WHITE PAPER

Resets and Reset Domain Crossings in ASIC and FPGA designs

Revision 1.0

2018-04-03

Alexander Gnusin

[Level: Advanced]

©[2018] Aldec, Inc.

Abstract

This white paper explains Reset-related ASIC and FPGA design issues as well as outlines commonly-used design techniques leading to safe reset implementations. It goes on to explain about Reset Domain Crossing effects and methods to mitigate their influence on design. LINT tools provide valuable help for designers in Resets and Reset Domain Crossings verification.

Table of Contents

Resets and Reset Domain Crossings in ASIC and FPGA designs 1
Table of Contents 2
Resets Implementation in ASIC and FPGA designs 3
Synchronous and Asynchronous Resets 3
Asynchronous Resets 4
Synchronous Resets 5
The Power-on-Reset in FPGA designs 5
Reset Synchronization Techniques 6
Asynchronous Reset Synchronization Techniques 7
Synchronous Reset Synchronization Techniques 7
Reset Synchronizers for Multiple-Clock Designs 8
Reset Techniques for Designs with Non-Resettable Storage Elements 9
Reset Domain Crossings 10
About Aldec, Inc. 12

Resets Implementation in ASIC and FPGA designs

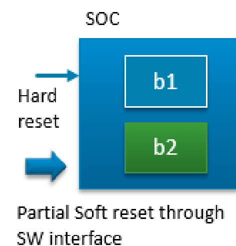
In ASIC or FPGA devices, a reset acts as a synchronization signal that sets all the storage elements to a known state. Designers normally implement a global reset as an external pin to initialize the design on power-up as well as re-initialize it later as needed. For example, the reset may be initiated by:

- Press switch: initiated by user request
- Power-on-Reset: initiated by power supply, signaling the "power ready" condition.
- Microprocessor, signaling system re-initialization caused by excessive interconnect noise

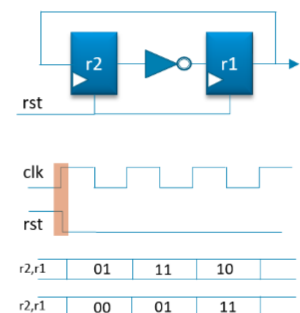
The power-on-reset (PoR) or press switch circuitry may use a Schmitt trigger to de-assert the reset signal cleanly once the rising voltage of the RC network passes the threshold voltage of the Schmitt trigger. The software-based reset techniques may use input ports or software interface to initiate the reset action.

Bringing a design into a known state ensures functional repeatability of the digital design. As a predictable synchronous automation, the system should operate exactly the same after the reset operation is followed by similar sequence of input stimulus.

The commonly-used convention is the differentiation between the Hard and Soft resets. Hard resets are initiated by other hardware components (power supply or press switch), while Soft resets are initiated by software-driven requests such as entering power-saving mode or re-booting the system due to a non-recoverable error event.



Normally, Hard resets are asserted and de-asserted asynchronously, while Soft resets may use synchronous operations. Also, Hard resets will affect the whole device, while Soft resets may reset only a portion of design while leaving the rest of design circuitry undisturbed:



Asynchronous reset operation may be a big issue for predictable and repeatable design initialization. In synchronous designs, asynchronous reset de-assertion causes metastability issues as well as unpredictable initialization values of memory elements. Reset signal de-assertions may be delivered to design storage elements in different clock cycles or during the setup/hold storage element region, and there is no predictability of the design. The following picture provides an example of an unpredictably-initialized design: In this example, the reset signal de-asserts during the active clock edge change, causing metastability issues as well as randomly initialized register values.

To avoid non-determinism in designs, reset signals should be always synchronized at de-assertion and design storage elements should receive reset de-assertion. Usually, this synchronization is implemented within the design.

