Asynchronous & Synchronous Reset Design Techniques - Part Deux

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

Don Mills
LCDM Engineering
mills@lcdm-eng.com

Steve Golson
Trilobyte Systems
sgolson@trilobyte.com

ABSTRACT

This paper will investigate the pros and cons of synchronous and asynchronous resets. It will then look at usage of each type of reset followed by recommendations for proper usage of each type.

1.0 Introduction

The topic of reset design is surprisingly complex and poorly emphasized. Engineering schools generally do an inadequate job of detailing the pitfalls of improper reset design. Based on our industry and consulting experience, we have compiled our current understanding of issues related to reset-design and for this paper have added the expertise of our colleague Steve Golson, who has done some very innovative reset design work. We continually solicit and welcome any feedback from colleagues related to this important design issue.

We presented our first paper on reset issues and techniques at the March 2002 SNUG conference[4] and have subsequently received numerous email responses and questions related to reset design issues.

We obviously did not adequately explain all of the issues related to the asynchronous reset synchronizer circuit because many of the emails we have received have asked if there are metastability problems related to the described circuit. The answer to this question is, no, there are no metastability issues related to this circuit and the technical analysis and explanation are now detailed in section 7.1 of this paper.

Whether to use synchronous or asynchronous resets in a design has almost become a religious issue with strong proponents claiming that their reset design technique is the only way to properly approach the subject.

In our first paper, Don and Cliff favored and recommended the use of asynchronous resets in designs and outlined our reasons for choosing this technique. With the help of our colleague, Steve Golson, we have done additional analysis on the subject and are now more neutral on the proper choice of reset implementation.

Clearly, there are distinct advantages and disadvantages to using either synchronous or asynchronous resets, and either method can be effectively used in actual designs. When choosing a reset style, it is very important to consider the issues related to the chosen style in order to make an informed design decision.

This paper presents updated techniques and considerations related to both synchronous and asynchronous reset design. This version of the paper includes updated Verilog-2001 ANSI-style ports in all of the Verilog examples.

The first version of this paper included an interesting technique for synchronizing the resetting of multiple ASICs of a high speed design application. That material has been deleted from this paper and readers are encouraged to read the first version of the paper if this subject is of interest.

2.0 Resets Purpose

Why be concerned with these annoying little resets anyway? Why devote a whole paper to such a trivial subject? Anyone who has used a PC with a certain OS loaded knows that the hardware reset comes in quite handy. It will put the computer back to a known working state (at least temporarily) by applying a system reset to each of the chips in the system that have or require a reset.

For individual ASICs, the primary purpose of a reset is to force the ASIC design (either behavioral, RTL, or structural) into a known state for simulation. Once the ASIC is built, the need for the ASIC to have reset applied is determined by the system, the application of the ASIC, and the design of the ASIC. For instance, many data path communication ASICs are designed to synchronize to an input data stream, process the data, and then output it. If sync is ever lost, the ASIC goes through a routine to re-acquire sync. If this type of ASIC is designed correctly, such that all unused states point to the "start acquiring sync" state, it can function properly in a system without ever being reset. A system reset would be required on power up for such an ASIC if the state machines in the ASIC took advantage of "don't care" logic reduction during the synthesis phase.

We believe that, in general, every flip-flop in an ASIC should be resetable whether or not it is required by the system. In some cases, when pipelined flip-flops (shift register flip-flops) are used in high speed applications, reset might be eliminated from some flip-flops to achieve higher performance designs. This type of environment requires a predetermined number of clocks during the reset active period to put the ASIC into a known state.

Many design issues must be considered before choosing a reset strategy for an ASIC design, such as whether to use synchronous or asynchronous resets, will every flip-flop receive a reset, how will the reset tree be laid out and buffered, how to verify timing of the reset tree, how to functionally test the reset with test scan vectors, and how to apply the reset across multiple clocked logic partitions.

3.0 General flip-flop coding style notes

3.1 Synchronous reset flip-flops with non reset follower flip-flops

Each Verilog procedural block or VHDL process should model only one type of flip-flop. In other words, a designer should not mix resetable flip-flops with follower flip-flops (flops with no resets) in the same procedural block or process[14]. Follower flip-flops are flip-flops that are simple data shift registers.

In the Verilog code of Example 1a and the VHDL code of Example 1b, a flip-flop is used to capture data and then its output is passed through a follower flip-flop. The first stage of this design is reset with a synchronous reset. The second stage is a follower flip-flop and is not reset, but because the two flip-flops were inferred in the same procedural block/process, the reset signal **`rst_n`** will be used as a data enable for the second flop. This coding style will generate extraneous logic as shown in Figure 1.

```
module badFFstyle (
  output reg q2,
  input      d, clk, rst_n);
  reg        q1;

  always @(posedge clk)
    if (!rst_n) q1 <= 1'b0;
    else begin
      q1 <= d;
      q2 <= q1;
    end
endmodule
```

Example 1a - Bad Verilog coding style to model dissimilar flip-flops

```
library ieee;
use ieee.std_logic_1164.all;
entity badFFstyle is
  port (
    clk   : in  std_logic;
    rst_n : in  std_logic;
    d     : in  std_logic;
    q2    : out std_logic);
end badFFstyle;

architecture rtl of badFFstyle is
  signal q1 : std_logic;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (rst_n = '0') then
        q1 <= '0';
      else
        q1 <= d;
        q2 <= q1;
      end if;
    end if;
  end process;
end rtl;
```

Example 1b - Bad VHDL coding style to model dissimilar flip-flops

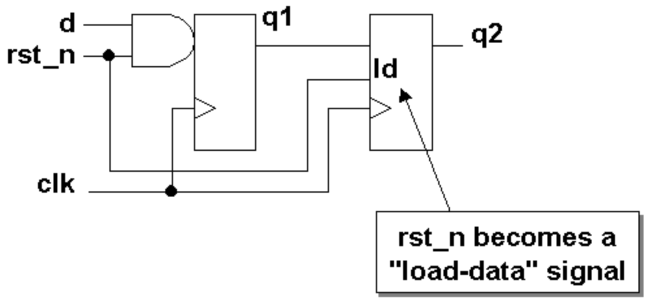


Figure 1 - Bad coding style yields a design with an unnecessary loadable flip-flop

The correct way to model a follower flip-flop is with two Verilog procedural blocks as shown in Example 2a or two VHDL processes as shown in Example 2b. These coding styles will generate the logic shown in Figure 2.

```
module goodFFstyle (
  output reg q2,
  input      d, clk, rst_n);
  reg        q1;

  always @(posedge clk)
    if (!rst_n) q1 <= 1'b0;
    else        q1 <= d;

  always @(posedge clk)
    q2 <= q1;
endmodule
```

Example 2a - Good Verilog-2001 coding style to model dissimilar flip-flops

```
library ieee;
use ieee.std_logic_1164.all;
entity goodFFstyle is
  port (
    clk   : in  std_logic;
    rst_n : in  std_logic;
    d     : in  std_logic;
    q2    : out std_logic);
end goodFFstyle;

architecture rtl of goodFFstyle is
  signal q1 : std_logic;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
```

```
          if (rst_n = '0') then
            q1 <= '0';
          else
            q1 <= d;
          end if;
        end if;
      end process;

      process (clk)
      begin
        if (clk'event and clk = '1') then
            q2 <= q1;
        end if;
      end process;
    end rtl;
```

Example 2b - Good VHDL coding style to model dissimilar flip-flops

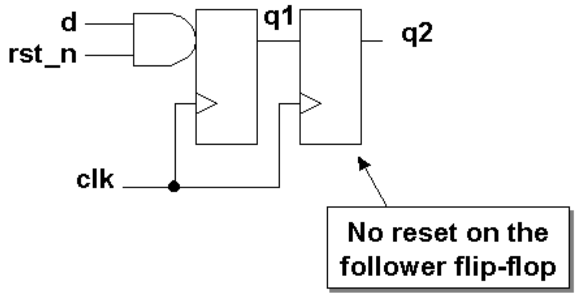


Figure 2 - Two different types of flip-flops, one with synchronous reset and one without

It should be noted that the extraneous logic generated by the code in Example 1a and Example 1b is only a result of using a synchronous reset. If an asynchronous reset approach had be used, then both coding styles would synthesize to the same design without any extra combinational logic. The generation of different flip-flop styles is largely a function of the sensitivity lists and **`if-else`** statements that are used in the HDL code. More details about the sensitivity list and **`if-else`** coding styles are detailed in section 4.1.

3.2 Flip-flop inference style

Each inferred flip-flop should **not** be independently modeled in its own procedural block/process. As a matter of style, all inferred flip-flops of a given function or even groups of functions should be described using a single procedural block/process. Multiple procedural blocks/processes should be used to model larger partitioned blocks within a given module/architecture. The exception to this guideline is that of follower flip-flops as discussed in section 3.1 where multiple procedural blocks/processes are required to efficiently model the function itself.

3.3 Assignment operator guideline

In Verilog, all assignments made inside the always block modeling an inferred flip-flop (sequential logic) should be made with nonblocking assignment operators[3]. Likewise, for VHDL, inferred flip-flops should be made using signal assignments.

4.0 Synchronous resets

As research was conducted for this paper, a collection of ESNUG and SOLV-IT articles was gathered and reviewed. Around 80+% of the gathered articles focused on synchronous reset issues. Many SNUG papers have been presented in which the presenter would claim something like, "we all know that the best way to do resets in an ASIC is to strictly use synchronous resets", or maybe, "asynchronous resets are bad and should be avoided." Yet, little evidence was offered to justify these statements. There are both advantages and disadvantages to using either synchronous or asynchronous resets. The designer must use an approach that is appropriate for the design.

