Intel® Stratix® 10 High-Performance Design Handbook

S10HPHB
2017.05.08

Subscribe
Send Feedback

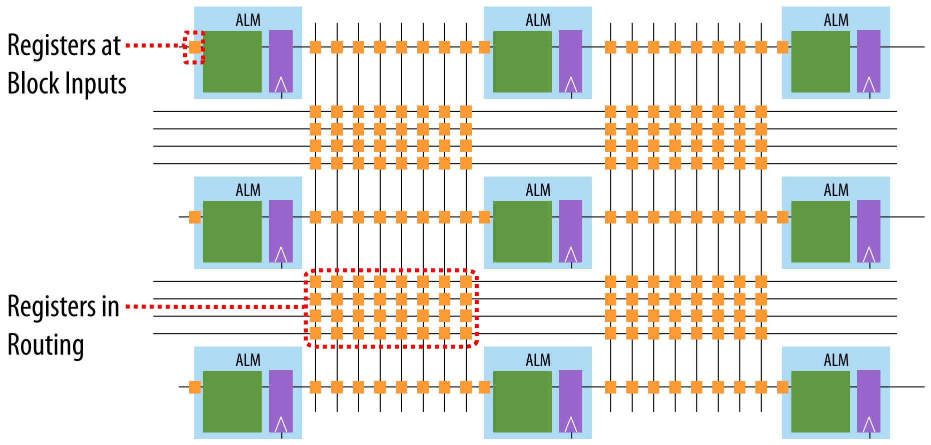
Contents

1 Intel® HyperFlex FPGA Architecture Introduction

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

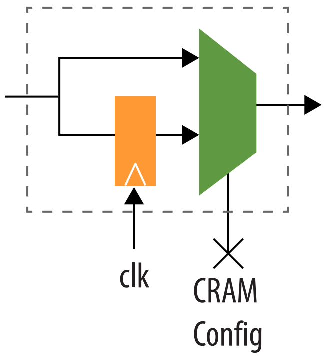
"Registers everywhere" is a key innovation of the HyperFlex FPGA architecture. Stratix 10 devices pack bypassable Hyper-Registers into every routing segment in the device core, and at all functional block inputs.

Figure 1. Registers Everywhere



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

Figure 2. Bypassable Hyper-Registers



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

- RTL Design Guidelines—provides fundamental high-performance RTL design techniques for Stratix 10 designs.
- Compiling Stratix 10 Designs—describes using the Intel Quartus® Prime Pro v17.1 Stratix 10 ES Editions to get the highest performance in Stratix 10 devices.
- HyperFlex Porting Guidelines—provides guidance for design migration to Stratix 10 devices.
- Design Example Walk-Through, Optimization Example, Appendices —demonstrate performance improvement techniques using real design examples.

1.1 Stratix 10 Basic Design Concepts

Table 1. Glossary

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

Related Links

- Hyper-Retiming (Facilitate Register Movement) on page 11
- Hyper-Pipelining (Add Pipeline Registers) on page 25
- Hyper-Optimization (Optimize RTL) on page 28

2 RTL Design Guidelines

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

2.1 High-Speed Design Methodology

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

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

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

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.

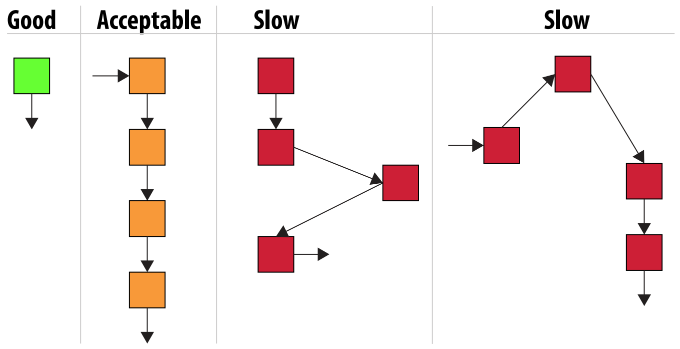
While setting a high-speed target, you must also maintain a comfortable guard band between the speed at which you can close timing, and the actual system speed required. Addressing the timing closure initially with margin is much easier.

2.1.1.1 Speed and Timing Closure

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

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

Figure 3. Acceptable and Slow Routing Delays



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

2.1.1.2 Speed and Latency

Running an FPGA 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

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

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

When working with streaming datapaths, the number of registers is a fair approximation of the latency of the pipeline in bits. Reducing the width by half creates the opportunity to double the number of pipeline stages without negatively impacting latency. 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 with settings and design changes if your design's performance does not initially meet performance requirements. Intel FPGA reprogrammability allows experimentation until you achieve your goals. Commonly, a design performance gradually becomes inadequate as technology requirements increase over time. For example, if you apply an existing design element to a new context at a wider parameterization, the speed performance likely declines.

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

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

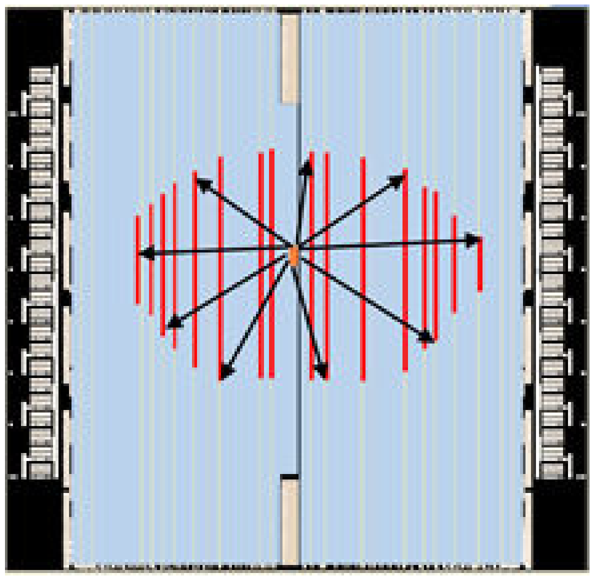
2.1.3 Compile Components Independently

Compile the design subcomponents as stand-alone entities to rapidly identify and optimize performance bottlenecks. Competition for resources and physical constraints (like pin locations) reduces 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. Base your timing margin targets on the context. For example, you can allow a timing margin of 10% for a high-level component representing half the chip. However, if the rule is not explicit, the margin erodes as 10% becomes 9%, then 6%, and so on.

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

Figure 4. M20K Spread in Chip Planner



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

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

2.1.4 Optimize Sub-Modules

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

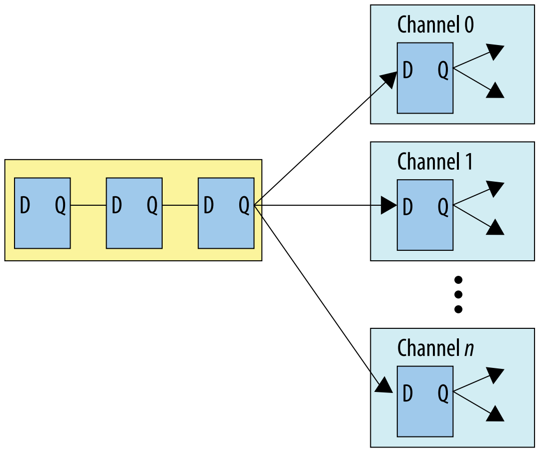
2.1.5 Avoid Broadcast Signals

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

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

Figure 5. Sub-Optimal Pipelining of Broadcast Signals

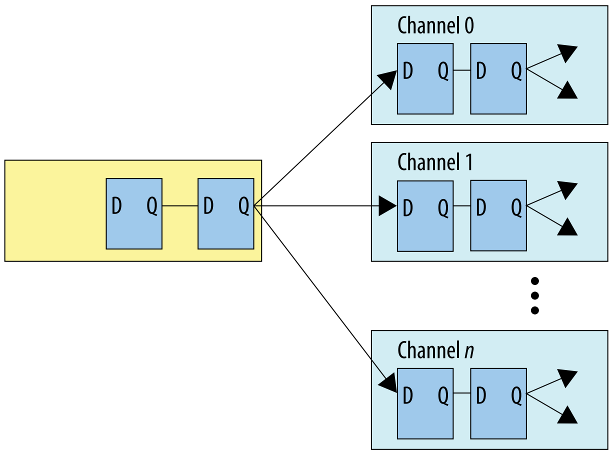
The yellow box highlights inserted registers in a module to help with timing. The block broadcasts the output to several transceiver channels. These extra registers may not improve timing sufficiently because the final register stage fans out to destinations over a wide area of the device.



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

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

Figure 6. Optimal Pipelining of Broadcast Signals



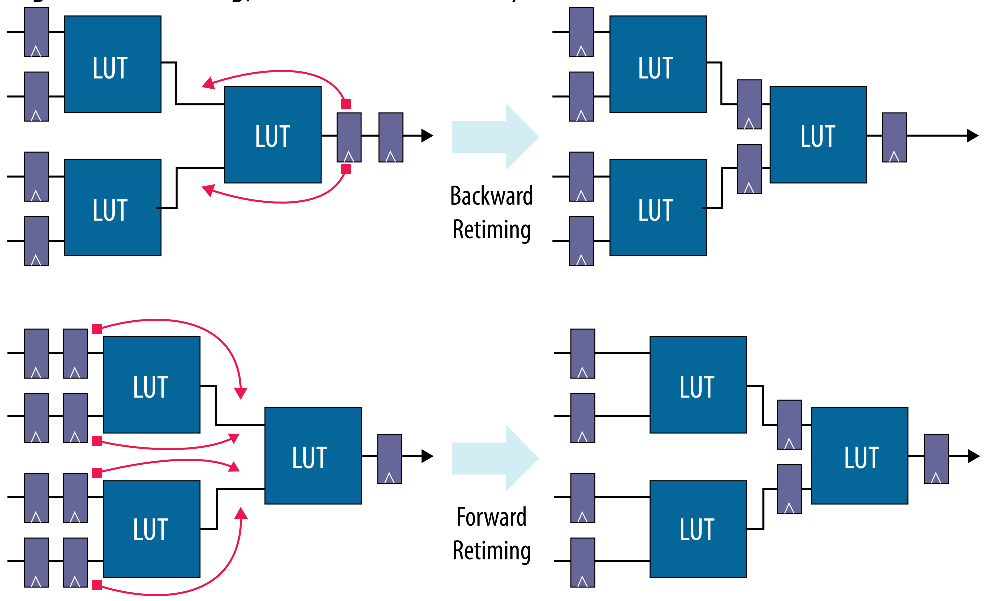
2.2 Hyper-Retiming (Facilitate Register Movement)

The Retime stage of the Fitter can balance register chains by retiming (moving) ALM registers into Hyper-Registers in the routing fabric. The Retime stage also performs sequential optimization by moving registers backwards and forwards across combinatorial logic. 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.

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

Figure 7. Moving Registers across LUTs

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



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

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

- Avoid asynchronous resets, except where necessary. Refer to the *Reset Strategies* section.
- Avoid synchronous clears. Synchronous clears are usually broadcast signals that are not conducive to retiming.
- Use 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 *Hyper-Pipelining (Add Pipeline Registers)* section.
- For information about addressing loops and other RTL restrictions to retiming, refer to the *Hyper-Optimization (Optimize RTL)* section.

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

Related Links

- Reset Strategies on page 12
- Timing Constraint Considerations on page 18
- Hyper-Pipelining (Add Pipeline Registers) on page 25
- Retiming Restrictions and Workarounds on page 88

2.2.1 Reset Strategies

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

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

Related Links

Synchronous Resets and Limitations on page 117

2.2.1.1 Removing Asynchronous Clears

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

Figure 8. Verilog HDL and VHDL Asynchronous Clear Examples

Verilog HDL

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

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

VHDL

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

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

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

Figure 9. Circuit Using Full Asynchronous Reset

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

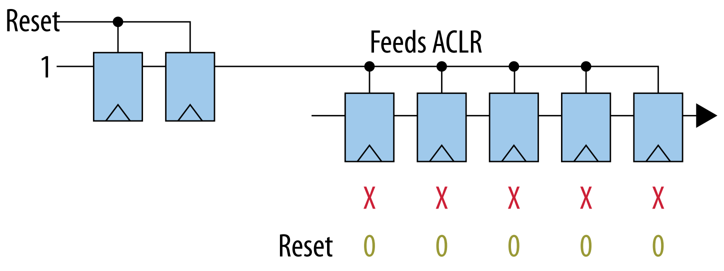


Figure 10. Partial Asynchronous Reset

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

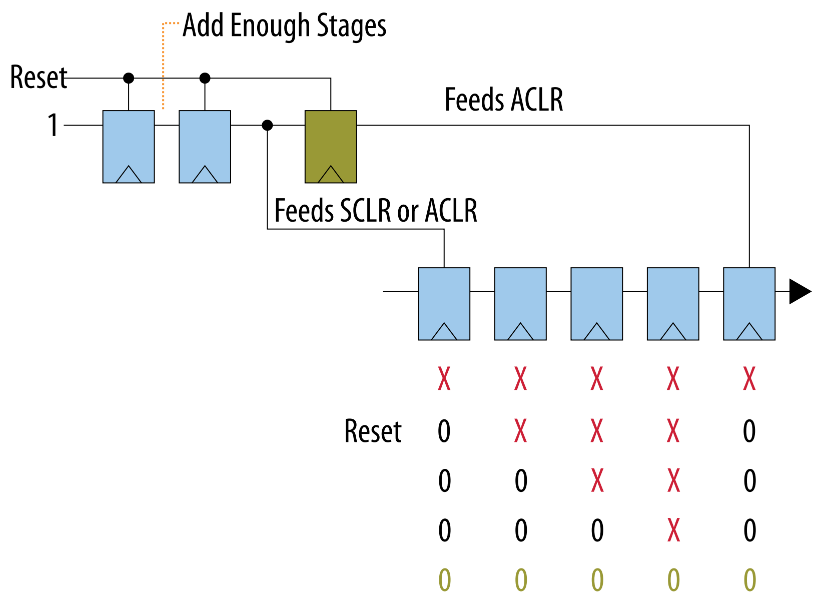


Figure 11. Circuit with an Inverter in the Register Chain

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

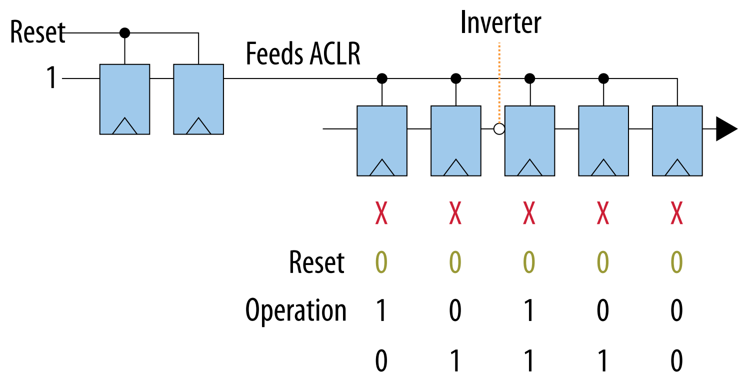


Figure 12. Circuit with an Inverter in the Register Chain with Asynchronous Clear

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

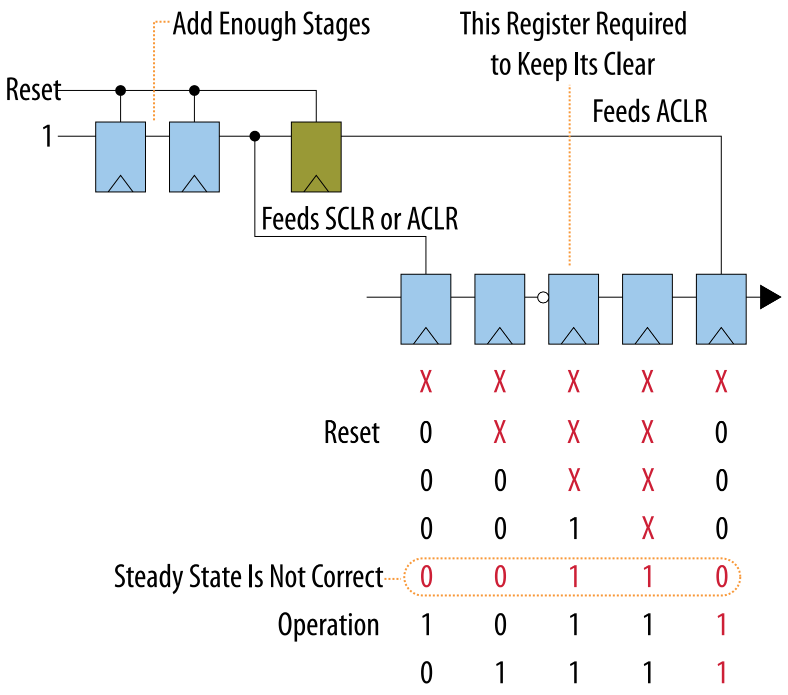
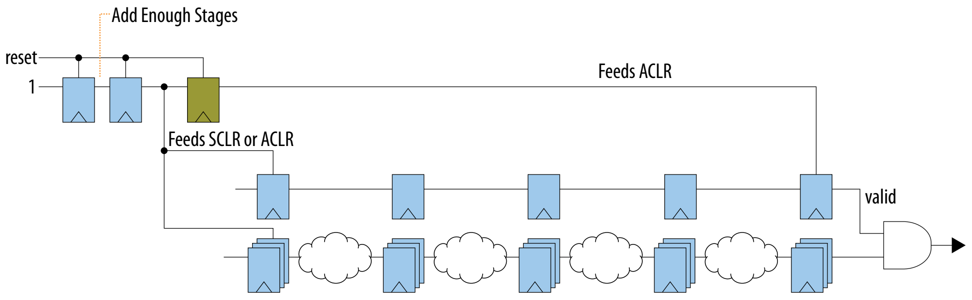


Figure 13. Validating the Output to Synchronize with Reset

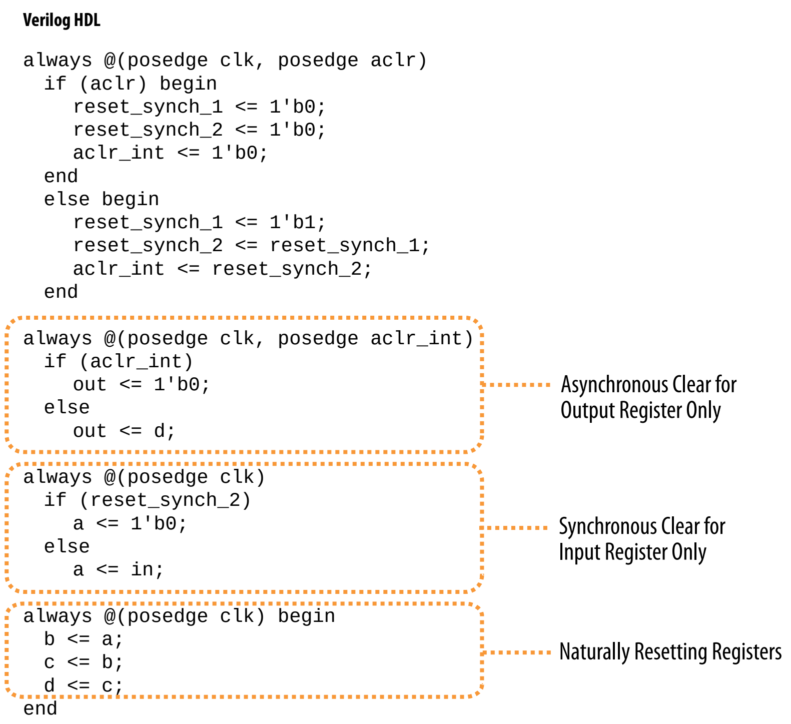
To avoid non-naturally resetting logic caused by inverting functions, validate the output to synchronize with reset removal. Then, if the validating pipeline can enable the output when the computational pipeline is actually valid, the behavior is equivalent with reset removal. This process is suitable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



The following figures show Verilog HDL and VHDL examples of the Figure 10 on page 14. Apply this example to your design to remove unnecessary asynchronous resets

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

The following figures show Verilog HDL and VHDL examples of the Figure 10 on page 14. Apply this example to your design to remove unnecessary asynchronous resets

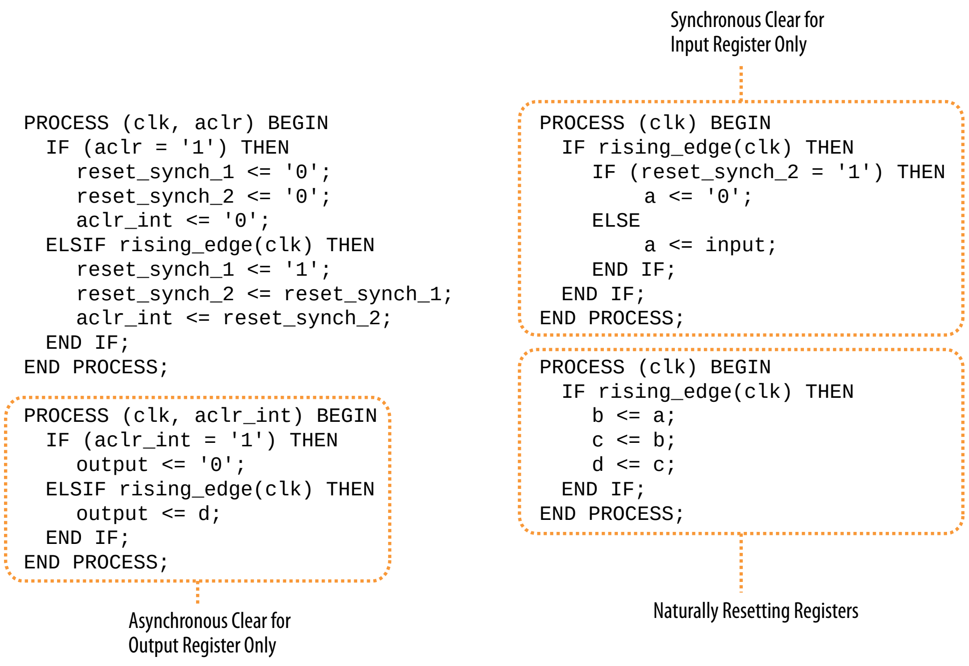


Verilog HDL

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

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

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

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

Asynchronous Clear for Output Register Only
Synchronous Clear for Input Register Only
Naturally Resetting Registers

Figure 15. VHDL Example Using Minimal or No Asynchronous Clears



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

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

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

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

Synchronous Clear for Input Register Only
Asynchronous Clear for Output Register Only
Naturally Resetting Registers

2.2.1.2 Synchronous Clears on Global Clock Trees

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

2.2.1.3 Synchronous Resets on I/O Ports

The Compiler does not retime registers driving an output port or 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 retiming. This section provides recommendations for the appropriate use of clock enables.

2.2.2.1 Localized Clock Enable

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

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

2.2.2.2 High Fan-Out Clock Enable

Avoid 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 120

2.2.2.3 Clock Enable with Timing Exceptions

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

2.2.3 Synthesis Attributes

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

Related Links

Preserve Registers During Synthesis on page 60
  Provides more information about Quartus Prime Synthesis Preserve Options

2.2.4 Timing Constraint Considerations

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

2.2.4.1 Optimize Multicycle Paths

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

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

2.2.4.2 Overconstraints

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

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

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

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

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

2.2.5 Clock Synchronization Strategies

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

Figure 16. Simple Clock Domain Crossing

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

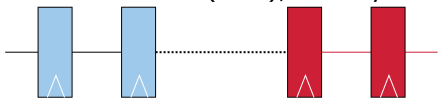
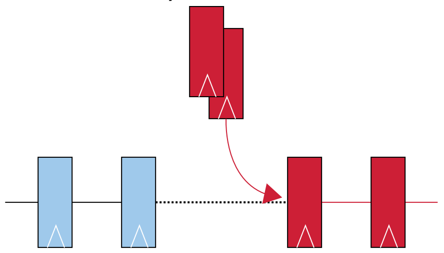


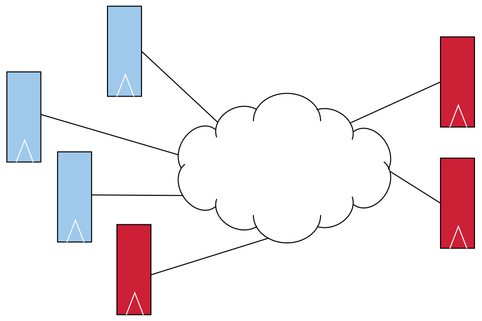
Figure 17. Simple Clock Domain Crossing After Adding Latency

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



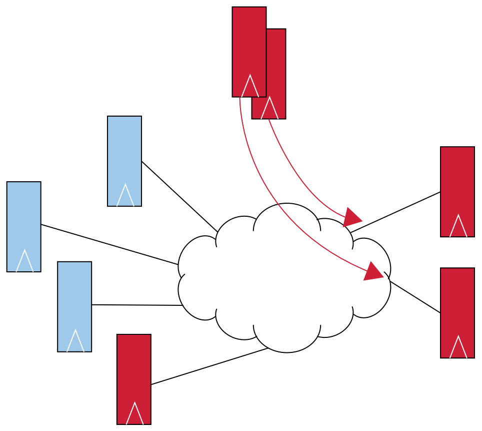
The following figure shows a domain crossing structure that is not recommended for use in Stratix 10 designs, but may exist in designs that target other device families. The design contains some combinational logic between the blue clock domain and the red clock domain. This logic is not properly synchronized 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 18. Clock Domain Crossing at Multiple Locations



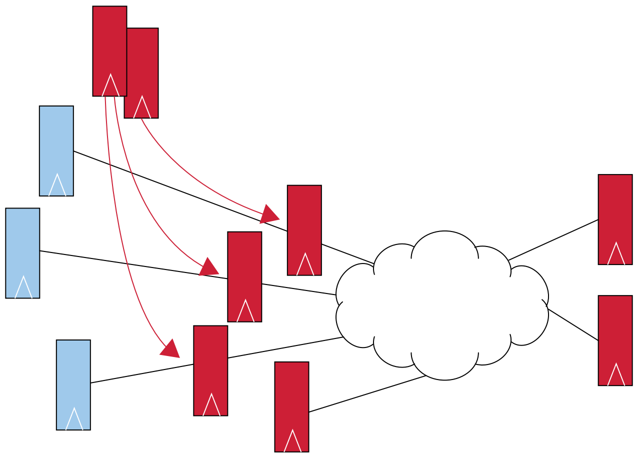
Add latency at the boundary of the red clock domain, but do not add registers on a red to red domain path. Otherwise, the paths become unbalanced, potentially changing design functionality. Although possible, adding latency in this scenario is risky. Thoroughly analyze the various paths before adding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

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



For Stratix 10 designs, synchronize the clock crossing paths before entering combinational logic. Adding latency is then more simple compared to the previous example. Blue domain registers synchronize to the red domain before entering the combinational logic. This method allows safe addition of pipeline registers in front of synchronizing registers, without contacting a red to red path inadvertently. This approach is the recommended synchronization method to take maximum advantage of the Stratix 10 architecture performance.

Figure 20. Improved Clock Domain Synchronization



2.2.6 Metastability Synchronizers

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

Metastability synchronizer chain length for S10 is 3. Registers required for metastability are now marked in the Critical Chain report as REG (Metastability required)

2.2.7 Initial Power-Up Conditions

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

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

2.2.7.1 Specifying Initial Conditions

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

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

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

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

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

2.2.7.2 Initial Conditions and Retiming

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

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

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

Figure 21. Circuit Before Retiming

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

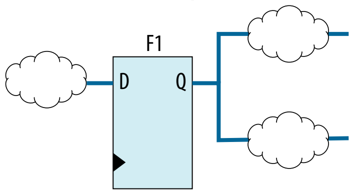
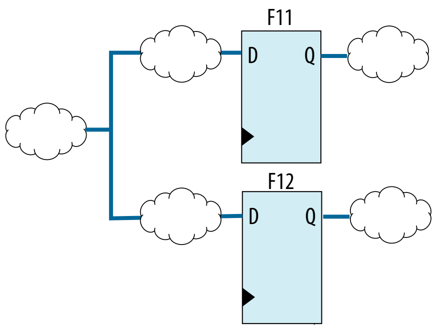


Figure 22. Circuit After Retiming Forward

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



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

Table 3. Possible Power-Up States After Retiming

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

C-Cycle Equivalence

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

Retiming Backward

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

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

2.2.7.3 Retiming Reset Sequences

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

Reset Retiming Behavior

The Compiler's has the following behavior when retiming resets:

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

Ignoring Initial Conditions

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

```
set_global_assignment -name ALLOW_POWER_UP_DONT_CARE ON
```

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

Modifying the Reset Sequence

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

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

Adding Clock Cycles to Reset

The Compiler does not report the c-cycle value of retiming transformations in the design. However, evaluate the number and length of pipelines in your design. Then, add enough clock cycles to guarantee the functional equivalence of the design when exiting the reset sequence.

Many of the transformations that need attention have a c-cycle of 1. For example, register duplication into multiple branches has this c-cycle. Regardless of the number of duplicate registers, the register is always one connection away from its original source. After one clock cycle, all the branches have the same value again.

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

Figure 23. Pipelining and Register Duplication

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

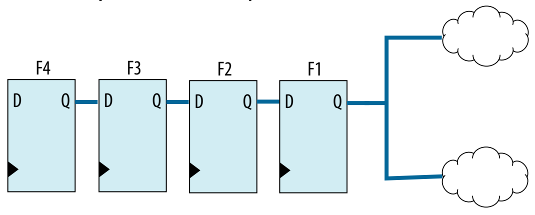


Figure 24. Impact of One Register Move

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

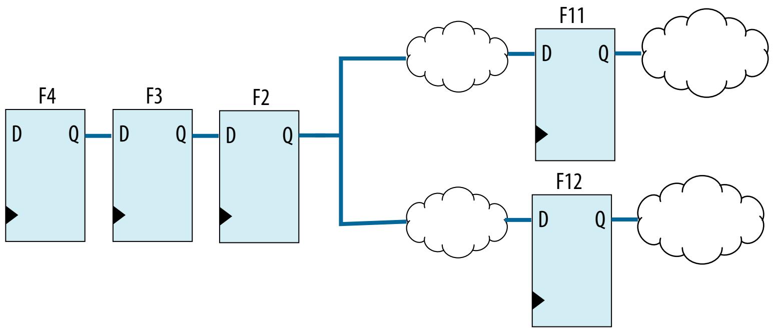
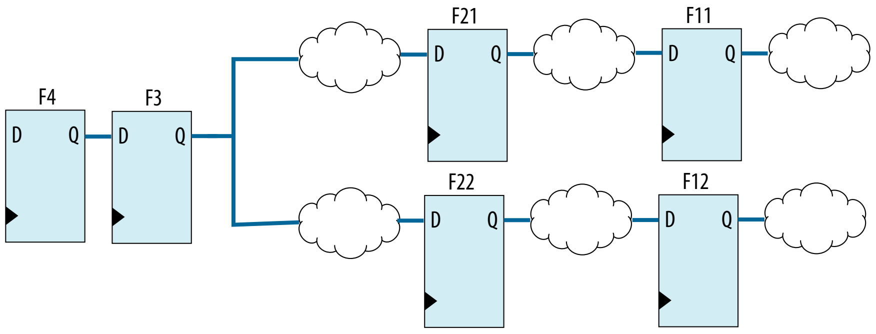


Figure 25. Impact of Two Register Moves

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



Each time a register from the pipeline is moved into the logic, the register duplicate and the C-cycle value of the design is increased by one.

