Intel® Hyperflex™ Architecture High-Performance Design Handbook

Updated for Intel® Quartus® Prime Design Suite: **23.4**

Answers to Top FAQs:

Online Version
Send Feedback

S10HPHB

683353
2023.12.08

Contents

1. Intel® Hyperflex™ FPGA Architecture Introduction

This document describes design techniques to achieve maximum performance with the Intel® Hyperflex™ FPGA architecture. The Intel Hyperflex FPGA architecture supports Hyper-Retiming, Hyper-Pipelining, and Hyper-Optimization design techniques that enable the highest clock frequencies in Intel Stratix® 10 and Intel Agilex® 7 devices.

Table 1. Intel Hyperflex Architecture FPGAs

Intel Hyperflex Architecture Devices	Intel Hyperflex Architecture Description
Intel Stratix 10 FPGAs	A "registers everywhere" architecture that packs bypassable Hyper-Registers into routing segments in the device core, and at all functional block inputs. The routing signal can travel through the register first, or bypass the register direct to the multiplexer, improving bandwidth and area and power efficiency.
Intel Agilex 7 FPGAs	

Figure 1. Registers Everywhere

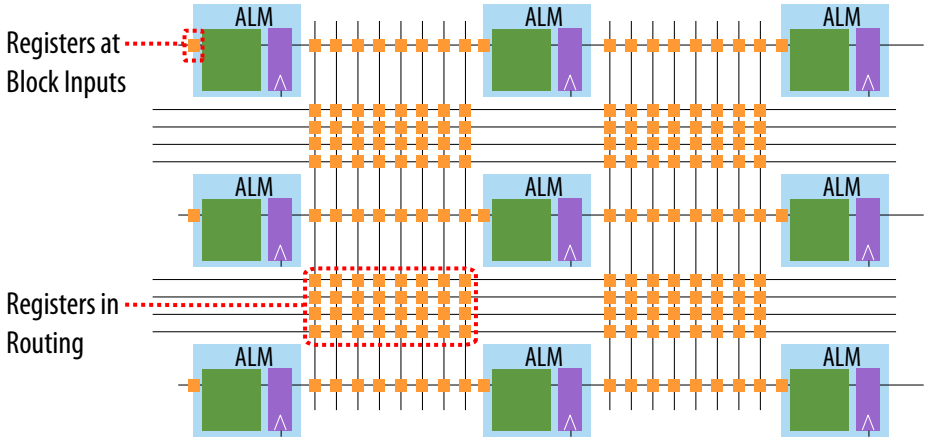
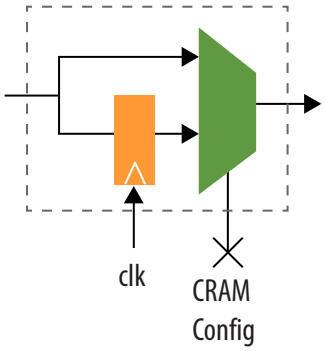


Figure 2. Bypassable Hyper-Registers



This document provides specific design guidelines, tool flows, and real world examples to take advantage of the Intel Hyperflex FPGA architecture:

- Intel Hyperflex Architecture RTL Design Guidelines on page 6—describes fundamental high-performance RTL design techniques for Intel Hyperflex FPGA architecture designs.
- Compiling Intel Hyperflex Architecture Designs on page 75—describes how to use the Intel Quartus Prime Pro Edition software to get the highest performance with Intel Hyperflex architecture FPGAs.
- Optimization Example on page 118—demonstrates performance improvement techniques using real world design examples.
- Intel Hyperflex Architecture Porting Guidelines on page 124—provides guidance for design migration to Intel Hyperflex architecture FPGAs.

