

**Guidelines
For
Design Synthesis
Using
Synopsys Design Compiler**

**Department of Computer Science Engineering
University of South Carolina
Columbia**

Revised: December 2000. By: Sreesa Akella

Design Synthesis

Introduction

One of the most important steps in ASIC design is the synthesis phase. Synthesis is an automatic method of converting a higher level of abstraction to a lower level of abstraction. In other words the synthesis process converts Register Transfer Level (RTL) descriptions to gate-level netlists. These gate-level netlists can be optimized for area, speed, testability, etc. The synthesis process is shown in Fig 1.0.

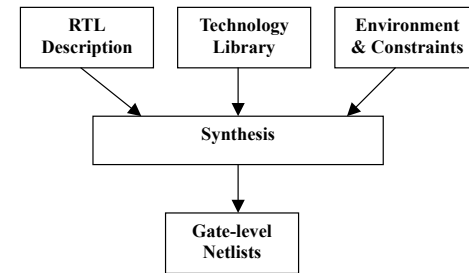


Figure 1. Synthesis process overview

The inputs to the synthesis process are RTL HDL description, circuit constraints and attributes for the design, and a technology library. The synthesis process produces an optimized gate-level netlist from all these inputs. Synthesizing a design is an iterative process and begins with defining the constraints for each block of the design. In addition to these constraints, a file defining the synthesis environment is also needed. The environment file specifies the technology cell libraries and other relevant information that the tool uses during synthesis.

Synthesis Environment

The most commonly used synthesis tool in the ASIC industry is *Synopsys Design Compiler*. Design constraints and other synthesis options are given as commands or default settings to the tool.

Synopsys synthesis tool is divided into two sections, *Design Analyzer* and *Design Compiler*. The former is the graphical interface and the latter is the command shell interface to the synthesis tool. These two interfaces can be invoked using the following commands.

Graphical interface, Design analyzer:
unix_prompt> design_analyzer

Shell Interface, Design compiler:
unix_prompt> dc_shell

Startup File

The Synopsys synthesis tool when invoked, either through Design analyzer or Design compiler command, reads a startup file, which must be present in the current working directory. This startup file is **.synopsys_dc.setup** file. There should be two startup files present, one in the current working directory and other in the root directory in which Synopsys is installed. The local startup file in the current working directory should be used to specify individual design specifications. This file does not contain design dependent data. Its function is to load the Synopsys technology independent libraries and other parameters. The user in the startup files specifies the design dependent data. The settings provided in the current working directory override the ones specified in the root directory.

There are four important parameters that should be setup before one can start using the tool. They are:

- **search_path**
This parameter is used to specify the synthesis tool all the paths that it should search when looking for a synthesis technology library for reference during synthesis.
- **target_library**
The parameter specifies the file that contains all the logic cells that should be used for mapping during synthesis. In other words, the tool during synthesis maps a design to the logic cells present in this library.
- **symbol_library**
This parameter points to the library that contains the “visual” information on the logic cells in the synthesis technology library. All logic cells have a symbolic representation and information about the symbols is stored in this library.
- **link_library**
This parameter points to the library that contains information on the logic gates in the synthesis technology library. The tool uses this library solely for reference but does not use the cells present in it for mapping as in the case of **target_library**.

An example on use of these four variables from a **.synopsys_dc.setup** file is given below.

```
search_path = “./synopsys/libraries/syn/cell_library/libraries/syn”  
target_library = class.db  
link_library = class.db  
symbol_library = class.db
```

Once these variables are setup properly, one can invoke the synthesis tool at the command prompt using any of the commands given for the two interfaces.

Design Objects

There are eight different types of objects categorized by Design Compiler.

Design: It corresponds to the circuit description that performs some logical function. The design may be stand-alone or may include other sub-designs. Although sub-design may be part of the design, it is treated as another design by the Synopsys.

Cell: It is the instantiated name of the sub-design in the design. In Synopsys terminology, there is no differentiation between the cell and instance; both are treated as cell.

Reference: This is the definition of the original design to which the cell or instance refers. For e.g., a leaf cell in the netlist must be referenced from the link library, which contains the functional description of the cell. Similarly an instantiated sub-design must be referenced in the design, which contains functional description of the instantiated sub-design.

Ports: These are the primary inputs, outputs or IO’s of the design.

Pin: It corresponds to the inputs, outputs or IO’s of the cells in the design. (Note the difference between port and pin)

Net: These are the signal names, i.e., the wires that hook up the design together by connecting ports to pins and/or pins to each other.

Clock: The port or pin that is identified as a clock source. The identification may be internal to the library or it may be done using **dc_shell** commands.

Library: Corresponds to the collection of technology specific cells that the design is targeting for synthesis; or linking for reference.

Design Entry

Before synthesis, the design must be entered into the Design Compiler or Design Analyzer (referred to as DC/DA from now on) in the RTL format. DC/DA provides the following two methods of design entry:

read command
analyze & elaborate commands

The **analyze & elaborate** commands are two different commands, allowing designers to initially analyze the design for syntax errors and RTL translation before building the generic logic for the design. The generic logic or GTECH components are part of Synopsys generic technology independent library. They are unmapped representation of boolean functions and serve as placeholders for the technology dependent library.

The **analyze** command also stores the result of the translation in the specified design library that maybe used later. So a design analyzed once need not be analyzed again and can be merely elaborated, thus saving time. Conversely **read** command performs the function of **analyze** and **elaborate** commands but does not store the analyzed results, therefore making the process slow by comparison.

