

# Understanding Clock Domain Crossing (CDC)

---

## Understanding Clock Domain Crossing (CDC) Checks and Techniques

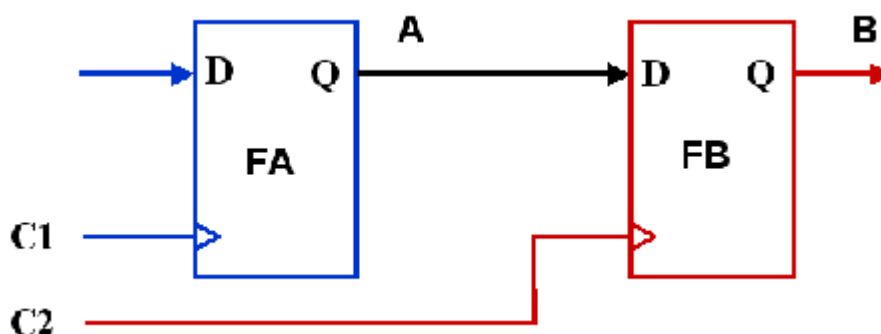
---

### Introduction: What is Clock Domain Crossing?

---

SoCs are becoming more complex these days. A lot of functionality is being added to chips and data is frequently transferred from one clock domain to another. Hence, clock domain crossing verification has become one of the major verification challenges in deep submicron designs.

What is clock domain crossing? A clock domain crossing occurs whenever data is transferred from a flop driven by one clock to a flop driven by another clock.



#### 1. Clock domain crossing.

In *Figure 1*, signal A is launched by the C1 clock domain and needs to be captured properly by the C2 clock domain. Depending on the relationship between the two clocks, there could be different types of problems in transferring data from the source clock to the destination clock. Along with that, the solutions to those problems can also be different.

Traditional methods like simulation and static timing analysis alone are not sufficient to verify that the data is transferred consistently and reliably across clock domains. Hence, new verification methodologies are required, but before devising a new methodology it is important to understand the issues related to clock domain crossings properly. Different types of clock domain crossings are discussed here along with the possible issues encountered in each one of them and their solutions. A new verification methodology is then proposed which will ensure that data is transferred correctly across clock domains.

In all the subsequent sections, the signal names shown in *Figure 1* are directly used. For example, C1 and C2 imply the source and destination clocks respectively. Similarly A and B are used as source and destination flop outputs respectively. Also, the source and destination flops are assumed to be positive edge triggered.

# Clock Domain Crossing Issues

This section describes three main issues which can possibly occur whenever there is a clock domain crossing. The solutions for those issues are also described.

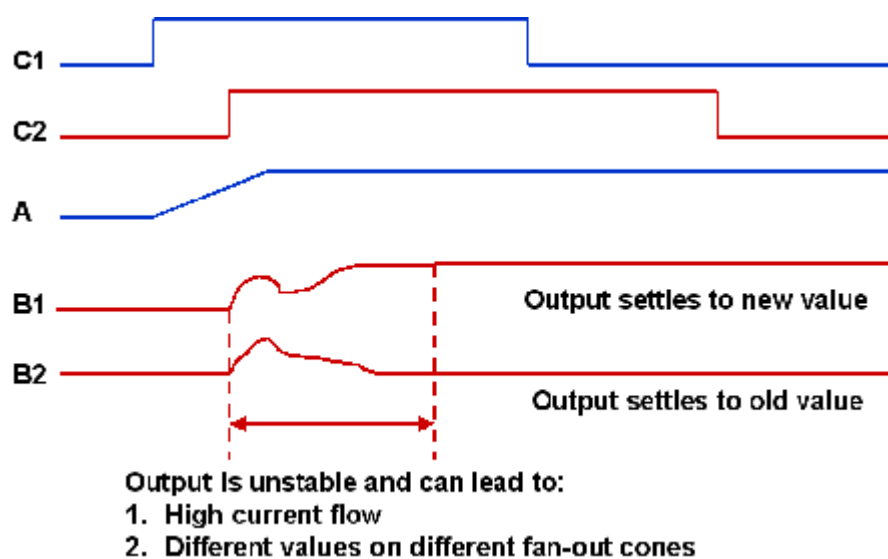
## 1. Metastability

**Problem.** If the transition on signal A happens very close to the active edge of clock C2, it could lead to setup or hold violation at the destination flop “FB”. As a result, the output signal B may oscillate for an indefinite amount of time. Thus the output is unstable and may or may not settle down to some stable value before the next clock edge of C2 arrives. This phenomenon is known as metastability and the flop “FB” is said to have entered a metastable state.

Metastability in turn can have the following consequences from a design perspective:

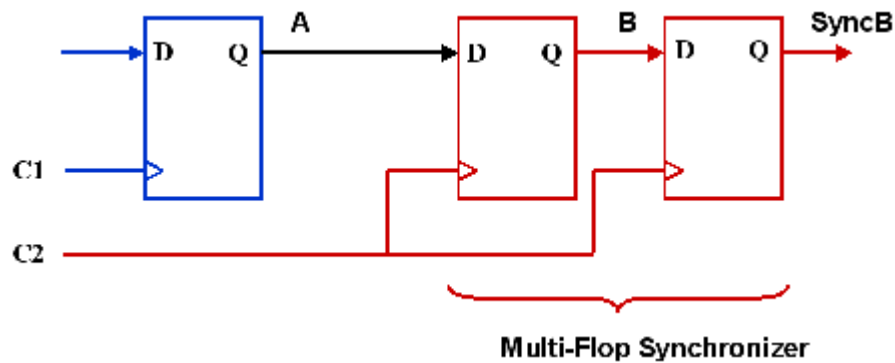
1. If the unstable data is fed to several other places in the design, it may lead to a high current flow and even chip burnout in the worst case.
2. Different fan-out cones may read different values of the signal, and may cause the design to enter into an unknown functional state, leading to functional issues in the design.
3. The destination domain output may settle down to the new value or may return to the old value. However, the propagation delay could be high leading to timing issues.

For example, see *Figure 2* . If the input signal A transitions very close to the posedge of clock C2, the output of the destination flop can be metastable. As a result it can be unstable and may finally settle to 1 or 0 as depicted by signals B1 and B2.



2. Metastability has consequences.

Solution. Metastability problems can be avoided by adding special structures known as synchronizers in the destination domain. The synchronizers allow sufficient time for the oscillations to settle down and ensure that a stable output is obtained in the destination domain. A commonly used synchronizer is a multi-flop synchronizer as shown in *Figure 3*.



