WP275 (v1.0.1) October 22, 2007

Get your Priorities Right – Make your Design Up to 50% Smaller

By: Ken Chapman

Would it be useful for your next design to be up to 50% smaller without significantly changing the ways you do things now? In these cost-sensitive days, you have to respond "yes". This white paper describes a rarely noticed design technique that can make a difference in the size and the performance of your FPGA design.

The ideas contained in this white paper might not yield the full 50% savings potential in your design, but should save a significant amount. A 20% reduction in size can mean your design fits into a smaller device. The smaller size also helps those designers who are looking for higher performance by potentially saving a speed grade.

This white paper provides some simple VHDL and Verilog examples to explain key points. More VHDL code examples are used due to the author's preference.

Single-Level Logic

Simple logic is implemented in a Xilinx FPGA using the look-up tables (LUTs) and flip-flops inside the logic "slices" that are contained in the configurable logic blocks (CLBs). Even a small device such as the XC3S50 has over 1,500 LUT and flip-flop pairs, so they can be thought of as free. However, the saying *look after the pennies and the pounds will look after themselves* is all too true when it comes to using these pairs in a real design.

The LUT can implement any function of four inputs regardless of how many gates are needed to describe it. The output of the LUT then feeds directly into the D input of the flip-flop (see Figure 1). Designs that pack together this well will be small and high-performance.

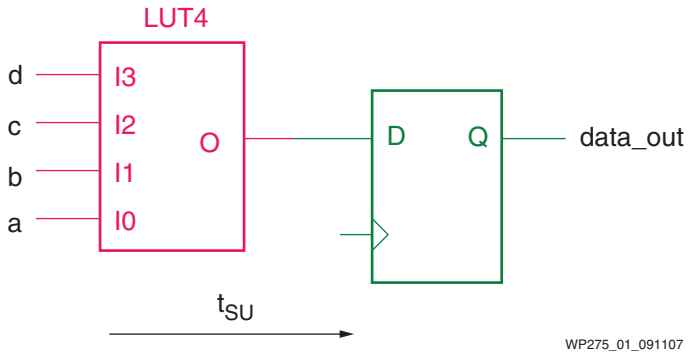

Figure 1: **Simple Logic Using a LUT and a Flip-Flop**

Whether you describe this implementation in VHDL or Verilog, the result should be the same. In the examples below, a simple AND gate has been used to represent the four-input logic function.

VHDL Example

```
process (clk)
begin
  if clk'event and clk='1' then
    data_out <= a and b and c and d;
  end if;
end process;
```

Verilog Example

```
always @(posedge clk) begin
  data_out <= a & b & c & d;
end
```

Although some programmable interconnect is used to connect to the LUT inputs, the function itself can be considered to be part of the setup time of the flip-flop, which is certainly less than 1 ns. Therefore, this is the ideal case to always aim to achieve.

Two-Level Logic

As soon as a function has more than four inputs, the synthesis tools have no choice but to split the logic between two or more LUTs in some way. Figure 2 shows the most likely way a six-input AND gate is implemented regardless of language or synthesis tool.