Synchronous and Asynchronous Resets

There exist two basic Reset implementations: Synchronous and Asynchronous. These implementations describe internal-to-device reset, connecting directly to storage elements reset pins. Despite their

ALDEC THE DESIGN VERIFICATION COMPANY

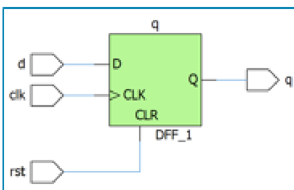
names, both resets are synchronous to the design clock in reset de-assertion. The only difference is reset assertion edge synchronization type, which is *clock-synchronous* in synchronous resets and *clock-asynchronous* in asynchronous resets.

Reset type selection is one of the main implementation considerations in digital logic. Both Asynchronous and Synchronous resets have their own advantages and drawbacks, which are discussed later in the paper.

Asynchronous Resets

Asynchronous resets are asynchronous only for the reset assertion. Reset de-assertion is still synchronous to internal design clock and the reset tree has to be taken into account during timing analysis. The following code demonstrates asynchronous reset implementation:

Schematics:



VHDL :

```
architecture rtl of rst_async is
  begin
    process (clk, rst) begin
      if (rst = '1') then
        q <= '0';
      elsif rising_edge(clk) then
        q <= d;
      end if;
    end process;
end rtl;
```

Verilog :

```
module rst_async (input clk, rst, d,
                  output reg q);

  always @(posedge clk or posedge rst)
    if (rst) q <= 1'b0;
    else     q <= d;

endmodule
```

As seen in the example, sequential processes contain reset in the sensitivity list. The reset condition is implemented at the top of the process, so reset-driven data change precedes clock-driven data change. This condition matches the implementation of an asynchronous reset in storage elements, by implementing the same preference functionality.

The advantages of asynchronous resets include:

- Resets are being implemented separately from the datapath, leaving datapath logic intact. This feature leads to better timing closure and smaller gate count, which reduces area and power.
- Resets may operate with or without clock and have preference over clock-driven storage element data changes. This enables reset operation with or without clocking, allowing designers to reset designs with non-locked clock generators.
- In FPGA, asynchronous reset nets could be mapped to global nets, allowing high-fanout connection of reset signal to thousands of storage elements as well as timing closure on reset de-assertion.

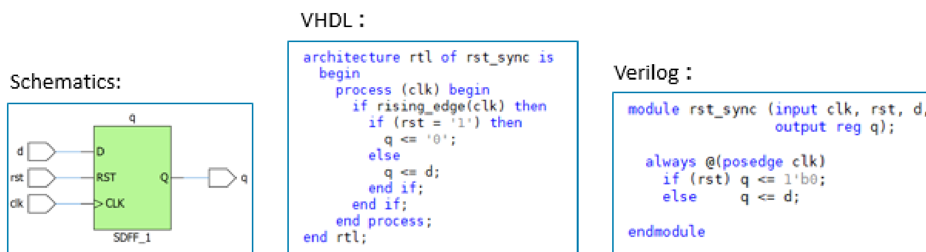
The disadvantages of asynchronous resets include:

- Although named “asynchronous”, the reset signal still requires synchronization and timing closure for the reset tree. The reset tree must be properly balanced to supply reset de-assertion on the same clock cycle. This may be problematic for the high-speed designs.
- Asynchronous resets may require a reset glitch filter. Reset pulses wide enough to meet the minimum reset pulse width for a flip-flop will cause the flip-flop to reset. So, an asynchronous reset glitch may reset the design and the reset glitch filter may be required to filter asynchronous reset pulses less than pre-defined length.

- Asynchronous resets introduce asynchronous events into the design. In the case of partial asynchronous reset, these asynchronous events may influence on operating logic, introducing metastability issues and non-determinism into the design functional operation.

Synchronous Resets

Synchronous resets are the other commonly-used reset implementation type. Here, both reset assertion and reset de-assertion edges are synchronous. The following example represents synchronous reset implementation:



In most cases, storage elements such as Flip-Flops do not implement special synchronous reset input. Also, reset has the same priority as incoming data relative to the clock active edge. It is important to note that synchronous reset is unable to operate without clock or during a PLL clock training period. Synchronous resets are treated like any other clock-synchronous data signals, and timing analysis has to be performed on synchronous reset like any other design signals.