1.1. Intel Hyperflex Architecture Design Concepts

Table 2. Glossary

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

Related Information

- Hyper-Retiming (Facilitate Register Movement) on page 11
- Hyper-Pipelining (Add Pipeline Registers) on page 29
- Hyper-Optimization (Optimize RTL) on page 42

2. Intel Hyperflex Architecture RTL Design Guidelines

This chapter describes RTL design techniques to achieve the highest clock rates possible in Intel Hyperflex architecture FPGAs. Intel Hyperflex architecture FPGAs support maximum clock rates significantly higher than previous FPGA generations.

Note: Avoiding RTL design rule violations improves the reliability, timing performance, and logic utilization of your design. The Intel Quartus Prime software includes the Design Assistant design rule checking tool to help avoid design rule violations. These rules include Hyper-Retimer Readiness Rules (HRR) that specifically target Intel Hyperflex architecture FPGA designs, as Design Assistant Design Rule Checking on page 79 describes.

Related Information

Intel Quartus Prime Pro Edition User Guide: Design Recommendations
For complete information on setup and use of Design Assistant

2.1. High-Speed Design Methodology

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

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

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

2.1.1. Set a High-Speed Target

For silicon efficiency, set your speed target as high as possible. The Intel Hyperflex architecture LUT is essentially a tiny ROM capable of a billion lookups per second. Operating this 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 a path includes many nodes, 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.

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

2.1.1.2. Speed and Latency

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

Table 3. 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 than linear area benefit. A ripple adder grows linearly as the bus width increases.

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

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

2.1.2. Experiment and Iterate

Experiment with settings and design changes if design performance does not initially meet performance requirements. Intel FPGA reprogrammability allows experimentation to achieve your goals. Design performance typically 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. You can add registers in functionally illegal locations to determine the effect on overall timing. If the prospective circuit then meets the timing objective, you can focus on design floorplanning.

If a circuit remains too slow, even when liberally inserting registers, you can reconsider more basic elements of the design. Moving up or down a speed grade, or compressing circuitry in Logic Lock regions are good techniques for investigating performance.

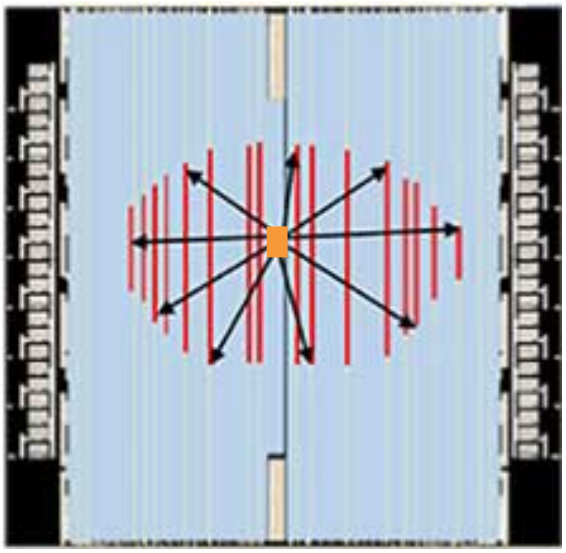
2.1.3. Compile Components Independently

To identify and optimize performance bottlenecks early, you can compile the design subcomponents as stand-alone entities. Individual component compilation allows you to test and optimize components in isolation, without the runtime and complexities of the entire system.

Establish a margin for the speed you require for each component. For example, when targeting a 20% timing margin, a component with 19.5% margin is a failure. Base your timing margin targets on the component 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 can erode.

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

Figure 3. M20K Spread in Chip Planner



The Chip Planner system view indicates nothing 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 becomes unsatisfactory when unrelated logic cells occupy the intervening area. Restructuring this block to physically distribute the control logic better relieves the high-level problem.

2.1.4. Optimize Sub-Modules

During design optimization, you can isolate the critical path in one or two sub-modules of 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.

