World Class Verilog & SystemVerilog Training

Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog

Clifford E. Cummings
Sunburst Design, Inc.
cliffc@sunburst-design.com

ABSTRACT

Important design considerations require that multi-clock designs be carefully constructed at Clock Domain Crossing (CDC) boundaries. This paper details some of the latest strategies and best known methods to address passing of one and multiple signals across a CDC boundary. Included in the paper are techniques related to CDC verification and an interesting 2-deep FIFO design for passing multiple control signals between clock domains. Although the design methods described in the paper can be generally implemented using any HDL, the examples are shown using efficient SystemVerilog techniques.

Table of Contents

1.0 Introduction 6
2.0 Metastability 6
2.1 Why is metastability a problem? 7
3.0 Synchronizers 8
3.1 Two synchronization scenarios 8
3.2 Two flip-flop synchronizer 8
3.3 MTBF - mean time before failure 9
3.4 Three flip-flop synchronizer 10
3.5 Synchronizing signals from the sending clock domain 10
3.6 Synchronizing signals into the receiving clock domain 11
4.0 Synchronizing fast signals into slow clock domains 13
4.1 Requirement for reliable signal passing between clock domains 13
4.1.1 The "three edge" requirement 13
4.2 Problem - passing a fast CDC pulse 14
4.3 Problem - sampling a long CDC pulse - but not long enough! 15
4.4 Open-loop solution - sampling signals with synchronizers 16
4.5 Closed loop solution - sampling signals with synchronizers 17
5.0 Passing multiple signals between clock domains 18
5.1 Multi-bit CDC strategies 18
5.2 Multi-bit signal consolidation 18
5.3 Problem - Two simultaneously required control signals 19
5.3.1 Solution - Consolidation 20
5.4 Problem - Two phase-shifted sequencing control signals. 21
5.4.1 Solution - consolidation and an extra flip-flop 22
5.5 Problem - Multiple CDC signals 23
5.5.1 Solutions for passing multiple CDC signals 23
5.6 Multi-Cycle Path (MCP) formulation 24
5.6.1 MCP formulation using a synchronized enable pulse 25
5.6.2 Closed-loop - MCP formulation with feedback 27
5.6.3 Closed-loop - MCP formulation with acknowledge feedback 28
5.7 Synchronizing counters 29
5.7.1 Binary counters 29
5.7.2 Gray codes 30
5.7.3 Gray-to-binary conversion 30
5.7.4 Binary-to-gray conversion 31
5.7.5 Gray code counter style #1 32
5.7.6 Gray code counter style #2 33
5.8 Additional multi-bit CDC techniques 34
5.8.1 Multi-bit CDC signal passing using asynchronous FIFOS 34
5.8.2 Multi-bit CDC signal passing using 1-deep / 2-register FIFO synchronizer 35
6.0 Naming conventions & design partitioning 36
6.1 Clock & signal naming conventions 36

6.1.1 Multi-clock / multi-source modules with no naming convention 37
6.2 Timing verification for each clock domain 37
6.3 Clock oriented design partitioning 37
6.3.1 Timing analysis of clock-partitioned modules 39
6.4 Partitioning with MCP formulations 40
7.0 Multi-clock gate-level simulation issues 41
7.1 Synchronizer gate-level CDC simulation issue 41
7.2 Strategies to remove X-propagation from gate-level simulations 41
7.2.1 Simulator command to turn off timing checks 42
7.2.2 Change flip-flop setup and hold times to 0 42
7.2.3 Copy and modify new flip-flop models 42
7.2.4 Synopsys set_annotated_check command 42
7.3 Additional strategies to remove X-propagation 43
7.3.1 Use multiple SDF files 43
7.3.2 Vendor synchronizer cell with supporting SDF generation tools 43
7.3.3 Vendors with built-in synchronizer support 44
7.4 Multiple SDF files for gate-level CDC simulations 44
7.5 Force synchronizer notifier inputs to a fixed value 44
7.6 ASIC & FPGA library cell synchronizers 45
7.7 Simulation model with random delay insertion 46
8.0 Summary & conclusions 47
8.1 Recommended 1-bit CDC techniques 47
8.2 Recommended multi-bit CDC techniques 48
8.3 Recommended naming conventions and design partitioning 48
8.4 Recommended solutions to multi-clock gate-level CDC simulations 48
9.0 Acknowledgements 48
10.0 References 48
11.0 Author & Contact Information 49
12.0 Appendix 50
12.1 Common sync2 model - used by MCP formulation and FIFO synchronizer 50
12.2 MCP formulation with ready-acknowledge source code 50
12.3 Multi-bit 1-deep / 2-register FIFO synchronizer source code 55

Table of Figures

Figure 1 - Asynchronous clocks and synchronization failure 6
Figure 2 - Metastable `bdat1` output propagating invalid data throughout the design..... 7
Figure 3 - Two flip-flop synchronizer..... 9
Figure 4 - Primary contributing factors to short MTBF values..... 10
Figure 5 - Three flip-flop synchronizer used in higher speed designs 10
Figure 6 - Unregistered signals sent across a CDC boundary..... 11
Figure 7 - Registered signals sent across a CDC boundary 12
Figure 8 - Short CDC signal pulse missed during synchronization 14
Figure 9 - Marginal CDC pulse that violates the destination setup and hold times..... 15
Figure 10 - Lengthened pulse to guarantee that the control signal will be sampled 16
Figure 11 - Signal with feedback to acknowledge receipt 17
Figure 12 - Problem - Passing multiple control signals between clock domains..... 19
Figure 13 - Solution - Consolidating control signals before passing between clock domains 20
Figure 14 - Problem - Passing sequential control signals between clock domains..... 21
Figure 15 - Solution - Logic to generate proper sequencing signals in the new clock domains... 22
Figure 16 - Problem - Encoded control signals passed between clock domains 23
Figure 17 - Logic to pass a synchronized enable pulse between clock domains 24
Figure 18 - Synchronized pulse generation logic..... 25
Figure 19 - Synchronized enable pulse generation logic and equivalent symbol 26
Figure 20 - Multi-Cycle Path (MCP) formulation toggle-pulse generation..... 26
Figure 21 - Multi-Cycle Path (MCP) formulation toggle-pulse generation with acknowledge... 27
Figure 22 - Multi-Cycle Path (MCP) formulation toggle-pulse generation with ready-ack 28
Figure 23 - Binary count values sampled in mid-transition..... 29
Figure 24 - 4-bit gray-to-binary conversion equations..... 30
Figure 25 - 4-bit gray-to-binary conversion equations - 2nd method 31
Figure 26 - 4-bit binary-to-gray conversion equations..... 31
Figure 27 - Gray code counter style #1 - only one gray code register..... 32
Figure 28 - Gray code counter style #2 - binary register and gray code register..... 33
Figure 29 - 1-deep / 2-register FIFO synchronizer block diagram..... 35
Figure 30 - Design partitioned on clock boundaries 38
Figure 31 - Partitioned design with MCP formulation 40
Figure 32 - Synchronizer gate-level CDC simulation waveforms 41
Figure 33 - Sample ASIC & FPGA synchronizer cell for synthesis and simulation 46

Table of Examples

Example 1 - Non-working but conceptually correct gray-to-binary SystemVerilog model 30
Example 2 - Parameterized and correct gray-to-binary SystemVerilog model..... 31
Example 3 - Parameterized binary-to-gray SystemVerilog model..... 32
Example 5 - Parameterized gray-code counter SystemVerilog model..... 33
Example 6 - Parameterized gray-code counter with binary counter 34
Example 7 - SystemVerilog model for ASIC & FPGA synchronizer cell..... 47
Example 8 - sync2.sv code..... 50
Example 9 - plsgen.sv code 50
Example 10 - asend_fsm.sv code..... 51
Example 11 - back_fsm.sv code 51
Example 12 - bmcp_recv.sv code 52
Example 13 - mcp_blk.sv code..... 53
Example 14 - acmp_send.sv code..... 54
Example 15 - wctl.sv code..... 55
Example 16 - cdc_syncfifo.sv code 55
Example 17 - Dual Port Ram code - dp_ram2.sv 56
Example 18 - rctl.sv code..... 56

1.0 Introduction

In 2001, I presented my first paper on multi-asynchronous clock design. At that time, I had not found any good sources to describe the design and synthesis techniques required to do proper multi-clock design. The 2001 paper was a collection of techniques that I had gathered over years from actual ASIC and FPGA design experiences. At the conclusion of the 2001 conference presentation, dozens of engineers and colleagues came forward and shared with me enough additional interesting ideas and techniques to write a sequel on the topic. Over the past eight years, I have included instruction on multi-clock design techniques in my Advanced and Expert Verilog and SystemVerilog training courses, and over that same period of time, more colleagues and students have shared with me additional interesting multi-clock design techniques. Since the release of the first multi-clock paper in 2001, the industry has largely identified these types of design methodologies as Clock Domain Crossing (CDC) techniques. I will use this common nomenclature in this paper.

This paper includes the best techniques described in the 2001 paper along with an updated collection of interesting and efficient multi-clock design techniques that have been shared with me over the past decade. The actual conference presentation slides will be mostly a collection of the new techniques incorporated since the original 2001 presentation, retaining only enough of the original slides to introduce the fundamental CDC design concepts and issues.

2.0 Metastability

Metastbility refers to signals that do not assume stable 0 or 1 states for some duration of time at some point during normal operation of a design. In a multi-clock design, metastability cannot be avoided but the detrimental effects of metastability can be neutralized.

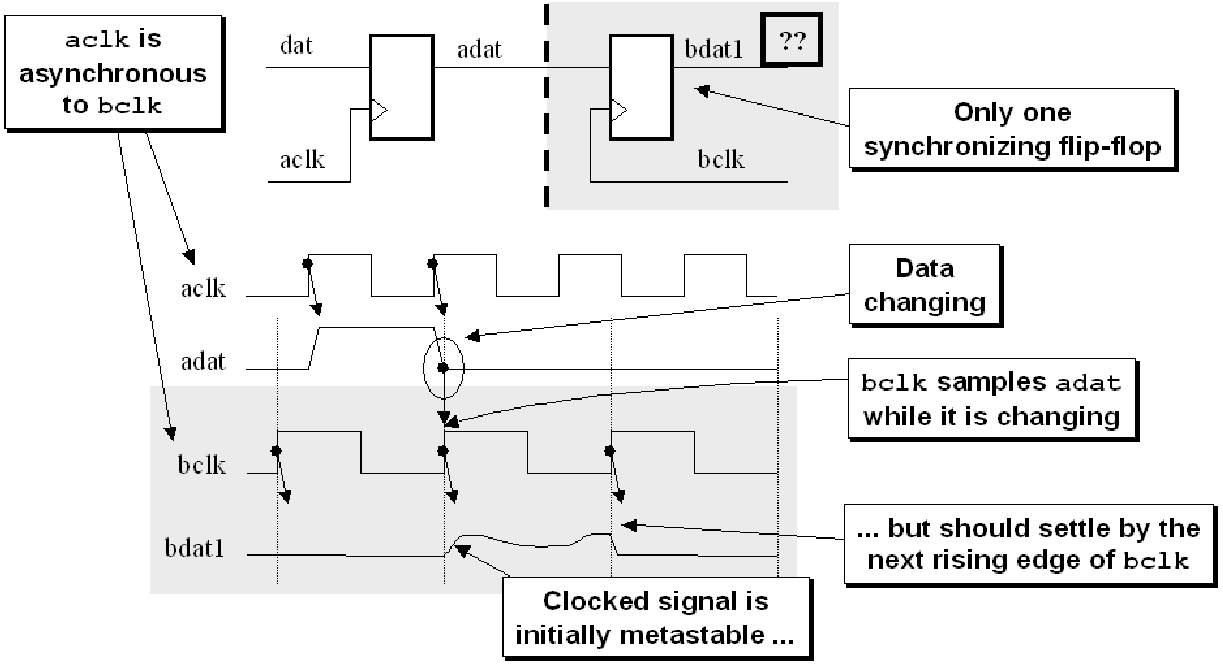

Figure 1 - Asynchronous clocks and synchronization failure

Quoting from Dally and Poulton's book[9] concerning metastability:

> "When sampling a changing data signal with a clock ... the order of the events determines the outcome. The smaller the time difference between the events, the longer it takes to determine which came first. When two events occur very close together, the decision process can take longer than the time allotted, and a synchronization failure occurs."

Figure 1 shows a synchronization failure that occurs when a signal generated in one clock domain is sampled too close to the rising edge of a clock signal from a second clock domain. Synchronization failure is caused by an output going metastable and not converging to a legal stable state by the time the output must be sampled again.

2.1 Why is metastability a problem?

So why is metastability a problem? Figure 2 shows that a metastable output that traverses additional logic in the receiving clock domain can cause illegal signal values to be propagated throughout the rest of the design. Since the CDC signal can fluctuate for some period of time, the input logic in the receiving clock domain might recognize the logic level of the fluctuating signal to be different values and hence propagate erroneous signals into the receiving clock domain.

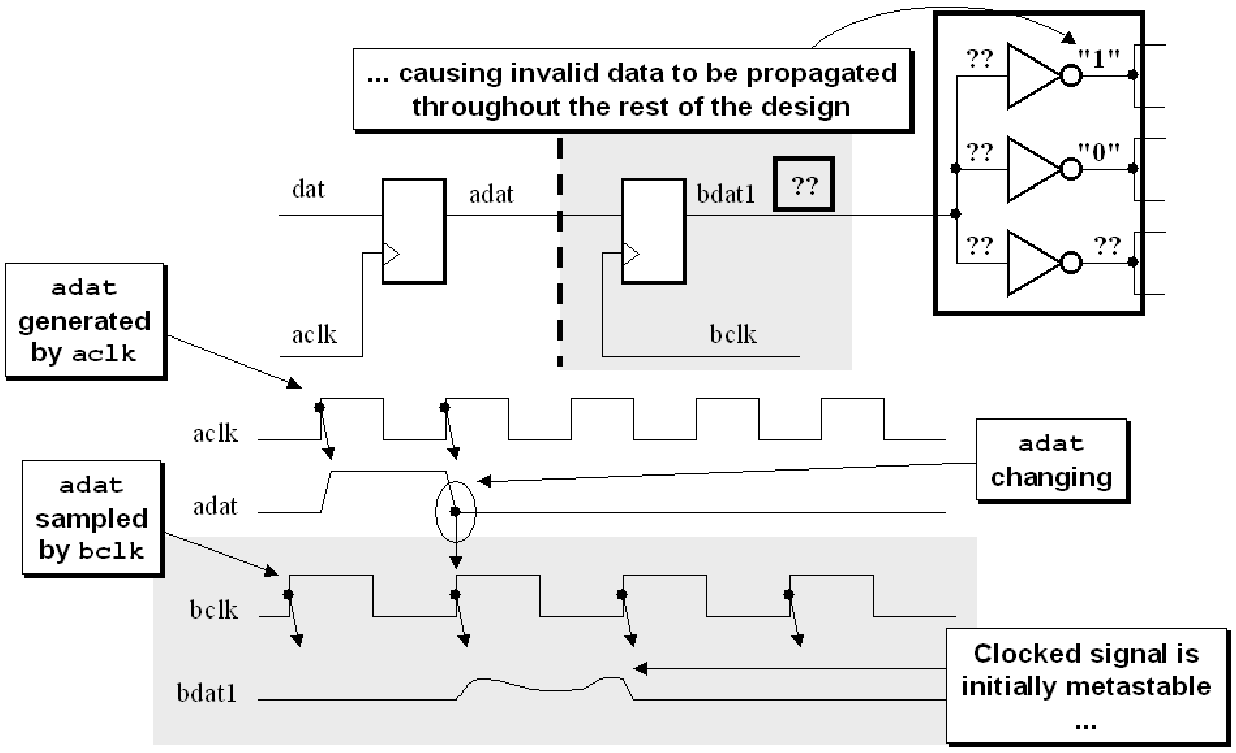

Figure 2 - Metastable `bdat1` output propagating invalid data throughout the design

Every flip-flop that is used in any design has a specified setup and hold time, or the time in which the data input is not legally permitted to change before and after a rising clock edge. This time

window is specified as a design parameter precisely to keep a data signal from changing too close to another synchronizing signal that could cause the output to go metastable.

3.0 Synchronizers

When passing signals between clock domains, an important question to ask is, do I need to sample every value of a signal that is passed from one clock domain to another?

3.1 Two synchronization scenarios

There are two scenarios that are possible when passing signals across CDC boundaries, and it is important to determine which scenario applies to your design:
(1) It is permitted to miss samples that are passed between clock domains.
(2) Every signal passed between clock domains must be sampled.

First scenario: sometimes it is not necessary to sample every value, but it is important that the sampled values are accurate. One example is the set of gray code counters used in a standard asynchronous FIFO design. In a properly designed asynchronous FIFO model, synchronized gray code counters do not need to capture every legal value from the opposite clock domain, but it is critical that sampled values be accurate to recognize when full and empty conditions have occurred.

Second scenario: a CDC signal must be properly recognized or recognized and acknowledged before a change is permitted on the CDC signal.

In both of these scenarios, the CDC signals will require some form of synchronization into the receiving clock domain.

3.2 Two flip-flop synchronizer

Quoting again from Dally and Poulton[9] concerning synchronizers:

> "A synchronizer is a device that samples an asynchronous signal and outputs a version of the signal that has transitions synchronized to a local or sample clock."

The simplest and most common synchronizer used by digital designers is a two-flip-flop synchronizer as shown in Figure 3.

The first flip-flop samples the asynchronous input signal into the new clock domain and waits for a full clock cycle to permit any metastability on the stage-1 output signal to decay, then the stage-1 signal is sampled by the same clock into a second stage flip-flop, with the intended goal that the stage-2 signal is now a stable and valid signal synchronized and ready for distribution within the new clock domain.

