Clock Domain Crossing Guidelines

Michael Csoppenszky

Last Updated: May 11th, 2017

Even with outstanding and robust clock domain crossing (CDC) components such as synchronizers and FIFO's, much of what one is trying to accomplish in attaining a particular reliability goal can be nullified if the synchronizers are used incorrectly. This document will outline a number of important guidelines to keep in mind.

(1) Partition the Design to Minimize the Total Number of Clock Domain Crossings

Before attempting to instantiate CDC components such as synchronizers or FIFO's to aide in reliably crossing asynchronous clock domains, it's important to first determine if a given crossing needs to even exist at all. The more clock domain crossings a given device has, even if handled correctly, will degrade the reliability of the overall device.

The mean time between failure (MTBF) of a synchronizer is expressed as:

$$MTBF = \frac{e^{T/\tau_{switch}}}{f_{\varnothing} f_{in} \Delta} \qquad \textbf{\textit{Equation 2}}$$

Where,

T is the time by which the output of the latch settles to a logic 0 or 1 (settling time)

$\mathbf{t_{switch}}$ is the time constant for the transition from a metastable state to a valid state

$\mathbf{f_{\theta}}$ is the synchronization clock frequency (sampling clock)

$\mathbf{f_{in}}$ is the data frequency (data input that is to be synchronized)

The above metric of reliability is for a single synchronizer. The entire device will have an MTBF of the synchronizer divided by how many of those synchronizers are used in the entire design. This assumes events leading to the metastability failure of any given synchronizer are independent of the other synchronizers, which is most often the case. For the most optimal reliability, as few synchronizers as possible should be used in the design. Also, each path that flows through a synchronizer will experience some latency, so design partitioning can be very important from a performance perspective as well. Imagine that a device requires three subtasks, A/B/C, to complete a particular operation, and that we know two clock domains will be involved. Assume subtask A must be in clock-domain 1 and C must be in clock-domain 2. Subtask B can be put into domain 1 or 2, but assume also that there is heavy communication between subtasks B and C. Thus, given the above assumptions, it would not make sense to locate logic for subtask B in clock domain 1, since potentially many additional synchronizations, and thus chances for failure, would be created. This would in effect create a synchronization loop from clock-domain 1 -> 2 -> 1 to complete a single subtask. Also, the overall task would be lower in performance, due to all the unnecessary clock domain crossing latencies. The central message here is to partition logic amongst the various clock domains at points of minimum communication, this will result in a design that is higher in both reliability and performance.

(2) Use FIFO CDC Components For Stream Oriented Data Crossings

If the particular design must shuttle a significant amount of data between clock domains and that data is fairly stream oriented, avoid synchronizing each and every item of data from one clock domain into another. For this situation, a multi-synchronous capable FIFO memory structure should be used to effectively straddle both clock domains. An example of such a FIFO is the ***jpl_cdc_fifo_a_fwft***. The clock domain receiving the data only needs to know when there is data in the FIFO to be read, and when it's empty. Otherwise, it can be read independently of the clock domain writing data into the memory. The writing side is symmetrical, it only needs to know if the FIFO is full or not. Again, utilizing this type of approach will result in lower average latency, fewer required synchronizers, fewer synchronized events, and thus better reliability and performance.

(3) Synchronize First, then Distribute

Figure 1 shows an effective implementation of a multi-bit clock domain crossing, when using a FIFO isn't appropriate. To preserve coherence of the of the group of signals crossing clock domains, it's effective to associate only one synchronized "tag" with a group of data inputs, a "packet". Going along with the theme of minimizing the total number of synchronizers, a tag should be associated with as large a packet of signals as possible.

