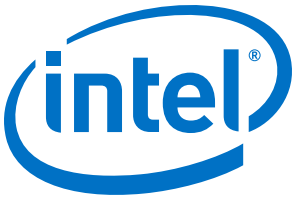
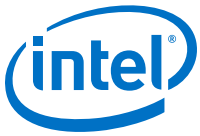


Stratix 10 High-Performance Design Handbook

S10HPHB
2016.08.07

Subscribe
Send Feedback

Contents



1 Introduction

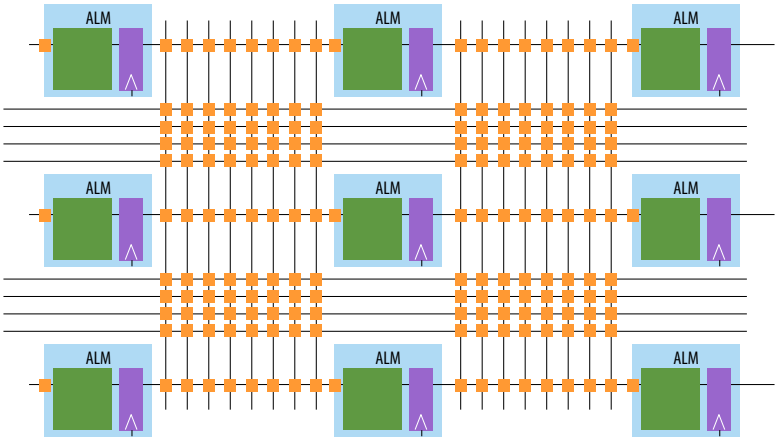
This document describes design techniques for achieving the highest performance with the Stratix® 10 HyperFlex® device architecture. The Stratix 10 architecture introduces new Hyper-Retiming, Hyper-Pipelining, and Hyper-Optimization design techniques for Intel FPGAs. Use these techniques to reach the highest clock frequencies in Stratix 10 devices.

- HyperFlex architecture overview—introduces the new Stratix 10 device architecture and supporting software features
- RTL design guidelines—provides fundamental high-performance RTL design techniques forStratix 10 designs
- Running Quartus® Prime Pro – Stratix 10 Edition Beta software—describes compiling for Stratix 10 devices and performance optimization features
- HyperFlex porting guidelines—design preparations to compile and optimize for Stratix 10 devices
- Design example walk-through—steps through the provided median filter Stratix 10design, analyzes the results, and demonstrates performance improvement techniques

1.1 HyperFlex Architecture Overview

The centerpiece of the Stratix 10 HyperFlex architecture is the innovative "registers everywhere" design. This architecture adds bypassable Hyper-Registers to every routing segment in the Stratix 10 device core, and at all functional block inputs.

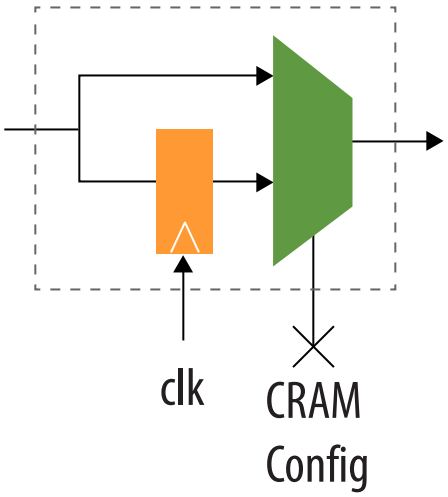
Figure 1. Registers Everywhere - HyperFlex Architecture



The Registers Everywhere illustration shows a small section of the Intel FPGA fabric, with nine ALMs and the interconnect routing that connects the ALMs. The squares at the intersection of each horizontal and vertical routing segment indicate the Hyper-Register location.

Figure 2. Bypassable Hyper-Register

The routing signal can bypass the register and go straight to the multiplexer, or go through the register first. One bit of the Intel FPGA configuration memory (CRAM) controls this multiplexer.

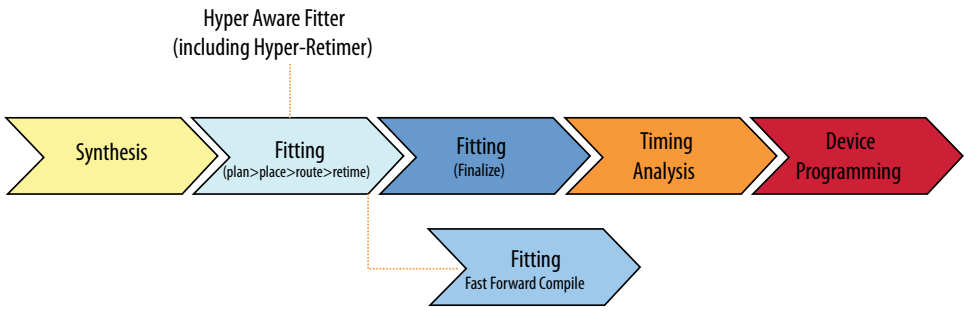


1.1.1 Hyper-Aware Design Flow

The Quartus Prime Pro – Stratix 10 Edition Beta software introduces a powerful suite of integrated capabilities to take full advantage of the HyperFlex architecture and maximize design productivity.
The Hyper-Aware Design Flow automatically optimizes designs to take advantage of the HyperFlex architecture. The Fitter anticipates the fine-grained Hyper-Retiming optimizations that occur in the subsequent retime stage of the Fitter. Therefore, you can focus the earlier stages of optimization on critical paths that do not benefit as much from retiming.

To improve design performance, enable more retiming optimizations by removing retiming restrictions (Hyper-Retiming), adding pipeline stages (Hyper-Pipelining) or modifying the structure of the design (Hyper-Optimization).

Figure 3. Hyper-Aware Design Flow



The Hyper-Registers allow you to use familiar design techniques to increase the performance of the design beyond other conventional FPGA architectures. As FPGA process geometries shrink, the interconnect delays between the ALMs become dominant and limit performance. Locating the Hyper-Registers in the interconnect routing—where they can best address this issue—is one of the key innovations of the HyperFlex architecture. When implementing these techniques in Hyper-Registers, rather than ALMs, the techniques are known as Hyper-Retiming, Hyper-Pipelining, and Hyper-Optimization.

Use this three-step process to maximize the performance of a design for the HyperFlex architecture:

Table 1. Three Steps to Maximize HyperFlex Architecture Performance

Step	Architecture Advantage	Required Changes	Core Performance (versus previous-generation high-performance Intel FPGA)
1	Hyper-Retiming	None, or minor RTL changes	1.4X
2	Hyper-Pipelining	Added pipelining	1.6X
3	Hyper-Optimization	Design dependent	2x or more

1.1.1.1 Hyper-Retiming

Hyper-Retiming improves the performance of critical paths by moving registers out of the ALMs and into the interconnect. This technique balances register-to-register delays, and allows the design to run at a faster clock frequency.
The Fitter automatically performs retiming optimizations to take advantage of the Hyper-Registers. However, you can enable further retiming optimizations by making the following types of changes to the RTL to remove retiming restrictions:

- Change the reset strategy to avoid using asynchronous clears
- Modifying timing constraints that prevent retiming

This process requires minimal effort, while resulting in an average performance gain of 1.4X for Stratix 10devices compared to previous generation high-performance FPGAs.

1.1.1.2 Hyper-Pipelining

Hyper-Pipelining 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.

After you modify the RTL and place the prescribed number of pipeline stages at the boundaries of each clock domain, the Hyper-Retimer automatically places the registers within the clock domain at the optimal locations to maximize the performance. The combination of auto-placement and Fast-Forward Compile helps to automate the process when compared with conventional pipelining. This process requires minimal effort, while resulting in an average performance gain of 1.6X for devices compared to previous generation high-performance FPGAs.

1.1.1.3 Hyper-Optimization

After accelerating data paths through Hyper-Retiming and Hyper-Pipelining, some designs face limitations of control logic, such as long feedback loops and state machines. To achieve higher performance, restructure such logic sections to use functionally equivalent feed-forward or pre-compute paths, rather than long combinatorial feedback paths.

The effort that Hyper-Optimization requires varies by design characteristics. However, the technique can result in performance gains in Stratix 10 devices.

1.1.1.4 Fast Forward Compile

Fast Forward Compile guides you through the performance optimization and identifies performance limiting areas of the design. Fast Forward Compile analyzes the design and provides detailed recommendations about removing retiming restrictions, about how many pipeline stages should be added on critical paths, and how the design may be limited by bottlenecks such as feedback loops in the RTL. Fast Forward compile provides a summary of the estimated fmax improvement for each of the recommended changes. Use Fast Forward Compile to easily predict the highest performance of your Hyper-Optimized design in a Stratix 10 device.

1.1.1.5 Hyper-Aware Algorithms

The Quartus Prime Pro – Stratix 10 Edition Beta software includes Hyper-Aware algorithms used during synthesis and place-and-route. These algorithms allow the Compiler to reduce logic resources by predicting which registers can move out of ALMs and into Hyper-Registers in the interconnect routing.

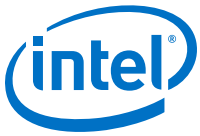
The suite of HyperFlex architecture features, combines with the Intel 14-nm Tri-Gate process technology, to enable Stratix 10 FPGAs and SoCs that deliver the highest levels of performance, density, and power efficiency in programmable logic.

1.2 Acknowledgments

Examples in this document include code from the following OpenCores projects:

- *10_100_1000 Mbps tri-mode Ethernet MAC*; by Jon Gao
- *128 bit AES Pipelined Cipher*; by Amr Salah
- *Turbo Decoder*; by David Brochart

The projects are distributed under the LGPL license, and the terms are reproduced below.



2 RTL Design Guidelines

This section recommends specific design techniques to achieve the highest clock rates possible with the HyperFlex architecture and the Hyper-Retimer. Most common techniques of high-speed design apply to designing for the HyperFlex architecture. In addition, you must use some new techniques to achieve the highest performance.

Follow these general RTL design guidelines to enable the Hyper-Retimer to optimize design performance:

- Design in a way that facilitates register retiming by the Hyper-Retimer.
- Use a latency-insensitive design that supports the addition of pipeline stages at clock domain boundaries, top-level I/Os, and at the boundaries of functional blocks.
- Restructure RTL to avoid performance-limiting loops.

2.1 HyperFlex Design Philosophy

The Stratix 10 architecture represents a big step forward in maximum clock rate compared to previous FPGA generations. Migrating to the Stratix 10 architecture generally requires a review of design best practices to obtain the most benefit from Stratix 10 FPGAs. However, increasing the speed of your circuitry can produce dramatic effects.

2.1.1 Set a High-Speed Target

For silicon efficiency, set your speed target as high as possible. The Stratix 10 LUT is essentially a tiny ROM capable of a billion lookups per second. Operating a 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

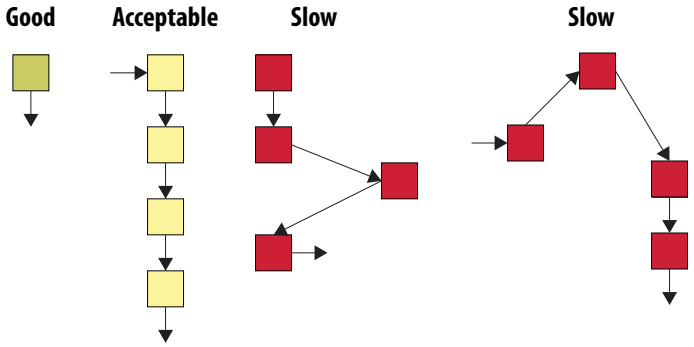
Timing closure difficulties occur when there is difference between what the circuit can naturally achieve, and the f_{MAX} requirement of your design. If the capability is sufficiently high, many possible placements are satisfactory, creating easy timing closure and short place-and-route runtime.

The timing in a slower circuit is not inherently easier to close than a faster one, because slow circuits tend to have large amounts of combinational logic between registers. When there are many nodes on a path, most possible placements involve

stretching nodes away from each other, resulting in significant routing delay. In contrast, a heavily pipelined circuit is much less dependent on placement, which makes closing timing easier, despite the higher speed.

Figure 4. Placement of Deep or Under-Pipelined Paths

Examples of acceptable and slow routing delays.



Be realistic about timing margin when building your design. An Intel FPGA comprises a common pool of physical resources. Portions of the design make contact and distort one other as logic is added to the system. Adding stress to the system is typically detrimental to speed. As a design project progresses, there are more ways to slow the system down than to speed it up. Allowing more timing margin at the start helps mitigate this problem.

2.1.1.2 Speed and Area or Latency

Running silicon at higher clock rates accomplishes more work with the same resources

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

	Circuit Function						
Bus Width (N)	**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

Most 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. Generally, the amount of additional registering required to go faster is significantly less than double, creating latency profit.

2.1.2 Experiment and Iterate

Experiment and iterate if your design's performance does not meet your requirements. The reprogrammability of Intel FPGAs allows you to experiment and optimize until you achieve your goals. Commonly, a design element's performance gradually becomes inadequate as requirements change over time. For example, when you apply the design element to a new context at a wider parameterization, perhaps the speed falls off.

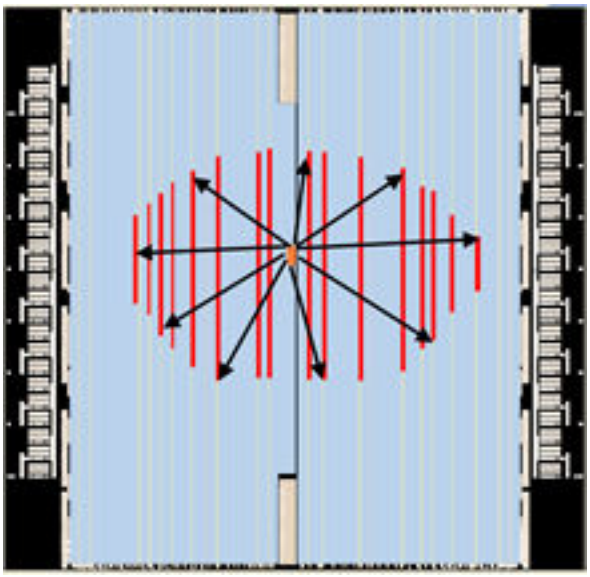
When experimenting with circuit timing, there is no permanent risk from experimentation that temporarily breaks the circuit to collect a data point. A common trick of experienced designers is to add registers illegally to determine the effect on overall timing. If the candidate circuit begins to meet the timing objective, you can make further investment to legalize the change. If a circuit remains too slow, even when taking considerable liberties with registers, you likely must reconsider more basic elements of the design. Moving up or down a speed grade, or compressing circuitry in LogicLock Plus regions, are other helpful methods for speed investigation.

2.1.3 Compile Components Independently

Compile the design subcomponents as stand-alone entities to find the trouble spots. Competition for resources and physical constraints (like pin locations) tend to slow the overall design performance. Once embedded 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 required component speed. For example, when targeting a 20% timing margin, a component with 19.5% margin is a failure. You can base the 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 5. M20K Spread in Chip Planner



The system level view does not show anything alarming about the resource ratios. However, there is a great deal of mysterious routing congestion. 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 fill up the intervening area. Restructuring this block to physically distribute the control logic better relieves the high-level problem.

Stand-alone compilation also prepares you for inevitable 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.

There are some caveats to this approach that you must be aware of to achieve the expected results for the entire design. Refer to the *Top-Level Considerations* section of the *HyperFlex Porting Guidelines* chapter for more information.

Related Links

Top-Level Design Considerations on page 98

2.1.5 Avoid Broadcast Signals

Avoid using broadcast signals whenever possible. Broadcast signals are typically high fan-out control nets and can create large latency differences between paths. This latency difference increases the difficulty for the Hyper-Retimer to find a suitable location for registers, and can result 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. When choosing your register duplication technique, use placement knowledge to maximize benefits. The following figures show two examples in which the design uses extra registers to help close timing. However, the locations of the extra registers make the second example more efficient than the first.

Figure 6. Adding a Pipeline Stage to Broadcast Signals

The yellow box indicates extra registers in a module you add to help with timing. The block broadcasts the output to several transceiver channels. Because the final register stage fans out to destinations that cover a large physical area of the chip, these extra registers may not improve timing sufficiently.

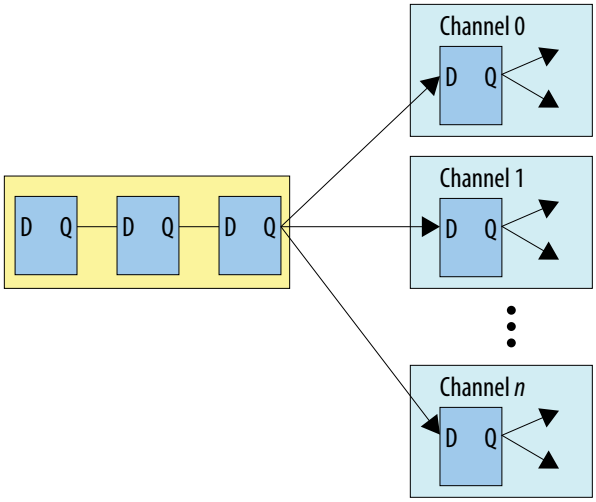
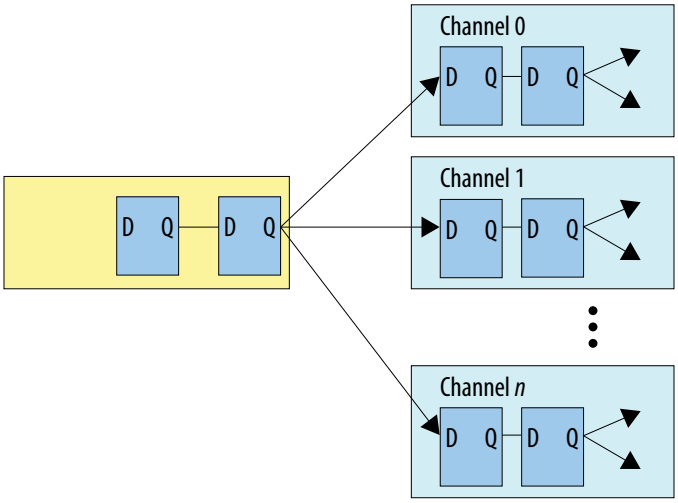


Figure 7. A Better Approach to Pipelining Broadcast Signals

A better approach to pipelining is to duplicate the last pipeline register and place a copy of the register in the destination module (the transceiver channels in this example). This method results in better placement and better timing. The improvement occurs because each channel's pipeline register is placed to help 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 a `dont_merge` synthesis attribute to avoid merging of the duplicate registers during synthesis, which eliminates any benefit.



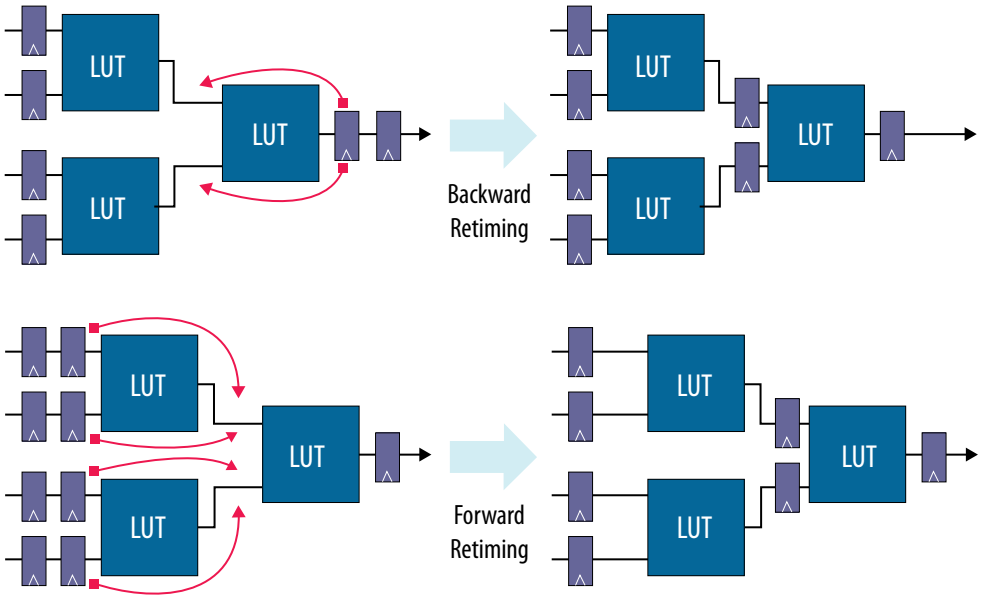
The recommendation to manually duplicate some registers may seem to contradict one of the benefits of the HyperFlex architecture—that you no longer need to manually insert pipeline registers in optimal locations in the RTL. The Hyper-Retimer runs after placement and routing and optimizes f_{MAX} performance for the placed and routed design. In some cases, such as this example, there are still steps you can take manually to help the placer get a better result. The manual duplication in this example helps the placer get a better result, and that result is then further optimized by the Hyper-Retimer.

2.2 Facilitate Register Movement (Hyper-Retiming)

This section discusses facilitating register movement in your design (Hyper-Retiming). The Hyper-Retimer balances register chains to increase f_{MAX} by retiming some ALM registers into Hyper-Registers in the routing fabric. The *retiming* refers to moving the physical location of design registers to balance the propagation delay between registers. Retiming also performs sequential optimization by moving registers backwards and forwards across combinatorial logic. Retiming across node splits and merges may involve register duplications or merges. By balancing the propagation delays between each stage in a series of registers, the retiming process shortens the critical paths, reduces the clock period, and increases the frequency of operation.

Figure 8. Moving Registers across LUTs

The left side represents the pre-retiming design, where the worst case delay is two LUTs. The right side represents the retiming design, which shows the worst case delay is one LUT.



When the Hyper-Retimer cannot retime a register, this is known as 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 situations that limit performance. Some limitations relate to hardware characteristics, software behavior, or the design. 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 the retimer.
- Use targeted 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 or don't touch register attributes. Refer to the *Retiming Restrictions and Workarounds* section.
- For information about adding pipeline registers, refer to the *Add Pipeline Registers (Hyper-Pipelining)* section.
- For information about addressing loops and other RTL restrictions to retiming, refer to the *Optimize RTL (Hyper-Optimization)* section.

Related Links

- Reset Strategies on page 16

This section recommends techniques to achieve maximum performance with resets.

- Timing Constraint Considerations on page 23
  This section recommends specific timing constraint techniques to maximize performance.
- esc1445881977928.xml
- Add Pipeline Registers (Hyper-Pipelining) on page 26
  This section discusses adding pipeline registers to increase performance.
- Retiming Restrictions and Workarounds on page 90
  This section describes RTL design techniques you can use to avoid retiming restrictions.

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 retime any register with an asynchronous clear to improve performance.

Using a synchronous clear instead of an asynchronous clear allows the Hyper-Retimer to retime 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 119

2.2.1.1 Removing Asynchronous Clears

You can 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.

The following figure shows Verilog HDL and VHDL examples of common circuits that implement asynchronous clears for the registers in a processing pipeline.

Figure 10. Verilog HDL and VHDL RTL Examples with Asynchronous Clears

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
```

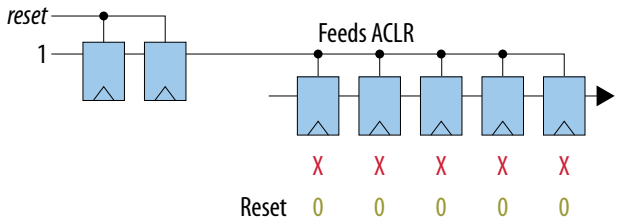
VHDL

```
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.

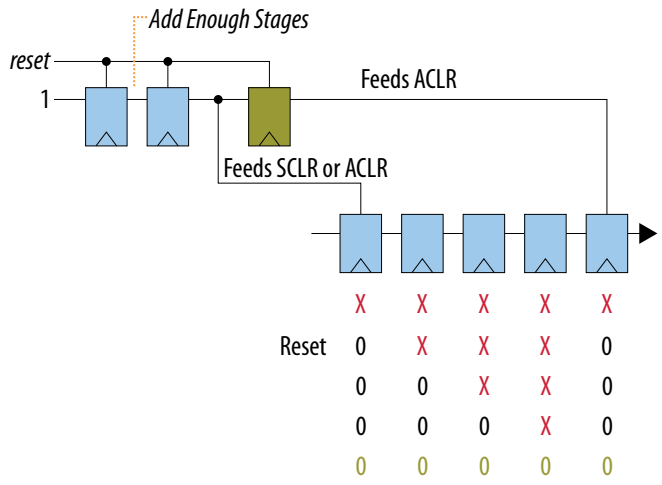
The following figure shows the same circuitry in schematic form, and shows the output behavior at reset.

Figure 11. Circuit Using Full Asynchronous Reset



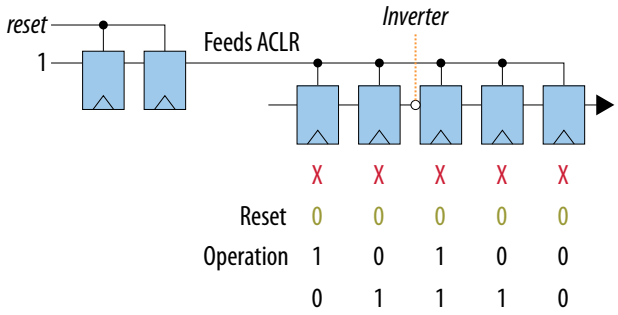
The following figure shows the removal of some of the asynchronous clears from the middle of the circuit. After a partial reset, if the modified circuit settles to the same steady state as the original circuit, then the modification is functionally equivalent.

Figure 12. Circuit Using Partial Asynchronous Reset



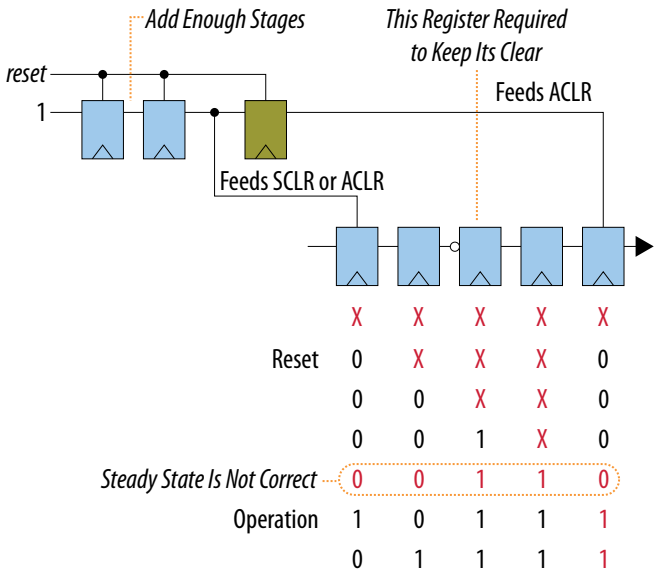
Cases involving inverting logic generally require additional synchronous clears to remain in the pipeline.

Figure 13. Circuit with an Inverter in the Register Chain



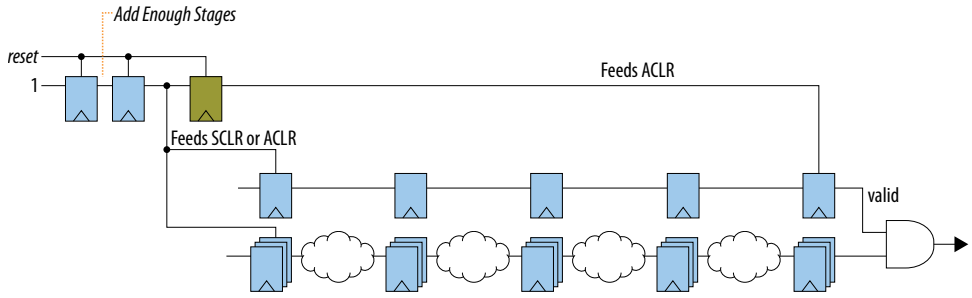
After removing the reset and applying the clock, the register outputs do not settle to the reset state, as in the circuit above. Rather, the inverting register cannot have its asynchronous clear removed to be equivalent to the above circuit after settling out of reset.

Figure 14. Circuit with an Inverter in the Register Chain with Asynchronous Clear



To avoid non-naturally resetting logic because of inverting functions, validate the output to synchronize with reset removal. Then, as long as 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.

Figure 15. Validating the Output to Synchronize with Reset



The next two figures show Verilog HDL and VHDL examples of the circuit shown in Figure 12 on page 18. You can apply this example to your design and remove unnecessary asynchronous resets.

