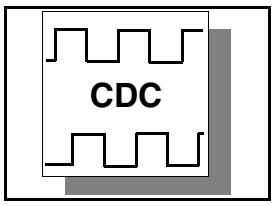
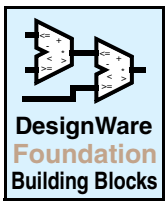




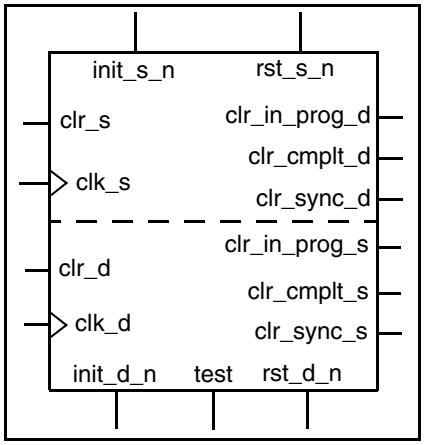
DW_reset_sync

Reset Sequence Synchronizer

Version, STAR and Download Information: IP Directory

Features and Benefits

- Interface between dual asynchronous clock domains
- Coordinated clearing between clock domains
- Parameterized number of synchronizing stage
- All outputs registered (except "in progress" outputs which are configurable)
- Parameterized test feature
- Ability to model missampling of data on source clock domain



Description

This synchronizer is a mechanism for a dual clock domain design to cleanly coordinate localized clearing of sequential elements in each domain. A clearing action initiated from either clock domain sets in motion a sequence of events which generates status back to each domain that allows for resetting of sequential elements. The sequence is such that synchronized notification is made to each domain. As a result, when each exits its clearing state, each domain knows when it is safe to begin normal operation again without the risk of system inconsistencies occurring between clock domains.

In general, the outputs to each domain are registered with the exception of `clr_in_prog_s` and `clr_in_prog_d`. `clr_in_prog_s` and `clr_in_prog_d` can both be registered or not based on the *reg_in_prog* parameter setting.

A unique built-in verification feature allows the designer to turn on a random sampling error mechanism that models skew between bits of the incoming data bus from the source domain (see section “Simulation Methodology” for more details). This facility provides an opportunity for determining system robustness during the early development phases and without having to develop special test stimulus.

Note

As of Release DWBB_201109.1 (F-2011.09-SP1), the DW_reset_sync clearing behavior has been enhanced to lengthen the assertion of `clr_in_prog_d`. Figure 1-2, Figure 1-3, Figure 1-4 and Figure 1-5 have been updated to reflect the new behavior. In addition, Figure 1-7 and Figure 1-8 have been added to show an example of the difference in behavior.

The main purpose of enhancing the DW_reset_sync was to lengthen the `clr_in_prog_d` assertion. However, in doing so, the timing behavior of outputs `clr_in_prog_s` and `clr_cmplt_d` have also been slightly altered.

Table 1-1 Pin Description

Pin Name	Width	Direction	Function
clk_s	1	Input	Source Domain clock source
rst_s_n	1	Input	Source Domain asynchronous reset (active low)
init_s_n	1	Input	Source Domain synchronous reset (active low)
clr_s	1	Input	Source Domain clear
clr_sync_s	1	Output	Source Domain clear for sequential logic (single `clk_s` cycle pulse)
clr_in_prog_s	1	Output	Source Domain clear sequence in progress
clr_cmplt_s	1	Output	Source Domain that clear sequence complete (single `clk_s` cycle pulse)
clk_d	1	Input	Destination Domain clock source
rst_d_n	1	Input	Destination Domain asynchronous reset (active low)
init_d_n	1	Input	Destination Domain synchronous reset (active low)
clr_d	1	Input	Destination Domain clear
clr_in_prog_d	1	Output	Destination Domain clear sequence in progress
clr_sync_d	1	Output	Destination Domain clear for sequential logic (single `clk_d` cycle pulse)
clr_cmplt_d	1	Output	Destination Domain that clear sequence complete (single `clk_d` cycle pulse)
test	1	Input	Scan test mode select input

Table 1-2 Parameter Description

Parameter	Values	Description
f_sync_type	0 to 4 Default: 2	Forward Synchronization Type Defines type and number of synchronizing stages: 0 – single clock design `clk_d` = `clk_s` 1 – negedge to posedge sync in destination domain 2 – posedge to posedge sync destination domain 3 – 3 posedge flops in destination domain 4 – 4 posedge flops in destination domain
r_sync_type	0 to 4 Default: 2	Reverse Synchronization Type (Destination to Source Domains) 0 – single clock design `clk_d = clk_s` 1 – negedge to posedge sync source domain 2 – posedge to posedge sync source domain 3 – 3 posedge flops in source domain 4 – 4 posedge flops in source domain
clk_d_faster	0 to 15 Default: 1	Obsolete parameter. The value setting is ignored. This parameter is kept in place only for backward compatibility.

Table 1-2 Parameter Description (Continued)

Parameter	Values	Description
reg_in_prog	0 or 1 Default: 1	Register the `clr_in_prog_s` and `clr_in_prog_d` Outputs 0 – unregistered 1 – registered
tst_mode	0 to 2 Default: 0	Test Mode 0 – no 'latch' is inserted for scan testing 1 – insert negative-edge capturing flip-flip on `data_s` input vector when `test` input is asserted. 2 – insert hold latch using active-low latch
verif_en*	0 to 2 Default: 1	Verification Enable Control 0 = no sampling errors inserted 1 = sampling errors randomly inserted with 0 or up to 1 destination clock cycle delays 2 = sampling errors randomly inserted with 0, 0.5, 1, or 1.5 destination clock cycle delays 3 = sampling errors randomly inserted with 0, 1, 2, or 3 destination clock cycle delays 4 = sampling errors randomly inserted with 0 or up to 0.5 destination clock cycle delays *See the Simulation Model section for more information about *verif_en*.

Table 1-3 Synthesis Implementations

Implementation Name	Function	License Feature Required
rtl	Synthesis model	DesignWare

Table 1-4 Simulation Models

Model	Function
DW03.DW_RESET_SYNC_CFG_SIM	Design unit name for VHDL simulation without missamplings
DW03.DW_RESET_SYNC_CFG_SIM_MS	Design unit name for VHDL simulation with missamplings enabled. (See "Simulation Methodology" below for details.)
dw/dw03/src/DW_reset_sync_sim.vhd	VHDL simulation model source code (modeling RTL)
dw/sim_ver/DW_reset_sync.v	Verilog simulation model source code

Simulation Methodology

For simulation, there are two methods available. One method is to utilize the simulation models as they emulate the RTL model. The other method is to enable modeling of random skew between bits on the `data_s` bus of the underlying DW_sync components source domain by the destination domain (denoted as “missampling” here on out) for signals traveling in the direction from source to destination domains. The same principle is applied for signals travelling in the opposite direction where skew is placed between bits on the `data_s` of the underlying DW_sync components destination domain by the source domain. When using the simulation models purely to behave as the RTL model, no special configuration is required. When using the simulation models to enable missampling, unique considerations must be made between Verilog and VHDL environments.