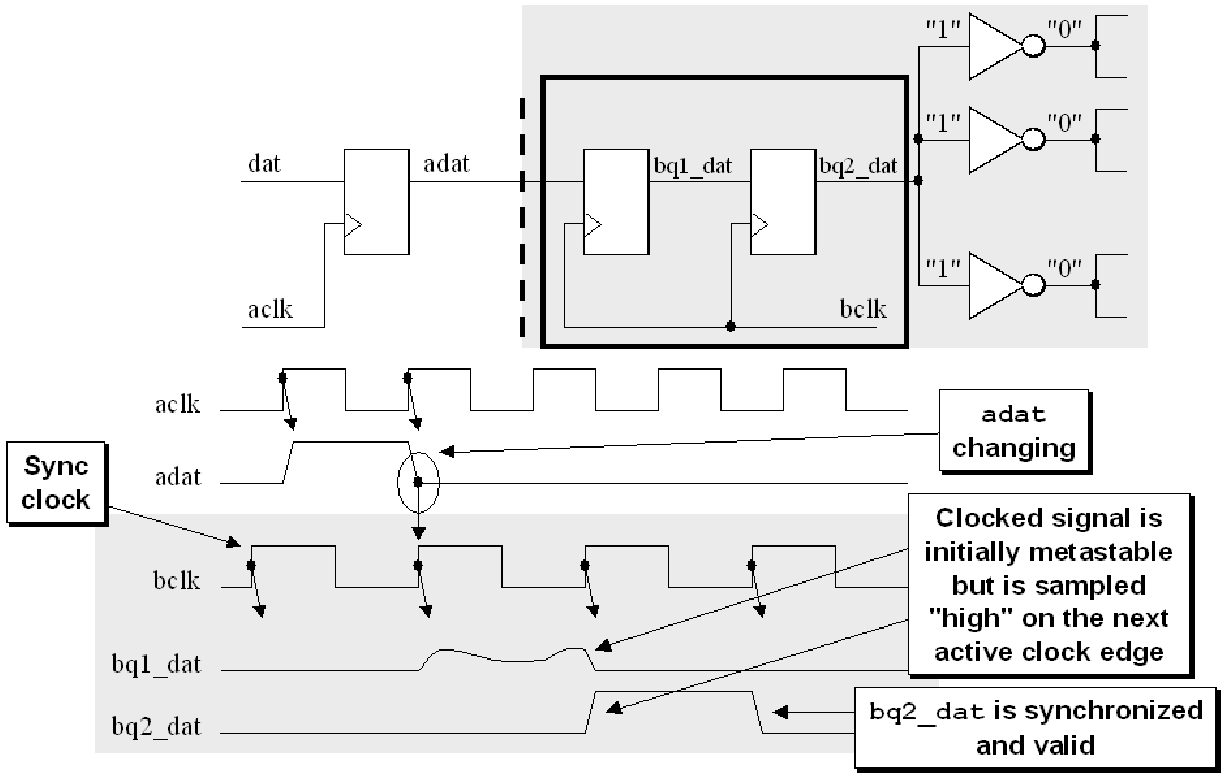

Figure 3 - Two flip-flop synchronizer

It is theoretically possible for the stage-1 signal to still be sufficiently metastable by the time the signal is clocked into the second stage to cause the stage-2 output signal to also go metastable. The calculation of the probability of the time between synchronization failures (MTBF) is a function of multiple variables including the clock frequencies used to generate the input signal and to clock the synchronizing flip-flops. One description of the MTBF calculation can be found in Dally and Poulton[9].

For most synchronization applications, the two flip-flop synchronizer is sufficient to remove all likely metastability.

3.3 MTBF - mean time before failure

For most applications, it is important to run a calculation of the Mean Time Before Failure (MTBF) for any signal crossing a CDC boundary. Failure in this sense means a signal that is passed to a synchronizing flip-flop, goes metastable on the first stage synchronizer flip-flop, and continues to be metastable one cycle later when it is sampled into the second stage synchronizer flip-flop. Since the signal did not settle to a known value after one clock cycle, the signal could still be metastable when sampled and passed to the receiving clock domain, causing potential failures to the corresponding logic.

When calculating MTBF numbers, larger numbers are preferred over smaller numbers. Larger MTBF numbers indicate longer periods of time between potential failures, while smaller MTBF

numbers indicate that metastability could happen frequently, similarly causing failures within the design.

Dally and Poulton[9] give a good equation with very thorough analysis of the calculation that can be performed to calculate the MTBF of a synchronizer circuit. Without repeating the equation and analysis, it should be pointed out that two of the most important factors that directly impact the MTBF of a synchronizer circuit are, the sample clock frequency (how fast are signals being sampled into the receiving clock domain) and the data change frequency (how fast is the data changing that crosses the CDC boundary).

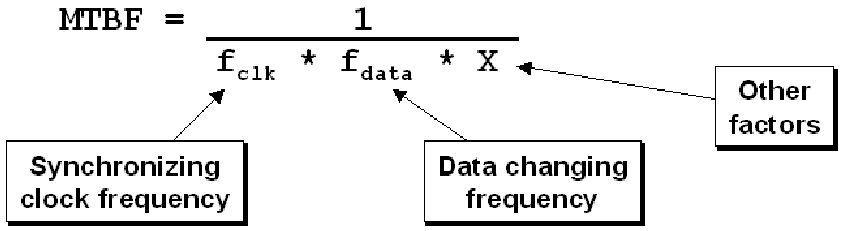

Figure 4 - Primary contributing factors to short MTBF values

From the above partial equation, it can be seen that failures occur more frequently (shorter MTBF) in higher speed designs, or when the sampled data changes more frequently.

3.4 Three flip-flop synchronizer

For some very high speed designs, the MTBF of a two-flop synchronizer is too short and a third flop is added to increase the MTBF to a satisfactory duration of time. Of course, satisfactory is determined by the architect of the design.

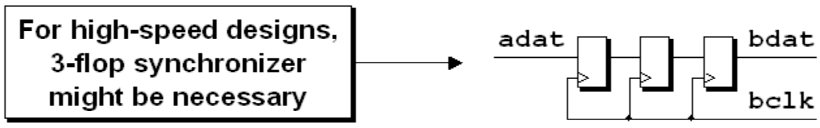

Figure 5 - Three flip-flop synchronizer used in higher speed designs

3.5 Synchronizing signals from the sending clock domain

Frequently asked question regarding CDC design: Is it a good idea to register signals from the sending clock domain before passing the signals to the receiving clock domain? Implied in the question is the assumption that CDC signals will be synchronized into the receiving clock domain; therefore, they do not require synchronization in the sending clock domain. This rationalization is incorrect and registering signals in the sending clock domain should generally be required.

Consider an example where the signals in the sending clock domain are not registered before being passed into the receiving clock domain, as shown in Figure 6.

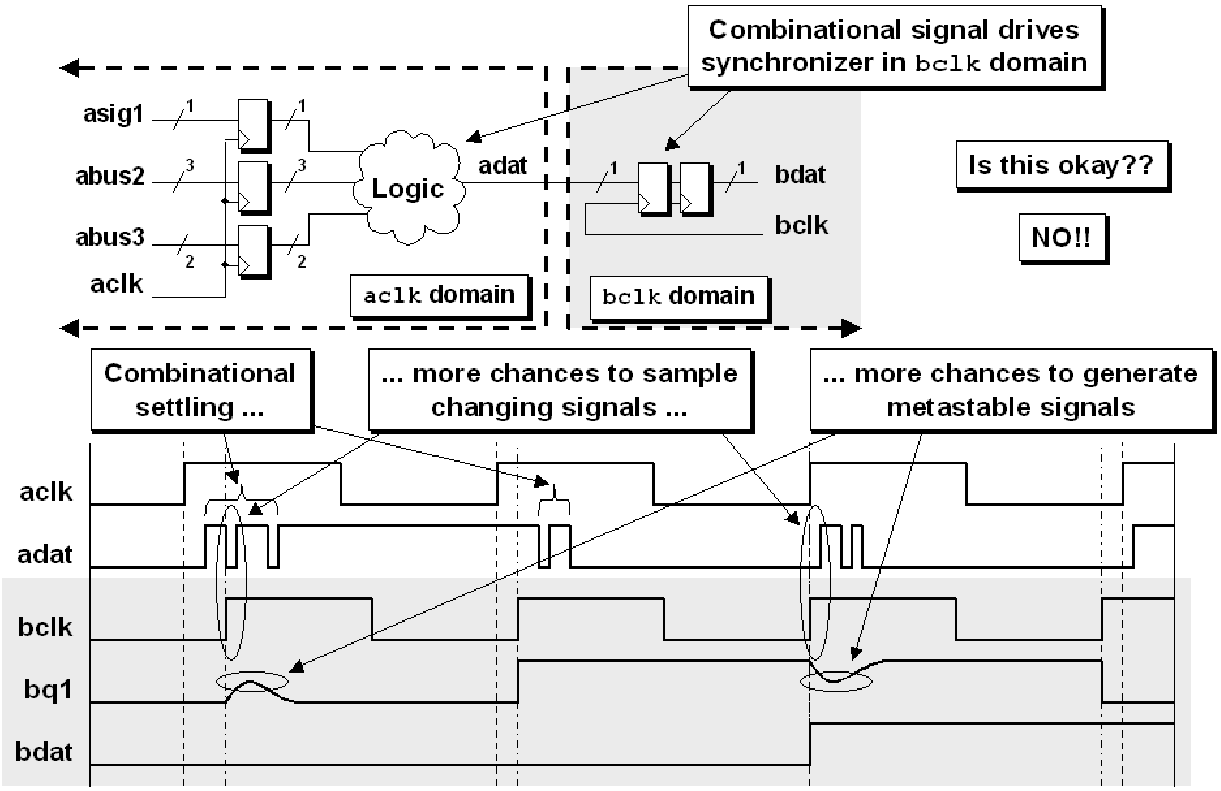

Figure 6 - Unregistered signals sent across a CDC boundary

In this example, the combinational output from the sending clock domain could experience combinational settling at the CDC boundary. This combinational settling effectively increases the data-change frequency potentially creating small bursts of oscillating data and thereby increasing the number of edges that could be sampled while changing, with a corresponding increase in the potential for sampling changing data and generating metastable signals.

3.6 Synchronizing signals into the receiving clock domain

Signals in the sending clock domain should be synchronized before being passed to a CDC boundary. The synchronization of signals from the sending clock domain reduces the number of edges that can be sampled in the receiving clock domain, effectively reducing the data-change frequency in the MTBF equation and hence increasing the time between calculated failures (see section 3.3 for a description of the impact of data change frequencies on MTBF).

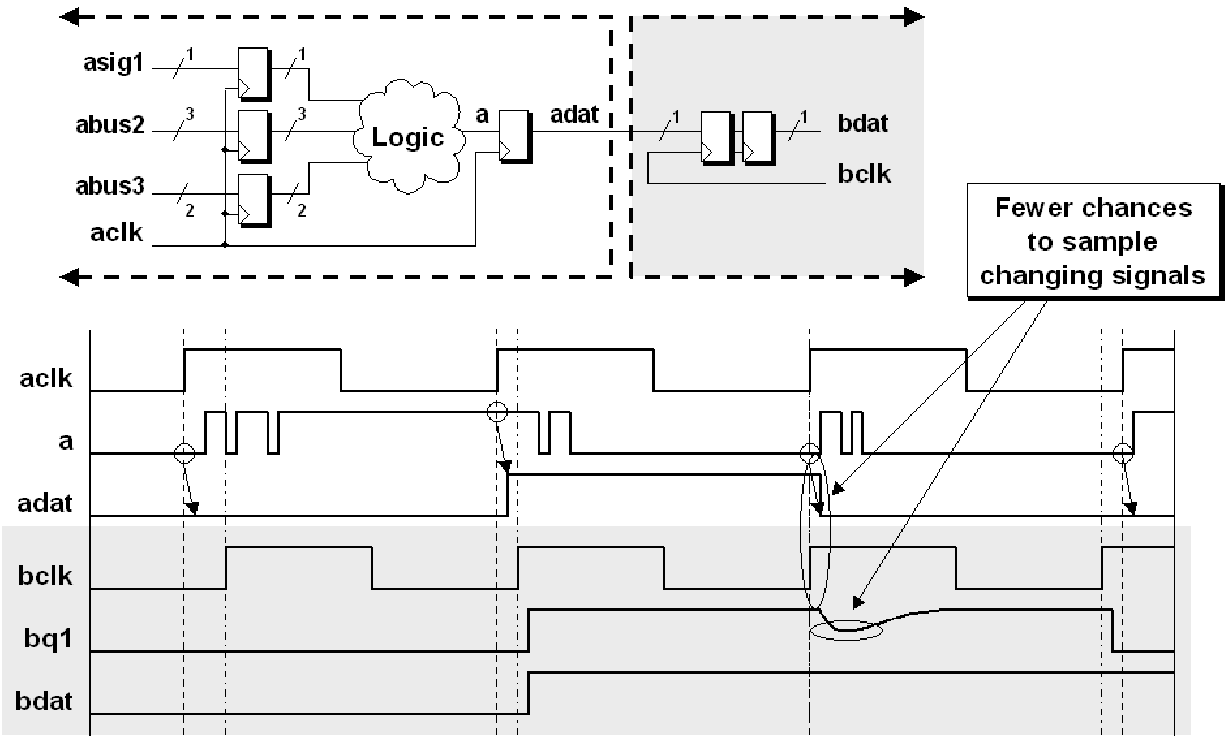

Figure 7 - Registered signals sent across a CDC boundary

In Figure 7, the **`aclk`** logic settles and sets up on the **`adat`** flip-flop before being passed into the **`bclk`** domain. The **`adat`** flip-flop filters out the combinational settling on the flip-flop input (**`a`**) and passes a clean signal to the **`bclk`** logic.

4.0 Synchronizing fast signals into slow clock domains

As discussed in section 3.1, if a CDC signal cannot be skipped when passed between clock domains, it is important to consider signal widths or synchronization techniques when they are passed between clock domains.

One issue associated with synchronizers is the possibility that a signal from a sending clock domain might change values twice before it can be sampled, or might be too close to the sampling edges of a slower clock domain. This possibility must be considered any time signals are sent from one clock domain to another and a determination must be made whether missed signals are or are not a problem for the design in question.

When missed samples are not allowed, there are two general approaches to the problem:
(1) An open-loop solution to ensure that signals are captured without acknowledgment.
(2) A closed-loop solution that requires acknowledgement of receipt of the signal that crosses a CDC boundary.
Both solutions are discussed in this section.

4.1 Requirement for reliable signal passing between clock domains

Synchronizing slower control signals into a faster clock domain is generally not a problem if the faster clock domain is 1.5X the frequency (or more) of the slower clock domain, since the faster clock signal will sample the slower CDC signal one or more times. Recognizing that sampling slower signals into faster clock domains causes fewer potential problems than sampling faster signals into slower clock domains, a designer might to take advantage of this fact by using simple two flip-flop synchronizers to pass single CDC signals between clock domains.

4.1.1 The "three edge" requirement

Mark Litterick[4] noted that when passing one CDC signal between clock domains through a two-flip-flop synchronizer, the CDC signal must be wider than 1-1/2 times the cycle width of the receiving domain clock period. Littereick described this requirement as "input data values must be stable for three destination clock edges."

For exceptionally long source and destination clock frequencies, this requirement could probably be safely relaxed to 1-1/4 times the cycle time of the receiving clock domain or less, but the "three edge" guideline is the safest initial design condition, and is easier to prove through the use of SystemVerilog assertions than to dynamically measure a fractional width of a CDC signal during simulation.

The "three edge" requirement actually applies to both open-loop and closed-loop solutions, but implementations of the closed-loop solution automatically ensure that at least three edges are detected for all CDC signals.

4.2 Problem - passing a fast CDC pulse

Consider the severely flawed condition where the sending clock domain has a higher frequency than the receiving clock domain and that a CDC pulse is only one cycle wide in the sending clock domain. If the CDC signal is only pulsed for one fast-clock cycle, the CDC signal could go high and low between the rising edges of a slower clock and not be captured into the slower clock domain as shown in Figure 8.

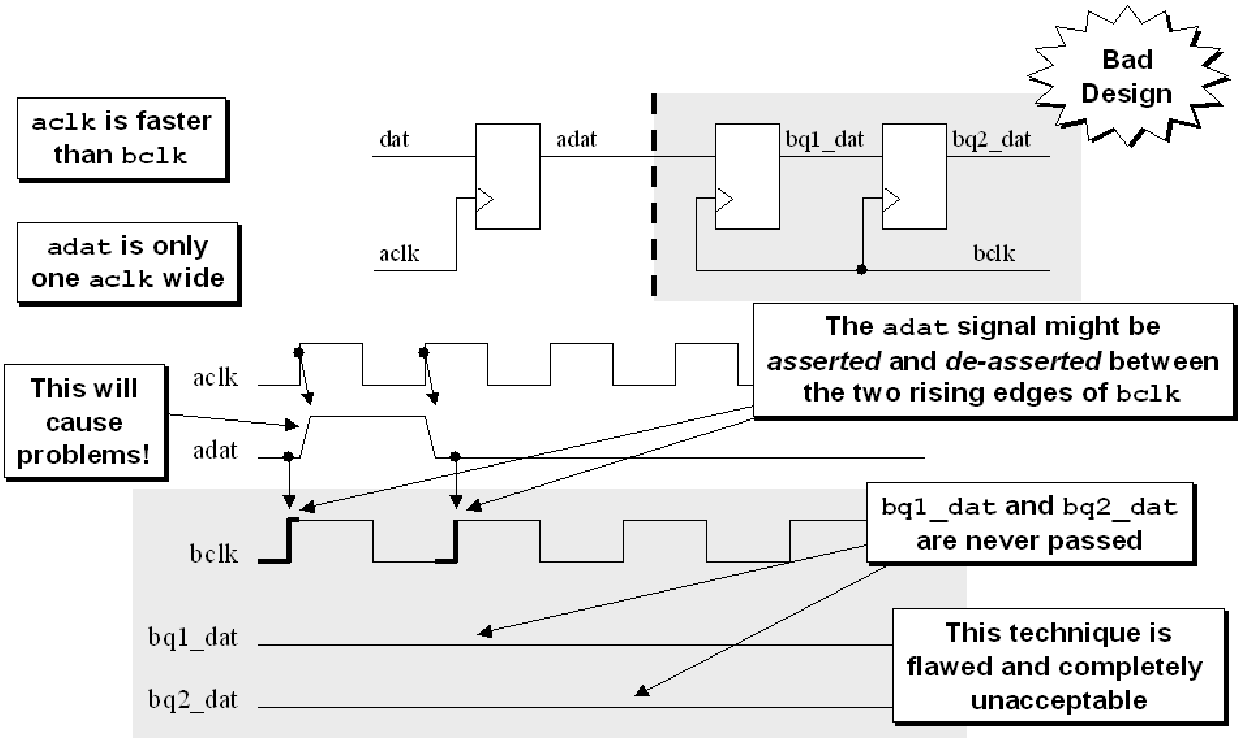

Figure 8 - Short CDC signal pulse missed during synchronization

4.3 Problem - sampling a long CDC pulse - but not long enough!

Consider the somewhat non-intuitive and flawed condition where the sending clock domain sends a pulse to the receiving clock domain that is slightly wider than the period of the receiving clock frequency. Under most conditions, the signal will be sampled and passed, but there is the small but real chance that the CDC pulse will change too close to the two rising clock edges of the receiving clock domain and thereby violate the setup time on the first clock edge and violate the hold time of the second clock edge and not form the anticipated pulse. This possible failure is shown in Figure 9.