Figure 16. Verilog HDL Example Using Minimal or No Asynchronous Clears

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;
```

Asynchronous clear for output register only

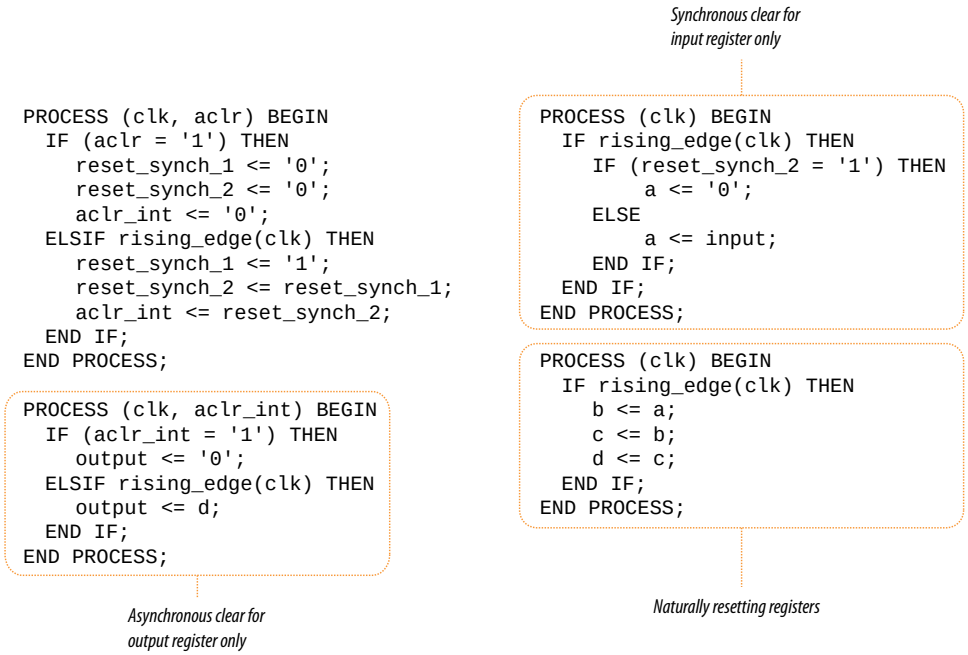
```
always @(posedge clk)
  if (reset_synch_2)
     a <= 1'b0;
  else
     a <= in;
```

Synchronous clear for input register only

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

Naturally resetting registers

Figure 17. VHDL Example Using Minimal or No Asynchronous Clears



2.2.1.2 Synchronous Clears on Global Clock Trees

Using a global clock tree to distribute a synchronous clear may limit the Hyper-Retimer's 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 Hyper-Retimer does not retime registers driving an output port or being driven by an input port. If a synchronous clear is on one of these I/O registers, you cannot retime the register. This restriction is not typical of practical designs in which logic drives resets. However, this issue may become apparent in benchmarking a smaller piece of logic, where the reset may come 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.

2.2.2 Clock Enable Strategies

High fan-out clock enable signals can limit the performance achievable by the Hyper-Retimer. This section provides recommendations for three situations where you can use 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, where 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 Section ➤ Control Signals** Compilation report and check the **Usage** column. Because the localized clock enable has a small fan-out, retiming it is quite easy and usually does not cause any timing issues.

2.2.2.2 High Fan-Out Clock Enable

Avoid a high fan-out signal 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 it disables. You can retime these extra registers into the logic if necessary.

Related Links

Synchronous Resets Summary on page 121

2.2.2.3 Clock Enable with Timing Exceptions

Another consideration is for clock enable logic accompanied by multicycle and false path (occasionally) timing constraints. 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. However, the Hyper-Retimer does not retime registers that are endpoints of these timing exceptions. 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, consider adding a register stage to the data path. Also consider removing 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 Hyper-Retimer does not retime registers with `preserve` or `dont_touch` attributes, because it respects the directive to prevent optimization. Consider whether you can remove the directives and allow the Hyper-Retimer to retime affected registers. If you preserve a register for debugging observability, consider keeping the `preserve` attribute. If you preserve a register to manage register duplication, consider using `dont_merge` instead.

If you use the `maxfan` synthesis attribute, there is a side effect of applying a `preserve` attribute to the duplicated registers, which prevents the Hyper-Retimer from retiming the registers. You can remove the `maxfan` attribute and evaluate the performance without the attribute. Alternatively, you can remove the attribute and specify the Maximum Fan-Out assignment in the Quartus Settings File (`.qsf`), which does not have the side effect.

2.2.4 Timing Constraint Considerations

This section recommends specific timing constraint techniques to maximize performance.

Synopsys Design Constraints (`.sdc`) and exceptions, such as false paths and multicycle paths, restrict the retiming optimizations of the Hyper-Retimer. The Hyper-Retimer does not retime registers that are the endpoints of an SDC constraint. Define any constraints or exceptions as specific as possible to avoid Hyper-Retimer restrictions.

2.2.4.1 Design Considerations for Multicycle Paths

This section describes special considerations for designs that include logic with multicycle exceptions. For example, your design may contain complex combinational logic that takes more than one clock cycle to process data. Some CRCs and arithmetic functions may include multicycle timing paths. You can reuse these modules and constraints unchanged in designs targeting Stratix 10 devices. However, the Hyper-Retimer does not retime registers that are endpoints of timing exceptions. Therefore, using actual register stages and removing the multicycle exception allows the Hyper-Retimer the most flexibility to improve performance.

For example, if you set up combinational logic with a multicycle exception of 3, you can remove the multicycle exception and insert two extra register stages before or after the combinational logic. This change allows the Hyper-Retimer to balance the extra register stages optimally through the logic.

2.2.4.2 Overconstraints

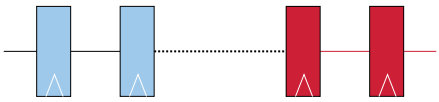
One timing closure technique is to add overconstraints to make the Fitter work harder on certain parts of a design. You may inadvertently limit the performance improvement if you use register-to-register overconstraints, because the Hyper-Retimer does retime registers that are the endpoints of an SDC constraint or exceptions. Overconstraints may be appropriate in some situations to improve performance. If you use overconstraints, do so sparingly as they may limit the achievable performance of the Hyper-Retimer. You may achieve higher performance without the overconstraint.

2.2.5 Clock Synchronization Strategies

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

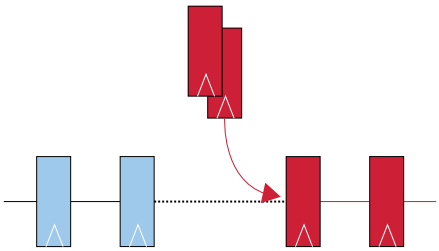
The following figure shows a straightforward synchronization scheme in which the path goes from one register of the first domain (blue), directly to a register of the next domain (red).

Figure 18. Simple Clock Domain Crossing



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

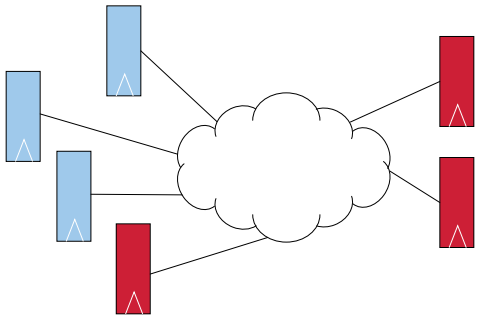
Figure 19. Simple Clock Domain Crossing After Adding Latency



The following figure shows a domain crossing structure that is not recommended for use in Stratix 10 designs, but may exist in designs that target other device families:

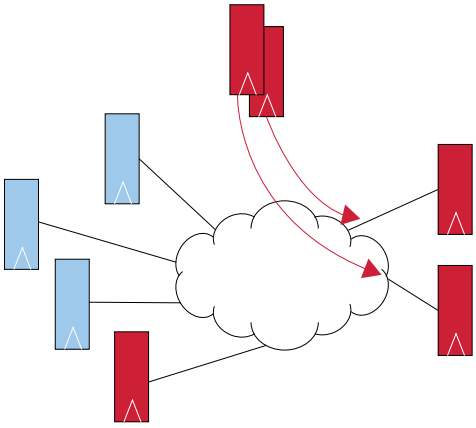
- The design contains some combinational logic between the blue clock domain and the red clock domain. This logic is not properly synchronized and you cannot add registers flexibly.
- The blue clock domain drives the combinational logic and the logic contains paths that are launched on the red domain.

Figure 20. Clock Domain Crossing at Multiple Locations



In this case, you can add latency at the boundary of the red clock domain as shown in the figure below, as long as you do not add registers on a red to red domain path. Otherwise, the paths become unbalanced, potentially breaking the design functionality. Although technically possible, it is risky to add latency in this scenario. Before doing so, thoroughly analyze the various paths.

Figure 21. Clock Domain Crossing at Multiple Locations After Adding Latency

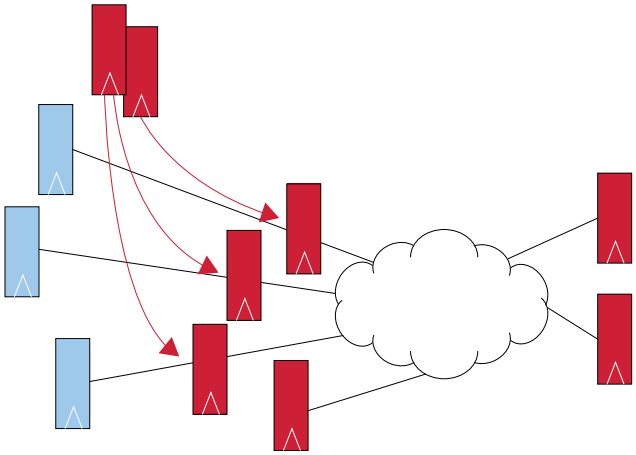


For Stratix 10 designs, synchronize the clock crossing paths before entering combinational logic. Adding latency is then more simple compared to the previous example.

In the following figure, the blue domain registers are synchronized to the red domain before entering the combinational logic. This design allows you to safely add extra pipeline registers in front of synchronizing registers without risking touching a red-red path inadvertently. This approach is the recommended synchronization method to take maximum advantage of the Stratix 10 architecture performance.

Figure 22. Improved Clock Domain Synchronization

Recommended synchronization method to take maximum advantage of the Stratix 10 architecture.



2.2.6 Synchronizers

The Quartus Prime compilation flow detects registers that are part of a synchronizer chain. The Fitter tries to optimize those registers to increase the mean time between failure (MTBF) for Metastability. The Hyper-Retimer does not retime the registers detected as part of a synchronizer chain. Therefore, to provide more flexibility for retiming, consider adding more pipeline registers at clock domain boundaries.

2.3 Add Pipeline Registers (Hyper-Pipelining)

This section discusses adding pipeline registers to increase performance. A proven way to increase f_{MAX} is to have a lot of registers and evenly distribute them throughout the circuit. At the high performance limit, a circuit has only one level of combinational logic between registers. If the register-to-combinational logic ratio is sufficient, the retimer can balance the registers to produce a design that is close to this ideal circuit.

Use Fast Forward Compilation during the design cycle to identify circuit boundaries that benefit from additional pipeline stages. Adding registers is much easier if you plan ahead to accommodate additional latency in your design. For more information about Fast Forward Compile, refer to the *Using Fast Forward Compilation* section.

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. For example, if Fast Forward Compile recommends adding two pipeline stages at an input bus, you can adjust a parameter and recompile. Refer to the *Appendix: Pipelining Examples* section for pre-written parameterizable pipeline modules in Verilog HDL, VHDL, and SystemVerilog.

Changing latency is more complicated than just adding pipeline stages. You might have to rework control logic, and other parts of the design or system software, to work properly with data arriving later. Making such changes could be difficult in existing RTL, but it may be easier in new parts of a design. Rather than hard-coding block latencies into control logic, try to make some of them parameters. In some types of systems, you may be able to add a "valid data" flag to pipeline stages in a processing pipeline and use that 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 be able to accommodate changes in latency.

Related Links

- Using Fast Forward Compilation on page 65
  This section describes using Fast Forward Compilation to guide you through the performance optimization process.
- Appendix: Parameterizable Pipeline Modules on page 50

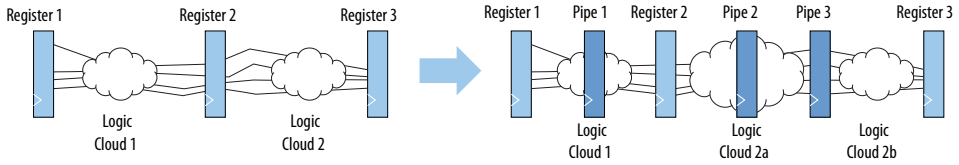
2.3.1 Conventional versus Hyper-Pipelining

This section describes how Hyper-Pipelining simplifies this process of conventional pipelining.

Conventional pipelining applies the following design modifications:

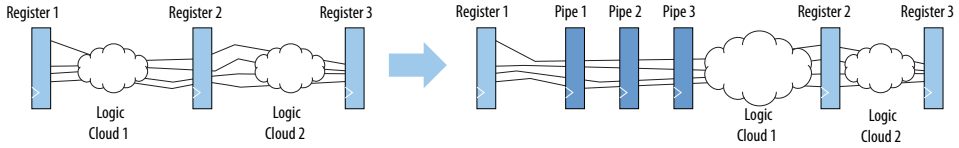
- Add two registers between logic clouds
- Modify HDL to insert a third register (or pipeline stage) into the design's logic cloud, which is Logic Cloud 2. This register insertion effectively creates Logic Cloud 2a and Logic Cloud 2b in the HDL

Figure 23. Conventional Pipelining User Modifications



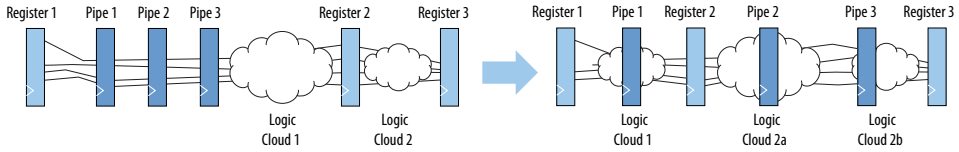
Hyper-Pipelining simplifies this process of adding registers. For the same design illustrated in the above figure, you add the registers—Pipe 1, Pipe 2, and Pipe 3—in aggregate at one location in the design RTL. Then, during design compilation, the Hyper-Retimer retimes the registers throughout the circuit to find the optimal placement along the path, as shown in the following figure. This optimization reduces path delay and maximizes the design's operating frequency.

Figure 24. Hyper-Pipelining User Modifications



The following figure shows implementation of additional registers after the Hyper-Retimer compilation stage has completes optimization of the design.

Figure 25. Hyper-Pipelining and Hyper-Retimer Implementation



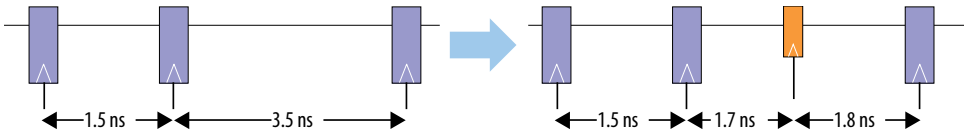
The resulting implementation in the Hyper-Pipelining flow differs from the conventional pipelining flow by the location of the Pipe 3 register. Because the Hyper-Retimer is aware of the current circuit implementation, including routing, it can more effectively locate the added aggregate registers to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

2.3.2 Pipelining and Latency

Adding pipeline registers in a design increases the number of clock cycles necessary for a signal value to propagate through the design. Increasing the clock frequency can offset the increased latency.

Consider a design for a previous generation FPGA, with a 275 MHz f_{MAX} requirement. In the following figure, the path on the left achieves 286 MHz, limited by the 3.5 ns delay. Data takes three cycles to propagate through the register pipeline. Three cycles at 275 MHz is 10.909 ns to propagate through the pipeline.

Figure 26. Hyper-Pipeline Reduced Latency



Assume that the design is being retargeted to a Stratix 10 device, and the f_{MAX} requirement has doubled to 550 MHz. The path on the right in the above figure shows an additional pipeline stage that has been added and retimed, and the path now achieves 555 MHz, limited by the 1.8 ns delay. The data takes four cycles to propagate through the register pipeline. Four cycles at 550 MHz is 7.273 ns to propagate through the pipeline.

If your goal is to maintain the time to propagate through the pipeline with four stages compared to three, you could meet the 10.909 ns delay of the first version by increasing the f_{MAX} of the second version to 367 MHz, a 33% increase from 275 MHz.

2.3.3 Use Registers Instead of Multicycle Exceptions

Sometimes designs contain modules with complex combinational logic (such as CRCs and other arithmetic functions) that can take more than one clock cycle to process. These modules are constrained with multicycle exceptions, to relax the timing requirements through the block. You can reuse these modules and constraints unchanged in designs targeting Stratix 10 devices. Refer to the *Design Considerations for Multicycle Paths* section for more information.

You can insert a number of register stages in one convenient place in a module, and the Hyper-Retimer 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 Hyper-Retimer can balance them.

Related Links

- Design Considerations for Multicycle Paths on page 23
  This section describes special considerations for designs that include logic with multicycle exceptions.
- Appendix: Parameterizable Pipeline Modules on page 50

2.4 Optimize RTL (Hyper-Optimization)

This section describes multiple RTL restructuring techniques to improve performance.

2.4.1 Deciding When to Rewrite the RTL

Often you have an existing RTL design that requires optimization to meet a new performance target. How do you decide between modifying or completely rewriting the RTL? The Quartus Prime Fast Forward Compile feature helps you make the decision. For information about running Fast Forward Compile, refer to the *Using Fast Forward Compilation* section.

If even Fast Forward Compile performance is significantly below your target, consider redesigning parts of your circuit. If the Fast Forward Compile performance is above your target, it is feasible to achieve the target, although you may be unable to implement some of the recommendations required to achieve this performance.

When you decide to rewrite the RTL, plan an appropriate implementation. Because adding pipelining stages is a common Hyper-Optimization step, choose a design style that allows you to accommodate varying amounts of latency. Using parameters to define and express the latency of blocks can be an effective way to retain design flexibility.

For more suggestions, refer to the *Experiment and Iterate* section.

Related Links

- Using Fast Forward Compilation on page 65
  This section describes using Fast Forward Compilation to guide you through the performance optimization process.
- Experiment and Iterate on page 11
  Experiment and iterate if your design's performance does not meet your requirements.

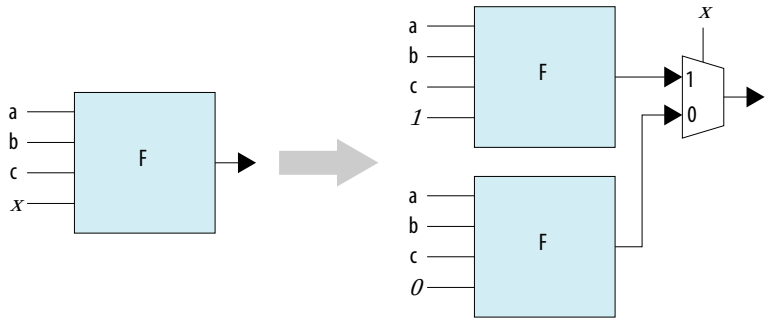
2.4.2 General Optimization Techniques

Use the general RTL techniques this section describes to optimize the design for the HyperFlex architecture and the Hyper-Retimer.

2.4.2.1 Shannon's Decomposition

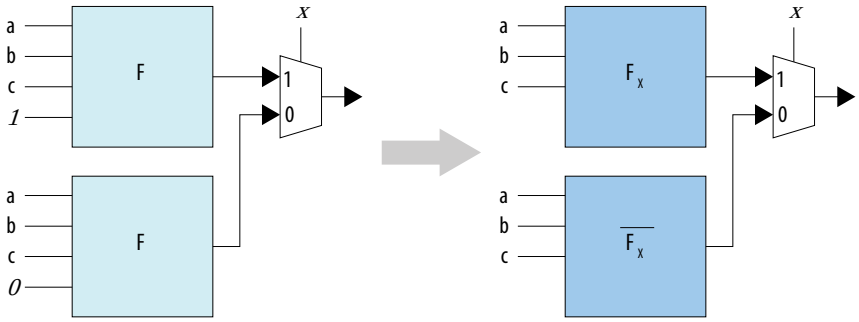
Shannon's decomposition plays a role in Hyper-Optimization. Shannon's decomposition, or Shannon's expansion, is a way of factoring a Boolean function. You can express a function as $F = x.F_x + x'F_x'$ where $x.F_x$ and $x'F_x'$ are the positive and negative co-factors of the function F with respect 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.

Figure 27. Shannon's Decomposition



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

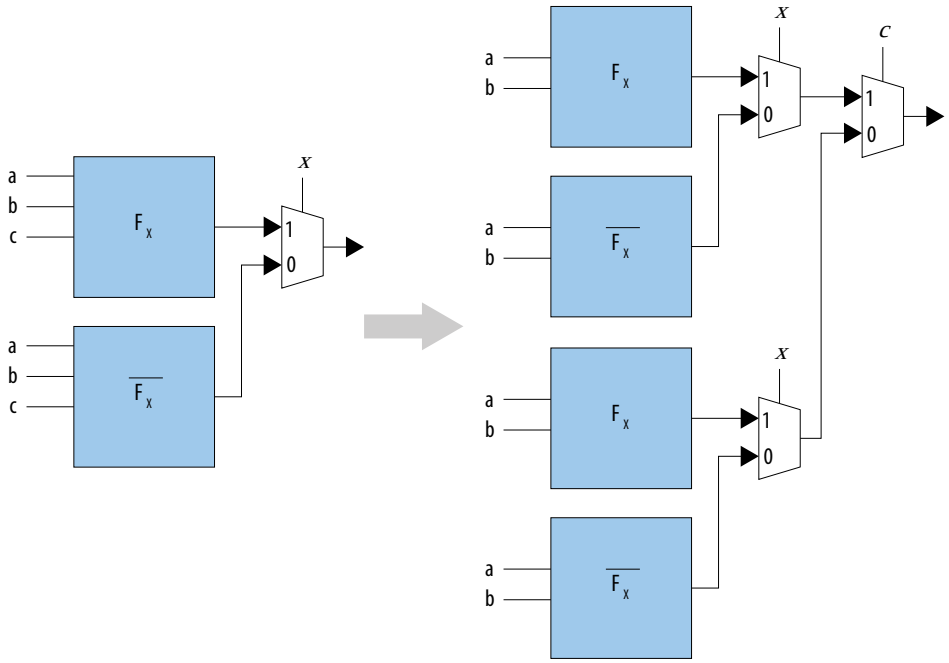
Figure 28. Shannon's Decomposition Logic Reduction



In Hyper-Optimization, the advantage of Shannon’s decomposition is that it 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.

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.

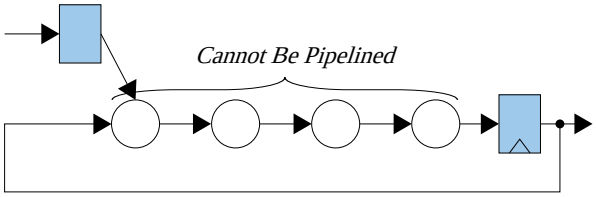
Figure 29. Repeated Shannon's Decomposition



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 Hyper-Retimer can now pipeline the logic moved outside the loop.

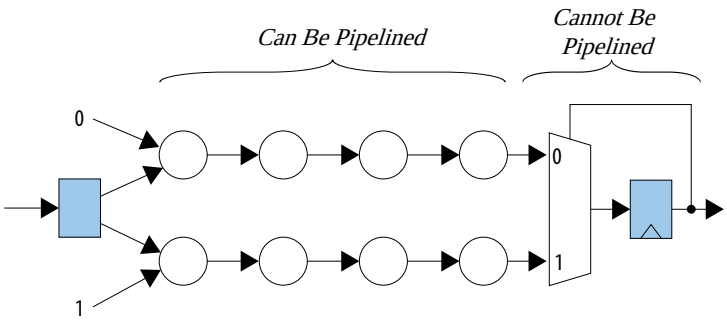
The following diagram shows a loop that contains a single register, four levels of combinational logic, and an additional input. Adding registers in the loop changes the functionality, but you can move the combinational logic outside the loop by performing Shannon's decomposition.

Figure 30. Loop Example before 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. The register in the loop then selects one of the two copies, as shown in the following diagram.

Figure 31. Loop Example after Shannon's Decomposition



Performing Shannon's decomposition on the logic in the loop reduces the amount of logic in the loop. The Hyper-Retimer can now perform Hyper-Retiming or Hyper-Pipelining on the logic removed from the loop, and increase the circuit performance.

2.4.2.1.1 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 1. 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 32. Fast Forward Compile Report before Shannon's Decomposition

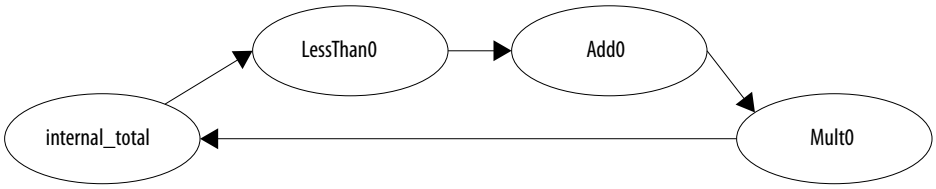
Fast Forward Summary for Clock Domain clk

	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 #1	29, including 1 pipeline stage	248.94 MHz	-3.017	0.970	Loop
3	Hyper-Optimization	29, including 1 pipeline stage	--	--	0.970	Loop

The Hyper-Retimer achieves 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 33. 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 2. 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 34. Fast Forward Compile 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

2.4.2.1.2 Identifying Circuits for Shannon's Decomposition

The circuits in which you can rearrange many inputs to control the final select stage are good candidates for Shannon's decomposition. Be aware of the new logic depths while 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 it is difficult to perfectly balance the number of inputs feeding each cloud of logic.

Another candidate for Shannon's decomposition is 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.

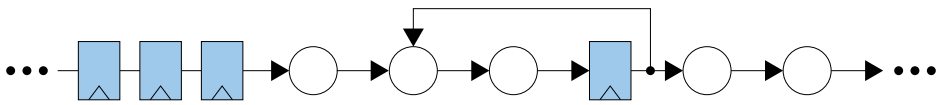
2.4.2.2 Time Domain Multiplexing

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

This technique replaces each register in a circuit by a set of C registers in series. Each extra copy of registers creates a new computation thread. One computation through the modified design takes C times as many clock cycles as the original circuit. However, the Hyper-Retimer 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.

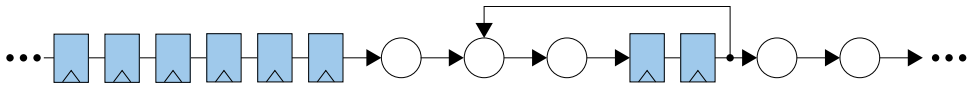
The following set of diagrams shows the process of C-slow retiming, beginning with an initial circuit.

Figure 35. C-slow Retiming Starting Point



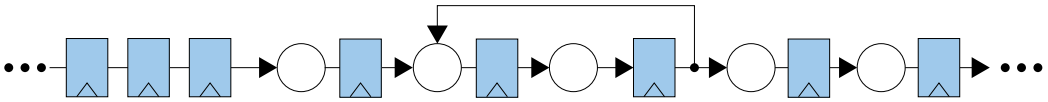
You edit the RTL design to replace every register, including registers in loops, with a set of C registers, made up of one register per independent thread of computation. If you replaced each register with two registers, the circuit is as shown in the following figure.

Figure 36. C-slow Retiming Intermediate Point



Compile the circuit at this point. When the Hyper-Retimer optimizes the circuit, it has more flexibility to perform Hyper-Retiming with the additional registers. The optimized circuit is as shown in the following figure.

Figure 37. 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.

2.4.2.2.1 Identifying Time Domain Multiplexing Optimization Opportunities

Use time domain multiplexing when a design includes multiple parallel threads, each limited by a loop. The module being optimized must not be sensitive to latency.

2.4.2.3 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.

2.4.2.4 Precomputation

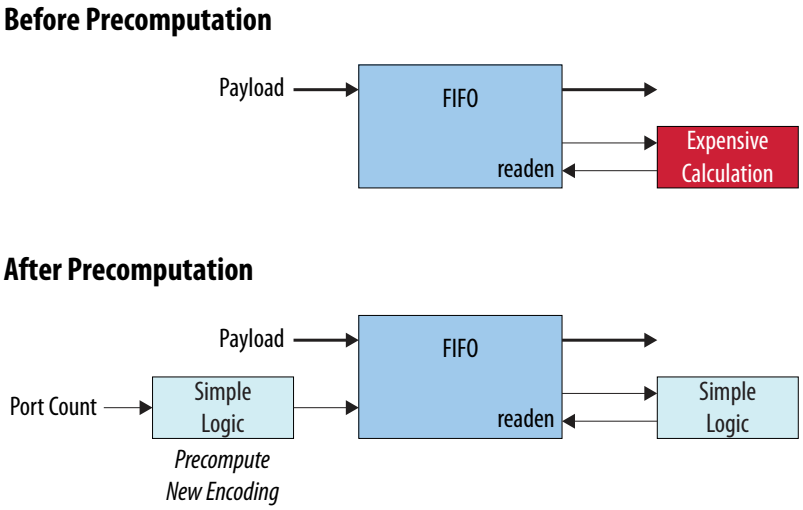
Precomputation is one of the easiest and most beneficial techniques for optimizing overall design speed. When confronted with critical logic, verify whether the signals the computation implies are available earlier. Always compute signals as early as possible to keep these computations outside of critical 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. Registers inside the loop cannot be moved outside of it; registers outside the loop cannot be retimed into the loop. Therefore, keep the logic inside the loop as small as possible so that it does not negatively impact your f_{MAX}.

The following figure shows a FIFO block diagram before and after precomputation. The diagram before precomputation depicts a payload going through a FIFO to some computational logic. The computational logic then sends control signals back to the FIFO for processing. If the loop created by the FIFO and calculation logic is large and depends on multiple signals, instead precompute the results to minimize the calculation logic.

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 it with pipelining or retiming. You cannot remove the loop, but can better control the effect of the loop on the design speed.

Figure 38. Restructuring a Design with an Expensive Loop



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 it 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;
```

2.4.3 Specific Design Structures

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

2.4.3.1 Restructuring Loops

Many restructuring techniques target loops, which fundamentally limit performance. A loop is a feedback path in a circuit. Loops may be simple and short, with a small amount of combinational logic on a feedback path. Loops may be much more complex, potentially going through multiple other registers on the way back to the original register. All useful circuits contain loops.

Loops limit performance because adding pipeline stages in a loop changes its functionality. The Hyper-Retimer never retimes registers into a loop. However, you can change RTL to restructure loops to improve performance. Perform loop optimizations after analyzing performance bottlenecks with Fast Forward Compile. You can also apply these techniques proactively as you write new RTL, to maximize performance potential. You can also use these techniques to improve performance in existing RTL, especially when you review performance bottlenecks in the Hyper-Retimer reports.

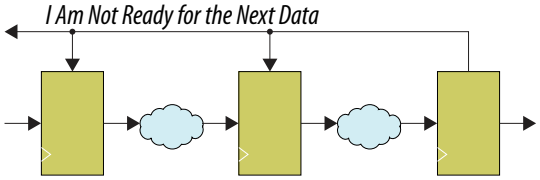
The following sections describe these techniques.

2.4.3.2 Control Signal Backpressure

This section describes RTL design techniques to control signal backpressure. The Stratix 10 architecture is extremely efficient at streaming 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 has the side effect of increasing backpressure on control signals. Data must flow forward as much as possible.

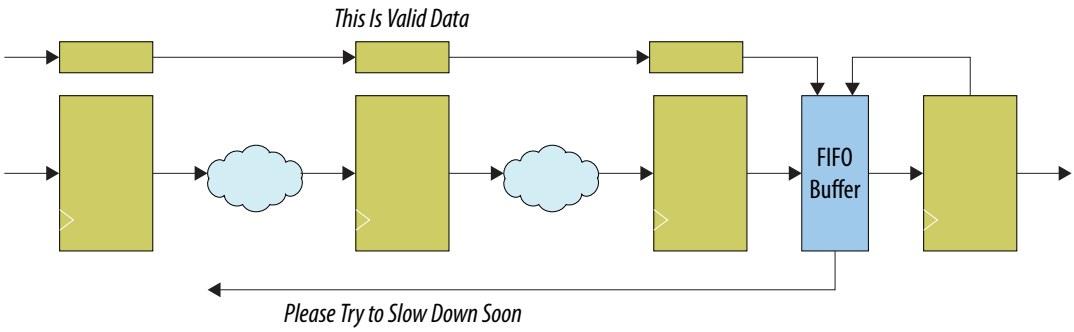
Figure 39. Control Signal Backpressure

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.



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 40. 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.

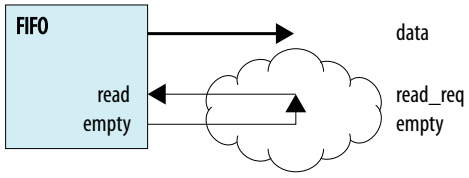
2.4.3.3 Flow Control with FIFO Status Signals

Because of the high clock speeds achievable by Stratix 10 devices, use extra care when dealing with flow control signals. This practice is particularly important with signals that gate a data path in multiple locations at the same time. For example, this practice is important with clock enable or FIFO full/empty signals. Instead of working with immediate control signals, use a delayed signal. Looking at the FIFO example, one can build a buffer within the FIFO block. The control signals indicate to the upstream data path that it 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 41. FIFO Flow Control Loop

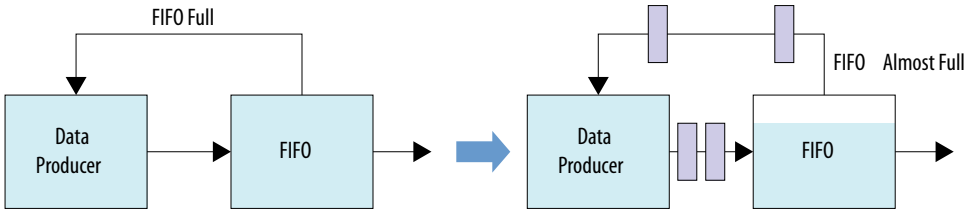
The loop is formed while reading control signals from the FIFO.



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

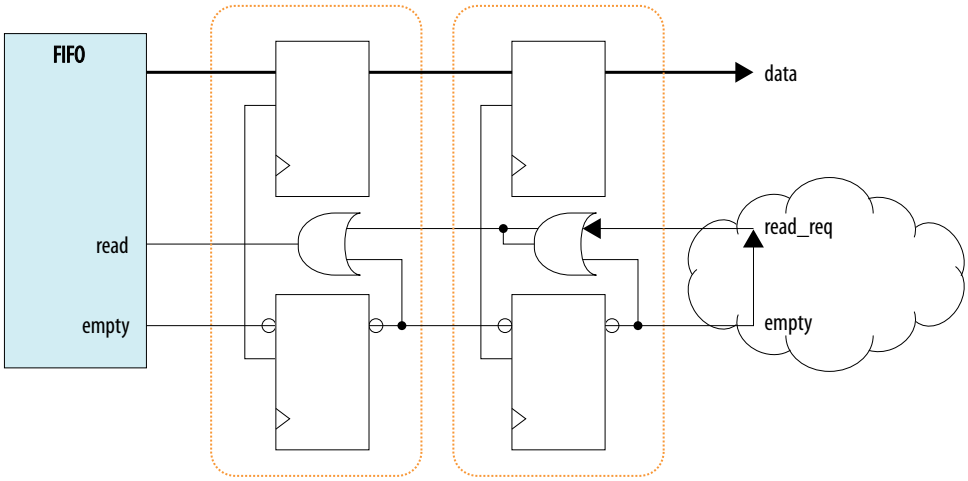
The following example has two extra registers in the full control flow signal. When the FIFO block signals that it is nearly full, the circuit requires two clock cycles to stop the data flow. Size the FIFO block to allow for proper storage of those extra valid data. The extra two pipeline registers in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

Figure 42. Improved FIFO Flow Control Loop with Almost Full instead of Full FIFO



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 43. 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, you should carefully consider whether there is a way to eliminate the broadcast signals. Pipeline broadcast control signals, and use almost full and almost empty status bits from FIFOs.

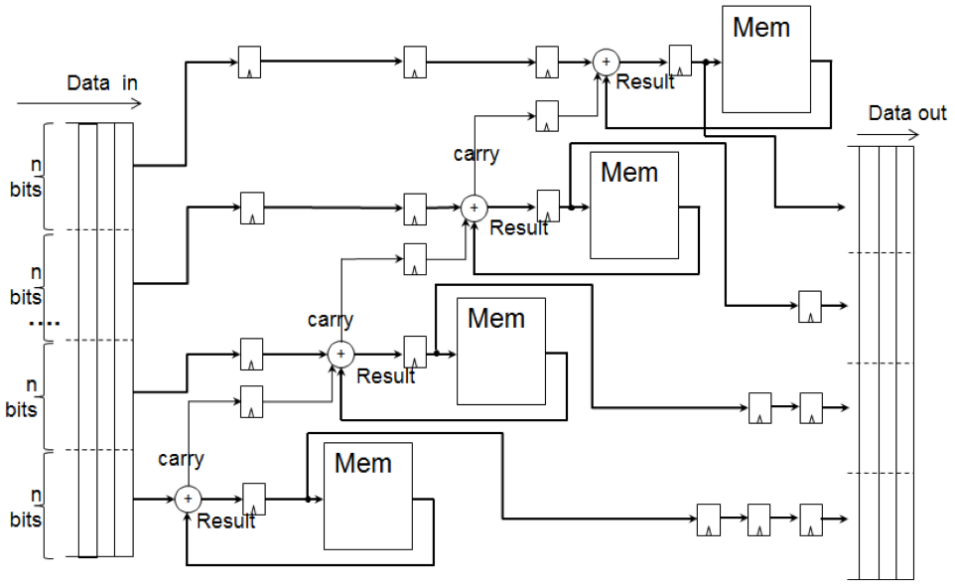
2.4.3.4 Read-Modify-Write Memory

Wireline networking applications often require updating counter values every clock cycle. This process is challenging because the number of queues and counters increases along with bandwidth requirements. If you pipeline a counter update over multiple clock cycles, maintain a cache to track recent activity with a particular counter. As you increase the number of cycles to update the value, the amount of logic required to handle caching and data combination increases, and can be difficult to scale. Loops are inherent in the logic for the cache, and they can be difficult to optimize.

A technique to improve performance, without extra complexity, is to break the modification operation into smaller blocks that can be completed in one clock cycle.

Figure 44. Pipelining Read-Modify-Write Memory

This figure shows a method to pipeline a read-modify-write memory to improve performance without maintaining a cache for tracking recent activity.



Data words are split into multiple n-bit chunks, where each chunk is small enough to be processed efficiently in one clock cycle. For the best performance, each chunk must be no wider than one M20K memory block.

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 unchanged regardless of the number of pipeline stages. If you need higher f_{MAX}, increase the number of memory blocks, use narrower memory blocks, and increase the latency. If you need lower latency, use fewer, wider memory blocks, and remove pipeline stages appropriately.

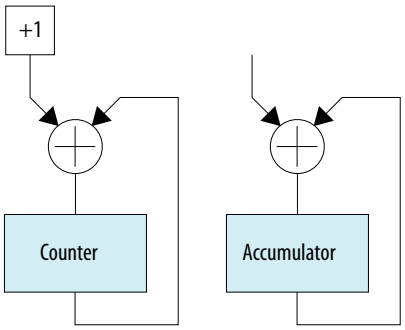
2.4.3.5 Counters and Accumulators

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 it 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 45. 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.



2.4.3.6 State Machines

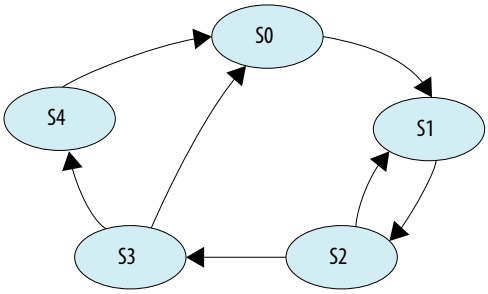
Loops due to state machines can be difficult to optimize. Carefully examine the state machine logic to determine whether you can precompute any signals used in the next state logic.

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

To optimize state machine loops, sometimes it may be necessary to write a new state machine.

Figure 46. State Machine Loop

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



Related Links

Precomputation on page 35
Precomputation is one of the easiest and most beneficial techniques for optimizing overall design speed.

2.4.3.7 Memory

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

2.4.3.7.1 True Dual-Port Memory

Stratix 10 devices support true dual-port memory structures. True Dual Port Memories allow you to perform two write and two read operations at once.