### 3. Multi-flop synchronization.

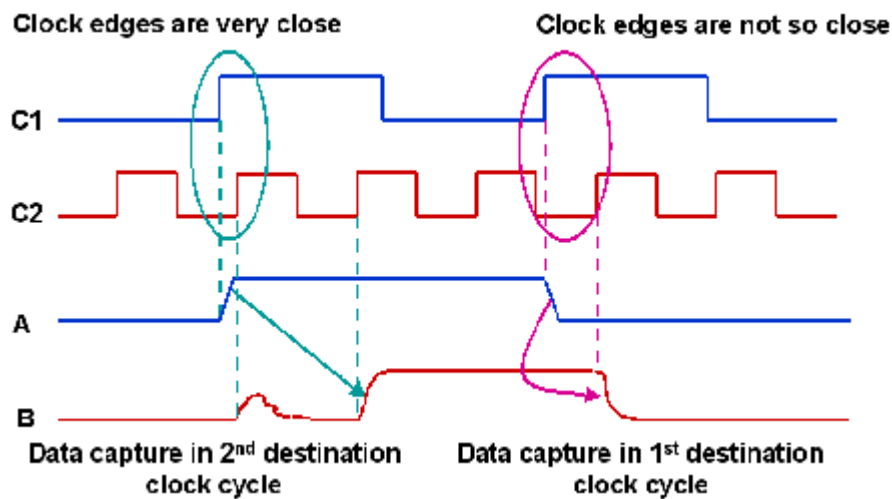
This structure is mainly used for single and multi-bit control signals and single bit data signals in the design. Other types of synchronization schemes are required for multi-bit data signals such as MUX recirculation, handshake, and FIFO.

## 2. Data Loss

**Problem.** Whenever a new source data is generated, it may not be captured by the destination domain in the very first cycle of the destination clock because of metastability. As long as each transition on the source signal is captured in the destination domain, data is not lost. In order to ensure this, the source data should remain stable for some minimum time, so that the setup and hold time requirements are met with respect to at least one active edge of destination clock.

If the active clock edges of C1 and C2 arrive close together, the first clock edge of C2, which comes after the transition on source data A, is not able to capture it. The data finally gets captured by the second edge of clock C2 (*Figure 4*).

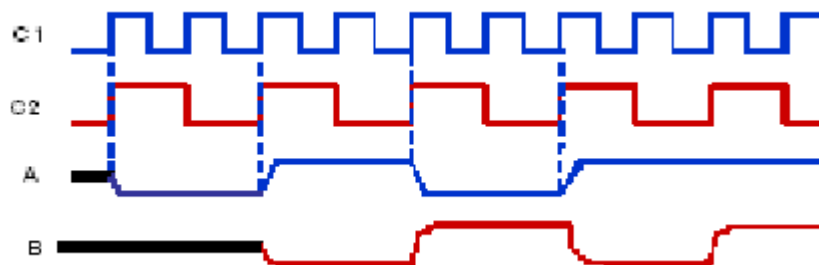
However, if there is sufficient time between the transition on data A and the active edge of clock C2, the data is captured in the destination domain in the first cycle of C2.



#### 4. Effect of metastability on data capture.

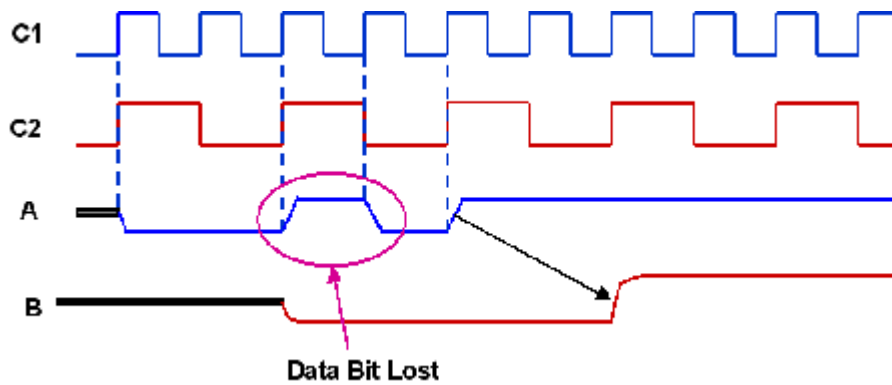
Hence, there may not be a cycle by cycle correspondence between the source and destination domain data. Whatever the case, it is important that each transition on the source data should get captured in the destination domain.

For example: Assume that the source clock C1 is twice as fast as the destination clock C2 and there is no phase difference between the two clocks. Further assume that the input data sequence “A” generated on the positive edge of clock C1 is “00110011”. The data B captured on the positive edge of clock C2 will be “0101”. Here, since all the transitions on signal A are captured by B, the data is not lost. This is depicted in *Figure 5*.



#### 5. No data is lost in this case.

However, if the input sequence is “00101111”, then the output in the destination domain will be “0011”. Here the third data value in the input sequence which is “1” is lost as shown in *Figure 6*.



6. *Data is lost in this case.*

**Solution.** In order to prevent data loss, the data should be held constant in the source domain long enough to be properly captured in the destination domain. In other words, after every transition on source data, at least one destination clock edge should arrive where there is no setup or hold violation so that the source data is captured properly in the destination domain. There are several techniques to ensure this.

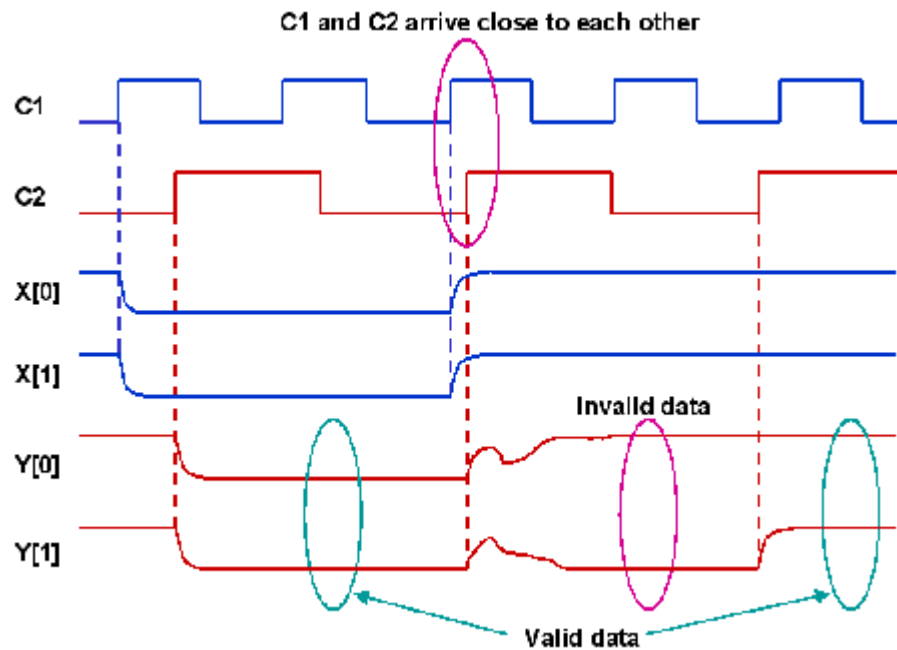
For example, a finite state machine (FSM) can be used to generate source data at a rate, such that it is stable for at least 1 complete cycle of the destination clock. This can be generally useful for synchronous clocks when their frequencies are known. For asynchronous clock domain crossings, techniques like handshake and FIFO are more suitable.

### 3. Data Incoherency

**Problem.** As seen in the previous section whenever new data is generated in the source clock domain, it may take 1 or more destination clock cycles to capture it, depending on the arrival time of active clock edges. Consider a case where multiple signals are being transferred from one clock domain to another and each signal is synchronized separately using a multi-flop synchronizer. If all the signals are changing simultaneously and the source and destination clock edges arrive close together, some of the signals may get captured in the destination domain in the first clock cycle while some others may be captured in the second clock cycle by virtue of metastability. This may result in an invalid combination of values on the signals at the destination side. Data coherency is said to have been lost in such a case.

If these signals are together controlling some function of the design, then this invalid state may lead to functional errors.

