

Intel® Stratix® 10 High-Performance Design Handbook

Updated for Intel® Quartus® Prime Design Suite: **17.1**



Contents

1 Intel® Hyperflex™ FPGA Architecture Introduction.....	4
1.1 Intel Stratix 10 Basic Design Concepts.....	5
2 RTL Design Guidelines.....	6
2.1 High-Speed Design Methodology.....	6
2.1.1 Set a High-Speed Target.....	6
2.1.2 Experiment and Iterate.....	8
2.1.3 Compile Components Independently.....	8
2.1.4 Optimize Sub-Modules.....	9
2.1.5 Avoid Broadcast Signals.....	9
2.2 Hyper-Retiming (Facilitate Register Movement).....	11
2.2.1 Reset Strategies.....	13
2.2.2 Clock Enable Strategies.....	18
2.2.3 Synthesis Attributes.....	19
2.2.4 Timing Constraint Considerations.....	19
2.2.5 Clock Synchronization Strategies.....	20
2.2.6 Metastability Synchronizers.....	22
2.2.7 Initial Power-Up Conditions.....	22
2.3 Hyper-Pipelining (Add Pipeline Registers).....	26
2.3.1 Conventional Versus Hyper-Pipelining.....	27
2.3.2 Pipelining and Latency.....	28
2.3.3 Use Registers Instead of Multicycle Exceptions.....	29
2.4 Hyper-Optimization (Optimize RTL).....	29
2.4.1 General Optimization Techniques.....	29
2.4.2 Optimizing Specific Design Structures.....	41
3 Compiling Intel Stratix 10 Designs.....	62
4 Design Example Walk-Through.....	64
4.1 Median Filter Design Example.....	64
4.1.1 Step 1: Compile the Base Design.....	65
4.1.2 Step 2: Add Pipeline Stages and Remove Asynchronous Resets.....	67
4.1.3 Step 3: Add More Pipeline Stages and Remove All Asynchronous Resets.....	69
4.1.4 Step 4: Optimize Short Path and Long Path Conditions.....	71
5 Retiming Restrictions and Workarounds.....	74
5.1 Interpreting Critical Chain Reports.....	76
5.1.1 Insufficient Registers.....	77
5.1.2 Short Path/Long Path.....	80
5.1.3 Fast Forward Limit.....	84
5.1.4 Loops.....	85
5.1.5 One Critical Chain per Clock Domain.....	89
5.1.6 Critical Chains in Related Clock Groups.....	89
5.1.7 Complex Critical Chains.....	89
5.1.8 Extend to locatable node.....	91
5.1.9 Domain Boundary Entry and Domain Boundary Exit.....	91
5.1.10 Critical Chains with Dual Clock Memories.....	93
5.1.11 Critical Chain Bits and Buses.....	94



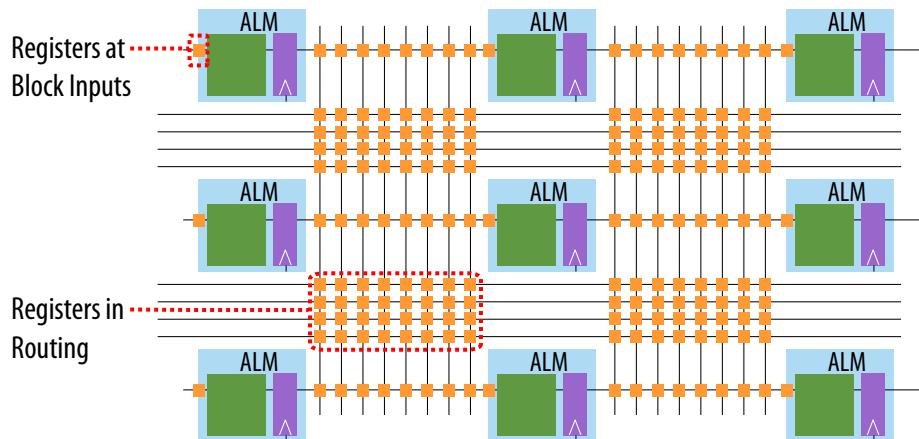
5.1.12 Delay Lines.....	94
6 Optimization Example.....	95
6.1 Round Robin Scheduler.....	95
7 Intel Hyperflex Architecture Porting Guidelines.....	101
7.1 Design Migration and Performance Exploration.....	101
7.1.1 Black-boxing Verilog HDL Modules.....	102
7.1.2 Black-boxing VHDL Modules.....	102
7.1.3 Clock Management.....	104
7.1.4 Pin Assignments.....	104
7.1.5 Transceiver Control Logic.....	105
7.1.6 Upgrade Outdated IP Cores.....	106
7.2 Top-Level Design Considerations.....	106
8 Appendices.....	108
8.1 Appendix A: Parameterizable Pipeline Modules.....	109
8.2 Appendix B: Clock Enables and Resets.....	111
8.2.1 Synchronous Resets and Limitations.....	111
8.2.2 Retiming with Clock Enables.....	115
8.2.3 Resolving Short Paths.....	119
9 Document Revision History.....	121

1 Intel® Hyperflex™ FPGA Architecture Introduction

This document describes design techniques to achieve maximum performance with the Intel® Hyperflex™ FPGA architecture. This architecture supports new Hyper-Retiming, Hyper-Pipelining, and Hyper-Optimization design techniques that enable the highest clock frequencies for Intel Stratix® 10 devices.

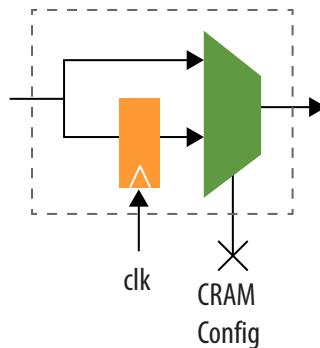
"Registers everywhere" is a key innovation of the Intel Hyperflex FPGA architecture. Intel Stratix 10 devices pack bypassable Hyper-Registers into every routing segment in the device core, and at all functional block inputs.

Figure 1. Registers Everywhere



With Intel Stratix 10 bypassable Hyper-Registers, the routing signal can travel through the register first, or bypass the register direct to the multiplexer. One bit of the FPGA configuration memory (CRAM) controls this multiplexer.

Figure 2. Bypassable Hyper-Registers





The chapters in this document provide specific design guidelines, tool flows, and real world examples to quickly take advantage of the Intel Hyperflex FPGA architecture:

- *RTL Design Guidelines*—provides fundamental high-performance RTL design techniques for Intel Stratix 10 designs.
- *Compiling Intel Stratix 10 Designs*—describes using the Intel Quartus® Prime Pro Edition software to get the highest performance in Intel Stratix 10 devices.
- *Intel Hyperflex Architecture Porting Guidelines*—provides guidance for design migration to Intel Stratix 10 devices.
- *Design Example Walk-Through, Optimization Example, and the Appendices*—demonstrate performance improvement techniques using real design examples.

1.1 Intel Stratix 10 Basic Design Concepts

Table 1. Glossary

Term/Phrase	Description
Critical Chain	Any design condition that prevents retiming of registers is the critical chain. The limiting factor may include more than one register-to-register path in a chain. The f_{MAX} of the critical chain and its associated clock domain is limited by the average delay of a register-to-register path, and quantization delays of indivisible circuit elements like routing wires. Fast Forward compilation breaks critical chains.
Fast Forward Compilation	Generates design-specific timing closure recommendations, and forward-looking performance results after removal of each timing restriction.
Hyper-Aware Design Flow	Design flow that enables the highest performance in Intel Stratix 10 devices through Hyper-Retiming, Hyper-Pipelining, Fast Forward compilation, and Hyper-Optimization.
Intel Hyperflex FPGA Architecture	Intel Stratix 10 device core architecture that includes additional registers, called Hyper-Registers, everywhere throughout the core fabric. Hyper-Registers provide increased bandwidth and improved area and power efficiency.
Hyper-Optimization	Design process that improves design performance through implementation of key RTL changes recommended by Fast Forward compilation, such as restructuring logic to use functionally equivalent feed-forward or pre-compute paths, rather than long combinatorial feedback paths.
Hyper-Pipelining	Design process that eliminates long routing delays by adding additional pipeline stages in the interconnect between the ALM registers. This technique allows the design to run at a faster clock frequency.
Hyper-Retiming	During Fast Forward compile, Hyper-Retiming speculatively removes signals from registers to enable mobility in the netlist for retiming.
Multiple Corner Timing Analysis	Analysis of multiple "timing corner cases" to verify your design's voltage, process, and temperature operating conditions. Fast-corner analysis assumes best-case timing conditions.

Related Links

- [Hyper-Retiming \(Facilitate Register Movement\)](#) on page 11
- [Hyper-Pipelining \(Add Pipeline Registers\)](#) on page 26
- [Hyper-Optimization \(Optimize RTL\)](#) on page 29



2 RTL Design Guidelines

This chapter describes RTL design techniques to achieve the highest clock rates possible in Intel Stratix 10 devices. The Intel Stratix 10 architecture supports maximum clock rates significantly higher than previous FPGA generations.

2.1 High-Speed Design Methodology

Migrating a design to the Intel Stratix 10 architecture requires implementation of high-speed design best practices to obtain the most benefit and preserve functionality. The Intel Stratix 10 high-speed design methodology produces latency-insensitive designs that support additional pipeline stages, and avoid performance-limiting loops. The following high-speed design best practices produce the most benefit for Intel Stratix 10 designs:

- Set a high-speed target
- Experiment and iterate
- Compile design components individually
- Optimize design sub-modules
- Avoid broadcast signals

The following sections describe specific RTL design techniques that enable Hyper-Retiming, Hyper-Pipelining, and Hyper-Optimization in the Intel Quartus Prime software.

2.1.1 Set a High-Speed Target

For silicon efficiency, set your speed target as high as possible. The Intel Stratix 10 LUT is essentially a tiny ROM capable of a billion lookups per second. Operating an Intel Stratix 10 LUT at 156 MHz uses only 15% of the capacity.

While setting a high-speed target, you must also maintain a comfortable guard band between the speed at which you can close timing, and the actual system speed required. Addressing the timing closure initially with margin is much easier.

2.1.1.1 Speed and Timing Closure

Failure to close timing occurs when actual circuit performance is lower than the f_{MAX} requirement of your design. If the target FPGA device has many available resources for logic placement, timing closure is easier and requires less processing time.

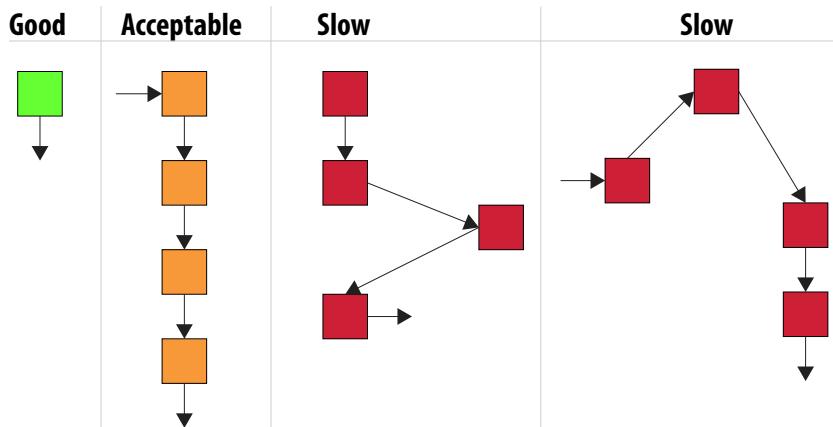
Timing closure of a slow circuit is not inherently easier than timing closure of a faster circuit, because slow circuits typically include more combinational logic between registers. When there are many nodes on a path, the Fitter must place nodes away from each other, resulting in significant routing delay. In contrast, a circuit that you heavily pipeline is much less dependent on placement, which simplifies timing closure.

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Empirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2008
Registered

Figure 3. Acceptable and Slow Routing Delays



Use realistic timing margins when creating your design. Consider that portions of the design make contact and distort one another as you add logic to the system. Adding stress to the system is typically detrimental to speed. Allowing more timing margin at the start of the design process helps mitigate this problem.

2.1.1.2 Speed and Latency

The following table illustrates the rate of growth for various types of circuits as the bus width increases. The circuit functions interleave with big O notations of area as a function of bus width, starting at sub-linear with $\log(N)$, to super-linear with $N*N$.

Table 2. Effect of Bus Width on Area

Bus Width (N)	Circuit Function						
	log N	Mux	ripple add	N*log N	barrel shift	Crossbar	N*N
16	4	5	16	64	64	80	256
32	5	11	32	160	160	352	1024
64	6	21	64	384	384	1344	4096
128	7	43	128	896	896	5504	16384
256	8	85	256	2048	2048	21760	65536

Typically, circuit components use more than 2X the area as the bus width doubles. For a simple circuit like a mux, the area grows sub-linearly as the bus width increases. Cutting the bus width of a mux in half provides slightly worse linear area benefit. A ripple adder grows linearly as the bus width increases.

More complex circuits, like barrel shifters and crossbars, grow super-linearly as bus width increases. If you cut the bus width of a barrel shifter, crossbar, or other complex circuit in half, the area benefit can be significantly better than half, approaching quadratic rates. For components in which all inputs affect all outputs, increasing the bus width can cause quadratic growth. The expectation is then that, if you take advantage of speed-up to work on half-width buses, you generate a design with less than half the original area.

When working with streaming datapaths, the number of registers is a fair approximation of the latency of the pipeline in bits. Reducing the width by half creates the opportunity to double the number of pipeline stages, without negatively impacting latency. This higher performance generally requires significantly less than double the amount of additional registering to create a latency profit.

2.1.2 Experiment and Iterate

Experiment with settings and design changes if your design's performance does not initially meet performance requirements. Intel FPGA reprogrammability allows experimentation until you achieve your goals. Commonly, a design performance gradually becomes inadequate as technology requirements increase over time. For example, if you apply an existing design element to a new context at a wider parameterization, the speed performance likely declines.

When experimenting with circuit timing, there is no permanent risk from experimentation that temporarily breaks the circuit to collect a data point. Add registers in illegal locations to determine the effect on overall timing. If the prospective circuit then meets the timing objective, you can make further investment to legalize the placement.

If a circuit remains too slow, even when liberally inserting registers, reconsider more basic elements of the design. Moving up or down a speed grade, or compressing circuitry in Logic Lock regions can help speed investigation.

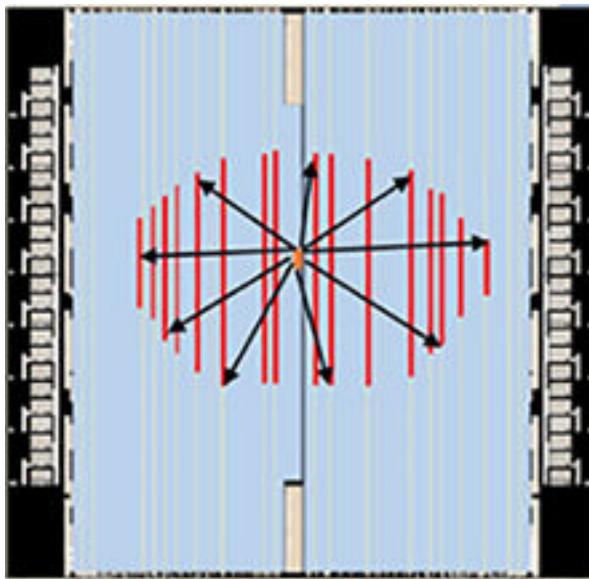
2.1.3 Compile Components Independently

Compile the design subcomponents as stand-alone entities to identify and optimize performance bottlenecks early on. Competition for resources and physical constraints (like pin locations) reduces overall design performance.

After you embed components at a higher level, the block speed may be the same. However, the speed may never be any faster with other components than alone. As a margin of safety, establish a bright line rule for the component speed you require. For example, when targeting a 20% timing margin, a component with 19.5% margin is a failure. Base your timing margin targets on the context. For example, you can allow a timing margin of 10% for a high-level component representing half the chip. However, if the rule is not explicit, the margin erodes as 10% becomes 9%, then 6%, and so on.

Use the Chip Planner to visualize the system level view. The following floorplan shows a component that uses 5% of the logic on the device (central orange) and 25% of the M20K blocks (red stripes).

Figure 4. M20K Spread in Chip Planner



The system level view does not show anything alarming about the resource ratios. However, significant routing congestion is apparent. The orange memory control logic fans out across a large physical span to connect to all of the memory blocks. The design functions satisfactorily alone, but struggles when unrelated logic cells occupy the intervening area. Restructuring this block to physically distribute the control logic better relieves the high-level problem.

Individual module compilation prepares you for subsequent hardware debug cycles. Independent, coherent operation of portions of the design is beneficial. This condition allows test and modification of only those sections, without the runtime and complexities of the entire system.

2.1.4 Optimize Sub-Modules

During design optimization, you can isolate the critical part in one or two sub-modules from a large design, and then compile the sub-modules. Compiling part of a design reduces compile time and allows you to focus on optimization of the critical part.

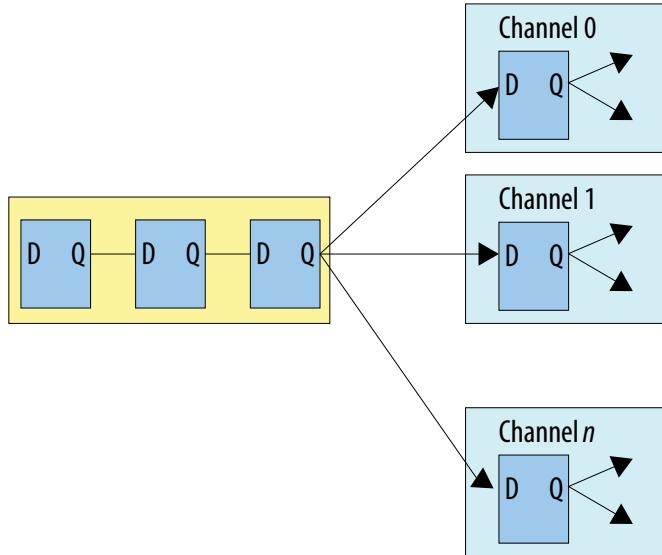
2.1.5 Avoid Broadcast Signals

Avoid using broadcast signals whenever possible. Broadcast signals are high fan-out control nets that can create large latency differences between paths. Path latency differences complicate the Compiler's ability to find a suitable location for registers, resulting in unbalanced delay paths. Use pipelining to address this issue and duplicate registers to drive broadcast signals.

Broadcast signals travel a large distance to reach individual registers. Because those fan-out registers may be spread out in the floorplan, use manual register duplication to improve placement. The correct placement of pipeline stages has a significant impact on performance.

Figure 5. Sub-Optimal Pipelining of Broadcast Signals

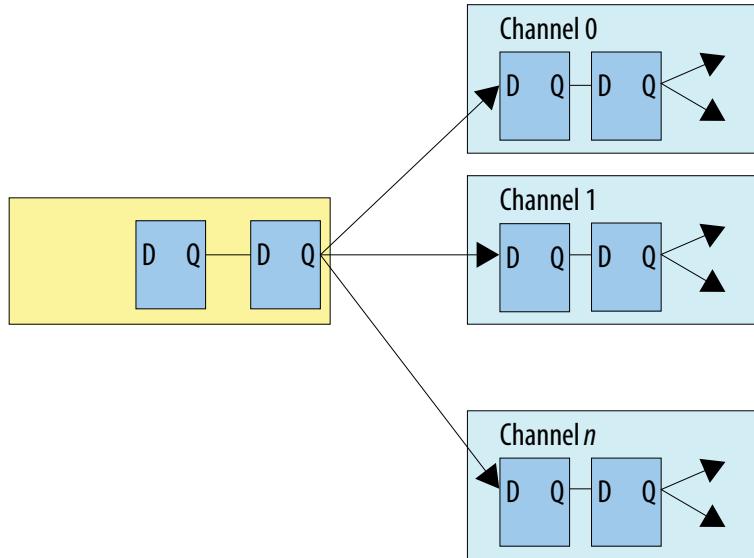
The yellow box highlights registers that you insert in a module to help with timing. The block broadcasts the output to several transceiver channels. These extra registers may not improve timing sufficiently because the final register stage fans out to destinations over a wide area of the device.



A better approach to pipelining is to duplicate the last pipeline register, and then place a copy of the register in the destination module (the transceiver channels in this example). This method results in better placement and timing. The improvement occurs because each channel's pipeline register placement helps cover the distance between the last register stage in the yellow module, and the registers in the transceivers, as needed.

In addition to duplicating the last pipeline register, apply the `dont_merge` synthesis attribute to avoid merging of the duplicate registers during synthesis, which eliminates any benefit. The Compiler automatically adds pipeline stages and moves registers into Hyper-Registers, whenever possible. You can also use manual pipelining to drive even better placement result.

Figure 6. Optimal Pipelining of Broadcast Signals



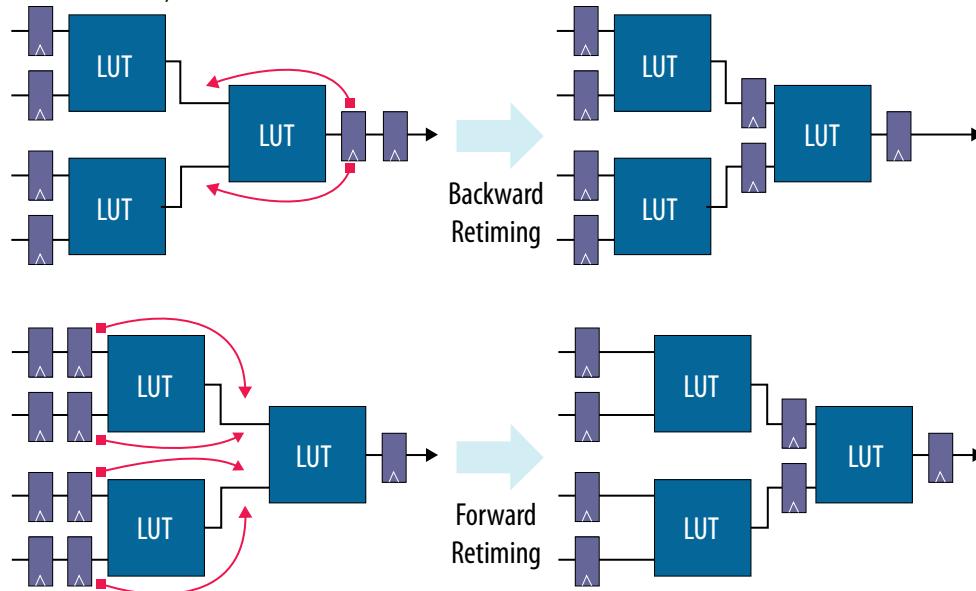
2.2 Hyper-Retiming (Facilitate Register Movement)

The Retime stage of the Fitter can balance register chains by retiming (moving) ALM registers into Hyper-Registers in the routing fabric. The Retime stage also performs sequential optimization by moving registers backward and forward across combinational logic. By balancing the propagation delays between each stage in a series of registers, retiming shortens the critical paths, reduces the clock period, and increases the frequency of operation.

The Retime stage then runs during Fitter processing to move the registers into ideal Hyper-Register locations. This Hyper-Retiming process requires minimal effort, while resulting in 1.1 – 1.3x performance gain for Intel Stratix 10 devices versus previous devices.

Figure 7. Moving Registers across LUTs

Registers on the left before retiming, with worst case delay of two LUTs. Registers on the right after retiming, with worst case delay of one LUT.



When the Compiler cannot retime a register, this is a retiming restriction. Such restrictions limit the design's f_{MAX} . Minimize retiming restrictions in performance-critical parts of your designs to achieve the highest performance.

There are a variety of design conditions that limit performance. Limitations can relate to hardware characteristics, software behavior, or the design characteristics. Use the following design techniques to facilitate register retiming and avoid retiming restrictions:

- Avoid asynchronous resets, except where necessary. Refer to the *Reset Strategies* section.
- Avoid synchronous clears. Synchronous clears are usually broadcast signals that are not conducive to retiming.
- Use wildcards or names in timing constraints and exceptions. Refer to the *Timing Constraint Considerations* section.
- Avoid single cycle (stop/start) flow control. Examples are clock enables and FIFO full/empty signals. Consider using valid signals and almost full/empty, respectively.
- Avoid *preserve* register attributes. Refer to the *Retiming Restrictions and Workarounds* section.
- For information about adding pipeline registers, refer to the *Hyper-Pipelining (Add Pipeline Registers)* section.
- For information about addressing loops and other RTL restrictions to retiming, refer to the *Hyper-Optimization (Optimize RTL)* section.

The following sections provide design techniques to facilitate register movement in specific design circumstances.



Related Links

- [Reset Strategies](#) on page 13
- [Timing Constraint Considerations](#) on page 19
- [Hyper-Pipelining \(Add Pipeline Registers\)](#) on page 26
- [Retiming Restrictions and Workarounds](#) on page 74

2.2.1 Reset Strategies

This section recommends techniques to achieve maximum performance with resets. For the best performance, avoid resets (asynchronous and synchronous), except when necessary. Because Hyper-Registers do not have asynchronous clears, you cannot move any register with an asynchronous clear into a Hyper-Register location.

Using a synchronous clear instead of an asynchronous clear allows retiming the register. Refer to the *Synchronous Resets and Limitations* section for more detailed information about retiming behavior for registers with synchronous clears. Some registers in your design require synchronous or asynchronous clears, but you must minimize the number for best performance.

Related Links

- [Synchronous Resets and Limitations](#) on page 111

2.2.1.1 Removing Asynchronous Clears

Remove asynchronous clears if a circuit naturally resets when the reset is held long enough to reach a steady-state equivalent of a full reset.

Figure 8. Verilog HDL and VHDL Asynchronous Clear Examples
Verilog HDL

```

  always @(posedge clk, aclr)
    if (aclr) begin
      reset_synch <= 1'b0;
      aclr_int <= 1'b0;
    end
    else begin
      reset_synch <= 1'b1;
      aclr_int <= reset_synch;
    end

  always @(posedge clk, aclr_int)
    if (aclr_int) begin
      a <= 1'b0;
      b <= 1'b0;
      c <= 1'b0;
      d <= 1'b0;
      out <= 1'b0;
    end
    else begin
      a <= in;
      b <= a;
      c <= b;
      d <= c;
      out <= d;
    end
  
```

VHDL

```

  PROCESS(clk, aclr) BEGIN
    IF (aclr = '1') THEN
      reset_synch <= '0';
      aclr_int <= '0';
    ELSIF rising_edge(clk) THEN
      reset_synch <= '1';
      aclr_int <= reset_synch
    END IF;
  END PROCESS;

  PROCESS(clk, aclr_int) BEGIN
    IF (aclr_int = '1') THEN
      a <= '0';
      b <= '0';
      c <= '0';
      d <= '0';
      output <= '0';
    ELSIF rising_edge(clk) THEN
      a <= input;
      b <= a;
      c <= b;
      d <= c;
      output <= d;
    END IF;
  END PROCESS;
  
```

Asynchronous clear clears all registers in the pipeline.
 They cannot be placed in Hyper-Registers.

Figure 9. Circuit Using Full Asynchronous Reset

The following figure shows the same logic as the *Asynchronous Clear Examples* in schematic form.

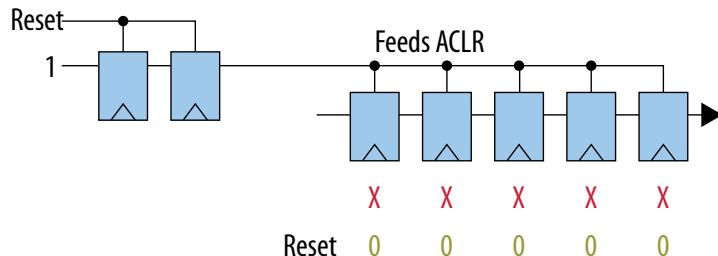




Figure 10. Partial Asynchronous Reset

After a partial reset, if the modified circuit settles to the same steady state as the original circuit, then the modification is functionally equivalent. The following figure shows removal of asynchronous clears from the middle of the circuit.

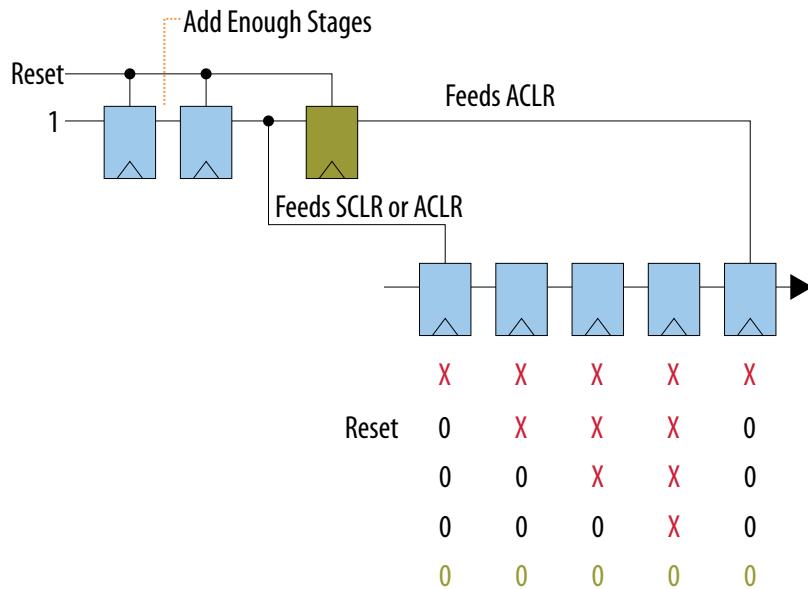


Figure 11. Circuit with an Inverter in the Register Chain

Circuits that include inverting logic typically require additional synchronous clears to remain in the pipeline, as illustrated in the following figure.

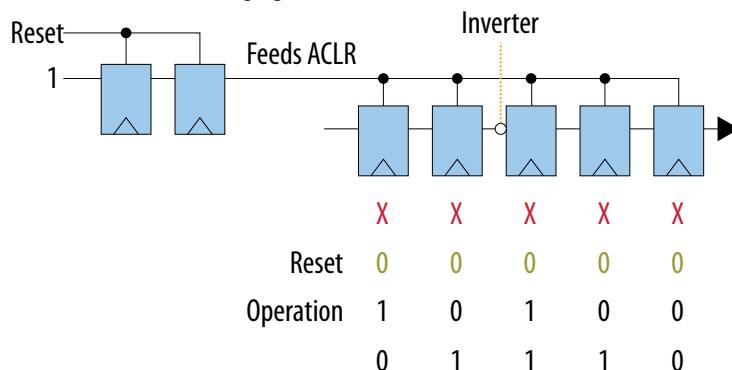
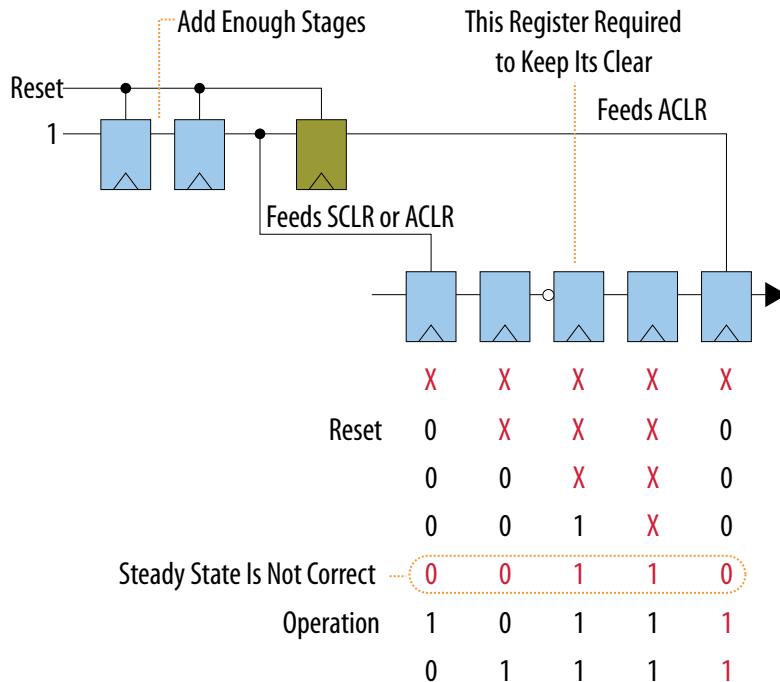
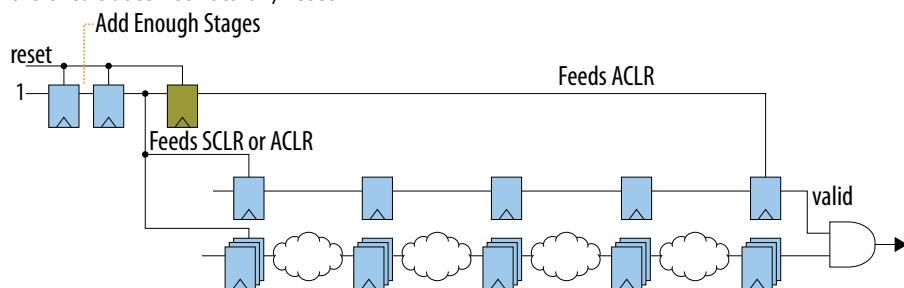


Figure 12. Circuit with an Inverter in the Register Chain with Asynchronous Clear

After removing the reset and applying the clock, the register outputs do not settle to the reset state. The inverting register cannot have its asynchronous clear removed to be equivalent to the above circuit after settling out of reset.


Figure 13. Validating the Output to Synchronize with Reset

To avoid non-naturally resetting logic caused by inverting functions, validate the output to synchronize with reset removal. Then, if the validating pipeline can enable the output when the computational pipeline is actually valid, the behavior is equivalent with reset removal. This process is suitable even if the computation portion of the circuit does not naturally reset.



The following figures show Verilog HDL and VHDL examples of the [Figure 10](#) on page 15. Apply this example to your design to remove unnecessary asynchronous resets



Figure 14. Verilog HDL Example Using Minimal or No Asynchronous Clears

The following figures show Verilog HDL and VHDL examples of the Figure 10 on page 15. Apply this example to your design to remove unnecessary asynchronous resets

Verilog HDL

```

always @(posedge clk, posedge aclr)
  if (aclr) begin
    reset_synch_1 <= 1'b0;
    reset_synch_2 <= 1'b0;
    aclr_int <= 1'b0;
  end
  else begin
    reset_synch_1 <= 1'b1;
    reset_synch_2 <= reset_synch_1;
    aclr_int <= reset_synch_2;
  end

  always @(posedge clk, posedge aclr_int)
    if (aclr_int)
      out <= 1'b0;
    else
      out <= d;

  always @(posedge clk)
    if (reset_synch_2)
      a <= 1'b0;
    else
      a <= in;

  always @(posedge clk) begin
    b <= a;
    c <= b;
    d <= c;
  end

```

Figure 15. VHDL Example Using Minimal or No Asynchronous Clears

```

PROCESS (clk, aclr) BEGIN
  IF (aclr = '1') THEN
    reset_synch_1 <= '0';
    reset_synch_2 <= '0';
    aclr_int <= '0';
  ELSIF rising_edge(clk) THEN
    reset_synch_1 <= '1';
    reset_synch_2 <= reset_synch_1;
    aclr_int <= reset_synch_2;
  END IF;
END PROCESS;

PROCESS (clk, aclr_int) BEGIN
  IF (aclr_int = '1') THEN
    output <= '0';
  ELSIF rising_edge(clk) THEN
    output <= d;
  END IF;
END PROCESS;

```

```

PROCESS (clk) BEGIN
  IF rising_edge(clk) THEN
    IF (reset_synch_2 = '1') THEN
      a <= '0';
    ELSE
      a <= input;
    END IF;
  END IF;
END PROCESS;

PROCESS (clk) BEGIN
  IF rising_edge(clk) THEN
    b <= a;
    c <= b;
    d <= c;
  END IF;
END PROCESS;

```

Naturally Resetting Registers

2.2.1.2 Synchronous Clears on Global Clock Trees

Using a global clock tree to distribute a synchronous clear may limit retiming performance improvements. Global clock trees do not have Hyper-Registers. As such, there is less flexibility to retime registers that fan out through a global clock tree compared to the routing fabric.

2.2.1.3 Synchronous Resets on I/O Ports

The Compiler does not retime registers driving an output port, or registers that an input port drives. If such an I/O register has a synchronous clear, you cannot retime the register. This restriction is not typical of practical designs that contain logic driving resets. However, this issue may arise in benchmarking a smaller piece of logic, where the reset originates from an I/O port. In this case, you cannot retime any of the registers that the reset drives. Adding some registers to the synchronous reset path corrects this condition.

2.2.1.4 Duplicate and Pipeline Synchronous Resets

If a synchronous clear signal causes timing issues, duplicating the synchronous clear signal between the source and destination registers can resolve the timing issue. The registers pushed forward need not contend for Hyper-Register locations with registers being pushed back. For small logic blocks of a design, this method is a valid strategy to improve timing.

2.2 Clock Enable Strategies

High fan-out clock enable signals can limit the performance achievable by retiming. This section provides recommendations for the appropriate use of clock enables.

2.2.2.1 Localized Clock Enable

The localized clock enable has a small fan-out. The localized clock enable often occurs in a clocked process or an always block. In these cases, the signal's behavior is undefined under a particular branch of a conditional case or if statement. As a result, the signal retains its previous value, which is a clock enable.