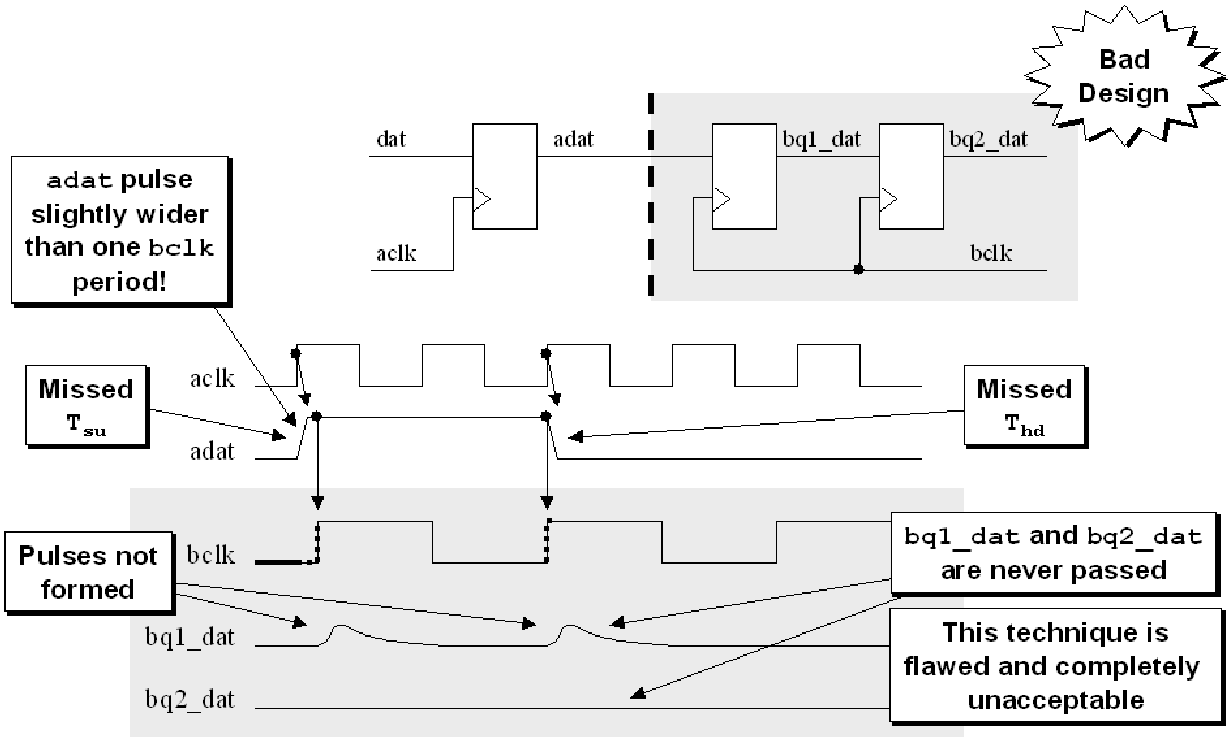

Figure 9 - Marginal CDC pulse that violates the destination setup and hold times

4.4 Open-loop solution - sampling signals with synchronizers

One potential solution to this problem is to assert CDC signals for a period of time that exceeds the cycle time of the sampling clock as shown in Figure 10. As discussed in section 4.1.1, the minimum pulse width is 1.5X the period of the receiving clock frequency. The assumption is that the CDC signal will be sampled at least once and possibly twice by the receiver clock.

Open-loop sampling can be used when relative clock frequencies are fixed and properly analyzed.

Advantage: the Open-loop solution is the fastest way to pass signals across CDC boundaries that does not require acknowledgement of the received signal.

Disadvantage: the largest potential problem related to an open-loop solution is that another engineer might mistake the solution for a general purpose solution, or the design requirements might change and an engineer might fail to re-analyze the original open loop solution. This problem can be minimized by adding a SystemVerilog Assertion to the model to detect if the input pulse ever fails to exceed the "three edges" design requirement.

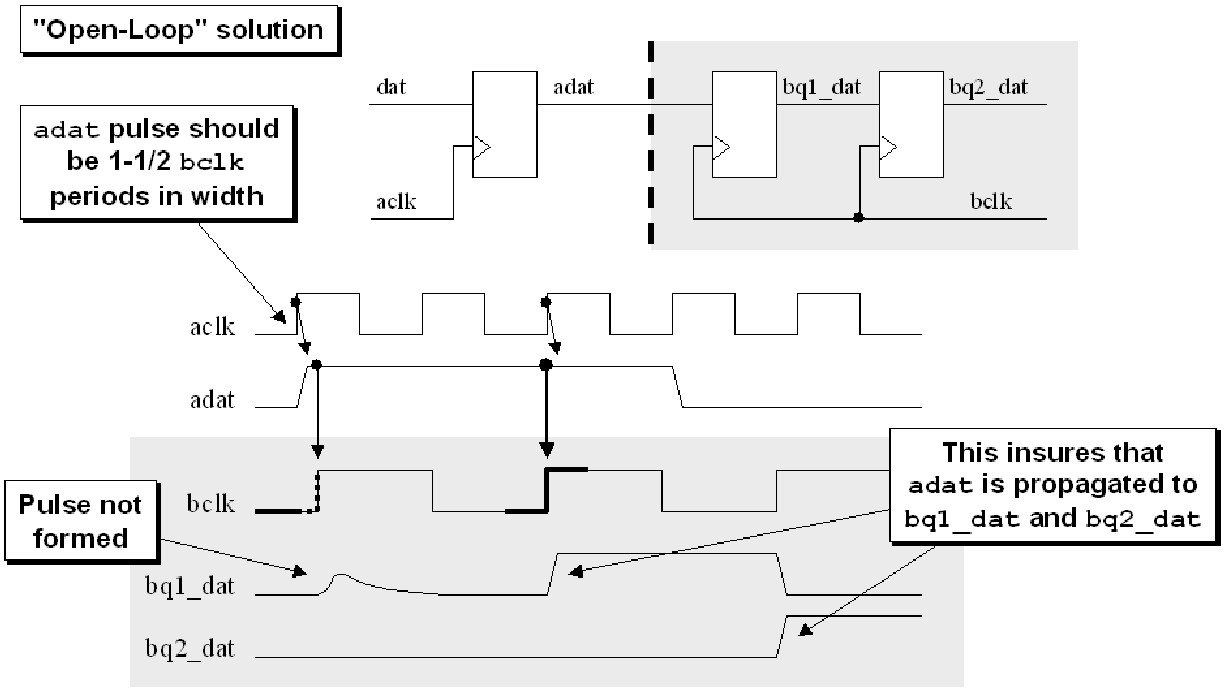

Figure 10 - Lengthened pulse to guarantee that the control signal will be sampled

4.5 Closed loop solution - sampling signals with synchronizers

A second potential solution to this problem is to send an enabling control signal, synchronize it into the new clock domain and then pass the synchronized signal back through another synchronizer to the sending clock domain as an acknowledge signal.

Advantage: synchronizing a feedback signal is a very safe technique to acknowledge that the first control signal was recognized and sampled into the new clock domain.

Disadvantage: there is potentially considerable delay associated with synchronizing control signals in both directions before allowing the control signal to change.

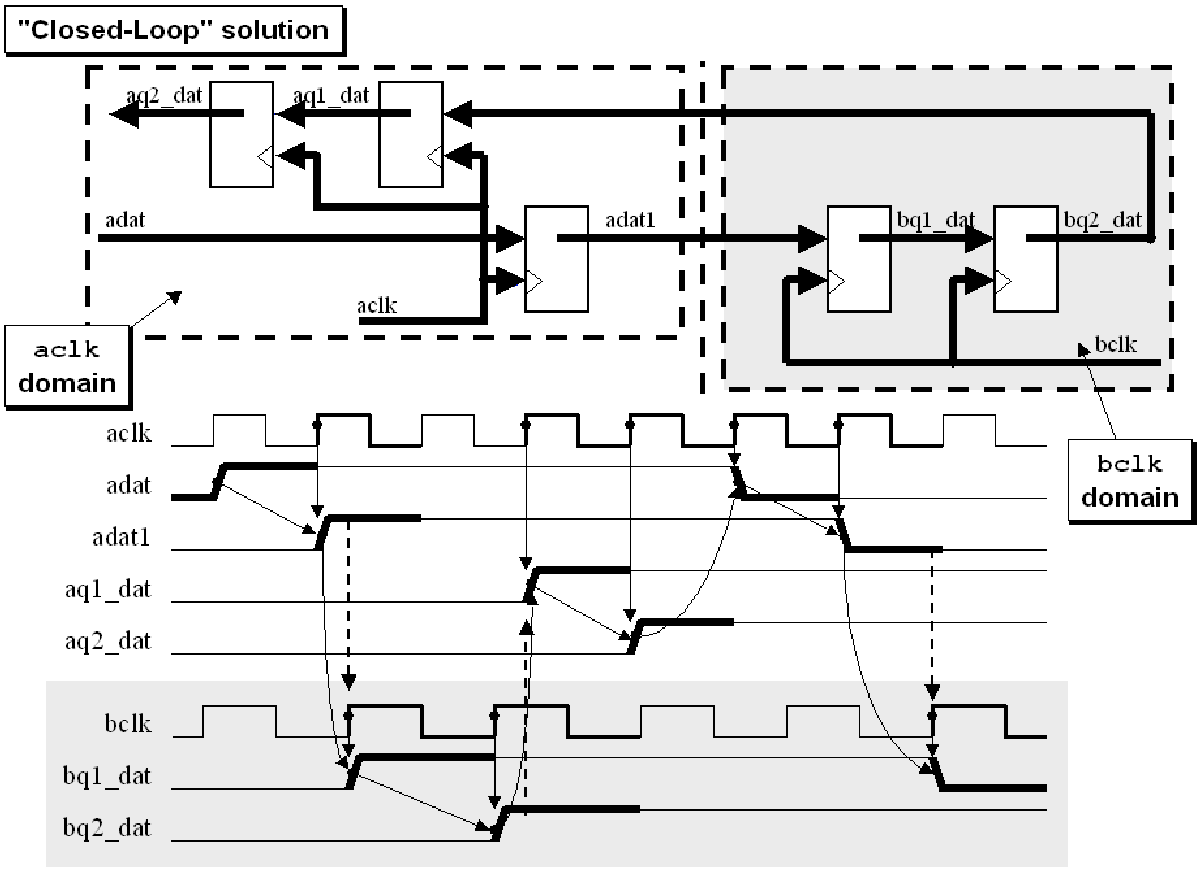

Figure 11 - Signal with feedback to acknowledge receipt

5.0 Passing multiple signals between clock domains

When passing multiple signals between clock domains, simple synchronizers do not guarantee safe delivery of the data.

A frequent mistake made by engineers when working on multi-clock designs is passing multiple CDC bits required in the same transaction from one clock domain to another and overlooking the importance of the *synchronized* sampling of the CDC bits.

The problem is that multiple signals that are synchronized to one clock will experience small data changing skews that can occasionally be sampled on different rising clock edges in a second clock domain. Even if we could perfectly control and match the trace lengths of the multiple signals, differences in rise and fall times as well as process variations across a die could introduce enough skew to cause sampling failures on otherwise carefully matched traces.

Multi-bit CDC strategies must be employed to avoid skewed sampling of the multi-bit value.

5.1 Multi-bit CDC strategies

To avoid multi-bit CDC skewed sampling scenarios, I have classified multi-bit CDC strategies into three main categories:

(1) Multi-bit signal consolidation. Where possible, consolidate multiple CDC bits into 1bit CDC signals.
(2) Multi-cycle path formulations. Use a synchronized load signal to safely pass multiple CDC bits.
(3) Pass multiple CDC bits using gray codes.

Each of these strategies is detailed in the remainder of this section.

5.2 Multi-bit signal consolidation

Where possible, consolidate multiple CDC signals into a 1bit CDC signal. Ask yourself the question, do I really need multiple bits to control logic across a CDC boundary?

Simply using synchronizers on all of the CDC bits is not always good enough as will be shown in the following examples.

If the order or alignment of the control signals is significant, care must be taken to correctly pass the signals into the new clock domain. All of the examples shown in this section are overly simplistic but they closely mimic situations that often arise in real designs.

5.3 Problem - Two simultaneously required control signals.

In the simple example shown in Figure 12, a register in the receiving clock domain requires both a *load* signal and an *enable* signal in order to load a data value into the register. If both the load and enable signals are driven on the same sending clock edge, there is a chance that a small skew between the control signals could cause the two signals to be synchronized into different clock cycles within the receiving clock domain. Under these conditions, the data would not be loaded into the register.

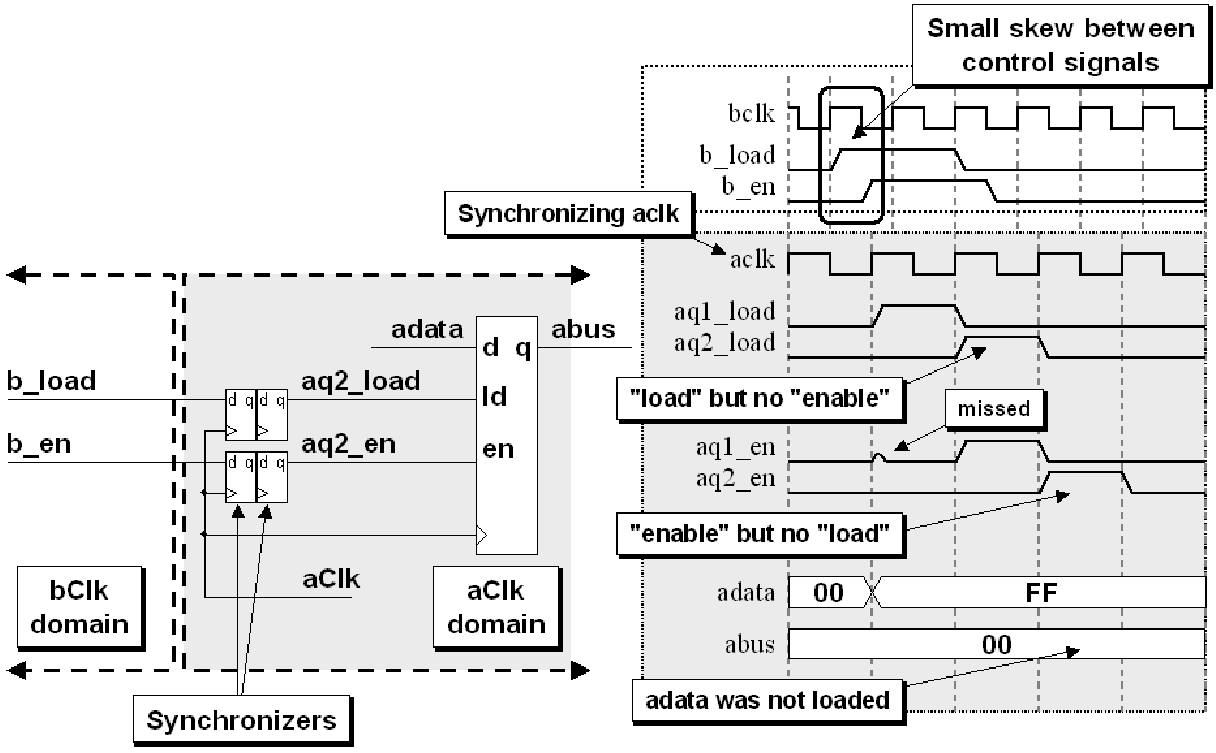

Figure 12 - Problem - Passing multiple control signals between clock domains

5.3.1 Solution - Consolidation

The solution to the problem in section 5.3 is simple, consolidate the control signals. As shown in Figure 13, drive both the load and enable register input signals in the receiving clock domain from just one load-enable signal. Consolidation will remove the potential of two control signals arriving shifted in time.

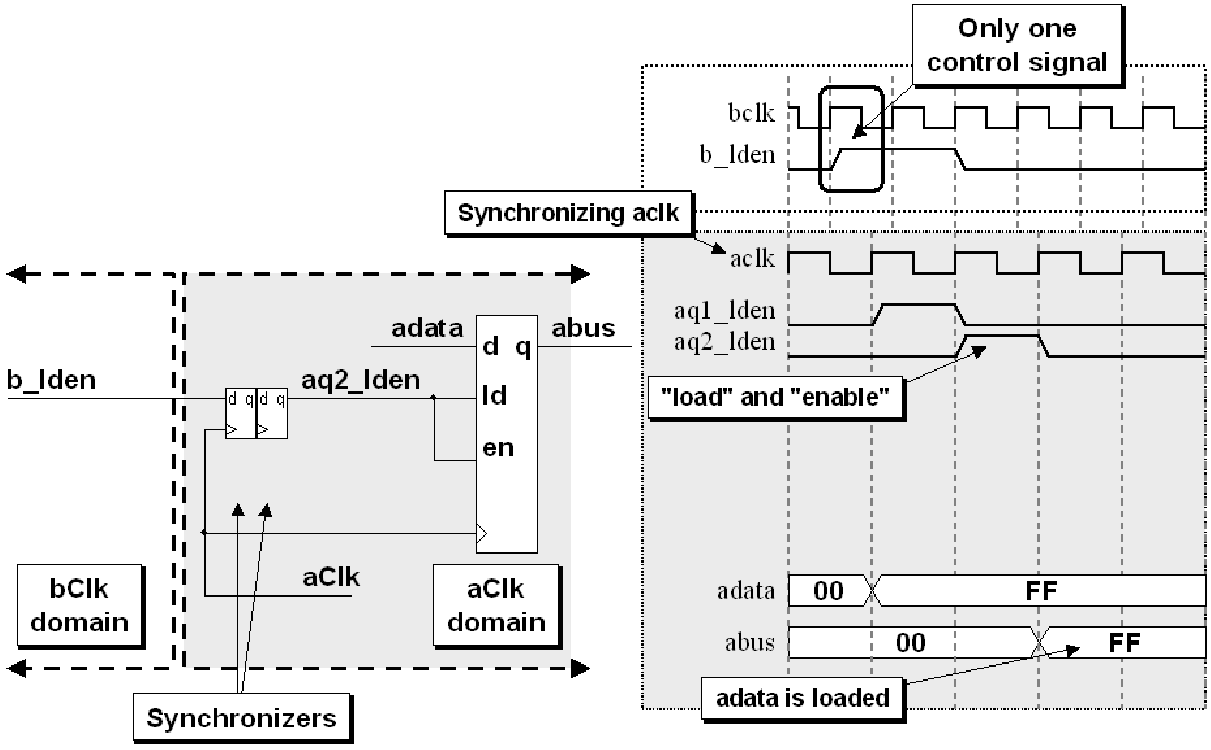

Figure 13 - Solution - Consolidating control signals before passing between clock domains

5.4 Problem - Two phase-shifted sequencing control signals.

The diagram in Figure 14, shows two enable signals, **aen1** and **aen2**, that are sequentially driven from a sending clock domain into the receiving clock domain to control the enable inputs of pipelined data registers. The problem is that in the first clock domain, the **aen1** control signal might terminate slightly before the **aen2** control signal is generated, and the rising edge of the receiving clock might occur in the slight gap between the **aen1** and **aen2** control signal pulses, causing a one-cycle gap to form in the enable control-signal chain in the receiving clock domain. This would cause the **a2** data value to be missed by the second register.