Synchronous resets are based on the premise that the reset signal will only affect or reset the state of the flip-flop on the active edge of a clock. The reset can be applied to the flip-flop as part of the combinational logic generating the d-input to the flip-flop. If this is the case, the coding style to model the reset should be an **`if`**/**`else`** priority style with the reset in the **`if`** condition and all other combinational logic in the **`else`** section. If this style is not strictly observed, two possible problems can occur. First, in some simulators, based on the logic equations, the logic can block the reset from reaching the flip-flop. This is only a simulation issue, not a hardware issue, but remember, one of the prime objectives of a reset is to put the ASIC into a known state for simulation. Second, the reset could be a "late arriving signal" relative to the clock period, due to the high fanout of the reset tree. Even though the reset will be buffered from a reset buffer tree, it is wise to limit the amount of logic the reset must traverse once it reaches the local logic. This style of synchronous reset can be used with any logic or library. Example 3 shows an implementation of this style of synchronous reset as part of a loadable counter with carry out.

```
module ctr8sr (
  output reg [7:0] q,
  output reg       co,
  input      [7:0] d,
  input            ld, clk, rst_n);

  always @(posedge clk)
    if      (!rst_n) {co,q} <= 9'b0;     // sync reset
    else if (ld)     {co,q} <= d;        // sync load
    else             {co,q} <= q + 1'b1; // sync increment
endmodule
```

Example 3a - Verilog-2001 code for a loadable counter with synchronous reset

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ctr8sr is
  port (
    clk        : in  std_logic;
    rst_n      : in  std_logic;
    d          : in  std_logic;
    ld         : in  std_logic;
    q          : out std_logic_vector(7 downto 0);
    co         : out std_logic);
end ctr8sr;

architecture rtl of ctr8sr is
  signal count : std_logic_vector(8 downto 0);
begin
  co <= count(8);
  q  <= count(7 downto 0);

  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (rst_n = '0') then
        count <= (others => '0');       -- sync reset
      elsif (ld = '1') then
        count <= '0' & d;               -- sync load
      else
        count <= count + 1;             -- sync increment
      end if;
    end if;
  end process;
end rtl;
```

Example 3b - VHDL code for a loadable counter with synchronous reset

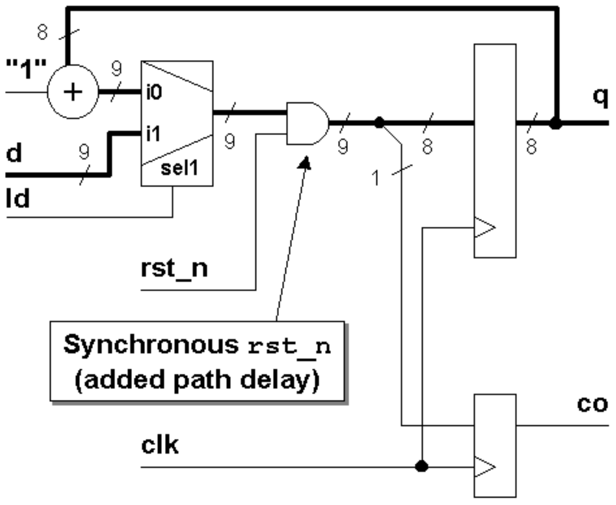


Figure 3 - Loadable counter with synchronous reset

4.1 Coding style and example circuit

The Verilog code of Example 4a and the VHDL code of 4b show the correct way to model synchronous reset flip-flops. Note that the reset is not part of the sensitivity list. For Verilog omitting the reset from the sensitivity list is what makes the reset synchronous. For VHDL omitting the reset from the sensitivity list and checking for the reset after the "**`if clk'event and clk = 1`**" statement makes the reset synchronous. Also note that the reset is given priority over any other assignment by using the **`if-else`** coding style.

```
module sync_resetFFstyle (
  output reg q,
  input      d, clk, rst_n);

  always @(posedge clk)
    if (!rst_n) q <= 1'b0;
    else        q <= d;
endmodule
```

Example 4a - Correct way to model a flip-flop with synchronous reset using Verilog-2001

```
library ieee;
use ieee.std_logic_1164.all;
entity syncresetFFstyle is
  port (
    clk   : in  std_logic;
    rst_n : in  std_logic;
    d     : in  std_logic;
    q     : out std_logic);
end syncresetFFstyle;

architecture rtl of syncresetFFstyle is
```

```
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (rst_n = '0') then
        q <= '0';
      else
        q <= d;
      end if;
    end if;
  end process;
end rtl;
```

Example 4b - Correct way to model a flip-flop with synchronous reset using VHDL

One problem with synchronous resets is that the synthesis tool cannot easily distinguish the reset signal from any other data signal. Consider the code from Example 3, which resulted in the circuit of Figure 3. The synthesis tool could alternatively have produced the circuit of Figure 4.

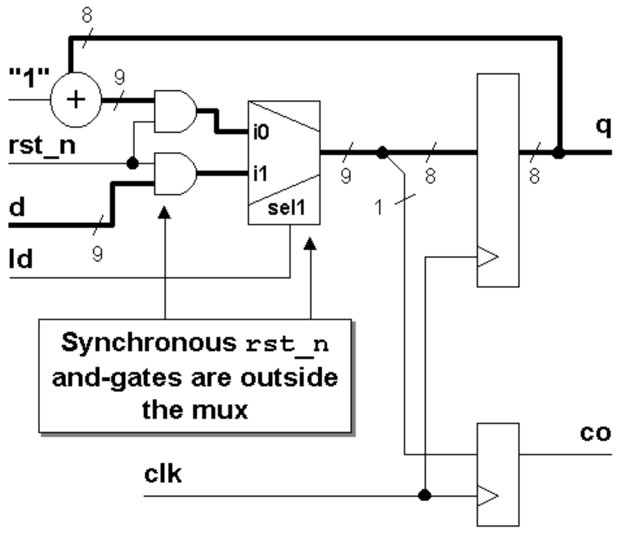


Figure 4 - Alternative circuit for loadable counter with synchronous reset

This is functionally identical to Figure 3. The only difference is that the reset and-gates are outside the MUX. Now, consider what happens at the start of a gate-level simulation. The inputs to both legs of the MUX can be forced to 0 by holding **`rst_n`** asserted low, however if **`ld`** is unknown (X) and the MUX model is pessimistic, then the flops will stay unknown (X) rather than being reset. Note this is only a problem during simulation! The actual circuit would work correctly and reset the flops to 0.

Synopsys provides the compiler directive **`sync_set_reset`** which tells the synthesis tool that a given signal is a synchronous reset (or set). The synthesis tool will "pull" this signal as close to the flop as possible to prevent this initialization problem from occurring. In this example the directive would be used by adding the following line somewhere inside the module:

```
// synopsys sync_set_reset "rst_n"
```

In general, we recommend only using Synopsys switches when they are required and make a difference; however the **`sync_set_reset`** directive does not affect the logical behavior of a design, instead it only impacts the functional implementation of a design. A wise engineer would prefer to avoid re-synthesizing the design late in the project schedule and would add the **`sync_set_reset`** directive to all RTL code from the start of the project. Since this directive is only required once per module, adding it to each module with synchronous resets is recommended.

Alternatively the synthesis variable **`hdlin_ff_always_sync_set_reset`** can be set to **`true`** prior to reading in the RTL, which will give the same result without requiring any directives in the code itself.

A few years back, another ESNUG contributor recommended adding the **`compile_preserve_sync_resets = "true"`** synthesis variable [15]. Although this variable might have been useful a few years ago, it was discontinued starting with Synopsys version 3.4b[38].

4.2 Advantages of synchronous resets

Synchronous reset logic will synthesize to smaller flip-flops, particularly if the reset is gated with the logic generating the d-input. But in such a case, the combinational logic gate count grows, so the overall gate count savings may not be that significant. If a design is tight, the area savings of one or two gates per flip-flop may ensure the ASIC fits into the die. However, in today's technology of huge die sizes, the savings of a gate or two per flip-flop is generally irrelevant and will not be a significant factor of whether a design fits into a die.

Synchronous resets generally insure that the circuit is 100% synchronous.

Synchronous resets insure that reset can only occur at an active clock edge. The clock works as a filter for small reset glitches; however, if these glitches occur near the active clock edge, the flip-flop could go metastable. This is no different or worse than every other data input; any signal that violates setup requirements can cause metastability.

In some designs, the reset must be generated by a set of internal conditions. A synchronous reset is recommended for these types of designs because it will filter the logic equation glitches between clocks.

By using synchronous resets and a pre-determined number of clocks as part of the reset process, flip-flops can be used within the reset buffer tree to help the timing of the buffer tree keep within a clock period.

According to the Reuse Methodology Manual (RMM)[32], synchronous resets might be easier to work with when using cycle based simulators. For this reason, synchronous resets are recommend in section 3.2.4(2nd edition, section 3.2.3 in the 1st edition) of the RMM. We believe using asynchronous resets with a good testbench coding style, where reset stimulus is only changed on clock edges, removes any simulation ease or speed advantages attributed to synchronous reset designs by the RMM. Translation: it is doubtful that reset style makes much difference in either ease or speed of simulation.

4.3 Disadvantages of synchronous resets

Not all ASIC libraries have flip-flops with built-in synchronous resets. However since synchronous reset is just another data input, you don't really need a special flop. The reset logic can easily be synthesized outside the flop itself.

Synchronous resets may need a pulse stretcher to guarantee a reset pulse width wide enough to ensure reset is present during an active edge of the clock[16]. This is an issue that is important to consider when doing multi-clock design. A small counter can be used that will guarantee a reset pulse width of a certain number of cycles.

A designer must work with simulator issues. A potential problem exists if the reset is generated by combinational logic in the ASIC or if the reset must traverse many levels of local combinational logic. During simulation, depending on how the reset is generated or how the reset is applied to a functional block, the reset can be masked by X's. A large number of the ESNUG articles address this issue. Most simulators will not resolve some X-logic conditions and therefore block out the synchronous reset[7][8][9][10][11][12][13][14][15][34]. Note this can also be an issue with asynchronous resets. The problem is not so much what type of reset you have, but whether the reset signal is easily controlled by an external pin.

By its very nature, a synchronous reset will require a clock in order to reset the circuit. This may not be a disadvantage to some design styles but to others, it may be an annoyance. For example, if you have a gated clock to save power, the clock may be disabled coincident with the assertion of reset. Only an asynchronous reset will work in this situation, as the reset might be removed prior to the resumption of the clock.

The requirement of a clock to cause the reset condition is significant if the ASIC/FPGA has an internal tristate bus. In order to prevent bus contention on an internal tristate bus when a chip is powered up, the chip should have a power-on asynchronous reset (see Figure 5). A synchronous reset could be used, however you must also directly de-assert the tristate enable using the reset signal (see Figure 6). This synchronous technique has the advantage of a simpler timing analysis for the reset-to-HiZ path.

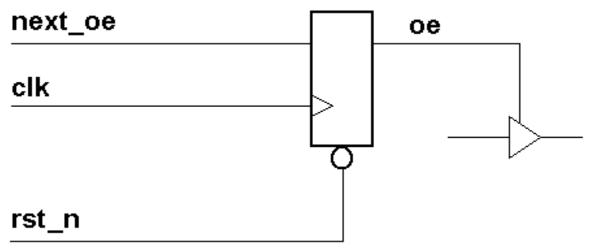


Figure 5 - Asynchronous reset for output enable

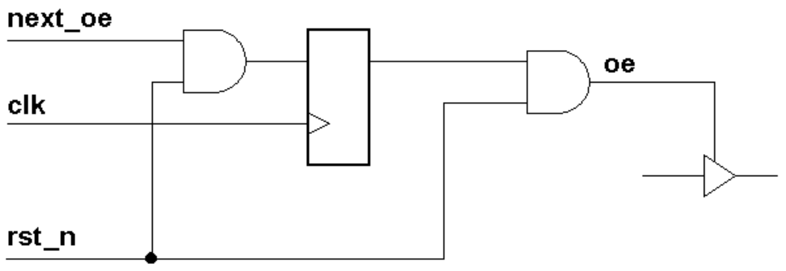


Figure 6 - Synchronous reset for output enable

5.0 Asynchronous resets

Improper implementation of asynchronous resets in digital logic design can cause serious operational design failures.

Many engineers like the idea of being able to apply the reset to their circuit and have the logic go to a known state. The biggest problem with asynchronous resets is the reset release, also called reset removal. The subject will be elaborated in detail in section 6.0.

Asynchronous reset flip-flops incorporate a reset pin into the flip-flop design. The reset pin is typically active low (the flip-flop goes into the reset state when the signal attached to the flip-flop reset pin goes to a logic low level.)

5.1 Coding style and example circuit

The Verilog code of Example 5a and the VHDL code of Example 5b show the correct way to model asynchronous reset flip-flops. Note that the reset *is* part of the sensitivity list. For Verilog, adding the reset to the sensitivity list is what makes the reset asynchronous. In order for the Verilog simulation model of an asynchronous flip-flop to simulate correctly, the sensitivity list should only be active on the leading edge of the asynchronous reset signal. Hence, in Example 5a, the always procedure block will be entered on the leading edge of the reset, then the **`if`** condition will check for the correct reset level.

Synopsys requires that if any signal in the sensitivity list is edge-sensitive, then all signals in the sensitivity list must be edge-sensitive. In other words, Synopsys forces the correct coding style. Verilog simulation does not have this requirement, but if the sensitivity list were sensitive to more than just the active clock edge and the reset leading edge, the simulation model would be incorrect[5]. Additionally, only the clock and reset signals can be in the sensitivity list. If other signals are included (legal Verilog, illegal Verilog RTL synthesis coding style) the simulation model would not be correct for a flip-flop and Synopsys would report an error while reading the model for synthesis.

For VHDL, including the reset in the sensitivity list and checking for the reset before the "**`if clk'event and clk = 1`**" statement makes the reset asynchronous. Also note that the reset is given priority over any other assignment (including the clock) by using the **`if/else`** coding style. Because of the nature of a VHDL sensitivity list and flip-flop coding style, additional signals can be included in the sensitivity list with no ill effects directly for simulation and synthesis. However, good coding style recommends that only the signals that can directly change the output of the flip-flop should be in the sensitivity list. These signals are the clock and the asynchronous reset. All other signals will slow down simulation and be ignored by synthesis.