For example: Assume that “00” and “11” are two valid values for a signal  $X[0:1]$  generated by clock C1. As shown in *Figure 7*, initially there is a transition from 1→0 on both the bits of X. Both the transitions get captured by clock C2 in the first cycle itself. Hence the signal  $Y[0:1]$  becomes “00”.



#### 7. Data coherency is lost in this case.

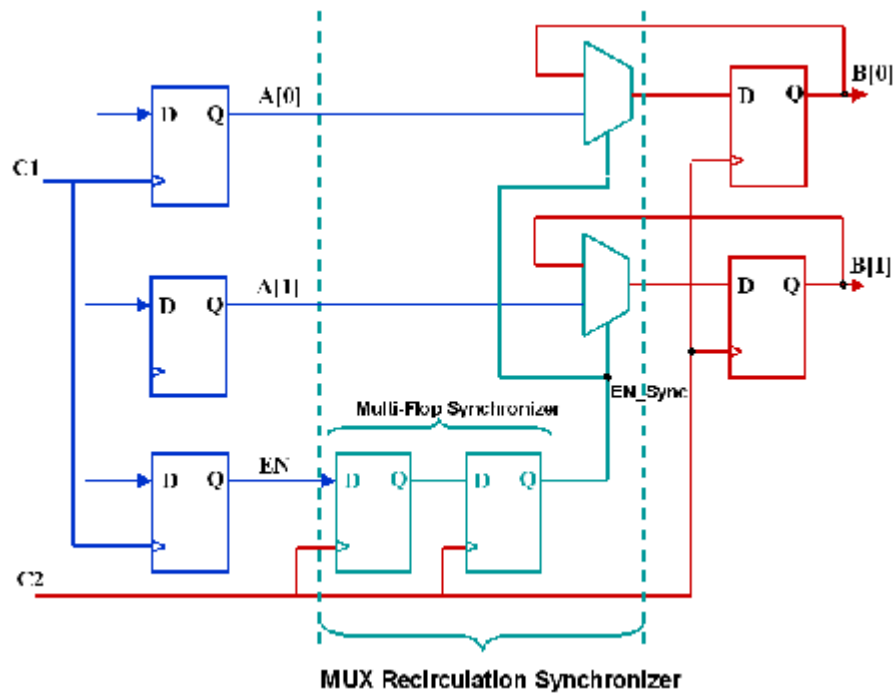
Next, there is a transition from 0->1 on both the bits of signal X. Here the rising edge of clock C2 comes close to the transition on signal X. While the transition on X[0] is captured in the first clock cycle, the transition on X[1] gets captured in second clock cycle of C2. This results in an intermediate value of “10” on Y[0:1] which is an invalid state. Data coherency is lost in this case.

**Solution.** In the above example, the problem results because all the bits are not changing to a new state in the same cycle of destination clock. If all the bits either retain their original value or change to the new value in the same cycle, then the design either remains in the original state or goes to a correct new state.

Now, if the circuit is designed in such a way that while changing the design from one state to another, only one bit change is required, then either that bit would change to a new value or would retain the original value. Since all the other bits have the same value in both the states, the complete bus will either change to the new value or retain the original value in this case.

This in turn implies that if the bus is Gray-encoded, the problem would get resolved and an invalid state would never be obtained.

However, this is applicable only for control busses as it may not be possible to Gray-encode the data busses. In such cases, other techniques like handshake, FIFO and MUX recirculation can be used to generate a common control logic to transfer data correctly. The MUX recirculation technique is shown in *Figure 8*.



### 8. MUX recirculation technique.

[Click here for a larger version](#)

Here, a control signal EN, generated in the source domain is synchronized in the destination domain using a multi-flop synchronizer. The synchronized control signal EN\_Sync drives the select pin of the muxes, thereby controlling the data transfer for all bits of the bus A. In this way, individual bits of the bus are not synchronized separately, and hence there is no data incoherency. However, it is important to ensure that when the control signal is active, the source domain data A[0:1] should be held constant.

## Synchronous Clock Domain Crossings

This section describes various types of synchronous clock domain crossings. Clocks which have a known phase and frequency relationship between them are known as synchronous clocks. These are essentially the clocks originating from the same clock-root. A clock crossing between such clocks is known as a synchronous clock domain crossing. It can be divided into several categories based on the phase and frequency relationship of the source and destination clocks as follows:

- Clocks with the same frequency and zero phase difference
- Clocks with the same frequency and constant phase difference
- Clocks with different frequency and variable phase difference
  - Integer multiple clocks
  - Rational multiple clocks

All the above sub categories may not be used in real designs but are being considered here for completeness and better understanding of the subject.

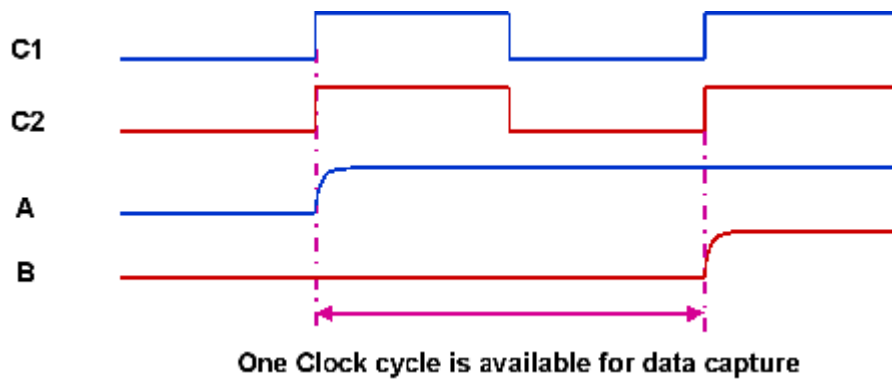
While describing all the above cases, it is assumed that the source clock (C1) and the destination clock (C2) have the same phase and frequency jitter and are balanced with the same specifications of clock latency and skew. It is also assumed that the clocks begin with a zero phase difference between them and the “clock to Q” delay of the flops is 0.

## Clocks with the same frequency and zero phase difference

---

This refers to two identical clocks, as the clocks C1 and C2 have the same frequency and 0 phase difference. Note, that as the clocks C1 and C2 are identical and generated from the same root clock, the data transfer from C1 to C2 is essentially not a clock domain crossing. For all practical purposes, this is the case of a single clock design and is considered here for completeness.

Whenever data is transferred from clock C1 to C2, one complete clock cycle of C1 (or C2) is available for data capture as shown in *Figure 9*.



### 9. Clocks with the same frequency and phase.

As long as the combinational logic delay between the source and destination flops is such that the setup and hold time of the circuit can be met, the data will be transferred correctly. The only requirement here is that the design should be STA (static timing analysis) clean. In that case, there will be no problem of metastability, data loss or data incoherency.

