AN 917: Reset Design Techniques for Intel® Hyperflex™ Architecture FPGAs

Updated for Intel® Quartus® Prime Design Suite: **20.4**

Online Version
Send Feedback

AN-917

ID: 683539
Version: 2021.01.05

Contents

1. AN 917: Reset Design Techniques for Intel® Hyperflex™ Architecture FPGAs

This document describes reset design techniques to ensure reliable power-up, reset release conditions, and maximum performance on Intel® Hyperflex™ architecture FPGAs. Proper application of reset design techniques allows you to take full advantage of the Hyper-Pipelining and Hyper-Retiming performance optimization features in Intel Stratix® 10 and Intel Agilex™ devices.

Refer to the following reset recommendations, strategies, and recommended coding techniques for effective reset implementation:

- General Reset Recommendations on page 3
- Reset Design Strategies on page 5
- Analyzing Resets with Design Assistant on page 9
- Implementing Reset Circuitry on page 16

1.1. General Reset Recommendations

The main purpose of providing a reset to an FPGA design is to provide stability, and to prevent power-on to an unknown state. Asserting reset forces all registers into a known state.

However, improper reset implementation can cause functional errors in FPGA designs and can also cause retiming restrictions during the Fitter's Retime stage. Such retiming restrictions limit the movement of registers to balance the propagation delays between the registers in a chain. Movement of these registers shortens the critical paths and increases operating frequency.

Reset usage can have a significant impact on the timing closure, area, and the routing congestion of your design. Systems with multiple clock domains further complicate reset issues. The lack of reset coordination in some designs can even cause intermittent failures on power-up. The following general reset recommendations apply to Intel Hyperflex architecture designs.

1.1.1. Limit Reset Usage

Limit resets to design conditions that actually require reset. Limiting resets has the following benefits, especially for a large data path that can cause excessive resource usage:

- Reduced logic utilization
- Lower routing congestion
- Lower data path routing delays
- Better timing performance

However, you must assert a reset for any register that requires power-up to a known state. The following are some of the design scenarios that require resets:

- If a watchdog timer detects a fault, then it can generate a reset to the whole system.
- Status registers that require reset at certain events after taking action to clear any pending errors.
- Simple counters or state machines that require return to initial states when conditions arise during a normal application run.
- A flip-flop within a feedback loop requires a reset to bring the register back to a reset state during a normal process.

The following are some of the blocks or registers that can function without resets:

- Shift registers and data buses accompanied by control signals to signify validity of current values.
- Designs that use pipeline or delay chains may only require reset at the beginning of the pipeline stage, but not at the succeeding stages. Hold the reset asserted for a duration long enough to flush the entire pipeline.

1.1.2. Hyper-Registers Require Synchronous Reset

For the best performance, avoid resets except when necessary. When design conditions require a reset, you must decide the type of reset to use, asynchronous or synchronous.

The Hyper-Retiming feature facilitates movement between registers to balance the propagation delays between them, with the aim of fixing critical path timing. Hyper-Retiming moves the ALM registers into the Hyper-Register locations that are available in every routing segment of the fabric.

By design, the Hyper-Registers do not have asynchronous resets. Therefore, the Intel Quartus® Prime Fitter cannot retime ALM registers with an asynchronous reset into a Hyper-Register location. Instead, use a synchronous reset to permit the usage of the Hyper-Registers during retiming. The ability to retime into Hyper-Registers helps achieve the best possible performance.

In blocks where the use of asynchronous resets is unavoidable, minimize the use of asynchronous resets within these blocks wherever possible. Asynchronous resets allow a circuit to reset with or without a clock signal present. There may be blocks that require this function, including any of the following:

- Clock source and distribution circuitry
- Power management block
- System reset generator

Use synchronous resets if a clock is present on every reset assertion. You may be able to use asynchronous resets for blocks using slow clocks when timing is not critical, especially if the asynchronous reset helps in decongesting data path routing.

1.2. Reset Design Strategies

The following topics describe recommended design strategies for asynchronous and synchronous resets:

- Asynchronous Reset Design Strategies on page 5
- Synchronous Reset Design Strategies on page 7

1.2.1. Asynchronous Reset Design Strategies

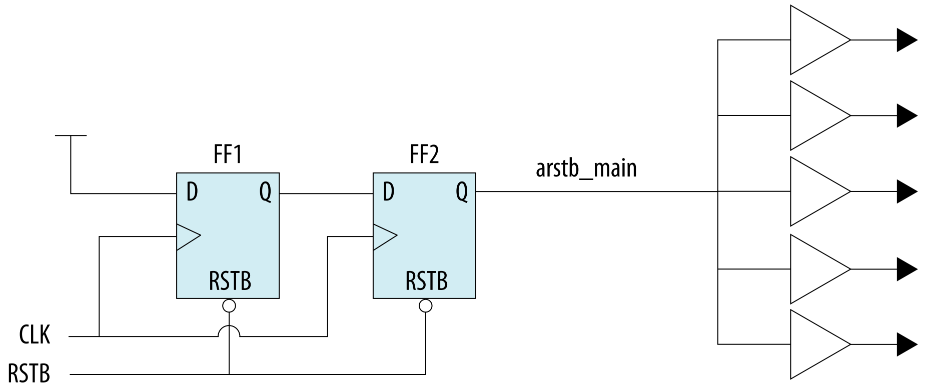
The primary disadvantage of using an asynchronous reset is that the reset is asynchronous both at the assertion and de-assertion of the signal.

The signal assertion is not the problem on the actual connected flip-flop. Even if the flip-flop moves to a metastable state, the flip-flop remains unstable only for a short period of time after assertion. After reset asserts long enough, all registers eventually settle to a reset state.

The problem of the asynchronous reset involves the signal de-assertion. If an asynchronous reset releases at or near the active clock edge, the output of the flip-flop could go to a metastable state, violating the reset recovery time of the flip-flop. The flip-flop then goes to an unknown state that can cause unexpected results upon entering normal operation.

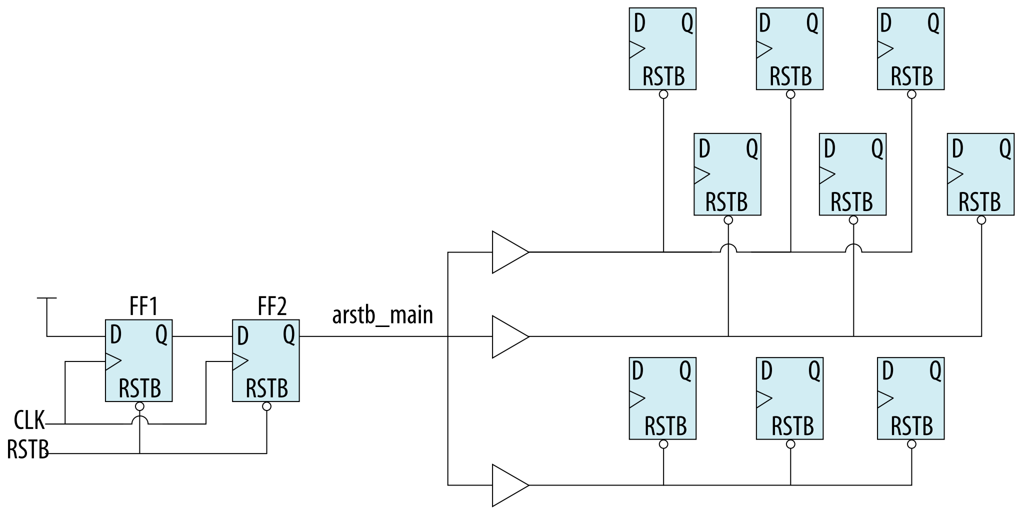
You can insert a synchronously de-asserted reset circuit to prevent this condition. Synchronization of the reset signal on a specific clock domain requires a minimum of two flops. Figure 1 on page 5 shows the first flip-flop (FF1) with output `Q` reset to `0`, and input `D` tied high. This flip-flop can go to a metastable state if `RSTB` is de-asserted near a `CLK` active edge. However, the second flip-flop (`FF2`) remains stable at `0`, since the input and output are both low, preventing any output change due to the input that might occur when a reset is removed.

Figure 1. VDD-based Reset Synchronizer



Using the VDD-based reset synchronizer, you can generate the main source (`arstb_main`) for the asynchronous reset. The reset is typically a high fan-out net. Implement a reset tree to maintain a good fan-out load to help meet the recovery and removal checks on driven registers during de-assertion. Figure 2 on page 6 shows how `arstb_main` is distributed to drive the asynchronous resets of the registers.

Figure 2. `arstb_main` as Asynchronous Reset Source

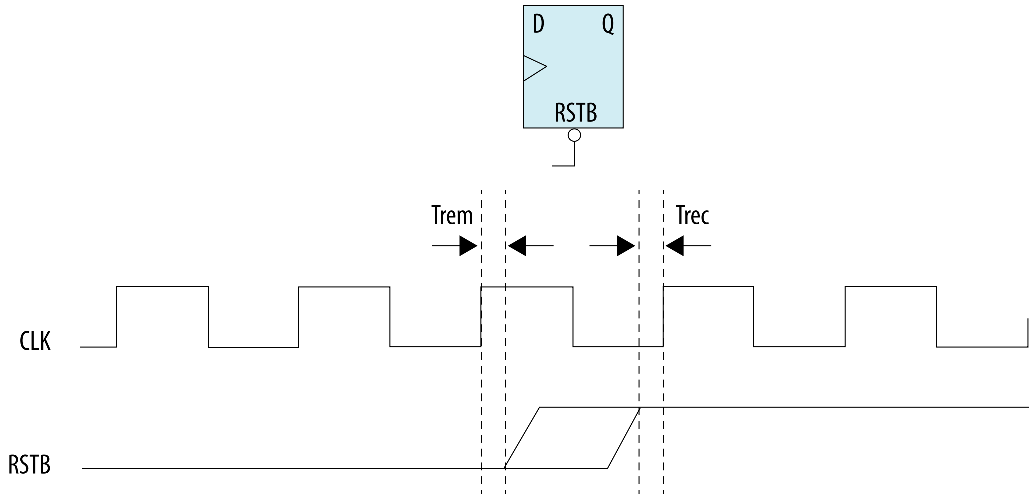


1.2.1.1. Recovery and Removal Checks

During the de-assertion of a reset, the control to the output of a flip-flop transfers from the reset line to the clock signal, like a regular D flip-flop. To avoid the register entering metastable state, you must ensure that the reset is not de-asserted in certain time frames of the active clock edge.

In Figure 3 on page 6, the Removal Time, `Trem`, refers to the minimum time, after the active clock edge, that the reset must be stable before being de-asserted. The reset Removal Check ensures that the de-asserted reset signal is not captured by the same clock edge that launches the reset.