```
module async_resetFFstyle (
  output reg q,
  input      d, clk, rst_n);

  // Verilog-2001: permits comma-separation
  // @(posedge clk, negedge rst_n)
  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d;
endmodule
```

Example 5a - Correct way to model a flip-flop with asynchronous reset using Verilog-2001

```
library ieee;
use ieee.std_logic_1164.all;
entity asyncresetFFstyle is
  port (
    clk   : in  std_logic;
    rst_n : in  std_logic;
    d     : in  std_logic;
    q     : out std_logic);
end asyncresetFFstyle;

architecture rtl of asyncresetFFstyle is
begin
  process (clk, rst_n)
  begin
    if (rst_n = '0') then
      q <= '0';
    elsif (clk'event and clk = '1') then
      q <= d;
    end if;
  end process;
end rtl;
```

Example 5b - Correct way to model a flip-flop with asynchronous reset using VHDL

The approach to synthesizing asynchronous resets will depend on the designers approach to the reset buffer tree. If the reset is driven directly from an external pin, then usually doing a **`set_drive 0`** on the reset pin and doing a **`set_dont_touch_network`** on the reset net will protect the net from being modified by synthesis. However, there is at least one ESNUG article that indicates this is not always the case[18].

One ESNUG contributor[17] indicates that sometimes **`set_resistance 0`** on the reset net might also be needed.

Alternatively rather than having **`set_resistance 0`** on the net, you can create a custom wireload model with resistance=0 and apply it to the reset input port with the command:

```
set_wire_load -port_list reset
```

A recently updated SolvNet article also notes that starting with Synopsys release 2001.08 the definition of ideal nets has slightly changed[41] and that a **`set_ideal_net`** command can be used to create ideal nets and "get no timing updates, get no delay optimization, and get no DRC fixing."

Our colleague, Chris Kiegle, reported that doing a set_disable_timing on a net for pre-v2001.08 designs helped to clean up timing reports[2], which seems to be supported by two other SolvNet articles, one related to synthesis and another related to Physical Synthesis, that recommend usage of both a **`set_false_path`** and a **`set_disable_timing`** command[35].

5.2 Modeling Verilog flip-flops with asynchronous reset and asynchronous set

One additional note should be made here with regards to modeling asynchronous resets in Verilog. The simulation model of a flip-flop that includes both an asynchronous set and an asynchronous reset in Verilog might not simulate correctly without a little help from the designer. In general, most synchronous designs do not have flop-flops that contain both an asynchronous set and asynchronous reset, but on the occasion such a flip-flop is required. The coding style of Example 6 can be used to correct the Verilog RTL simulations where both reset and set are asserted simultaneously and reset is removed first.

First note that the problem is only a simulation problem and not a synthesis problem (synthesis infers the correct flip-flop with asynchronous set/reset). The simulation problem is due to the always block that is only entered on the active edge of the set, reset or clock signals. If the reset becomes active, followed then by the set going active, then if the reset goes inactive, the flip-flop should first go to a reset state, followed by going to a set state. With both these inputs being asynchronous, the set should be active as soon as the reset is removed, but that will not be the case in Verilog since there is no way to trigger the always block until the next rising clock edge.

For those rare designs where reset and set are both permitted to be asserted simultaneously and then reset is removed first, the fix to this simulation problem is to model the flip-flop using self-correcting code enclosed within the translate_off/translate_on directives and force the output to the correct value for this one condition. The best recommendation here is to avoid, as much as possible, the condition that requires a flip-flop that uses both asynchronous set and asynchronous reset. The code in Example 6 shows the fix that will simulate correctly and guarantee a match between pre- and post-synthesis simulations. This code uses the translate_off/translate_on directives to force the correct output for the exception condition[5].

```
// Good DFF with asynchronous set and reset and self-
// correcting set-reset assignment
module dff3_aras (
  output reg q,
  input      d, clk, rst_n, set_n);

  always @(posedge clk or negedge rst_n or negedge set_n)
    if      (!rst_n) q <= 0; // asynchronous reset
    else if (!set_n) q <= 1; // asynchronous set
    else             q <= d;

  // synopsys translate_off
  always @(rst_n or set_n)
    if (rst_n && !set_n) force   q = 1;
    else                 release q;
  // synopsys translate_on
endmodule
```

Example 6 – Verilog Asynchronous SET/RESET simulation and synthesis model

5.3 Advantages of asynchronous resets

The biggest advantage to using asynchronous resets is that, as long as the vendor library has asynchronously reset-able flip-flops, the data path is guaranteed to be clean. Designs that are pushing the limit for data path timing, can not afford to have added gates and additional net delays in the data path due to logic inserted to handle synchronous resets. Using an asynchronous reset, the designer is guaranteed not to have the reset added to the data path. The code in Example 7 infers asynchronous resets that will not be added to the data path.

```
module ctr8ar (
  output reg [7:0] q,
  output reg       co;
  input      [7:0] d;
  input            ld, rst_n, clk;

  always @(posedge clk or negedge rst_n)
    if      (!rst_n) {co,q} <= 9'b0;     // async reset
    else if (ld)     {co,q} <= d;        // sync load
    else             {co,q} <= q + 1'b1; // sync increment
endmodule
```

Example 7a - Verilog-2001 code for a loadable counter with asynchronous reset

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ctr8ar is
  port (
    clk         : in  std_logic;
    rst_n       : in  std_logic;
    d           : in  std_logic;
    ld          : in  std_logic;
    q           : out std_logic_vector(7 downto 0);
    co          : out std_logic);
end ctr8ar;

architecture rtl of ctr8ar is
  signal count : std_logic_vector(8 downto 0);
begin
  co <= count(8);
  q  <= count(7 downto 0);

  process (clk)
  begin
    if (rst_n = '0') then
      count <= (others => '0');       -- sync reset
      elsif (clk'event and clk = '1') then
        if (ld = '1') then
          count <= '0' & d;           -- sync load
        else
          count <= count + 1;         -- sync increment
        end if;
      end if;
  end process;
end rtl;
```

Example 7b- VHDL code for a loadable counter with asynchronous reset

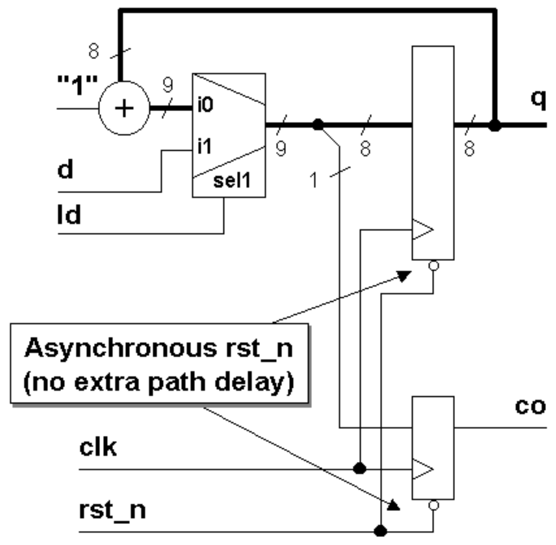


Figure 7 - Loadable counter with asynchronous reset

Another advantage favoring asynchronous resets is that the circuit can be reset with or without a clock present.

The experience of the authors is that by using the coding style for asynchronous resets described in this section, the synthesis interface tends to be automatic. That is, there is generally no need to add any synthesis attributes to get the synthesis tool to map to a flip-flop with an asynchronous reset pin.

5.4 Disadvantages of asynchronous resets

There are many reasons given by engineers as to why asynchronous resets are evil.

The Reuse Methodology Manual (RMM) suggests that asynchronous resets are not to be used because they cannot be used with cycle based simulators. This is simply not true. The basis of a cycle based simulator is that all inputs change on a clock edge. Since timing is not part of cycle based simulation, the asynchronous reset can simply be applied on the inactive clock edge.

For DFT, if the asynchronous reset is not directly driven from an I/O pin, then the reset net from the reset driver must be disabled for DFT scanning and testing. This is required for the synchronizer circuit shown in section 6.

Some designers claim that static timing analysis is very difficult to do with designs using asynchronous resets. The reset tree must be timed for both synchronous and asynchronous resets to ensure that the release of the reset can occur within one clock period. The timing analysis for a reset tree must be performed after layout to ensure this timing requirement is met. This timing analysis can be eliminated if the design uses the distributed reset synchronizer flip-flop tree discussed in section 8.2.