Stratix 10 embedded memory components (M20k) have slightly different modes of operation compared to previous device technology. Consider differences in the following areas for Stratix 10 designs:

- True dual-port memory
- Mixed-width ratio for read/write access

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

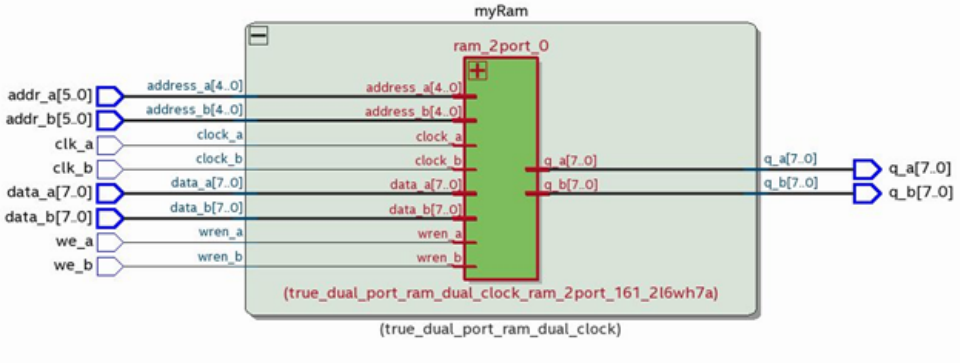
2.4.3.7.2 Use Simple Dual-Port Memories

When migrating a design to a 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, you can restructure it using two simple dual clock memories.

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

Figure 47. Arria® 10 True Dual-Port Memory Implementation

Quartus Prime Pro Edition version 16.1 generates this true dual-port memory structure for Arria® 10 devices.



Example 3. 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];
```

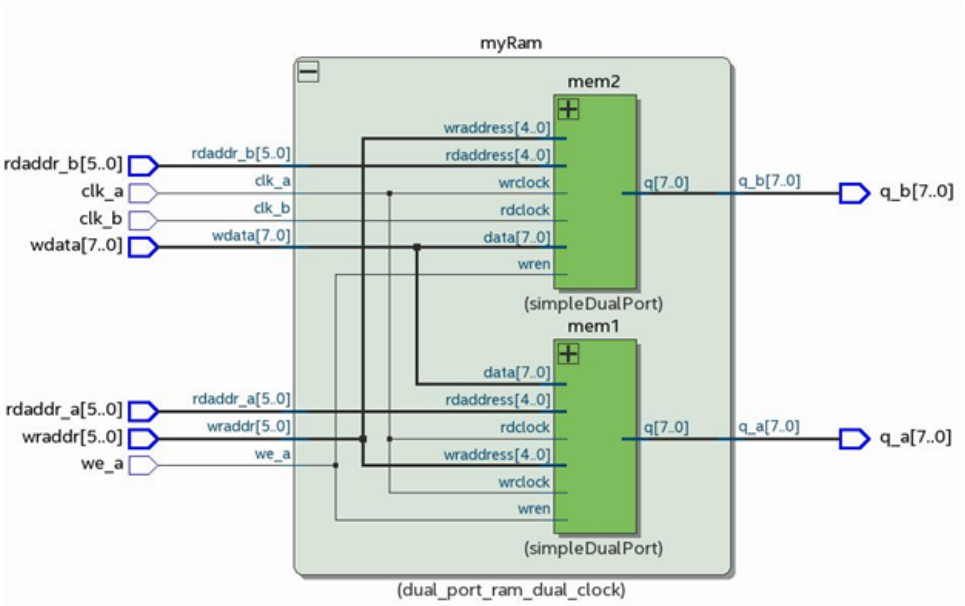
```
    always @ (posedge clk_a)
    begin
        // Port A
        if (we_a)
        begin
            ram[addr_a] <= data_a;
            q_a <= data_a;
        end
        else
        begin
            q_a <= ram[addr_a];
        end
    end

    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.

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 Stratix 10 devices.

Figure 48. 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 Quartus Prime Pro – Stratix 10 Edition Beta GUI.

Example 4. 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 5. 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
```

Memory Mixed Port Width Ratio Limits

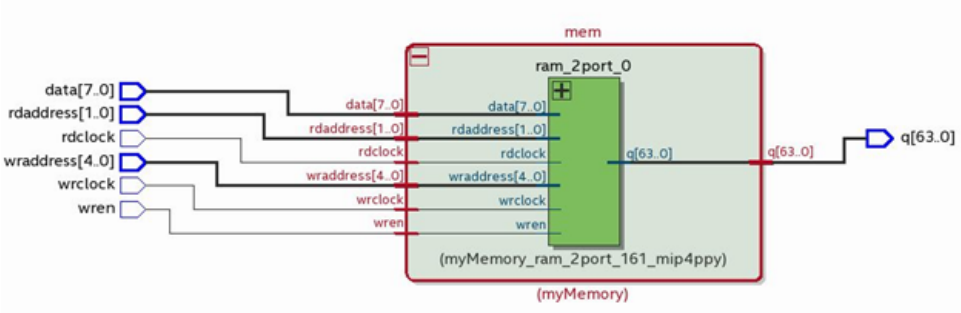
To enable clocks speeds of up to 1GHz, Stratix 10 devices block RAMs on silicon are different from previous generation devices.

The new RAM block design is more restrictive with respect to use of mixed ports data width. The redesigned Stratix 10 device block RAMs do not support 1/32, 1/16, or 1/8 mixed port ratios. The only valid ratios are 1, ½, and ¼ mixed port ratios. The Quartus Prime Pro – Stratix 10 Edition Beta generates an error message for implementation of invalid mixed port ratios.

If you are migrating a design that used invalid port width ratios for Stratix 10 devices, modify the RTL to create the desired ratio. starting from components with valid ratios.

.

Figure 49. Dual-Port Memory with Invalid 1/8 Mixed Port Ratio

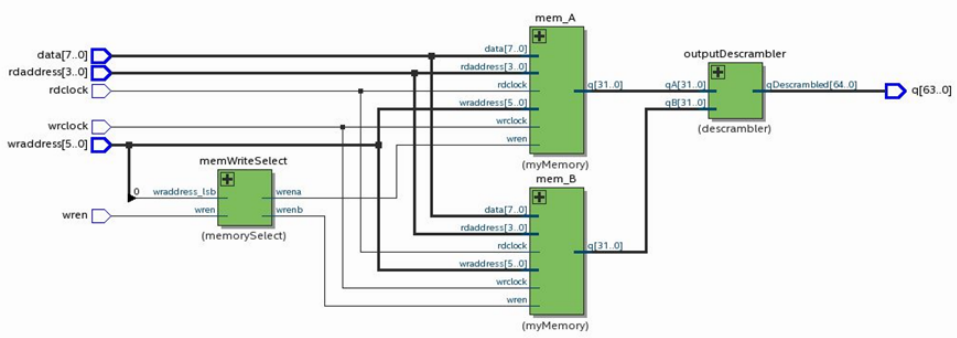


To create a functionally equivalent design for Stratix 10 devices, create and combine smaller memories with valid mixed port width ratios. For example, the following steps implement a mixed port width ratio:

1. Create two memories with ¼ mixed port width ratio by instantiating the 2-Ports memories from the IP Catalog.
2. Define some 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 50. 1/8 Width Ratio Example

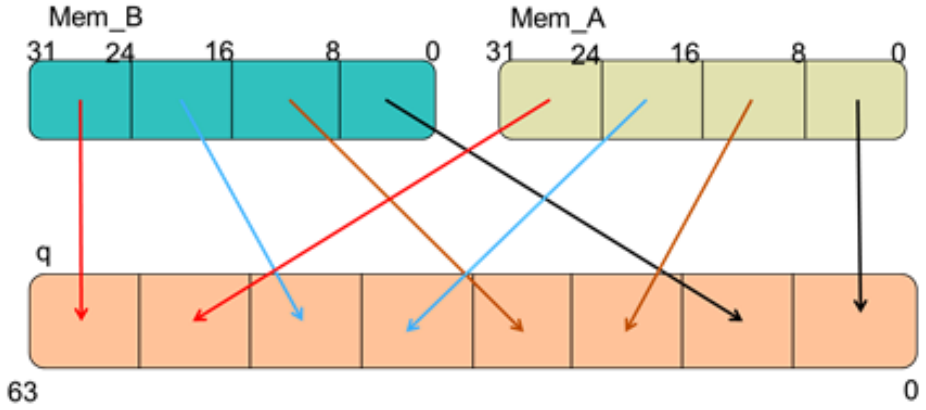
This example shows the interleaving of two memories and the write logic. The chosen write logic uses the least significant bit of the write address to decide which memory to write. Even addresses write in memory `mem_A`, odd addresses write in memory `mem_B`.



Because of the scheme that controls writing to the memories, you must take care in reconstructing 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 51. 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 6. 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
)(
    data, wraddress, rdaddress,    wren,
    wrclock, rdclock,    q
);

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),
    .wrclock(wrclock),
    .rdclock(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 7. 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
```

2.4.3.7.3 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 characterized by 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. This mode is implemented with soft logic outside 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.

2.4.3.8 DSP Blocks

DSP blocks support frequencies up to 1 GHz, but you must use all of the registers (two input registers and the output register).

2.4.3.9 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 the Hyper-Retimer can retime through the cloud of logic.

2.4.3.10 Modulus and Division

The modulus and division operators are costly in terms of area and speed, unless they use powers of two. Carefully consider whether there is an alternate 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 111

2.4.3.11 Resets

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

2.4.3.12 Hardware Re-use

One of the most effective ways to deal with loops due to hardware re-use is to unroll the loops.

2.4.3.13 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

2.4.3.14 FIFOs

FIFOs always contain loops, but there are some efficient ways to implement the internal FIFO logic to 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 48

2.5 Appendix: 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 as part of a latency-insensitive design style. You can readily change the latency of your circuit with these blocks.

Example 8. 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 9. Parameterizable Hyper-Pipelining Verilog HDL Instance

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

Example 10. 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 11. Parameterizable Hyper-Pipelining SystemVerilog Instance

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

Example 12. 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;
            dout <= HR(HR'high);
        end generate hp;

    end arch;
```

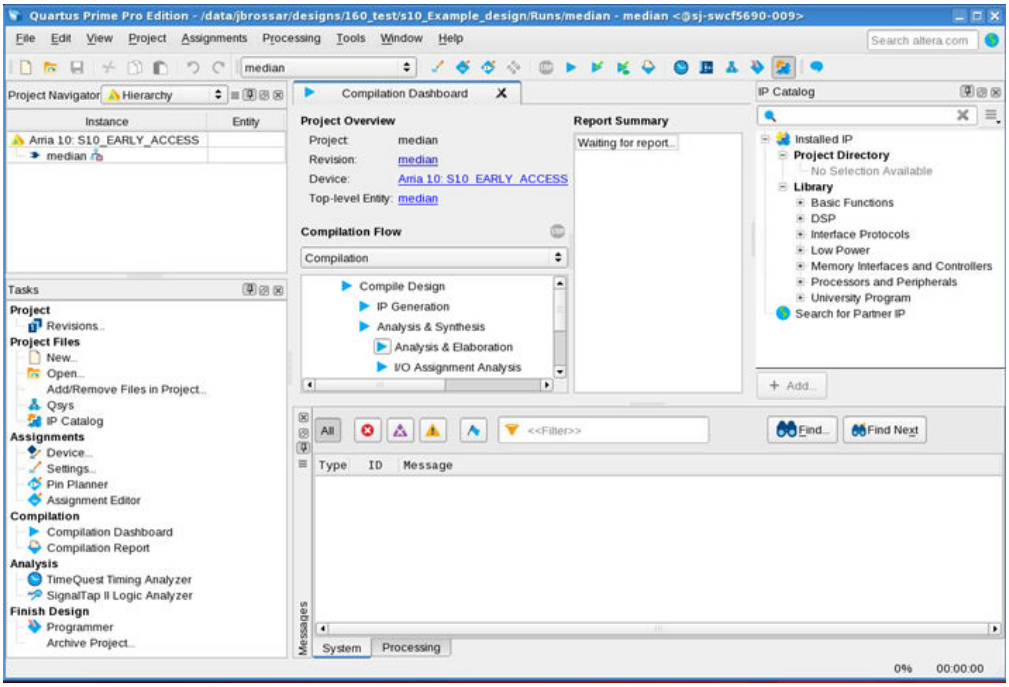
Example 13. Parameterizable Hyper-Pipelining VHDL Instance

