Verilog Code Review #2

Scott Borisch

Jan 8, 2021

Overview

- Multiple Clock Domains
- Resets – Sync vs. Async

Clock Domains

- FPGA Designs may require multiple clocks for many reasons, but the most common are:
  - Using a protocol that requires a specific speed
    - SPI interface – use a clock that is multiple (usually at least 4x) of the SPI Clock
    - Protocols that use “embedded clocks”: Ethernet, PCIe
      - Use a SERDES + PLL to extract the embedded clock
  - Have a certain processing rate requirement

Clock Crossings

- If the design requires multiple clock domains, determine which functional blocks will run at what frequency.
  - Determine how to do clock crossings.
  - These are important architectural choices – do up front.
  - In general, minimize number of clock domains.
    - However, there are often reasons you can't do this:
      - You need to do some processing at a very high clock rate. But other logic does not need to run this fast, and doing so will make static timing much more difficult.

Sync vs. Async Clock Crossings

- You can do a sync crossing: IF your clocks have
  - an integer multiple frequency relationship
  - Are frequency locked
  - Are phase locked
- You **can** (but don't have to) do a synchronous clock crossing
  - This means you can't change the clock frequencies in a way that violates the above
  - Also, timing constraints at the crossing will be more demanding.
  - For these reasons, always doing asynch crossings are my preference.

Sync vs. Async Clock Crossings

- Types of Clock Crossings
  - Single bit: synchronizer flops (at least two)
  - Multi-bit
    - Use synchronizer flops for a “go strobe”
    - Cross the multi-bit value as controlled by the single-bit “go strobe”
  - Multi-bit: Gray Code
    - Works for values that can only stay the same/increment/decrement (counters/address pointers – sometimes)
  - Dual Clock (Async Fifo)
    - Provides much higher transfer rate than multi-bit with go strobe
    - As fast as the slower of the two clocks involved
    - FPGA vendor often provides this as IP (for free)

Resets: async vs. sync

- Sync vs. async – most FPGAs will provide both options
- However, they often recommend one will give better performance
  - This is often a good reason to one vs. the other
- In general, pick one style and use it for all your resets (unless there is a good reason not to)
- Sync reset requires a (typically rising) clock edge for the reset to take effect.
- Async reset does not.
- However, just because a reset is async does not mean you can’t violate timing constraints !! Don’t toggle the reset too close to the active (rising) clock edge.
  - This is called removal/recovery time. Analogous to setup/hold time.

Resets: async vs. sync

- Your reset must be synchronized to your clock – if it doesn’t come this way, use synchronizer flops (just like a clock crossing) to make it synchronous.
- Resets typically have a VERY high fanout. This can be a static timing problem.
  - To get around this, you can:
    - Add duplication flops – this way each flop drives a lower fanout.
    - Add pipeline delayed versions of reset – drive different modules from different pipeline stages.
    - A combination of the above.
- If possible, do NOT have logic that requires that the entire design comes out of reset at the same time in order to work properly. If all your logic works this way, you can have (almost) arbitrary levels of pipelining on your resets.

Static Timing Tricks

- Another way to work around static timing problems caused by resets:
  - Timing violations only happen when reset toggles near a clock edge, so:
    - For async resets, simply disable your clock while asserting/deasserting the reset !
    - For synch resets, you can achieve the same effect by:
      - Disable clock
      - Enable reset
      - Enable clock => wait
      - Disable clock
      - Disable reset
      - Re-enable clock

“Clean” Clocks

- In order for static timing analysis to work properly it assumes:
  - Your clock is always at a specified frequency (or slower)
  - Your clock does not glitch (runt-pulses)
- If at all possible, make sure your clock will NEVER violate the above assumptions.
- If this is impossible to avoid at startup, you must issue your reset AFTER the startup condition has passed.
- If this happens during normal operating conditions – your design will not work properly !

"Clean" Clocks

- What can cause a clock to run at an incorrect speed (transiently)?
  - Startup condition of a PLL. Some PLLs may have this issues, others won't. Check datasheet carefully.
- What can cause a clock to glitch?
  - Some designs require muxing between different clock frequencies. Changing frequencies would be a configuration change – this wouldn't happen during normal operation (at least for any design I have ever seen…)
  - Some FPGAs provides glitchless clock muxes to avoid this issue.
  - Not all clock muxes are glitchless – read the documentation.