To check whether a design has clock enables, view the **Fitter Report > Plan Stage > Control Signals** Compilation report and check the **Usage** column. Because the localized clock enable has a small fan-out, retiming is easy and usually does not cause any timing issues.

2.2.2.2 High Fan-Out Clock Enable

Avoid high fan-out signals whenever possible. The high fan-out clock enable feeds a large amount of logic. The amount of logic is so large that the registers that you retime are pushing or pulling registers up and down the clock enable path for their specific needs. This pushing and pulling can result in conflicts along the clock enable line. This condition is similar to the aggressive retiming in the *Synchronous Resets Summary* section. Some of the methods discussed in that section, like duplicating the enable logic, are also beneficial in resolving conflicts along the clock enable line.



You typically use these high fan-out signals to disable a large amount of logic from running. These signals might occur when a FIFO's full flag goes high. You can often design around these signals. For example, you can design the FIFO to specify almost full a few clock cycles earlier, and allow the clock enable a few clock cycles to propagate back to the logic that disables. You can retime these extra registers into the logic if necessary.

Related Links

[Synchronous Resets Summary](#) on page 114

2.2.2.3 Clock Enable with Timing Exceptions

The Compiler cannot retime registers that are endpoints of multicycle or false path timing exceptions. Clock enables are sometimes used to create a sub-domain that runs at half or quarter the rate of the main clock. Sometimes these clock enables control a single path with logic that changes every other cycle. Because you typically use timing exceptions to relax timing, this case is less of an issue. If a clock enable validates a long and slow data path, and the path still has trouble meeting timing, add a register stage to the data path. Remove the multicycle timing constraint on the path. The Hyper-Aware CAD flow allows the Retimer to retime the path to improve timing.

2.2.3 Synthesis Attributes

Your design may include registers with synthesis attributes such as `preserve` or `dont_touch`. The Compiler does not retime registers with `preserve` or `dont_touch` attributes, because the Compiler respects the directive to prevent optimization. Consider whether you can remove the directives and allow the Compiler to retime affected registers. To preserve a register for debugging observability, keep the `preserve` attribute. If you preserve a register to manage register duplication, use `dont_merge` instead.

2.2.4 Timing Constraint Considerations

The use of timing constraints impacts compilation results. Timing constraints influence how the Fitter places logic. This section describes timing constraint techniques that maximize design performance.

2.2.4.1 Optimize Multicycle Paths

The Compiler does not retime registers that are the endpoints of an `.sdc` timing constraint, including multicycle or false path timing constraints. Therefore, assign any timing constraints or exceptions as specifically as possible to avoid retiming restrictions.

Using actual register stages, rather than a multicycle constraint, allows the Compiler the most flexibility to improve performance. For example, rather than specifying a multicycle exception of 3 for combinational logic, remove the multicycle exception and insert two extra register stages before or after the combinational logic. This change allows the Compiler to balance the extra register stages optimally through the logic.

2.2.4.2 Overconstraints

Overconstraints direct the Fitter to spend more time optimizing specific parts of a design. Overconstraints are appropriate in some situations to improve performance. However, because legacy overconstraint methods restrict retiming optimization, Intel Stratix 10 devices support a new `is_post_route` function that allows retiming. The `is_post_route` function allows the Fitter to adjust slack delays for timing optimization.

Example 1. Intel Stratix 10 Overconstraints Syntax (Allows Hyper-Retiming)

```
if { ! [is_post_route] } {
    # Put overconstraints here
}
```

Example 2. Legacy Overconstraints Example (Prevents Hyper-Retiming)

```
### Over Constraint ###
# if ${::quartus(nameofexecutable) == "quartus_fit"} {
#     set_min_delay 0.050 -from [get_clocks {CPRI|PHY|TRX*|*|rx_pma_clk}] -to
[get_clocks {CPRI|PHY|TRX*|*|rx_clkout}]
# }
```

2.2.5 Clock Synchronization Strategies

Use a simple synchronization strategy to reach maximum speeds in the Intel Stratix 10 architecture. Adding latency on paths with simple synchronizer crossings is straightforward. However, adding latency on other crossings is more complex.

Figure 16. Simple Clock Domain Crossing

This example shows a simple synchronization scheme with a path from one register of the first domain (blue), directly to a register of the next domain (red).

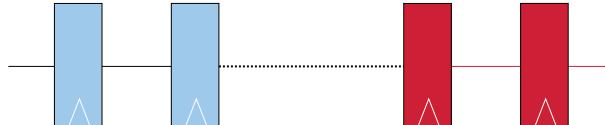
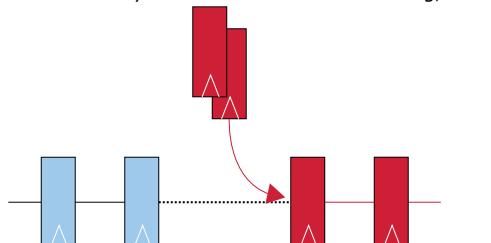


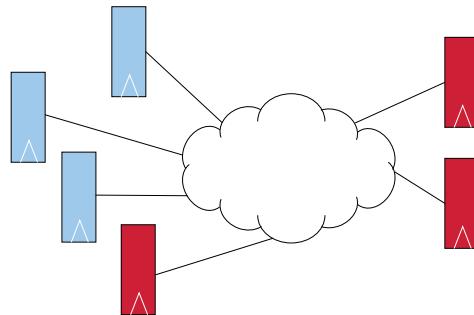
Figure 17. Simple Clock Domain Crossing After Adding Latency

To add latency in the red domain for retiming, add the registers as shown.



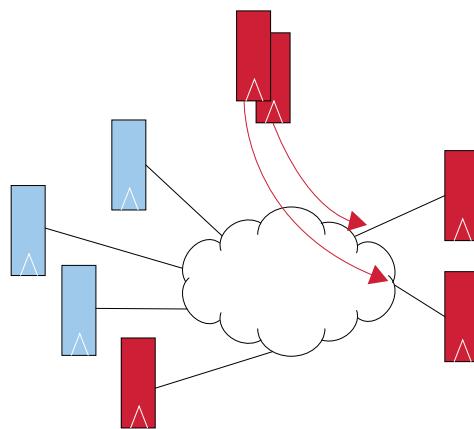
The following figure shows a domain crossing structure that is not optimum in Intel Stratix 10 designs, but exists in designs that target other device families. The design contains some combinational logic between the blue clock domain and the red clock domain. The design not properly synchronize the logic and you cannot add registers flexibly. The blue clock domain drives the combinational logic and the logic contains paths that the red domain launches.

Figure 18. Clock Domain Crossing at Multiple Locations



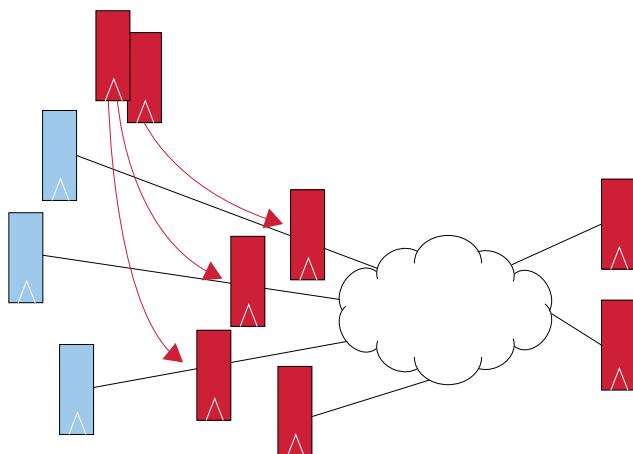
Add latency at the boundary of the red clock domain, but do not add registers on a red to red domain path. Otherwise, the paths become unbalanced, potentially changing design functionality. Although possible, adding latency in this scenario is risky. Thoroughly analyze the various paths before adding latency.

Figure 19. Clock Domain Crossing at Multiple Locations After Adding Latency



For Intel Stratix 10 designs, synchronize the clock crossing paths before entering combinational logic. Adding latency is then more simple when you compare with the previous example. Blue domain registers synchronize to the red domain before entering the combinational logic. This method allows safe addition of pipeline registers in front of synchronizing registers, without contacting a red to red path inadvertently. Implement this synchronization method for the highest Intel Stratix 10 architecture performance.

Figure 20. Clock Domain Synchronization Improvement



2.2.6 Metastability Synchronizers

The Compiler detects registers that are part of a synchronizer chain. The Compiler cannot retime the registers in a synchronizer chain. To allow retiming of the registers in a synchronizer chain, add more pipeline registers at clock domain boundaries.

Metastability synchronizer chain length for Intel Stratix 10 devices is three. The Critical Chain report now marks the registers that metastability requires with REG (Metastability required).

2.2.7 Initial Power-Up Conditions

The initial condition of your design at power-up represents the state of the design at clock cycle 0. The initial condition is highly dependent on the underlying device technology. Once the design leaves the initial state, there is no automated method to return to that state. In other words, the initial condition state is a transitional rather than functional state. In addition, other design components can affect the validity of the initial state. For example, a PLL that is not yet locked upon power-up can impact the initial state.

Therefore, do not rely on initial conditions when designing for Intel Stratix 10 FPGAs. Rather, use a single reset signal to place the design in a known, functional state until all the interfaces have powered up, locked, and trained.

2.2.7.1 Specifying Initial Conditions

You can specify initial power-up conditions by inference in your RTL code. Intel Quartus Prime synthesis automatically converts default values for registered signals into Power-Up Level constraints. Alternatively, specify the Power-Up Level constraints manually.

Example 3. Initial Power-Up Conditions Syntax (Verilog HDL)

```

reg q = 1'b1; //q has a default value of '1'
always @ (posedge clk)
begin
    q <= d;
end
  
```

Example 4. Initial Power-Up Conditions Syntax (VHDL)

```
SIGNAL q : STD_LOGIC := '1'; -- q has a default value of '1'  
PROCESS (clk, reset)  
BEGIN  
    IF (rising_edge(clk)) THEN  
        q <= d;  
    END IF;  
END PROCESS;
```

2.2.7.2 Initial Conditions and Retiming

The initial power-up conditions can limit the Compiler's ability to:

- Perform logic optimization during synthesis
- Move registers into Hyper-Registers during retiming

The following examples show how setting initial conditions to a known state ensures that circuits are functionality equivalent after retiming.

Figure 21. Circuit Before Retiming

This sample circuit shows register F1 at power-up can have either state '0' or state '1'. Assuming the clouds of logic are purely combinational, there are two possible states in the circuit C1 ($F1='0'$ or $F1='1'$).

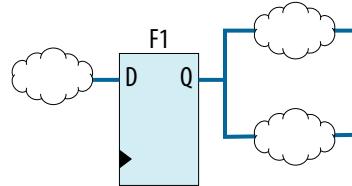
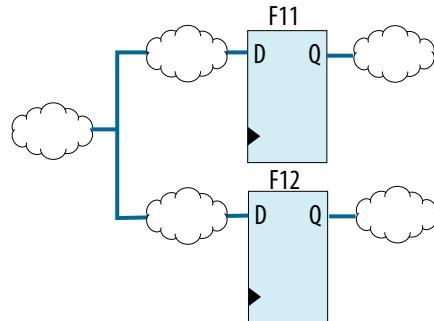


Figure 22. Circuit After Retiming Forward

If the Retimer pushes register F1 forward, the Retimer must duplicate the register in each of the branches that F1 drives.



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 design functionality.

Table 3. Possible Power-Up States After Retiming

F11 States	F12 States
0	0
0	1
1	0
1	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 your design's reset sequence. To ensure the design's functional equivalence after retiming, apply an extra clock cycle after power-up. This extra clock cycle ensures that the states of F11 and F12 are always identical. This technique results in only two possible states for the registers, 0/0 or 1/1, assuming the combinational logic is non-inverting on both paths.

Retiming Backward

Retiming registers backward is always a safe operation with a c-cycle value of 0. In this scenario, the Compiler merges F11 and F12 together. If you do not specify initial conditions for F11 and F12, the Compiler always permits merging. If you specify initial conditions, the Compiler accounts for the initial state of F11 and F12. In this case, the retiming transformation only occurs if the transformation preserves the initial states.

If the Compiler transformation cannot preserve the initial states of F11 and F12, the Compiler does not allow the retiming operation. To avoid changing circuit functionality during retiming, apply an extra clock cycle after power-up to ensure the content of F11 and F12 are always identical.

2.2.7.3 Retiming Reset Sequences

Under certain conditions, the Retime stage performs transformation of registers with a c-cycle value greater than zero. This ability can help improve the maximum frequency of the design. However, register retiming with a c-cycle equivalence value greater than zero requires extra precaution to ensure functional equivalence after retiming. To retain functional equivalence, reuse existing reset sequences, and add the appropriate number of clock, as the following sections describe:

Reset Retiming Behavior

The Compiler has the following behavior when retiming resets:

- Backward retiming with reset is safe and occurs, taking into consideration any initial conditions.
- Forward retiming with reset always preserves the initial conditions.
- Register retiming assumes that registers with no initial conditions power up to 0 for retiming purpose. Retiming preserves this initial condition.



Ignoring Initial Conditions

Retime more registers by ignoring initial conditions on registers. Specify the ALLOW_POWER_UP_DONT_CARE option in the .qsf to ignore initial reset conditions and continue with retiming:

```
set_global_assignment -name ALLOW_POWER_UP_DONT_CARE ON
```

When using ALLOW_POWER_UP_DONT_CARE, ensure that the registers your reset sequence covers do not have initial conditions in RTL code.

Modifying the Reset Sequence

Follow these recommendations to maximize operating frequency of resets during retiming:

- Remove `sclr` signals from all registers that reset naturally. This removal allows the registers to move freely in the logic during retiming.
- Assign the power-up state of the registers that the reset sequence covers as `don't care`. Ignore initial conditions on those registers.
- Set the ALLOW_POWER_UP_DONT_CARE global assignment to ON. This setting maximizes register movement.
- Compute and add to the reset synchronizer the relevant amount of extra clock cycles due to c-cycle equivalence.

Adding Clock Cycles to Reset

The Compiler reports the number of clock cycles to add to your reset sequence in the **Fitter > Retime Stage > Reset Sequence Requirement** report. The report lists the number of cycles to add on a clock domain basis.

Figure 23. Reset Sequence Requirement Report

Clock Name	Number of additional cycles
1 clk	10

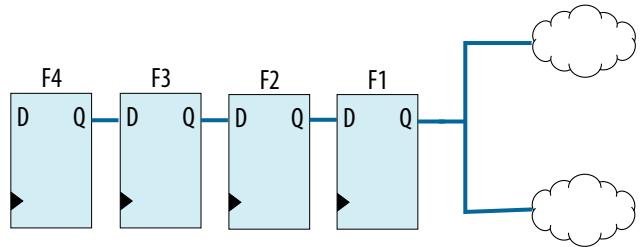
Note: Due to retiming optimizations, a clock domain may require a longer reset sequence to ensure correct functionality. The table above indicates the minimum number of additional reset sequence cycles needed for each clock domain.

Register duplication into multiple branches has a c-cycle of 1. Regardless of the number of duplicate registers, the register is always one connection away from its original source. After one clock cycle, all the branches have the same value again.

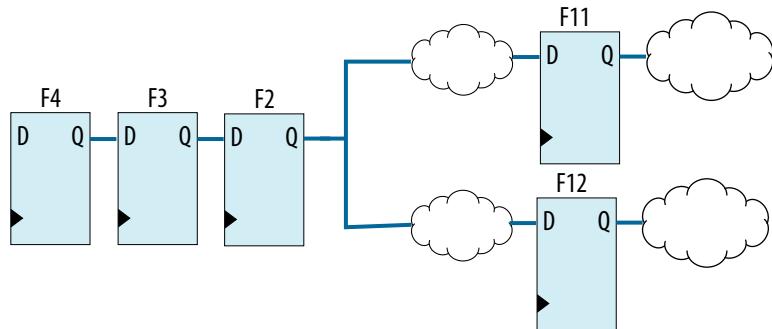
The following examples show how adding clock cycles to the reset sequence ensures the functional equivalence of the design after retiming.

Figure 24. Pipelining and Register Duplication

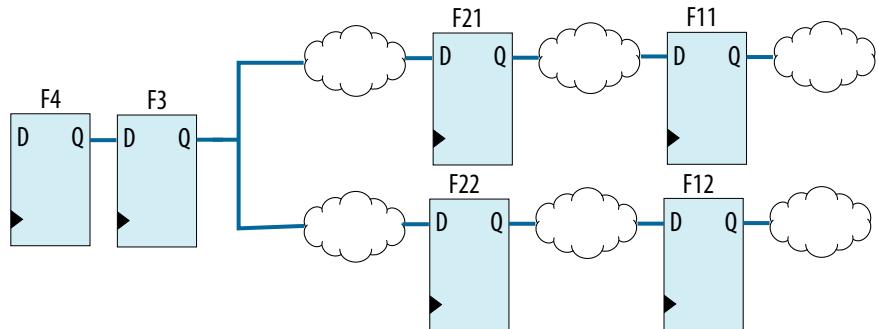
This example shows pipelining of registers with potential for forward retiming. The c-cycle value equals 0.


Figure 25. Impact of One Register Move

This example shows a pipelining of registers after forward retiming of one register. Because the c-cycle value equals 1, the reset sequence for this circuit requires one additional clock cycle for functional equivalence after reset.


Figure 26. Impact of Two Register Moves

This example shows a pipelining of registers after forward retiming of two registers. Because the c-cycle value equals 2, the reset sequence for this circuit requires two additional clock cycles for functional equivalence after reset.



Each time a register from the pipeline moves into the logic, the register duplicates and the C-cycle value of the design increases by one.

2.3 Hyper-Pipelining (Add Pipeline Registers)

Hyper-Pipelining is a design process that eliminates long routing delays by adding additional pipeline stages in the interconnect between the ALMs. This technique allows the design to run at a faster clock frequency. First run Fast-Forward compilation to determine the best location and performance you can expect from adding pipeline stages. This process requires minimal effort, resulting in 1.3 – 1.6x performance gain for Intel Stratix 10 devices, versus previous generation high-performance FPGAs.

Adding registers in your RTL is much easier if you plan ahead to accommodate additional latency in your design. At the most basic level, planning for additional latency means using parameterizable pipelines at the inputs and outputs of the clock domains in your design. Refer to the *Appendix: Pipelining Examples* for pre-written parameterizable pipeline modules in Verilog HDL, VHDL, and SystemVerilog.

Changing latency is more complex than simply adding pipeline stages. Changing latency can require reworking control logic, and other parts of the design or system software, to work properly with data arriving later. Making such changes is often difficult in existing RTL, but is typically easier in new parts of a design. Rather than hard-coding block latencies into control logic, implement some latencies as parameters. In some types of systems, a “valid data” flag is present to pipeline stages in a processing pipeline to trigger various computations, instead of relying on a high-level fixed concept of when data is valid.

Additional latency may also require changes to testbenches. When you create testbenches, use the same techniques you use to create latency-insensitive designs. Do not rely on a result becoming available in a predefined number of clock cycles, but consider checking a “valid data” or “valid result” flag.

Latency-insensitive design is not appropriate for every part of a system. Interface protocols that specify a number of clock cycles for data to become ready or valid must conform to those requirements and may not accommodate changes in latency.

After you modify the RTL and place the appropriate number of pipeline stages at the boundaries of each clock domain, the Retiming stage automatically places the registers within the clock domain at the optimal locations to maximize the performance. The combination of Hyper-Retiming and Fast-Forward compilation helps to automate the process in comparison with conventional pipelining.

Related Links

- [Appendix A: Parameterizable Pipeline Modules](#) on page 109
- [Precomputation](#) on page 39

2.3.1 Conventional Versus Hyper-Pipelining

Hyper-Pipelining simplifies this process of conventional pipelining. Conventional pipelining includes the following design modifications:

- Add two registers between logic clouds.
- Modify HDL to insert a third register (or pipeline stage) into the design’s logic cloud, which is Logic Cloud 2. This register insertion effectively creates Logic Cloud 2a and Logic Cloud 2b in the HDL

Figure 27. Conventional Pipelining User Modifications

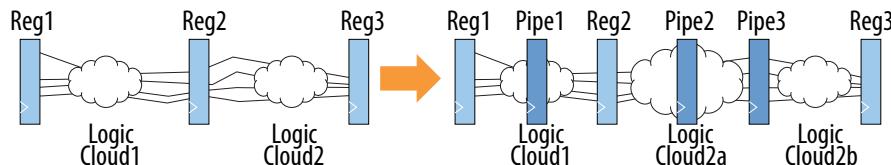
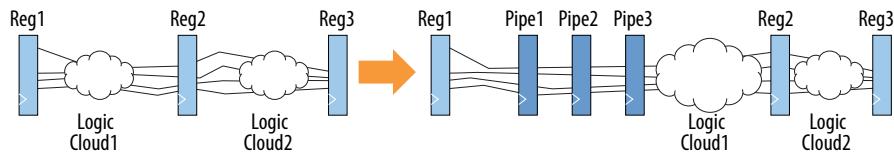
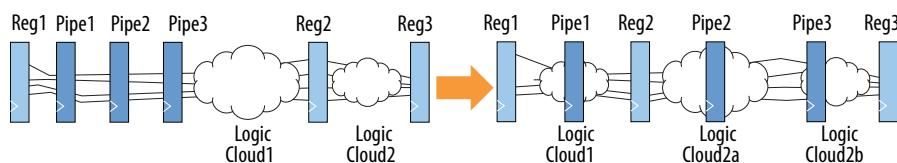


Figure 28. Hyper-Pipelining User Modifications

Hyper-Pipelining simplifies the process of adding registers. Add the registers—Pipe 1, Pipe 2, and Pipe 3—in aggregate at one location in the design RTL. The Compiler retimes the registers throughout the circuit to find the optimal placement along the path. This optimization reduces path delay and maximizes the design's operating frequency.


Figure 29. Hyper-Pipelining and Hyper-Retiming Implementation

The following figure shows implementation of additional registers after the retiming stage completes optimization.



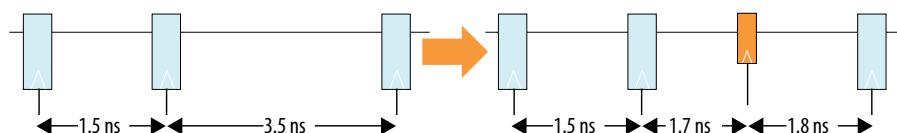
The resulting implementation in the Hyper-Pipelining flow differs from the conventional pipelining flow by the location of the Pipe 3 register. Because the Compiler is aware of the current circuit implementation, including routing, the Compiler can more effectively locate the aggregate registers to meet the design's maximum operating frequency. Hyper-Pipelining requires significantly less effort than conventional pipelining techniques because you can place registers at a convenient location in a data path. The Compiler optimizes the register placements automatically.

2.3.2 Pipelining and Latency

Adding pipeline registers within a path increases the number of clock cycles necessary for a signal value to propagate along the path. Increasing the clock frequency can offset the increased latency.

Figure 30. Hyper-Pipeline Reduced Latency

This example shows a previous generation Intel FPGA, with a 275 MHz f_{MAX} requirement. The path on the left achieves 286 MHz because the 3.5 ns delay limits the path. Data requires three cycles to propagate through the register pipeline. Three cycles at 275 MHz calculates to 10.909 ns requirement to propagate through the pipeline.



If re-targeting an Intel Stratix 10 device doubles the f_{MAX} requirement to 550 MHz, the path on the right side of the figure shows how an additional pipeline stage retimes. The path now achieves 555 MHz, due to the limits of the 1.8 ns delay. The data requires four cycles to propagate through the register pipeline. Four cycles at 550 MHz equals 7.273 ns to propagate through the pipeline.

To maintain the time to propagate through the pipeline with four stages compared to three, meet the 10.909 ns delay of the first version by increasing the f_{MAX} of the second version to 367 MHz. This technique results in a 33% increase from 275 MHz.



2.3.3 Use Registers Instead of Multicycle Exceptions

Often designs contain modules with complex combinational logic (such as CRCs and other arithmetic functions) that require multiple clock cycles to process. You constrain these modules with multicycle exceptions that relax the timing requirements through the block. You can use these modules and constraints in designs targeting Intel Stratix 10 devices. Refer to the *Design Considerations for Multicycle Paths* section for more information.

Alternatively, you can insert a number of register stages in one convenient place in a module, and the Compiler balances them automatically for you. For example, if you have a CRC function to pipeline, you do not need to identify the optimal decomposition and intermediate terms to register. Add the registers at its input or output, and the Compiler balances them.

Related Links

- [Optimize Multicycle Paths](#) on page 19
- [Appendix A: Parameterizable Pipeline Modules](#) on page 109

2.4 Hyper-Optimization (Optimize RTL)

After you accelerate data paths through Hyper-Retiming, Fast Forward compilation, and Hyper-Pipelining, the design may still have limits of control logic, such as long feedback loops and state machines.

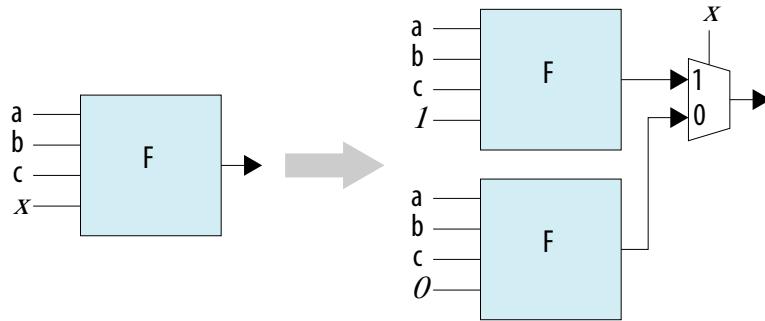
To overcome such limits, use functionally equivalent feed-forward or pre-compute paths, rather than long combinatorial feedback paths. The following sections describe specific Hyper-Optimization for various design structures. This process can result in 2x performance gain for Intel Stratix 10 devices, compared to previous generation high-performance FPGAs.

2.4.1 General Optimization Techniques

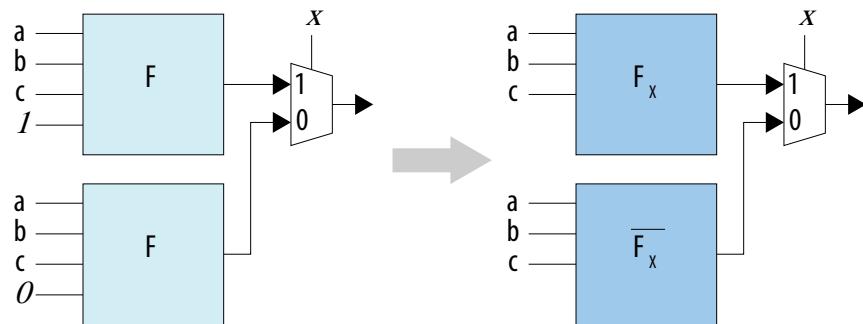
Use the following general RTL techniques to optimize your design for the Intel Hyperflex FPGA architecture.

2.4.1.1 Shannon's Decomposition

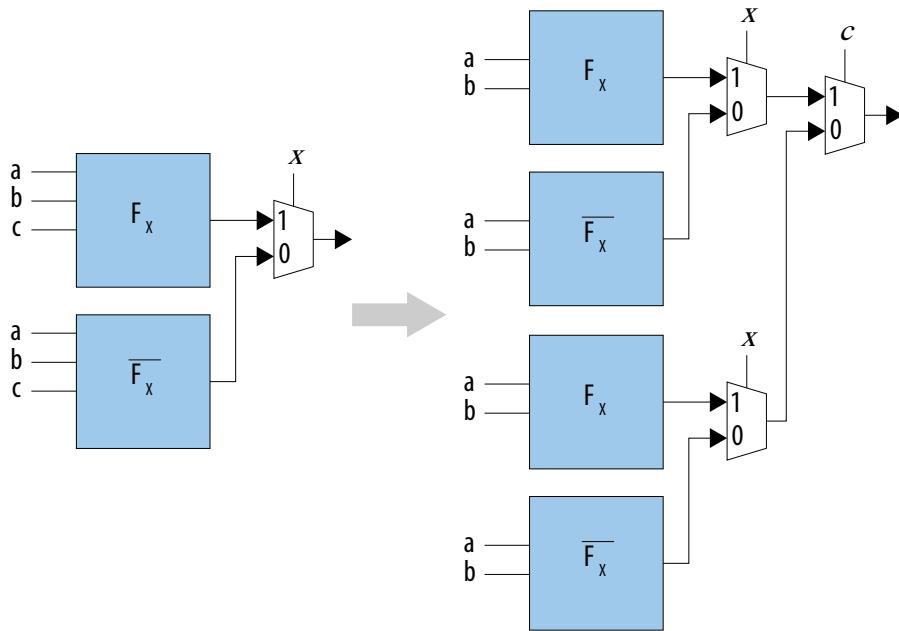
Shannon's decomposition plays a role in Hyper-Optimization. Shannon's decomposition, or Shannon's expansion, is a way of factoring a Boolean function. You can express a function as $F = x.F_x + x'F_x'$ where $x.F_x$ and $x'F_x'$ are the positive and negative co-factors of the function F with respect to x . You can factor a function with four inputs as, $(a, b, c, x) = x.(a, b, c, 1) + x'.F(a, b, c, 0)$, as shown in the following diagram. In Hyper-Optimization, Shannon's decomposition pushes the x signal to the head of the cone of input logic, making the x signal the fastest path through the cone of logic. The x signal becomes the fastest path at the expense of all other signals. Using Shannon's decomposition also doubles the area cost of the original function.

Figure 31. Shannon's Decomposition

Figure 32. Shannon's Decomposition Logic Reduction

Logic synthesis can take advantage of the constant-driven inputs and slightly reduce the cofactors, as shown in the following diagram.


Figure 33. Repeated Shannon's Decomposition

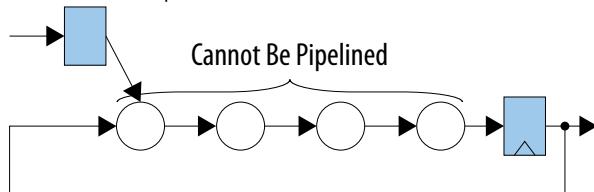
The following diagram shows how you can repeatedly use Shannon's decomposition to decompose functions with more than one critical input signal, thus increasing the area cost.



Shannon's decomposition can be an effective optimization technique for loops. When you perform Shannon's decomposition on logic in a loop, the logic in the loop moves outside the loop. The Compiler can now pipeline the logic moved outside the loop.

Figure 34. Loop Example before Shannon's Decomposition

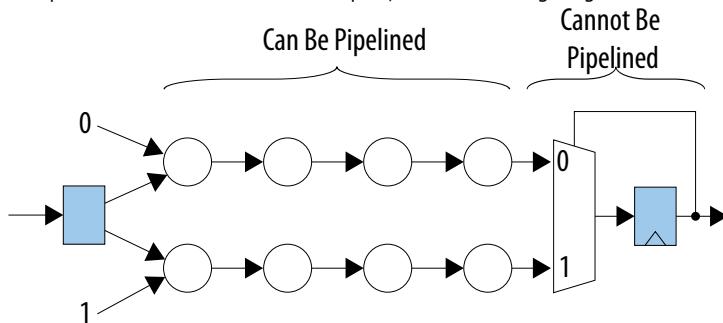
This diagram shows a loop that contains a single register, four levels of combinational logic, and an additional input. Adding registers in the loop changes the functionality, but you can move the combinational logic outside the loop by performing Shannon's decomposition.



The output of the register in the loop is 0 or 1. You can duplicate the combinational logic that feeds the register in the loop, tying one copy's input to 0, and the other copy's input to 1.

Figure 35. Loop Example after Shannon's Decomposition

The register in the loop then selects one of the two copies, as the following diagram shows.



Performing Shannon's decomposition on the logic in the loop reduces the amount of logic in the loop. The Compiler can now perform register retiming or Hyper-Pipelining on the logic you remove from the loop, thereby increasing the circuit performance.

2.4.1.1.1 Shannon's Decomposition Example

The sample circuit adds or subtracts an input value from the `internal_total` value based on its relationship to a target value. The core of the circuit is the `target_loop` module, shown in the following example.

Example 5. Source Code before Shannon's Decomposition

```
module target_loop (clk, sclr, data, target, running_total);
parameter WIDTH = 32;

input clk;
input sclr;
input [WIDTH-1:0] data;
input [WIDTH-1:0] target;
output [WIDTH-1:0] running_total;

reg [WIDTH-1:0] internal_total;

always @(posedge clk) begin
    if (sclr)
```

```

begin
    internal_total <= 0;
end
else begin
    internal_total <= internal_total + ((( internal_total > target) ? -
data:data)* target/4));
    end
end
assign running_total = internal_total;
end module

```

The module uses a synchronous clear, based on the recommendations to enable Hyper-Retiming.

The following figure shows the Fast Forward Compile report for the target_loop module instantiated in a register ring.

Figure 36. Fast Forward Compile Report before Shannon's Decomposition

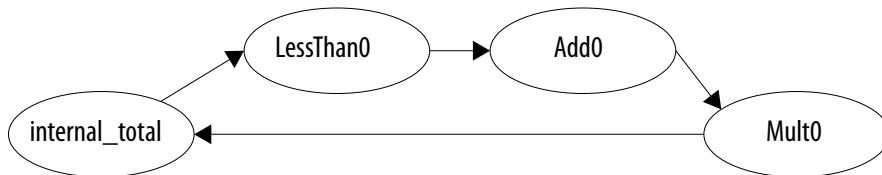
Fast Forward Summary for Clock Domain clk					
Step	Fast Forward Optimizations Applied	To Achieve Fmax	Slack	Requirement	Limiting Reason
1 Base Performance	0, including 0 pipeline stages	247.1 MHz	-3.047	0.970	Short Path/Long Path
2 Fast Forward Step #3	29, including 1 pipeline stage	248.94 MHz	-3.017	0.970	Loop
3 Hyper-Optimization	29, including 1 pipeline stage	=	=	0.970	Loop

Hyper-Retiming reports about 248 MHz by adding a pipeline stage in the Fast Forward Compile. The Limiting Reason column indicates that the critical chain is a loop.

Examining the critical chain report reveals that there is a repeated structure in the chain segments. The repeated structure is shown as an example in the *Optimizing Loops* section.

The following diagram shows a structure that implements the expression in the previous example code. The functional blocks correspond to the comparison, addition, and multiplication operations. The zero in each arithmetic block's name is part of the synthesized name in the netlist. The zero is because the blocks are the first zero-indexed instance of those operators created by synthesis.

Figure 37. Elements of a Critical Chain Sub-Loop



This expression is a candidate for Shannon's decomposition. Instead of performing only one addition with the positive or negative value of data, you can perform the following two calculations simultaneously:

- $internal_total - (data * target/4)$
- $internal_total + (data * target/4)$

You can then use the result of the comparison `internal_total > target` to select which calculation result to use. The modified version of the code that uses Shannon's decomposition to implement the `internal_total` calculation is shown in the following example.



Example 6. Source Code after Shannon's Decomposition

```
module target_loop_shannon (clk, sclr, data, target, running_total);
    parameter WIDTH = 32;

    input clk;
    input sclr;
    input [WIDTH-1:0] data;
    input [WIDTH-1:0] target;
    output [WIDTH-1:0] running_total;

    reg [WIDTH-1:0] internal_total;
    wire [WIDTH-1:0] total_minus;
    wire [WIDTH-1:0] total_plus;

    assign total_minus = internal_total - (data * (target / 4));
    assign total_plus = internal_total + (data * (target / 4));

    always @(posedge clk) begin
        if (sclr)
            begin
                internal_total <= 0;
            end
        else begin
            internal_total <= (internal_total > target) ? total_minus:total_plus;
        end
    end

    assign running_total = internal_total;
endmodule
```

As shown in the following figure, the performance almost doubles after recompiling the design with the code change.

Figure 38. Fast Forward Summary Report after Shannon's Decomposition

Fast Forward Summary for Clock Domain clk						
	Step	Fast Forward Optimizations Applied	To Achieve Fmax	Slack	Requirement	Limiting Reason
1	Base Performance	0, including 0 pipeline stages	486.85 MHz	-1.054	0.970	Insufficient Registers
2	Fast Forward Step #1	37, including 1 pipeline stage	495.79 MHz	-1.017	0.970	Short Path, Long Path
3	Hyper-Optimization	37, including 1 pipeline stage	0.970	Short Path, Long Path

2.4.1.1.2 Identifying Circuits for Shannon's Decomposition

Shannon's decomposition is a good solution for circuits in which you can rearrange many inputs to control the final select stage. Account for new logic depths when restructuring logic to use a subset of the inputs to control the select stage. Ideally, the logic depth to the select signal is similar to the logic depth to the selector inputs. Practically, there is a difference in the logic depths because of difficulty in perfectly balancing the number of inputs feeding each cloud of logic.

Shannon's decomposition may also be a good solution for a circuit with only one or two signals in the cone of logic that are truly critical, and others are static, or with clearly lower priority.

Shannon's decomposition can incur a significant area cost, especially if the function is complex. There are other optimization techniques that have a lower area cost, as described in this document.

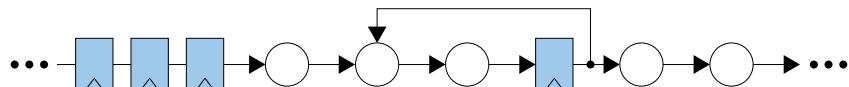
2.4.1.2 Time Domain Multiplexing

Time domain multiplexing increases circuit throughput by using multiple threads of computation. This technique is also known as C-slow retiming, or multithreading.