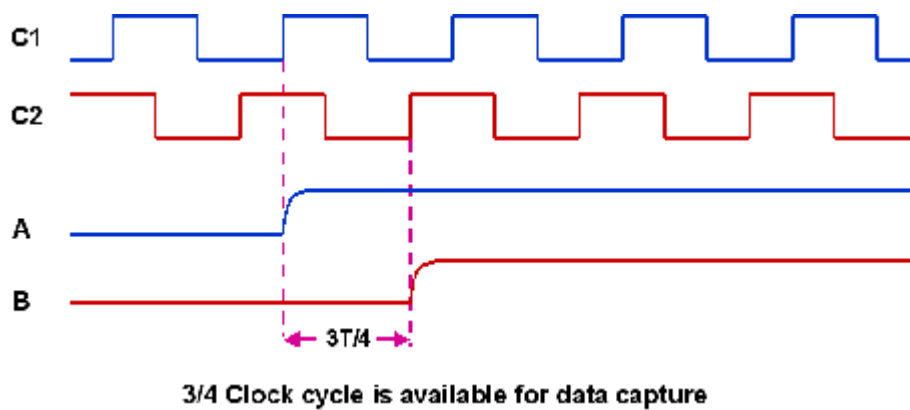
## Clocks with the same frequency and constant phase difference

---

These are the clocks having the same time period but a constant phase difference. A typical example is the use of a clock and its inverted clock. Another example is a clock which is phase shifted from its parent clock, for example by  $T/4$  where  $T$  is the time period of the clocks.

Clocks C1 and C2 have the same frequency but are phase shifted and C1 is leading C2 by  $3T/4$  time units (*Figure 10*).





#### 10. Same frequency, phase shifted clocks.

Whenever data is transferred from clock C1 to C2, there is more restriction on the combinational logic delay due to smaller setup/hold margins. If the logic delay at the crossing is such that the setup and hold time requirements can be met, data will be transferred properly and there will be no metastability. In all such cases, there is no need for a synchronizer. The only requirement here is that the design should be STA clean.

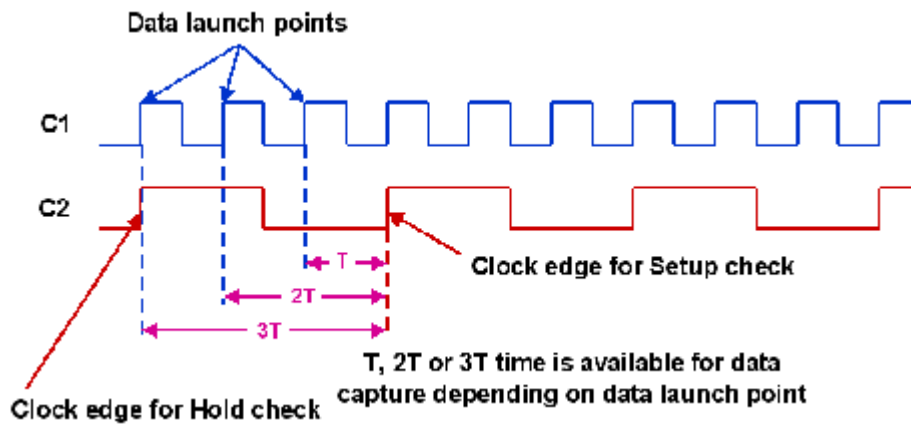
### Clocks with different frequency and variable phase difference

These are clocks which have a different frequency and a variable phase difference. There can be two sub-categories here, one where the time period of one clock is an integer multiple of the other and a second where the time period of one clock is a non-integer (rational) multiple of the other. In both cases, the phase difference between the active edges of clocks is variable. These two cases are described in detail below.

#### A. Integer multiple clocks

In this case, the frequency of one clock is an integer multiple of the other and the phase difference between their active edges is variable. Here the minimum possible phase difference between the active edges of 2 clocks would always be equal to the time period of the fast clock.

In *Figure 11* clock C1 is 3 times faster than clock C2. Assuming  $T$  is the time period of clock C1, the time available for data capture by clock C2 could be  $T$ ,  $2T$  or  $3T$  depending on which edge of clock C1 the data is launched. Hence, the worst case delay of any path should meet the setup time with respect to the edge with a phase difference of  $T$ . The worst case hold check would be made with respect to the edge with 0 phase difference.



### 11. Integer multiple clocks.

In all such cases, one complete cycle of the faster clock is always available for data capture, hence it should always be possible to meet the setup and hold requirements. As a result there will be no metastability or data incoherency and a synchronizer is not needed.

However, there can still be a problem of data loss in the case of fast to slow clock crossing. (That is, the source clock is faster than the destination clock.) In order to prevent this, the source data should be held constant for at least one cycle of the destination clock. This can be ensured by using some control circuit, for example, a simple finite state machine (FSM) would work in this case. In the example shown in *Figure 11*, if the source data is generated once in every 3 cycles of the source clock, there would be no data loss.

For the case of slow to fast crossings, there will anyways be no data loss.

## B. Rational multiple clocks

In this case, the frequency of one clock is a rational or non-integer multiple of the other clock and the phase difference between the active clock edges is variable.

Unlike the situation where one clock is an integer multiple of the other, here the minimum phase difference between the two clocks can be very small- small enough to cause metastability. Whether or not a metastability problem will occur depends on the value of the rational multiple, and the design technology. Three different cases are being considered here with the help of examples.

In the first case, there is a sufficient phase difference between the active edges of the source and destination clocks such that there will be no metastability.

In the second case, the active clock edges of the two clocks can come very close together, close enough to cause metastability problem. However, in this case the frequency multiple is such that, once the clock edges come close together, there would be sufficient margin in the next cycle to capture data properly without any setup or hold violation.

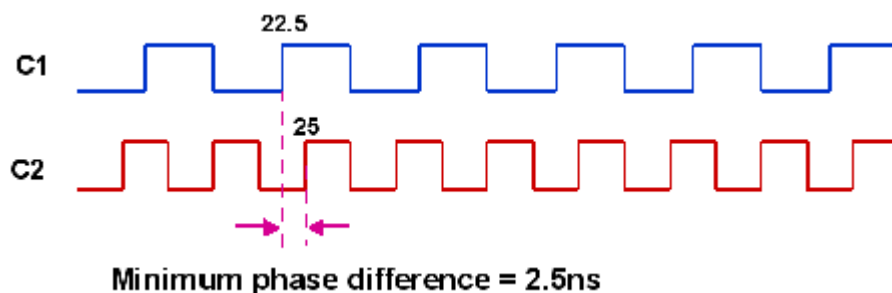
In the third case, the clock edges of the two clocks can be close enough for many consecutive cycles. This is similar to the behavior of asynchronous clocks except that here the clock-root for both the clocks is the same and hence the phase difference between the clocks can be calculated.

Note that in all the examples given below, some delay values are used and it is assumed that a phase margin of less than or equal to 1.5ns between the clock edges can cause metastability. This is just a placeholder value and in real designs, it would be a function of many things including technology used, flop characteristics, etc.

### Example 1

This is the case when the active clock edges of both the clocks will never come very close together, and in all cases there would be a sufficient margin to meet the setup and hold requirements of the circuit.

Consider a clock C from which 2 clocks C1 and C2 are derived with a frequency of divide-by-3 and divide-by-2 respectively with respect to clock C. Here clock C1 is 1.5 times slower than clock C2. As shown in *Figure 12*, the time period of clock C1 is 15ns and of C2 is 10ns. The least possible phase difference between the two clock edges is 2.5ns which should be sufficient to meet setup and hold time requirements.



### 12. Clock edges never come very close together.

However, additional combinational logic should not be added at the crossing due to the very small setup/hold margins. If there is any logic, its delay should meet the setup and hold time requirements. If this condition can be met, there will be no metastability and no synchronizer would be required.

