Synchronous Resets? Asynchronous Resets?
I am so confused!
How will I ever know which to use?

Clifford E. Cummings
Sunburst Design, Inc.

Don Mills
LCDM Engineering

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.

This paper will also detail an interesting synchronization technique using digital calibration to synchronize reset removal on a multi-ASIC design.

1.0 resets, Resets, RESETS, and then there's RESETS

One cannot begin to consider a discussion of reset usage and styles without first saluting the most common reset usage of all. This undesired reset occurs almost daily in systems that have been tested, verified, manufactured, and integrated into the consumer, education, government, and military environments. This reset follows what is often called "The Blue Screen of Death" resulting from software incompatibilities between the OS from a certain software company, the software programs the OS is servicing, and the hardware on which the OS software is executing.

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.

It is the opinion of the authors that in general, every flip-flop in an ASIC should be resetable whether or not it is required by the system. Further more, the authors prefer to use asynchronous resets following the guidelines detailed in this paper. There are exceptions to these guidelines. In some cases, when follower 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 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 among multiple clock zones.

In addition, when applying resets between multiple ASICs that require a specific reset release sequence, special techniques must be employed to adjust to variances of chip and board manufacturing. The final sections of this paper will address this latter issue.

2.0 General flip-flop coding style notes

2.1 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)[12]. 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 (q2, d, clk, rst_n);
  output q2;
  input  d, clk, rst_n;
  reg    q2, 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

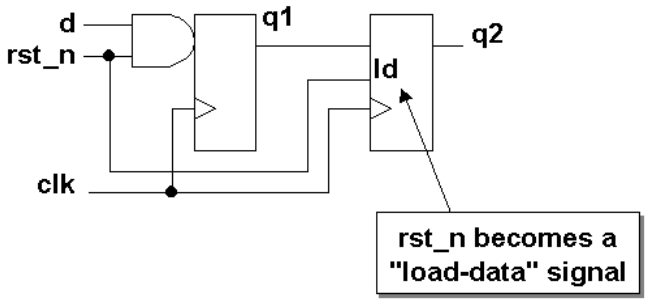

Figure 1 - Bad coding style yields a design with an unnecessary loadable flip-flop

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

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

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

Example 2a - Good Verilog 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

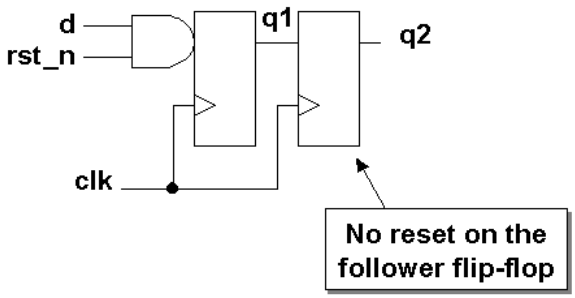

Figure 2 - Two different types of flip-flops, one with synchronous reset and one without

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 3.1.

2.2 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 macro level functional divisions within a given module/architecture. The exception to this guideline is that of follower flip-flops as discussed in the previous section (section 2.1) where multiple procedural blocks/processes are required to efficiently model the function itself.

2.3 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.

3.0 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 some advantages to using synchronous resets, but there are also disadvantages. The same is true for asynchronous resets. The designer must use the 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 ( q, co, d, ld, rst_n, clk);
  output [7:0] q;
  output       co;
  input  [7:0] d;
  input        ld, rst_n, clk;
  reg    [7:0] q;
  reg          co;

  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 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

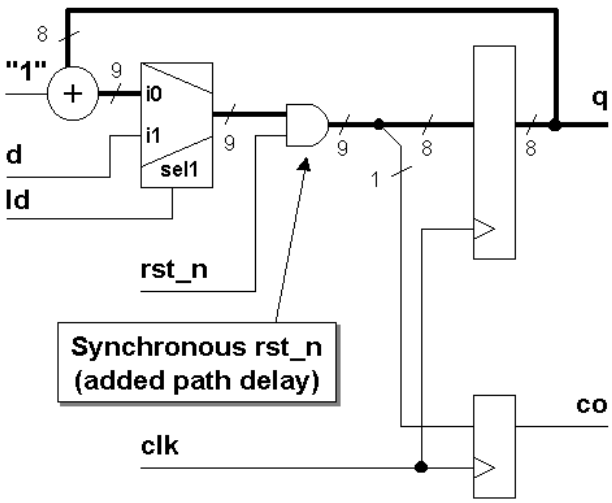

Figure 3 - Loadable counter with synchronous reset

A second style of synchronous resets is based on the availability of flip-flops with synchronous reset pins and the ability of the designer and synthesis tool to make use of those pins. This is sometimes the case, but more often the first style discussed above is the implementation used[22][26].

3.1 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 (q, d, clk, rst_n);
  output q;
  input  d, clk, rst_n;
  reg    q;

  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

```
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

For flip-flops designed with synchronous reset style #1 (reset is gated with data to the d-input), Synopsys has a switch that the designer can use to help infer flip-flops with synchronous resets.

Compiler directive: **`sync_set_reset`**

In general, the authors recommend only using Synopsys switches when they are required and make a difference; however, our colleague Steve Golson pointed out that the **`sync_set_reset`** directive does not affect the functionality of a design, so its omission would not be recognized until gate-level simulation, when discovery of a failure would require re-synthesizing the design late in the project schedule. Since this directive is only required once per module, adding it to each module with synchronous resets is recommended[19].

A few years back, another ESNUG contributor recommended adding the **`compile_preserve_sync_resets = "true"`** compiler directive[13]. Although this directive might have been useful a few years ago, it was discontinued starting with Synopsys version 3.4b[22].

3.2 Advantages of synchronous resets

Synchronous reset 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 reset can be much easier to work with when using cycle based simulators. For this very reason, synchronous resets are recommend in section 3.2.4(2nd edition, section 3.2.3 in the 1st edition) of the Reuse Methodology Manual (RMM)[18].

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.

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 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.

3.3 Disadvantages of synchronous resets

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[14].

A designer must work with pessimistic vs. optimistic simulators. This can be an issue 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, based 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 addressed this issue. Most simulators will not resolve some X-logic conditions and therefore block out the synchronous reset[5][6][7][8][9][10][11][12][13][20].

By it's 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. 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 a tristate bus when a chip is powered up, the chip must have a power on asynchronous reset[17].

4.0 Asynchronous resets

Asynchronous resets are the authors preferred reset approach. However, asynchronous resets alone can be very dangerous. 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 5.0.

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.)

4.1 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[4]. 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 (q, d, clk, rst_n);
  output q;
  input  d, clk, rst_n;
  reg    q;

  // 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

```
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[16].

One ESNUG contributor[15] indicates that sometimes **`set_resistance 0`** on the reset net might also be needed.

And our colleague, Steve Golson, has pointed out that you can **`set_resistance 0`** on the net, or 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[24] 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."

Another 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[21][25].

4.2 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[4].

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

  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

4.3 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. Of course this argument does not hold if the vendor library has flip-flops with synchronous reset inputs *and* the designer can get Synopsys to actually use those pins. 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 ( q, co, d, ld, rst_n, clk);
  output [7:0] q;
  output       co;
  input  [7:0] d;
  input        ld, rst_n, clk;
  reg    [7:0] q;
  reg          co;

  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 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

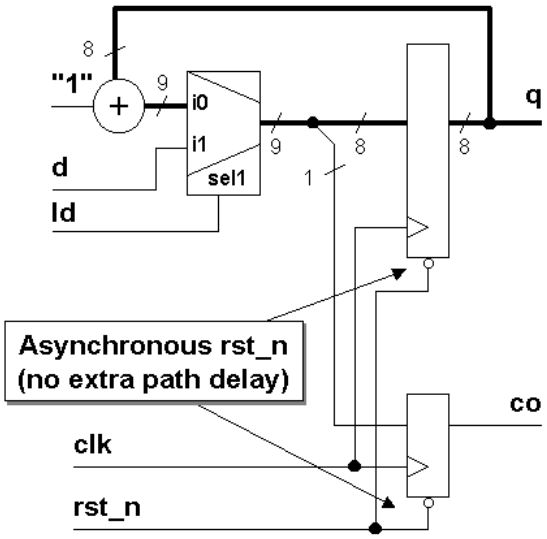

Figure 4 - Loadable counter with asynchronous reset

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.