Parameterized designs (such as usage of *generic* statement in VHDL) must use the **analyze** and **elaborate** commands in order to pass required parameters, while elaborating the design. The **read** command should be used for entering pre-compiled designs or netlists in DC/DA.

One other major difference between the two methods is that, in **analyze** and **elaborate** design entry of a design in VHDL format, one can specify different architectures during elaboration for the same analyzed design. This option is not available in the **read** command.

The commands used for both the methods in DC are as given below:

Read command:

```
dc_shell>read -format <format> <list of file names>
```

“-format” option specifies the format in which the input file is in, e.g. VHDL

Sample command for a reading “adder.vhd” file in VHDL format is given below

```
dc_shell>read -format vhdl adder.vhd
```

Analyze and Elaborate commands:

```
dc_shell>analyze -format <format> <list of file names>
```

```
dc_shell>elaborate <.syn file> -arch “<architecture >” -param “<parameter>”
```

.syn file is the file in which the analyzed information of the design analyzed is stored.

e.g: The adder entity in the adder.vhd has a generic parameter “width” which can be specified while elaboration. The architecture used is “beh” defined in the adder.vhd file.

The commands for analyze and elaborate are as given below:

```
dc_shell> analyze -format vhdl adder.vhd
```

```
dc_shell> elaborate adder -arch “beh” -param “width = 32”
```

Technology Library

Technology libraries contain the information that the synthesis tool needs to generate a netlist for a design based on the desired logical behavior and constraints on the design. The tool referring to the information provided in a particular library would make appropriate choices to build a design. The libraries contain not only the logical function of an ASIC cell, but the area of the cell, the input-to-output timing of the cell, any constraints on fanout of the cell, and the timing checks that are required for the cell. Other information stored in the technology library may be the graphical symbol of the cell for use in creating the netlist schematic.

The **target_library**, **link_library**, and **symbol_library** parameters in the **start-up** file are used to set the technology library for the synthesis tool.

The Synopsys® .lib technology library contains the following information

- Wire-load models for net length and data estimation. Wire-load models available in the technology library are statistical and hence inaccurate when estimating data.
- Operating Conditions along with scaling k-factors for different delay components to model the effects of temperature, process, and voltage on the delay numbers.
- Specific delay models like piece-wise linear, non-linear, cmos2 etc. for calculation of delay values.

For each of the technology primitive cells the following information is modeled

- Interface pin names, direction and other information.
- Functional descriptions for both combinational and sequential cells which can be modeled in Synopsys®
- Pin capacitance and drive capabilities
- Pin to pin timing
- Area

Register Transfer-Level Description

A Register Transfer-Level description is a style that specifies a particular design in terms of registers and combinational logic in between. This is shown by the “register and cloud” diagram in Fig 2.0

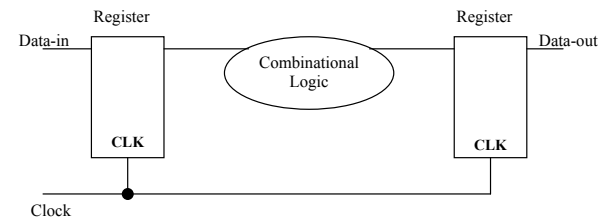


Figure 2. Register and cloud diagram

The registers can be described explicitly, through component instantiation, or implicitly, through inference. The combinational logic is described either by logical equations, sequential control statements (CASE, IF then ELSE, etc.), subprograms, or through concurrent statements and are represented by cloud objects in figure 2.0 between the registers.

RTL is the most popular form of high-level design specification. A good coding style would help the synthesis tool generate a design with minimal area and maximum performance.

General Guidelines

Following are given some guidelines which if followed might improve the performance of the synthesized logic, and produce a cleaner design that is suited for automating the synthesis process.

- Clock logic including clock gating and reset generation should be kept in one block – to be synthesized once and not touched again. This helps in a clean specification of the clock constraints. Another advantage is that the modules that are being driven by the clock logic can be constrained using the ideal clock specifications
- No glue logic at the top: The top block is to be used only for connecting modules together. It should not contain any combinational glue logic. This removes the time consuming top-level compile, which can now be simply stitched together without undergoing additional synthesis.
- Module name should be same as the file name and one should avoid describing more than one module or entity in a single file. This avoids any confusion while compiling the files and during the synthesis.
- While coding finite state machines, the state names should be described using the enumerated types. The combinational logic for computing the next state should be in its own process, separate from the state registers. Implement the next-state combinational logic with a case statement. This helps in optimizing the logic much better and results in a cleaner design.
- Incomplete sensitivity lists must be avoided as this might result in simulation mismatches between the source RTL and the synthesized logic.
- Memory elements, latches and flip-flops: A latch is inferred when an incomplete *if* statement with a missing *else* part is specified. A flip-flop or a register as it is referred to, is inferred when an edge sensitive statement is specified in the process body. A latch is more troublesome than a flip-flop as it makes static timing analysis on designs containing latches. So designers try to avoid latches and prefer flip-flops more to latches.
- Multiplexer Inference: A *case* statement is used for implementing multiplexers. To prevent latch inferences in case statements the *default* part of the *case* statement should always be specified. On the other hand an *if* statement is used for writing priority encoders. Multiple *if* statements with multiple branches result in the creation of a priority encoder structure.

```
Ex: process(A, B, C)
begin
    if A= 0 then D = B; end if;
    if A= 1 then D = C; end if;
end process;
```

The above example infers a priority encoder with the first *if* statement given the precedence. The same code can be written using a *case* statement to implement a multiplexer as follows.

```
process(A, B, C)
begin
    case (A) is
        when 0    => D = B;
        when others => D = C;
    end case;
end process;
```

The same code can be written using *if* statement along with *elsif* statements to cover all possible branches.

