In HDLs, what you see isn't always what you get

Steve Knapp

What you see isn't always what you get. This cliché applies particularly to digital design with hardware description languages (HDLs). VHDL and Verilog offer tremendous abstraction and productivity benefits. However, just as with any good power tool, HDLs present some potential dangers. Used inappropriately, a power saw could just as easily rip through your arm as a sheet of plywood. Likewise, HDLs provide previously undreamed of productivity—but used incorrectly, they promise a hellish design experience.

Think back to the days when you began programming or drawing schematics. That first program or chip was probably much slower and far bigger than intended. With experience, though, your design techniques improved. Using HDLs follows a similar path: first, learn the tool's capabilities and limitations, then recognize that HDLs aren't a panacea. They can't transform a slow, inefficient design into a model of perfection just as a power saw won't make you a master carpenter. Drawing from my experience, this column addresses some issues to be aware of when using VHDL to design programmable logic.

Steven K Knapp is the founder and president of OptiMagic Inc (Aptos, CA, www.optimagic.com), a firm that develops intellectual property and design software for programmable logic. Prior to founding this firm he held various applications, engineering and management positions at Xilinx and Intel's former programmable logic division.

Inferred latches

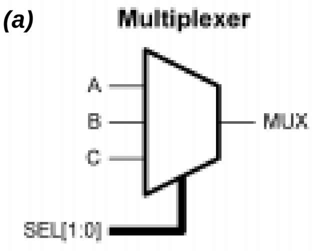
One interesting trap beginners typically fall into is allowing unintended logic to creep into designs. For instance, it's possible to inadvertently specify a latch in VHDL source code. While not a catastrophe, these "inferred" latches consume valuable real estate. A simple example demonstrates the concept.

Assume a multiplexer selects between three input signals (A, B and C) using two select lines (SEL1 and SEL0) as in Fig 1a. The VHDL code to specify the multiplexer might look something like the text in Fig 1b. The code functionally simulates correctly, and the logic performs as you might expect. However, when synthesizing the design, software might issue a warning indicating that it generated a latch, probably caused by a missing assignment in an *If* or a *Case* statement. What exactly does this mean?

Examine the design in Fig 1a more

Fig 1—A 3-input multiplexer function in classic form (a) follows the intuitive structure and datasheet view a designer experiences with schematic-based design. The VHDL source code (b) that implements the 3-input multiplexer implicitly creates a latch unless the designer specifies a default assignment.

(a)



SEL1	SEL0	MUX
0	0	A
0	1	B
1	0	C

(b)

```
library IEEE;
use IEEE.std_logic_1164.  all;

entity inferred_latch   is
   port (A,B,C:      in STD_LOGIC;
         SEL:        in STD_LOGIC_VECTOR(1    downto 0);
         MUX:        out STD_LOGIC);
end inferred_latch;

architecture BEHAV  of inferred_latch   is
begin
process (SEL,A,B,C)
   begin
      if (SEL="00")   then
         MUX <= A;
      elsif (SEL="01")   then
         MUX <= B;
      elsif (SEL="10")   then
         MUX <= C;

      end if;
   end process;
end STRUCT;
```