```
-- 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 =>
        );
```

3 Running the Quartus Prime Pro – Stratix 10 Edition Beta Software

This document describes use of the Hyper-Retimer and Fast Forward Compile features for architecture evaluation, performance exploration, and timing closure in the Quartus Prime Pro – Stratix 10 Edition Beta software. These features include the new Hyper-Retimer and Fast Forward Compile optimization steps that enable the highest performance of the Stratix 10 HyperFlex architecture.

Figure 52. Quartus Prime Pro – Stratix 10 Edition Beta GUI



- ***Hyper-Retimer***—retimes registers during fitting to optimize f_{MAX} performance and balance the delay between register stages. The Hyper-Retimer can retime at each routing element in the HyperFlex fabric, while maintaining the functionality of your design.
- ***Fast Forward Compile***— identifies bottlenecks in the design structure that limit performance, recommends RTL changes to break bottlenecks, and predicts the design performance after recommended RTL changes.

3.1 System Requirements

To experience the fastest compilation time, install the Quartus Prime Pro – Stratix 10 Edition Beta on a system that meets or exceeds the following specifications:

- Intel Core i7 Dual or Quad core 64-bit processor
- 36 GB to 96 GB RAM
- SSD – Solid State Drive

For node-locked licenses, compile times vary by PC configuration.

3.1.1 Licensing

The Quartus Prime Pro – Stratix 10 Edition Beta is a special version of the Quartus Prime Pro Edition software that provides advance support for only Stratix 10 devices. Please contact your Intel® sales representative to obtain a license to enable the Quartus Prime Pro – Stratix 10 Edition Beta software.

Note: This document assumes basic familiarity with the Quartus Prime Pro Edition software and the basic Intel FPGA design flow.

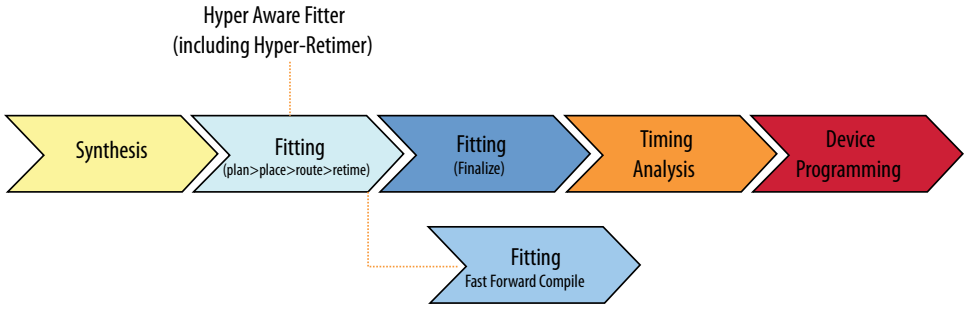
Related Links

- www.altera.com/support
- www.altera.com/training
- www.altera.com/literature

3.2 Recommended Design Flow

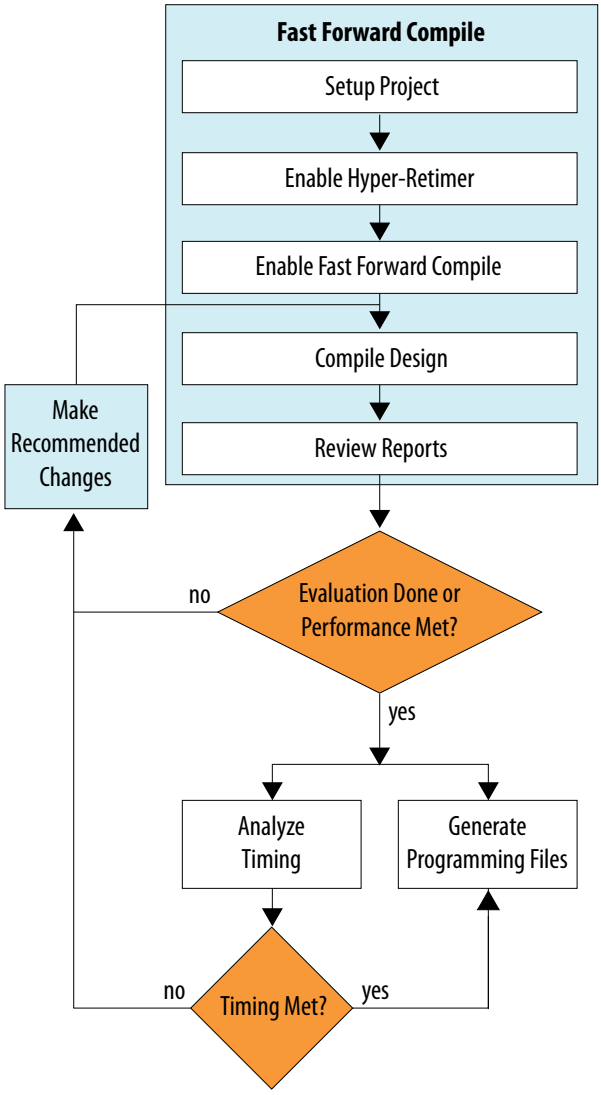
The Quartus Prime Pro – Stratix 10 Edition Beta design flow introduces the Hyper-Retimer and Fast Forward Compilation stages to maximize performance for Stratix 10 designs.
The Hyper-Retimer runs during the Fitter, after place and route, to retime registers into Hyper-Registers for fine-grained performance improvements. Use Fast Forward Compile to identify and break through performance bottlenecks that limit retiming ability.

Figure 53. Hyper-Aware Design Flow Including Hyper-Retiming and Fast Forward Compile



After the Hyper-Retimer or Fast Forward Compile steps, view the results in the Compilation Report to evaluate performance and determine where to implement key RTL performance improvements. After implementing any RTL changes, recompile the design and rerun Fast Forward Compile until performance and timing analysis results are satisfactory.

Figure 54. Fast-Forward Compilation Flow



Depending on your work flow, you can run Fast Forward Compile automatically during every compilation, or you can run Fast Forward Compile as a separate process. For example, automating Fast Forward Compilation helps when iterating through various RTL changes. Conversely, when not focusing on design performance, such as when performing board layout or pin and interface planning, save time by disabling Fast Forward Compile.

Fast Forward Compilation results only predict performance after RTL changes. The Compiler cannot use the Fast Forward results to generate programming files for TimeQuest timing analysis, or for EDA netlist generation. To achieve results similar to the Fast Forward Compile results, you must modify your RTL accordingly and recompile the design.

Related Links

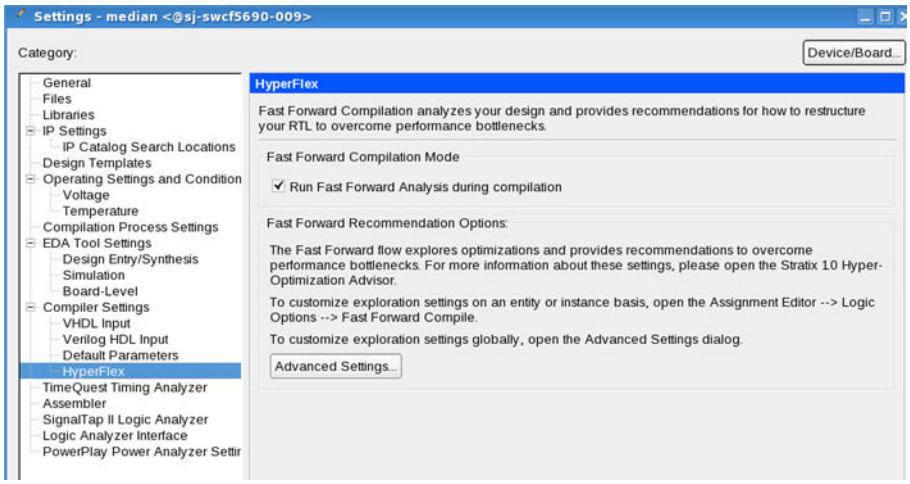
- Step 1: Enable and Run Fast Forward Compile on page 56
- Step 2: Review Critical Chain Reports on page 57
- Step 3: Implement Performance Recommendations on page 58

3.2.1 Step 1: Enable and Run Fast Forward Compile

You can enable Fast-Forward Compile and the Hyper-Retimer for architecture evaluation, performance exploration, and timing closure. To setup your project for Hyper-Retiming and run Fast Forward Compile:

1. Create or open a project in the Quartus Prime Pro – Stratix 10 Edition Beta.
2. Click **Assign ➤ Device**, and then select **Stratix 10** for the **Device family**.
3. The Hyper-Retimer is enabled by default for Stratix 10 designs. To manually enable or disable the Hyper-Retimer, use the following setting in the Quartus Settings File (`.qsf`):
   ```
   set_global_assignment -name HYPER_RETIMER ON
   ```
4. To enable automatic Fast Forward analysis during every compilation, click **Assignments ➤ Settings ➤ Compiler Settings ➤ HyperFlex ➤ Run Fast Forward Analysis during compilation**.

Figure 55. HyperFlex Settings



5. To run Fast Forward Compile, click **Processing ➤ Start Compilation**.

   Note: Alternatively, you can run Fast Forward Compile as a separate process by double-clicking **Generate Fast Forward Timing Closure Recommendations** in the **Tasks** pane. You can run Fast Forward Compile at the command line by typing `quartus_fit --fastforward`.
6. View the results in the Compilation Report to evaluate performance and implement key RTL performance improvements.

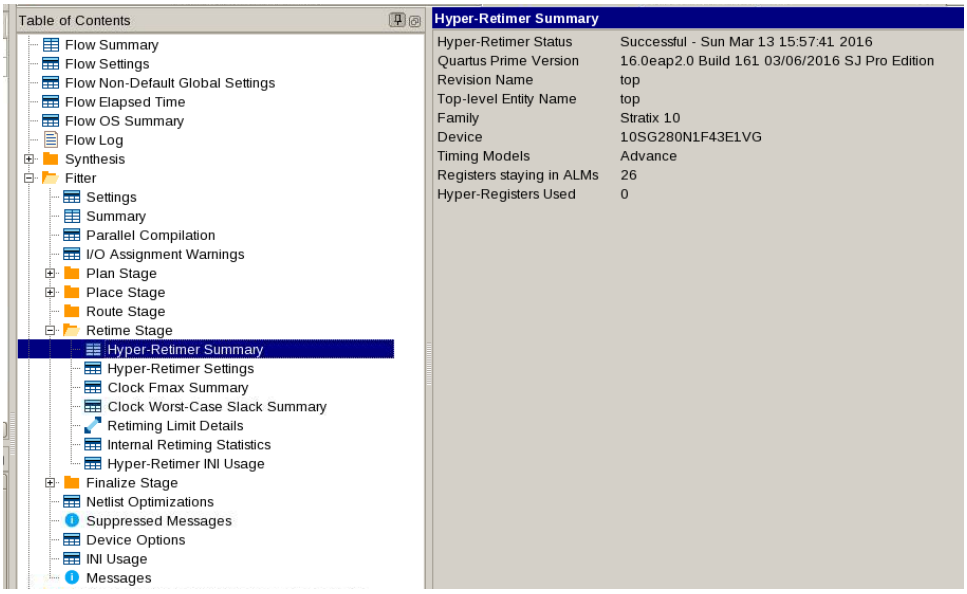
3.2.2 Step 2: Review Critical Chain Reports

Fast Forward Compile and the Hyper-Retimer report information about critical chains in your design. The management of critical chains is vital to achieve the highest performance for Stratix 10 designs. A critical chain is the element of your design that prevents the Hyper-Retimer from further performance improvement. Often, the critical chain includes more than two registers. You can then resolve these critical chains to achieve the highest performance. The Hyper-Retimer also generates a text report with the extension `.retime.rpt`

Hyper-Retimer Summary Report

The Hyper-Retimer generates detailed reports about each clock domain's slack and f_{MAX} performance. The **Hyper-Retimer Summary** report includes recommendations about RTL changes you can make to achieve higher f_{MAX} performance. The report includes details about the part of each clock domain that prevents the Hyper-Retimer from achieving higher performance, and the number of registers that are not retimed. The Hyper-Retimer report is under **Retime Stage** in the **Fitter** report. The report also shows the total number of Hyper-Registers the Fitter uses. These values are for information only, and does not affect changes you make to your RTL or design.

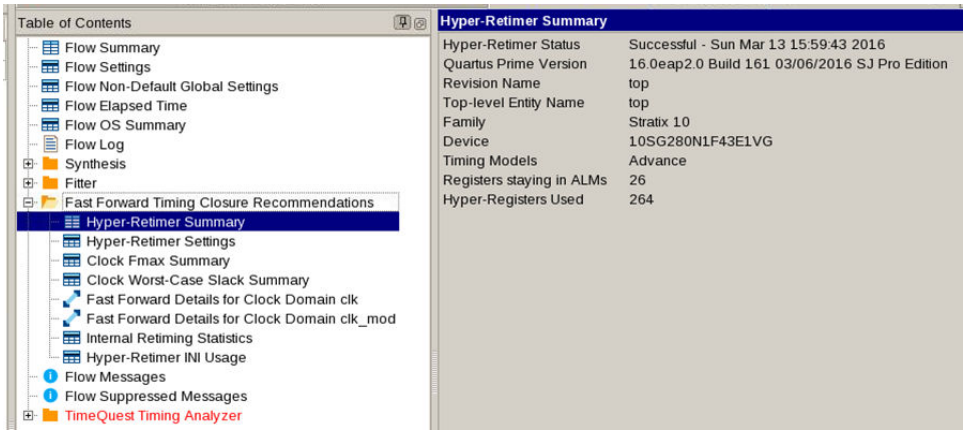
Figure 56. Hyper-Retimer Summary Report



Fast Forward Timing Closure Recommendations Report

Fast Forward Compile generates detailed reports about the f_{MAX} performance gains during each stage of Fast Forward Compile. The **Fast Forward Compile Timing Closure Recommendations** reports include recommendations about RTL changes you can make to achieve higher f_{MAX} performance. Use the **Fast Forward Timing Closure Recommendations** in the Compilation report to optimize your RTL.

Figure 57. Fast Forward Timing Closure Recommendations



Related Links

- Retiming Restrictions and Workarounds on page 90
  This section describes RTL design techniques you can use to avoid retiming restrictions.
- Interpreting Critical Chain Reports on page 69
  This section describes critical chain reports. Use the recommendations in the report, and optimization techniques in this document, to improve the performance of your design.

3.2.3 Step 3: Implement Performance Recommendations

Fast Forward Compile performs the Hyper-Retiming, Hyper-Pipelining, and Hyper-Optimization in your design. Review and implement the suggested changes in your design RTL to realize the predictive performance gains.

After implementing RTL changes, recompile the design and view the impact in the Compilation report. The amount and type of changes that you implement depends on your performance goals. For example, if you can achieve the target f_{MAX} with simple asynchronous clear removal or conversion, your optimization can end. However, if you require additional performance, you must implement more significant Fast Forward recommendations. Explore performance and implement the RTL changes to your code until you reach the desired performance target.

3.3 Using the Hyper-Retimer

The Hyper-Retimer optimizes f_{MAX} performance during fitting by retiming registers to balance the delay between register stages. When you enable the Hyper-Retimer, the Fitter prioritizes paths that cannot be retimed, such as loops. The Fitter then uses the path delays predicted by the Hyper-Retimer to adjust placement. The Hyper-Aware Fitter accounts for the Hyper-Retimer in prioritizing the critical paths. This process exposes the true performance bottleneck that cannot be fixed by the retimer. Based on the critical chains in the Hyper-Retimer report, you can make RTL or settings changes to improve performance on subsequent compiles.

To turn off the Hyper-Retimer, so it does not run during fitting, use the following line in your `.qsf` file:

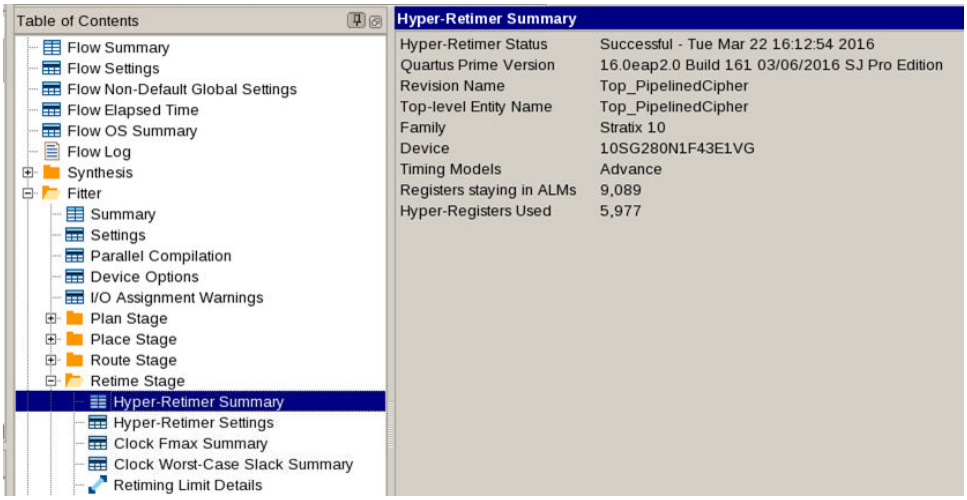
```
set_global_assignment -name HYPER_RETIMER OFF
```

In general, leave the setting at its default value, `ON`. You could turn off the Hyper-Retimer if you easily meet your timing requirements without it, and want to save compile time.

3.3.1 Interpreting Hyper-Retimer Reports

The Hyper-Retimer reports details of each clock domain's slack and f_{MAX} performance. The report includes details about the part of each clock domain that prevented the Hyper-Retimer from achieving higher performance. The Hyper-Retimer report is in **Retime Stage** in the **Fitter** report. The Hyper-Retimer also generates a text format report with the extension `.retime.rpt`.

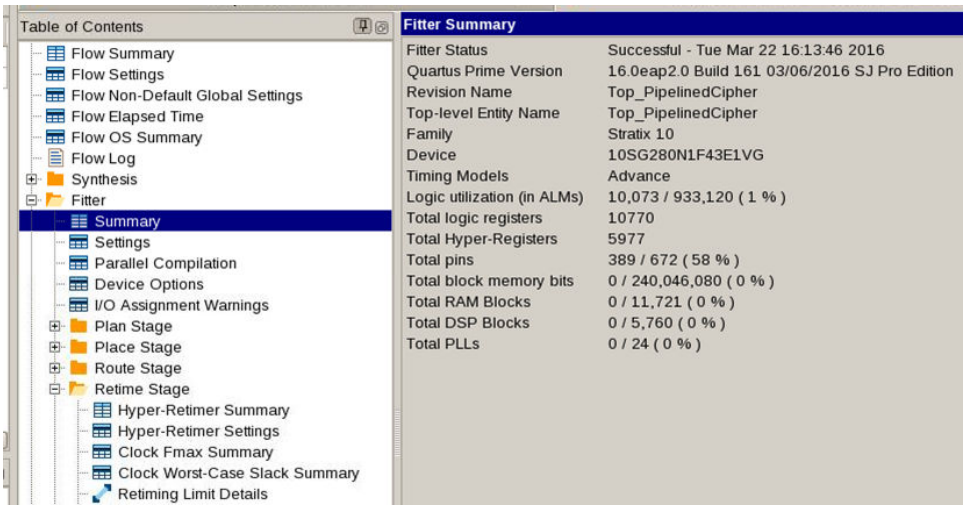
Figure 58. Hyper-Retimer Summary Report (Retime Stage)



The **Summary** report includes a brief overview of the project information, including two lines about register and Hyper-Register usage. **Registers staying in ALMs** gives the number of registers that were not retimed out of ALMs by the Hyper-Retimer. **Hyper-Registers used** gives the number of Hyper-Registers in the routing fabric that were used by the Hyper-Retimer. Compare the **Hyper-Retimer Summary** report with the **Fitter Summary** report that shows Total logic registers and Total Hyper-Registers after fitting.

Total logic registers is the number of registers the Compiler uses after placement and routing, but before the Hyper-Retimer runs. Total Hyper-Registers is the number of Hyper-Registers the Compiler used after the Hyper-Retimer runs.

Figure 59. Fitter Summary Report



Of the 10770 logic registers after the placement and routing completes, 9089 remain in ALMs. This data means that 1681 (1077 - 9089) registers retime into Hyper-Registers in the HyperFlex fabric. During retiming, a total of 5977 Hyper-Registers implement the 1681 that retime out of ALMs. Again, these values are informational only and does not affect changes you make to your RTL or settings.

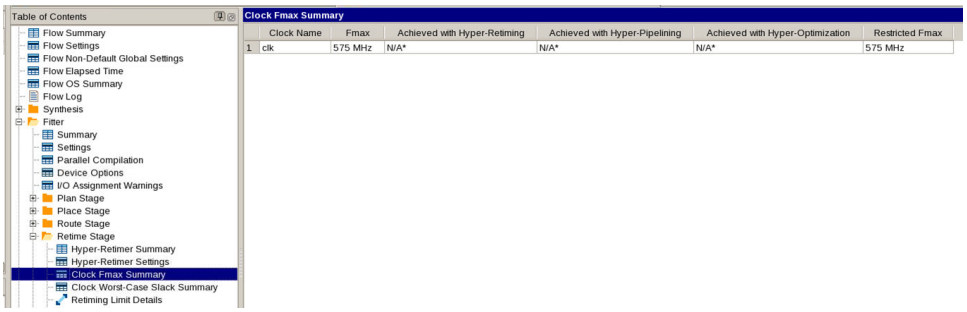
Related Links

Using Fast Forward Compilation on page 65
This section describes using Fast Forward Compilation to guide you through the performance optimization process.

3.3.1.1 Hyper-Retimer Clock Fmax Summary Report

The **Clock Fmax Summary** reports the actual f_{MAX} after the Hyper-Retimer runs and retimes registers to Hyper-Registers. The **Clock Fmax Summary** reports one row for each clock domain.
This report indicates the f_{MAX} performance you achieve if you program the device with a programming file that generates from this compilation.

Figure 60. Hyper-Retimer - Clock Fmax Summary



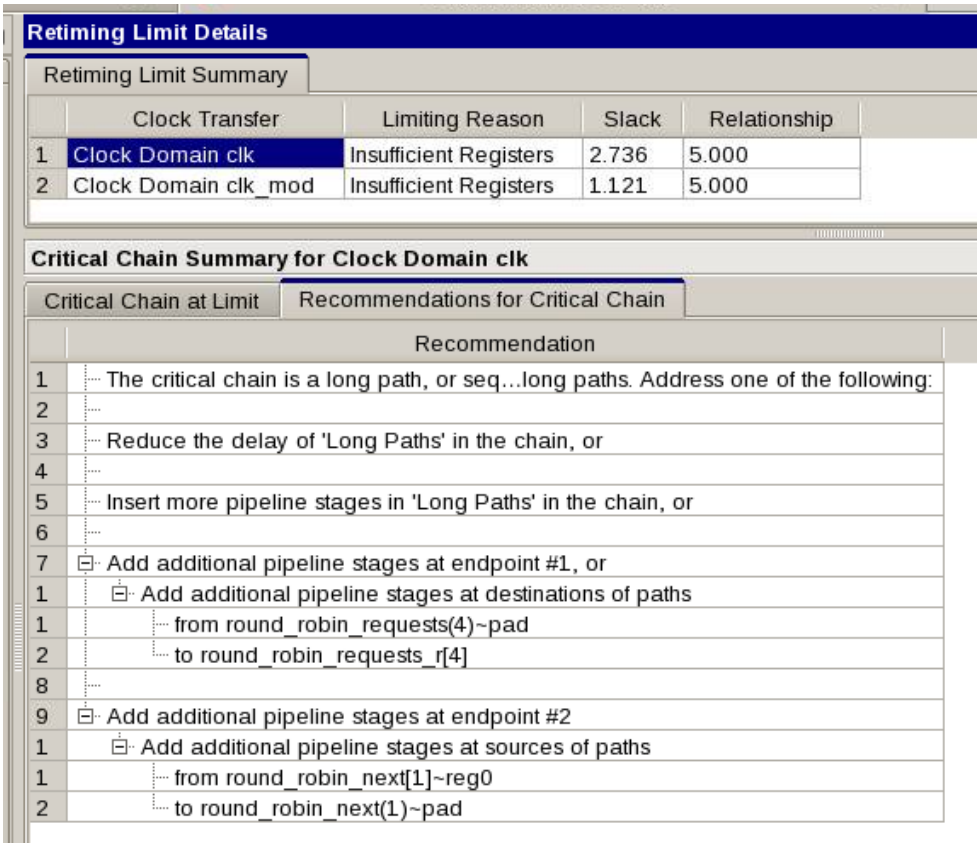
The **Fmax** column shows the f_{MAX} the Hyper-Retimer achieves. The next three columns (**Achieved with Hyper-Retiming**, **Achieved with Hyper-Pipelining**, and **Achieved with Hyper-Optimization**) are blank because the Compiler performs Hyper-Retiming, Hyper-Pipelining, and Hyper-Optimization only during Fast Forward Compile. If the **Restricted Fmax** column shows a lower value than any of the other columns, this typically occurs when some hard block in the clock domain (such as an M20K block) has a lower maximum performance specification than the Hyper-Retimer can achieve.

3.3.1.2 Retiming Limit Details Report

The **Retiming Limit Summary** lists the following data:

- Each clock domain in your design (in the Clock Transfer column)
- The reason the Hyper-Retimer could not make it run faster (in the Limiting Reason column)
- The timing slack for the domain (in the Slack column)
- The clock period requirement (in the Relationship column)

Figure 61. Retiming Limit Details



When you click each clock domain line, a critical chain summary displays below. The critical chain summary contains two tabs:

- **Critical Chain at Limit**—displays the critical chain, which is the part of your design that limited the Hyper-Retimer from achieving higher performance.
- **Recommendations for Critical Chain**—lists specific places in your design to make changes that improve the performance of the critical chain.

The critical chain summaries contain the most important information about the performance-limiting parts of your design, and recommendations to guide your performance optimization.

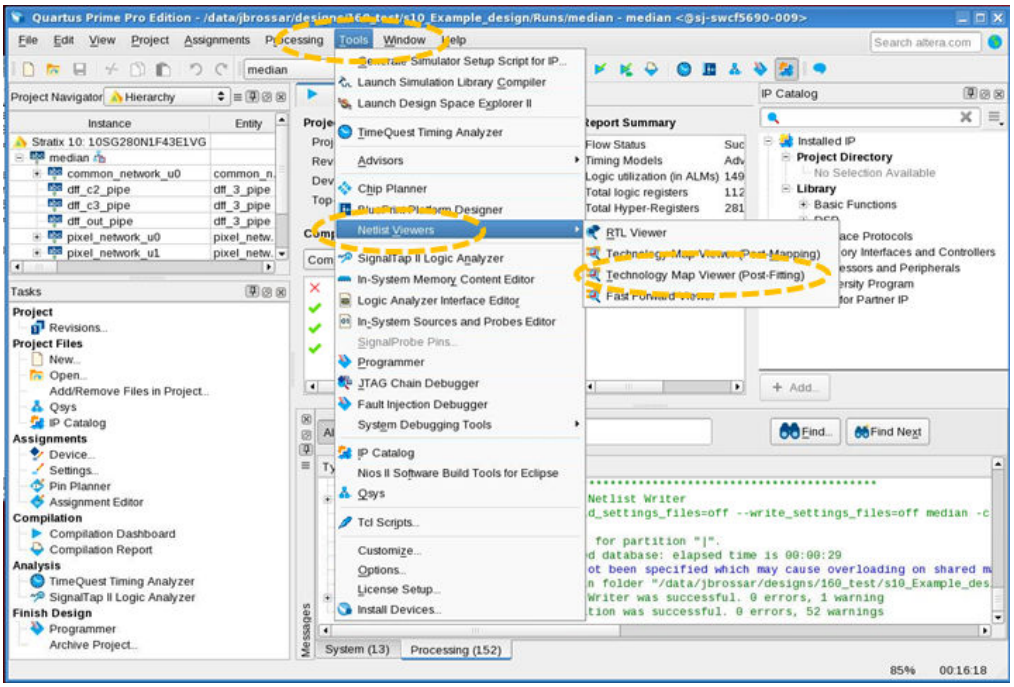
Related Links

Interpreting Critical Chain Reports on page 69
This section describes critical chain reports. Use the recommendations in the report, and optimization techniques in this document, to improve the performance of your design.

3.3.2 Viewing the Hyper-Retimer Netlist

You can display and analyze the design netlist captured after the Hyper-Retimer stage (post-fit). Click **Tools ➤ Netlist Viewers ➤ Technology Map Viewers (post-fitting)** to display the netlist showing the Hyper-Registers and bypassed ALMs in the design.

Figure 62. Accessing the Post-Hyper-Retimer Viewer



You can also right-click a critical chain in the **Retiming Limit Details** panel in the Hyper-Retimer report, and choose **Locate Critical Chain**. The following images compare the netlist views before synthesis (RTL Viewer), after the Fitter but before the Hyper-Retimer (Technology Map (Post-Fit) Viewer), and after the Hyper-Retimer (**Post-Hyper-Retimer Viewer Showing Bypassed Flipflops**).

Figure 63. Locate Critical Chain

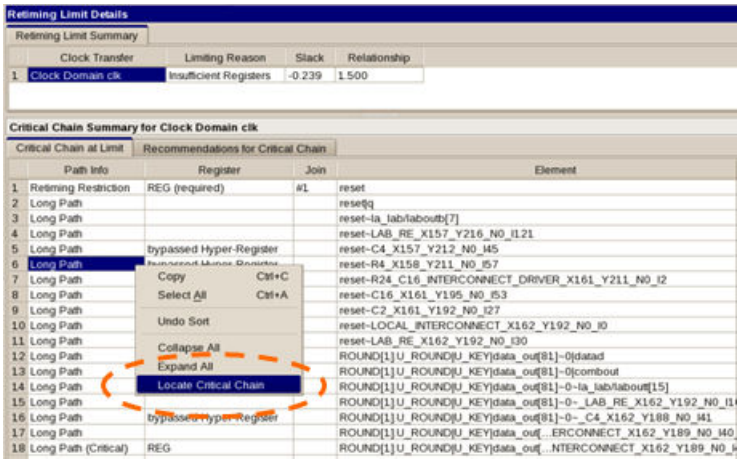
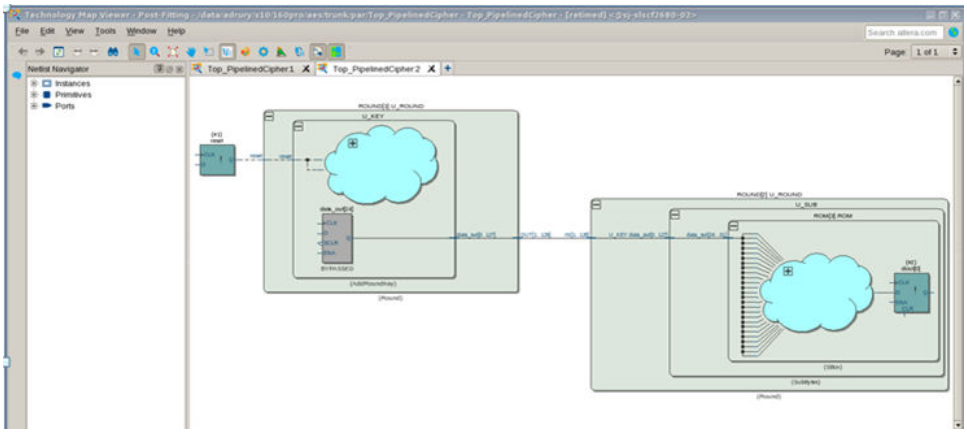
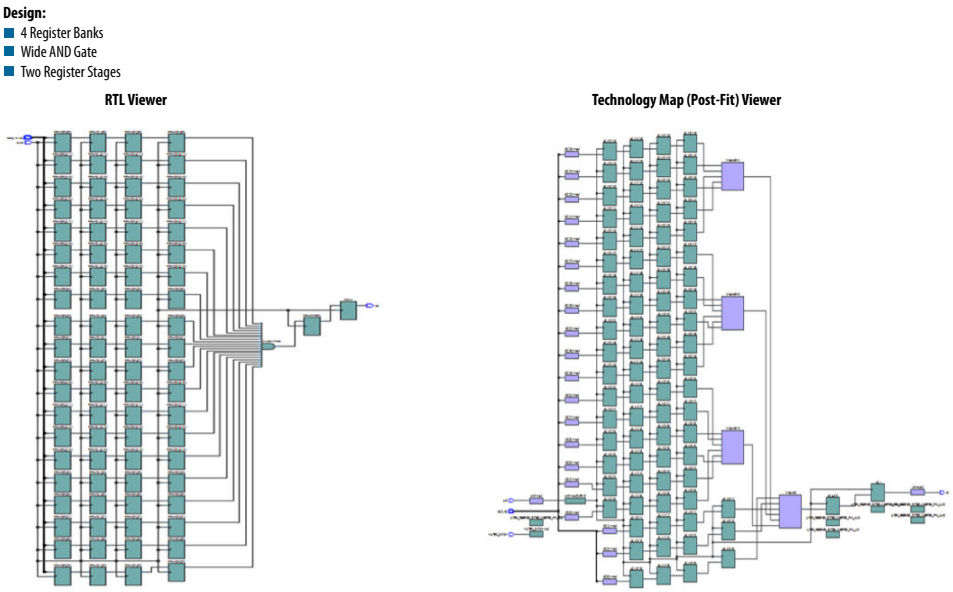


Figure 64. Technology Map Viewer



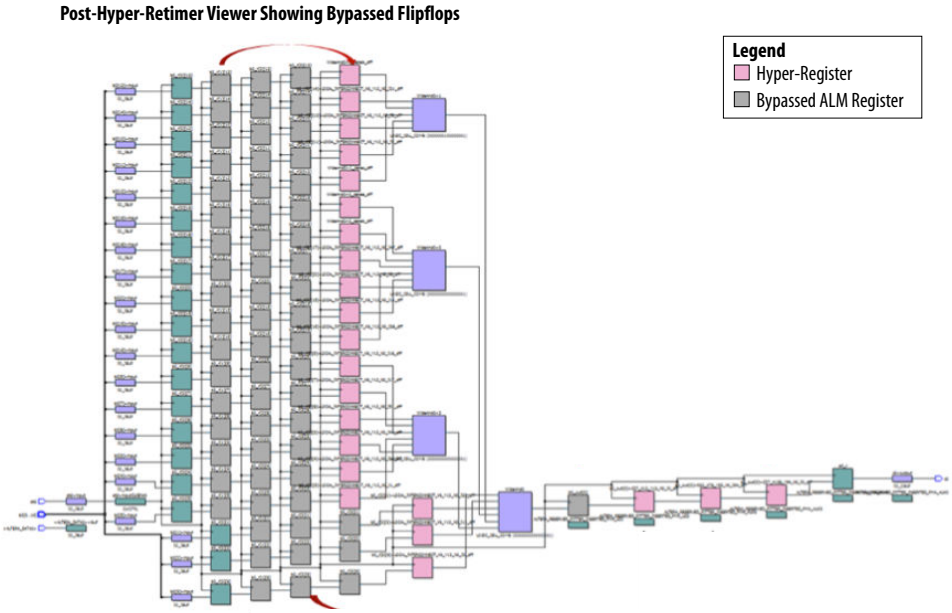
The example design has four banks of registers, a wide AND gate, and two output register stages.

Figure 65. RTL Viewer and Post-Fit Viewer Before Retiming



After retiming, the second stage of the register banks is retimed forward into the first bank of Hyper-Registers. The third and fourth stages are retimed forward across AND gates (shown in purple) into Hyper-Registers. The first output stage is moved to a Hyper-Register (the right-most register shown in pink).

Figure 66. Post-Fit Viewer After Retiming



3.4 Using Fast Forward Compilation

This section describes using Fast Forward Compilation to guide you through the performance optimization process.

Fast Forward Compile provides the following feedback to help optimize design performance:

- Reports the design's performance bottlenecks
- Recommends RTL changes to avoid the bottlenecks
- Predicts design performance after RTL changes

Use Fast Forward Compile iteratively to break through the following types of performance bottlenecks with Hyper-Retiming, Hyper-Pipelining, and Hyper-Optimization:

- Situations that prevent registers from being retimed are avoided with Hyper-Retiming
- Clock domain boundaries without enough registers to maximize performance are modified with Hyper-Pipelining
- Circuit structures that prevent further optimization are modified with Hyper-Optimization

These changes are suggested in the retiming report. Based on the report, you can decide how to implement these changes in a functionally valid way. With Fast Forward Compile, you can see how specific design changes improve performance before any actual modifications are made.

Use Hyper-Retiming

The Hyper-Retimer exposes true performance bottlenecks that you cannot correct through conventional retiming. The first performance bottleneck is any condition that prevents the retiming of registers. During Hyper-Retiming, the Fast Forward Compiler ignores timing restrictions that typically prevent register retiming, such as registers with asynchronous clears, or preserve synthesis attributes. Fast Forward Compile reports each register that ignores retiming restrictions, so you can make the changes to the corresponding RTL and remove the performance bottleneck.

Use Hyper-Pipelining

Another performance bottleneck occurs at clock domain boundaries without enough registers to maximize performance. Clock domain boundaries include top-level I/Os and asynchronous domain crossings. During Hyper-Pipelining, Fast Forward Compile adds pipeline stages at clock domain boundaries, until adding them no longer increases performance. Fast Forward Compile reports all registers at clock domain boundaries where it adds pipeline stages during Hyper-Pipelining, so you can make the changes to your RTL to remove the performance bottlenecks.

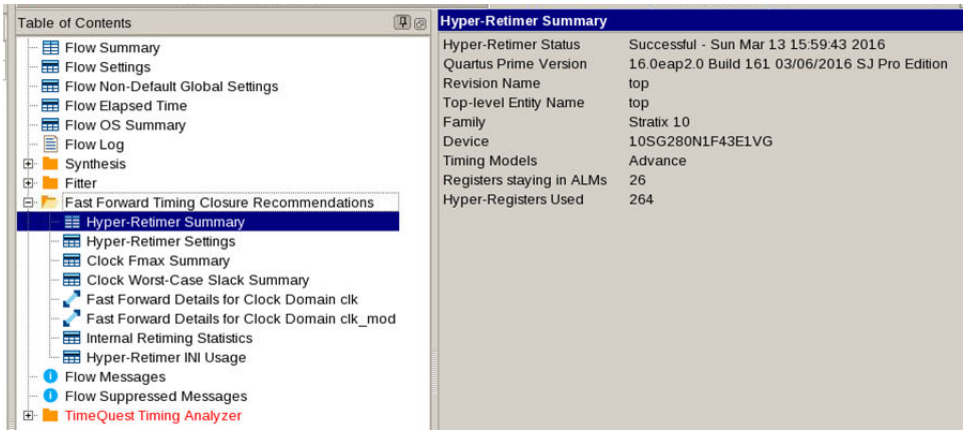
Use Hyper-Optimization

A third performance bottleneck can occur when circuit structures prevent further optimization. During Hyper-Optimization, Fast Forward Compile assumes that all M20K memory blocks and DSP blocks are fully registered. Fast Forward Compile reports all structures it changes during Hyper-Optimization, so you can make the changes to your RTL to remove the performance bottlenecks.

3.4.1 Interpreting Fast Forward Compile Reports

Fast Forward Compile generates detailed reports about the current Fast Forward Compile settings, and a summary of the f_{MAX} performance gains during each stage of Fast Forward Compile. Use the performance improvement recommendations to modify your RTL to achieve higher f_{MAX} performance. Review the Fast Forward Compile information in **Fast Forward Timing Closure Recommendations** in the Compilation report.

Figure 67. Fast Forward Timing Closure Recommendations - Hyper-Retimer Summary

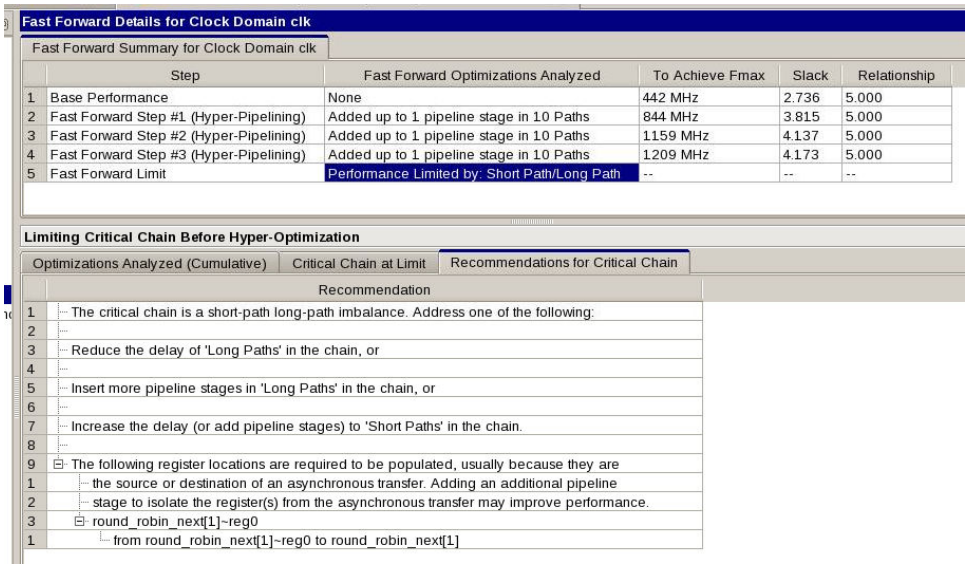


Fast Forward Compile also generates report data in text format, with the extension `.fastforward.rpt`. The **Summary** report includes a brief overview of the project information, including two lines about register and Hyper-Register usage. **Registers staying in ALMs** gives the number of registers that were not retimed out of ALMs during Fast Forward Compile. **Hyper-Registers used** gives the number of Hyper-Registers in the routing fabric that were used during Fast Forward Compile. The values for **Registers staying in ALMs**, and **Hyper-Registers used**, may be different than the values in the Retime stage of the Fitter report. The difference is because Fast Forward Compile performs Hyper-Retiming, Hyper-Pipelining, and Hyper-Optimization to allow more ALM registers to move into Hyper-Registers, to add more pipeline registers, and to fully register RAM and DSP blocks. Therefore, the Fast Forward Compile report typically shows fewer registers staying in ALMs, and more Hyper-Registers used than the Retime stage.

3.4.1.1 Fast Forward Details Report

The **Fast Forward Timing Closure Recommendations** report includes separate reports that summarize the Fast Forward Compile results for each clock domain in your design. Each clock domain report includes a **Fast Forward Summary** that lists the Fast Forward optimization step (**Step** column), a summary of the optimizations applied (**Fast Forward Optimizations Analyzed** column), the potential f_{MAX} with those optimizations (**To Achieve Fmax** column), the timing slack for the domain (**Slack** column), and the clock period requirement (**Relationship** column).

Figure 68. Fast Forward Details for Clock Domain



Click on each step to display more details about the step, including some or all of the following data:

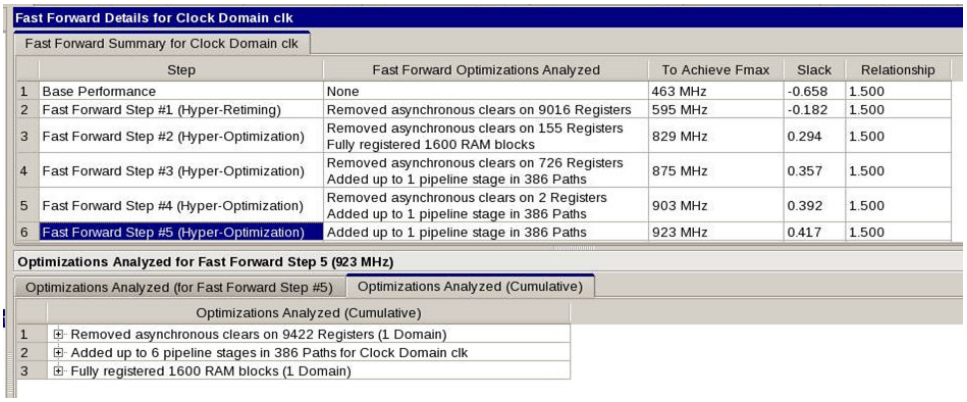
Table 3. Fast Forward Details Report

Report	Description
Optimizations Analyzed (cumulative)	Lists all the optimizations analyzed in the Fast Forward Compile steps up to and including the specific step.
Optimizations Analyzed (For Fast Forward Step #<n>)	Lists only the optimizations analyzed in the specific Fast Forward Compile step.
Critical Chain at Limit	Displays the critical chain that limited the Hyper-Retimer from achieving higher performance. Only the Fast Forward Limit step shows the critical chain tab.
Recommendations for Critical Chain	Lists specific places in your design to make changes that improve the performance of the critical chain. Only the Fast Forward Limit step shows the recommendations tab.

3.4.1.2 Fast Forward Optimizations Analyzed Report

The **Optimizations Analyzed** tab report lists the specific Hyper-Retiming, Hyper-Pipelining, and Hyper-Optimizations performed during Fast Forward Compile. The report organizes the Fast Forward optimizations hierarchically so that you can browse from the high-level system overview to increasing amounts of detail.

Figure 69. Fast Forward Step 5 Optimizations Analyzed (Cumulative)



Fast Forward Details for Clock Domain clk

Fast Forward Summary for Clock Domain clk

	Step	Fast Forward Optimizations Analyzed	To Achieve Fmax	Slack	Relationship
1	Base Performance	None	463 MHz	-0.658	1.500
2	Fast Forward Step #1 (Hyper-Retiming)	Removed asynchronous clears on 9016 Registers	595 MHz	-0.182	1.500
3	Fast Forward Step #2 (Hyper-Optimization)	Removed asynchronous clears on 155 Registers Fully registered 1600 RAM blocks	829 MHz	0.294	1.500
4	Fast Forward Step #3 (Hyper-Optimization)	Removed asynchronous clears on 726 Registers Added up to 1 pipeline stage in 386 Paths	875 MHz	0.357	1.500
5	Fast Forward Step #4 (Hyper-Optimization)	Removed asynchronous clears on 2 Registers Added up to 1 pipeline stage in 386 Paths	903 MHz	0.392	1.500
6	Fast Forward Step #5 (Hyper-Optimization)	Added up to 1 pipeline stage in 386 Paths	923 MHz	0.417	1.500

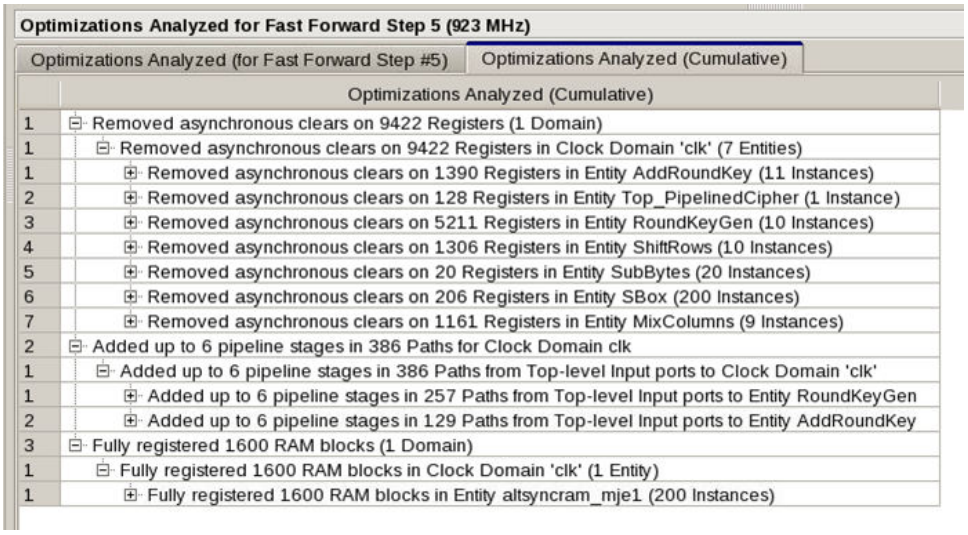
Optimizations Analyzed for Fast Forward Step 5 (923 MHz)

Optimizations Analyzed (for Fast Forward Step #5) | Optimizations Analyzed (Cumulative)

	Optimizations Analyzed (Cumulative)
1	⊞ Removed asynchronous clears on 9422 Registers (1 Domain)
2	⊞ Added up to 6 pipeline stages in 386 Paths for Clock Domain clk
3	⊞ Fully registered 1600 RAM blocks (1 Domain)

The report organizes Hyper-Retiming, Hyper-Pipelining, and Hyper-Optimization step data separately. The Hyper-Retiming analysis in Fast Forward Compile steps 1, 2, 3, and 4 in the example removes asynchronous clears on a total of 9422 registers in one clock domain. The Hyper-Pipelining analysis in Fast Forward Compile steps 3, 4, and 5 added up to 6 pipeline stages on 386 paths in one clock domain. The Hyper-Optimization analysis in Fast Forward Compile step 2 fully registered 1600 RAM blocks. You can expand each grouping of optimizations for more details.

Figure 70. Fast Forward Report Optimizations Summary



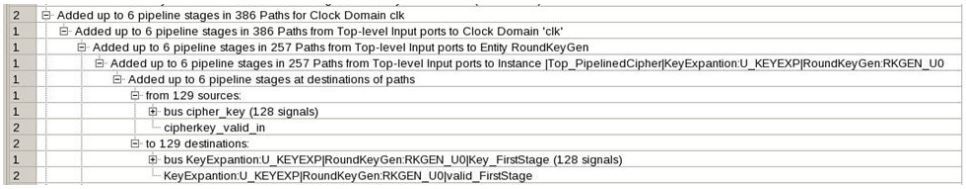
Optimizations Analyzed for Fast Forward Step 5 (923 MHz)

Optimizations Analyzed (for Fast Forward Step #5) | Optimizations Analyzed (Cumulative)

	Optimizations Analyzed (Cumulative)
1	⊟ Removed asynchronous clears on 9422 Registers (1 Domain)
1	⊟ Removed asynchronous clears on 9422 Registers in Clock Domain 'clk' (7 Entities)
1	⊞ Removed asynchronous clears on 1390 Registers in Entity AddRoundKey (11 Instances)
2	⊞ Removed asynchronous clears on 128 Registers in Entity Top_PipelinedCipher (1 Instance)
3	⊞ Removed asynchronous clears on 5211 Registers in Entity RoundKeyGen (10 Instances)
4	⊞ Removed asynchronous clears on 1306 Registers in Entity ShiftRows (10 Instances)
5	⊞ Removed asynchronous clears on 20 Registers in Entity SubBytes (20 Instances)
6	⊞ Removed asynchronous clears on 206 Registers in Entity SBox (200 Instances)
7	⊞ Removed asynchronous clears on 1161 Registers in Entity MixColumns (9 Instances)
2	⊟ Added up to 6 pipeline stages in 386 Paths for Clock Domain clk
1	⊟ Added up to 6 pipeline stages in 386 Paths from Top-level Input ports to Clock Domain 'clk'
1	⊞ Added up to 6 pipeline stages in 257 Paths from Top-level Input ports to Entity RoundKeyGen
2	⊞ Added up to 6 pipeline stages in 129 Paths from Top-level Input ports to Entity AddRoundKey
3	⊟ Fully registered 1600 RAM blocks (1 Domain)
1	⊟ Fully registered 1600 RAM blocks in Clock Domain 'clk' (1 Entity)
1	⊞ Fully registered 1600 RAM blocks in Entity altsyncram_mje1 (200 Instances)

The report also groups optimizations by clock domain, entity, instance, and register name.