For Verilog simulation enabling missampling a preprocessing variable named DW_MODEL_MISSAMPLES must be defined as follows:

```
`define DW_MODEL_MISSAMPLES
```

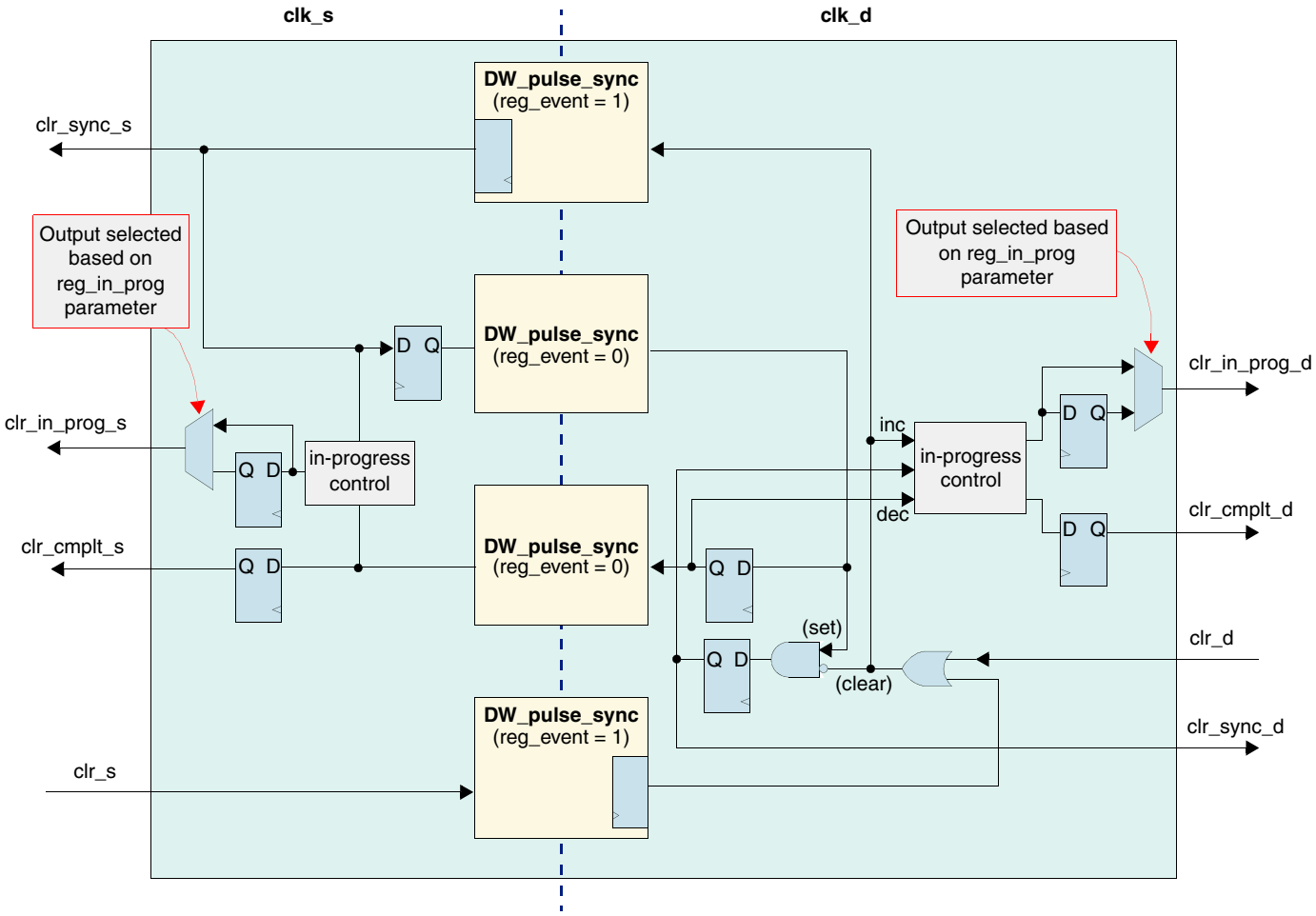
Once `DW_MODEL_MISSAMPLES is defined, the value of the *verif_en* parameter comes into play and configures the simulation model. Note: If `DW_MODEL_MISSAMPLES is not defined, the Verilog simulation model behaves as if *verif_en* was set to '0'.

For VHDL simulation enabling missampling, an alternative simulation architecture is provided. This architecture is named "sim_ms". The parameter *verif_en* only has meaning when utilizing the "sim_ms". That is, when binding the "sim" simulation architecture the *verif_en* value is ignored and the model effectively behaves as though *verif_en* is set to 0. See the "HDL Usage Through Component Instantiation - VHDL" on page 14 for an example utilizing each architecture.

Block Diagram

Figure 1-1 is the block diagram for the Reset Sequence Synchronizer.

Figure 1-1 DW_reset_sync Basic Block Diagram



Reset Considerations

System Resets (synchronous and asynchronous)

The instantiated DW_pulse_sync modules convert the clearing inputs (`clr_s` and `clr_d`) to 'toggle' events (from the source and destination domains, respectively). These 'toggle' events do not rely on state value but rather on state change from the originating domain. As a result, an assertion of `rst_s_n` or `init_s_n` could cause an erroneous 'toggle' event to occur that translates over into the destination domain in a 'false' assertion of `clr_sync_d`. Similarly, an assertion of `rst_d_n` or `init_d_n`, could result in a 'false' assertion of `clr_sync_s`. (Refer to Figure 1-6 as an example of a 'false' `clr_sync_d` event occurring as a by-product of asserting `rst_s_n`.) This inherently could cause inconsistencies when the activation of these resets is not coordinated between the two domains at the system level. Therefore, as a requirement to insure clean set and release of the reset state, both source and destination domains must be, at some point, in the active reset state simultaneously when performing system resets. That is, when asserting `rst_s_n` and/or `init_s_n`, then `rst_d_n` and/or `init_d_n` must also be asserted at the same time and vice versa. As a general requirement the length of the system reset signal(s) assertion should be a minimum of 4 clock cycles of the slowest clock between the two domains. System reset signals for both clock domains, when asserted, should overlap for a minimum of *f_sync_type*+1 or *r_sync_type*+1 cycles (whichever is larger) of the slowest clock of the two domains.

