Pragmatic Simulation-Based Verification of Clock Domain Crossing Signals and Jitter using SystemVerilog Assertions

Mark Litterick, Verilab, Munich, Germany. (mark.litterick@verilab.com)

Abstract

Recent advances in automated formal solutions for verification of clock domain crossing signals go far towards reducing the risk of clock related defects in multi-clock system-on-chip devices. However the vast majority of multiple clock-domain devices still utilize a flow which does not involve these specialized tools or formal verification techniques. This paper presents a pragmatic alternative methodology, using SystemVerilog Assertions in a simulation-based verification flow, to validate the correct operation and use of synchronizers while emulating the effects of CDC jitter in order to stress the functional operation of the rest of the device.

1 Introduction

Integrated circuits with multiple clocks require special verification techniques to ensure that Clock Domain Crossing (CDC) signals, which cross from one clock domain to another, are handled correctly. The issues are related to analogue effects in the real-world transistor-level circuits and do not typically manifest themselves in standard RTL simulation flows. The problems are well documented in [2][6][7], and include:

- metastability in flip-flops where the setup and hold-times are violated
- uncertainty and jitter related to metastability
- functional errors due to convergence of synchronized signals
- functional errors due to divergence through multiple synchronizers

Recent advances in automated formal solutions [3][4][5][6] go far towards reducing the risk of clock related defects and currently represent the best-in-class methodology. However, the vast majority of multi-clock FPGA and ASIC devices use a tool flow which uses neither formal verification nor dedicated CDC tools. This paper presents a pragmatic alternative Assertion-Based Verification (ABV) methodology which can be used in a simulation based flow to improve the quality of CDC verification and minimize the associated risks. By analyzing the CDC issues and considerations in detail, and presenting SystemVerilog Assertions (SVA) [1] for the necessary checks, the paper provides useful background information and should help demystify the state-of-the-art automated formal methodologies.

The basic steps for CDC analysis and checking are the same irrespective of toolset implementation:

1. structural analysis to identify CDC signals
2. classification of CDC signal types
3. determination of appropriate properties
4. validation of the property assertions

This paper has a look at each of these steps in turn and then considers the issues related to validating correct functional behavior of the device in the presence of jitter on the CDC signals.

2 Structural Analysis

Whichever verification methodology is adopted it is necessary to perform structural analysis to identify all of the signals that cross clock domain boundaries. Structural analysis can be performed by various approaches including:

- commercial automated CDC tools
- script based in-house tools
- manual inspection and code reviews

The more automated the solution the better; however some degree of manual intervention may be required to classify the CDC signals. Structural analysis should be performed on both the RTL implementation and also on the post-synthesis gate-level netlist, and the results compared.

For flows that do not use automated CDC tools it is recommended that RTL designers instantiate CDC synchronization modules, rather than infer them from distributed RTL code. This approach has the following advantages:

- ensures metastable nets are not abused
- assertion of properties is better encapsulated
- eases error prone manual structural analysis
- allows better emulation of CDC jitter

3 Classification of CDC Signal Types

Once all the CDC signals have been identified they can be classified into different types, depending on design intent, for example:

- static signal (only changes during configuration and not during operation)
- regular signal needing synchronization (e.g. interrupt request)
- level or pulse based handshake interface (e.g. request-acknowledge)
- asynchronous FIFO

For the simple cases the automatic structural analysis tools may be able to determine the signal type from the context: although manual intervention is typically required to discriminate between static signals and CDC signals that have missing synchronization circuits.

4 Property Definitions

Just having a synchronization circuit connected is only part of the solution: the associated logic must interact correctly with the synchronization circuit to ensure valid operation. The ABV approach allows assertions to be specified that check correct functionality of the synchronization circuits as well as validating correct use of the synchronizers by the environment in which they are instantiated. A structural approach to CDC design allows these properties to be specified once and automatically attaches them to all instances of the corresponding synchronization building blocks. Typical property checks for common CDC synchronization circuit elements are listed in the following sections with some example coding using SVA.

4.1 Checks for Static Signals

If a signal is classified as static then it means it does not require any special synchronization circuit since it is guaranteed not to change during normal functionality of the device. A signal that changes very infrequently violates this rule and cannot be regarded as static. In order to ensure that static signals are used correctly throughout the entire regression run the following property should be asserted:

- value must not change during run mode

Note that this property cannot normally be validated formally, since it is really a high level protocol requirement on the operation of the device (although some implementations would physically disallow signal changes during run mode, which could be checked formally). **Figure 1** provides example SVA code for such a property and demonstrates assertion of this property on all static signals (*x, y, z*) related to a single *run* signal.