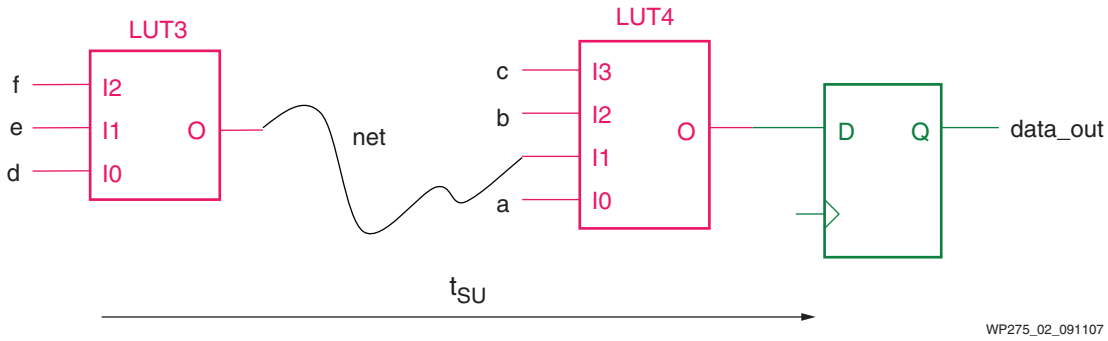

Figure 2: **Two-Level Logic**

Notice how this function is *twice* the cost of the single-level logic. Because of the granularity of the FPGA, there is no way that the function can be just 5% bigger. As soon as the function cannot fit into one LUT, two LUTs must be used. We can also see a significant impact on performance. Although the delay through the additional LUT is also less than 1 ns, the "net" to join the functions adds yet more delay, and the overall impact on setup time can be significant. Of course, this is where time specifications and the place-and-route (PAR) tools play their part in keeping the net delays within limits; however, avoiding this situation obviously makes a design faster as well as smaller.

Adding a Reset

When adding controls to their designs, designers often include a global reset. This reset is especially liked for simulation. However, because a Xilinx FPGA already starts up in a known state following configuration, the global reset is really not necessary. The following VHDL and Verilog examples show how the global reset is implemented.

VHDL Example

```
process (clk, reset)
begin
  if reset='1' then
    data_out <= '0';
  elsif clk'event and clk='1' then
    data_out <= a and b and c and d;
  end if;
end process;
```

Verilog Example

```
always @(posedge clk or posedge reset) begin
  if (reset) begin
    data_out <= 1'b0;
  end
  else begin
    data_out <= a & b & c & d;
  end
```

Each flip-flop has a set of dedicated control inputs to support *set*, *reset*, and *clock enable* controls. Clearly, the synthesis tool should be sensible enough to use these inputs when it can, meaning that the LUT is reserved for the actual functions we are trying to control. As shown in Figure 3, the asynchronous clear (CLR) input to the flip-flop has been used, which means that there is still an optimum single level of logic.

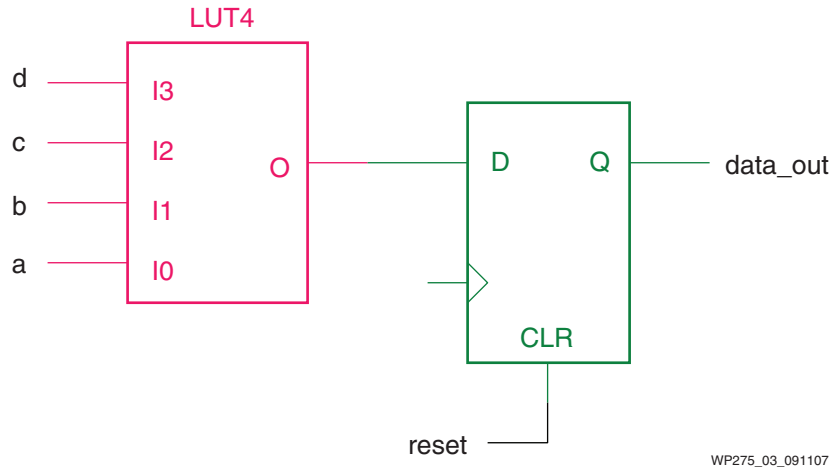

Figure 3: **Addition of Reset**

Adding More Controls

This section shows some additional controls similar to what must be included in real designs. The code below has the same four-input function, but the flip-flop needs reset, set, and clock-enable controls. The remaining examples use VHDL only. The examples are simple and build on the same function used in the previous example.

```
process (clk,reset)
begin
  if reset='1' then                                  } Reset
     data_out <= '0';
   elsif clk'event and clk='1' then
     if enable='1' then                              } Enable
       if force_high='1' then                        } Set
          data_out <= '1';
        else
          data_out <= a and b and c and d;           } Logic
       end if;
     end if;
  end if;
end process;
```

Look at what the Xilinx Synthesis Tool (XST) does with this code (Figure 4). Why has it failed to use the control directly on the flip-flop, which causes our single-level logic to become two-level logic at *twice* the cost and at lower performance?