Figure 71. Fast Forward Report Optimizations Register Details



2	⊟ Added up to 6 pipeline stages in 386 Paths for Clock Domain clk
1	⊟ Added up to 6 pipeline stages in 386 Paths from Top-level Input ports to Clock Domain 'clk'
1	⊟ Added up to 6 pipeline stages in 257 Paths from Top-level Input ports to Entity RoundKeyGen
1	⊟ Added up to 6 pipeline stages in 257 Paths from Top-level Input ports to Instance |Top_PipelinedCipher|KeyExpantion:U_KEYEXP|RoundKeyGen:RKGEN_U0
1	⊟ Added up to 6 pipeline stages at destinations of paths
1	⊟ from 129 sources:
1	⊞ bus cipher_key (128 signals)
2	cipherkey_valid_in
2	⊟ to 129 destinations:
1	⊞ bus KeyExpantion:U_KEYEXP|RoundKeyGen:RKGEN_U0|Key_FirstStage (128 signals)
2	KeyExpantion:U_KEYEXP|RoundKeyGen:RKGEN_U0|valid_FirstStage

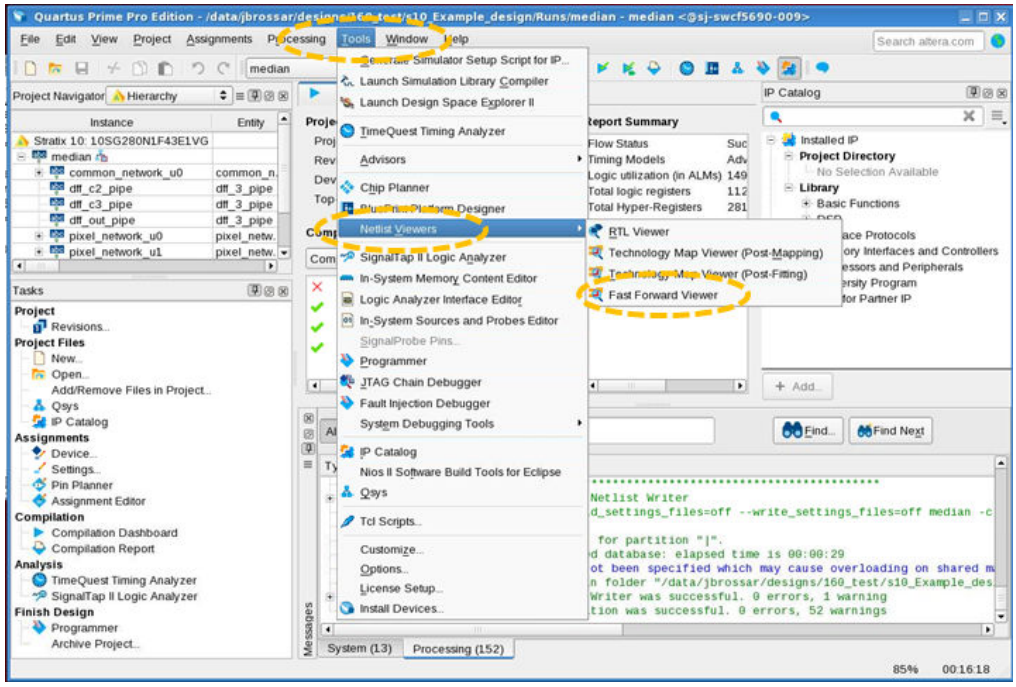
The **Optimizations Analyzed** report lists RTL changes you can make to reach a particular performance level. If you make the same changes manually in your RTL and recompile the design, you can expect to achieve the same performance.

For best performance, implement as many of the Fast Forward optimizations as you can. The Hyper-Retimer executes the changes and predicts a certain level of performance. The recommendations provided for improving the performance of critical chains can have varying results. Optimizing one critical chain may uncover a second, or multiple other critical chains, that have to be optimized to break through a performance barrier.

3.4.2 Viewing the Fast Forward Compile Netlist

You can display and analyze the design netlist captured after the Fast Forward Compile stage. This netlist reflects your design as if you implement all Fast Forward recommendations. Use this netlist to visualize your optimized design implementation.

Figure 72. Accessing the Fast Forward Netlist Viewer



3.5 Interpreting Critical Chain Reports

This section describes critical chain reports. Use the recommendations in the report, and optimization techniques in this document, to improve the performance of your design.

Critical chains are an important new concept for designs in the Stratix 10 architecture. A critical chain is the specific part of your design that prevents the Hyper-Retimer from making it run any faster. Critical chain reports list the registers and combinational nodes that comprise the critical chains in your design, and include recommendations about steps you can take to improve performance.

Figure 73. Sample Critical Chain

In the above example, the thick red line from register A through combinational logic and two registers to register B represents a sample critical chain. Register A cannot be retimed forward, and register B cannot be retimed backward due to timing restrictions. Sometimes register A and register B can be the same register, in which case the critical chain is called a loop. 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.

A critical path is the limiting factor that prevents a design from running faster with conventional retiming techniques. A critical path is a register-to-register path, often with combinational logic. With the Hyper-Retimer, the limiting factor is called a critical chain because it often includes more than one register-to-register path. A critical chain is a higher level abstraction of a critical path. You can analyze critical paths in HyperFlex designs with the TimeQuest Timing Analyzer. However, the Hyper-Retimer critical chain reports are the best place to see how to improve design performance. With the Hyper-Retimer, you can focus on higher level optimization, because the Hyper-Retimer uses Hyper-Registers to evenly balance slacks on all the registers in a critical chain.

The performance recommendations for these chains can include one or more of the following steps:

- Reduce the delay of 'Long Paths' in the chain. Use standard timing closure techniques to reduce delay, but it might be more fruitful to look at other recommendations too. Paths can have delay from too much combinational logic, or from sub-optimal placement, or routing congestion, among other reasons.
- 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.

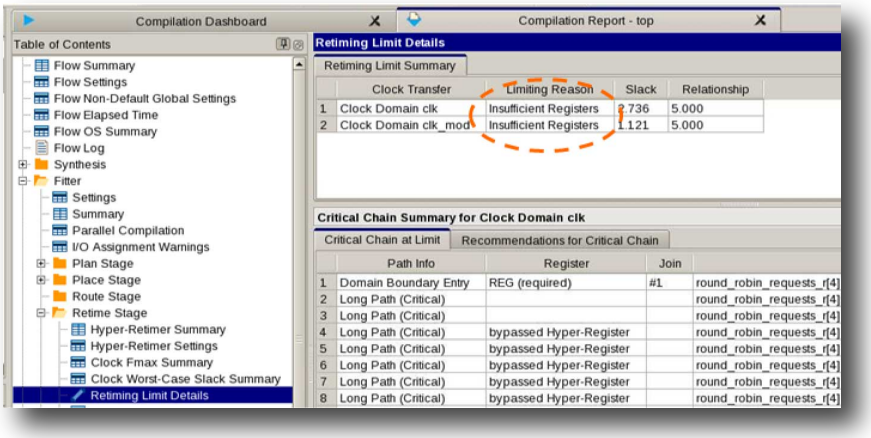
3.5.1 Types of Critical Chains

Depending on how they limit performance, critical chains are classified by the Hyper-Retimer in one of the following ways:

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

The following figure shows the location of the critical chain in the Hyper-Retimer Report:

Figure 74. Types of Critical Chains Listed as the Limiting Reason in the Hyper-Retimer Report



Report

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.

Related Links

- Insufficient Registers on page 71
- Short Path/Long Path on page 75
- Path Limit on page 79
- Loops on page 81

3.5.1.1 Insufficient Registers

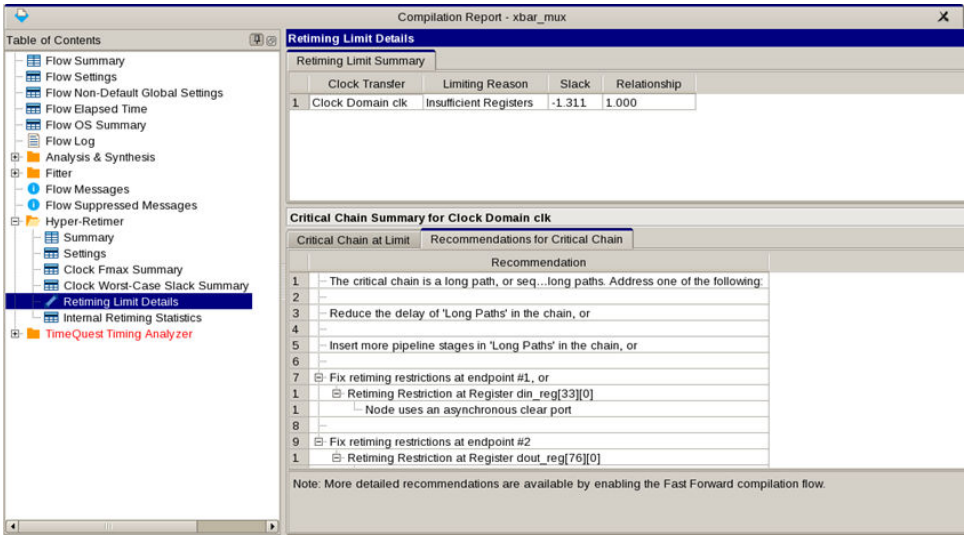
When registers at neither end of the chain can be retimed, and adding more registers can improve performance, the limiting reason reported is Insufficient Registers.

3.5.1.1.1 Insufficient Registers Example

The following screenshots show the relevant parts of the Hyper-Retimer report and the logic contained in the critical chain.

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

Figure 75. Retiming Limit Details



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.

The **Requirement** column contains the period requirement of the clock domain, including any setup and hold uncertainty.

Figure 76. Critical Chain in 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.

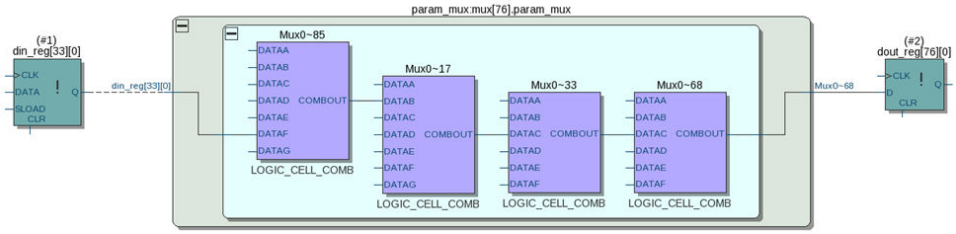
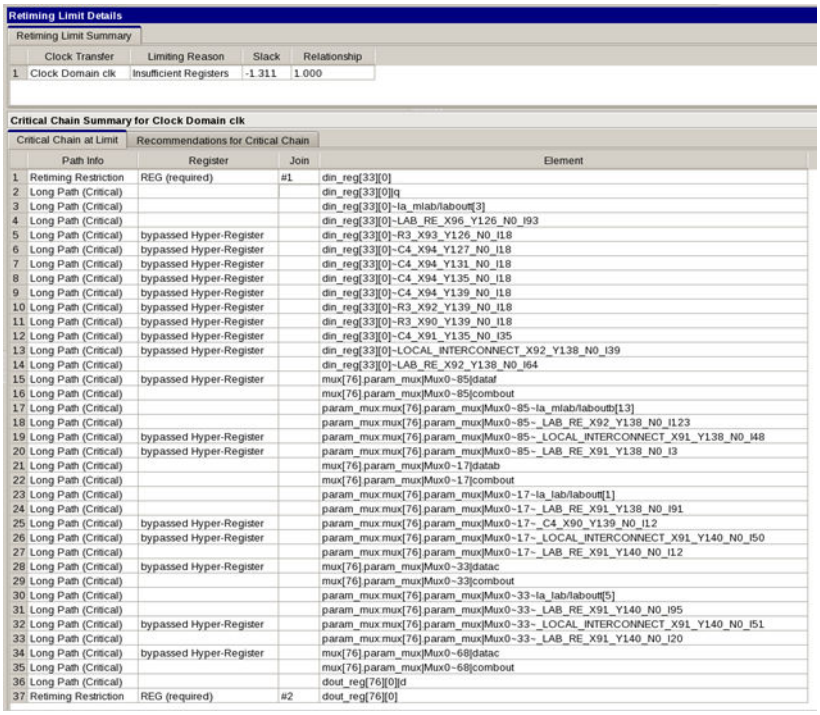


Figure 77. Critical Chain with Insufficient Registers Reported by the Hyper-Retimer



Retiming Limit Details

Retiming Limit Summary

	Clock Transfer	Limiting Reason	Slack	Relationship
1	Clock Domain clk	Insufficient Registers	-1.311	1.000

Critical Chain Summary for Clock Domain clk

Critical Chain at Limit | Recommendations for Critical Chain

	Path Info	Register	Join	Element
1	Retiming Restriction	REG (required)	#1	din_reg[33][0]
2	Long Path (Critical)			din_reg[33][0]|q
3	Long Path (Critical)			din_reg[33][0]~la_mlab/laboutt[3]
4	Long Path (Critical)			din_reg[33][0]~LAB_RE_X96_Y126_N0_I93
5	Long Path (Critical)	bypassed Hyper-Register		din_reg[33][0]~R3_X93_Y126_N0_I18
6	Long Path (Critical)	bypassed Hyper-Register		din_reg[33][0]~C4_X94_Y127_N0_I18
7	Long Path (Critical)	bypassed Hyper-Register		din_reg[33][0]~C4_X94_Y131_N0_I18
8	Long Path (Critical)	bypassed Hyper-Register		din_reg[33][0]~C4_X94_Y135_N0_I18
9	Long Path (Critical)	bypassed Hyper-Register		din_reg[33][0]~C4_X94_Y139_N0_I18
10	Long Path (Critical)	bypassed Hyper-Register		din_reg[33][0]~R3_X92_Y139_N0_I18
11	Long Path (Critical)	bypassed Hyper-Register		din_reg[33][0]~R3_X90_Y139_N0_I18
12	Long Path (Critical)	bypassed Hyper-Register		din_reg[33][0]~C4_X91_Y135_N0_I35
13	Long Path (Critical)	bypassed Hyper-Register		din_reg[33][0]~LOCAL_INTERCONNECT_X92_Y138_N0_I39
14	Long Path (Critical)			din_reg[33][0]~LAB_RE_X92_Y138_N0_I64
15	Long Path (Critical)	bypassed Hyper-Register		mux[76].param_mux|Mux0~85|dataf
16	Long Path (Critical)			mux[76].param_mux|Mux0~85|combout
17	Long Path (Critical)			param_mux:mux[76].param_mux|Mux0~85~la_mlab/laboutb[13]
18	Long Path (Critical)			param_mux:mux[76].param_mux|Mux0~85~_LAB_RE_X92_Y138_N0_I123
19	Long Path (Critical)	bypassed Hyper-Register		param_mux:mux[76].param_mux|Mux0~85~_LOCAL_INTERCONNECT_X91_Y138_N0_I48
20	Long Path (Critical)	bypassed Hyper-Register		param_mux:mux[76].param_mux|Mux0~85~_LAB_RE_X91_Y138_N0_I3
21	Long Path (Critical)			mux[76].param_mux|Mux0~17|datab
22	Long Path (Critical)			mux[76].param_mux|Mux0~17|combout
23	Long Path (Critical)			param_mux:mux[76].param_mux|Mux0~17~la_lab/laboutt[1]
24	Long Path (Critical)			param_mux:mux[76].param_mux|Mux0~17~_LAB_RE_X91_Y138_N0_I91
25	Long Path (Critical)	bypassed Hyper-Register		param_mux:mux[76].param_mux|Mux0~17~_C4_X90_Y139_N0_I12
26	Long Path (Critical)	bypassed Hyper-Register		param_mux:mux[76].param_mux|Mux0~17~_LOCAL_INTERCONNECT_X91_Y140_N0_I50
27	Long Path (Critical)			param_mux:mux[76].param_mux|Mux0~17~_LAB_RE_X91_Y140_N0_I12
28	Long Path (Critical)	bypassed Hyper-Register		mux[76].param_mux|Mux0~33|datac
29	Long Path (Critical)			mux[76].param_mux|Mux0~33|combout
30	Long Path (Critical)			param_mux:mux[76].param_mux|Mux0~33~la_lab/laboutt[5]
31	Long Path (Critical)			param_mux:mux[76].param_mux|Mux0~33~_LAB_RE_X91_Y140_N0_I95
32	Long Path (Critical)	bypassed Hyper-Register		param_mux:mux[76].param_mux|Mux0~33~_LOCAL_INTERCONNECT_X91_Y140_N0_I51
33	Long Path (Critical)			param_mux:mux[76].param_mux|Mux0~33~_LAB_RE_X91_Y140_N0_I20
34	Long Path (Critical)	bypassed Hyper-Register		mux[76].param_mux|Mux0~68|datac
35	Long Path (Critical)			mux[76].param_mux|Mux0~68|combout
36	Long Path (Critical)			dout_reg[76][0]|d
37	Retiming Restriction	REG (required)	#2	dout_reg[76][0]

Note that the beginning of the critical chain, on line 1, has Retiming Restriction listed in the **Path Info** column. Also, the end of the critical chain, on line 37, has Retiming Restriction listed also. The retiming restrictions are because of the asynchronous resets on the two registers.

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

Table 4. 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
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

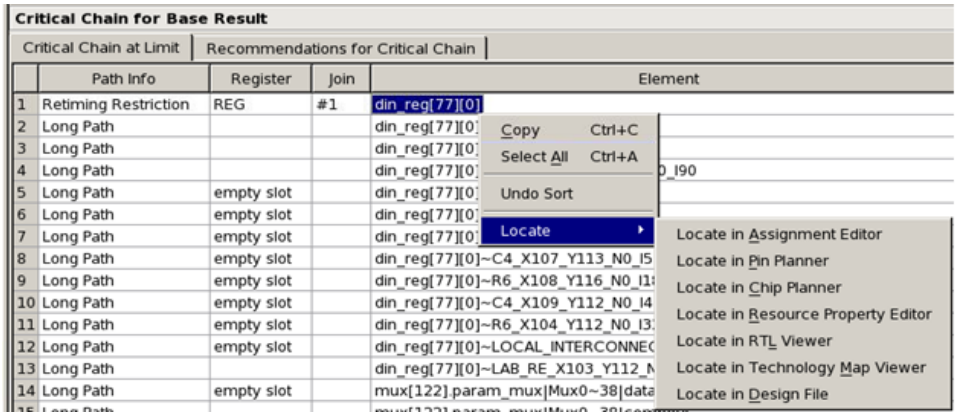
In the critical chain report in Figure 77 on page 73, there are 17 lines that list bypass Hyper-Register in the **Register** column. Bypassed Hyper-Register indicates the location of a Hyper-Register the Hyper-Retimer can use if there are more registers in the chain, or if there are no restrictions on the endpoints. If there are no restrictions on the endpoints, the Hyper-Retimer 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 can be retimed. The Fast Forward Compile process could 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 *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 Quartus Prime Pro – Stratix 10 Edition Beta with the **Locate** option, as shown in the following figure.

Figure 78. Cross Probe from the Critical Chain Report



Related Links

- Short Path/Long Path on page 75
- Complex Critical Chains on page 84
- Facilitate Register Movement (Hyper-Retiming) on page 14
  This section discusses facilitating register movement in your design (Hyper-Retiming).

3.5.1.1.2 Optimizing Insufficient Registers

Evaluate the recommendations from the Hyper-Retimer to improve performance. To fix critical chains with the limiting reason of insufficient registers, use Hyper-Retiming and Hyper-Pipelining techniques that this documents describes.

Related Links

- Facilitate Register Movement (Hyper-Retiming) on page 14
  This section discusses facilitating register movement in your design (Hyper-Retiming).
- Add Pipeline Registers (Hyper-Pipelining) on page 26
  This section discusses adding pipeline registers to increase performance.

Critical Chains with Dual Clock Memories

The Hyper-Retimer does not retime registers through dual clock memories. Therefore, it is possible that a functional block in your design that is between two dual clock FIFOs or memories could be reported as the critical chain, with a limiting reason of Insufficient Registers, even when you perform a Fast Forward Compile. If you have a critical chain with a limiting reason of Insufficient Registers, and the chain is between dual clock memories, you can add pipeline stages to the functional block. You can also add a bank of registers in the RTL design and allow the Hyper-Retimer to balance the registers. Refer to *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 Hyper-Retimer 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: Parameterizable Pipeline Modules on page 50

3.5.1.2 Short Path/Long Path

When the critical chain has related paths with conflicting characteristics where one path could 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 the Hyper-Retimer to satisfy. Short paths and long paths are always connected in some way, with at least one common node. The Hyper-Retimer needs to maintain functional correctness by ensuring identical relative latency through both critical chains, which results 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.

3.5.1.2.1 Hyper-Register Locations Not Available

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

In the following example, the short path includes two Hyper-Register locations, both of which have been used by the Hyper-Retimer. One is indicated on line 38, and the other on line 41. Lines 38 and 41 indicate REG in the **Register** column. The names in the **Element** column end in `_dff`, indicating that the Hyper-Registers in those locations have been used. The `_dff` represents the D flop-flop. No other Hyper-Register locations are available for the Hyper-Retimer to use in that chain segment. If there were Hyper-Register locations available, particular lines would indicate that with a bypassed Hyper-Register entry in the **Register** column. Line 45 is not a Hyper-Register; it is an ALM register.

Figure 79. 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]|q
3	Short Path	unusable (hold)		rrm|Mux4~0|datae
4	Short Path			rrm|Mux4~0|combout
5	Short Path			round_robin_modulo:rrm|Mux4~0la_mlab/laboutt[13]
6	Short Path			round_robin_modulo:rrm|Mux4~0_LAB_RE_X76_Y198_N0_I103
7	Short Path	REG	#2	round_robin_modulo:rrm|Mux4~0_LO..._INTERCONNECT_X75_Y198_N0_I3_dff
8	----------	----------	----	----------

3.5.1.2.2 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 indicated on line 3. However, it cannot be used because using it does not meet hold time requirements as indicated on line 3. The register on line 1 cannot be retimed forward, and the register on line 7 cannot be retimed backward.

Figure 80. Critical Chain with Short Path/Long Path

Limiting Critical Chain Before Hyper-Optimization

Optimizations Analyzed (Cumulative) | Critical Chain at Limit | Recommendations for Critical Chain

	Path Info	Register	Join	Element
1	Short Path	REG	#1	round_robin_mod_last_r[1]~C4_X73_Y130_N0_I6_dff
2	Short Path			round_robin_mod_last_r[1]~C4_X73_Y130_N0_I6
3	Short Path	unusable (hold)		round_robin_mod_last_r[1]~LOCAL_INTERCONNECT_X74_Y133_N0_I51
4	Short Path			round_robin_mod_last_r[1]~LAB_RE_X74_Y133_N0_I9
5	Short Path			round_robin_modulo:rm|next[1]~3~_datae_dff.d
6	Short Path	REG		round_robin_modulo:rm|next[1]~3~_datae_dff
7	Short Path			rm|next[1]~3|datae
8	Short Path			rm|next[1]~3|combout
9	Short Path			round_robin_mod_next[1]~reg0|d
10	Short Path	REG (required)	#2	round_robin_mod_next[1]~reg0
11	----------	----------	----	----------
12	Long Path (Critical)	REG	#1	round_robin_mod_last_r[1]~C4_X73_Y130_N0_I6_dff
13	Long Path (Critical)			round_robin_mod_last_r[1]~C4_X73_Y130_N0_I6
14	Long Path (Critical)	bypassed Hyper-Register		round_robin_mod_last_r[1]~LOCAL_INTERCONNECT_X74_Y132_N0_I40
15	Long Path (Critical)			round_robin_mod_last_r[1]~LAB_RE_X74_Y132_N0_I0
16	Long Path (Critical)	bypassed Hyper-Register		rm|Mod3|auto_generated|divider|divider|op_3~5|dataf
17	Long Path (Critical)			rm|Mod3|auto_generated|divider|divider|op_3~9|cout

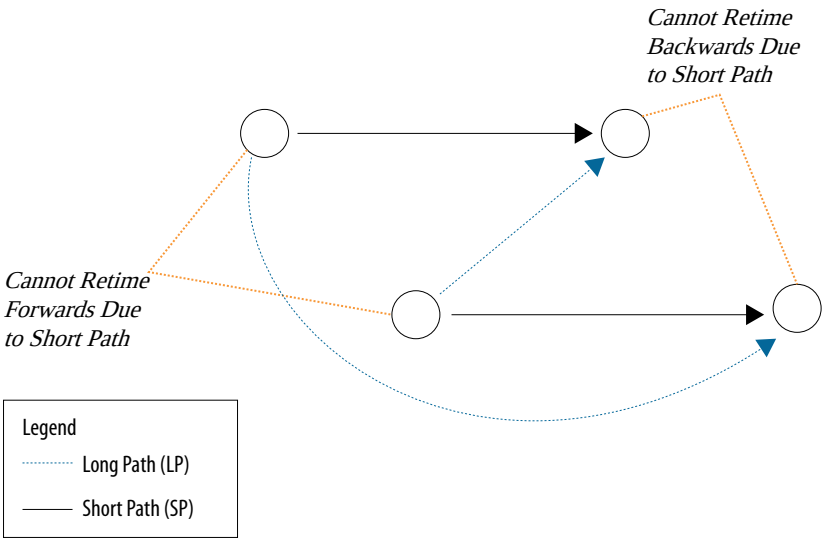
3.5.1.2.3 Optimizing Short Path/Long Path

Evaluate the recommendations from the Hyper-Retimer to optimize performance limitations due to short path/long path.

Add Registers

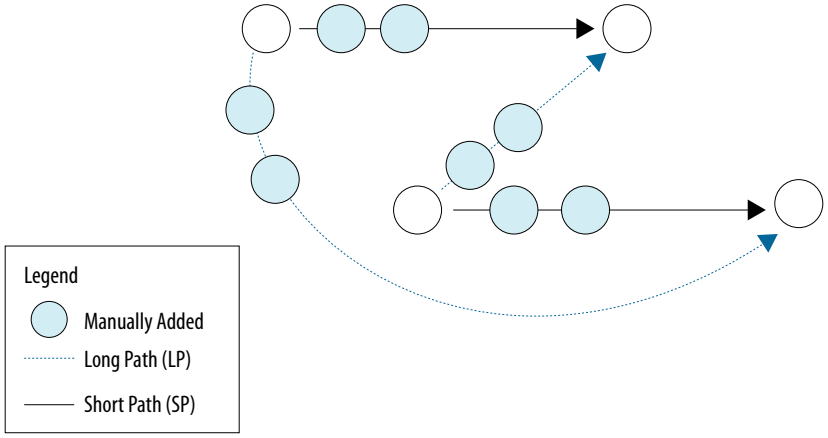
Manually adding registers on both the short and long paths can be helpful if you can accommodate the extra latency in the critical chain.

Figure 81. Critical Chain with Alternating Short Path/Long Path



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

Figure 82. Sample Short Path/Long Path with Additional Latency



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 83. Critical Chain with Alternating Short Path/Long Path

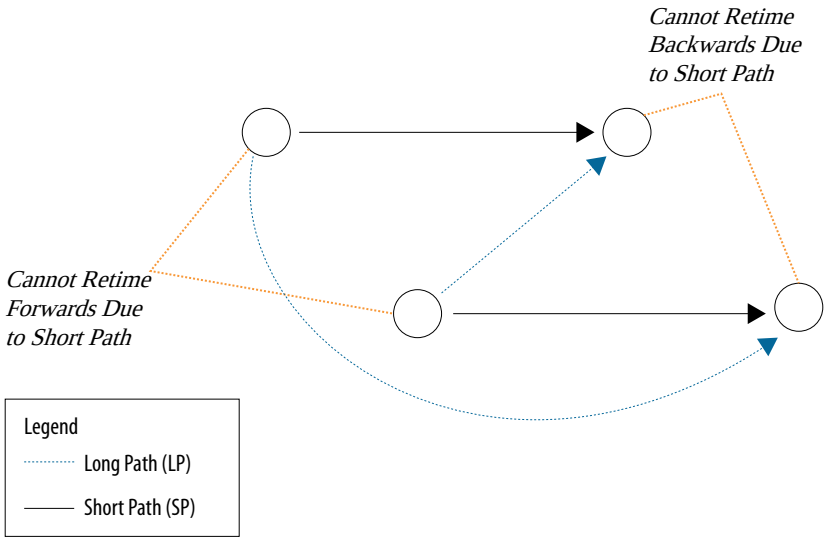
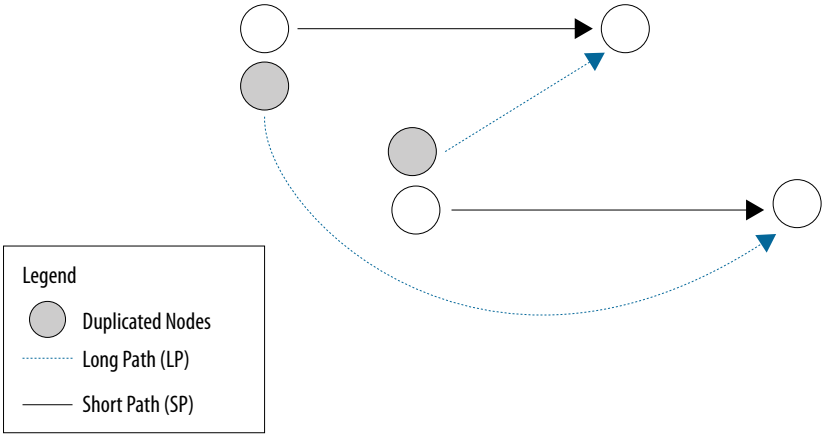


Figure 84. Short Path/Long Path with Two Duplicate Nodes



The fitter and Hyper-Retimer 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 could 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, so it might be better to use a maximum fan-out directive.

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.

3.5.1.3 Path 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 be retimed into. Path Limit also indicates that you have reached a performance limit of the current place and route result.

As shown in the following figure, the critical chain goes from a hard memory block to the first Hyper-Register available outside the hard memory block. The fact that the source is a hard memory block can be inferred from parts of the names on lines 1, 2, and 3. Lines 1 and 2 refer to `ram_block1a0`, and line 3 contains a reference to `MEDIUM_EAB_RE`, which refers to a medium embedded array block routing element. The medium embedded array block is one of the hard memory blocks in Stratix 10 devices.

Figure 85. Critical Chain with Path Limit

Limiting Critical Chain Before Hyper-Optimization

Fast Forward Optimizations Applied in Clock Domain clk | Critical Chain at Limit | Recommendations for Critical Chain

	Path Info	Register	Join	Element
1	Long Path	REG (required)	#1	Round:ROUND[0].U_ROUND|SubBytes:...auto_generated|ram_block1a0~reg0
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

When the critical chain is a Path Limit, it shows Long Path in the **Path Info** column. This indicates that the chain is too long, and it could go faster if the Hyper-Retimer could retime a register into the chain. No entries marked as bypassed Hyper-Register in the **Register** column indicate that there are no Hyper-Register locations available for the Hyper-Retimer to use.

The limiting reason of Path Limit does not imply that the critical chain has reached the inherent silicon performance limit. It simply means that the current place and route result has the reported performance limit. Another compilation could result in a different placement that allows the Hyper-Retimer to achieve better performance on the particular critical chain. One common reason for a path limit is when registers have not been packed into dedicated input or output registers in a hard DSP or RAM block.

3.5.1.3.1 Optimizing Path Limit

Evaluate the recommendations from the Hyper-Retimer to see what optimizations are recommended by the Quartus Prime Pro – Stratix 10 Edition Beta.

If your critical chain has a limiting reason of Path Limit and it 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 it 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. When 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 to the Hyper-Retimer for optimization, and are not shown 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. The performance can be improved by enabling optional input and output registers.

Turn on optional registers using the IP parameter editor to parameterize hard DSP or memory blocks. If DSP or memory functions are inferred from your RTL, ensure you follow the recommended coding styles described in *Recommended HDL Coding Styles* so that the optional input and output registers of the hard blocks are used. The Hyper-Retimer does not retime into or out of DSP and hard memory block registers. Hence, it is important to instantiate the optional registers in order to achieve maximum performance.

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

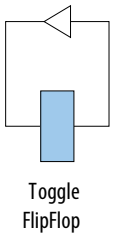
Related Links

- Recommended HDL Coding Styles
- True Dual-Port Memory on page 42

3.5.1.4 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 86. Simple Loop



When the critical chain is a feedback loop, the number of registers in a loop cannot be changed by the Hyper-Retimer without changing functionality. Retiming can be performed around a loop without changing functionality, but additional registers cannot be put in the loop. To explore performance gains, the Fast Forward Compile process adds registers at particular boundaries of the circuit, such as clock domain boundaries.

Figure 87. FIFO Flow Control Loop

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

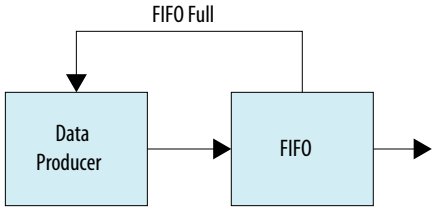


Figure 88. Counter and Accumulator Loop

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.

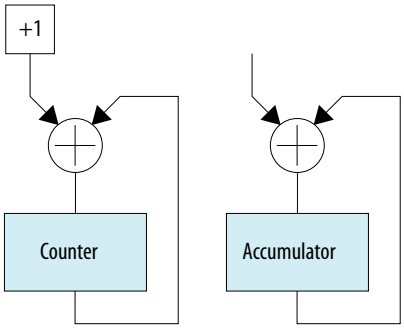


Figure 89. State Machine Loop

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

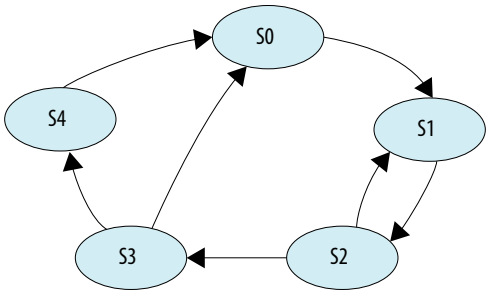
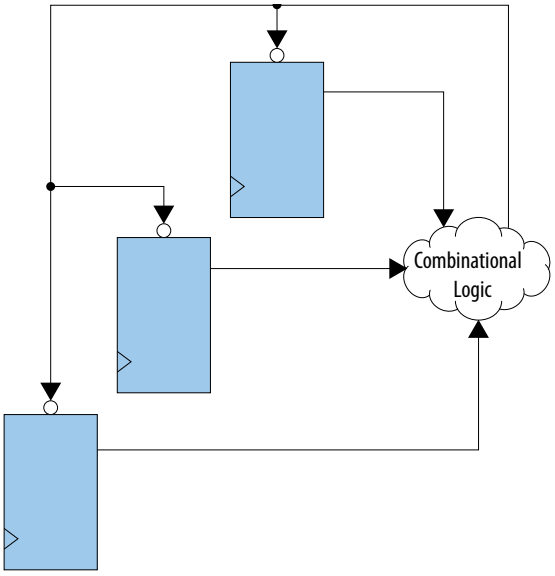


Figure 90. Reset Circuit Loop

Reset circuit loops include monitoring logic to reset if they get into an error condition.



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

3.5.1.4.1 Example of Critical Chain with Loops as the Limiting Reason

The following screenshots show the relevant panels from the Hyper-Retimer report and the logic contained in the critical chain.

Figure 91. Fast Forward Compile 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

	Step	Fast Forward Optimizations Analyzed	To Achieve Fmax	Slack	Relationship
1	Base Performance	None	284 MHz	4.480	8.000
2	Fast Forward Step #1 (Hyper-Retiming)	Removed asynchronous clears on 197 Registers	426 MHz	5.651	8.000
3	Fast Forward Step #2 (Hyper-Pipelining)	Removed user preserve assignments from 51 Registers 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, thus making it a loop.

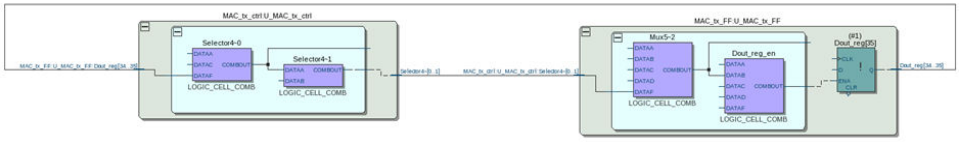
Figure 92. Critical Chain with Loop as Reported by the Hyper-Retimer

Limiting Critical Chain Before Hyper-Optimization

Optimizations Analyzed (Cumulative) | Critical Chain at Limit | Recommendations for Critical Chain

	Path Info	Register	Join	Element
1	Long Path (Critical)	REG	#1	MAC_tx|U_MAC_tx|MAC_tx_FF|U_MAC_tx_FF|Dout_reg[35]
2	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|Dout_reg[35]|q
3	Long Path (Critical)			MAC_tx|U_MAC_tx|MAC_tx_FF|U_MAC_tx_FF|Dout_reg[35]~la_lab/laboutb[10]
4	Long Path (Critical)			MAC_tx|U_MAC_tx|MAC_tx_FF|U_MAC_..._reg[35]~LAB_RE_X136_Y98_N0_I120
5	Long Path (Critical)	bypassed Hyper-Register		MAC_tx|U_MAC_tx|MAC_tx_FF|U_MAC_t...F|Dout_reg[35]~R3_X134_Y98_N0_I20
6	Long Path (Critical)	bypassed Hyper-Register		MAC_tx|U_MAC_tx|MAC_tx_FF|U_MAC_t...F|Dout_reg[35]~C4_X135_Y99_N0_I25
7	Long Path (Critical)	bypassed Hyper-Register		MAC_tx|U_MAC_tx|MAC_tx_FF|U_MAC...L_INTERCONNECT_X136_Y101_N0_I31
8	Long Path (Critical)			MAC_tx|U_MAC_tx|MAC_tx_FF|U_MAC_..._reg[35]~LAB_RE_X136_Y101_N0_I23
9	Long Path (Critical)	bypassed Hyper-Register		U_MAC_tx|U_MAC_tx_ctrl|Selector4~0|dataf
10	Long Path (Critical)			U_MAC_tx|U_MAC_tx_ctrl|Selector4~0|combout
11	Long Path (Critical)			MAC_tx|U_MAC_tx|MAC_tx_ctrl|U_MAC_tx_ctrl|Selector4~0~la_lab/laboutt[11]
12	Long Path (Critical)			MAC_tx|U_MAC_tx|MAC_tx_ctrl|U_MAC...ctor4~0~_LAB_RE_X136_Y101_N0_I101
13	Long Path (Critical)	bypassed Hyper-Register		MAC_tx|U_MAC_tx|MAC_tx_ctrl|U_MAC...|Selector4~0~_C4_X135_Y102_N0_I26
14	Long Path (Critical)	bypassed Hyper-Register		MAC_tx|U_MAC_tx|MAC_tx_ctrl|U_MAC_...rl|Selector4~0~_R3_X136_Y102_N0_I7
15	Long Path (Critical)	bypassed Hyper-Register		MAC_tx|U_MAC_tx|MAC_tx_ctrl|U_MA...AL_INTERCONNECT_X136_Y102_N0_I37
16	Long Path (Critical)	bypassed Hyper-Register		MAC_tx|U_MAC_tx|MAC_tx_ctrl|U_MAC...ector4~0~_LAB_RE_X136_Y102_N0_I10
17	Long Path (Critical)			U_MAC_tx|U_MAC_tx_ctrl|Selector4~1|dataa
18	Long Path (Critical)			U_MAC_tx|U_MAC_tx_ctrl|Selector4~1|combout
19	Long Path (Critical)			MAC_tx|U_MAC_tx|MAC_tx_ctrl|U_MAC_tx_ctrl|Selector4~1~la_lab/laboutt[7]
20	Long Path (Critical)			MAC_tx|U_MAC_tx|MAC_tx_ctrl|U_MAC...ector4~1~_LAB_RE_X136_Y102_N0_I97
21	Long Path (Critical)	bypassed Hyper-Register		MAC_tx|U_MAC_tx|MAC_tx_ctrl|U_MAC_...rl|Selector4~1~_C4_X135_Y98_N0_I35
22	Long Path (Critical)	bypassed Hyper-Register		MAC_tx|U_MAC_tx|MAC_tx_ctrl|U_MA...AL_INTERCONNECT_X136_Y101_N0_I39
23	Long Path (Critical)			MAC_tx|U_MAC_tx|MAC_tx_ctrl|U_MAC...ector4~1~_LAB_RE_X136_Y101_N0_I48
24	Long Path (Critical)	bypassed Hyper-Register		U_MAC_tx|U_MAC_tx_FF|Mux5~2|dataf
25	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|Mux5~2|combout
26	Long Path (Critical)			MAC_tx|U_MAC_tx|MAC_tx_FF|U_MAC_tx_FF|Mux5~2~la_lab/laboutb[4]
27	Long Path (Critical)			MAC_tx|U_MAC_tx|MAC_tx_FF|U_MAC...ux5~2~_LAB_RE_X136_Y101_N0_I114
28	Long Path (Critical)	bypassed Hyper-Register		MAC_tx|U_MAC_tx|MAC_tx_FF|U_MAC_tx_FF|Mux5~2~_R3_X134_Y101_N0_I15
29	Long Path (Critical)	bypassed Hyper-Register		MAC_tx|U_MAC_tx|MAC_tx_FF|U_MAC...L_INTERCONNECT_X136_Y101_N0_I47
30	Long Path (Critical)	bypassed Hyper-Register		MAC_tx|U_MAC_tx|MAC_tx_FF|U_MAC...Mux5~2~_LAB_RE_X136_Y101_N0_I35
31	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|Dout_reg_en|datab

Figure 93. 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 other inputs to the logic cone for the `RetryCnt[0]` register are not shown for clarity, but the following source code shows parts of the `MAC_tx_ctrl` source and some of the inputs to the `RetryCnt` registers.

Example 14. 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;
```