- Three state buffers: A tri-state buffer is inferred whenever a high impedance (Z) is assigned to an output. Tri-state logic is generally not always recommended because it reduces testability and is difficult to optimize – since it cannot be buffered.
- Signals versus Variables in VHDL: Signal assignments are order independent, i.e. the order in which they are placed within the process statement does not have any effect on the order in which they are executed as all the signal assignments are done at the end of the process. The variable assignments on the other hand are order dependent. The signal assignments are generally used within the sequential processes and variable assignments are used within the combinational processes.

Design Attributes and Constraints

A designer, in order to achieve optimum results, has to methodically constrain the design, by describing the design environment, target objectives and design rules. The constraints contain timing and/or area information, usually derived from the design specifications. The synthesis tool uses these constraints to perform synthesis and tries to optimize the design with the aim of meeting target objectives.

Design Attributes

Design attributes set the environment in which a design is synthesized. The attributes specify the process parameters, I/O port attributes, and statistical wire-load models. The most common design attributes and the commands for their setting are given below:

Load: Each output can specify the drive capability that determines how many loads can be driven within a particular time. Each input can have a load value specified that determines how much it will slow a particular driver. Signals that are arriving later than the clock can have an attribute that specifies this fact. The load attribute specifies how much capacitive load exists on a particular output signal. The load value is specified in the units of the technology library in terms of picofarads or standard loads, etc... The command for setting this attribute is given below:

set_load <value> <object_list>
e.g. **dc_shell> set_load 1.5 x_bus**

Drive: The drive specifies the drive strength at the input port. It is specified as a resistance value. This value controls how much current a particular driver can source. The larger a driver is, i.e 0 resistance, the faster a particular path will be, but a larger driver will take more area, so the designer needs to trade off speed and area for the best performance. The command for setting the drive for a particular object is given below

set_drive <value> <object_list>
e.g. **dc_shell> set_drive 2.7 ybus**

Design Constraints

Design constraints specify the goals for the design. They consist of area and timing constraints. Depending on how the design is constrained the DC/DA tries to meet the set objectives. Realistic specification is important, because unrealistic constraints might result in excess area, increased power and/or degrading in timing. The basic commands to constrain the design are

set_max_area: This constraint specifies the maximum area a particular design should have. The value is specified in units used to describe the gate-level macro cells in the technology library.

e.g. **dc_shell> set_max_area 0**

Specifying a 0 area might result in the tool to try its best to get the design as small as possible

create_clock: This command is used to define a clock object with a particular period and waveform. The **-period** option defines the clock period, while the **-waveform** option controls the duty cycle and the starting edge of the clock. This command is applied to a pin or port, object types.

Following example specifies that a port named CLK is of type "clock" that has a period of 40 ns, with 50% duty cycle. The positive edge of the clock starts at time 0 ns, with the falling edge occurring at 20 ns. By changing the falling edge value, the duty cycle of the clock may be altered.

e.g. **dc_shell> create_clock -period 40 -waveform {0 20} CLK**

set_dont_touch_network: This is a very important command, usually used for clock networks and resets. This command is used to set a **dont_touch** property on a port, or on the net. Note setting this property will also prevent DC from buffering the net. In addition any gate coming in contact with the "don't_touch" net will also inherit the attribute.

e.g. **dc_shell> set_dont_touch_network {CLK, RST}**

set_dont_touch: This is used to set a don_touch property on the **current_design**, cells, references, or nets. This command is frequently used during hierarchical compilation of blocks for preventing the DC from optimizing the don't_touch object.

e.g. **dc_shell> set_dont_touch current_design**

current_design is the variable referencing the current working design. It can be set using the **current_design** command as follows

dc_shell>current_design <design_name>

set_input_delay: It specifies the input arrival time of a signal in relation to the clock. It is used at the input ports, to specify the time it takes for the data to be stable after the clock edge. The timing specification of the design usually contains this information, as the setup/hold time requirements for the input signals. From the top-level timing specifications the sub-level timing specifications may also be extracted.

e.g. **dc_shell> set_input_delay -max 23.0 -clock CLK {datain}**

dc_shell> set_input_delay -min 0.0 -clock CLK {datain}

The CLK has a period of 30 ns with 50% duty cycle. For the above given specification of max and min input delays for the datain with respect to CLK, the setup-time requirement for the input signal datain is 7ns, while the hold-time requirement is 0ns.

set_output_delay: This command is used at the output port, to define the time it takes for the data to be available before the clock edge. This information is usually is provided in the timing specification.

e.g. **dc_shell> set_output_delay - max 19.0 -clock CLK {dataout}**

The CLK has a period of 30 ns with 50% duty cycle. For the above given specification of max output delay for the dataout with respect to CLK, the data is valid for 11 ns after the clock edge.

set_max_delay: It defines the maximum delay required in terms of time units for a particular path. In general it is used for blocks that contain combination logic only. However it may also be used to constrain a block that is driven by multiple clocks, each with a different frequency. This command has precedence over DC derived timing requirements.

e.g. **dc_shell> set_max_delay 5 -from all_inputs() - to_all_outputs()**

set_min_delay: It defines the minimum delay required in terms of time units for a particular path.. It is the opposite of the set_max_delay command. This command has precedence over DC derived timing requirements.

e.g. **dc_shell> set_max_delay 3 -from all_inputs() - to_all_outputs()**

Optimizing Designs

A fully optimized design is one, which has met the timing requirements and occupies the smallest area. The optimization can be done in two stages one at the code level, the other during synthesis. The optimization at the code level involves modifications to RTL code that is already been simulated and tested for its functionality.