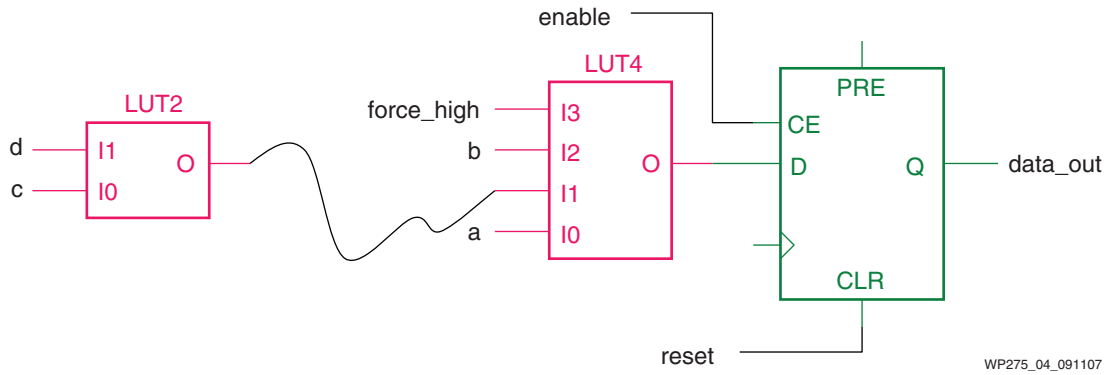

Figure 4: **Additional Controls**

The synthesis tool has done the best that it possibly can. The cause of the problem is the code, which describes something that is not natural to implement directly in the logic slices. The code might have described what the designer wanted it to do, but the synthesis tool was forced to emulate that functionality using the resources available to it. Rather like using a knife blade to tighten a loose screw: it might get the job done, but the screw and the knife will get damaged—and performing the operation is not without a element of danger for the person doing it!

Do Not Mix your Drinks

The flip-flops can support both asynchronous and synchronous reset and set controls. However, they cannot support a mixture of asynchronous and synchronous controls on the same flip-flop. The synthesis tool must therefore choose between a synchronous flip-flop with *SET* and *RST* controls or an asynchronous flip-flop with *PRE* and *CLR* controls (see Figure 5). An asynchronous reset always takes priority and forces the selection.

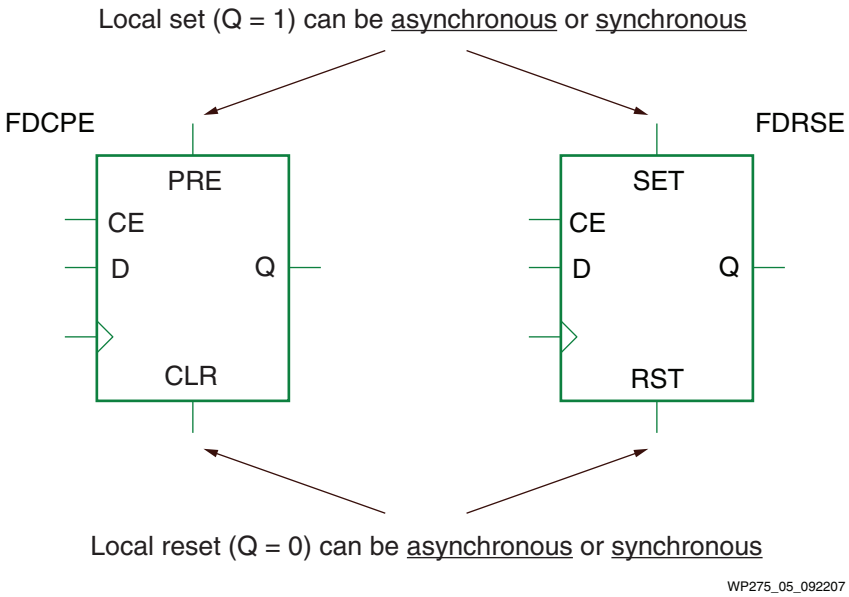

Figure 5: **Asynchronous versus Synchronous Controls**

The asynchronous reset returns us to the problem with having a global reset. All too often that global reset is defined as being asynchronous. Look at your code and verify that it is asynchronous. The previous example used both set and reset controls, which might not be so common; however, there are also cases in which there are two reset conditions for the same flip-flop. One is the global reset, which you have *by default* in the coding style, and the second is a local reset required for operational purposes (i.e., a BCD counter that must roll over to *0* following a *9*).

If the global reset is asynchronous, the local synchronous reset must be emulated using the LUT, with the potential to force two levels of logic at *twice* the cost and at lower performance.