Note that reset operation, assertion labels and error messages have been omitted from the SVA shown in this paper to simplify the code examples. Refer to [1] for a detailed explanation of the SVA syntax and coding guidelines.

```
property p_static(clk, run, data);
  @(posedge clk)
    !$stable(data) |-> !run;
endproperty : p_static

assert property (p_static(clk_a, run_a, {x, y, z}));
```

Figure 1: SVA for Static Signals

4.2 Checks for 2-Flip-Flop Synchronizer

The 2-flip-flop synchronizer, shown in **Figure 2**, is the basic building block for most synchronization circuits. Operation of this circuit, including failure mode analysis, is well documented in [7] and associated papers.

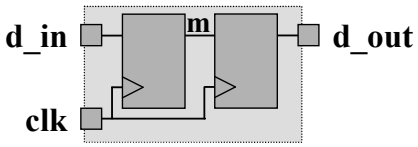

Figure 2: 2-Flip-Flop Synchronizer

If no assumptions are made about the relationship between the source and destination clock domains, then the following property ensures that all input signal values can be reliably transported to the synchronizer output:

- input data values must be stable for three destination clock edges

There are three specific cases where input values might not get transported to the synchronizer output; values that are:

- never sampled by the destination clock
- sampled by only one clock edge
- sampled by only two successive clocks

The timing diagram shown in **Figure 3** illustrates potential filtering due to metastability in each case. The first high pulse on *d_in* is never sampled by the clock and it is not propagated during simulation; note however that if the rising edge of this pulse violated the previous hold-time, or the falling edge violated the current setup-time, then this value might get propagated in real life. The second pulse on *d_in* is sampled by the destination clock; but if the signal changes just before the clock edge (a setup-time violation) then the simulation will propagate the new value, but the real circuit might become metastable and decay to original low value. The third pulse on *d_in* illustrates the case where even though the signal is sampled by two successive clock edges, it can still

be filtered by the synchronizer. In this case the rising edge of *d_in* violates the setup-time for the first clock edge (and decays to previous value; low) and the falling edge violates the hold-time for the next clock edge (and decays to the new value; low); under these circumstances the simulation will propagate a 2-clock wide pulse but the real circuit may filter it out completely.

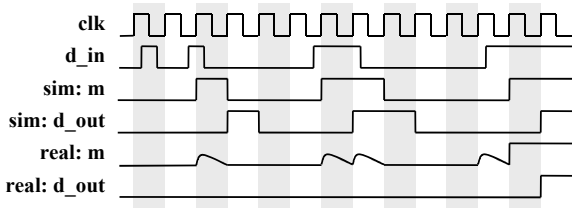

Figure 3: Metastability Filtering

Example SVA coding for properties which cover all cases is shown in **Figure 4**. The *p_no_glitch* property states that when the *d_in* value changes, it must have the same value when sampled at the next rising edge of *clk*. Note that since *@(d_in)* is being used as the first *EventExpression* in *p_no_glitch* and the SVA scheduling semantics mean that the property samples signals in the *preponed* region (just prior to the clock event) the local *data* variable has to be assigned the inverse of *d_in*. For this to work reliably, the *d_in* value must also be checked for unknown (*X* or *Z*) values using *$isunknown* (not shown). The *p_no_glitch* assertion starts a different thread for each signal transition; hence when there are two transitions between clock edges the thread for the trailing edge passes, whereas the thread for the leading edge fails.

```
property p_stability;
  @(posedge clk)
    !$stable(d_in) |=> $stable(d_in) [*2];
endproperty : p_stability

property p_no_glitch;
  logic data;
  @(d_in)
    (1, data = !d_in) |=>
  @(posedge clk)
    (d_in == data);
endproperty : p_no_glitch

assert property(p_stability);
assert property(p_no_glitch);
```

Figure 4: SVA for 2-Flip-Flop Synchronizer

4.3 Checks for Handshake Synchronizers

There are many different types of handshake synchronizers in use but most come down to the same fundamental working principle of synchronizing a single-bit request into the destination clock domain and waiting for a synchronized version of the acknowledge to come back. The differences come when you consider the higher level protocols for the associated interfaces, data management and how the logic is partitioned. This abundance of different implementations makes the job of fully automating the protocol checking of handshake interfaces difficult. Analysis of level-based request-acknowledge full-handshake protocols is provided by [4][5][6].