The biggest problem with asynchronous resets is that they are asynchronous, both at the assertion and at the de-assertion of the reset. The assertion is a non issue, the de-assertion is the issue. If the asynchronous reset is released at or near the active clock edge of a flip-flop, the output of the flip-flop could go metastable and thus the reset state of the ASIC could be lost.

Another problem that an asynchronous reset can have, depending on its source, is spurious resets due to noise or glitches on the board or system reset. See section 8.0 for a possible solution to reset glitches. If this is a real problem in a system, then one might think that using synchronous resets is the solution. A different but similar problem exists for synchronous resets if these spurious reset pulses occur near a clock edge, the flip-flops can still go metastable (but this is true of any data input that violates setup requirements).

6.0 Asynchronous reset problem

In discussing this paper topic with a colleague, the engineer stated first that since all he was working on was FPGAs, they do not have the same reset problems that ASICs have (a misconception). He went on to say that he always had an asynchronous system reset that could override everything, to put the chip into a known state. The engineer was then asked what would happen to the FPGA or ASIC if the release of the reset occurred on or near a clock edge such that the flip-flops went metastable.

Too many engineers just apply an asynchronous reset thinking that there are no problems. They test the reset in the controlled simulation environment and everything works fine, but then in the system, the design fails intermittently. The designers do not consider the idea that the release of the reset in the system (non-controlled environment) could cause the chip to go into a metastable unknown state, thus voiding the reset all together. Attention must be paid to the release of the reset so as to prevent the chip from going into a metastable unknown state when reset is released. When a synchronous reset is being used, then both the leading and trailing edges of the reset must be away from the active edge of the clock.

As shown in Figure 8, an asynchronous reset signal will be de-asserted asynchronous to the clock signal. There are two potential problems with this scenario: (1) violation of reset recovery time and, (2) reset removal happening in different clock cycles for different sequential elements.

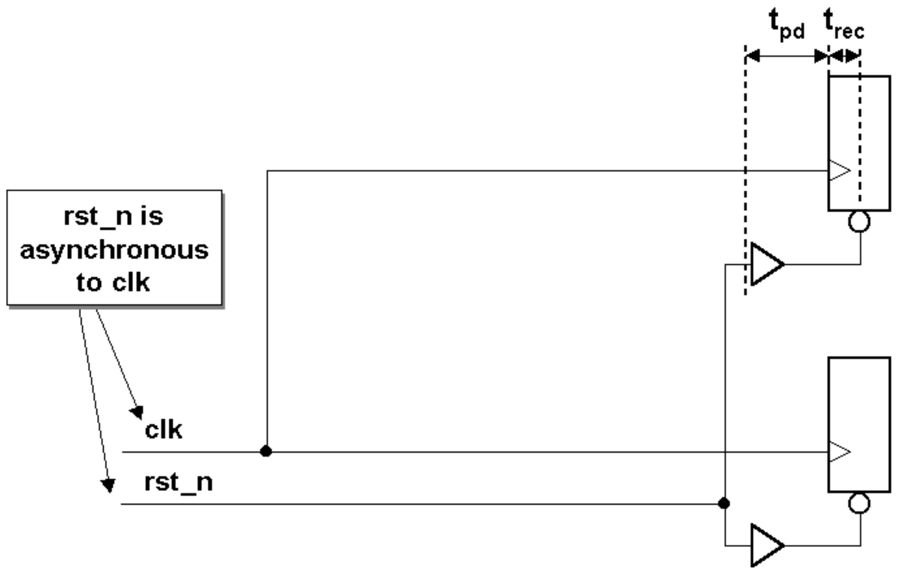


Figure 8 - Asynchronous reset removal recovery time problem

6.1 Reset recovery time

Reset recovery time refers to the time between when reset is de-asserted and the time that the clock signal goes high again. The Verilog-2001 Standard[29] has three built-in commands to model and test recovery time and signal removal timing checks: $recovery, $removal and $recrem (the latter is a combination of recovery and removal timing checks).

Recovery time is also referred to as a **tsu** setup time of the form, “PRE or CLR inactive setup time before CLK↑”[1].

Missing a recovery time can cause signal integrity or metastability problems with the registered data outputs.

6.2 Reset removal traversing different clock cycles

When reset removal is asynchronous to the rising clock edge, slight differences in propagation delays in either or both the reset signal and the clock signal can cause some registers or flip-flops to exit the reset state before others.

7.0 Reset synchronizer

Guideline: EVERY ASIC USING AN ASYNCHRONOUS RESET SHOULD INCLUDE A RESET SYNCHRONIZER CIRCUIT!!

Without a reset synchronizer, the usefulness of the asynchronous reset in the final system is void even if the reset works during simulation.

The reset synchronizer logic of Figure 9 is designed to take advantage of the best of both asynchronous and synchronous reset styles.

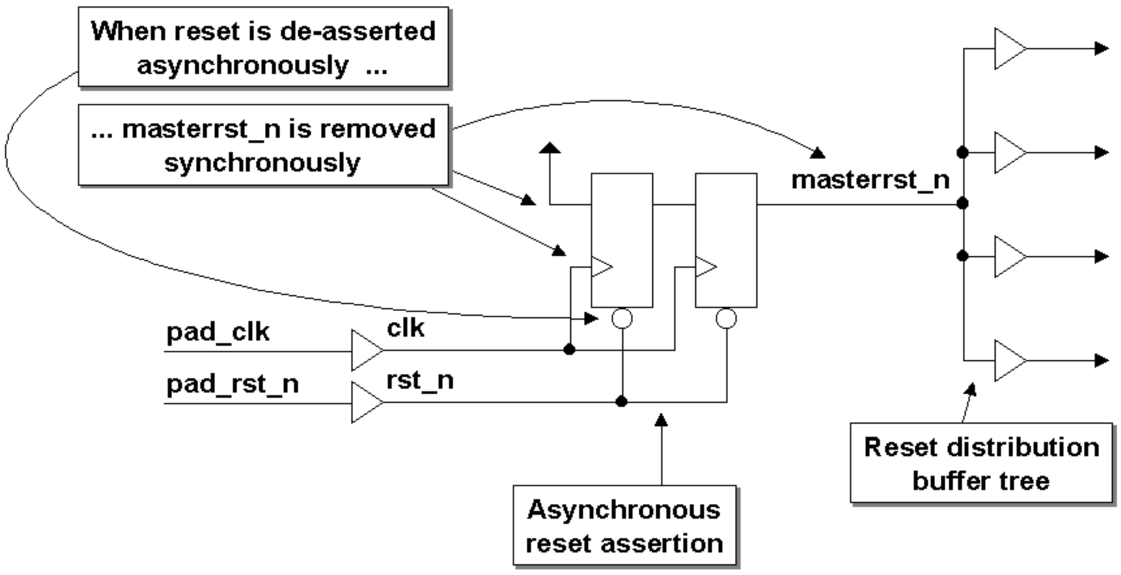

When reset is de-asserted asynchronously ...
... masterrst_n is removed synchronously
masterrst_n
pad_clk
clk
pad_rst_n
rst_n
Reset distribution buffer tree
Asynchronous reset assertion

Figure 9 - Reset Synchronizer block diagram

An external reset signal asynchronously resets a pair of master reset flip-flops, which in turn drive the master reset signal asynchronously through the reset buffer tree to the rest of the flip-flops in the design. The entire design will be asynchronously reset.

Reset removal is accomplished by de-asserting the reset signal, which then permits the d-input of the first master reset flip-flop (which is tied high) to be clocked through a reset synchronizer. It typically takes two rising clock edges after reset removal to synchronize removal of the master reset.

Two flip-flops are required to synchronize the reset signal to the clock pulse where the second flip-flop is used to remove any metastability that might be caused by the reset signal being removed asynchronously and too close to the rising clock edge. As discussed in section 5.4, these synchronization flip-flops must be kept off of the scan chain.

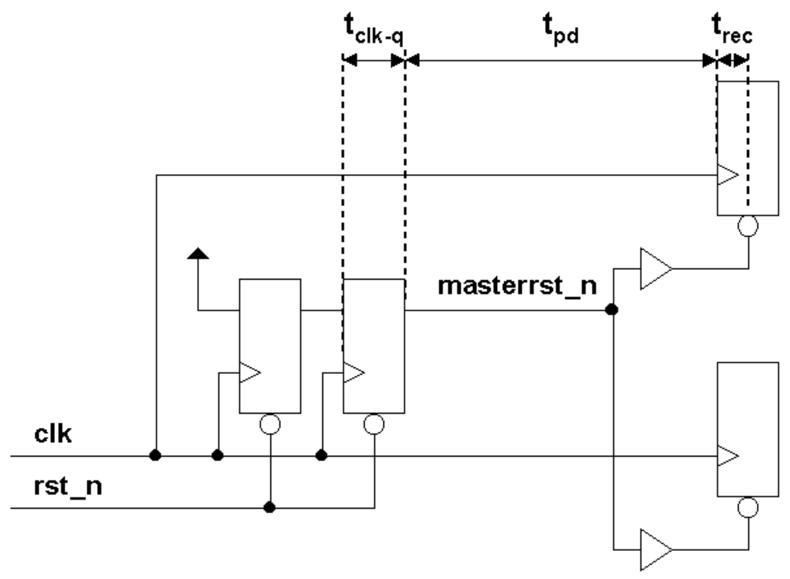


Figure 10 - Predictable reset removal to satisfy reset recovery time

A closer examination of the timing now shows that reset distribution timing is the sum of the a clk-to-q propagation delay, total delay through the reset distribution tree and meeting the reset recovery time of the destination registers and flip-flops, as shown in Figure 10.

The code for the reset synchronizer circuit is shown in Example 8.

```
module async_resetFFstyle2 (
  output reg rst_n,
  input      clk, asyncrst_n);
  reg        rff1;

  always @(posedge clk or negedge asyncrst_n)
    if (!asyncrst_n) {rst_n,rff1} <= 2'b0;
    else             {rst_n,rff1} <= {rff1,1'b1};
endmodule
```

Example 8a - Properly coded reset synchronizer using Verilog-2001

```
library ieee;
use ieee.std_logic_1164.all;
entity asyncresetFFstyle is
  port (
    clk        : in  std_logic;
    asyncrst_n : in  std_logic;
    rst_n      : out std_logic);
end asyncresetFFstyle;
```

```
architecture rtl of asyncresetFFstyle is
  signal rff1 : std_logic;
begin
  process (clk, asyncrst_n)
  begin
    if (asyncrst_n = '0') then
      rff1  <= '0';
      rst_n <= '0';
    elsif (clk'event and clk = '1') then
      rff1  <= '1';
      rst_n <= rff1;
    end if;
  end process;
end rtl;
```

Example 8b - Properly coded reset synchronizer using VHDL

7.1 Reset Synchronizer Metastability??

Ever since the publication of our first resets paper[4], we have received numerous email messages asking if the reset synchronizer has potential metastability problems on the second flip-flop when reset is removed. The answer is that the reset synchronizer DOES NOT have reset metastability problems. The analysis and discussion of related issues follows.