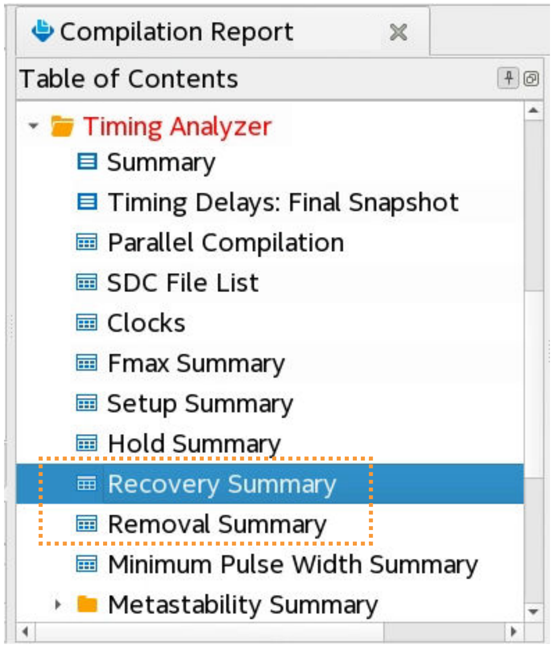
Figure 3. Recovery and Removal Check Timing Diagram



Reset Recovery Time, `Trec`, is the minimum time between the de-assertion of a reset and the clock signal being high again. The reset Recovery Check ensures that the reset signal is stable for a minimum time after de-assertion, before the next active clock edge.

The Intel Quartus Prime Timing Analyzer computes recovery and removal slacks separately from set-up and hold slacks. You can find the summary and individual reports in the Timing Analyzer reports.

Figure 4. Compilation Report Window - Timing Analyzer Reports



1.2.2. Synchronous Reset Design Strategies

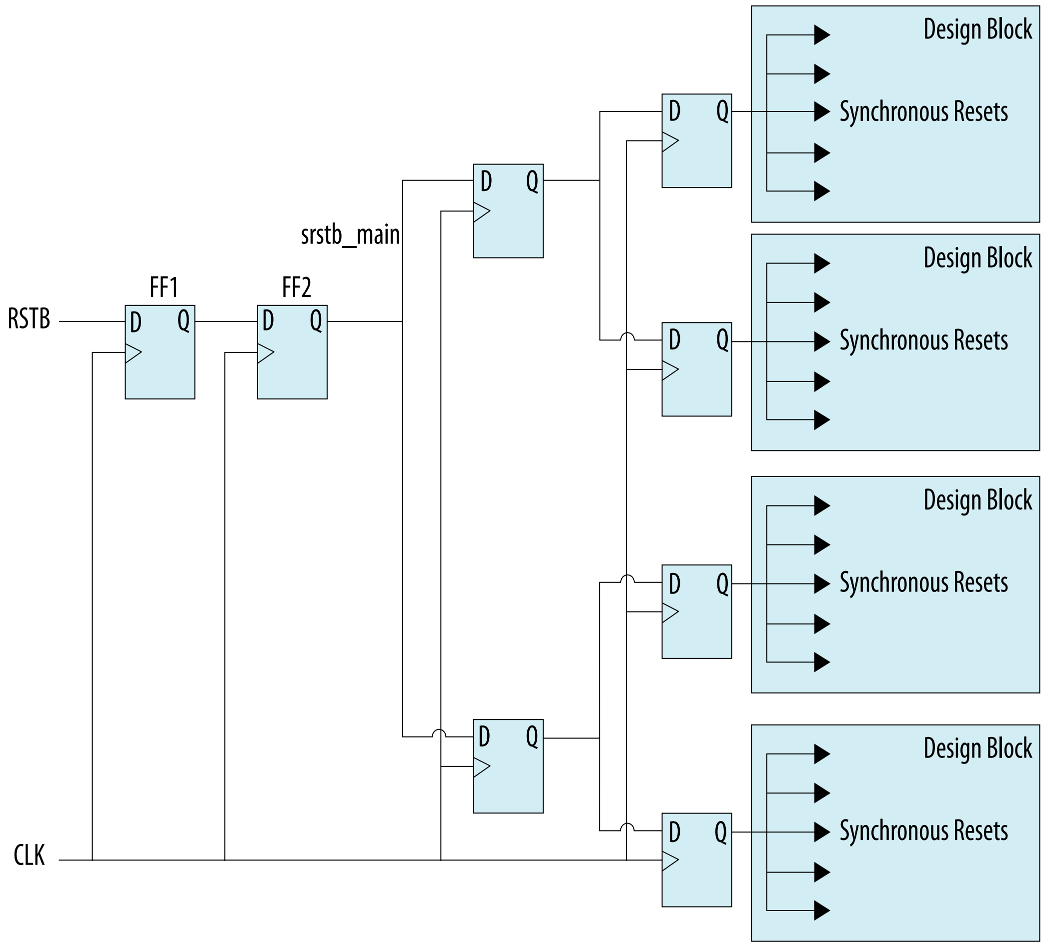
The use of synchronous resets helps to ensure a fully synchronous design. The Intel Quartus Prime Timing Analyzer can more efficiently analyze synchronous designs.

If you use an asynchronous reset source, synchronize the source before feeding it to the register's reset input. Depending on the design specifications, you can externally synchronize the reset source. Whether you synchronize the reset source internally or externally, it is part of the data logic path. Hence, you can easily determine the data arrival and data required times for proper slack analysis.

Like any broadcast signal, a synchronous reset has a high fan-out. The high fan-out makes it difficult (and at times impossible) to close timing without pipelining and duplicating the synchronous reset source.

A good technique for synchronous reset is to build a balanced reset tree, as Figure 5 on page 8 shows. In this example, `srstb_main` output of the reset synchronizer serves as the main source of the distributed synchronous reset tree. The reduction of fan-out at each stage can ease the delay paths passed through by each reset branch, thereby helping to close timing.

Figure 5. `srstb_main` as Synchronous Reset Source



You can use Hyper-Pipelining and Hyper-Retiming in your design to generate a similar reset distribution. The main goal of this design technique is to address the critical paths connecting the synchronous reset line. The Fitter accomplishes this by moving the registers on the pipeline and retiming them as needed to resolve timing issues.

Another design technique to reduce fan-out involves specifying the `DUPLICATE_HIERARCHY_DEPTH` assignment in the Intel Quartus Prime settings file (`.qsf`). This assignment can distribute duplicate registers of any high fan-out broadcast signals using hierarchical proximity to guide the duplication process.

1.3. Analyzing Resets with Design Assistant

The Intel Quartus Prime Design Assistant is a design rule checking tool that helps you to detect and correct any violations against a standard set of Intel FPGA-recommended design guidelines. Correcting design rule violations improves the reliability, timing performance, and logic utilization of your design.

The Design Assistant includes several rule categories to help you check for functional and timing closure related reset issues. Following analysis, Design Assistant provides recommendations for correcting any violations. The following table lists the Design Assistant rule categories that help detect reset issues:

Table 1. Design Assistant Rules Categories for Reset, Clock Domain Crossing, and Timing Closure

Categories	Rule Prefix
Reset Rules	RES
Clock Domain Crossing Rules	CDC
Timing Closure Rules	TMC

For descriptions of all rules, refer to "Design Assistant Rule Categories" in *Intel Quartus Prime Pro Edition User Guide: Design Recommendations*.

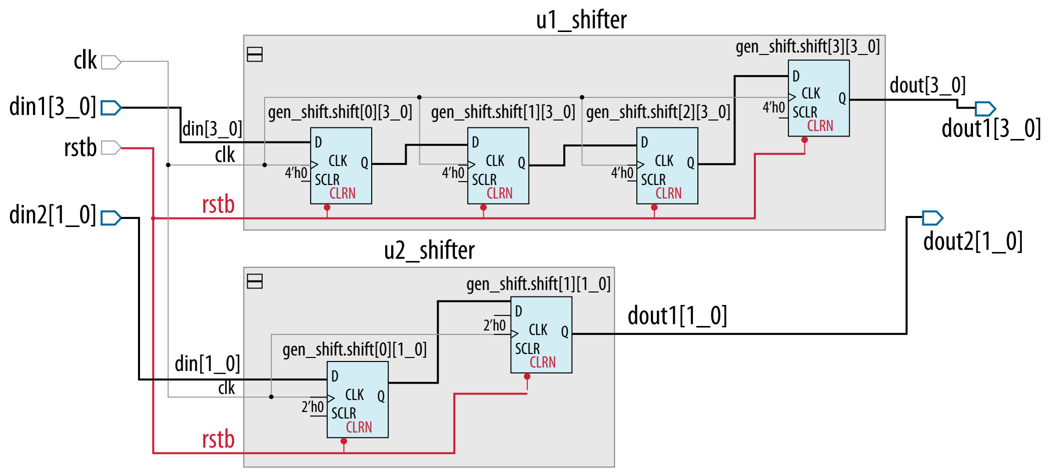
Related Information

Intel Quartus Prime Pro Edition User Guide: Design Recommendations

1.3.1. Example: Asynchronous Reset Design Assistant Analysis

You can use Design Assistant to identify and correct problematic asynchronous reset conditions. For this example, Figure 6 on page 9 shows a circuit with two separate groups of registers, with asynchronous resets driven by the input port `rstb`.

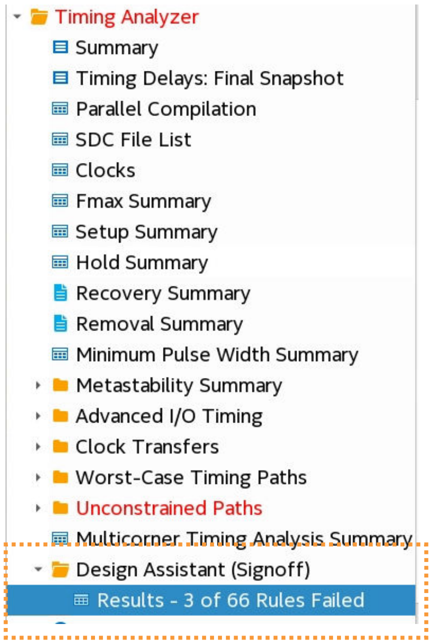
Figure 6. Circuit Design with Asynchronous Reset



The following steps describe Design Assistant reset analysis of the circuit:

1. After running the Fitter or a full compilation, view the **Design Assistant** reports in the **Timing Analyzer** report folder of the Compilation Report.