This level of modifications to the RTL code is generally avoided as sometimes it leads to inconsistencies between simulation results before and after modifications. However, there are certain standard model optimization techniques that might lead to a better synthesized design.

Model Optimization

Model optimizations are important to a certain level, as the logic that is generated by the synthesis tool is sensitive to the RTL code that is provided as input. Different RTL codes generate different logic. Minor changes in the model might result in an increase or decrease in the number of synthesized gates and also change its timing characteristics.

A logic optimizer reaches different endpoints for best area and best speed depending on the starting point provided by a netlist synthesized from the RTL code. The different starting points are obtained by rewriting the same HDL model using different constructs. Some of the optimizations, which can be used to modify the model for obtaining a better quality design, are listed below.

Resource Allocation

This method refers to the process of sharing a hardware resource under mutually-exclusive conditions. Consider the following *if* statement.

```

if A = '1' then
    E = B + C;
else
    E = B + D;
end if;

```

The above code would generate two ALUs one for the addition of B+C and other for the addition B + D which are executed under mutually exclusive conditions. Therefore a single ALU can be shared for both the additions. The hardware synthesized for the above code is given below in Figure 3 (a).

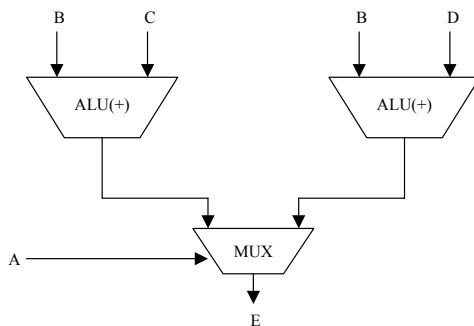


Figure 3 (a). Without resource allocation.

The above code is rewritten with only one addition operator being employed. The hardware synthesized is given in Figure 3(b).

```

if A = '1' then
    temp := C; // A temporary variable introduced.
else
    temp := D;
end if;
E = B + temp;

```

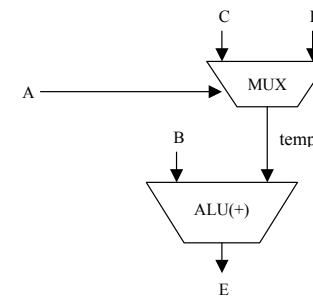


Figure 3 (b). With resource allocation.

It is clear from the figure that one ALU has been removed with one ALU being shared for both the addition operations. However a multiplexer is induced at the inputs of the ALU that contributes to the path delay. Earlier the timing path of the select signal goes through the multiplexer alone, but after resource sharing it goes through the multiplexer and the ALU datapath, increasing its path delay. However due to resource sharing the area of the design has decreased. This is therefore a trade-off that the designer may have to make. If the design is timing-critical it would be better if no resource sharing is performed.

Common sub-expressions and Common factoring

It is often useful to identify common subexpressions and to reuse the computed values wherever possible. A simple example is given below.

```

B := R1 + R2;
....
C <= R3 - (R1 + R2);

```

Here the subexpression R1 + R2 in the signal assignment for C can be replaced by B as given below. This might generate only one adder for the computation instead of two.

```

C <= R3 - B;

```

Common factoring is the extraction of common subexpressions in mutually-exclusive branches of an *if* or *case* statement.

```
if (test)
  A <= B & (C + D);
else
  J <= (C + D) | T;
end if;
```

In the above code the common factor C + D can be placed out of the *if* statement, which might result in the tool generating only one adder instead of two as in the above case.

```
temp := C + D; // A temporary variable introduced.
if (test)
  A <= B & temp;
else
  J <= temp | T;
end if;
```

Such minor changes if made by the designer can cause the tool to synthesize better logic and also enable it to concentrate on optimizing more critical areas.

Moving Code

In certain cases an expression might be placed, within a *for/while* loop statement, whose value would not change through every iteration of the loop. Typically a synthesis tool handles the a *for/while* loop statement by unrolling it the specified number of times. In such cases redundant code might be generated for that particular expression causing additional logic to be synthesized. This could be avoided if the expression is moved outside the loop, thus optimizing the design. Such optimizations performed at a higher level, that is, within the model, would help the optimizer to concentrate on more critical pieces of the code. An example is given below.

```
C := A + B;
.....
for c in range 0 to 5 loop
.....
  T := C - 6;
  // Assumption : C is not assigned a new value within the loop, thus the above
  expression would remain constant on every iteration of the loop.
.....
end loop;
```

The above code would generate six subtracters for the expression when only one is necessary. Thus by modifying the code as given below we could avoid the generation of unnecessary logic.

```
C := A + B;
.....
temp := C - 6; // A temporary variable is introduced

for c in range 0 to 5 loop
.....
  T := temp;
  // Assumption : C is not assigned a new value within the loop, thus the above
  expression would remain constant on every iteration of the loop.
.....
end loop;
```

Constant folding and Dead code elimination

There are possibilities where the designer might leave certain expressions which are constant in value. This can be avoided by computing the expressions instead of the implementing the logic and then allowing the logic optimizer to eliminate the additional logic.

```
Ex:
  C := 4;
  ....
  Y = 2 * C;
```

Computing the value of Y as 8 and assigning it directly within your code can avoid the above unnecessary code. This method is called constant folding.

The other optimization, dead code elimination refers to those sections of code, which are never executed.

```
Ex.
A := 2;
B := 4;
if(A > B) then
.....
end if;
```

The above *if* statement would never be executed and thus should be eliminated from the code.