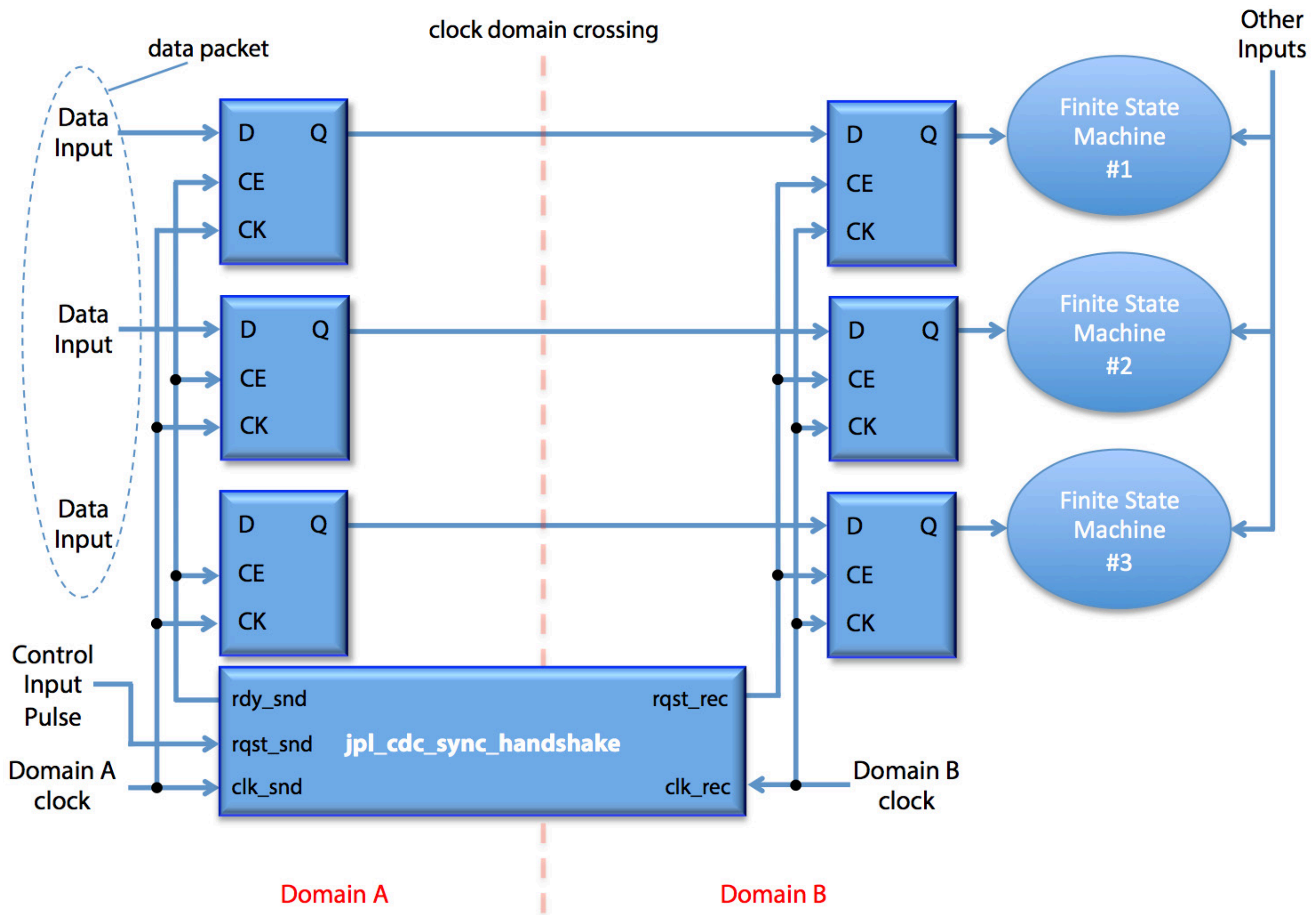

Figure 1 - Proper Use of a Synchronizer

Imagine that the packet of input signals in Figure 1 are related, such as a data or address bus for example. Although this same issue holds if it had been a coherent group of control signals. Without careful thought, it would have been possible for each bit of the packet to pass through an independent synchronizer, ie. distribute first, then synchronize approach. Recall that the exact latency through a synchronizer is not deterministic, although the variance is bounded. If that particular approach had been followed, it's equivalent to having done nothing at all, as the data bits moving across the clock domain boundary can easily not remain a coherent group, as some bits may affect the FSM's (Finite State Machine) shown in the example faster than the others. The impact of a loss of coherency can easily be devastating for a system. A much more effective way of handling coherent groups of signals is to associate a **single tag** with a **single packet** of signals, where only the tag is synchronized and **not** the packet itself. With this approach, the packet of data will remain coherent when it arrives into the destination clock domain.

The only scenarios where it's possible to pass a group of signals through individual synchronizers (one synchronizer per bit) and achieve a correct CDC outcome is when the group of signals is (1) encoded in Gray code, or (2) the signals have absolutely no relationship to one other - ie. each bit is totally independent of, and unrelated to, the other bits.

Note that the example shown in Figure 1 makes use of the ***jpl_cdc_sync_handshake*** synchronizer from the JPL CDC library, which contains useful built-in flow-control features. Using that particular synchronizer, the example in Figure 1 could be between any clock frequency arrangement (fast to slow, slow to fast, etc) and still work reliably. **The key is that the sending domain must keep the packet constant until the receiving domain has consumed it**. The "rdy_snd" output of the handshake synchronizer, which is the built-in handshake mechanism, serves that exact purpose. Only when the "rdy_snd" signal is asserted high is the sending domain allowed to update the data packet to new values. And the sending domain must keep the data packet constant until the next time the "rdy_snd" signal becomes asserted high. Note that the "rqst_snd" input to the ***jpl_cdc_sync_handshake*** synchronizer is expected to be a one-cycle asserted high pulse, and the "rqst_rec" output it produces will also be a one-cycle pulse.