An alternative approach is to isolate the CDC related aspects of the protocol to synchronization circuits which do not form part of the communicating modules. A popular implementation is to convert pulses from the source clock domain into pulses in the destination clock domain using a pulse synchronizer like the one shown in **Figure 5**.

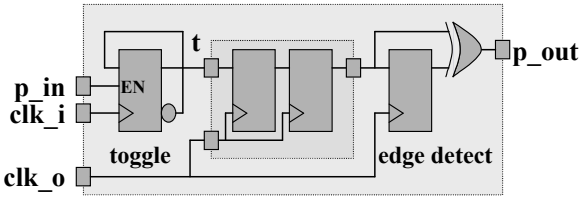

Figure 5: Pulse Synchronizer

In this case the data transfer is managed using a pulse-based handshake synchronizer as shown in **Figure 6**. The source requests a transfer of the bundled data by asserting *req_o* high for a single clock; this request is synchronized into the destination domain as a *req_i* pulse, which is used to latch the data. The destination responds by generating an acknowledge pulse, *ack_o*, which is synchronized back into the source domain to signal the end of the transfer.

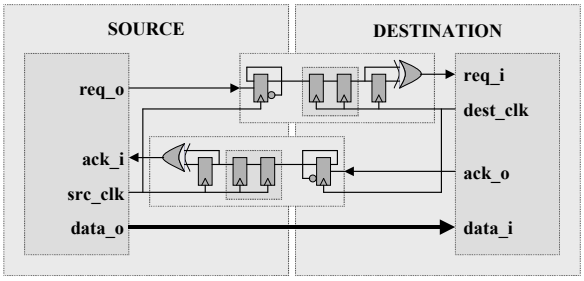

Figure 6: Pulse-Based Handshake

This solution is popular for the following reasons:

- source and destination modules use a simple pulse-based request-acknowledge protocol
- interface protocol can be used throughout device for all inter-module interfaces
- CDC logic encapsulated external to module
- good reuse strategy since the synchronization layer could be removed completely if the modules are instantiated in the same clock domain in another device

It is possible to define a generic handshake checker that can be instantiated at each module interface to check such things as the following:

- every request gets acknowledged
- no acknowledge without a request
- data stability

Some example SVA coding for such a request-acknowledge interface protocol is given in **Figure 7**. The sequence *s_transfer* is used to check that every *req* gets an *ack* (*p_req_gets_ack* will fail if two *req*s occur without an intermediate *ack*; since a valid *s_transfer* must start on every *req*) and that every *ack* was preceded by a *req* (*p_ack_had_req* will fail if two *ack*s occur without an intermediate *req*; since every *ack* must be the endpoint of a valid *s_transfer*). The *p_data_stable* property states that *data* must be stable from *req* up to and including the next *ack*.

```
sequence s_transfer;
  @(posedge clk)
    req ##1 !req [*1:max] ##0 ack;
endsequence : s_transfer

property p_req_gets_ack;
  @(posedge clk)
    req |-> s_transfer;
endproperty : p_req_gets_ack

property p_ack_had_req;
  @(posedge clk)
    ack |-> s_transfer.ended;
endproperty : p_ack_had_req

property p_data_stable;
  @(posedge clk)
    req |=> $stable(data) [*1:max] ##0 ack;
endproperty : p_data_stable

assert property(p_req_gets_ack);
assert property(p_ack_had_req);
assert property(p_data_stable);
```

Figure 7: SVA for Req-Ack Protocol

In addition to validating the request-acknowledge protocol at each module interface, the following properties can be asserted for the pulse synchronizer:

- *p_in* must be 1-source-clock wide
- *p_out* must be 1-destination-clock wide
- every *p_in* results in a *p_out*
- *p_in* separation is checked by the *p_stability* property (from 2FF-sync) on toggle net, *t*

Figure 8 provides some example SVA code for the property definitions related to the pulse synchronizer. Note that the range specified in *p_in_out* allows correct operation with CDC jitter emulation which is explained in Section 6.

```
property p_single_pulse(clk,pulse);
  @(posedge clk)
    pulse |=> !pulse;
endproperty : p_single_pulse

property p_in_out;
  @(posedge clk_i)
    p_in |=>
  @(posedge clk_o)
    1 ##[1:3] p_out;
endproperty : p_in_out

assert property (p_single_pulse(clk_i,p_in));
assert property (p_single_pulse(clk_o,p_out));
assert property (p_in_out);
```

Figure 8: SVA for Pulse Synchronizer

4.4 Checks for Asynchronous FIFOs

Another common CDC synchronization circuit, which is used when the high latency of the handshake protocols cannot be tolerated, is the dual-clock asynchronous FIFO as shown in **Figure 9**. The design principles behind such circuits are well explained in [8]. Although many implementation variations exist, the basic operation is the same; data is written into the FIFO from the source clock domain and read from the FIFO in the destination clock domain. Gray-coded read and write pointers are passed into the alternate clock domain to generate full and empty status flags.