Default Assignment

```
else
   MUX <= '0';
```

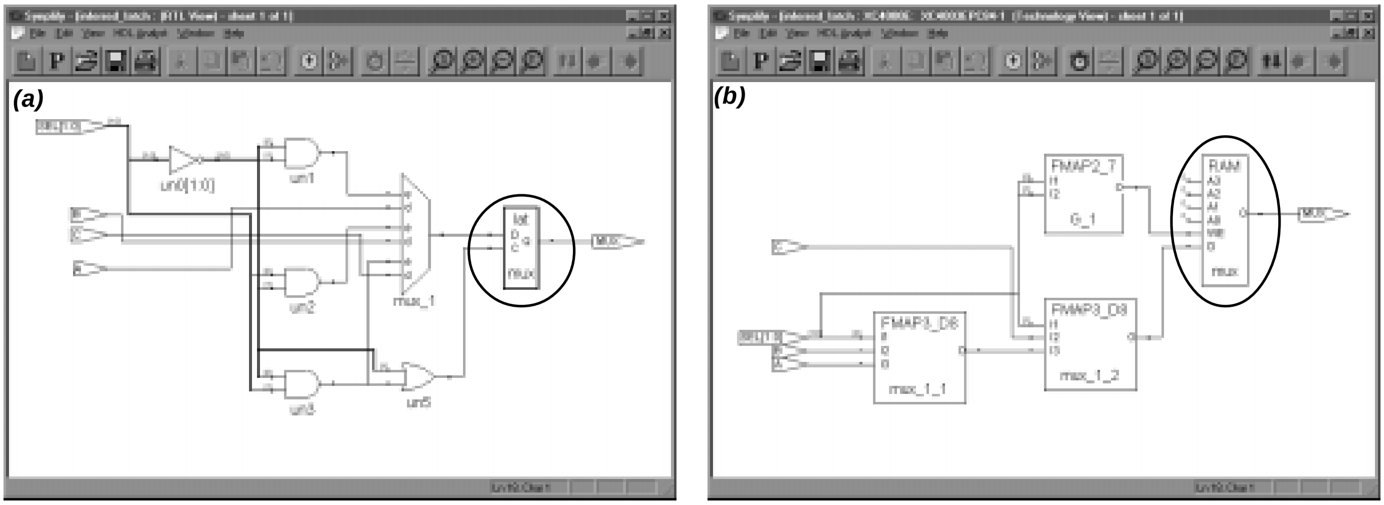


Fig 2—The Synplify HDL Analyst design viewer for the inferred latch function presents two views of the synthesized logic. The register-transfer-level (RTL) view (a) shows an equivalent schematic, while a technology view (b) shows the physical implementation in a Xilinx XC4000E FPGA. Note that the synthesizer implements a latch (circled in a) using the XC4000E's level-sensitive RAM function (circled in b).

closely. What happens when SEL1 and SEL0 are both asserted High? The explicit VHDL code indicates that nothing happens. VHDL implicitly uses a latch to retain a prior state if there isn't an explicit transition, which means that whatever appears on the MUX output remains there as long as both SEL1 and SEL0 are High. How would a logic synthesizer resolve this logic physically? It would use a gated latch.

Without closely examining the resulting netlist, this extra logic might go unnoticed. However, during place and route you might note that the design consumes a bit more logic than expected. Some of the better logic-synthesis packages provide a schematic viewer to display the resulting synthesized design. For example, Synplicity's (Sunnyvale, CA) Synplify offers an optional design viewer, HDL Analyst, that provides both register-transfer-level (RTL) and technology mapping views of a VHDL or Verilog source file.

Fig 2a shows HDL Analyst's RTL view for the design of Fig 1b. It contains mostly gate-level primitives and provides a look at how the synthesis software interprets the HDL source

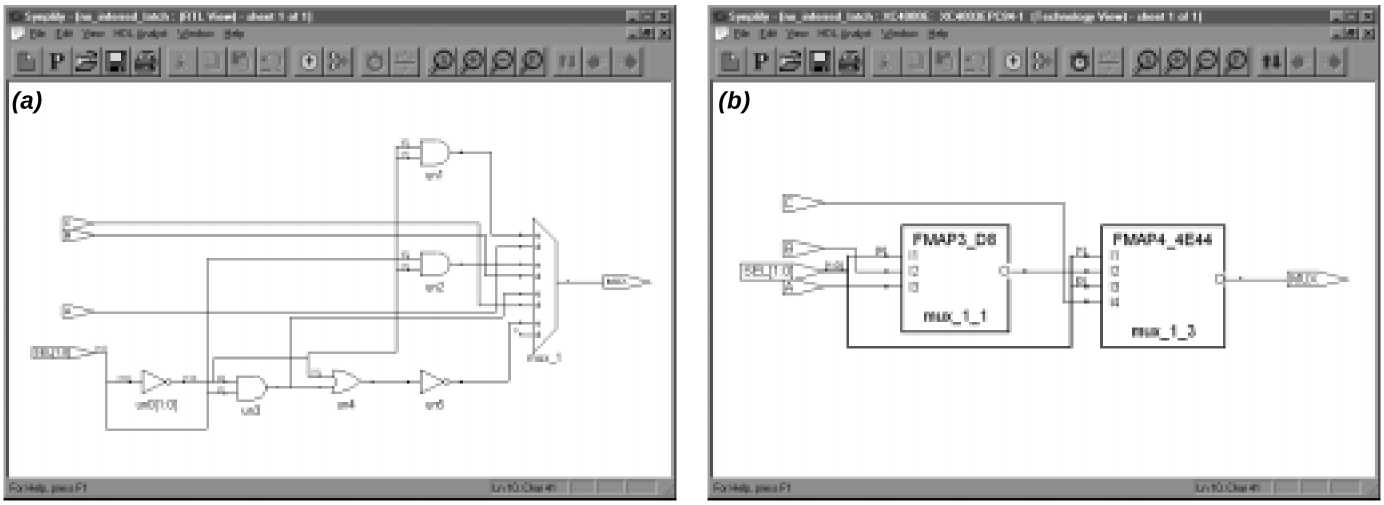
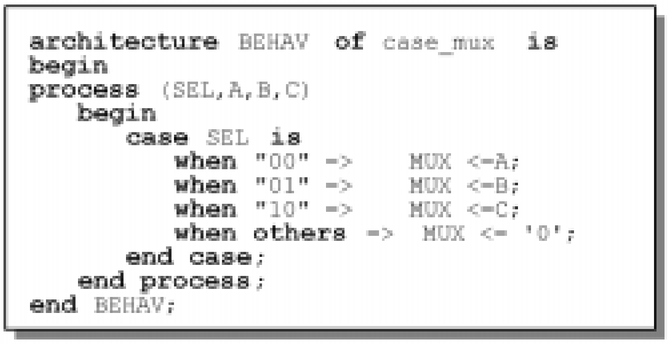