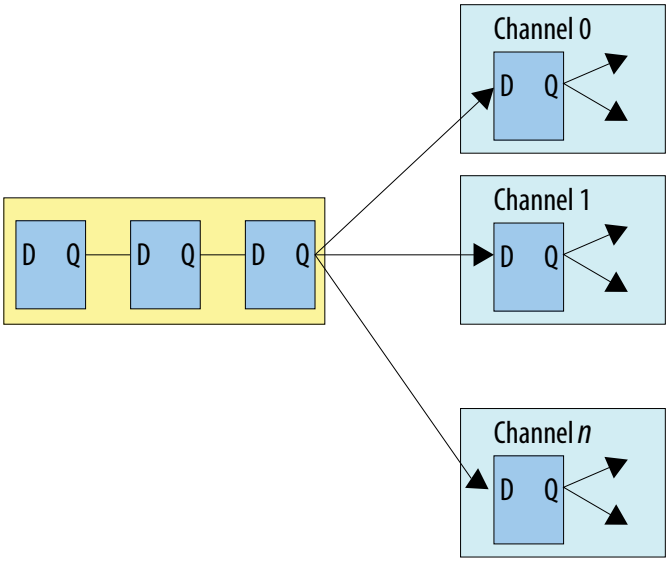
2.1.5. Avoid Broadcast Signals

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.

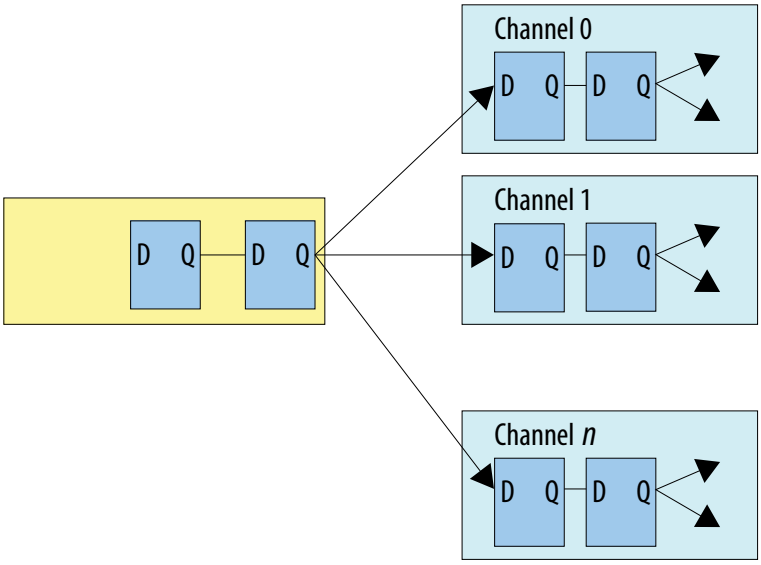
In Sub-Optimal Pipelining of Broadcast Signals, the yellow box highlights registers inserted in a module to help with timing. The block broadcasts the output to several transceiver channels. These extra registers may not improve timing sufficiently because the final register stage fans out to destinations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

Figure 4. Sub-Optimal Pipelining of Broadcast Signals



Optimal Pipelining of Broadcast Signals shows a better approach with duplicating the last pipeline register, and then placing 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.

Figure 5. Optimal Pipelining of Broadcast Signals



In addition to duplicating the last pipeline register, you can 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.