The first flip-flop of the reset synchronizer *does have* potential metastability problems because the input is tied high, the output has been asynchronously reset to a 0 and the reset could be removed within the specified reset recovery time of the flip-flop (the reset may go high too close to the rising edge of the clock input to the same flip-flop). This is why the second flip-flop is required.

The second flip-flop of the reset synchronizer *is not* subject to recovery time metastability because the input and output of the flip-flop are both low when reset is removed. There is no logic differential between the input and output of the flip-flop so there is no chance that the output would oscillate between two different logic values.

7.2 Erroneous ASIC Vendor Modeling

One engineer emailed to tell us that he had run simulations with four different ASIC libraries and that the flip-flop outputs of two of the ASIC libraries were going unknown during gate-level simulation when the reset was removed too close to the rising clock edge[44]. This is typically an ASIC library modeling problem. Some ASIC vendors make the mistake of applying a general recovery time specification without consideration of the input and output values being the same. When we asked the engineer to examine the transistor-level version of the model, he emailed back that the circuit was indeed not susceptible to metastability problems if the d-input was low when a reset recovery violation occurred; translation, the vendor had mistakenly applied a general reset recovery time to the flip-flop model.

7.3 Flawed Reset De-Metastabilization Circuit

One engineer suggested using the circuit in Figure 11 to remove metastability. The flip-flop in the circuit is an asynchronously reset flip-flop.

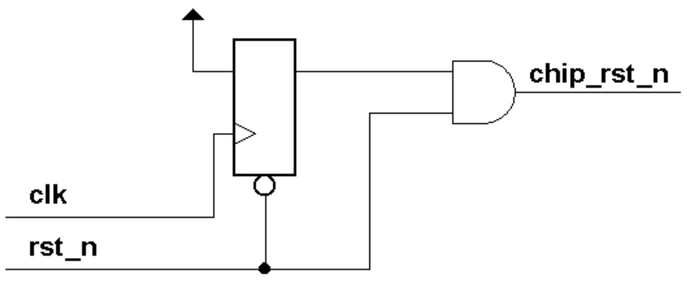


Figure 11 - Flawed reset synchronizer #1

Upon further query, the engineer reported that the output and-gate was used to remove metastability if reset is asserted too close to an active clock edge[28]. This is not necessary. There is no reset metastability issue when reset is asserted because the reset signal bypasses the clock signal in a flip-flop circuit to cleanly force the output low. The metastability issue is always related to reset removal.

This engineer handled reset recovery issues as a post place & route task. The reset delays would be measured and if necessary, a falling-clock flip-flop would be substituted for the flip-flop shown in Figure 11.

We are not convinced that this is a robust solution to the problem because min-max process variations may cause some reset circuits to fail if they have significantly different timing characteristics than the measured prototype device.

7.4 Simulation testing with resets

One EDA support engineer reported that design engineers are running simulations and releasing reset on the active edge of the clock. It should be noted that most of the time, this is a Verilog race condition and is almost always a real hardware race condition.

On real hardware, if the reset signal is removed coincident with a rising clock edge, the reset signal will violate the reset recovery time specification for the device and the output of the flip-flop could go metastable. This is another important reason why the reset synchronizer circuit described in section 7.0 is used for designs that include asynchronous reset logic.

In a simulation, if reset is removed on a posedge clock, there is usually no guarantee what the simulation result will be. Even if the RTL code behaves as expected, the gate-level simulation may behave differently due to event scheduling race conditions and different IEEE-Verilog compliant simulators may even yield different RTL simulation results. Most ASIC libraries will drive an X-output from the gate-level flip-flop simulation model when a reset recovery time violation occurs (typically modeled using a User Defined Primitive, or UDP for short).

Since one important goal related to testbench creation is to make sure that the same testbench can be used to verify the same results for both pre- and post-synthesis simulations, in our testbenches we always change the reset signal on the inactive clock edge, far away from any potential recovery time violation and simulation race condition.

Guideline: In general, change the testbench reset signal on the inactive clock edge using blocking assignments.

Another good testbench strategy is to assert reset at time 0 to initialize all resetable registers and flip-flops. Asserting reset at time zero could also cause a Verilog race condition but this race condition can be easily avoided by making the first testbench assignment to reset using a nonblocking assignment as shown in Example 9. Using a time-0 nonblocking assignment to reset causes the reset signal to be updated in the nonblocking update events region of the Verilog event queue at time 0, forcing all procedural blocks to become active before the reset signal is asserted, which means all reset-sensitive procedural blocks are guaranteed to trigger at time 0 (no Verilog race issues).

```
initial begin              // clock oscillator
  clk <= 0;                // time 0 nonblocking assignment
  forever #(`CYCLE/2) clk = ~clk;
end

initial begin
  rst_n <= 0;              // time 0 nonblocking assignment
  @(posedge clk);          // Wait to get past time 0
  @(negedge clk) rst = 1; // rst_n low for one clock cycle
  ...
end
```

Example 9 - Good coding style for time-0 reset assertion

One EDA tool support engineer who receives complaints about Verilog race conditions by engineers that release reset coincident with the active clock edge in their testbenches (as noted above, this is a real hardware race condition, a Verilog simulation race condition, and in our opinion a sign of a poorly trained Verilog engineer) recommended that design engineers avoid asynchronous-reset flip-flops to eliminate the potential Verilog race conditions related to reset removel. He then showed a typical asynchronous reset flip-flop model similar to the one shown in Example 10.

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

Example 10 - Typical coding style for flip-flops with asynchronous resets

He correctly noted that either the clk would go high while rst_n is low, causing q to be reset, or clk could go high after rst_n is released, causing q to be assigned the value of d.

We pointed out that synchronous reset flip-flops can experience the same non-deterministic simulation results for the exact same reason and that synchronous reset flip-flops do not change the fact that this would still be a real hardware problem. Conclusion: do not release reset coincident with the active clock edge of the design from a testbench. This might make a good interview question for design and verification engineers!

8.0 Reset distribution tree

The reset distribution tree requires almost as much attention as a clock distribution tree, because there are generally as many reset-input loads as there are clock-input loads in a typical digital design, as shown in Figure 12. The timing requirements for reset tree are common for both synchronous and asynchronous reset styles.

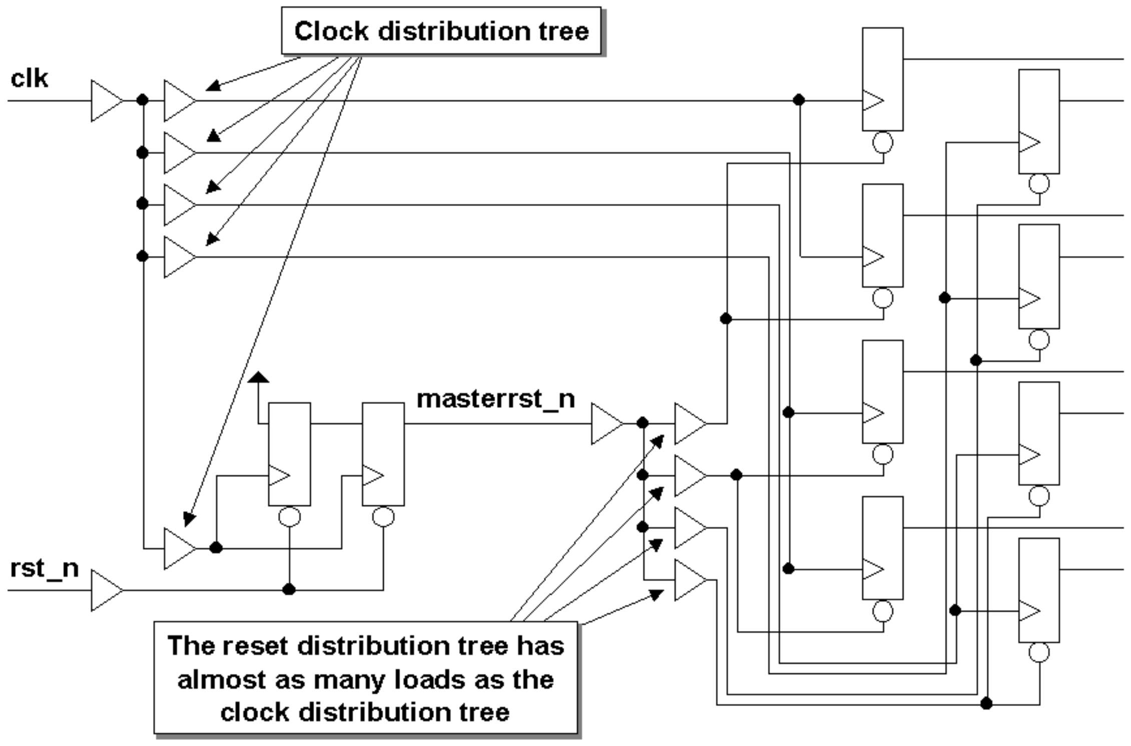

Clock distribution tree
clk
masterrst_n
rst_n
The reset distribution tree has almost as many loads as the clock distribution tree

Figure 12 - Reset distribution tree

One important difference between a clock distribution tree and a reset distribution tree is the requirement to closely balance the skew between the distributed resets. Unlike clock signals, skew between reset signals is not critical as long as the delay associated with any reset signal is short enough to allow propagation to all reset loads within a clock period and still meet recovery time of all destination registers and flip-flops.

Care must be taken to analyze the clock tree timing against the clk-q-reset tree timing. The safest way to clock a reset tree (synchronous or asynchronous reset) is to clock the internal-master-reset flip-flop from a leaf-clock of the clock tree as shown in Figure 13. If this approach will meet timing, life is good. In most cases, there is not enough time to have a clock pulse traverse the clock tree, clock the reset-driving flip-flop and then have the reset traverse the reset tree, all within one clock period.