The logic optimizer performs these optimizations by itself, but nevertheless if the designer optimizes the code accordingly the tool optimization time would be reduced resulting in faster tool running times.

Flip-flop and Latch optimizations

Earlier in the RTL code section, it has been described how flip-flops and latches are inferred through the code by the synthesis tool. However there are only certain cases where the inference of the above two elements is necessary. The designer thus should try to eliminate all the unnecessary flip-flop and latch elements in the design. Placing only the clock sensitive signals under the edge sensitive statement can eliminate the

unnecessary flip-flops. Similarly the unwanted latches can be avoided by specifying the values for the signals under all conditions of an *if/case* statement.

Using Parentheses.

The usage of parentheses is critical to the design as the correct usage might result in better timing paths.

Ex.

$$\text{Result} \leq R1 + R2 - P + M;$$

The hardware generated for the above code is as given below in Figure 4 (a).

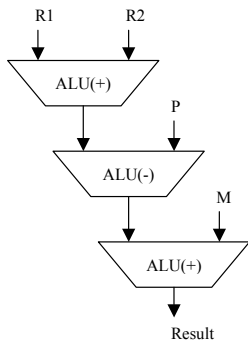


Figure 4 (a) Without using parentheses

If the expression has been written using parentheses as given below, the hardware synthesized would be as given in Figure 4 (b).

$$\text{Result} \leq (R1 + R2) - (P - M);$$

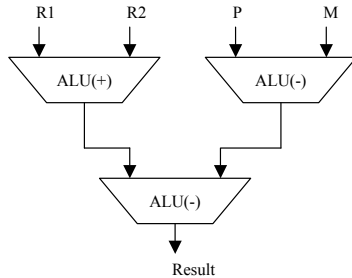


Figure 4 (b). After using parentheses

It is clear that after using the parentheses the timing path for the datapath has been reduced as it does not need to go through one more ALU as in the earlier case.

Partitioning and structuring the design.

A design should always be structured and partitioned as it helps in reducing design complexity and also improves the synthesis run times since it smaller sub blocks synthesis synthesizes faster. Good partitioning results in the synthesis of a good quality design. General recommendations for partitioning are given below.

- Keep related combinational logic in the same module
- Partition for design reuse.
- Separate modules according to their functionality.
- Separate structural logic from random logic.
- Limit a reasonable block size (perhaps a maximum of 10K gates per block).
- Partition the top level.
- Do not add glue-logic at the top level.
- Isolate state-machine from other logic.
- Avoid multiple clocks within a block.
- Isolate the block that is used for synchronizing the multiple clocks.

Optimization using Design Compiler/Design Analyzer

For the optimization of design, to achieve minimum area and maximum speed, a lot of experimentation and iterative synthesis is needed. The process of analyzing the design for speed and area to achieve the fastest logic with minimum area is termed – design space exploration.

For the sake of optimization, changing of HDL code may impact other blocks in the design or test benches. For this reason, changing the HDL code to help synthesis is less desirable and generally is avoided. It is now the designer’s responsibility to minimize the area and meet the timing requirements through synthesis and optimization. The later versions of DC, starting from DC98 have their compile flow different from previous versions. In the DC98 and later versions the timing is prioritized over area. Another difference is that DC98 performs compilation to reduce “total negative slack” instead of “worst negative slack”. This ability of DC98 produces better timing results but has some impact on area. Also DC98 requires designers to specify area constraints explicitly as opposed to the previous versions that automatically handled area minimization. Generally some area cleanup is performed but better results are obtained when constraints are specified.

The DC has three different compilation strategies. It is up to user discretion to choose the most suitable compilation strategy for a design.

- a) Top-down hierarchical compile method.
- b) Time-budget compile method.
- c) Compile-characterize-write-script-recompile (CCWSR) method.

Top-down hierarchical Compile

Prior to the release of DC98 this method was used to synthesize small designs as this method was extremely memory intensive and took a lot of time for large designs. In this method the source is compiled by reading the entire design with constraints and attributes applied, only at the top level.

DC98 provided Synopsys the capability to synthesize million gate designs by tackling much larger blocks (>100K) at a time. This approach is feasible for some designs depending on the design style (single clock etc.) and other factors. One may use this technique to synthesize larger blocks at a time by grouping the sub-blocks together and flattening them to improve timing.

Advantages

- Only top level constraints are needed.
- Better results due to optimization across entire design.

Disadvantages

- Long compile time.
- Incremental changes to the sub-blocks require complete re-synthesis.
- Does not perform well, if design contains multiple clocks or generated clocks.

Time-budgeting compile.

This process is best for designs properly partitioned designs with timing specifications defined for each sub-block. Due to specifying of timing requirements for each block, multiple synthesis scripts for individual blocks are produced. The synthesis is usually performed bottom-up i.e., starting at the lowest level and going up to the top most level. This method is useful for medium to very large designs and does not require large amounts memory.

Advantages

- Design easier to manage due to individual scripts.
- Incremental changes to sub-blocks do not require complete re-synthesis.
- Can be used for any style of design, e.g. multiple and generated clocks.

Disadvantages

- Difficult to keep track of multiple scripts.
- Critical paths seen at top level may not be critical at lower level.
- Incremental compilations may be needed for fixing DRC's.

Compile-Characterize-Write-Script-Recompile

This is an advanced synthesis approach, useful for medium to very large designs that do not have good inter-block specifications defined. It requires constraints to be applied at the top level of the design, with each sub-block compiled beforehand. The sub-blocks are then characterized using the top-level constraints. This in effect propagates the required timing information from the top-level to the sub-blocks. Performing a **write_script** on the characterized sub-blocks generates the constraint file for each sub-block. The constraint files are then used to re-compile each block of the design.