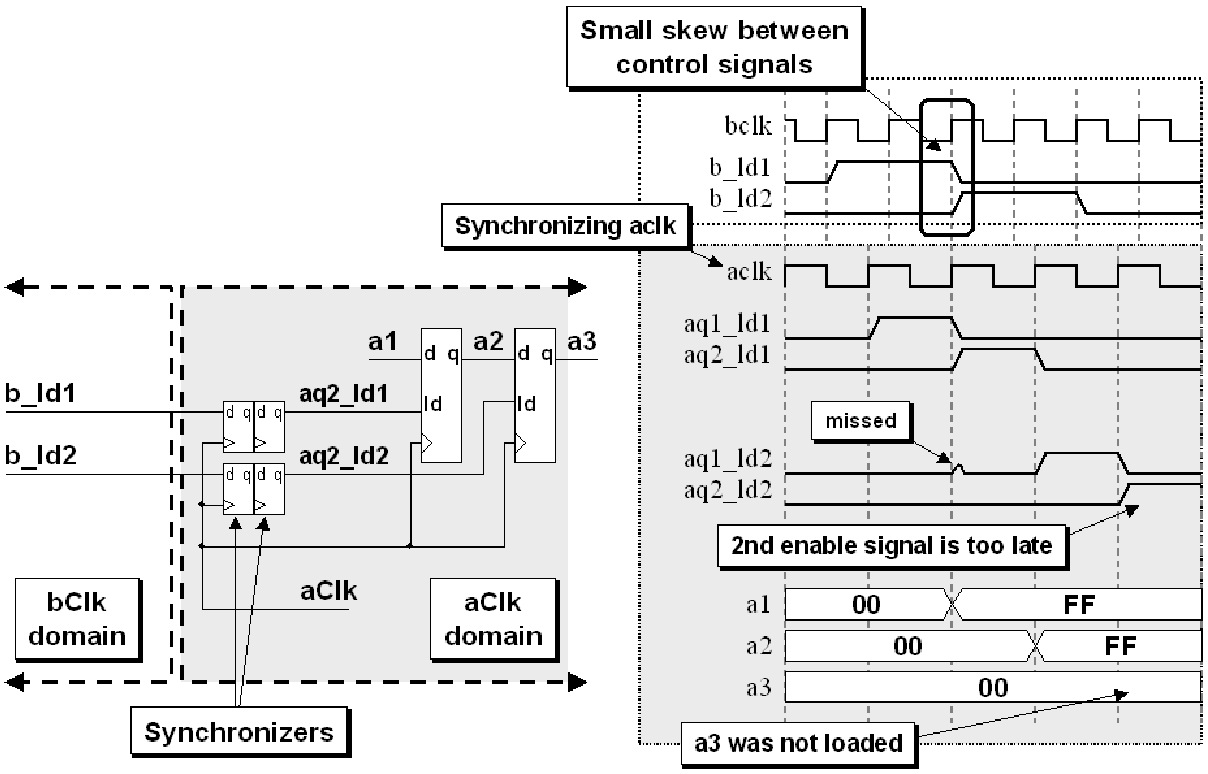

Figure 14 - Problem - Passing sequential control signals between clock domains

5.4.1 Solution - consolidation and an extra flip-flop

The solution to this problem, as shown in Figure 15, is to send only one control signal into the receiving clock domain and generate the second phase-shifted pipelined enable signal within the receiving clock domain.

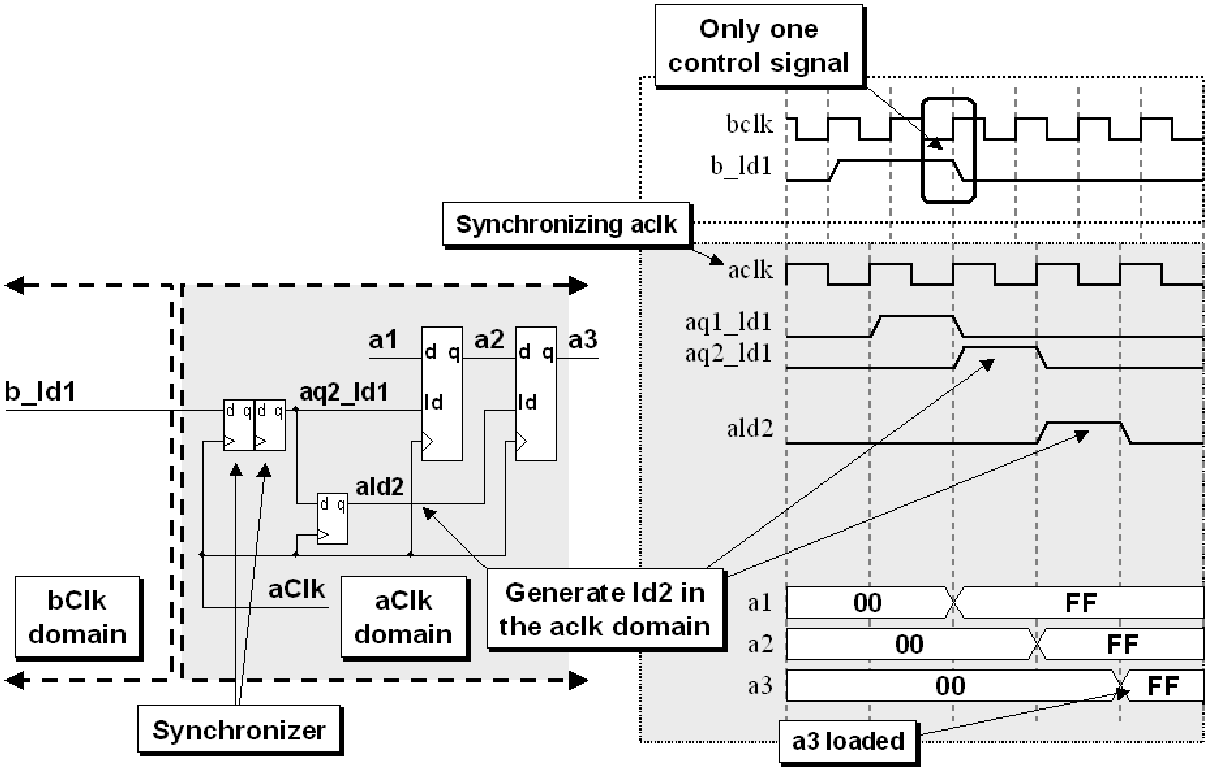

Figure 15 - Solution - Logic to generate proper sequencing signals in the new clock domains

5.5 Problem - Multiple CDC signals

The diagram in Figure 16 shows two encoded control signals being passed between clock domains. If the two encoded signals are slightly skewed when sampled, an erroneous decoded output could be generated for one clock period in the receiving clock domain.

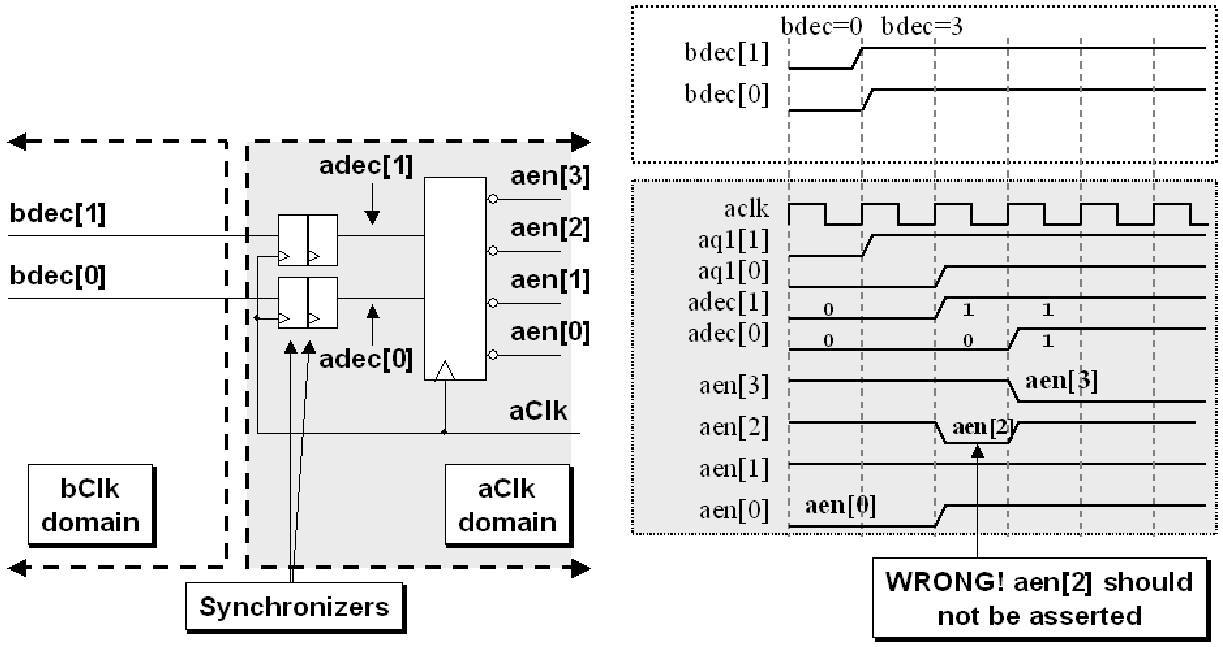

Figure 16 - Problem - Encoded control signals passed between clock domains

5.5.1 Solutions for passing multiple CDC signals

Multi-Cycle Path (MCP) formulations and FIFO techniques can be used to address problems related to passing multiple CDC signals. A description and definition of an MCP formulation is given in section 5.6

There are at least two Multi-Cycle Path (MCP) formulations that can be used to fix this problem:
(1) Closed-loop - MCP formulation with feedback.
(2) Closed-loop - MCP formulation with acknowledge feedback.
The MCP formulation implementation techniques are described starting in the next section.

There are also at least two FIFO strategies that act as closed loop solutions to this problem:
(1) Asynchronous FIFO implementation.
(2) 2-deep FIFO implementation.
The FIFO implementation techniques are described starting in section 5.8.

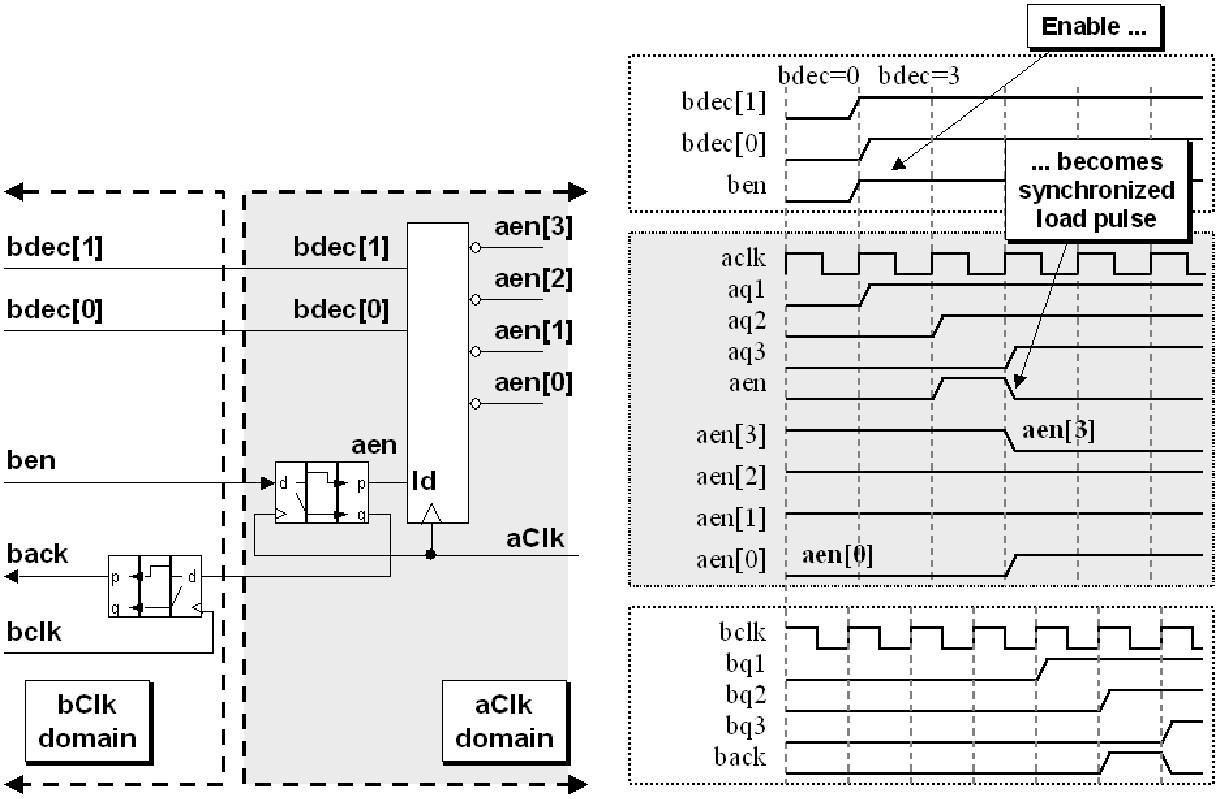

Figure 17 - Logic to pass a synchronized enable pulse between clock domains

5.6 Multi-Cycle Path (MCP) formulation

Using an MCP formulation is a common technique for safely passing multiple CDC signals.

An MCP formulation refers to sending unsynchronized data to a receiving clock domain paired with a synchronized control signal. The data and control signals are sent simultaneously allowing the data to setup on the inputs of the destination register while the control signal is synchronized for two receiving clock cycles before it arrives at the load input of the destination register.

Advantages:
(1) The sending clock domain is not required to calculate the appropriate pulse width to send between clock domains.
(2) The sending clock domain is only required to toggle an enable into the receiving clock domain to indicate that data has been passed and is ready to be loaded. The enable signal is not required to return to its initial logic level.

This strategy passes multiple CDC signals without synchronization, and simultaneously passes a synchronized enable signal to the receiving clock domain. The receiving clock domain is not allowed to sample the multi-bit CDC signals until the synchronized enable passes through synchronization and arrives at the receiving register.

This strategy is called a Multi-Cycle Path Formulation[8] due to the fact that the unsynchronized data word is passed directly to the receiving clock domain and held for multiple receiving clock cycles, allowing an enable signal to be synchronized and recognized into the receiving clock domain before permitting the unsynchronized data word to change.

Because the unsynchronized data is passed and held stable for multiple clock cycles before being sampled, there is no danger that the sampled value will go metastable.

5.6.1 MCP formulation using a synchronized enable pulse

Perhaps the most common method to pass a synchronized enable signal between clock domains is to employ a toggling enable signal that is passed to a synchronized pulse generator to indicate that the unsynchronized multi-cycle data word can be captured on the next receiving clock edge as shown in Figure 18.

A key feature of this synchronized enable pulse generation is that the polarity of the input signal does not matter. In Figure 18, the d-input is toggled high in cycle 1 and by cycle 4 a high signal has propagated through the three synchronizing flip-flops. In cycle 3 the outputs of the q2 and q3 flip-flops have a different polarity causing the synchronized enable pulse to form on the output of the exclusive-or gate in that same cycle. Similarly, the d-input is toggled low in cycle 7 and by cycle 10 a high signal has propagated through the three synchronizing flip-flops. And again in cycle 9 the outputs of the q2 and q3 flip-flops have a different polarity causing the synchronized enable pulse to form on the output of the exclusive-or gate.

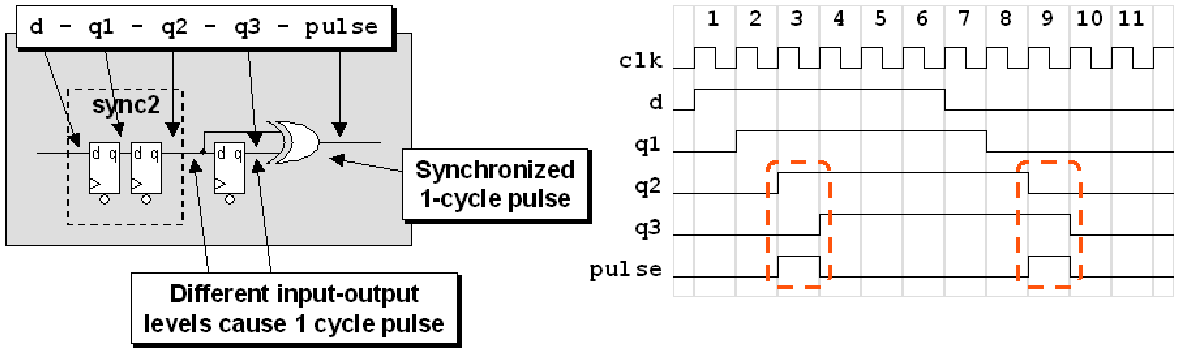

Figure 18 - Synchronized pulse generation logic

Since all of the MCP formulations described in section 5.0 use the synchronized enable pulse generation circuit, it was deemed useful to create and use a smaller equivalent symbol to represent the synchronized enable pulse generation circuit. The equivalent symbol is shown in Figure 19.