The advantages of synchronous resets include:

- Synchronous resets enable development of completely synchronous devices. Avoidance of asynchronous events leads to higher design reliability, which is required in mission-critical devices.
- Generally, storage elements without asynchronous reset will have a smaller area. In some cases, synchronous reset implementation leads to power and area savings.
- In FPGA, the no-reset flip-flop could be implemented in BRAM or SRL, allowing better device utilization.

The disadvantages of synchronous resets include:

- Synchronous resets add extra logic to the datapath between flip-flops. This may impact timing and could be undesirable in high-speed designs.
- For multiple-clock designs, synchronous resets have to be re-synchronized for each clock domain. Also, since each clock domain has its own minimum pulse width requirements, each clock domain requires a reset pulse stretcher.
- Synchronous resets require a clock edge for reset operation. Without the clock, synchronous resets are unable to operate. This may be an issue for designs with clock gating, as it is impossible to reset the design with an already gated clock.

The Power-on-Reset in FPGA designs

Unlike ASICs, FPGA devices implement the Power-on-Reset function. It initiates the program load or configuration. Initialization values from INIT statements are loaded to into the bitstream. In addition to configuring the LUT's and routing, the bitstream contains initial values for every flip-flop and RAM bit in the device. Since this is a serial process, registers are initialized at different times throughout the configuration process, however the logic is not active yet so it is unimportant. The Global Set/Reset (GSR) signal keeps the device in non-operational mode until the end of the configuration phase. After a pre-defined number of clock cycles, and the completion of the configuration phase, the GSR signal goes inactive and the FPGA is allowed to run as programmed.

The GSR signal gates all functional clocks, so GSR signal de-assertion is safe from a metastability standpoint. However, it may introduce non-determinism in the design, because there is large skew in the GSR net to various parts of the FPGA. This skew may be greater than one or more functional clock cycles, causing reset de-assertion at different clock cycles. This behavior initializes design logic with sequential loopback in the wrong state.

Unlike ASICs, FPGA developers may use VHDL/Verilog initialization constructs to define register values after the GSR release. In case initialization values are not defined, the default value (usually, low value) will be used for initialization. The following code examples demonstrate setting default values for “data” in a flip-flop in both Verilog and VHDL:

Verilog:

```verilog
module rst_init (input clk, din,
                 output dout);

reg data;
initial data = 1'b1;

always @(posedge clk)
  data <= din;

assign dout = data;

endmodule
```

VHDL:

```vhdl
architecture Behavioral of rst_init is
signal data : std_logic := '1';
begin
   process (clk) is
   begin
      if rising_edge(clk) then
      data <= din;
      end if;
   end process;
   dout <= data;
end architecture Behavioral;
```

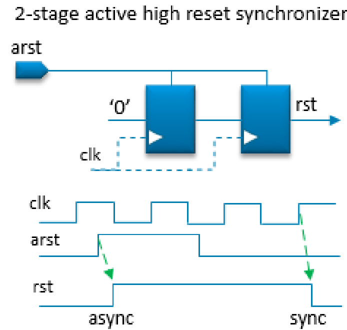
FPGA vendors encourage customers to use non-resettable Flip-Flops due to better device utilization. The first reason is to save on reset connections, and the second one is the ability to implement non-resettable flip-flops within Shift Register LUTs (SRL) as well as in Block RAMs (BRAM). In RTL simulation, however, a non-initialized flip-flop will hold an undefined “X” value. These “X” values should not affect the predictability and determinism of the device operation. The RTL simulation, however, is “X-optimistic”, converting X’s to known values. In this case, possible issues may remain hidden. It is good practice to completely eliminate X values from digital simulation by using initialization constructs or by resetting all design storage elements to known values.