Besides satisfying simultaneous assertion of each domains system reset signals for a minimum number of cycles, the timing of the assertion between these signals needs some consideration. To prevent erroneous `clr_sync_s` and `clr_sync_d` pulses from occurring when system resets are asserted, it is recommended that:

1. If the source domain reset is asserted first, then the destination domain should assert its reset within *f_sync_type*+1 `clk_d` cycles from the time the assertion of the source domain reset occurred OR
2. If the destination domain reset is asserted first, then the source domain should assert its within *r_sync_type*+1 `clk_s` cycles from the time the assertion of the destination domain reset occurred.
3. If both domains can tolerate a false `clr_sync_s` or `clr_sync_d` (whichever the case) during system reset conditions, then this recommendation can be ignored as long as both clock domains eventually have overlapping active reset conditions.

There are no restrictions on when to release the reset condition on either side. However, to be completely safe, it is recommended, though not required, to release the source clock domain's reset last.

Additionally, the clearing signals (`clr_s` and `clr_d`) should not be asserted sooner than one clock cycle after their respective domain's system reset de-assertion. In fact, if possible, the first assertion of `clr_s` or `clr_d` shouldn't occur until one clock cycle within its domain from the last de-assertion of system reset between both domains.

Detail Clearing Functionality (synchronous clearing)

The DW_reset_sync contains one clearing signal for each domain called `clr_s` and `clr_d`. A minimum of a single clock cycle pulse on either one of these clearing signals initiates a synchronized clearing sequence to each domain for resetting of sequential elements of the system. This clearing sequence is orchestrated to ensure that the destination domain interface is completely cleared and ready before the source domain is permitted to initiating new activity.

In general, independent of which domain initiates the clearing sequence, the destination domain always goes into the active clearing state before the source domain does as indicated by `clr_in_prog_d` and `clr_in_prog_s` going to '1', respectively. Similarly, the destination domain exits the active clearing state before the source domain does as indicated by `clr_in_prog_d` and `clr_in_prog_s` going to '0', respectively.

Refer to Figure 1-2 through Figure 1-5 that show various timing of `clr_s` and `clr_d`.

It is imperative for system integrity to cease source domain activity after asserting `clr_s` (and while waiting for a subsequent `clr_cmplt_s` pulse) and/or when observing an active `clr_in_prog_s`. From the destination domain, accepting activity after `clr_d` is asserted and/or observing an active `clr_in_prog_d` would result in corrupting system integrity. Bottom-line, it is very important to halt source domain and destination domain activity during the clearing sequence and only start source domain activity after the `clr_cmplt_s` pulse is observed.

There is no restriction on how often or how long `clr_s` and `clr_d` can be asserted. The clearing operation is maintained if in progress and subsequent `clr_s` and/or `clr_d` initiations are made. Once the final assertion of `clr_s` and/or `clr_d` is made, the sustained clearing sequence eventually comes to completion and all 'in progress' flags de-assert.

Test

The synthesis parameter, *tst_mode*, controls the insertion of lock-up latches at the points where signals cross between the clock domains, `clk_s` and `clk_d`. Lock-up latches are used to ensure proper cross-domain operation during the capture phase of scan testing in devices with multiple clocks. When *tst_mode*=1, lock-up latches will be inserted during synthesis and will be controlled by the input, test.

With *tst_mode*=1, the input, `test`, controls the bypass of the latches for normal operation where `test`=0 bypasses latches and `test`=1 includes latches. In order to assist DFT compiler in the use of the lock-up latches, use the "`set_test_hold 1 tst_mode`" command before using the `insert_scan` command.

When *tst_mode*=0 (which is its default value when not set in the design) no lock-up latches are inserted and the test input is not connected.

Attention The insertion of lock-up latches requires the availability of an active low enable latch cell. If the target library does not have such a latch or if latches are not allowed (using dont_use commands for instance), synthesis of this module with *tst_mode*=1 will fail.

Timing Diagrams

Figure 1-2 shows an example of `clr_s` initiating a local clearing sequence. In this case, `clr_s` is a single `clk_s` cycle, but the length of `clr_s` assertions is not restricted. When `clr_s` is asserted, it propagates to the destination domain where it is synchronized and produces the assertion of `clr_in_prog_d` (which stays active until the clearing sequence from the destination domain's perspective completes). The event of the `clr_in_prog_d` assertion is fed back to the source domain where it is synchronized and, in turn, starts the source domain clearing event in the form of `clr_sync_s` and `clr_in_prog_s` assertions. As with the `clr_in_prog_d` in the destination domain, `clr_in_prog_s` stays asserted until the completion of the clearing sequence in the source domain.

The event of the `clr_sync_s` assertion then gets routed back to the destination domain to initiate the synchronized completion of the clearing sequence on that end. This is indicated by the `clr_sync_d` pulse and de-assertion of `clr_in_prog_d` followed by the `clr_cmplt_d`. At this point, the destination domain is in its initialized state and ready to receive activity from the source domain.

To finish up the clearing sequence between the two domains, the `clr_sync_d` pulse is sent back to the source domain where it is synchronized and de-asserts `clr_in_prog_s` which is followed by the `clr_cmplt_s` pulse. Once the `clr_cmplt_s` gets asserted, it is the indication to the source domain that the destination domain is cleared and ready for the new activity.

For this waveform example *f_sync_type* is 2, *r_sync_type* is 2, and *reg_in_prog* is 1. *tst_mode* and *verif_en* have no impact in this example.

