5.1 Partitioning for Synthesis

Partitioning can be viewed as, utilizing the "Divide and Conquer" concept to reduce complex designs into simpler and manageable blocks. Promoting design reuse is one of the most significant advantages to partitioning the design.

Apart from the ease in meeting timing constraints for a properly partitioned design, it is also convenient to distribute and manage different blocks of the design between team members.

The following recommendations achieve best synthesis results and reduction in compile time.

a) Keep related combinational logic in the same module.

b) Partition for design reuse.

c) Separate modules according to their functionality.

d) Separate structural logic from random logic.

e) Limit a reasonable block size (perhaps, maximum of 10K gates per block)

f) Partition the top level (separate I/O Pads, Boundary Scan and core logic).

g) Do not add glue-logic at the top level.

h) Isolate state-machine from other logic.

i) Avoid multiple clocks within a block.

j) Isolate the block that is used for synchronizing multiple clocks.

k) WHILE PARTITIONING, THINK OF YOUR LAYOUT STYLE.

The group and ungroup commands provide the designer with the capability of altering the partitions in DC, after the design hierarchy has already been defined by the previously written HDL code. Figure 5-1, illustrates such an action.

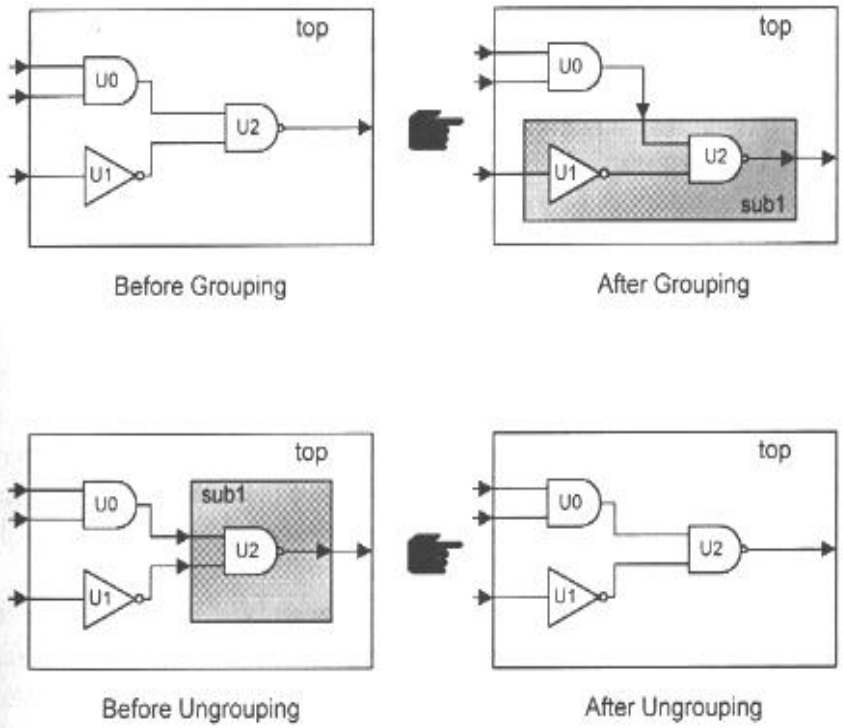

Figure 5-1. Changing Partitions

The group command combines the specified instances into a separate block. In Figure 5-1, instances U1 and U2 are grouped together to form a sub-block named sub1, using the following command.

```
dc_shell> current_design top

dc_shell> group {U1 U2} –design_name sub1
```

The ungroup command performs the opposite function. It is used to remove the hierarchy, as shown in Figure 5-1, by using the following command.

```
dc_shell> current_design top

dc_shell> ungroup –all
```

The designer can also use the ungroup command along with the –flatten and –all options to flatten the entire hierarchy. This is illustrated below:

```
dc_shell> ungroup –flatten –all
```

5.2 What is RTL?

Today, RTL or the Register Transfer Level is the most popular form of high-level design specification. An RTL description of a design describes the design in terms of transformation and transfer of logic from one register to another. Logic values are stored in registers where they are evaluated through some combinational logic, and then re-stored in the next register.