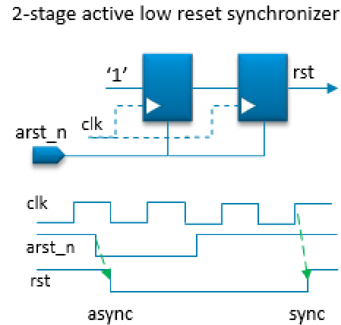
Reset Synchronization Techniques

In most cases, external-to-device reset signals are fully asynchronous. Synchronization techniques are required to convert an external-to-device reset to the internal-to-device one, either asynchronous-by-assertion or fully synchronous.

Asynchronous Reset Synchronization Techniques

The internal-to-design asynchronous reset still requires synchronization for reset de-assertion. The following synchronization design patterns perform this synchronization:

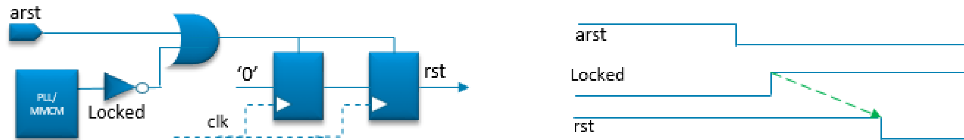




As we can see, the above synchronizers ensure the minimum width of the reset pulse to be at least two clock cycles. This may not be sufficient to reset designs with non-resettable flip-flops in pipelines. The following code snippet presents an example of a pulse synchronizer with a minimum pulse assertion width of 10 clock cycles:

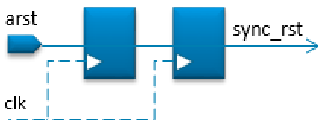
```
module rst_sync (input clk, arst, output rst_sync);
  reg [3:0] count;
  always @(posedge clk or posedge arst) begin
    if (arst)
      count <= 4'b0;
    else if (count < 15)
      count <= count+1;
  end
  assign rst_sync = (count > 10);
endmodule
```

In both ASIC and FPGA designs, it is important to guarantee that a reset is de-asserted only when design clocks are stable and running. In this case, PLL/DLL or FPGA MMCM "Locked" signals postpone reset de-assertion in reset synchronizers until the end of clock training:



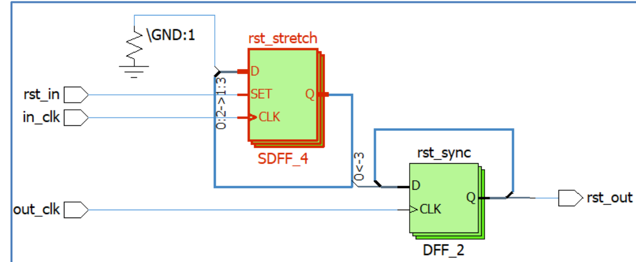
Synchronous Reset Synchronization Techniques

Synchronous resets can be easily created from asynchronous using a well-known NDFF synchronizer:



The same synchronization technique may be used to re-synchronize reset signals from one clock domain to another one. Please note that each clock domain logic has their own minimum reset pulse width requirements. A pulse stretcher could be implemented prior to reset synchronizer to ensure required pulse width, as shown in the following code example:

```
module rst_stretcher (input in_clk, out_clk, rst_in, output rst_out);

    reg [3:0] rst_stretch;
    always @ (posedge in_clk)
    begin
        rst_stretch <= rst_in ? 4'b1111 : {rst_stretch[2:0],1'b0};
    end

    reg [1:0] rst_sync;
    always @ (posedge out_clk)
    begin
        rst_sync <= {rst_sync[0],rst_stretch[3]};
    end
    assign rst_out = rst_sync[1];

endmodule
```



In FPGA, the NDFF reset synchronization Flip-Flops may be non-resettable by themselves. In order to avoid possible reset glitches, set the initial values for these Flip-flops similar to the initial value for the external asynchronous reset signal. The following FPGA-specific reset synchronizer example assumes that immediately after device configuration, the external asynchronous reset stays at logical "0", setting the initial reset synchronizer values accordingly:

```
module rst_sync (clk, din, dout);
  input clk;
  input      din;
  output     dout;

  reg data_meta, data;
  initial {data_meta,data} = 2'b0;

  always @(posedge clk) begin
      data_meta <= din;
      data      <= data_meta;
  end
  assign dout = data;

endmodule
```