Time domain multiplexing replaces each register in a circuit with a set of C registers in series. Each extra copy of registers creates a new computation thread. One computation through the design requires C times as many clock cycles as the original circuit. However, the Compiler can retime the additional registers to improve the f_{MAX} by a factor of C. For example, instead of instantiating two modules running at 400 MHz, you can instantiate one module running at 800 MHz.

The following set of diagrams shows the process of C-slow retiming, beginning with an initial circuit.

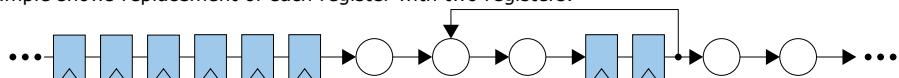
Figure 39. C-slow Retiming Starting Point



Edit the RTL design to replace every register, including registers in loops, with a set of C registers, comprising one register per independent thread of computation.

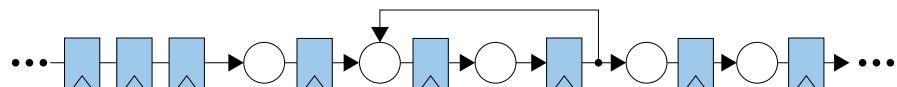
Figure 40. C-slow Retiming Intermediate Point

This example shows replacement of each register with two registers.



Compile the circuit at this point. When the Compiler optimizes the circuit, there is more flexibility to perform retiming with the additional registers.

Figure 41. C-Slow Retiming Ending Point



In addition to replacing every register with a set of registers, you must also multiplex the multiple input data streams into the block, and demultiplex the output streams out of the block. Use time domain multiplexing when a design includes multiple parallel threads, for which a loop limits each thread. The module you optimize must not be sensitive to latency.

2.4.1.3 Loop Unrolling

Loop unrolling moves logic out of the loops and into feed-forward flows. You can further optimize the logic with additional pipeline stages.

2.4.1.4 Loop Pipelining

Loops are omnipresent and an integral part of design functionality. However, loops are a limiting factor to Hyper-Retiming optimization. The Compiler cannot automatically pipeline any logic inside of a loop. Adding or removing a sequential element inside the loop potentially breaks the functionality of the design.

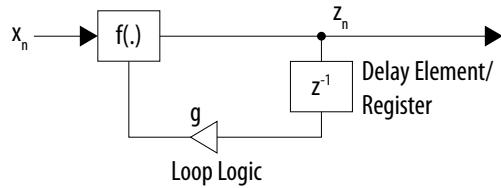
However, you can modify the loop structure to allow the Compiler to insert pipeline stages, without changing the functionality of the design, as the following topics demonstrate. Properly pipelining a loop involves the following steps:

1. Restructure loop and non-loop logic
2. Manually add pipeline stages to the loop
3. Cascade the loop logic

2.4.1.4.1 Loop Pipelining Theory

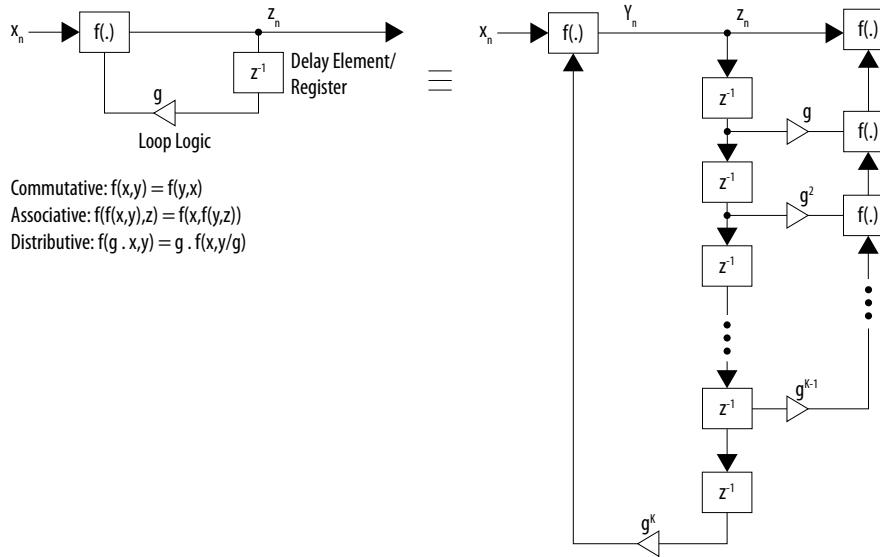
The following figure illustrates the definition of a logical loop. The result (z_n) is a function of input x_n and a delayed version of that input.

Figure 42. Simple Loop Example



If the function $f(\cdot)$ satisfies commutative, associative, and distributive properties (for example, addition, XOR, maximum), the equivalence of the following figures is mathematically provable.

Figure 43. Equivalent Loop After Modification

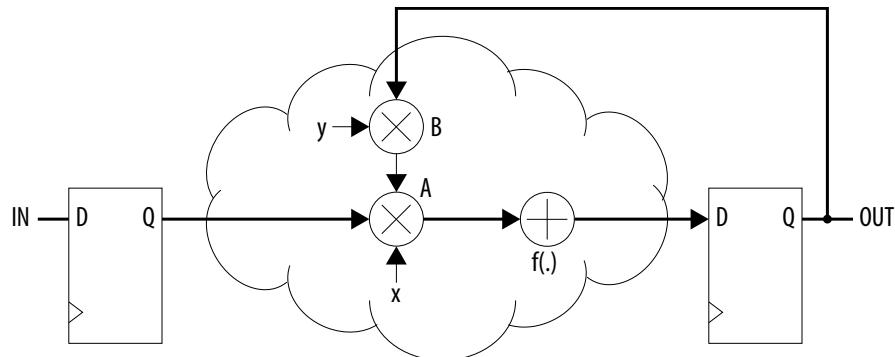


2.4.1.4.2 Loop Pipelining Demonstration

The following demonstrates proper loop pipelining to optimize an accumulator in an example design. In the original implementation, the accumulator data input in multiplies by x , adds to the previous value out , multiplied by y . This demonstration improves performance using these techniques:

1. Implement separation of forward logic
2. Retime the loop register
3. Create the feedback loop equivalence with cascade logic

Figure 44. Original Loop Structure



Example 7. Original Loop Structure Example Verilog HDL Code

```
module orig_loop_strct (rstn, clk, in, x, y, out);
    input clk, rstn, in, x, y;
    output out;
    reg out;
    reg in_reg;

    always @ ( posedge clk )
        if ( !rstn ) begin
            in_reg <= 1'b0;
        end else begin
            in_reg <= in;
        end

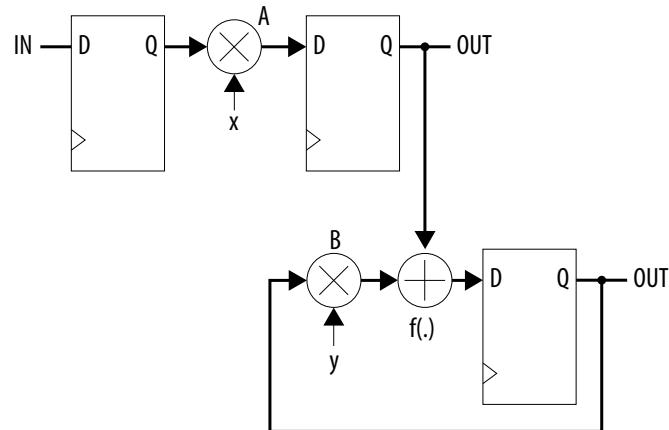
    always @ ( posedge clk )
        if ( !rstn ) begin
            out <= 1'b0;
        end else begin
            out <= y*out + x*in_reg;
        end
endmodule //orig_loop_strct
```

The first stage of optimization is to rewrite logic to remove as much logic as possible from the loop, and create a forward logic block. The goal of rewriting is to remove as much work as possible from the feedback loop. The Compiler cannot automatically optimize any logic in a feedback loop. Consider the following recommendations in removing logic from the loop:

- Evaluate as many decisions and perform as many calculations in advance of the loop, that do not directly rely on the loop value.
- Potentially pass logic into the register stage before passing into the loop.

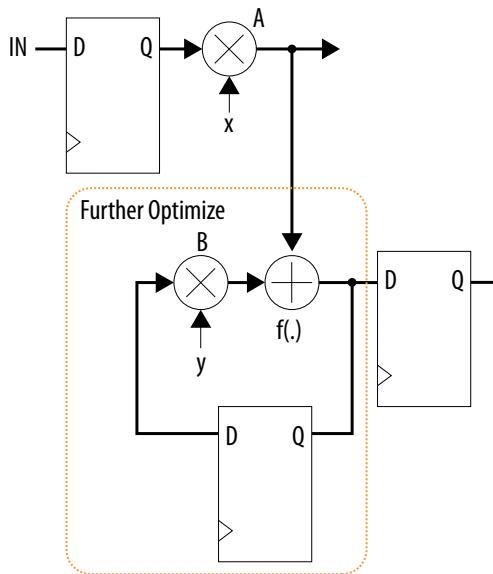
After rewriting the logic, the Compiler can now freely retime the logic that you move to the forward path.

Figure 45. Separation of Forward Logic from the Loop



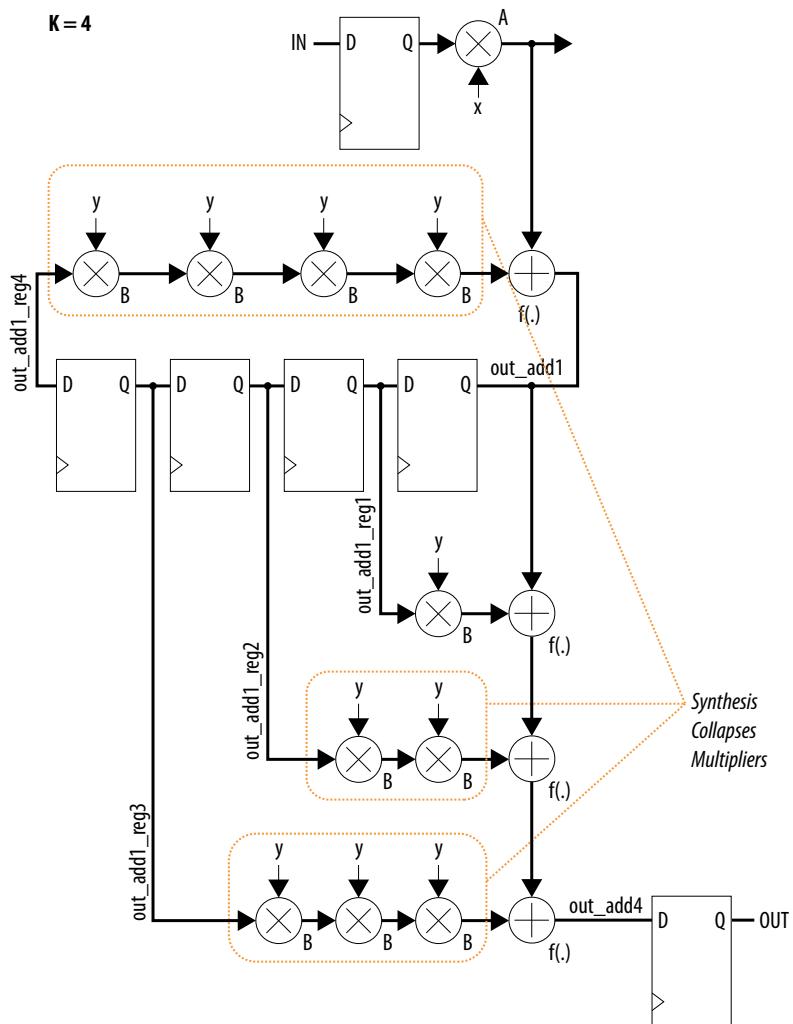
In the next optimization stage, you retime the loop register to ensure that the design functions the same as the original loop circuitry.

Figure 46. Retime Loop Register



Finally, further optimize the loop by repeating the first optimization steps with the logic in the highlighted boundary.

Figure 47. Results of Cascade Loop Logic, Hyper-Retimer, and Synthesis Optimizations (Four Level Optimization)



Example 8. Four Level Optimization Example Verilog HDL Code

```

module cll_hypr_rtm_synopt ( rstn, clk, x, y, in, out);
    input rstn, clk, x, y, in;
    output out;
    reg     out;
    reg in_reg;

    wire out_add1;
    wire out_add2;
    wire out_add3;
    wire out_add4;

    reg out_add1_reg1;
    reg out_add1_reg2;
    reg out_add1_reg3;
    reg out_add1_reg4;

    always @ ( posedge clk )

```



```

if ( !rstn ) begin
    in_reg <= 0;
end else begin
    in_reg <= in;
end

always @ ( posedge clk )
if ( !rstn ) begin
    out_add1_reg1 <= 0;
    out_add1_reg2 <= 0;
    out_add1_reg3 <= 0;
    out_add1_reg4 <= 0;
end else begin
    out_add1_reg1 <= out_add1;
    out_add1_reg2 <= out_add1_reg1;
    out_add1_reg3 <= out_add1_reg2;
    out_add1_reg4 <= out_add1_reg3;
end

assign out_add1 = x*in_reg + (((y*out_add1_reg4)*y)*y);
assign out_add2 = out_add1 + (y*out_add1_reg1);
assign out_add3 = out_add2 + ((y*out_add1_reg2)*y);
assign out_add4 = out_add3 + (((y*out_add1_reg3)*y)*y);

always @ ( posedge clk ) begin
    if ( !rstn )
        out <= 0;
    else
        out <= out_add4;
end
endmodule //c11_hypr_rtm_synopt

```

2.4.1.4.3 Loop Pipelining and Synthesis Optimization

The loop pipelining technique initially appears to create more logic to optimize this loop, resulting in less devices resources. While this technique may increase logic use in some cases, design synthesis further reduces logic through during optimization.

Synthesis optimizes the various clouds of logic. In the preceding example, synthesis ensure that the cloud of logic containing $g*g*g*g$ is smaller than implementing four instances of block g. This reduction in size is because the LUT actually has six inputs, and logic collapses, sharing some LUTs. In addition, the Hyper-Retimer retimes registers in and around this smaller cloud of logic, thus making the logic less timing-critical.

2.4.1.5 Precomputation

Precomputation is one of the easiest and most beneficial techniques for optimizing overall design speed. When confronted with critical logic, verify whether the signals the computation implies are available earlier. Always compute signals as early as possible to keep these computations outside of critical logic.

When trying to keep critical logic outside your loops, try precomputation first. The Compiler cannot optimize logic within a loop easily using retiming only. The Compiler cannot move registers inside the loop to the outside of the loop. The Compiler cannot retime registers outside the loop into the loop. Therefore, keep the logic inside the loop as small as possible so that the logic does not negatively impact f_{MAX} .

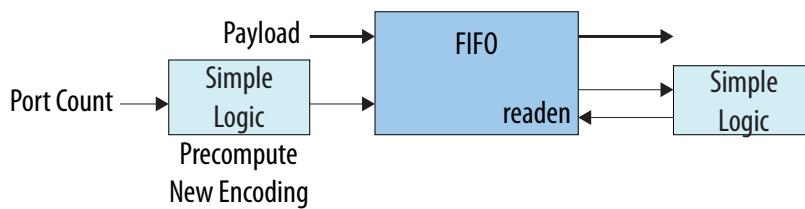
After precomputation, logic is minimized in the loop and the design precomputes the encodings. The calculation is outside of the loop, and you can optimize the calculation with pipelining or retiming. You cannot remove the loop, but can better control the effect of the loop on the design speed.

Figure 48. Restructuring a Design with an Expensive Loop

Before Precomputation



After Precomputation



The following code example shows a similar problem. The original loop contains comparison operators.

```

StateJam:if
    (RetryCnt <=MaxRetry&&JamCounter==16)
        Next_state=StateBackOff;
    else if (RetryCnt>MaxRetry)
        Next_state=StateJamDrop;
    else
        Next_state=Current_state;
    
```

Precomputing the values of `RetryCnt<=MaxRetry` and `JamCounter==16` removes the expensive computation from the `StateJam` loop and replaces the computation with simple boolean operations. The modified code is:

```

reg RetryCntGTMaxRetry;
reg JamCounterEqSixteen;
StateJam:if
    (!RetryCntGTMaxRetry && JamCounterEqSixteen)
        Next_state=StateBackOff;
    else if (RetryCntGTMaxRetry)
        Next_state=StateJamDrop;
    else
        Next_state=Current_state;
always @ (posedge Clk or posedge Reset)
if (Reset)
    JamCounterEqSixteen <= 0;
else if (Current_state!=StateJam)
    JamCounterEqSixteen <= 0;
else
    JamCounterEqSixteen <= (JamCounter == 15) ? 1:0;
always @ (posedge Clk or posedge Reset)
if (Reset)
    RetryCntGTMaxRetry <= 0;
else if (Current_state==StateSwitchNext)
    RetryCntGTMaxRetry <= 0;
else if (Current_state==StateJam&&Next_state==StateBackOff)
    RetryCntGTMaxRetry <= (RetryCnt >= MaxRetry) ? 1: 0;
    
```



2.4.2 Optimizing Specific Design Structures

This section describes common performance bottleneck structures, and recommendations to improve f_{MAX} performance for each case.

2.4.2.1 High-Speed Clock Domains

Intel Stratix 10 devices support very high-speed clock domains. The Compiler uses programmable clock tree synthesis to minimize clock insertion delay, reduce dynamic power dissipation, and provide clocking flexibility in the device core.

Device minimum pulse width constraints can limit the highest performance of Intel Stratix 10 clocks. As the number of resources on a given clock path increase, uncertainty and skew increases on the clock pulse. If clock uncertainty exceeds the minimum pulse width of the target device, this lowers the minimum viable clock period. This effect is a function of total clock insertion delay on the path. To counter this effect for high-speed clock domains, use the Chip Planner and Timing Analyzer reports to optimize clock source placement in your design.

If reports indicate limitation from long clock routes, adjust the clock pin assignment or use Clock Region or Logic Lock Region assignments to constrain fan-out logic closer to the clock source. Use Clock Region assignments to specify the clock sectors and optimize the size of the clock tree.

After making any assignment changes, recompile the design and review the clock route length and clock tree size. Review the Compilation Report to ensure that the clock network does not restrict the performance of your design.

2.4.2.1.1 Visualizing Clock Networks

After running the Fitter, visualize clock network implementation in the Chip Planner. The Chip Planner shows the source clock pin location, clock routing, clock tree size, and clock sector boundaries. Use these views to make adjustment and reduce the total clock tree size.

To visualize design clock networks in the Chip Planner:

1. Open a project.
2. On the Compilation Dashboard, click **Fitter**, **Early Place**, **Place**, **Route**, or **Retime** to run the Fitter.
3. On the Tasks pane, double-click **Chip Planner**. The Chip Planner loads device information and displays color coded chip resources.
4. On the Chip Planner Tasks pane, click **Report Clock Details**. The Chip Planner highlights the clock pin location, routing, and sector boundaries. Click elements under the **Clock Details Report** to display general and fan-out details for the element.
5. To visualize the clock sector boundaries, click the **Layers Settings** tab and enable **Clock Sector Region**. The green lines indicate the boundaries of each sector.

Figure 49. Clock Network in Chip Planner

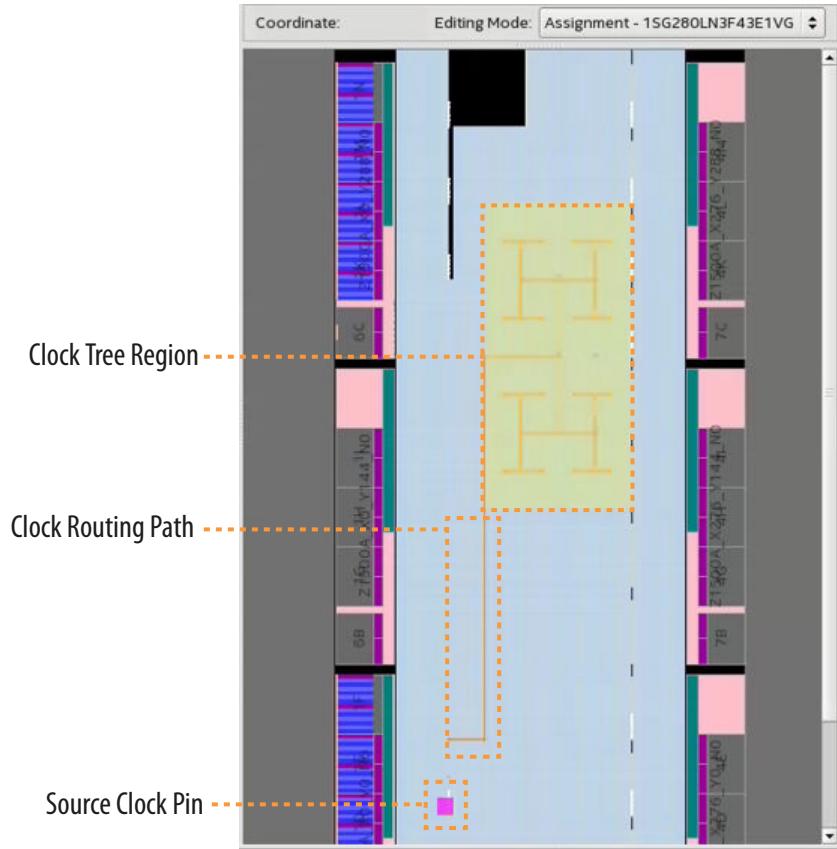


Figure 50. Clock Sector Boundary Layer in Chip Planner



2.4.2.1.2 Viewing Clock Networks in the Fitter Report

The Compilation Report provides detailed information about clock network implementation following Fitter placement. View the Global & Other Fast Signals Details report to display the length and depth of the clock path from the source clock pin to the clock tree.

To view clock network implementation in Fitter reports:

1. Open a project.
2. On the Compilation Dashboard, click **Fitter, Place, Route** to run the Fitter.
3. On the Compilation Dashboard, click the **Report** icon for the completed stage.
4. Click **Global & Other Fast Signals Details**. The table displays the length of the clock route from source to the clock tree, and the clock region depth.

Figure 51. Clock Network Details in Fitter Report

Property		Value
1	Name	dut dut hip altera_pcie_s10_hip_ast_..._chan
1	-- SDC Name	dut dut hip altera_pcie_s10_hip_ast_pipen1b
2	-- Source Type	HSSI PLDADAPT TX
3	-- Source Location	HSSIPLDADAPTTX_1CO
4	-- Fan-Out	36674
5	-- Promotion Reason	Standard promotion candidate (required due to
6	-- Clock Region	Sectors (0, 0) to (1, 1)
7	-- Clock Region Size (in Sectors)	2 x 2 (4 total)
8	-- Spine Index used in each Sector	1
9	-- Path Length from...ource to Clock Tree	1 clock sector wire(s)
10	-- Clock Tree Depth	1.5 clock sector wire(s)
2		
3	Name	auto_fab_0 alt_sld_fab_0 alt_sld_fab_0 s10
4		
5	Name	dut dut hip altera_pcie_s10_hip_ast_..._chan
6		

To visualize how these signals are routed, use the Chip Planner task "Report Clock Details".
To manually specify the size and placement of Global Signals, use the CLOCK_REGION assignment

2.4.2.1.3 Viewing Clocks in the Timing Analyzer

The Timing Analyzer reports high speed clocks that are limited by long clock paths. Open the Fmax Summary report to view any clock f_{MAX} that is restricted by high minimum pulse width violations (t_{CH}), or low minimum pulse width violation (t_{CL}).

To view clock network data in the Timing Analyzer:

1. Open a project.
2. On the Compilation Dashboard, click **Timing Analysis**. After timing analysis is complete, the **Timing Analyzer** folder appears in the Compilation Report.
3. Under the **Slow 900mV 100C Model** folder, click the **Fmax Summary** report.
4. To view path information details for minimum pulse width violations, in the Compilation Report, right-click the **Minimum Pulse Width Summary** report and click **Generate Report in Timing Analyzer**. The Timing Analyzer loads the timing netlist.
5. Click **Reports > Custom Reports > Report Minimum Pulse Width**.
6. In the **Report Minimum Pulse Width** dialog box, specify options to customize the report output and then click **OK**.
7. Review the data path details for report of long clock routes in the **Slow 900mV 100C Model** report.

Figure 52. Minimum Pulse Width Details Show Long Clock Route

The screenshot displays the Intel Timing Analyzer interface with the following sections:

- Summary of Paths** table (Top):

	Slack	Actual Width	Required Width	Type	Clock	Clock Edge	Target
1	-0.386	-0.348	0.038	High Pulse Width	clk	Rise	ff_1_source dout
2	-0.386	-0.348	0.038	High Pulse Width	clk	Rise	ff_2_hip dout
3	-0.386	-0.348	0.038	High Pulse Width	clk	Rise	ff_3_hip dout
4	-0.384	-0.346	0.038	High Pulse Width	clk	Rise	ff_4_destination dout
5	-0.348	-0.348	0.000	High Pulse Width	clk	Rise	ff_1_source dout clk
6	-0.348	-0.348	0.000	High Pulse Width	clk	Rise	ff_2_hip dout clk
7	-0.348	-0.348	0.000	High Pulse Width	clk	Rise	ff_3_hip dout clk
8	-0.346	-0.346	0.000	High Pulse Width	clk	Rise	ff_4_destination dout clk
9	-0.268	-0.225	0.043	Low Pulse Width	clk	Rise	ff_1_source dout
10	-0.268	-0.225	0.043	Low Pulse Width	clk	Rise	ff_2_hip dout
- Path #1: slack is -0.386 (VIOLATED)** message.
- Late Clock Arrival Path** table:

Total	Incr	RF	Type	Element
1	0.000	0.000		launch edge time
2	0.000	0.000		source latency
3	0.000	0.000		clk
4	0.000	0.000	RR	IC clk-input j
5	0.000	0.000	RR	CELL clk-input o
6	0.240	0.240	RR	CELL clk-input-io48tilelvds_0/m1_15_0_io12buf1_to_jopll_loclkin1
7	3.962	3.722	RR	IC ff_1_source dout clk
8	3.962	0.000	RR	CELL ff_1_source dout
- Early Clock Arrival Path** table:

Total	Incr	RF	Type	Element
1	0.250	0.250		launch edge time
2	0.250	0.000		source latency
3	0.250	0.000		clk
4	0.250	0.000	FF	IC clk-input j
5	0.250	0.000	FF	CELL clk-input o
6	0.446	0.196	FF	CELL clk-input-io48tilelvds_0/m1_15_0_io12buf1_to_jopll_loclkin1
7	3.614	3.168	FF	IC ff_1_source dout clk
8	3.614	0.000	FF	CELL ff_1_source dout

An annotation with an orange arrow points from the text "Reports show long clock delay" to the "Late Clock Arrival Path" table, specifically highlighting row 7.

2.4.2.2 Restructuring Loops

Loops are a primary target of restructuring techniques because loops fundamentally limit performance. A loop is a feedback path in a circuit. Some loops are simple and short, with a small amount of combinational logic on a feedback path. Other loops are very complex, potentially traveling through multiple registers before returning to the original register.

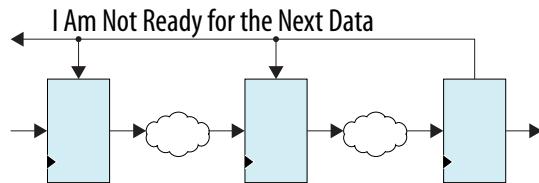
The Compiler never retimes registers into a loop, because adding a pipeline stage to a loop changes functionality. However, change your RTL manually to restructure loops to improve performance. Perform loop optimization after analyzing performance bottlenecks with Fast Forward compile. Also apply these techniques to any new RTL in your design.

2.4.2.3 Control Signal Backpressure

This section describes RTL design techniques to control signal backpressure. The Intel Stratix 10 architecture efficiently streams data. Because the architecture supports very high clock rates, it is difficult to send feedback signals to reach large amounts of logic in one clock cycle. Inserting extra pipeline registers also increases backpressure on control signals. Data must flow forward as much as possible.

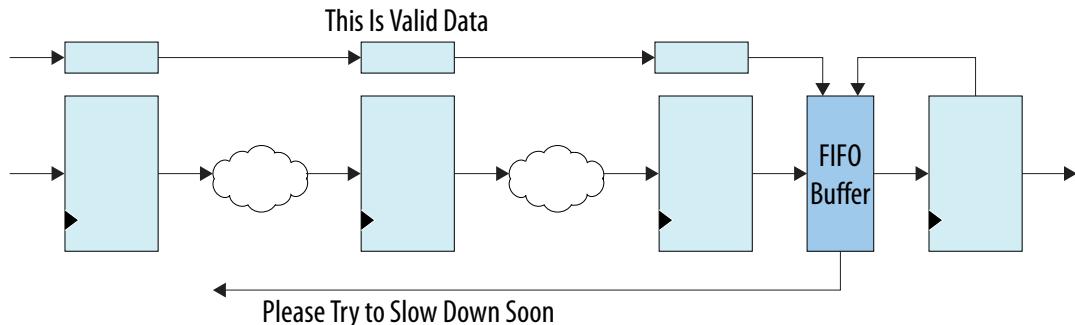
Single clock cycle control signals create loops that can prevent or reduce the effectiveness of pipelining and register retiming. This example depicts a ready signal that notifies the upstream register of readiness to consume data. The ready signals must freeze multiple data sources at the same time.

Figure 53. Control Signal Backpressure



Modifying the original RTL to add a small FIFO buffer that relieves the pressure upstream is a straightforward process. When the logic downstream of this block is not ready to use the data, the FIFO stores the data.

Figure 54. Using a FIFO Buffer to Control Backpressure



The goal is for data to reach the FIFO buffer every clock cycle. An extra bit of information decides whether the data is valid and should be stored in the FIFO buffer. The critical signal now resides between the FIFO buffer and the downstream register that consumes the data. This loop is much smaller. You can now use pipelining and register retiming to optimize the section upstream of the FIFO buffer.

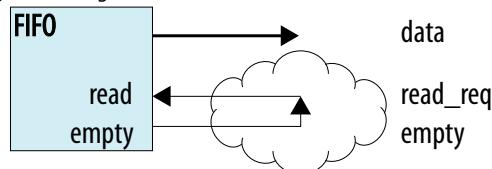
2.4.2.4 Flow Control with FIFO Status Signals

High clock speeds require consideration when dealing with flow control signals. This consideration is particularly important with signals that gate a data path in multiple locations at the same time. For example, with clock enable or FIFO full/empty signals. Instead of working with immediate control signals, use a delayed signal. You can build a buffer within the FIFO block. The control signals indicate to the upstream data path that the path is almost full, leaving a few clock cycles for the upstream data to receive their gating signal. This approach alleviates timing closure difficulties on the control signals.

When you use FIFO full and empty signals, you must process these signals in one clock cycle to prevent overflow or underflow.

Figure 55. FIFO Flow Control Loop

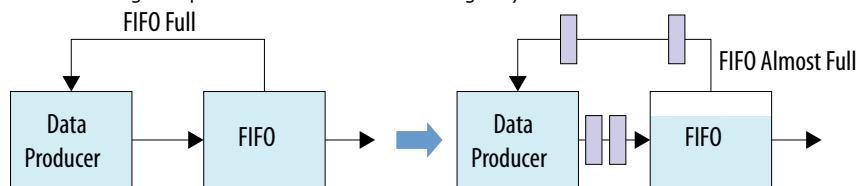
The loop forms while reading control signals from the FIFO.



If you use an almost full or almost empty signal instead, you can add pipeline registers in the flow control loop. The lower the almost full threshold, and the higher the almost empty threshold, the more registers you can add to the signal.

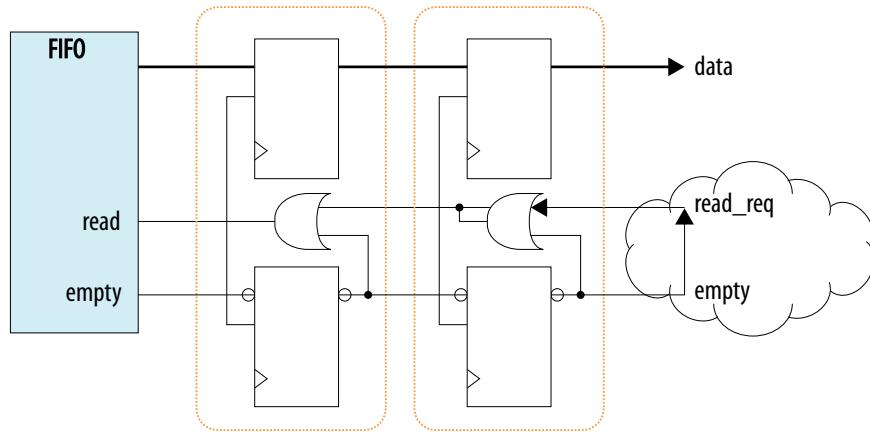
Figure 56. Improved FIFO Flow Control Loop with Almost Full instead of Full FIFO

The following example shows two extra registers in the full control flow signal. When the FIFO block signals that the path is nearly full, the circuit requires two clock cycles to stop the data flow. Size the FIFO block to allow for proper storage of those extra valid data. The extra two pipeline registers in the control path help with routing, and enable higher speed than with traditional single-cycle FIFO control scheme.



You can use skid buffers to pipeline a FIFO. If necessary, you can cascade skid buffers. When you insert skid buffers, they unroll the loop that includes the FIFO control signals. The skid buffers do not eliminate the loop in the flow control logic, but the loop transforms into a series of shorter loops. In general, switch to almost empty and almost full signals instead of using skid buffers.

Figure 57. FIFO Flow Control Loop with Two Skid Buffers in a Read Control Loop



If you have loops involving FIFO control signals, and they are broadcast to many destinations for flow control, consider whether you can eliminate the broadcast signals. Pipeline broadcast control signals, and use almost full and almost empty status bits from FIFOs.

Example 9. Skid Buffer Example

```

/ synopsys translate_off
//`timescale 1 ps / 1 ps
// synopsys translate_on

module singleclock_fifo_lowell
#(
    parameter DATA_WIDTH      = 8,
    parameter FIFO_DEPTH     = 16,
    parameter SHOWAHEAD       = "ON", // "ON" = showahead mode ('pop' is an acknowledgement); /
                                         // "OFF" = normal mode ('pop' is a request).
    parameter RAM_TYPE        = "AUTO", // "AUTO" or "MLAB" or "M20K".
    // Derived
    parameter ADDR_WIDTH      = $clog2(FIFO_DEPTH) + 1 // e.g. clog2(64) =
6, but 7 bits /
                                         needed to store 64 value
)
(
    input wire                  clk,
    input wire                  rst,
    input wire [DATA_WIDTH-1:0] in_data,    // write data
    input wire                  pop,        // rd request
    input wire                  push,       // wr request
    output wire                 out_valid,  // not empty
    output wire                 in_ready,   // not full
    output wire [DATA_WIDTH-1:0] out_data,   // rd data
    output wire [ADDR_WIDTH-1:0] fill_level
);
    wire scfifo_empty;
    wire scfifo_full;
    wire [DATA_WIDTH-1:0] scfifo_data_out;
    wire [ADDR_WIDTH-1:0] scfifo_usedw;

    logic [DATA_WIDTH-1:0] out_data_1q;
    logic [DATA_WIDTH-1:0] out_data_2q;
    logic                 out_empty_1q;
    logic                 out_empty_2q;
    logic                 e_pop_1;
    logic                 e_pop_2;

```



```
logic e_pop_qual;

assign out_valid      = ~out_empty_2q;
assign in_ready       = ~scfifo_full;
assign out_data       = out_data_2q;
assign fill_level     = scfifo_usedw + !out_empty_1q + !out_empty_2q;

// add output pipe
assign e_pop_1         = out_empty_1q || e_pop_2;
assign e_pop_2         = out_empty_2q || pop;
assign e_pop_qual = !scfifo_empty && e_pop_1;
always_ff@(posedge clk)
begin
    if(rst == 1'b1)
    begin
        out_empty_1q <= 1'b1; // empty is 1 by default
        out_empty_2q <= 1'b1; // empty is 1 by default
    end
    else begin
        if(e_pop_1)
        begin
            out_empty_1q <= scfifo_empty;
        end
        if(e_pop_2)
        begin
            out_empty_2q <= out_empty_1q;
        end
    end
end
always_ff@(posedge clk)
begin
    if(e_pop_1)
        out_data_1q <= scfifo_data_out;
    if(e_pop_2)
        out_data_2q <= out_data_1q;
end

scfifo scfifo_component
(
    .clock      (clk),
    .data       (in_data),
    .rdreq      (e_pop_qual),
    .wrreq      (push),
    .empty      (scfifo_empty),
    .full       (scfifo_full),
    .q          (scfifo_data_out),
    .usedw     (scfifo_usedw),
    // .aclr      (rst),
    // .aclr      (1'b0),
    .almost_empty (),
    .almost_full (),
    .eccstatus  (),
    // .sclr      (1'b0)
    .sclr      (rst) // switch to sync reset
);
defparam
    scfifo_component.add_ram_output_register = "ON",
    scfifo_component.enable_ecc             = "FALSE",
    scfifo_component.intended_device_family = "Stratix",
    scfifo_component.lpm_hint              = (RAM_TYPE == "MLAB") ?
"RAM_BLOCK_TYPE=MLAB" : /
    ((RAM_TYPE == "M20K") ? "RAM_BLOCK_TYPE=M20K" : ""),
    scfifo_component.lpm_numwords          = FIFO_DEPTH,
    scfifo_component.lpm_showahead        = SHOWAHEAD,
    scfifo_component.lpm_type             = "scfifo",
    scfifo_component.lpm_width            = DATA_WIDTH,
    scfifo_component.lpm_widthu          = ADDR_WIDTH,
    scfifo_component.overflow_checking   = "ON",
    scfifo_component.underflow_checking = "ON",
```

```
scfifo_component.use_eab = "ON";  
  
endmodule
```

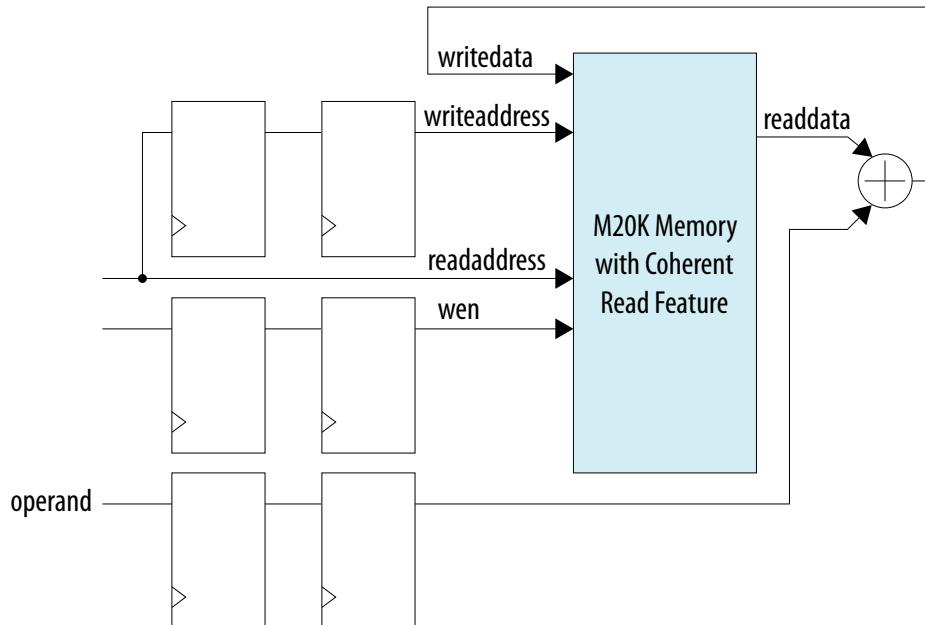
2.4.2.5 Read-Modify-Write Memory

Intel Stratix 10 M20K memory blocks support coherent reads to simplify implementing read-modify-write memory. Read-modify-write memory is useful in applications such as networking statistics counters. Read-modify-write memory is also useful in any application that stores a value in memory, that requires incrementing and re-writing in a single cycle.