Further, if the crossing is a slow to fast crossing, there will be no data loss. However, in case of a fast to slow clock crossing, there can be data loss. In order to prevent this, the source data needs to be held constant for at least one cycle of the destination clock so that at least one active edge of the destination clock arrives between two consecutive transitions on the source data.

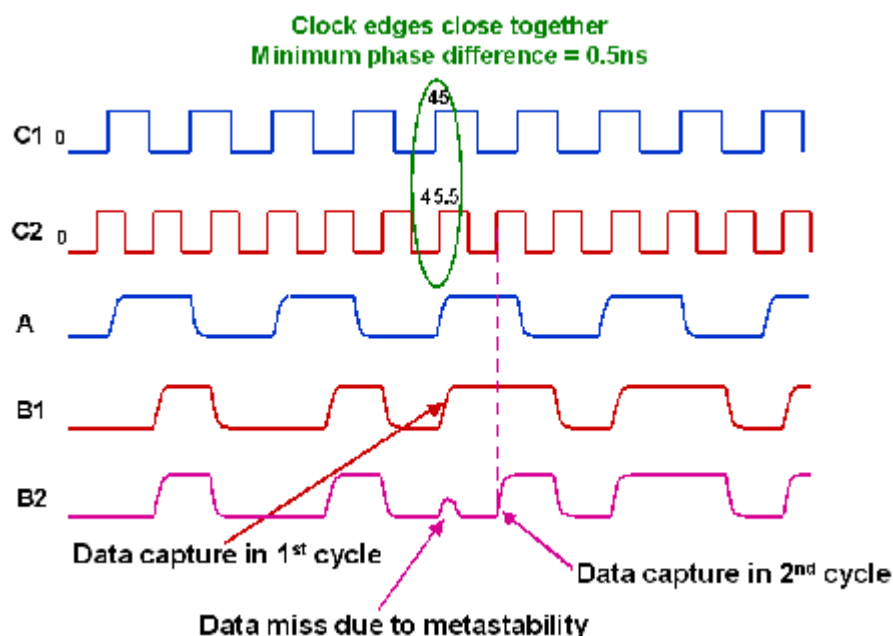
### Example 2

In this case, the active clock edges of both the clocks can come very close together intermittently. In other words, the clock edges come close together once and then there

would be sufficient margin between the edges for the next few cycles (to capture data properly) before they come close again. Here the word “close” implies close enough to cause metastability.

In *Figure 13* , clocks C1 and C2 have time periods 10ns and 7ns respectively. Notice, that the minimum phase difference between the two clocks is 0.5ns, which is very small. So, there are chances of metastability and a synchronizer would be required.

Due to metastability, the data may not be captured in the destination domain when the clock edges are very close together. However, in this case, note that once the clock edges come very close together, in the next cycle there is a sufficient margin so that the data can be captured properly by the destination clock. This is shown by signal B2 in *Figure 13* . While the expected output would be B1, the actual waveform could look like B2, but still there is no data loss in this case. However there can be an issue of data incoherency as described previously.



### 13. Clock edges come close together intermittently.

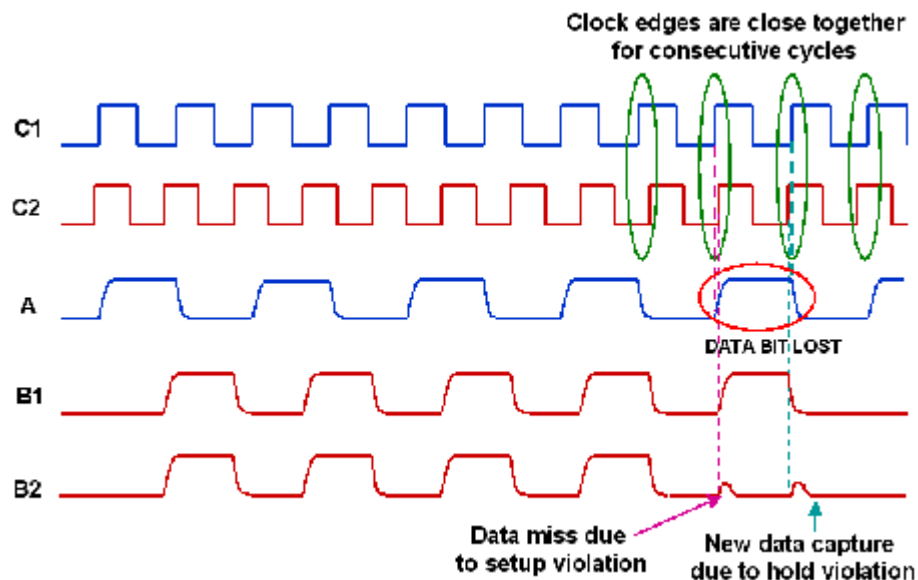
For a fast to slow crossing, data loss can occur, and in order to prevent this, the source data should be held constant for a minimum of one destination clock cycle. Again, this can be done by the use of a simple FSM.

### Example 3

This is the case when the phase difference between the clocks can be very small at times and can remain like that for several cycles. This is very similar to asynchronous clocks except that the variable phase differences will be known and will repeat periodically.

In *Figure 14* , clocks C1 and C2 have time periods 10ns and 9ns respectively. It can be seen that the active clock edges of both the clocks come very close together for 4 consecutive cycles. In the first two cycles there is a possibility of a setup violation (as the

source clock is leading the destination clock) and in the next two cycles there is a possibility of hold violation (as the destination clock is leading the source clock).



14. Clock edges are close for consecutive cycles.

[Click here for a larger version](#)

In this case, there will be an issue of metastability and hence synchronization needs to be done.

Apart from metastability there can be an issue of data loss also, even though it is a slow to fast clock domain crossing. As can be seen from *Figure 14*, B1 is the expected output if there would have been no metastability. However, the actual output can be B2. Here the data value '1' is lost, because in the first cycle the value '1' is not captured due to setup violation and in the second cycle the new value '0' is incorrectly captured due to hold violation.

In order to prevent data loss, the data needs to be held constant for a minimum of two cycles of the destination clock. This is applicable for both fast to slow as well as slow to fast clock domain crossings. This can be done by controlling the source data generation using a simple FSM. However, the data incoherency issue can still be there.

In such cases, standard techniques like handshake and FIFO are more useful to control data transfer as they will also take care of the data incoherency issue.

## Asynchronous Clock Domain Crossings

Clocks which do not have a known phase or frequency relationship between them are known as asynchronous clocks. Whenever there is a clock crossing between two asynchronous clocks, their active edges can arrive very close together leading to metastability. Here the phase difference between the clocks can be variable and unlike synchronous clocks it is unpredictable.

Proper synchronization needs to be done in the destination domain to prevent metastability. Apart from that, there can be problems of data loss and data incoherency (in both fast to slow as well as slow to fast clock crossings). If the source and destination clock frequencies are known, data loss can be prevented by holding the source data constant for two cycles of the destination clock. However, if the circuit is to be designed to be independent of clock frequencies, handshake or FIFO techniques should be used to prevent metastability, data loss and data incoherency.

## **Verification Methodology**

---

This section describes a methodology which will ensure that the circuit has been designed properly to handle the clock domain crossing issues.

The validation activity can be divided into two categories, namely structural and functional. Structural validation ensures that appropriate synchronization logic has been added wherever it is required and functional validation ensures that the logic which has been added is actually performing the intended function.