Figure 1-2 Example of clr_s Initiated Clearing

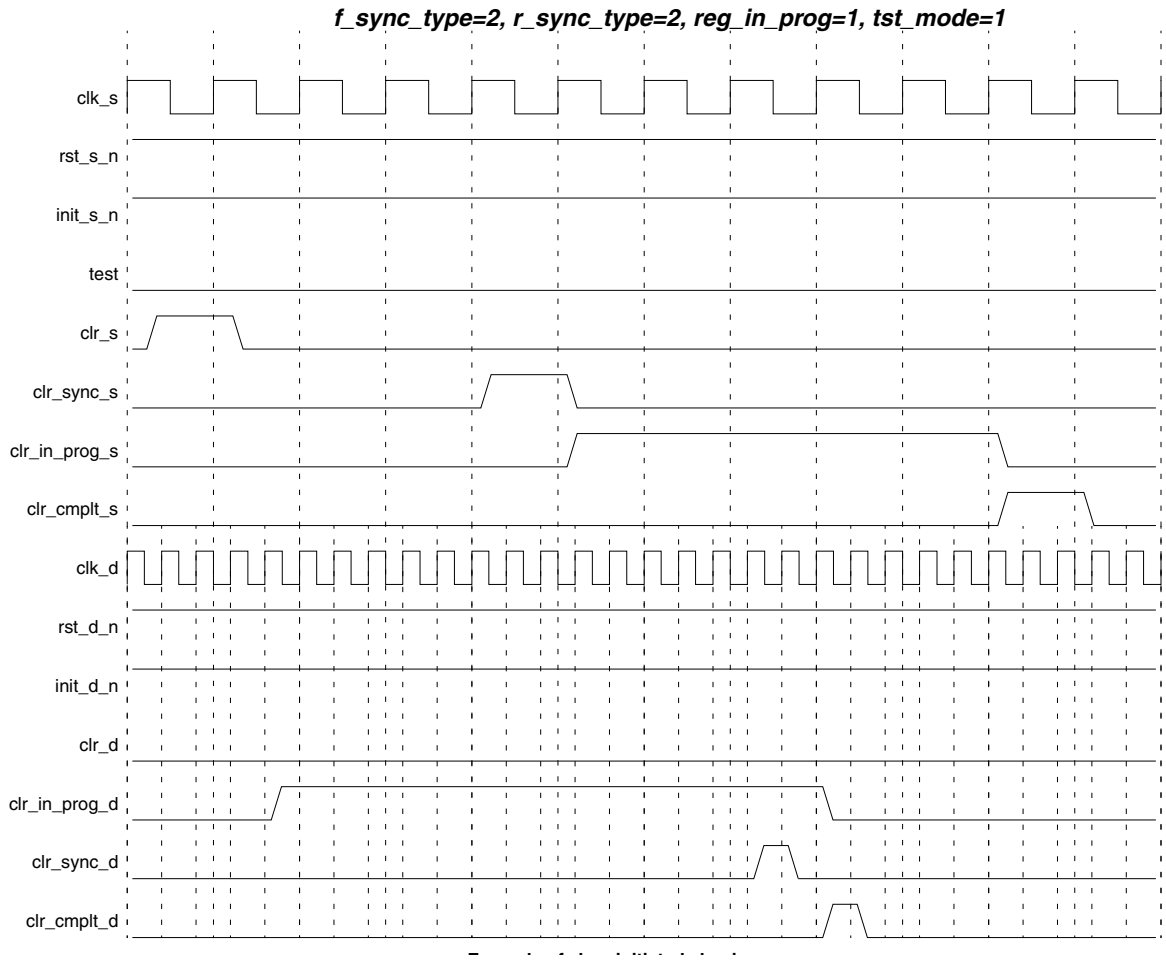


Figure 1-3 shows an example of `clr_d` initiating a clearing sequence. The `clr_d` initiated pulse places the destination domain in the active clearing state as indicated by the assertion of `clr_in_prog_d`. At the same time, `clr_d` gets sent to the source domain where it is synchronized and triggers the `clr_sync_s` and `clr_in_prog_s` signals. At this point, the source domain needs to realize, based on `clr_sync_s` or `clr_in_prog_s` assertions that the destination domain is in the active clearing state and is not receiving any more activity.

Once the `clr_sync_s` assertion occurs it is re-synchronized back in the destination domain to generate the `clr_sync_d` output as well as the de-assertion of the `clr_in_prog_d` output followed by the active pulse of `clr_cmplt_d`. At this point, the destination domain is in its initialized state and ready to receive activity from the source domain.

The assertion of `clr_sync_d` is sent back to the source domain which triggers the de-assertion of `clr_in_prog_s` followed by the `clr_cmplt_s` pulse activation. Once the `clr_cmplt_s` gets asserted, it is the indication to the source domain that the destination domain is cleared and ready for the new activity.