4.4 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.

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.

5.0 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 5, 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.

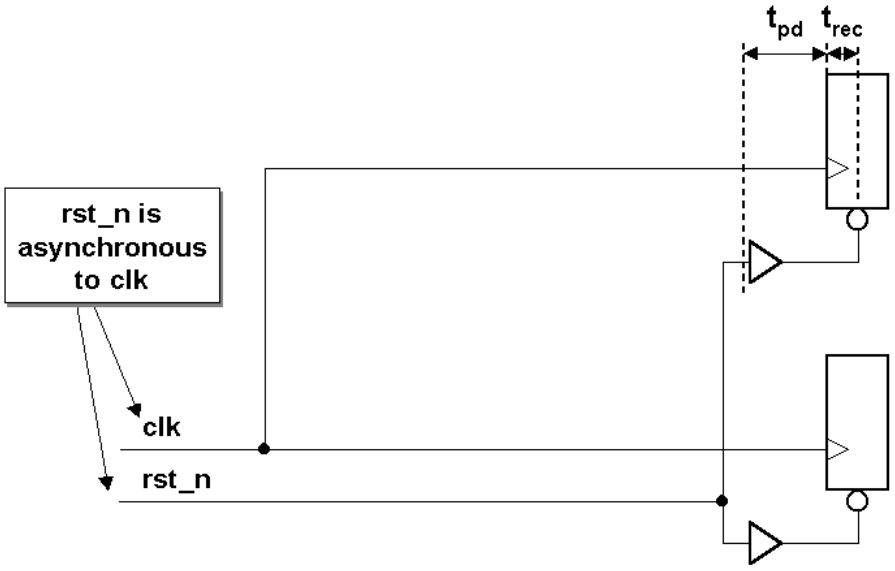

Figure 5 - Asynchronous reset removal recovery time problem

5.1 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[17] 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.

5.2 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.

6.0 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 6 is designed to take advantage of the best of both asynchronous and synchronous reset styles.

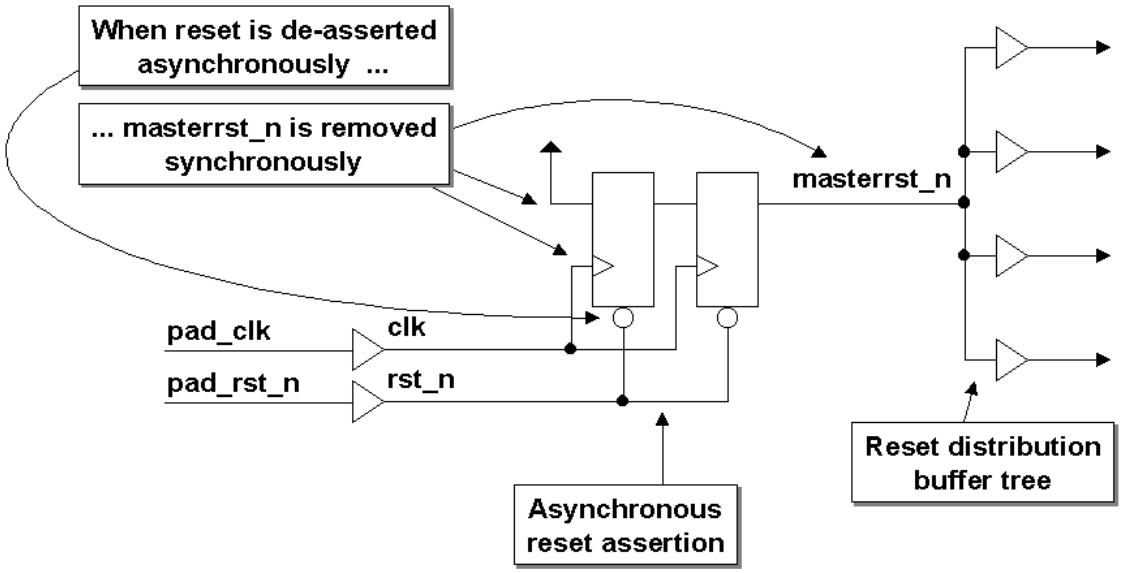

Figure 6 - Reset Synchronizer block diagram

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 4.4, these synchronization flip-flops must be kept off of the scan chain.

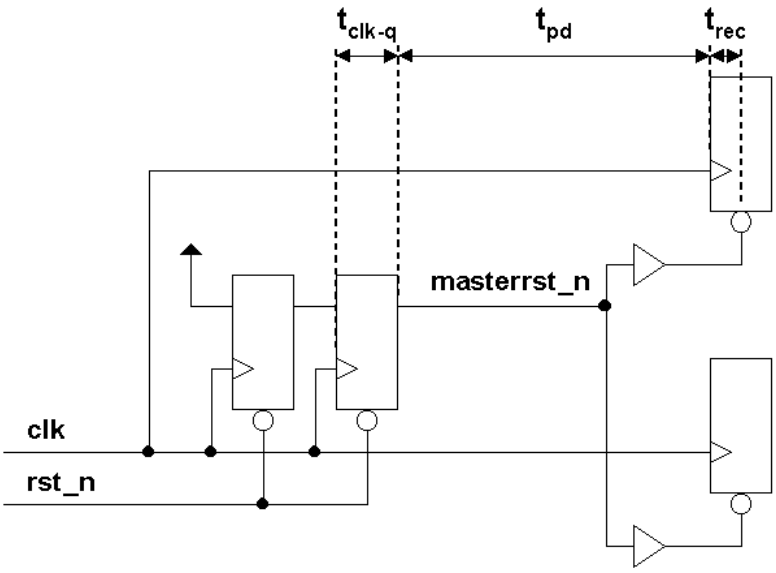

Figure 7 - Predictable reset removal to satisfy reset recovery time

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 7.