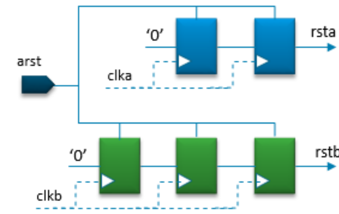
Xilinx uses a special "ASYNC_REG" placement constraint to place the NDFF synchronizers close to each other. Synchronous reset synchronizers require the usage of this constraint in CDC as well.

Similar to asynchronous reset implementation, the synchronous reset operation may be postponed until the end of clock training process. This guarantees the quality of the clock signal required for proper synchronous reset operation. Unlike asynchronous resets, synchronous resets cannot be asserted prior to or during clock training process: both reset assertion and de-assertion require proper design clocking. In this case, external asynchronous resets should not toggle until design clocks are completely locked. One solution is to route the "Locked" notification signal to the design port and implement external-to-device logic, ensuring that reset is supplied only after the "Locked" signal is asserted. The other possible solution may be to use the "Locked" signal change as a trigger for a synchronous reset, as shown in the following code snippet:

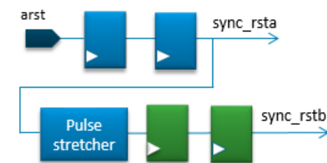
```
module rst_gen (input clk, locked, output rst_out);

    reg locked_prev;
    reg [3:0] rst_stretch;
    always @ (posedge clk) begin
        locked_prev <= locked;
        rst_stretch <= (locked_prev != locked)? 4'b1111 : {rst_stretch[2:0],1'b0};
    end
    assign rst_out = rst_stretch[3];
endmodule
```

Reset Synchronizers for Multiple-Clock Designs

Multiple-clock designs require correct reset synchronization for each clock domain. Resets for each clock domain should be properly synchronized; if needed, reset pulse stretchers can be implemented. In order to enable simultaneous asynchronous reset assertion for all clock domains, asynchronous resets have to be synchronized from the common source port, as shown in the picture. Here, pulse synchronizers provide the minimum reset length of two clka clock periods to the “clka” domain and the minimum reset length of three clkb clock periods to the “clkb” domain.



Synchronous resets, unlike asynchronous ones, may be re-synchronized from one clock domain to another one. Unlike asynchronous resets, synchronous resets do not use global nets, so this approach may be more convenient for placement & routing. A pulse stretcher may be used to accommodate for all clock domain frequencies and requirements. The following picture presents synchronous reset re-synchronization technique:



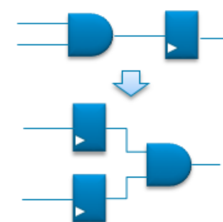
In the above example, the sync_rsta pulse needs to be stretched in the case a clock period in clkb domain is bigger than a clock period in clka domain.

Reset Techniques for Designs with Non-Resettable Storage Elements

Non-resettable storage elements may exist in either ASIC or FPGA designs. In ASIC designs, the non-resettable storage elements may initialize with non-deterministic high or low binary value. In FPGA designs, the non-resettable storage elements initialize to deterministic (usually, low) values or to the user-defined initial value (provided by HDL language initialization statements such as “initial” Verilog construct).

Nowadays, FPGA vendors encourage designers to use non-resettable flip-flops, as FPGA resources contain structures for non-resettable flip-flops mappings only. These structures include Shift Register LUTs (SRL) and distributed Block RAMs (BRAM). However, non-resettable storage elements may introduce non-determinism to the FPGA design, as the Global Set-Reset net (GSR net) may release different storage elements in different clock cycles. So, some storage elements may “wake up” one or more clock cycles earlier than the others. In the presence of sequential loopback, this behavior could corrupt initialization values and launch the design from an un-predictable state.