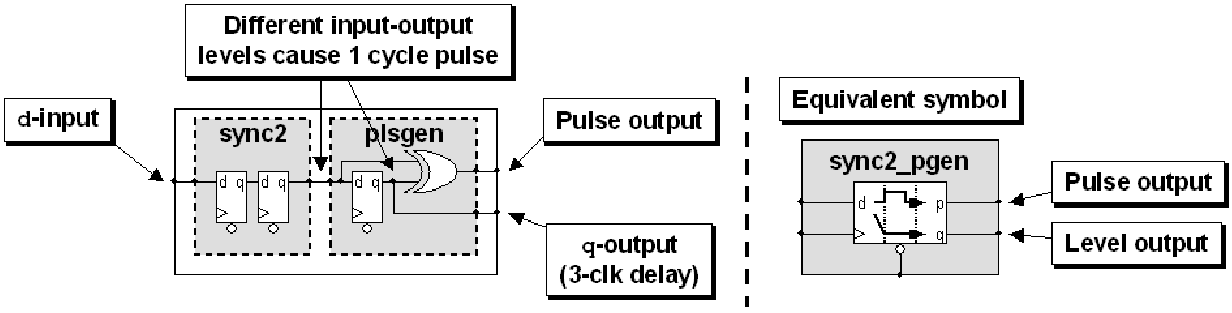

Figure 19 - Synchronized enable pulse generation logic and equivalent symbol

In addition to generating a pulse off of any d-input polarity, the synchronized enable pulse generation circuit also has a q-output that follows the d-input delayed by three clock cycles. The q-output is frequently used as a feedback signal and passed as an acknowledge signal through another synchronized enable pulse generation circuit in the sending clock domain.

Figure 20 shows a typical send-receive toggle-pulse generation design.

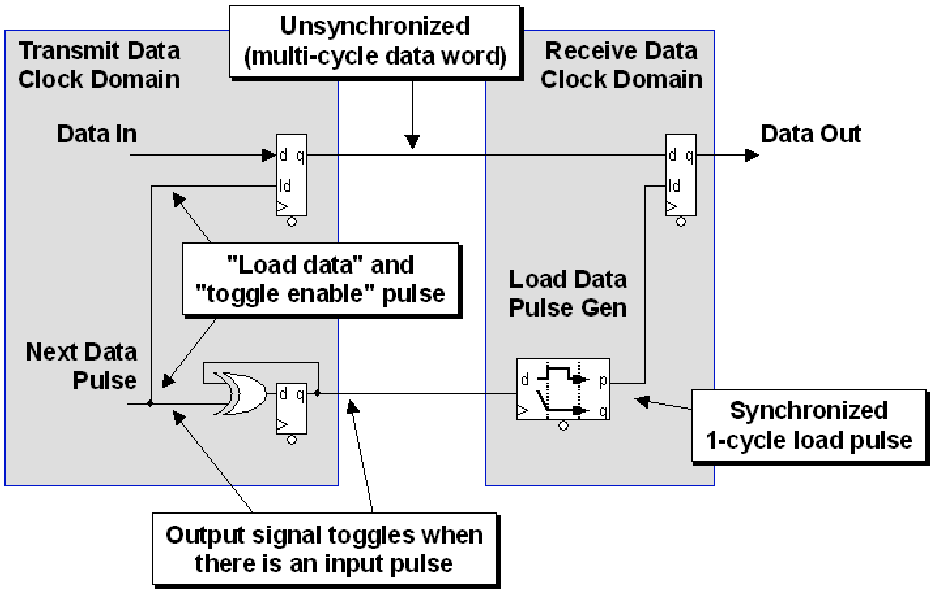

Figure 20 - Multi-Cycle Path (MCP) formulation toggle-pulse generation

Using this technique, it is required that the receiving clock domain have logic in place to capture the data when the pulse is detected, because the pulse will only be valid for one receiving clock cycle per multi-cycle data word.

5.6.2 Closed-loop - MCP formulation with feedback

An important technique when using an MCP formulation is to pass the enable signal back to the sending clock domain as an acknowledge signal as shown in Figure 21.

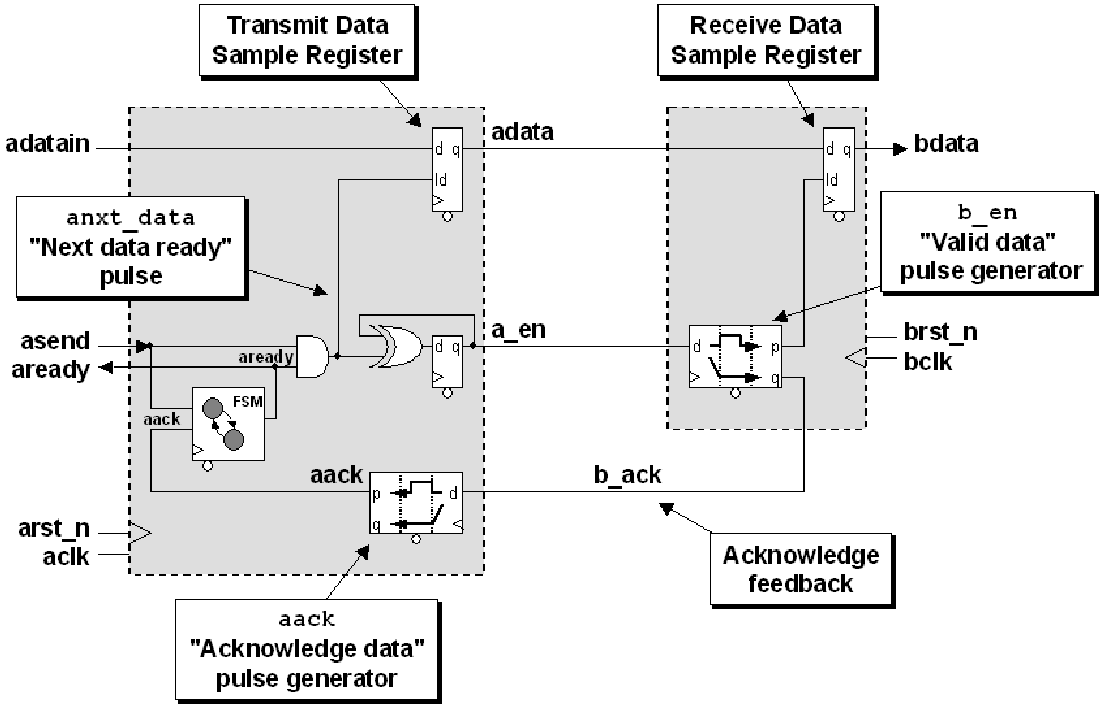

Figure 21 - Multi-Cycle Path (MCP) formulation toggle-pulse generation with acknowledge

For the example in Figure 21, the acknowledge feedback signal (**`b_ack`**) generates an acknowledge pulse (**`aack`**) that is used as an input to a small **`READY-BUSY`**, 1-state FSM block that generates a ready signal (**`aready`**) to indicate that it is now safe to change the data input (**`adatain`**) value again. Once the **`aready`** signal goes high, the sender is free to send new data (**`adatain`**) and the accompanying **`asend`** control signal.

This is an automatic feedback path that assumes that the receiving clock domain will always be ready for the next data word synchronized through an MCP formulation.

5.6.3 Closed-loop - MCP formulation with acknowledge feedback

A fully responsive variation of the technique described in section 5.6.2 uses an MCP formulation is to pass the enable signal back to the sending clock domain as an acknowledge signal only after the receiving clock domain acknowledges receipt of the data with a **`bload`** pulse as shown in Figure 22.

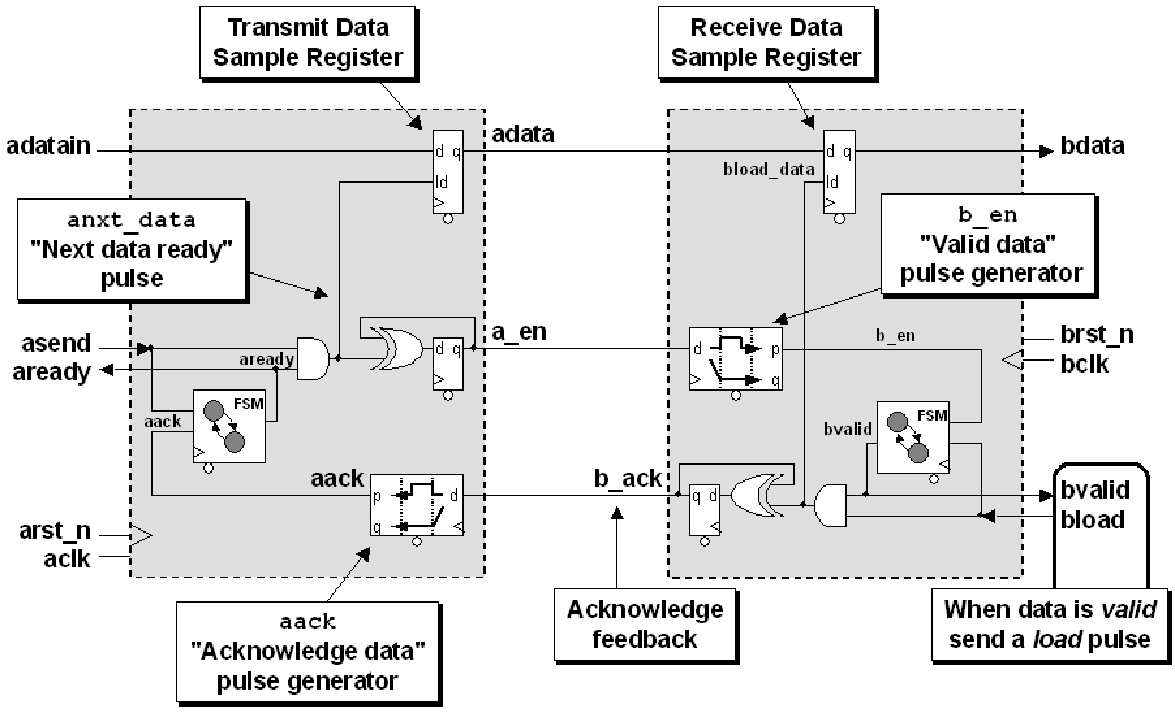

Figure 22 - Multi-Cycle Path (MCP) formulation toggle-pulse generation with ready-ack

For the example in Figure 22, the receiving clock domain has a small **`WAIT-READY`**, 1-state FSM that sends a valid signal (**`bvalid`**) to the receiving logic when data is valid on the input to the data register. The data is not actually loaded until the receiving logic acknowledges that the data should be loaded by asserting the **`bload`** signal. There is no feedback to the sending clock domain until the data has been loaded, then the **`b_ack`** signal is sent back the same as the MCP formulation with automatic feedback.

This is an feedback path requires action on the part of the receiving clock domain before data is captured and feedback is sent.

5.7 Synchronizing counters

As mentioned earlier, when passing multiple signals between clock domains, an important question to ask is, do I need to sample every value of a signal that is passed from one clock domain to another? With counters, the answer is frequently, no!

Reference [1] details FIFO design techniques where gray code counters are sampled between clock domains and intermediate gray count values are often missed. For this FIFO design, the greater consideration is to make sure that the counters cannot overrun their boundaries, which could cause missed full and empty flag detection. Even though the sampled gray count values between clock domains are often missed, the design is robust and all important gray count values are appropriately sampled. See [1] for details.

Since a valid design might be allowed to skip some count value samples, can any counter be used to pass count values across a CDC boundary? The answer is no.

5.7.1 Binary counters

One characteristic of binary counters is that half of all sequential binary incrementing operations require that two or more counter bits must change. Trying to synchronize a binary counter across a CDC boundary is the same as trying to synchronize multiple CDC signals into a new clock domain. If a simple 4-bit binary counter changes from address 7 (binary 0111) to address 8 (binary 1000), all four counter bits will change at the same time. If a synchronizing clock edge comes in the middle of this transition, it is possible that any 4-bit binary pattern could be sampled and synchronized into the new clock domain as shown in Figure 23.

Binary Count Values	
00	0 0 0 0
01	0 0 0 1
02	0 0 1 0
03	0 0 1 1
04	0 1 0 0
05	0 1 0 1
06	0 1 1 0
07	0 1 1 1
08	1 0 0 0
09	1 0 0 1
10	1 0 1 0
11	1 0 1 1
12	1 1 0 0
13	1 1 0 1
14	1 1 1 0
15	1 1 1 1

07 -> 08 possible binary transitions		
0 1 1 1	-> 1 0 0 0	(07->08)
0 1 1 1	-> 0 0 0 0	(07->00)
0 1 1 1	-> 0 0 0 1	(07->01)
0 1 1 1	-> 0 0 1 0	(07->02)
0 1 1 1	-> 0 0 1 1	(07->03)
0 1 1 1	-> 0 1 0 0	(07->04)
0 1 1 1	-> 0 1 0 1	(07->05)
0 1 1 1	-> 0 1 1 0	(07->06)
0 1 1 1	-> 0 1 1 1	(07->07)
0 1 1 1	-> 1 0 0 0	(07->08)
0 1 1 1	-> 1 0 0 1	(07->09)
0 1 1 1	-> 1 0 1 0	(07->10)
0 1 1 1	-> 1 0 1 1	(07->11)
0 1 1 1	-> 1 1 0 0	(07->12)
0 1 1 1	-> 1 1 0 1	(07->13)
0 1 1 1	-> 1 1 1 0	(07->14)
0 1 1 1	-> 1 1 1 1	(07->15)

Figure 23 - Binary count values sampled in mid-transition

In a FIFO design, the new synchronized binary value might trigger a false full or empty flag, or even worse, it might *not* trigger a *real* full or empty flag causing data to be lost due to FIFO

overflow or causing invalid data to be read from the FIFO due to an attempt to read data when the FIFO is really empty.

5.7.2 Gray codes

Gray codes are named after Frank Gray[4] and the safest counters that can be used in multi-clock designs are Gray code counters. Gray codes only allow one bit to change for each clock transition, eliminating the problem associated with trying to synchronize multiple changing CDC bits across a clock domain.

Standard gray codes have very nice translation properties to convert gray-to-binary and back again. Using these conversions, it is simple to design efficient gray code counters.

5.7.3 Gray-to-binary conversion

To convert a gray-code value to an equivalent binary-code value, using an n-bit gray code value as an example, binary bit 0 is equal to the exclusive-or of gray code bit 0 exclusive-ored with all other gray code bits from 1 to n. Binary bit 1 is equal gray code bit 1 exclusive-ored with all other gray code bits from 2 to n, etc. The most significant binary bit is just equal to the most significant gray code bit.

The equations for a sample 4-bit gray-to-binary conversion are shown in Figure 24.

```
bin[0] = gray[3] ^ gray[2] ^ gray[1] ^ gray[0];
bin[1] = gray[3] ^ gray[2] ^ gray[1];
bin[2] = gray[3] ^ gray[2];
bin[3] = gray[3];
```

Figure 24 - 4-bit gray-to-binary conversion equations

The easiest way to code a gray-to-binary converter is to code a for-loop and do an exclusive-or reduction on a gray code vector with variable index range, where each time through the loop the LSB of the index range increases until we are left with a simple assignment of bin[MSB] = ^gray[MSB:MSB] (just the 1-bit MSB of the gray code vector), as shown in Example 1.

```
module gray2bin_bad #(parameter SIZE = 4)
 (output logic [SIZE-1:0] bin,
  input  logic [SIZE-1:0] gray);

  // Syntax Error - variable index range
  always_comb
    for (int i=0; i<SIZE; i++)
      bin[i] = ^(gray[SIZE-1:i]);
endmodule
```

Example 1 - Non-working but conceptually correct gray-to-binary SystemVerilog model

Unfortunately, Verilog and SystemVerilog do not permit part selects using a variable index range so the code in Example 1, although conceptually correct, will not compile.

To address this issue, remember that an exclusive-or gate is really a programmable inverter. If one input is tied high, the other input is inverted and passed to the output. Similarly, if one input is tied low, the other input is passed to the output without inversion (no change from input to output).

Taking advantage of the fact that any added exclusive-or operation that involves a 0-input does not change the outcome of the operation, the way to approach of a gray-to-binary conversion is to exclusive-or the significant gray-code bits with padded 0's as shown in Figure 25.

```
bin[0] = gray[3] ^ gray[2] ^ gray[1] ^ gray[0] ; // gray>>0
bin[1] =   1'b0  ^ gray[3] ^ gray[2] ^ gray[1] ; // gray>>1
bin[2] =   1'b0  ^   1'b0  ^ gray[3] ^ gray[2] ; // gray>>2
bin[3] =   1'b0  ^   1'b0  ^   1'b0  ^ gray[3] ; // gray>>3
```

Figure 25 - 4-bit gray-to-binary conversion equations - 2nd method

The corresponding parameterized SystemVerilog model for this simplified algorithm is shown in Example 2. This example is syntactically correct, will compile and does work.

```
module gray2bin #(parameter SIZE = 4)
 (output logic [SIZE-1:0] bin,
  input  logic [SIZE-1:0] gray);

  always_comb
    for (int i=0; i<SIZE; i++)
      bin[i] = ^(gray>>i);
endmodule
```

Example 2 - Parameterized and correct gray-to-binary SystemVerilog model

What happens to all of the extra exclusive-or operations with inputs tied to 0? Synthesis tools recognize that exclusive-or gates with a constant-0 on one input can be optimized away to infer a very efficient implementation of the design.

5.7.4 Binary-to-gray conversion

To convert a binary value to an equivalent gray-code value, using an n-bit binary value as an example, gray-code bit 0 is equal to the exclusive-or of binary bits 0 and 1. Gray-code bit 1 is equal to the exclusive-or of binary bits 1 and 2, etc. The most significant gray-code bit is just equal to the most significant binary bit.

The equations for a sample 4-bit binary-to-gray conversion are shown in Figure 26.

```
gray[0] = bin[0] ^ bin[1];
gray[1] = bin[1] ^ bin[2];
gray[2] = bin[2] ^ bin[3];
gray[3] = bin[3] ^  1'b0 ; // same as gray[3] = bin[3];
```

Figure 26 - 4-bit binary-to-gray conversion equations

The easiest way to code a binary-to-gray converter is to code a simple continuous assignment that performs a bit-wise exclusive-or operation between the binary vector and a right-shifted version of the same binary vector as shown in
Example 3. This example is syntactically correct, will compile and does work.

```
module bin2gray #(parameter SIZE = 4)
 (output logic [SIZE-1:0] gray,
  input  logic [SIZE-1:0] bin);

  assign gray = (bin>>1) ^ bin;
endmodule
```

Example 3 - Parameterized binary-to-gray SystemVerilog model

5.7.5 Gray code counter style #1

We can build a gray code counter by using the conversions that were shown in sections 5.7.3 and 5.7.4. For any gray code counter, it is important to remember that the gray-output must be registered to eliminate any combinational settling in the design.