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

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

```
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

7.0 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 8. The timing requirements for reset tree are common for both synchronous and asynchronous reset styles.

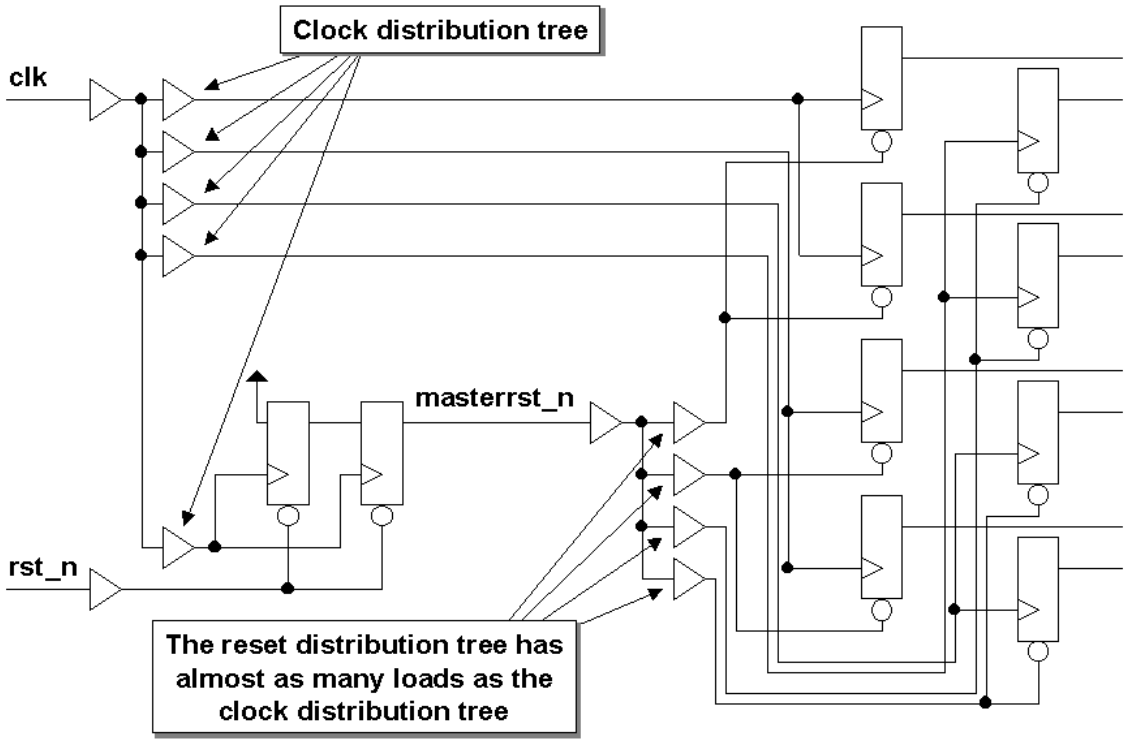

Figure 8 - Reset distribution tree

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 9. 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.

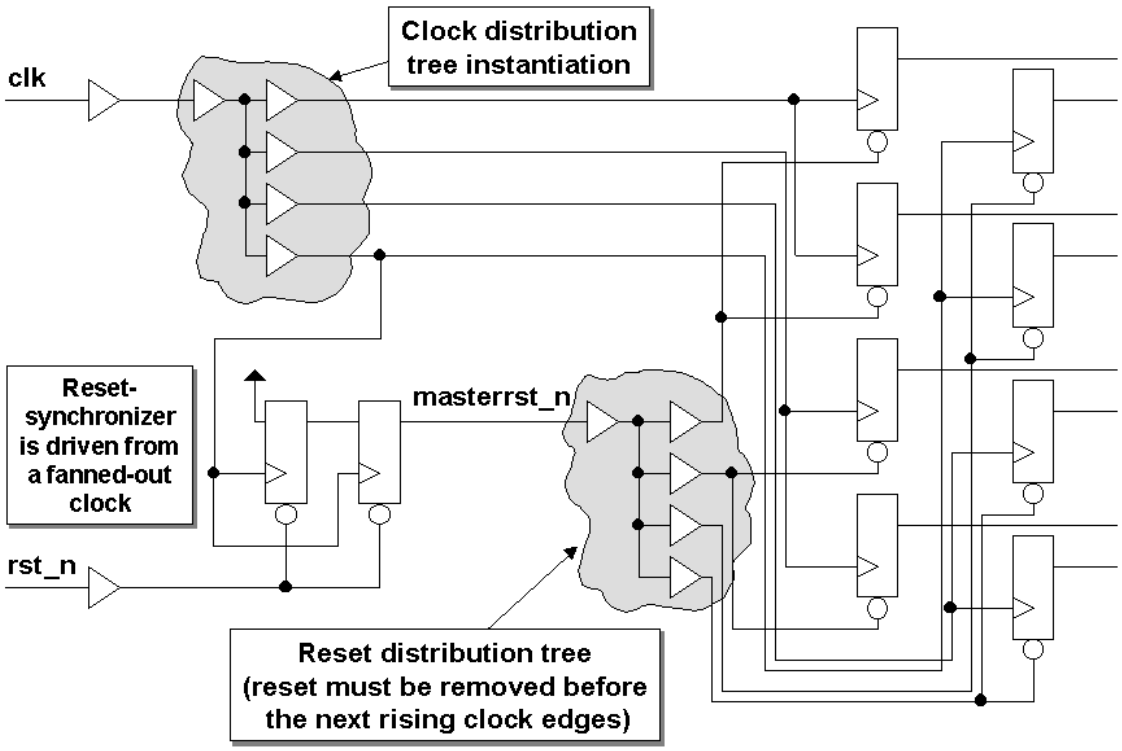

Figure 9 - Reset tree driven from a delayed, buffered clock

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 10. 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.

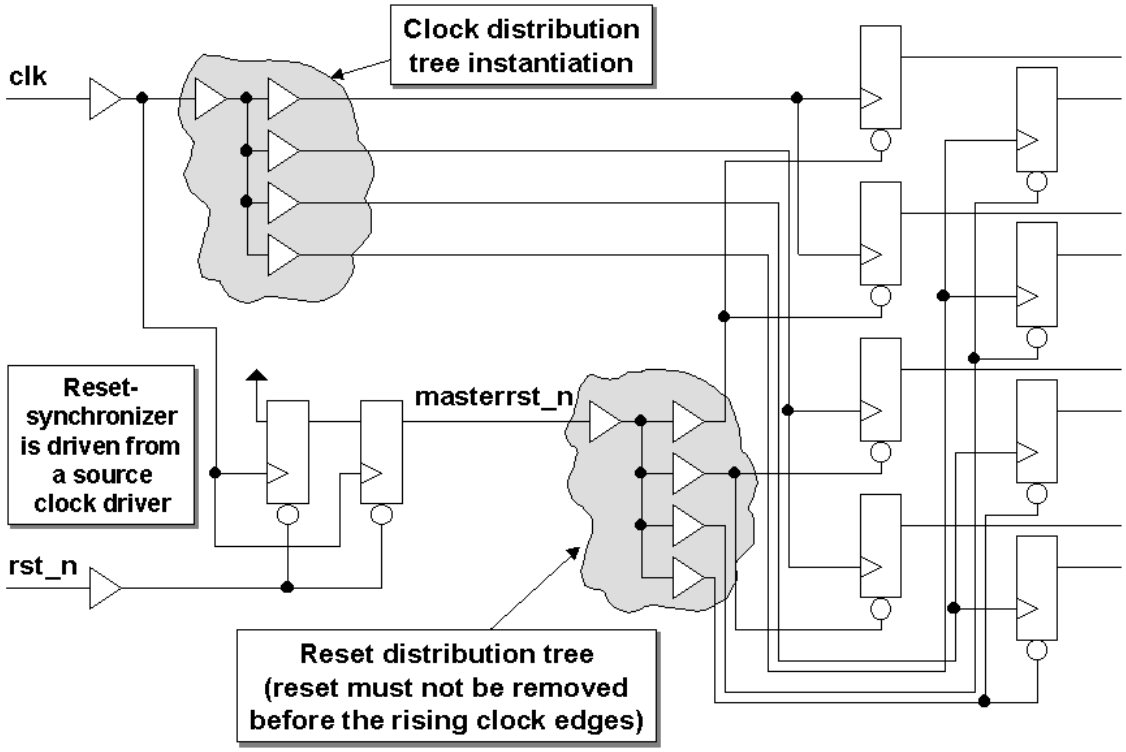

Figure 10 - Reset synchronizer driven in parallel to the clock distribution tree

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

8.0 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 11 shows the implementation of this approach.

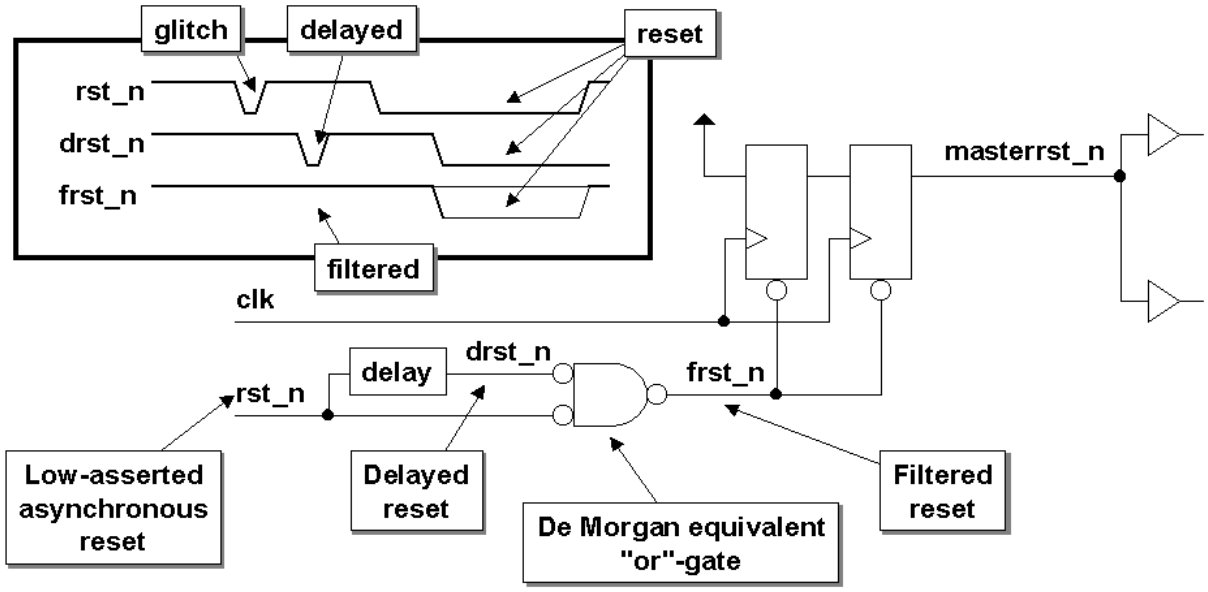

Figure 11 - Reset glitch filtering

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.

9.0 DFT for asynchronous resets

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.

10.0 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.

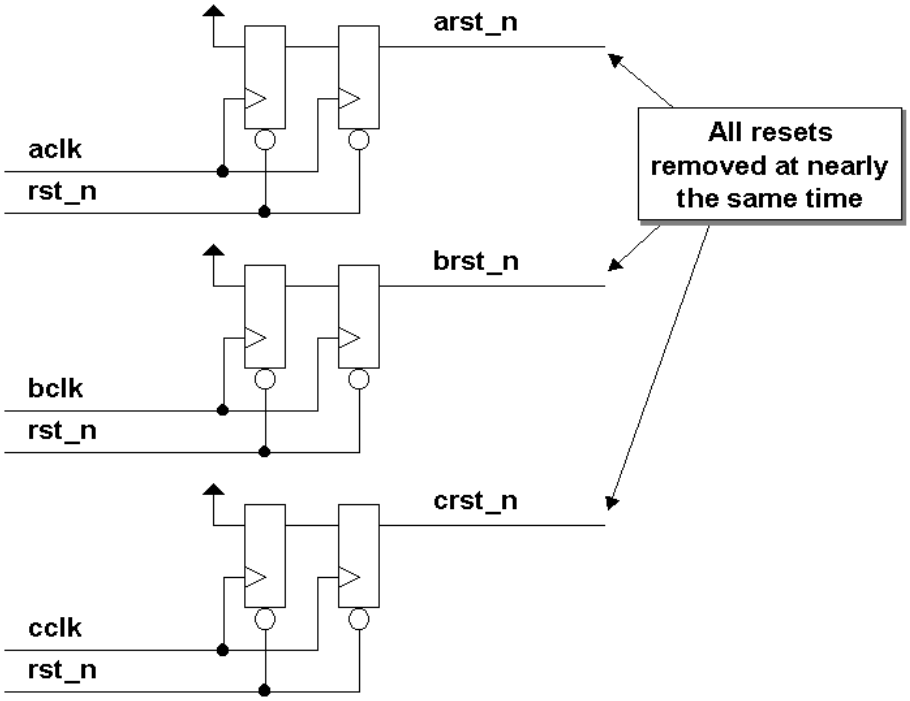

Figure 12 - Multi-clock reset removal

10.1 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 12 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.

10.2 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 13 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.

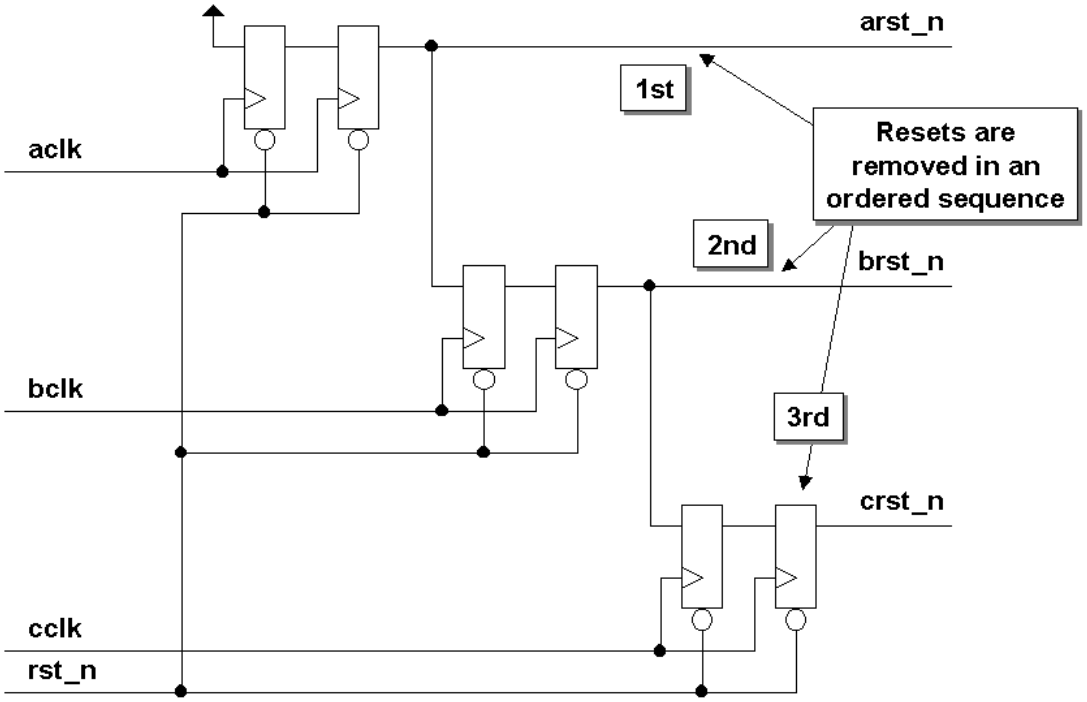

Figure 13 - Multi-clock ordered reset removal

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.

11.0 Multi-ASIC reset synchronization

There are designs with multiple ASICs that require precise synchronization of reset removal across all of the multiple ASICs. One approach to satisfy this type of design, described in this section, is to use a different asynchronous reset synchronization scheme, one that only requires one reset removal flip-flop instead of the two flip-flops described in section 6.0, plus a digitally calibrated synchronization delay to properly sequence reset removal from the multiple ASICs.

Consider the actual design of a data acquisition board on a Digital Storage Oscilloscope (DSO). In rudimentary terms, a DSO is a test instrument that probes an analog signal, continuously does sampling and Analog-to-Digital (A2D) conversion of the signal, and continuously stores the sampled digital data into memory as fast as it can. After the requested trigger condition occurs, the rest of the data associated with the trigger condition is stored to memory and then DSO control logic (typically a commercial microprocessor) accesses the data and draws a waveform of the data values onto a screen for visual inspection.