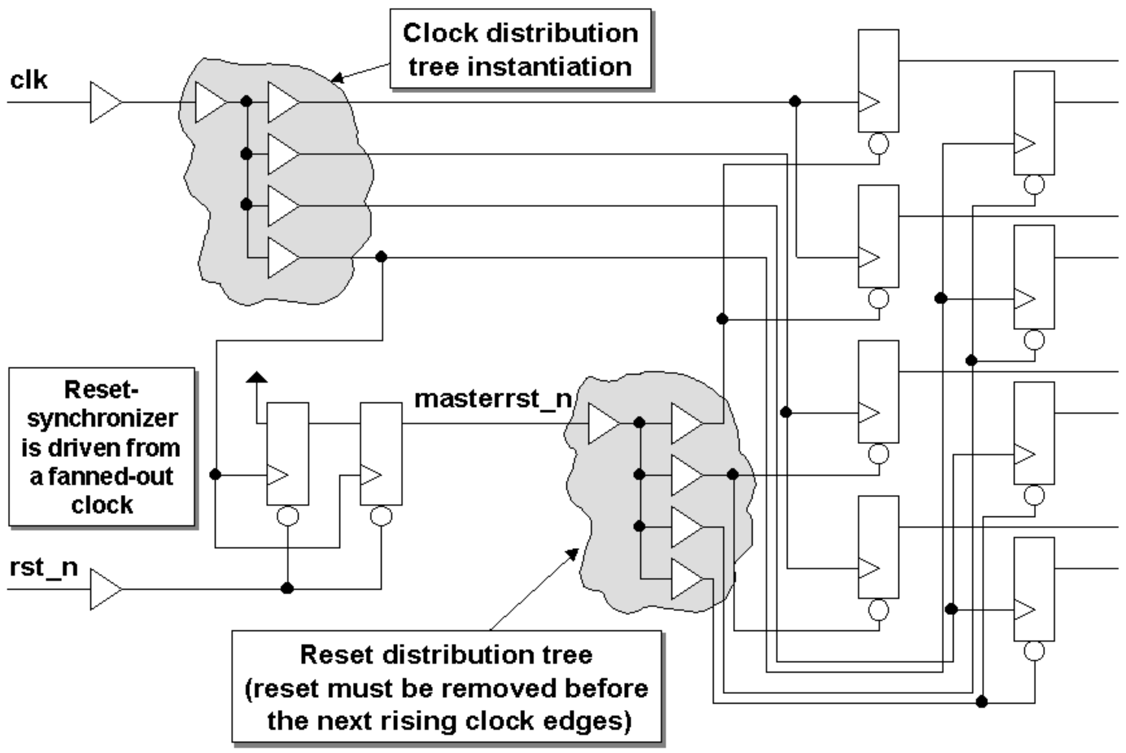
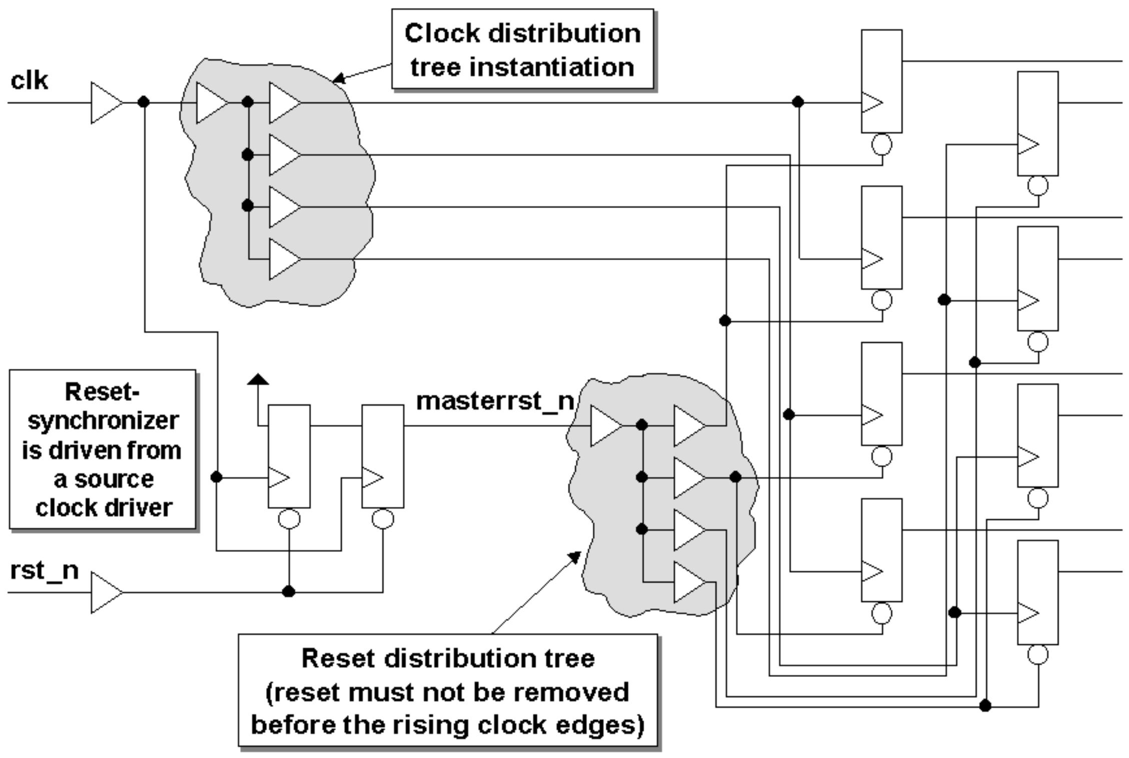

Clock distribution tree instantiation
clk
Reset-synchronizer is driven from a fanned-out clock
masterrst_n
rst_n
Reset distribution tree (reset must be removed before the next rising clock edges)

Figure 13 - Reset tree driven from a delayed, buffered clock

In order to help speed the reset arrival to all the system flip-flops, the reset-driver flip-flop is clocked with an early clock as shown in Figure 14. Post layout timing analysis must be made to ensure that the reset release for asynchronous resets and both the assertion and release for synchronous reset do not beat the clock to the flip-flops; meaning the reset must not violate setup and hold on the flops. Often detailed timing adjustments like this can not be made until the layout is done and real timing is available for the two trees.



Clock distribution tree instantiation
clk
Reset-synchronizer is driven from a source clock driver
masterrst_n
rst_n
Reset distribution tree (reset must not be removed before the rising clock edges)

Figure 14 - Reset synchronizer driven in parallel to the clock distribution tree

Ignoring this problem will not make it go away. Gee, and we all thought resets were such a basic topic.

8.1 Synchronous reset distribution technique

For synchronous resets, one technique is to build a distributed reset buffer tree with flops embedded in the tree. This keeps the timing requirements fairly simple, because you don't have to reach every flip-flop in one clock period. In each module, the reset input to the module is run through a simple D-flip-flop, and then this delayed reset is used to reset logic inside the module *and* to drive the reset input of any submodules. Thus it may take several clocks for all flip-flops in the design to be reset (Note: similar problems are seen with multi-clock designs where the reset signal must cross clock domains). Thus each module would contain code such as

```
input reset_raw;

// synopsys sync_set_reset "reset"
always @ (posedge clk) reset <= reset_raw;
```

where **`reset`** is used to synchronously reset all logic within the enclosed module, and is also connected to the **`reset_raw`** port of any submodules.

With such a technique the synchronous reset signal can be treated like any other data signal, with easy timing analysis for every module in the design, and reasonable fanouts at any stage of the reset tree.

8.2 Asynchronous reset distribution technique

For asynchronous resets, an interesting technique is to again use a distributed asynchronous reset synchronizer scheme, similar to the reset tree described in section 8.1, to replace the reset buffer tree.

This approach for asynchronous resets places reset synchronizers at every level of hierarchy of the design. This is the same approach as distributing synchronous reset flip-flops as discussed in section 8.1. The difference, is that there are two flip-flops per reset synchronizer at each level instead of one flip-flop used for the synchronous reset approach. The local reset drives the asynchronous reset inputs to local flip-flops instead of being gated into the data path as done with the synchronous reset technique.

This method of distributed reset synchronizers will reset the same as having one reset synchronizer at the top level, in that the design will asynchronously reset when reset is applied and will be synchronously released from the reset. However, the design will be released from reset over a number of clock cycles as the release of reset trickles through the hierarchical reset tree.

Note that using this technique, the whole design may not come out of reset at the same time (within the same clock cycle). Whether or not this is a problem is design dependent. Most designs can deal with the release of reset across many clocks. If the design functionality is such that the whole design must come out of reset within the same clock cycle, then the reset tree of reset synchronizers must be balanced at all end points. This is true for both synchronous and asynchronous resets.

Section 8.0 discussed details about buffering the global asynchronous reset tree. The biggest problem with this approach is the timing verification of the reset tree to ensure that the release of the reset occurs within one clock period. Preliminary analysis can be done prior to place and route, but the reset tree from section 8.0 must be analyzed after place & route (P&R). Unfortunately, if timing adjustments are required, the designer most often must make these adjustments by hand in the P&R domain and then re-time the routed design, repeating this process until the timing requirements are met. The approach discussed in this section using the distributed reset synchronizers removes the backend manual adjustments and will allow the synthesis tools to do the job of timing and buffering the design automatically. Using this distribution technique, the reset buffering is completely local to the current level (the same as with the synchronous approach discussed in section 8.1).

When using asynchronous resets, it is vitally important that the designer uses the proper variables set to the proper settings in both DC and PT to ensure that the asynchronous reset driven from the q-output of the reset synchronizing flip-flops are buffered (if needed) and timed. Details on these settings can be found in SolvNet article #901989[43]. The article states, both DC and PT can and will time to the asynchrnoous reset input against the local clock if the following variables are set:

```
pt_shell> set timing_disable_recovery_removal_checks "false"
dc_shell> enable_recovery_removal_arcs "true"
```

These settings should be the default setting from Synopsys (just make sure they are set for your environment). With these flags set correctly, and the distributed reset synchronizers, the clock-tree-like task of building a buffered reset tree is eliminated.

If you are designing FPGAs, then the reset synchronizer distribution method discussed in this section may be preferred[30]. There are two good reasons this may be true: (1) The Global Set/Reset (GSR) buffer on most FPGAs does both asynchronous reset *and asynchronous reset removal* with all of the associated problems related to asynchronous reset removal already detailed in this paper. Unless an FPGA vendor has implemented a reset synchronizer on-chip, the engineer will need to implement an off-chip asynchronous reset synchronizer and the inter-chip pin-pad delays may be too slow to effectively implement. (2) It is not unusual to have multiple clock buffers with multiple clock domains but only one GSR buffer and each clock domain should control a corresponding reset synchronizer (discussed in section 11.0).

There is also a good reason to consider using asynchronous resets instead of synchronous resets in an FPGA device. In general, FPGAs have an abundance of flip-flops, but FPGA design speed is often limited by the size of the combinational blocks required for the design. If a block of combinational logic does not fit into a single cell of FPGA lookup tables, the combinational logic must be continued in additional lookup tables with corresponding lookup delays and inter-cell routing delays. The use of synchronous resets typically requires at least part of a lookup table that might be needed by a combinational logic block.

And finally, DFT for FPGAs is a non-issue since FPGA designs do not include DFT internal scan, thus the issues regarding DFT with asynchronous resets on an FPGA do not exist.