RTL functions like a bridge between the software and hardware. It is text with strong graphical connotations – text that implies graphics or structure. It can be described as technology independent, textual structural description, similar to a netlist.

5.2.1 Software versus Hardware

A frequent obstacle to writing HDL code is the software mind-set. HDLs have evolved from logic netlist representations. HDLs in their initial form (the Register Transfer Level) were a forum to represent logic in a format independent from any particular technology library. A higher level of HDL abstraction is the behavioral level that allows the design to be independent of timing and explicit sequencing. Lately, many system designers have adapted to HDLs to describe the full system.

Frequently, the expectation is that the synthesis tool will synthesize the HDL to the minimal area and maximum performance, regardless of how the HDL is written. The problem remains that at high level there are numerous ways of writing code to perform the same function. For example, a conditional expression could be written using *case* statements or *if* statements. Logically, these expressions are responsible for performing the same task, but when synthesized they can give drastically different results, as far as type of logic inferred, area, and timing are concerned. A reasonable caveat told to recent adopters of synthesis is – THINK HARDWARE!

5.3 General Guidelines

The following are general guidelines that every designer should be aware of. There is no fixed rule to adhere to these guidelines, however, following them vastly improves the performance of the synthesized logic, and produces a cleaner design that is well suited for automating the synthesis process.

5.3.1 Technology Independence

HDL should be written in a technology independent fashion. Hard-coded instances of library gates should be minimized. Preference should be given to inference rather than instantiation. The benefit being that the RTL code can be implemented with any ASIC library and new technology through re-synthesis. This is especially important for synthesizable IP cores that are commonly used by many designs.

In cases where placement of library gates is unavoidable, all the instantiated gates may be grouped together to form their own module. This helps in management of library specific aspects of a design.

5.3.2 Clock Logic

a) Clock logic including clock gating logic 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 ideal clock specifications.

b) Avoid multiple clocks per block – try keeping one clock per block. Such restrictions later help avoid difficulties that may arise while constraining a block containing multiple clocks. It also helps in managing clock skew that may arise at the physical level. Sometimes this becomes unavoidable, for instance where synchronization logic is present to sync signals from one clock domain to the other. For such cases, it is recommended that designer isolate the sync logic, and synthesize it separately using special techniques. This includes setting a dont_touch attribute on the sync logic before instantiating it in the main block.

c) Clocks should be given meaningful names. A suggestion is to keep the name of the clock that reflects its functionality in addition to its frequency. Another good practice is to keep the same name for the clock, uniform throughout the hierarchy, i.e., the clock name should not change as it traverses through the hierarchy. This simplifies the script writing and helps in automating the synthesis process.

d) For DFT scan insertion, it is a requirement that the clocks be controlled from primary inputs. This may involve adding a mux at the clock source for controllability. Although not a hard and fast rule, it is recommended to hand-instantiate the clock-mux in the RTL (preferably at the top-level of the design). This allows the designer to know the instance name of the clock-mux at the stage of RTL elaboration, which in turn allows selective clock-mux timing path to be disabled, using set_disable_timing.

5.3.3 No Glue Logic at the Top

The top-level should only be used for connecting modules together. It should not contain any combinational glue logic. One of the benefits of this style is that it makes redundant the very time consuming top-level compile, which can now be simply stitched together without undergoing additional synthesis. Absence of glue logic at the top-level also facilitates layout, if performing hierarchical place and route.

5.3.4 Module Name Same as File Name

A good practice is to keep the module name (or entity name), same as the file name. Never describe more than one module or entity in a single file. A single file should only contain a single module/entity definition for synthesis. This has enormous benefits in defining a clean methodology using scripting languages like PERL, AWK etc.

5.3.5 Pads Separate from Core Logic

Divide the top-level into two separate blocks "pads" and "core". Pads are usually instantiated and not inferred, therefore it is preferred that they be kept separate from the core logic. This simplifies the setting of the dont_touch attribute on all the pads of the design, simultaneously. By keeping the pads in a separate block, we are isolating the library dependent part of RTL code.