For an actual design of this type, the data acquisition board contained four digital demultiplexer (demux) ASICs, each of which captured one-fourth of the **`datain`** samples to send to memory, as shown in Figure 14.

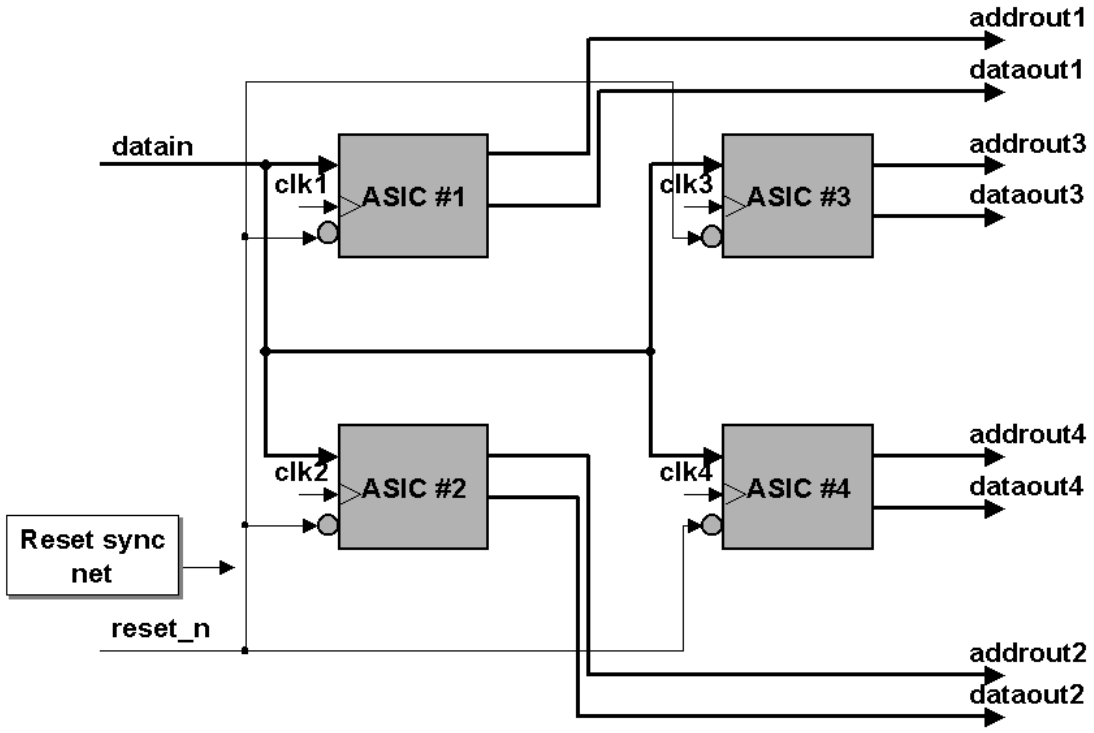

Figure 14 - Multi-ASIC design with synchronized reset removal problem

For this digital acquisition system, as soon as reset is removed, the ASICs must start capturing data and generating memory addresses to write the data to memory. Both data acquisition and address generation are continuously running, capturing data samples and overwriting previous written memory locations until a trigger circuit causes the address counters to stop and hold the data that has been most recently captured. Frequently, the trigger is set to hold and show 90% of the waveform as pre-trigger data and 10% of the waveform as post-trigger data. Since it is generally impossible to predict when the trigger will occur, it is necessary to continuously acquire data after reset removal until a trigger signal stops the data acquisition.

The approach that was used in this design to do high-speed data acquisition was to use four demux ASICs that capture every fourth point of the digitized waveform. Since the demux ASICs typically ran at very fast clock rates, and since each demux ASIC also had to generate accompanying address count values to store the data samples to memory, it was important that all four demux ASICs start their respective address counters in the correct sequence to insure that the data samples stored in memory could be easily read-back to draw waveforms on the DSO display.

The problem with this type of design was to accurately remove the reset signal from the four ASIC devices at the same time (in the same relative clock period) so that the four ASICs captured the correctly sequenced data samples that corresponded to address-#0 on all four ASICs, followed by address-#1 on all four ASICs, etc., so that the data stored to memory could be read back from memory (after triggering the DSO) in the correct sequence to display an accurate waveform on the DSO screen.

For this type of design, there are a number of factors that work against correct reset-removal and hence correct sequencing of the data values being written to memory.

First, for very high-speed designs (DSOs are typically very high-speed designs in order to capture an adequate number of data samples while probing other high-speed circuits), the relative board trace length of reset signals to the four ASICs would have to be held to a very tight tolerance; hence, board layout is an issue.