Advantages

- Less memory intensive.
- Good quality of results because of optimization between sub-blocks of the design.
- Produces individual scripts, which may be modified by the user.

Disadvantages

- The generated scripts are not easily readable.
- It is difficult to achieve convergence between blocks
- Lower block changes might need complete re-synthesis of entire design.

Resolving Multiple instances

Before proceeding for optimization, one needs to resolve multiple instances of the sub-block of your design. This is a necessary step as Dc does not permit compilation until multiple instances are resolved.

Ex: Lets say moduleA has been synthesized. Now moduleB that has two instantiations of moduleA as U1 and U2 is being compiled. The compilation will be stopped with an error message stating that moduleA is instantiated 2 times in moduleB. There are two methods of resolving this problem. You can set a **don_touch** attribute on moduleA before synthesizing moduleB, or **uniquify** moduleB. **uniquify** a **dc_shell** command creates unique definitions of multiple instances. So if for the above case it generates moduleA-u1 and moduleA_u2 (in VHDL), corresponding to instance U1 and U2 respectively.

Optimization Techniques

Various optimization techniques that help in achieving better area and speed for your design are given below.

Compile the design

The compilation process maps the HDL code to actual gates specified from the target library. This is done through the **compile** command. The syntax is given below

```
compile -map_effort <low | medium | high>
-incremental_mapping
-in_place
-no_design_rule | -only_design_rule
-scan
```

The compile command by default uses the `-map_effort medium` option. This usually produces the best results for most of the designs. It also default settings for the structuring and flattening attributes. The `map_effort high` should only be used, if target objectives are not met through default compile.

The `-incremental_mapping` is used only after initial compile as it works only at gate-level. It is used to improve timing of the logic.

Flattening and structuring

Flattening implies reducing the logic of a design to a 2-level AND/OR representation. This approach is used to optimize the design by removing all intermediate variables and parenthesis. This option is set to “false” by default.

The optimization is performed in two stages. The first stage involves the flattening and structuring and the second stage involves mapping of the resulting design to actual gates, using mapping optimization techniques.

Flattening

Flattening reduces the design logic in to a two level, sum-of-products of form, with few logic levels between the input and output. This results in faster logic. It is recommended for unstructured designs with random logic. The flattened design then can be structured before final mapping optimization to reduce area. This is important as flattening has significant impact on area of the design.

In general one should compile the design using default settings (flatten and structure are set as false). If timing objectives are not met flattening and structuring should be employed. If the design is still failing goals then just flatten the design without structuring it. The command for flattening is given below

```
set_flatten <true | false>
    -design <list of designs>
    -effort <low | medium | high>
    -phase <true | false>
```

The `-phase` option if set to true enables the DC to compare the logic produced by inverting the equation versus the non-inverted form of the equation.

Structuring

The default setting for this is “true”. This method adds intermediate variables that can be factored out. This enables sharing of logic that in turn results in reduction of area. For ex.

Before structuring	After structuring
$P = ax + ay + c$	$P = aI + c$
$Q = x + y + z$	$Q = I + z$
	$I = x + y$

The shared logic generated might effect the total delay of the logic. Thus one should be careful enough to specify realistic timing constraints, in addition to using default settings.

Structuring can be set for timing(default) or Boolean optimization. The latter helps in reducing area, but has a greater impact on timing. Thus circuits that are timing sensitive should not be structured for Boolean optimization. Good examples for Boolean optimization are random logic structures and finite state machines. The command for structuring is given below.

```
set_structure <true | false>
    -design <list of designs>
    -boolean <low | medium | high>
    -timing <true | false>
```

If the design is not timing critical and you want to minimize for area only, then set the area constraints (`set_max_area 0`) and perform Boolean optimization. For all other case structure with respect to timing only.

Removing hierarchy

DC by default maintains the original hierarchy that is given in the RTL code. The hierarchy is a logic boundary that prevents DC from optimizing across this boundary. Unnecessary hierarchy leads to cumbersome designs and synthesis scripts and also limits the DC optimization within that boundary, without optimizing across hierarchy.

To allow DC to optimize across hierarchy one can use the following commands.

```
dc_shell> current_design <design name>
```

```
dc_shell> ungroup -flatten -all
```

This allows the DC to optimize the logic separated by boundaries as one logic resulting in better timing and an optimal solution.

Optimizing for Area

DC by default tries to optimize for timing. Designs that are not timing critical but area intensive can be optimized for area. This can be done by initially compiling the design with specification of area requirements, but no timing constraints. In addition, by using the `don_touch` attribute on the high-drive strength gates that are larger in size, used by default to improve timing, one can eliminate them, thus reducing the area considerably.

Once the design is mapped to gates, the timing and area constraints should again be specified (normal synthesis) and the design re-compiled incrementally. The incremental compile ensures that DC maintains the previous structure and does not bloat the logic unnecessarily.

The following points can be kept in mind for further area optimization:

- 1) Bind all combinational logic as much as possible. If combinational logic were spread over different blocks of the design the optimization of the logic would not be perfect resulting in large areas. So better partitioning of the design with combinational logic not spread out among different blocks would result in better area.
- 2) At the top level avoid any kind of glue logic. It is better to incorporate glue logic in one of the sub-components thus letting the tool to optimize the logic better.

Timing issues

There are two kind of timing issues that are important in a design- setup and hold timing violations.