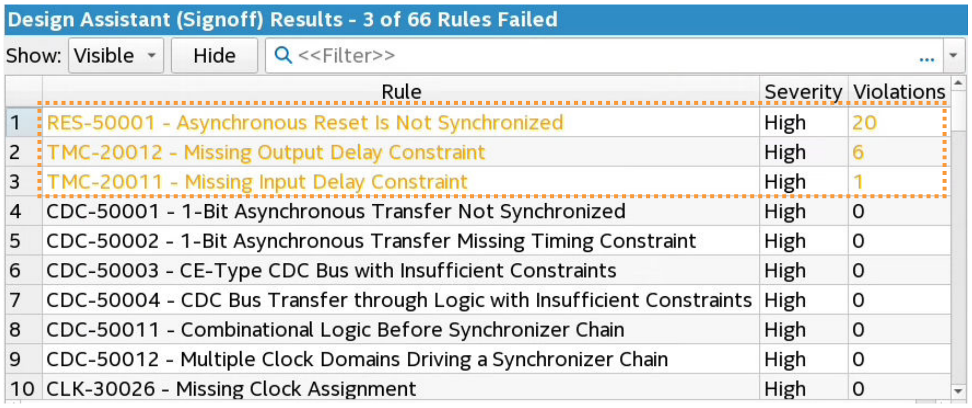
Figure 7. Design Assistant Report in Timing Analyzer Folder of Compilation Report



- Timing Analyzer
  - Summary
  - Timing Delays: Final Snapshot
  - Parallel Compilation
  - SDC File List
  - Clocks
  - Fmax Summary
  - Setup Summary
  - Hold Summary
  - Recovery Summary
  - Removal Summary
  - Minimum Pulse Width Summary
  - Metastability Summary
  - Advanced I/O Timing
  - Clock Transfers
  - Worst-Case Timing Paths
  - Unconstrained Paths
  - Multicorner Timing Analysis Summary
  - Design Assistant (Signoff)
    - Results - 3 of 66 Rules Failed

2. Review any rule violations in the report. This example shows 20 violations of rule **RES-50001 – Asynchronous Resets Not Synchronized**. Select **RES-50001** in the list to show all instances violating the specific rule in your design.

Figure 8. Design Assistant (Signoff) Results



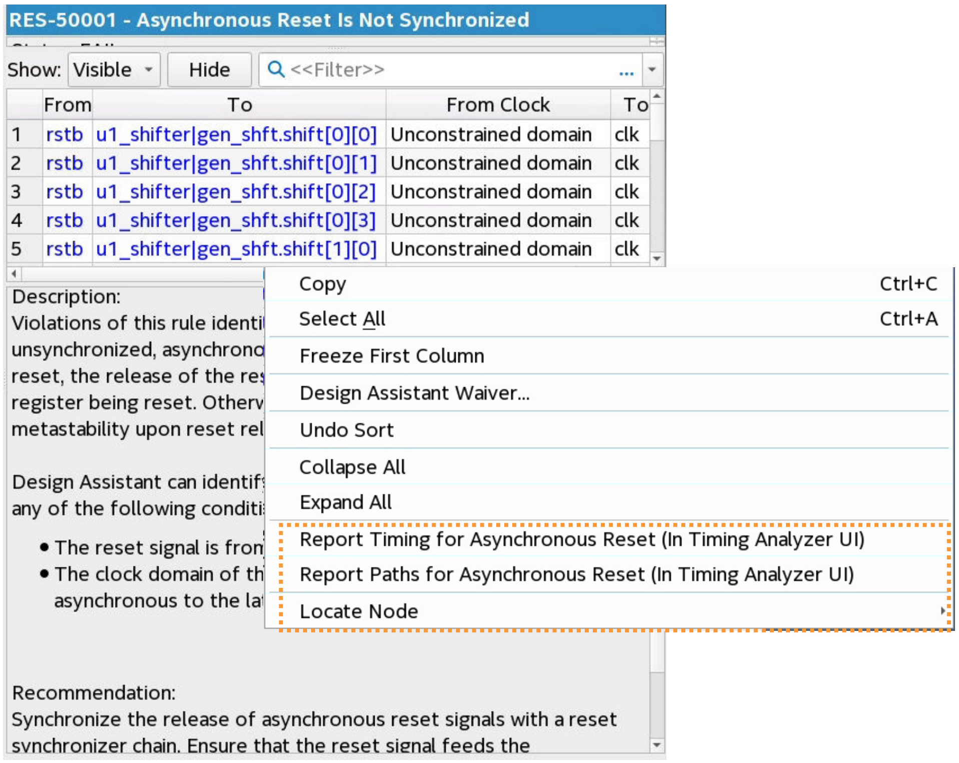
Design Assistant (Signoff) Results - 3 of 66 Rules Failed

Show: Visible | Hide | <<Filter>>

	Rule	Severity	Violations
1	RES-50001 - Asynchronous Reset Is Not Synchronized	High	20
2	TMC-20012 - Missing Output Delay Constraint	High	6
3	TMC-20011 - Missing Input Delay Constraint	High	1
4	CDC-50001 - 1-Bit Asynchronous Transfer Not Synchronized	High	0
5	CDC-50002 - 1-Bit Asynchronous Transfer Missing Timing Constraint	High	0
6	CDC-50003 - CE-Type CDC Bus with Insufficient Constraints	High	0
7	CDC-50004 - CDC Bus Transfer through Logic with Insufficient Constraints	High	0
8	CDC-50011 - Combinational Logic Before Synchronizer Chain	High	0
9	CDC-50012 - Multiple Clock Domains Driving a Synchronizer Chain	High	0
10	CLK-30026 - Missing Clock Assignment	High	0

3. Find the location of rule violations in your design. You can cross probe each instance to locate the node.

Figure 9. Locating to RES-50001 Rule Violations


RES-50001 - Asynchronous Reset Is Not Synchronized
Show: Visible
Hide
<<Filter>>
Copy Ctrl+C
Select All Ctrl+A
Freeze First Column
Design Assistant Waiver...
Undo Sort
Collapse All
Expand All
Report Timing for Asynchronous Reset (In Timing Analyzer UI)
Report Paths for Asynchronous Reset (In Timing Analyzer UI)
Locate Node

	From	To	From Clock	To
1	rstb	u1_shifter|gen_shft.shift[0][0]	Unconstrained domain	clk
2	rstb	u1_shifter|gen_shft.shift[0][1]	Unconstrained domain	clk
3	rstb	u1_shifter|gen_shft.shift[0][2]	Unconstrained domain	clk
4	rstb	u1_shifter|gen_shft.shift[0][3]	Unconstrained domain	clk
5	rstb	u1_shifter|gen_shft.shift[1][0]	Unconstrained domain	clk

Description:
Violations of this rule identi
unsynchronized, asynchrono
reset, the release of the re
register being reset. Otherv
metastability upon reset rel

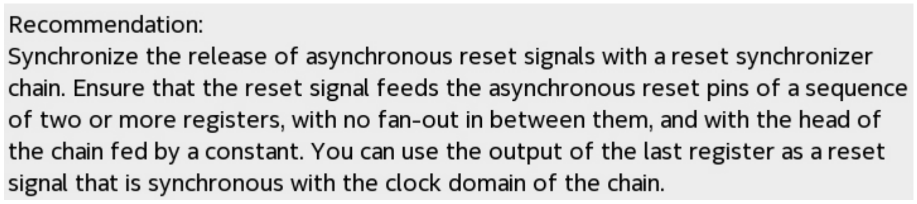
Design Assistant can identif
any of the following conditi

- The reset signal is fron
- The clock domain of th asynchronous to the la

Recommendation:
Synchronize the release of asynchronous reset signals with a reset synchronizer chain. Ensure that the reset signal feeds the

4. Identify the root cause of the violation and the recommendation for correction in the Design Assistant Recommendation.

Figure 10. RES-50001 Rule Recommendation

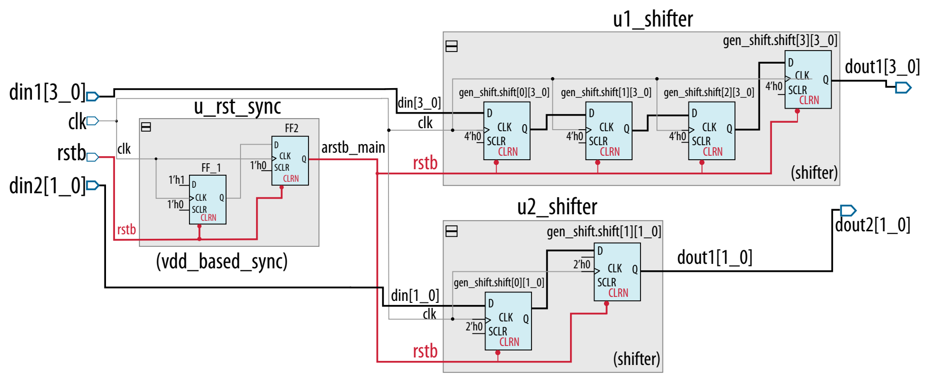


Recommendation:
Synchronize the release of asynchronous reset signals with a reset synchronizer chain. Ensure that the reset signal feeds the asynchronous reset pins of a sequence of two or more registers, with no fan-out in between them, and with the head of the chain fed by a constant. You can use the output of the last register as a reset signal that is synchronous with the clock domain of the chain.

5. Implement the rule Recommendation in your design RTL. In this example, Design Assistant recommends inserting a synchronizer to feed all asynchronous reset pins the report specifies, as the following example shows.

   Note: To time the recovery and removal going to the reset pin, the clock domain of the synchronizer must match the latching domain of the register being reset.

Figure 11. Adding a VDD-Based Reset Synchronizer

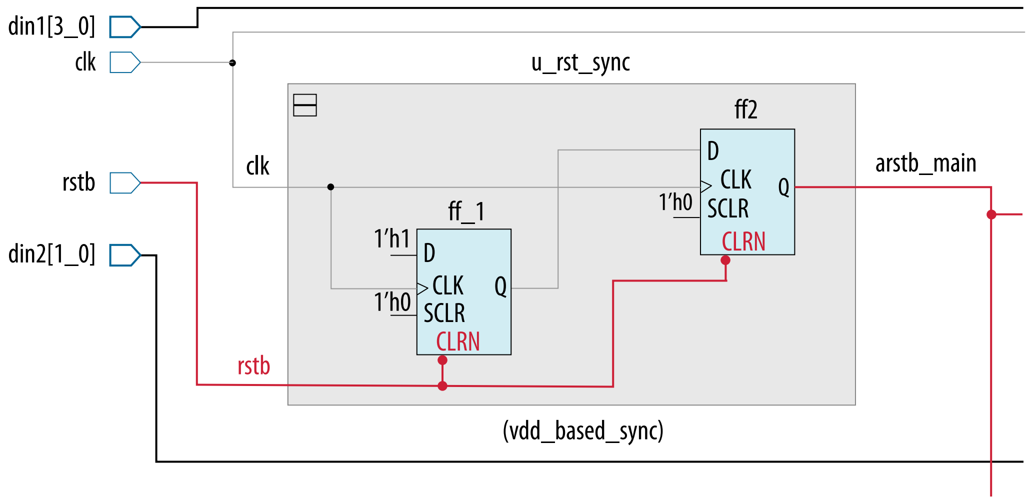


6. Recompile the design with your changes, and rerun Design Assistant to confirm corrections. Repeat steps 1 through 6 until all issues are fixed.

1.3.1.1. Adding SDC Constraints to Asynchronous Reset Circuitry

After correcting the asynchronous related issues in the reset circuitry for this example, Design Assistant reports two violations for the two new registers because `rstb` drives their asynchronous reset pins. However, the `rstb` clock domain is unrelated or asynchronous to the latching domain of the register.