Figure 1-3 Example of clr_d Initiated Clearing Sequence

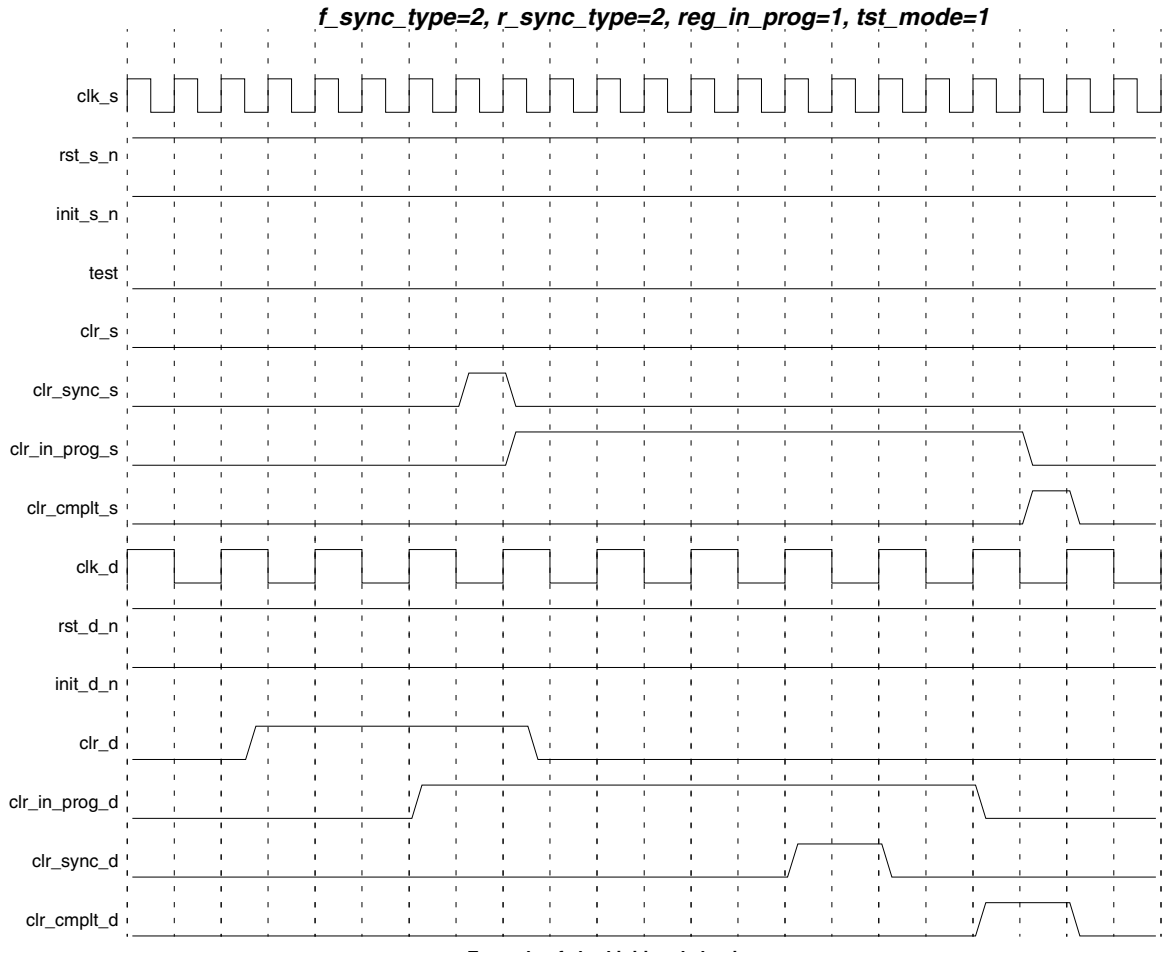


Figure 1-4 shows an example of both `clr_s` and `clr_d` being asserted around the same time. In this example the `clk_s` rate is faster than the `clk_d` rate. Once one of the clearing signals is sampled, the clearing sequence is starting. As long as either `clr_s` or `clr_d` is sustained 'high' and the clearing sequence is in progress, clearing is also sustained until both `clr_s` and `clr_d` are de-asserted. Following the de-assertion of both `clr_s` and `clr_d`, the clearing sequence progresses and completes as indicated by the single clock cycle pulses of `clr_cmplt_d` and `clr_cmplt_s` from each domain.

Figure 1-4 Example of clr_s and clr_d Initiated Clearing Sequence

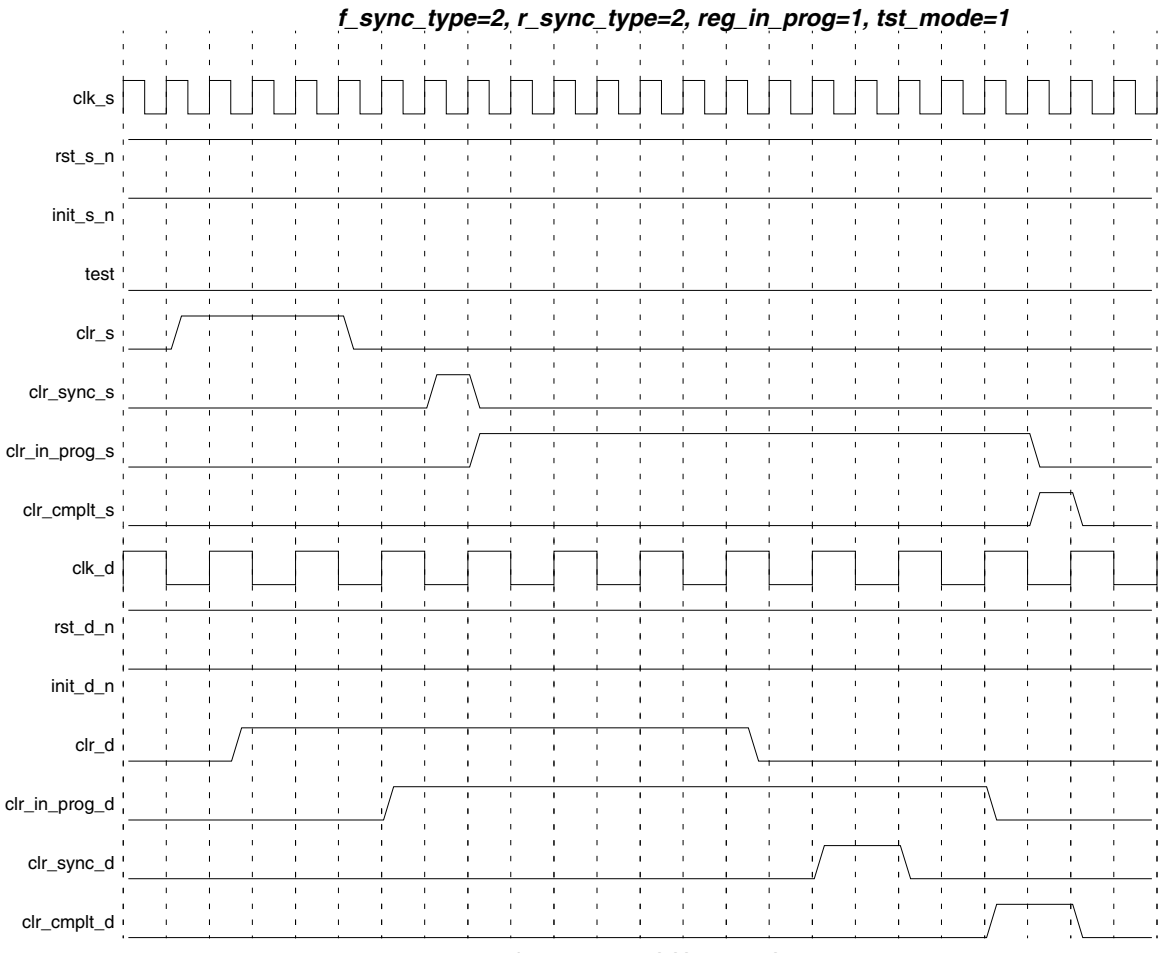


Figure 1-5 shows an initiation of a `clr_s` where its duration is much longer than one 'clk_s' cycle. From this, the behavior of the `clr_in_prog_s` and `clr_in_prog_d` flags are sustained longer than those seen in Figure 1-2 in which `clr_s` was only asserted a single `clk_s` cycle.