Intel Stratix 10 M20K memory blocks simplify implementation by eliminating any need for hand-written caching circuitry. Caching circuitry that pipelines the modify operation over multiple clock cycles becomes complex because of high clock speeds or large counters.

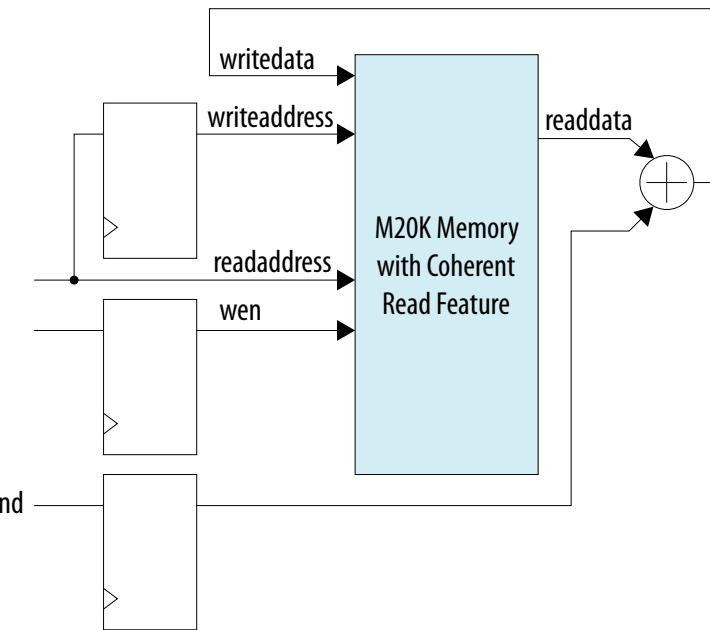
To use the coherent read feature, connect memory according to whether you register the output data port. If you register the output data port, add two register stages to the write enable and write address lines when you instantiate the memory.

Figure 58. Registered Output Data Requires Two Register Stages



If you do not register the output data port, add one register stage to the write enable and write address lines when you instantiate the memory.

Figure 59. Unregistered Output Data Requires One Register Stage



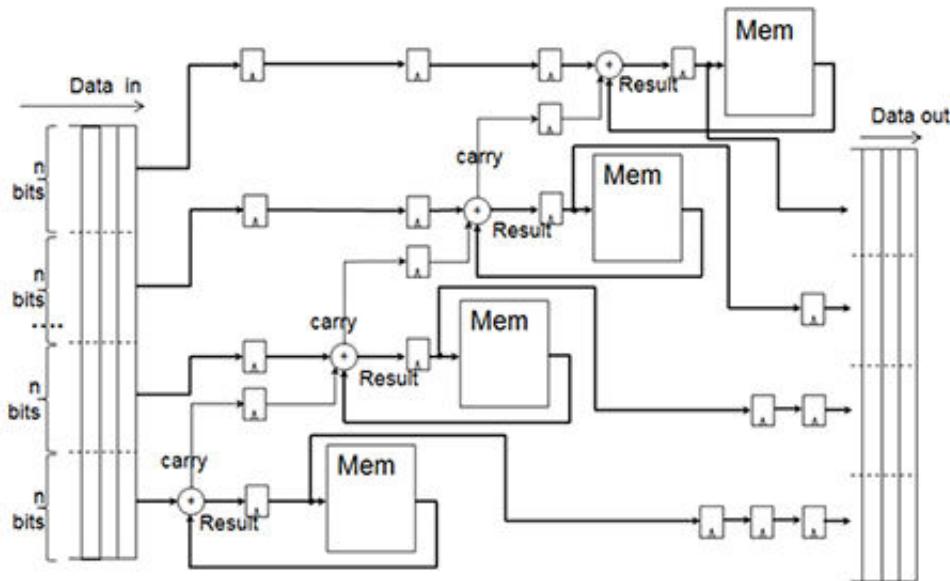
Use of coherent read has the following restrictions:

- Must use the same clock for reading and writing.
- Must use the same width for read and write ports.
- Cannot use ECC.
- Cannot use byte enable.

Figure 60. Pipelining Read-Modify-Write Memory

The following diagram shows a pipelining method for a read-modify-write memory that improves performance, without maintaining a cache for tracking recent activity. If you require M20K features that are incompatible with coherent read, or if you do not want to use coherent read, use the following alternative approaches to improve the f_{MAX} performance of memory:

- Break the modification operation into smaller blocks that can complete in one clock cycle.
- Ensure that each chunk is no wider than one M20K memory block. The Compiler splits data words into multiple n -bit chunks, where each chunk is small enough for efficient processing in one clock cycle.
- To increase f_{MAX} , increase the number of memory blocks, use narrower memory blocks, and increase the latency. To decrease latency, use fewer and wider memory blocks, and remove pipeline stages appropriately. A loop in a read-modify-write circuit is unavoidable because of the nature of the circuit, but the loop in this solution is small and short. This solution is scalable, because the underlying structure remains the same regardless of the number of pipeline stages.



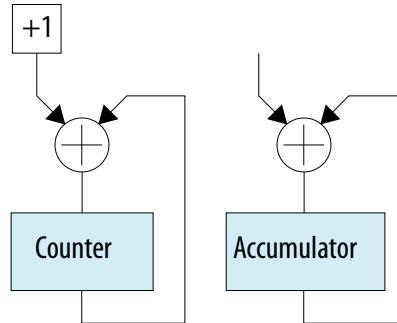
2.4.2.6 Counters and Accumulators

Performance-limiting loops occur rarely in small, simple counters. Counters with unnatural rollover conditions (not a power of two), or irregular increments, are more likely to have a performance-limiting critical chain. When a performance-limiting loop occurs in a small counter (roughly 8 bits or less), write the counter as a fully decoded state machine, depending on all the inputs that control the counter. The counter still contains loops, but they are smaller, and not performance-limiting. When the counter is small (roughly 8 bits or less), the Fitter implements the counter in a single LAB. This implementation makes the counter fast because all the logic is placed close together.

You can also use loop unrolling to improve counter performance.

Figure 61. Counter and Accumulator Loop

In a counter and accumulator loop, a register's new value depends on its old value. This includes variants like LFSRs (linear feedback shift register) and gray code counters.



2.4.2.7 State Machines

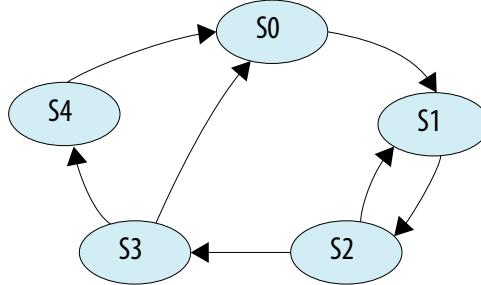
Loops related to state machines can be difficult to optimize. Carefully examine the state machine logic to determine whether you can precompute any signals that the next state logic uses.

To effectively pipeline the state machine loop, consider adding skips states to a state machine. Skips states are states that you use to allow more transition time between two adjacent states.

Optimization of state machine loops may require a new state machine.

Figure 62. State Machine Loop

In a state machine loop, the next state depends on the current state of the circuit.



Related Links

- [Appendix A: Parameterizable Pipeline Modules](#) on page 109
- [Precomputation](#) on page 39

2.4.2.8 Memory

The section covers various topics about optimization for hard memory blocks in Intel Stratix 10 devices.

2.4.2.8.1 True Dual-Port Memory

Intel Stratix 10 devices support true dual-port memory structures. True dual-port memories allow two write and two read operations at once.



Intel Stratix 10 embedded memory components (M20K) have slightly different modes of operation compared to previous Intel FPGA technology, including mixed-width ratio for read/write access.

Intel Stratix 10 devices do not support true dual-port memories in independent clock mode. However, Intel Stratix 10 devices fully support true dual-port memories in single clock mode with an operation frequency of up to 1 GHz.

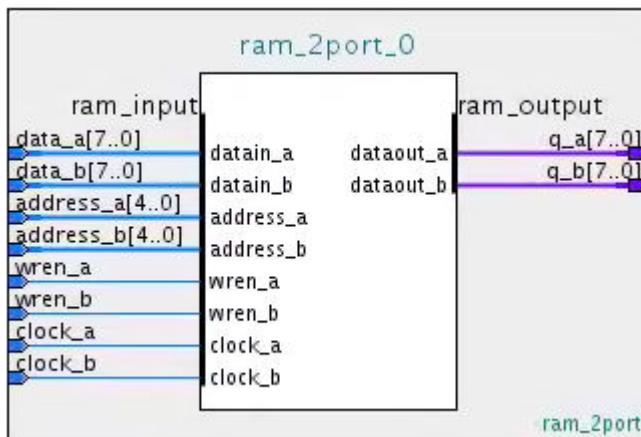
2.4.2.8.2 Use Simple Dual-Port Memories

When migrating a design to an Intel Stratix 10 device, consider whether your original design contains a dual-port memory that uses different clocks on each port. If your design is actually using the same clock on both write ports, restructure it using two simple dual-clock memories.

The advantage of this method is that the simple dual-port blocks support frequencies up to 1 GHz. The disadvantage is the doubling of the number of memory blocks required to implement your memory.

Figure 63. Intel Arria® 10 True Dual-Port Memory Implementation

Previous versions of the Intel Quartus Prime Pro Edition software generate this true dual-port memory structure for Intel Arria® 10 devices.



Example 10. Dual Port, Dual Clock Memory Implementation

```
module true_dual_port_ram_dual_clock
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=6)
(
    input [(DATA_WIDTH-1):0] data_a, data_b,
    input [(ADDR_WIDTH-1):0] addr_a, addr_b,
    input we_a, we_b, clk_a, clk_b,
    output reg [(DATA_WIDTH-1):0] q_a, q_b
);

// Declare the RAM variable
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

always @ (posedge clk_a)
begin
    // Port A
    if (we_a)
        begin
            ram[addr_a] <= data_a;
        end
    end
end

// Port B
always @ (posedge clk_b)
begin
    // Port B
    if (we_b)
        begin
            ram[addr_b] <= data_b;
        end
    end
end
endmodule
```

```

        q_a <= data_a;
    end
    else
    begin
        q_a <= ram[addr_a];
    end
end

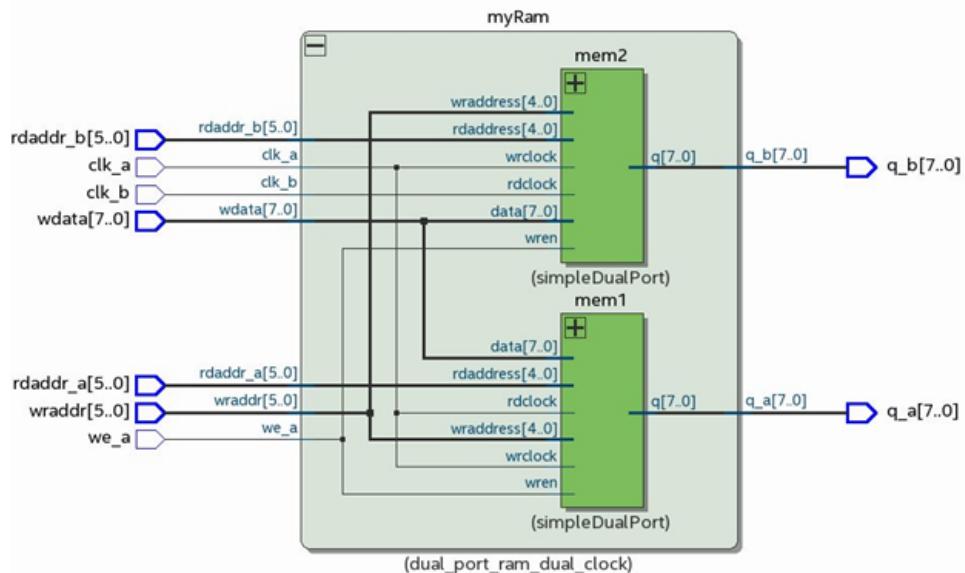
always @ (posedge clk_b)
begin
    // Port B
    if (we_b)
    begin
        ram[addr_b] <= data_b;
        q_b <= data_b;
    end
    else
    begin
        q_b <= ram[addr_b];
    end
end
endmodule
    
```

Synchronizing dual-port memory that uses different write clocks can be difficult. Ensure that both ports do not simultaneously write to a given address. In many designs the dual-port memory often performs a write operation on one of the ports, followed by two read operations using both ports (1W2R). You can model this behavior by using two simple dual-port memories. In simple dual-port memories, a write operation always writes in both memories, while a read operation is port dependent.

2.4.2.8.3 Simple Dual-Port Memory Example

Using two simple dual-port memories can double the use of M20K blocks in the device. However, this memory structure can perform at a frequency up to 1 GHz. This frequency is not possible when using true dual-port memory with independent clocks in Intel Stratix 10 devices.

Figure 64. Simple Dual-Port Memory Implementation





You can achieve similar frequency results by inferring simple dual-port memory in RTL, rather than by instantiation in the GUI.

Example 11. Simple Dual-Port RAM Inference

```
module simple_dual_port_ram_with_SDPs
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=6)
(
    input [(DATA_WIDTH-1):0] wrdata,
    input [(ADDR_WIDTH-1):0] wraddr, rdaddr,
    input we_a, wrclock, rdclock,
    output reg [(DATA_WIDTH-1):0] q_a
);

// Declare the RAM variable
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

always @ (posedge wrclock)
begin
    // Port A is for writing only
    if (we_a)
        begin
            ram[wraddr] <= wrdata;
        end
end

always @ (posedge rdclock)
begin
    // Port B is for reading only
    begin
        q_a <= ram[rdaddr];
    end
end
endmodule
```

Example 12. True Dual-Port RAM Behavior Emulation

```
module test (wrdata, wraddr, rdaddr_a, rdaddr_b,
            clk_a, clk_b, we_a, q_a, q_b);

    input [7:0] wrdata;
    input clk_a, clk_b, we_a;
    input [5:0] wraddr, rdaddr_a, rdaddr_b;
    output [7:0] q_a, q_b;

    simple_dual_port_ram_with_SDPs myRam1 (
        .wrdata(wrdata),
        .wraddr(wraddr),
        .rdaddr(rdaddr_a),
        .we_a(we_a),
        .wrclock(clk_a), .rdclock(clk_b),
        .q_a(q_a)
    );

    simple_dual_port_ram_with_SDPs myRam2 (
        .wrdata(wrdata),
        .wraddr(wraddr),
        .rdaddr(rdaddr_b),
        .we_a(we_a),
        .wrclock(clk_a), .rdclock(clk_a),
        .q_a(q_b)
    );

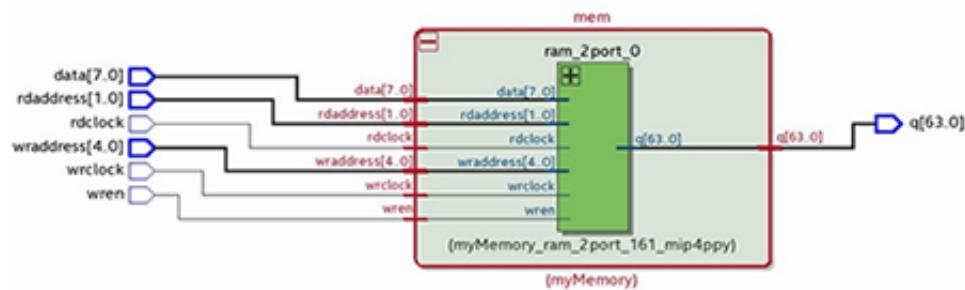
endmodule
```

2.4.2.8.4 Memory Mixed Port Width Ratio Limits

Intel Stratix 10 device block RAMs enable clocks speeds of up to 1GHz. The new RAM block design is more restrictive with respect to use of mixed ports data width. Intel Stratix 10 device block RAMs do not support 1/32, 1/16, or 1/8 mixed port ratios. The only valid ratios are 1, 1/2, and 1/4 mixed port ratios. The Compiler generates an error message for implementation of invalid mixed port ratios.

When migrating a design that uses invalid port width ratios for Intel Stratix 10 devices, modify the RTL to create the desired ratio.

Figure 65. Dual-Port Memory with Invalid 1/8 Mixed Port Ratio

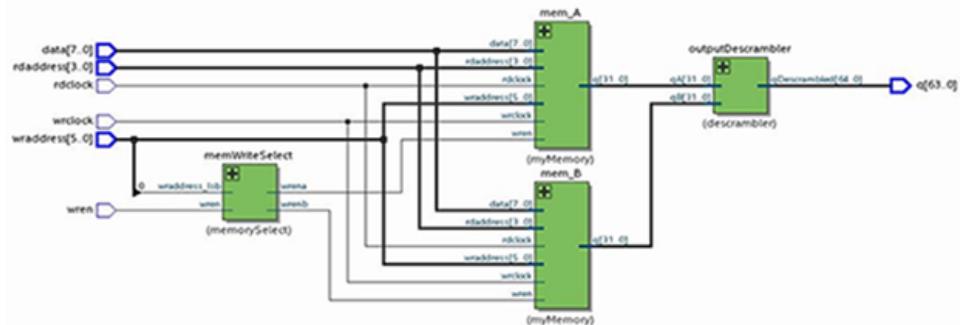


To create a functionally equivalent design for Intel Stratix 10 devices, create and combine smaller memories with valid mixed port width ratios. For example, the following steps implement a mixed port width ratio:

1. Create two memories with 1/4 mixed port width ratio by instantiating the 2-Ports memory IP core from the IP Catalog.
2. Define write enable logic to ping-pong writing between the two memories.
3. Interleave the output of the memories to rebuild a 1/8 ratio output.

Figure 66. 1/8 Width Ratio Example

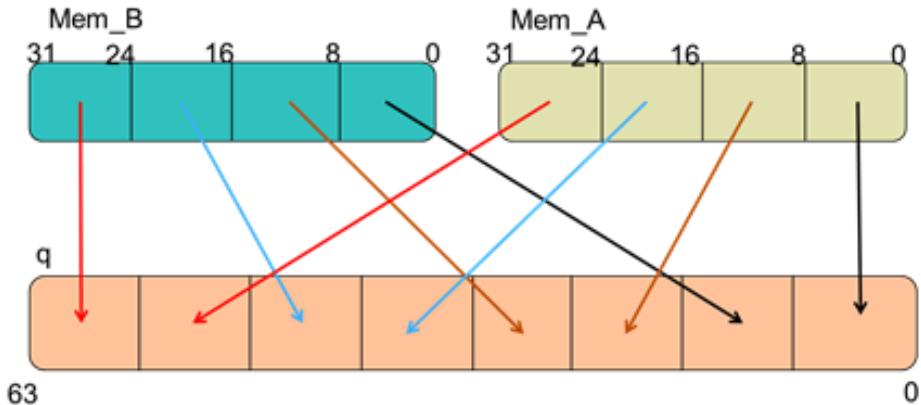
This example shows the interleaving of two memories and the write logic. The chosen write logic uses the least significant bit of the write address to decide which memory to write. Even addresses write in memory mem_A, odd addresses write in memory mem_B.



Because of the scheme that controls writing to the memories, carefully reconstruct the full 64-bit output during a write. You must account for the interleaving of the individual 8-bit words in the two memories.

Figure 67. Memory Output Descrambling Example

This example shows the descrambled output when attempting to read at address 0h0.



The following RTL examples implement the extra stage to descramble the data from memory on the read side.

Example 13. Top-Level Descramble RTL Code

```

module test
#(
    parameter WR_DATA_WIDTH = 8,
    parameter RD_DATA_WIDTH = 64,
    parameter WR_DEPTH = 64,
    parameter RD_DEPTH = 4,
    parameter WR_ADDR_WIDTH = 6,
    parameter RD_ADDR_WIDTH = 4
) (
    input      [WR_DATA_WIDTH-1:0]      data,
    input      [WR_ADDR_WIDTH-1:0]      wraddress,
    input      [RD_ADDR_WIDTH-1:0]      rdaddress;
    input      wren;
    input      wrclock;
    input      rdclock;
    output     [RD_DATA_WIDTH-1:0]      q;

    wire wrena, wrenb;
    wire [(RD_DATA_WIDTH/2)-1:0] q_A, q_B;

    memorySelect memWriteSelect (
        .wraddress_lsb(wraddress[0]),
        .wren(wren),
        .wrena(wrena),
        .wrenb(wrenb)
    );

    myMemory mem_A (
        .data(data),
        .wraddress(wraddress),
        .rdaddress(rdaddress),
        .wren(wrena),
        .wrclk(wrclock),
        .rdclk(rdclock),
        .q(q_A)
    );
    myMemory mem_B (

```

```

    .data(data),
    .wraddress(wraddress),
    .rdaddress(rdaddress),
    .wren(wrenb),
    .wrclock(wrclock),
    .rdclock(rdclock),
    .q(q_B)
);

descrambler #(
    .WR_WIDTH(WR_DATA_WIDTH),
    .RD_WIDTH(RD_DATA_WIDTH)
) outputDescrambler (
    .qA(q_A),
    .qB(q_B),
    .qDescrambled(q)
);

endmodule

```

Example 14. Supporting RTL Code

```

module memorySelect (wraddress_lsb, wren, wrena, wrenb);
    input wraddress_lsb;
    input wren;
    output wrena, wrenb;

    assign wrena = !wraddress_lsb && wren;
    assign wrenb = wraddress_lsb && wren;
endmodule

module descrambler #(
    parameter WR_WIDTH = 8,
    parameter RD_WIDTH = 64
) (
    input [(RD_WIDTH/2)-1 : 0] qA,
    input [(RD_WIDTH/2)-1 : 0] qB,
    output [RD_WIDTH:0] qDescrambled
);

genvar i;
generate
    for (i=WR_WIDTH*2; i<=RD_WIDTH; i += WR_WIDTH*2) begin: descramble
        assign qDescrambled[i-WR_WIDTH-1:i-(WR_WIDTH*2)] = qA[(i/2)-1:(i/2)-WR_WIDTH];
        assign qDescrambled[i-1:i-WR_WIDTH] = qB[(i/2)-1:(i/2)-WR_WIDTH];
    end
endgenerate
endmodule

```

2.4.2.8.5 Unregistered RAM Outputs

To achieve the highest performance, register the output of memory blocks before using the data in any combinational logic. Driving combinational logic directly, with unregistered memory outputs, can result in a critical chain with insufficient registers.

You can unknowingly use unregistered memory outputs, followed by combinational logic, if you implement a RAM using the read-during-write new data mode. The Compiler implements this mode with soft logic outside of the memory block that



compares the read and write addresses. This mode muxes the write data straight to the output. If you want to achieve the highest performance, do not use the read-during-write new data mode.

2.4.2.9 DSP Blocks

DSP blocks support frequencies up to 1 GHz. However, you must use all of the registers, including the input register, two stages of pipeline registers, and the output register.

2.4.2.10 General Logic

Avoid using one-line logic functions that while structurally sound, generate multiple levels of logic. The only exception to this is adding a couple of pipeline registers on either side, so that Hyper-Retiming can retime through the cloud of logic.

2.4.2.11 Modulus and Division

The modulus and division operators are costly in terms of device area and speed performance, unless they use powers of two. If possible, use an implementation that avoids a modulus or division operator. The *Round Robin Scheduler* topic shows the replacement of a modulus operator with a simple shift, resulting in a dramatic performance increase.

Related Links

[Round Robin Scheduler](#) on page 95

2.4.2.12 Resets

Use resets for circuits with loops in monitoring logic to detect erroneous conditions, and pipeline the reset condition.

2.4.2.13 Hardware Re-use

To resolve loops caused by hardware re-use, unroll the loops.

2.4.2.14 Algorithmic Requirements

These loops can be difficult to improve, but can sometimes benefit from a combination of optimization techniques described in the *General Optimization Techniques* section.

Related Links

[General Optimization Techniques](#) on page 29

2.4.2.15 FIFOs

FIFOs always contain loops. There are efficient methods to implement the internal FIFO logic that provide excellent performance.

One feature of some FIFOs is a bypass mode where data bypasses the internal memory completely when the FIFO is empty. If you implement this mode in any of your FIFOs, be aware of the possible performance limitations inherent in unregistered memory outputs.

Related Links

[Unregistered RAM Outputs](#) on page 58

2.4.2.16 Ternary Adders

Implementing ternary adders can increase resource usage in Intel Stratix 10 devices. However, unless your design heavily relies on ternary adder structure, additional resource usage may not be noticeable at the top design level. However, a review of the design level where you add a ternary adder structure shows an increase in LUT count. In addition, the amount of resource increase directly correlates to the size of the adder. Small width adders (size < 16 bits) do not cause much resource difference. However, increasing the size of the adder increases the resource count differential for Intel Stratix 10 devices in comparison with older FPGA technology.

Ternary Adder RTL Code

```
module ternary_adder (CLK, A, B, C, OUT);
    parameter WIDTH = 16;
    input [WIDTH-1:0] A, B, C;
    input CLK;
    output [WIDTH-1:0] OUT;

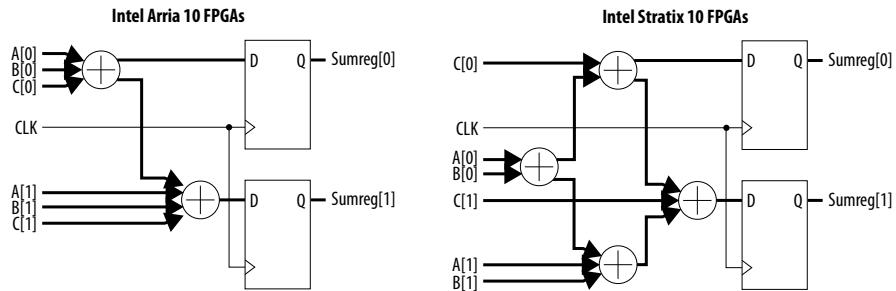
    wire [WIDTH-1:0] sum1;
    reg [WIDTH-1:0] sumreg1;

    // 3-bit additions
    assign sum1 = A + B + C;
    assign OUT = sumreg1;

    // Registers
    always @ (posedge CLK)
        begin
            sumreg1 <= sum1;
        end
endmodule
```

This increase in device resource use occurs because the Intel Stratix 10 ALM does not have a shared arithmetic mode that previous FPGA technologies have. The ALM in shared arithmetic mode can implement a three-input add in the ALM. By contrast, the Intel Stratix 10 ALM can implement only a two-input add in the ALM.

Figure 68. RTL View of Intel Arria 10 versus Intel Stratix 10 to add 2 LSBs from a three 8-bit input adder



In shared arithmetic mode, the Intel Arria 10 ALM allows a three-input adder to use three adaptive LUT (ALUT) inputs: CIN, SHAREIN, COUT, SUMOUT, and SHAREOUT. The absence of the shared arithmetic mode restricts ALM use with only two ALUT inputs: CIN, COUT and SUMOUT. The figure below shows the resulting implementation of a ternary adder on both Intel Arria 10 and Intel Stratix 10 FPGAs.

Figure 69. Intel Arria 10: ALMs used to add 2 LSBs from a three 8-bit input adder

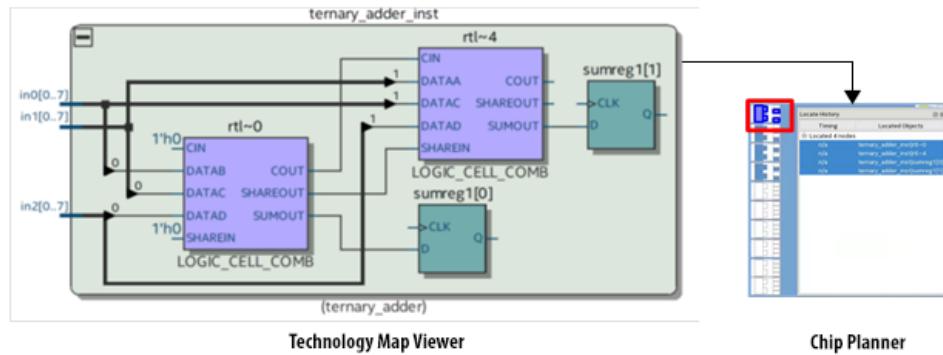
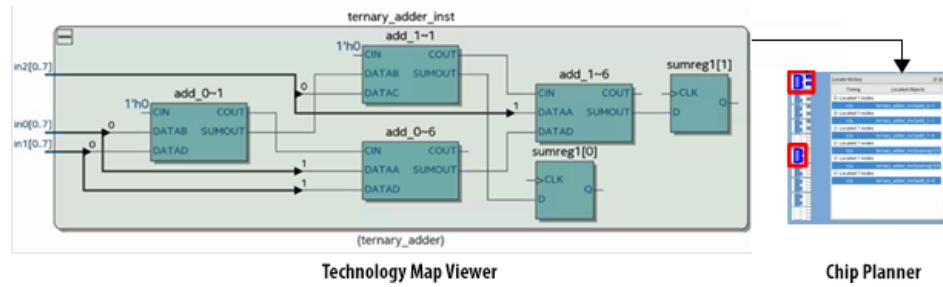


Figure 70. Intel Stratix 10: ALMs used to add 2 LSBs from a three 8-bit input adder



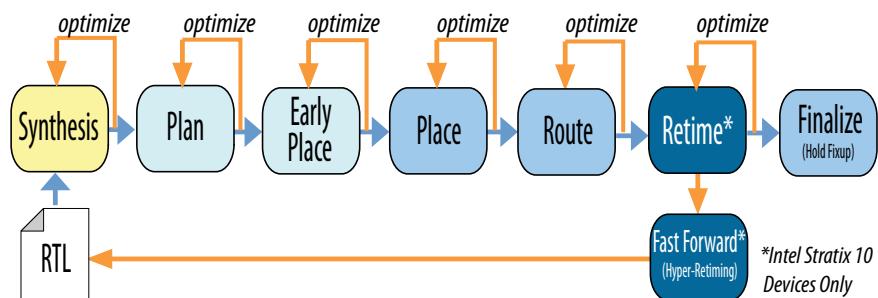
3 Compiling Intel Stratix 10 Designs

The Intel Quartus Prime Pro Edition Compiler is optimized to take full advantage of the Intel Hyperflex FPGA architecture in Intel Stratix 10 devices. The Intel Quartus Prime Pro Edition Compiler supports the Hyper-Aware design flow, in which the Compiler maximizes retiming of registers into Hyper-Registers.

Hyper-Aware Design Flow

Use the Hyper-Aware design flow to shorten design cycles and optimize performance. The Hyper-Aware design flow combines automated register retiming, with implementation of targeted timing closure recommendations (Fast Forward compilation), to maximize use of Hyper-Registers and drive the highest performance for Intel Stratix 10 designs.

Figure 71. Hyper-Aware Design Flow



Register Retiming

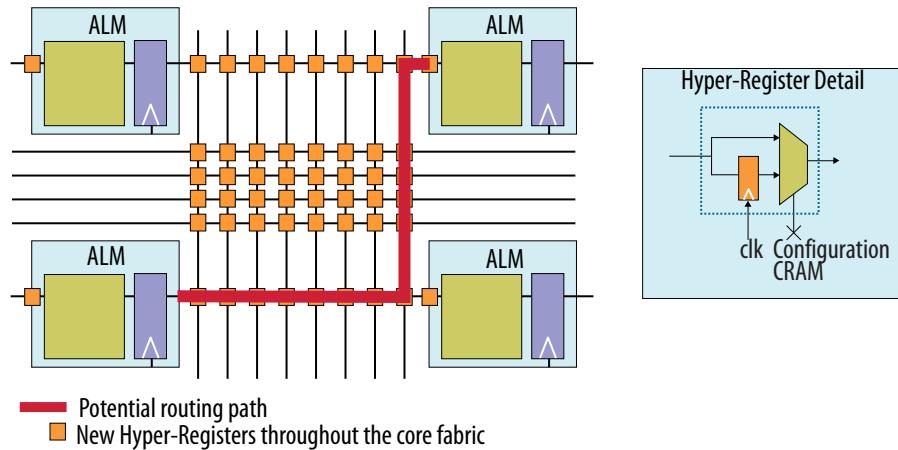
A key innovation of the Intel Stratix 10 architecture is the addition of multiple Hyper-Registers in every routing segment and block input. Maximizing the use of Hyper-Registers improves design performance. The prevalence of Hyper-Registers improves balance of time delays between registers and mitigates critical path delays. The Compiler's **Retime** stage moves registers out of ALMs and retimes them into Hyper-Registers, wherever advantageous. Register retiming runs automatically during the Fitter, requires minimal effort, and can result in significant performance improvement. Following retiming, the **Finalize** stage corrects connections with hold violations.

Fast Forward Compilation

If you require optimization beyond simple register retiming, run Fast Forward compilation to generate timing closure recommendations that break key performance bottlenecks that prevent further movement into Hyper-Registers. For example, Fast Forward recommends removing specific retiming restrictions that prevent further retiming into Hyper-Registers. Fast Forward compilation shows precisely where to make the most impact with RTL changes, and reports the predictive performance benefits you can expect from removing restrictions and retiming into Hyper-Registers.

(Hyper-Retiming). The Fitter does not automatically retime registers across RAM and DSP blocks. However, Fast Forward analysis shows the potential performance benefit from this optimization.

Figure 72. Hyper-Register Architecture



Fast-Forward compilation identifies the best location to add pipeline stages (Hyper-Pipelining), and the expected performance benefit in each case. After you modify the RTL to place pipeline stages at the boundaries of each clock domain, the **Retime** stage automatically places the registers within the clock domain at the optimal locations to maximize performance. Implement the recommendations in RTL to achieve similar results. After implementing any changes, re-run the **Retime** stage until the results meet performance and timing requirements. Fast Forward compilation does not run automatically as part of a full compilation. Enable or run **Fast Forward compilation** in the Compilation Dashboard.

Table 4. Optimization Steps

Optimization Step	Technique	Description
Step 1	Register Retiming	Retime stage moves existing registers into Hyper-Registers.
Step 2	Fast Forward Compile	Compiler generates design-specific timing closure recommendations and predicts performance improvement with removal of all barriers to Hyper-Registers (Hyper-Retiming).
Step 3	Hyper-Pipelining	Use Fast Forward compilation to identify where to add new registers and pipeline stages in RTL.
Step 4	Hyper-Optimization	Design optimization beyond Hyper-Retiming and Hyper-Pipelining, such as restructuring loops, removing control logic limits, and reducing the delay along long paths.