5.3.6 Minimize Unnecessary Hierarchy

Do not create unnecessary hierarchy. Every hierarchy sets a boundary. Performance is degraded, if unnecessary hierarchies are created. This is because DC is unable to optimize efficiently across hierarchies. One may use the ungroup command to flatten the unwanted hierarchies, before compiling the design to achieve better results.

5.3.7 Register All Outputs

This is a well-known Synopsys recommendation. The outputs of a block should originate directly from registers. Although not always practical, this coding/design style simplifies constraint specification and also helps optimization. This style prevents combinational logic from spanning module boundaries. It also increases the effectiveness of the characterize-write-script synthesis methodology by preventing the pin-pong effect that is common to this type of compilation technique.

5.3.8 Guidelines for FSM Synthesis

The following guidelines are presented for writing finite state machines that may help in optimizing the logic:

a) State names should be described using "enumerated types" in VHDL, or "parameters" in Verilog.

b) Combinational logic for computing the next state should be in its own *process* or *always* block, separate from the state registers.

c) Implement the next-state combinational logic with a *case* statement.

5.4 Logic Inference

High-level Description Languages (HDLs) like VHDL and Verilog are front-ends to synthesis. HDLs allow a design to be represented in a technology independent fashion. However, synthesis imposes certain restrictions on the manner in which HDL description of a design is written. Not all HDL constructs can be synthesized. Not only that, synthesis expects HDLs to be coded in a specific way so as to get the desired results. We can say that synthesis is template driven – if the code is written using the templates that are understood and expected by the synthesis tool, then the results will be correct and predictable. The templates and other coding patterns for synthesis are called coding styles. For quality results it is imperative that designers possess a keen understanding of the coding styles, logic inferences, and the corresponding logic structures that DC generates.

5.4.1 Incomplete Sensitivity Lists

This is one of the most common mistakes made by designers. Incomplete sensitivity lists may cause simulation mismatches between the source RTL and the synthesized logic. DC issues a warning for signals that are present in the *process* or *always* block, but are absent from the sensitivity list. This is primarily a simulation problem since the process does not trigger when sensitized (because of the missing signal in the sensitivity list). The synthesized logic, however, is generally correct for blocks containing incomplete sensitivity lists.

Verilog Example

```
always @(weekend or go_to_beach or go_to_work)
begin
    if (weekend)
        action = go_to_beach
    else if (weekday)
        action = go_to_work;
```

VHDL Example

```
process (weekend, go_to_beach, go_to_work)
begin
    if (weekend) then
        action <= go_to_beach;
    elsif (weekday) then
        action <= go_to_work;
    end if;
end process;
```

The examples illustrated above do not contain the signal "weekday" in their sensitivity lists. The synthesized logic may still be accurate, however, during simulation the process will not trigger each time the signal "weekday" changes value. This may cause a mismatch between the simulation result of the source RTL and the synthesized logic.

5.4.2 Memory Element Inference

There are two types of memory elements – latches and flip-flops. Latches are level-sensitive memory elements, while flip-flops in general are edge-sensitive. Latches are transparent as long as the enable to the latch is active. At the time the latch is disabled, it holds the value present at the D input, at

its Q output. Flip-flops on the other hand, respond to rising or falling edge of the clock.

Latches are simple devices, therefore they cover less area as compared to their counterparts, flip-flops. However, latches in general are more troublesome because their presence in a design makes DFT scan insertion difficult, although not impossible. It is also complicated to perform static timing analysis on designs containing latches, due to their ability of being transparent when enabled. For this reason, designers generally prefer flip-flops over latches.

The following sub-sections provide detailed information on how to avoid latches, as well as how to infer them, if desired.