Related Information

Setting the dont_merge Synthesis Attribute on page 100

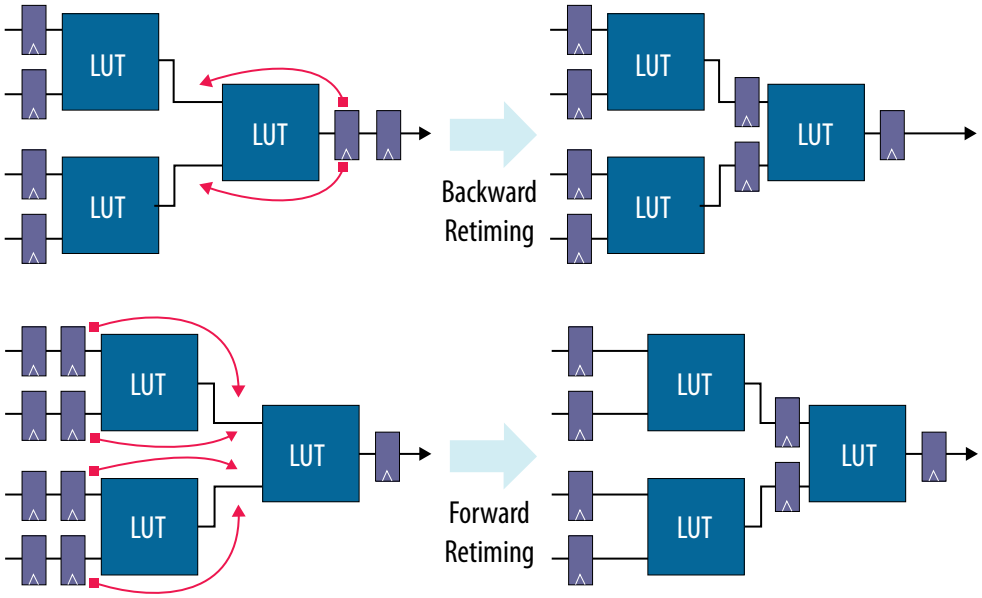
2.2. Hyper-Retiming (Facilitate Register Movement)

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

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

In Moving Registers across LUTs, registers on the left show before retiming, with the worst case delay of two LUTs. Registers on the right show after retiming, with the worst case delay of one LUT.

Figure 6. Moving Registers across LUTs



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

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

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

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

Related Information

- Reset Strategies on page 12
- Timing Constraint Considerations on page 19
- Hyper-Pipelining (Add Pipeline Registers) on page 29
- Retiming Restrictions and Workarounds on page 98

2.2.1. Reset Strategies

This section recommends techniques to achieve maximum performance when using reset signals. To hold your design in reset until configuration is complete, you must implement the Reset Release Intel FPGA IP, or the `INIT_DONE` signal (routed back in through a pin). Refer to the *Intel Agilex 7 Configuration User Guide* or *Intel Stratix 10 Configuration User Guide* for more details on reset for your device.

For the best performance, avoid resets (asynchronous and synchronous), except when necessary. Because Hyper-Registers do not have asynchronous resets, the Compiler cannot retime any register with an asynchronous reset into a Hyper-Register location.

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

Related Information

- Synchronous Resets and Limitations on page 134
- Intel Agilex 7 Configuration User Guide

- Intel Stratix 10 Configuration User Guide

2.2.1.1. Removing Asynchronous Resets

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

Verilog HDL and VHDL Asynchronous Reset Examples shows how the asynchronous reset (shown in **bold**) resets all registers in the pipeline, preventing placement in Hyper-Registers.

Table 4. Verilog HDL and VHDL Asynchronous Reset Examples

Verilog HDL

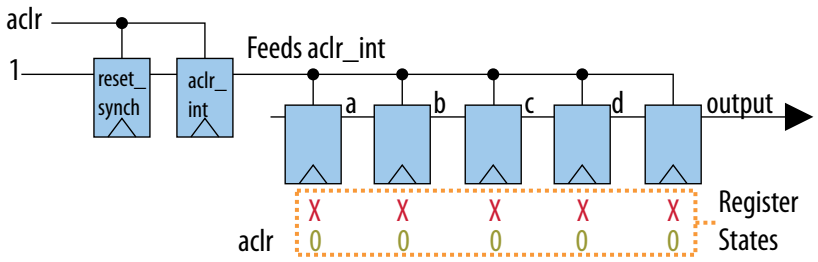
```
always @(posedge clk, aclr)
   if (aclr) begin
      reset_synch <= 1'b0;
      aclr_int <= 1'b0;
   end
   else begin
      reset_synch <= 1'b1;
      aclr_int <= reset_synch;
   end

always @(posedge clk, aclr_int)
// Asynchronous reset===============
   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 = '0') 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
// Asynchronous reset===============
   IF (aclr_int = '0') 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;
```