Second, process variations within or between batches of manufactured ASICs can create delays that exceed the ultra-short ASIC clock periods. Choosing four ASICs to insert during manufacture can result in selection of four

devices with different relative delays being placed on the same data acquisition board. The relative process speeds of the four ASICs placed on a board cannot be guaranteed (which of the four ASICs will always be the fastest? Who knows!)

Third, temperature swings in different test environments can also add to differences in delays. Relative positioning of the ASICs inside of a DSO enclosure might account for significant differences in temperature for this high-speed system.

Fourth, removing the covers of the DSO to troubleshoot prototypes could introduce different temperature variations across the four ASICs than when the covers are closed.

For the actual design, a common reset signal (**`reset_n`**) was routed to all four demux ASICs to assert reset, but the reset signal did not de-assert reset from the demux ASICs. A separate sync signal was used to flag reset removal permission on each demux ASIC.

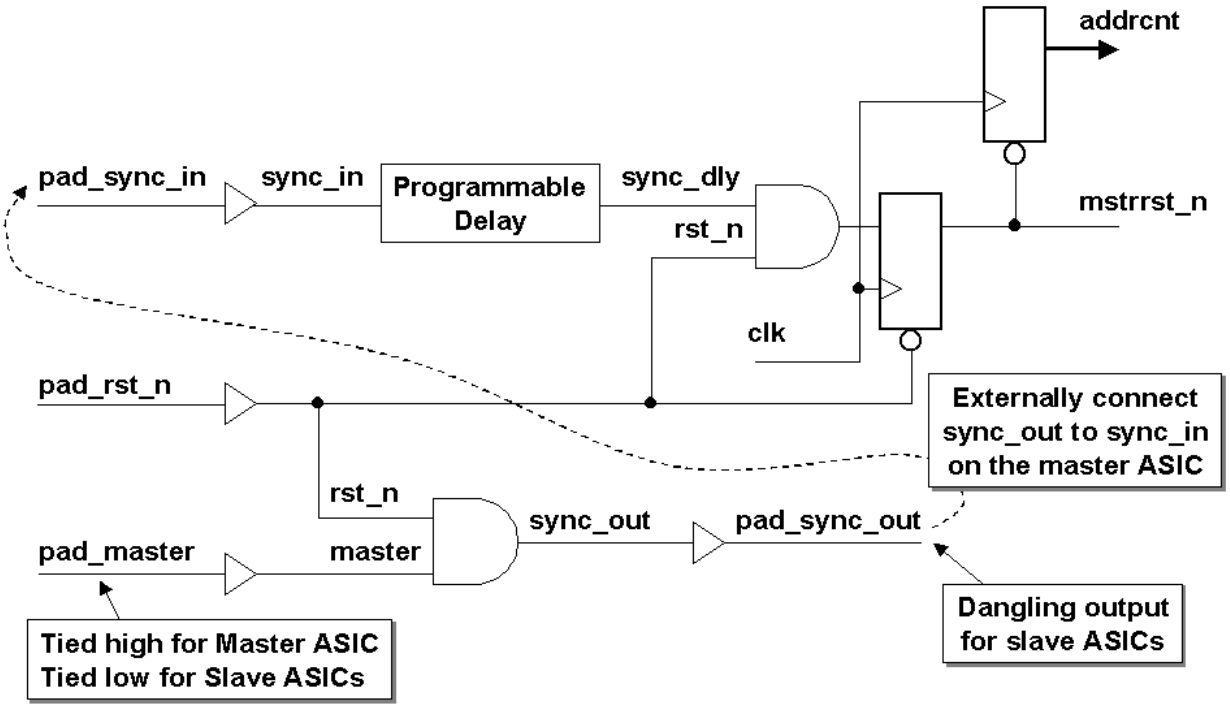

Figure 15 - Reset removal synchronization logic block diagram

The multi-ASIC reset removal synchronization logic is handled using the logic shown in Figure 15. This logic is common to both master and slave ASICs.

Asserting reset (**`rst_n`** going low in Figure 15) asynchronously resets the master reset signal, **`mstrrst_n`**, which is driven through a reset-tree to the rest of the resetable logic on all ASICs (both master and slave ASICs); therefore, reset is asynchronous and immediate.

Each ASIC has three pins dedicated to reset-removal synchronization.

The first pin on each ASIC is a dedicated master/slave selection pin. When this pin is tied high, the ASIC is placed into master mode. When the pin is tied low, the ASIC is placed into slave mode.

The second pin on each ASIC is the **`sync_out`** pin. On the slave ASICs, the **`sync_out`** pin is unused and left dangling. The master ASIC generates the **`sync_out`** pulse when reset is removed (when **`reset_n`** goes high). The **`sync_out`** signal is driven out of the master ASIC and is tied to the **`sync_in`** input on both master and

slave ASICs through board-trace connections. The **sync_out** pin is the pin that controls reset removal on both the master ASIC and the slave ASICs.