3.5.2 Details about Critical Chain Reports

The topics below apply to any type of critical chain.

3.5.2.1 One Critical Chain per Clock Domain

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

Look at other chains in your design to check if there are other areas of the design that you could optimize. You can see other chains by looking through the critical chains in each step of the Fast Forward Compile report. Each step of the Fast Forward Compile tests a set of changes such as removing or converting asynchronous clears and adding pipeline stages and reports the performance based on those changes.

Related Links

Critical Chains in Related Clock Groups on page 84

3.5.2.2 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-Retimer reports a critical chain for a Related Clock Group. The optimization techniques critical chain types also apply to critical chains in related clock groups.

3.5.2.3 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 Hyper-Retimer report panels. 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 5. 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
			f
	REG	#4	g
----------	----------	----------	----------
			g
			h
			a

Figure 94. 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.

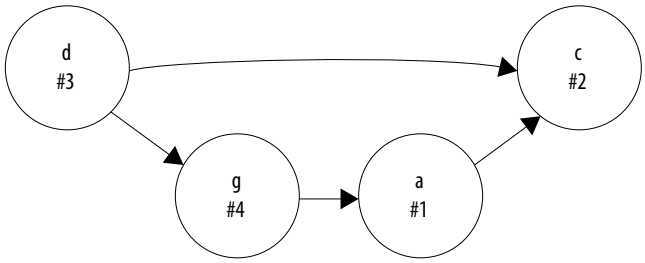
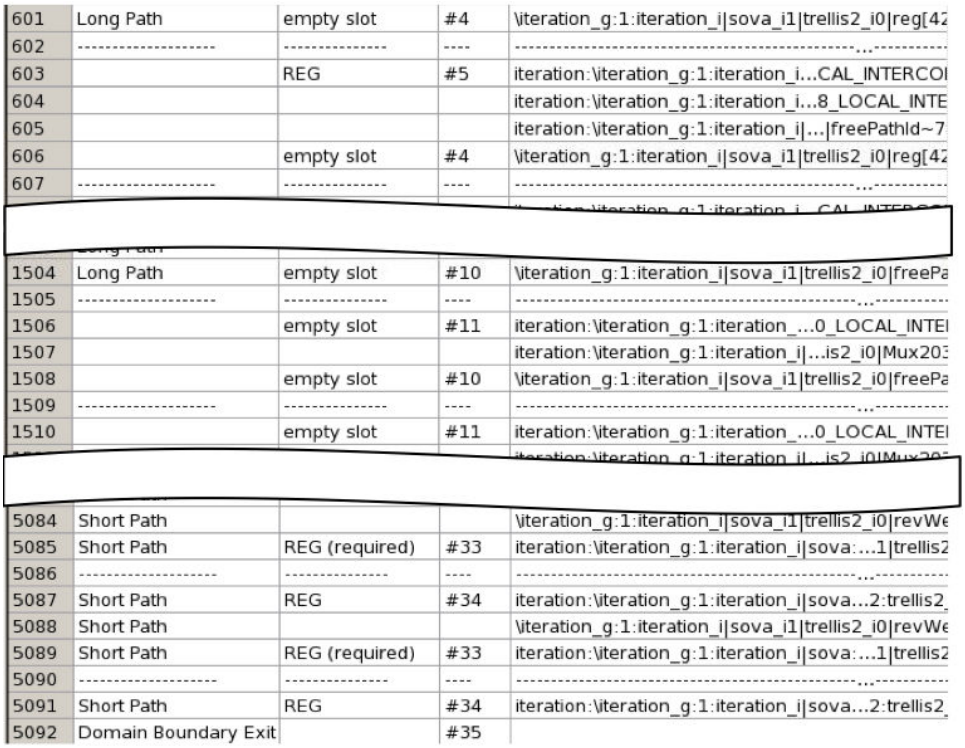


Figure 95. Complex Critical Chain

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	\iteration_g:1:iteration_i|sova_i1|trellis2_i0|reg[42
602	----------	----------	----	----------...----------
603		REG	#5	iteration:\iteration_g:1:iteration_i...CAL_INTERCOI
604				iteration:\iteration_g:1:iteration_i...8_LOCAL_INTE
605				iteration:\iteration_g:1:iteration_i|...|freePathId~7
606		empty slot	#4	\iteration_g:1:iteration_i|sova_i1|trellis2_i0|reg[42
607	----------	----------	----	----------...----------
1504	Long Path	empty slot	#10	\iteration_g:1:iteration_i|sova_i1|trellis2_i0|freePa
1505	----------	----------	----	----------...----------
1506		empty slot	#11	iteration:\iteration_g:1:iteration_...0_LOCAL_INTEI
1507				iteration:\iteration_g:1:iteration_i|...is2_i0|Mux203
1508		empty slot	#10	\iteration_g:1:iteration_i|sova_i1|trellis2_i0|freePa
1509	----------	----------	----	----------...----------
1510		empty slot	#11	iteration:\iteration_g:1:iteration_...0_LOCAL_INTEI
5084	Short Path			\iteration_g:1:iteration_i|sova_i1|trellis2_i0|revWe
5085	Short Path	REG (required)	#33	iteration:\iteration_g:1:iteration_i|sova:...1|trellis2
5086	----------	----------	----	----------...----------
5087	Short Path	REG	#34	iteration:\iteration_g:1:iteration_i|sova...2:trellis2_
5088	Short Path			\iteration_g:1:iteration_i|sova_i1|trellis2_i0|revWe
5089	Short Path	REG (required)	#33	iteration:\iteration_g:1:iteration_i|sova:...1|trellis2
5090	----------	----------	----	----------...----------
5091	Short Path	REG	#34	iteration:\iteration_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.

3.5.2.4 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 occurs most frequently with join points. When that happens, the critical chain segment is extends if necessary until it reaches a line that can be located to an RTL file.

3.5.2.5 Domain Boundary Entry and Domain Boundary Exit

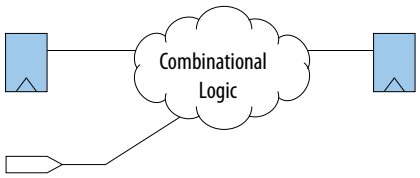
You can see Domain Boundary Entry or Domain Boundary Exit indicated in the **Path Info** column for a critical chain. The terms domain boundary entry and domain boundary exit refer to paths that are unconstrained, paths between asynchronous clock domains, or between a clock domain and top-level device 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 the Hyper-Retimer 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. The Hyper-Retimer can insert register stages at the input of a module, and perform forward retiming pushes, and it can 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 the Hyper-Retimer can remove register stages and the latency can exit the clock domain, if Hyper-Pipelining is enabled. The Hyper-Retimer removing a register may be counter intuitive. However, it may be necessary to retain functional correctness, depending on other optimizations performed by the Hyper-Retimer.

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 96. Domain Boundary with Unregistered Input/Output



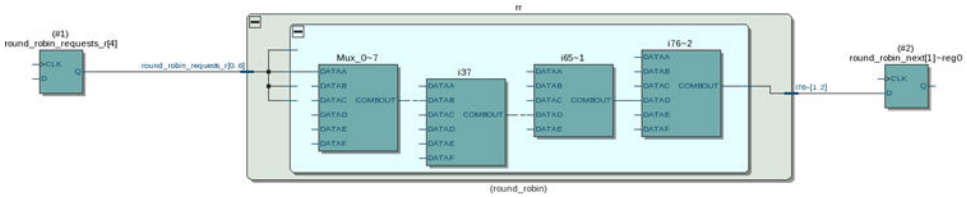
The register-to-register path might be shown as a critical chain segment with a domain boundary entry or a domain boundary exit, depending on how it restricted the Hyper-Retimer. The unregistered input prevents the Hyper-Retimer from inserting register stages at the domain boundary, because the input is unregistered. Likewise, the unregistered input can also prevent the Hyper-Retimer 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 prevented the Hyper-Retimer from retiming a register out of the clock domain. To determine why a register could not be retimed, you must look in your 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 97. Critical Chain Schematic with Domain Boundary



The limiting reason for the base compile is Insufficient Registers.

Figure 98. Fast Forward Compile Report with Insufficient Registers

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

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

Figure 99. Critical Chain with Domain Boundary Entry

Critical Chain Summary for Clock Domain clk

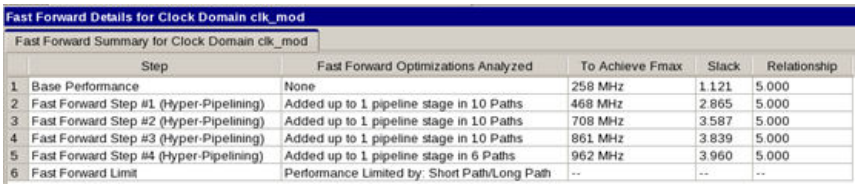
Critical Chain at Limit | Recommendations for Critical Chain

	Path Info	Register	Join	Element
1	Domain Boundary Entry	REG (required)	#1	round_robin_requests_r[4]
2	Long Path (Critical)			round_robin_requests_r[4]|q
3	Long Path (Critical)			round_robin_requests_r[4]~LAB_RE_X225_Y48_N0_I108
4	Long Path (Critical)	bypassed Hyper-Register		round_robin_requests_r[4]~C4_X224_Y48_N0_I10
5	Long Path (Critical)	bypassed Hyper-Register		round_robin_requests_r[4]~C4_X224_Y52_N0_I10
6	Long Path (Critical)	bypassed Hyper-Register		round_robin_requests_r[4]~C4_X224_Y56_N0_I10
7	Long Path (Critical)	bypassed Hyper-Register		round_robin_requests_r[4]~C4_X224_Y61_N0_I10
8	Long Path (Critical)	bypassed Hyper-Register		round_robin_requests_r[4]~LOCAL_INTERCONNECT_X225_Y63_N0_I14
9	Long Path (Critical)	bypassed Hyper-Register		round_robin_requests_r[4]~LAB_RE_X225_Y63_N0_I51
10	Long Path (Critical)			rr|Mux_0~7|dataa
11	Long Path (Critical)			rr|Mux_0~7|combout
12	Long Path (Critical)			rr|Mux_0~7~_LAB_RE_X225_Y63_N0_I119
13	Long Path (Critical)	bypassed Hyper-Register		rr|Mux_0~7~_C4_X225_Y63_N0_I17
14	Long Path (Critical)	bypassed Hyper-Register		rr|Mux_0~7~_LOCAL_INTERCONNECT_X225_Y65_N0_I50
15	Long Path (Critical)	bypassed Hyper-Register		rr|Mux_0~7~_LAB_RE_X225_Y65_N0_I76
16	Long Path (Critical)			rr|i37|datab
17	Long Path (Critical)			rr|i37|combout
18	Long Path (Critical)			rr|i37~LAB_RE_X225_Y65_N0_I130
19	Long Path (Critical)	bypassed Hyper-Register		rr|i37~LOCAL_INTERCONNECT_X225_Y65_N0_I28
20	Long Path (Critical)	bypassed Hyper-Register		rr|i37~LAB_RE_X225_Y65_N0_I49
21	Long Path (Critical)			rr|i65~1|datad
22	Long Path (Critical)			rr|i65~1|combout
23	Long Path (Critical)			rr|i65~1~_LAB_RE_X225_Y65_N0_I119
24	Long Path (Critical)	bypassed Hyper-Register		rr|i65~1~_LOCAL_INTERCONNECT_X225_Y65_N0_I49
25	Long Path (Critical)	bypassed Hyper-Register		rr|i65~1~_LAB_RE_X225_Y65_N0_I9
26	Long Path (Critical)			rr|i76~2|datad
27	Long Path (Critical)			rr|i76~2|combout
28	Long Path (Critical)			round_robin_next[1]~reg0|d
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 could insert 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 100. 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 #1 (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	--	--	--

3.5.2.6 Critical Chains with Dual Clock Memories

The Hyper-Retimer does not retime registers through dual clock memories. Therefore, it is possible that a functional block in your design that is between two dual clock FIFOs or memories could be reported as the critical chain, with a limiting reason of Insufficient Registers, even when you perform a Fast Forward Compile. If you have a critical chain with a limiting reason of Insufficient Registers, and the chain is between dual clock memories, you can add pipeline stages to the functional block. You can also add a bank of registers in the RTL design and allow the Hyper-Retimer to balance the registers. Refer to *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 Hyper-Retimer 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: Parameterizable Pipeline Modules on page 50

3.5.2.7 Critical Chain Bits and Buses

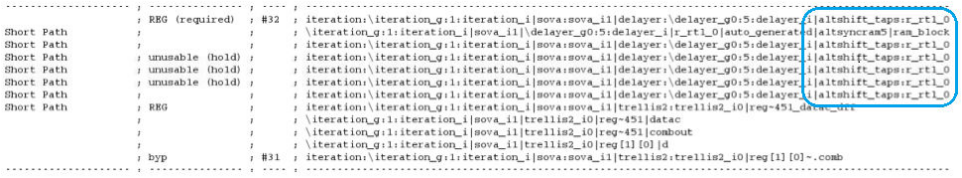
The critical chain of a design commonly includes registers that are single bits in a wider bus or register bank. When you analyze such a critical chain, focus on the bus as a whole, instead of analyzing the structure related to the single bit. For example, a critical chain that refers to bit 10 in a 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.

3.5.2.7.1 Delay Lines

You may have a parameterized module that delays a bus by some number of clock cycles. Sometimes that kind of structure is converted during synthesis to an ALTSHIFT_TAPS Megafunction. The following screenshot shows part of a critical chain with a delay module that has been converted to an ALTSHIFT_TAPS Megafunction. The highlighted section at the right-hand end shows a design hierarchy of `altshift_taps:r_rtl_0`, indicating that synthesis replaces the bank of registers with the ALTSHIFT_TAPS IP core. Parts of the ALTSHIFT_TAPS IP core cause the critical chain segment categorization as a short path.

Figure 102. Critical Chain Report with Delay Line



The Fitters places the chain of registers so close together that the hold time cannot be met if the Hyper-Retimer uses any of the intermediate Hyper-Register locations. Turning off **Auto Shift Register Replacement** for the bank of registers would prevent synthesis from using the ALTSHIFT_TAPS Megafunction and probably resolve the 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. A RAM-based implementation can be a compact way to delay a bus of data if you do not need to move it a long distance across the chip.

3.6 Retiming Restrictions and Workarounds

This section describes RTL design techniques you can use to avoid retiming restrictions. There are a variety of situations that cause retiming restrictions. Some exist because of hardware characteristics, some exist because of software behavior, and some are inherent in a design. You can avoid many of the following retiming restrictions with RTL design techniques, although some restrictions are inherent in a design.

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 situation occurs in the design, the registers involved are not retimed. This ensures that the Hyper-Retimer 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. Any register that has an asynchronous clear cannot be retimed into a Hyper-Register. 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 in 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 Hyper-Retimer 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 Hyper-Retimer 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 Hyper-Retimer retimes registers in ALMs, DDIOs, single port RAMs, and DSP blocks.

continued...

Design Condition	Hyper-Register Support
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 for incremental compilation	This condition is inherent in any design that uses incremental compilation. If this retiming restriction prevents you from achieving the required performance, add additional registers inside the partition boundary for the Hyper-Retimer to optimize.
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. Sometimes, latches are inferred because of RTL coding issues, such as with 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 the Hyper-Retimer to optimize.
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	Hyper-Registers are clocked by only the dedicated clock network. 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 Hyper-Retimer does not retime registers that are sources or destinations of SDC constraints.
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 Hyper-Retimer 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 Hyper Retimer.
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 once frequency, and fans out to registers running at another frequency. The Hyper-Retimer 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 the Hyper-Retimer.

Related Links

- Timing Constraint Considerations on page 23
  This section recommends specific timing constraint techniques to maximize performance.
- Synchronizers on page 25

4 HyperFlex Porting Guidelines

This document and the Quartus Prime Pro – Stratix 10 Edition Beta software provide step-by-step recommendations to increase design performance using the following code optimization techniques to leverage the Stratix 10 HyperFlex architecture:

- Hyper-Retiming
- Hyper-Pipelining
- Hyper-Optimization

The Quartus Prime Pro – Stratix 10 Edition Beta software focuses on core performance exploration. To experiment with performance exploration, use a large, second level module that does not contain periphery IP (transceiver, memory, etc.). For designs that include specialized IP modules, this document provides Stratix 10 migration guidelines for Stratix V or Arria 10 designs.

For the purposes of performance exploration, focus on the relative performance improvements that can be made when you take advantage of the HyperFlex architecture. Once you implement the design optimizations suggested by Fast Forward Compile in your design, you can realize the performance gains and move toward timing closure at the full chip level.

4.1 Suggested Scope for Performance Exploration and Design Migration

You can make specific RTL changes to optimize performance for Stratix 10 designs. This increased speed can help you close timing, or provide flexibility to add additional functionality to your design.

When you convert a design for Stratix 10 FPGAs, the *required* RTL changes are minor and similar to the changes required any time you switch to a new device family. Required changes include device-specific changes, such as updating PLLs, high-speed I/O pins, and other resources. These components have the same general functionality. However, these components include features to enable higher operational speeds. For example:

- DSP blocks have added pipeline registers and support a floating point mode.
- Memory blocks have additional logic for coherency and some restrictions related to the width.

Some additional design modifications can enable retiming optimizations to take advantage of the Stratix 10 architecture and achieve dramatic performance improvements.

Start the migration process by picking a lower level block in the design. This block should not contain any specialized IPs, such as transceivers. Migrating all the individual IPs could be quite time consuming. This migration is not the most effective use of the performance exploration at this stage.

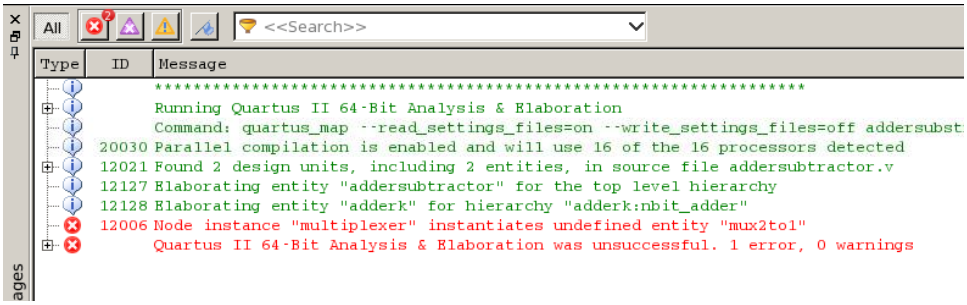
Black-box any special IP component and only keep components which are required for the current level you have selected. Only keep the following key blocks for core performance exploration:

- PLLs for generating clocks
- Core Blocks (Logic, Registers, Memories, DSPs)

Note: If you are migrating the design from a previous version of the Quartus software, you might have to replace some components if they are incompatible with or unavailable in the current software version.

When black-boxing components, maintain the module port definition. You cannot simply remove the source file from the project. You must specify the port definition and direction of every component used in the design to the synthesis software. Failure define the ports results in compilation errors. Check the error messages and fix any missing port/module definitions.

Figure 103. Compilation Error Messages



The easiest way to black-box a module is to empty its functional content. Below are examples for black-boxing content depending on whether you are using Verilog HDL or VHDL:

4.1.1 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
```

4.1.2 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 mux2to1 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 mux2to1 IS
BEGIN
END Behavior ;
```

In addition to black-boxing the modules you are not interested in, you must put them into their own partition to separate them from the rest of the design. Putting black-boxed modules into an empty partition prevents the logic connected to the black-boxed modules from being optimized away during synthesis.

To create a new partition:

1. Create a new partition.
2. Set it to **Empty**.
3. Add all the black-box modules into this partition.

Figure 104. Create New Partition

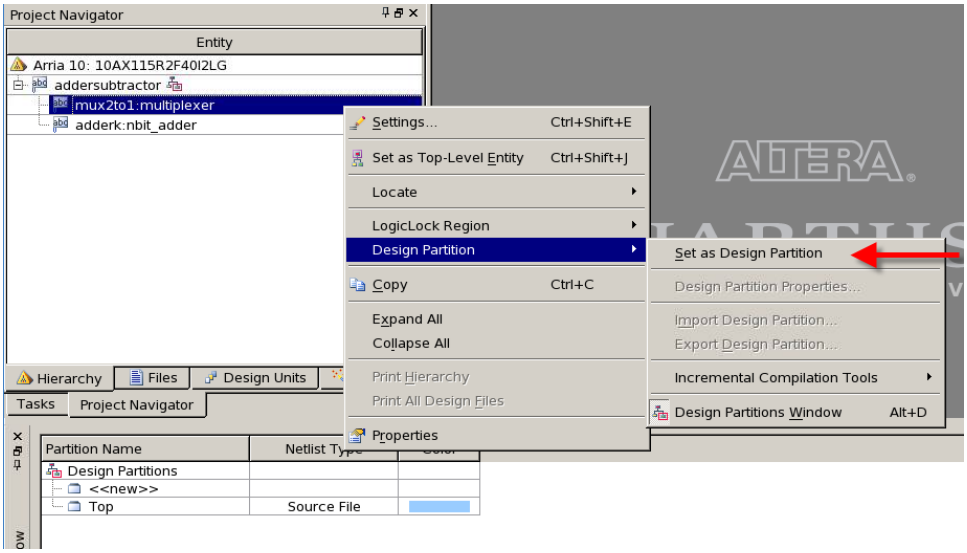
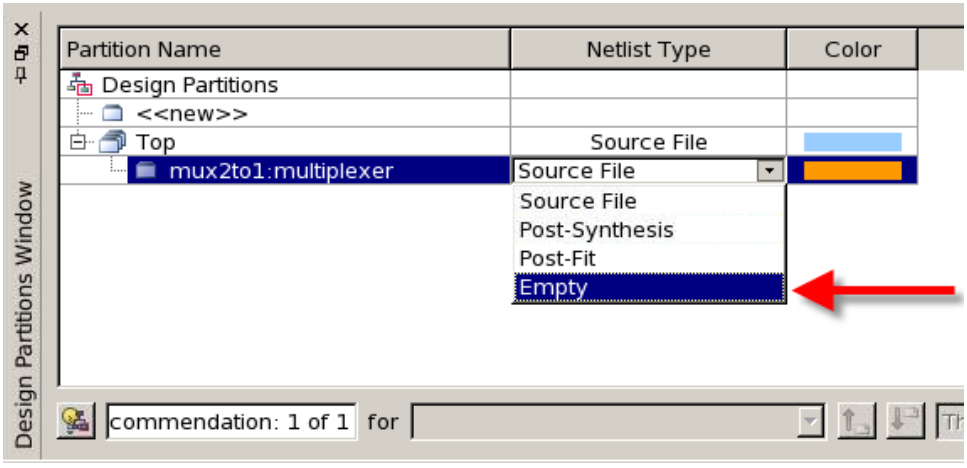


Figure 105. Set Partition to Empty



4.1.3 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. Pay close attention to any clock which could be coming out of a black-boxed module. If this occurs in your design, you have to recreate this clock. Failure to recreate the clock marks any register downstream as unclocked. This changes the logic function of your design, because registers that do not receive a clock could be removed by the synthesis engine during optimization.

Examine the clock definitions in the SDC file to determine if a clock is created in one of the black-boxed modules. Looking at a particular module, several cases can happen:

- 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 you do not have to do anything.
    - Yes: create a clock on the output pin of that module matching the definition you found in the SDC.
- There is no clock definition in that module
  - Is there a clock feedthrough path in that module?
    - No: you do not have to do anything.
    - Yes: create a new clock on the feedthrough output pin of the module.

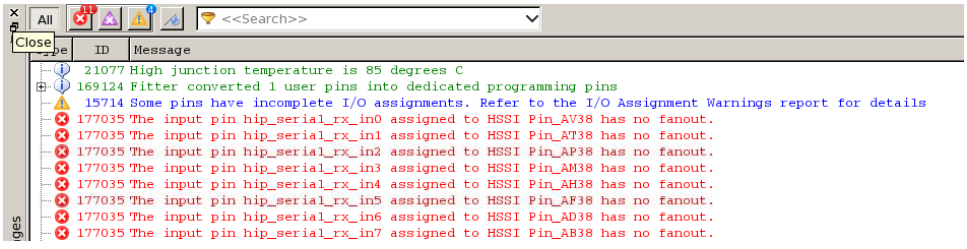
4.1.4 Pin Assignments

Once you start black-boxing logic, you might encounter some pin assignment issues. Below are common situations to look for:

- Input pins which have been configured for high speed communication

The Quartus Prime Pro – Stratix 10 Edition Beta checks for the status of high-speed pins and generates some errors if these pins are unconnected in the design. 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. You might have to change the I/O bank as well.

Figure 106. High-speed Pin Error Messages

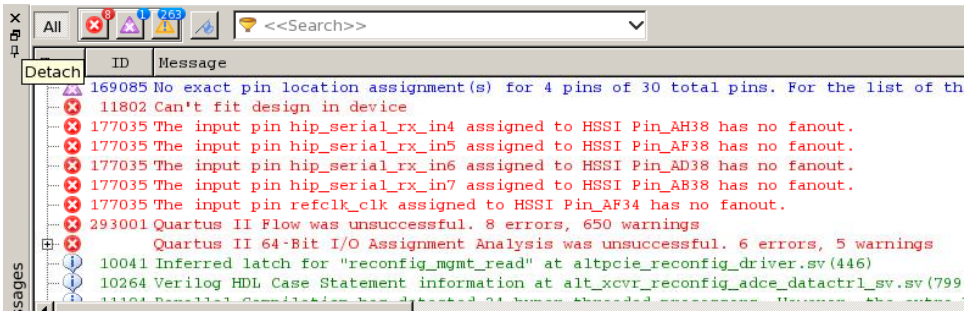


In the `.qsf` file, it translates to the following:

```
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 107. Pins Error Messages



Dangling pins

If you have high-speed I/O pins dangling because of black-boxing components, set them to virtual pins. You can 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 *
```

4.1.5 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.

4.1.6 Upgrade Outdated IP Cores

If you have outdated IP components in your design, click **Project ➤ Upgrade IP Components** to upgrade the components to the latest version. For example, EMIF logic is a good candidate for this. You must upgrade every IP component that is not black-boxed to the current version.

Note: Some IP components might not be supported in Stratix 10 designs. If those components are critical (for example, PLL), you must modify your design and replace them with Stratix 10-compatible IP components.

4.2 Top-Level Design Considerations

Consider the following top-level design issues:

- I/O constraints
- Reset logic
- DSP and M20K blocks

Wrap the top-level in a register ring.

I/O constraints

In order to get the maximum performance of the Hyper-Retimer engine, remove the following constraints from your SDC file:

- `set_input_delay`
- `set_output_delay`

These constraints model how much time out of a given clock period is used outside of the block itself. For the purposes of analyzing the effect of design optimizations, you want to use all the available slack within the block itself. This helps maximize performance at the module level. These constraints can be added back when moving to full chip timing closure.

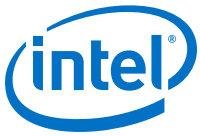
Resets

If you remove reset generation from the design, you have to provide a replacement signal. The easiest way to do this is to connect it directly to an input pin of your design.

Although this is technically correct, it might affect the retiming capabilities in Stratix 10 architectures. Add a couple of pipeline stages to your reset signal. This helps the Hyper-Retimer 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, you can manually recompile these blocks. Look for the following conditions:



- DSPs: Watch the pipelining depth. More pipeline stages results in a faster design. If you see that retiming is limited by the logic levels in a DSP block, add more pipeline stages.
- M20Ks: Retiming relies heavily on the presence of registers to move logic around. With M20K blocks, you can help the Quartus Prime Pro – Stratix 10 Edition Beta 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 the way the block is driven when embedded in the full design. The ring also avoids the retiming restriction associated with registers connected to inputs/outputs. The first and last level of registers should now be able to retime more realistically.

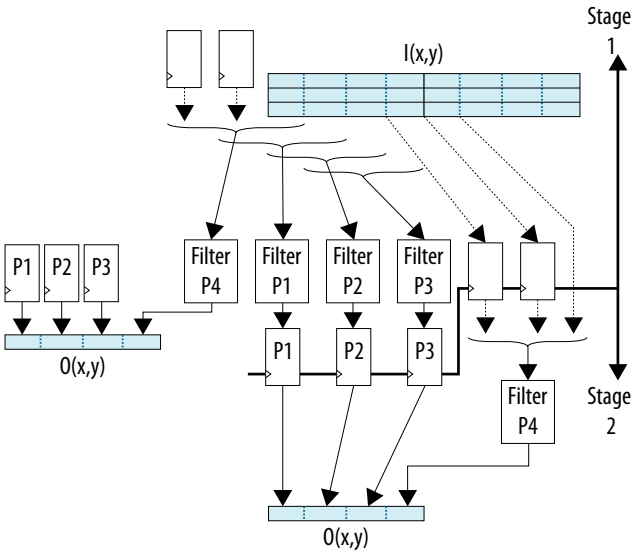
4.3 Summary

These guidelines allow you to quickly and easily evaluate the benefits of design optimizations that leverage the Stratix 10 HyperFlex architecture, while still preserving your design's functional intent. Even though these suggestions require minor modifications to the design, they are meant to get your design to a stage where you can quickly apply the suggested design optimizations, and achieve major performance gains in your design's most critical modules before going to the full chip level for timing closure.

5 Design Example Walk-Through

This section illustrates use of Fast-Forward Compilation and register movement (Hyper-Retiming) to improve performance in a real-world median filter design example. This walk-through describes project setup, design compilation, interpreting results, and optimizing RTL.

Figure 108. Median Filter Operational Diagram



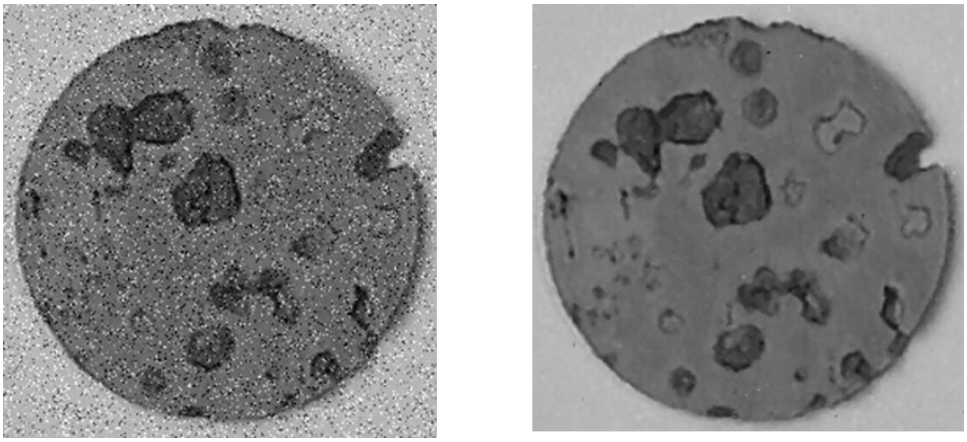
5.1 Median Filter Design Example

This walk-through provides a small image processing median filter design example to illustrate use of Fast-Forward Compilation and Hyper-Retiming.

Note: Intel provides supporting design example project and design files for this walk-through. Download the supporting `median.zip` file available with this document. Unzip the file to use and refer to a complete, verified design example, including all project files, constraint files, design files, and example RTL.

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.[1]

Figure 109. Before and After Images Processed with Median Filtering



5.1.1 Step 1: Setup the Project

Follow these instructions to setup the Median Filter design example project. The design example project includes the `median.sdc` file that defines the single clock single that drives the design. The design example uses this clock definition throughout the design flow.