The SystemVerilog code for gray-code counter style #1 incorporates a gray-to-binary converter, a binary-to-gray converter and increments the binary value between conversions as shown in Figure 27.

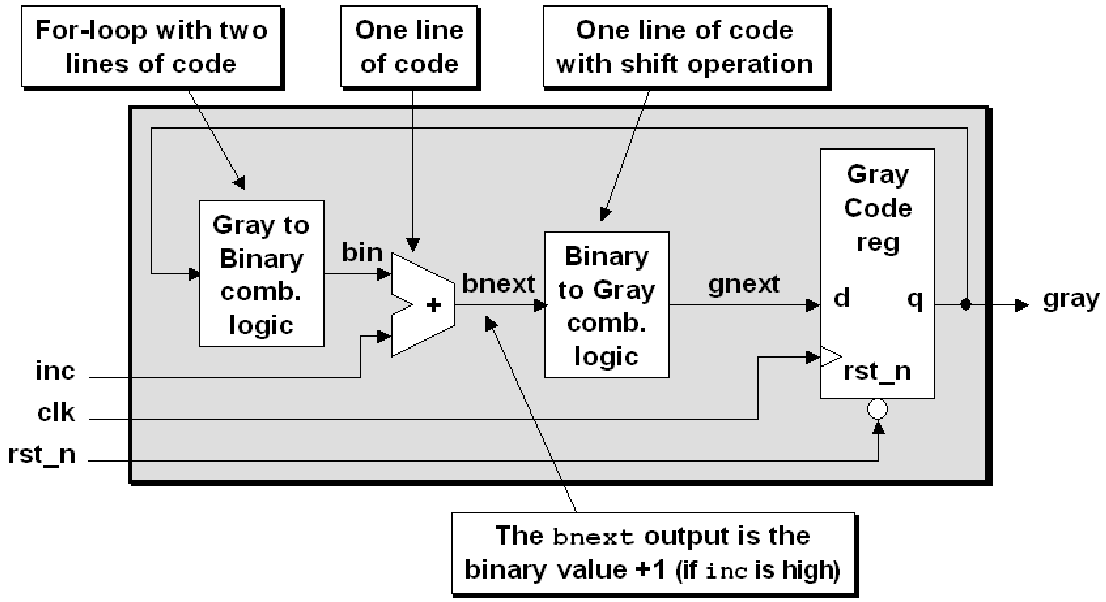

Figure 27 - Gray code counter style #1 - only one gray code register

The corresponding parameterized SystemVerilog model for the gray-code counter style #1 is shown in Example 4.

```
module graycntr #(parameter SIZE = 5)
 (output logic [SIZE-1:0] gray,
  input  logic            clk, inc, rst_n);

  logic [SIZE-1:0] gnext, bnext, bin;

  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) gray <= '0;
    else        gray <= gnext;

  always_comb begin
    for (int i=0; i<SIZE; i++)
      bin[i] = ^(gray>>i);
    bnext = bin + inc;
    gnext = (bnext>>1) ^ bnext;
  end
endmodule
```

Example 4 - Parameterized gray-code counter SystemVerilog model

5.7.6 Gray code counter style #2

We can build the second style of gray code counter by using just the binary-to-gray conversions that were shown in section 5.7.4. This gray code counter actually both a binary count register and a gray code count register.

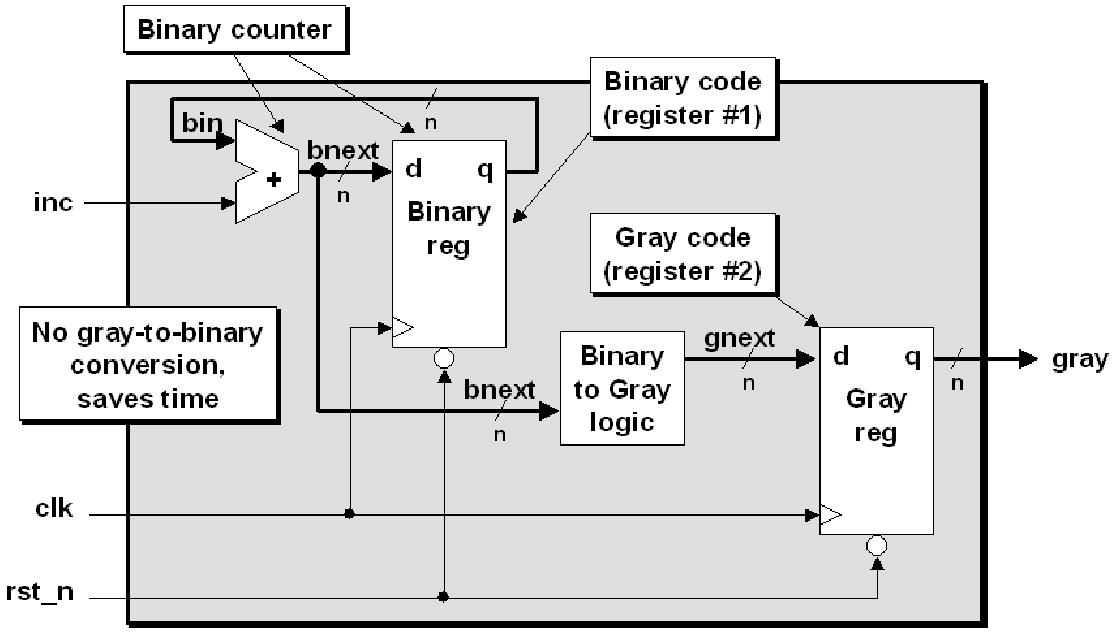

Figure 28 - Gray code counter style #2 - binary register and gray code register

The SystemVerilog code for gray-code counter style #2 incorporates a binary counter to eliminate the need for the gray-to-binary conversion, and uses the next binary count value to do the binary-to-gray conversion that is then registered into the gray code register. This style uses twice as

many flip-flops but a shorter combinational logic path to generate the next gray code value, which makes this implementation faster than gray code counter style #1. The block diagram for gray code counter style #2 is shown in Figure 28,

The corresponding parameterized SystemVerilog model for the gray-code counter style #2 is shown in Example 5.

```
module graycntr #(parameter SIZE = 5)
 (output logic [SIZE-1:0] gray,
  input  logic            clk, full, inc, rst_n);

  logic [SIZE-1:0] gnext, bnext, bin;

  always_ff @(posedge clk or negedge rst_n)
    if (!wrst_n) {bin, gray} <= '0;
    else         {bin, gray} <= {bnext, gnext};

  assign bnext = !full ? bin + inc : bin;
  assign gnext = (bnext>>1) ^ bnext;
endmodule
```

Example 5 - Parameterized gray-code counter with binary counter

5.8 Additional multi-bit CDC techniques

In addition to the MCP formulation techniques described in earlier sections, I have found a number of engineers that use standard FIFOs to pass data and control signals between clock domains.

There are at least two interesting FIFO implementation strategies that can be used to address multi-bit CDC signal integrity:
(1) Asynchronous FIFO implementations.
(2) 2-deep FIFO implementation.

5.8.1 Multi-bit CDC signal passing using asynchronous FIFOS

Passing multiple bits, whether data bits or control bits, can be done through an asynchronous FIFO. An asynchronous FIFO is a shared memory or register buffer where data is inserted from the write clock domain and data is removed from the read clock domain. Since both sender and receiver operate within their own respective clock domains, using a dual-port buffer, such as a FIFO, is a safe way to pass multi-bit values between clock domains.

A standard asynchronous FIFO device allows multiple data or control words to be inserted as long as the FIFO is not full, and the receiver and then extract multiple data or control words when convenient as long as the FIFO is not empty.

Most of the hard work in a FIFO design is done through the synchronization of gray code counters and a proven FIFO design technique is described in [1].

5.8.2 Multi-bit CDC signal passing using 1-deep / 2-register FIFO synchronizer

Another interesting variation on passing multiple control and data bits across CDC boundaries involves the use of a 1-deep two register FIFO as shown in Figure 29.

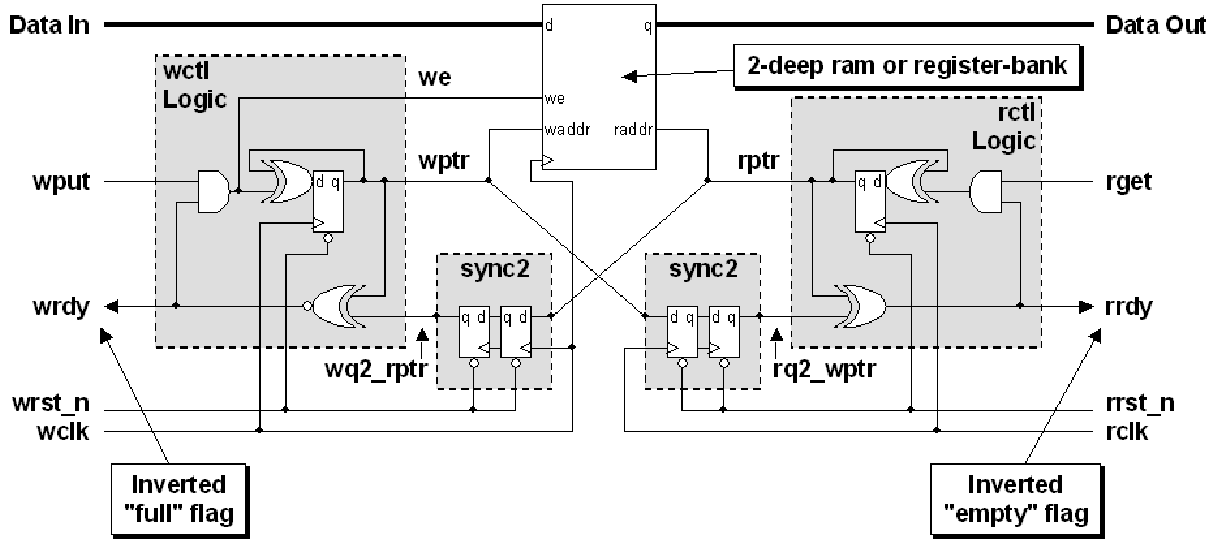

Figure 29 - 1-deep / 2-register FIFO synchronizer block diagram

This 1-deep two register FIFO has a number of interesting characteristics. Since the FIFO is built using only two registers or a 2-deep dual port RAM, the gray code counters used to detect full and empty are simple toggle flip-flops, which is really nothing more than 1-bit binary counters (remember, the MSB of a standard gray code is the same as the MSB of a binary code).

On reset, both pointers are cleared and the FIFO is empty and hence the FIFO is *not full*. We use the inverted not-full condition to indicate that the FIFO is ready to receive a data or control word (**`wrdy`** is high). After a data or control word is put into the FIFO (using **`wput`**), the **`wptr`** toggles and the FIFO becomes full, or in other words, the **`wrdy`** signal goes low, which also disables the ability to toggle the **`wptr`** and therefore also disables the ability to put another word into the 2-register FIFO until the first word is removed from the FIFO by the receiving clock-domain logic.

What is especially interesting about this design is that the **`wptr`** is now pointing to the second location in the 2-register FIFO, so when the FIFO does again become ready (when **`wrdy`** is high), the **`wptr`** is already pointing to the next location to write.

The same concept is replicated on receiving side of the FIFO. When a data or control word is written into the FIFO, the FIFO becomes not empty. We use the inverted not-empty condition to indicate that the FIFO is has a data or control word that is ready to be received (**`rrdy`** is high).

By using two registers to store the multi-bit CDC values, we are able to remove one clock cycle from the send MCP formulation and another cycle from the acknowledge feedback path.

6.0 Naming conventions & design partitioning

Naming conventions help to ensure good team communication and also facilitate the use of scripting languages to gather and group all signals in a design that are associated with a particular clock. Good design partitioning can significantly reduce the effort to synthesize and verify the timing of a multi-clock design. Recommended naming conventions and design partitioning are discussed in this section.

There are two approaches to address potential CDC problems: (1) verify that the design meets qualified CDC rules, (2) avoid the problem. Both approaches are valuable and should be used to ensure an error-free design.

The first approach, verification of CDC design rules, typically requires the use of special tools to check the design for possible CDC violations. When I wrote my first paper on multi-clock design in 2001, I was unaware of any tool on the market that performed checking of CDC rules. Today there are a number of companies that provide such tools (see [11] for a list of companies and tools in the CDC verification space).

The second approach, avoid the problem, can be done by employing a few good coding guidelines as outlined below.

6.1 Clock & signal naming conventions

A number of useful clock and signal naming conventions have been used by various design teams.

Guideline: Use a clock naming convention to identify the clock source of every signal in a design.

Reason: A naming convention helps all team members to identify the clock domain for every signal in a design and also makes grouping of signals for timing analysis easier to do using regular expression "wild-carding" from within a synthesis script.

One proven naming convention requires that a leading prefix character be used to identify the various asynchronous clock domains. Examples included: **`uClk`** for the microprocessor clock, **`vClk`** for the video clock and **`dClk`** for the display clock.

Each signal is then synchronized to one of the clock domains in the design and each signal-name is labeled with a prefix character to identify the clock domain used to generate that signal. For example, any signal that is generated by the **`uClk`** is labeled with a **`u`**-prefix in the signal name, such as **`uaddr`**, **`udata`**, **`uwrite`**, etc. Any signal that is generated by the **`vClk`** is similarly labeled with a **`v`**-prefix in the signal name, such as **`vdata`**, **`vhsync`**, **`vframe`**, etc. The same signal naming convention is used for all signals generated by any of the other clocks in the design.

Using this technique, any engineer on the design team can easily identify the clock-domain source of any signal in the design and either use the signals directly or pass the signals through proper synchronization so that the signals can be used within a new clock domain.

The exact naming convention is not important, but it is vital that every engineer on the project agrees to adhere to the naming convention chosen by the team. A naming convention will significantly contribute to the productivity of the design team.

6.1.1 Multi-clock / multi-source modules with no naming convention

If your team does not using any particular clock-oriented signal naming convention and if modules are allowed to have multiple clock inputs, there is always the danger that the CDC analysis tool might not be setup correctly and it is easy to miss bad CDC design practices.

Even if your team has access to good CDC analysis tools, I strongly recommend that you take a few simple steps to make analysis and recognition of potential CDC design problems easier to identify and debug.

6.2 Timing verification for each clock domain

To verify the timing of any design, one must verify the that timing is met for each clock domain in a design. Although tools have improved over the past decade to help automate the analysis and verification of signals in separate clock domains, it is still a good practice to approach multi-clock design using good partitioning and naming conventions.

By partitioning a design to permit only one clock per module, static timing analysis becomes a significantly easier task for each domain in the design.

6.3 Clock oriented design partitioning

Some of the simplest and best design partitioning methodologies are implemented using design partitioning at clock boundaries.

Guideline: Only allow one clock per module[9].

Reason: Static timing analysis and creating synthesis scripts is more easily accomplished on single-clock modules or groups of single-clock modules.

Exception: The top-level module that connects together the signals from all of the different clock domains will naturally have all of the clocks as inputs to this module. Minimize your multi-clock verification effort and only allow the top-module to have multiple clock inputs.

Guideline: Partition the design blocks into one-clock modules.

Reason: The timing verification of completely synchronous sub-blocks can be easily verified using STA (Static Timing Analysis) tools and partitioning the design blocks into multiple one-

clock domain sub-blocks turns a large complicated timing analysis task into multiple, completely synchronous, one-clock designs.

Guideline: Create synchronizer modules to pass signals from one clock domain into another clock domain and only allow one clock per synchronizer module.

Reason: It is given that any signal passing from one clock domain to another clock domain will eventually experience setup and hold time problems. Isolating the CDC boundary logic can significantly reduce the design and verification effort of multi-clock designs.

Under most conditions, the synchronizer modules will be the only blocks in the design that will experience intentional setup and hold time violations. When passing signals between asynchronous clock domains, it is given that timing violations will occur, that is the whole reason why synchronizers must be added to a design.

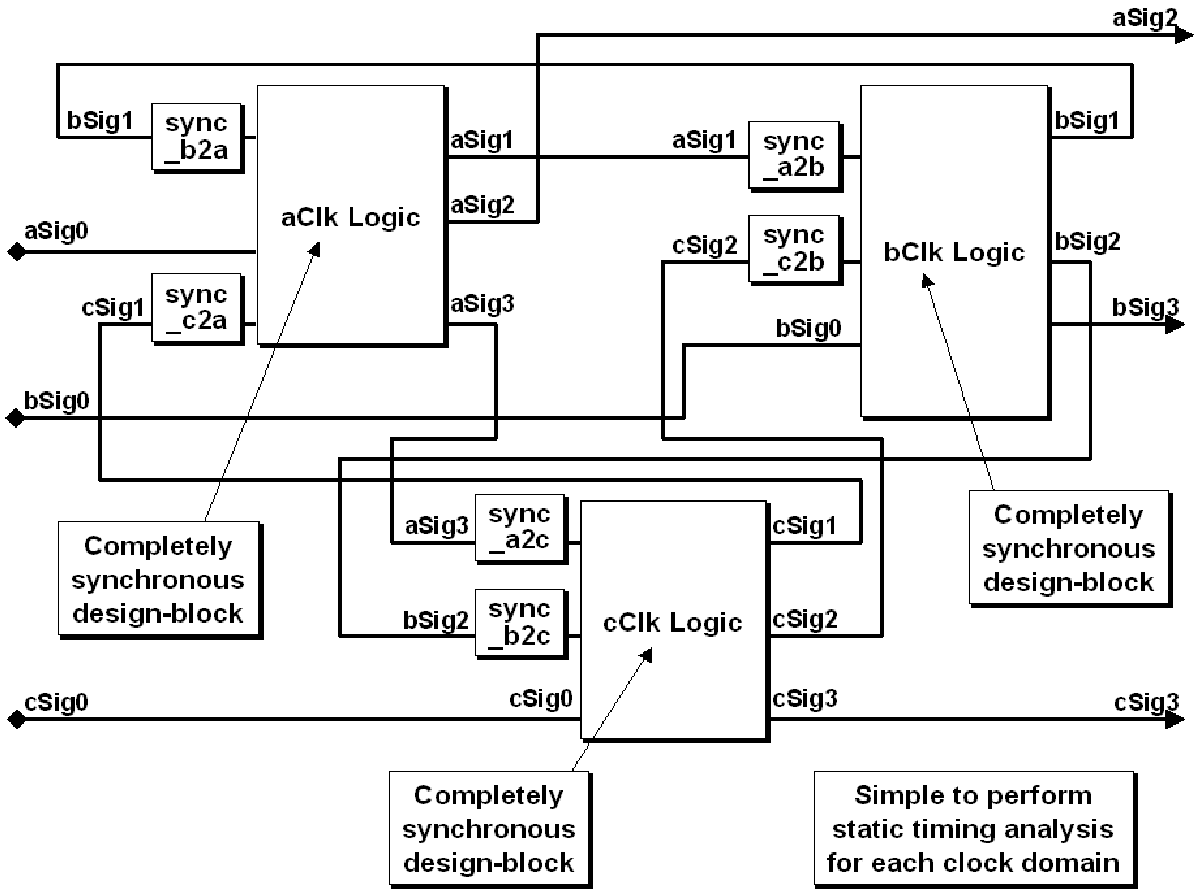

Figure 30 - Design partitioned on clock boundaries

Consider an example design with three clock domains, labeled **`aClk`**, **`bClk`** and **`cClk`** and shown in Figure 30. In this design, all of the **`aClk`** design blocks have been grouped into a single **`aClk`** Logic block. All of the **`bClk`** design blocks have been grouped into a single **`bClk`** Logic block and similarly we have created a **`cClk`** Logic block. Any signal that originates in an asynchronous

clock domain passes through a synchronizer module before it is permitted to drive an input of another Logic block.