The third pin on each ASIC is the **sync_in** pin. The **sync_in** pin is the input pin that is used to control reset removal on both master and slave ASICs. The **sync_in** signal is connected to a programmable delay block and is then enabled by a high-assertion on the reset input, that is then passed to a synchronous reset removal flip-flop. The next rising clock edge on the ASIC will cause the reset to be synchronously removed, permitting the address counters on each ASIC to start counting in a synchronized and orderly manner.

The problem, as explained earlier, is to insure that the **sync_in** signal removes the reset on the four ASICs in the correct order.

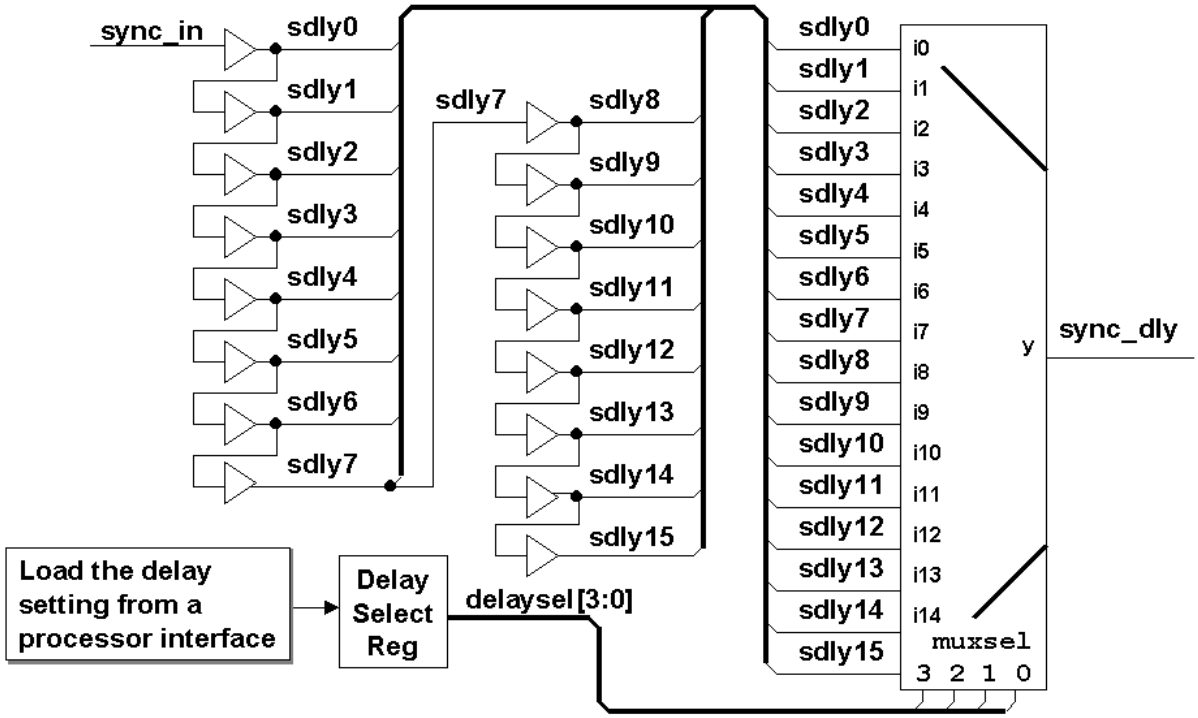

Figure 16 - Programmable digital delay block diagram

The programmable digital delay block, shown in Figure 16, is a set of delay stages connected in series with each delay-stage output driving both the next delay-stage input and an input on a multiplexer. The delay stages could be simple buffers or they could be pairs of inverters. The number of delay stages selected was equal to almost three ASIC clock cycles.

A processor interface is used to program the delay select register, which enables the multiplexer select lines to choose which delayed **sync_in** signal (**sdly0** to **sdly15**) would be driven to the mux output and used to remove the reset on the ASIC.

In order to determine the correct delay settings for each ASIC, a software digital calibration technique was employed.

To help calibrate the demux ASICs, as well as other analog devices on the data acquisition board, the board was designed to capture a selectable on-board ramp signal through the data acquisition path. The ramp signal was used to calibrate the delays on the four demux ASICs.

In Figure 17-Figure 19, the software programmable, digital calibration procedure is shown with just two of the four demux ASICs.

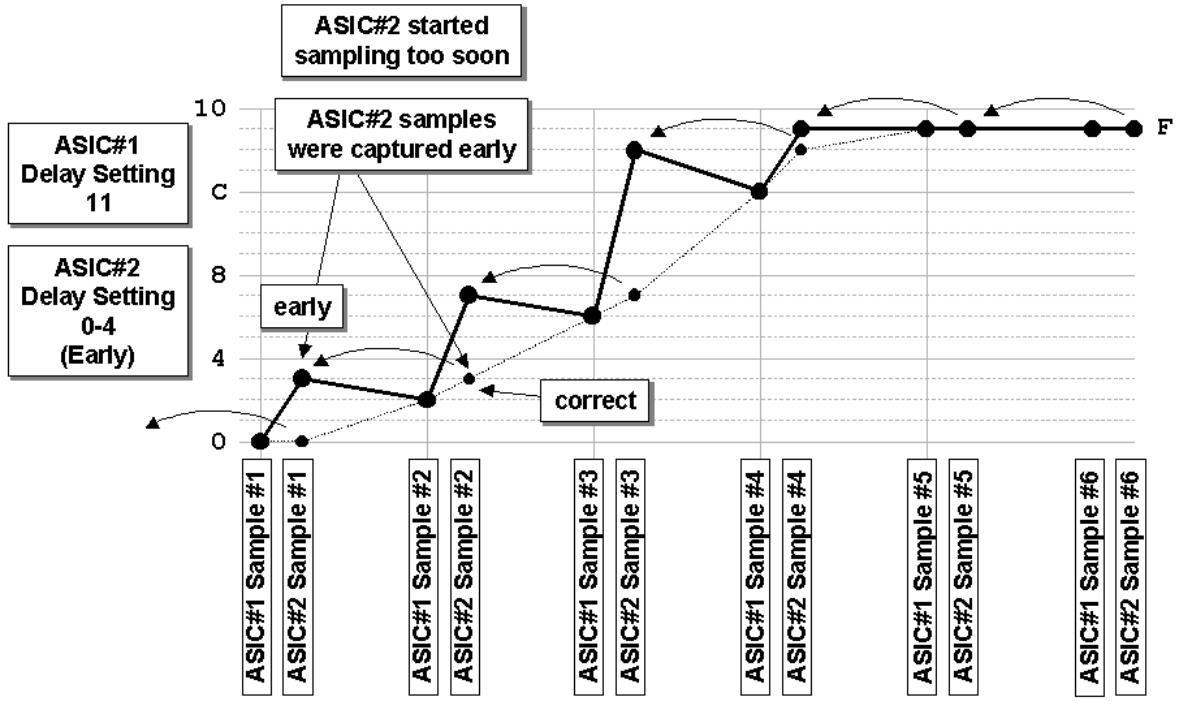

Figure 17 - Two-ASIC reset-removal calibration - early data sampling on ASIC #2

ASIC #1 is given the initial delay setting of 11 (to drive the **`sdly11`** signal to the mux output). ASIC #2 is given another delay setting and a ramp signal is captured by the data acquisition board. If the delay setting on ASIC #2 is too small, such as a delay value of 0-4 as shown in Figure 17, the ramp values captured by ASIC #2 will be sampled early compared to the data points sampled by ASIC #1. This is manifest by the fact that each ramp data point captured by ASIC #2 is larger than the next data point captured by ASIC #1.

If the delay setting on ASIC #2 is in the correct range, such as a delay value of 5-11 as shown in Figure 18, the ramp values captured by ASIC #2 will be sampled in the correct order compared to the data points sampled by ASIC #1. This is manifest by the fact that each ramp data point captured by ASIC #2 is larger than the previous data point captured by ASIC #1 and smaller than the next data point captured by ASIC #1.