2.3 Hyper-Pipelining (Add Pipeline Registers)

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

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

Changing latency is more 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, implement some latencies as parameters. In some types of systems, you may be able to add a "valid data" flag to pipeline stages in a processing pipeline to trigger various computations, instead of relying on a high-level fixed concept of when data is valid.

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

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

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

Related Links

- Appendix A: Parameterizable Pipeline Modules on page 115
- Precomputation on page 34

2.3.1 Conventional Versus Hyper-Pipelining

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

Conventional pipelining includes the following design modifications:

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

Figure 26. Conventional Pipelining User Modifications

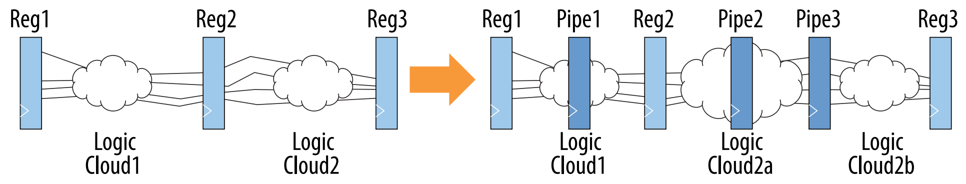


Figure 27. Hyper-Pipelining User Modifications

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

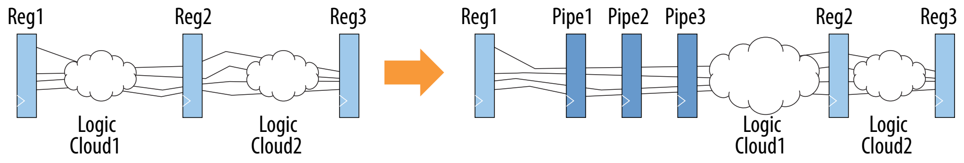
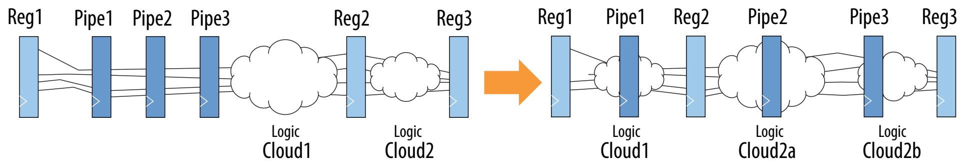


Figure 28. Hyper-Pipelining and Hyper-Retiming Implementation

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



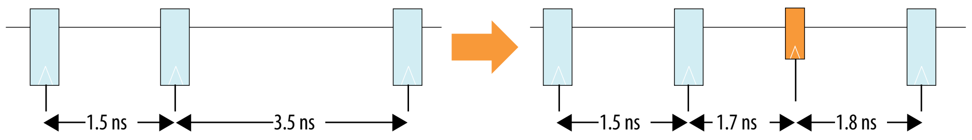
The resulting implementation in the Hyper-Pipelining flow differs from the conventional pipelining flow by the location of the Pipe 3 register. Because the Compiler is aware of the current circuit implementation, including routing, it can more effectively locate the added aggregate registers to meet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

2.3.2 Pipelining and Latency

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

Figure 29. Hyper-Pipeline Reduced Latency

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



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

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

2.3.3 Use Registers Instead of Multicycle Exceptions

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

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

Related Links

- Optimize Multicycle Paths on page 18
- Appendix A: Parameterizable Pipeline Modules on page 115

2.4 Hyper-Optimization (Optimize RTL)

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

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

2.4.1 General Optimization Techniques

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

2.4.1.1 Shannon's Decomposition

Shannon's decomposition plays a role in Hyper-Optimization. Shannon's decomposition, or Shannon's expansion, is a way of factoring a Boolean function. You can express a function as $F = x.F_x + x'F_x'$ where $x.F_x$ and $x'F_x'$ are the positive and negative co-factors of the function F with respect to x. You can factor a function with four inputs as, (a, b, c, x) = x.(a, b, c, 1) + x'.F(a, b, c, 0), as shown in the following diagram. In Hyper-Optimization, 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.

Figure 30. Shannon's Decomposition

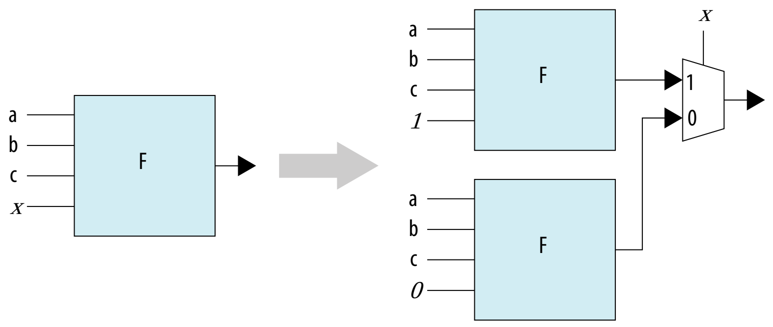


Figure 31. Shannon's Decomposition Logic Reduction

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

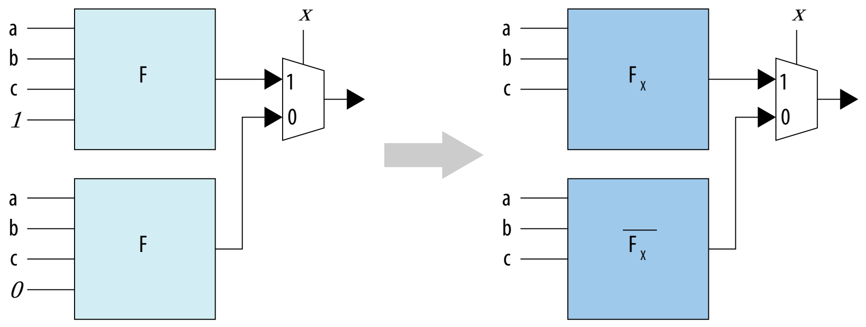
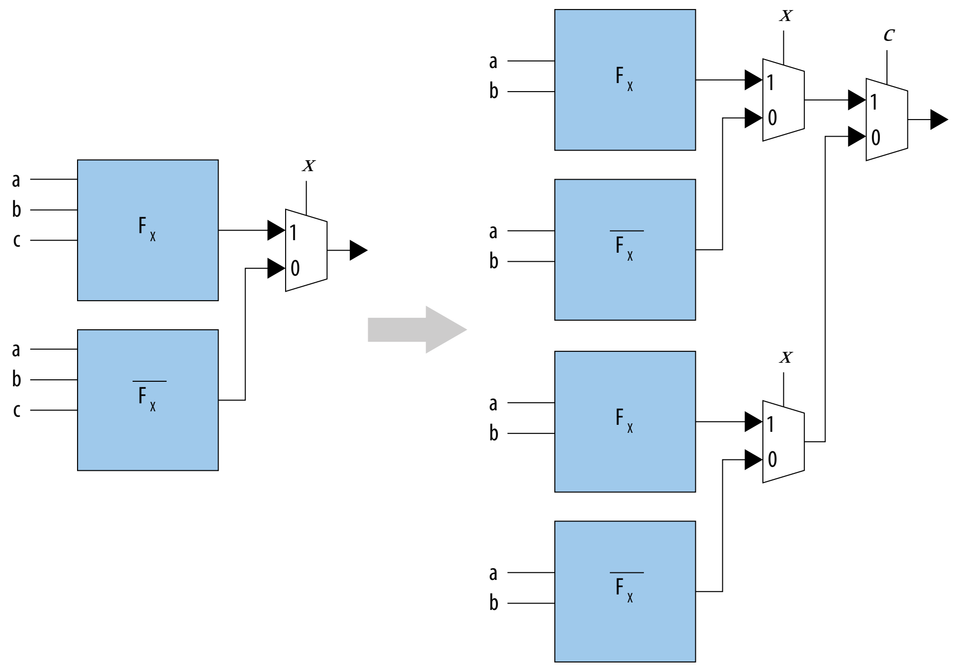


Figure 32. Repeated Shannon's Decomposition

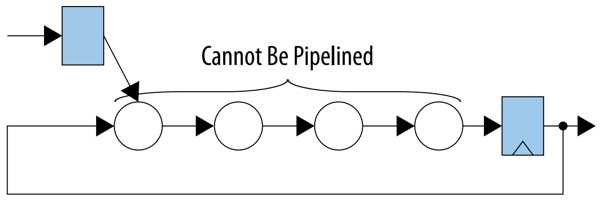
The following diagram shows how you can repeatedly use Shannon's decomposition to decompose functions with more than one critical input signal, thus increasing the area cost.



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

Figure 33. Loop Example before Shannon's Decomposition

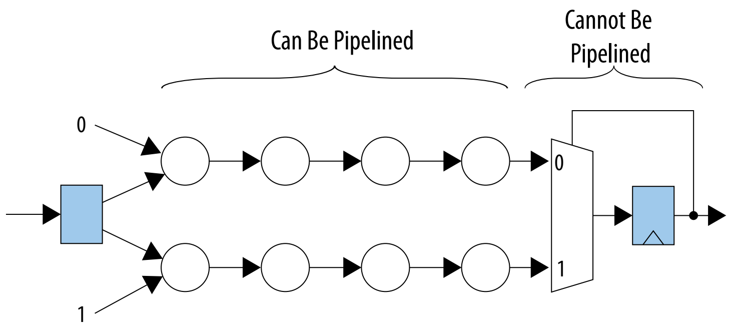
This diagram shows a loop that contains a single register, four levels of combinational logic, and an additional input. Adding registers in the loop changes the functionality, but you can move the combinational logic outside the loop by performing Shannon’s decomposition.



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

Figure 34. Loop Example after Shannon's Decomposition

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



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

2.4.1.1.1 Shannon’s Decomposition Example

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

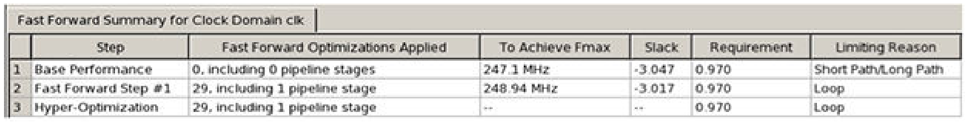
Example 5. Source Code before Shannon's Decomposition

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

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

reg [WIDTH-1:0] internal_total;

always @(posedge clk) begin
        if (sclr)
        begin
             internal_total <= 0;
        end
        else begin
          internal_total <= internal_total + ((( internal_total > target) ? -
data:data)* target/4));
        end
end
assign running_total = internal_total;
end module
```

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

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

Figure 35. Fast Forward Compile Report before Shannon’s Decomposition



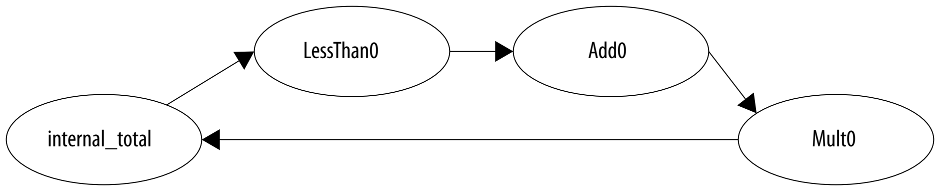
Fast Forward Summary for Clock Domain clk

	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

Hyper-Retiming reports about 248 MHz by adding a pipeline stage in the Fast Forward Compile. The Limiting Reason column indicates that the critical chain is a loop. Examining the critical chain report reveals that there is a repeated structure in the chain segments. The repeated structure is shown as an example in the *Optimizing Loops* section.

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

Figure 36. Elements of a Critical Chain Sub-Loop



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

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

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

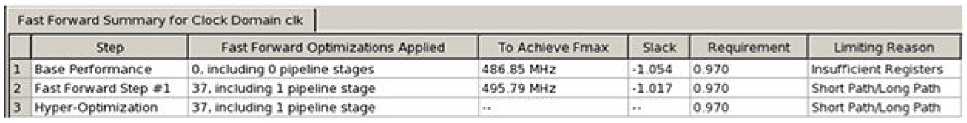
Example 6. Source Code after Shannon's Decomposition

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

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

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

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

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

assign running_total = internal_total;
endmodule
```

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

Figure 37. Fast Forward Summary Report after Shannon's Decomposition



Fast Forward Summary for Clock Domain clk

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

2.4.1.1.2 Identifying Circuits for Shannon's Decomposition

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

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

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

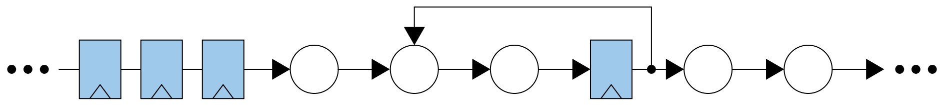
2.4.1.2 Time Domain Multiplexing

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

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

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

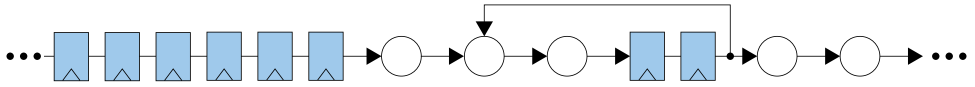
Figure 38. C-slow Retiming Starting Point



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

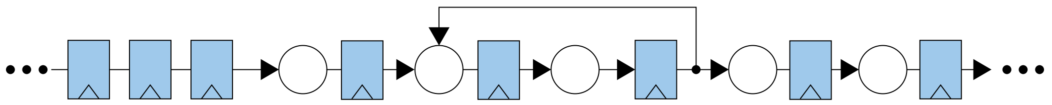
Figure 39. C-slow Retiming Intermediate Point

This example shows replacement of each register with two registers.



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

Figure 40. C-Slow Retiming Ending Point



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

2.4.1.3 Loop Unrolling

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

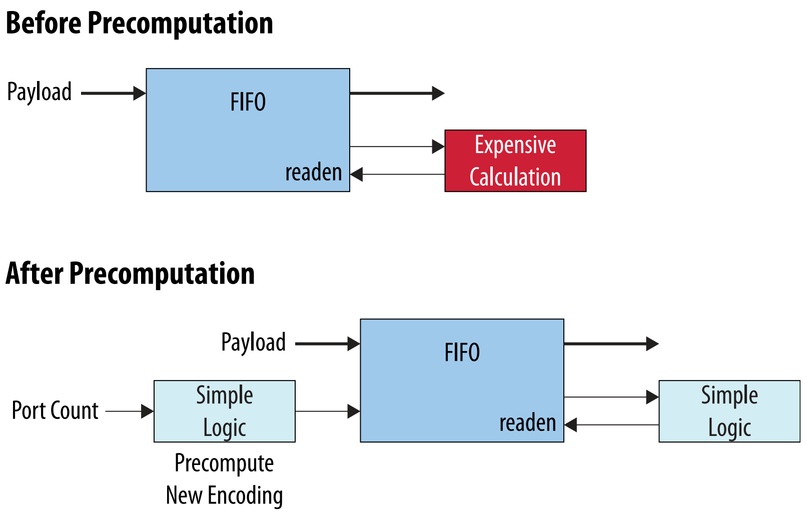
2.4.1.4 Precomputation

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

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

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

Figure 41. 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;
```

2.4.2 Optimizing Specific Design Structures

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

2.4.2.1 High-Speed Clock Domains

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

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

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

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

2.4.2.1.1 Visualizing Clock Networks

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

To visualize design clock networks in the Chip Planner:

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

Figure 42. Clock Network in Chip Planner

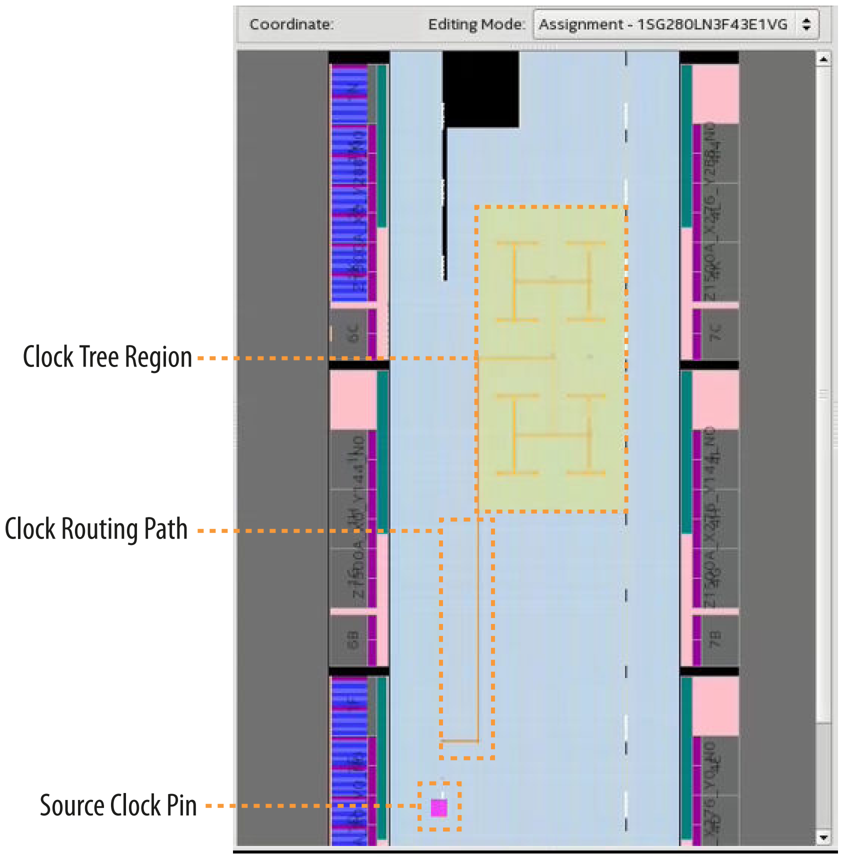
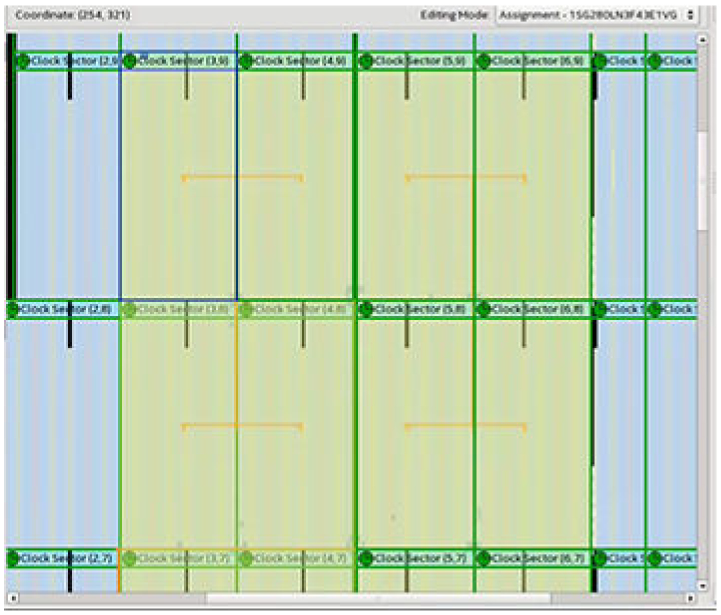


Figure 43. Clock Sector Boundary Layer in Chip Planner



2.4.2.1.2 Viewing Clock Networks in the Fitter Report

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

To view clock network implementation in Fitter reports:

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

Figure 44. Clock Network Details in Fitter Report

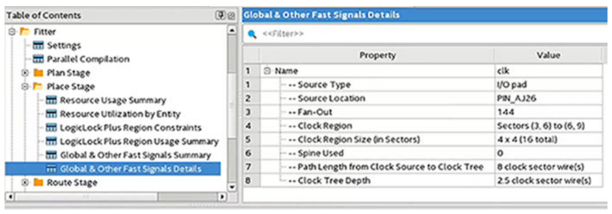


Table of Contents
Fitter
Settings
Parallel Compilation
Plan Stage
Place Stage
Resource Usage Summary
Resource Utilization by Entity
LogicLock Plus Region Constraints
LogicLock Plus Region Usage Summary
Global & Other Fast Signals Summary
Global & Other Fast Signals Details
Route Stage
Global & Other Fast Signals Details
<<Filter>>

	Property	Value
1	Name	clk
1	-- Source Type	I/O pad
2	-- Source Location	PIN_AJ26
3	-- Fan-Out	144
4	-- Clock Region	Sectors (3, 6) to (6, 9)
5	-- Clock Region Size (in Sectors)	4 x 4 (16 total)
6	-- Spine Used	0
7	-- Path Length from Clock Source to Clock Tree	8 clock sector wire(s)
8	-- Clock Tree Depth	2.5 clock sector wire(s)

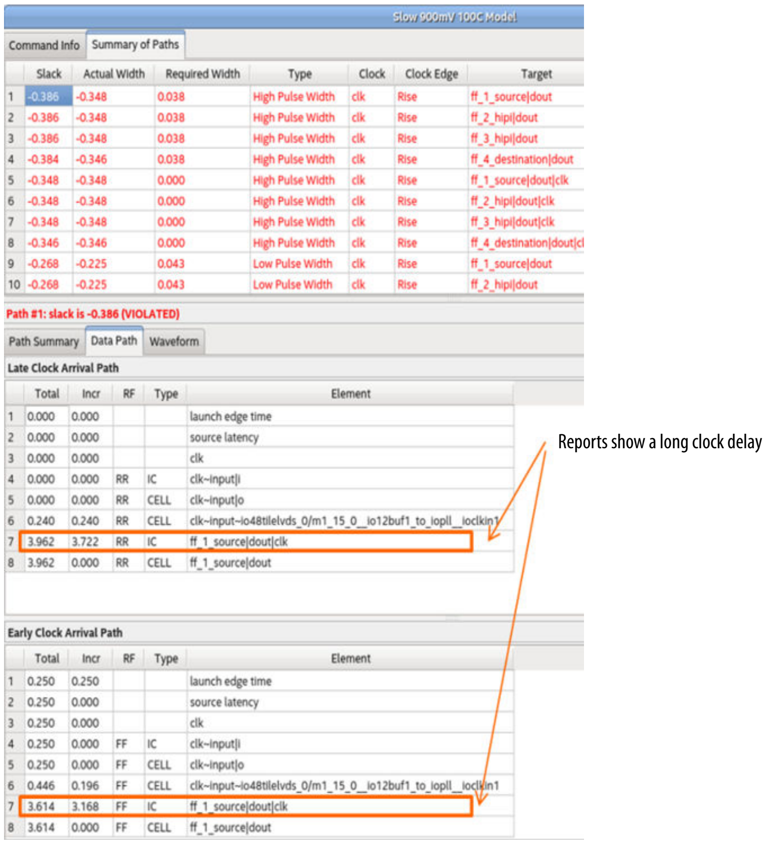
2.4.2.1.3 Viewing Clocks in TimeQuest

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

Top view clock network data in TimeQuest:

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

Figure 45. Minimum Pulse Width Details Show Long Clock Route



	Slack	Actual Width	Required Width	Type	Clock	Clock Edge	Target
1	-0.386	-0.348	0.038	High Pulse Width	clk	Rise	ff_1_source|dout
2	-0.386	-0.348	0.038	High Pulse Width	clk	Rise	ff_2_hipi|dout
3	-0.386	-0.348	0.038	High Pulse Width	clk	Rise	ff_3_hipi|dout
4	-0.384	-0.346	0.038	High Pulse Width	clk	Rise	ff_4_destination|dout
5	-0.348	-0.348	0.000	High Pulse Width	clk	Rise	ff_1_source|dout|clk
6	-0.348	-0.348	0.000	High Pulse Width	clk	Rise	ff_2_hipi|dout|clk
7	-0.348	-0.348	0.000	High Pulse Width	clk	Rise	ff_3_hipi|dout|clk
8	-0.346	-0.346	0.000	High Pulse Width	clk	Rise	ff_4_destination|dout|cl
9	-0.268	-0.225	0.043	Low Pulse Width	clk	Rise	ff_1_source|dout
10	-0.268	-0.225	0.043	Low Pulse Width	clk	Rise	ff_2_hipi|dout

Path #1: slack is -0.386 (VIOLATED)

Late Clock Arrival Path

	Total	Incr	RF	Type	Element
1	0.000	0.000			launch edge time
2	0.000	0.000			source latency
3	0.000	0.000			clk
4	0.000	0.000	RR	IC	clk~input|i
5	0.000	0.000	RR	CELL	clk~input|o
6	0.240	0.240	RR	CELL	clk~input~io48tilelvds_0/m1_15_0__io12buf1_to_iopll__ioclkin1
7	3.962	3.722	RR	IC	ff_1_source|dout|clk
8	3.962	0.000	RR	CELL	ff_1_source|dout

Early Clock Arrival Path

	Total	Incr	RF	Type	Element
1	0.250	0.250			launch edge time
2	0.250	0.000			source latency
3	0.250	0.000			clk
4	0.250	0.000	FF	IC	clk~input|i
5	0.250	0.000	FF	CELL	clk~input|o
6	0.446	0.196	FF	CELL	clk~input~io48tilelvds_0/m1_15_0__io12buf1_to_iopll__ioclkin1
7	3.614	3.168	FF	IC	ff_1_source|dout|clk
8	3.614	0.000	FF	CELL	ff_1_source|dout

2.4.2.2 Restructuring Loops

Loops are a primary target of restructuring techniques because they 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 very complex, potentially traveling through multiple registers before returning to the original register. All useful circuits contain loops.

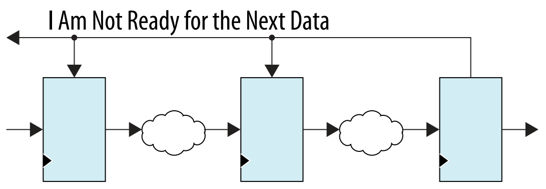
The Compiler never retimes registers into a loop because adding a pipeline stage to a loop would change functionality. However, change your RTL manually to restructure loops to improve performance. Perform loop optimization after analyzing performance bottlenecks with Fast Forward compile. Also apply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

2.4.2.3 Control Signal Backpressure

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

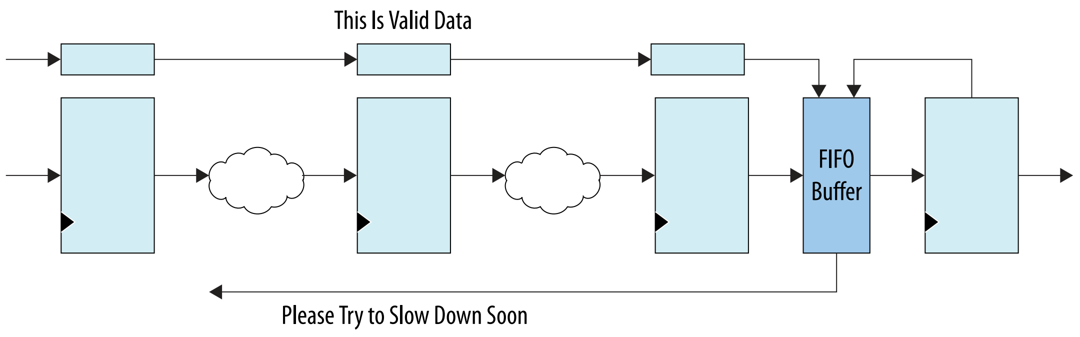
Single clock cycle control signals create loops that can prevent or reduce the effectiveness of pipelining and register retiming. This example depicts a ready signal that notifies the upstream register of readiness to consume data. The ready signals must freeze multiple data sources at the same time.

Figure 46. Control Signal Backpressure



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

Figure 47. Using a FIFO Buffer to Control Backpressure



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

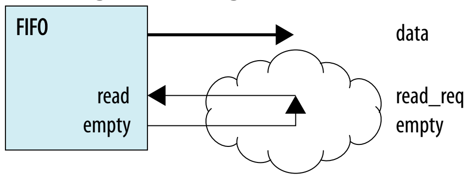
2.4.2.4 Flow Control with FIFO Status Signals

High clock speeds require consideration when dealing with flow control signals. This consideration is particularly important with signals that gate a data path in multiple locations at the same time. For example, with clock enable or FIFO full/empty signals. Instead of working with immediate control signals, use a delayed signal. You can build a buffer within the FIFO block. The control signals indicate to the upstream data path that 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 48. FIFO Flow Control Loop

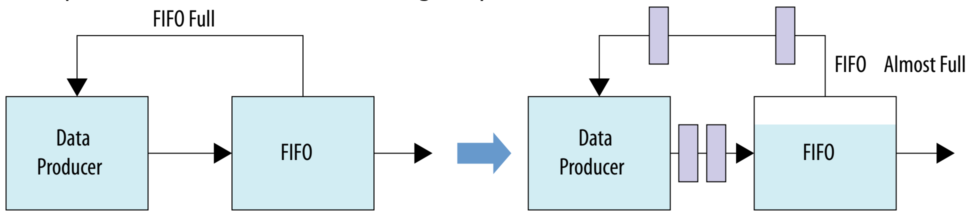
The loop is formed while reading control signals from the FIFO.



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

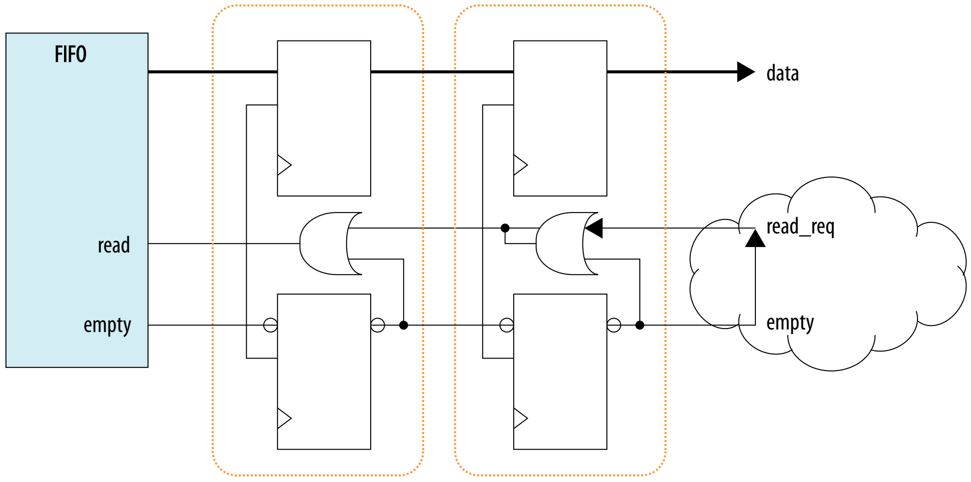
Figure 49. Improved FIFO Flow Control Loop with Almost Full instead of Full FIFO

The following example shows two extra registers in the full control flow signal. When the FIFO block signals that 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.



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



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

Example 7. Skid Buffer Example

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

module  singleclock_fifo_lowell
#(

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

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

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

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

    scfifo scfifo_component
    (
        .clock        (clk),
        .data         (in_data),

        .rdreq        (e_pop_qual),
        .wrreq        (push),

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


endmodule
```

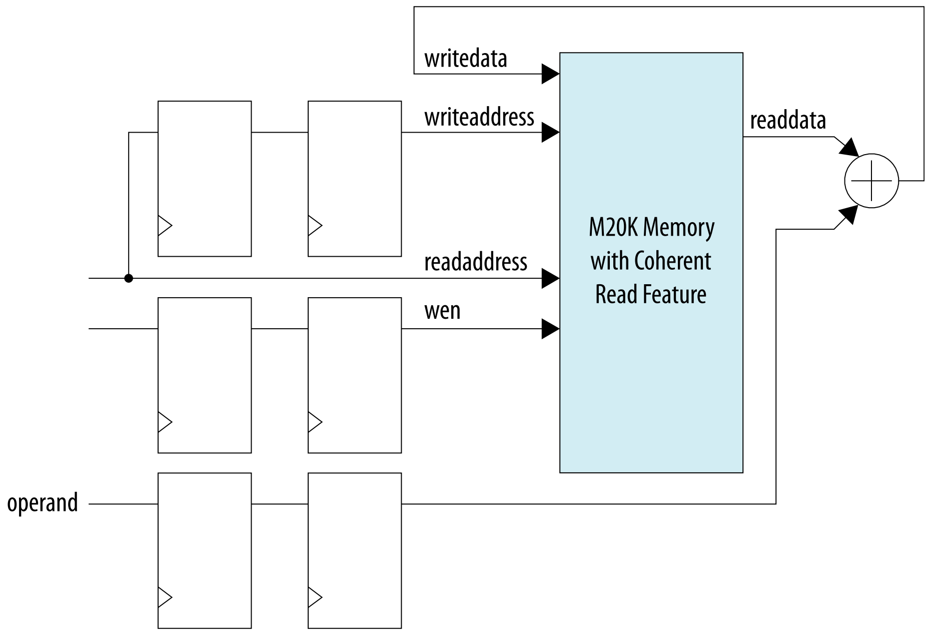
2.4.2.5 Read-Modify-Write Memory

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

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

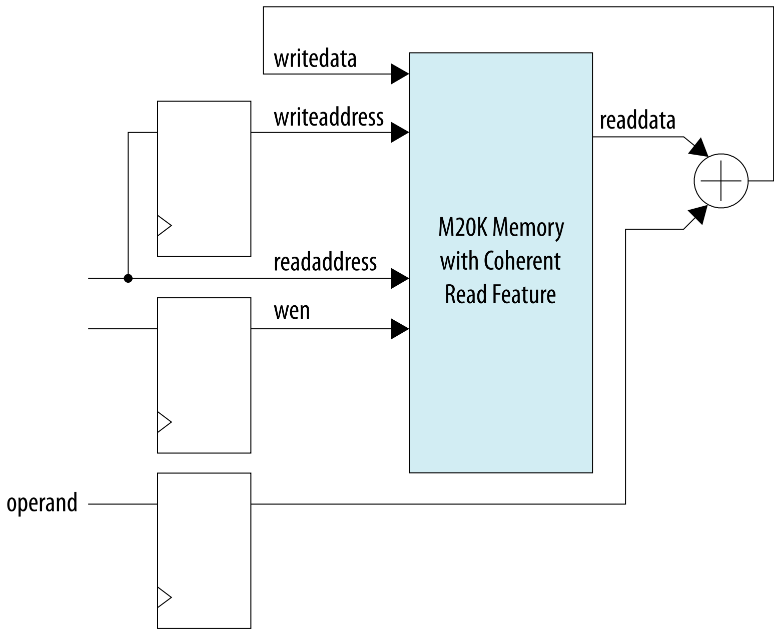
To use the coherent read feature, connect memory according to whether you register the output data port. If you register the output data port, add two register stages to the write enable and write address lines when you instantiate the memory.