A number of CDC problems can be detected just by performing structural validation. These checks are simpler and much faster than the functional validation. Moreover, if there are structural issues, most of the functional validation would have no relevance anyway. Hence, verification should begin with the structural checks and the problems detected there should be corrected before moving on to functional validation.

Rule-based checking is a very efficient way to perform structural validation. Assertion-based verification techniques can be used to perform functional validation. Assertions can be inferred automatically in the design using some EDA tools, or they can be inserted in the RTL using any of the standard assertion languages like OVL, PSL and SVA. These languages are supported by many EDA vendors.

These assertions can either be simulated in the functional simulation environment or can be verified using formal verification techniques. Both these techniques have their own advantages and disadvantages.

The simulation results are dependent on the quality of test vectors used. A problem may go undetected if the vectors applied cannot stimulate it, and it is very difficult to determine the right set of test vectors which will give good coverage.

As compared to simulation, formal techniques give a much better coverage and there is no need to provide any test vectors. However, formal techniques have some performance issues because of state space explosion which is a well known problem in formal analysis (see reference 4). So, these checks are not suitable for full chip analysis but they work reasonably well at the block level.

A step-by-step approach for verifying clock domain crossings is described here.

## Step 1

---

Check for the presence of valid synchronizers in:

All asynchronous clock domain crossings, and,

Those cases of synchronous clock domain crossings where there can be metastability as described in the section on rational multiple clocks.

A multi-flop synchronizer is sufficient to ensure that there will be no metastability.

However, there can still be a problem of data incoherency. So, it is advisable to check at this stage only, that multi-flop synchronizers are used only for scalar signals. They can also be used for control busses. They should not be used for data busses however.

A rule-based checker can be used to automatically detect all clock domain crossings and to check for the presence of valid synchronizers at all places where they are required.

If there are missing synchronizers, the designer should modify the design to add appropriate synchronization logic.

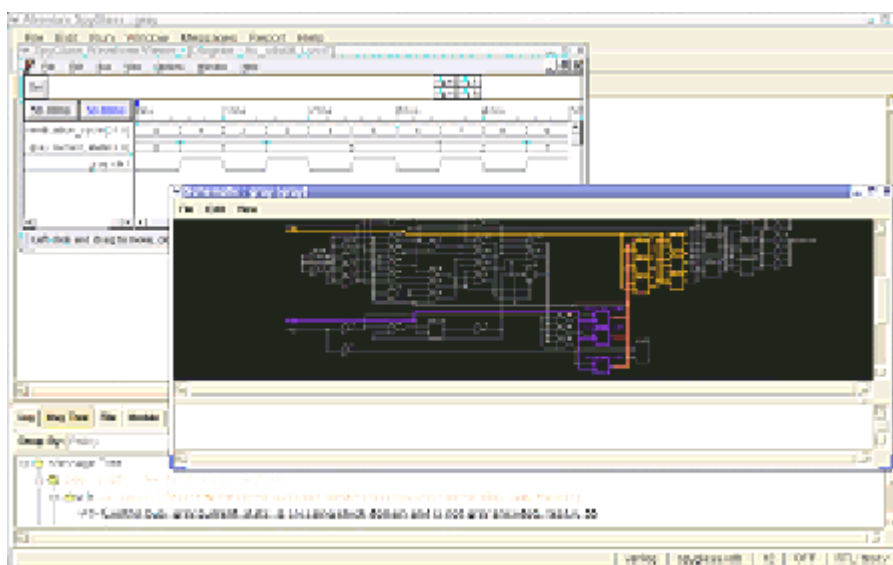
## Step 2

---

Check for the presence of separately synchronized signals which are converging. These are probable candidates for data incoherency. These candidates can be identified by doing structural analysis of the design.

The candidate signals for data incoherency should be verified to be Gray-encoded. This validation can be done through assertions. The assertion itself could even be generated by a structural checking tool " whenever it sees signals which are candidates for data incoherency.

*Figure 15* shows a control bus clock domain crossing, which is synchronized using a multi-flop synchronizer but is not Gray-encoded. A waveform trace is generated for the assertion failure.



15. Formal verification helps catch gray-encoding failure.

In case the converging signals cannot be Gray-encoded, change the synchronization scheme to one which uses a common control signal, for example, MUX recirculation, FIFO or handshake. These schemes still need to be validated for proper functionality as described in Step 4.

### Step 3

Once the proper synchronization logic is in place and the Gray-encoding checks have been done, the next step is to verify that there is no data loss while transferring data from one clock domain to the other. This needs to be checked for the following two cases:

- Synchronous clock domain crossings
  - All fast to slow crossings
  - Slow to fast crossings where the clock edges can be close together for continuous cycles
- All asynchronous clock domain crossings

These can be validated by asserting that each source data launch is always captured in the destination domain.

In the case of fast to slow synchronous clock domain crossings, where a synchronizer is not required and for the simple cases of multi-flop synchronization, check that after every transition on the source data an active edge of the destination clock arrives where there is no setup or hold violation.

For other synchronization schemes, some standard functional checks can be done to ensure that there is no data loss, which are described in Step 4.

### Step 4

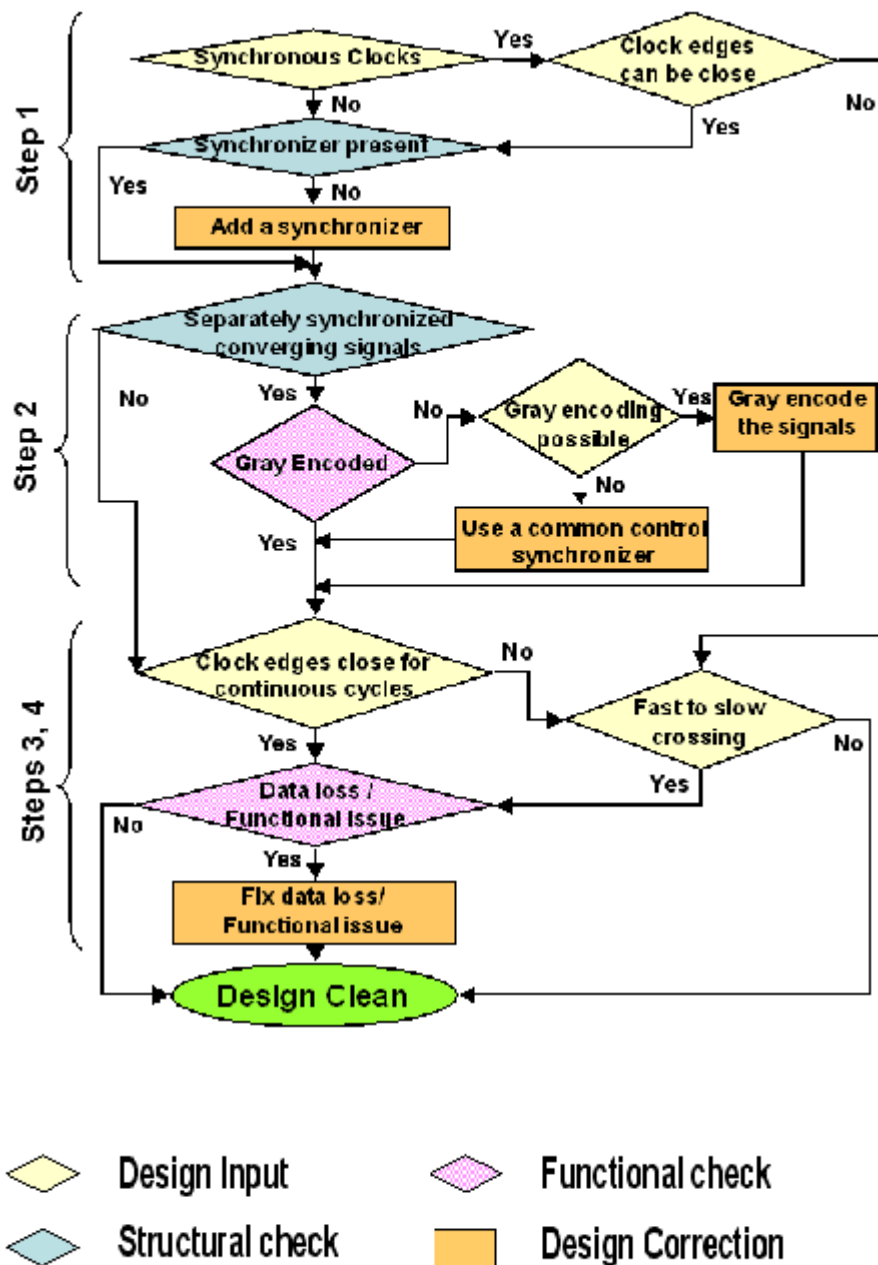
In all cases, where some special synchronization schemes are used, it is necessary to verify that they are performing the intended function correctly. This is important to ensure that there will be no metastability, data incoherency or data loss problem.