5.4.2.1 Latch Inference

A latch is inferred when a conditional statement is incompletely specified. An *if* statement with a missing *else* part is an example of incompletely specified conditional. Here is an example, both in Verilog and VHDL:

Verilog Example

```
always @(weekend)
begin
    if (weekend)
        action <= go_to_beach;
end
```

VHDL Example

```
process (weekend)
begin
    if (weekend = '1') then
        action <= go_to_beach;
end process;
```

The above statement will cause the DC to infer a latch enabled by a signal called "weekend". In the above example, "action" is not given any value when the signal "weekend" is 0. Always cover all the cases in order to avoid unintentional latch inference. This may be achieved by using an *else* statement, or using a *default* statement outside the *if* branch.

A latch may also get inferred from an incompletely specified *case* statement in Verilog.

```
`define sunny  2'b00
`define snowy  2'b01
`define windy  2'b10

wire [1:0] weather;

case (weather)
    sunny  : action <= go_motorcycling;
    snowy  : action <= go_skiing;
    windy  : action <= go_paragliding;
endcase;
```

In the above case statement only 3 of the 4 possible values of "weather" are covered. This causes a latch to be inferred on the signal "action". Note, for the above example the Synopsys full_case directive may also be used to avoid the latch inference as explained in Chapter 3. The following example contains the *default* statement that provides the fourth condition, thereby preventing the latch inference.

```
case (weather)
    sunny  : action <= go_motorcycling;
    snowy  : action <= go_skiing;
    windy  : action <= go_paragliding;
    default : action <= go_paragliding;
endcase;
```

VHDL does not allow incomplete case statements. This often means that the *others* clause must be used, consequently the above problem does not occur in VHDL. However, latches may still be inferred by VHDL, if a particular

output signal is not assigned a value in each branch of the *case* statement. The inference being that outputs must be assigned a value in all branches to prevent latch inference in VHDL.

```
case (weather) is
      when sunny    => action <= go_motorcycling;
      when snowy    => action <= go_skiing;
      when windy    => action <= go_paragliding;
      when others   => null;
end case;
```

The above example, although containing the *others* clause will infer latches because the output signal "action" is not assigned a particular value in the *others* clause. To prevent this, all branches should be completely specified, as follows:

```
case (weather) is
      when sunny    => action <= go_motorcycling;
      when snowy    => action <= go_skiing;
      when windy    => action <= go_paragliding;
      when others   => action <= go_paragliding;
end case;
```

5.4.2.2 Register Inference

DC provides a wide variety of templates for register inference. This is to support different edge-types of the clock and reset mechanisms. A register is inferred, when there is an edge specified in the sensitivity list. The edge could be a positive edge or a negative edge.

5.4.2.2.1 Register Inference in Verilog

In Verilog, a register is inferred when an edge is specified in the sensitivity list of an *always* block. One register is inferred for each of the variables assigned in the *always* block. All variable assignments, not directly dependent on the clock-edge should be made in a separate *always* block, which does not have an edge specification in its sensitivity list.

A plain and simple positive edge-triggered D flip-flop is inferred using the following template:

```
always @(posedge clk)
   reg_out <= data;
```

In order to infer registers with resets, the reset signal is added to the sensitivity list, with reset logic coded within the *always* block. Following is an example of a D flip-flop with an asynchronous reset:

```
always @(posedge clk or reset)
   if ( reset )
      reg_out <= 1'b0;
   else
      reg_out <= data;
```

Having a synchronous reset is a simple matter of removing the "reset" signal from the sensitivity list. In this case, since the block responds only to the clock edge, the reset is also, only recognized at the clock edge.

```
always @(posedge clk)
   if ( reset )
      reg_out <= 1'b0;
   else
      reg_out <= data;
```

Negative edge-triggered flop may be inferred by using the following template:

```
always @(negedge clk)
   reg_out <= data;
```

Absence of negative edge-triggered flop in the technology library will result in DC inferring a positive edge-triggered flop with an additional inverter to invert the clock signal.

5.4.2.2.2 Register Inference in VHDL

In VHDL a register is inferred when an edge is specified in the *process* body. The following example illustrates the VHDL template to infer a D flip-flop:

```
reg1: process (clk )
begin
    if ( clk'event and clk = '1' ) then
        reg_out <= data;
    end if;
end process Reg1;
```

DC does not infer latches for variables declared inside functions, since variables declared inside functions are reassigned each time the function is called.