Figure 1-5 clr_s duration much longer than one `clk_s` cycle

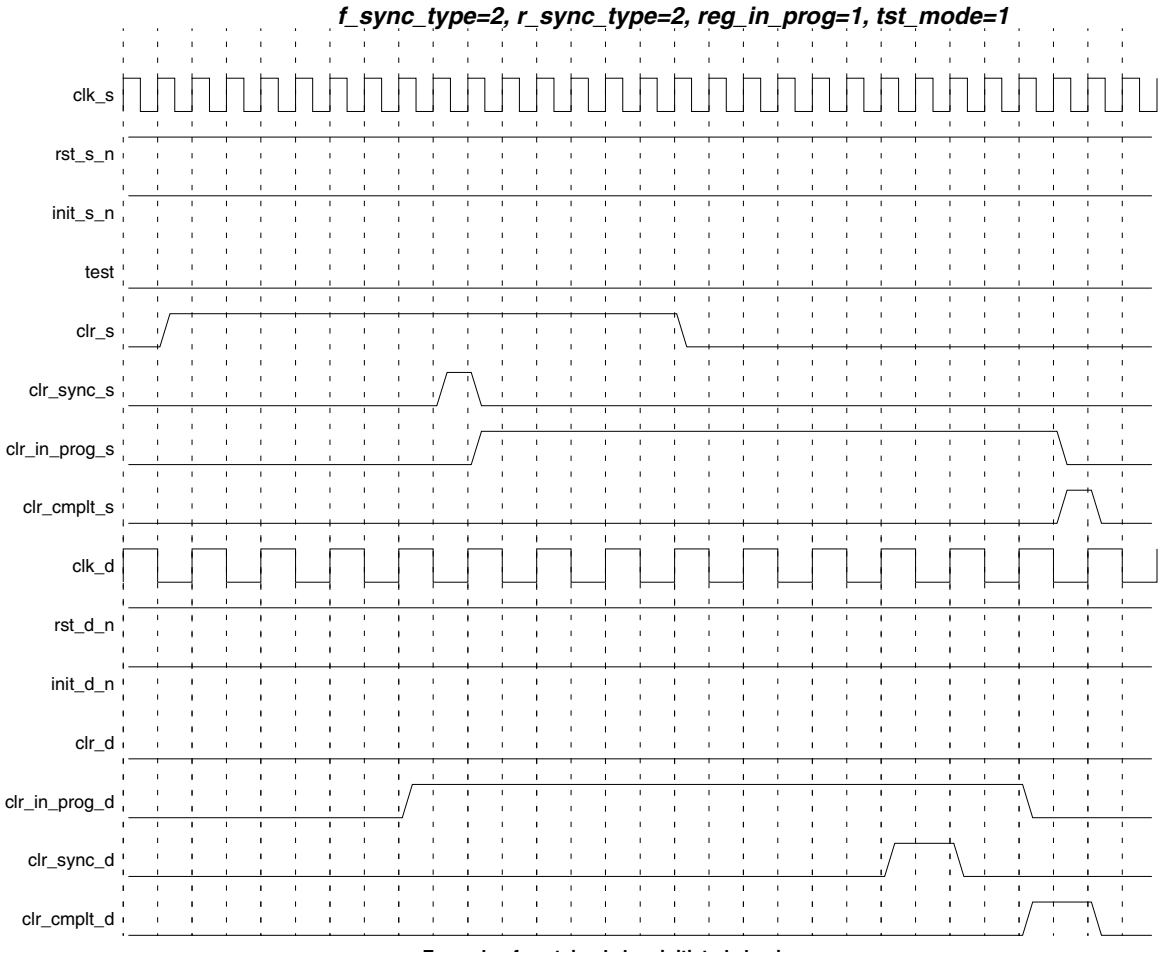
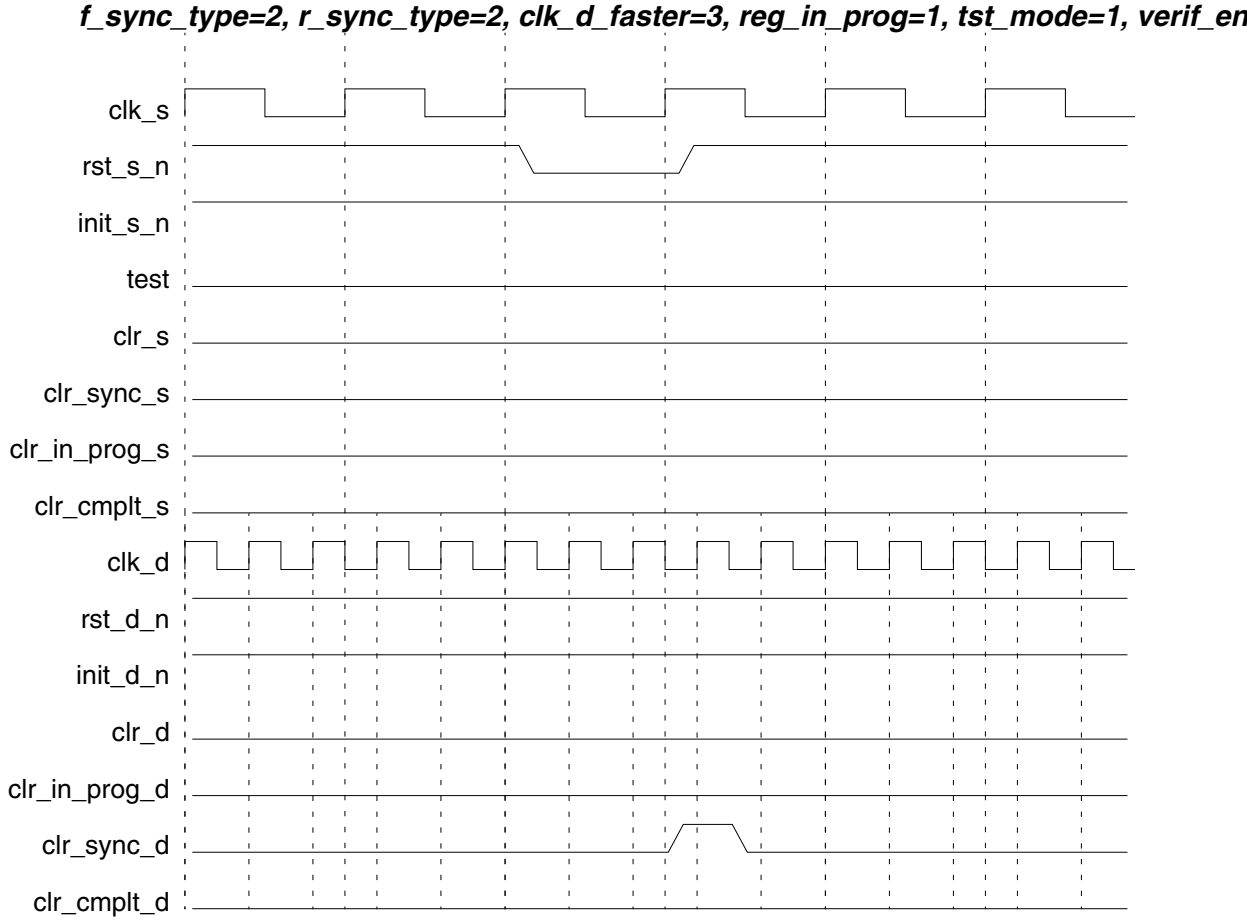