Figure 51. Registered Output Data Requires Two Register Stages



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

Figure 52. Unregistered Output Data Requires One Register Stage



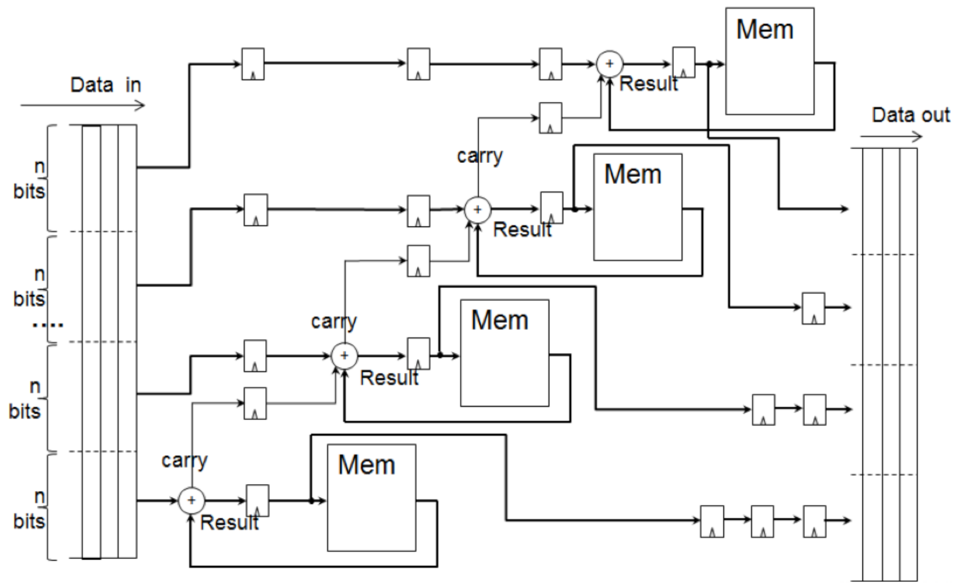
Use of coherent read has the following restrictions:

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

Figure 53. Pipelining Read-Modify-Write Memory

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

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



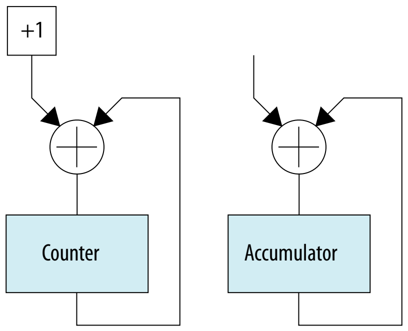
2.4.2.6 Counters and Accumulators

Performance-limiting loops occur rarely in small, simple counters. Counters with unnatural rollover conditions (not a power of two), or irregular increments, are more likely to have a performance-limiting critical chain. When a performance-limiting loop occurs in a small counter (roughly 8 bits or less), write the counter as a fully decoded state machine, depending on all the inputs that control the counter. The counter still contains loops, but they are smaller, and not performance-limiting. When the counter is small (roughly 8 bits or less), the fitter implements 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 54. Counter and Accumulator Loop

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



2.4.2.7 State Machines

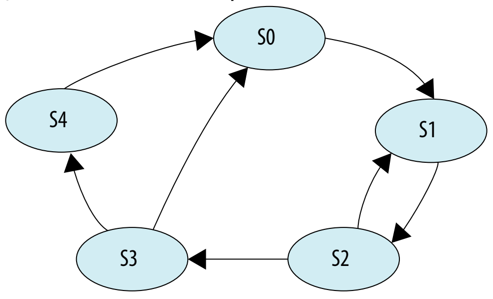
Loops related to state machines can be difficult to optimize. Carefully examine the state machine logic to determine whether you can precompute any signals used in the next state logic.

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 55. State Machine Loop

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



Related Links

- Appendix A: Parameterizable Pipeline Modules on page 115
- Precomputation on page 34

2.4.2.8 Memory

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

2.4.2.8.1 True Dual-Port Memory

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

Stratix 10 embedded memory components (M20k) have slightly different modes of operation compared to previous Intel FPGA technology, including mixed-width ratio for read/write access.

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.

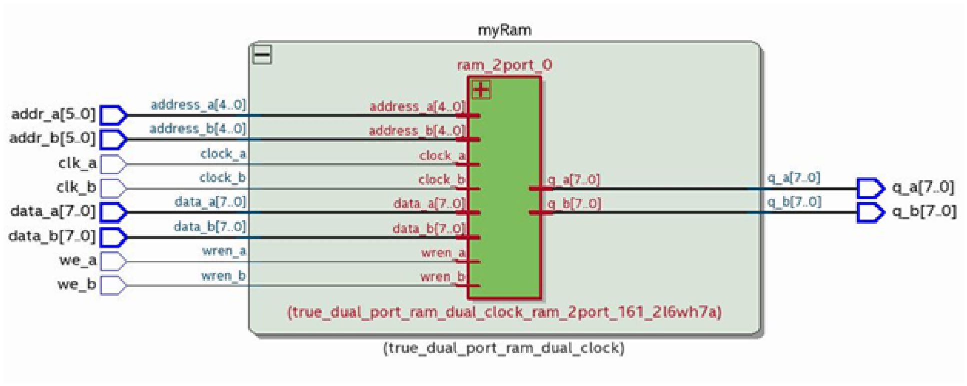
2.4.2.8.2 Use Simple Dual-Port Memories

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, restructure it using two simple dual-clock memories.

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

Figure 56. Arria® 10 True Dual-Port Memory Implementation

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



Example 8. 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
);
```

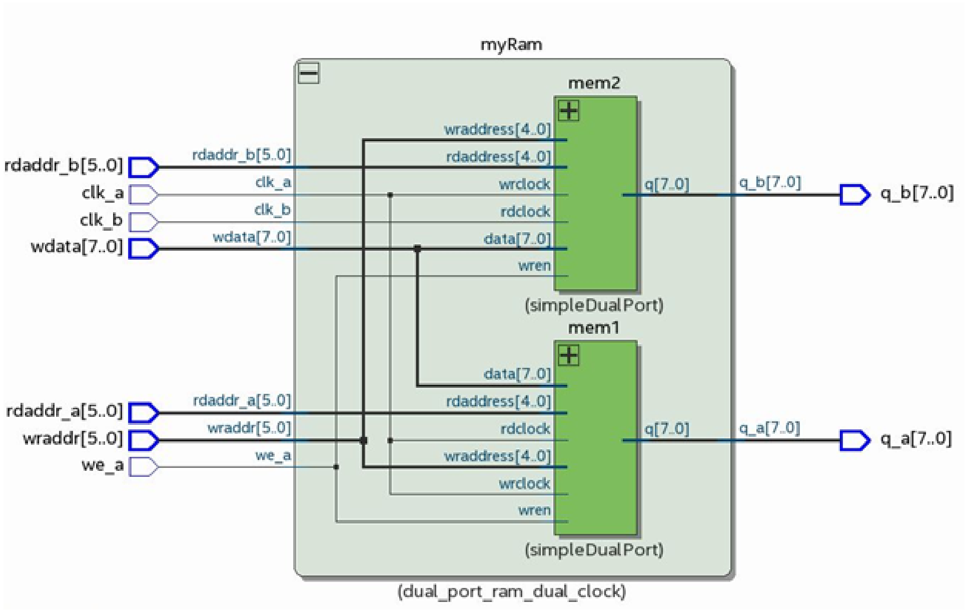
```
    // Declare the RAM variable
    reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

    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 57. Simple Dual-Port Memory Implementation



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

Example 9. 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 10. True Dual-Port RAM Behavior Emulation

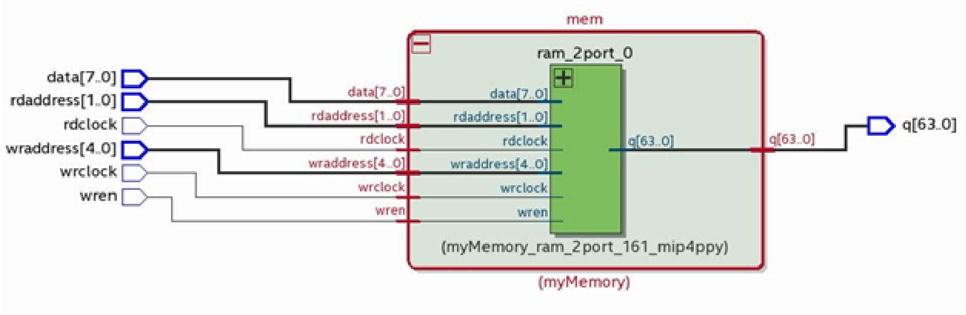
```
module test (wrdata, wraddr, rdaddr_a, rdaddr_b,
    clk_a, clk_b, we_a, q_a, q_b);

    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

Stratix 10 device block RAMs enable clocks speeds of up to 1GHz. The new RAM block design is more restrictive with respect to use of mixed ports data width. 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 generates an error message for implementation of invalid mixed port ratios.

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

Figure 58. Dual-Port Memory with Invalid 1/8 Mixed Port Ratio

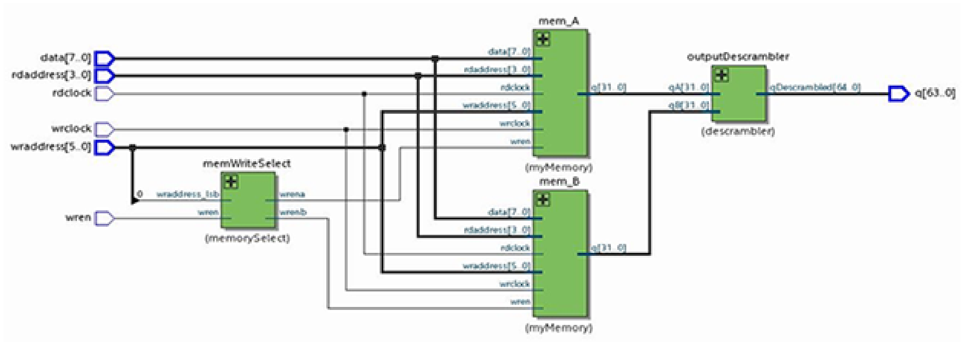


To create a functionally equivalent design for Stratix 10 devices, create and combine smaller memories with valid mixed port width ratios. For example, the following steps implement a mixed port width ratio:

1. Create two memories with ¼ mixed port width ratio by instantiating the 2-Ports memory IP core from the IP Catalog.
2. Define write enable logic to ping-pong writing between the two memories.
3. Interleave the output of the memories to rebuild a 1/8 ratio output.

Figure 59. 1/8 Width Ratio Example

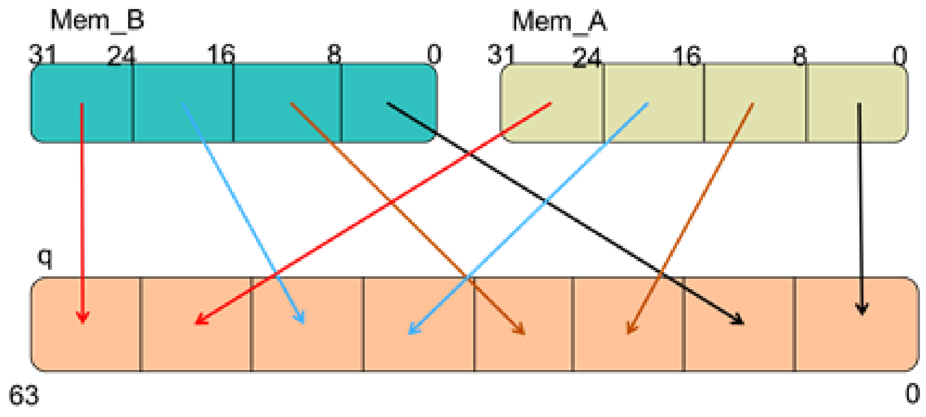
This example shows the interleaving of two memories and the write logic. The chosen write logic uses the least significant bit of the write address to decide which memory to write. Even addresses write in memory `mem_A`, odd addresses write in memory `mem_B`.



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

Figure 60. 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 11. 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 12. Supporting RTL Code

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

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

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

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

endmodule
```

2.4.2.8.3 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.

2.4.2.9 DSP Blocks

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

2.4.2.10 General Logic

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

2.4.2.11 Modulus and Division

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

Related Links

Round Robin Scheduler on page 108

2.4.2.12 Resets

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

2.4.2.13 Hardware Re-use

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

2.4.2.14 Algorithmic Requirements

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

Related Links

General Optimization Techniques on page 28

2.4.2.15 FIFOs

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

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

Related Links

Unregistered RAM Outputs on page 53

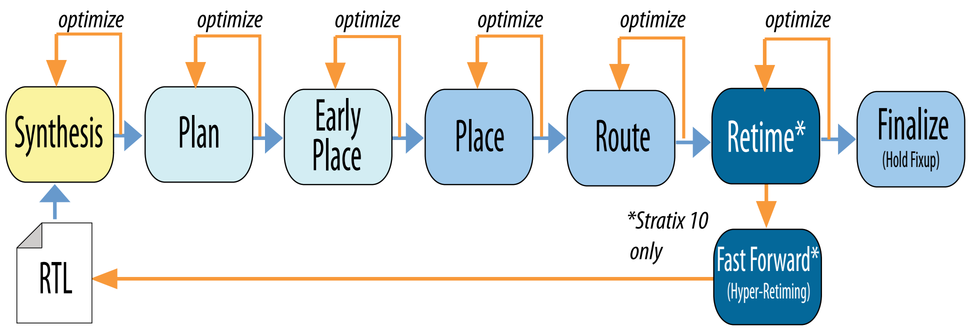
3 Compiling Stratix 10 Designs

This chapter describes how to use the Quartus Prime Compiler to take full advantage of the Intel HyperFlex FPGA architecture in Stratix 10 devices.

Hyper-Aware Design Flow

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

Figure 61. Hyper-Aware Design Flow



Register Retiming

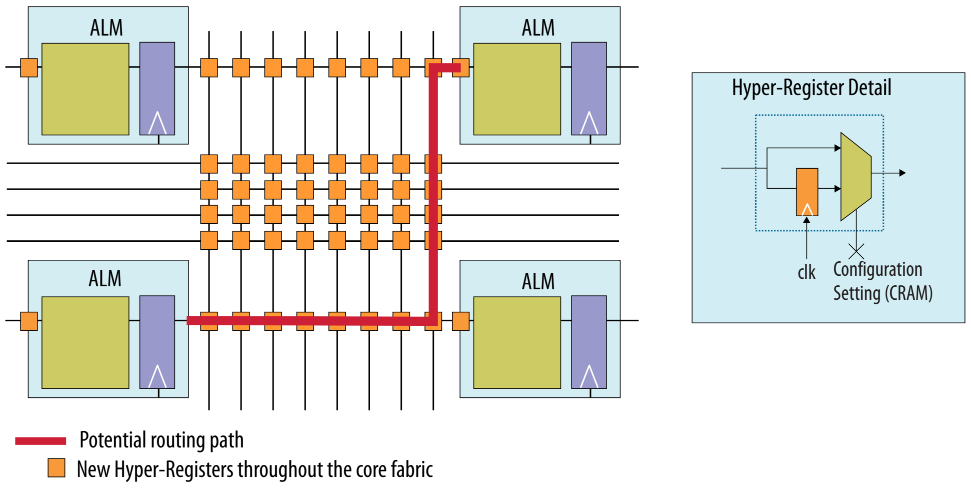
A key innovation of the Stratix 10 architecture is the addition of multiple Hyper-Registers in every routing segment and block input. Maximizing the use of Hyper-Registers improves design performance. The prevalence of Hyper-Registers improves balance of time delays between registers and mitigates critical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

Fast Forward Compilation

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

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

Figure 62. Hyper-Register Architecture


ALM
ALM
ALM
ALM
Hyper-Register Detail
clk
Configuration Setting (CRAM)
Potential routing path
New Hyper-Registers throughout the core fabric

Fast-Forward compilation identifies the best location to add pipeline stages (Hyper-Pipelining), and the expected performance benefit in each case. After you modify the RTL to place pipeline stages at the boundaries of each clock domain, the **Retime** stage automatically places the registers within the clock domain at the optimal locations to maximize performance. Implement the recommendations in RTL to achieve similar results. After implementing any changes, re-run the **Retime** stage until the results meet performance and timing requirements.

Table 4. Optimization Steps

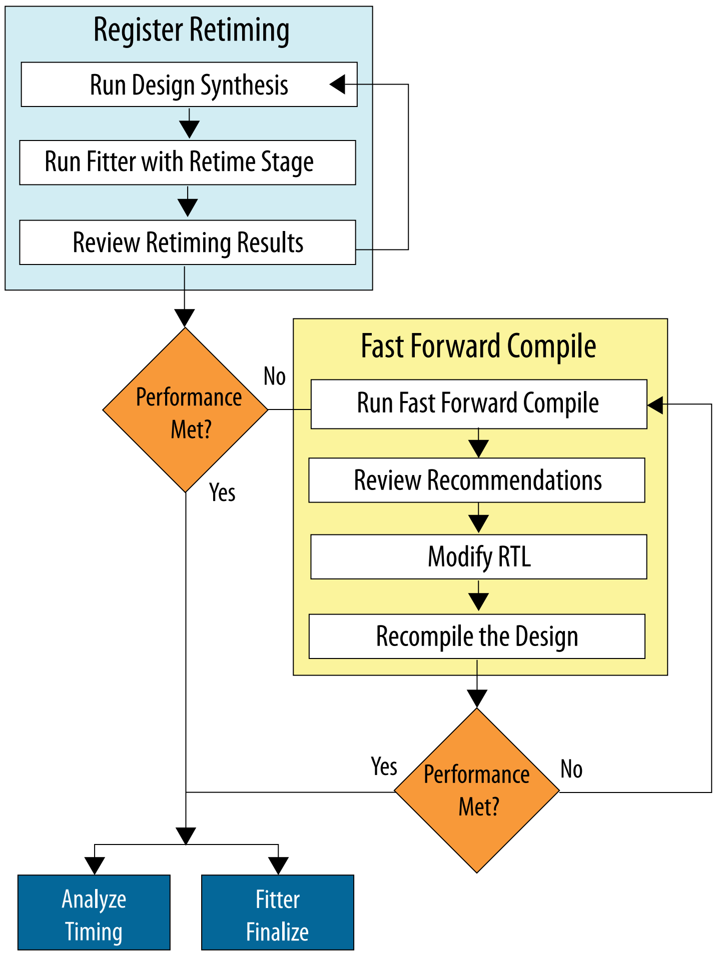
Optimization Step	Technique	Description
Step 1	Register Retiming	**Retime** stage moves existing registers into Hyper-Registers.
Step 2	Fast Forward Compile	Compiler generates design-specific timing closure recommendations and predicts performance improvement with removal of all barriers to Hyper-Registers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

Note: The Stratix 10 Hyper-Optimization Advisor provides step-by-step instructions to run Fast Forward compilation and implement Hyper-Optimization. Click **Tools ➤ Advisors ➤ Stratix 10 Hyper-Optimization Advisor** to view advice.

3.1 Running the Hyper-Aware Design Flow

The Hyper-Aware design flow combines register retiming with Fast Forward Compilation to maximize use of available Stratix 10 Hyper-Registers.

Figure 63. Hyper-Aware Design Flow



The Hyper-Aware design flow includes the following high-level steps this chapter covers in detail:

1. Run the **Retime** stage during the Fitter to automatically retime ALM registers into Hyper-Registers.
2. Review Retiming Results in the Compilation Report.
3. If you require further performance optimization, run Fast Forward compilation.
4. Review Fast Forward timing closure recommendations.
5. Implement appropriate Fast Forward recommendations in your RTL.
6. Recompile the design through the **Retime** stage.

Related Links

- Step 1: Run Register Retiming on page 58
- Step 2: Review Retiming Results on page 60
- Step 3: Run Fast Forward Compile and Hyper-Retiming on page 63
- Step 4: Review Hyper-Retiming Results on page 65

- Step 5: Implement Fast Forward Recommendations on page 68

3.1.1 Step 1: Run Register Retiming

Register retiming improves design performance by moving registers out of ALMs and retimes them into Hyper-Registers in the Stratix 10 device interconnect.

The Fitter runs the **Retime** stage automatically following place and route when you target a Stratix 10 device. Alternatively, start or stop the individual **Retime** stage in the Compilation Dashboard. After running register retiming, view the Fitter reports to optimize remaining critical paths.

To run register retiming:

1. Create or open a Quartus Prime project that is ready for design synthesis and fitting.
2. To run register retiming, click **Retime** on the Compilation Dashboard. The Compiler runs prerequisite stages automatically, as needed. The Compiler generates detailed reports and timing analysis data for each stage. Click the **Report** or **TimeQuest** icons to review results of each stage. Rerun any stage to apply any setting or design changes.
3. If register retiming achieves all performance goals for your design, proceed to **Fitter (Finalize)** and TimeQuest timing analysis stages of compilation. If your design requires further optimization, run **Fast Forward Timing Closure Recommendations**.

Figure 64. Retiming Stage in Compilation Dashboard

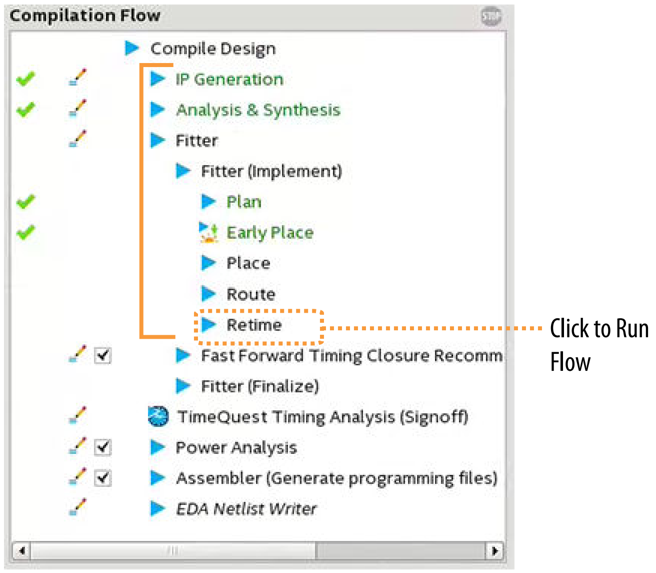


Table 5. Fitter Stage Commands

Command	Description
Fitter (Implement)	Runs the Plan, Early Place, Place, Route and Retime stages. Click the adjacent **TimeQuest** icon after this stage to analyze the subset of timing corners needed for timing closure.
Start Fitter (Plan)	Loads synthesized periphery placement data and constraints, and assigns periphery elements to device I/O resources. After this stage, you can run post-Plan timing analysis to verify timing constraints, check periphery timing, and validate cross-clock timing windows. This command creates the Planned snapshot.
Start Fitter (Early Place)	Begins assigning core design logic to device resources. After this stage, the Chip Planner can display initial high-level placement of design elements. Use this information to guide your floorplanning decisions. This command creates the Early Placed snapshot. Early Place does not run during the full compilation flow.
Start Fitter (Place)	Completes assignment of core design logic placement to device resources. This command creates the Placed snapshot.
Start Fitter (Route)	Performs core routing. This stage creates a fully routed design to validate delay chain settings and analyze routing resources. After this stage, perform detailed setup and hold timing closure in The TimeQuest Timing Analyzer and view routing congestion via the Chip Planner. This command creates the Routed snapshot.
Start Fitter (Retime)	Retimes existing registers in the design into Hyper-Registers to increase performance by removing retiming restrictions and eliminate critical paths. The Compiler may report hold violations for short paths following the Retime stage. The Fitter identifies and corrects the short paths with hold violations during the **Fitter (Finalize)** stage by adding routing wire along the paths.
Start Fitter (Finalize)	Finalizes the place and route process after timing closure. This command creates the Final snapshot. The Fitter also runs post-route fix-up to correct any short path hold violations remaining from retiming.

3.1.1.1 Prevent Register Retiming

Enable the **Prevent register retiming** option if you want to globally prevent automatic retiming of registers for design performance improvement. When disabled, the Compiler automatically performs register retiming optimizations that move combinational logic across register boundaries. The Compiler maintains the overall logic of the design component, and also balances the datapath delays between each register. Optionally, assign **Allow Register Retiming** to any design entity or instance to override **Prevent register retiming** for specific portions of the design. Click **Assignments ➤ Assignment Editor** to specify entity- and instance-level assignments, or use the following syntax to make the assignment in the `.qsf` directly.

Example 13. Disable register retiming for entity abc

```
set_global_assignment –name ALLOW_REGISTER_RETIMING ON

set_instance_assignment –name ALLOW_REGISTER_RETIMING OFF –to “abc|”

set_instance_assignment –name ALLOW_REGISTER_RETIMING ON –to “abc|def|”
```

Example 14. Disable register retiming for the whole design, except for registers in entity abc

```
set_global_assignment –name ALLOW_REGISTER_RETIMING OFF

set_instance_assignment –name ALLOW_REGISTER_RETIMING ON –to “abc|”

set_instance_assignment –name ALLOW_REGISTER_RETIMING OFF –to “abc|def|”
```

3.1.1.2 Preserve Registers During Synthesis

Quartus Prime synthesis minimizes gate count, merges redundant logic, and ensures efficient use of device resources. If you need to preserve specific registers through synthesis processing, you can specify any of the following entity-level assignments. Use **Preserve Resisters in Synthesis** or **Preserve Fan-Out Free Register Node** to allow Fitter optimization of the preserved registers. **Preserve Resisters** restricts Fitter optimization of the preserved registers. Specify synthesis preservation assignments by clicking **Assignments ➤ Assignment Editor**, in the `.qsf` file, or as synthesis attributes in your RTL.

Table 6. Synthesis Preserve Options

Assignment	Description	Allows Fitter Optimization?	Assignment Syntax
Preserve Resisters in Synthesis	Prevents removal of registers during synthesis. This settings does not affect retiming or other optimizations in the Fitter.	Yes	• `PRESERVE_REGISTER_SYN_ONLY ON|Off -to <entity>` (.qsf) • `preserve_syn_only` or `syn_preservesyn_only` (synthesis attributes)
Preserve Fan-Out Free Register Node	Prevents removal of assigned registers without fan-out during synthesis.	Yes	• `PRESERVE_REGISTER_FANOUT_FREE_NODE ON|Off -to <entity>` (.qsf) • `no_prune on` (synthesis attribute)
Preserve Resisters	Prevents removal and sequential optimization of assigned registers during synthesis. Sequential netlist optimizations can eliminate redundant registers and registers with constant drivers.	No	• `PRESERVE_REGISTER ON|Off -to <entity>` (.qsf) • `preserve`, `syn_preserve`, or `keep on` (synthesis attributes)

3.1.2 Step 2: Review Retiming Results

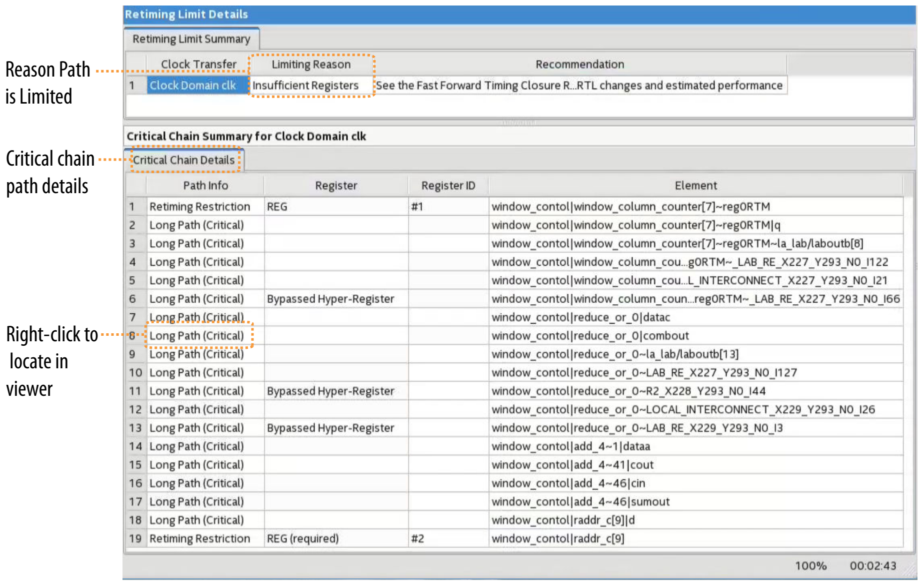
The Fitter generates detailed reports showing the results of the Retime stage. Follow these steps to review the results and make additional performance improvements with register retiming.