Coding style template for registers with asynchronous and synchronous resets are similar in nature to that of Verilog templates, shown in previous section.

Negative edge-triggered flop may be inferred by using the following template:

```
reg1: process (clk )
begin
    if ( clk'event and clk = '0' ) then
        reg_out <= data;
    end if;
end process Reg1;
```

Absence of negative edge-triggered flop in the technology library will result in DC inferring a positive edge-triggered flop with an additional inverter to invert the clock signal.

5.4.3 Multiplexer Inference

Depending upon the design requirements, the HDL may be coded in different ways to infer a variety of architectures using muxes. These may comprise of a single mux with all inputs having the same delay to reach the output, or a priority encoder that uses a cascaded structure of muxes to prioritize the input signals. A mixture of the above techniques is also commonly used to place the late arriving signal closer to the output.

The correct use of *if* and *case* statements is a complex topic that is outside the scope of this chapter. There are application notes (from Synopsys) and other published materials currently available that explain the proper usage of these statements. It is therefore the intent of this chapter to refer the users to outside sources for this information. Only brief discussion is provided in this section.

5.4.3.1 Use *case* Statements for Muxes

In general, *if* statements are used for latch inferences and priority encoders, while *case* statements are used for implementing muxes. It is recommended to infer muxes exclusively through *case* statements. The *if* statements may be used for latch inferencing and priority encoders. They may also be effectively used to prioritize late arriving signals. This kind of prioritizing may be implementation dependent. It also limits reusability.

To prevent latch inference in *case* statements the *default* part of the *case* statement should always be specified. For example, in case of a state machine, the default action could be that all states covered by the *default* clause cause a jump to the "start" state. Having a *default* clause in the *case* statement is the preferred way to write *case* statements, since it makes the HDL independent of the synthesis tool. Using directives like full_case etc makes the code dependent on the synthesis tool.

If the default action is to assign don't-cares, then a difference in behavior between RTL simulation and synthesized result may occur. This is because, DC may optimize the don't-cares randomly causing the resulting logic to differ.

5.4.3.2 *if* versus *case* Statements – A Case of Priorities

Multiple *if* statements with multiple branches result in the creation of priority encoder structure.

```
always @(weather or go_to_work or go_to_beach)
begin
    if (weather[0]) action = go_to_work;
    if (weather[1]) action = go_to_beach;
end
```

In the above example, the signal "weather" is a two-bit input signal and is used to select the two inputs, "go_to_work" and "go_to_beach", with "action" as the output. When synthesized, the cascaded mux structure of the priority encoder is produced as shown in Figure 5-2.

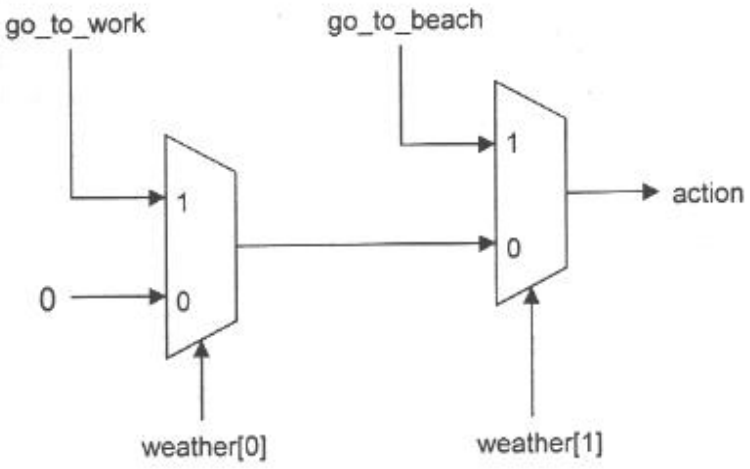

Figure 5-2. Result of using Multiple *if* Statements

If the above example used the *case* statement (instead of multiple *if* statements) in which all possible values of the selection index were covered and were exclusive, then it would have resulted in a single multiplexer as shown in Figure 5-3.