If the delay setting on ASIC #2 is too large, such as a delay value of 12-15 as shown in Figure 19, the ramp values captured by ASIC #2 will be sampled late compared to the data points sampled by ASIC #1. This is manifest by the fact that each ramp data point captured by ASIC #2 is smaller than the next data point captured by ASIC #1.

Once the correct range is determined for ASIC #2, the center point in the range is chosen to be the ASIC #2 **`sync_in`** delay setting. The center point is the safest setting in the range since this setting is approximately a half-cycle between the previous and next rising clock edges for the reset-removal synchronization flip-flop.

After determining the correct ASIC #2 setting, the correct ASIC #1 range surrounding the initial setting (the setting of 11 is used in Figure 17) must be determined to find the correct ASIC #1 mid-point setting. After determining the correct ASIC #1 setting, a similar process is used to find the correct delay setting for ASIC #3, followed by finding the correct setting for ASIC #4.

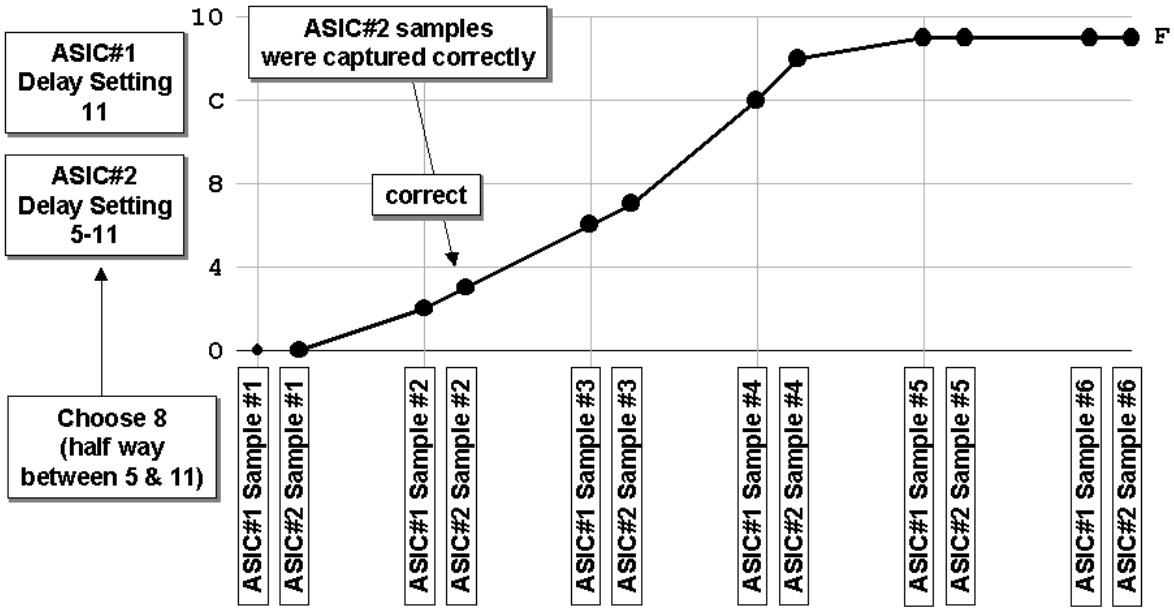

Figure 18 - Two-ASIC reset-removal calibration - correctly timed data sampling on ASIC #2

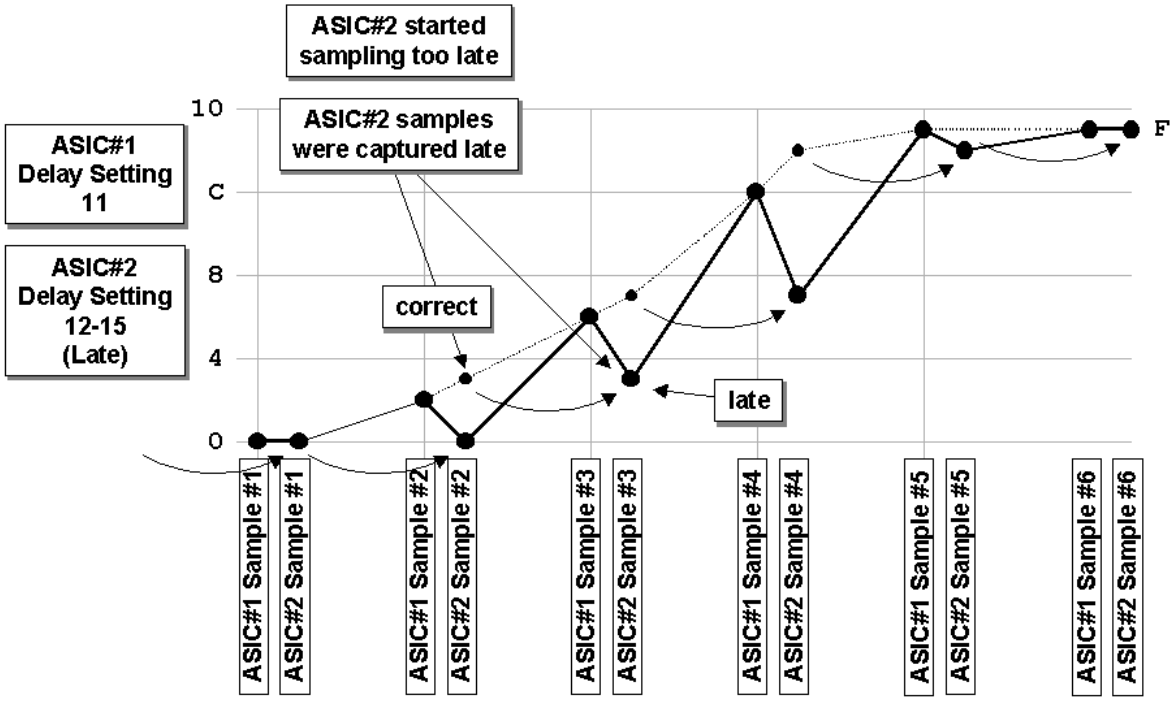

Figure 19 - Two-ASIC reset-removal calibration - late data sampling on ASIC #2

After digital calibration, there was no need to use a second reset-removal synchronization flip-flop because a mid-clock setting was used to insure that the flip-flop recovery time was met and to insure that no metastability problems would arise.

The full block diagram of the four-demux ASIC design with master/slave pin and **`sync_in`**/**`sync_out`** pins on each ASIC and how they were connected is shown in Figure 20.

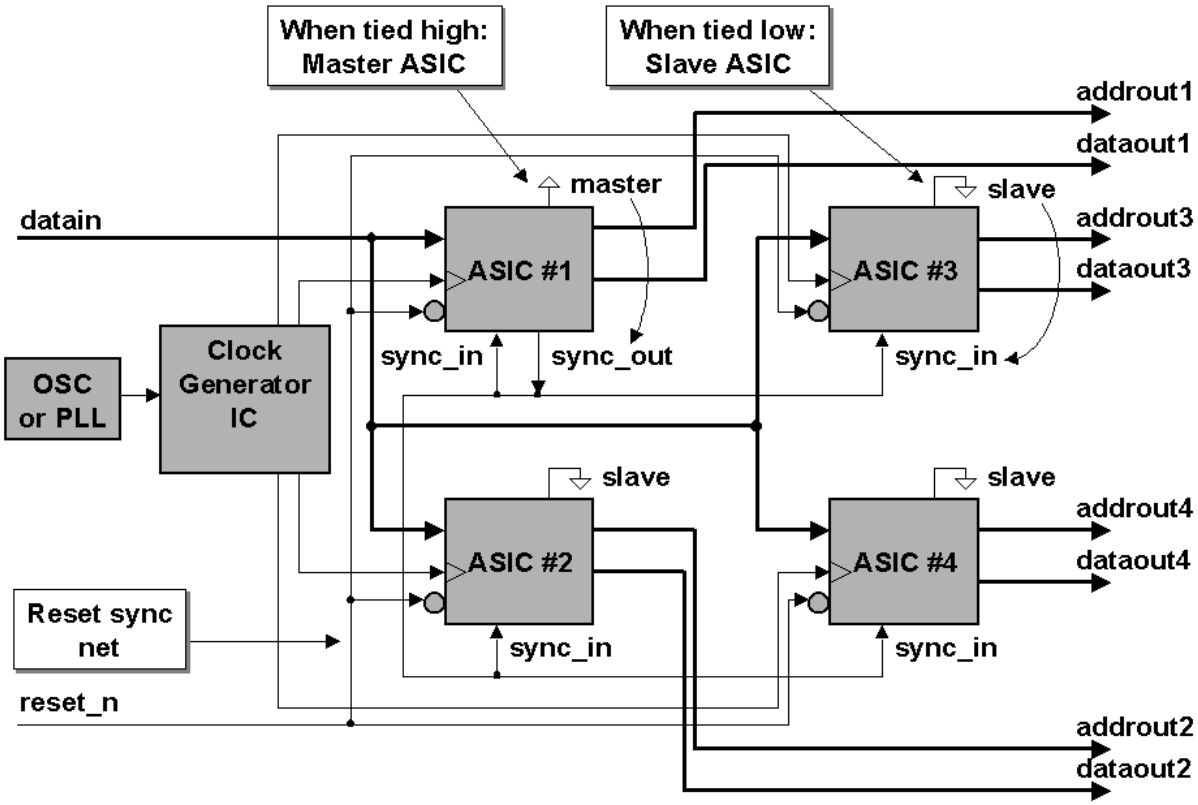

Figure 20 - Multi-ASIC synchronized reset removal solution

In the actual design, after determining a valid set of mid-point delay settings for the four ASICs on one of the data acquisition prototype boards, these values were programmed into a ROM and used as initial settings for all manufactured boards and variations from the initial settings were tracked. What was interesting was that the calibrated delay values for each board rarely strayed more than one or two delay stages up or down from the original settings of the initial data acquisition prototype board.

12.0 Conclusions

Using asynchronous resets is the surest way to 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.

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] 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

[5] ESNUG #60, Item 1- http://www.deepchip.com/posts/0060.html

[6] ESNUG #240, Item 7- http://www.deepchip.com/posts/0240.html

