Inferring Multiplexers in FPGA Compiler II and FPGA Express

How to get better results by automatically inferring multiplexers that fully utilize architecture-specific FPGA resources.

by Alan Ma
Senior Corporate Applications Engineer, Synopsys, Inc.
alanma@synopsys.com

In general, multiplexers can be implemented by using Look Up Tables (LUTs). To obtain the best quality of results (QoR), Synopsys FPGA Compiler II™ and FPGA Express™ (FCII/FE) take it one step further by utilizing the built-in multiplexer resources in high-density FPGAs, which produces significantly better results in both area and speed.

The Process

During elaboration, the process of translating the text-based description of a design to an architecture-independent gate-level representation, FCII/FE infers a generic primitive called MUX_OP when it encounters multiplexers in the Hardware Description Language (HDL). It is during optimization where MUX_OPs are mapped to architecture-specific multiplexer resources. The following sections describe the requirements for MUX_OP to be inferred.

General Implementation

Our research indicates that using architecture-specific multiplexer resources is only beneficial when the number of inputs meets certain requirements. Table 1 illustrates the multiplexer sizes and the primitives FCII/FE utilizes for Xilinx Virtex-II, Virtex, and XC4000 FPGAs (and their derivatives). FCII/FE automatically maps to these hardware resources (primitives) when you follow the recommended coding guidelines.

Coding Guidelines

Synopsys recommends the use of CASE statements to describe multiplexer logic. When the requirements on the number of inputs for the target architecture are met (as shown in Table 1), FCII/FE maps the design to architecture-specific multiplexer resources if at least 75% of all possible cases are specified.

Figure 1 shows an example of an eight-to-one multiplexer in Verilog. Figure 2 illustrates its VHDL equivalent. Note that the control signal sel has three bits so there can be as many as eight possible cases. As a result, at least six (75% of eight) cases need to be specified for multiplexers to be inferred.

Architecture	Min. Inputs	Max. Inputs	Primitives Used
Virtex-II	4	256	LUT, MUXF5, MUXF6
Virtex	4	256	LUT, MUXF5, MUXF6
XC4000	4	256	FMAP, HMAP

Table 1 - Multiplexer size requirements for automatic inference

Using the infer_mux Directive

Figure 3 shows a similar eight-to-one multiplexer with the addition of several arithmetic operators; Figure 4 shows its VHDL counterpart. To allow operator sharing, multiplexers are generally not automatically inferred for CASE statements which contain more than one operator (regardless of the number of cases specified). However, you have the option to override FCII/FE by using the infer_mux directive.

The infer_mux directive forces FCII/FE to infer multiplexers as long as at least 50% of all possible cases are specified. It can be used when:

- The requirements on the number of inputs (as shown in Table 1) are not met.
- The CASE statement contains more than one arithmetic operator.

It is important to understand that FCII/FE generally makes intelligent decisions on multiplexer inference based on the cost of doing so. For example, it may choose not to infer multiplexers, to allow operator sharing for better performance. As a result, QoR is likely to suffer if you override that decision by using infer_mux. Please use this directive with caution.

Conclusion

FPGA Compiler II and FPGA Express take advantage of Xilinx-specific multiplexer resources to deliver the best quality of results. The tools automatically infer multiplexers if the design complies with the coding guidelines and meets the requirements for the target architecture. You also have the option to force multiplexer inference by using the infer_mux directive.

Visit the Synopsys FPGA website at www.synopsys.com/fpga for other information on the latest FPGA synthesis technologies.

```
module mux_8to1  (
a, b, c, d, e, f, sel,
                    mux_out
);

 input                    a, b, c, d, e, f;
 input      [2:0]         sel;
 output     [1:0]         mux_out;

 reg  [1:0]         mux_out;

 always @(sel or a or b or c or d or e or f)
case (sel)// synopsys infer_mux
             3'b000     : mux_out = a + b;
             3'b001     : mux_out = a + c;
             3'b010     : mux_out = d - e;
             default    : mux_out = d - f;
endcase
 endmodule
```

Figure 1 - Using CASE statements for multiplexers in Verilog

```
library ieee;
 use ieee.std_logic_1164.all;

 entity mux_8to1 is port  (
                    a, b, c, d, e, f: in std_logic;
sel: in std_logic_vector(2 downto 0);
                    mux_out: out std_logic
);
 end mux_8to1;

 architecture rtl of mux_8to1 is
 begin
process (sel, a, b, c, d, e, f)
begin
case sel is
                    when "000"    => mux_out <= a;
                    when "001"    => mux_out <= b;
                    when "010"    => mux_out <= c;
                    when "011"    => mux_out <= d;
                    when "100"    => mux_out <= e;
                    when others   => mux_out <= f;
end case;
end process;
 end rtl;
```

Figure 2 - Using CASE statements for multiplexers in VHDL

```
module mux_8to1  (
a, b, c, d, e, f, sel,
                    mux_out
);

 input                    a, b, c, d, e, f;
 input      [2:0]         sel;
 output                   mux_out;

 reg                mux_out;

 always @(sel or a or b or c or d or e or f)
case (sel)
             3'b000     : mux_out = a;
             3'b001     : mux_out = b;
             3'b010     : mux_out = c;
             3'b011     : mux_out = d;
             3'b100     : mux_out = e;
             default    : mux_out = f;
endcase
 endmodule
```

Figure 3 - Using infer_mux for multiplexer inference in Verilog

```
library ieee;
 use ieee.std_logic_1164.all;

 entity mux_8to1 is port  (
             a, b, c, d, e, f: in std_logic;
             sel: in std_logic_vector(2 downto 0);
             mux_out: out std_logic_vector(1 downto 0)
             );
 end mux_8to1;

 architecture rtl of mux_8to1 is
 begin
process (sel, a, b, c, d, e, f)
begin
case sel is      -- synopsys infer_mux
             when "000"    => mux_out <= a + b;
             when "001"    => mux_out <= a + c;
             when "010"    => mux_out <= d - e;
             when others   => mux_out <= d - f;
end case;
end process;
 end rtl;
```

Figure 4 - Using infer_mux for multiplexer inference in VHDL