1. To open the **Retiming Limit Details** report, click the **Report** icon for the Retime stage in the Compilation Dashboard. The **Retiming Limit Details** lists the number of registers moved, the path(s) involved, and the limiting reason that prevents further retiming.
2. To further optimize, resolve any **Limiting Reason** in your design and rerun the **Retime** stage, as necessary.
3. If register retiming achieves all performance goals for your design, proceed to **Fitter (Finalize)** and **TimeQuest Timing Analysis** stages of compilation.
4. If your design requires further optimization, run Fast Forward compilation.

Table 7. Retiming Limit Details Report Data

Report Data	Description
Clock Transfer	Lists each clock domain in your design. Click the domain to display data about each entry.
Limiting Reason	Specifies the design condition(s) that prevent further register retiming improvement, such as any of the following conditions: • **Insufficient Registers**—indicates insufficient quantity of registers at either end of the chain for retiming. Adding more registers can improve performance. • **Short Path/Long Path**—indicates that the critical chain has dependent paths with conflicting characteristics. For example, one path could improve performance with more registers, and another path has no place for additional registers. • **Path Limit**—indicates that there are no further Hyper-Register locations available on the critical path, or the design reached a performance limit of the current place and route. • **Loops**—indicates a feedback path in a circuit. When the critical chain includes a feedback loop, retiming cannot change the number of registers in the loop without changing functionality. The Compiler can retime around the loop without changing functionality, but additional registers cannot be put in the loop.
Critical Chain Details	Lists register timing path associated with the retiming limitations. Right-click any path to **Locate Critical Chain in Technology Map Viewer**.

Figure 65. Retiming Limit Details



	Clock Transfer	Limiting Reason	Recommendation
1	Clock Domain clk	Insufficient Registers	See the Fast Forward Timing Closure R...RTL changes and estimated performance

Critical Chain Summary for Clock Domain clk

	Path Info	Register	Register ID	Element
1	Retiming Restriction	REG	#1	window_contol|window_column_counter[7]~reg0RTM
2	Long Path (Critical)			window_contol|window_column_counter[7]~reg0RTM|q
3	Long Path (Critical)			window_contol|window_column_counter[7]~reg0RTM~la_lab/laboutb[8]
4	Long Path (Critical)			window_contol|window_column_cou...g0RTM~_LAB_RE_X227_Y293_N0_I122
5	Long Path (Critical)			window_contol|window_column_cou...L_INTERCONNECT_X227_Y293_N0_I21
6	Long Path (Critical)	Bypassed Hyper-Register		window_contol|window_column_coun...reg0RTM~_LAB_RE_X227_Y293_N0_I66
7	Long Path (Critical)			window_contol|reduce_or_0|datac
8	Long Path (Critical)			window_contol|reduce_or_0|combout
9	Long Path (Critical)			window_contol|reduce_or_0~la_lab/laboutb[13]
10	Long Path (Critical)			window_contol|reduce_or_0~LAB_RE_X227_Y293_N0_I127
11	Long Path (Critical)	Bypassed Hyper-Register		window_contol|reduce_or_0~R2_X228_Y293_N0_I44
12	Long Path (Critical)			window_contol|reduce_or_0~LOCAL_INTERCONNECT_X229_Y293_N0_I26
13	Long Path (Critical)	Bypassed Hyper-Register		window_contol|reduce_or_0~LAB_RE_X229_Y293_N0_I3
14	Long Path (Critical)			window_contol|add_4~1|dataa
15	Long Path (Critical)			window_contol|add_4~41|cout
16	Long Path (Critical)			window_contol|add_4~46|cin
17	Long Path (Critical)			window_contol|add_4~46|sumout
18	Long Path (Critical)			window_contol|raddr_c[9]|d
19	Retiming Restriction	REG (required)	#2	window_contol|raddr_c[9]

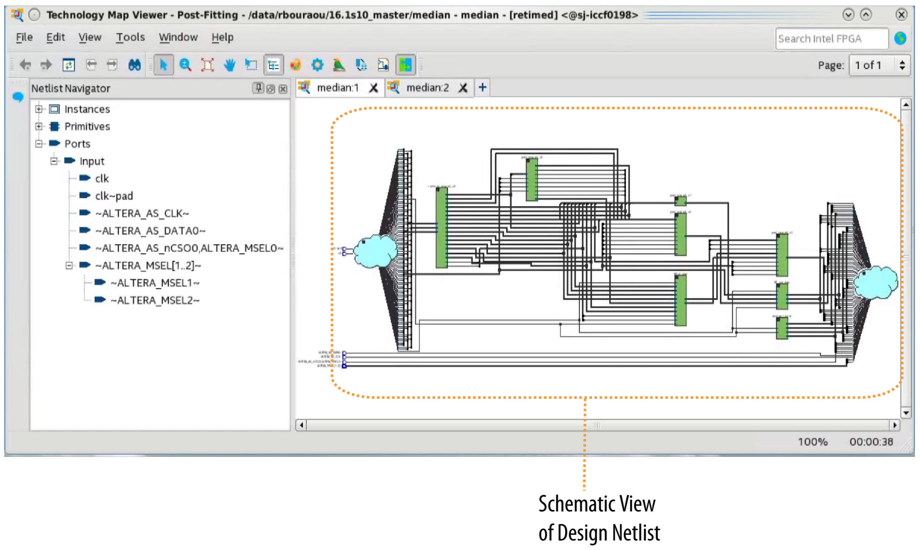
Note: The Compiler may report hold violations for short paths following the Retime stage. The Fitter identifies and corrects the short paths with hold violations during the **Fitter (Finalize)** stage by adding routing wire along the paths.

3.1.2.1 Locate Critical Chains

The **Retiming Limit Details** reports the design path(s) that limit further register retiming. Right-click any path to locate to the path in the Technology Map Viewer - Post-fitting view. This viewer displays a schematic representation of the complete design after place, route, and register retiming. To view the retimed netlist in the Technology Map Viewer, follow these steps:

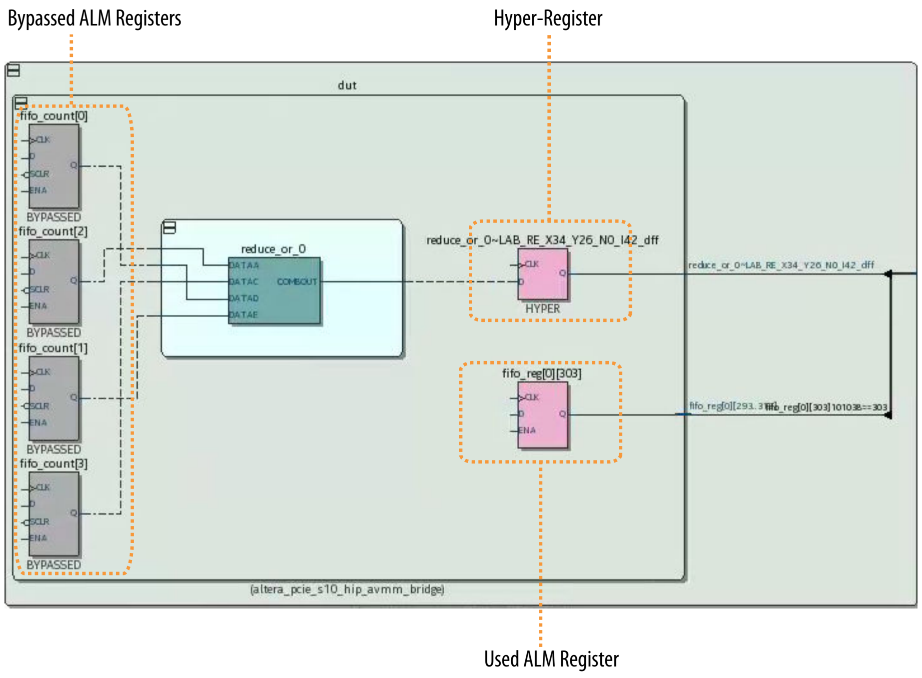
1. To open the **Retiming Limit Details** report, click the **Report** icon next to the **Retime** stage in the Compilation Dashboard.
2. Right-click any path in the **Retiming Limit Details** report and click **Locate Critical Chain in Technology Map Viewer**. The netlist displays as a schematic in the Technology Map Viewer.

Figure 66. Technology Map Viewer



Register retiming moves the register banks forward into Hyper-Registers.

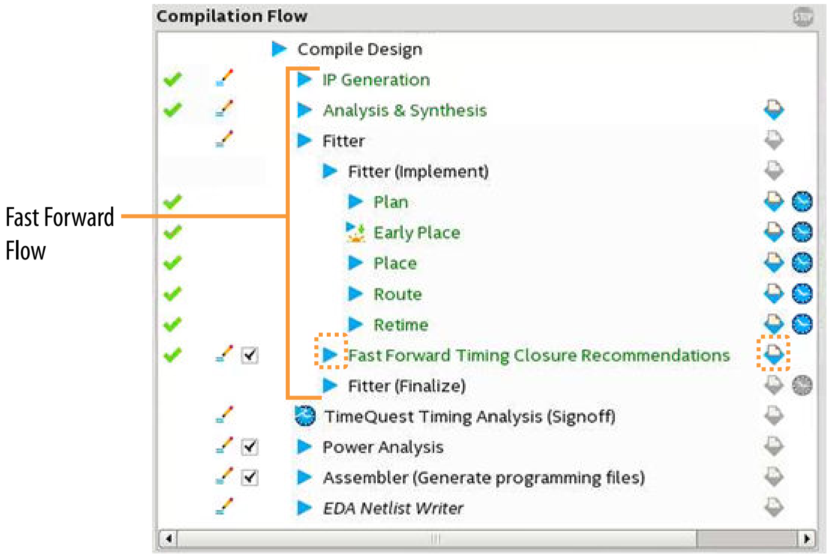
Figure 67. Post-Fit Viewer After Retiming



3.1.3 Step 3: Run Fast Forward Compile and Hyper-Retiming

When you run Fast Forward compilation, the Compiler predictively removes signals from registers to allow mobility within in the netlist for subsequent retiming. Fast Forward compilation generates design-specific timing closure recommendations, and predicts maximum performance with removal of all timing restrictions. After you complete Fast Forward explorations, determine which recommendations you can implement to provide the most benefit. Implement appropriate recommendations in your RTL, and recompile the design to realize the performance levels that Fast Forward reports.

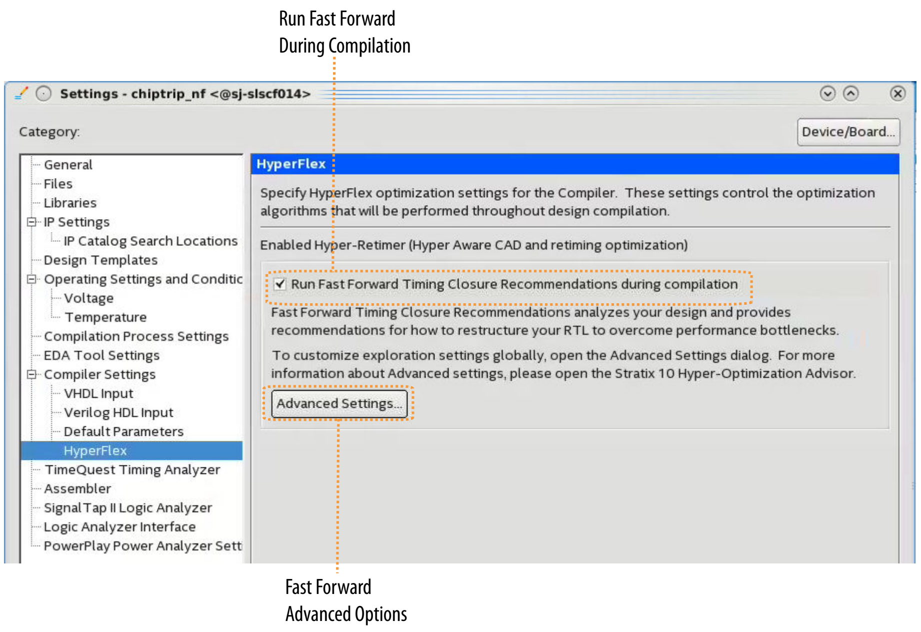
Figure 68. Running Fast Forward Compilation



To generate Fast Forward timing closure recommendations, follow these steps:

1. On the Compilation Dashboard, click **Fast Forward Timing Closure Recommendations**. The Compiler runs prerequisite synthesis or Fitter stages automatically, as needed, and generates timing closure recommendations in the Compilation Report.
2. View timing closure recommendations in the Compilation Report to evaluate design performance and implement key RTL performance improvements.
3. Optionally, specify any of the following any of the following options if you want to automate or refine Fast Forward analysis:
   - If you want to run Fast Forward compilation during each full compilation, click **Assignments ➤ Settings ➤ Compiler Settings ➤ HyperFlex** and enable **Run Fast Forward Timing Closure Recommendations during compilation.**
   - If you want to modify how Fast Forward compilation interprets specific I/O and block types, click **Assignments ➤ Settings ➤ Compiler Settings ➤ HyperFlex ➤ Advanced Settings**.

Figure 69. HyperFlex Settings


Run Fast Forward During Compilation
Settings - chiptrip_nf <@sj-slscf014>
Category:
Device/Board...
General
Files
Libraries
IP Settings
IP Catalog Search Locations
Design Templates
Operating Settings and Conditic
Voltage
Temperature
Compilation Process Settings
EDA Tool Settings
Compiler Settings
VHDL Input
Verilog HDL Input
Default Parameters
HyperFlex
TimeQuest Timing Analyzer
Assembler
SignalTap II Logic Analyzer
Logic Analyzer Interface
PowerPlay Power Analyzer Sett
Advanced Settings...
Fast Forward Advanced Options

HyperFlex

Specify HyperFlex optimization settings for the Compiler. These settings control the optimization algorithms that will be performed throughout design compilation.

Enabled Hyper-Retimer (Hyper Aware CAD and retiming optimization)

☑ Run Fast Forward Timing Closure Recommendations during compilation

Fast Forward Timing Closure Recommendations analyzes your design and provides recommendations for how to restructure your RTL to overcome performance bottlenecks.

To customize exploration settings globally, open the Advanced Settings dialog. For more information about Advanced settings, please open the Stratix 10 Hyper-Optimization Advisor.

3.1.3.1 Advanced HyperFlex Settings

The **Advanced HyperFlex Settings** control how Fast Forward Compilation analyzes and reports results for specific logical structures in a Stratix 10 design. To access the settings, click **Assignments ➤ Settings ➤ HyperFlex ➤ Advanced Settings**.

Table 8. Advanced HyperFlex Settings

Option	Description
Fast Forward Compile Asynchronous Clears	Specifies how Fast Forward analysis accounts for registers with asynchronous clear signals. The options are: • **Auto**—the Compiler assumes asynchronous clears are asynchronous until they limit timing performance during Fast Forward Compilation, at which point they are assumed removed • **Preserve**—the Compiler never assumes that it can remove or convert asynchronous clears for Fast Forward analysis.
Fast Forward Compile Fully Registered DSP Blocks	Specifies how Fast Forward analysis accounts for DSP blocks that limit performance. Enable this option to generate results as if all DSP blocks are fully registered.
Fast Forward Compile Fully Registered RAM Blocks	Specifies how Fast Forward analysis accounts for RAM blocks that limit performance. Enable this option to analyze the blocks as fully registered.
Fast Forward Compile Maximum Additional Pipeline Stages	Specifies the maximum number of pipeline stages that Fast Forward compilation explores.
Fast Forward Compile User Preserve Directives	Specifies how Fast Forward compilation accounts for restrictions from user-preserve directives.

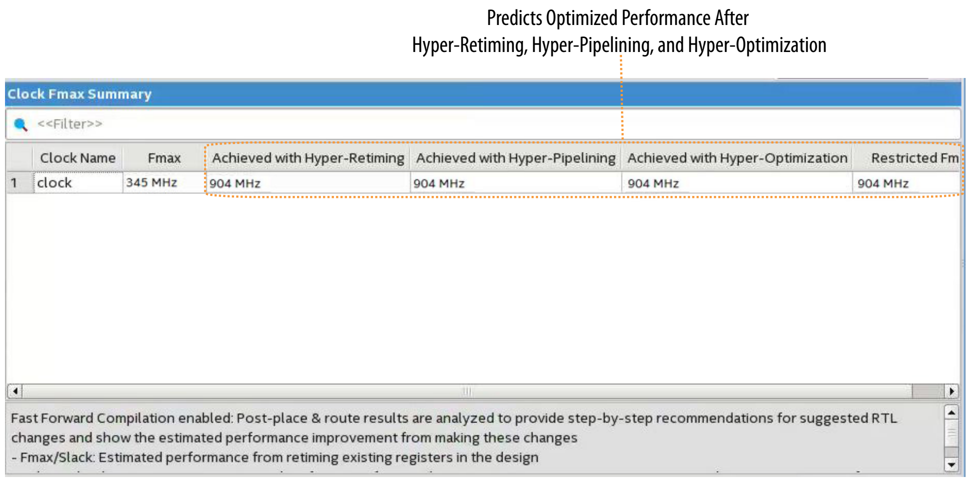
3.1.4 Step 4: Review Hyper-Retiming Results

After running Fast Forward Compilation, review the reports to determine which recommendations are appropriate and practical for your design functionality and performance goals.

3.1.4.1 Clock Fmax Summary Report

The **Clock Fmax Summary** reports the current f_{max} and potential performance achievable for each clock domain after Hyper-Retiming, Hyper-Pipelining, and Hyper-Optimization steps. Review the **Clock Fmax Summary** data to determine whether each potential performance improvement warrants further investigation and potential optimization of design RTL.

Figure 70. Current and Potential Performance in Clock Fmax Summary


Predicts Optimized Performance After
Hyper-Retiming, Hyper-Pipelining, and Hyper-Optimization
Clock Fmax Summary
<<Filter>>

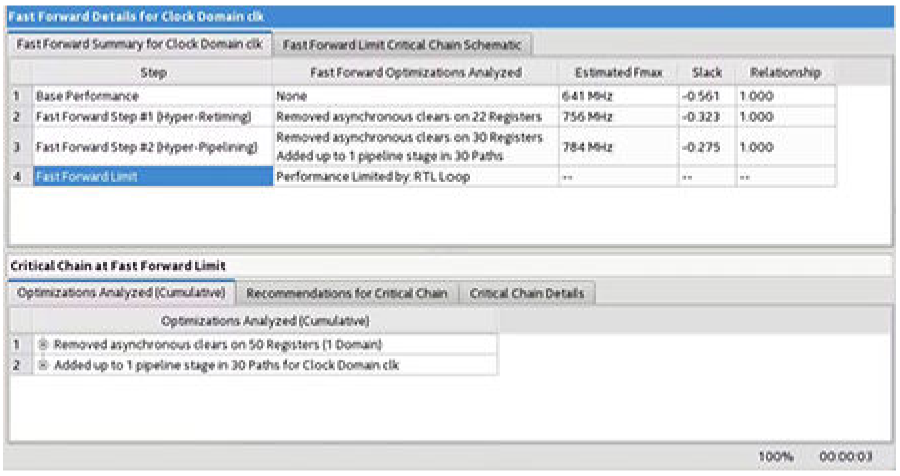
	Clock Name	Fmax	Achieved with Hyper-Retiming	Achieved with Hyper-Pipelining	Achieved with Hyper-Optimization	Restricted Fm
1	clock	345 MHz	904 MHz	904 MHz	904 MHz	904 MHz

Fast Forward Compilation enabled: Post-place & route results are analyzed to provide step-by-step recommendations for suggested RTL changes and show the estimated performance improvement from making these changes
- Fmax/Slack: Estimated performance from retiming existing registers in the design

3.1.4.2 Fast Forward Details Report

The **Fast Forward Details** report recommends the design modifications necessary to achieve Fast Forward compilation performance levels. Some recommendations may be functionally impossible or impractical for your design. Consider which recommendations you can implement in RTL to achieve similar performance improvement. Click on any optimization **Step** to view the implementation details and performance calculations for that step.

Figure 71. Fast-Forward Details Report



Fast Forward Details for Clock Domain clk

Fast Forward Summary for Clock Domain clk | Fast Forward Limit Critical Chain Schematic

	Step	Fast Forward Optimizations Analyzed	Estimated Fmax	Slack	Relationship
1	Base Performance	None	641 MHz	-0.561	1.000
2	Fast Forward Step #1 (Hyper-Retiming)	Removed asynchronous clears on 22 Registers	756 MHz	-0.323	1.000
3	Fast Forward Step #2 (Hyper-Pipelining)	Removed asynchronous clears on 30 Registers Added up to 1 pipeline stage in 30 Paths	784 MHz	-0.275	1.000
4	Fast Forward Limit	Performance Limited by: RTL Loop	--	--	--

Critical Chain at Fast Forward Limit

Optimizations Analyzed (Cumulative) | Recommendations for Critical Chain | Critical Chain Details

	Optimizations Analyzed (Cumulative)
1	⊞ Removed asynchronous clears on 50 Registers (1 Domain)
2	⊞ Added up to 1 pipeline stage in 30 Paths for Clock Domain clk

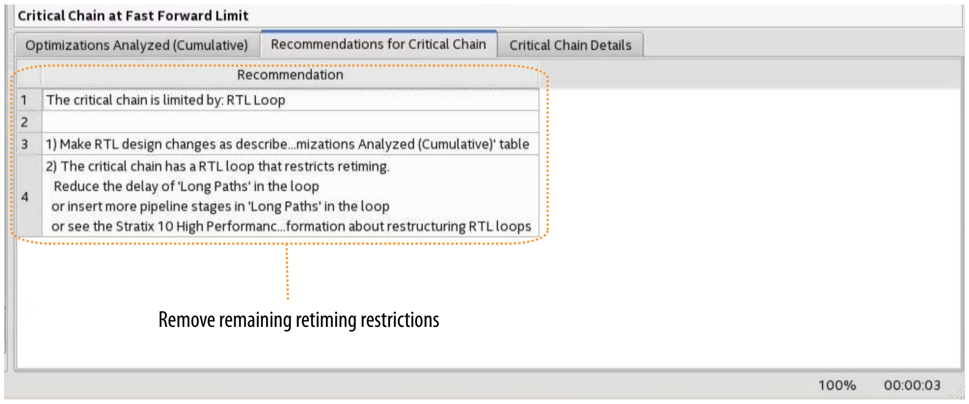
100% 00:00:03

Table 9. Fast Forward Details Report Data

Report Field	Description
Step	Displays the pre-optimized Base Performance f_{MAX}, the recommended Fast Forward optimization steps, and the Fast Forward Limit critical path that prevents further optimization.
Fast Forward Optimizations Analyzed	Summarizes the optimizations necessary to implement each optimization step.
Estimated Fmax	Specifies the potential f_{MAX} performance if you implement all Fast Forward optimization steps.
Optimizations Analyzed For Fast Forward Step	Lists design recommendations hierarchically for the selected **Step**. Click the text to expand the report and view the clock domain, the affected module, and the bus and bits that require modification.
Optimizations Analyzed (Cumulative)	Accumulated list of all design changes necessary to reach the selected **Step**.
Critical Chain at Fast Forward Limit	Displays information about any path that continues to limit Hyper-Retiming even after application of all Fast Forward steps. The critical chain is any path that limits further Hyper-Retiming. Click the **Fast Forward Limit** step to display this field.
Recommendations for Critical Chain	Lists register timing path associated with the retiming limitations. Right-click any path to **Locate Critical Chain in Fast Forward Viewer**.

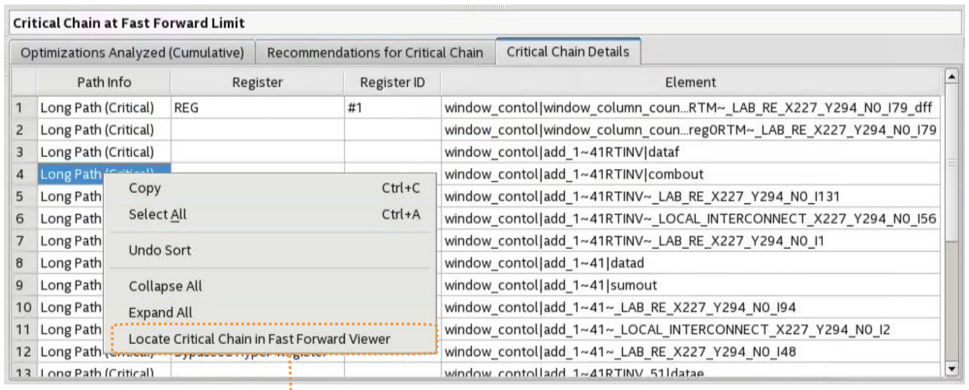
Right-click any path to locate to the critical chain in the Fast Forward Viewer. The Fast Forward Viewer displays a predictive representation of the complete design, after implementation of all Fast Forward recommendations.

Figure 72. Recommendations for Critical Chain


Critical Chain at Fast Forward Limit
Optimizations Analyzed (Cumulative)
Recommendations for Critical Chain
Critical Chain Details
Remove remaining retiming restrictions
100%
00:00:03

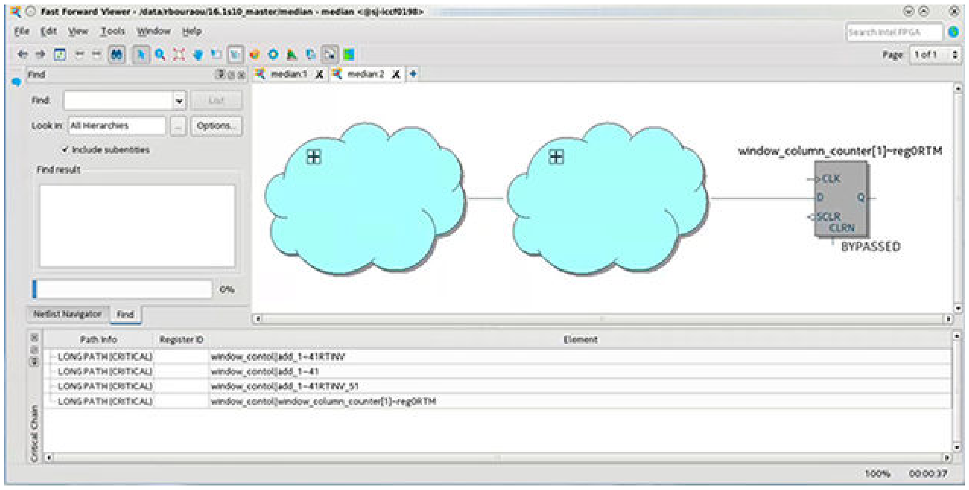
	Recommendation
1	The critical chain is limited by: RTL Loop
2	
3	1) Make RTL design changes as describe...mizations Analyzed (Cumulative)' table
4	2) The critical chain has a RTL loop that restricts retiming. Reduce the delay of 'Long Paths' in the loop or insert more pipeline stages in 'Long Paths' in the loop or see the Stratix 10 High Performanc...formation about restructuring RTL loops

Figure 73. Locate Critical Chain in Fast Forward Viewer


Critical Chain at Fast Forward Limit
Optimizations Analyzed (Cumulative)
Recommendations for Critical Chain
Critical Chain Details
Copy Ctrl+C
Select All Ctrl+A
Undo Sort
Collapse All
Expand All
Locate Critical Chain in Fast Forward Viewer
Locate Critical Chain in Fast Forward Viewer

	Path Info	Register	Register ID	Element
1	Long Path (Critical)	REG	#1	window_contol|window_column_coun...RTM~_LAB_RE_X227_Y294_N0_I79_dff
2	Long Path (Critical)			window_contol|window_column_coun...reg0RTM~_LAB_RE_X227_Y294_N0_I79
3	Long Path (Critical)			window_contol|add_1~41RTINV|dataf
4	Long Path (Critical)			window_contol|add_1~41RTINV|combout
5	Long Path			window_contol|add_1~41RTINV~_LAB_RE_X227_Y294_N0_I131
6	Long Path			window_contol|add_1~41RTINV~_LOCAL_INTERCONNECT_X227_Y294_N0_I56
7	Long Path			window_contol|add_1~41RTINV~_LAB_RE_X227_Y294_N0_I1
8	Long Path			window_contol|add_1~41|datad
9	Long Path			window_contol|add_1~41|sumout
10	Long Path			window_contol|add_1~41~_LAB_RE_X227_Y294_N0_I94
11	Long Path			window_contol|add_1~41~_LOCAL_INTERCONNECT_X227_Y294_N0_I2
12	Long Path (Critical)	Bypassed Hyper-Register		window_contol|add_1~41~_LAB_RE_X227_Y294_N0_I48
13	Long Path (Critical)			window_contol|add_1~41RTINV_51|datae

Figure 74. Fast Forward Viewer Shows Predictive Results



Path Info	Register ID	Element
LONG PATH (CRITICAL)		window_contol|add_1~41RTINV
LONG PATH (CRITICAL)		window_contol|add_1~41
LONG PATH (CRITICAL)		window_contol|add_1~41RTINV_51
LONG PATH (CRITICAL)		window_contol|window_column_counter[1]~reg0RTM

3.1.5 Step 5: Implement Fast Forward Recommendations

Implement the Fast Forward timing closure recommendations in your design RTL and rerun the **Retime** stage to realize the predictive performance gains.

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, you can stop design optimization after making those changes. However, if you require additional performance, implement more Fast Forward recommendations, such as any of the following techniques:

- Remove limitations of control logic, such as long feedback loops and state machines.
- Restructure logic to use functionally equivalent feed-forward or pre-compute paths, rather than long combinatorial feedback path.
- Reduce the delay of 'Long Paths' in the chain. Use standard timing closure techniques to reduce delay. Excessive combinational logic, sub-optimal placement, and routing congestion cause delay on paths.
- Insert more pipeline stages in 'Long Paths' in the chain. Long paths have the most delay between registers in the critical chain.
- Increase the delay (or add pipeline stages to 'Short Paths' in the chain).

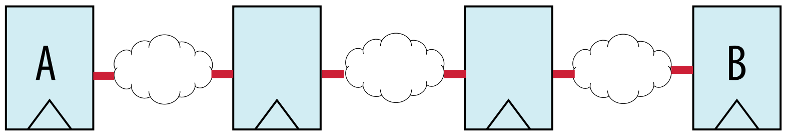
Explore performance and implement the RTL changes to your code until you reach the desired performance target.

3.1.5.1 Interpreting Critical Chain Reports

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

Figure 75. Sample Critical Chain

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



Fast Forward recommendations include:

- 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.

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

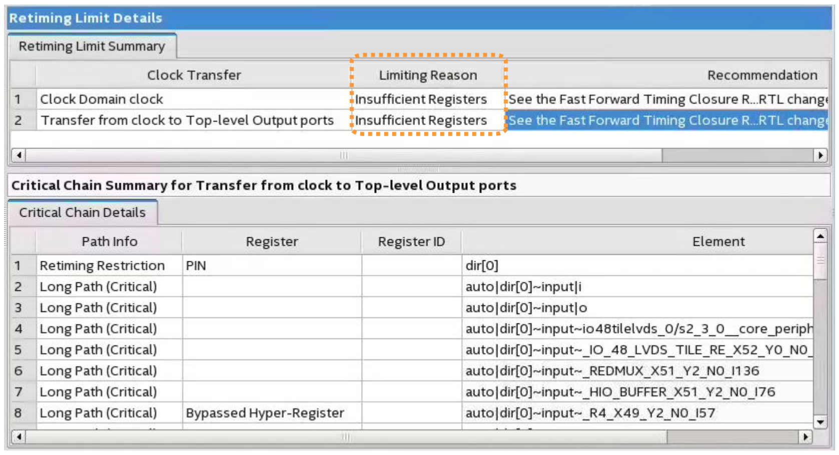
- Insufficient Registers
- Loop
- Short path/long path
- Path limit

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

3.1.5.1.1 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.