Synchronous Design

If you still insist on having a global reset (unless the reasons against it above convinced you otherwise), try the same example using a synchronous reset to see if it makes things better.

```
process (clk)
begin
  if clk'event and clk='1' then
    if reset='1' then                                  } Synchronous Reset
       data_out <= '0';
     else
       if enable='1' then                              } Enable
         if force_high='1' then                        } Synchronous Set
            data_out <= '1';
          else
            data_out <= a and b and c and d;           } Logic
         end if;
       end if;
    end if;
  end if;
end process;
```

Figure 6 shows what XST does with the synchronous code. Again, it looks like the tool has wasted the dedicated *SET* control on the flip-flop and implemented this control in logic. The result is once again two-level logic at *twice* the cost and with lower performance.

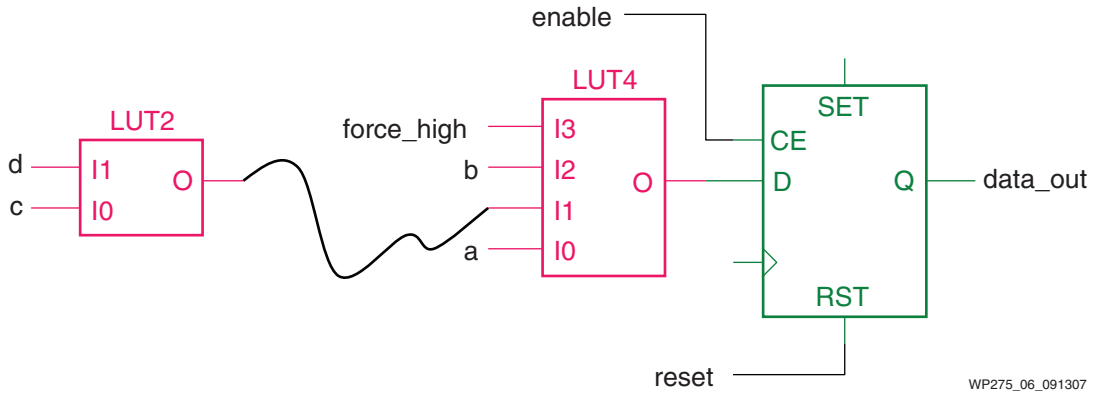

Figure 6: **XST Synchronous Implementation**

Again, XST has done the best that it possibly can. The code asked the synthesis tool to implement something that is not natural, and XST was forced to emulate what was described.

Get your Priorities Right

The key to a successful implementation is to understand the ways the flip-flops can work. Although the FPGA as a whole is programmable, each low-level feature is actually fixed. The programmability comes in the ability to decide if the feature is used or not.

Imagine you are going to use a discrete device. Before using it, you would have to study the data sheet to see the connections and how the controls work. Study Table 1 for a short while. Whatever this device is, its inputs and outputs are consistent with a flip-flop.

Table 1: **Example Discrete Device Characteristics**

Inputs					Outputs
R	**S**	**CE**	**D**	**C**	**Q**
1	X	X	X	↑	0
0	1	X	X	↑	1
0	0	0	X	X	No Change
0	0	1	1	↑	1
0	0	1	0	↑	0

Table 1 shows that the *R* input, which has the highest priority, will reset the *Q* output on the rising edge of *C*. Only if the *R* input is Low can anything else happen. Observe that *S* has the next highest priority, forcing the *Q* output High. Finally, the *CE* input must be High for the *Q* output to follow the *D* input. Not exactly rocket science, but the following information might come as a bit of a shock.

Table 1 was taken from the online Libraries Guide, which is provided with the ISE™ software tools, from the page describing a flip-flop known as an *FDRSE*. This name makes more sense when read from left to right (see Figure 7).