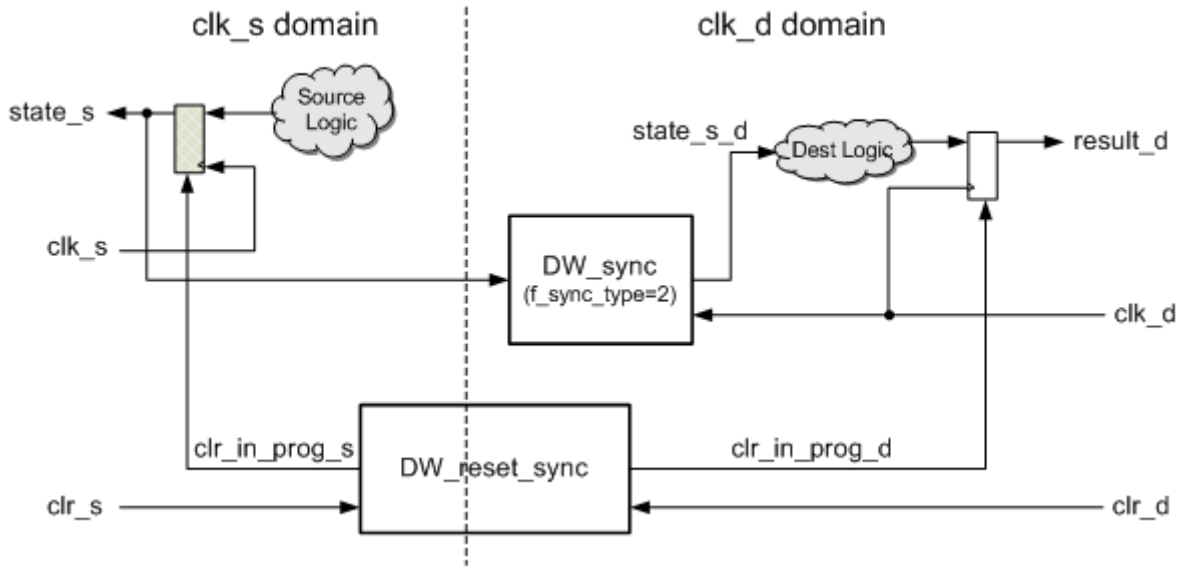
Figure 1-6 shows an example of `rst_s_n` being asserted and how it can induce a `clr_sync_d` event that is not initiated by either a `clr_s` or `clr_d` assertion.

Figure 1-6 Example of asserted rst_s_n Causing Abnormal Clearing Sequence



Example DW_reset_sync Usage

Figure 1-7 Block Diagram for Example Usage of DW_reset_sync



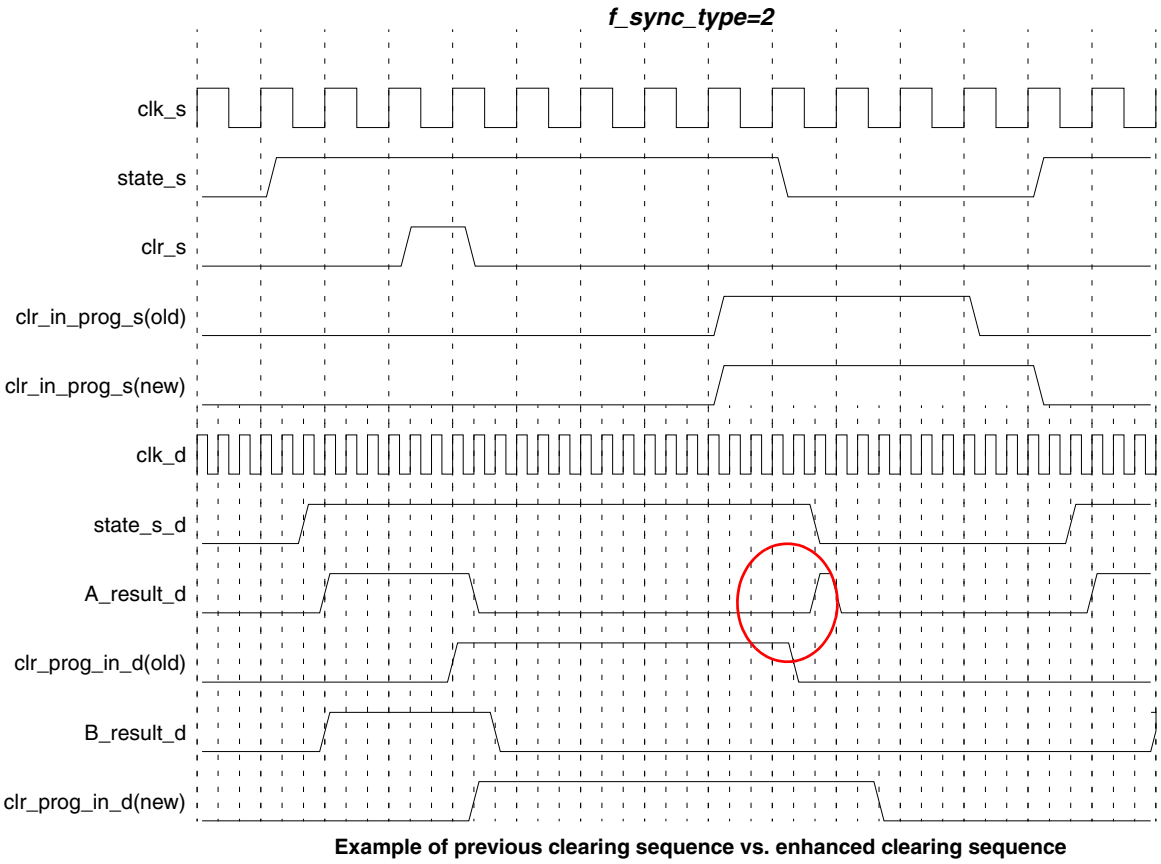
In referring to Figure 1-7, the following timing diagram (Figure 1-8) shows the difference in behavior between the previous version of DW_reset_sync (old) and the current version of DW_reset_sync (new) as they relate to `clr_in_prog_d`.