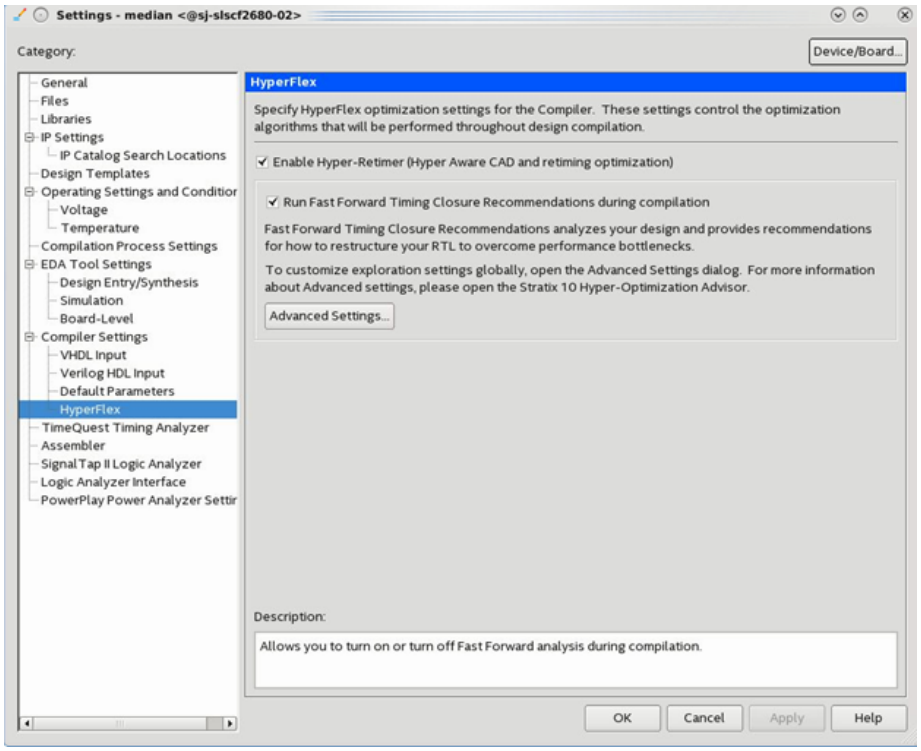
1. Download and extract the Median Filter design example.
2. Open the `median.qsf` project in the Quartus Prime Pro – Stratix 10 Edition Beta software.
3. Click **Assign ➤ Device** and confirm the following device assignment settings:
   - **Device family**: **Stratix 10 (GX/SX).**
   - **Device**: **1SG280LN3F43E1VG (Advanced)**
4. All I/Os in the example design are set as "virtual" pins, meaning that the Fitter does not actually connect them to real device pins. To view the state of these virtual pins, click **Assignments ➤ Assignment Editor**.
5. To enable the Hyper-Retimer and perform Hyper-Optimizations, confirm the following setting in the `median.qsf` settings file for the project:
   `set_global_assignment -name HYPER_RETIMER ON`
6. Click **Assignments ➤ Settings ➤ Compiler Settings ➤ HyperFlex** and confirm the following settings:

1 This median filter design was first presented in a paper titled "An FPGA-Based Implementation for Median Filtering Meeting the Real-Time Requirements of Automated Visual Inspection Systems" at 10th Mediterranean Conference on Control and Automation, Lisbon, Portugal, 2002. The design is publicly available under GNU General Public License as published by the Free Software Foundation.

- Enable **Hyper-Retimer**
- Enable **Run Fast Forward Timing Closure Recommendations**

7. Click **Tools ➤ TimeQuest Timing Analyzer** and define all clocks and specify a 1 GHz clock frequency requirement. TimeQuest settings save to a Synopsys Design Constraints (`.sdc`) file. Alternatively, you can create the `.sdc` file manually.
8. Click **Assignments ➤ Assignment Editor** and assign the **Virtual Pin** option to all pins.

Figure 110. HyperFlex Settings



5.1.2 Step 2: Run Fast-Forward Compilation

To run Fast Forward Compile, click **Processing ➤ Start Compilation**. Alternatively, you can run Fast Forward Compile as a separate process by double-clicking **Generate Fast Forward Timing Closure Recommendations** in the **Tasks** pane. You can run Fast Forward Compile at the command line by typing `quartus_fit --fastforward`. Run Fast Forward Compile after running the Fitter. Your design must successfully run through the previous compilation stages before Fast Forward Compile.

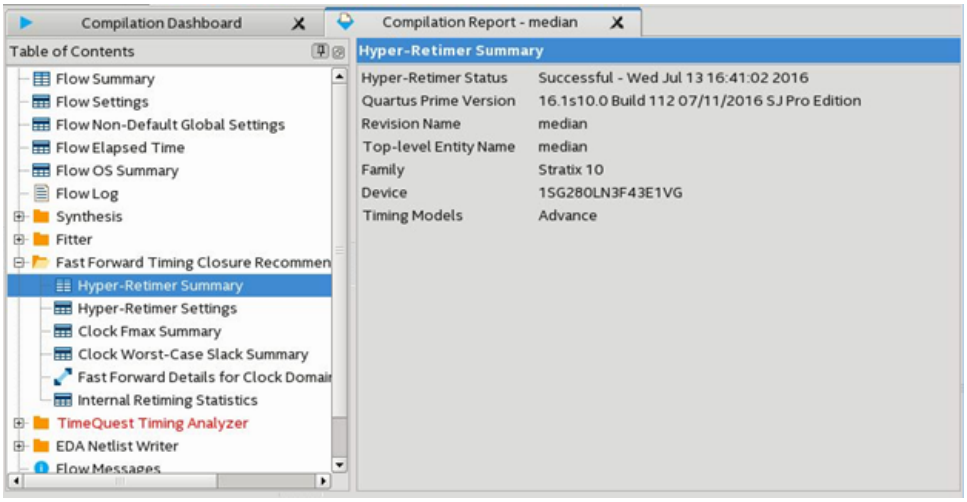
5.1.3 Step 3: View Fast-Forward Recommendations

The Compiler generates detailed reports following processing. View the results of Hyper-Retiming in the Hyper-Retimer section of the Compilation Report.

5.1.3.1 Hyper-Retimer Summary Report

The Hyper-Retimer Summary reports provides an overview of conditions during the Hyper-Retimer stage.

Figure 111. Hyper-Retimer Summary Report



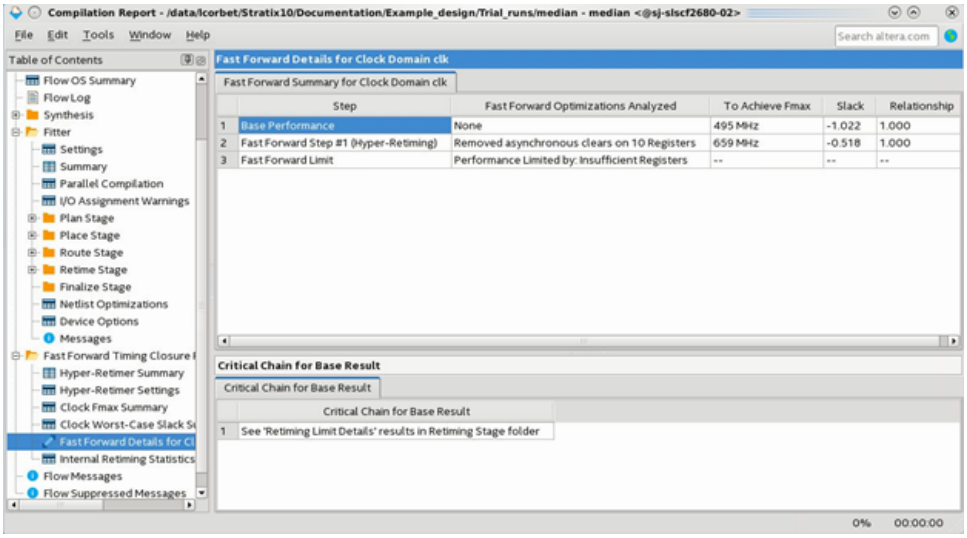
The report lists the number of registers remaining in adaptive logic modules (ALM), compared with the number of registers moved to Hyper-Registers (located in the routing network). In this example, 56 registers remain in traditional locations in the ALM and 281 Hyper-Register locations are used.

5.1.3.2 Fast-Forward Details Report

The Fast-Forward Details reports the following information about each clock domain:

- **Base Performance**—the first line of the report lists the "Base Performance, reflecting the current f_{MAX} without any RTL changes.
- **Fast Forward Steps**—the next few lines of the report list the specific Hyper-Retiming steps the Compiler implements to improve design performance. The report numbers these lines in order of execution.
- **Fast Forward Limit**—the last line of the report lists the critical limiting factor remaining after the Hyper-Retiming modifications are complete.

Figure 112. Fast-Forward Details Report

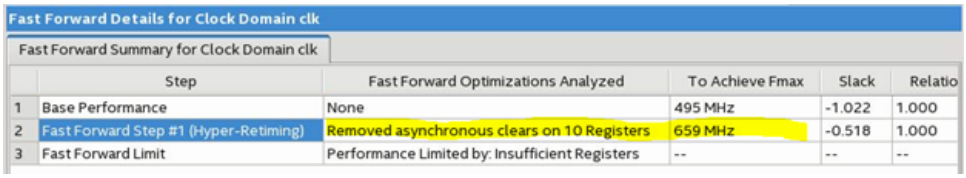


5.1.3.3 Critical Chain Reports

Select any report line to display detailed information in the critical chain report. The **Base Performance** step refers to the Retime Stage -> Retiming Limit Details report that shows the current timing restrictions for this example design.

Fast-Forward Step #1 suggests significant improvement is achievable after removal of asynchronous signals.

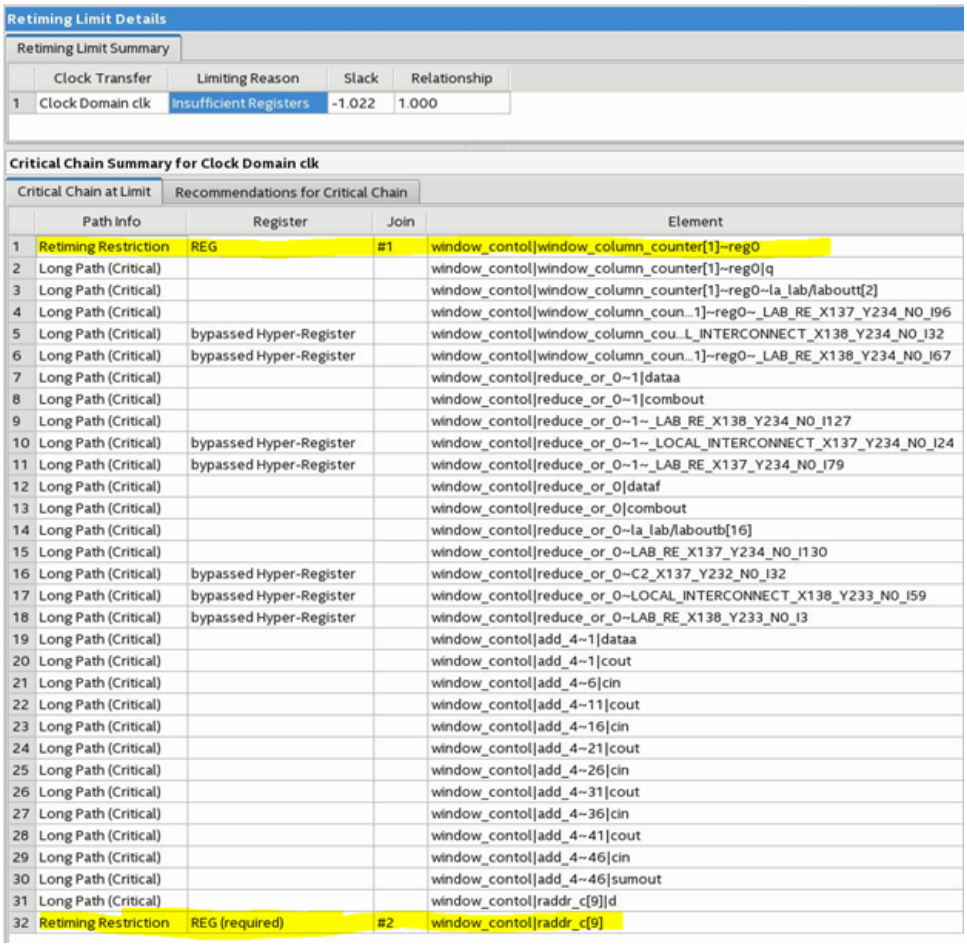
Figure 113. Fast-Forward Step #1



Removal of this asynchronous registers enables an estimated performance increase of 160MHz in the f_{MAX} of this example design. The report provides the location of the asynchronous signals for easy modification.

Click the **Critical Chain at Limit** tab in the Retiming Limit Details report to display detailed descriptions of the path(s) responsible for f_{MAX} limitations. In this example design, some registers have the "REG (required)" distinction. This timing restriction is related to the presence of asynchronous signals

Figure 114. Critical Chain for Base Result



Retiming Limit Details

Retiming Limit Summary

	Clock Transfer	Limiting Reason	Slack	Relationship
1	Clock Domain clk	Insufficient Registers	-1.022	1.000

Critical Chain Summary for Clock Domain clk

Critical Chain at Limit | Recommendations for Critical Chain

	Path Info	Register	Join	Element
1	Retiming Restriction	REG	#1	window_contol|window_column_counter[1]~reg0
2	Long Path (Critical)			window_contol|window_column_counter[1]~reg0|q
3	Long Path (Critical)			window_contol|window_column_counter[1]~reg0~la_lab/laboutt[2]
4	Long Path (Critical)			window_contol|window_column_coun...1]~reg0~_LAB_RE_X137_Y234_N0_I96
5	Long Path (Critical)	bypassed Hyper-Register		window_contol|window_column_cou...L_INTERCONNECT_X138_Y234_N0_I32
6	Long Path (Critical)	bypassed Hyper-Register		window_contol|window_column_coun...1]~reg0~_LAB_RE_X138_Y234_N0_I67
7	Long Path (Critical)			window_contol|reduce_or_0~1|dataa
8	Long Path (Critical)			window_contol|reduce_or_0~1|combout
9	Long Path (Critical)			window_contol|reduce_or_0~1~_LAB_RE_X138_Y234_N0_I127
10	Long Path (Critical)	bypassed Hyper-Register		window_contol|reduce_or_0~1~_LOCAL_INTERCONNECT_X137_Y234_N0_I24
11	Long Path (Critical)	bypassed Hyper-Register		window_contol|reduce_or_0~1~_LAB_RE_X137_Y234_N0_I79
12	Long Path (Critical)			window_contol|reduce_or_0|dataf
13	Long Path (Critical)			window_contol|reduce_or_0|combout
14	Long Path (Critical)			window_contol|reduce_or_0~la_lab/laboutb[16]
15	Long Path (Critical)			window_contol|reduce_or_0~LAB_RE_X137_Y234_N0_I130
16	Long Path (Critical)	bypassed Hyper-Register		window_contol|reduce_or_0~C2_X137_Y232_N0_I32
17	Long Path (Critical)	bypassed Hyper-Register		window_contol|reduce_or_0~LOCAL_INTERCONNECT_X138_Y233_N0_I59
18	Long Path (Critical)	bypassed Hyper-Register		window_contol|reduce_or_0~LAB_RE_X138_Y233_N0_I3
19	Long Path (Critical)			window_contol|add_4~1|dataa
20	Long Path (Critical)			window_contol|add_4~1|cout
21	Long Path (Critical)			window_contol|add_4~6|cin
22	Long Path (Critical)			window_contol|add_4~11|cout
23	Long Path (Critical)			window_contol|add_4~16|cin
24	Long Path (Critical)			window_contol|add_4~21|cout
25	Long Path (Critical)			window_contol|add_4~26|cin
26	Long Path (Critical)			window_contol|add_4~31|cout
27	Long Path (Critical)			window_contol|add_4~36|cin
28	Long Path (Critical)			window_contol|add_4~41|cout
29	Long Path (Critical)			window_contol|add_4~46|cin
30	Long Path (Critical)			window_contol|add_4~46|sumout
31	Long Path (Critical)			window_contol|raddr_c[9]|d
32	Retiming Restriction	REG (required)	#2	window_contol|raddr_c[9]

5.1.4 Step 4: Implement Fast-Forward Recommendations

Fast Forward Compile recommends Hyper-Retiming, Hyper-Pipelining, and Hyper-Optimization in your design. Review and implement the suggested changes in your design RTL to realize the predictive performance gains. After implementing RTL changes, recompile the design and view the impact in the Compilation report.

Fast Forward Compile recommends RTL changes to improve performance. For example, the presence of asynchronous signals (resets in the design) affect retiming abilities. As a starting point for handing resets, ensure that resets are synchronous. Refer to other sections of this document for detailed information about reset strategies.

Figure 115. Asynchronous Reset in State Machine RTL

```
always @(posedge clk or negedge rst_n)
begin : out_memory_counter
    if(~rst_n) begin
        waddr <= {LUT_ADDR_WIDTH{1'b0}};
    end else if(valid) begin
        waddr <= waddr + 1'b1;
    end
end

always @(posedge clk or negedge rst_n)
begin : addr_counter
    if(~rst_n) begin
        window_column_counter <= 10'd0;
        window_line_counter <= 2'b00;
        raddr_a <= {LUT_ADDR_WIDTH{1'b0}};
        raddr_b <= {LUT_ADDR_WIDTH{1'b0}};
        raddr_c <= {LUT_ADDR_WIDTH{1'b0}};
    end else begin
        if(window_column_counter != ((IMG_WIDTH/4)-1)) begin
            window_column_counter <= window_column_counter + 1'b1;
            valid <= 1'b1;
            raddr_a <= raddr_a + 1'b1;
            raddr_b <= raddr_b + 1'b1;
            raddr_c <= raddr_c + 1'b1;
```

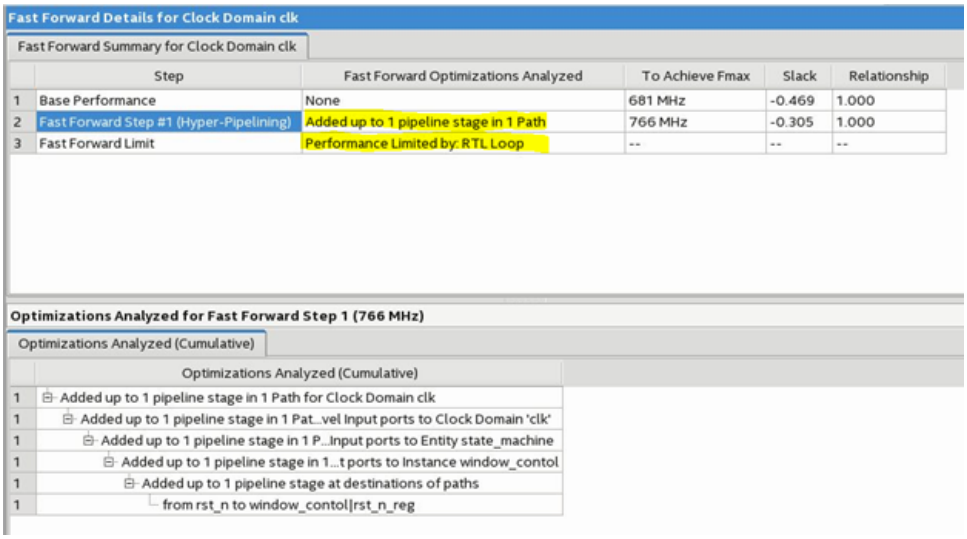
Figure 116. Synchronous Reset in State Machine RTL

```
always @(posedge clk)
begin : out_memory_counter
    if(~rst_n) begin
        waddr <= {LUT_ADDR_WIDTH{1'b0}};
    end else if(valid) begin
        waddr <= waddr + 1'b1;
    end
end

always @(posedge clk)
begin : addr_counter
    if(~rst_n) begin
        window_column_counter <= 10'd0;
        window_line_counter <= 2'b00;
        raddr_a <= {LUT_ADDR_WIDTH{1'b0}};
        raddr_b <= {LUT_ADDR_WIDTH{1'b0}};
        raddr_c <= {LUT_ADDR_WIDTH{1'b0}};
```

You can modify the RTL as suggested to improve performance. Asynchronous signals, combination of short and long paths where extra registers are required, are common RTL structures that limit design performance. Logical loops also present challenges for design performance. The Compiler cannot retime any register into or out of a loop. This restriction helps to ensure that Hyper-Retiming does not change the logic function of the design.

After changes to synchronous reset, the design example still benefits from an addition stage of pipeline registers. The Hyper-Retimer reports the specific location to add these registers. You can add registers at either the path source or destination. The Compiler choses the optimal location automatically.

Figure 117. Fast Forward Compilation Improvement



Fast Forward Details for Clock Domain clk

Fast Forward Summary for Clock Domain clk

	Step	Fast Forward Optimizations Analyzed	To Achieve Fmax	Slack	Relationship
1	Base Performance	None	681 MHz	-0.469	1.000
2	Fast Forward Step #1 (Hyper-Pipelining)	Added up to 1 pipeline stage in 1 Path	766 MHz	-0.305	1.000
3	Fast Forward Limit	Performance Limited by: RTL Loop	--	--	--

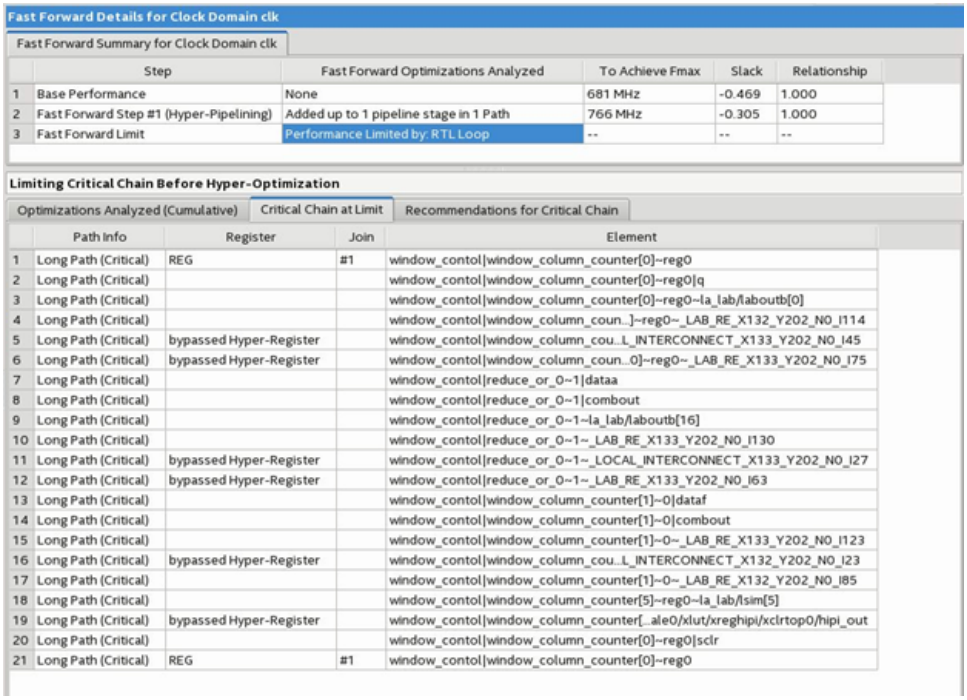
Optimizations Analyzed for Fast Forward Step 1 (766 MHz)

Optimizations Analyzed (Cumulative)

	Optimizations Analyzed (Cumulative)
1	Added up to 1 pipeline stage in 1 Path for Clock Domain clk
1	Added up to 1 pipeline stage in 1 Pat...vel Input ports to Clock Domain 'clk'
1	Added up to 1 pipeline stage in 1 P...Input ports to Entity state_machine
1	Added up to 1 pipeline stage in 1...t ports to Instance window_contol
1	Added up to 1 pipeline stage at destinations of paths
1	from rst_n to window_contol|rst_n_reg

RTL loops are one of the most significant factors impacting f_{MAX} performance. You can view more detail about this limit in the Fast Forward Limit report. The report lists the paths in the loop and all the convergence points. Depending on the size of the loop, it can be somewhat difficult to visualize from the report. You can alternatively visualize the critical chains by **Right-clicking ➤ Locate Critical Chain**. The Technology Map Viewer abstracts combinational logic with cloud icons. Expand this logic by clicking on the + sign.

Figure 118. Fast Forward Limit



Fast Forward Details for Clock Domain clk

Fast Forward Summary for Clock Domain clk

	Step	Fast Forward Optimizations Analyzed	To Achieve Fmax	Slack	Relationship
1	Base Performance	None	681 MHz	-0.469	1.000
2	Fast Forward Step #1 (Hyper-Pipelining)	Added up to 1 pipeline stage in 1 Path	766 MHz	-0.305	1.000
3	Fast Forward Limit	Performance Limited by: RTL Loop	--	--	--

Limiting Critical Chain Before Hyper-Optimization

Optimizations Analyzed (Cumulative) | Critical Chain at Limit | Recommendations for Critical Chain

	Path Info	Register	Join	Element
1	Long Path (Critical)	REG	#1	window_contol|window_column_counter[0]~reg0
2	Long Path (Critical)			window_contol|window_column_counter[0]~reg0|q
3	Long Path (Critical)			window_contol|window_column_counter[0]~reg0~la_lab/laboutb[0]
4	Long Path (Critical)			window_contol|window_column_coun...]~reg0~_LAB_RE_X132_Y202_N0_I114
5	Long Path (Critical)	bypassed Hyper-Register		window_contol|window_column_cou...L_INTERCONNECT_X133_Y202_N0_I45
6	Long Path (Critical)	bypassed Hyper-Register		window_contol|window_column_coun...0]~reg0~_LAB_RE_X133_Y202_N0_I75
7	Long Path (Critical)			window_contol|reduce_or_0~1|dataa
8	Long Path (Critical)			window_contol|reduce_or_0~1|combout
9	Long Path (Critical)			window_contol|reduce_or_0~1~la_lab/laboutb[16]
10	Long Path (Critical)			window_contol|reduce_or_0~1~_LAB_RE_X133_Y202_N0_I130
11	Long Path (Critical)	bypassed Hyper-Register		window_contol|reduce_or_0~1~_LOCAL_INTERCONNECT_X133_Y202_N0_I27
12	Long Path (Critical)	bypassed Hyper-Register		window_contol|reduce_or_0~1~_LAB_RE_X133_Y202_N0_I63
13	Long Path (Critical)			window_contol|window_column_counter[1]~0|dataf
14	Long Path (Critical)			window_contol|window_column_counter[1]~0|combout
15	Long Path (Critical)			window_contol|window_column_counter[1]~0~_LAB_RE_X133_Y202_N0_I123
16	Long Path (Critical)	bypassed Hyper-Register		window_contol|window_column_cou...L_INTERCONNECT_X132_Y202_N0_I23
17	Long Path (Critical)			window_contol|window_column_counter[1]~0~_LAB_RE_X132_Y202_N0_I85
18	Long Path (Critical)			window_contol|window_column_counter[5]~reg0~la_lab/lsim[5]
19	Long Path (Critical)	bypassed Hyper-Register		window_contol|window_column_counter[...ale0/xlut/xreghipi/xclrtop0/hipi_out
20	Long Path (Critical)			window_contol|window_column_counter[0]~reg0|sclr
21	Long Path (Critical)	REG	#1	window_contol|window_column_counter[0]~reg0

As shown in the report, the loop involves register `window_column_counter`. We can review and modify the RTL to improve performance.

Figure 119. Non-Optimized RTL

```
always @(posedge clk)
begin : addr_counter
    if(~rst_n_reg) begin
        window_column_counter <= 10'd0;
        window_line_counter <= 2'b00;
        raddr_a <= {LUT_ADDR_WIDTH{1'b0}};
        raddr_b <= {LUT_ADDR_WIDTH{1'b0}};
        raddr_c <= {LUT_ADDR_WIDTH{1'b0}};
    end else begin
        if(window_column_counter != ((IMG_WIDTH/4)-1)) begin
            window_column_counter <= window_column_counter + 1'b1;
            valid <= 1'b1;
            raddr_a <= raddr_a + 1'b1;
            raddr_b <= raddr_b + 1'b1;
            raddr_c <= raddr_c + 1'b1;
        end else begin
            window_column_counter <= 10'd0;
            case (window_line_counter)
                2'b00 :
                begin
                    raddr_a <= raddr_a + 1'b1;
                    raddr_b <= raddr_b - window_column_counter;
                    raddr_c <= raddr_c - window_column_counter;
                    window_line_counter = window_line_counter + 1'b1;
                end
                2'b01 :
                begin
                    raddr_b <= raddr_b + 1'b1;
                    raddr_a <= raddr_a - window_column_counter;
                    raddr_c <= raddr_c - window_column_counter;
                    window_line_counter = window_line_counter + 1'b1;
                end
                2'b10 :
```

Notice that `window_column_counter` performs some arithmetic operations inside multiple condition statements. However, the condition test is constant and can be computed once. We can also pre-compute the `window_column_counter + 1` on each clock. This technique avoids arithmetic operations inside the critical chain loop, and simply selects the result within the if-then-else statement. This strategy results in a more efficient implementation.

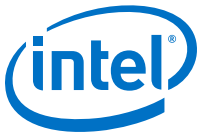
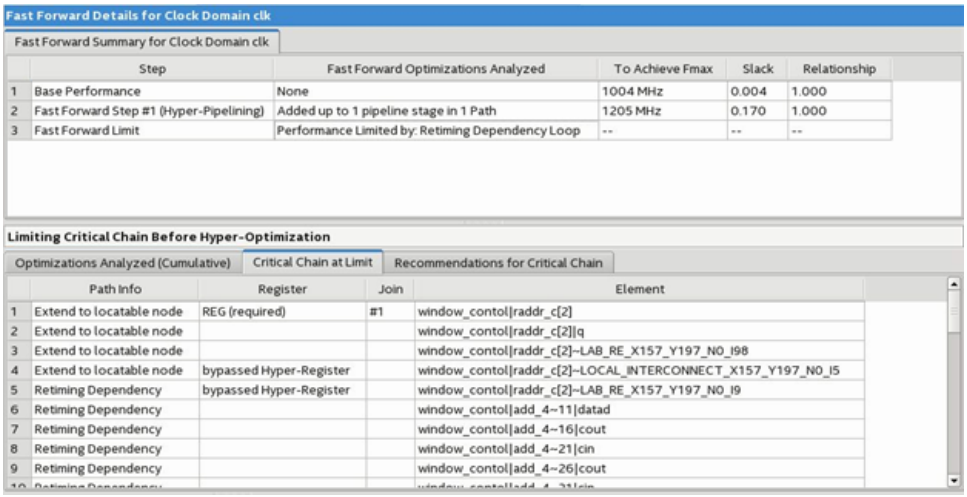


Figure 120. Optimized RTL

```
reg [9:0] window_column_counter_plus_one;
reg rst_n_reg;
wire window_column_counter_test;
assign window_column_counter_test = ((IMG_WIDTH/4)-1) ? 1 : 0;


always @(posedge clk)
begin : out_memory_counter
    if(~rst_n_reg) begin
        waddr <= {LUT_ADDR_WIDTH{1'b0}};
    end else if(valid) begin
        waddr <= waddr + 1'b1;
    end
end

always @(posedge clk)
begin
  rst_n_reg <= rst_n;
  window_column_counter_plus_one <= window_column_counter + 1'b1;
end


always @(posedge clk)
begin : addr_counter
    if(~rst_n_reg) begin
        window_column_counter <= 10'd0;
        window_line_counter <= 2'b00;
        raddr_a <= {LUT_ADDR_WIDTH{1'b0}};
        raddr_b <= {LUT_ADDR_WIDTH{1'b0}};
        raddr_c <= {LUT_ADDR_WIDTH{1'b0}};
    end else begin
        if(!window_column_counter_test) begin
            window_column_counter <= window_column_counter_plus_one;
            valid <= 1'b1;
            raddr_a <= raddr_a + 1'b1;
            raddr_b <= raddr_b + 1'b1;
            raddr_c <= raddr_c + 1'b1;
        end else begin
            window_column_counter <= 10'd0;
            case (window_line_counter)
                2'b00 :
                begin
```

After making RTL changes, click **Processing ➤ Start Compilation** to compile the design. Correcting the asynchronous reset conditions, adding pipeline stages, and avoiding large loops significantly improves this design example performance, from 500MHz to 1GHz.

Figure 121. Fast Forward Details Report



Fast Forward Details for Clock Domain clk

Fast Forward Summary for Clock Domain clk

	Step	Fast Forward Optimizations Analyzed	To Achieve Fmax	Slack	Relationship
1	Base Performance	None	1004 MHz	0.004	1.000
2	Fast Forward Step #1 (Hyper-Pipelining)	Added up to 1 pipeline stage in 1 Path	1205 MHz	0.170	1.000
3	Fast Forward Limit	Performance Limited by: Retiming Dependency Loop	--	--	--

Limiting Critical Chain Before Hyper-Optimization

Optimizations Analyzed (Cumulative) | Critical Chain at Limit | Recommendations for Critical Chain

	Path Info	Register	Join	Element
1	Extend to locatable node	REG (required)	#1	window_contol|raddr_c[2]
2	Extend to locatable node			window_contol|raddr_c[2]|q
3	Extend to locatable node			window_contol|raddr_c[2]~LAB_RE_X157_Y197_N0_I98
4	Extend to locatable node	bypassed Hyper-Register		window_contol|raddr_c[2]~LOCAL_INTERCONNECT_X157_Y197_N0_I5
5	Retiming Dependency	bypassed Hyper-Register		window_contol|raddr_c[2]~LAB_RE_X157_Y197_N0_I9
6	Retiming Dependency			window_contol|add_4~11|datad
7	Retiming Dependency			window_contol|add_4~16|cout
8	Retiming Dependency			window_contol|add_4~21|cin
9	Retiming Dependency			window_contol|add_4~26|cout

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 basic example uses a modulus operator as part of the logic to determine the next client for service. The modulus operator can be relatively slow and area inefficient because it performs division.