9.0 Reset-glitch filtering

As stated earlier in this paper, one of the biggest issues with asynchronous resets is that they are asynchronous and therefore carry with them some characteristics that must be dealt with depending on the source of the reset. With asynchronous resets, any input wide enough to meet the minimum reset pulse width for a flip-flop will cause the flip-flop to reset. If the reset line is subject to glitching, this can be a real problem. Presented here is one approach that will work to filter out the glitches, but it is ugly! This solution requires that a digital delay (meaning the delay will vary with temperature, voltage and process) to filter out small glitches. The reset input pad should also be a Schmidt triggered pad to help with glitch filtering. Figure 15 shows the implementation of this approach.

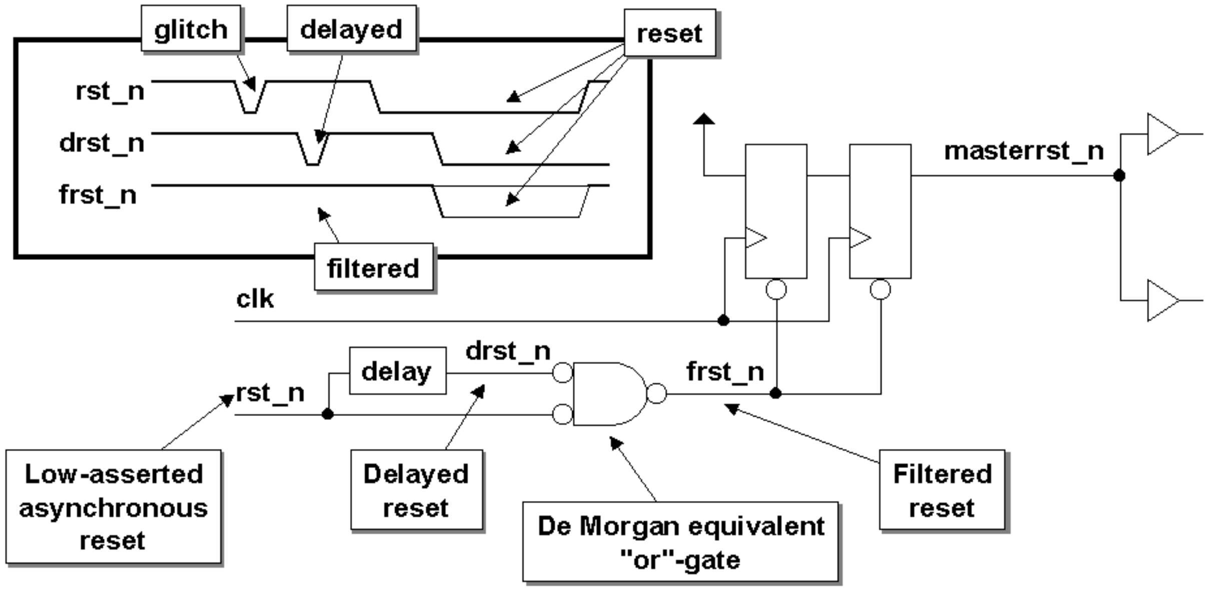


Figure 15 - Reset glitch filtering

In order to add the delay, some vendors provide a delay hard macro that can be hand instantiated. If such a delay macro is not available, the designer could manually instantiate the delay into the synthesized design after optimization – remember not to optimize this block after the delay has been inserted or it will be removed. Of course the elements could have don't touch attributes applied to prevent them from being removed. A second approach is to instantiated a slow buffer in a module and then instantiated that module multiple times to get the desired delay. Many variations could expand on this concept.

This glitch filter is not needed in all systems. The designer must research the system requirements to determine whether or not a delay is needed.

10.0 DFT for asynchronous resets

One important issue related to asynchronous resets has to do with the use of Design For Test (DFT). Engineers have commented that the toughest part about using asynchronous resets in a design is related to DFT[26] while other engineers that are using DFT with asynchronous resets claim it is not difficult[6]. If an asynchronous reset is being gated and used as an active functional input, DFT becomes difficult.

If the asynchronous reset is used as part of the functional design, then all the comments in ESNUG #409 item 11[26] regarding DFT difficulties are correct. DFT will be hard, if not impossible. The functional design is no longer following the base design guidelines for synchronous design that the synthesis and timing analysis tools require for accurate and correct results. The guidelines recommended in this paper with regards to asynchronous reset are based on the reset being only an initialization reset and that reset is not part of the functionality of the device. The only logic in the reset path would be the reset synchronizers. If this design approach is used, then the DFT approach discussed below provides a very simple and thorough approach to DFT for asynchronous reset.

Applying Design for Test (DFT) functionality to a design is a two step process. First, the flips-flops in the design are stitched together into a scan chain accessible from external I/O pins, this is called scan insertion. The scan chain is typically not part of the functional design. Second, a software program is run to generate a set of scan vectors that, when applied to the scan chain, will test and verify the design. This software program is called Automatic Test Program Generation or ATPG. The primary objective of the scan vectors is to provide foundry vectors for manufacture tests of the wafers and die as well as tests for the final packaged part.

The process of applying the ATPG vectors to create a test is based on:

1. scanning a known state into all the flip-flops in the chip,
2. switching the flip-flops from scan shift mode, to functional data input mode,
3. applying one functional clock,
4. switching the flip-flops back to scan shift mode to scan out the result of the one functional clock while scanning in the next test vector.

The DFT process usually requires two control pins. One that puts the design into "test mode." This pin is used to mask off non-testable logic such as internally generated asynchronous resets, asynchronous combinational feedback loops, and many other logic conditions that require special attention. This pin is usually held constant during the entire test. The second control pin is the shift enable pin.

In order for the ATPG vectors to work, the test program must be able to control all the inputs to the flip-flops on the scan chain in the chip. This includes not only the clock and data, but also the reset pin (synchronous or asynchronous). If the reset is driven directly from an I/O pin, then the reset is held in a non-reset state. If the reset is internally generated, then the master internal reset is held in a non-reset state by the test mode signal. If the internally generated reset were not masked off during ATPG, then the reset condition might occur during scan causing the flip-flops in the chip to be reset, and thus lose the vector data being scanned in.

Even though the asynchronous reset is held to the non-reset state for ATPG, this does not mean that the reset/set cannot be tested as part of the DFT process. Before locking out the reset with test mode and generating the ATPG vectors, a few vectors can be manually generated to create reset/set test vectors. The process required to test asynchronous resets for DFT is very straight forward and may be automatic with some DFT tools. If the scan tool does not automatic test the asynchronous resets/sets, then they must be setup manually. The basic steps to manually test the asynchronous resets/sets are as follows:

1. scan in all ones into the scan chain
2. issue and release the asynchronous reset
3. scan out the result and scan in all zeros
4. issue and release the reset
5. scan out the result
6. set the reset input to the non reset state and then apply the ATPG generated vectors.

This test approach will scan test for both asynchronous resets and sets. These manually generated vectors will be added to the ATPG vectors to provide a higher fault coverage for the manufacture test. If the design uses flip-flops with synchronous reset inputs, then modifying the above manual asynchronous reset test slightly will give a similar test for the synchronous reset environment. Add to the steps above a functional clock while the reset is applied. All other steps would remain the same.

For the reset synchronizer circuit discussed in this paper, the two synchronizer flips-flops should not be included in the scan chain, but should be tested using the manual process discussed above.

11.0 Multi-clock reset issues

For a multi-clock design, a separate asynchronous reset synchronizer circuit and reset distribution tree should be used for each clock domain. This is done to insure that reset signals can indeed be guaranteed to meet the reset recovery time for each register in each clock domain.

As discussed earlier, asynchronous reset assertion is not a problem. The problem is graceful removal of reset and synchronized startup of all logic after reset is removed.

Depending on the constraints of the design, there are two techniques that could be employed: (1) non-coordinated reset removal, and (2) sequenced coordination of reset removal.

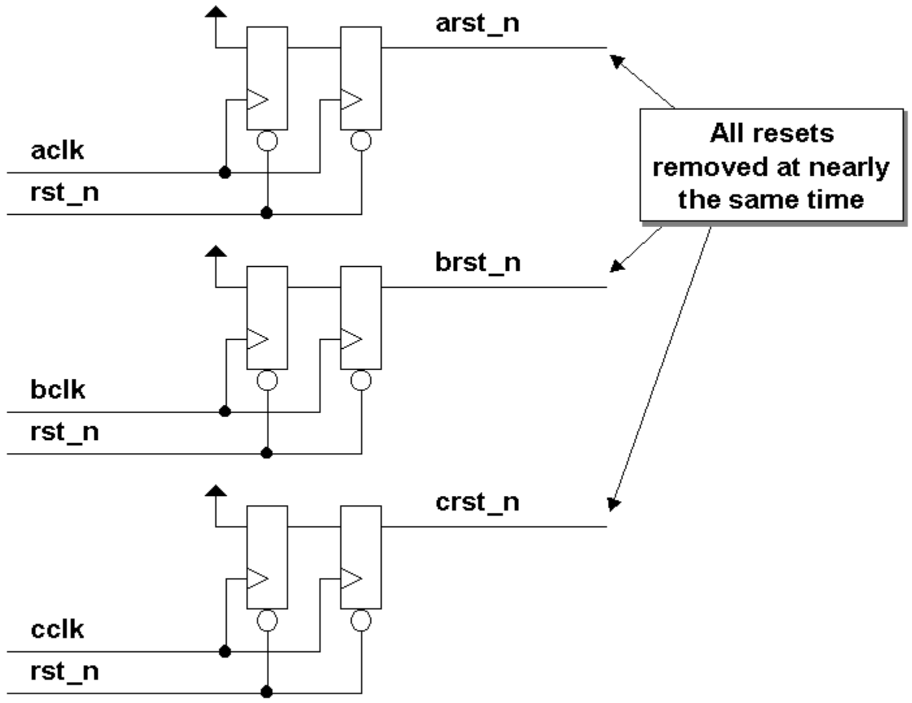


Figure 16 - Multi-clock reset removal

11.1 Non-coordinated reset removal

For many multi-clock designs, exactly when reset is removed within one clock domain compared to when it is removed in another clock domain is not important. Typically in these designs, any control signals crossing clock boundaries are passed through some type of request-acknowledge handshaking sequence and the delayed acknowledge from one clock domain to another is not going to cause invalid execution of the hardware. For this type of design, creating separate asynchronous reset synchronizers as shown in Figure 16 is sufficient, and the fact that **`arst_n`**, **`brst_n`** and **`crst_n`** could be removed in any sequence is not important to the design.

11.2 Sequenced coordination of reset removal

For some multi-clock designs, reset removal must be ordered and proper sequence. For this type of design, creating prioritized asynchronous reset synchronizers as shown in Figure 17 might be required to insure that all **`aclk`** domain logic is activated after reset is removed before the **`bclk`** logic, which must also be activated before the **`cclk`** logic becomes active.