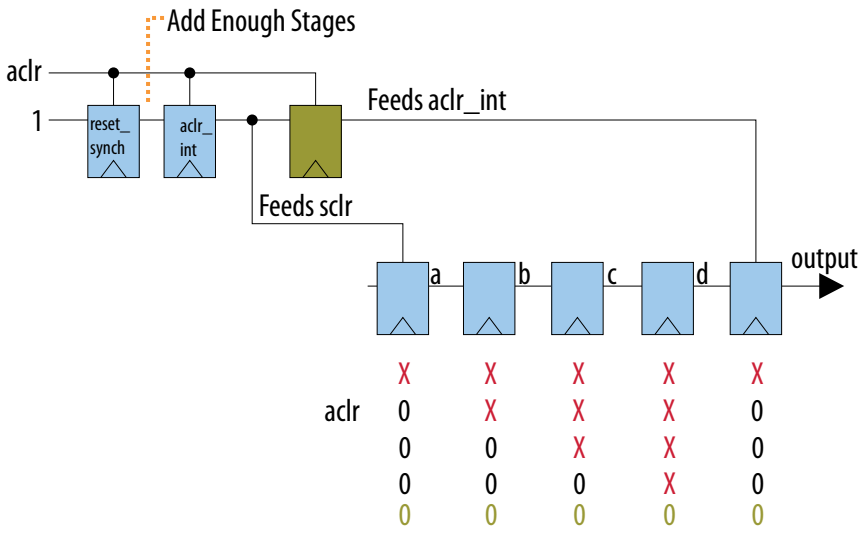
Circuit with Full Asynchronous Reset shows the logic of Verilog HDL and VHDL Asynchronous Reset Examples in schematic form. When `aclr` is asserted, all the outputs of the flops are zeros. Releasing `aclr` and applying two clock pulses causes all flops to enter functional mode.

Figure 7. Circuit with Full Asynchronous Reset



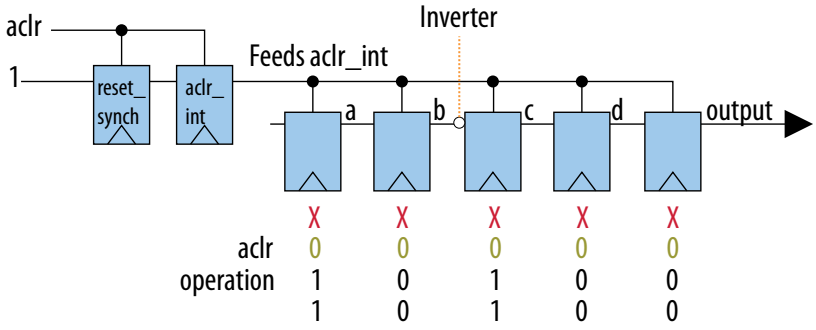
Partial Asynchronous Reset illustrates the removal of asynchronous resets from the middle of the circuit. After a partial reset, if the modified circuit settles to the same steady state as the original circuit, the modification is functionally equivalent.