Related Links

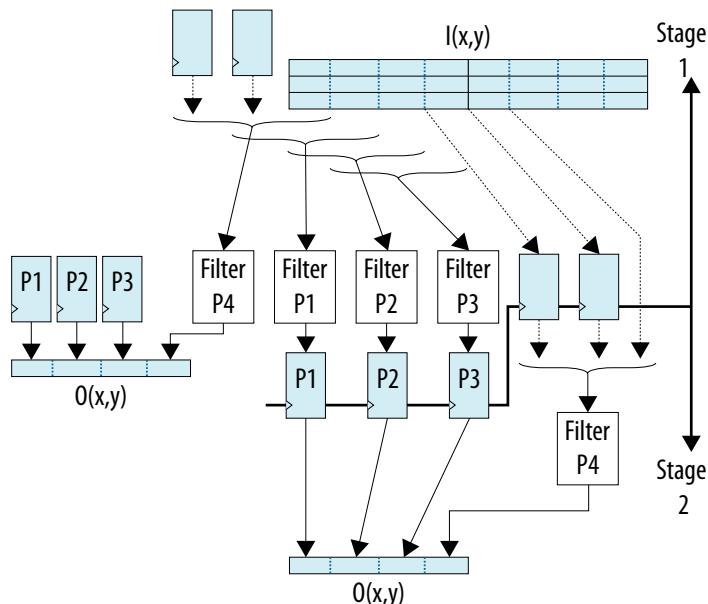
[Design Compilation, Intel Quartus Prime Pro Edition Handbook](#)

For complete step by step information about compiling Intel Stratix 10 designs.

4 Design Example Walk-Through

This walk-through illustrates performance optimization after Fast-Forward compilation and Hyper-Retiming techniques on a real-world Median Filter image processing design. Fast Forward compilation generates recommendations for design RTL changes to achieve the highest performance with the Intel Hyperflex architecture. This walk-through describes project setup, design compilation, results analysis, and RTL optimization.

Figure 73. Median Filter Operational Diagram



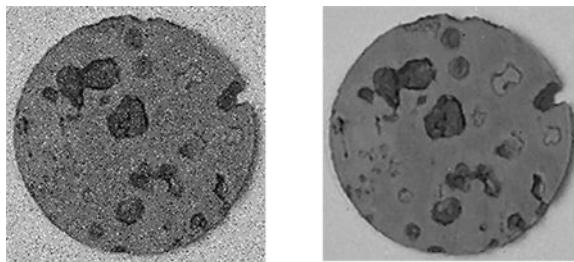
4.1 Median Filter Design Example

The Median filter is a non-linear filter that removes impulsive noise from an image. These filters require the highest performance. The design requirement is to perform real time image processing on a factory floor.

Intel provides the supporting design example project and design files for this walk-through at <https://github.com/intel/fpga-median-s10>. You can download and unzip the verified project, constraint, design, RTL from the median.zip to complete this walkthrough.⁽¹⁾



Figure 74. Before and After Images Processed with Median Filtering



Median Filter Design Example Files

After download and extraction, the Median filter design example .zip file contains the following directories under the Median_filter_design_example directory:

Table 5. Median Filter Design Example Files

File Name	Description
Base	Contains the original version of the design and project files.
Final	Contains the final version of the design and project files with all RTL optimizations in place.
fpga-median.ORIGINAL	Contains the original OpenSource version of the Median filter and the associated research paper.
Step_1	Incremental RTL design changes and project files for Fast Forward optimization step 1.
Step_2	Incremental RTL design changes and project files for Fast Forward optimization step 2.

This walk-through covers the following steps:

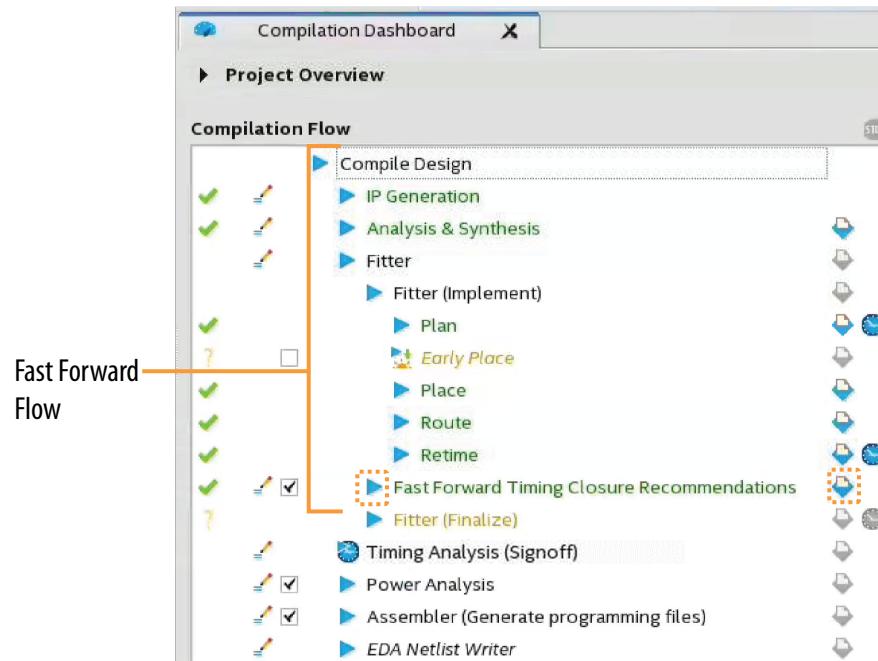
1. [Step 1: Compile the Base Design](#) on page 65
2. [Step 2: Add Pipeline Stages and Remove Asynchronous Resets](#) on page 67
3. [Step 3: Add More Pipeline Stages and Remove All Asynchronous Resets](#) on page 69
4. [Step 4: Optimize Short Path and Long Path Conditions](#) on page 71

4.1.1 Step 1: Compile the Base Design

Follow these steps to compile the base design:

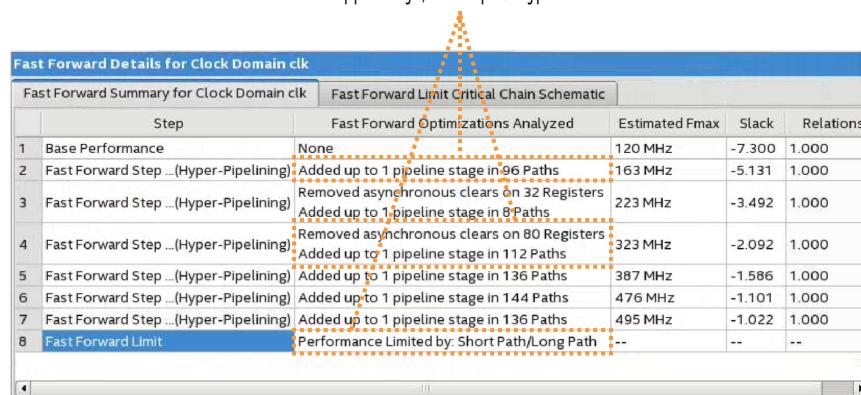
(1) The paper *An FPGA-Based Implementation for Median Filtering Meeting the Real-Time Requirements of Automated Visual Inspection Systems* first presented this design at the *10th Mediterranean Conference on Control and Automation*, Lisbon, Portugal, 2002. The design is publicly available under GNU General Public License that the Free Software Foundation publishes.

1. Download and unzip the median design example files at <https://github.com/intel/fpga-median-s10>.
2. In the Intel Quartus Prime Pro Edition software, click **File > Open Project** and select the Median_filter_<version>/Base/median.qpf project file. The base version of the design example opens.
3. To compile the base design, click **Compile Design** on the Compilation Dashboard. By default, the Fast Forward Timing Recommendations stage runs during the Fitter, and generates detailed recommendations in the **Fast Forward Timing Closure Recommendations > Fast Forward Details** report.



4. In the Fast Forward Details report, view the compilation results for the Clk clock domain.

Fast Forward reports asynchronous registers,
a need for pipeline stages, and short path/long path combinations





The report indicates a **Base Performance** of 120 MHz, with the following design conditions limiting further optimization:

- The design contains asynchronous resets (clears).
- Additional pipeline stages (registers) can improve performance.
- Short path and long path combinations limit further optimization.

The following steps describe implementation of these recommendations in the design RTL.

4.1.2 Step 2: Add Pipeline Stages and Remove Asynchronous Resets

This first optimization step adds five levels of pipeline registers in the design locations that Fast Forward suggests, and removes the asynchronous resets present in a design module. This optimization step increases f_{MAX} performance to the level that Fast Forward estimates.

To add pipeline stages and remove asynchronous resets from the design:

1. Open the Median_filter_<version>/Step_1/rtl/hyper_pipe.sv. This file defines a parameterizable hyper_pipe pipeline component that you can easily use in any design. The following shows this component's code with parameterizable width (WIDTH) and depth (NUM_PIPES):

```
module hyper_pipe #(
    parameter WIDTH = 1,
    parameter NUM_PIPES = 1
)
(
    input clk,
    input [WIDTH-1:0] din,
    output [WIDTH-1:0] dout);

reg [WIDTH-1:0] hp [NUM_PIPES-1:0];

genvar i;
generate
    if (NUM_PIPES == 0) begin
        assign dout = din;
    end
    else begin
        always @ (posedge clk)
            hp[0] <= din;
        for (i=1;i < NUM_PIPES;i++) begin : hregs
            always @ (posedge clk) begin
                hp[i] <= hp[i-1];
            end
        end
        assign dout = hp[NUM_PIPES-1];
    end
endgenerate
endmodule
```

2. Use the parameterizable module to add some levels of pipeline stages to the locations that Fast Forward recommends. The following example shows how to add latency before the q output of the dff_3_pipe module:

```
...
hyper_pipe #(
    .WIDTH (DATA_WIDTH),
    .NUM_PIPES(4)
) hp_d0 (
    .clk(clk),
```



```
    .din(d0),
    .dout(q0_int)
);

.

always @(posedge clk)
begin : register_bank_3u
    if(~rst_n) begin
        q0 <= {DATA_WIDTH{1'b0}};
        q1 <= {DATA_WIDTH{1'b0}};
        q2 <= {DATA_WIDTH{1'b0}};
    end else begin
        q0 <= q0_int;
        q1 <= q1_int;
        q2 <= q2_int;
    end
end
```

3. Remove the asynchronous resets inside the dff_3_pipe module by simply changing the registers to synchronous registers, as shown below. Refer to [Reset Strategies](#) on page 13 for general examples of efficient reset implementations.

```
always @(posedge clk or negedge rst_n) // Asynchronous reset

begin : register_bank_3u
    if(~rst_n) begin
        q0 <= {DATA_WIDTH{1'b0}};
        q1 <= {DATA_WIDTH{1'b0}};
        q2 <= {DATA_WIDTH{1'b0}};
    end else begin
        q0_reg <= d0;
        q1_reg <= d1;
        q2_reg <= d2;
        q0 <= q0_reg;
        q1 <= q1_reg;
        q2 <= q2_reg;
    end
end

.

always @(posedge clk)
begin : register_bank_3u
    if(~rst_n_int) begin // Synchronous reset
        q0 <= {DATA_WIDTH{1'b0}};
        q1 <= {DATA_WIDTH{1'b0}};
        q2 <= {DATA_WIDTH{1'b0}};
    end else begin
        q0 <= q0_int;
        q1 <= q1_int;
        q2 <= q2_int;
    end
end
```

These RTL changes add five levels of pipeline to the inputs of the median_wrapper design (word0, word1, and word2 buses), and five levels of pipeline into the dff_3_pipe module. The following steps show the results of these changes.

4. To implement the changes, save all design changes and click **Compile Design** on the Compilation Dashboard.
5. Following compilation, once again view the compilation results for the Clk clock domain in the Fast Forward Details report.



Fast Forward Details for Clock Domain clk					
Fast Forward Summary for Clock Domain clk		Fast Forward Limit Critical Chain Schematic			
Step	Fast Forward Optimizations Analyzed	Estimated Fmax	Slack	Relationship	
1 Base Performance	None	495 MHz	-1.022	1.000	
2 Fast Forward Step ... (Hyper-Pipelining)	Added up to 1 pipeline stage in 96 Paths	537 MHz	-0.861	1.000	
3 Fast Forward Step ... (Hyper-Pipelining)	Added up to 1 pipeline stage in 99 Paths	588 MHz	-0.701	1.000	
4 Fast Forward Step ... (Hyper-Pipelining)	Removed asynchronous clears on 20 Registers	627 MHz	-0.594	1.000	
5 Fast Forward Step ... (Hyper-Pipelining)	Removed asynchronous clears on 20 Registers Added up to 1 pipeline stage in 99 Paths	687 MHz	-0.456	1.000	
6 Fast Forward Step ... (Hyper-Pipelining)	Added up to 1 pipeline stage in 99 Paths	691 MHz	-0.448	1.000	
7 Fast Forward Limit	Performance Limited by: Short Path/Long Path	--	--	--	

Critical Chain at Fast Forward Limit					
Optimizations Analyzed (Cumulative)		Recommendations for Critical Chain		Critical Chain Details	
Path Info	Register	Register ID	Element		
1 Short Path	REG	#1	hp_word1 hp[2][25]~RTM~_LAB_RE_X264_Y350_d		
2 Short Path			hp_word1 hp[2][25]~RTM~_LAB_RE_X264_Y350_d		
3 Short Path			median_inst common_network_u0 node_u3 data_u3 hp_word0 hp[3][29]~RTM~_LAB_RE_X264_Y350_d		
4 Short Path			median_inst common_network_u0 node_u3 data_u3 hp_word0 hp[3][29]~RTM~_LAB_RE_X264_Y350_d		
5 Short Path			hp_word0 hp[3][29]~RTM~_LAB_RE_X264_Y350_d		

The report shows the effect of the RTL changes on the **Base Performance** f_{MAX} of the design. The design performance now increases to 495 MHz.

The report indicates that you can achieve further performance improvement by removing more asynchronous registers, adding more pipeline registers, and addressing optimization limits of short path and long path. The following steps describe implementation of these recommendations in the design RTL.

Note: As an alternative to completing the preceding steps, you can open and compile the Median_filter_<version>/Step_1/median.qpf project file that already includes these changes, and then observe the results.

4.1.3 Step 3: Add More Pipeline Stages and Remove All Asynchronous Resets

The Fast Forward Timing Closure Recommendations suggest further changes that you can make to enable additional optimization during retiming. The **Optimizations Analyzed** tab reports the specific registers in the analysis for you to modify. The report indicates that state_machine.v still contains asynchronous resets that limit optimization. Follow these steps to remove remaining asynchronous resets in state_machine.v, and add more pipeline stages:

1. Use the techniques and examples in [Step 2: Add Pipeline Stages and Remove Asynchronous Resets](#) on page 67 to change all asynchronous resets to synchronous resets in state_machine.v. These resets are in multiple locations in the file, as the report indicates.
2. In the Fast Forward Details report, select the last optimization row before the **Fast Forward Limit** row, and then click the **Optimizations Analyzed** tab. **Optimizations Analyzed** indicates the location and number of registers to add.

Fast Forward Details for Clock Domain clk

Fast Forward Summary for Clock Domain clk		Fast Forward Limit Critical Chain Schematic		
Step	Fast Forward Optimizations Analyzed	Estimated Fmax	Slack	Relationship
1 Base Performance	None	495 MHz	-1.022	1.000
2 Fast Forward Step ... (Hyper-Pipelining)	Added up to 1 pipeline stage in 96 Paths	537 MHz	-0.861	1.000
3 Fast Forward Step ... (Hyper-Pipelining)	Added up to 1 pipeline stage in 99 Paths	588 MHz	-0.701	1.000
4 Fast Forward Step ... (Hyper-Pipelining)	Removed asynchronous clears on 20 Registers	627 MHz	-0.594	1.000
5 Fast Forward Step ... (Hyper-Pipelining)	Removed asynchronous clears on 20 Registers Added up to 1 pipeline stage in 99 Paths	687 MHz	-0.456	1.000
6 Fast Forward Step ... (Hyper-Pipelining)	Added up to 1 pipeline stage in 99 Paths	691 MHz	-0.448	1.000
7 Fast Forward Limit	Performance Limited by: Short Path/Long Path	--	--	--

Optimizations Analyzed for Fast Forward Step 5 (691 MHz)

Optimizations Analyzed (for Fast Forward Step #5)		Optimizations Analyzed (Cumulative)
Optimizations Analyzed (Cumulative)		
1	Added up to 4 pipeline stages in 99 Paths for Clock Domain clk	
1	Added up to 4 pipeline stages in 99 Paths from Top-level Input ports to Clock Domain 'clk'	
1	Added up to 4 pipeline stages in 99 Paths from Top-level Input ports to Entity hyper_pipe	
1	Added up to 4 pipeline stages in 1 Path from Top-level Input ports to Instance median_inst dff_out_pipe hp_rst	
2	Added up to 4 pipeline stages in 1 Path from Top-level Input ports to Instance median_inst dff_c2_pipe hp_rst	
3	Added up to 4 pipeline stages in 1 Path from Top-level Input ports to Instance median_inst dff_c3_pipe hp_rst	

3. Use the techniques and examples in [Step 2: Add Pipeline Stages and Remove Asynchronous Resets](#) on page 67 to add the number of pipeline stages at the locations in the **Optimizations Analyzed** tab.
4. Once again, compile the design and view the Fast Forward Details report. The performance increase is similar to the estimates, but short path and long path combinations still limit further performance. The next step addresses this performance limit.

Fast Forward Details for Clock Domain clk

Fast Forward Summary for Clock Domain clk		Fast Forward Limit Critical Chain Schematic		
Step	Fast Forward Optimizations Analyzed	Estimated Fmax	Slack	Relationship
1 Base Performance	None	656 MHz	-0.524	1.000
2 Fast Forward Limit	Performance Limited by: Short Path/Long Path	--	--	--

Critical Chain at Fast Forward Limit

Optimizations Analyzed (Cumulative)		Recommendations for Critical Chain	Critical Chain Details
Path Info	Register	Register ID	Element
1 Extend to locatable node	REG	#1	hp_word0 hp[9][0]-RTM~_C4_X271_Y336_NO_I4
2 Extend to locatable node			hp_word0 hp[9][0]-RTM~_C4_X271_Y336_NO_I4
3 Short Path	Bypassed Hyper-Register		hp_word0 hp[9][0]-RTM~_R10_X262_Y337_NO_I5
4 Short Path			hp_word0 hp[9][0]-RTM~_LOCAL_INTERCONNECT
5 Short Path			hp_word0 hp[9][0]-RTM~_LAB_RE_X270_Y337_N
6 Short Path	REG	#2	hp_word0 hp[9][0]-RTM~_LAB_RE_X270_Y337_N
7 Short Path			hp_word0 hp[9][0]-RTM~_LAB_RE_X270_Y337_N

Note: As an alternative to completing the preceding steps, you can open and compile the Median_filter_<version>/Step_2/median.qpf project file that already includes these changes, and then observe the results.



4.1.4 Step 4: Optimize Short Path and Long Path Conditions

After removing asynchronous registers and adding pipeline stages, the Fast Forward Details report suggests that short path and long path conditions limit further optimization. In this example, the longest path limits the f_{MAX} for this specific clock domain. To increase the performance, follow these steps to reduce the length of the longest path for this clock domain.

1. To view the long path information, click the **Critical Chain Details** tab in the Fast Forward Details report. Review the structure of the logic around this path, and consider the associated RTL code. This path involves the node module of the node.v file. The critical path relates to the computation of registers data_hi and data_lo, which are part of several comparators.

Optimizations Analyzed (Cumulative)		Recommendations for Critical Chain		Critical Chain Details
	Path Info	Register	Register ID	Element
75	Long Path (Critical)			hp_word0 hp[9][0]~RTM~_C4_X271_Y336_N0_148
76	Long Path (Critical)	REG	#1	hp_word0 hp[9][0]~RTM~_C4_X271_Y336_N0_148
77	Long Path (Critical)	Bypassed Hyper-Register		hp_word0 hp[9][0]~RTM~_LAB_RE_X271_Y338_N0
78	Long Path (Critical)			hp_word0 hp[9][0]~RTM~_LOCAL_INTERCONNECT
79	Long Path (Critical)			median_inst common_network_u0 node_u0 data_l
80	Long Path (Critical)			median_inst common_network_u0 node_u0 data_l
81	Long Path (Critical)	Bypassed Hyper-Register		median_inst common_network_u0 node..._lo[0]~3~_L
82	Long Path (Critical)			median_inst common_network_u0 no..._o[0]~3~_LA
83	Long Path (Critical)			median_inst common_network_u0 no..._AL_INTERCO
84	Long Path (Critical)			median_inst common_network_u0 node_u0 data_l
85	Long Path (Critical)			median_inst common_network_u0 node_u4 data_l
86	Long Path (Critical)			median_inst common_network_u0 node_u4 data_l
87	Long Path (Critical)	Bypassed Hyper-Register		median_inst common_network_u0 nod..._hi[0]~3~_L
88	Long Path (Critical)			median_inst common_network_u0 nod..._hi[0]~3~_L
89	Long Path (Critical)			median_inst common_network_u0 no..._AL_INTERCO
90	Long Path (Critical)			median_inst common_network_u0 node_u8 LessT
91	Long Path (Critical)			median_inst common_network_u0 node_u8 LessT
92	Long Path (Critical)	REG	#4	median_inst common_network_u0 no..._0~1~_LAB
93	Long Path (Critical)			median_inst common_network_u0 nod...~1~_LAB
94	Long Path (Critical)			median_inst common_network_u0 no..._an_0~1~_L
95	Long Path (Critical)			median_inst common_network_u0 no..._AL_INTERCO
96	Long Path (Critical)			median_inst common_network_u0 node_u8 LessT
97	Long Path (Critical)			median_inst common_network_u0 node_u8 LessT
98	Long Path (Critical)			median_inst common_network_u0 node_u8 LessT
99	Long Path (Critical)			median_inst common_network_u0 node_u8 LessT
100	Long Path (Critical)			median_inst common_network_u0 node_u8 LessT
101	Long Path (Critical)			median_inst common_network_u0 node_u8 LessT

The following shows the original RTL for this path:

```
always @(*)
begin : comparator
    if(data_a < data_b) begin
        sel0 = 1'b0; // data_a : lo / data_b : hi
    end else begin
        sel0 = 1'b1; // data_b : lo / data_a : hi
    end
end

always @(*)
begin : mux_lo_hi
    case (sel0)
        1'b0 :
        begin
            if(LOW_MUX == 1)
                data_lo = data_a;
            if(HI_MUX == 1)
                data_hi = data_b;
        end
        1'b1 :
        begin
            if(LOW_MUX == 1)
```

```

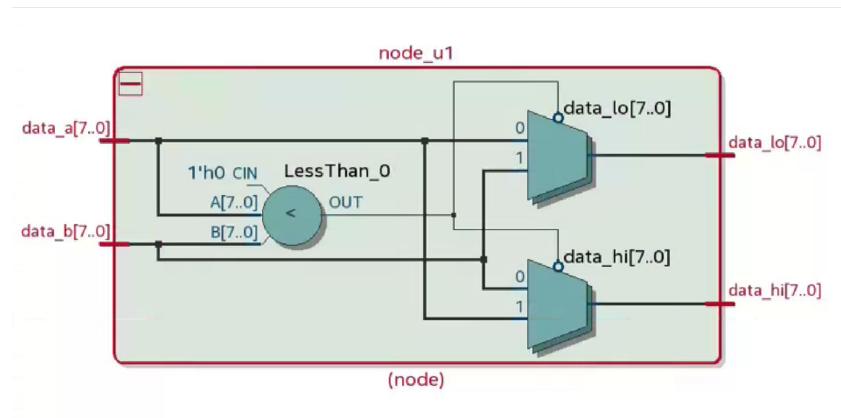
        data_lo = data_b;
        if(HI_MUX == 1)
            data_hi = data_a;
        end
        default :
        begin
            data_lo = {DATA_WIDTH{1'b0}};
            data_hi = {DATA_WIDTH{1'b0}};
        end
        endcase
    end

```

The Compiler infers the following logic from this RTL:

- A comparator that creates the sel0 signal
- A pair of muxes that create the data_hi and data_lo signals, as the following figure shows:

Figure 75. Node Component Connections



2. Review the `pixel_network.v` file that instantiates the node module. The node module's outputs are unconnected when you do not use them. These unconnected outputs result in no use of the `LOW_MUX` or `HI_MUX` code. Rather than inferring muxes, use bitwise logic operation to compute the values of the `data_hi` and `data_lo` signals, as the following example shows:

```

reg [DATA_WIDTH-1:0] sel0;

always @(*)
begin : comparator
    if(data_a < data_b) begin
        sel0 = {DATA_WIDTH{1'b0}}; // data_a : lo / data_b : hi
    end else begin
        sel0 = {DATA_WIDTH{1'b1}}; // data_b : lo / data_a : hi
    end
end

data_lo = (data_b & sel0) | (data_a & ~sel0);
data_hi = (data_a & sel0) | (data_b & ~sel0);

```

3. Once again, compile the design and view the Fast Forward Details report. The performance increase is similar to the estimates, and short path and long path combinations no longer limit further performance. After this step, only a logical loop limits further performance.



Figure 76. Short Path and Long Path Conditions Optimized

Fast Forward Details for Clock Domain clk					
Fast Forward Summary for Clock Domain clk			Fast Forward Limit Critical Chain Schematic		
Step	Fast Forward Optimizations Analyzed	Estimated Fmax	Slack	Relationship	
1 Base Performance	None	740 MHz	-0.351	1.000	
2 Fast Forward Limit	Performance Limited by RTL Loop	--	--	--	

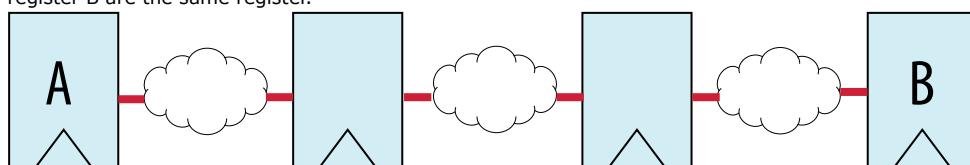
Note: As an alternative to completing the preceding steps, you can open and compile the Median_filter_<version>/Final/median.qpf project file that already includes these changes, and then observe the results.

5 Retiming Restrictions and Workarounds

The Compiler identifies the register chains in your design that limit further optimization through Hyper-Retiming. The Compiler refers to these related register-to-register paths as a critical chain. The f_{MAX} of the critical chain and its associated clock domain is limited by the average delay of a register-to-register path, and quantization delays of indivisible circuit elements like routing wires. There are a variety of situations that cause retiming restrictions. Retiming restrictions exist because of hardware characteristics, software behavior, or are inherent to the design. The **Retiming Limit Details** report the limiting reasons preventing further retiming, and the registers and combinational nodes that comprise the chain. The Fast Forward recommendations list the steps you can take to remove critical chains and enable additional register retiming.

Figure 77. Sample Critical Chain

In this figure the red line represents a same critical chain. Timing restrictions prevent register A from retiming forward. Timing restrictions also prevent register B from retiming backwards. A loop occurs when register A and register B are the same register.



Fast Forward recommendations for the critical chain include:

- Reduce the delay of 'Long Paths' in the chain. Use standard timing closure techniques to reduce delay. Combinational logic, sub-optimal placement, and routing congestion, are among the reasons for path delay..
- Insert more pipeline stages in 'Long Paths' in the chain. Long paths are the parts of the critical chain that have the most delay between registers.
- Increase the delay (or add pipeline stages to 'Short Paths' in the chain).

Particular registers in critical chains can limit performance for many other reasons. The Compiler classifies the following types of reasons that limit further optimization by retiming:

- Insufficient Registers
- Loop
- Short path/long path
- Path limit

After understanding why a particular critical chain limits your design's performance, you can then make RTL changes to eliminate that bottleneck and increase performance.

**Table 6. Hyper-Register Support for Various Design Conditions**

Design Condition	Hyper-Register Support
Initial conditions that cannot be preserved	Hyper-Registers do have initial condition support. However, you cannot perform some retiming operations while preserving the initial condition stage of all registers (that is, the merging and duplicating of Hyper-Registers). If this condition occurs in the design, the Fitter does not retime those registers. This retiming limit ensures that the register retiming does not affect design functionality.
Register has an asynchronous clear	Hyper-Registers support only data and clock inputs. Hyper-Registers do not have control signals such as asynchronous clears, presets, or enables. The Fitter cannot retime any register that has an asynchronous clear. Use asynchronous clears only when necessary, such as state machines or control logic. Often, you can avoid or remove asynchronous clears from large parts of a datapath.
Register drives an asynchronous signal	This design condition is inherent to any design that uses asynchronous resets. Focus on reducing the number of registers that are reset with an asynchronous clear.
Register has don't touch or preserve attributes	The Compiler does not retime registers with these attributes. If you use the preserve attribute to manage register duplication for high fan-out signals, try removing the preserve attribute. The Compiler may be able to retime the high fan-out register along each of the routing paths to its destinations. Alternatively, use the dont_merge attribute. The Compiler retimes registers in ALMs, DDIOs, single port RAMs, and DSP blocks.
Register is a clock source	This design condition is uncommon, especially for performance-critical parts of a design. If this retiming restriction prevents you from achieving the required performance, consider whether a PLL can generate the clock, rather than a register.
Register is a partition boundary	This condition is inherent to any design that uses design partitions. If this retiming restriction prevents you from achieving the required performance, add additional registers inside the partition boundary for Hyper-Retiming.
Register is a block type modified by an ECO operation	This restriction is uncommon. Avoid the restriction by making the functional change in the design source and recompiling, rather than performing an ECO.
Register location is an unknown block	This restriction is uncommon. You can often work around this condition by adding extra registers adjacent to the specified block type.
Register is described in the RTL as a latch	Hyper-Registers cannot implement latches. The Compiler infers latches because of RTL coding issues, such as incomplete assignments. If you do not intend to implement a latch, change the RTL.
Register location is at an I/O boundary	All designs contain I/O, but you can add additional pipeline stages next to the I/O boundary for Hyper-Retiming.
Combinational node is fed by a special source	This condition is uncommon, especially for performance-critical parts of a design.
Register is driven by a locally routed clock	Only the dedicated clock network clocks Hyper-Registers. Using the routing fabric to distribute clock signals is uncommon, especially for performance-critical parts of a design. Consider implementing a small clock region instead.
Register is a timing exception end-point	The Compiler does not retime registers that are sources or destinations of .sdc constraints.

continued...

Design Condition	Hyper-Register Support
Register with inverted input or output	This condition is uncommon.
Register is part of a synchronizer chain	The Fitter optimizes synchronizer chains to increase the mean time between failure (MTBF), and the Compiler does not retime registers that are detected or marked as part of a synchronizer chain. Add more pipeline stages at the clock domain boundary adjacent to the synchronizer chain to provide flexibility for the retiming.
Register with multiple period requirements for paths that start or end at the register (cross-clock boundary)	This situation occurs at any cross-clock boundary, where a register latches data on a clock at one frequency, and fans out to registers running at another frequency. The Compiler does not retime registers at cross-clock boundaries. Consider adding additional pipeline stages at one side of the clock domain boundary, or the other, to provide flexibility for retiming.

The following topics describes RTL design techniques that you can use to remove retiming restrictions.

Related Links

[Timing Constraint Considerations](#) on page 19

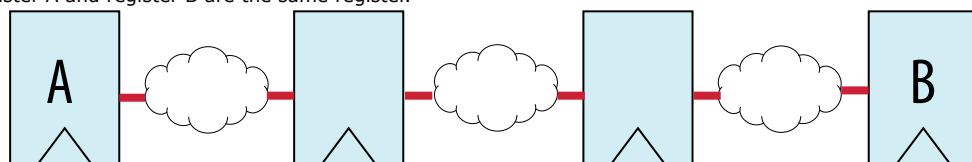
5.1 Interpreting Critical Chain Reports

The Compiler identifies the register chains in your design that limit further optimization through Hyper-Retiming. The Compiler refers to these related register-to-register paths as a critical chain. The f_{MAX} of the critical chain and its associated clock domain is limited by the average delay of a register-to-register path, and quantization delays of indivisible circuit elements like routing wires.

The Retiming Limit Details report the limiting reasons preventing further retiming, and the registers and combinational nodes that comprise the chain. The Fast Forward recommendations list the steps you can take to remove critical chains and enable additional register retiming.

Figure 78. Sample Critical Chain

In this figure the red line represents the same critical chain. Timing restrictions prevent register A from retiming forward. Timing restrictions also prevent register B from retiming backwards. A loop occurs when register A and register B are the same register.



Fast Forward recommendations include:

- Reduce the delay of 'Long Paths' in the chain. Use standard timing closure techniques to reduce delay. Combinational logic, sub-optimal placement, and routing congestion, are among the reasons for path delay..
- Insert more pipeline stages in 'Long Paths' in the chain. Long paths are the parts of the critical chain that have the most delay between registers.
- Increase the delay (or add pipeline stages to 'Short Paths' in the chain).

Particular registers in critical chains can limit performance for many other reasons.



The Compiler classifies the following types of reasons that limit further optimization by retiming:

- Insufficient Registers
- Loop
- Short path/long path
- Path limit

After understanding why a particular critical chain limits your design's performance, you can then make RTL changes to eliminate that bottleneck and increase performance.

5.1.1 Insufficient Registers

When registers at neither end of the chain can be retimed, but adding more registers can improve performance, the limiting reason reported is Insufficient Registers.

Figure 79. Insufficient Registers Reported as Limiting Reason

Retiming Limit Details			
Retiming Limit Summary			
Clock Transfer	Limiting Reason	Recommendation	
1 Clock Domain clock	Insufficient Registers	See the Fast Forward Timing Closure R...RTL changes and estimate	
2 Transfer from clock to Top-level Output ports	Insufficient Registers	See the Fast Forward Timing Closure R...RTL changes and estimate	

Critical Chain Summary for Transfer from clock to Top-level Output ports			
Critical Chain Details			
Path Info	Register	Register ID	Element
1 Retiming Restriction	PIN		dir[0]
2 Long Path (Critical)			auto dir[0]~input i
3 Long Path (Critical)			auto dir[0]~input o
4 Long Path (Critical)			auto dir[0]~input~io48tilevds_0/s2_3_0__core_periph
5 Long Path (Critical)			auto dir[0]~input~IO_48_LVDS_TILE_RE_X52_Y0_NO
6 Long Path (Critical)			auto dir[0]~input~REDMUX_X51_Y2_NO_I136
7 Long Path (Critical)			auto dir[0]~input~HIO_BUFFER_X51_Y2_NO_I76
8 Long Path (Critical)	Bypassed Hyper-Register		auto dir[0]~input~R4_X49_Y2_NO_I57

5.1.1.1 Insufficient Registers Example

The following screenshots show the relevant parts of the Retiming Limit Details report and the logic in the critical chain.

The Retiming Limit Details report indicates that the performance of the `clk` domain fails to meet the timing requirement of 1ns period (1GHz f_{MAX}) with a slack of -1.311ns, corresponding to a f_{MAX} of 432.7 MHz.

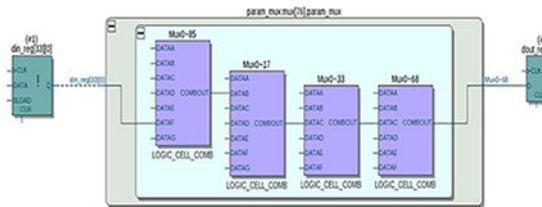
Figure 80. Retiming Limit Details

Retiming Limit Details			
Retiming Limit Summary			
Clock Transfer	Limiting Reason	Recommendation	
1 Clock Domain clock	Insufficient Registers	See the Fast Forward Timing Closure R...RTL changes and estimate	
2 Transfer from clock to Top-level Output ports	Insufficient Registers	See the Fast Forward Timing Closure R...RTL changes and estimate	

The circuit has an inefficient crossbar switch implemented with one stage of input registers, one stage of output registers, and purely combinational logic to route the signals. The input and output registers have asynchronous resets. Because the multiplexer in the crossbar is not pipelined, the implementation is inefficient and the performance is limited.

Figure 81. Critical Chain in Post-Fit Technology Map Viewer

The critical chain goes from the input register, through a combinational logic cloud, to the output register. The critical chain contains only one register-to-register path.



In the following figure, line 1 shows a timing restriction in the **Path Info** column. Line 37 also lists a retiming restriction. The asynchronous resets on the two registers cause the retiming restrictions.