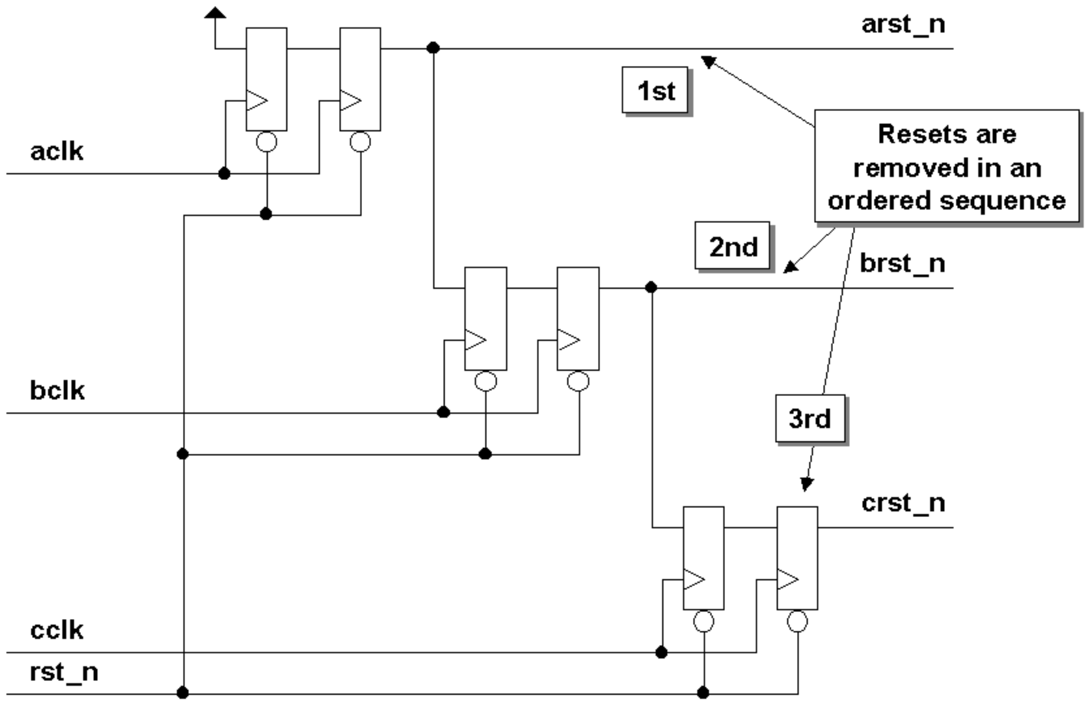


Figure 17 - Multi-clock ordered reset removal

For this type of design, only the highest priority asynchronous reset synchronizer input is tied high. The other asynchronous reset synchronizer inputs are tied to the master resets from higher priority clock domains.

12.0 Conclusions

Properly used, synchronous and asynchronous resets can each guarantee reliable reset assertion. Although an asynchronous reset is a safe way to reliably reset circuitry, removal of an asynchronous reset can cause significant problems if not done properly.

The proper way to design with asynchronous resets is to add the reset synchronizer logic to allow asynchronous reset of the design and to insure synchronous reset removal to permit safe restoration of normal design functionality.

Using DFT with asynchronous resets is still achievable as long as the asynchronous reset can be controlled during test.

Whether the design uses synchronous or asynchronous resets, using one of the distributed flip-flop trees as described in this paper may be worthy of consideration by the designer since they remove many of the issues related to buffering, timing and layout of a reset tree.

In conclusion, simple little resets ... aren't!

References

[1] *ALS/AS Logic Data Book*, Texas Instruments, 1986, pg. 2-78.

[2] Chris Kiegle, personal communication

[3] Clifford E. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!," *SNUG (Synopsys Users Group) 2000 User Papers*, section-MC1 (1st paper), March 2000. Also available at www.sunburst-design.com/papers

[4] Clifford E. Cummings and Don Mills, "Synchronous Resets? Asynchronous Resets? I am so confused! How will I ever know which to use?" *SNUG (Synopsys Users Group) San Jose, 2002 User Papers*, March 2002. Also available at www.sunburst-design.com/papers and www.lcdm-eng.com/papers.htm

[5] Don Mills and Clifford E. Cummings, "RTL Coding Styles That Yield Simulation and Synthesis Mismatches," *SNUG (Synopsys Users Group) 1999 Proceedings*, section-TA2 (2nd paper), March 1999. Also available at www.lcdm-eng.com/papers.htm and www.sunburst-design.com/papers

[6] Erick Pew, personal communication

[7] ESNUG #60, Item 1- www.deepchip.com/items/0060-01.html

[8] ESNUG #240, Item 7- www.deepchip.com/items/0240-07.html

[9] ESNUG #242, Item 6 - www.deepchip.com/items/0242-06.html

[10] ESNUG #243, Item 4 - www.deepchip.com/items/0243-04.html

[11] ESNUG #244, Item 5 - www.deepchip.com/items/0244-05.html

[12] ESNUG #246, Item 5 - www.deepchip.com/items/0246-05.html

[13] ESNUG #278, Item 7 - www.deepchip.com/items/0278-07.html

[14] ESNUG #280, Item 4 - www.deepchip.com/items/0280-04.html

[15] ESNUG #281, Item 2 - www.deepchip.com/items/0281-02.html

[16] ESNUG #355, Item 2 - www.deepchip.com/items/0355-02.html

[17] ESNUG #356, Item 4 - www.deepchip.com/items/0356-04.html

[18] ESNUG #373, Item 6 - www.deepchip.com/items/0373-06.html

[19] ESNUG #375, Item 14 - www.deepchip.com/items/0375-14.html

[20] ESNUG #379, Item 14 - www.deepchip.com/items/0379-14.html

[21] ESNUG #380, Item 13 - www.deepchip.com/items/0380-13.html

[22] ESNUG #381, Item 13 - www.deepchip.com/items/0381-13.html

[23] ESNUG #393 Item 1 - www.deepchip.com/items/0393-01.html

[24] ESNUG #396, Item 1 - www.deepchip.com/items/0396-01.html

[25] ESNUG #404, Item 15 - www.deepchip.com/items/0404-15.html

[26] ESNUG #409, Item 11 - www.deepchip.com/items/0409-11.html

[27] ESNUG #410, Item 3 - www.deepchip.com/items/0410-03.html

[28] Gzim Derti, personal communication

[29] *IEEE Standard Verilog Hardware Description Language*, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-2001.

[30] Ken Chapman, "Get Smart About Reset (Think Local, Not Global)," Xilinx TechXclusives, downloaded from www.xilinx.com/support/techxclusives/global-techX19.htm

[31] Lee Tatistcheff, personal communication

[32] Michael Keating, and Pierre Bricaud, *Reuse Methodology Manual*, Second Edition, Kluwer Academic Publishers, 1999, pg. 35.

[33] Synopsys SolvNet, Doc Name: 902298, "Recovery and Removal timing checks on Primetime," Updated 02/13/2002 - solvnet.synopsys.com/retrieve/902298.html

[34] Synopsys SolvNet, Doc Name: 903391, "Methodology and limitations of synthesis for synchronous set and reset," Updated 09/07/2001 - solvnet.synopsys.com/retrieve/903391.html

[35] Synopsys SolvNet, Doc Name: 900214, "Handling High Fanout Nets in 2001.08" Updated: 11/01/2001 - solvnet.synopsys.com/retrieve/900214.html

[36] Synopsys SolvNet, Doc Name: 004186, "Reset Pros and Cons" Updated: 03/10/2003 - solvnet.synopsys.com/retrieve/004186.html

[37] Synopsys SolvNet, Doc Name: 901644, "Multiple Synchronous Resets" Updated: 09/07/2001 - solvnet.synopsys.com/retrieve/901644.html

[38] Synopsys SolvNet, Doc Name: 901093, "Is the compile_preserve_sync_reset Switch Still Valid?," Updated: 09/07/2001 - solvnet.synopsys.com/retrieve/901093.html

[39] Synopsys SolvNet, Doc Name: 902448, "Is the compile_preserve_sync_reset Switch Still Valid?," Updated: 05/22/1998 - solvnet.synopsys.com/retrieve/902448.html (this article and the previous reference have the same name but are different articles.)

[40] Synopsys SolvNet, Doc Name: 901811, "Why can't I synthesize synchronous reset flip-flops?," Updated: 08/16/1999 - solvnet.synopsys.com/retrieve/901811.html

[41] Synopsys SolvNet, Doc Name: 901241, "Commands for High Fanout Nets: high_fanout_net_threshold and report_high_fanout" Updated: 01/31/2003 - solvnet.synopsys.com/retrieve/901241.html

[42] Synopsys SolvNet, Doc Name: 901264, "Data and Synchronous Reset Swapped," Updated: 06/18/2003 - solvnet.synopsys.com/retrieve/901264.html

[43] Synopsys SolvNet, Doc Name: 901989, “Default Settings for Recovery/Removal Arcs in PrimeTime and Design Compiler,” Updated: 01/12/1999 - solvnet.synopsys.com/retrieve/901989.html

[44] Zeeshan Sarwar, personal communication

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 21 years of ASIC, FPGA and system design experience and 11 years of Verilog, synthesis and methodology training experience.

Mr. Cummings, a member of the IEEE 1364 Verilog Standards Group (VSG) since 1994, is the only Verilog and SystemVerilog trainer to co-develop and co-author the IEEE 1364-1995 & IEEE 1364-2001 Verilog Standards, the IEEE 1364.1-2002 Verilog RTL Synthesis Standard and the Accellera SystemVerilog 3.0 & 3.1 Standards.

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

Email address: cliffc@sunburst-design.com

Don Mills is an independent EDA consultant, ASIC designer, and Verilog/VHDL trainer with 17 years of experience.

Don has inflicted pain on Aart De Geuss for too many years as SNUG Technical Chair. Aart was more than happy to see him leave! Not really, Don chaired three San Jose SNUG conferences: 1998-2000, the first Boston SNUG 1999, and three Europe SNUG conferences 2001- 2003.

Don holds a BSEE from Brigham Young University.

E-mail address: mills@lcdm-eng.com

Steve Golson designed his first IC in 1982 in 4-micron NMOS. Since 1986 he has provided contract engineering services in VLSI design (full custom, semi-custom, gate array, FPGA); computer architecture and memory systems; and digital hardware design. He has extensive experience developing synthesis methodologies for large ASICs using a variety of design tools including Verilog and Synopsys. Other services include Synopsys and Verilog training classes, patent infringement analysis, reverse engineering, and expert witness testimony.

Steve holds a BS in Earth, Atmospheric, and Planetary Science from Massachusetts Institute of Technology. Hey, if a seismologist can design ASICs, it can’t be that hard!

E-mail address: sgolson@trilobyte.com

An updated version of this paper can be downloaded from the web sites: www.sunburst-design.com/papers, www.lcdm-eng.com or from www.trilobyte.com

(Data accurate as of August 12, 2003)