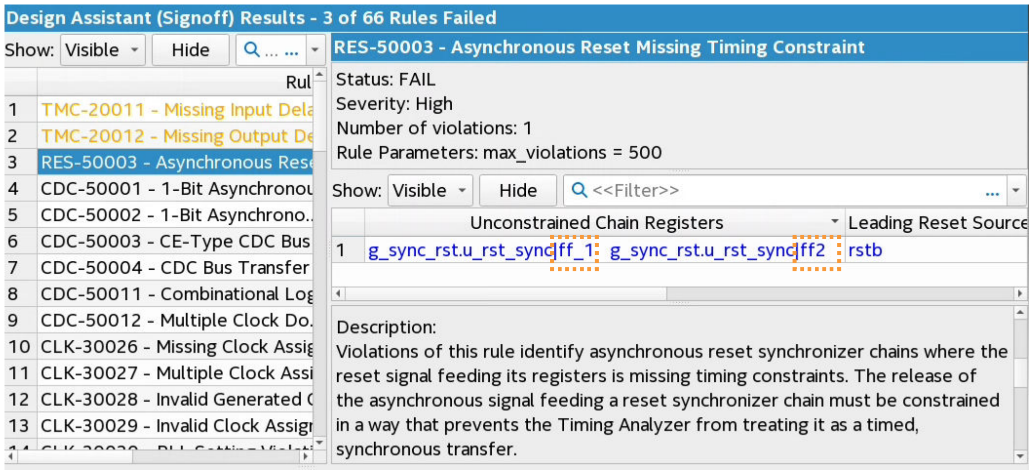
Figure 12. New Registers with Asynchronous Reset Driven by `rstb`



Design Assistant detects that the two registers are reset synchronizers in this instance, causing another Design Assistant rule violation for rule-ID **RES-50003 Asynchronous Reset Missing Timing Constraint**.

Design Assistant reports that the reset signal feeding the asynchronous reset synchronizer is missing timing constraints:

Figure 13. RES-50003 Rule Violations



Design Assistant (Signoff) Results - 3 of 66 Rules Failed

Show: Visible | Hide

	Rule
1	TMC-20011 - Missing Input Dela
2	TMC-20012 - Missing Output De
3	RES-50003 - Asynchronous Rese
4	CDC-50001 - 1-Bit Asynchronou
5	CDC-50002 - 1-Bit Asynchrono..
6	CDC-50003 - CE-Type CDC Bus
7	CDC-50004 - CDC Bus Transfer
8	CDC-50011 - Combinational Log
9	CDC-50012 - Multiple Clock Do.
10	CLK-30026 - Missing Clock Assig
11	CLK-30027 - Multiple Clock Assi
12	CLK-30028 - Invalid Generated C
13	CLK-30029 - Invalid Clock Assigr

RES-50003 - Asynchronous Reset Missing Timing Constraint

Status: FAIL
Severity: High
Number of violations: 1
Rule Parameters: max_violations = 500

Show: Visible | Hide | <<Filter>>

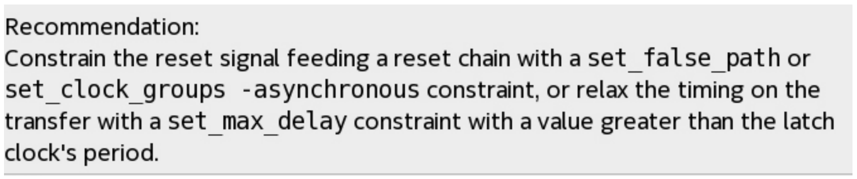
	Unconstrained Chain Registers	Leading Reset Source
1	g_sync_rst.u_rst_sync|ff_1; g_sync_rst.u_rst_sync|ff2	rstb

Description:
Violations of this rule identify asynchronous reset synchronizer chains where the reset signal feeding its registers is missing timing constraints. The release of the asynchronous signal feeding a reset synchronizer chain must be constrained in a way that prevents the Timing Analyzer from treating it as a timed, synchronous transfer.

To resolve the violations in this example, perform the following steps:

1. Check the rule Description and Recommendation.

Figure 14. RES-50003 Rule Recommendation



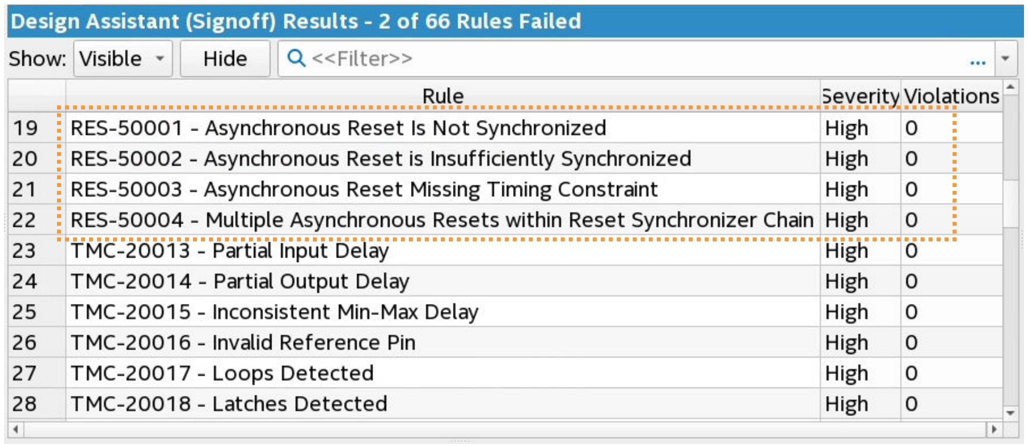
Recommendation:
Constrain the reset signal feeding a reset chain with a `set_false_path` or `set_clock_groups -asynchronous` constraint, or relax the timing on the transfer with a `set_max_delay` constraint with a value greater than the latch clock's period.

2. To implement the Recommendation, add the following constraint to the project SDC file:

```
set_false_path -from [get_ports {rstb}]
```

The RES violations disappear after compiling the sample circuitry with the fixes.

Figure 15. RES Violations Resolved



Design Assistant (Signoff) Results - 2 of 66 Rules Failed

Show: Visible | Hide | <<Filter>>

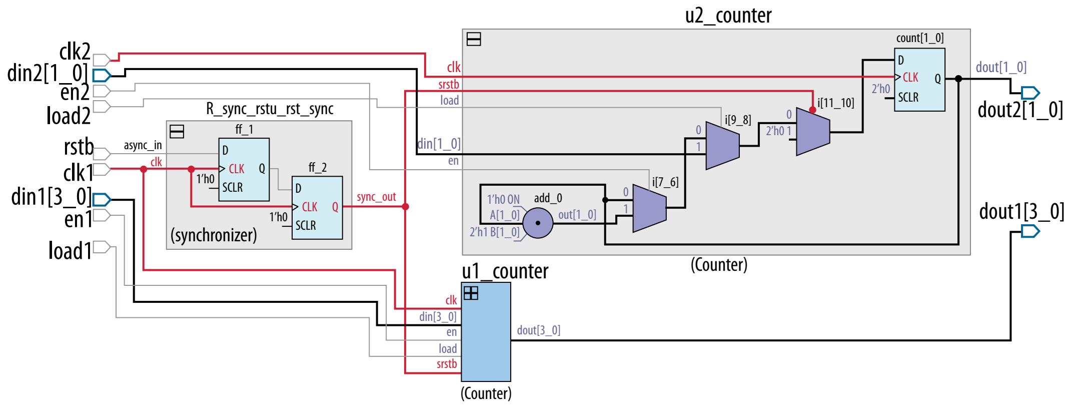
	Rule	Severity	Violations
19	RES-50001 - Asynchronous Reset Is Not Synchronized	High	0
20	RES-50002 - Asynchronous Reset is Insufficiently Synchronized	High	0
21	RES-50003 - Asynchronous Reset Missing Timing Constraint	High	0
22	RES-50004 - Multiple Asynchronous Resets within Reset Synchronizer Chain	High	0
23	TMC-20013 - Partial Input Delay	High	0
24	TMC-20014 - Partial Output Delay	High	0
25	TMC-20015 - Inconsistent Min-Max Delay	High	0
26	TMC-20016 - Invalid Reference Pin	High	0
27	TMC-20017 - Loops Detected	High	0
28	TMC-20018 - Latches Detected	High	0

1.3.2. Example: Synchronous Reset Design Assistant Analysis

You can also use Design Assistant to identify and correct problematic synchronous reset conditions. In the following example, the reset synchronizer output incorrectly connects to a register reset using a different clock domain. Such a connection error can create functional design issues, especially if one domain is reset unnecessarily during normal operation.

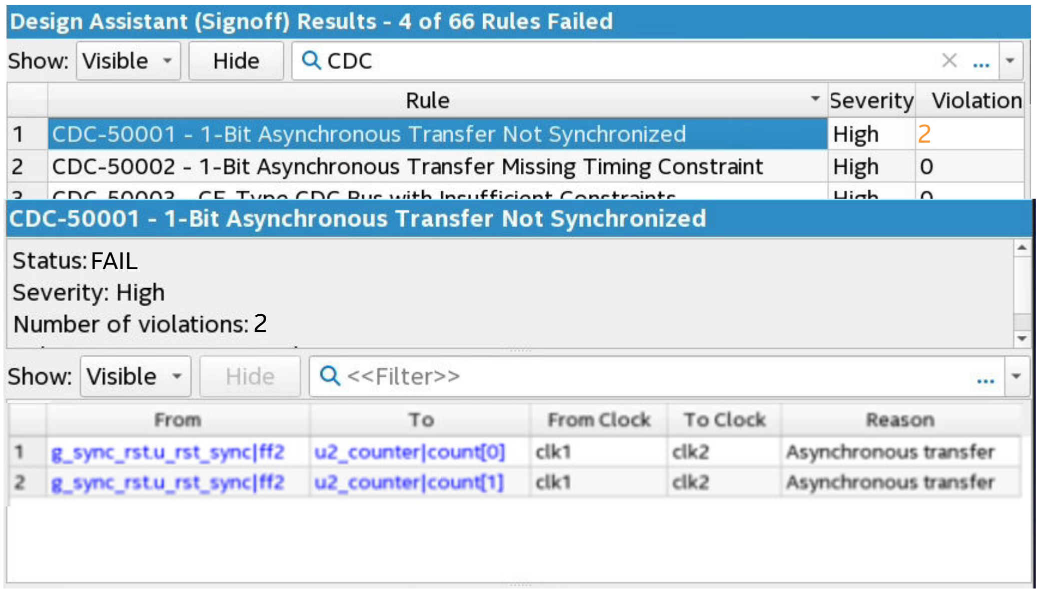
Figure 16 on page 14 shows an example where the reset synchronizer clocked on the `clk1` domain connects to the `u2_counter` instance from the `clk2` domain.