6.3.1 Timing analysis of clock-partitioned modules

Using the clock-oriented design partitioning strategy, all of the inputs and outputs of each design block are completely synchronous to just one clock. This is the easiest type of design to verify using static timing analysis (STA) tools because there are no false paths in the design.

Group together all design modules that are clocked within each clock domain. One group should be formed for each clock domain in the design. These groups will be timing verified as if each were a separate, completely synchronous design. For each clock domain, we have a single design block and we can easily perform worst-case (max-time / setup time checking) timing analysis, and best-case (min-time / hold time checking) timing analysis.

Also using this clock-oriented partitioning strategy, each of the CDC boundaries has been isolated using a synchronizer module. Each synchronizer module only includes either synchronizer cells provided by the ASIC or FPGA vendor (preferred), or is built using flip-flops connected as pairs to form synchronizer-equivalent cells.

If synchronizer cells are available from the ASIC or FPGA vendor, and instantiated into the design, there will be no need to verify setup and hold times on these modules, since the vendor should have already created a cell layout that does not violate setup or hold times between the flip-flop stages.

If the synchronizers are synthesized from RTL code, it is most important to perform best-case timing analysis to make sure that the flip-flops are not placed too close together in such a way that the output from the first stage might change too quickly to satisfy the hold time requirement of the input of the second stage. A colleague recently pointed out that worst-case timing analysis should also be performed just in case the layout tools happen to place the two synchronizer flip-flops far apart on the ASIC or FPGA die. I agree with this updated recommendation.

Because of the partitioning of the separate synchronizers, gate-level simulations can more easily be configured to ignore setup and hold time violations on the first stage of each synchronizer

The static timing analysis of the RTL synchronizers requires simple set_false_path commands to remove the inputs from the STA. We know that there are timing problems at the inputs of the synchronizers, that is why the synchronizers are used.

By partitioning design and synchronizer blocks to permit only one clock per module, static timing analysis becomes a significantly easier task to perform. Synthesis script commands used to address multiple clock domain issues now become a matter of grouping, identifying false paths and performing min-max timing analysis.

6.4 Partitioning with MCP formulations

Partitioning a design at clock boundaries into separate design blocks and synchronizer blocks works well most of the time, but if multiple signals need to be passed between clock domains using an MCP formulation, then some of the signals that are passed to a design block may come from a different clock domain as shown in Figure 31.

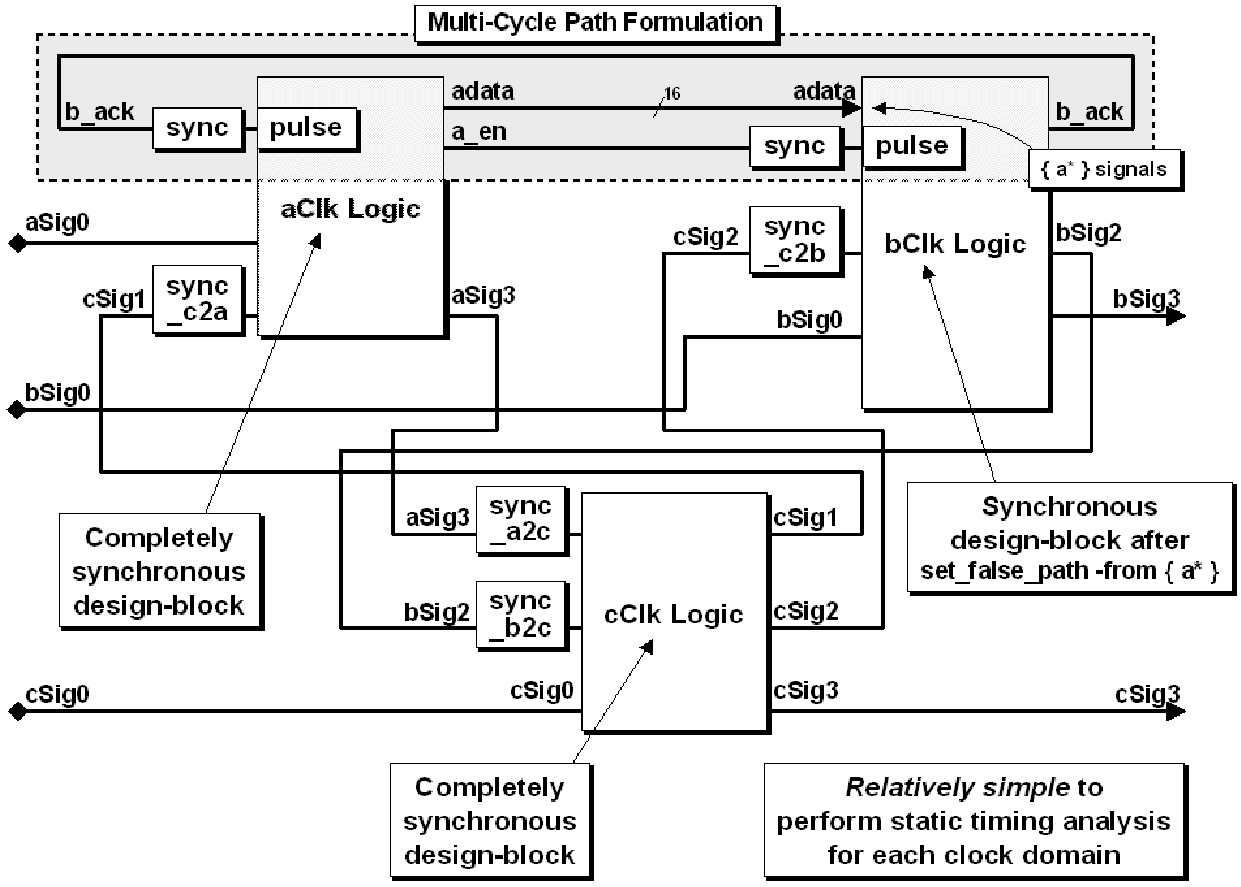

Figure 31 - Partitioned design with MCP formulation

Design blocks with asynchronous inputs can still be easily timed if a clock based naming convention has been used for the signals in the design. Before performing STA on the design block in question, simply exclude the asynchronous inputs from the analysis.

In general, only the inputs to the synchronizers and MCP formulation data paths require "set_false_path" commands. If a clock-prefix naming scheme is used, then wild-cards can be used to easily identify all asynchronous inputs. In Figure 31, to exclude the **`adata`** bus from STA within the **`bClk`** Logic block, first execute the command:

```
set_false_path -from { a* }
```

This command should be sufficient to eliminate all asynchronous inputs from **`bClk`** STA.

7.0 Multi-clock gate-level simulation issues

Digital simulation models typically generate X's when synchronizers recognize setup and hold time violations on CDC signals. This can frequently cause gate-level simulations to fail. What techniques exist to address this problem?

As mentioned in section 6.3.1, signals crossing clock boundaries through a synchronizer will experience setup and hold violations. That is why synchronizers are added to a design, to filter out the metastability effects of a signal that changes too close to the rising edge of a receiving clock domain clock signal.

7.1 Synchronizer gate-level CDC simulation issue

When doing gate-level simulations on a multi-clock design, the ASIC library models of flip-flops are modeled with setup and hold time expressions to match the timing specifications of the actual flip-flops. ASIC libraries typically model flip-flops to drive X's (unknowns) on the flip-flop outputs when a timing violation occurs. When simulating gate-level synchronizers, setup and hold time violations might cause ASIC libraries to issue setup and hold time error messages and the offending signals are frequently driven to an X value. These X-values propagate to the rest of the design causing problems when trying to verify the functionality of the entire gate-level design as shown in Figure 31.

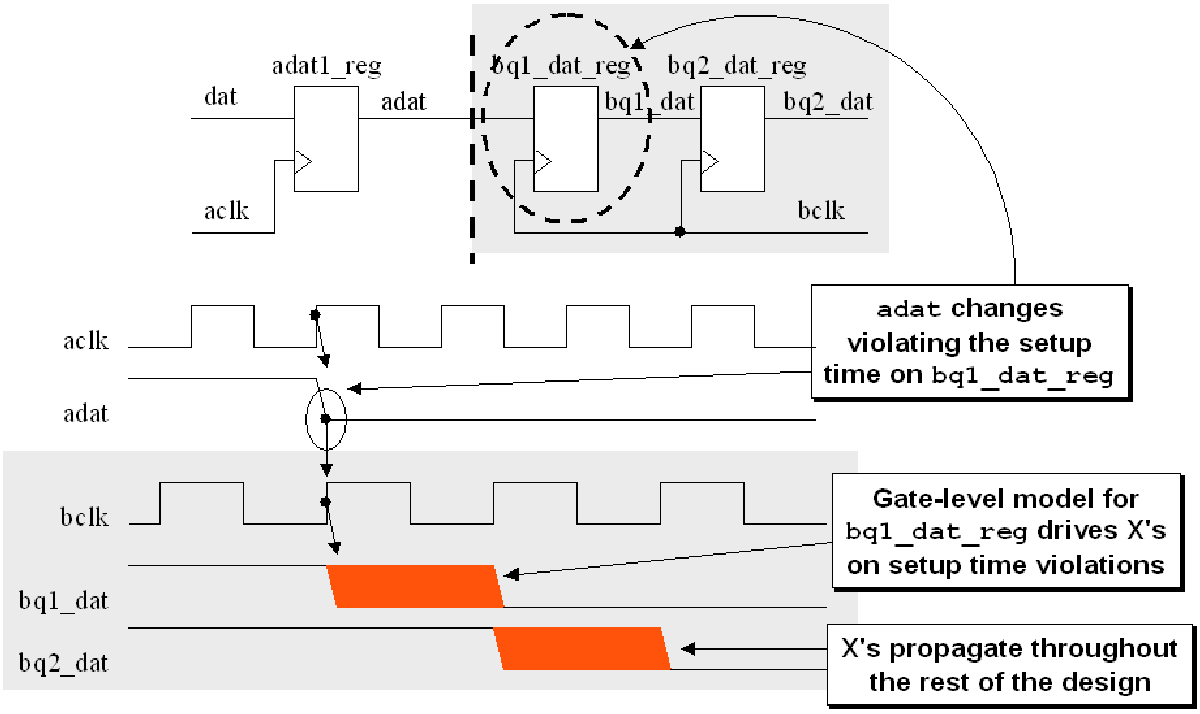

Figure 32 - Synchronizer gate-level CDC simulation waveforms

7.2 Strategies to remove X-propagation from gate-level simulations

There are a number of strategies that have been shared with me by esteemed colleagues over the past 10 years to address issues related to unwanted propagation of X's every time a signal violates a setup or hold time on the first stage of the synchronizer.

Since X-propagation happens when a setup or hold time is violated, almost all of the approaches to address this issue involve changing the setup and hold times to 0 so that there can be no setup or hold time violation, and hence, no X-propagation.

Some of the approaches are bad and others are good. Below are some of the strategies that have been considered to address the X-propagation problem.

7.2.1 Simulator command to turn off timing checks

Most SystemVerilog simulators have a command option to ignore all timing checks, but this would also ignore the desired timing checks for the rest of the design.

7.2.2 Change flip-flop setup and hold times to 0

It is possible to change the setup and hold time setting to zero for any ASIC library flip-flop that is used in a synchronizer, but that would cause all setup and hold time checks of all instances of that same type of flip-flop to be set to zero, including the flip-flops that you might want to use to test the rest of the design.

7.2.3 Copy and modify new flip-flop models

You could make copies of flip-flops from an ASIC library and store them into a new SystemVerilog library with different names, set to zero all setup and hold times, then modify the design gate-level netlist, replacing all first stage synchronizer ASIC library flip-flops with the modified library flip-flops without timing checks, but this could be an error prone and tedious process that might have to be repeated each time a new netlist is generated or it might require the creation of a makefile and scripts to automatically make the modifications each time a new netlist is generated.

7.2.4 Synopsys set_annotated_check command

A useful approach to this problem suggested by Bhatnagar[5] is to use Synopsys commands to modify the SDF backannotation of the setup and hold time on just the first stage flip-flop cells in the design. Bhatnagar points out that the SDF file is instance based and therefore targeting the setup and hold times for the offending cells is more easily accomplished. Bhatnagar notes:

> Instead of manually removing the setup and hold-time constructs from the SDF file, a better way is to zero out the setup and hold-times in the SDF file, only for the violating flops, i.e., replace the existing setup and hold-time numbers with zero's.

Bhatnagar further points out that setup hold times of zero means that there can be no timing violation, therefore no unknowns propagated to the rest of the design. The following dc_shell-t command, given by Bhatnagar, is used to make setup and hold times zero:

```
set_annotated_check 0 -setup -hold -from REG1/CLK -to REG1/D
```

Using a creative naming convention for the output of the first stage flip-flop of a synchronizer might make wild card expressions possible to easily backannotate all first stage flip-flop SDF setup and hold time values to zero using very few dc_shell-t commands.

This technique works if the design is being done with Synopsys DesignCompiler tools, but what about non-Synopsys flows?

7.3 Additional strategies to remove X-propagation

All of the strategies described in sections 7.2 through 7.2.4 were shared in my first multi-clock design paper given in 2001. After the initial presentation a number of engineers came forward to share additional techniques to remove X-propagation from gate level simulations. Engineers from at least three companies described the technique very similar to the description in section ### (kudos to the engineers who attended San Jose SNUG-2001).

Since then, other engineers from many companies have shared additional techniques. Those techniques are described in this section and I am very grateful to all the engineers who continue to share interesting techniques with me each year. Kudos to you all!

7.3.1 Use multiple SDF files

Remember, a key to removing unwanted X-propagation is to force the setup and hold times of the synchronizer inputs to 0, thereby removing all possible setup and hold time violations on the synchronizer inputs.

Many engineers have told me that they actually generate two SDF files. The first SDF file has all of the actual delays, including accurate setup and hold times, for the entire design. Then engineers generate a second SDF file with only the first stage flip-flops included in the file. In this file, the setup and hold times are set to 0. Some engineers build this file by hand and other generated this file using scripts.

Engineers then read in the first SDF file using the $sdf_annotate command. Then they read in the second SDF file, which overwrites the setup and hold times for the data inputs of the first stage synchronizers. When reading in the two SDF files, last SDF file for each instance wins. All timing was accurately annotated, and then the timing checks for the first stage synchronizers were modified.

This is a clever technique that can be used with any tools flow that generates SDF files.

7.3.2 Vendor synchronizer cell with supporting SDF generation tools

Other engineers have told of a great way to approach the X-propagation problem, but the method requires either (a) control over the cell library, or (b) a good working relationship with your ASIC vendor.

This technique requires that a separate synchronizer cell be created with proper placement relationship between the two flip-flop stages. To make this method work, the vendor must provide:
(1) the actual synchronizer cell - these will be instatiated into the design.
(2) the SystemVerilog model for the synchronizer cell for simulation.
(3) SDF file generation tools that will generate an SDF file with 0-setup and 0-hold for the synchronizer cells.

If a vendor can provide this cell and these capabilities, it will only be necessary to generate a single SDF file with proper timing checks for the synchronizer cells.

Any ASIC of FPGA vendor who provides this capability is doing a huge favor for their customer base. I have heard that some ASIC vendors provide this capability. I do not know of any FPGA vendor who provides this capability. Recognizing that most modern designs are multi-clock designs, I strongly urge all ASIC and even all FPGA vendors to provide synchronizer cells with appropriate simulation and SDF file tool support.

7.3.3 Vendors with built-in synchronizer support

If anyone knows of a vendor who offers this support, please let me know who the vendor is with appropriate contact information and I will periodically update this paper to honor vendors who offer this capability to us, the design community.

Vendor list:
(No vendors listed as of this release of this paper)

7.4 Multiple SDF files for gate-level CDC simulations

Immediately after giving my first multi-clock presentation in 2001, engineers from at least three different companies came up after my presentation to share the following great technique to address X-propagation in gate-level simulations.

The technique involved writing out the full SDF timing file and then either manually or by using a script, generate a second SDF file for just the first-stage flip-flops of all synchronizer modules. The second SDF file set all setup and hold times to 0 and then the two SDF files are applied to the design using $sdf_annotate commands. The first SDF file annotates all of the actual timing to the entire design and then the second SDF file is read to over-write the setup and hold times for the first stage synchronizers.

The advantage of this technique is that it can be used for all designs using all tools, not just Synopsys ASIC designs. This is a highly recommended technique.

7.5 Force synchronizer notifier inputs to a fixed value

The built-in timing checks for Verilog and SystemVerilog setup and hold time checks ($setup, $hold, and $setuphold) have an optional notifier output. This notifier output toggles from 0-1-X-Z whenever a timing violation is detected.

Most ASIC and FPGA flip-flop models are built from Verilog User Defined Primitives (UDPs) and the notifier signal is typically listed as one of the inputs to the UDP table. Whenever the notifier input toggles (caused by a timing violation), the flip-flop output goes unknown and that unknown is what is visible on the output of the gate-level flip-flop models. The notifier on these first-stage flip-flop models can be force to a logic level to prevent them from toggling and causing the flip-flop outputs to go unknown during simulation.