Figure 76. Insufficient Registers Reported as Limiting Reason



Retiming Limit Details

Retiming Limit Summary

	Clock Transfer	Limiting Reason	Recommendation
1	Clock Domain clock	Insufficient Registers	See the Fast Forward Timing Closure R...RTL chang
2	Transfer from clock to Top-level Output ports	Insufficient Registers	See the Fast Forward Timing Closure R...RTL chang

Critical Chain Summary for Transfer from clock to Top-level Output ports

Critical Chain Details

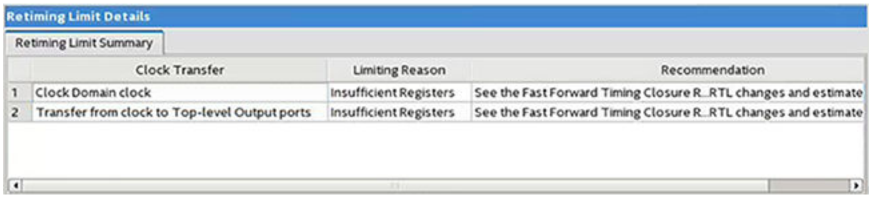
	Path Info	Register	Register ID	Element
1	Retiming Restriction	PIN		dir[0]
2	Long Path (Critical)			auto|dir[0]~input|i
3	Long Path (Critical)			auto|dir[0]~input|o
4	Long Path (Critical)			auto|dir[0]~input~io48tilelvds_0/s2_3_0__core_periph
5	Long Path (Critical)			auto|dir[0]~input~_IO_48_LVDS_TILE_RE_X52_Y0_N0_
6	Long Path (Critical)			auto|dir[0]~input~_REDMUX_X51_Y2_N0_I136
7	Long Path (Critical)			auto|dir[0]~input~_HIO_BUFFER_X51_Y2_N0_I76
8	Long Path (Critical)	Bypassed Hyper-Register		auto|dir[0]~input~_R4_X49_Y2_N0_I57

Insufficient Registers Example

The following screenshots show the relevant parts of the Retiming Limit Details 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 77. Retiming Limit Details



Retiming Limit Details

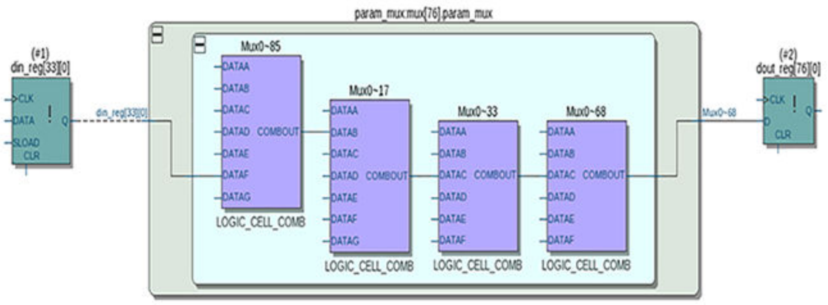
Retiming Limit Summary

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

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

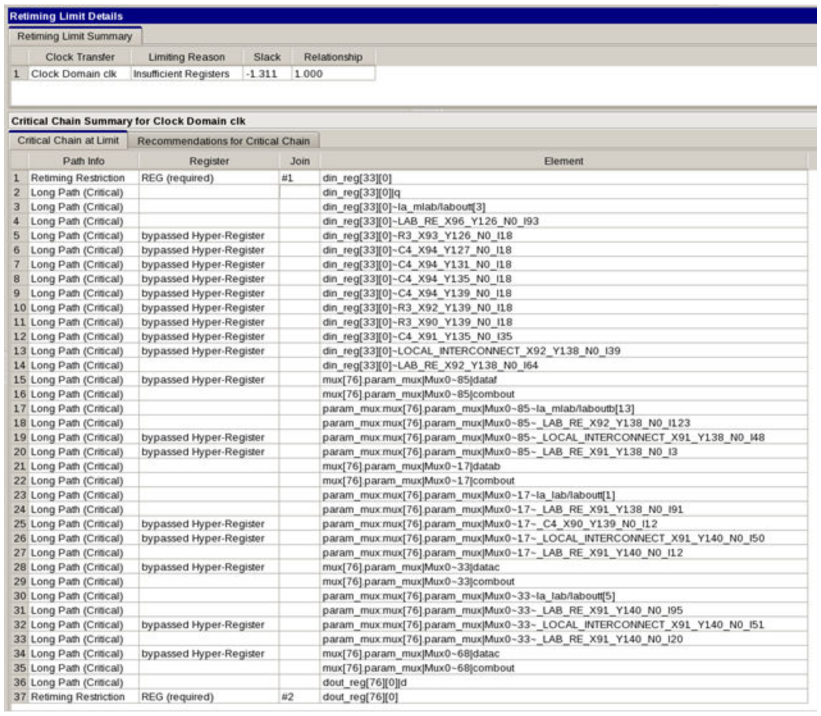
Figure 78. Critical Chain in Post-Fit Technology Map Viewer

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



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

Figure 79. Critical Chain with Insufficient Registers Reported During Hyper-Retiming



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]

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

Table 10. 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

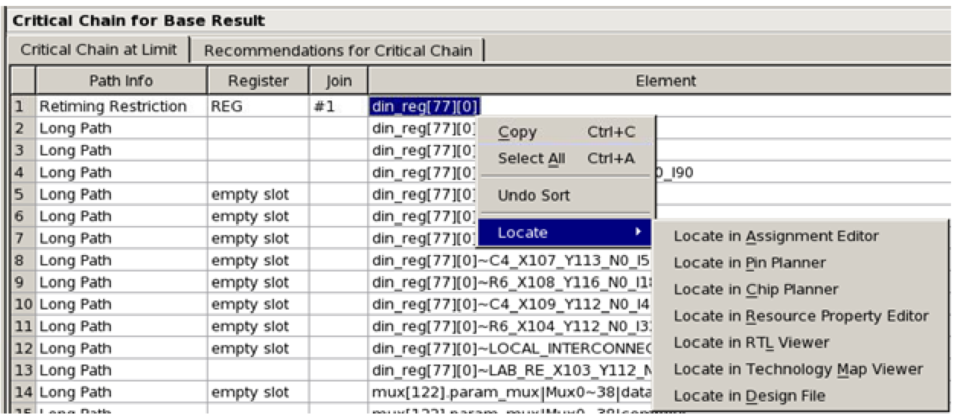
In the critical chain report in Figure 79 on page 71, there are 17 lines that list bypass Hyper-Register in the **Register** column. Bypassed Hyper-Register indicates the location of a Hyper-Register for use if there are more registers in the chain, or if there are no restrictions on the endpoints. If there are no restrictions on the endpoints, the Compiler can retime the endpoint registers, or retime other registers from outside the critical chain into the critical chain. If the RTL design contains more registers through the crossbar switch, there are more registers that 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 with the **Locate** option, as shown in the following figure.

Figure 80. Cross Probe from the Critical Chain Report



Critical Chain for Base Result

Critical Chain at Limit | Recommendations for Critical Chain

	Path Info	Register	Join	Element
1	Retiming Restriction	REG	#1	din_reg[77][0]
2	Long Path			din_reg[77][0]
3	Long Path			din_reg[77][0]
4	Long Path			din_reg[77][0] 0_I90
5	Long Path	empty slot		din_reg[77][0]
6	Long Path	empty slot		din_reg[77][0]
7	Long Path	empty slot		din_reg[77][0]
8	Long Path	empty slot		din_reg[77][0]~C4_X107_Y113_N0_I5
9	Long Path	empty slot		din_reg[77][0]~R6_X108_Y116_N0_I1
10	Long Path	empty slot		din_reg[77][0]~C4_X109_Y112_N0_I4
11	Long Path	empty slot		din_reg[77][0]~R6_X104_Y112_N0_I3
12	Long Path	empty slot		din_reg[77][0]~LOCAL_INTERCONNE
13	Long Path			din_reg[77][0]~LAB_RE_X103_Y112_N
14	Long Path	empty slot		mux[122].param_mux|Mux0~38|data

Related Links

- Short Path/Long Path on page 73
- Complex Critical Chains on page 82
- Hyper-Retiming (Facilitate Register Movement) on page 11

Optimizing Insufficient Registers

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

Related Links

- Hyper-Retiming (Facilitate Register Movement) on page 11
- Hyper-Pipelining (Add Pipeline Registers) on page 25

Critical Chains with Dual Clock Memories

Hyper-Retiming 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 after Fast Forward compile.

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

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

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

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

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

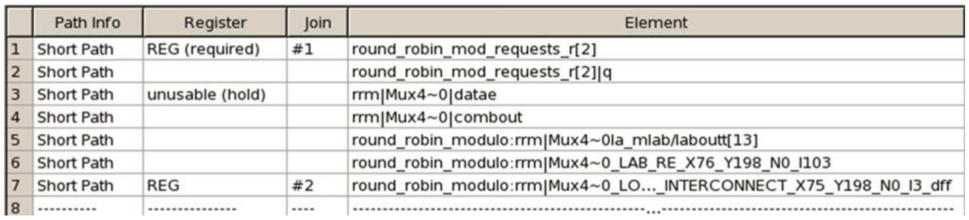
Short path/long path critical chains are a new optimization focus associated with post-fit retiming. In conventional retiming, the structure of the netlist can be changed during synthesis or placement and routing. However, during Hyper-Retiming, short path/long path can occur because the netlist structure,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

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, 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 are in use. 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 are in use. The `_dff` represents the D flop-flop. No other Hyper-Register locations are available for use in that chain segment. Available Hyper-Register locations 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 81. Critical Chain Short Path Segment with no Available Hyper-Register Locations

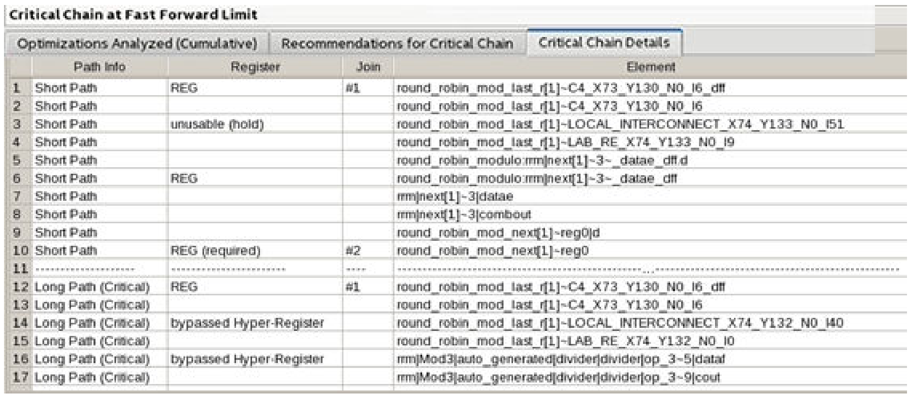


	Path Info	Register	Join	Element
1	Short Path	REG (required)	#1	round_robin_mod_requests_r[2]
2	Short Path			round_robin_mod_requests_r[2]|q
3	Short Path	unusable (hold)		rrm|Mux4~0|datae
4	Short Path			rrm|Mux4~0|combout
5	Short Path			round_robin_modulo:rrm|Mux4~0la_mlab/laboutt[13]
6	Short Path			round_robin_modulo:rrm|Mux4~0_LAB_RE_X76_Y198_N0_I103
7	Short Path	REG	#2	round_robin_modulo:rrm|Mux4~0_LO..._INTERCONNECT_X75_Y198_N0_I3_dff
8	----------	---------------	----	--...--

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 82. Critical Chain with Short Path/Long Path



Critical Chain at Fast Forward Limit

Optimizations Analyzed (Cumulative) | Recommendations for Critical Chain | Critical Chain Details

	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:rrm|next[1]~3~_datae_dff.d
6	Short Path	REG		round_robin_modulo:rrm|next[1]~3~_datae_dff
7	Short Path			rrm|next[1]~3|datae
8	Short Path			rrm|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		rrm|Mod3|auto_generated|divider|divider|op_3~5|dataf
17	Long Path (Critical)			rrm|Mod3|auto_generated|divider|divider|op_3~9|cout

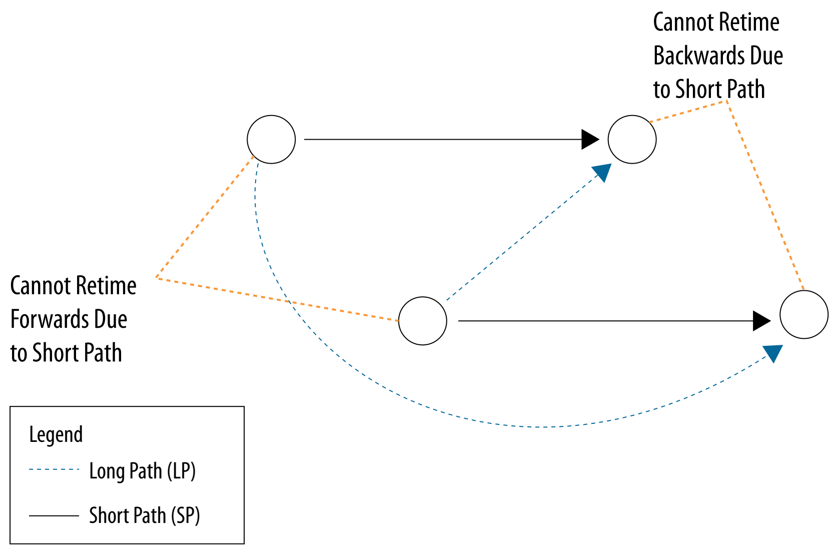
Optimizing Short Path/Long Path

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

Add Registers

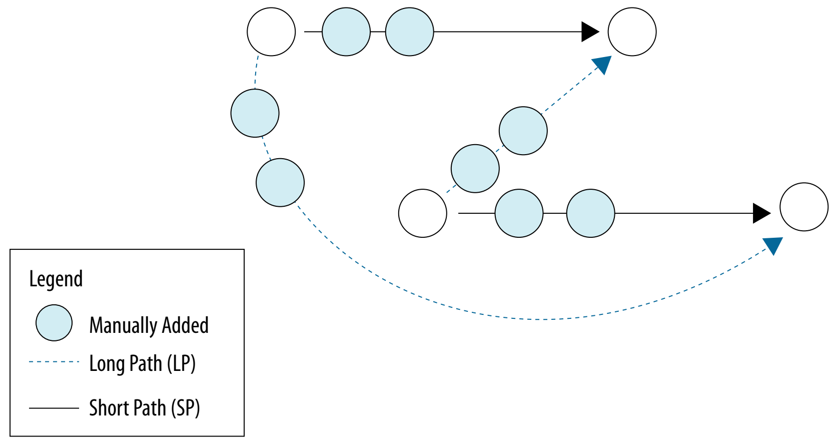
Manually adding registers on both the short and long paths can be helpful if you can accommodate the extra latency in the critical chain.

Figure 83. Critical Chain with Alternating Short Path/Long Path


Cannot Retime Backwards Due to Short Path
Cannot Retime Forwards Due to Short Path
Legend
Long Path (LP)
Short Path (SP)

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

Figure 84. 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 85. Critical Chain with Alternating Short Path/Long Path

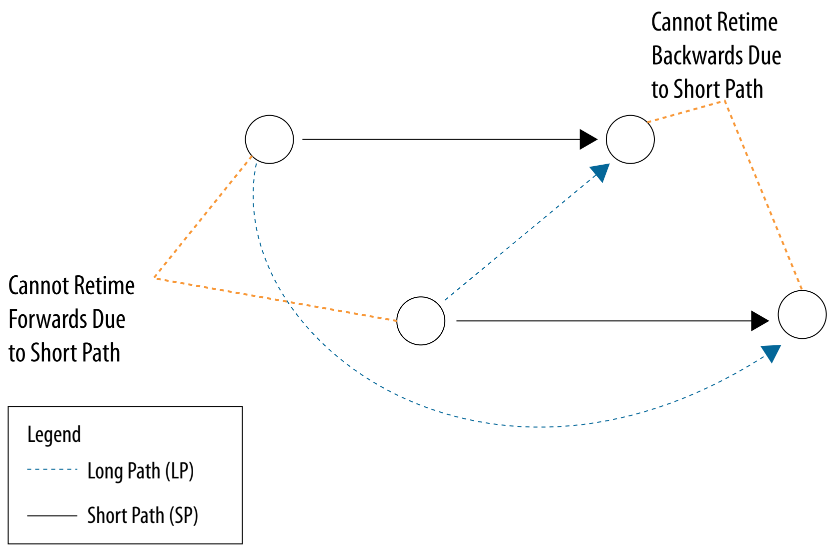
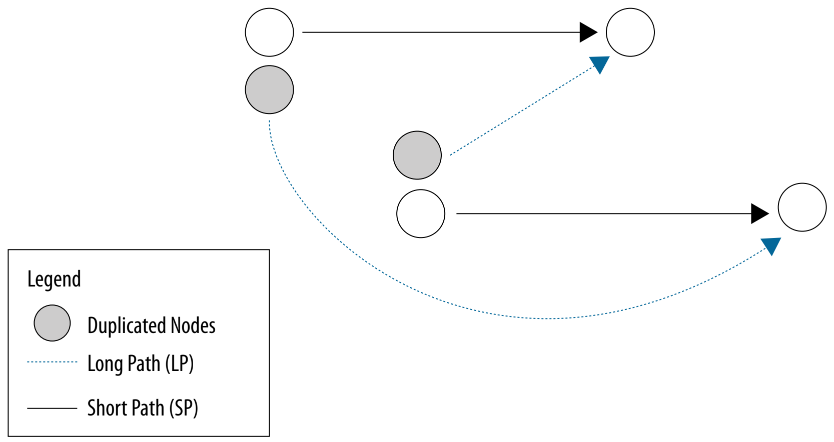

Cannot Retime Backwards Due to Short Path
Cannot Retime Forwards Due to Short Path
Legend
Long Path (LP)
Short Path (SP)

Figure 86. Short Path/Long Path with Two Duplicate Nodes



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

You can apply a maximum fan-out synthesis directive to the common source registers. Use a value of one, because a value greater than one 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

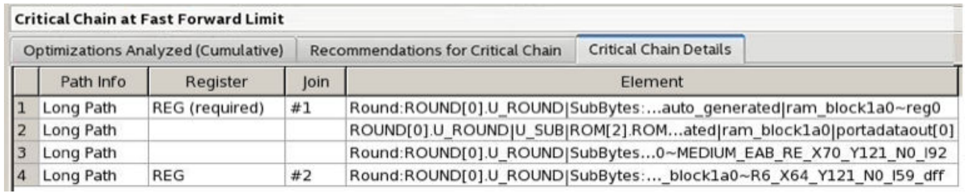
Sometimes, the long path can be in the data plane, and the short path can be in the control plane. If you add registers to the data path, change the control logic. This can be a time-consuming process. In cases where the control logic is based on the number of clock cycles in the data path, you can add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

3.1.5.1.3 Fast Forward Limit

The critical chain has the limiting reason of Path Limit, when there are no more Hyper-Register locations available on the critical path, and the design cannot run any faster or 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 87. Critical Chain at Fast Forward Limit



Critical Chain at Fast Forward Limit

Optimizations Analyzed (Cumulative) | Recommendations for Critical Chain | Critical Chain Details

	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 Hyper-Retiming 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.

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 Hyper-Retiming 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.

Optimizing Path Limit

Evaluate the Fast Forward recommendations. 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 for Hyper-Retiming, 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 Compiler 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.

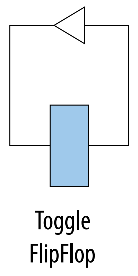
Related Links

- Recommended HDL Coding Styles
- True Dual-Port Memory on page 47

3.1.5.1.4 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 88. Simple Loop



When the critical chain is a feedback loop, the number of registers in a loop cannot be changed by register retiming 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 89. 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.

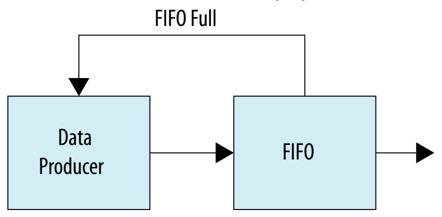


Figure 90. Counter and Accumulator Loop

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.

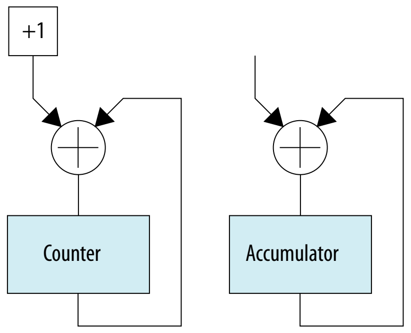


Figure 91. State Machine Loop

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

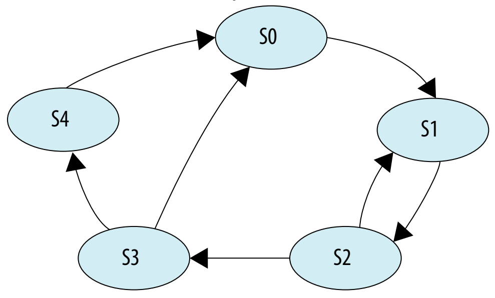
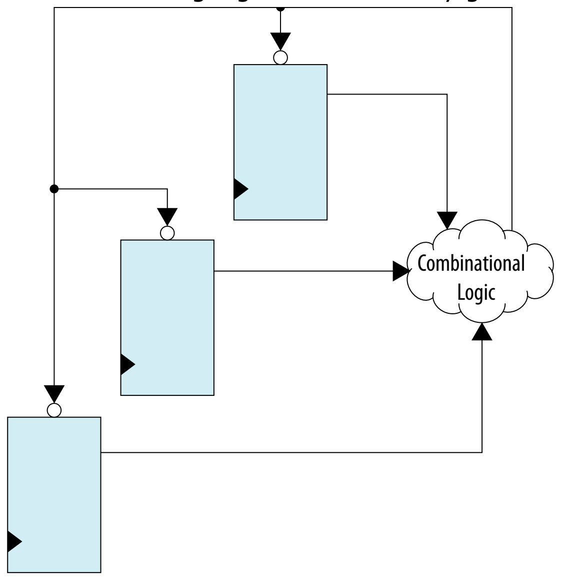


Figure 92. Reset Circuit Loop

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

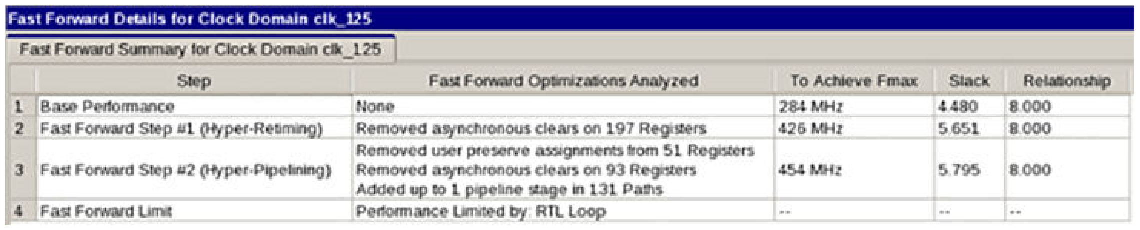


Use loops to save area through hardware re-use. Components that are re-used over several cycles typically involve loops. For example reused components include CRC calculations, filters, floating point dividers, and word aligners. Loops are also used in closed loop feedback designs such as IIR filters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

Example of Critical Chain with Loops as the Limiting Reason

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

Figure 93. Fast Forward Details Report showing Limiting Reason for Hyper-Optimization is a Loop



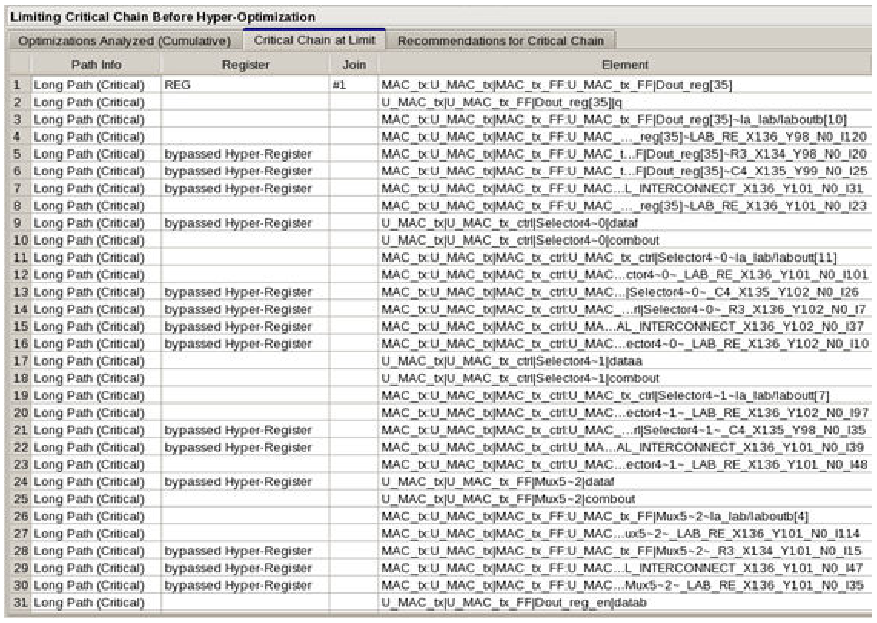
Fast Forward Details for Clock Domain clk_125

Fast Forward Summary for Clock Domain clk_125

	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 94. Critical Chain with Loop as Reported During Hyper-Retiming



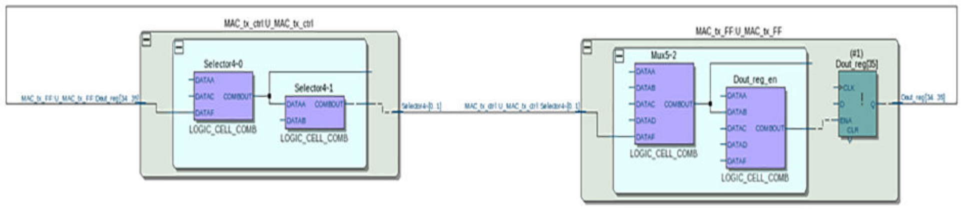
Limiting Critical Chain Before Hyper-Optimization

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/labout[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/labout[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 95. 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 15. Source Code for Critical Chain