Figure 16. Reset Synchronizer used on Multiple Clock Domain



Design Assistant reports the following CDC rule violations in the **Timing Analyzer** folder of the **Compilation Report** for this condition:

Figure 17. Design Rule CDC-50001 Violations


Design Assistant (Signoff) Results - 4 of 66 Rules Failed

Show: Visible | Hide | CDC

	Rule	Severity	Violation
1	CDC-50001 - 1-Bit Asynchronous Transfer Not Synchronized	High	2
2	CDC-50002 - 1-Bit Asynchronous Transfer Missing Timing Constraint	High	0
3	CDC-50003 - CE-Type CDC Bus with Insufficient Constraints	High	0

CDC-50001 - 1-Bit Asynchronous Transfer Not Synchronized

Status: FAIL
Severity: High
Number of violations: 2

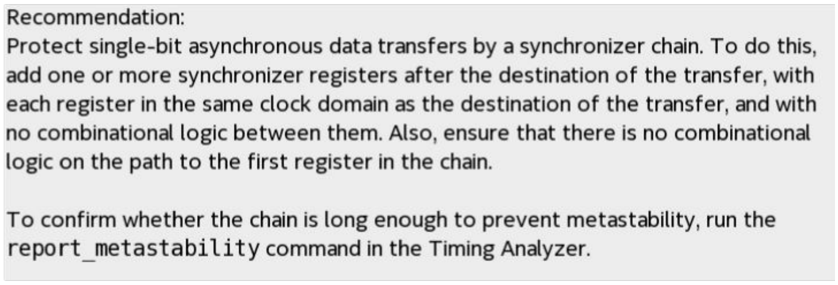
Show: Visible | Hide | <<Filter>>

	From	To	From Clock	To Clock	Reason
1	g_sync_rst.u_rst_sync|ff2	u2_counter|count[0]	clk1	clk2	Asynchronous transfer
2	g_sync_rst.u_rst_sync|ff2	u2_counter|count[1]	clk1	clk2	Asynchronous transfer

Rule check CDC-50001 identifies two paths originating from FF2 of the reset synchronizer clocked by `clk1`, driving the `u2_counter` registers clocked by `clk2`. To resolve the violation for this example:

1. Review the Design Assistant rule Description and Recommendation:

Figure 18. Rule CDC-50001 Recommendation

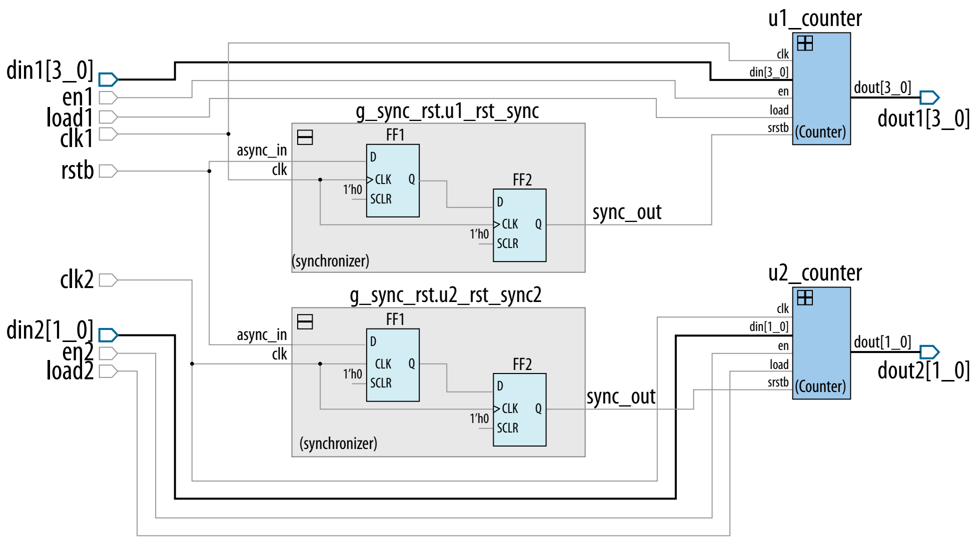


Recommendation:
Protect single-bit asynchronous data transfers by a synchronizer chain. To do this, add one or more synchronizer registers after the destination of the transfer, with each register in the same clock domain as the destination of the transfer, and with no combinational logic between them. Also, ensure that there is no combinational logic on the path to the first register in the chain.

To confirm whether the chain is long enough to prevent metastability, run the `report_metastability` command in the Timing Analyzer.

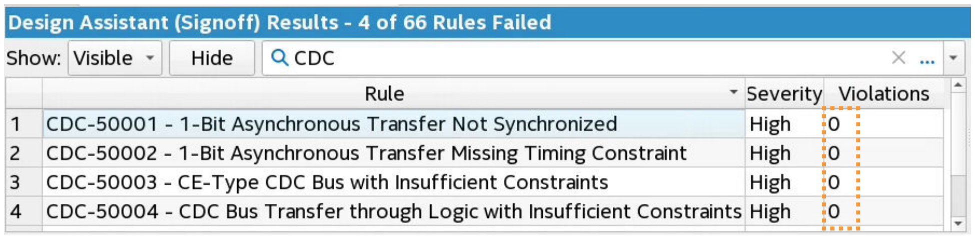
2. Implement a separate reset synchronizer for every clock domain, as the following figure shows:

Figure 19. Separate Reset Synchronizer per Clock Domain



3. Recompile the design changes. The synchronizer implementation corrects the CDC rule violations.

Figure 20. CDC Rule Violations Corrected



	Rule	Severity	Violations
1	CDC-50001 - 1-Bit Asynchronous Transfer Not Synchronized	High	0
2	CDC-50002 - 1-Bit Asynchronous Transfer Missing Timing Constraint	High	0
3	CDC-50003 - CE-Type CDC Bus with Insufficient Constraints	High	0
4	CDC-50004 - CDC Bus Transfer through Logic with Insufficient Constraints	High	0

1.4. Implementing Reset Circuitry

Improper application of resets in your design can cause operational design failures. The following sections provide reset implementation coding techniques and examples that comply with the following implementation guidelines. Follow these guidelines to properly implement reset and obtain the best timing possible:

- Ensure that every register that must power-up or reset to a known value connects to a reset.
- Avoid unnecessary resets to lessen possibility of routing congestion.
- Use synchronous resets to allow retiming into Hyper-Registers. Hyper-Registers used on retiming cannot be reset asynchronously.
- Use a regular two register chain synchronizer for resets coming in asynchronously, before feeding them to synchronous reset pins of registers. Provide a reset synchronizer for every clock domain.
- Use a VDD-based synchronizer for resets that connect to asynchronous reset pins. Provide a synchronizer for every clock domain.

Refer to the following topics for implementing reset circuitry:

- Reset Coding Techniques on page 16
- Avoiding Common Reset Coding Issues on page 17
- Generating Synchronous Reset Trees on page 20
- Reset Removal on Multi Clock Domain Designs on page 24
- Using the Reset Release Intel FPGA IP on page 26
- Adding Clock Cycles to the Reset Sequence on page 28

1.4.1. Reset Coding Techniques

Use the following coding techniques for reset implementation and conversion of asynchronous to synchronous resets.

1.4.1.1. Converting to Synchronous Resets

Complete the coding of your design before targeting an Intel Hyperflex architecture device. If you are using asynchronous resets in your design, convert them to synchronous resets to benefit from Hyper-Retiming performance optimization. Hyper-Retiming helps to meet the timing requirements for fast running clocks. The design blocks that are running slow clocks, and are already passing timing requirements, can continue using asynchronous resets.

Switching your code to use synchronous resets is simple, as the following example shows:

```
//Asynchronous reset on control register
always @(posedge clk or negedge rstb)
    begin
        if (!rstb)
            control <= ‘d0;
        else
            control <= new_value;
    end
```

```
//Synchronous reset on control register
always @(posedge clk)
    begin
        if (!rstb)
            control <= ‘d0;
        else
            control <= new_value;
    end
```

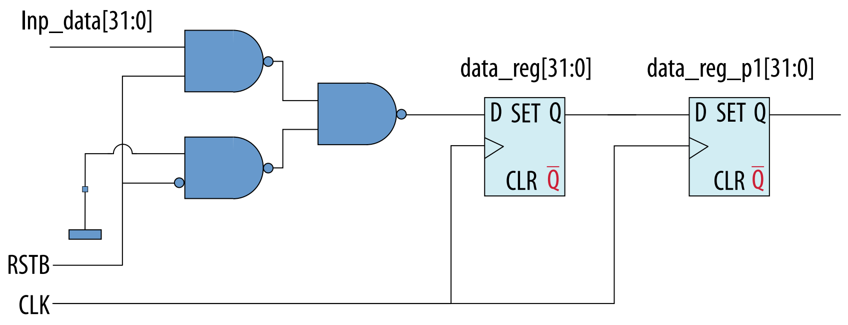
1.4.2. Avoiding Common Reset Coding Issues

The following sections describe how to avoid common reset coding issues that can limit performance or introduce unnecessary logic.

1.4.2.1. Coding Synchronous Reset with Follower Registers

Some registers with synchronous resets have follower registers or a pipeline immediately following, as the following example shows:

Figure 21. Resettable Flop with Register Follower

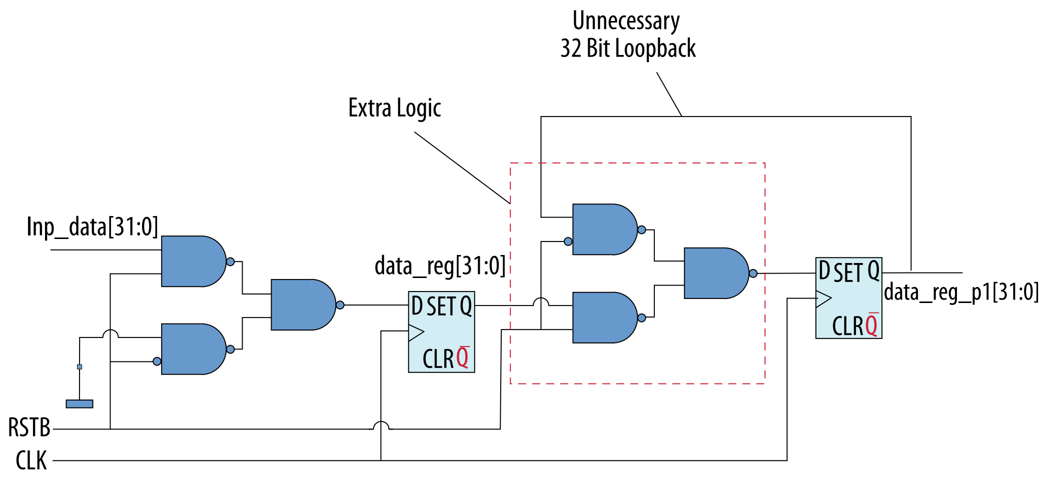


The following example shows incorrect coding for this circuit:

```
always @(posedge clk)
begin
    if (!rstb)
        data_reg[31:0] <= 32‘d0;
    else
        begin
            data_reg[31:0] <= inp_data[31:0];
            data_reg_p1[31:0] <= data_reg[31:0];
        end
end
```

Unfortunately, the incorrect coding results in unnecessary logic, as the following figure shows:

Figure 22. Synthesized Circuit for Incorrect Register Follower Code



The incorrect coding produces unnecessary logic before the 32-bit follower register `data_reg_p1`. The code uses `rstb` as a condition to load the output of the previous register `data_reg` or retains its current value, which creates an unnecessary 32-bit loopback. Aside from the routing congestion that the loopback creates, this loopback limits the ability of the register `data_reg_p1` from being retimed to help resolve critical path issues.

