HDL Synthesis
and
Built-In Clock Enables

The internal flip-flops in Xilinx FPGA architectures have built-in, dedicated clock enable (CE) inputs. Appropriate use of these clock enables avoids the need for gating clocks, facilitating good synchro-

nous design techniques. Using these dedicated clock enable resources also avoids having to use the combinatorial logic resources in the logic blocks to implement the same functionality, potentially eliminating an extra level of logic from the design. This, in turn, can minimize delays along critical paths and save valuable logic resources.

The built-in clock enable function is implemented using a multiplexer in front of the flip-flop's data input, as shown in **Figure 1**. When the clock enable signal is not asserted, the Q output of the flip-flop is fed to the D input, holding the flip-flop in its current state regardless of activity on the data and clock inputs. Implementing clock enables in this manner avoids the race conditions that could result if the clock enable line was used to directly gate the clock input. Implementing the same clock enable functionality using a look-up table (LUT) within a configurable logic block (CLB) would consume three inputs to the LUT; in most cases, this would add an additional LUT to the data path to the flip-flop as well as using additional routing resources.

However, to take advantage of the built-in clock enable function, users of HDLs and logic synthesis tools need to be very careful when coding flip-flops. Different coding techniques can yield significantly different circuit implementations. The choice of user options for the synthesis compiler also can affect the synthesis results.

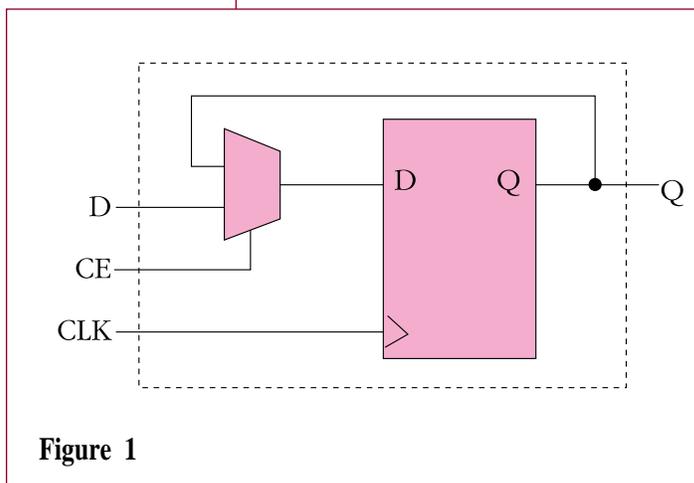Using an HDL, there are a many ways to describe a flip-flop with a clock

Figure 1

CODING EXAMPLE 1

```
always @ (posedge clk or posedge rst)
begin
     if (rst)
          q <= 1'b0;
     else if (clken)
          q <= d;
end
```

CODING EXAMPLE 2

```
always @ (posedge clk or posedge rst)
begin
     if (rst)
          q <= 1'b0;
     else if (clken)
          q <= d;
     else
          q <= q;
end
```

enable. The examples show four unique ways of describing a flip-flop with an asynchronous reset and a synchronous clock enable. *(The information presented below also applies if the asynchronous reset is removed from the description.)* While Verilog is used in the examples, the discussion applies to VHDL coding as well.

These different coding styles can yield differing results, as shown in **Table 1**. Furthermore, user-selected options in the synthesis compiler also can affect the results. For example, in the case of the Synopsys FPGA Compiler synthesis tool, two compilation variables affect flip-flop implementations; these variables control feedback paths in sequential circuits. The variables and their default values are:

```
hdlin_keep_feedback "FALSE"
hdlin_keep_inv_feedback "TRUE"
```

The hdlin_keep_feedback variable does not seem to affect the implementation of flip-flops for Xilinx FPGAs, but the hdlin_keep_inv_feedback variable does have a significant effect. (Incidentally, if you type "help hdlin_keep_inv_feedback" at the dc_shell prompt, it informs you that the default value for this variable is FALSE. This is not accurate; the default value is TRUE. This was changed with release 3.2b, and has been TRUE ever since.)