Figure 82. Critical Chain with Insufficient Registers Reported During Hyper-Retiming

Retiming Limit Summary			
Clock Transfer	Limiting Reason	Stack	Relationship
1. Clock Domain clk	Insufficient Registers	-1.311	1.000
Critical Chain Summary for Clock Domain clk			
Critical Chain at Limit			
Path Info	Register	Join	Element
1. Retiming Restriction	REG (required)	#2	din_reg[33][0]
2. Long Path (Critical)			din_reg[33][0]
3. Long Path (Critical)			param_mux[0~85]_param_mux
4. Long Path (Critical)			param_mux[0~85]_param_mux
5. Long Path (Critical)	bypassed Hyper Register		din_reg[33][0]-LAB_FE_X96_Y126_NO_93
6. Long Path (Critical)	bypassed Hyper Register		din_reg[33][0]-A3_X93_Y126_NO_98
7. Long Path (Critical)	bypassed Hyper Register		din_reg[33][0]-A3_X93_Y126_NO_100
8. Long Path (Critical)	bypassed Hyper Register		din_reg[33][0]-A3_X93_Y126_NO_118
9. Long Path (Critical)	bypassed Hyper Register		din_reg[33][0]-C_X94_Y126_NO_119
10. Long Path (Critical)	bypassed Hyper Register		din_reg[33][0]-C_X94_Y126_NO_120
11. Long Path (Critical)	bypassed Hyper Register		din_reg[33][0]-E3_X95_Y126_NO_128
12. Long Path (Critical)	bypassed Hyper Register		din_reg[33][0]-C_X95_Y126_NO_128
13. Long Path (Critical)	bypassed Hyper Register		din_reg[33][0]-A3_X95_Y126_NO_129
14. Long Path (Critical)	bypassed Hyper Register		din_reg[33][0]-A3_X95_Y126_NO_129
15. Long Path (Critical)	bypassed Hyper Register		inst[76]_param_muxA0~0@pcat
16. Long Path (Critical)			inst[76]_param_muxA0~0@pcat
17. Long Path (Critical)			param_mux[0~85]_param_muxA0~0@pcat
18. Long Path (Critical)			param_mux[0~85]_param_muxA0~0@pcat
19. Long Path (Critical)	bypassed Hyper Register		param_mux[0~85]_param_muxA0~0@pcat
20. Long Path (Critical)	bypassed Hyper Register		param_mux[0~85]_param_muxA0~0@pcat
21. Long Path (Critical)			inst[76]_param_muxA0~0@pcat
22. Long Path (Critical)			inst[76]_param_muxA0~0@pcat
23. Long Path (Critical)			param_mux[0~85]_param_muxA0~0@pcat
24. Long Path (Critical)			param_mux[0~85]_param_muxA0~0@pcat
25. Long Path (Critical)	bypassed Hyper Register		param_mux[0~85]_param_muxA0~0@pcat
26. Long Path (Critical)	bypassed Hyper Register		param_mux[0~85]_param_muxA0~0@pcat
27. Long Path (Critical)	bypassed Hyper Register		param_mux[0~85]_param_muxA0~0@pcat
28. Long Path (Critical)	bypassed Hyper Register		param_mux[0~85]_param_muxA0~0@pcat
29. Long Path (Critical)	bypassed Hyper Register		param_mux[0~85]_param_muxA0~0@pcat
30. Long Path (Critical)	bypassed Hyper Register		param_mux[0~85]_param_muxA0~0@pcat
31. Long Path (Critical)	bypassed Hyper Register		param_mux[0~85]_param_muxA0~0@pcat
32. Long Path (Critical)	bypassed Hyper Register		param_mux[0~85]_param_muxA0~0@pcat
33. Long Path (Critical)	bypassed Hyper Register		param_mux[0~85]_param_muxA0~0@pcat
34. Long Path (Critical)	bypassed Hyper Register		param_mux[0~85]_param_muxA0~0@pcat
35. Long Path (Critical)	bypassed Hyper Register		param_mux[0~85]_param_muxA0~0@pcat
36. Long Path (Critical)	bypassed Hyper Register		param_mux[0~85]_param_muxA0~0@pcat
37. Retiming Restriction	REG (required)	#2	dout_reg[33][0]

The following table shows the correlation between critical chain elements and the Technology Map Viewer examples.

Table 7. Correlation Between Critical Chain Elements and Technology Map Viewer

Line Numbers in Critical Chain Report	Circuit Element in the Technology Map Viewer
1-3	din_reg[33][0] source register and its output
4-14	FPGA routing fabric between din_reg[33][0] and Mux0~85, the first stage of mux in the crossbar
15-17	Combinational logic implementing Mux0~85
18-20	Routing between Mux0~85 and Mux0~17, the second stage of mux in the crossbar
<i>continued...</i>	



Line Numbers in Critical Chain Report	Circuit Element in the Technology Map Viewer
21-23	Combinational logic implementing Mux0~17
24-27	Routing between Mux0~17 and Mux0~33, the third stage of mux in the crossbar
28-29	Combinational logic implementing Mux0~17
30-33	Routing between Mux 0~33 and Mux0~68, the fourth stage of mux in the crossbar
34-35	Combinational logic implementing Mux0~68
36-37	dout_reg[76][0] destination register

In the critical chain report in [Figure 82](#) on page 78, there are 17 lines that list bypass Hyper-Register in the **Register** column. Bypassed Hyper-Register indicates the location of a Hyper-Register for use if there are more registers in the chain, or if there are no restrictions on the endpoints. If there are no restrictions on the endpoints, the Compiler can retime the endpoint registers, or retime other registers from outside the critical chain into the critical chain. If the RTL design contains more registers through the crossbar switch, there are more registers that the Compiler can retime. Fast Forward compilation can also insert more registers to increase the performance.

In the critical chain report, lines 2 to 36 list "Long Path (Critical)" in the **Path Info** column. This indicates that the path is too long to run above the listed frequency. The "Long Path" designation is also related to the Short Path/Long Path type of critical chain. Refer to the *Short Path/Long Path* section for more details. The (Critical) designation exists on one register-to-register segment of a critical chain. The (Critical) designation indicates that the register-to-register path is the most critical timing path in the clock domain.

The **Join** column contains a "#1" on line 1, and a "#2" on line 29. The information in the **Join** column helps interpret more complex critical chains. For more details, refer to *Complex Critical Chains* section.

The **Element** column shows the name of the circuit element or routing resource at each step in the critical chain. You can right-click the names to copy them, or cross probe to other parts of the Intel Quartus Prime software with the **Locate** option, as shown in the following figure.

Related Links

- [Short Path/Long Path](#) on page 80
- [Complex Critical Chains](#) on page 89
- [Hyper-Retiming \(Facilitate Register Movement\)](#) on page 11

5.1.1.2 Optimizing Insufficient Registers

Use the Hyper-Pipelining techniques that this documents describes to resolve critical chains limited by reported insufficient registers.

Related Links

- [Hyper-Retiming \(Facilitate Register Movement\)](#) on page 11
- [Hyper-Pipelining \(Add Pipeline Registers\)](#) on page 26



5.1.1.3 Critical Chains with Dual Clock Memories

Hyper-Retiming does not retime registers through dual clock memories. Therefore, the Compiler can report a functional block between two dual clock FIFOs or memories, as the critical chain. The report specifies a limiting reason of Insufficient Registers, even after Fast Forward compile.

If the limiting reason is Insufficient Registers, and the chain is between dual clock memories, you can add pipeline stages to the functional block. Alternatively, add a bank of registers in the RTL, and then allow the Compiler to balance the registers. Refer to the *Pipeline Stages* section for a technique to introduce registers in that critical chain with a software setting.

A functional block between two single-clock FIFOs is not affected by this behavior, because the FIFO memories are single-clock. The Compiler can retime registers across a single-clock memory. Additionally, a functional block between a dual-clock FIFO and registered device I/Os is not affected by this behavior, because the Fast Forward Compile can pull registers into the functional block through the registers at the device I/Os.

Related Links

[Appendix A: Parameterizable Pipeline Modules](#) on page 109

5.1.2 Short Path/Long Path

When the critical chain has related paths with conflicting characteristics, where one path can improve performance with more registers, and another path has no place for additional registers, the limiting reason reported is Short Path/Long Path.

A critical chain is categorized as short path/long path when there are conflicting optimization goals for Hyper-Retiming. Short paths and long paths are always connected in some way, with at least one common node. Retimed registers must maintain functional correctness and ensure identical relative latency through both critical chains. This requirement can result in conflicting optimization goals. Therefore, one segment (the long path) can accept the retiming move, but the other segment (the short path) cannot accept the retiming move. The retiming move is typically retiming an additional register into the short and long paths.

Critical chains are categorized as short path/long path for the following reasons:

- When Hyper-Register locations are not available on the short path to retime into.
- When retiming a register into both paths to improve the performance of the long path does not meet hold time requirement on the short path. Sometimes, short path/long path critical chains exist as a result of the circuit structures used in a design, such as broadcast control signals, synchronous clears, and clock enables.

Short path/long path critical chains are a new optimization focus associated with post-fit retiming. In conventional retiming, the structure of the netlist can be changed during synthesis or placement and routing. However, during Hyper-Retiming, short path/long path can occur because the netlist structure, and the placement and routing cannot be changed.



5.1.2.1 Hyper-Register Locations Not Available

The Fitter may place the elements in a critical chain segment very close together, or route them such that there are no Hyper-Register locations available. When all Hyper-Register locations in a critical chain segment are in use, there are no more locations available for further optimization.

In the following example, the short path includes two Hyper-Register locations that are in use. One or more names in the **Element** column end in _dff, indicating that the Hyper-Registers in those locations are in use. The _dff represents the D flop-flop. No other Hyper-Register locations are available for use in that chain segment. Available Hyper-Register locations indicate status with a bypassed Hyper-Register entry in the **Register** column.

Figure 83. Critical Chain Short Path Segment with no Available Hyper-Register Locations

Path Info	Register	Join	Element
1 Short Path	REG (required)	#1	round_robin_mod_requests_r[2]
2 Short Path			round_robin_mod_requests_r[2]j/q
3 Short Path	unusable (hold)		rmm mux4~0 state
4 Short Path			rmm mux4~0 combout
5 Short Path			round_robin_modulo rmm mux4~0 a_mlabtabout[13]
6 Short Path			round_robin_modulo rmm mux4~0 _lab_re_x76_y198_no_103
7 Short Path	REG	#2	round_robin_modulo rmm mux4~0 _lo....INTERCONNECT_X75_Y198_NO_13_dff
8	*****	***	*****

5.1.2.2 Example for Hold Optimization

On line 3 in the following example, the **Register** column indicates unusable (hold). There is a Hyper-Register location available at the datae LUT input for the rmm|_mux4~0 combinational node, as line 3 indicates. However, you cannot use this register location because the location does not meet hold time requirements as, line 3 indicates. The Compiler cannot retime the register on line 1 forward, and the Compiler cannot retime the register on line 7 backward.

Figure 84. Critical Chain with Short Path/Long Path

Critical Chain at Fast Forward Limit			
Optimizations Analyzed (Cumulative)		Recommendations for Critical Chain	
Path Info	Register	Join	Element
1 Short Path	REG	#1	round_robin_mod_last_d[1]-C4_X73_Y130_NO_16_dff
2 Short Path			round_robin_mod_last_d[1]-C4_X73_Y130_NO_16
3 Short Path	unusable (hold)		round_robin_mod_last_d[1]-LOCAL_INTERCONNECT_X74_Y133_NO_151
4 Short Path			round_robin_mod_last_d[1]-LAB_RE_X74_Y133_NO_19
5 Short Path			round_robin_modulo rmm next[1]-3-_datae_dff
6 Short Path	REG		round_robin_modulo rmm next[1]-3-_datae_dff
7 Short Path			rmm next[1]-3 datae
8 Short Path			rmm next[1]-3 combout
9 Short Path			rmm next[1]-3 datae
10 Short Path	REG (required)	#2	round_robin_mod_next[1]-reg0d
11	*****	***	*****
12 Long Path (Critical)	REG	#1	round_robin_mod_last_d[1]-C4_X73_Y130_NO_16_dff
13 Long Path (Critical)			round_robin_mod_last_d[1]-C4_X73_Y130_NO_16
14 Long Path (Critical)	bypassed Hyper Register		round_robin_mod_last_d[1]-LOCAL_INTERCONNECT_X74_Y132_NO_140
15 Long Path (Critical)			round_robin_mod_last_d[1]-LAB_RE_X74_Y132_NO_19
16 Long Path (Critical)	bypassed Hyper Register		rmm Mod3 auto_generated divider divider op_3-5 state
17 Long Path (Critical)			rmm Mod3 auto_generated divider divider op_3-9 cout

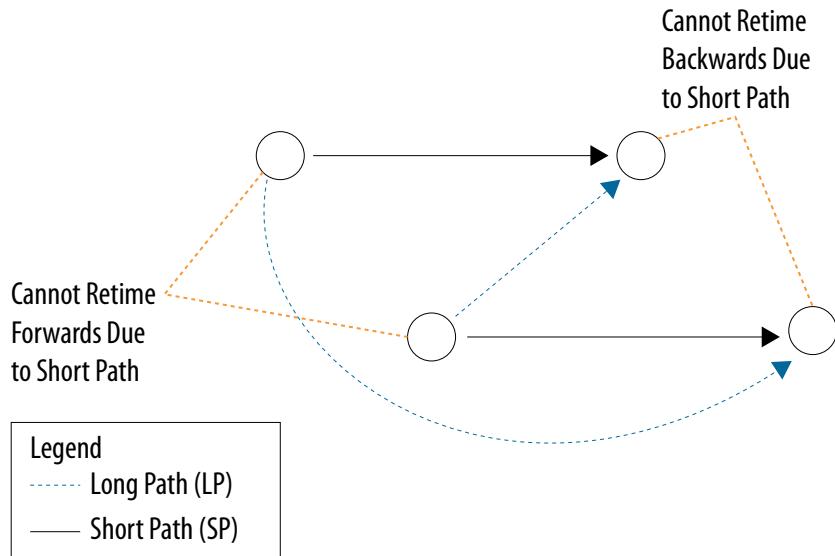
5.1.2.3 Optimizing Short Path/Long Path

Evaluate the Fast Forward recommendations to optimize performance limitations due to short path/long path constraints.

5.1.2.4 Add Registers

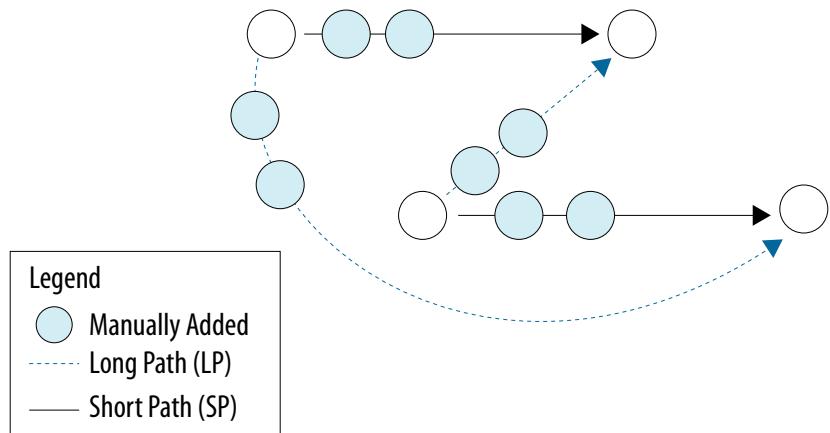
Manually adding registers on both the short and long paths can be helpful if you can accommodate the extra latency in the critical chain.

Figure 85. Critical Chain with Alternating Short Path/Long Path



If you add registers to the four chain segments, the Compiler can optimize the critical chain. When additional registers are available in the RTL, the Compiler can optimize their positions.

Figure 86. Sample Short Path/Long Path with Additional Latency



5.1.2.5 Duplicate Common Nodes

When the short path/long path critical chain contains common segments originating from same register, you can duplicate the register so one duplicate feeds the short path and one duplicate feeds the long path.

Figure 87. Critical Chain with Alternating Short Path/Long Path

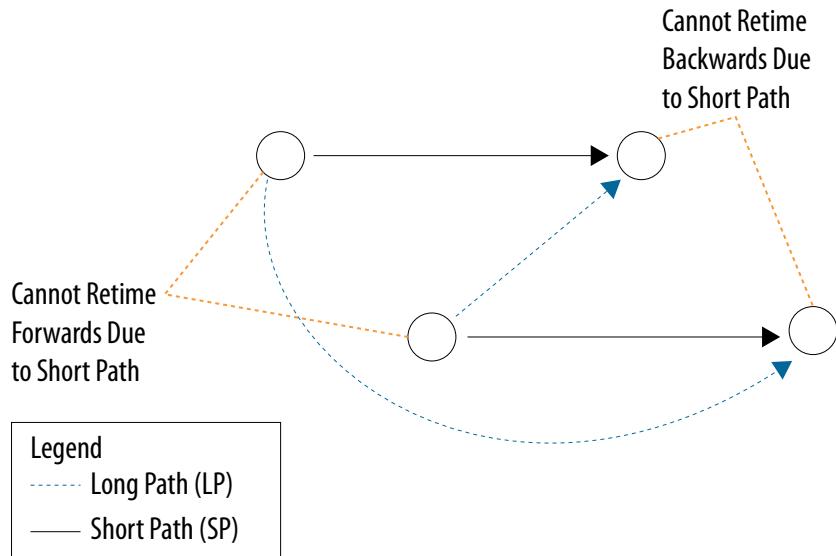
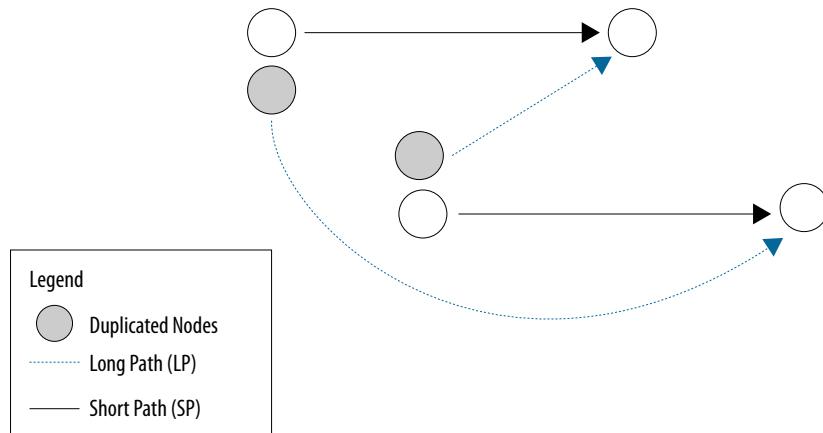


Figure 88. Short Path/Long Path with Two Duplicate Nodes



The Fitter can optimize the newly-independent segments separately. The duplicated registers have common sources themselves, so they are not completely independent, but the optimization is easier with an extra, independent register in each part of the critical chain.

You can apply a maximum fan-out synthesis directive to the common source registers. Use a value of one, because a value greater than one can result in the short and long path segments having the same source node, which you tried to avoid.

Alternately, use a synthesis directive to preserve the duplicate registers if you manually duplicate the common source register in a short path/long path critical chain. Otherwise, the duplicates may get merged during synthesis. Using a synthesis directive to preserve the duplicate registers can cause an unintended retiming restriction. Use a maximum fan-out directive.

5.1.2.6 Data and Control Plane

Sometimes, the long path can be in the data plane, and the short path can be in the control plane. If you add registers to the data path, change the control logic. This can be a time-consuming process. In cases where the control logic is based on the number of clock cycles in the data path, you can add registers in the data path (the long path) and modify a counter value in the control logic (the short path) to accommodate the increased number of cycles used to process the data.

5.1.3 Fast Forward Limit

The critical chain has the limiting reason of Path Limit, when there are no more Hyper-Register locations available on the critical path, and the design cannot run any faster or implement further retiming. Path Limit also indicates reaching a performance limit of the current place and route result.

The following figure shows the critical chain changing, from a hard memory block, to the first Hyper-Register available outside the hard memory block. Parts of the names in lines 1, 2, and 3 indicate that the source is a hard memory block. Lines 1 and 2 refer to ram_block1a0. Line 3 contains a reference to MEDIUM_EAB_RE, which denotes a medium embedded array block routing element. The medium embedded array block is one of the hard memory block in Intel Stratix 10 devices.

Figure 89. Critical Chain at Fast Forward Limit

Critical Chain at Fast Forward Limit			
Optimizations Analyzed (Cumulative)		Recommendations for Critical Chain	
	Path Info	Register	Join
1	Long Path	REG (required)	#1
2	Long Path		ROUND[0] U_ROUND[U_SUB ROM 2] ROM...ated ram_block1a0 portadataout[0]
3	Long Path		Round:ROUND[0] U_ROUND SubBytes...0~MEDIUM_EAB_RE_X70_Y121_N0_I92
4	Long Path	REG	#2 Round:ROUND[0] U_ROUND SubBytes:..._block1a0~R6_X64_Y121_N0_I59_dff

The **Path Info** column displays the information when the critical chain is a Path Limit. This column indicates that the chain is too long, but can you can improve performance by retiming a register into the chain. If the report lists no entries for bypassed Hyper-Register in the **Register** column, this absence indicates that there are no Hyper-Register locations available.

Path Limit does not imply that the critical chain reaches the inherent silicon performance limit. Path Limit indicates that the current place and route result reaches a performance limit. Another compilation can result in a different placement that allows Hyper-Retiming to achieve better performance on the particular critical chain. Typically, path limit occurs when registers do not pack into dedicated input or output registers in a hard DSP or RAM block.

5.1.3.1 Optimizing Path Limit

Evaluate the Fast Forward recommendations. If your critical chain has a limiting reason of Path Limit, and the chain is entirely in the core logic and in the routing elements of the Intel FPGA fabric, the design can run at the maximum performance of the core fabric. When the critical chain has a limiting reason of Path limit, and chain is through a DSP block or hard memory block, you can improve performance by optimizing the path limit.

To optimize path limit, enable the optional input and output registers for DSP blocks and hard memory blocks. If you do not use the optional input and output registers for DSP blocks and memory blocks, the locations for the optional registers are not available for Hyper-Retiming, and do not appear as bypassed Hyper-Registers in the critical chain. The path limit is the silicon limit of the path, without the optional input or output registers. You can improve the performance by enabling optional input and output registers.

Turn on optional registers using the IP parameter editor to parameterize hard DSP or memory blocks. If you infer DSP or memory functions from your RTL, ensure that you follow the *Recommended HDL Coding Styles* to ensure that you use the optional input and output registers of the hard blocks. The Compiler does not retime into or out of DSP and hard memory block registers. Instantiate the optional registers to achieve maximum performance.

If your critical chain includes true dual port memory, refer to *True Dual-Port Memory* for optimizing techniques.

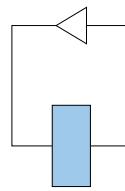
Related Links

- [Recommended HDL Coding Styles](#)
- [True Dual-Port Memory](#) on page 52

5.1.4 Loops

A loop is a feedback path in a circuit. When a circuit is heavily pipelined, loops are often a limiting reason to increasing design f_{MAX} through register retiming. A loop may be very short, containing only a single register or much longer, containing dozens of registers and combinational logic clouds. A register in a divide-by-two configuration is a short loop.

Figure 90. Simple Loop

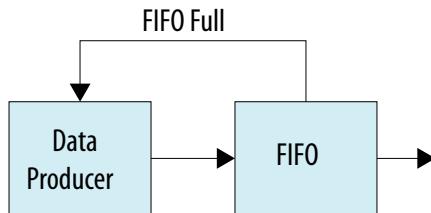


Toggle
FlipFlop

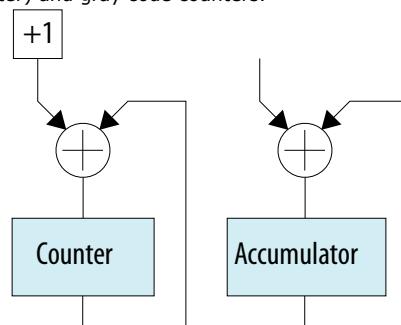
When the critical chain is a feedback loop, register retiming cannot change the number of registers in a loop without changing functionality. Registers can retime around a loop without changing functionality, but adding registers to the loop changes functionality. To explore performance gains, the Fast Forward Compile process adds registers at particular boundaries of the circuit, such as clock domain boundaries.

Figure 91. FIFO Flow Control Loop

In a FIFO flow control loop, upstream processing stops when the FIFO is full, and downstream processing stops when the FIFO is empty.


Figure 92. Counter and Accumulator Loop

In a counter and accumulator loop, a register's new value depends on the old value. This includes variants like LFSRs (linear feedback shift register) and gray code counters.


Figure 93. State Machine Loop

In a state machine loop, the next state depends on the current state of the circuit.

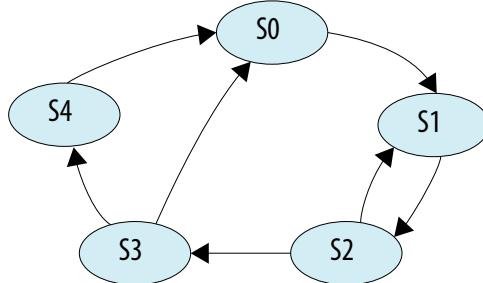
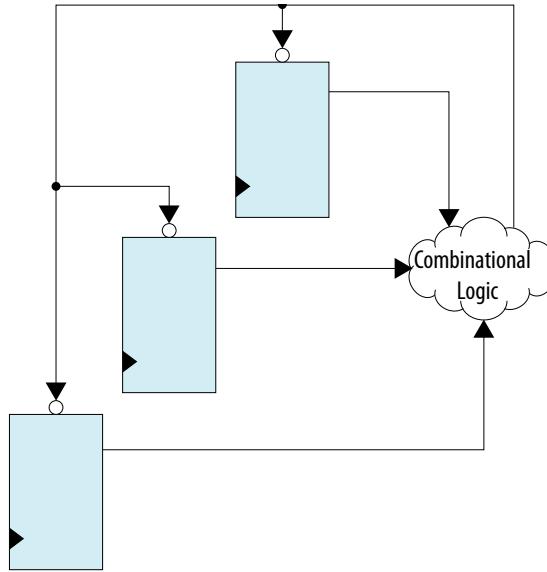


Figure 94. Reset Circuit Loop

Reset circuit loops include monitoring logic to reset on an error condition.



Use loops to save area through hardware re-use. Components that you re-use over several cycles typically involve loops. Such components include CRC calculations, filters, floating point dividers, and word aligners. Closed loop feedback designs, such as IIR filters and automatic gain control for transmitter power in remote radiohead designs, also use loops.

5.1.4.1 Example of Loops Limiting the Critical Chain

The following screenshots show the relevant panels from the Fast Forward Details report and the logic contained in the critical chain.

Figure 95. Fast Forward Details Report showing Limiting Reason for Hyper-Optimization is a Loop

Fast Forward Details for Clock Domain clk_125					
Fast Forward Summary for Clock Domain clk_125		Fast Forward Optimizations Analyzed	To Achieve Fmax	Stack	Relationship
1	Base Performance	None	284 MHz	4,480	8,000
2	Fast Forward Step #1 (Hyper-Retiming)	Removed asynchronous clears on 197 Registers Removed user preserve assignments from 51 Registers	426 MHz	5,651	8,000
3	Fast Forward Step #2 (Hyper-Pipelining)	Removed asynchronous clears on 93 Registers Added up to 1 pipeline stage in 131 Paths	454 MHz	5,795	8,000
4	Fast Forward Limit	Performance Limited by: RTL loop	-	-	-

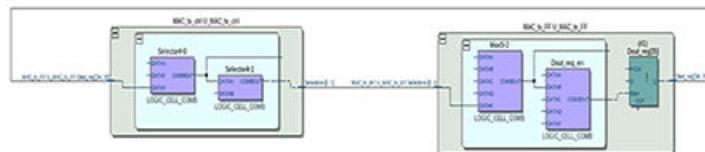
In the following figure, the Join ID for the start and end points is the same, which is #1. This case indicates that the start and end points of the chain are the same, thereby creating a loop.

Figure 96. Critical Chain with Loop as Reported During Hyper-Retiming

Limiting Critical Chain Before Hyper-Optimization			
Path Info	Register	Join	Element
1. Long Path (Critical)	REG	#1	MAC_txU_MAC_tx_FF_MAC_tx_FF Out_Reg[25]
2. Long Path (Critical)			U_MAC_txU_MAC_tx_FF Out_Reg[25]#0
3. Long Path (Critical)			MAC_txU_MAC_tx_FF_MAC_tx_FF Out_Reg[25]-la_lab/about[10]
4. Long Path (Critical)			MAC_txU_MAC_tx_FF_MAC_tx_FF Out_Reg[25]-LAB_Re_X136_Y10_N0_I120
5. Long Path (Critical)	bypassed Hyper-Register		MAC_txU_MAC_tx_FF_MAC_tx_FF Out_Reg[25]-R3_X134_Y09_N0_I20
6. Long Path (Critical)	bypassed Hyper-Register		MAC_txU_MAC_tx_FF_MAC_tx_FF Out_Reg[25]-C4_X135_Y09_N0_I25
7. Long Path (Critical)	bypassed Hyper-Register		MAC_txU_MAC_tx_FF_MAC_tx_L_INTERCONNECT_X136_Y101_N0_I31
8. Long Path (Critical)	bypassed Hyper-Register		MAC_txU_MAC_tx_FF_MAC_tx_FF Out_Reg[25]-LAB_Re_X136_Y101_N0_I23
9. Long Path (Critical)	bypassed Hyper-Register		U_MAC_txU_MAC_tx_c0 Select04-0 data
10. Long Path (Critical)			U_MAC_txU_MAC_tx_c0 Select04-0 combout
11. Long Path (Critical)			MAC_txU_MAC_tx_c0 MAC_tx_c0 Select04-0-la_lab/about[11]
12. Long Path (Critical)			MAC_txU_MAC_tx_c0 MAC_tx_c0 LAB_Re_X136_Y101_N0_I101
13. Long Path (Critical)	bypassed Hyper-Register		MAC_txU_MAC_tx_c0 MAC_tx_c0 Select04-0-C4_X135_Y102_N0_I26
14. Long Path (Critical)	bypassed Hyper-Register		MAC_txU_MAC_tx_c0 MAC_tx_c0 Select04-0-R3_X136_Y102_N0_I17
15. Long Path (Critical)	bypassed Hyper-Register		MAC_txU_MAC_tx_c0 MAC_tx_c0 U_INTERCONNECT_X136_Y102_N0_I37
16. Long Path (Critical)	bypassed Hyper-Register		MAC_txU_MAC_tx_c0 MAC_tx_c0 Select04-0-LAB_Re_X136_Y102_N0_I10
17. Long Path (Critical)			U_MAC_txU_MAC_tx_c0 Select04-1 data
18. Long Path (Critical)			U_MAC_txU_MAC_tx_c0 Select04-1 combout
19. Long Path (Critical)			MAC_txU_MAC_tx_c0 MAC_tx_c0 Select04-1-la_lab/about[7]
20. Long Path (Critical)			MAC_txU_MAC_tx_c0 MAC_tx_c0 c0 data[1-1 LAB_Re_X136_Y102_N0_I97
21. Long Path (Critical)	bypassed Hyper-Register		MAC_txU_MAC_tx_c0 MAC_tx_c0 MAC_tx_c0 reselect04-1-C4_X135_Y09_N0_I35
22. Long Path (Critical)	bypassed Hyper-Register		MAC_txU_MAC_tx_c0 MAC_tx_c0 MAC_tx_c0 reselect04-1-R3_X136_Y101_N0_I39
23. Long Path (Critical)			MAC_txU_MAC_tx_c0 MAC_tx_c0 U_INTERCONNECT_X136_Y101_N0_I39
24. Long Path (Critical)	bypassed Hyper-Register		MAC_txU_MAC_tx_c0 MAC_tx_c0 Select04-1-LAB_Re_X136_Y101_N0_I48
25. Long Path (Critical)			U_MAC_txU_MAC_tx_c0 FF data[2 data
26. Long Path (Critical)			U_MAC_txU_MAC_tx_c0 FF data[2 combout
27. Long Path (Critical)			MAC_txU_MAC_tx_c0 MAC_tx_c0 FF data[2-la_lab/about[4]
28. Long Path (Critical)			MAC_txU_MAC_tx_c0 MAC_tx_c0 FF data[2-LAB_Re_X136_Y101_N0_I14
29. Long Path (Critical)	bypassed Hyper-Register		MAC_txU_MAC_tx_c0 MAC_tx_c0 FF data[2-R3_X134_Y101_N0_I15
30. Long Path (Critical)	bypassed Hyper-Register		MAC_txU_MAC_tx_c0 MAC_tx_c0 U_INTERCONNECT_X136_Y101_N0_I47
31. Long Path (Critical)	bypassed Hyper-Register		MAC_txU_MAC_tx_c0 MAC_tx_c0 FF data[2-LAB_Re_X136_Y101_N0_I35
			U_MAC_txU_MAC_tx_c0 FF data[2 data

Figure 97. Critical Chain in Technology Map Viewer

The output of the RetryCnt[0] register feeds back to its enable input through two levels of combinational logic. The figure does not show the other inputs to the logic cone for the RetryCnt[0] register, but the following source code shows portions of the MAC_tx_ctrl source, and some inputs to the RetryCnt registers.



Example 15. Source Code for Critical Chain

```

StateJam:
if (RetryCnt<=MaxRetry&&JamCounter==16)
    Next_state=StateBackOff;
else if (RetryCnt>MaxRetry)
    Next_state=StateJamDrop;
else
    Next_state=Current_state;

always @ (posedge Clk or posedge Reset)
    if (Reset)
        JamCounter      <=0;
    else if (Current_state!=StateJam)
        JamCounter      <=0;
    else if (Current_state==StateJam)
        JamCounter      <=JamCounter +1;

always @ (posedge Clk or posedge Reset)
    if (Reset)
        RetryCnt      <=0;
    else if (Current_state==StateSwitchNext)
        RetryCnt      <=0;
    else if (Current_state==StateJam&&Next_state==StateBackOff)
        RetryCnt      <=RetryCnt +1;

```



5.1.5 One Critical Chain per Clock Domain

Hyper-Retiming reports one critical chain per clock domain, except in a special case that *Critical Chains in Related Clock Groups* describes. If you perform a Fast Forward compile, Hyper-Retiming reports show one critical chain per clock domain per Fast Forward optimization step. Hyper-Retiming does not report multiple critical chains per clock domain, because only one chain is the critical chain.

Review other chains in your design for potential optimization. View other chains in each step of the Fast Forward compilation report. Each step in the report tests a set of changes, such as removing or converting asynchronous clears, and adding pipeline stages. The reports detail the performance, assuming implementation of those changes.

Related Links

[Critical Chains in Related Clock Groups](#) on page 89

5.1.6 Critical Chains in Related Clock Groups

When two or more clock domains have the exact same timing requirement, and there are paths between the domains, and the registers on the clock domain boundaries do not have a Don't Touch attribute, the Hyper-Retiming reports a critical chain for a Related Clock Group. The optimization techniques critical chain types also apply to critical chains in related clock groups.

5.1.7 Complex Critical Chains

Complex critical chains consist of several segments connected with multiple join points. A join point is indicated with a positive integer in the Join column in the Fitter reports. Join points are listed at the ends of segments in a critical chain, and they indicate where segments diverge or converge. Join points indicate connectivity between chain segments when the chain is listed in a line-oriented text-based report. Join points correspond to elements in your circuit, and show how they are connected to other elements to form a critical chain.

The following example shows how join points correspond to circuit connectivity, using the sample critical chain in the following table.

Table 8. Sample Critical Chain

Path Info	Register	Join	Element
	REG	#1	a
			b
	REG	#2	c
-----	-----	-----	-----
	REG	#3	d
			e
	REG	#2	c
-----	-----	-----	-----
	REG	#3	d

continued...

Path Info	Register	Join	Element
			f
	REG	#4	g
-----	-----	-----	-----
			g
			h
			a

Figure 98. Visual Representation of Sample Critical Chain

Each circle in the diagram contains the element name and the join point number from the critical chain table.

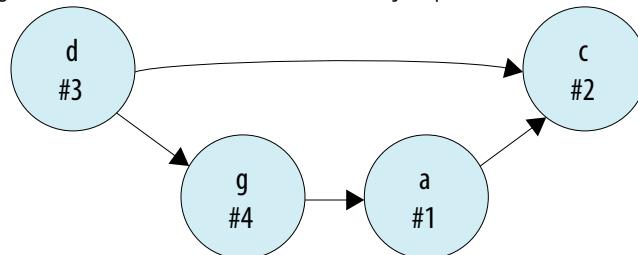


Figure 99. Complex Critical Chain

A critical chain can include dozens of join points. The complex critical chain shown below has 35 join points.

601	Long Path	empty slot	#4	Viteration_g_1:iteration_i sova_i1 trellis2_i0 reg[4]
602	-----	-----	---	-----
603		REG	#5	Iteration:Viteration_g_1:iteration_i...CAL_INTERCOL
604				Iteration:Viteration_g_1:iteration_i...8_LOCAL_INTE
605				Iteration:Viteration_g_1:iteration_i ...freePathId=7
606		empty slot	#4	Viteration_g_1:iteration_i sova_i1 trellis2_i0 reg[4]
607	-----	-----	---	-----

1504	Long Path	empty slot	#10	Viteration_g_1:iteration_i sova_i1 trellis2_i0 freePa
1505	-----	-----	---	-----
1506		empty slot	#11	Iteration:Viteration_g_1:iteration_i...0_LOCAL_INTE
1507				Iteration:Viteration_g_1:iteration_i ...ls2_i0 Mux20:
1508		empty slot	#10	Viteration_g_1:iteration_i sova_i1 trellis2_i0 freePa
1509	-----	-----	---	-----
1510		empty slot	#11	Iteration:Viteration_g_1:iteration_i...0_LOCAL_INTE

5084	Short Path			Viteration_g_1:iteration_i sova_i1 trellis2_i0 revWe
5085	Short Path	REG (required)	#33	Iteration:Viteration_g_1:iteration_i sova...1 trellis2
5086	-----	-----	---	-----
5087	Short Path	REG	#34	Iteration:Viteration_g_1:iteration_i sova...2:trellis2
5088	Short Path			Viteration_g_1:Iteration_i sova_i1 trellis2_i0 revWe
5089	Short Path	REG (required)	#33	Iteration:Viteration_g_1:iteration_i sova...1 trellis2
5090	-----	-----	---	-----
5091	Short Path	REG	#34	Iteration:Viteration_g_1:iteration_i sova...2:trellis2,
5092	Domain Boundary Exit		#35	

For long critical chains, identify smaller parts of the critical chain for optimization. Recompile the design and analyze the changes in the critical chain. Refer to *Optimizing Loops* for other approaches to focus your optimization effort on part of a critical chain.

5.1.8 Extend to locatable node

You may see a path info entry of "Extend to locatable node" in a critical chain. This is a convenience feature to allow you to correlate nodes in the critical chain to design names in your RTL.