Fig 3—Adding a default assignment simplifies both the RTL (a) and technology views (b) for the multiplexer function from Fig 1a.



```
architecture BEHAV  of case_mux  is
begin
process (SEL,A,B,C)
   begin
      case SEL is
         when "00" =>    MUX <=A;
         when "01" =>    MUX <=B;
         when "10" =>    MUX <=C;
         when others =>  MUX <= '0';
      end case;
   end process;
end BEHAV;
```

Fig 4—Another implementation of the 3:1 multiplexer using VHDL's Case construct is often the preferred method of specifying this function.

code. Fig 2b shows the corresponding technology view for the same design but indicates how the logic maps into functions available on the target device—in this case, a Xilinx XC4000E FPGA. The blocks labeled FMAP... in Fig 2b represent lookup tables (LUTs) in a Xilinx XC4000E.

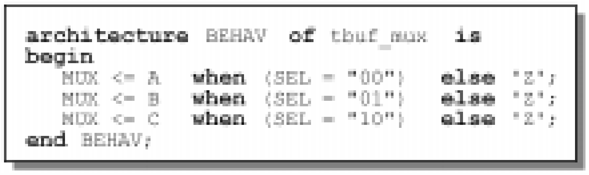
Note the latch primitive in Fig 2a. The resulting implementation uses the XC4000E's level-sensitive RAM element to build it. Consequently, the code in Fig 1b results in the use of four LUTs—a less than optimal solution. While not ultimately efficient, this implementation functions correctly, albeit a bit slow. Although the machine-generated schematic for the design isn't as clear as one created by a human, it's far more understandable than digging through a text-based netlist to figure out the functionality—not a pleasant experience.

So, is there a better way to build this design? You bet! The inferred latch appears because the VHDL source didn't specify all possible conditions. Adding a default assignment to the bottom of the nested-*If* statement (see Fig 1b) results in a more efficient result (Fig 3). The latch disappears and the technology view is greatly simplified. Adding the default assignment reduces the resource requirements from four LUTs down to only two—a near optimal solution.

Coding style also affects the efficiency of the resulting logic. Generally, *Case* statements tend to be more efficient than nested-*If* statements for multiplexer functions. Fig 4 shows the VHDL code fragment implementing the 3:1 multiplexer using a *Case* construct. Note the default assignment at the end of the statement using the Others keyword.

Using 3-state buffers

These seemingly minor points of style illustrate a common misconception among design novices. They assume that a logic synthesizer and optimizer always find the best implementation. In fact, much of the result



```
architecture BEHAV  of tbuf_mux  is
begin
   MUX <= A   when (SEL = "00")   else 'Z';
   MUX <= B   when (SEL = "01")   else 'Z';
   MUX <= C   when (SEL = "10")   else 'Z';
end BEHAV;
```

Fig 5—Using 3-state buffers for multiplexers can be more efficient when the underlying technology supports it and is most appropriate for bus-oriented designs.

Space left blank intentionally.