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

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

always @ (posedge Clk or posedge Reset)
            if (Reset)
```

```
                        RetryCnt   <=0;
                else if (Current_state==StateSwitchNext)
                        RetryCnt   <=0;
                else if (Current_state==StateJam&&Next_state==StateBackOff)
                        RetryCnt  <=RetryCnt +1;
```

3.1.5.2 Details about Critical Chain Reports

The topics below apply to any type of critical chain.

3.1.5.2.1 One Critical Chain per Clock Domain

Hyper-Retiming 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, Hyper-Retiming reports show one critical chain per clock domain per Fast Forward optimization step. Hyper-Retiming does not report multiple critical chains per clock domain, because only one chain is the critical chain.

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

Related Links

Critical Chains in Related Clock Groups on page 82

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

3.1.5.2.3 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 Fitter reports. Join points are listed at the ends of segments in a critical chain, and they indicate where segments diverge or converge. Join points indicate connectivity between chain segments when the chain is listed in a line-oriented text-based report. Join points correspond to elements in your circuit, and show how they are connected to other elements to form a critical chain.

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

Table 11. Sample Critical Chain

Path Info	Register	Join	Element
	REG	#1	a
			b
	REG	#2	c

continued...

Path Info	Register	Join	Element
------------	------------	------------	------------
	REG	#3	d
			e
	REG	#2	c
------------	------------	------------	------------
	REG	#3	d
			f
	REG	#4	g
------------	------------	------------	------------
			g
			h
			a

Figure 96. 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.

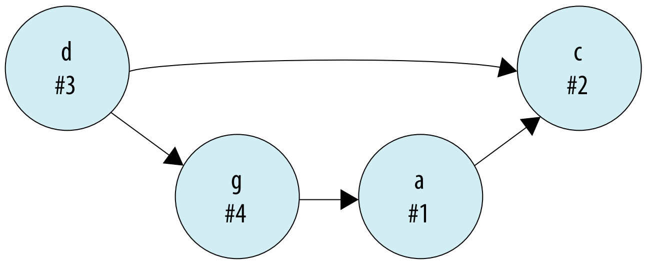
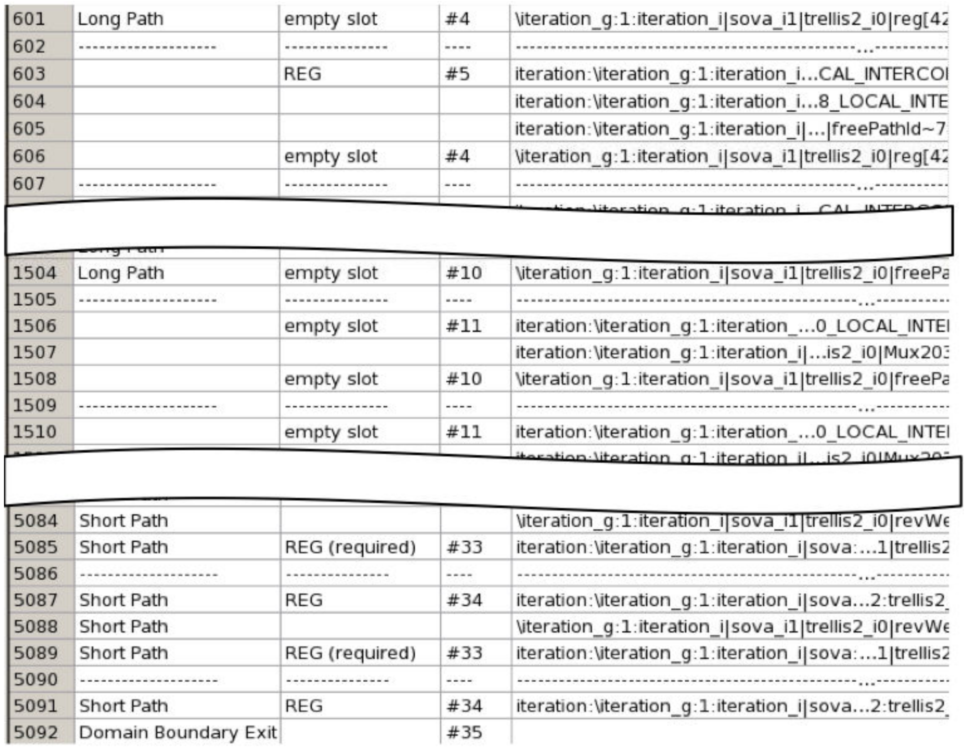


Figure 97. Complex Critical Chain

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.

3.1.5.2.4 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.

3.1.5.2.5 Domain Boundary Entry and Domain Boundary Exit

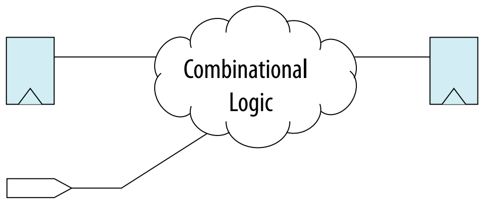
The **Path Info** column lists the Domain Boundary Entry or Domain Boundary Exit for a critical chain. Domain boundary entry and domain boundary exit refer to paths that are unconstrained, paths between asynchronous clock domains, or between a clock domain and top-level device input-outputs. Domain boundary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

A domain boundary entry refers to a point in the design topology, at a clock domain boundary, where Hyper-Retiming can insert register stages (where latency can enter the clock domain) if Hyper-Pipelining is enabled. The concept of a domain boundary entry is independent of the dataflow direction. Hyper-Retiming can insert register stages at the input of a module, and perform forward retiming pushes, 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 Hyper-Retiming can remove register stages and the latency can exit the clock domain, if Hyper-Pipelining is enabled. Removing a register may be counter intuitive. However, it may be necessary to retain functional correctness, depending on other optimizations performed by Hyper-Retiming.

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 98. Domain Boundary with Unregistered Input/Output



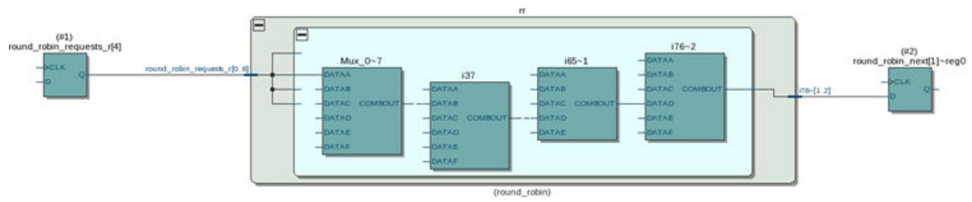
The register-to-register path might be shown as a critical chain segment with a domain boundary entry or a domain boundary exit, depending on how it restricted Hyper-Retiming. The unregistered input prevents the Hyper-Retiming from inserting register stages at the domain boundary, because the input is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

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

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

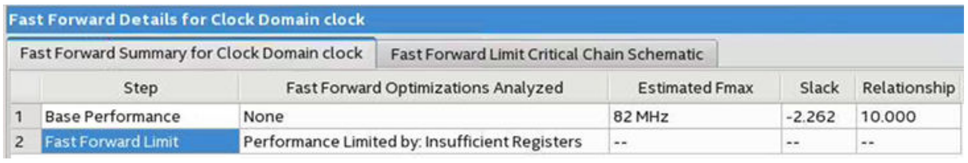
The following critical chain begins and ends with domain boundary entry. The domain boundary entries are the input and output registers connecting to top-level device I/Os. The input register is `round_robin_mod_last_r` and the output register is `round_robin_mod_next`.

Figure 99. Critical Chain Schematic with Domain Boundary



The limiting reason for the base compile is Insufficient Registers.

Figure 100. Fast Forward Compile Report with Insufficient Registers



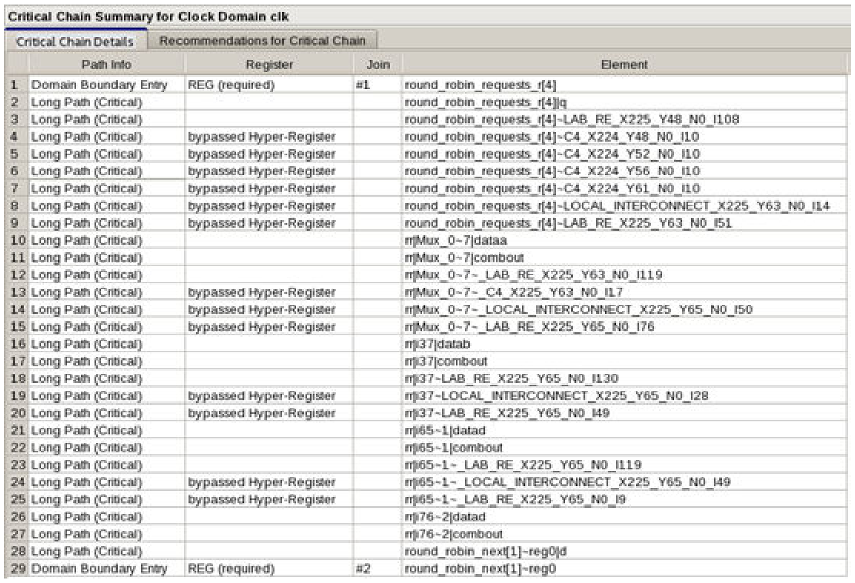
Fast Forward Details for Clock Domain clock

Fast Forward Summary for Clock Domain clock | Fast Forward Limit Critical Chain Schematic

	Step	Fast Forward Optimizations Analyzed	Estimated Fmax	Slack	Relationship
1	Base Performance	None	82 MHz	-2.262	10.000
2	Fast Forward Limit	Performance Limited by: Insufficient Registers	--	--	--

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

Figure 101. Critical Chain with Domain Boundary Entry



Critical Chain Summary for Clock Domain clk

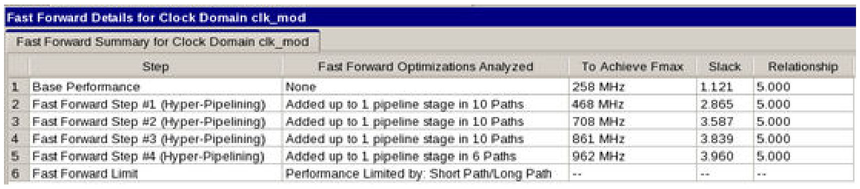
Critical Chain Details | Recommendations for Critical Chain

	Path Info	Register	Join	Element
1	Domain Boundary Entry	REG (required)	#1	round_robin_requests_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 102. 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	--	--	--

3.1.5.2.6 Critical Chains with Dual Clock Memories

Hyper-Retiming 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 after Fast Forward compile.

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

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

Related Links

Appendix A: Parameterizable Pipeline Modules on page 115

3.1.5.2.7 Critical Chain Bits and Buses

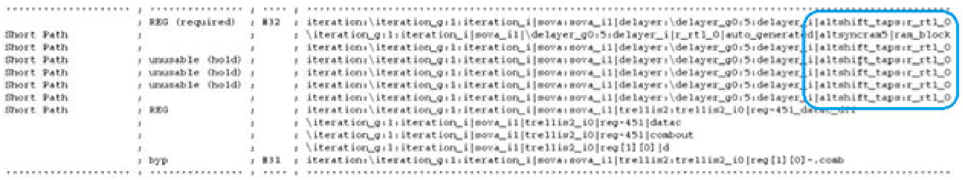
The critical chain of a design commonly includes registers that are single bits in a wider bus or register bank. When you analyze such a critical chain, focus on the bus as a whole, instead of analyzing the structure related to the single bit. For example, a critical chain that refers to bit 10 in a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

3.1.5.2.8 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 103. Critical Chain Report with Delay Line



```
-------------------- ; --------------- ; ---- ; ---------------------------------------------------------------------------------------------------------------
                     ; REG (required)  ; #32  ; iteration:\iteration_g:1:iteration_i|mova:mova_i1|delayer:\delayer_g0:5:delayer_i|altshift_taps:r_rtl_0
Short Path           ;                 ;      ; \iteration_g:1:iteration_i|mova_i1|\delayer_g0:5:delayer_i|r_rtl_0|auto_generated|altsyncram5|ram_block
Short Path           ;                 ;      ; iteration:\iteration_g:1:iteration_i|mova:mova_i1|delayer:\delayer_g0:5:delayer_i|altshift_taps:r_rtl_0
Short Path           ; unusable (hold) ;      ; iteration:\iteration_g:1:iteration_i|mova:mova_i1|delayer:\delayer_g0:5:delayer_i|altshift_taps:r_rtl_0
Short Path           ; unusable (hold) ;      ; iteration:\iteration_g:1:iteration_i|mova:mova_i1|delayer:\delayer_g0:5:delayer_i|altshift_taps:r_rtl_0
Short Path           ; unusable (hold) ;      ; iteration:\iteration_g:1:iteration_i|mova:mova_i1|delayer:\delayer_g0:5:delayer_i|altshift_taps:r_rtl_0
Short Path           ;                 ;      ; iteration:\iteration_g:1:iteration_i|mova:mova_i1|delayer:\delayer_g0:5:delayer_i|altshift_taps:r_rtl_0
Short Path           ; REG             ;      ; iteration:\iteration_g:1:iteration_i|mova:mova_i1|trellis2:trellis2_i0|reg~451_datac_ff
                     ;                 ;      ; \iteration_g:1:iteration_i|mova_i1|trellis2_i0|reg~451|datac
                     ;                 ;      ; \iteration_g:1:iteration_i|mova_i1|trellis2_i0|reg~451|combout
                     ;                 ;      ; \iteration_g:1:iteration_i|mova_i1|trellis2_i0|reg[1][0]|d
                     ; byp             ; #31  ; iteration:\iteration_g:1:iteration_i|mova:mova_i1|trellis2:trellis2_i0|reg[1][0]~.comb
-------------------- ; --------------- ; ---- ; ---------------------------------------------------------------------------------------------------------------
```

The Fitters places the chain of registers so close together that the hold time cannot be met if the Fitter 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.

3.1.5.3 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. Retiming restrictions exist because of hardware characteristics, software behavior, or are inherent to the design.

Table 12. 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 register retiming does not affect design functionality.
Register has an asynchronous clear	Hyper-Registers support only data and clock inputs. Hyper-Registers do not have control signals such as asynchronous clears, presets, or enables. 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 to any design that uses asynchronous resets. Focus on reducing the number of registers that are reset with an asynchronous clear.
Register has don't touch or preserve attributes	The Compiler does not retime registers with these attributes. If you use the preserve attribute to manage register duplication for high fan-out signals, try removing the preserve attribute. The Compiler may be able to retime the high fan-out register along each of the routing paths to its destinations. Alternatively, use the dont_merge attribute. The Compiler retimes registers in ALMs, DDIOs, single port RAMs, and DSP blocks.
Register is a clock source	This design condition is uncommon, especially for performance-critical parts of a design. If this retiming restriction prevents you from achieving the required performance, consider whether a PLL can generate the clock, rather than a register.

continued...

Design Condition	Hyper-Register Support
Register is a partition boundary	This condition is inherent to any design that uses design partitions. If this retiming restriction prevents you from achieving the required performance, add additional registers inside the partition boundary for Hyper-Retiming.
Register is a block type modified by an ECO operation	This restriction is uncommon. Avoid the restriction by making the functional change in the design source and recompiling, rather than performing an ECO.
Register location is an unknown block	This restriction is uncommon. You can often work around this condition by adding extra registers adjacent to the specified block type.
Register is described in the RTL as a latch	Hyper-Registers cannot implement latches. 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 Hyper-Retiming.
Combinational node is fed by a special source	This condition is uncommon, especially for performance-critical parts of a design.
Register is driven by a locally routed clock	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 Compiler 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 Compiler does not retime registers that are detected or marked as part of a synchronizer chain. Add more pipeline stages at the clock domain boundary adjacent to the synchronizer chain to provide flexibility for the 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 one frequency, and fans out to registers running at another frequency. The Compiler does not retime registers at cross-clock boundaries. Consider adding additional pipeline stages at one side of the clock domain boundary, or the other, to provide flexibility for retiming.

Related Links

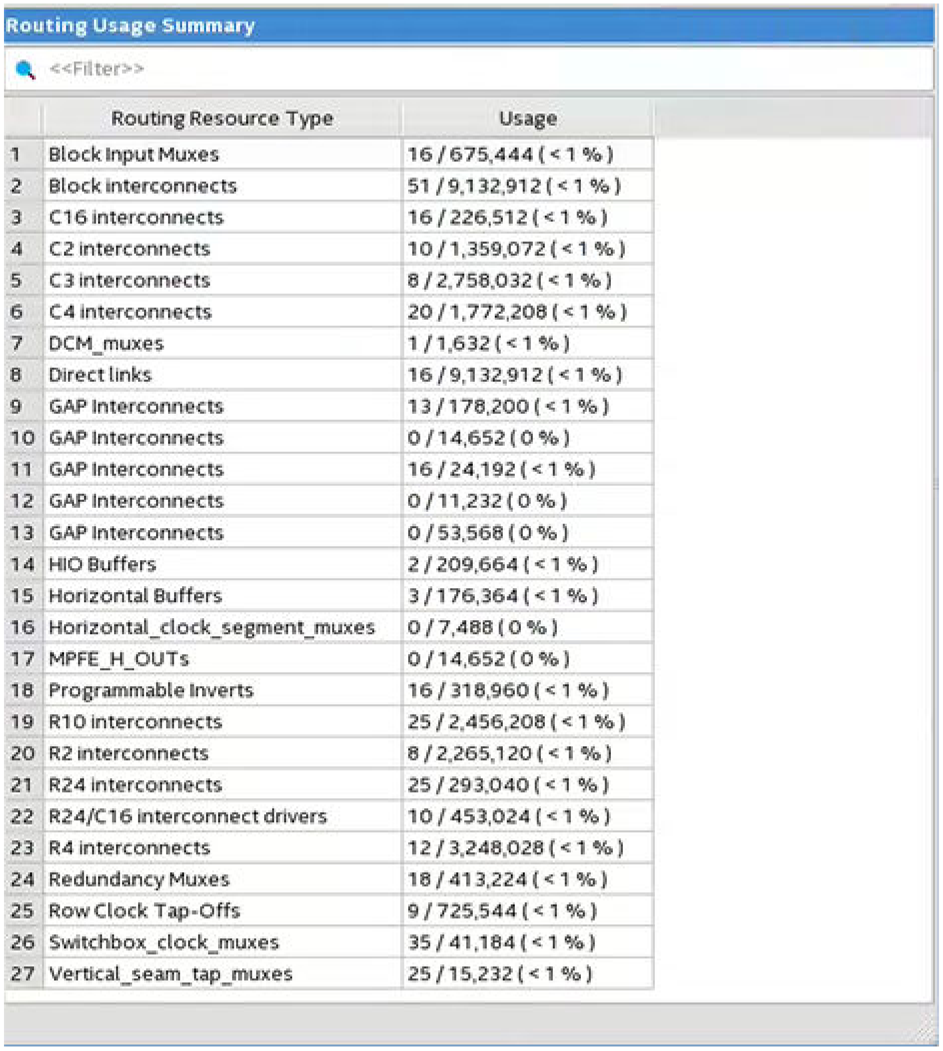
Timing Constraint Considerations on page 18

3.1.5.4 Finalize Stage Reports

The Finalize stage reports describe final placement and routing operations, including:

- Delay chain summary information
- Post-route hold fix-up data. The Compiler may report hold violations for short paths following the Retime stage. The Fitter identifies and corrects the short paths with hold violations during the **Fitter (Finalize)** stage by adding routing wire along the paths.

Figure 104. Finalize Stage Reports



Routing Usage Summary

<<Filter>>

	Routing Resource Type	Usage
1	Block Input Muxes	16 / 675,444 (< 1 %)
2	Block interconnects	51 / 9,132,912 (< 1 %)
3	C16 interconnects	16 / 226,512 (< 1 %)
4	C2 interconnects	10 / 1,359,072 (< 1 %)
5	C3 interconnects	8 / 2,758,032 (< 1 %)
6	C4 interconnects	20 / 1,772,208 (< 1 %)
7	DCM_muxes	1 / 1,632 (< 1 %)
8	Direct links	16 / 9,132,912 (< 1 %)
9	GAP Interconnects	13 / 178,200 (< 1 %)
10	GAP Interconnects	0 / 14,652 (0 %)
11	GAP Interconnects	16 / 24,192 (< 1 %)
12	GAP Interconnects	0 / 11,232 (0 %)
13	GAP Interconnects	0 / 53,568 (0 %)
14	HIO Buffers	2 / 209,664 (< 1 %)
15	Horizontal Buffers	3 / 176,364 (< 1 %)
16	Horizontal_clock_segment_muxes	0 / 7,488 (0 %)
17	MPFE_H_OUTs	0 / 14,652 (0 %)
18	Programmable Inverts	16 / 318,960 (< 1 %)
19	R10 interconnects	25 / 2,456,208 (< 1 %)
20	R2 interconnects	8 / 2,265,120 (< 1 %)
21	R24 interconnects	25 / 293,040 (< 1 %)
22	R24/C16 interconnect drivers	10 / 453,024 (< 1 %)
23	R4 interconnects	12 / 3,248,028 (< 1 %)
24	Redundancy Muxes	18 / 413,224 (< 1 %)
25	Row Clock Tap-Offs	9 / 725,544 (< 1 %)
26	Switchbox_clock_muxes	35 / 41,184 (< 1 %)
27	Vertical_seam_tap_muxes	25 / 15,232 (< 1 %)

4 HyperFlex Porting Guidelines

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

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

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

4.1 Design Migration and Performance Exploration

Migrate your Stratix V or Arria 10 design to a Stratix 10 device and evaluate performance improvement. Migrating a design for Stratix 10 devices *requires* only minor changes. However, additional *non-required* changes can help achieve dramatic performance improvements. This increased speed can help you close timing, or provide flexibility to add additional functionality to your design.

Required design changes as similar to any device upgrade. These changes include updating PLLs, high-speed I/O pins, and other device resources. The Stratix 10 version of these components have the same general functionality. However, the Stratix 10 components include features to enable higher operational speeds:

- 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.

The high level steps in the migration process are:

1. Select for migration a lower-level block in the design, without any specialized IP.
2. Black-box any special IP component and only keep components which are required for the current level you have selected. Only keep the following key blocks for core performance evaluation:

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

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

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

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:

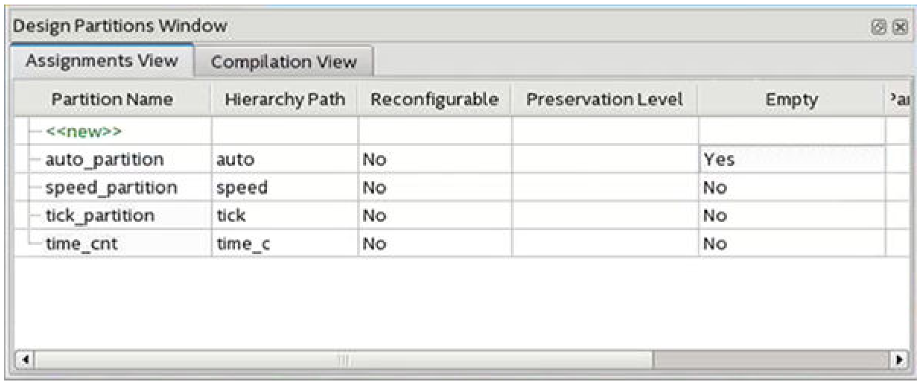
```
-- k-bit 2-to-1 multiplexer
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

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

To create a new partition:

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

Figure 105. Create New Empty Partition



Partition Name	Hierarchy Path	Reconfigurable	Preservation Level	Empty	ʾaı
<<new>>					
auto_partition	auto	No		Yes	
speed_partition	speed	No		No	
tick_partition	tick	No		No	
time_cnt	time_c	No		No	

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. Identify any clock existing a black-boxed module. If this occurs in your design, recreate this clock. Failure to recreate the clock marks any register downstream as unclocked. This changes the logic function of your design, because registers that do not receive a clock could be removed by synthesis. 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 no action required.
    - Yes: create a clock on the output pin of that module matching the definition in the `.sdc`.
- There is no clock definition in that module:
  - Is there a clock feedthrough path in that module?
    - No: there is no action required.
    - Yes: create a new clock on the feedthrough output pin of the module.

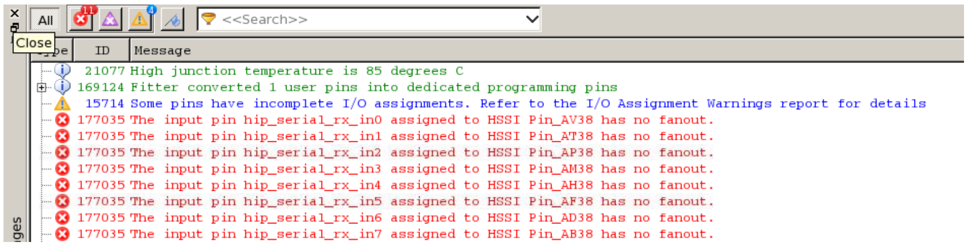
4.1.4 Pin Assignments

Black-boxing logic can cause some pin assignment issues. Use the following guidelines to resolve pin assignments.

- Reassign high-speed communication input pins

The 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. Verify and change as needed the I/O bank.

Figure 106. High-speed Pin Error Messages

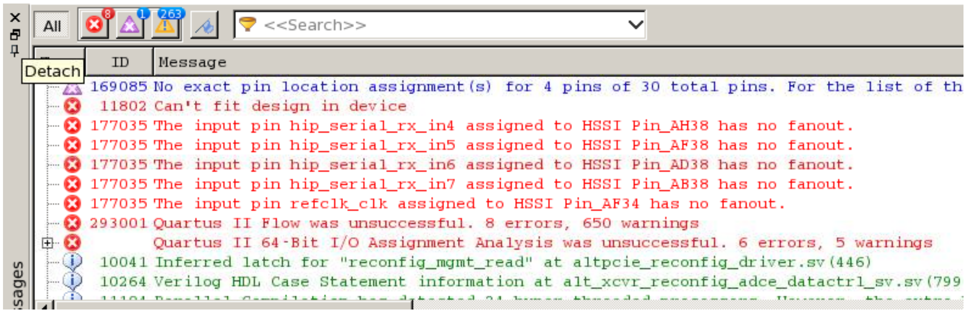


```
21077 High junction temperature is 85 degrees C
169124 Fitter converted 1 user pins into dedicated programming pins
15714 Some pins have incomplete I/O assignments. Refer to the I/O Assignment Warnings report for details
177035 The input pin hip_serial_rx_in0 assigned to HSSI Pin_AV38 has no fanout.
177035 The input pin hip_serial_rx_in1 assigned to HSSI Pin_AT38 has no fanout.
177035 The input pin hip_serial_rx_in2 assigned to HSSI Pin_AP38 has no fanout.
177035 The input pin hip_serial_rx_in3 assigned to HSSI Pin_AM38 has no fanout.
177035 The input pin hip_serial_rx_in4 assigned to HSSI Pin_AH38 has no fanout.
177035 The input pin hip_serial_rx_in5 assigned to HSSI Pin_AF38 has no fanout.
177035 The input pin hip_serial_rx_in6 assigned to HSSI Pin_AD38 has no fanout.
177035 The input pin hip_serial_rx_in7 assigned to HSSI Pin_AB38 has no fanout.
```

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



```
169085 No exact pin location assignment(s) for 4 pins of 30 total pins. For the list of th
 11802 Can't fit design in device
177035 The input pin hip_serial_rx_in4 assigned to HSSI Pin_AH38 has no fanout.
177035 The input pin hip_serial_rx_in5 assigned to HSSI Pin_AF38 has no fanout.
177035 The input pin hip_serial_rx_in6 assigned to HSSI Pin_AD38 has no fanout.
177035 The input pin hip_serial_rx_in7 assigned to HSSI Pin_AB38 has no fanout.
177035 The input pin refclk_clk assigned to HSSI Pin_AF34 has no fanout.
293001 Quartus II Flow was unsuccessful. 8 errors, 650 warnings
       Quartus II 64-Bit I/O Assignment Analysis was unsuccessful. 6 errors, 5 warnings
 10041 Inferred latch for "reconfig_mgmt_read" at altpcie_reconfig_driver.sv(446)
 10264 Verilog HDL Case Statement information at alt_xcvr_reconfig_adce_datactrl_sv.sv(799
```

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

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

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

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

4.2 Top-Level Design Considerations

I/O constraints

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

- `set_input_delay`
- `set_output_delay`

These constraints model 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, provide a replacement signal by direct connection to an input pin of your design. This configuration may affect the retiming capabilities in Stratix 10 architectures. Add a couple of pipeline stages to your reset signal. This helps the Compiler to optimize between the reset input and the first level of registers.

Special Blocks

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

- DSPs: Watch the pipelining depth. More pipeline stages results in a faster design. If 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 by registering the logic memory twice:
  - Once inside the M20K block directly
  - Once in the fabric, at the pins of the block

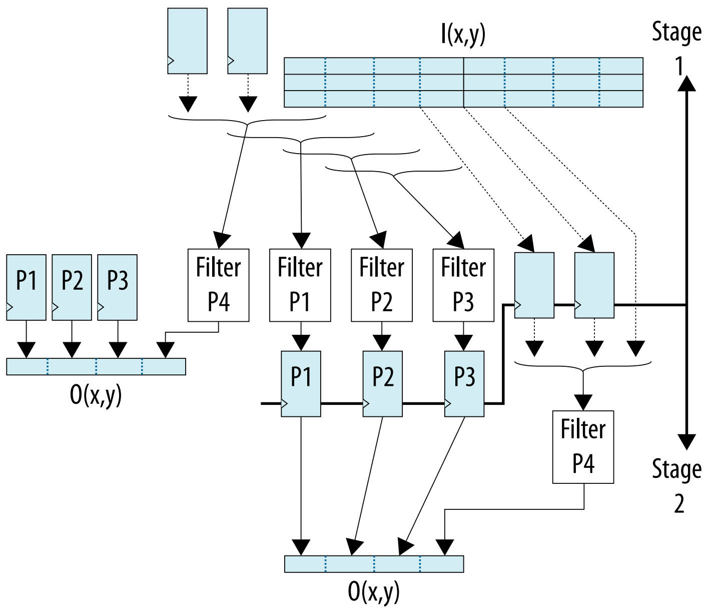
Register the Block

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.

5 Design Example Walk-Through

The Median Filter design example illustrates how Hyper-Retiming and Fast-Forward compilation improve performance in a real-world design. This walk-through describes project setup, design compilation, interpreting results, and optimizing RTL.

Figure 108. Median Filter Operational Diagram



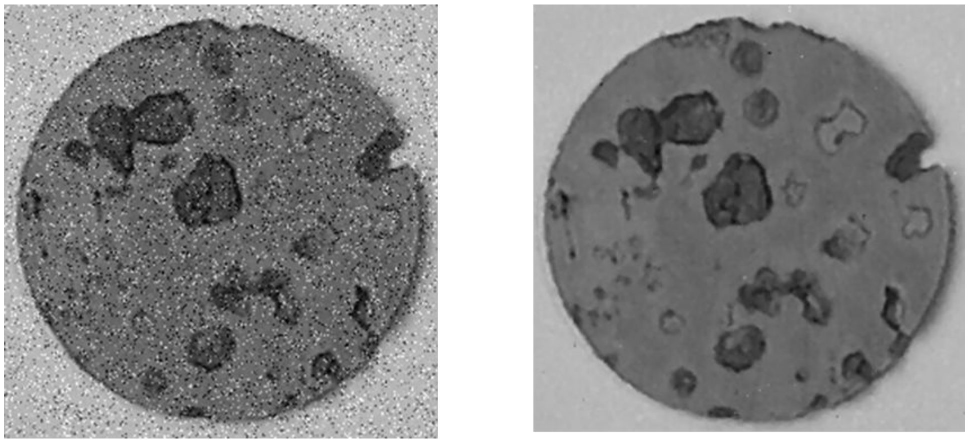
5.1 Median Filter Design Example

This walk-through uses an image processing median filter design to illustrate use of Fast-Forward compilation and Hyper-Retiming. 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]

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.

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.

Figure 109. Before and After Images Processed with Median Filtering

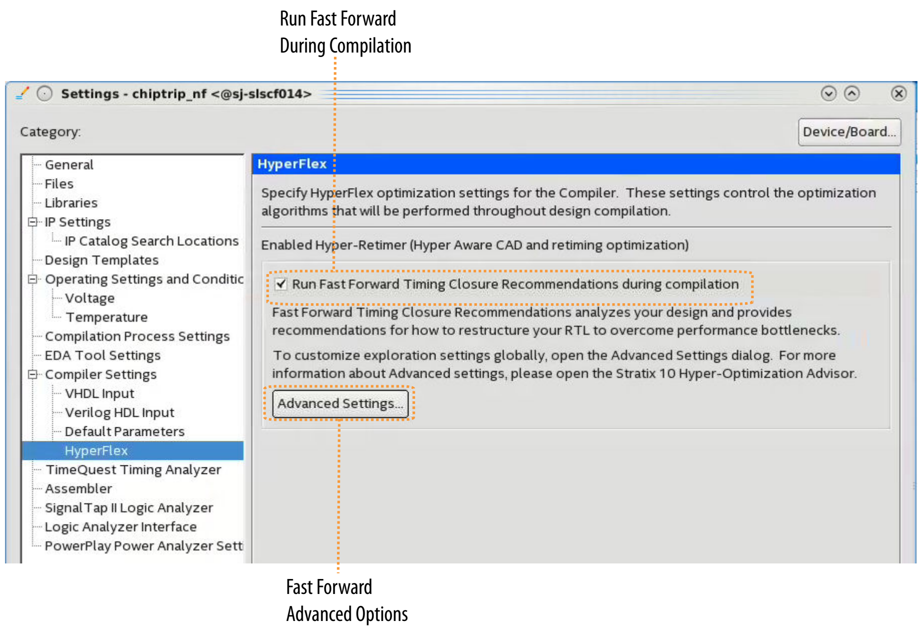


5.1.1 Step 1: Setup the Project

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.

1. Download and extract the Median Filter design example.
2. Open the `median.qsf` project in the Quartus Prime Pro v17.1 Stratix 10 ES Editions 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. The Compiler runs the Retime and Fast Forward compilation stages by default during full compilation. To verify these settings, click **Assignments ➤ Settings ➤ HyperFlex**.
6. Click **Tools ➤ TimeQuest Timing Analyzer**. Define all clocks and specify a 1 GHz clock frequency requirement. TimeQuest saves these settings in a Synopsys Design Constraints (`.sdc`) file. Alternatively, you can create the `.sdc` file manually.
7. Click **Assignments ➤ Assignment Editor** and assign the **Virtual Pin** option to all pins.

Figure 110. HyperFlex Settings



Specify HyperFlex optimization settings for the Compiler. These settings control the optimization algorithms that will be performed throughout design compilation.

Enabled Hyper-Retimer (Hyper Aware CAD and retiming optimization)

[✓] Run Fast Forward Timing Closure Recommendations during compilation

Fast Forward Timing Closure Recommendations analyzes your design and provides recommendations for how to restructure your RTL to overcome performance bottlenecks.

To customize exploration settings globally, open the Advanced Settings dialog. For more information about Advanced settings, please open the Stratix 10 Hyper-Optimization Advisor.

5.1.2 Step 2: Run Fast-Forward Compilation

Fast Forward compilation runs during full compilation, or you can run the process separately. Your design must successfully run through the previous compilation stages before Fast Forward Compile.

1. On the Compilation Dashboard, click **Fast Forward Timing Closure Recommendations**. The Compiler runs all modules in sequence through Fast Forward compilation.
2. View the analysis in the **Fast Forward Timing Closure Recommendations** folder of the Compilation Report.

5.1.3 Step 3: View Fast-Forward Recommendations

The Compiler generates detailed reports following register retiming and Fast Forward compilation. View the results of retiming in the **Retime Stage** section of the Fitter Report. View the results of Fast Forward analysis in the **Fast Forward Timing Closure Recommendations** reports.

5.1.4 Step 4: Implement Fast Forward Recommendations

Fast Forward compilation recommends steps to improve performance on specific paths. Consider Fast Forward recommendations, make appropriate changes in your design RTL, and then recompile the design to realize the performance gains.