[7] ESNUG #242, Item 6 - http://www.deepchip.com/posts/0242.html

[8] ESNUG #243, Item 4 - http://www.deepchip.com/posts/0243.html

[9] ESNUG #244, Item 5 - http://www.deepchip.com/posts/0244.html

[10] ESNUG #246, Item 5 - http://www.deepchip.com/posts/0246.html

[11] ESNUG #278, Item 7 - http://www.deepchip.com/posts/0278.html

[12] ESNUG #280, Item 4 - http://www.deepchip.com/posts/0280.html

[13] ESNUG #281, Item 2 - http://www.deepchip.com/posts/0281.html

[14] ESNUG #355, Item 2 - http://www.deepchip.com/posts/0355.html

[15] ESNUG #356, Item 4 - http://www.deepchip.com/posts/0356.html

[16] ESNUG #373, Item 6 - http://www.deepchip.com/posts/0373.html

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

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

[19] Steve Golson, personal communication

[20] Synopsys SolvNet, Doc Name: METH-933.html, "Methodology and limitations of synthesis for synchronous set and reset," Updated 09/07/2001.

[21] Synopsys SolvNet, Doc Name: Physical_Synthesis-231.html, "Handling High Fanout Nets in 2001.08" Updated: 11/01/2001.

[22] Synopsys SolvNet, Doc Name: Star-15.html, "Is the compile_preserve_sync_reset Switch Still Valid?," Updated: 09/07/2001.

[23] Synopsys SolvNet, Doc Name: Synthesis-452.html, "Why can't I synthesize synchronous reset flip-flops?," Updated: 08/16/1999.

[24] Synopsys SolvNet, Doc Name: Synthesis-780.html, "How can I use the high_fanout_net_threshold commands to simplify the net delay calculation?" Updated: 01/25/2002.

[25] Synopsys SolvNet, Doc Name: Synthesis-482109.html, "How to Eliminate Transition Time Calculation Side Effects From Arcs That Are Fal" Updated: 08/11/1997

[26] Synopsys SolvNet, Doc Name: Synthesis-799.html, "Data and Synchronous Reset Swapped," Updated: 05/01/2001.

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 19 years of ASIC, FPGA and system design experience and nine years of Verilog, synthesis and methodology training experience.

Mr. Cummings, a member of the IEEE 1364 Verilog Standards Group (VSG) since 1994, chaired the VSG Behavioral Task Force, which was charged with proposing enhancements to the Verilog language. Mr. Cummings is also a member of the IEEE Verilog Synthesis Interoperability Working Group.

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

E-mail Address: cliffc@sunburst-design.com

Don Mills is an independent EDA consultant, ASIC designer, and Verilog/VHDL trainer with 16 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 is currently chair of the Europe SNUG 2001- present.

Don holds a BSEE from Brigham Young University.

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

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

(Data accurate as of February 5th, 2002)