Setup Time: It indicates the time before the clock edge during which the data should be valid i.e. it should be stable during this period and should not change. Any change during this period would trigger a setup timing violation. Figure 5(a) illustrates an example with setup time equal to 2 ns. This means that signal DATA must be valid 2 ns before the clock edge; i.e. it should not change during this 2ns period before the clock edge.

Hold Time: It indicates the time after the clock edge during which the data should be held valid i.e. it should not change but remain stable. Any change during this period would trigger a hold timing violation. Figure 5(b) illustrates an example with hold time equal to 1 ns. This means that signal DATA must be held valid 1 ns after the clock edge; i.e. it should not change during the 1 ns period after the clock edge.

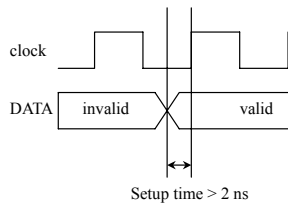


Figure 5(a) Timing diagram for setup time on DATA

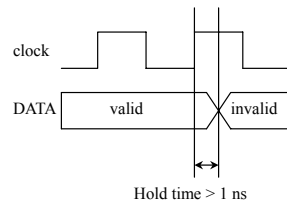


Figure 5(b) Timing diagram for hold time on DATA

The synthesis tool automatically runs its internal static timing analysis engine to check for setup and hold time violations for the paths, that have timing constraints set on them. It mostly uses the following two equations to check for the violations.

$$T_{prop} + T_{delay} < T_{clock} - T_{setup} \quad (1)$$

$$T_{delay} + T_{prop} > T_{hold} \quad (2)$$

Here T_{prop} is the propagation delay from input clock to output of the device in question (mostly a flip-flop); T_{delay} is the propagation delay across the combinational logic through which the input arrives; T_{setup} is the setup time requirement of the device; T_{clock} is clock period; T_{hold} the hold time requirement of the device.

So if the propagation delay across the combinational logic, T_{delay} is such that the equation (1) fails i.e. $T_{prop} + T_{delay}$ is more than $T_{clock} - T_{setup}$ then a setup timing violation is reported. Similarly if $T_{delay} + T_{prop}$ is greater than T_{hold} then a hold timing violation is reported. In the case of the setup violation the input data arrives late due to

large T_{delay} across the combinational logic and thus is not valid/unstable during the setup time period triggering a violation. The flip-flop needs a certain time to read the input. During this period the data must remain stable and unchanged and any change would result in improper working of the device and thus a violation. In case of hold timing violation the data arrives faster than usual because the $T_{delay} + T_{prop}$ is not enough to delay the data enough. The flip-flop needs some time to store the data, during which the data should remain stable. Any change during this period would result in a violation. The data changes faster without giving the flip-flop sufficient time to read it thus triggering a violation.

When the synthesis tool reports timing violations the designer needs to fix them. There are three options for the designer to fix these violations.

- 1) **Optimization using synthesis tool:** this is the easiest of all the other options. Few of the techniques have been discussed in the section Optimization Techniques above. Few other techniques will be dealt with later in this section.
- 2) **Microarchitectural Tweaks:** This is a manual approach compared to the previous one. Here the designer should modify code to make microarchitectural changes that effect the timing of the design. Some of these techniques were discussed in the section Optimization Techniques and few new ones would be dealt with in this section.
- 3) **Architectural changes:** This is the last option as the designer needs to change the whole architecture of the design under consideration and would take up a long time.

Optimization using synthesis tool

The tool can be used to tweak the design for improving performance. A designer for performance optimization can employ the following ways.

- a) Compilation with a `map_effort` high option;
- b) Group critical paths together and give them a weight factor;
- c) Register balancing;
- d) Choose a specific implementation for a module;
- e) Balancing heavy loading.

Compilation with a `map_effort` high

The initial compilation of a design is done with `map_effort` as **medium** when employing design constraints. This usually gives the best results with flattening and structuring options. In case the desired results are not met i.e. the design generates some timing violations then the `map_effort` of **high** can be set. This usually takes a long time to run and thus is not used as the first option. This compilation could improve design performance by about 10%

Group critical paths and assign a weight factor

We can use the `group_path` command to group critical timing paths and set a weight factor on these critical paths. The weight factor indicates the effort the tool needs

to spend to optimize these paths. Larger the weight factor the more the effort. This command allows the designer to prioritize the critical paths for optimization using the weight factor.

`group_path -name <group_name> -from <starting_point> -to <ending_point> -weight <value>`

Register balancing

This command is particularly useful with designs that are pipelined. The command reshuffles the logic from one pipeline stage to another. This allows extra logic to be moved away from overly constrained pipeline stages to less constrained ones with additional timing. The command is simply **balance_registers**.

Choose a specific implementation for a module

A synthesis tool infers high-level functional modules for operators like '+', '-', '*', etc... however depending upon the map_effort option set, the design compiler would choose the implementation for the functional module. For example the adder has the following kinds of implementation.

- a) Ripple carry - *rpl*
- b) Carry look ahead - *cla*
- c) Fast carry look ahead - *clf*
- d) Simulation model - *sim*

The implementation type *sim* is only for simulation. Implementation types *rpl*, *cla*, and *clf* are for synthesis; *clf* is the faster implementation followed by *cla*; the slowest being *rpl*.

If compilation of map_effort low is set the designer can manually set the implementation using the **set_implementation** command. Otherwise the selection will not change from current choice. If the map_effort is set to medium the design compiler would automatically choose the appropriate implementation depending upon the optimization algorithm. A choice of medium map_effort is suitable for better optimization or even a manual setting can be used for better performance results.