(4) Use the Appropriate Synchronizer Type

It's critical to know the possible range of frequencies on both sides of a given clock domain crossing before you decide which type of synchronizer to use. The only situation where it's appropriate to use a unidirectional synchronizer, such as the **jpl_cdc_sync_bit**, is when you know with 100% certainty that it's never possible for the sending domain clock to operate at a faster frequency than the receiving domain clock. There are very few exceptions to this rule, and the vast majority of those situations involve detailed knowledge of design modes, design states, etc.

If you aren't extremely familiar with any such modes and/or states, it's extremely ill-advised to try and use a unidirectional synchronizer for situations where the sending domain clock is operating at a faster frequency than the receiving domain. In that case, it's best to use a full-handshake synchronizer such as the following one that's built for such a scenario: ***jpl_cdc_sync_handshake***.

So in summary, the following are the appropriate synchronizers to use in these situations :

- Sending domain clock is *__always__* slower than receiving domain clock: **jpl_cdc_sync_bit**
- Sending domain clock may be faster than receiving domain clock: **jpl_cdc_sync_handshake**

(5) Ensure the Input to a Synchronizer is From an Identity Path

Ensuring a design provides a signal to the input of a synchronizer that only contains true transitions is very important. In other words, if the input to a synchronizer comes via combinational logic, it is very likely that the signal may transition multiple times during a cycle before settling on a final value. This due to variations in the timing paths through the combinational logic, as well as variations in routing delays. Such "false" transitions, or glitches, can easily cause false information to pass through the synchronizer, which can be fatal. The solution is simple, ensure that every path to a synchronizer input does indeed come from an identity path from a flop output. An identity path is one that begins from a register output and only passes through buffers or inverters, and does not involve any combinational logic whatsoever.

Note that the ***jpl_cdc_sync_handshake*** has a built-in input stage flop, so this is already handled for you. However, the ***jpl_cdc_sync_bit*** synchronizer does not have such an input stage flop, so the designer must ensure the input to that synchronizer should come directly from a flop output - with NO combinational logic involved.

(6) Ensure the Tag/Packet Timing Relationship at the Destination Domain

Figure 1 is a good example of the general use of a synchronizer. One must guarantee that the signals comprising the packet arrive at the flip-flops in the destination clock domain **no later** than when the tag arrives. In addition, one must also ensure that the packet remains stable until the destination domain has clocked-in, or consumed, the packet. When sending packets from a domain that is lower in frequency to one that is higher, that last rule is fairly simple to observe. However, when sending packets from a higher-frequency domain to one that is of lower frequency, concepts like flow-control handshakes must be considered. Thus the bidirectional synchronizer, such as the ***jpl_cdc_sync_handshake***, offers the "rdy_snd" output to implement an effective and low-latency flow-control mechanism that surrounding logic can make use of.

(7) Use a Formal CDC Tool to Analyze the Design along with Functional Simulation

Although the use of a CDC (Clock Domain Crossing) tool that employs formal methods is most definitely useful and can add significant value, it's important to know that passing cleanly through a formal CDC tool's analysis is not a guarantee of CDC correctness. So then what's the point of using such a tool? Using a CDC tool is far better than not using one - in that today's CDC tools are good enough to expose a good variety of CDC issues that would be extremely time consuming to uncover manually. So there's definitely a lot of value in using such a tool, but it's important to know that one cannot rely on a CDC tool alone to "bless" a design for CDC correctness.

The design team must still understand all of the guidelines presented in this paper, and be very cognizant of these guidelines throughout the entire micro-architecture, logic design, and physical design phases - and then use a CDC tool along with special randomized functional simulation as a means of providing a double-check of correctness. Note that if the synchronizers and FIFO's used in a design are not from the golden JPL CDC library, the ability to perform meaningful CDC analysis will be quite a bit more complex and very time consuming.

(8) All Synchronizers and FIFO's Must be from the Golden JPL CDC Library

Encapsulating the clock domain crossings with approved CDC components is extremely important on a number of different levels. And the only officially approved CDC components are those in the golden CDC library. First and foremost, the crossings require synchronizers and FIFO's that are implemented correctly and robustly. There is no value-add, and substantial risk and other very significant drawbacks in each designer coming up with their own CDC components. As such, doing that isn't allowed. Next, the efficacy of the synchronizer itself, as discussed previously, is heavily influenced on physical design parasitics (capacitance, RC delays, etc). Minimizing those parasitics is highly important to achieve a robust MTBF, and as such, it's extremely important that the synchronizers used in a design be built as self-contained components in a manner in which the PAR tools will honor as an indivisible unit during placement. And by "indivisible unit" it's meant the PAR tools will guarantee that all the sub-components used *within* the synchronizer design always reside in extremely close relative physical proximity to one another on the chip. Doing that will minimize the physical design parasitics that are so detrimental to MTBF performance. And all of this is already taken into consideration with the JPL CDC library components.

Further, ensuring the synchronizers and FIFO's used in a design all come from the JPL CDC library is highly important to enable the time-efficient verification of the correctness of the clock domain crossings employed on a chip. Without that, verifying the correctness of the clock domain crossings will be complex and very time consuming.