One of the important arguments towards non-resettable flip-flop usage is synthesis optimizations based on register retiming, pipelining and some other register-related netlist modifications. These optimization techniques are widely used in Stratix 10 devices. Register-specific optimizations are possible only in the absence of an asynchronous reset. The following picture presents a register re-timing example, where, to optimize critical timing paths an AND gate is pushed on the other side of register:

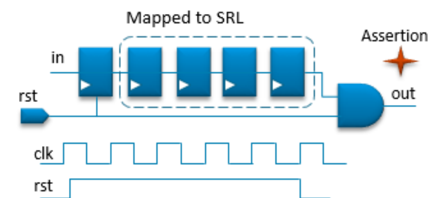


Usually, the design control logic relies on previous states to contain sequential loopbacks. Finite state machines, counters, protocols etc. require strict determinism in their operation. The datapath structures may rely on the previous states too. For example, convergence-based algorithms such as an Infinite Impulse Response (IIR) filters rely on the previous data processing cycles results. Similarly, many arithmetic operations such as division rely on the previous iteration's results.

Proper device initialization should completely eliminate non-deterministic behaviors, so designs with non-resettable elements need to be treated with special care. The following design practices are essential to completely eliminate non-determinism from designs with non-resettable storage elements:

- For control logic implementation, use resettable storage elements only.
- For datapath logic with sequential feedbacks, use resettable storage elements only. LINT tools may help designers to identify logic with sequential feedback.
- Pipelines, either in control logic or in the datapath, may still contain non-resettable flip-flops. Special treatment is required to guarantee fully deterministic behavior while resetting pipelines.

The main reason to use pipelines in ASIC design is to minimize design area and improve timing. In FPGA, the pipelines may be mapped to SRL or BRAM structures, improving device utilization and timing. Special reset-specific logic is required to guarantee proper pipeline reset operation. To propagate reset values through the pipeline, the first pipeline flip-flop should remain resettable. The pipeline output should keep the proper reset value throughout the pipeline reset process. Finally, the reset pulse length should be sufficiently wide to ensure all pipeline stages receive the correct reset values. The following picture illustrates an example of a proper resettable pipeline design:



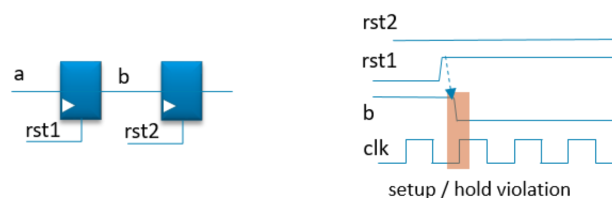
In simulation, the pipeline reset failure leads to unknown values in the pipeline output. RTL simulation is known to be "X-optimistic", so these values may remain undiscovered and may not lead to the test failure. The following picture presents the "X-optimism" RTL code simulation issue, where the X-value "disappears" once passing to the argument of "if" process:

```
// X-optimism:
// if a == 1'bx, then b = 1'b0
if (a) b <= 1'b1;
else   b <= 1'b0;

// correct code for X-propagation
if (a)       b <= 1'1;
else if (~a) b <= 1'b0;
else         b <= 1'bx;
```

The "X-checkers" design assertions help designers to ensure an absence of X-values on pipeline outputs. LINT tools help designers identify design pipelines and automatically generate design assertions to connect them to pipeline outputs.

Reset Domain Crossings

Reset Domain Crossing appears in designs receiving multiple reset signals that target different design structures. These signals introduce asynchronous reset assertion events in each reset domain. These asynchronous reset events may lead to metastability defects as well as unpredictable design initialization. The following picture illustrates a Reset Domain Crossing issue:



The asynchronous reset assertion of the “rst2” signal causes an immediate data change in the first flip-flop. If second flip-flop is still in operational mode, the asynchronous data change at “b” causes a setup or hold violation during the second flip-flop data capture. These violations may cause the second flip-flop to go to a metastable state and non-deterministic at data capture.