The rules are:

- Each verilog procedural block or VHDL process must model only one type of flip-flop.
- Do not combine resettable and non-resettable flip-flops in the same procedural block or process.

The following example shows proper coding for this circuit:

```
always @(posedge clk)
begin
    if (!rstb)
        data_reg[31:0] <= 32’d0;
    else
        begin
            data_reg[31:0] <= inp_data[31:0];
        end
end

always @(posedge clk)
begin
        data_reg_p1[31:0] <= data_reg[31:0];
end
```

1.4.2.2. Coding Reset-Related Optimizations

As mentioned, you can implement the following reset related coding changes to fully enable Hyper-Retiming optimization across your design:

- Convert asynchronous resets to synchronous resets.
- Remove resets on datapath registers or pipeline registers when the functionality allows, to lessen routing congestion in areas where the data registers are involved.

However, if applied incorrectly, these reset related optimization can actually cause retiming restrictions.

For example, consider the following initial code with a mix of control signals and data registers:

```
always @(posedge clk or negedge rstb)
begin
    if (!rstb)
        begin
            any_control <= 1’b0;
            data[31:0] <= 32‘d0;
            data_d1[31:0] <= 32‘d0;
            data_d2[31:0] <= 32‘d0;
        end
    else
        begin
            any_control <= s_valid & s_ok;
            data[31:0] <= sdata[31:0];
            data_d1[31:0] <= data[31:0];
            data_d2[31:0] <= data_d1[31:0];
        end
end
```

After converting the code to use synchronous reset, the code may appear as follows:

```
always @(posedge clk)
begin
    if (!rstb)
        begin
            any_control <= 1’b0;
            data[31:0] <= 32‘d0;
            data_d1[31:0] <= 32‘d0;
            data_d2[31:0] <= 32‘d0;
        end
    else
        begin
            any_control <= s_valid & s_ok;
            data[31:0] <= sdata[31:0];
            data_d1[31:0] <= data[31:0];
            data_d2[31:0] <= data_d1[31:0];
        end
end
```

However, if you decide to only remove the resets on data pipelines `data_d1[31:0]` and `data_d2[31:0]`, the result mixes registers with and without reset in the same if-else procedure. The following example code shares characteristics with the example in Coding Synchronous Reset with Follower Registers on page 17:

```
always @(posedge clk)
begin
    if (!rstb)
        begin
            any_control <= 1’b0;
            data[31:0] <= 32‘d0;
        end
    else
        begin
            any_control <= s_valid & s_ok;
            data[31:0] <= sdata[31:0];
            data_d1[31:0] <= data[31:0];
            data_d2[31:0] <= data_d1[31:0];
        end
end
```

data_d1[31:0] and data_d2[31:0] use rstb as a condition, whether to update or retain their previous values. This coding creates unnecessary logic and 32-bit loopbacks on the registers.

The following example shows a more effective method for code with mixed control signals and data pipelines:

```
always @(posedge clk)
begin

    any_control <= s_valid & s_ok;
    data[31:0] <= sdata[31:0];
    data_d1[31:0] <= data[31:0];
    data_d2[31:0] <= data_d1[31:0];

    if (!rstb)
    begin
        any_control <= 1'b0;
        data[31:0] <= 32'd0;
        ~~data_d1[31:0] <= 32'd0;~~
        ~~data_d2[31:0] <= 32'd0;~~
    end
end
```

Removing resets on data_d1[31:0] and data_d2[31:0] in this code is safe. This method also avoids creating unnecessary loopbacks and does not introduce any retiming restrictions.

Signals any_control and data[31:0] have multiple assignments when rstb is '0'. The Verilog HDL standard defines the Verilog HDL coding of multiple non-blocking assignments to the same variable, in the same always block. The last non-blocking assignment to the same variable takes precedence.

Related Information

Coding Synchronous Reset with Follower Registers on page 17

1.4.3. Generating Synchronous Reset Trees

A synchronous reset can easily fan-out to a large destination, like any broadcast signal. While the use of synchronous reset over asynchronous reset eases timing restrictions, this technique does not remove all performance restrictions. You must use reset in a careful manner. The logic that does not require synchronous reset can help with timing. You need not remove all resets, especially those that are running on slower clock domains. Focus reset strategies on design blocks that must run at high-speed.

To fix a large fan-out on a synchronous reset, you must duplicate and pipeline the reset to provide retiming ability on all the data paths using the reset as part of the logic.

1.4.3.1. Reset Distribution through Hyper-Pipelining and Hyper-Retiming

You can create a synchronous reset tree by duplicating the reset on every major design submodule that requires a reset on the same clock edge. You perform this duplication at the top level. You must add enough pipeline registers on every duplicate branch for use during retiming. You must ensure a balanced reset tree to have the same latency at all endpoints.

This manual process is the minimum duplication requirement to limit the fan-out of the synchronous reset. Thereafter, the Compiler attempts to retime the reset with the connected local logic. Depending on the logic size of every submodule driven by the duplicate branch, the Compiler may need to work harder to resolve all the reset path timing for every local duplicate branch.

You can add the `preserve_syn_only` attribute to the inserted pipeline registers to prevent the Compiler from optimizing out the registers during synthesis. For example:

```
Verilog HDL

(*preserve_syn_only*) reg dup_reg;
```

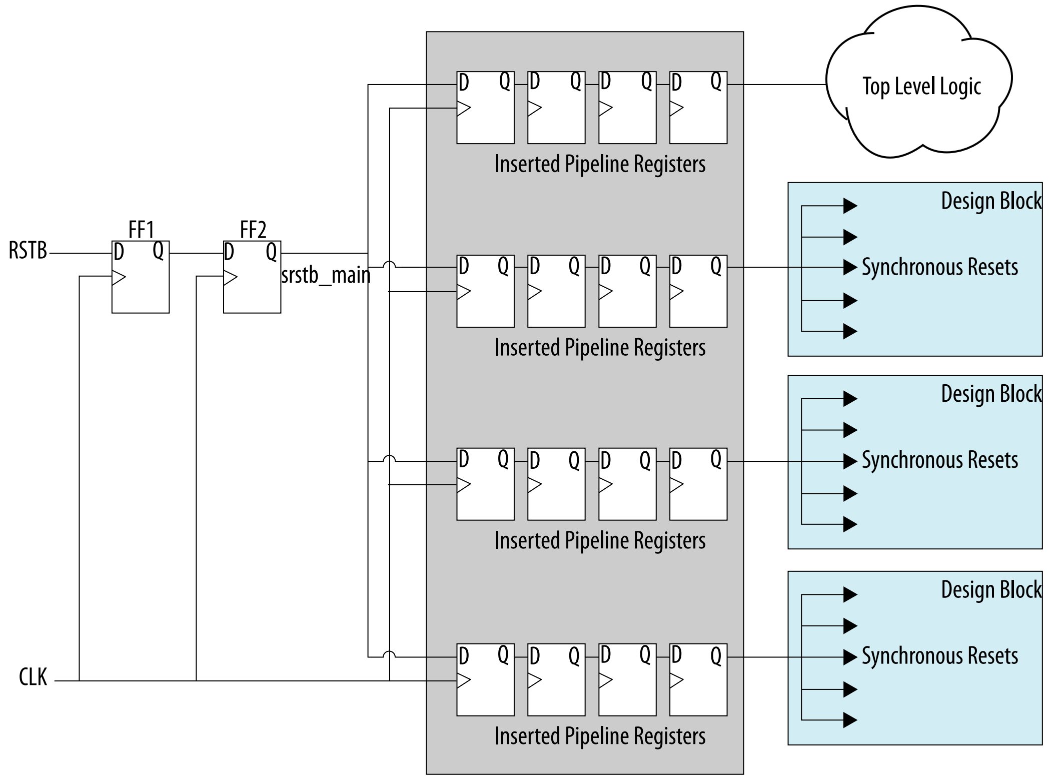
```
VHDL

signal dup_reg : std_logic;
attribute preserve_syn_only : boolean;
attribute preserve_syn_only of dup_reg : signal is true;
```

Figure 23 on page 21 shows the circuit structure after inserting pipeline registers and duplicating this on every synchronous reset branch at the top level. After adding the registers, you compile the design to allow retiming of the registers as needed to resolve timing issues.

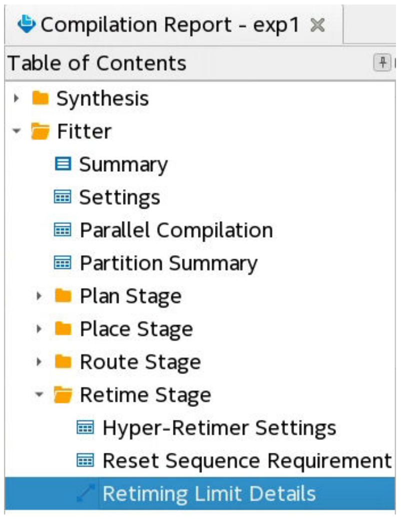
Important: To reset all the registers of a clock domain in the same clock cycle, the number of inserted pipeline registers must match in each branch.

Figure 23. Inserting Pipeline Registers on Synchronous Reset Tree



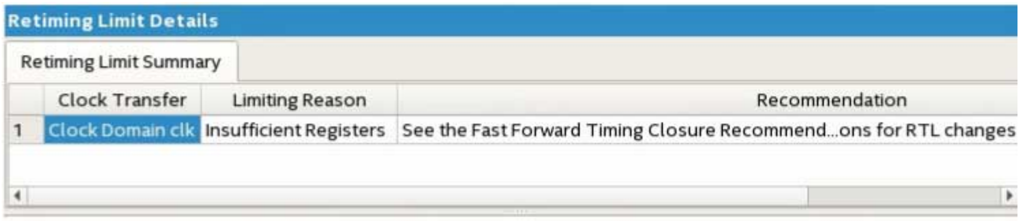
After the Retime stage, you can review the Retiming Limit Details Report to determine whether the number of pipeline registers is a limiting factor in meeting timing requirements, as the following example shows:

Figure 24. Retiming Limit Details in Fitter Retime Stage Folder



The **Limiting Reason** column specifies `Insufficient Registers`. The **Critical Chain Details** shows if the critical path is caused by the pipeline registers. You can add an additional pipeline register on all branches to keep them balanced and then recompile.

Figure 25. Retiming Limit Detail Report


Retiming Limit Details

Retiming Limit Summary

	Clock Transfer	Limiting Reason	Recommendation
1	Clock Domain clk	Insufficient Registers	See the Fast Forward Timing Closure Recommend...ons for RTL changes

1.4.3.2. Adding Pipeline Registers and Automatic Register Duplication

The `DUPLICATE_HIERARCHY_DEPTH` assignment uses design hierarchy information to guide the creation of duplicates and their fan-out assignments during synthesis.

```
set_instance_assignment -name DUPLICATE_HIERARCHY_DEPTH -to <register_name>
<num_levels>
```

Where `register_name` is the last register in a chain that fans out to multiple hierarchies. To create a register tree, ensure that there are sufficient simple registers behind the node, and those simple registers are automatically pulled into the tree.