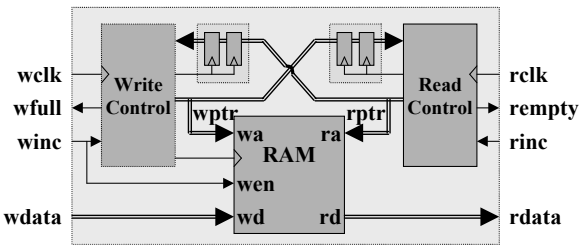

Figure 9: Dual-Clock Asynchronous FIFO

The following properties can be asserted for the dual-clock asynchronous FIFO:

- never write when full
- never read when empty
- pointers must be gray-coded at source
- data integrity (values and order)

The SVA code shown in **Figure 10** provides a possible implementation of these checks, using the signal names from [8]. The *p_data_integrity* assertion starts a new thread with a unique *cnt* value whenever *wdata* is written to the FIFO and checks the *rdata* value against the local *data* variable whenever the corresponding read occurs. The *first_match* is required in *p_data_integrity* to ensure the property checks *rdata* on the first occurrence of a read with the corresponding *cnt*, otherwise it waits for ever for a *rdata* match at the *rcnt* value.

```
property p_bad_access(clk,inc,flag);
  @(posedge clk)
    inc |-> !flag;
endproperty : p_bad_access

property p_gray_code(clk,rst_n,data);
  @(posedge clk) disable iff (!rst_n)
   !$stable(data) |-> $onehot(data ^ $past(data));
endproperty : p_gray_coded

int wcnt, rcnt;
always @(posedge wclk or negedge wrst_n)
    if (!wrst_n) wcnt = 0;
    else if (winc)  wcnt = wcnt + 1;
always @(posedge rclk or negedge rrst_n)
    if (!rrst_n) rcnt = 0;
    else if (rinc)  rcnt = rcnt + 1;

property p_data_integrity
  int cnt; logic [DSIZE-1:0] data;
  disable iff (!wrst_n || !rrst_n)
  @(posedge wclk)
      (winc, cnt = wcnt, data = wdata) |=>
  @(posedge rclk)
    first_match(##[0:$] (rinc && (rcnt == cnt)))
    ##0 (rdata == data);
endproperty : p_data_integrity

assert property(p_bad_access(wclk,winc,wfull));
assert property(p_bad_access(rclk,rinc,rempty));
assert property(p_gray_code(wclk,wrst_n,wptr));
assert property(p_gray_code(rclk,rrst_n,rptr));
assert property (p_data_integrity);
```

Figure 10: SVA for Asynchronous FIFO

5 Assertion Validation

The properties asserted on the CDC synchronization circuits and associated interfaces can be validated either dynamically (by simulation) or statically (using a formal property checking tool). However, the property definitions provided in this paper are aimed at enhancing simulation-based verification and some of them are not suitable for formal verification, for example *p_data_integrity* uses an unbounded range.

6 CDC Jitter Emulation

One of the main functional problems that can arise from synchronization is called CDC jitter [3]. Even when all signals are properly synchronized into the destination clock domain, the actual arrival time in real life is subject to uncertainty if the signal goes metastable in the first stage synchronizing flip-flop. This CDC jitter can lead to functional failures in the destination logic, particularly where there is convergence of signals from different synchronization elements or divergence of a single CDC signal through more than one synchronizer (and subsequent re-convergence of seemingly unrelated signals). Normally this jitter would not show up on a simulation but must be considered to conclude the analysis.

One possible method to emulate most of the effects of CDC jitter is to replace the 2-flip-flop synchronizer circuit with an alternative implementation which randomly adjusts the output signal edge transitions; an example of a possible implementation is shown in **Figure 11** with corresponding SystemVerilog code shown in **Figure 12**. This circuit represents an improvement over the implementations suggested in [9] (since it can emulate negative jitter) and [3] (since it is completely encapsulated in the destination clock domain - making it more suitable for the structural CDC methodology discussed in this paper).

