Tales from the Helpdesk

Cleaning Dirty Signals

Doulos - April 2009

All engineers who attend our courses are invited to send their questions to us in the weeks and months that follow the course, as they learn to apply the knowledge they gained during the few days of the course. The issue we're looking at here came from an email from an attendee.

The Problem

Most engineers know that they need to pay attention when a signal crosses from one clock domain to another, and they know how to do this: feed the signal through a couple of flip-flops clocked by the target domain and all is well, more or less. However, sometimes the source signal is a bit dirty and contains glitches that mean you get rather more transitions than you either expect or want; such a source might be a sensor, for example. Sometimes the source of the signal is something that by its very nature will produce a signal that bounces or oscillates for some time before reaching a stable high or low state. How do we deal with these?

Solution #1

If your source signal carries glitches for only a few clock cycles then the easy way to clean it is to extend the usual synchronizer shift-register. Choose how long it is based on how long it takes for the signal to stabilize.

Leaving the first stage in the shifter for synchronization, update the output of the shifter when all the other stages agree on the value. Here's an example in Verilog:

```
always @ (posedge clock) begin
  if (shifter[filter_width:1] == 0) begin
    filtered <= 0;
  end // if
  else if (shifter[filter_width:1] == 1) begin
    filtered <= 1;
  end  // else
  shifter <= { shifter[filter_width-1:0], raw};
end // always
```

Note that in this example the length of the shifter is not fixed. It's hard to over-emphasize the importance of making your code as flexible as you reasonably can. In this case we've given the module a parameter called `filter_width`.

Solution #2

The deglitcher outlined above is not great when the signal may bounce for a long time. If you have a mechanical switch as an input to a system with a fast clock, the signal may be bouncing for hundreds of cycles. How do we deal with that? We don't want to create a deglitching shift-register with hundreds of stages, so we need something that makes more efficient use of resources.

Instead of a shift-register we use a counter. Every time the input changes value we reset the counter, and the output is updated only when the counter reaches its timeout. Here's some code, in VHDL this time:

```
if rising_edge(clock) then
  if raw /= prev_raw then
    -- The input just changed.  Reset the timeout.
    counter := filter_width;
  elsif counter /= 0 then
    -- Input stable, but timer not yet expired.  Keep timing.
    counter := counter - 1;
  else
    -- Input stable, and counter has expired.  Update the output.
    filtered <= prev_raw;
  end if;
  -- Keep track of the most recent input.
  prev_raw := raw;
end if;
```

If you're not familiar with VHDL, please note that you'll need to choose the appropriate packages and/or types to be able to do that decrement operation (and those accustomed to VHDL's uncompromising exactitude will probably have done this without thinking about it).

In this case, and as you can see if you download the code, we chose to use an integer type for the counter. This saves us from having to calculate a vector size and leaves that job to the synthesis tool. It also means that we can use the "-" operator without having to use a vector arithmetic package. In the Verilog equivalent a vector is all we have, of course, so the choice becomes a little different. Do we use an integer and rely on the synthesis tool to optimize away the unused bits? That's not very elegant and can contribute distracting noise to the synthesis log, so instead we calculate the vector size from the desired filter width using a function:

```
function automatic integer bits_to_fit (input integer N);
  if (N==0)
    bits_to_fit = 0;
  else
    bits_to_fit = 1 + bits_to_fit(N/2);
endfunction
```

Note that the function is recursive, which requires the declaration to include the word automatic, meaning that storage for arguments and so on is allocated for each call, not each declaration. The function could be implemented without recursion, but that's less interesting...