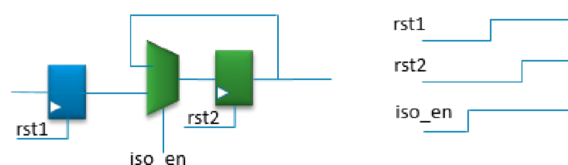
The frequency of the reset operation increases the influence of the Reset Domain Crossing effect. It is common practice to dynamically reset temporary non-functional blocks in low-power designs for power optimization purposes. Such a design should be developed considering the Reset Domain Crossing effects.

Reset Domain Crossing influences only where the reset domains cross, sending domain reset assertions asynchronously while the receiving domain reset remains un-asserted. The best design practice to avoid Reset domain crossing issues is to supply resets in the correct order, as shown in the following picture:

Here, the design has three asynchronous reset domains. The arrows represent Reset Domain Crossing directions. As we see, there are no Reset Domain Crossing issues on rst1->rst2 crossings, as rst2 is asserted after the rst1 signal. Similarly, there is no issues on rst3->rst1 and rst2->rst3 crossings. The rst3->rst2 crossing has to be considered as a real issue; the design has to be resilient to the metastability and non-determinism issues created by the Reset Domain Crossing.

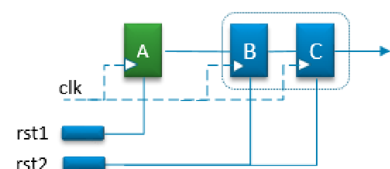
A proper reset ordering sequence may minimize the RDC (Reset Domain Crossing) influence on a designs behavior. LINT tools help designers determine the content of reset domains as well as the crossings between them. RDC crossings with proper reset sequences can be excluded from consideration. Note that it is designer’s task to properly implement reset sequences. LINT tools are able to automatically generate design assertions to confirm the proper reset sequences are given in the design.

One of the best RDC synchronization methods is to isolate the receiving domain register from the source domain register. This requires an enable signal to be generated in the receiving register’s clock and reset domains, isolating the receiving register from the transmitting register when the reset is asserted. The following picture presents the isolation cell implementation used in Low-Power designs:



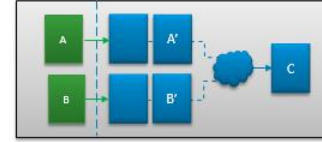
In the above example, the iso_en signal must be asserted prior to the assertion of rst1. The receiving flip-flop holds its value during an asynchronously sent flip-flop data change. LINT tools help customers identify isolation cells in Reset Domain Crossings as well as to generate verification code to ensure correct operation of “isolation-enable” signals relative to reset signal assertions.

Another technique solves the metastability-related issues, caused by Reset Domain Crossings. The well-known Clock Domain Crossing technique to beat metastability is using an NDFF synchronizer. The same technique is used to remove the metastability effect influence on Reset Domain Crossings, as it is shown in the following picture:



ALDEC
THE DESIGN VERIFICATION COMPANY

Please note that that, unlike the isolation cell, this scheme cannot prevent the non-determinism in a design because of one-clock reset shifts between reset domains. Here, LINT tools may verify the re-convergence of signals crossing reset domains, as shown on the following picture:



In the case no re-convergence is detected, reset domain signals crossed through the synchronizers may be safe.

Also, please note that Reset Domains may not necessarily match the boundaries of Clock Domains. There may be two or more independent to-each-other Reset Domains belonging to the same Clock Domain.

About Aldec, Inc.

Aldec Inc., an industry leader in Electronic Design Verification, provides a patented verification technology tool suite including: RTL Design, RTL Simulation, Hardware-Assisted Verification, SoC/ASIC Emulation & Prototyping, Design Rule Checking, CDC/RDC Analysis, IP Cores, Requirements Lifecycle Management, DO-254 Functional Verification, Embedded Solutions, High-Performance Computing and Military/Aerospace solutions. *www.aldec.com*