Figure 8. Partial Asynchronous Reset



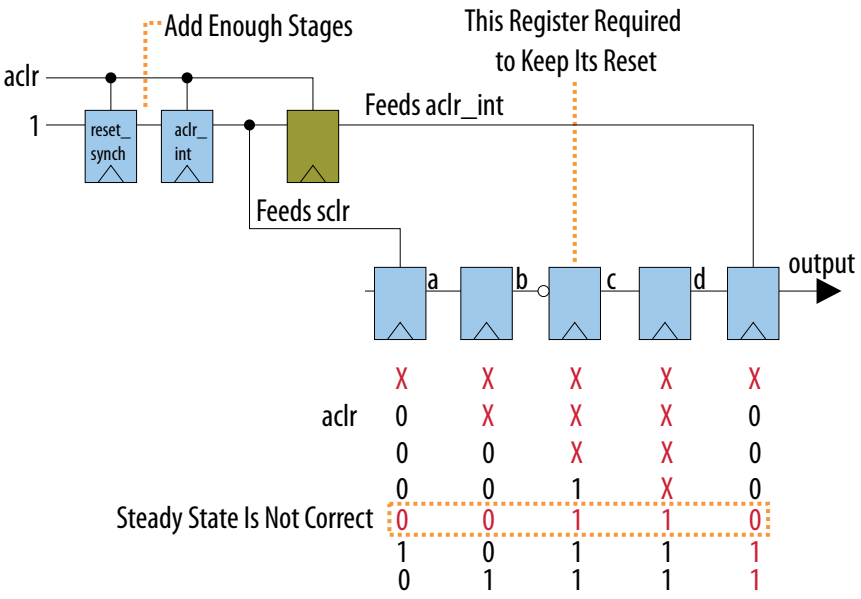
Circuit with an Inverter in the Register Chain shows how circuits that include inverting logic typically require additional synchronous resets to remain in the pipeline.

Figure 9. Circuit with an Inverter in the Register Chain



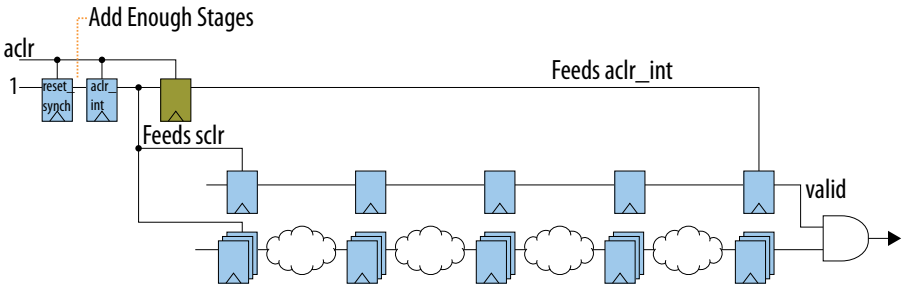
After removing reset and applying the clock, the register outputs do not settle to the reset state. If the asynchronous reset is removed from the inverting register, the circuit cannot remain equivalent with Circuit with an Inverter in the Register Chain after settling out of reset.

Figure 10. Circuit with an Inverter in the Register Chain with Asynchronous Reset



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

Figure 11. Validating the Output to Synchronize with Reset



Verilog HDL Example Using Minimal or No Asynchronous Resets shows Verilog HDL and VHDL examples of Partial Asynchronous Reset. You can adapt this example to your design to remove unnecessary asynchronous resets.

Table 5. Verilog HDL Example Using Minimal or No Asynchronous Resets

Verilog HDL	VHDL

`always @(posedge clk, 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;`
	`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;`

Verilog HDL	VHDL

`aclr_int <= reset_synch_2;`
`end`
 	
`// Asynchronous reset for output register=====`
`always @(posedge clk, posedge aclr_int)`
`if (aclr_int)`
`out <= 1'b0;`
`else`
`out <= d;`
 	
`// Synchronous reset for input register=====`
`always @(posedge clk)`
`if (reset_synch_2)`
`a <= 1'b0;`
`else`
`a <= in;`
 	
`// Naturally resetting registers=====`
`always @(posedge clk) begin`
`b <= a;`
`c <= b;`
`d <= c;`
 	
`end`
	`END IF;`
	`END PROCESS;`
	
	`// Asynchronous reset for output register=====`
	`PROCESS (clk, aclr_int) BEGIN`
	`IF (aclr_int = '1') THEN`
	`output <= '0';`
	`ELSIF rising_edge(clk) THEN`
	`output <= d;`
	`END IF;`
	`END PROCESS;`
	