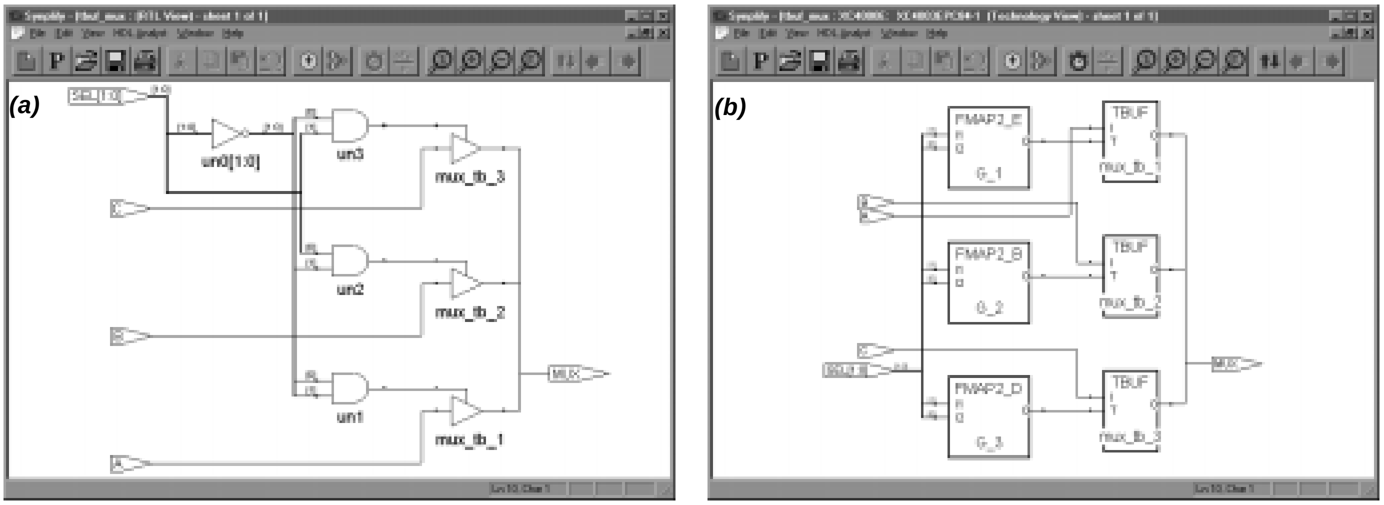


Fig 6—Both the RTL (a) and technology views (b) of the implementation of the logic from Fig 5 show a greatly simplified multiplexer.

depends on the initial source code, which is especially true with arithmetic functions, where various carry-look-ahead techniques result in different speed-vs-area implementations. This column's multiplexer example demonstrates this idea. Note that all the implementations built so far (Figs 2 and 3) use logic gates to construct the multiplexer. The Xilinx XC4000E family also has on-chip 3-state buffers intended primarily to implement bidirectional data buses. In reality, a bidirectional bus is nothing more than a multiplexer structure. However, most synthesis tools won't even consider a 3-state buffer implementation without some source-code changes.

The VHDL fragment in Fig 5 causes the logic-synthesis software to build a 3-state buffer multiplexer (Fig 6). Depending on the state of the SEL inputs, the MUX output either drives the values on A, B, or C, or the output is in the high-impedance state. This particular implementation is available only for devices containing internal 3-state buffers such as the Xilinx XC4000 family and the Lucent Orca family.

General HDL tips

Based on the earlier examples, keep in mind a few general tips for HDL designs:

• Know your synthesis tool, its capabilities and limitations.

• A bad design won't get better after running it through logic synthesis. Synthesis provides an efficient means to enter a design, whether it's good or bad. Spaghetti code in any programming language is still spaghetti code after a compiler is finished with it.

• Designing with HDL is an ongoing learning process. As you gain experience, your designs will be faster and consume less logic. Don't be afraid to try trial implementations to understand how a tool responds. Also look at other people's code for ideas.

• Investigate how your coding affects the final implementation. Schematic viewers are helpful. Without them, you must reconstruct the design's structure from a text-based netlist.

• Always include a default assignment in all *If* and *Case* statements, even if you're sure they cover all conditions. Without a default assignment, a synthesis tool can generate inferred latches. Watch for warnings from the software.

• It's generally better to code multiplexers using a *Case* construct rather than a nested-*If* construct.

• Each programmable-logic family has unique features. Some special methods might be required to use these features, such as those needed to build a multiplexer using 3-state buffers. ***PE&IN***

Editorial Feedback
This article's value to me was:
High—269 Average—270 Low—271