Hierarchy Management in Synplify

A look at how Synplify automatically manages hierarchy for all Xilinx architectures while giving you additional controls if required.

by Allen Drost, Corporate Applications Manager, and Jim Tatsukawa, Partner Programs Manager, Synplicity, allen@synplicity.com, jimt@synplicity.com

As your designs reach ASIC complexities with the Xilinx Virtex devices, you will face new design problems. Of particular importance is how synthesis tools handle your design's hierarchy. Simple solutions of just turning on or off hierarchy are inefficient and cumbersome. Synthesis tools need to look beyond just optimization and be used as part of an overall design flow that includes constant changes and recompiles.

Hierarchy Pro's and Con's

Traditional synthesis tools treat hierarchy as "artificial" hard boundaries. Thus preventing logic on one level of hierarchy from being combined with logic on another level. For example, if the output of an inverter crosses a level of hierarchy and drives a second inverter, this hierarchy boundary would prevent a traditional synthesis tool from optimizing the double inversion, resulting in a final circuit that has two inverters instead of a more optimal circuit with zero inverters. See **Figures 1 and 2** for more examples.

Synthesis tools need to look beyond just optimization and be used as part of an overall design flow with constant changes and recompiles.

Alternatively you could try and dissolve the design so that there were no hierarchical boundaries. This flattening of the design may produce a smaller or faster design from a logic perspective, but it creates other complications in the design flow. First of all, blindly flattening the entire design may produce a single block of logic large enough to overwhelm the capacity of the synthesis tool, resulting in unmanageable runtimes, or a sub-optimal netlist. To deal with this problem, you may need to specify which levels of hierarchy to dissolve, but this requires you to decide what the optimal hierarchical boundaries should be.

Another issue with flattening hierarchy is that, although it may lead to a more optimal netlist from synthesis, the simple fact that the topology of the design has changed may cause problems with tools downstream. For example, you may invest significant amounts of time and effort developing constraints for placement and routing, as well as a simulation testbench. By altering the topology of the design, the grouping of large blocks of logic may change, as well as the names of the registers within the design. These changes could invalidate both the place and route constraints and the simulation testbench. If this occurs,

Continued on the following page

RTL & Technology Views of Hierarchical Design

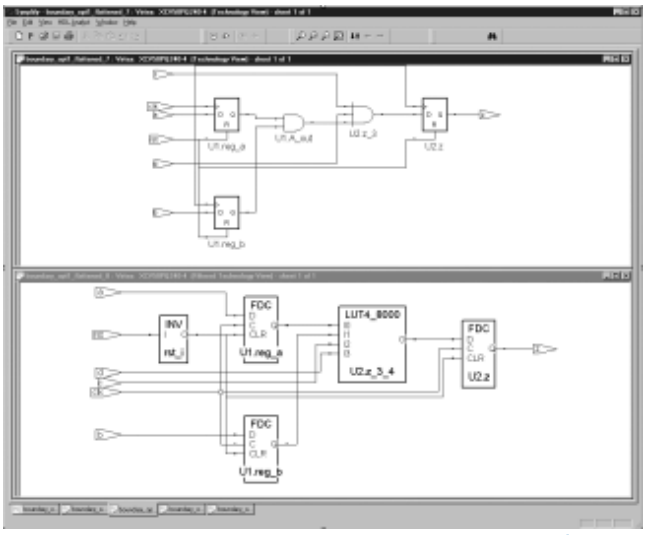

Figure 1

This design simply ANDs four signals together, however as shown in the top (RTL) view, two of the signals are ANDed in hierarchical block U1 and two more are ANDed with that result in block U2. Notice in the bottom (Technology) view that the two AND structures were merged together into a single 4-input LUT in block U2. This type of simple boolean optimization occurs regardless of the value of syn_hier.