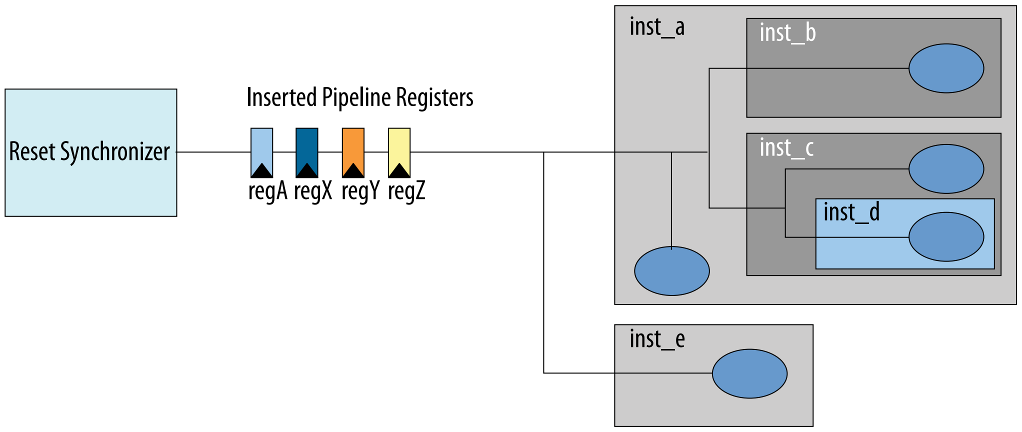
`num_levels` corresponds to the upper bound of the number of registers that exist in the chain, to use for duplicating down the hierarchies.

The registers in the chain must satisfy all the following conditions to be included in duplication:

- Registers must be fed only by another register and cannot be fed by combinational logic.
- Registers cannot be part of a synchronizer chain.
- Registers cannot have any secondary signals.
- Registers cannot have a `preserve` attribute or a PRESERVE_REGISTER assignment.
- All registers in the chain except the last one must have only one fan-out.

Consider the following design with pipeline registers `regA`, `regX`, `regY`, and `regZ` added after the reset synchronizer block:

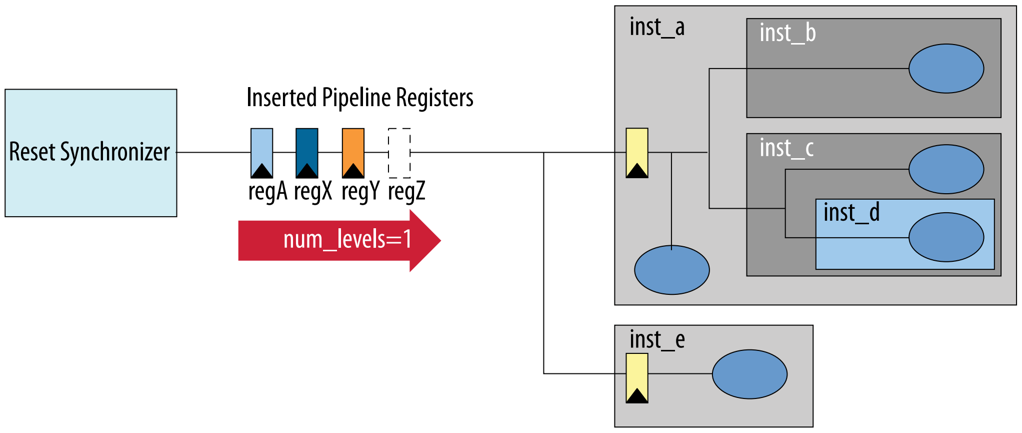
Figure 26. Pipeline Registers Added to Reset Synchronizer Block



```
set_instance_assignment -name DUPLICATE_HIERARCHY_DEPTH -to regZ 1
```

When `num_levels` is set to 1, only `regZ` is pulled out of the chain and pushed down by one hierarchy level into its fan-out tree.

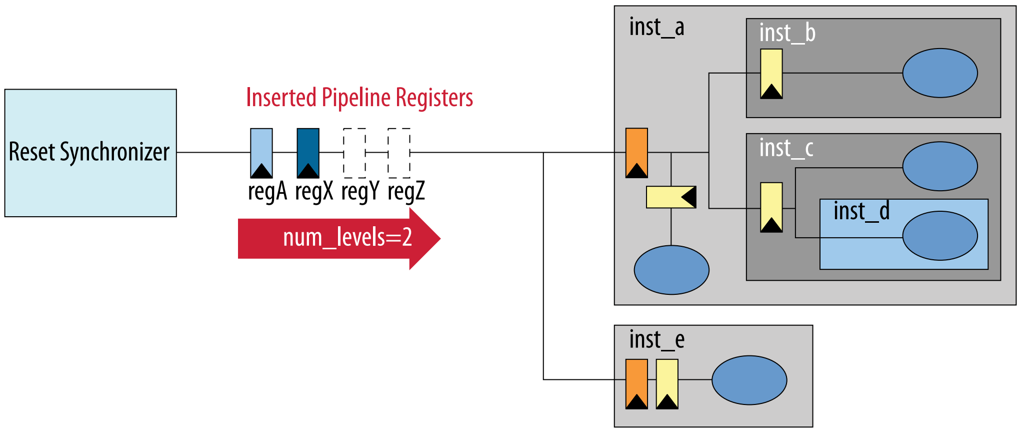
Figure 27. num_levels set to 1



```
set_instance_assignment -name DUPLICATE_HIERARCHY_DEPTH -to regZ 2
```

When `num_levels` is set to 2, both `regY` and `regZ` are pulled out of the chain. `regZ` stops at the maximum hierarchy depth 2 and `regY` stops at hierarchy depth 1.

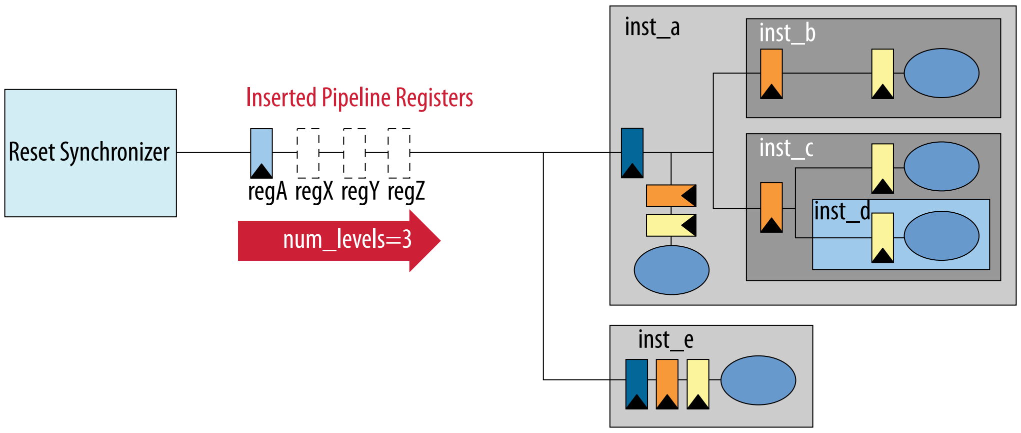
Figure 28. num_levels is set to 2



```
set_instance_assignment -name DUPLICATE_HIERARCHY_DEPTH -to regZ 3
```

When `num_levels` is set to 3, `regX`, `regY`, and `regZ` are pulled out of the chain and pushed to a maximum depth of 1, 2, and 3 levels respectively.

Figure 29. num_levels is set to 3



The Synthesis report includes the Hierarchical Tree Duplication Summary report that provides information on the registers impacted by the `DUPLICATE_HIERARCHY_DEPTH` assignment. You can refer to the report to set or modify the assignments and further improve the results.

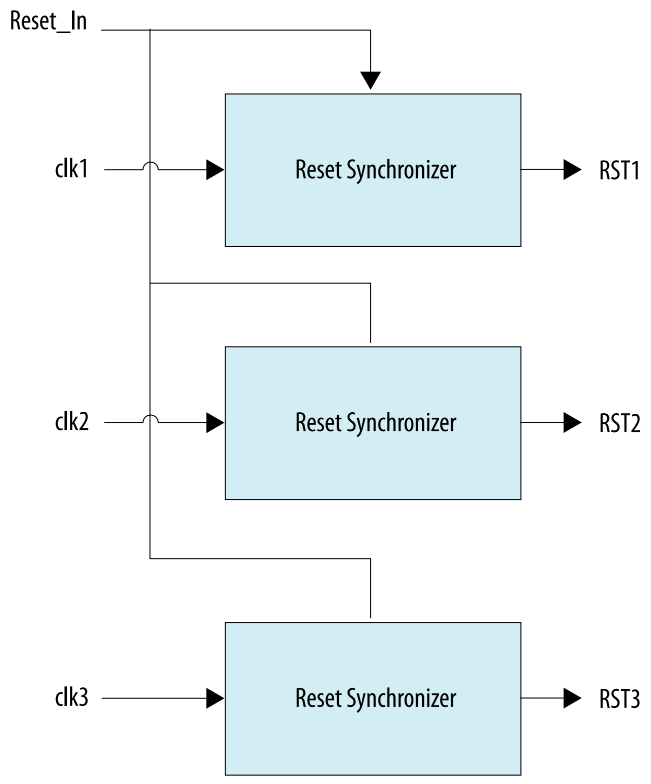
1.4.4. Reset Removal on Multi Clock Domain Designs

Depending on the specifications of your design, you can use the following techniques for reset removal on multi clock domain designs:

- Non-Coordinated Reset Removal
- Sequenced or Ordered Reset Removal

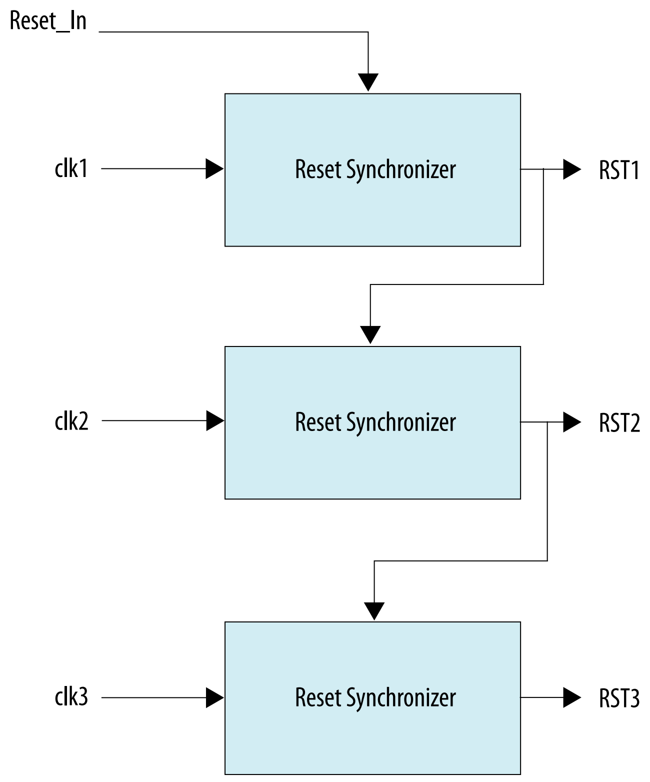
In a design with multiple clock domains, you must have a reset synchronizer for each clock domain. If the whole design is not sensitive to the removal of reset sequencing, the resets can be non-coordinated. In this case, every clock domain can generate multiple reset domain signals to synchronize the main reset, as shown in the following figure.