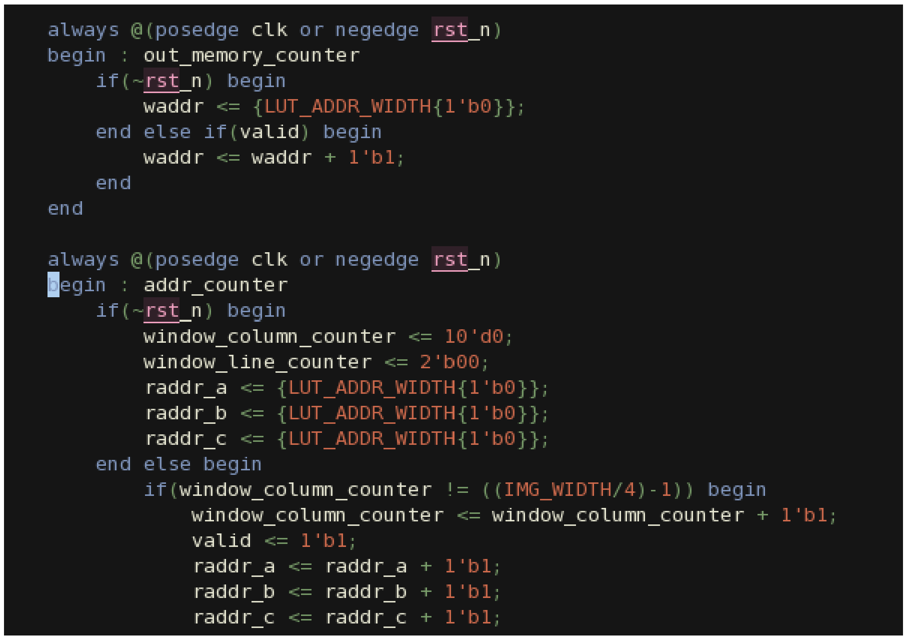
The amount and type of changes that you implement depends on your performance goals. For example, if you can achieve the target f_{MAX} with simple asynchronous clear removal or conversion, your optimization can end. However, if you require additional performance, implement more Fast Forward recommendations, such as any of the following techniques:

- Remove limitations of control logic, such as long feedback loops and state machines.
- Restructure logic to use functionally equivalent feed-forward or pre-compute paths, rather than long combinatorial feedback path.
- Reduce the delay of 'Long Paths' in the chain. Use standard timing closure techniques to reduce delay. Excessive combinational logic, sub-optimal placement, and routing congestion cause delay on paths.
- Insert more pipeline stages in 'Long Paths' in the chain. Long paths have the most delay between registers in the critical chain.
- Increase the delay (or add pipeline stages to 'Short Paths' in the chain).

Explore performance and implement the RTL changes to your code until you reach the desired performance target.

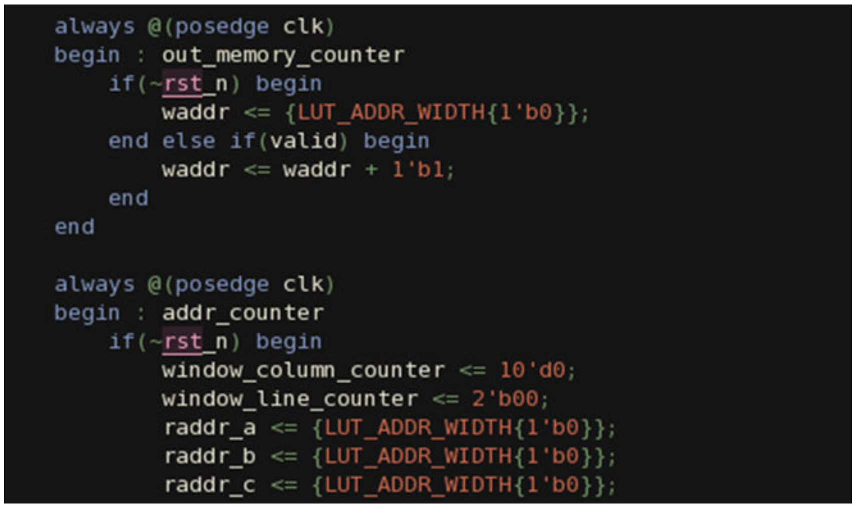
In the design example, the presence of asynchronous signals (resets in the design) affects retiming abilities. As a starting point for handing resets, ensure that resets are synchronous. The RTL guidelines in this document describe appropriate reset strategies.

Figure 111. 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;
```

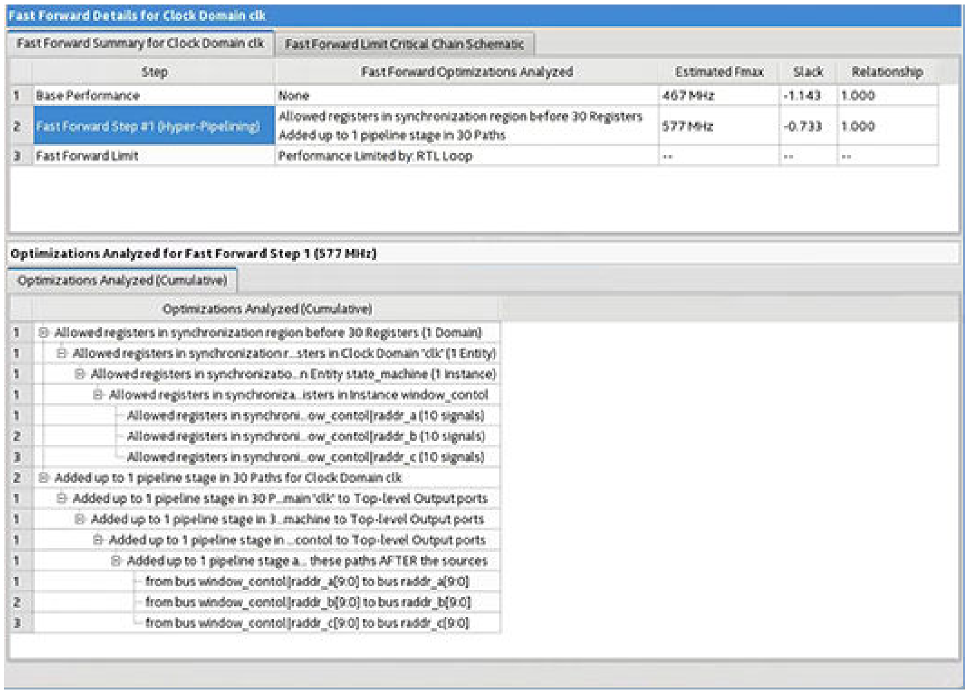
Figure 112. Synchronous Reset in State Machine RTL



```
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}};
```

After changes to synchronous reset, the design example can still benefit from an additional stage of pipeline registers. The **Fast Forward Details** report lists the specific location to add these registers. Add registers at either the path source or destination. The Fitter choses the optimal placement automatically.

Figure 113. Fast Forward Compilation Improvement



Fast Forward Details for Clock Domain clk

Fast Forward Summary for Clock Domain clk | Fast Forward Limit Critical Chain Schematic

	Step	Fast Forward Optimizations Analyzed	Estimated Fmax	Slack	Relationship
1	Base Performance	None	467 MHz	-1.143	1.000
2	Fast Forward Step #1 (Hyper-Pipelining)	Allowed registers in synchronization region before 30 Registers Added up to 1 pipeline stage in 30 Paths	577 MHz	-0.733	1.000
3	Fast Forward Limit	Performance Limited by: RTL Loop	--	--	--

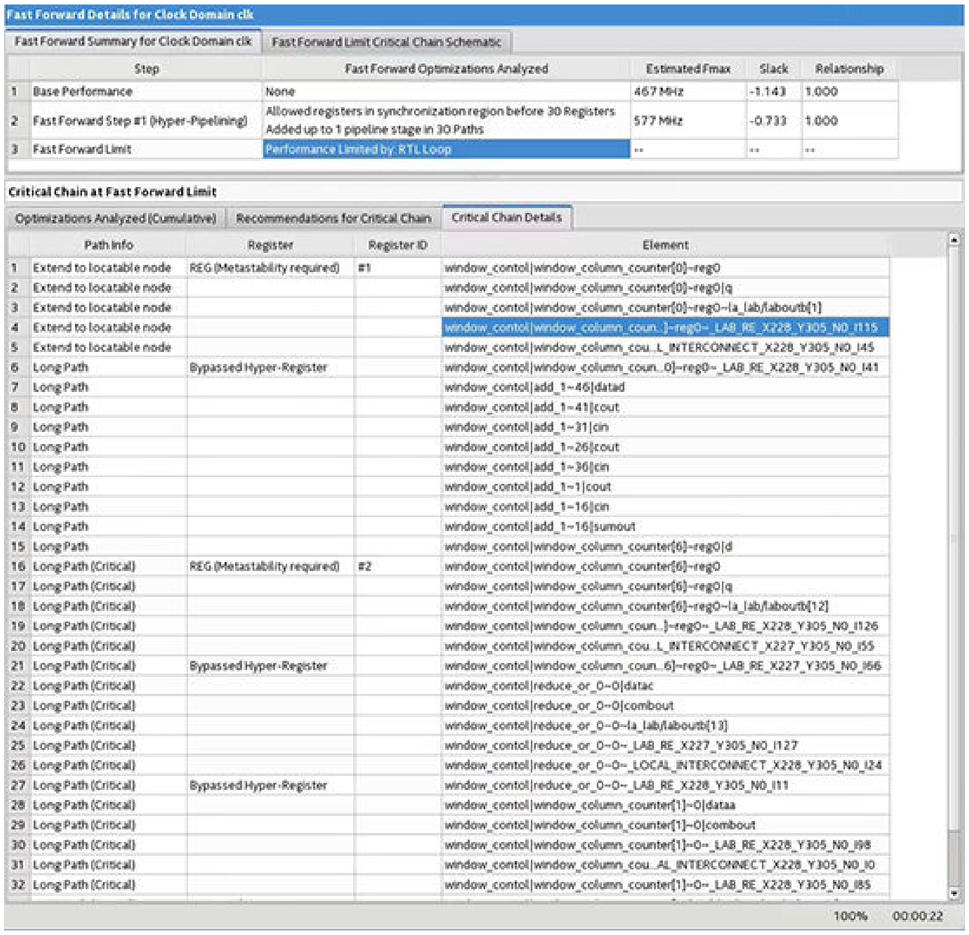
Optimizations Analyzed for Fast Forward Step 1 (577 MHz)

Optimizations Analyzed (Cumulative)

	Optimizations Analyzed (Cumulative)
1	⊟ Allowed registers in synchronization region before 30 Registers (1 Domain)
1	⊟ Allowed registers in synchronization r...sters in Clock Domain 'clk' (1 Entity)
1	⊟ Allowed registers in synchronizatio...n Entity state_machine (1 Instance)
1	⊟ Allowed registers in synchroniza...isters in Instance window_contol
1	Allowed registers in synchroni...ow_contol|raddr_a (10 signals)
2	Allowed registers in synchroni...ow_contol|raddr_b (10 signals)
3	Allowed registers in synchroni...ow_contol|raddr_c (10 signals)
2	⊟ Added up to 1 pipeline stage in 30 Paths for Clock Domain clk
1	⊟ Added up to 1 pipeline stage in 30 P...main 'clk' to Top-level Output ports
1	⊟ Added up to 1 pipeline stage in 3...machine to Top-level Output ports
1	⊟ Added up to 1 pipeline stage in ...contol to Top-level Output ports
1	⊟ Added up to 1 pipeline stage a... these paths AFTER the sources
1	from bus window_contol|raddr_a[9:0] to bus raddr_a[9:0]
2	from bus window_contol|raddr_b[9:0] to bus raddr_b[9:0]
3	from bus window_contol|raddr_c[9:0] to bus raddr_c[9:0]

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 Details** 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. 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 114. Fast Forward Details



Fast Forward Details for Clock Domain clk

Fast Forward Summary for Clock Domain clk | Fast Forward Limit Critical Chain Schematic

	Step	Fast Forward Optimizations Analyzed	Estimated Fmax	Slack	Relationship
1	Base Performance	None	467 MHz	-1.143	1.000
2	Fast Forward Step #1 (Hyper-Pipelining)	Allowed registers in synchronization region before 30 Registers Added up to 1 pipeline stage in 30 Paths	577 MHz	-0.733	1.000
3	Fast Forward Limit	Performance Limited by: RTL Loop	--	--	--

Critical Chain at Fast Forward Limit

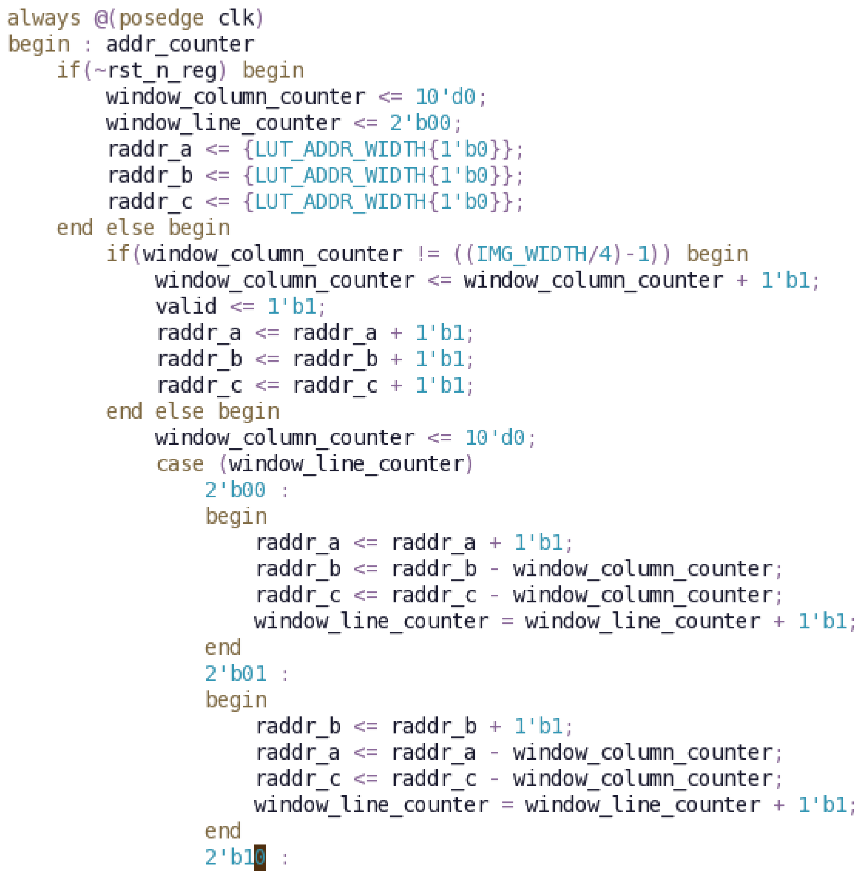
Optimizations Analyzed (Cumulative) | Recommendations for Critical Chain | Critical Chain Details

	Path Info	Register	Register ID	Element
1	Extend to locatable node	REG (Metastability required)	#1	window_contol|window_column_counter[0]~reg0
2	Extend to locatable node			window_contol|window_column_counter[0]~reg0|q
3	Extend to locatable node			window_contol|window_column_counter[0]~reg0~la_lab/laboutb[1]
4	Extend to locatable node			window_contol|window_column_coun...]~reg0~_LAB_RE_X228_Y305_N0_I115
5	Extend to locatable node			window_contol|window_column_cou...L_INTERCONNECT_X228_Y305_N0_I45
6	Long Path	Bypassed Hyper-Register		window_contol|window_column_coun...0]~reg0~_LAB_RE_X228_Y305_N0_I41
7	Long Path			window_contol|add_1~46|datad
8	Long Path			window_contol|add_1~41|cout
9	Long Path			window_contol|add_1~31|cin
10	Long Path			window_contol|add_1~26|cout
11	Long Path			window_contol|add_1~36|cin
12	Long Path			window_contol|add_1~1|cout
13	Long Path			window_contol|add_1~16|cin
14	Long Path			window_contol|add_1~16|sumout
15	Long Path			window_contol|window_column_counter[6]~reg0|d
16	Long Path (Critical)	REG (Metastability required)	#2	window_contol|window_column_counter[6]~reg0
17	Long Path (Critical)			window_contol|window_column_counter[6]~reg0|q
18	Long Path (Critical)			window_contol|window_column_counter[6]~reg0~la_lab/laboutb[12]
19	Long Path (Critical)			window_contol|window_column_coun...]~reg0~_LAB_RE_X228_Y305_N0_I126
20	Long Path (Critical)			window_contol|window_column_cou...L_INTERCONNECT_X227_Y305_N0_I55
21	Long Path (Critical)	Bypassed Hyper-Register		window_contol|window_column_coun...6]~reg0~_LAB_RE_X227_Y305_N0_I66
22	Long Path (Critical)			window_contol|reduce_or_0~0|datac
23	Long Path (Critical)			window_contol|reduce_or_0~0|combout
24	Long Path (Critical)			window_contol|reduce_or_0~0~la_lab/laboutb[13]
25	Long Path (Critical)			window_contol|reduce_or_0~0~_LAB_RE_X227_Y305_N0_I127
26	Long Path (Critical)			window_contol|reduce_or_0~0~_LOCAL_INTERCONNECT_X228_Y305_N0_I24
27	Long Path (Critical)	Bypassed Hyper-Register		window_contol|reduce_or_0~0~_LAB_RE_X228_Y305_N0_I11
28	Long Path (Critical)			window_contol|window_column_counter[1]~0|dataa
29	Long Path (Critical)			window_contol|window_column_counter[1]~0|combout
30	Long Path (Critical)			window_contol|window_column_counter[1]~0~_LAB_RE_X228_Y305_N0_I98
31	Long Path (Critical)			window_contol|window_column_cou...AL_INTERCONNECT_X228_Y305_N0_I0
32	Long Path (Critical)			window_contol|window_column_counter[1]~0~_LAB_RE_X228_Y305_N0_I85

100% 00:00:22

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

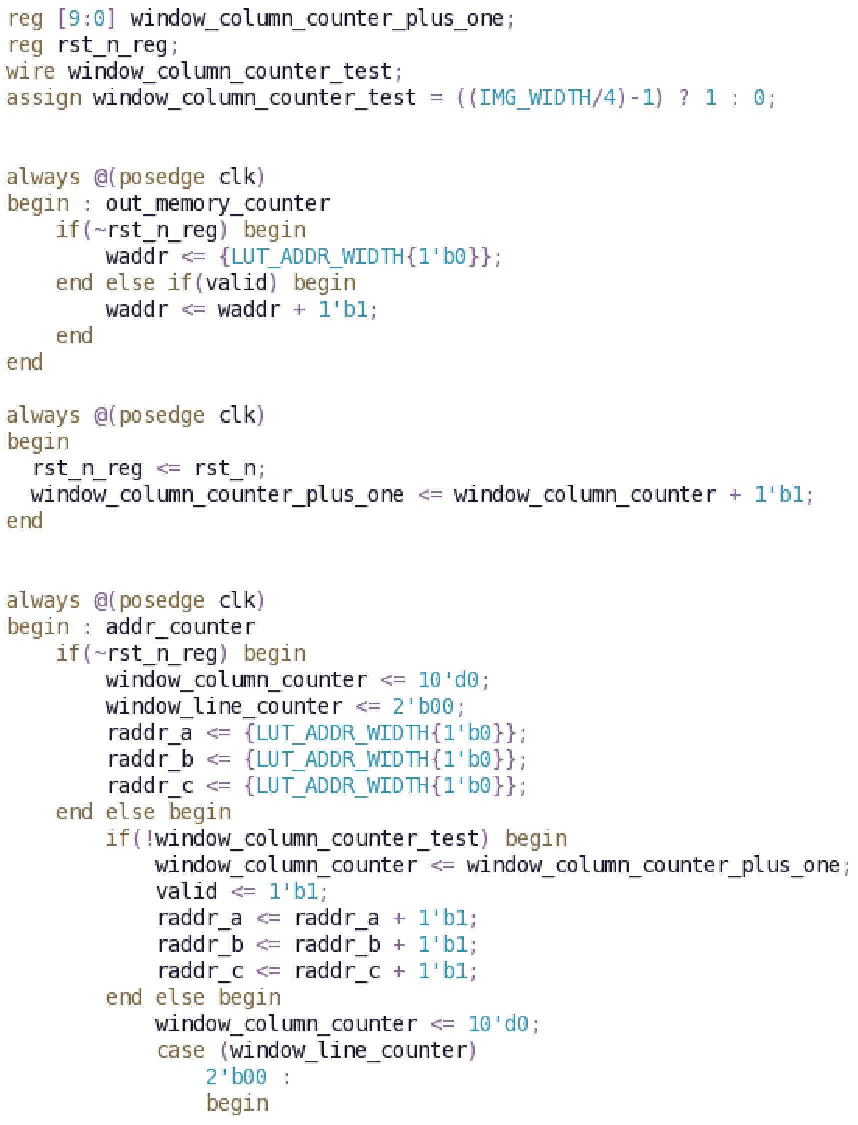
Figure 115. Non-Optimized RTL



```
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.

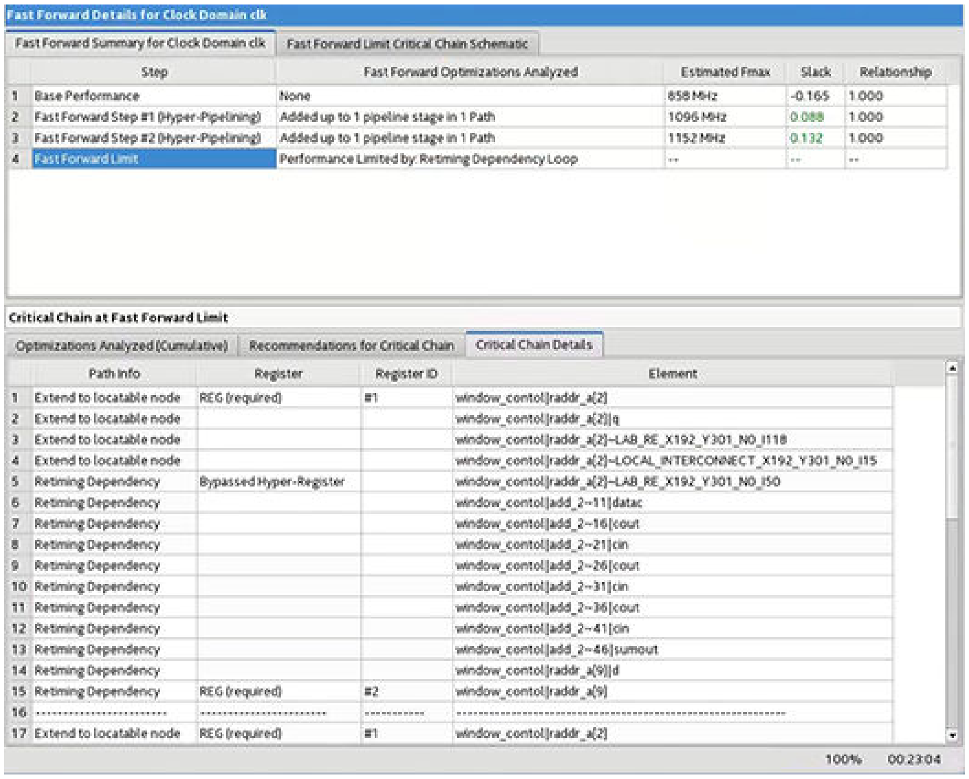
Figure 116. Optimized RTL



```
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 to 1152 MHz on Fast Forward Step #2 (Hyper-Pipelining).

Figure 117. Fast Forward Details Report



Fast Forward Details for Clock Domain clk

Fast Forward Summary for Clock Domain clk | Fast Forward Limit Critical Chain Schematic

	Step	Fast Forward Optimizations Analyzed	Estimated Fmax	Slack	Relationship
1	Base Performance	None	858 MHz	-0.165	1.000
2	Fast Forward Step #1 (Hyper-Pipelining)	Added up to 1 pipeline stage in 1 Path	1096 MHz	0.088	1.000
3	Fast Forward Step #2 (Hyper-Pipelining)	Added up to 1 pipeline stage in 1 Path	1152 MHz	0.132	1.000
4	Fast Forward Limit	Performance Limited by: Retiming Dependency Loop	--	--	--

Critical Chain at Fast Forward Limit

Optimizations Analyzed (Cumulative) | Recommendations for Critical Chain | Critical Chain Details

	Path Info	Register	Register ID	Element
1	Extend to locatable node	REG (required)	#1	window_contol|raddr_a[2]
2	Extend to locatable node			window_contol|raddr_a[2]|q
3	Extend to locatable node			window_contol|raddr_a[2]~LAB_RE_X192_Y301_N0_I118
4	Extend to locatable node			window_contol|raddr_a[2]~LOCAL_INTERCONNECT_X192_Y301_N0_I15
5	Retiming Dependency	Bypassed Hyper-Register		window_contol|raddr_a[2]~LAB_RE_X192_Y301_N0_I50
6	Retiming Dependency			window_contol|add_2~11|datac
7	Retiming Dependency			window_contol|add_2~16|cout
8	Retiming Dependency			window_contol|add_2~21|cin
9	Retiming Dependency			window_contol|add_2~26|cout
10	Retiming Dependency			window_contol|add_2~31|cin
11	Retiming Dependency			window_contol|add_2~36|cout
12	Retiming Dependency			window_contol|add_2~41|cin
13	Retiming Dependency			window_contol|add_2~46|sumout
14	Retiming Dependency			window_contol|raddr_a[9]|d
15	Retiming Dependency	REG (required)	#2	window_contol|raddr_a[9]
16	------------------------	------------------------	------------	--
17	Extend to locatable node	REG (required)	#1	window_contol|raddr_a[2]

100% 00:23:04

6 Optimization Example

This section contains a round robin scheduler optimization example.

6.1 Round Robin Scheduler

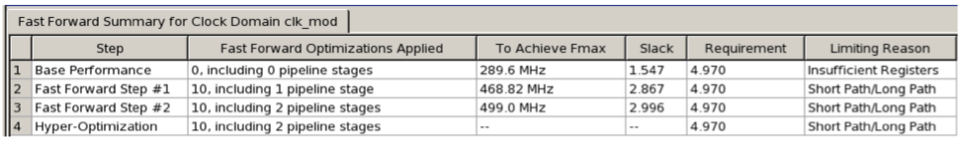
The round robin scheduler is a basic functional block. The following example uses a modulus operator to determine the next client for service. The modulus operator is relatively slow and area inefficient because it performs division.

Example 16. 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 wire [LOG2_CLIENTS -1:0]  last,

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

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

};

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

always @*
begin
   integer J, K;

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

 endmodule
```

Figure 118. Fast Forward Compile Report for Round Robin Scheduler

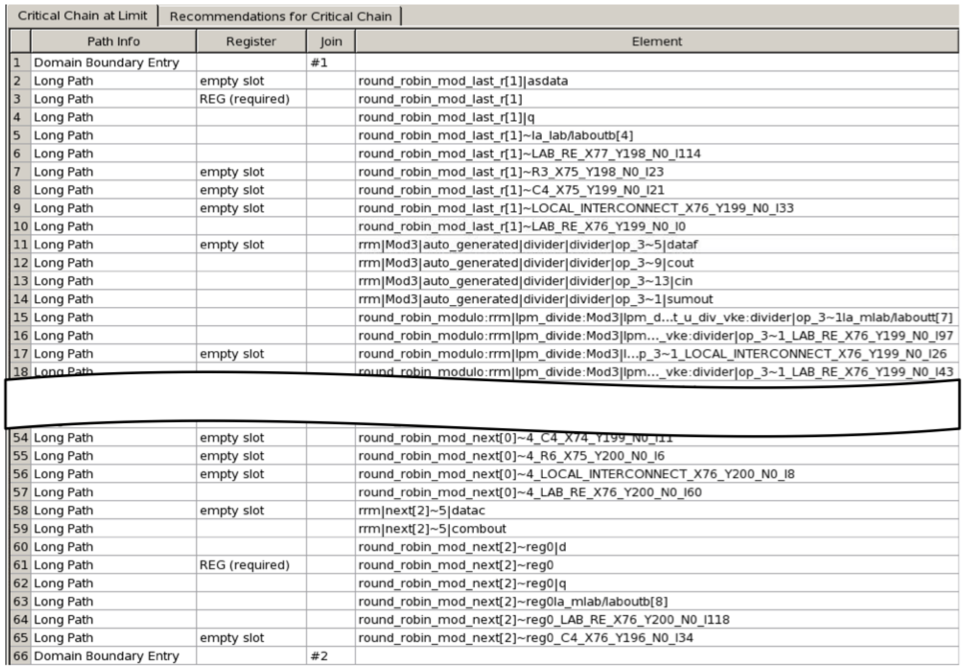


Fast Forward Summary for Clock Domain clk_mod

	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 Fast Forward Summary report identifies insufficient registers limiting Hyper-Retiming on the critical chain. 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 119. Critical Chain for Base Performance for Round Robin Scheduler

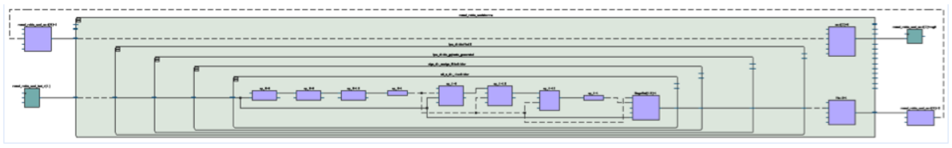


Critical Chain at Limit | Recommendations for Critical Chain

	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	

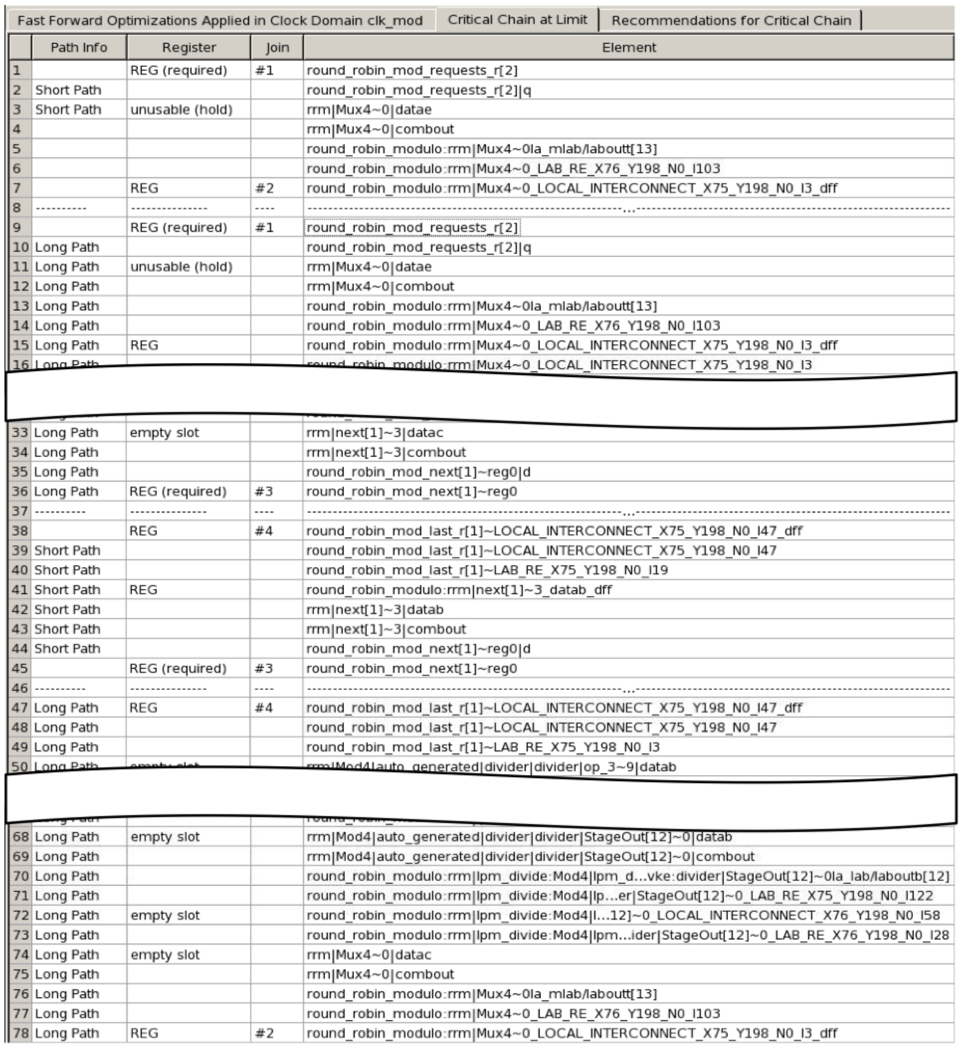
The 66 elements in the critical chain above, correspond to the circuit diagram below with 13 levels of logic. The modulus operator contributes significantly to the low performance. Nine of the 13 levels of logic are part of the implementation for the modulus operator.