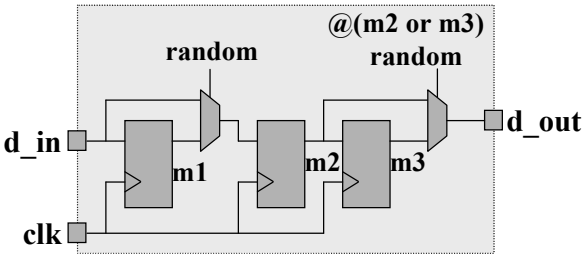

Figure 11: Synchronizer with Jitter Emulation

```
always @(posedge clk) begin
  randcase
    1: m2 <= $past(d_in);
    1: m2 <= d_in;
  endcase
  m3 <= m2;
end
always @(m2 or m3)
  randcase
    1: d_out = m2;
    1: d_out = m3;
  endcase
```

Figure 12: SystemVerilog for Jitter Emulation

The timing diagram of **Figure 13** demonstrates the effect of CDC jitter emulation on the output of the 2-flip-flop synchronizer circuit during RTL simulations. Note that the jitter emulation can cause the *d_out* transition to randomly appear one clock early (representing a hold-time violation on previous clock), at the nominal position, or one clock late (representing a setup-time violation on current clock) relative to the normal position during RTL simulation without jitter emulation. This circuit is also capable of filtering one and two-clock wide pulses.

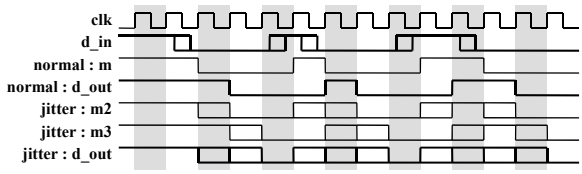

Figure 13: CDC Jitter Timing Diagram

Both formal and dynamic verification methodologies can be used to determine if the design still works correctly with CDC jitter emulation. In both cases ABV can be used to instrument the areas of convergence, identified by structural analysis, with appropriate additional assertions since the jitter results may not be readily visible to higher-level checkers in the verification environment. Note that formal techniques can prove the design properties never get violated as a result of all possible jitter conditions, whereas simulation can only demonstrate that with a particular jitter combination no assertions or HVL checks were violated. Actual violations can also be difficult to debug in simulation compared to formal methods. Nonetheless, over the complete regression run, CDC jitter emulation still provides an improvement in overall verification quality with simulation-based flows.

7 Conclusion

This paper presented a pragmatic assertion-based methodology that can be used with simulation-based flows to improve the quality and confidence in verification of clock domain crossing signals. The methodology has been successfully applied to many projects to help isolate CDC implementation, usage and jitter related defects. It is proposed as an alternative to automated formal verification solutions because it is more accessible to the vast majority of teams currently implementing multi-clock designs targeted at FPGA and ASIC devices. The SVA code examples and methodology make no assumptions about the relationships between the clock domains, enabling more reliable reuse at the expense of stricter design rules. Careful partitioning and isolation of the CDC structural code is key to minimizing the risk and enables a few important properties to be specified and asserted on blocks that are instantiated where required.

8 References

[1] Accellera, *SystemVerilog 3.1a Language Reference Manual*, http://www.accellera.org

[2] Cadence, *Clock Domain Crossing: closing the loop on clock domain functional implementation problems*, White Paper

[3] Tai Ly, Neil Hand & Chris Ka-kei Kwok, *Formally Verifying Clock Domain Crossing Jitter Using Assertion-Based Verification*, DVCon 2004

[4] Chris Ka-kei Kwok, Vijay Vardhan Gupta & Tai Ly, *Using Assertion-Based Verification to Verify Clock Domain Crossing Signals*, DVCon 2003

[5] Tsachy Kapschitz, Ran Ginosar & Richard Newton, *Verifying Synchronization in Multi-Clock Domain SoC*, DVCon 2004

[6] Tsachy Kapshitz & Ran Ginosar, *Formal Verification of Synchronizers*, CHARME 2005

[7] Ran Ginosar, *Fourteen Ways to Fool Your Synchronizer*, ASYNC 2003

[8] Cliff Cummings, *Simulation and Synthesis Techniques for Asynchronous FIFO Design*, SNUG San Jose 2002

[9] Janick Bergeron, *Writing Testbenches, Functional Verification of HDL Models*, Kluwer Academic Publishers, 2003