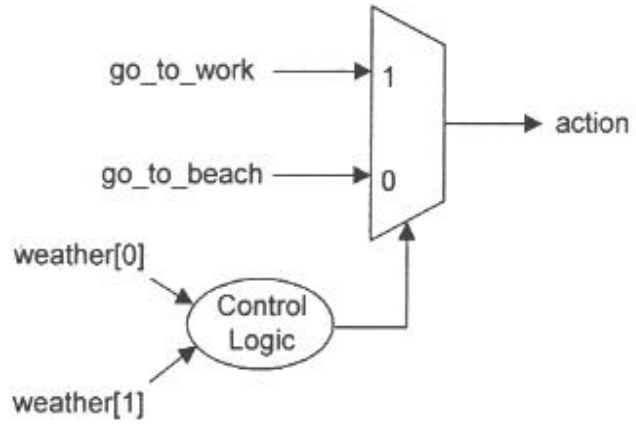

Figure 5-3. Result of using *case* Statement, or a Single *if* Statement

The same structure (Figure 5-3) is produced, if a single *if* statement is used, along with *elsif* statements to cover all possible branches.

5.4.4 Three-State Inference

Tri-state logic is inferred when high impedance (Z) is assigned to an output. Arbitrary use of tri-state logic is generally not recommended because of the following reasons:

a) Tri-state logic reduces testability.

b) Tri-state logic is difficult to optimize – since it cannot be buffered. This can lead to max_fanout violations and heavily loaded nets.

On the upside however, tri-state logic can provide significant savings in area.

Verilog

```
assign tri_out = enable ? tri_in : 1'bz;
```

VHDL

```
tri_out <= tri_in when (enable = '1') else 'Z';
```

5.5 Order Dependency

Both, Verilog and VHDL provide variable assignments that are order dependent/independent. Correct usage of these produces desired results, while incorrect usage may cause synthesized logic to behave differently than the source RTL.

5.5.1 Blocking versus Non-Blocking Assignments in Verilog

It is important to use non-blocking statements when doing sequential assignments like pipelining and modeling of several mutually exclusive data transfers. Use of blocking assignments within sequential processes may cause race conditions, because the final result depends on the order in which the assignments are evaluated. The non-blocking assignments are order independent; therefore they match closely to the behavior of the hardware.

Non-blocking assignment is done using the "<=" operator, while the "=" operator is used for blocking assignments.

```
always @(posedge clk)
begin
    firstReg    <= data;
    secondReg   <= firstReg;
    thirdReg    <= secondReg;
end
```

In hardware, the register updates will occur in the reverse order as shown above. The use of non-blocking assignments causes the assignments to occur in the same manner as hardware i.e., thirdReg will get updated with the old value of secondReg and the secondReg will get updated with the old value of firstReg. If blocking assignments were used in the above example, the signal "data" would have propagated all the way through to the thirdReg concurrently during simulation.

The blocking assignments should generally be used within the combinational *always* block.

5.5.2 Signals versus Variables in VHDL

Similar to Verilog, VHDL also provides order dependency through the use of signals and variables. The signal assignments may be equated to Verilog's non-blocking assignments, i.e., they are order independent. The variable assignments are order sensitive and correlate to Verilog's blocking assignments.

Variable assignments are done using the ":=" operator, whereas the "<=" operator is used for signal assignments.

The following example illustrates the usage of the signal assignments within the sequential *process* block. The resulting hardware contains three registers, with signal "data" propagating from firstReg to secondReg and then to the thirdReg. The RTL simulation will also show the same result.

```
process(clk)
begin
    if (clk'event and clk = '1') then
        firstReg    <= data;
        secondReg   <= firstReg;
        thirdReg    <= secondReg;
    end if;
end process;
```

A general recommendation is to only use signal assignments within sequential processes and variable assignments within the combinational processes.