Table 1 summarizes the results of synthesizing code fragment examples 1-4, in terms of whether the built-in clock enable or a multiplexer circuit external to the flip-flop is used

CODING EXAMPLE 3

```
assign d_in = clken ? d : q;
always @ (posedge clk or posedge rst)
 begin
     if (rst)
            q <= 1'b0;
      else
            q <= d_in;
   end
```

CODING EXAMPLE 4

```
assign d_in = ((clken & d) | (~clken & q));
always @ (posedge clk or posedge rst)
begin
     if (rst)
            q <= 1'b0;
     else
            q <= d_in;
```

to implement the clock enable function. The results came from using v3.3b of the Synopsys compiler. *Other logic synthesis compilers may yield different results. If this information cannot be obtained from the documentation supplied with your synthesis tools, than you may want to use the HDL code examples given here to test the operation of your synthesis tool.*

33

Table 1: Synthesis Results Using Synopys FPGA Compiler

hdlin_keep_inv_feedback variable	register size	Example (1)	(2)	(3)	(4)
False	single-bit	CE	CE	CE	MUX
False	multi-bit	CE	CE	CE	MUX
True	single-bit	CE	CE	CE	MUX
True	multi-bit	CE	MUX	MUX	MUX

CE = dedicated clock enable synthesized   MUX = external multiplexer synthesized

34

> **"...*it is* important** *for the designer to understand which circuit implementation will be produced by the synthesis compiler, which is often a function of the user's coding style.* **"**

Example (4) uses an explicit multiplexer equation that is external to the flip-flop code. This will **always** cause the compiler to generate a flip-flop with an external multiplexer, and will not use the dedicated clock enable flip-flop.

When FALSE, the hdlin_keep_inv_feedback variable will **always** cause the compiler to generate a flip-flop with a dedicated clock enable for **examples (1-3)**.

When TRUE (the default), the hdlin_keep_inv_feedback variable causes the compiler to generate circuits that are dependent on both the design and the coding style, as follows:

• If the register is a single-bit entity, the compiler will generate a flip-flop with a dedicated clock enable when using the coding styles of Examples (1-3). As stated above, an external multiplexer will be always be generated for example (4).

• If the register is part of a multi-bit bus (or vector), the synthesis results depend on how the code is written. Example (1) will generate a set of flip-flops with dedicated clock enable (CE) pins. Examples (2) and (3) will generate simple flip-flops with a multiplexer driving the D input. Basically, if you include the "else ➡ q gets q" clause in the code, the compiler interprets this (correctly) as a multiplexer, and will generate the multiplexer externally using feedback from the flip-flop. If you don't, the compiler will use flip-flops with dedicated clock enables.

Which is better? That depends upon the application. Using flip-flops with the dedicated clock enables is most efficient for highest speed and minimal area, and is usually recommended. As noted above, implementing an external multiplexer requires three inputs to a look-up table to generate the multiplexer. In an FPGA's fan-in limited architecture, this wastes resources and often causes an additional LUT delay.

However, there are occasions when an external multiplexer is better than a dedicated clock enable (CE) for placement reasons. Flip-flops within a single configurable logic block (CLB) share a common dedicated CE signal. If only the dedicated CE mechanism is used, there is no physical way to place two flip-flops with different clock enable signals in the same CLB. If the design has many unique, dedicated clock enables, placement problems may result, because once a flip-flop with a dedicated CE is placed in a CLB, no other flip-flop with a dedicated CE can be placed in that same CLB, possibly resulting in "wasted," unusable flip-flops. Flip-flops that do not use the dedicated CE lines have no such restrictions. This phenomenon is more prevalent in the XC5200 family, with four flip-flops per CLB, than in the XC4000 series, with only two flip-flops per CLB.

(A similar placement problem can occur with flip-flops having different clock or asynchronous control signals, since these inputs are also common to all the flip-flops in a CLB. However, most designs do not have enough unique clocks or resets to make this a problem.)

In summary, using coding styles that take advantage of the FPGA's built-in clock enable function usually results in smaller, faster designs. However, placement considerations may dictate the use of clock enable logic implemented in the CLBs look-up tables. In either case, it is important for the designer to understand which circuit implementation will be produced by the synthesis compiler, which is often a function of the user's coding style. ◆