The required checks are given here for three commonly used schemes:

- Handshake synchronization: Check that the request-data and request-acknowledge protocols are working as per the specifications.
- FIFO synchronization: Check that there is no FIFO overflow or underflow.
- Mux recirculation: With reference to *Figure 8*, check that while the synchronized control signal EN\_Sync is active, the following two conditions hold:
  - Source data A[0:1] is stable, and,
  - at least one active edge of destination clock arrives

The methodology described in the above four steps is also depicted in *Figure 16*.





16. The flow of the verification methodology.

[Click here for a larger version](#)

### Summary

Traditional verification methods like simulation and static timing analysis are not sufficient to detect all types of problems which can occur in clock domain crossings. The problems which can occur depend on the types of clock domain crossings. Similarly, the solutions to those problems are also different and hence the verification techniques required are different as well. Some of the basic problems of clock domain crossings have been discussed here. The solutions to those issues are also discussed and a verification methodology is proposed which will ensure that data is correctly transferred across clock domains.

### References

- [1] Sanjay Churiwala, "Tackling multiple clocks in SoCs", EE Times March 15, 2004.
- [2] Shaker Sarwary, "Solving the toughest problems in CDC analysis", EE Times August 28, 2006.
- [3] <http://www.asic-world.com/tidbits/metastablity.html>
- [4] K. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, Boston, 1993.

**About the Authors:**

**Saurabh Verma** Is an engineering manager at Atrenta. He has a bachelor degree from Indian Institute of Technology, Kanpur. He can be reached at: Verma

**Ashima S. Dabare** Ashima is a Consulting Applications Engineer at Atrenta. She has a masters degree from Indian Institute of Technology, Delhi. She can be reached at: Dabare.

# Solving the toughest problems in CDC analysis

---

Among the many verification challenges confronting system-on-chip (SOC) designers these days, clock domain crossings (CDCs) rank near the top in difficulty. The latest SOCs may have dozens or even thousands of clock domains, many of them difficult to verify using conventional tools such as simulation. Detecting these bugs using traditional simulation tools requires long simulation runs and a chance encounter. As a consequence, CDCs have become a leading cause of design errors. Such errors can add significant time and expense to the design-and-debug cycle, and may even find their way into silicon, necessitating costly re-spins.

Two particularly troublesome CDC-related issues involve FIFO- and handshake-based synchronization mechanisms. Both can be difficult or impossible to accurately verify using simulation. And conventional static CDC analysis tools do too little and too much at the same time, simultaneously overlooking *real* design errors and over-reporting large numbers of *false* violations. As a result the user is forced into an endless bug-hunting process, which often discourages the designer and leaves the real bugs undetected.

The success of static CDC verification tools is determined by two critical measures—the time taken to signoff and the completeness of CDC verification. Conventional CDC analysis tools fall short in both areas. They generate large amounts of noise (false violations)—extending the verification cycle—and provide poor coverage of complex CDC synchronization schemes.

Fortunately, a new class of CDC tool, using static analysis techniques, offers an effective automated solution to these problems. Such tools combine functional and structural analysis to both identify and verify FIFO and handshake synchronizers, while weeding out the false violations that plague traditional CDC tools.

## Separating True From False Violations

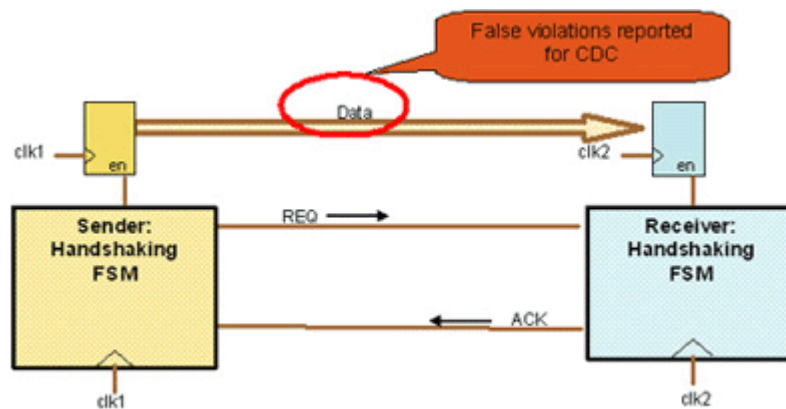
In order to isolate real clock domain crossing issues, it is necessary to detect various synchronization schemes—not just basic two-flop or multi-flop synchronizers, but more complex mechanisms, such as handshakes and FIFO-based schemes. Once detected, these synchronizers need to be verified as working correctly.

Both detection and verification have their own set of challenges, but when they are done properly, one can confidently claim correctness of these clock domain crossings. This knowledge, in turn, can be used to filter out false violations, which typically result when a tool fails to recognize properly synchronized crossings and instead reports them as unsynchronized. Many tools lack the ability to detect, much less functionally verify, handshake structures. Given that it is now commonplace to find up to 80 percent of CDCs controlled by handshakes in large design modules, the false violations can be plentiful,

numbering in the hundreds. They may number in the many hundreds. No wonder designers complain about false violations & handshakes! Similar problems arise with FIFO synchronizers.

## Verifying Handshakes

Handshake synchronizers rely on request-acknowledge protocols (*Fig.1*) .



### 1. Typical handshake structure can generate false CDC violations

A request signal is sent from the source domain when the data is stable. The destination domain waits for the request signal to capture the data, then sends back an acknowledge signal to the source domain. The source domain can now de-assert the request and prepare for the next data transfer.