	`// Synchronous reset for input register=====`
	`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;`
	
	`// Naturally resetting registers=====`
	`PROCESS (clk) BEGIN`
	`IF rising_edge(clk) THEN`
	`b <= a;`
	`c <= b;`
	`d <= c;`
	`END IF;`
	`END PROCESS;`

2.2.1.2. Synchronous Resets on Global Clock Trees

Using a global clock tree to distribute a synchronous reset may limit retiming performance improvements by the Compiler. 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 with fan-out to the routing fabric.

2.2.1.3. Synchronous Resets on I/O Ports

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

2.2.1.4. Duplicate and Pipeline Synchronous Resets

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

2.2.2. Clock Enable Strategies

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

2.2.2.1. Localized Clock Enable

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

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

2.2.2.2. High Fan-Out Clock Enable

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

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

Related Information

Synchronous Resets Summary on page 137

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. Preserving Registers During Synthesis

You can specify entity-level assignments and synthesis attributes to preserve specific registers during synthesis processing.

For example, the **Preserve Registers in Synthesis** assignment preserves the register that you assign during synthesis, without restricting Hyper-Retiming optimization. Similarly, you can specify the `dont_merge` or `preserve_syn_only` synthesis attributes to preserve registers without restricting retiming optimization, as the following example shows.

```
logic  hip_data; /* synthesis preserve_syn_only */
(*preserve_syn_only*) logic hip_data;
```

The **Preserve Registers** assignment also preserves registers, but does not allow Hyper-Retimer optimization of the registers that you assign. This assignment can be useful when you want to preserve a register for debugging observability.

Specify any of the following synthesis preservation assignments by clicking **Assignments ➤ Assignment Editor**, by modifying the `.qsf` file, or by specifying synthesis attributes in your RTL.

Table 6. Synthesis Preserve Options

Assignment	Description	Allows Fitter Optimization?	Assignment Syntax
Preserve Registers in Synthesis	Prevents removal of registers during synthesis without restricting any optimization after synthesis, such as Hyper-Retiming or physical synthesis optimizations.	Yes	• `set_global_assignment -name PRESERVE_REGISTER_SYN_ONLY ON|OFF -entity <entity name>` `set_instance_assignment -name PRESERVE_REGISTER_SYN_ONLY ON|OFF -to <to> -entity <entity name>` • `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. The `PRESERVE_FANOUT_FREE_NODE` assignment cannot preserve a fanout-free register that has no fanout inside the Verilog HDL or VHDL module in which you define it. To preserve these fanout-free registers, implement the `noprune` pragma in the source file: `(*noprune*)reg r;` If there are multiple instances of this module, with only some instances requiring preservation of the fanout-free register, set a dummy pragma on the register in the HDL and also set the `PRESERVE_FANOUT_FREE_NODE` assignment. This dummy pragma allows the register synthesis to implement the assignment. For	Yes	• `set_instance_assignment -name PRESERVE_FANOUT_FREE_NODE ON|OFF -to <to> -entity <entity name>` • `noprune on` (see Verilog or VHDL synthesis attribute for syntax)

continued...

Assignment	Description	Allows Fitter Optimization?	Assignment Syntax
	example, set the following dummy pragma for a register r in Verilog HDL: `(*dummy*)reg r;`		
Preserve Registers	Prevents removal and sequential optimization of assigned registers during synthesis. Sequential netlist optimizations can eliminate redundant registers and registers with constant drivers.	No	• `set_global_assignment -name PRESERVE_REGISTER ON|OFF -entity <entity name>` `set_instance_assignment -name PRESERVE_REGISTER ON|OFF -to <to> -entity <entity name>` • `preserve`, `syn_preserve`, or `keep on` (synthesis attributes)