Not every line in a critical chain report corresponds to a design entry name in an RTL file. For example, individual routing wires have no correlation with names in your RTL. Typically that is not a problem, because another name on a nearby or adjacent line corresponds with, and is locatable to, a name in an RTL file. Sometimes a line in a critical chain report may not have an adjacent or nearby line that you can locate in an RTL file. This condition occurs most frequently with join points. When this condition occurs, the critical chain segment extends as necessary until the critical chain reaches a line that you can be located in an HDL file.

5.1.9 Domain Boundary Entry and Domain Boundary Exit

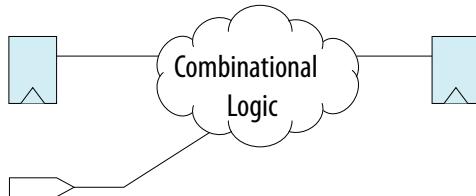
The **Path Info** column lists the Domain Boundary Entry or Domain Boundary Exit for a critical chain. Domain boundary entry and domain boundary exit refer to paths that are unconstrained, paths between asynchronous clock domains, or between a clock domain and top-level device input-outputs. Domain boundary entry and exit can also be indicated for some false paths as well.

A domain boundary entry refers to a point in the design topology, at a clock domain boundary, where Hyper-Retiming can insert register stages (where latency can enter the clock domain) if Hyper-Pipelining is enabled. The concept of a domain boundary entry is independent of the dataflow direction. Hyper-Retiming can insert register stages at the input of a module, and perform forward retiming pushes. Hyper-Retiming can also insert register stages at the output of a module, and perform backward retiming pushes. These insertions occur at domain boundary entry points.

A domain boundary exit refers to a point in the design topology, at a clock domain boundary, where Hyper-Retiming can remove register stages and the latency can exit the clock domain, if Hyper-Pipelining is enabled. Removing a register seems counter intuitive. However, this method is often necessary to retain functional correctness, depending on other optimizations that Hyper-Retiming performs.

Sometimes a critical chain indicates a domain boundary entry or exit when there is an unregistered I/O feeding combinational logic on a register-to-register path as shown in the following figure.

Figure 100. Domain Boundary with Unregistered Input/Output



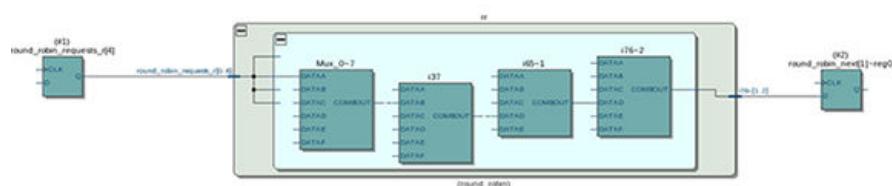
The register-to-register path might be shown as a critical chain segment with a domain boundary entry or a domain boundary exit, depending on how the path restricts Hyper-Retiming. The unregistered input prevents the Hyper-Retiming from inserting register stages at the domain boundary, because the input is unregistered. Likewise, the unregistered input can also prevent Hyper-Retiming from removing register stages at the domain boundary.

Critical chains with a domain boundary exit do not provide complete information for you to determine what prevents retiming a register out of the clock domain. To determine why a register cannot retime, review the design to identify the signals that connect to the other side of a register associated with a domain boundary exit.

Domain boundary entry and domain boundary exit can appear independently in critical chains. They can also appear in combination such as, a domain boundary exit without a domain boundary entry, or a domain boundary entry at the beginning and end of a critical chain.

The following critical chain begins and ends with domain boundary entry. The domain boundary entries are the input and output registers connecting to top-level device I/Os. The input register is `round_robin_mod_last_r` and the output register is `round_robin_mod_next`.

Figure 101. Critical Chain Schematic with Domain Boundary



The limiting reason for the base compile is Insufficient Registers.

Figure 102. Fast Forward Compile Report with Insufficient Registers

Fast Forward Details for Clock Domain clock				
Fast Forward Summary for Clock Domain clock		Fast Forward Limit Critical Chain Schematic		
Step	Fast Forward Optimizations Analyzed	Estimated Fmax	Slack	Relationship
1 Base Performance	None	82 MHz	-2.262	10.000
2 Fast Forward Limit	Performance Limited by: Insufficient Registers	--	--	--

The following parts of the critical chain report show that the endpoints are labeled with Domain Boundary Entry.



Figure 103. Critical Chain with Domain Boundary Entry

Critical Chain Summary for Clock Domain clk			
Critical Chain Details		Recommendations for Critical Chain	
Path Info	Register	Join	Element
1. Domain Boundary Entry	REG (required)	#1	round_robin_requests_i[4]
2. Long Path (Critical)			round_robin_requests_i[4]@t
3. Long Path (Critical)			round_robin_requests_i[4]-LAB.RE_X225_Y48_N0_I109
4. Long Path (Critical)	bypassed Hyper-Register		round_robin_requests_i[4]-C4_X224_Y48_N0_I10
5. Long Path (Critical)	bypassed Hyper-Register		round_robin_requests_i[4]-C4_X224_Y52_N0_I50
6. Long Path (Critical)	bypassed Hyper-Register		round_robin_requests_i[4]-C4_X224_Y56_N0_I10
7. Long Path (Critical)	bypassed Hyper-Register		round_robin_requests_i[4]-C4_X224_Y61_N0_I10
8. Long Path (Critical)	bypassed Hyper-Register		round_robin_requests_i[4]-LOCAL_INTERCONNECT_X225_Y63_N0_I14
9. Long Path (Critical)	bypassed Hyper-Register		round_robin_requests_i[4]-LAB.RE_X225_Y63_N0_I51
10. Long Path (Critical)	bypassed Hyper-Register		m937jdataa
11. Long Path (Critical)			m937jcombout
12. Long Path (Critical)			m937j- LAB.RE_X225_Y63_N0_I119
13. Long Path (Critical)	bypassed Hyper-Register		m937j-CA_X225_Y63_N0_I17
14. Long Path (Critical)	bypassed Hyper-Register		m937j- LOCAL_INTERCONNECT_X225_Y65_N0_I50
15. Long Path (Critical)	bypassed Hyper-Register		m937j- LAB.RE_X225_Y65_N0_I76
16. Long Path (Critical)			m937jdatab
17. Long Path (Critical)			m937jcombout
18. Long Path (Critical)			m937-LAB.RE_X225_Y65_N0_I130
19. Long Path (Critical)	bypassed Hyper-Register		m937- LOCAL_INTERCONNECT_X225_Y65_N0_I28
20. Long Path (Critical)	bypassed Hyper-Register		m937-LAB.RE_X225_Y65_N0_I49
21. Long Path (Critical)			m955-1jdatad
22. Long Path (Critical)			m955-1jcombout
23. Long Path (Critical)			m955-1- LAB.RE_X225_Y65_N0_I119
24. Long Path (Critical)	bypassed Hyper-Register		m955-1- LOCAL_INTERCONNECT_X225_Y65_N0_I49
25. Long Path (Critical)	bypassed Hyper-Register		m955-1- LAB.RE_X225_Y65_N0_I9
26. Long Path (Critical)			m976-2jdatad
27. Long Path (Critical)			m976-2jcombout
28. Long Path (Critical)			round_robin_next[1]-reg0jd
29. Domain Boundary Entry	REG (required)	#2	round_robin_next[1]-reg0

Both the input and output registers are indicated as Domain Boundary Entry because the Fast Forward Compile often inserts register stages at these boundaries if Hyper-Pipelining were enabled. Because the critical chain for the base compile does not contain any Fast Forward optimizations, no additional register stages were inserted at either the input to the chain, or the output of the chain.

A similar path in the same circuit has an endpoint indicated as Domain Boundary Exit in a critical chain reported after two steps of Fast Forward optimization. The following screenshot shows that the limiting reason for Fast Forward Step #2 is Short path/Long path.

Figure 104. Fast Forward Compile Report with Short Path/Long Path

Fast Forward Details for Clock Domain clk_mod					
Fast Forward Summary for Clock Domain clk_mod					
Step	Fast Forward Optimizations Analyzed	To Achieve Fmax	Slack	Relationship	
1. Base Performance	None	258 MHz	1.121	5.000	
2. Fast Forward Step #2 (Hyper-Pipelining)	Added up to 1 pipeline stage in 10 Paths	468 MHz	2.865	5.000	
3. Fast Forward Step #2 (Hyper-Pipelining)	Added up to 1 pipeline stage in 10 Paths	708 MHz	3.587	5.000	
4. Fast Forward Step #3 (Hyper-Pipelining)	Added up to 1 pipeline stage in 10 Paths	861 MHz	3.839	5.000	
5. Fast Forward Step #4 (Hyper-Pipelining)	Added up to 1 pipeline stage in 6 Paths	962 MHz	3.960	5.000	
6. Fast Forward Limit	Performance Limited by Short Path/Long Path	--	--	--	

5.1.10 Critical Chains with Dual Clock Memories

Hyper-Retiming does not retime registers through dual clock memories. Therefore, the Compiler can report a functional block between two dual clock FIFOs or memories, as the critical chain. The report specifies a limiting reason of Insufficient Registers, even after Fast Forward compile.

If the limiting reason is Insufficient Registers, and the chain is between dual clock memories, you can add pipeline stages to the functional block. Alternatively, add a bank of registers in the RTL, and then allow the Compiler to balance the registers. Refer to the *Pipeline Stages* section for a technique to introduce registers in that critical chain with a software setting.

A functional block between two single-clock FIFOs is not affected by this behavior, because the FIFO memories are single-clock. The Compiler can retime registers across a single-clock memory. Additionally, a functional block between a dual-clock FIFO and registered device I/Os is not affected by this behavior, because the Fast Forward Compile can pull registers into the functional block through the registers at the device I/Os.

Related Links

[Appendix A: Parameterizable Pipeline Modules](#) on page 109

5.1.11 Critical Chain Bits and Buses

The critical chain of a design commonly includes registers that are single bits in a wider bus or register bank. When you analyze such a critical chain, focus on the bus as a whole, instead of analyzing the structure related to the single bit. For example, a critical chain that refers to bit 10 in a 512 bit bus probably corresponds to similar structures for all the bits in the bus. A technique that can help with this approach is to mentally replace each bit index, such as [10], with [*].

If the critical chain includes a register in a bus where different slices go through different logic, then focus your analysis on the appropriate slice based on which register is reported in the critical chain.

5.1.12 Delay Lines

You may have a module that delays a bus by some number of clock cycles. The Compiler often converts this type of structure to the altshift_taps Intel FPGA IP during synthesis. The following screenshot shows part of a critical chain after conversion of a delay module to an altshift_taps IP core. The section in highlight shows a design hierarchy of altshift_taps:r_rtl_0, indicating that synthesis replaces the bank of registers with the altshift_taps IP core.

Figure 105. Critical Chain Report with Delay Line

```
***** z ***** z ***** z *****
z RRG (required) z #12 z iteration\iteration_q\iteration_i\mova\mova_ll\delayer\delayer_q0\5\delayer\altshift_taps:r_rtl_0
Short Path z z z iteration_q\iteration_i\mova_ll\delayer\delayer_q0\5\delayer\altshift_taps:r_rtl_0
Short Path z z iteration\iteration_q\iteration_i\mova_ll\delayer\delayer_q0\5\delayer\altshift_taps:r_rtl_0
Short Path z unusable (hold0) z iteration\iteration_q\iteration_i\mova\mova_ll\delayer\delayer_q0\5\delayer\altshift_taps:r_rtl_0
Short Path z unusable (hold0) z iteration\iteration_q\iteration_i\mova\mova_ll\delayer\delayer_q0\5\delayer\altshift_taps:r_rtl_0
Short Path z z iteration\iteration_q\iteration_i\mova\mova_ll\delayer\delayer_q0\5\delayer\altshift_taps:r_rtl_0
Short Path z RRG z iteration\iteration_q\iteration_i\mova\mova_ll\delayer\delayer_q0\5\delayer\altshift_taps:r_rtl_0
***** z ***** z ***** z *****
z RPP z #31 z iteration\iteration_q\iteration_i\mova\mova_ll\trellis2_0\0\reg=451\de_m***[redacted]
z z z iteration_q\iteration_i\mova\mova_ll\trellis2_0\0\reg=451\data
z z z iteration_q\iteration_i\mova\mova_ll\trellis2_0\0\reg=451\comout
z z z iteration_q\iteration_i\mova\mova_ll\trellis2_0\0\reg[1]\fold
z RPP z #31 z iteration\iteration_q\iteration_i\mova\mova_ll\trellis2_0\0\reg[1][0]=.comb
```

When the Fitter places the chain of registers so close together, the Fitter cannot meet hold time when using any intermediate Hyper-Register locations. Turning off **Auto Shift Register Replacement** for the bank of registers prevents synthesis from using the altshift_taps IP core, and resolves any short path part of that critical chain.

Consider whether a RAM-based FIFO implementation is an acceptable substitute for a register delay line. If one function of the delay line is pipelining routing (to move signals a long distance across the chip), then a RAM-based implementation is typically not an acceptable substitute. If you do not require movement of data over long distance, A RAM-based implementation is a compact method to delay a bus of data.



6 Optimization Example

This section contains a round robin scheduler optimization example.

6.1 Round Robin Scheduler

The round robin scheduler is a basic functional block. The following example uses a modulus operator to determine the next client for service. The modulus operator is relatively slow and area inefficient because the modulus operator performs division.

Example 16. Source Code for Round Robin Scheduler

```

module round_robin_modulo # (
    parameter LOG2_CLIENTS = 3,
    parameter CLIENTS      = 7)
{
    // previous client to be serviced
    input wire [LOG2_CLIENTS -1:0] last,

    // Client requests:-
    input wire [CLIENTS -1:0] requests,

    // Next client to be serviced: -
    output reg [LOG2_CLIENTS -1:0] next,
};

//Schedule the next client in a round robin fashion, based on the previous

always @*
begin
    integer J, K;

    begin : find_next
    next = last; // Default to staying with the previous
    for (J = 1; J < CLIENTS; J=J+1)
        begin
            K = (last + J) % CLIENTS;
            if (requests[K] == 1'b1)
                begin
                    next = K[0 +: Log2_CLIENTS];
                    disable find_next;
                end
            end // of 'find_next'
        end
    endmodule

```

Figure 106. Fast Forward Compile Report for Round Robin Scheduler

Fast Forward Summary for Clock Domain clk_mod					
Step	Fast Forward Optimizations Applied	To Achieve Fmax	Slack	Requirement	Limiting Reason
1 Base Performance	0, including 0 pipeline stages	289.6 MHz	1.547	4.970	Insufficient Registers
2 Fast Forward Step #3	10, including 3 pipeline stage	458.82 MHz	2.867	4.970	Short Path, Long Path
3 Fast Forward Step #2	10, including 2 pipeline stages	499.0 MHz	2.996	4.970	Short Path, Long Path
4 Hyper-Optimization	10, including 2 pipeline stages	4.970	Short Path, Long Path

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Empirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2008
Registered

The Fast Forward Summary report identifies insufficient registers limiting Hyper-Retiming on the critical chain. The chain starts from the register that connects to the last input, through the modulus operator implemented with an LPM_DIVIDE IP core, and continuing to the register that connects to the next output.

Figure 107. Critical Chain for Base Performance for Round Robin Scheduler

Critical Chain at Limit		Recommendations for Critical Chain		
	Path Info	Register	Join	Element
1	Domain Boundary Entry		#1	
2	Long Path	empty slot		round_robin_mod_last_r[1][asdata]
3	Long Path	REG (required)		round_robin_mod_last_r[1]
4	Long Path			round_robin_mod_last_r[1]/q
5	Long Path			round_robin_mod_last_r[1]~i4_lab[laboutb(4)]
6	Long Path			round_robin_mod_last_r[1]~LAB.RE_X77.Y199.N0.I34
7	Long Path	empty slot		round_robin_mod_last_r[1]~R3.X75.Y199.N0.I23
8	Long Path	empty slot		round_robin_mod_last_r[1]~C4_X75.Y199.N0.I21
9	Long Path	empty slot		round_robin_mod_last_r[1]~LOCAL_INTERCONNECT_X76.Y199.N0.I33
10	Long Path			round_robin_mod_last_r[1]~LAB.RE_X76.Y199.N0.I0
11	Long Path	empty slot		mm[Mod3]auto_generated@divider/divider[jop_3~5]@stataf
12	Long Path			mm[Mod3]auto_generated@divider/divider[jop_3~9]@out
13	Long Path			mm[Mod3]auto_generated@divider/divider[jop_3~13]@in
14	Long Path			mm[Mod3]auto_generated@divider/divider[jop_3~1]@sumout
15	Long Path			round_robin_modulo_rrm[pm_divide.Mod3]@pm_d...f_u_div_vke.divider[jop_3~1]a_lab[laboutb(7)]
16	Long Path			round_robin_modulo_rrm[pm_divide.Mod3]@pm_vke.divider[jop_3~1]_LAB.RE_X76.Y199.N0.I97
17	Long Path	empty slot		round_robin_modulo_rrm[pm_divide.Mod3]@pm_vke.divider[jop_3~1]_LOCAL_INTERCONNECT_X76.Y199.N0.I26
18	Long Path			round_robin_modulo_rrm[pm_divide.Mod3]@pm_vke.divider[jop_3~1]_LAB.RE_X76.Y199.N0.I43
<hr/>				
54	Long Path	empty slot		round_robin_mod_next[0]~4_C4_X74.Y199.N0.I0
55	Long Path	empty slot		round_robin_mod_next[0]~4_R6_X75.Y200.N0.I6
56	Long Path	empty slot		round_robin_mod_next[0]~4_LOCAL_INTERCONNECT_X76.Y200.N0.I8
57	Long Path			round_robin_mod_next[0]~4_LAB.RE_X76.Y200.N0.I60
58	Long Path	empty slot		mm[next[2]~5]@dataf
59	Long Path			mm[next[2]~5]@combotb
60	Long Path			round_robin_mod_next[2]~reg0[d]
61	Long Path	REG (required)		round_robin_mod_next[2]~reg0
62	Long Path			round_robin_mod_next[2]~reg0[q]
63	Long Path			round_robin_mod_next[2]~reg0a_lab[laboutb(8)]
64	Long Path			round_robin_mod_next[2]~reg0_LAB.RE_X76.Y200.N0.I18
65	Long Path	empty slot		round_robin_mod_next[2]~reg0_C4_X76.Y196.N0.I34
66	Domain Boundary Entry		#2	

The 66 elements in the critical chain above correspond to the circuit diagram below that has 13 levels of logic. The modulus operator contributes significantly to the low performance. Nine of the 13 levels of logic are part of the implementation for the modulus operator.

Figure 108. Schematic for Critical Chain



Fast Forward compilation estimates a 70% performance improvement from adding two pipeline stages at the module inputs, for retiming through the logic cloud. At this point, the critical chain is a short path/long path and the chain involves the modulus operator.



Figure 109. Critical Chain Fast Forward Compile for Round Robin Scheduler

Fast Forward Optimizations Applied in Clock Domain clk_mod			Critical Chain at Limit	Recommendations For Critical Chain
	Path Info	Register	Join	Element
3	REG (required)	#1		round_robin_mod_requests_r[2]
2	Short Path			round_robin_mod_requests_r[2] q
3	Short Path	unusable (hold)		rmm Mux4~0 datae
4				rmm Mux4~0 combout
5				round_robin_module rmm Mux4~0 la_milab about[13]
6				round_robin_module rmm Mux4~0_LAB_REL_X76_Y198_N0_I103
7	REG	#2		round_robin_module rmm Mux4~0_LOCAL_INTERCONNECT_X75_Y198_N0_I3_dff
8	*****	*****	*****	*****
9	REG (required)	#1		round_robin_mod_requests_r[2]
10	Long Path			round_robin_mod_requests_r[2] q
11	Long Path	unusable (hold)		rmm Mux4~0 datae
12	Long Path			rmm Mux4~0 combout
13	Long Path			round_robin_module rmm Mux4~0 la_milab about[13]
14	Long Path			round_robin_module rmm Mux4~0_LAB_REL_X76_Y198_N0_I103
15	Long Path	REG		round_robin_module rmm Mux4~0_LOCAL_INTERCONNECT_X75_Y198_N0_I3_dff
16	Long Path			round_robin_module rmm Mux4~0_LOCAL_INTERCONNECT_X75_Y198_N0_I3
17	*****	*****	*****	*****
33	Long Path	empty slot		rmm next[1]~3 datac
34	Long Path			rmm next[1]~3 combout
35	Long Path			round_robin_mod_next[1]~reg0 d
36	Long Path	REG (required)	#3	round_robin_mod_next[1]~reg0
37	*****	*****	*****	*****
38		REG	#4	round_robin_mod_last_r[1]~LOCAL_INTERCONNECT_X75_Y198_N0_I47_dff
39	Short Path			round_robin_mod_last_r[1]~LOCAL_INTERCONNECT_X75_Y198_N0_I47
40	Short Path			round_robin_mod_last_r[1]~LAB_REL_X75_Y198_N0_I19
41	Short Path	REG		round_robin_module rmm next[1]~3_datab_dff
42	Short Path			rmm next[1]~3 datab
43	Short Path			rmm next[1]~3 combout
44	Short Path			round_robin_mod_next[1]~reg0 d
45	REG (required)	#3		round_robin_mod_next[1]~reg0
46	*****	*****	*****	*****
47	Long Path	REG	#4	round_robin_mod_last_r[1]~LOCAL_INTERCONNECT_X75_Y198_N0_I47_dff
48	Long Path			round_robin_mod_last_r[1]~LOCAL_INTERCONNECT_X75_Y198_N0_I47
49	Long Path			round_robin_mod_last_r[1]~LAB_REL_X75_Y198_N0_I3
50	Long Path	empty slot		***** auto_generated divider divider op_3~9 datab
51	*****	*****	*****	*****
68	Long Path	empty slot		rmm Mod4 auto_generated divider divider StageOut[12]~0 datab
69	Long Path			rmm Mod4 auto_generated divider divider StageOut[12]~0 combout
70	Long Path			round_robin_module rmm pm_divide Mod4 pm_d...vke divider StageOut[12]~0 la_lab about[12]
71	Long Path			round_robin_module rmm pm_divide Mod4 pm_d...er StageOut[12]~0_LAB_REL_X75_Y198_N0_I122
72	Long Path	empty slot		round_robin_module rmm pm_divide Mod4 ...12 0_LOCAL_INTERCONNECT_X75_Y198_N0_I58
73	Long Path			round_robin_module rmm pm_divide Mod4 pm_d...er StageOut[12]~0_LAB_REL_X75_Y198_N0_I28
74	Long Path	empty slot		rmm Mux4~0 datac
75	Long Path			rmm Mux4~0 combout
76	Long Path			round_robin_module rmm Mux4~0 la_milab about[13]
77	Long Path			round_robin_module rmm Mux4~0_LAB_REL_X75_Y198_N0_I103
78	Long Path	REG	#2	round_robin_module rmm Mux4~0_LOCAL_INTERCONNECT_X75_Y198_N0_I3_dff

The divider in the modulus operation is the bottleneck that requires RTL modification. Paths through the divider exist in the critical chain for all steps in the Fast Forward compile. Consider alternate implementations to calculate the next client to service, and avoid the modulus operator. If you switch to an implementation that specifies the number of clients as a power of two, determining the next client to service does not require a modulus operator. When you instantiate the module with fewer than 2^n clients, tie the unused request inputs to logic 0.

Example 17. Source Code for Round Robin Scheduler with Performance Improvement with 2^n Client Inputs

```
module round_robin # (
    parameter LOG2_CLIENTS = 3,
    parameter CLIENTS = 2**LOG2_CLIENTS)
{
    // Previous client to be serviced:-
    input wire [LOG2_CLIENTS -1:0] last,
```

```

// Client requests:-
  input wire [CLIENTS -1:0] requests,
  // Next client to be serviced:-
  output reg [LOG2_CLIENTS -1:0] next
};

//Schedule the next client in a round robin fashion, based on the
previous
  always @(next or last or requests)
begin
  integer J,K;

  begin : find_next
  next = last; // Default to staying with the previous
  for (J=1; J<CLIENTS; J = J+1)
    begin
      K = last + J;
      if (requests[k]0 +: LOG2_CLIENTS]] == 1'b1)
        begin
          next = K[0 +: LOG2_CLIENTS];
          disable find_next;
        end
      end// of 'find_next'
    end
  endmodule

```

Even without any Fast Forward optimizations, this round robin implementation runs almost 15% faster than the best Fast Forward compilation result with the modulus operator.

Figure 110. Fast Forward Summary Report for Round Robin Scheduler with Performance Improvement with 2^n Client Inputs

Fast Forward Summary for Clock Domain clk						
	Step	Fast Forward Optimizations Applied	To Achieve Fmax	Slack	Requirement	Limiting Reason
1	Base Performance	0, including 0 pipeline stages	570.13 MHz	3.246	4.970	Insufficient Registers
2	Fast Forward Step #1	10, including 1 pipeline stage	976.56 MHz	3.976	4.970	Insufficient Registers
3	Fast Forward Step #2	10, including 2 pipeline stages	1011.12 MHz	4.013	4.970	Short Path, Long Path
4	Hyper-Optimization	10, including 2 pipeline stages	--	--	4.970	Short Path, Long Path

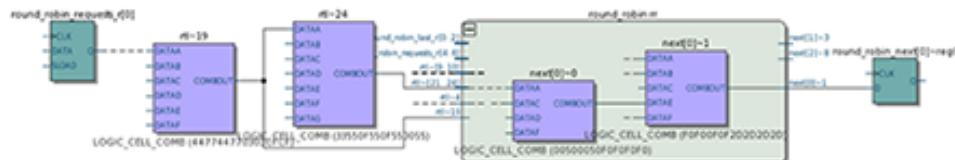
Without any Fast Forward optimization (the Base Performance step), the critical chain in this version also has the performance limiting reason of insufficient registers. Although critical chains in both versions contain only two registers, the critical chain for the 2^n version contains only 38 elements, compared to 66 elements in the modulus version.

Figure 111. Critical Chain for Round Robin Scheduler with Performance Improvement

Critical Chain at Limit		Recommendations for Critical Chain		
	Path Info	Register	Join	Element
1	Domain Boundary Entry		#1	
2	Long Path	empty slot		round_robin_requests_r[0].asdata
3	Long Path	REG (required)		round_robin_requests_r[0]
4	Long Path			round_robin_requests_r[0].q
5	Long Path			round_robin_requests_r[0].fa_lab.labout[18]
6	Long Path			round_robin_requests_r[0].LAB.RE_X77_Y198_N0_I08
7	Long Path	empty slot		round_robin_requests_r[0].R6_X72_Y198_N0_I44
8	Long Path	empty slot		round_robin_requests_r[0].C4_X75_Y199_N0_I19
9	Long Path	empty slot		round_robin_requests_r[0].LOCAL_INTERCONNECT_X76_Y200_N0_I45
10	Long Path			round_robin_requests_r[0].LAB.RE_X76_Y200_N0_I50
11	Long Path	empty slot		rtf~19 dataa
<hr/>				
12	Long Path	empty slot		round_robin_rr.next[0]~0.LOCAL_INTERCONNECT_X77_Y200_N0_I55
28	Long Path	empty slot		round_robin_rr.next[0]~0.LAB.RE_X77_Y200_N0_I1
29	Long Path			rr[next[0]~1].datae
30	Long Path	empty slot		rr[next[0]~1].combout
31	Long Path			round_robin_next[0].reg0.d
32	Long Path	REG (required)		round_robin_next[0].reg0
33	Long Path			round_robin_next[0].reg0.q
34	Long Path			round_robin_next[0].reg0.lab.labout[1]
35	Long Path			round_robin_next[0].reg0.LOCAL_INTERCONNECT_X77_Y200_N0_I91
36	Long Path			round_robin_next[0].reg0.LAB.RE_X77_Y200_N0_I1
37	Long Path	empty slot		round_robin_next[0].reg0.R3_X77_Y200_N0_I1
38	Domain Boundary Entry		#2	

The 38 elements in the critical chain above correspond to the following circuit diagram with only four levels of logic.

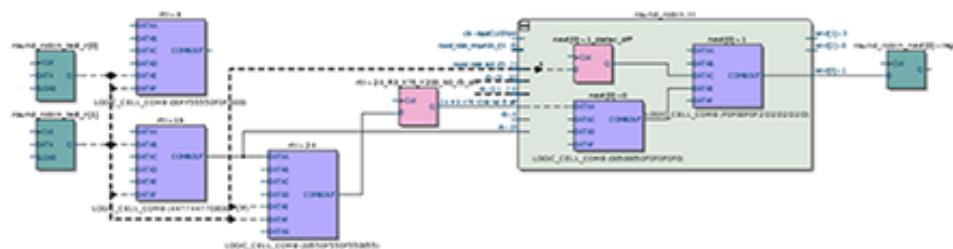
Figure 112. Schematic for Critical Chain with Performance Improvement



By adding two register stages at the input, for retiming through the logic cloud, Fast Forward Compile takes the circuit performance to 1 GHz, which is the architectural limit of Intel Stratix 10 devices. As with the modulus version, the final critical chain after Fast Forward optimization has a limiting reason of short path/long path, but the performance is double the performance of the modulus version.

Figure 113. Critical Chain for Round Robin Scheduler with Best Performance

Fast Forward Optimizations Applied in Clock Domain clk			Critical Chain at Limit	Recommendations for Critical Element
Path Info	Register	Join		
1	empty slot	#1	rti~4 dataf	
2		#2	rti~4 combout	
3	
4	REG (required)	#3	round_robin_last_r[1]	
5 Short Path			round_robin_last_r[1] q	
6 Short Path	unusable (hold)		rti~4 datae	
7		#2	rti~4 combout	
8	
9	REG (required)	#3	round_robin_last_r[1]	
10 Long Path			round_robin_last_r[1] q	
11 Long Path			round_robin_last_r[1]~ia_milab/laboutb[16]	
12 Long Path			round_robin_last_r[1]~LAB_RE_X76_Y200_N0_I126	
13 Long Path	empty slot		round_robin_last_r[1]~R6_X71_Y200_N0_I14	
14 Long Path	empty slot		round_robin_last_r[1]~LOCAL_INTERCONNECT_X76_Y200_N0_I13	
Long Path				
35 Long Path	empty slot		round_robin_rr next[0]~0_LAB_RE_X77_Y200_N0_I1	
36 Long Path	empty slot		rr next[0]~1 datae	
37 Long Path			rr next[0]~1 combout	
38 Long Path			round_robin_next[0]~reg0 d	
39 Long Path	REG (required)	#4	round_robin_next[0]~reg0	
40	
41		#5	round_robin_last_r[0]~LAB_RE_X76_Y200_N0_I22	
42 Short Path	unusable (hold)		round_robin_last_r[0]~LOCAL_INTERCONNECT_X77_Y200_N0_I34	
43 Short Path			round_robin_last_r[0]~LAB_RE_X77_Y200_N0_I4	
44 Short Path	REG		round_robin_rr next[0]~1_datac_dff	
45			rr next[0]~1 datac	
46			rr next[0]~1 combout	
47			round_robin_next[0]~reg0 d	
48	REG (required)	#4	round_robin_next[0]~reg0	
49	
50		#5	round_robin_last_r[0]~LAB_RE_X76_Y200_N0_I22	
51	empty slot		round_robin_last_r[0]~LOCAL_INTERCONNECT_X76_Y200_N0_I21	
52	empty slot		round_robin_last_r[0]~LAB_RE_X76_Y200_N0_I72	
53	empty slot	#1	rti~4 dataf	

Figure 114. Schematic for Critical Chain with Best Performance


Removing the modulus operator, and switching to a power-of-two implementation, are small design changes that provide a dramatic performance increase.

- Use natural powers of two for math operations whenever possible
- Explore alternative implementations for seemingly basic functions.

In this example, changing the implementation of the round robin logic provides more performance increase than adding pipeline stages.



7 Intel Hyperflex Architecture Porting Guidelines

This chapter provides guidelines for migrating a Stratix V or an Intel Arria 10 design to the Intel Stratix 10 Intel Hyperflex FPGA architecture. These guidelines allow you to quickly evaluate the benefits of design optimization in the Intel Hyperflex architecture, while still preserving your design's functional intent.

Porting requires minor modifications to the design, but can achieve major performance gains for your design's most critical modules.

To experiment with performance exploration, select for migration a large, second-level module that does not contain periphery IP (transceiver, memory, etc.). During performance exploration, review reported performance improvements.

7.1 Design Migration and Performance Exploration

You can migrate a Stratix V or an Intel Arria 10 design to an Intel Stratix 10 device to evaluate performance improvement. Migrating a design for Intel Stratix 10 devices *requires* only minor changes. However, additional *non-required* changes further performance improvement. This performance improvement helps you to close timing and add functionality to your design.

Any device migration typically requires some common design changes. These changes include updating PLLs, high-speed I/O pins, and other device resources. The Intel Stratix 10 version of these components have the same general functionality as in previous device families. However, the Intel Stratix 10 components include features to enable higher operational speeds:

- DSP blocks include pipeline registers and support a floating point mode.
- Memory blocks include additional logic for coherency, and width restrictions.

The high level steps in the migration process are:

1. Select for migration a lower-level block in the design, without any specialized IP.
2. Black-box any special IP component and only retain components that the current level requires. Only retain the following key blocks for core performance evaluation:

- PLLs for generating clocks
- Core blocks (logic, registers, memories, DSPs)

Note: If you migrate a design from a previous version of the Intel Quartus Prime software, some Intel FPGA IP may require replacement if incompatible with the current software version.

3. Maintain module port definitions when black-boxing components. Do not simply remove the source file from the project.
4. Specify the port definition and direction of every component that the design uses to the synthesis software. Failure to define the ports results in compilation errors.
5. During design synthesis, review the error messages and correct any missing port or module definitions.

The easiest way to black-box a module is to empty the functional content. The following examples show black-boxing in Verilog HDL or VHDL.

7.1.1 Black-boxing Verilog HDL Modules

In black-boxing Verilog HDL, keep the module definition but delete the functional description.

Before:

```
// k-bit 2-to-1 multiplexer
module mux2tol (V, W, Sel, F);
    parameter k = 8;
    input [k-1:0] V, W;
    input Sel;
    output [k-1:0] F;
    reg [k-1:0] F;

    always @(V or W or Sel)
        if (Sel == 0)
            F = V;
        else
            F = W;
endmodule
```

After:

```
// k-bit 2-to-1 multiplexer
module mux2tol (V, W, Sel, F);
    parameter k = 8;
    input [k-1:0] V, W;
    input Sel;
    output [k-1:0] F;
endmodule
```

7.1.2 Black-boxing VHDL Modules

In black-boxing VHDL, keep the entity as-is, but delete the architecture. In the case when you have multiple architectures, make sure you remove all of them.

Before:

```
-- k-bit 2-to-1 multiplexer
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2tol IS
```



```

GENERIC ( k : INTEGER := 8 ) ;
PORT (      V, W : IN      STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
            Sel    : IN      STD_LOGIC ;
            F      : OUT     STD_LOGIC_VECTOR(k-1 DOWNTO 0) ) ;
END mux2tol ;

ARCHITECTURE Behavior OF mux2tol IS
BEGIN
    PROCESS ( V, W, Sel ) 
    BEGIN
        IF Sel = '0' THEN
            F <= V ;
        ELSE
            F <= W ;
        END IF ;
    END PROCESS ;
END Behavior ;

```

After:

```

-- k-bit 2-to-1 multiplexer
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2tol IS
GENERIC ( k : INTEGER := 8 ) ;
PORT (      V, W : IN      STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
            Sel    : IN      STD_LOGIC ;
            F      : OUT     STD_LOGIC_VECTOR(k-1 DOWNTO 0) ) ;
END mux2tol ;

ARCHITECTURE Behavior OF mux2tol IS
BEGIN
END Behavior ;

```

In addition to black-boxing modules, you must assign the modules to an empty design partition. The partition prevents the logic connected to the black-boxed modules from being optimized away during synthesis.

To create a new partition:

1. In the Project Navigator **Hierarchy** tab, right-click the black-boxed module, and then click **Design Partition > Set as Design Partition**.
2. For **Empty**, select **Yes**.
3. Add all the black-box modules into this partition.

Figure 115. Create New Empty Partition

Design Partitions Window						
Assignments View		Compilation View				
Partition Name	Hierarchy Path	Type	Preservation Level	Empty	Partition Database File	Color
<<new>>						
auto_max	auto	Default	Not Set	Yes		purple
speed_ch	speed	Default	Not Set	No		yellow
tick_cnt	tick	Default	Not Set	No		orange
time_cnt	time_c	Default	Not Set	No		blue

7.1.3 Clock Management

After black-boxing appropriate logic, ensure that all registers in the design are still receiving a clock signal. All the PLLs must still be present. Identify any clock existing a black-boxed module. If this occurs in your design, recreate this clock. Failure to recreate the clock marks any register downstream as unclocked. This condition changes the logic function of your design, because synthesis can remove registers that do not receive a clock. Examine the clock definitions in the .sdc file to determine if this file specifies a clock definition in one of the black-boxed modules. Looking at a particular module, several conditions can occur:

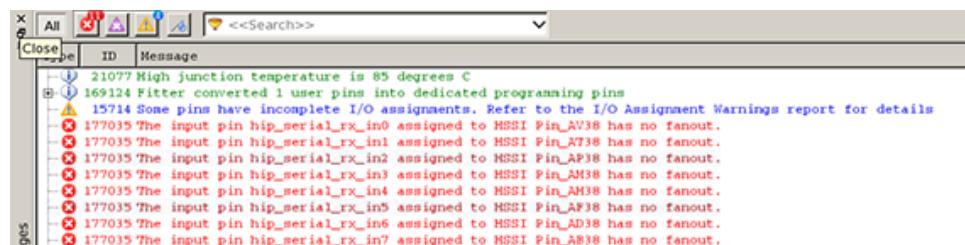
- There is a clock definition in that module:
 - Does the clock signal reach the primary output of the module and a clock pin of a register downstream of the module?
 - No: this clock is completely internal and no action required.
 - Yes: create a clock on the output pin of that module matching the definition in the .sdc.
- There is no clock definition in that module:
 - Is there a clock feedthrough path in that module?
 - No: there is no action required.
 - Yes: create a new clock on the feedthrough output pin of the module.