Source code

```
/**** Sub-block A description ****/
module block_A (clk, rst, a, b, A_out);

input clk, rst, a, b;
output A_out;

reg  reg_a, reg_b;

always @(posedge clk or negedge rst) begin
 if (!rst) begin
    reg_a <= 1'b0;
    reg_b <= 1'b0;
 end
 else begin
    reg_a <= a;
    reg_b <= b;
 end
end

assign A_out = reg_a & reg_b;

endmodule
```

```
/**** Sub-block B description ****/
module block_B (clk, rst, c, d, A_out, z);

input clk, rst, c, d, A_out;
output z;

reg  z;

always @(posedge clk or negedge rst) begin
 if (!rst)
    z <= 1'b0;
 else
    z <= c & d & A_out;
end

endmodule

/**** Top level description ****/
module boundary_opt1 (clk, rst, a, b, c, d, z);

input clk, rst, a, b, c, d;
output z;

wire  A_out;

block_A U1 (clk, rst, a, b, A_out);
block_B U2 (clk, rst, c, d, A_out, z);

endmodule
```

Figure 2

you would need to update the constraints and testbench, creating a lot of extra work, particularly if it is necessary to iterate this process several times.

Hierarchy Solution

Synplify handles these hierarchical issues automatically. During the synthesis process, Synplify dissolves as much of the design's hierarchy as possible to allow efficient optimization of logic across hierarchical boundaries while maintaining fast runtimes. Synplify then rebuilds the hierarchy to be as close as possible to the original source.

With the exception of any optimizations that occurred on the logic that straddles the hierarchy boundaries, the final netlist will have the same hierarchy as the original source code, ensuring that hierarchical register names remain consistent, and that major blocks of logic remain grouped together. This method of handling hierarchical boundaries, combined with the architecture-specific mapping, creates an efficient and effective optimization engine.

Additional Control for Hierarchy

Although Synplify does a good job at managing hierarchical designs automatically, from time to time you may have a specific reason to want manual control over your design's hierarchy. Synplify offers the "syn_hier" attribute to provide this manual control. The syn_hier attribute is applied to instances, modules, or architectures. It takes one of three values:

- The "remove" option dissolves a level of hierarchy.
- The "soft" option is the default option, and gives Synplify control over which hierarchical boundaries to dissolve.
- The "firm" option prevents a level of hierarchy from being dissolved, however, simple boolean optimizations will still take place across hierarchical boundaries (see **Figure 1**).

Notice that these options control the way Synplify handles a design during optimization only. Regardless of which option is selected (remove, soft, or firm), Synplify will rebuild the

Optimization With Soft Hierarchy

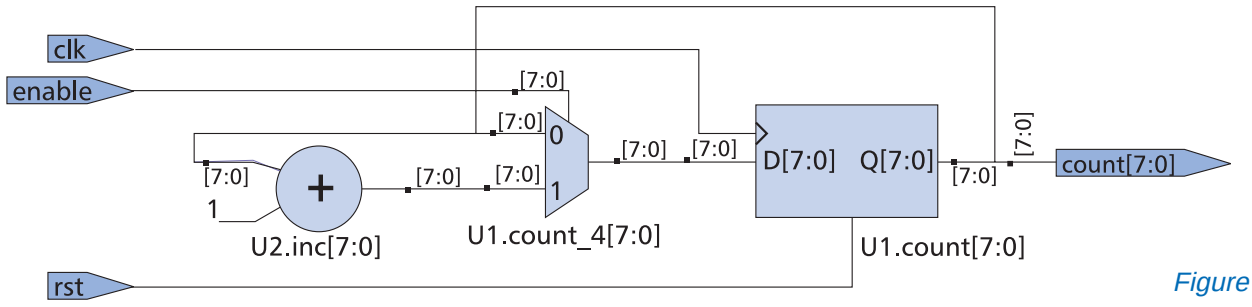

Figure 3

hierarchy before the final netlist is created, ensuring that the netlist created by Synplify is efficient with regard to hierarchical boundary optimizations, and structurally as close as possible to the source code.

The syn_hier attribute can be placed directly in the source code, in the Synplify constraints file (.sdc), or in the graphical constraint editor in SCOPE. See **Figure 4** for example usage. The syntax is shown below:

Verilog:

```
module block_A (clk, rst, enable, inc, count) /*
 synthesis syn_hier = "firm" */;
```

VHDL:

```
attribute syn_hier: string;
attribute syn_hier of block_A : architecture is
 "firm";
```

Constraint file (.sdc):

```
define_attribute { U1} syn_hier { firm}
```

Summary

To control Virtex designs over multiple compiles, synthesis tools need a hierarchical solution that does not compromise the overall performance of the design. Synplify understands the hierarchies of a design, and automatically produces a netlist with efficient logic optimization across hierarchical boundaries, while preserving a topology as close as possible to the source code, thus ensuring that the entire design flow remains intact. See www.synplicity.com for more information.