Balancing heavy loading

Designs generally have certain nets with heavy fanout generating a heavy load on a certain point. A large load would be difficult to drive by a single net. This leads to unnecessary delays and thus timing violations. The **balance_buffers** command comes in hand to solve such problems. this command would make the design compiler to create buffer trees to drive the large fanout and thus balance the heavy load.

Microarchitectural Tweaks

The design can be modified for both setup timing violations as well as hold timing violations. Lets deal with setup timing violations.

When a design with setup violations cannot be fixed with tool optimizations the code or microarchitectural implementation changes should be employed.

The following methods can be used for this purpose.

- a) Logic duplication to generate independent paths
- b) Balancing of logic between flip-flops
- c) Priority decoding versus multiplex decoding

Logic duplication to generate independent paths

Consider the figure 5(a). Assuming a critical path exists from A to Q2, logic optimization on combinational logic X, Y, and Z would be difficult because X is shared with Y and Z. We can duplicate the logic X as shown in figure 5(b). In this case Q1 and Q2 have independent paths and the path for Q2 can be optimized in a better fashion by the tool to ensure better performance.

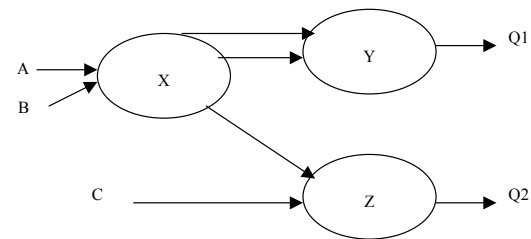


Figure 5(a). Logic with Q2 critical path.

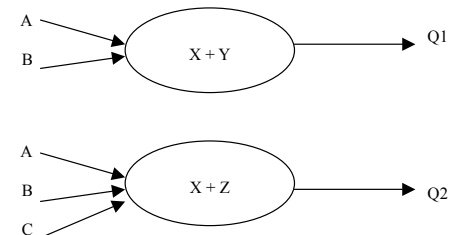


Figure 5(b). Logic duplication giving allowing Q2 an independent path.

Logic duplication can also be used in cases where a module has one signal arriving late compared to other signals. The logic can be duplicated in front of the fast - arriving signals such that timing of all the signals is balanced. Figure 6(a)&(b) illustrate this fact quite well. The signal Q might generate a setup violation as it might be delayed due to the late-arriving select signal of the multiplexer. The combinational logic present at the output could be put in front of the inputs (fast arriving). This would cause the delay

due the combinational logic to be used appropriately to balance the timing of the inputs of the multiplexer and thus avoiding the setup violation for Q.

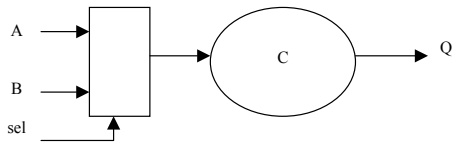


Figure 6(a). Multiplexer with late-arriving sel signal

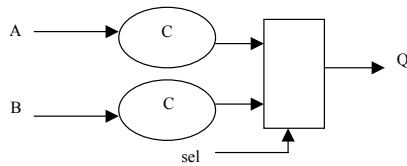


Figure 6(b). Logic-duplication for balancing of timing between the signals.

Balancing of logic between flip-flops

This concept is similar to the **balance_registers** command we have come across in the Tool optimization section. The difference is that the designer does this at the code-level. To fix setup violations in designs using pipeline stages the logic between each stage should be balanced. Consider a pipeline stage consisting of three flip-flops and two combinational logic modules in between each flip-flop. If the delay of the first logic module is such that it violates the setup time of the second flip-flop by a large margin and the delay of the second logic module is so less that the data on the third flip-flop is comfortably meeting the setup requirement. We can move part of the first logic module to the second logic module so that the setup time requirement of both the flip-flops is met. This would ensure better performance without any violations taking place. Figure 7 illustrates the example.

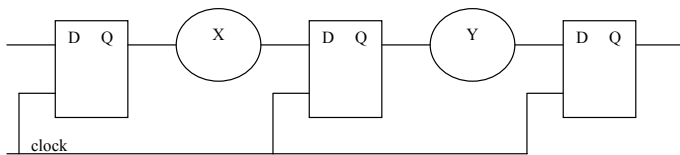


Figure 7. Logic with pipeline stages

Priority encoding versus multiplex encoding

When a designer knows for sure that a particular input signal is arriving late then priority encoding would be a good bet. The signals arriving earlier could be given more priority and thus can be encoded before the late arriving signals. Consider the boolean equation:

$$Q = A.B.C.D.E.F$$

It can be designed using five and gates with A, B at the first gate. The output of first gate is anded with C and output of the second gate with D and so on. This would ensure proper performance if signal F is most late arriving and A is the earliest to arrive. If propagation delay of each and gate were 1 ns this would ensure the output signal Q would be valid only 5 ns after A is valid or only 1 ns after signal H is valid.

Multiplex decoding is useful if all the input signals arrive at the same time. This would ensure that the output would be valid at a faster rate. Thus multiplex decoding is faster than priority decoding if all input signals arrive at the same time. In this case for the boolean equation above the each of the two inputs would be anded parallelly in the form of A.B, C.D and E.F each these outputs would then be anded again to get the final output. This would ensure Q to be valid in about 2 ns after A is valid.

Fixing Hold time violations

Hold time violations occur when signals arrive to fast causing them to change before they are read in by the devices. The best method to fix paths with hold time violations is to add buffers in those paths. The buffers generate additional delay slowing the path considerably. One has to careful while fixing hold time violations. Too many buffers would slow down the signal a lot and might result in setup violations which is a problem again.