One clever technique used by at least one company forces the timing violation notifiers of the first-stage synchronizer flip-flops to be forced to one logic level so they can never toggle and trigger X's into the flip-flop models.

7.6 ASIC & FPGA library cell synchronizers

ASIC and FPGA providers could make CDC design much easier to do if they would provide fully characterized synchronizer cells that could be instantiated into a design. Advanced ASIC vendors provide:
(1) the characterized synchronizer cell.
(2) the Verilog model to simulate the synchronizer cell.
(3) the SDF generator to generate SDF files that annotate the setup and hold time on a synchronizer cell to 0 to avoid the X-generation when a signal violates setup or hold times while crossing CDC boundaries.

I know of no FPGA provider that provides this capability, but a forward thinking FPGA provider would provide such cells for their advanced multi-clock design customers.

7.7 Simulation model with random delay insertion

An interesting model that synthesizes to the correct synchronizer for design, but simulates with random cycle delays has been suggested by multiple colleagues.
The block diagram for the model is shown in Figure 33, and the SystemVerilog code to support this model is shown in Example 6.

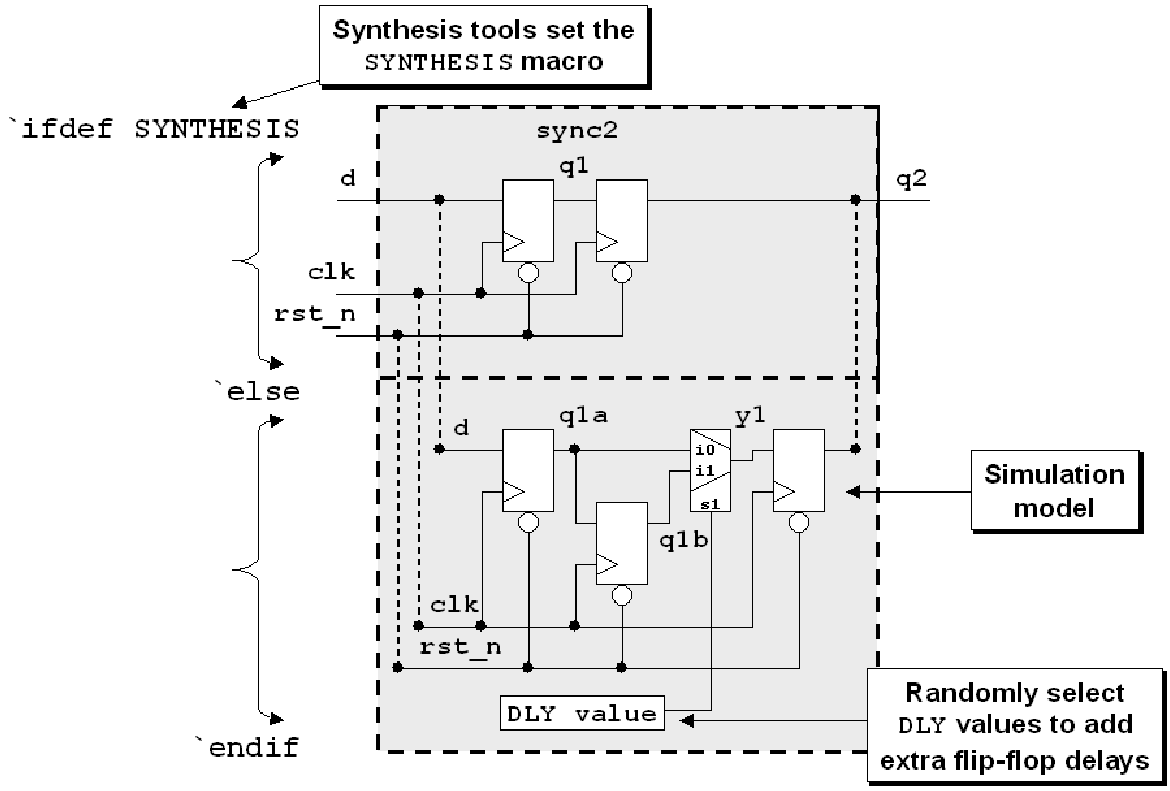

Figure 33 - Sample ASIC & FPGA synchronizer cell for synthesis and simulation

As can be seen in the block diagram, the model is designed to produce either a synthesizble synchronizer model or to be used as a simulation model with selectable delays.

The IEEE Std 1364.1-2002 Verilog RTL Synthesis Standard[6] requires that a compliant synthesis tool set the **`SYNTHESIS`** macro before reading in any Verilog models. Although most synthesis tools have largely ignored many of the requirements of the IEEE Verilog synthesis standard, most tools have implemented this nice **`SYNTHESIS`** macro requirement.

Tools that set the **`SYNTHESIS`** macro before reading this sync2 SystemVerilog code will select the code to infer the two flip-flop synchronizer.

Simulators, which do not set the **`SYNTHESIS`** macro, will read the sync2 model, ignore the code intended for a synthesizable model and will simulate the model in the `else portion of the code.

The model is parameterized so the same model can be used with the default parameter **SIZE** of 1-bit in width for simple 1-bit CDC signals, or the model can be instantiated with the **SIZE** parameter set to a multi-bit width so that the synchronizer can be used to capture and synchronize multi-bit buses such as gray code counters.

```
module sync2 #(parameter SIZE=1)
 (output logic [SIZE-1:0] q2,
  input  logic [SIZE-1:0] d,
  input  logic            clk, rst_n);

  `ifdef SYNTHESIS
  logic [SIZE-1:0] q1;

  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) {q2,q1} <= '0;
    else        {q2,q1} <= {q1,d};

  `else
  logic [SIZE-1:0] y1, q1a, q1b;
  logic [SIZE-1:0] DLY = '0;

  assign y1 = (~DLY & q1a) | (DLY & q1b);

  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) {q2,q1b,q1a} <= '0;
    else        {q2,q1b,q1a} <= {y1,q1a,d};
  `endif
endmodule
```

Example 6 - SystemVerilog model for ASIC & FPGA synchronizer cell

The simulation portion of the model includes a default declaration for a **SIZE**-ed variable called **DLY**. By default the **DLY** variable is initialized to **0**, which causes the entire sync2 model to simulate with the default of two flip-flop delays, but the **DLY** variable can be hierarchically set from the testbench to a reproducible 1's and 0's random value to cause some of the bits in the bus to pass through three flip-flop stages while others pass through only two flip-flop stages. This can model the behavior of a set of synchronizers where some bits are captured on an earlier clock edge than others and allow the simulation to observe how well the design behaves with small skews in the multi-bit data path.

8.0 Summary & conclusions

Clock Domain Crossing (CDC) errors can cause serious design failures. These expensive failures can be avoided by following a few critical guidelines and using well established verification techniques.

8.1 Recommended 1-bit CDC techniques

When passing one bit between clock domains:

- register the signal in the sending clock domain to remove combinational settling.
- synchronize the signal into the receiving clock domain. A Multi-Cycle Path (MCP) formulation may be necessary.

8.2 Recommended multi-bit CDC techniques

When passing multiple control or data signals between clock domains, use one of the following strategies:

- Consolidate - first attempt to combine multiple signals into a 1-bit representation in the sending clock domain before synchronizing the signal into the receiving domain.
- Use Multi-Cycle Path (MCP) formulations to pass multiple signals across clock domains
- Use FIFOs to pass multi-bit buses, either data or control buses.
- Use gray code counters.

8.3 Recommended naming conventions and design partitioning

Use a clock-based naming convention.

As much as possible, partition the design sub-blocks into completely synchronous 1-clock designs.

8.4 Recommended solutions to multi-clock gate-level CDC simulations

There are multiple useful solutions to the CDC X-propagation simulation issues during gate-level simulation:

- Use a Synopsys switch to generate 0-setup and 0-hold times for first stage flip-flops on synchronizers. Works okay with Synopsys tools only.
- Use multiple SDF files - good technique described later in this section.
- Vendor provides a synchronizer cell and appropriate SDF tools - great solution if your ASIC or FPGA vendor provides the models and tools (very few do - ask you ASIC & FPGA vendors to support this feature)
- Use creative SystemVerilog models to model synchronization problems.

The techniques described in this paper were designed to facilitate robust development and verification of multi-clock designs.

9.0 Acknowledgements

My thanks to the hundreds of colleagues and students who have shared interesting multi-asynchronous clock CDC design techniques over the past eight years.

10.0 References

[1] Clifford E. Cummings, “Simulation and Synthesis Techniques for Asynchronous FIFO Design,” SNUG 2002 - www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf

[2] Clifford E. Cummings, “Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs,” SNUG 2001 - www.sunburst-design.com/papers/CummingsSNUG2001SJ_AsyncClk.pdf

[3] Don Mills & Clifford E. Cummings, “RTL Coding Styles That Yield Simulation and Synthesis Mismatches” SNUG 1999 - www.sunburst-design.com/papers/CummingsSNUG1999SJ_SynthMismatch.pdf

[4] Frank Gray, "Pulse Code Communication." United States Patent Number 2,632,058. March 17, 1953.

[5] Himanshu Bhatnagar, Advanced ASIC Chip Synthesis, Second Edition, Kluwer Academic Publishers, 2002.

[6] "IEEE Std. 1364.1 - 2002 IEEE Standard for Verilog Register Transfer Level Synthesis," IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364.1-2002

[7] Mark Litterick, “Pragmatic Simulation-Based Verification of Clock Domain Crossing Signals and Jitter Using SystemVerilog Assertions,” DVCon 2006 www.verilab.com/files/sva_cdc_paper_dvcon2006.pdf

[8] Real Intent, Inc. (white paper), “Clock Domain Crossing Demystified: The Second Generation Solution for CDC Verification,” February 2008 - www.realintent.com

[9] Steve Golson, personal communication

[10] William J. Dally and John W. Poulton, Digital Systems Engineering, Cambridge University Press, 1998

[11] Wikipedia: http://en.wikipedia.org/wiki/Clock_Domain_Crossing_Verification

11.0 Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 26 years of ASIC, FPGA and system design experience and 16 years of SystemVerilog, synthesis and methodology training experience.

Mr. Cummings has presented more than 80 SystemVerilog seminars and training classes in the past five years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Mr. Cummings has participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee, and has presented more than 40 papers on SystemVerilog & SystemVerilog related design, synthesis and verification techniques.

Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

Sunburst Design, Inc. offers World Class Verilog & SystemVerilog training courses. For more information, visit the www.sunburst-design.com web site.
Email address: cliffc@sunburst-design.com

An updated version of this paper can be downloaded from the web site: www.sunburst-design.com/papers
(Last updated September 26, 2008)

12.0 Appendix

This appendix includes the source code for the MCP formulation with acknowledge feedback and the 1-deep, 2-register FIFO synchronizer.

12.1 Common sync2 model - used by MCP formulation and FIFO synchronizer

The **sync2** model is common to both the MCP formulation with ready-acknowledge design (source code in section 12.2) and the multi-bit 1-deep / 2-register FIFO synchronizer (source code in section 12.3).

```
// sync signal to different clock domain
module sync2 (
  output logic q,
  input  logic d, clk, rst_n);

  logic q1; // 1st stage ff output

  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) {q,q1} <= '0;
    else        {q,q1} <= {q1,d};
endmodule
```

Example 7 - sync2.sv code

12.2 MCP formulation with ready-acknowledge source code

This model requires the **sync2** model shown in section 12.1.

```
// Pulse Generator
module plsgen (
  output logic pulse, q,
  input  logic d,
  input  logic clk, rst_n);

  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) q <= '0;
    else        q <= d;

  assign pulse = q ^ d;
endmodule
```

Example 8 - plsgen.sv code

```
module asend_fsm (
  output logic aready,  // ready to send next data
  input  logic asend,   // send adata
  input  logic aack,    // acknowledge receipt of adata
  input  logic aclk, arst_n);

  enum logic {READY = '1,
              BUSY  = '0} state, next;

  always_ff @(posedge aclk or negedge arst_n)
    if (!arst_n) state <= READY;
    else         state <= next;

  always_comb begin
    case (state)
      READY: if (asend) next = BUSY;
             else       next = READY;
      BUSY : if (aack)  next = READY;
             else       next = BUSY;
    endcase
  end

  assign aready = state;
endmodule
```

Example 9 - asend_fsm.sv code

```
module back_fsm (
  output logic bvalid,  // data valid / ready to load
  input  logic bload,   // load data / send acknowledge
  input  logic b_en,    // enable receipt of adata
  input  logic bclk, brst_n);

  enum logic {READY = '1,
              WAIT  = '0} state, next;

  always_ff @(posedge bclk or negedge brst_n)
    if (!brst_n) state <= WAIT;
    else         state <= next;

  always_comb begin
    case (state)
      READY: if (bload) next = WAIT;
             else       next = READY;
      WAIT : if (b_en)  next = READY;
             else       next = WAIT;
    endcase
  end

  assign bvalid = state;
endmodule
```

Example 10 - back_fsm.sv code

```
module bmcp_recv (
  output logic [7:0] bdata,
  output logic       bvalid, // bdata valid
  output logic       b_ack,  // acknowledge signal
  input  logic [7:0] adata,  // unsynchronized adata
  input  logic       bload,  // load data and acknowledge receipt
  input  logic       bq2_en, // synchornized enable input
  input  logic       bclk, brst_n);

  logic  b_en;  // enable pulse from pulse generator

  // Pulse Generator
  plsgen       pg1 (.pulse(b_en), .q(), .d(bq2_en),
                    .clk(bclk), .rst_n(brst_n), .*);

  // data ready/acknowledge FSM
  back_fsm fsm (.*);

  // load next data word
  assign bload_data = bvalid & bload;

  // toggle-flop controlled by bload_data
  always_ff @(posedge bclk or negedge brst_n)
    if      (   !brst_n) b_ack <= '0;
    else if (bload_data) b_ack <= ~b_ack;

  always_ff @(posedge bclk or negedge brst_n)
    if      (   !brst_n) bdata <= '0;
    else if (bload_data) bdata <= adata;
endmodule
```

Example 11 - bmcp_recv.sv code

```
module mcp_blk #(parameter type dat_t = logic [7:0]) (
  output logic       aready,  // ready to receive next data
  input  logic [7:0] adatain,
  input  logic       asend,
  input  logic       aclk, arst_n,

  output logic [7:0] bdata,
  output logic       bvalid,  // bdata valid (ready)
  input  logic       bload,
  input  logic       bclk, brst_n);

  logic [7:0] adata;   // internal data bus

  logic       b_ack;   // acknowledge enable signal
  logic       a_en;    // control enable signal
  logic       bq2_en;  // control  - sync output
  logic       aq2_ack; // feedback - sync output

  sync2     async  (.q(aq2_ack), .d(b_ack), .clk(aclk), .rst_n(arst_n));
  sync2     bsync  (.q(bq2_en),  .d(a_en),  .clk(bclk), .rst_n(brst_n));
  amcp_send alogic (.*);
  bmcp_recv blogic (.*);
endmodule
```

Example 12 - mcp_blk.sv code

```
module amcp_send (
  output logic [7:0] adata,
  output logic       a_en, aready,
  input  logic [7:0] adatain,
  input  logic       asend,
  input  logic       aq2_ack,
  input  logic       aclk, arst_n);

  logic  aack; // acknowledge pulse from pulse generator

  // Pulse Generator
  plsgen       pg1 (.pulse(aack), .q(), .d(aq2_ack),
                    .clk(aclk), .rst_n(arst_n));

  // data ready/acknowledge FSM
  asend_fsm fsm (.*);

  // send next data word
  assign anxt_data = aready & asend;

  // toggle-flop controlled by anxt_data
  always_ff @(posedge aclk or negedge arst_n)
    if      (  !arst_n) a_en <= '0;
    else if (anxt_data) a_en <= ~a_en;

  always_ff @(posedge aclk or negedge arst_n)
    if      (  !arst_n) adata <= '0;
    else if (anxt_data) adata <= adatain;
endmodule
```

Example 13 - acmp_send.sv code

12.3Multi-bit 1-deep / 2-register FIFO synchronizer source code

This model requires the **sync2** model shown in section 12.1.

```
module wctl (
  output logic wrdy, wptr, we,
  input  logic wput, wq2_rptr,
  input  logic wclk, wrst_n);

  assign we    = wrdy & wput;
  assign wrdy  = ~(wq2_rptr ^ wptr);

  always_ff @(posedge wclk or negedge wrst_n)
    if (!wrst_n) wptr <= '0;
    else         wptr <= wptr ^ we;
endmodule
```

Example 14 - wctl.sv code

```
`timescale 1ns/1ns
module cdc_syncfifo #(parameter type dat_t = logic [7:0]) (
  // Write clk interface
  input  dat_t wdata,
  output logic wrdy,
  input  logic wput,
  input  logic wclk, wrst_n,
  // Read clk interface
  output dat_t rdata,
  output logic rrdy,
  input  logic rget,
  input  logic rclk, rrst_n);

  logic wptr, we, wq2_rptr;
  logic rptr, rq2_wptr;

  wctl  wctl  (.*);

  rctl  rctl  (.*);

  sync2 w2r_sync (.q(rq2_wptr), .d(wptr), .clk(rclk), .rst_n(rrst_n));

  sync2 r2w_sync (.q(wq2_rptr), .d(rptr), .clk(wclk), .rst_n(wrst_n));

  // dual-port 2-deep ram
  dp_ram2 #(dat_t) dpram (.q(rdata), .d(wdata),
                          .waddr(wptr), .raddr(rptr),
                          .we(we), .clk(wclk), .*);
endmodule
```

Example 15 - cdc_syncfifo.sv code

```
// dual-port 2-deep ram
```

```
module dp_ram2 #(parameter type dat_t = logic [7:0])
 (output dat_t q,
  input  dat_t d,
  input  logic waddr, raddr, we, clk);

  dat_t mem [0:1];

  always_ff @(posedge clk)
    if (we) mem[waddr] <= d;

  assign q = mem[raddr];
endmodule
```

Example 16 - Dual Port Ram code - dp_ram2.sv

```
module rctl (
  output logic rrdy, rptr,
  input  logic rget,rq2_wptr,
  input  logic rclk, rrst_n);

  typedef enum {xxx, VALID} status_e;
  status_e status;

  assign status = status_e'(rrdy);

  assign rinc  = rrdy & rget;
  assign rrdy  = (rq2_wptr ^ rptr);

  always_ff @(posedge rclk or negedge rrst_n)
    if   (!rrst_n) rptr <= '0;
    else           rptr <= rptr ^ rinc;
endmodule
```

Example 17 - rctl.sv code