analyzing the Area Report: Note down the area and the critical path delay.

```
****************************************
Report : area
Design : dec5x32
Date   : Mon Feb 16 16:49:37 2009
****************************************

Library(s) Used:

cx4001_core_max (File: /home/vlsilab/synopsys/lib_cx4001/cx4001_core.max.db)
cx4551_lib_max (File: /home/vlsilab/synopsys/lib_cx4001/cx4551/cx4551.max.db)

Number of ports: 37
Number of nets: 87
Number of cells: 82
Number of references: 12

Combinational area: 242.500000
Noncombinational area: 0.000000
Net Interconnect area: undefined (Wire load has zero net area)

Total cell area: 242.500000
Total area: undefined

***** End Of Report *****
```
**Report : timing**

- path full
- delay max
- max_paths 1
- sort_by group

Design : dec5x32
Date : Mon Feb 16 16:51:10 2009

Operating Conditions: quick_max Library: cx4551_lib_max
Wire Load Model Mode: top

Startpoint: op[2] (input port)
Endpoint: opsig[5] (output port)
Path Group: (none)
Path Type: max

<table>
<thead>
<tr>
<th>Des/Clust/Port</th>
<th>Wire Load Model</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>dec5x32</td>
<td>estimated_max</td>
<td>cx4551_lib_max</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Point</th>
<th>Incr</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>input external delay</td>
<td>0.00</td>
<td>0.00 f</td>
</tr>
<tr>
<td>op[2] (in)</td>
<td>0.04</td>
<td>0.04 f</td>
</tr>
<tr>
<td>U126/Z (nd21)</td>
<td>1.51</td>
<td>1.55 r</td>
</tr>
<tr>
<td>U85/Z (iv10s)</td>
<td>0.29</td>
<td>1.84 f</td>
</tr>
<tr>
<td>U86/Z (iv10s)</td>
<td>1.33</td>
<td>3.16 r</td>
</tr>
<tr>
<td>U95/Z (nr20)</td>
<td>0.47</td>
<td>3.63 f</td>
</tr>
<tr>
<td>U56/Z (cdqm0)</td>
<td>0.21</td>
<td>3.84 f</td>
</tr>
</tbody>
</table>
of 15

36. Now let us apply timing constraint on the design. 5.23 ns is too long for us. We need the work to be done in 4.5 ns. Choose Attributes -> Optimization Constraints -> Timing Constraints. Note the critical path was from op(2) to opsig(5). So choose op(2) in the from field, and opsig(5) into field. Enter 4.5 in the Maximum Delay - Rise field and click Apply. Click Cancel. Optimize and get the reports. ( Or just type the commands set_max_delay 4.5 -rise -to opsig[5] -from op[2] )

Total cell area: 238.250000

-----------------------------------------------------------

data required time 4.50

data arrival time -2.32

-----------------------------------------------------------

slack (MET) 2.18

37. Flatten logic controls whether the flatten attribute is set on the current design. Select this option to enable flattening during design optimization. (Type the command like this: design_vision> set_flatten -effort low)

38. Optimize and get the reports. (These numbers will not be exactly the same when you perform optimization in the lab.)

Total cell area: ?

-----------------------------------------------------------

-----------------------------------------------------------

slack (MET) ?

-----------------------------------------------------------


Total cell area: ?

slack (MET) ?

40. Now try structuring again but add Boolean Optimization also.

design_vision> set_structure -boolean true

design_vision> set_structure -timing false
You should get

Total cell area:  

 slack (MET)  

41. You can see how the area and speed of the synthesized design depends on the optimization directives that you apply. Table all the results and analyze. Note that you might get slightly different results if you have not followed the steps exactly. Also the circuit that we analyzed was a simple, homogeneous one. In an actual design with dissimilar circuits, the differences in synthesis results may not be very marked as here.

42. We have also showed the commands that get echoed in the Command Window while setting the directives using the menu. You can follow the syntax and write your own script for your project. Refer the online Design Vision Tutorial by going to the menu Help -> Online Help .

43. If you need to optimize a subdesign rigorously like this for your project, then proceed till you get satisfactory results. Save the subdesign as a .db file. Later, when you optimize the top design, set the don’t touch attribute on the subdesign. That is all for today.

**Submission**

Schematic view, area and timing reports of various optimization directives that you tried in the lab. Summarize data (Area and time slack for each step 36, 38, and 40) into a table. Some analysis and conclusions based on the results.