7.1.4 Pin Assignments

Black-boxing logic can be the cause of some pin assignment errors. Use the following guidelines to resolve pin assignments. Reassign high-speed communication input pins to correct such errors.

The checks for the status of high-speed pins and generates some errors if you do not connect these pins. When you black-box transceivers, you may encounter this situation. To address these errors, re-assign the HSSI pins to a standard I/O pin. Verify and change the I/O bank if necessary.

Figure 116. High-speed Pin Error Messages

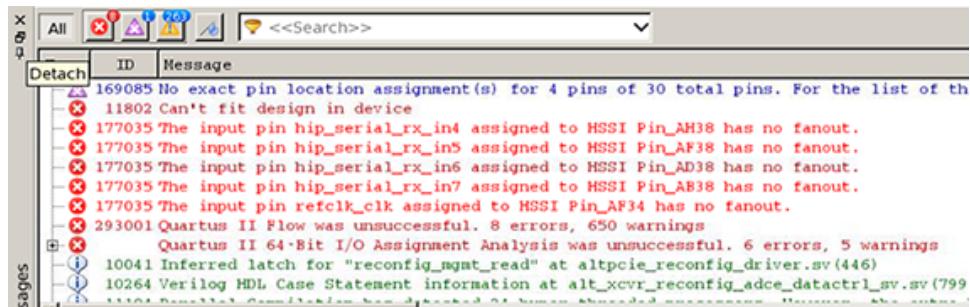


In the .qsf file, the assignment translates to the following:

```
set_instance_assignment -name IO_STANDARD "2.5 V" -to hip_serial_rx_in1
set_instance_assignment -name IO_STANDARD "2.5 V" -to hip_serial_rx_in2
set_instance_assignment -name IO_STANDARD "2.5 V" -to hip_serial_rx_in3
set_location_assignment IOBANK_4A -to hip_serial_rx_in1
set_location_assignment IOBANK_4A -to hip_serial_rx_in2
set_location_assignment IOBANK_4A -to hip_serial_rx_in3
```



Figure 117. Pins Error Messages



Dangling pins

If you have high-speed I/O pins dangling because of black-boxing components, set them to virtual pins. Enter this assignment in the Assignment Editor, or in the .qsf file directly, as shown below:

```
set_instance_assignment -name VIRTUAL_PIN ON -to hip_serial_tx_in1
set_instance_assignment -name VIRTUAL_PIN ON -to hip_serial_tx_in2
set_instance_assignment -name VIRTUAL_PIN ON -to hip_serial_tx_in3
```

GPIO pins

If you have GPIO pins, make them virtual pins using this qsf assignment:

```
set_instance_assignment VIRTUAL_PIN -to *
```

7.1.5 Transceiver Control Logic

Your design may have some components with added logic that controls them. For example, you might have a small design which controls the reset function of a transceiver. You can leave these blocks in the top-level design and their logic is available for optimization.



7.1.6 Upgrade Outdated IP Cores

The Intel Quartus Prime software alerts you to outdated IP components in your design. Unless black-boxed, upgrade every outdated IP component to the current version:

1. Click **Project > Upgrade IP Components** to upgrade the components to the latest version.
2. To upgrade one or more IP cores that support automatic upgrade, ensure that you turn on the **Auto Upgrade** option for the IP core, and click **Perform Automatic Upgrade**. The **Status** and **Version** columns update when upgrade is complete. Example designs provided with any IP core regenerate automatically whenever you upgrade an IP core.
3. To manually upgrade an individual IP core, select the IP core and click **Upgrade in Editor** (or simply double-click the IP core name). The parameter editor opens, allowing you to adjust parameters and regenerate the latest version of the IP core.

Note: Some IP components cannot upgrade for Intel Stratix 10 devices. If those components are critical (for example, PLL), modify your design and replace them with Intel Stratix 10-compatible IP components.

7.2 Top-Level Design Considerations

I/O constraints

In order to get the maximum performance from register retiming, wrap the top-level in a register ring and remove the following constraints from your .sdc file:

- set_input_delay
- set_output_delay

These constraints model external delay outside of the block. For the purposes of analyzing the effect of design optimizations, use all the available slack within the block. This technique helps maximize performance at the module level. Replace these constraints when moving to full chip timing closure.

Resets

If you remove reset generation from the design, provide a replacement signal by direct connection to an input pin of your design. This configuration may affect the retiming capabilities in Intel Stratix 10 architectures. Add two pipeline stages to your reset signal. This technique allows the Compiler to optimize between the reset input and the first level of registers.



Special Blocks

Retiming does not automatically change some components. Some examples are DSP and M20K blocks. In order to achieve higher performance through retiming, manually recompile these blocks. Look for the following conditions:

- DSPs: Watch the pipelining depth. More pipeline stages results in a faster design. If the logic levels in a DSP block limits retiming, add more pipeline stages.
- M20Ks: Retiming relies heavily on the presence of registers to move logic. With M20K blocks, you can help the Compiler by registering the logic memory twice:
 - Once inside the M20K block directly
 - Once in the fabric, at the pins of the block

Register the Block

Register all inputs and all outputs of your block. This register ring mimics driving the block when embedded in the full design. The ring also avoids the retiming restriction with registers connected to inputs or outputs. The Compiler can now retime the first and last level of registers more realistically.



8 Appendices

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Empirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2008
Registered



8.1 Appendix A: Parameterizable Pipeline Modules

The following examples show parameterizable pipeline modules in Verilog HDL, SystemVerilog, and VHDL. Use these code blocks at top-level I/Os and clock domain boundaries to change the latency of your circuit.

Example 18. Parameterizable Hyper-Pipelining Verilog HDL Module

```
// Hyper-pipelining module HyperPipe Intel Version 2014/08/12
(* altera_attribute = "-name AUTO_SHIFT_REGISTER_RECOGNITION off" *)
module hyperpipe #(
    parameter CYCLES,
    parameter WIDTH,
) (
    input  clk,
    input  [WIDTH-1:0] din,
    output [WIDTH-1:0] dout
);

generate
    if (CYCLES==0) begin : GEN_COMB_INPUT
        assign dout = din;
    end
    else begin : GEN_REG_INPUT
        integer i;
        reg [WIDTH-1:0] R_data [CYCLES-1:0];

        always @(posedge clk) begin
            R_data[0] <= din;
            for(i=1;i<CYCLES;i=i+1) R_data[i] <= R_data[i-1];
        end
        assign dout = R_data[CYCLES-1];
    end
endgenerate
endmodule
```

Example 19. Parameterizable Hyper-Pipelining Verilog HDL Instance

```
// Instantiation Template:
hyperpipe #(
    .CYCLES ( ),
    .WIDTH   ( ),
) hp (
    .clk     ( ),
    .din     ( ),
    .dout    ( )
);
```

Example 20. Parameterizable Hyper-Pipelining SystemVerilog Module

```
// Hyper-pipelining module HyperPipe Intel Version 2014/08/12
(* altera_attribute = "-name AUTO_SHIFT_REGISTER_RECOGNITION off" *)
module hyperpipe_2d #(
    parameter CYCLES,
    parameter PACKED_WIDTH,
    parameter UNPACKED_WIDTH,
) (
    input  clk,
    input  [PACKED_WIDTH-1:0] din  [UNPACKED_WIDTH-1:0],
    output [PACKED_WIDTH-1:0] dout [UNPACKED_WIDTH-1:0]
);

generate
    if (CYCLES==0) begin : GEN_COMB_INPUT
        assign dout = din;
    end
    else begin : GEN_REG_INPUT
```



```
integer i;
reg [PACKED_WIDTH-1:0] R_data
    [CYCLES-1:0][UNPACKED_WIDTH-1:0];

always @(posedge clk) begin
    R_data[0] <= din;
    for(i=1; i<CYCLES; i=i+1)
        R_data[i] <= R_data[i-1];
    end
    assign dout = R_data[CYCLES-1];
end
endgenerate
```

Example 21. Parameterizable Hyper-Pipelining SystemVerilog Instance

```
// Instantiation Template:
hyperpipe #((
    .CYCLES      ( ),
    .PACKED_WIDTH ( ),
    .UNPACKED_WIDTH ( ),
) hp (
    .clk  ( ),
    .din  ( ),
    .dout ( )
);
```

Example 22. Parameterizable Hyper-Pipelining VHDL Entity

```
-- HyperPipe Intel Version 2014/08/12
library IEEE;
use IEEE.std_logic_1164.all;
library altera;
use altera.altera_syn_attributes.all;

entity hyperpipe is
    generic (
        CYCLES : integer
        WIDTH   : integer;
    );
    port (
        clk  : in std_logic;
        din  : in std_logic_vector (WIDTH - 1 downto 0);
        dout : out std_logic_vector (WIDTH - 1 downto 0)
    );
end hyperpipe;

architecture arch of hyperpipe is

    -- Prevent large hyperpipes from going into memory-based
    altshift_taps,
    -- since that won't take advantage of Hyper-Registers
    attribute altera_attribute of hyperpipe :
        entity is "-name AUTO_SHIFT_REGISTER_RECOGNITION off";

    type hyperpipe_t is array(CYCLES-1 downto 0) of
        std_logic_vector(WIDTH-1 downto 0);
    signal HR : hyperpipe_t;

begin
    wire : if CYCLES=0 GENERATE
        -- The 0 bit is just a pass-thru, when CYCLES is set to 0
        dout <= din;
    end generate wire;

    hp : if CYCLES>0 GENERATE
        process (clk) begin
            if (clk'event and clk='1')then
                HR <= HR(HR'high-1 downto 0) & din;
            end if;
        end process;
    end generate hp;
end architecture;
```



```
        end process;
        dout <= HR(HR'high);
    end generate hp;

end arch;
```

Example 23. Parameterizable Hyper-Pipelining VHDL Instance

```
-- Template Declaration
component hyperpipe
    generic (
        CYCLES : integer;
        WIDTH   : integer;
    );
    port (
        clk   : in  std_logic;
        din   : in  std_logic_vector(WIDTH - 1 downto 0);
        dout  : out std_logic_vector(WIDTH - 1 downto 0)
    );
end component;

-- Instantiation Template:
hp : hyperpipe
    generic map (
        CYCLES =>
        WIDTH   => ,
    )
    port map (
        clk   => ,
        din   => ,
        dout  =>
    );
```

8.2 Appendix B: Clock Enables and Resets

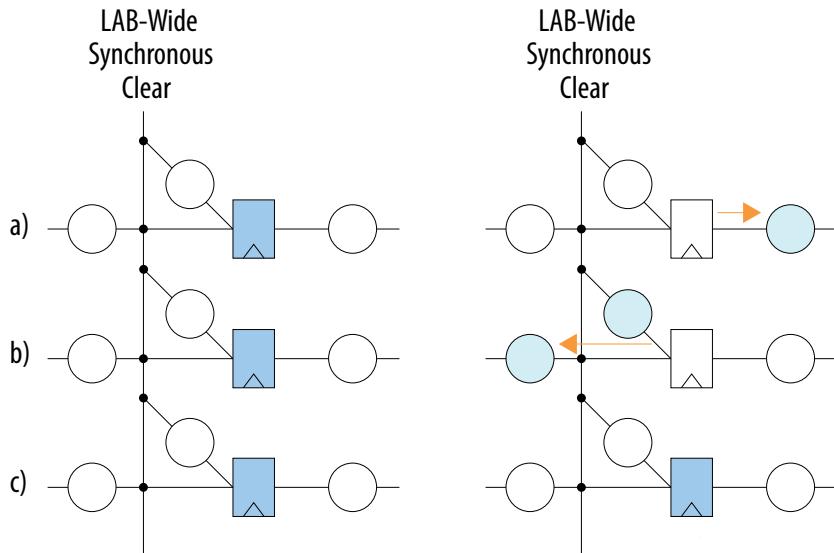
8.2.1 Synchronous Resets and Limitations

Converting asynchronous resets to synchronous eases retiming restrictions, but does not remove all performance restrictions. The ALM register's dedicated LAB-wide signal often performs synchronous clears. The signal's fan-out determines use of this signal during synthesis. The Compiler typically implements a synchronous clear with a small fan-out in logic. Larger fan-outs use this dedicated signal. Even if synthesis uses the synchronous clear, the Compiler still retimes the register into Hyper-Registers. The bypass mode of the ALM register enables this functionality. When the Compiler bypasses the register, the `sclr` signal and other control signals remain accessible.

In the following example, the LAB-wide synchronous clear feeds multiple ALM registers. A Hyper-Register is available along the synchronous clear path for every register.

Figure 118. Retiming Example for Synchronous Resets

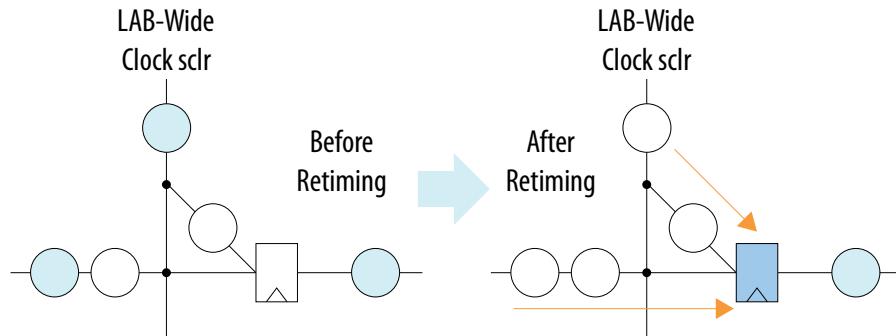
Circles represent Hyper-Registers and rectangles represent ALM registers. An unfilled object represents an unoccupied location and a blue-filled object is occupied.



During retiming, the Compiler pushes top register in row (a) into a Hyper-Register. The Compiler implements this by bypassing the ALM register, but still using the SCLR logic that feeds that register. When you use the LAB-wide SCLR signal, an ALM register must exist on the data path, but you need not use the register.

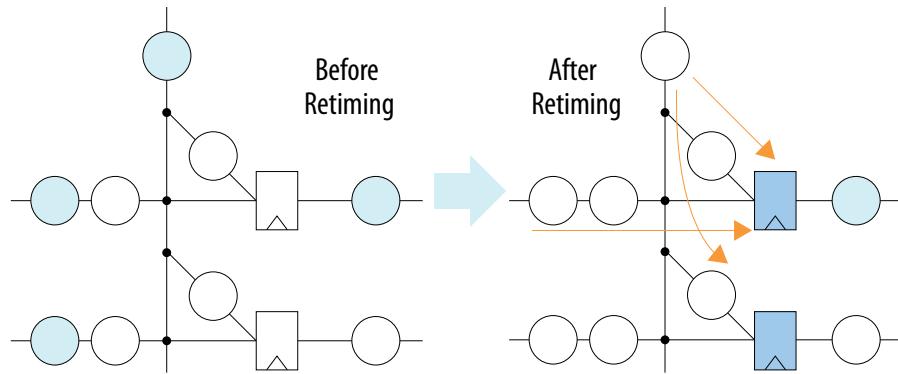
Register retiming pushes the register in row (b) left into its data path. The register pushes through a signal split of the data path and synchronous clear. The Compiler must push this register onto both nets: one register in the data path, and one register in the synchronous clear path. This implementation is possible because each path has a Hyper-Register.

Retiming is more complex when another register pushes forward into the ALM. As shown in the following figure, a register from the asynchronous clear port, and a register from the data path, merge together.

Figure 119. Retiming Example – Second Register Pushes out of ALM


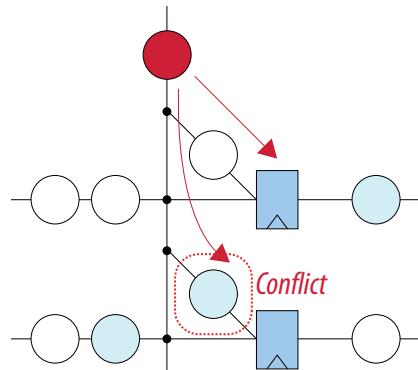
Because other registers share the synchronous clear path, the register splits on the path to other synchronous clear ports.

Figure 120. Retiming Example – Register Splits on the Path to other Synchronous Clear Ports



In the following figure, the Hyper-Register at a synchronous clear is in use and cannot accept another register. The Compiler cannot retime this register for the second time through the ALM.

Figure 121. Retiming Example – Conflict at Synchronous Clear



Two key architectural components enable movement of ALM registers with a synchronous clear forward or backward:

- The ability to bypass the ALM register
- A Hyper-Register on the synchronous clear path

To push more registers through, retiming becomes difficult. Performance improvement is better with asynchronous reset removal than conversion to synchronous resets. Synchronous clears are often difficult to retime because of their wide broadcast nature.

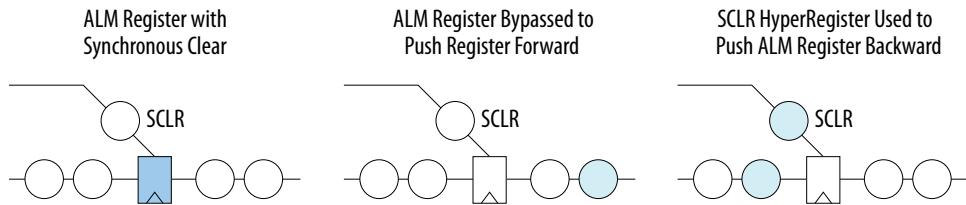
8.2.1.1 Synchronous Resets Summary

Synchronous clears can limit the amount of retiming. There are two issues with synchronous clears that cause problems for retiming:

- A short path, usually traveling directly from the source register to the destination register without any logic between them. Short paths are not normally a problem, because the Compiler retimes the positive slack to longer paths. This retiming improves performance. However, short paths typically connect to long data paths that require retiming. By retiming many registers along long paths, registers push down or pull up this short path. This problem is not significant in normal logic, but becomes significant when synchronous clears have large fan-outs.
- Synchronous clears have large fan-outs. When aggressive retiming pushes registers up or down the synchronous clear path, paths can clutter until they cannot accept more registers. This situation results in path length imbalances, and the Compiler can pull no more registers from the synchronous clear paths.

Aggressive retiming occurs when the Compiler retimes a second register through the ALM register.

Figure 122. Aggressive Retiming



Intel Stratix 10 devices have a dedicated Hyper-Register on the SCLR path, with the ability to place the ALM register into bypass mode. This ability allows you to push and pull this register. If you push the register forward, then you must pull a register down the SCLR path and merge the two. If you push the register back, then you must push a duplicate register up the SCLR path. You can use both of these options. However, you create bottlenecks when multiple registers push and pull registers up and down the synchronous clear routing.

Use resets in a practical manner. Control logic mostly requires synchronous reset. Logic that may not require a synchronous reset helps with timing. Refer to the following guidelines about synchronous resets:

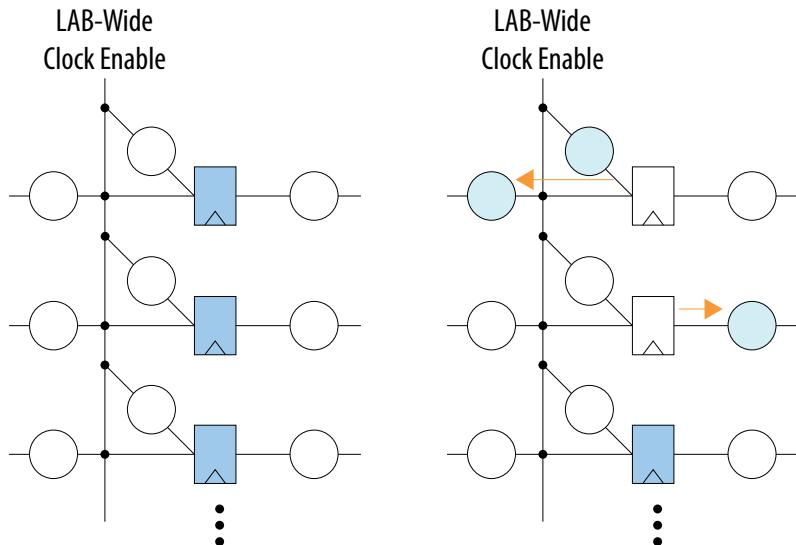
- Avoid synchronous resets in new code that must run at high speed. This limitation generally applies to data path logic that flushes out while the system is in reset, or logic with values that the system ignores when coming out of reset.
- Control logic often requires a synchronous reset, so there is no avoiding the reset in that situation.
- For existing logic that runs at high speeds, remove the resets wherever possible. If you do not understand the logic behavior at reset, retain the synchronous reset. Only remove the synchronous clear if a timing issue arises.
- Pipeline the synchronous clear. This technique does not help when you must pull registers back, but can help when you need to pull registers forward into the data path.
- Duplicate synchronous clear logic for different hierarchies. This technique limits the fan-out of the synchronous clear, so that the Compiler can retime the clear with the local logic. Apply this technique only after you determine that an existing synchronous clear with large fan-out limits retiming. This technique is not difficult on the back-end, because the technique does not change design functionality.
- Duplicate synchronous clear for different clock domain and inverted clocks. This technique can overcome some retiming restrictions due to boundary or multiple period requirement issues.

8.2.2 Retiming with Clock Enables

Like synchronous resets, clock enables use a dedicated LAB-wide resource that feeds a specific function in the ALM register. Similarly, Intel Stratix 10 devices support special logic that simplifies retiming logic with clock enables. However, wide broadcast control signals, such as clock enables (and synchronous clears), are difficult to retime.

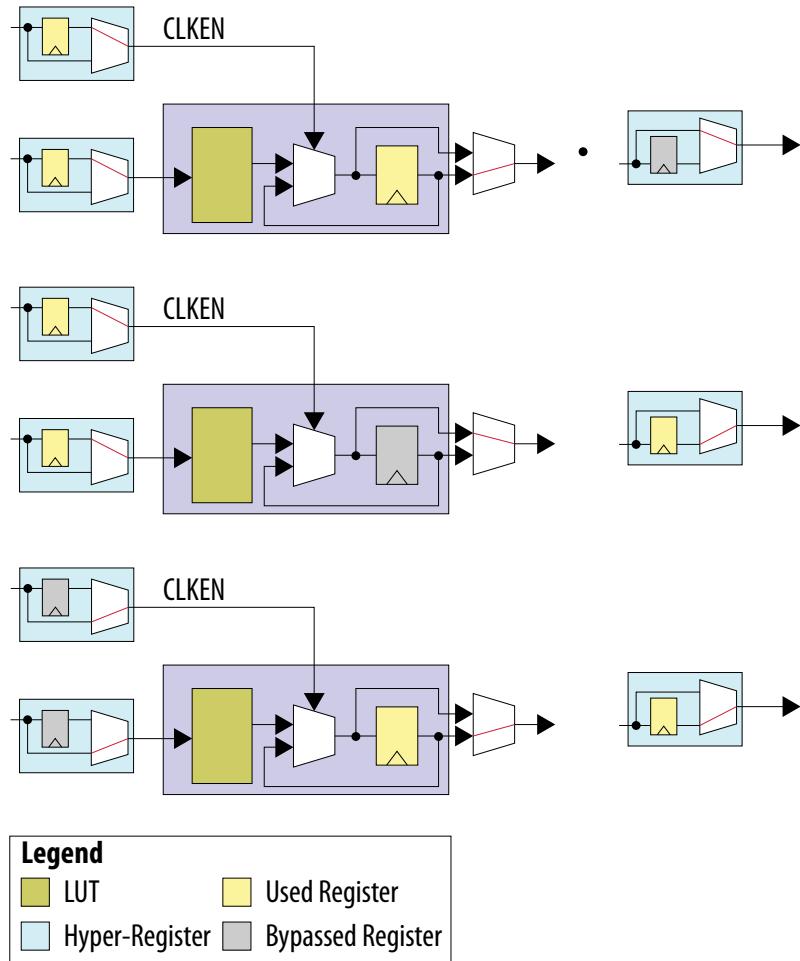
Figure 123. ALM Representing Clock Enables

The following figure shows the sequence of retiming moves for the asynchronous clears in the *Synchronous Resets and Limitations* section.



The top circuit contains a dedicated Hyper-Register on the clock enable path. To push back the register, the Compiler must split the register, so that another register pushes up the clock enable path. In this case, the Hyper-Register location absorbs the register without problem. These features allow the Compiler to easily retime an ALM register with a clock enable backward or forward (middle circuit), to improve timing. A useful feature of a clock enable is that logic usually generates by synchronous signals, so that the Compiler can retime the clock enable path alongside the data path.

Figure 124. Retiming Steps and Structure with an ALM register and Hyper-Registers



The figure shows retiming of the clock enable signal `clken` typical broadcast type control signal. In the top circuit, before retiming, the circuit uses an ALM register. The circuit also uses the Hyper-Registers on the clock enable and data paths. In the middle circuit, the ALM register retimes forward into a Hyper-Register outside the ALM, into the routing fabric. The circuit still uses the ALM register, but the register is not on the data path through the ALM. The ALM holds the previous value of the register. The clock enable mux now selects between this previous value and the new value, based on the clock enable. The diagram shows retiming forward of a second register from the clock enable and data paths into the ALM register. The circuit now uses the ALM register in the path. You can repeat this process and iteratively retime multiple registers across an enabled ALM register.

Related Links

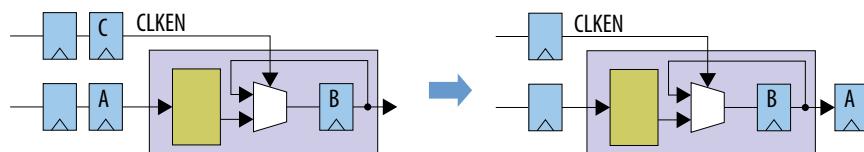
[Synchronous Resets and Limitations](#) on page 111

8.2.2.1 Example for Broadcast Control Signals

Broadcast control signals that fan-out to many destinations limit retiming. Asynchronous clears can limit retiming due to device support of specific register control signals. However, even synchronous signals, such as synchronous clear and clock enable, can limit retiming when part of a short path or long path critical chain. The use of a synchronous control signal is not a limiting reason by itself; rather the structure and placement of the circuit causes the limit.

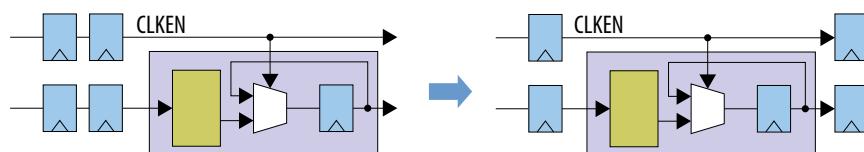
A register must be available on all of the node's inputs to forward retime a register over a node. To retime register A over register B in the following diagram, the Compiler must pull a register from all inputs, including register C on the clock enable input. Additionally, if the Compiler retimes a register down one side of a branch point, the Compiler must retime a copy of the register down all sides of a branch point. This requirement is the same for conventional retiming and Hyper-Retiming.

Figure 125. Retiming through a Clock Enable



There is a branch point at the clock enable input of register B. The branch point consists of additional fan-out to other destinations besides the clock enable. To retime register A over register B, the operation is the same as the previous diagram. However, the presence of the branch point means that a copy of register C must retime along the other side of the branch point, to register C.

Figure 126. Retiming through a Clock Enable with a Branch Point

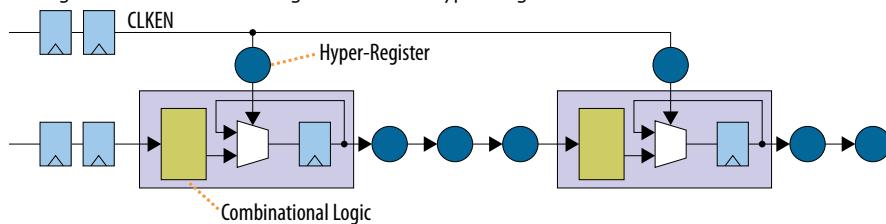


Retiming Example

The following diagrams combine the previous two steps to illustrate the process of a forward Hyper-Retiming push in the presence of a broadcast clock enable signal or a branch point.

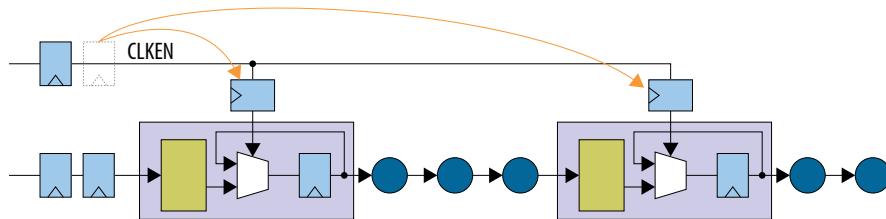
Figure 127. Retiming Example Starting Point

Hyper-Retiming can move a retimed register into the Hyper-Registers.

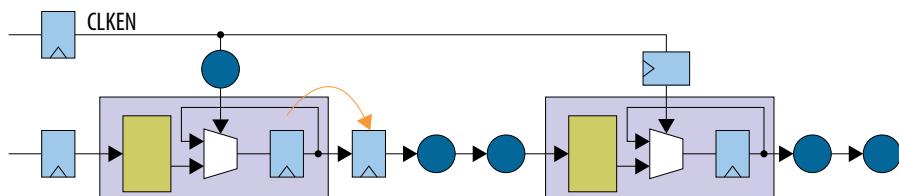


Each register's clock enable has one Hyper-Register location at its input. Because of the placement and routing, the register-to-register path includes three Hyper-Register locations. A different compilation can result in more or fewer Hyper-Register locations. Additionally, there are registers on the data and clock enable inputs to this chain that Hyper-Retiming can retime. These registers exist in the RTL, or you can define them with options that the *Pipeline Stages* section describes.

One stage of the input registers retime into a Hyper-Register location between the two registers. [Figure 128](#) on page 118 shows one part of the Hyper-Retiming forward push. One of the registers on the clock enable input retimes over the branch point, with a copy going to a Hyper-Register location at each clock enable input.

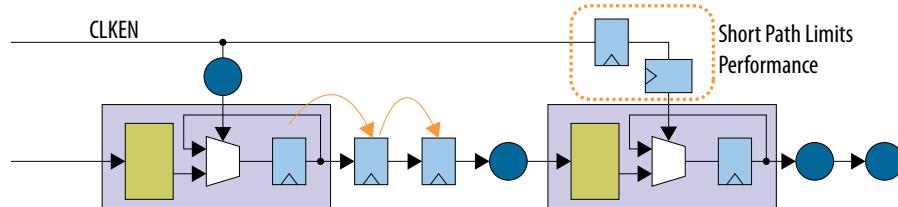
Figure 128. Retiming Example Intermediate Point


[Figure 129](#) on page 118 shows the positions of the registers in the circuit after Hyper-Retiming completes the forward push. The two registers at the inputs of the left register retime to a Hyper-Register location. This diagram is functionally equivalent to the two previous diagrams. The one Hyper-Register location at the clock enable input of the second register remains occupied. There are no other Hyper-Register locations on the clock enable path to the second register, yet there is still one register at the inputs that the Compiler can retime.

Figure 129. Retiming Example Ending Point


[Figure 130](#) on page 119 shows the register positions Hyper-Retiming uses if a short path/long path critical chain do not limit the path. However, because no Hyper-Registers are available on the right-hand clock enable path, Hyper-Retiming cannot retime the circuit as shown in the diagram.

Figure 130. Retiming Example Limiting condition



Because the clock enable path to the second register has no more Hyper-Register locations available, the Compiler reports this as the short path. Because the register-to-register path is too long to operate at the performance requirement, although having more available Hyper-Register locations for the retimed registers, the Compiler reports this as the long path.

The example is intentionally simple to show the structure of a short path/long path critical chain. In reality, a two-fan-out load is not the critical chain in a circuit. However, broadcast control signals can become the limiting critical chains with higher fan-out. Avoid or rewrite such structures to improve performance.

Related Links

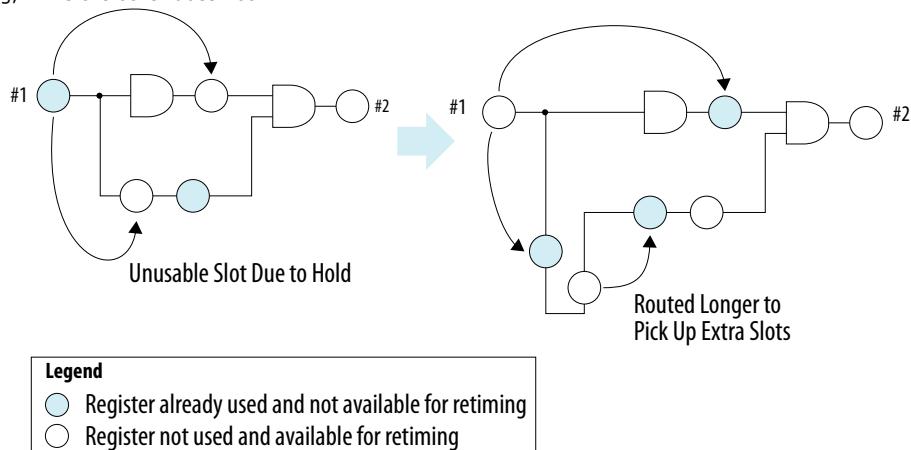
[Appendix A: Parameterizable Pipeline Modules](#) on page 109

8.2.3 Resolving Short Paths

Retiming registers that are close to each other can potentially trigger hold violations at higher speeds. The following figure shows how a short path limits retiming.

Figure 131. Short Paths Limiting Retiming

In this example, forward retiming pushes a register onto two paths, but one path has an available register for retiming, while the other does not.



In the circuit on the left, if register #1 retimes forward, the top path has an available slot. However, the lower path cannot accept a retimed register. The retimed register is too close to an adjacent used register, causing hold time violations. The Compiler detects these short paths, and routes the registers to longer paths, as shown in the circuit on the right. This practice ensures that sufficient slots are available for retiming.

The following two examples address short paths:

Case 1: A design runs at 400 MHz. Fast Forward compile recommends adding a pipeline stage to reach 500 MHz and a second pipeline stage to achieve 600 MHz performance.

The limiting reason is the short path / long path. Add the two-stage pipelining the Compiler recommends to reach 600 MHz performance. If the limiting reason is short path / long path again, this means the Router reaches a limitation fixing the short paths in the design. At this point you may have already reached your target performance, or this is no longer the critical path.

Case 2: A design runs at 400 MHz. Fast Forward compile does not make any recommendations to add pipeline stages.

If the short path / long path is the immediate limiting reason for retiming, this means that the Router reaches a limitation in trying to fix the short paths. Adding pipeline stages to the reported path does not help. You must optimize the design.

Retiming registers that are close to each other can potentially trigger hold violations at higher speeds. The Compiler reports this situation in the retiming report under **Path Info**. The Compiler also reports short paths if enough Hyper-Registers are not available. When nodes involve both a short path and a long path, adding pipeline registers to both paths helps with retiming.

9 Document Revision History

This document has the following revision history.

Table 9. Document Revision History

Date	Version	Changes
2017.11.06	17.1.0	<ul style="list-style-type: none"> Revised <i>Design Example Walkthrough</i> steps and results. Provided link to available design example files for each stage. Moved step-by-step design compilation instructions to <i>Design Compilation</i> chapter, Intel Quartus Prime Pro Edition Handbook. Added <i>Ternary Adders</i> topic and examples. Added <i>Loop Pipelining</i> topic and examples. Added description of Reset Sequence Requirement report. Updated for latest Intel branding conventions.
2017.05.08	Quartus Prime Pro v17.1 Stratix 10 ES Editions	<ul style="list-style-type: none"> Updated software support version to Quartus Prime Pro v17.1 Stratix 10 ES Editions. Added Initial Power-Up Conditions topic. Added Retiming Reset Sequences topic. Added guidelines for high-speed clock domains. Added Fitter Overconstraints topic. Described Hold Fix-up in Fitter Finalize stage. Added statement about Fast Forward compilation support for retiming across RAM and DSP blocks. Added details on coherent RAM to read-modify-write memory description. Added description of Fast Forward Viewer and Hyper-Optimization Advisor. Added Advanced HyperFlex Settings topic. Added Prevent Register Retiming topic. Added Preserve Registers During Synthesis topic. Added Fitter Commands topic. Added Finalize Stage Reports topic. Replaced command line instructions with new GUI steps in compilation flows. Described concurrent analysis controls in Compilation Dashboard. Consolidated duplicate content and grouped Appendices together. Updated diagrams and screenshots.
2016.08.07	2016.08.07	<ul style="list-style-type: none"> Added clock crossing and initial condition timing restriction details. Described true dual-port memory support and memory width ratio with examples Updated code samples and narrative in Design Example Walk-through Added reference to provided Design Example files Re-branded for Intel Updated for latest changes to software GUI and capabilities.
2016.03.16	2016.03.16	First public release.

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Empirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.

ISO
9001:2008
Registered