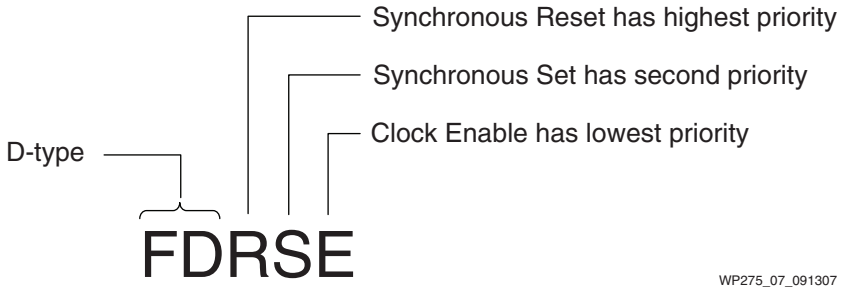

Figure 7: **FDRSE Definition**

All flip-flops inside the FPGA have these same controls, although the asynchronous flip-flop is called an *FDCPE* to reflect the asynchronous clear and preset controls. When more than one control is used, each follows the priority it has been assigned.

In the VHDL code example in "Synchronous Design", the *enable* had a higher priority than the *set*, which did not conform with the order supported by the dedicated

controls. The synthesis tool worked around this issue by using a combination of the dedicated controls, which were in the correct order, and some LUT logic to emulate the rest of the flip-flop that was being described.

Writing Sympathetic Code

We are all very aware of controls and their priorities when we connect external components. Schematic designers are also very aware of the control priorities of flip-flops inside an FPGA, simply because they select the flip-flops manually from the library and then must follow the priority rules to make the components work in their circuits. These aspects are all too easy to ignore when writing HDL code at a higher level of abstraction. Fortunately, since all flip-flop components are the same, once we know their simple rules, it becomes relatively easy to write generic code that is sympathetic to the way in which they work as shown in the following code.

```
process (clk)
begin
  if clk'event and clk='1' then
    if reset='1' then                                  } Synchronous Reset
       data_out <= '0';
     else
      if force_high='1' then                           } Synchronous Set
         data_out <= '1';
        else
        if  enable='1' then                            } Enable
           data_out <= a and b and c and d;            } Logic
        end if;
      end if;
    end if;
  end if;
end process;
```

With the code now correctly describing what can be achieved naturally, XST is able to implement the single-level logic we had been expecting all along—a small and high-performance implementation (see Figure 8).

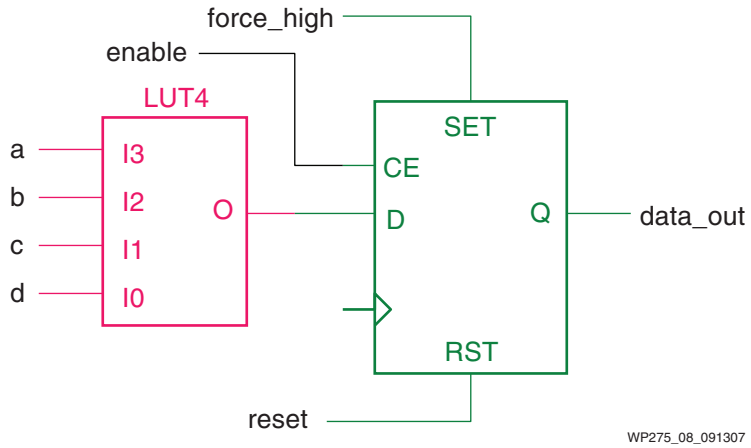

Figure 8: **Smaller, Higher Performance Implementation**

The Challenge

Look at the HDL code that you have been writing for your most recent design and observe whether or not you *get your priorities right*. How many times do you have an asynchronous reset that prevents any subsequent synchronous reset or set from being implemented directly at the flip-flop? How many times do you see an enable with higher priority than a synchronous reset? Not all of these issues force another level of logic, but the potential certainly exists.

If your design is working, do not change it! Remember the phrase *if it isn't broken, don't fix it*. For your next design, just think about how you write the code, and you will find many ways to both reduce the size and speed up your design.

Summary

There are always cases that require the priority rules to be broken—this is what programmable logic is all about. However, if you can learn to treat these cases as exceptions and write code that is sympathetic to the priorities, the results will be rewarding. If you remove that unnecessary global asynchronous reset, there is one less priority to worry about every single time you design.

To provide feedback on this White Paper, refer to the Xilinx Forums at http://forums.xilinx.com/.

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
09/21/07	1.0	Initial Xilinx release. Some content taken from previous web postings as a TechXclusive.
10/22/07	1.0.1	Corrected comments next to code following "Adding More Controls".

Notice of Disclaimer