Tales from the Helpdesk

Synchronization and Edge detection

Doulos - January 2009

The Problem

Quite often your code needs to react to a change on some control signal. That can be an external input, something saying that another part of the circuit has done it's job and that we can continue. All sort of scenarios exist that call for a signal generate by one part of a system to be detected by another part or a different system.
A little while ago, we received a question from an engineer who was engaged in the task of condensing an old system implemented on a board into a single FPGA. He was having trouble with clocks in his FPGA implementation, and a small investigation showed us the problem. The system board consisted of a processor and various peripherals (all separate ICs,) communicating using a tristate bus and all manner of asynchronous control strobes. He had found some HDL IP to implement the peripherals in the FPGA, but alas the IP used all the same tristates, latches and strobes.

Why is it a problem?

All this asynchronous stuff is fine and dandy if you're still in 1980, but not so good if you want to use an FPGA. Why? Well, every time I use code like this to detect an edge on a strobe:

```
if rising_edge(strobe) then
```

Or this

```
always @(posedge strobe)
```

Then the FPGA tools think "Aha! A clock!" and act accordingly. They route the signal in question through a clock buffer, and if you have a few of these edge-detect fragments you pretty soon run out of clock buffers. Then your tools start using general routing for clocks and your nice, dependable, low-skew clocks go completely out of the window.

How do I solve the problem?

Don't do these things.

Oh. You want more? Well, the basic principle is that FPGAs do asynchronous stuff really very badly. They thrive on synchronicity (is that a word?). You've almost certainly heard the old maxim of "Keep It Simple, Stupid", or KISS. Well, for FPGAs we can adapt that to "Keep It Synchronous, Stupid". Yes, unless your design is very small you're going to have to have more than one clock domain, but keep the number of clocks as small as humanly possible. Instead of generating a new divided or gated clock, consider using a clock enable to control one widespread system clock. That makes best use of the clocking resources on the chip.

Don't use internal tristates at all. Don't use latches unless you know exactly why you need them; your FPGA synthesis tool gives you a warning about latches for a good reason: they're almost always a mistake and they really, really, don't help the implementation or timing checking of the design during place and route. Make sure you are familiar with the sort of HDL code that generates latches, so you can avoid writing them .

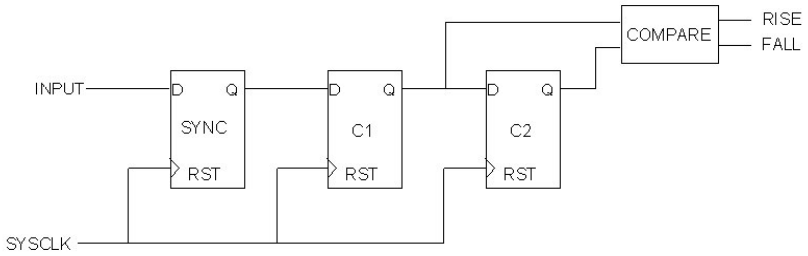
So how do I do that edge detection?

How can you provide a synchronous notification of an asynchronous edge? First thing we need to do is to get a synchronous version of that asynchronous input signal.

Why? (If you know, skip this paragraph.) Because if the input signal changes within a flip-flop's setup and hold times the output may be unusable for various reasons; the flip-flop may or may not have

propagated the new value, or perhaps got confused for a while and not settled to any real output value ("metastability"). If the input signal changes within the "metastability window" the output could take a long (theoretically infinite) time to settle to a stable value. That time could well be longer than one clock cycle, so we add another flip-flop just in case. It's vanishingly unlikely for the second flip-flop to get hit by metastability. Now read on...

Synchronization is conventionally done with a two-stage shift-register that is clocked by the target domain's clock. That's great, our input is now synchronous to the sysclk. We need to detect a change, and we can do that by extending the shift-register a little and comparing the values of different bits:



The first stage of the shifter takes the hit of timing errors, and we compare the next two stages (C1 and C2) to see whether we have a rise or a fall on the input signal. We output two signals that indicate a rise or a fall; each will be high for one sysclk cycle. The compare logic is trivially simple.

Other things to think about

How are you going to use the riser and fall flags? They are high for only one cycle at a time, so you if you need the indication for longer you may need to register their values. They are synchronous to sysclk, so to use them in blocks with other clocks you will of course need to synchronize them.

What is the nature of the thing generating the input signal? Is it a clean change from one value to another or does it glitch or bounce? You might need to clean it up with some sort of glitch filter before you use this edge-detection. If your signal comes from something really messy like a mechanical switch you may need something more than the usual shift-register deglitcher, but that's for another article.

Another lesson about synchronization: A colleague tells me about a horrible bug he uncovered for a customer. The customer's design had two FSMs that kept getting into deadlock, and he'd burned a few fuse-based chips trying to diagnose the problem. Like the nastiest bugs it was intermittent and hard to reproduce. RTL simulation was absolutely perfect, so what was the issue? The engineer had very carefully synchronized signals between clock domains, as you do, but in his eagerness to obey the rules he had synchronized one signal into a clock domain in two different places. As a result, minute timing differences in the paths to these two synchronizers meant that the two FSMs working together saw different vales for this signal. Hence the deadlock. Bottom line: only synchronize a signal once per clock domain.

Conclusion

Don't do asynchronous edge detection, ever. This circuit is a nice edge detector that gives you synchronous notification of edges on your input signal. There's no excuse for not doing this; it's a tiny circuit in just five lines of Verilog. A bit more in VHDL, obviously...

Remember: Keep It Synchronous, Stupid.