Figure 120. Schematic for Critical Chain



Fast Forward compilation 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 121. 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	empty slot		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

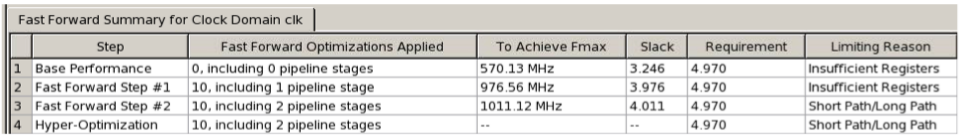
The divider in the modulus operation is the bottleneck that requires RTL modification. Paths through the divider exist in the critical chain for all steps in the Fast Forward compile. Consider alternate implementations to calculate the next client to service and avoid the modulus operator. If you switch to an implementation that specifies the number of clients as a power of two, 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 17. 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 compilation result with the modulus operator.

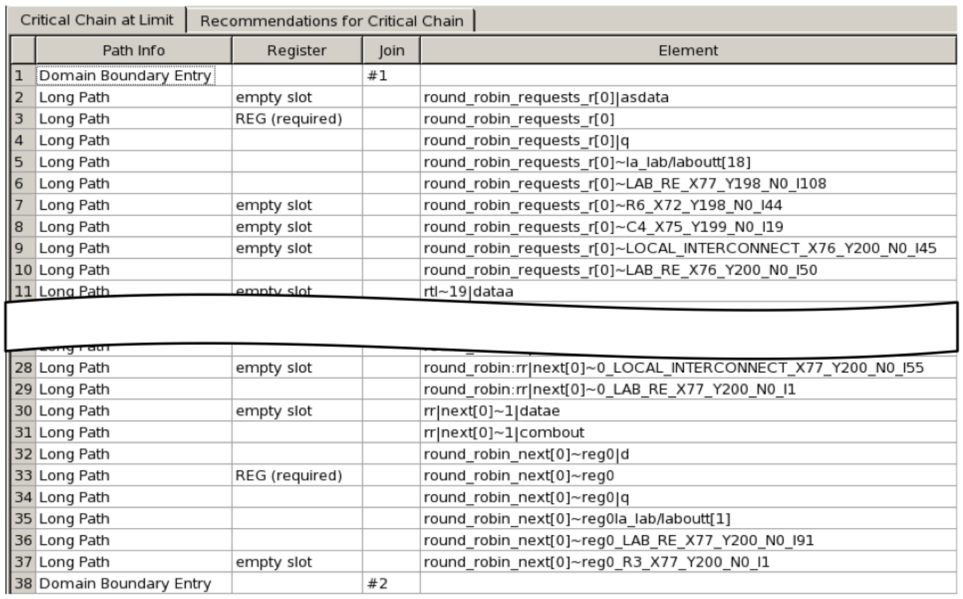
Figure 122. Fast Forward Summary Report for Round Robin Scheduler with Improved Performance with 2^n Client Inputs



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.



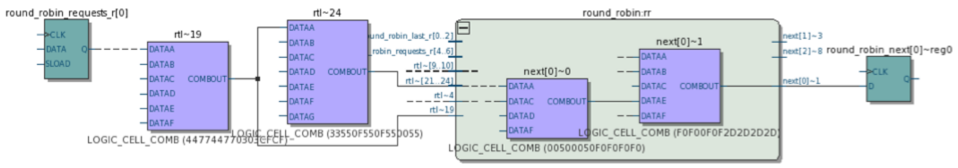
Critical Chain at Limit | Recommendations for Critical Chain

	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	

Figure 123. Critical Chain for Round Robin Scheduler with Improved Performance

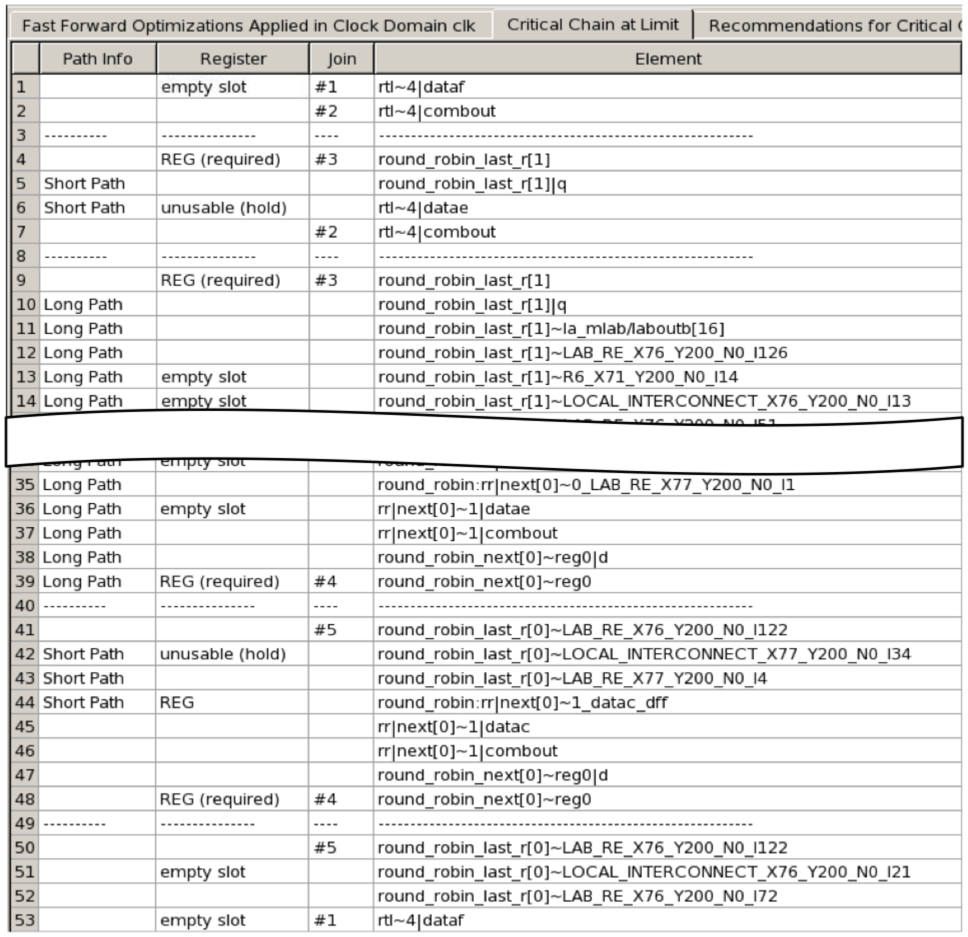
The 38 elements in the critical chain above correspond to the following circuit diagram with only four levels of logic.

Figure 124. 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.

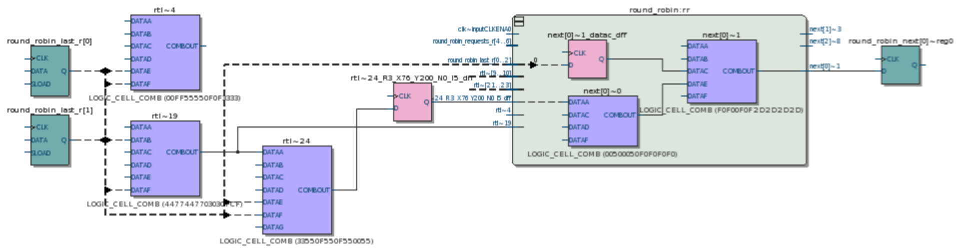
Figure 125. Critical Chain for Round Robin Scheduler with Best Performance



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 126. Schematic for Critical Chain with Best Performance


round_robin_last_r[0]
round_robin_last_r[1]
rtl~4
rtl~19
rtl~24
LOGIC_CELL_COMB (00FF55550F0F3333)
LOGIC_CELL_COMB (447744770303CFCF)
LOGIC_CELL_COMB (33550F550F550055)
rtl~24_R3_X76_Y200_N0_I5_dff
round_robin:rr
next[0]~1_datac_dff
next[0]~1
next[0]~0
LOGIC_CELL_COMB (F0F00F0F2D2D2D2D)
LOGIC_CELL_COMB (00500050F0F0F0F0)
round_robin_next[0]~reg0
CLK
DATA
SLOAD
Q
COMBOUT

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 Appendices

7.1 Appendix A: Parameterizable Pipeline Modules

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

Example 18. Parameterizable Hyper-Pipelining Verilog HDL Module

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

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

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

Example 19. Parameterizable Hyper-Pipelining Verilog HDL Instance

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

Example 20. Parameterizable Hyper-Pipelining SystemVerilog Module

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

    generate
        if (CYCLES==0) begin : GEN_COMB_INPUT
```

```
            assign dout = din;
        end
        else begin : GEN_REG_INPUT
            integer i;
            reg [PACKED_WIDTH-1:0] R_data
                [CYCLES-1.0][UNPACKED_WIDTH-1:0];

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

Example 21. Parameterizable Hyper-Pipelining SystemVerilog Instance

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

Example 22. Parameterizable Hyper-Pipelining VHDL Entity

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

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

architecture arch of hyperpipe is

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

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

    begin
        wire : if CYCLES=0 GENERATE
            -- The 0 bit is just a pass-thru, when CYCLES is set to 0
            dout <= din;
        end generate wire;
        hp : if CYCLES>0 GENERATE
            process (clk) begin
                if (clk'event and clk='1')then
                    HR <= HR(HR'high-1 downto 0) & din;
                end if;
            end process;
            dout <= HR(HR'high);
        end generate hp;

    end arch;
```

Example 23. Parameterizable Hyper-Pipelining VHDL Instance

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

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

7.2 Appendix B: Clock Enables and Resets

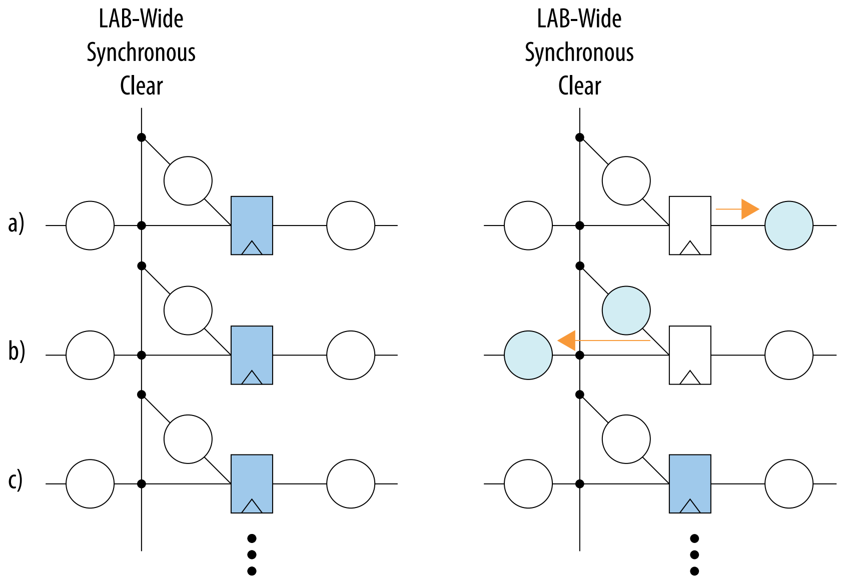
7.2.1 Synchronous Resets and Limitations

Converting asynchronous resets to synchronous helps retiming, but there are still performance restrictions. The ALM register's dedicated LAB-wide signal often performs synchronous clears. The signal’s fan-out determines use of this signal during synthesis. A synchronous clear with a small fan-out is usually performed in logic, while larger fan-outs use this dedicated signal. Even if synthesis determines use of the synchronous clear, the Compiler still retimes the register into Hyper-Registers. The bypass mode of the ALM register enables this functionality. When the Compiler bypasses the register, the `sclr` signal and other control signals remain accessible.

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

Figure 127. 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.

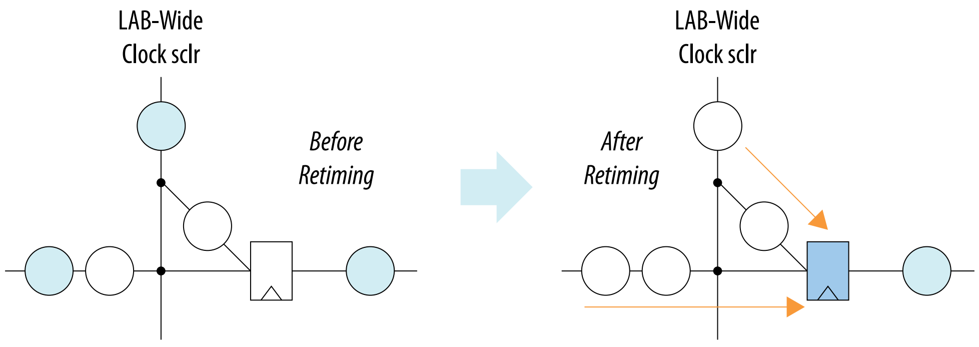


During retiming, the top register in row (a) is pushed right into a Hyper-Register. This is achieved by bypassing the ALM register, but still using the SCLR logic that feeds that register. When the LAB-wide SCLR signal is used, an ALM register must exist on the data path, but it does not have to be used.

Register retiming pushes the register in row (b) left into its data path. The register pushes through a signal split of the data path and synchronous clear. 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.

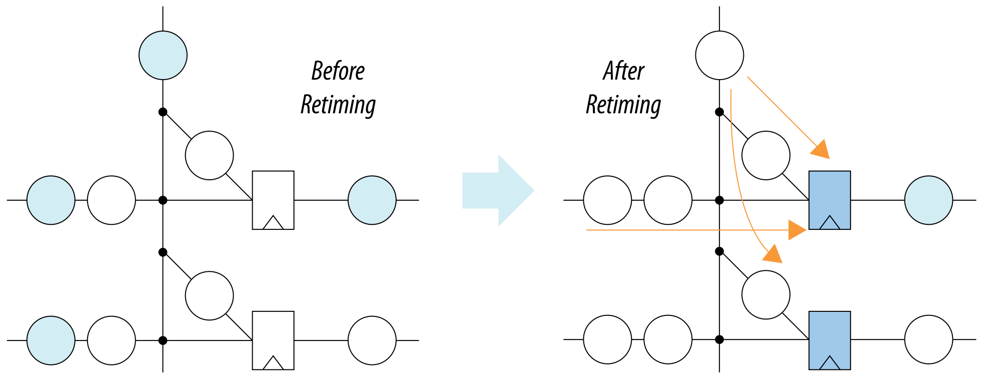
Retiming becomes complicated if another register is pushed forward into the ALM. As shown in the following figure, a register from the asynchronous clear port and a register from the data path must be merged together.

Figure 128. Retiming Example – Second Register Pushed out of ALM



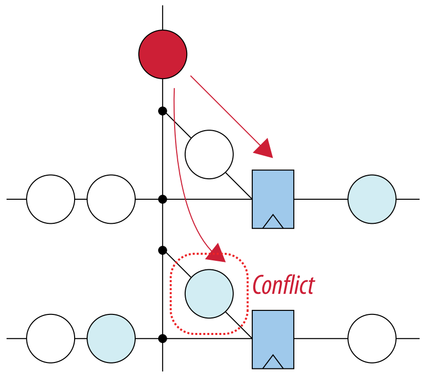
Because the register on the synchronous clear path is shared with other registers, the register splits on the path to other synchronous clear ports as well.

Figure 129. Retiming Example – Register Splits on the Path to other Synchronous Clear Ports



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

Figure 130. Retiming Example – Conflict at Synchronous Clear



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

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

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

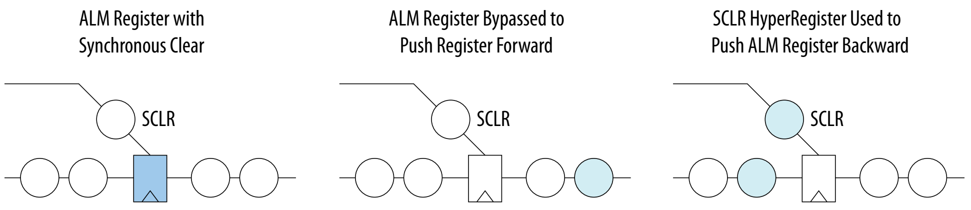
7.2.1.1 Synchronous Resets Summary

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

- A short path, usually going directly from the source register to the destination register without any logic between them. Short paths are not normally a problem, because their positive slack can be retimed out to longer paths. This makes the whole design run faster. However, short paths typically connect to long data paths that require retiming. By retiming many registers along long paths, registers are pushed down or pulled up this short path. This problem is not significant 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 become 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 occurs when a second register must be retimed through the ALM register.

Figure 131. Aggressive Retiming


ALM Register with Synchronous Clear
SCLR
ALM Register Bypassed to Push Register Forward
SCLR
SCLR HyperRegister Used to Push ALM Register Backward
SCLR

Consider an ALM register with an 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. Stratix 10 devices have a dedicated Hyper-Register on the SCLR path, and the ability to put the ALM register into bypass mode. This allows you to 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.

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:

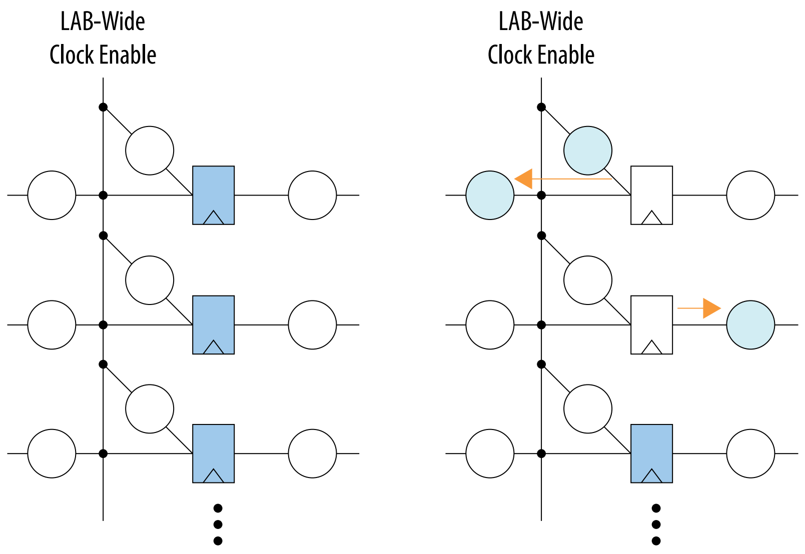
- When writing new code that needs to run at high speeds, avoid synchronous resets wherever possible. This is generally in data path logic that either flushes out while the system is in reset, or its values are ignored when the system comes out of reset, until new, valid logic filters through.
- Control 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.

7.2.2 Retiming with Clock Enables

Like synchronous resets, clock enables use a dedicated LAB-wide resource that feeds a specific function in the ALM register. Similarly, Stratix 10 devices support 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.

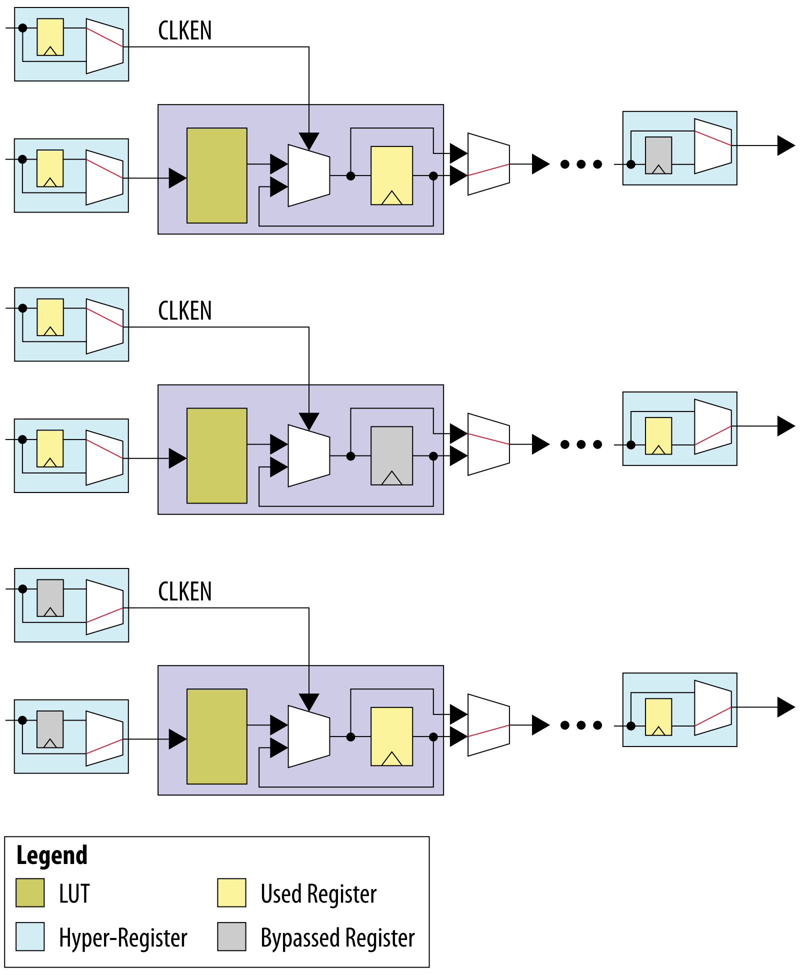
Figure 132. ALM Representing Clock Enables

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.



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 133. 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

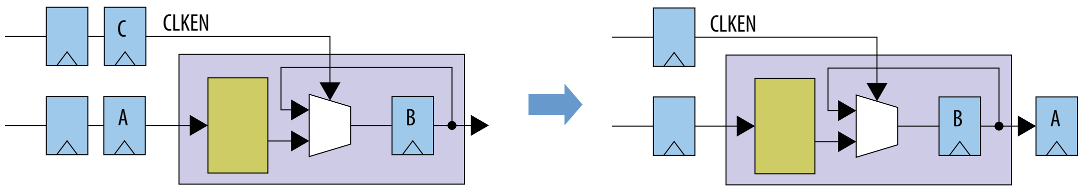
Synchronous Resets and Limitations on page 117

7.2.2.1 Example for Broadcast Control Signals

Broadcast control signals that fan-out to many destinations limit retiming. Asynchronous clears can limit retiming due to silicon support for certain register control signals. However, even synchronous signals, such as synchronous clear and clock enable, can limit retiming when 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.

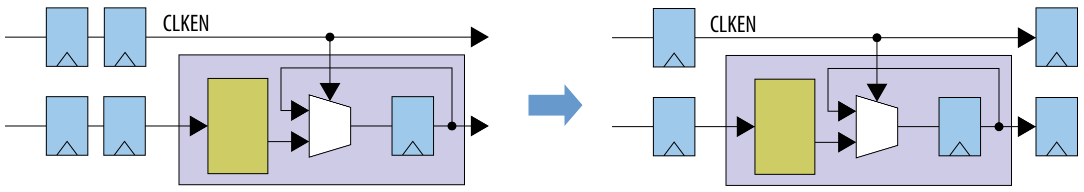
To forward retime a register over a node, there must be a register available on all of the node's inputs. This requirement is the same for conventional retiming and Hyper-Retiming. To retime register A over register B in the following diagram, a register must be pulled from all inputs, including register 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 134. Retiming through a Clock Enable



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

Figure 135. Retiming through a Clock Enable with a Branch Point

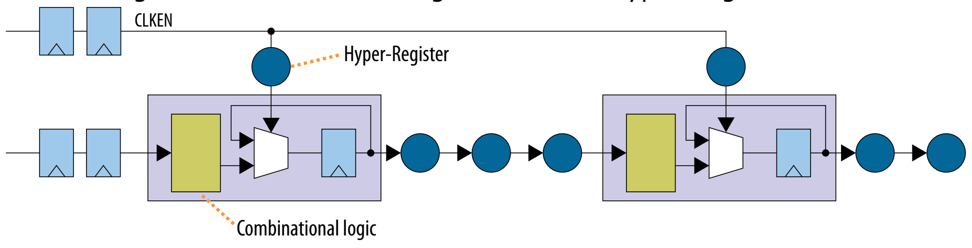


Retiming Example

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 136. Retiming Example Starting Point

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



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

One stage of the input registers retime into a Hyper-Register location between the two registers. Figure 137 on page 125 shows one part of the Hyper-Retiming forward push. One of the registers on the clock enable input is retimes over the branch point, with a copy going to a Hyper-Register location at each clock enable input.

Figure 137. Retiming Example Intermediate Point

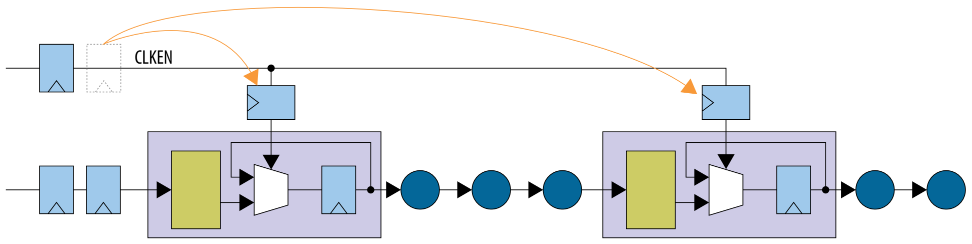


Figure 138 on page 125 shows the positions of the registers in the circuit after Hyper-Retiming completes the forward push. The two registers at the inputs of the left register retime to a Hyper-Register location. This diagram is functionally equivalent to the two previous diagrams. The one Hyper-Register location at the clock enable input of the second register remains occupied. There are no other Hyper-Register locations on the clock enable path to the second register, yet there is still one register at the inputs that could be retimed.

Figure 138. Retiming Example Ending Point

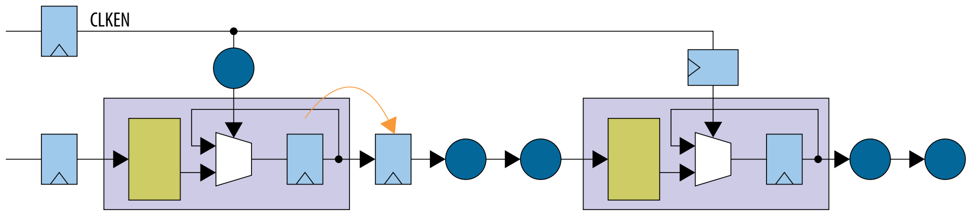
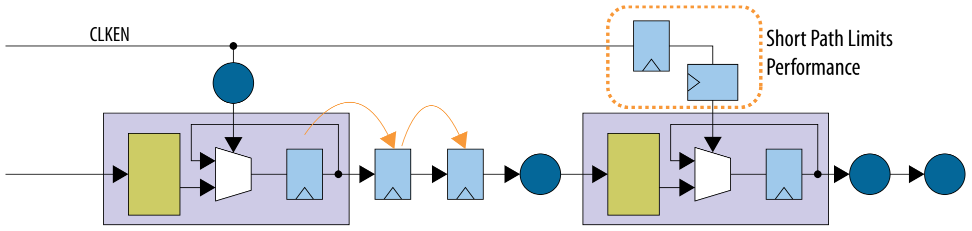


Figure 139 on page 126 shows the register positions Hyper-Retiming could use if not limited by a short path/long path critical chain. However, because no Hyper-Registers are available on the right-hand clock enable path, Hyper-Retiming cannot retime the circuit as shown in the diagram.

Figure 139. Retiming Example Limiting condition



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

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

Related Links

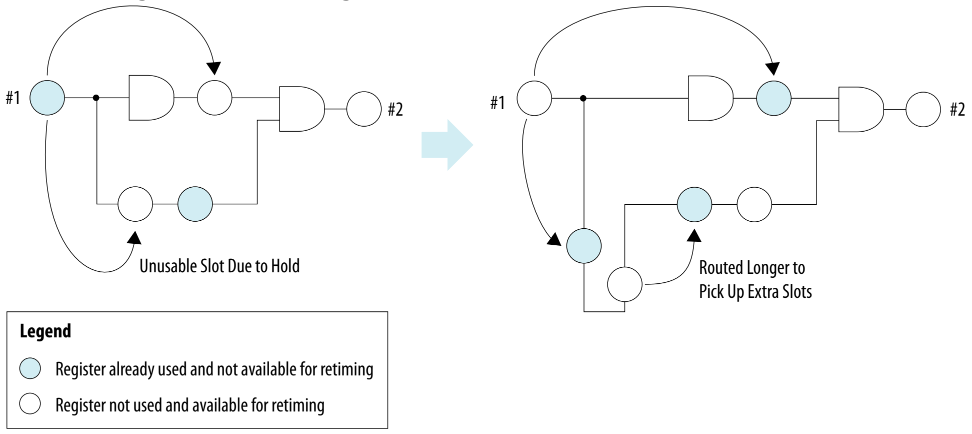
Appendix A: Parameterizable Pipeline Modules on page 115

7.2.3 Resolving Short Paths

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

Figure 140. Short Paths Limiting Retiming

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


#1
#2
Unusable Slot Due to Hold
#1
#2
Routed Longer to
Pick Up Extra Slots
Legend
Register already used and not available for retiming
Register not used and available for 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 Compiler detects these short paths, and routes the registers to longer paths, as shown in the circuit on the right. This practice ensures that sufficient slots are available for retiming.

The following two examples address short paths:

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

The limiting reason is the short path / long path. Add the 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. Fast Forward compile does not make any recommendations to add pipeline stages.

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

8 Document Revision History

This document has the following revision history.

Table 13. Document Revision History

Date	Version	Changes
2017.05.08	Quartus Prime Pro v17.1 Stratix 10 ES Editions	• Updated software support version to Quartus Prime Pro v17.1 Stratix 10 ES Editions. • Added Initial Power-Up Conditions topic. • Added Retiming Reset Sequences topic. • Added guidelines for high-speed clock domains. • Added Fitter Overconstraints topic. • Described Hold Fix-up in Fitter Finalize stage. • Added statement about Fast Forward compilation support for retiming across RAM and DSP blocks. • Added details on coherent RAM to read-modify-write memory description. • Added description of Fast Forward Viewer and Hyper-Optimization Advisor. • Added Advanced HyperFlex Settings topic. • Added Prevent Register Retiming topic. • Added Preserve Registers During Synthesis topic. • Added Fitter Commands topic. • Added Finalize Stage Reports topic. • Replaced command line instructions with new GUI steps in compilation flows. • Described concurrent analysis controls in Compilation Dashboard. • Consolidated duplicate content and grouped Appendices together. • Updated diagrams and screenshots.
2016.08.07	2016.08.07	• Added clock crossing and initial condition timing restriction details. • Described true dual-port memory support and memory width ratio with examples • Updated code samples and narrative in Design Example Walk-through • Added reference to provided Design Example files • Re-branded for Intel • Updated for latest changes to software GUI and capabilities.
2016.03.16	2016.03.16	First public release.