Figure 30. Non-Coordinated Reset Removal



Carefully determine if different block domains require reset in sequence. Identify the proper reset removal sequencing to ensure that the block domain dependency is not corrupted by the removal of a different reset domain. The following figure shows an example of reset removal sequencing.

Figure 31. Ordered Reset Removal



1.4.5. Using the Reset Release Intel FPGA IP

During the device configuration, the global configuration control signals hold the core fabric in a frozen state to prevent electrical contention. Sectors that comprise multiple logic array block (LAB) rows are all asynchronously unfrozen by different Local Sector Managers (LSM). Within each sector, LAB rows and registers are released sequentially by each LSM. Consequently, different logic can operate while other logic remains frozen during the process.

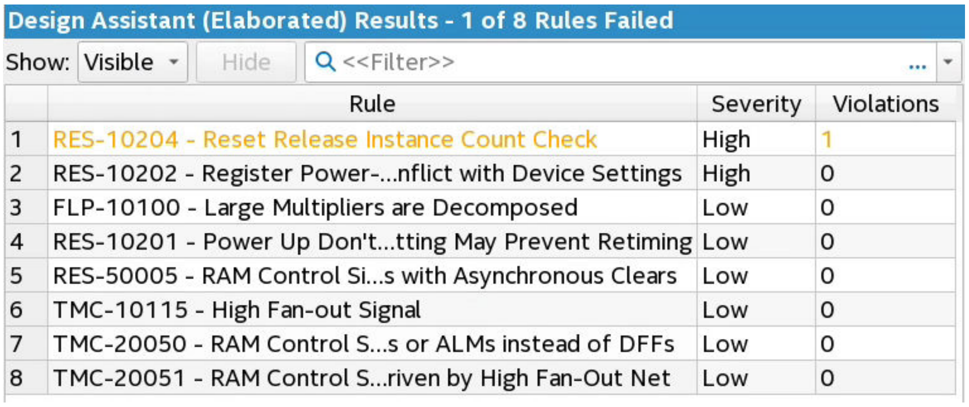
If the activity of the logic partly becomes operational, this could potentially cause some control logic or state machines without a reset strategy to enter an illegal or unknown state, once the entire fabric goes into the user mode.

Therefore, after every compile, the Intel Quartus Prime software generates the following critical warning message:

```
Use the Reset Release IP in Intel Stratix 10 FPGA
 designs to ensure a successful configuration. For more information about the
Reset Release IP, refer to the Intel Stratix 10 Configuration User Guide.
```

The Design Assistant report in the Synthesis folder also includes RES-10204 rule violation - Reset Release Instance Count Check.

Figure 32. RES-10204 Rule Violation - Reset Release Instance Count Check



Design Assistant (Elaborated) Results - 1 of 8 Rules Failed

	Rule	Severity	Violations
1	RES-10204 - Reset Release Instance Count Check	High	1
2	RES-10202 - Register Power-...nflict with Device Settings	High	0
3	FLP-10100 - Large Multipliers are Decomposed	Low	0
4	RES-10201 - Power Up Don't...tting May Prevent Retiming	Low	0
5	RES-50005 - RAM Control Si...s with Asynchronous Clears	Low	0
6	TMC-10115 - High Fan-out Signal	Low	0
7	TMC-20050 - RAM Control S...s or ALMs instead of DFFs	Low	0
8	TMC-20051 - RAM Control S...riven by High Fan-Out Net	Low	0

Design Assistant categorizes the lack of Reset Release Intel FPGA IP as HIGH severity. Without the IP, intermittent functional issues could result on every design power-up.

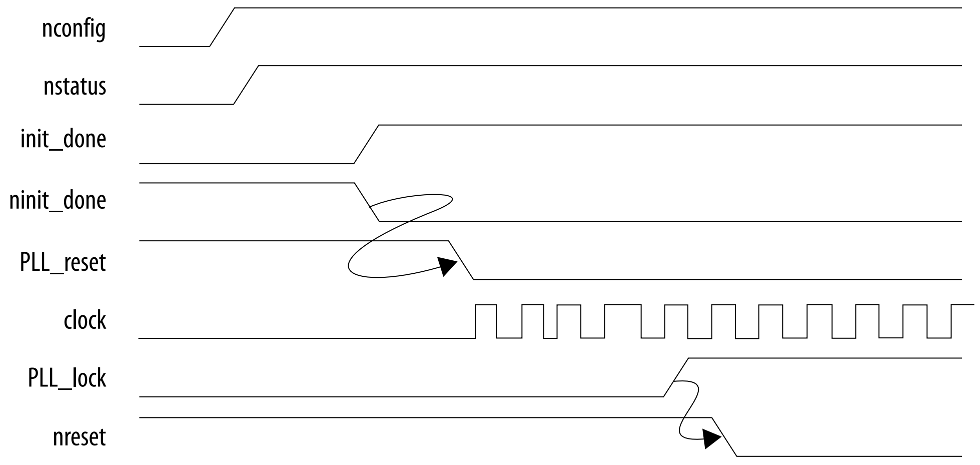
The Reset Release IP can hold the design at reset until the device has fully entered the user mode. You can specify two options to signal the safe entry to the user mode:

- Using the `INIT_DONE` output PIN signal
- Using the `nINIT_DONE` output from the Reset Release IP

Refer to the Related Information for detailed information about connecting your design with the Reset Release Intel FPGA IP.

Designs commonly use the PLL lock signal to hold the design logic in reset until the PLL is locked. To make sure that the lock happens after the device completes initialization, you must gate the PLL reset input with `nINIT_DONE`, as shown in the following figure.

Figure 33. Using `nINIT_DONE` on PLL_Reset Signal to Generate System `nReset`



Related Information

- Intel Stratix 10 FPGA Configuration Flow
- Intel Agilex Configuration User Guide

1.4.6. Adding Clock Cycles to the Reset Sequence

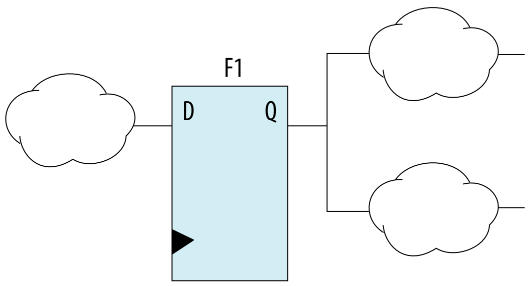
You may need to apply one or more extra clock cycles to the reset sequence after power-up to ensure the functional equivalence of the design after retiming. The number of clock cycles a that a design requires after power-up is the "c-cycle" value. The following describes how to add clock cycles to the reset sequence.

1.4.6.1. C-Cycle Equivalence

The c-cycle refers to the number of clock cycles a design requires after power-up to ensure functional equivalence. The c-cycle value is an important consideration in structuring the reset sequence of your design.

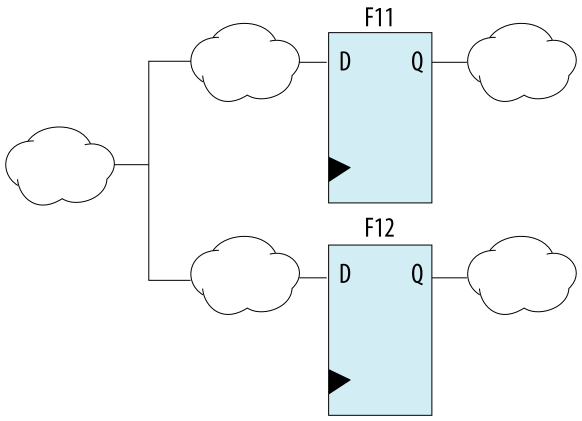
Consider the following simple circuit where register F1 at power-up can have either state ‘0’ or state ‘1’. Assuming the clouds of logic are purely combinational, there are 2 possible states in the circuit, F1 = ‘0’ or F1 = ‘1’.

Figure 34. Circuit Before Retiming



If the retimer pushes F1 forward, then the register is duplicated in each branch that F1 drives.

Figure 35. Register Duplicated



After retiming and register duplication, the circuit now has four possible states at power-up. The addition of two potential states in the circuit after retiming, potentially changes the deign functionality.

Table 2. Registers F11 and F12 States

F11 States	F12 States
0	0
0	1
1	0
1	1

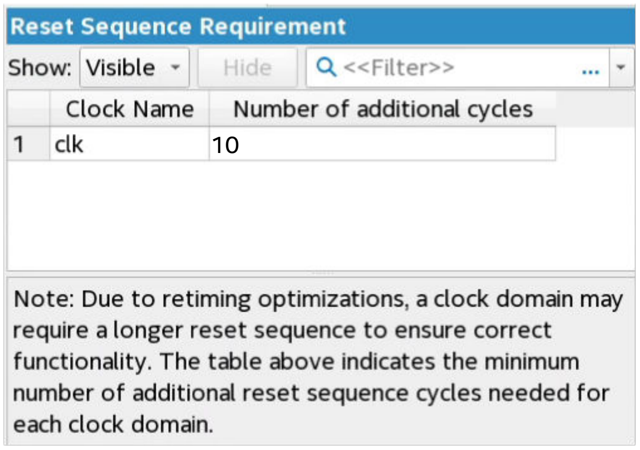
Apply an extra clock cycle after power-up to ensure the functional equivalence of the design after retiming. The extra cycle ensures that the states of F11 and F12 are always identical resulting in two possible states for the registers, 0/0 or 1/1, assuming the combinational logic is non-inverting on both paths.

Backward retiming is always a safe operation with c-cycle value of 0. The Compiler always permits merging if you do not specify initial conditions for F11 and F12. If you specify initial conditions, the Compiler accounts for those initial states and retiming transformation occurs only if the initial states are preserved.

1.4.6.2. Determining the Reset Sequence Requirement

You can view the c-cycle value in the Reset Sequence Requirement section of the Compilation Report. The report lists the number of cycles to add on a clock domain basis. The following Reset Sequence Requirement report indicates that the `clk` domain requires adding 10 additional cycles to the reset sequence to ensure correct functionality.

Figure 36. Reset Sequence Requirement Report



	Clock Name	Number of additional cycles
1	clk	10

Note: Due to retiming optimizations, a clock domain may require a longer reset sequence to ensure correct functionality. The table above indicates the minimum number of additional reset sequence cycles needed for each clock domain.

For more information about initial power-up conditions and retiming reset sequences, refer to the *Intel Hyperflex Architecture High-Performance Design Handbook*.

Related Information

- Intel Hyperflex Architecture High-Performance Design Handbook
- Asynchronous and Synchronous Reset Design Techniques, SNUG Boston 2003

1.5. Document Revision History for AN 917: Reset Design Techniques for Intel Hyperflex Architecture FPGAs

Document Version	Intel Quartus Prime Version	Changes
2021.01.05	20.4	Initial release of the document.