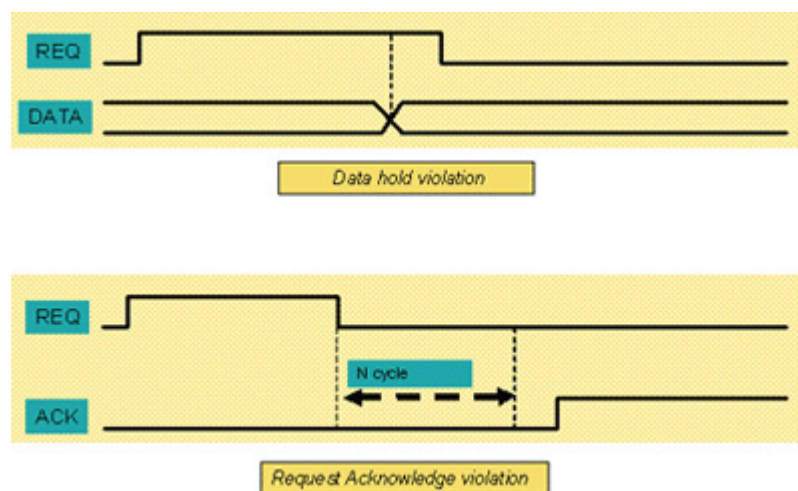
The control, request and acknowledge signals may be synchronized using standard dual-rank synchronizers. However, since the protocol requires the request to be high until an acknowledge signal is received, the designer may determine that the crossing is safe and that no synchronization of REQ-ACK signals is necessary. Traditional CDC tools that look only for a double-flop synchronizer in such cases will report these crossings as unsynchronized, resulting in a large number of false violations. These false violations, in turn, will be embedded in a long list of unsynchronized crossings, and it becomes a tedious process to go through and identify the buses that are properly synchronized using a handshake synchronizer.

Is there a better solution? Yes: automatic identification and verification of handshake schemes can eliminate a large number of false violations. This requires several capabilities:

1. Handshake recognition: In order to correctly recognize a handshake scheme, the data transfer and all components of the handshake need to be identified. Identification includes crossing detection, MUX or flop enable detection, source and destination controller identification and finally request and acknowledge recognition. Due to the variety of handshaking design styles, detection of all the components requires thorough and flexible structural and functional analysis of a design.

2. Handshake verification: The crossing cannot be considered safe until its functionality is proven to be correct. Two aspects of handshake functionality can be verified (*Fig. 2*)

- Request and acknowledge behavior: Each request should result in an acknowledge within some time frame. The time frame can be detected during the handshake recognition and designer may be prompted to verify the correctness before actual verification. In addition, request and acknowledge signals may be active low or high, which can also be analyzed during handshake detection.
- Data capture window check: Data should be stable at the time a request is sent and should not change until acknowledge is asserted and request is taken off. Although this requirement verifies the correctness at the source and is not directly related to a crossing problem, a violation indicates a serious functional problem in a design.



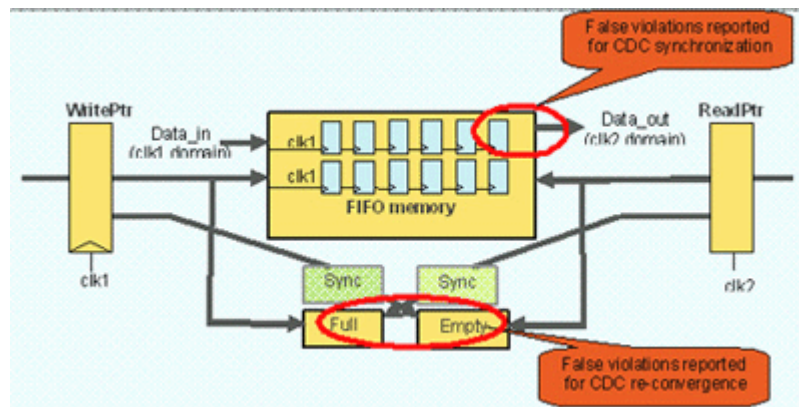
## 2. Functional issues may violate Handshake protocol

Static verification techniques are the ideal way to perform the above verification exhaustively; Alternatively, verification can be carried out using simulation and a user-defined testbench. The difference is that the simulation approach requires a great deal of manual effort and will not exhaustively verify these schemes, leaving many bugs unresolved.

FIFOs are commonly used to transfer data generated by a source to a destination where the source and destination are running at different or variable rates. Very often the source and destination reside in different clock domains, in which case an asynchronous FIFO is needed. In some cases, synchronous FIFOs are also used when the source and the destination are in the same clock domain.

Asynchronous FIFOs involve multiple clock domain crossings for empty and full flag calculation as well as data read to the destination domain. In a FIFO, these crossings are not always synchronized using traditional synchronization methods, and may be falsely reported as unsynchronized. The following two false violations are commonly reported by structural CDC analysis (*Fig. 3*) :

1. Unsynchronized crossing at the output: The data is directly read from the FIFO (in the write domain) into a destination register (in the read domain). This creates an unsynchronized crossing but should be considered a correct crossing because of the FIFO.
2. CDC re-convergence: The empty and full flags involve synchronized read/write pointers that normally converge into comparators for empty and full flag calculation. Although there is convergence of separately synchronized signals, these are not a design issue and can be safely waived.



### 3. False CDC violations complicate Asynchronous FIFO debugging

Conventional tools don't recognize FIFO synchronizers, so designers have to comb through unsynchronized crossing reports and figure out which ones are part of FIFO synchronizers. As with handshakes, manual filtering of these false violations is tedious and error-prone. On the other hand, not performing full FIFO verification is highly risky. Many bugs may be hidden in the full and empty flag logic, causing rare FIFO overflows or underflows that can easily go undetected by simulators and cause silicon re-spins.

CDC verification of FIFO synchronizers can be addressed with the following steps:

1. FIFO detection: FIFOs can be detected automatically by looking into logic around register banks and/or buses crossing clock domains. There are challenges in performing this analysis due to the variety of FIFOs and design styles. Thorough structural and functional analysis is needed to examine all components of a FIFO, including the memory block, register files, pointer counters, gray encoder, comparators and other glue logic.
2. FIFO verification: Once recognized, FIFOs may be efficiently verified for overflow and underflow conditions. Static functional verification can be used to cover the functionality exhaustively. This is a good application of static functional verification due to the local nature of full and empty logic and the FIFO behavior contained in a particular area of the design. An intelligent verification approach can perform such verification using only the read and write pointers, sparing the designer the effort of specifying FIFO attributes such as FIFO depth.

## Conclusion

FIFO- and handshake-based synchronization of clock domain crossings pose particular problems for design verification. These design features have resisted efficient detection and verification by conventional automated approaches, such as simulation and traditional CDC-checking tools. However, a new class of tools is emerging that, using combined structural and functional analysis, offers the first effective automated solution to these problems. By automatically detecting and verifying FIFO- and handshake-based CDC problems, such tools close a major gap in SOC verification. The result will be a big improvement in productivity and equally big reduction of risk in SOC development. The benefits are hard to overstate: by weeding out serious bugs early in the design cycle—prior to implementation and fabrication—these new-generation CDC tools can potentially save weeks of design-and-debug and hundreds of thousands or even millions of dollars in redesigns and respins.

## About the author

**Shaker Sarwary** is a technology director at Atrenta. He has a doctorate from Paris University (France). He has held senior engineering positions in the areas of Synthesis and Verification at Lattice Semiconductor, Get2Chip, and Cadence. He can be reached at .