Example 15. Source Code for Round Robin Scheduler

The module is instantiated in a register ring and compiled.

```
module round_robin_modulo # (
            parameter LOG2_CLIENTS  = 3,
            parameter CLIENTS       = 7)
  {
    // previous client to be serviced
       input wirte [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 122. 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 #1	10, including 1 pipeline stage	468.82 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

The critical chain for the base performance, without Fast Forward optimizations, identifies as the performance limiting critical chain because of insufficient registers. The chain is from the register that connects to the last input, through the modulus operator implemented with an LPM_DIVIDE IP core, to the register connected to the next output.

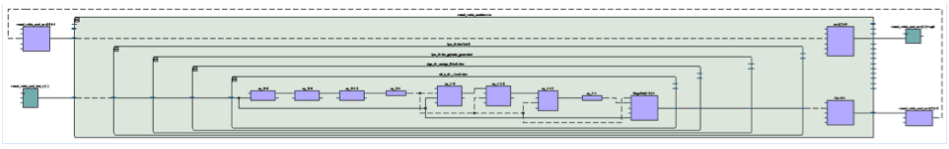
Figure 123. Critical Chain for Base Performance for Round Robin Scheduler

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]~la_lab/laboutb[4]
6	Long Path			round_robin_mod_last_r[1]~LAB_RE_X77_Y198_N0_I114
7	Long Path	empty slot		round_robin_mod_last_r[1]~R3_X75_Y198_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		rrm|Mod3|auto_generated|divider|divider|op_3~5|dataf
12	Long Path			rrm|Mod3|auto_generated|divider|divider|op_3~9|cout
13	Long Path			rrm|Mod3|auto_generated|divider|divider|op_3~13|cin
14	Long Path			rrm|Mod3|auto_generated|divider|divider|op_3~1|sumout
15	Long Path			round_robin_modulo:rrm|lpm_divide:Mod3|lpm_d...t_u_div_vke:divider|op_3~1la_mlab/laboutt[7]
16	Long Path			round_robin_modulo:rrm|lpm_divide:Mod3|lpm..._vke:divider|op_3~1_LAB_RE_X76_Y199_N0_I97
17	Long Path	empty slot		round_robin_modulo:rrm|lpm_divide:Mod3|l...p_3~1_LOCAL_INTERCONNECT_X76_Y199_N0_I26
18	Long Path			round_robin_modulo:rrm|lpm_divide:Mod3|lpm..._vke:divider|op_3~1_LAB_RE_X76_Y199_N0_I43
54	Long Path	empty slot		round_robin_mod_next[0]~4_C4_X74_Y199_N0_I11
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		rrm|next[2]~5|datac
59	Long Path			rrm|next[2]~5|combout
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]~reg0la_mlab/laboutb[8]
64	Long Path			round_robin_mod_next[2]~reg0_LAB_RE_X76_Y200_N0_I118
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 with 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 124. Schematic for Critical Chain



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

Figure 125. 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
1		REG (required)	#1	round_robin_mod_requests_r[2]
2	Short Path			round_robin_mod_requests_r[2]|q
3	Short Path	unusable (hold)		rrm|Mux4~0|datae
4				rrm|Mux4~0|combout
5				round_robin_modulo:rrm|Mux4~0la_mlab/laboutt[13]
6				round_robin_modulo:rrm|Mux4~0_LAB_RE_X76_Y198_N0_I103
7		REG	#2	round_robin_modulo:rrm|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)		rrm|Mux4~0|datae
12	Long Path			rrm|Mux4~0|combout
13	Long Path			round_robin_modulo:rrm|Mux4~0la_mlab/laboutt[13]
14	Long Path			round_robin_modulo:rrm|Mux4~0_LAB_RE_X76_Y198_N0_I103
15	Long Path	REG		round_robin_modulo:rrm|Mux4~0_LOCAL_INTERCONNECT_X75_Y198_N0_I3_dff
16	Long Path			round_robin_modulo:rrm|Mux4~0_LOCAL_INTERCONNECT_X75_Y198_N0_I3
33	Long Path	empty slot		rrm|next[1]~3|datac
34	Long Path			rrm|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_RE_X75_Y198_N0_I19
41	Short Path	REG		round_robin_modulo:rrm|next[1]~3_datab_dff
42	Short Path			rrm|next[1]~3|datab
43	Short Path			rrm|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_RE_X75_Y198_N0_I3
50	Long Path			rrm|Mod4|auto_generated|divider|divider|op_3~9|datab
68	Long Path	empty slot		rrm|Mod4|auto_generated|divider|divider|StageOut[12]~0|datab
69	Long Path			rrm|Mod4|auto_generated|divider|divider|StageOut[12]~0|combout
70	Long Path			round_robin_modulo:rrm|lpm_divide:Mod4|lpm_d...vke:divider|StageOut[12]~0la_lab/laboutb[12]
71	Long Path			round_robin_modulo:rrm|lpm_divide:Mod4|lp...er|StageOut[12]~0_LAB_RE_X75_Y198_N0_I122
72	Long Path	empty slot		round_robin_modulo:rrm|lpm_divide:Mod4|l...12]~0_LOCAL_INTERCONNECT_X76_Y198_N0_I58
73	Long Path			round_robin_modulo:rrm|lpm_divide:Mod4|lpm...ider|StageOut[12]~0_LAB_RE_X76_Y198_N0_I28
74	Long Path	empty slot		rrm|Mux4~0|datac
75	Long Path			rrm|Mux4~0|combout
76	Long Path			round_robin_modulo:rrm|Mux4~0la_mlab/laboutt[13]
77	Long Path			round_robin_modulo:rrm|Mux4~0_LAB_RE_X76_Y198_N0_I103
78	Long Path	REG	#2	round_robin_modulo:rrm|Mux4~0_LOCAL_INTERCONNECT_X75_Y198_N0_I3_dff

The performance can improve with additional manual changes to the source code. The divider in the modulus operation is the bottleneck for focus. Paths through the divider exist in the critical chain for all steps in the Fast Forward compile.

A better approach is to consider alternate implementations to calculate the next client to service, to avoid the modulus operator. If you switch to an implementation that specifies the number of clients as a power of two, a modulus operator is not required to determine the next client to service. When you instantiate the module with fewer than 2^n clients, tie the unused request inputs to logic 0.

Example 16. Source Code for Round Robin Scheduler with Improved Performance 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
                  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 Compile result from the version with the modulus operator.

Figure 126. Fast Forward Compile Report for Round Robin Scheduler with Improved Performance 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	4.011	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.

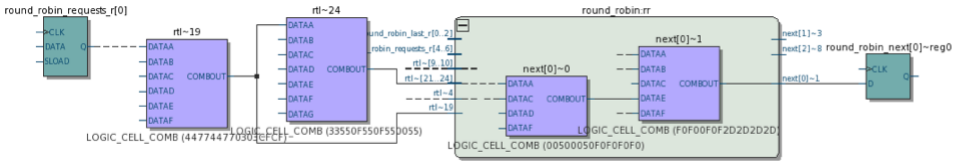
Figure 127. Critical Chain for Round Robin Scheduler with Improved Performance

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]~la_lab/laboutt[18]
6	Long Path			round_robin_requests_r[0]~LAB_RE_X77_Y198_N0_I108
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		rtl~19|dataa
28	Long Path	empty slot		round_robin:rr|next[0]~0_LOCAL_INTERCONNECT_X77_Y200_N0_I55
29	Long Path			round_robin:rr|next[0]~0_LAB_RE_X77_Y200_N0_I1
30	Long Path	empty slot		rr|next[0]~1|datae
31	Long Path			rr|next[0]~1|combout
32	Long Path			round_robin_next[0]~reg0|d
33	Long Path	REG (required)		round_robin_next[0]~reg0
34	Long Path			round_robin_next[0]~reg0|q
35	Long Path			round_robin_next[0]~reg0la_lab/laboutt[1]
36	Long Path			round_robin_next[0]~reg0_LAB_RE_X77_Y200_N0_I91
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 128. Schematic for Critical Chain with Improved Performance



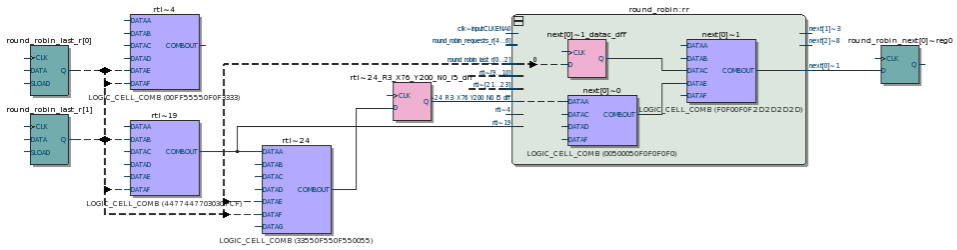
By adding two register stages at the input, to be retimed through the logic cloud, Fast Forward Compile takes the circuit performance to 1 GHz, which is the architectural limit of Stratix 10 devices. As with the modulus version, the final critical chain after Fast Forward Optimization has a limiting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

Figure 129. Critical Chain for Round Robin Scheduler with Best Performance

Fast Forward Optimizations Applied in Clock Domain clk | Critical Chain at Limit | Recommendations for Critical (

	Path Info	Register	Join	Element
1		empty slot	#1	rtl~4|dataf
2			#2	rtl~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)		rtl~4|datae
7			#2	rtl~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]~la_mlab/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
35	Long Path			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_I122
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_I122
51		empty slot		round_robin_last_r[0]~LOCAL_INTERCONNECT_X76_Y200_N0_I21
52				round_robin_last_r[0]~LAB_RE_X76_Y200_N0_I72
53		empty slot	#1	rtl~4|dataf

Figure 130. Schematic for Critical Chain with Best Performance



Removing the modulus operator and switching to a power-of-two implementation is a very small design change that provides 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 provided more performance increase than adding pipeline stages.

7 Glossary

Table 7. Glossary

Term/Phrase	Description
Critical chain	A critical chain is any design condition that prevents Hyper-Retiming from improving performance. In Hyper-Retiming, the limiting factor may include more than one register-to-register path in a chain. A critical chain is a higher level abstraction of a critical path. You can analyze and report critical paths in HyperFlex designs with the TimeQuest timing analyzer.
Fast-corner timing analysis	The TimeQuest timing analyzer performs multicorner timing analysis to verify your design under a variety of operating conditions—such as voltage, process, and temperature. Fast-corner analysis assumes best-case timing conditions.
Fast-Forward Compilation	Analyzes potential performance of HyperFlex architecture by virtually moving registers (Hyper-Retiming) and adding pipeline registers (Hyper-Pipelining). Fast-Forward Compilation identifies design bottlenecks for Hyper-Optimization and identifies methods for further optimization.
Hyper-Aware design tools	Intel design methodologies and tools that enable the Stratix 10 HyperFlex architecture.
HyperFlex architecture	Stratix 10 device core architecture that includes additional registers, called Hyper-Registers, everywhere throughout the core fabric. These registers are available in every interconnect routing segment and at the inputs of all functional blocks. The Hyper-Registers provide increased bandwidth and improved area and power efficiency. With many more registers that are easy to access, you can retime registers to eliminate critical paths, add pipeline registers to remove routing delays, and optimize your design for best-in-class performance.
Hyper-Optimization	The process of analyzing and improving design performance by making changes to the design that are enabled by retiming.
Hyper-Retiming	Optimizes the placement of existing registers to balance the propagation delay between the registers. Also performs sequential optimizations by moving registers back and forward across combinational logic
Performance recommendations	Fast Forward Compilation recommends design changes based on the potential implementation of Hyper-Retiming and Hyper-Pipelining in your design.
Retiming reports	Reports the f_{MAX} achievable with Hyper-Retiming and lists recommendations to optimize the design.

8 Document Revision History

This document has the following revision history.

Table 8. Document Revision History

Date	Version	Changes
2016.08.07	2016.08.07	• 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 • Revised software name to Quartus Prime Pro – Stratix 10 Edition Beta. • Updated for latest changes to software GUI and capabilities.
2016.03.16	2016.03.16	First public release.

A Appendix: Clock Enables and Resets

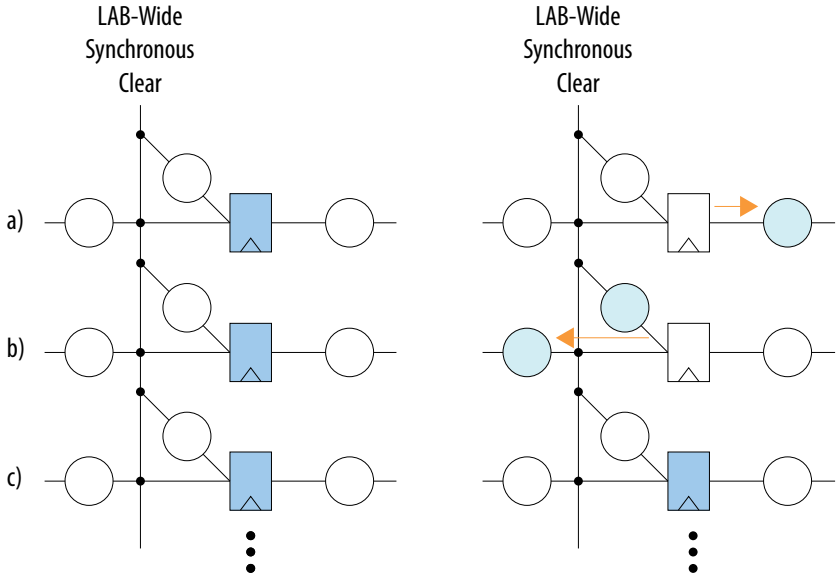
A.1 Synchronous Resets and Limitations

Converting asynchronous resets to synchronous helps retiming, but there are still performance restrictions. The ALM has a dedicated LAB-wide signal which is often used for synchronous clears. Using the signal is determined by synthesis, but is usually dependent on the signal's fan-out. A synchronous clear with a small fan-out is usually done in logic, while larger fan-outs use this dedicated signal. Even if the dedicated synchronous clear is used, the register can still be pushed into Hyper-Registers. This process is achieved through the bypass mode of the ALM register, where a signal can go right up to the register and still bypass it. When the register is bypassed, the `sclr` signal and other control signals can still be accessed.

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 131. 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 one is occupied.

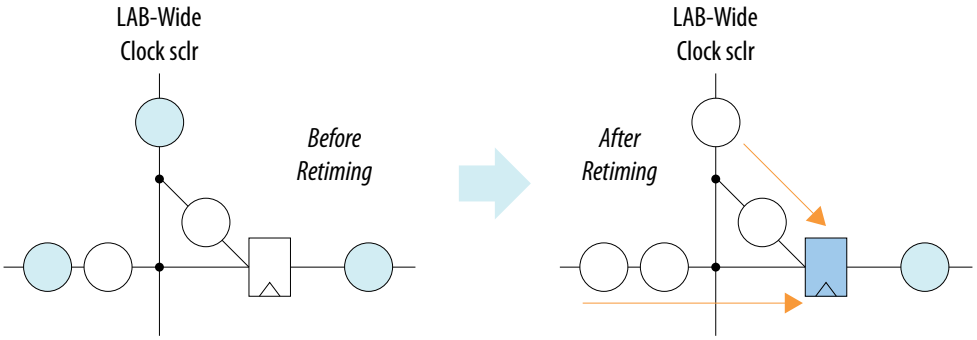


During retiming, the top register in row (a) is pushed right into a Hyper-Register. This is achieved by bypassing the ALM register, but still using the SCLR logic that feeds that register. When the LAB-wide SCLR signal is used, an ALM register must exist on the data path, but it does not have to be used.

The retimer pushes the register in row (b) left into its data path. The register is pushed through a signal split of the data path and synchronous clear. So this register must be pushed onto both nets, one in the data path and one in the synchronous clear path. This can be implemented because each path has a Hyper-Register.

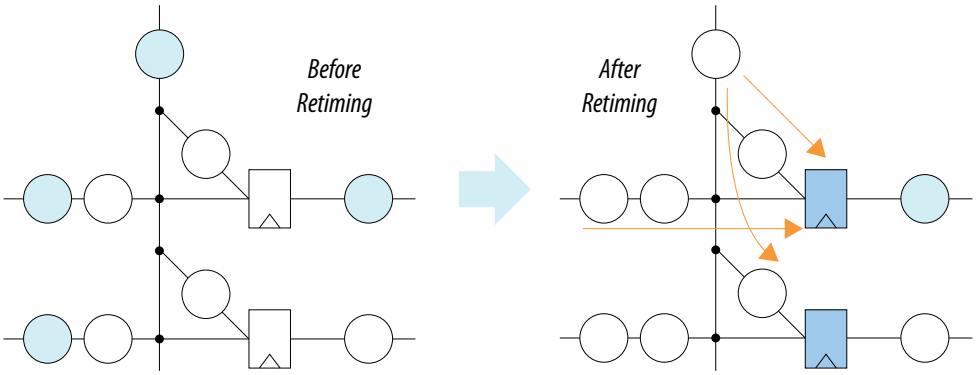
Retiming becomes complicated if another register is pushed forward into the ALM. As shown in the following figure, a register from the asynchronous clear port and a register from the data path must be merged together.

Figure 132. Retiming Example – Second Register Pushed out of ALM



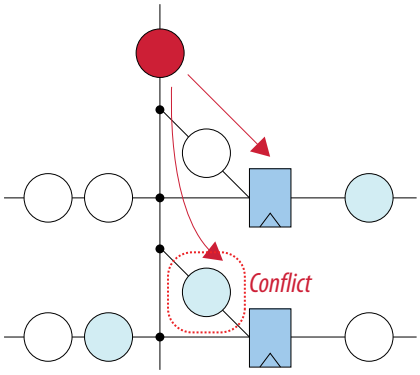
Because the register on the synchronous clear path is shared with other registers, the register splits on the path to other synchronous clear ports as well.

Figure 133. Retiming Example – Register Splits on the Path to other Synchronous Clear Ports



In the following figure, the Hyper-Register at a synchronous clear is already being used and cannot accept another register. In this case, you cannot retime this register for the second time through the ALM.

Figure 134. Retiming Example – Conflict at Synchronous Clear



There are two key architectural components that make it easy to move an ALM register with a synchronous clear forward or backward:

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

If you want to push more registers through, retiming becomes difficult. Because of this, performance improvement is expected to be better with asynchronous reset removal than conversion to synchronous resets. Synchronous clears are often difficult to retime because of their wide broadcast nature.

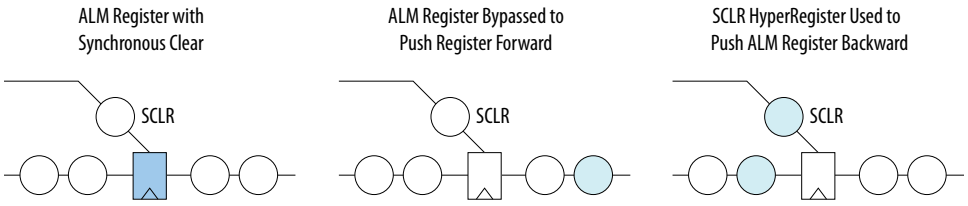
A.1.1 Synchronous Resets Summary

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

- A short path, usually going directly from the source register to the destination register without any logic between them. This is not a problem by itself. Short paths are normally good, because their positive slack can be retimed out to longer paths, making the whole design run faster. But short paths are typically connected to long data paths that must be retimed. By retiming lots of registers up and down these long paths, registers are getting pushed down or pulled up this short path. This issue isn't a big problem in normal logic, but is aggravated because synchronous clears typically have large fan-outs.
- Synchronous clears have large fan-outs. When an aggressive retiming requires registers to be pushed up or down the synchronous clear paths, the paths can get cluttered until they can no longer accept more registers. This situation results in path length imbalances (also referred to as short path / long path), or no more registers can be pulled from the synchronous clear paths.

Aggressive retiming is when a second register must be retimed through the ALM register.

Figure 135. Aggressive Retiming



Consider an ALM register that has a synchronous clear signal, as shown in the picture on the left. The middle picture shows that register has been retimed forward and the ALM register is bypassed. The picture on the right shows the register being retimed backwards, in which case a register must be pushed up the SCLR path. Because the HyperFlex hardware has these special features, a dedicated Hyper-Register on the SCLR path, and the ability to put the ALM register into bypass mode, you can push and pull this register. If pushed forward, then you must pull a register down the SCLR path and merge the two. If pushed back, then you must push a duplicate register up the SCLR path. You can use both of these options. However, bottlenecks can be created when multiple registers are pushing and pulling registers up and down the synchronous clear routing.

In summary, be practical about where to use resets. Control logic mostly requires synchronous reset. Logic that may not require a synchronous reset helps with timing. Refer to the following guidelines for dealing with synchronous resets:

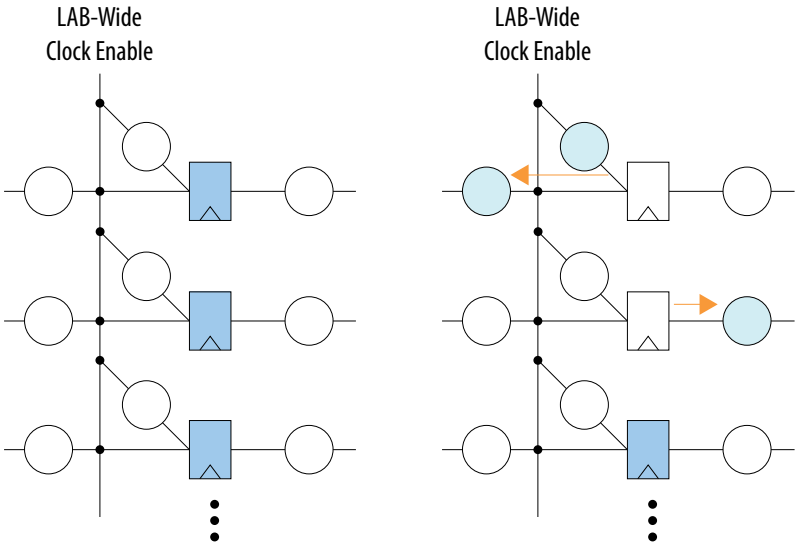
- When writing new code that needs to run at high speeds, avoid synchronous resets wherever possible. This is generally in data path logic that either flushes out while the system is in reset, or its values are ignored when the system comes out of reset, until new, valid logic filters through.
- Control logic often requires a synchronous reset, so there is no avoiding it in that situation.
- For existing logic that runs at high speeds, remove the resets wherever possible. When you reach a point where you do not understand the logic well enough or aren't confident with how it behaves when reset, leave the synchronous reset in. Only if it becomes a timing issue in your design should you spend time analyzing if and how the synchronous clear can be removed.
- Pipeline the synchronous clear. This does not help if registers must be pushed back, but can help when registers must be pulled forward into the data path.
- Duplicate synchronous clear logic for different hierarchies. This limits the fan-out of the synchronous clear so that it can be retimed with the local logic. Again, this may be done only after you determine the existing synchronous clear with a large fan-out is limiting how the design can be retimed. This is not difficult to do on the back-end because it does not change the design functionality.
- Duplicate synchronous clear for different clock domain and inverted clocks. This can overcome some retiming restrictions due to boundary or multiple period requirement issues.

A.2 Retiming with Clock Enables

Like synchronous resets, clock enables use a dedicated LAB-wide resource that feed a specific function in the ALM register. Similarly, the HyperFlex architecture has some special logic that makes retiming logic with clock enables easier. However, wide broadcast control signals such as clock enables (and synchronous clears) are difficult to retime.

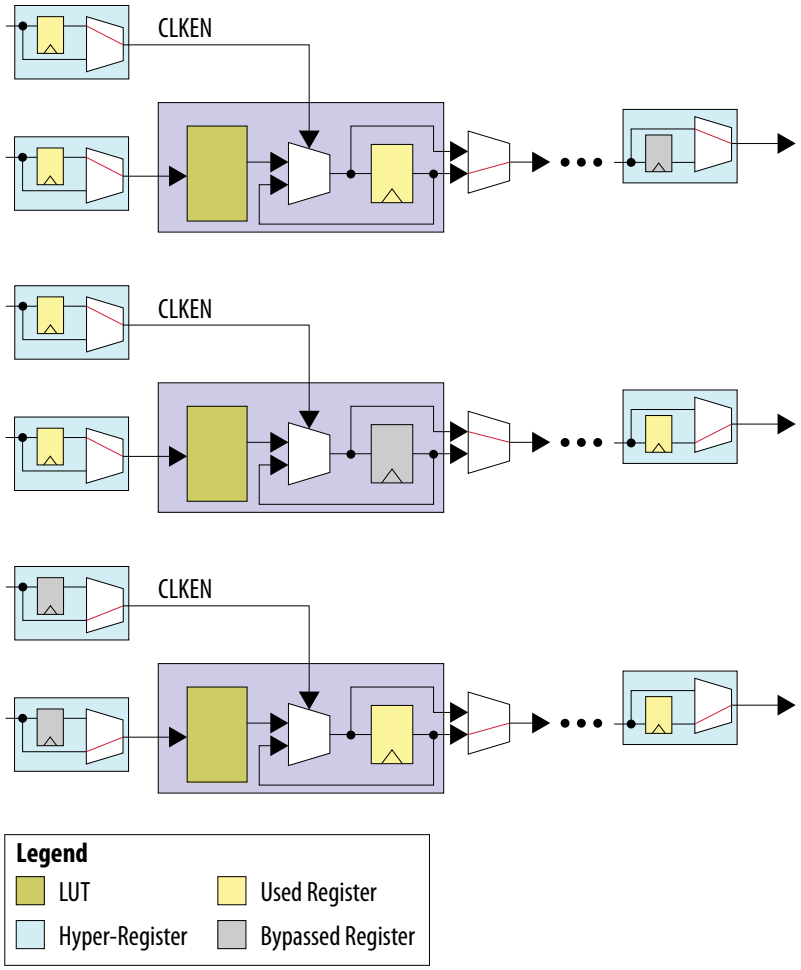
The following figure shows that the sequence of retiming moves for the asynchronous clears in the *Synchronous Resets and Limitations* section apply to the clock enable control signals.

Figure 136. ALM Representing Clock Enables



In the top circuit, there is a dedicated Hyper-Register on the clock enable path. If the register needs to be pushed back, it must be split so that another register is pushed up the clock enable path. Here, the Hyper-Register location can absorb it without problem. These features allow an ALM register with a clock enable to be easily retimed backward or forward (middle circuit), to improve timing. A useful feature of a clock enable is that its logic is usually generated by synchronous signals, so that the clock enable path can be retimed alongside the data path.

Figure 137. Retiming Steps and Structure with an ALM register and Associated Hyper-Registers



The figure shows how the clock enable signal `clken`, which is a typical broadcast type of control signal, gets retimed. In the top circuit, before retiming, an ALM register is used. The Hyper-Registers on the clock enable and data paths are also used. In the middle circuit, the ALM register has been retimed forward into a Hyper-Register outside the ALM, into the routing fabric. The ALM register is still being used, but it is not on the data path through the ALM. It is used to hold 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 bottom diagram shows when a second register is retimed forward from the clock enable and data paths into the ALM register. The ALM register is now used in the path. This process can be repeated and multiple registers can be iteratively retimed across an enabled ALM register.

The clock enable structure can be divided into the following three categories.

Related Links

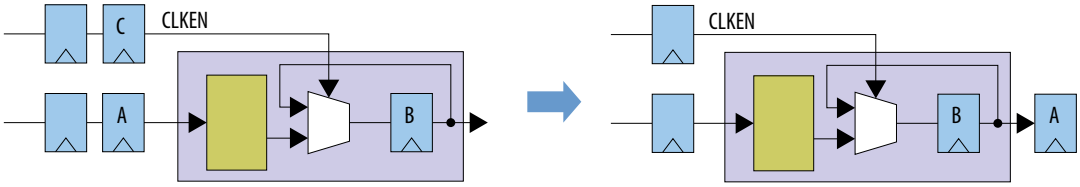
Synchronous Resets and Limitations on page 119

A.2.1 Example for Broadcast Control Signals

Broadcast control signals that fan out to large numbers of destinations can be a limiting factor to retiming. Asynchronous clears can limit retiming due to silicon support for certain register control signals. However, even synchronous signals, such as synchronous clear and clock enable, can limit retiming when they are part of a short path/long path critical chain. The use of a synchronous control signal is not a limiting reason by itself; rather it is the structure of the circuit combined with the particular placement.

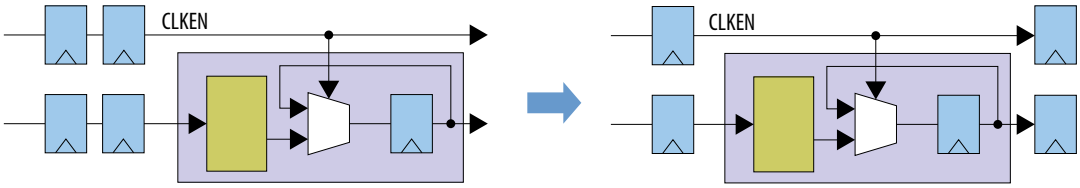
To forward retime a register over a node, there must be a register available on all of the node's inputs. This requirement is the same for conventional retiming and Hyper-Retiming. To retime register A over register B in the following diagram, a register must be pulled from all inputs, including register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

Figure 138. 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 that is shown. To retime register A over register B, the operation is the same as the previous diagram, but the presence of the branch point means that a copy of register C must be retimed along the other side of the branch point, to register C.

Figure 139. Retiming through a Clock Enable with a Branch Point

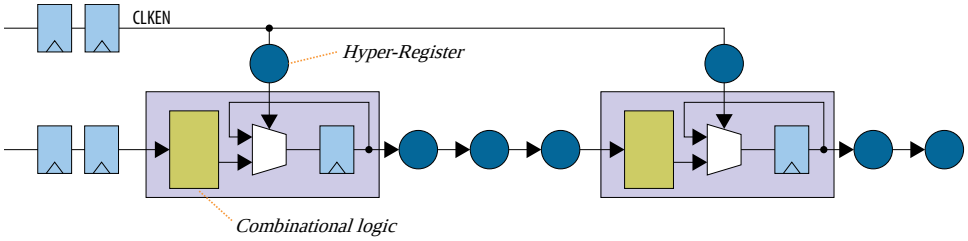


Retiming Example

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 140. Retiming Example Starting Point

The Hyper-Retimer can move a retimed register into the Hyper-Registers.



Per the HyperFlex architecture, 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 could include more or fewer Hyper-Register locations. Additionally, there are registers on the data and clock enable inputs to this chain that can be retimed by the Hyper-Retimer. These registers exist in the RTL, or can be specified with options described in *Pipeline Stages* section.

One stage of the input registers are retimed into a Hyper-Register location between the two registers. Figure 141 on page 126 shows one part of the Hyper-Retiming forward push. One of the registers on the clock enable input is retimed over the branch point, with a copy going to a Hyper-Register location at each clock enable input.

Figure 141. Retiming Example Intermediate Point

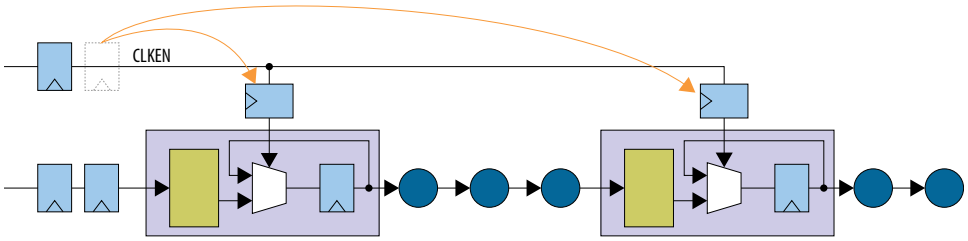


Figure 142 on page 126 shows the positions of the registers in the circuit after the Hyper-Retimer completes the forward push. The two registers at the inputs of the left register have been retimed to a Hyper-register location. This diagram is functionally equivalent to the two previous diagrams. The 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 could be retimed.

Figure 142. Retiming Example Ending Point

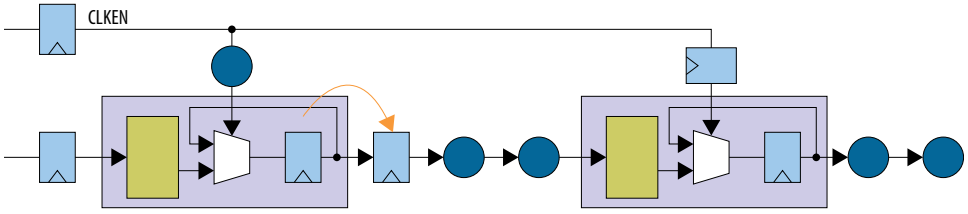
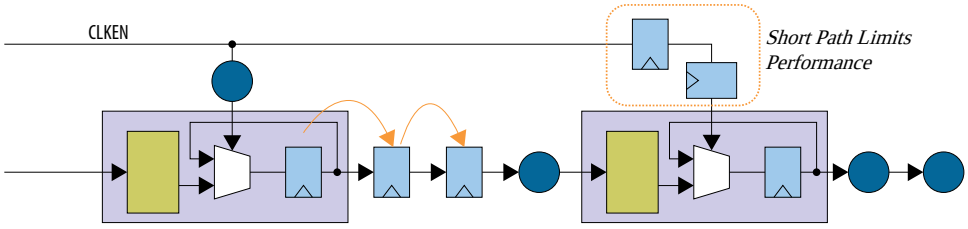


Figure 143 on page 127 shows the register positions the Hyper-Retimer could use if it were not limited by a short path/long path critical chain. However, because no Hyper-Registers are available on the right-hand clock enable path, the Hyper-Retimer cannot retime the circuit as shown in the diagram.

Figure 143. Retiming Example Limiting condition



Because the clock enable path to the second register has no more Hyper-Register locations available, it is be reported as the short path. Because the register-to-register path is too long to operate above the reported performance, although having more available Hyper-Register locations for the retimed registers, the path is reported 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, and you should take steps to avoid or rewrite the structures.

Related Links

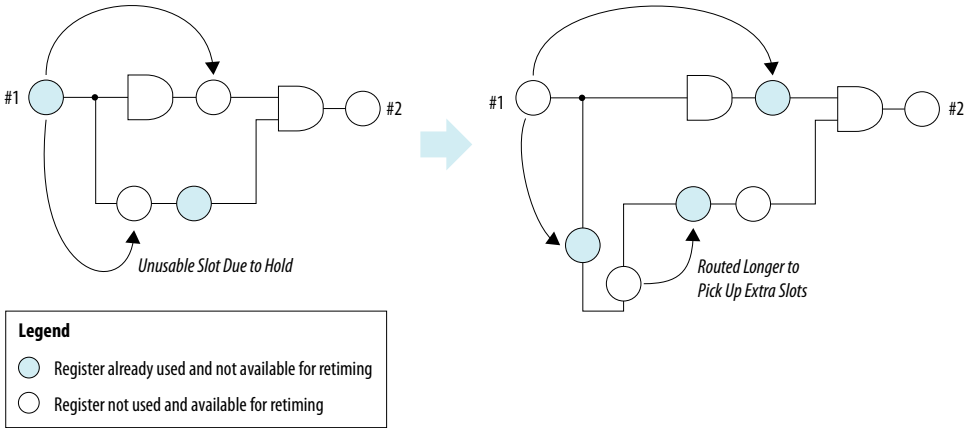
Appendix: Parameterizable Pipeline Modules on page 50

A.3 Resolving Short Paths

Traditionally, retiming registers that are close to each other can potentially trigger hold violations at higher speeds.

The following figure shows how a short path can typically limit 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.

Figure 144. Short Paths Limiting Retiming



In the circuit on the left, if register #1 is to be retimed forward, the top path has an available slot. However, the lower path can't accept a retimed register because it is too close to an adjacent register already in use, causing hold time violations. The place and route tool is aware of these short paths, so it can route the registers to longer paths, as shown in the circuit on the right. This practice ensures that sufficient slots are available for retiming. This feature interacts with the Fast Forward Compile feature.

The following two examples address short paths:

Case 1: A design works at 400 MHz and the Fast Forward Compile report 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 recommended two-stage pipelining to reach 600 MHz performance. Then, if the limiting reason is again short path / long path, the router has reached a limitation in trying to fix the short paths in the design. However, at this point you may have already reached your target performance, or this is no longer the critical path.

Case 2: A design works at 400 MHz and the Fast Forward Compile report does not make any recommendations to add pipeline stages.

If the short path / long path is the immediate limiting reason for retiming, the router has reached 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.