What Figure 1-8 depicts is the different “result_d” behavior when using the version DWBB_201109.1 (F-2011.09) and later (new) DW_reset_sync compared to version DWBB_201012.5 (E-2010.12-SP5) and earlier (old) DW_reset_sync. In the figure, 'state_s' is a signal in the 'clk_s' domain that gets synchronized through a DW_sync into the 'clk_d' domain as 'state_s_d'. Signal “A_result_d” and “B_result_d” represent “result_d” from Figure 1-7 but for different applications of the 'clr_in_prog_d' versions. “A_result_d” is connected to the old DW_reset_sync and its 'A_clr_in_prog_d' (old) and “B_result_d” is connected to the current DW_reset_sync and its ‘B_clr_in_prog_d’ (new). In both versions of DW_reset_sync, `clr_in_prog_s` are identical and only listed once in the timing diagram.

As can be seen, “A_result_d” contains a single-cycle `clk_d` pulse (circled in red) after ‘A_clr_in_prog_d’ (old) deasserts. Since 'A_clr_in_prog_d' is inactive, this blip of “A_result_d” comes from 'state_s_d' being sampled in its 'non-cleared' state. Eventually the cleared value of 'state_s_d' is seen but too late. This could be undesirable and could cause some inadvertent activity downstream in the system because all indications show that the clearing sequence is done in the 'clk_d' domain.

This problem is fixed with the current DW_reset_sync since the 'clr_in_prog_d' (B_clr_in_prog_d in this example) is extended and prevents “B_result_d' from registering that unwanted 'non-cleared' value produced from `state_s_d`.

Figure 1-8 Example: Fixed Possible Incorrect Result with new clr_in_prog_d behavior



Example of previous clearing sequence vs. enhanced clearing sequence

Related Topics

- Memory – Registers Overview
- DesignWare Building Block IP Documentation Overview

HDL Usage Through Component Instantiation - VHDL

```
library IEEE,DWARE;
use IEEE.std_logic_1164.all;
use DWARE.DW_Foundation_comp.all;

entity DW_reset_sync_inst is
      generic (
        inst_f_sync_type : INTEGER := 2;
            inst_r_sync_type : NATURAL := 2;
            inst_clk_d_faster: NATURAL := 1;
            inst_reg_in_prog : NATURAL := 1;
        inst_tst_mode : INTEGER := 0;
        inst_verif_en : INTEGER := 1
        );
      port (
        inst_clk_s : in std_logic;
        inst_rst_s_n : in std_logic;
        inst_init_s_n : in std_logic;
        inst_clr_s : in std_logic;
        clr_sync_s_inst : out std_logic;
        clr_in_prog_s_inst : out std_logic;
        clr_cmplt_s_inst : out std_logic;

        inst_clk_d : in std_logic;
        inst_rst_d_n : in std_logic;
        inst_init_d_n : in std_logic;
        inst_clr_d : in std_logic;
            clr_in_prog_d_inst : out std_logic;
            clr_sync_d_inst : out std_logic;
            clr_cmplt_d_inst : out std_logic;

        inst_test : in std_logic
        );
    end DW_reset_sync_inst;


architecture inst of DW_reset_sync_inst is
begin

    -- Instance of DW_reset_sync
    U1 : DW_reset_sync
    generic map ( f_sync_type => inst_f_sync_type, r_sync_type => inst_r_sync_type,
clk_d_faster => inst_clk_d_faster,
                      reg_in_prog => inst_reg_in_prog, tst_mode => inst_tst_mode,
verif_en => inst_verif_en )
    port map ( clk_s => inst_clk_s, rst_s_n => inst_rst_s_n, init_s_n => inst_init_s_n,
clr_s => inst_clr_s,
```

```
                    clr_sync_s => clr_sync_s_inst, clr_in_prog_s => clr_in_prog_s_inst,
clr_cmplt_s => clr_cmplt_s_inst,
              clk_d => inst_clk_d, rst_d_n => inst_rst_d_n, init_d_n => inst_init_d_n,
clr_d => inst_clr_d,
              clr_in_prog_d => clr_in_prog_d_inst, clr_sync_d => clr_sync_d_inst,
clr_cmplt_d => clr_cmplt_d_inst,
                test => inst_test );


end inst;

-- Configuration for use with a VHDL simulator
-- pragma translate_off
library DW03;
configuration DW_reset_sync_inst_cfg_inst of DW_reset_sync_inst is
  for inst
    -- NOTE: If desiring to model missampling, uncomment the following
    -- line.  Doing so, however, will cause inconsequential errors
    -- when analyzing or reading this configuration before synthesis.
    -- for U1 : DW_reset_sync use configuration DW03.DW_reset_sync_cfg_sim_ms;  end
for;
  end for; -- inst
end DW_reset_sync_inst_cfg_inst;
-- pragma translate_on
```

HDL Usage Through Component Instantiation - Verilog

```
module DW_reset_sync_inst( inst_clk_s, inst_rst_s_n, inst_init_s_n, inst_clr_s,
                clr_sync_s_inst, clr_in_prog_s_inst, clr_cmplt_s_inst,
          inst_clk_d, inst_rst_d_n, inst_init_d_n, inst_clr_d, clr_in_prog_d_inst,
          clr_sync_d_inst, clr_cmplt_d_inst, inst_test );

parameter f_sync_type = 2;
parameter r_sync_type  = 2;
parameter clk_d_faster = 1;
parameter reg_in_prog = 1;
parameter tst_mode = 0;
parameter verif_en = 1;


input inst_clk_s;
input inst_rst_s_n;
input inst_init_s_n;
input inst_clr_s;
output clr_sync_s_inst;
output clr_in_prog_s_inst;
output clr_cmplt_s_inst;

input inst_clk_d;
input inst_rst_d_n;
input inst_init_d_n;
input inst_clr_d;
output clr_in_prog_d_inst;
output clr_sync_d_inst;
output clr_cmplt_d_inst;

input inst_test;

    // Instance of DW_reset_sync
    DW_reset_sync #(f_sync_type, r_sync_type, clk_d_faster, reg_in_prog, tst_mode,
verif_en)
      U1 ( .clk_s(inst_clk_s), .rst_s_n(inst_rst_s_n), .init_s_n(inst_init_s_n),
.clr_s(inst_clr_s), .clr_sync_s(clr_sync_s_inst), .clr_in_prog_s(clr_in_prog_s_inst),
.clr_cmplt_s(clr_cmplt_s_inst), .clk_d(inst_clk_d), .rst_d_n(inst_rst_d_n),
.init_d_n(inst_init_d_n), .clr_d(inst_clr_d), .clr_in_prog_d(clr_in_prog_d_inst),
.clr_sync_d( clr_sync_d_inst), .clr_cmplt_d(clr_cmplt_d_inst), .test(inst_test) );

endmodule
```

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.

All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043

www.synopsys.com