Source Code:

```
/**** Sub-block A description ****/
module block_A (clk, rst, enable, inc, count);

input    clk, rst, enable;
input [7:0] inc;
output [7:0] count;

reg  [7:0] count;

always @(posedge clk or negedge rst) begin
 if (!rst) count = 8'h00;
 else if (enable) count = inc;
end

endmodule

/**** Sub-block B description ****/
module block_B (count, inc);

input [7:0] count;
output [7:0] inc;

assign inc = count + 1;

endmodule

/**** Top level description ****/
module boundary_opt1 (clk, rst, enable, count);

input    clk, rst, enable;
output [7:0] count;

wire  [7:0] inc;

block_A U1 (clk, rst, enable, inc, count);
block_B U2 (count, inc);

endmodule
```

Figure 4

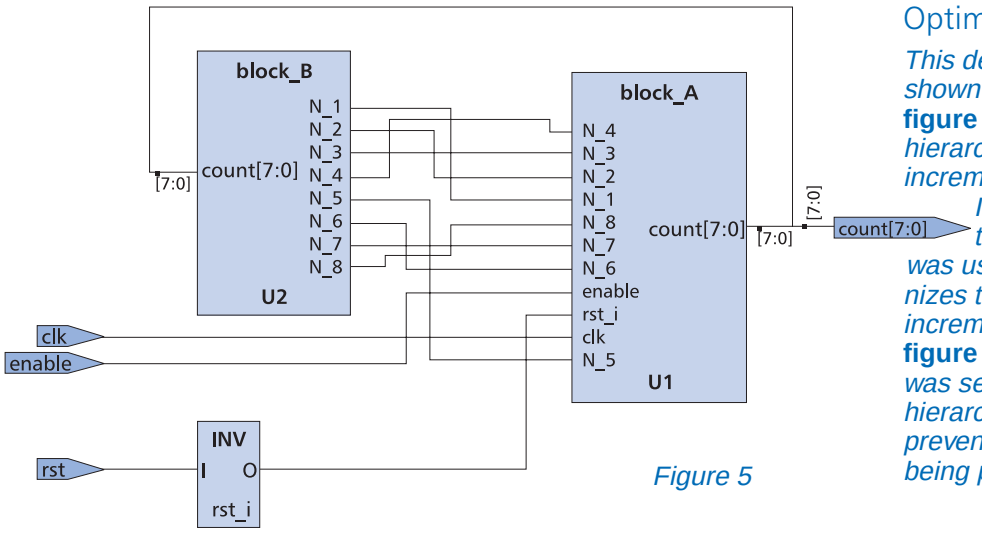

Figure 5

Optimization With Firm Hierarchy

This design is a simple counter. As shown by the component names in **figure 3** *(RTL View), the register is in hierarchical block U1 and the incrementor is in hierarchical block U2. In* **figure 6** *(Technology View), the default value of syn_hier was used (soft), and Synplify recognizes the counter and pulls the increment logic into block U1. In* **figure 5** *(Technology View), syn_hier was set to the value firm. This kept the hierarchical boundary in tact and prevented the increment logic from being pulled into block U1.*

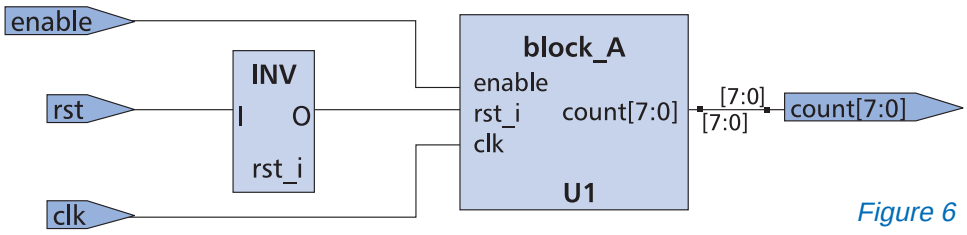

Figure 6