2.2.4. Timing Constraint Considerations

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

2.2.4.1. Optimize Multicycle Paths

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

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

2.2.4.2. Overconstraints

Overconstraints direct the Fitter to spend more time optimizing specific parts of a design. Overconstraints are appropriate in some situations to improve performance. However, because legacy overconstraint methods restrict retiming optimization, Intel Hyperflex architecture FPGAs 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. Overconstraints Syntax (Allows Hyper-Retiming)

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

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

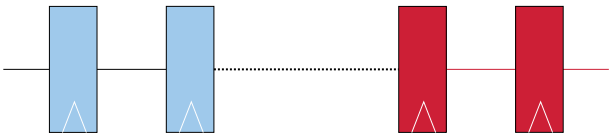
```
### Over Constraint ###
# if {$::quartus(nameofexecutable) == "quartus_fit"} {
#     set_min_delay 0.050 -from [get_clocks {CPRI|PHY|TRX*|*|rx_pma_clk}] -to \
     [get_clocks {CPRI|PHY|TRX*|*|rx_clkout}]
# }
```

2.2.5. Clock Synchronization Strategies

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

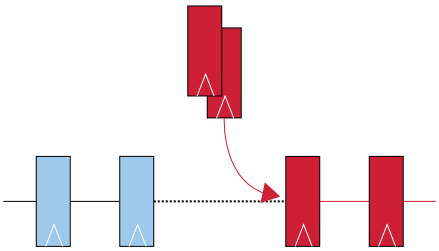
Simple Clock Domain Crossing shows a simple synchronization scheme with a path from one register of the first domain (blue), directly to a register of the next domain (red).

Figure 12. Simple Clock Domain Crossing



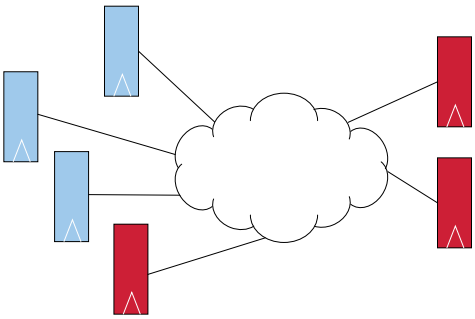
To add latency in the red domain for retiming, add the registers as Adding Latency to Simple Clock Domain Crossing shows.

Figure 13. Adding Latency to Simple Clock Domain Crossing



Clock Domain Crossing at Multiple Locations shows a domain crossing structure that is not optimum in Intel Hyperflex architecture FPGAs, but exists in designs that target other device families. The design contains some combinational logic between the blue clock domain and the red clock domain. The design does not properly synchronize the logic and you cannot add registers flexibly. The blue clock domain drives the combinational logic and the logic contains paths that the red domain launches.

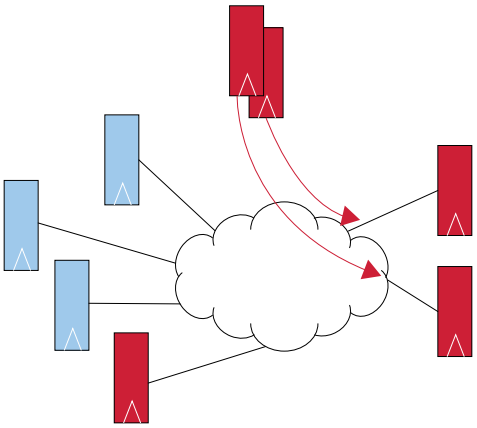
Figure 14. Clock Domain Crossing at Multiple Locations



Adding Latency at Multiple Clock Domain Crossing Locations shows adding latency at the boundary of the red clock domain, without adding registers on a red to red domain path. Otherwise, the paths become unbalanced (with respect to the cycle behavior on

clock edges), potentially changing design functionality. Although possible, adding latency in this scenario is risky. Thoroughly analyze the various paths before adding latency.

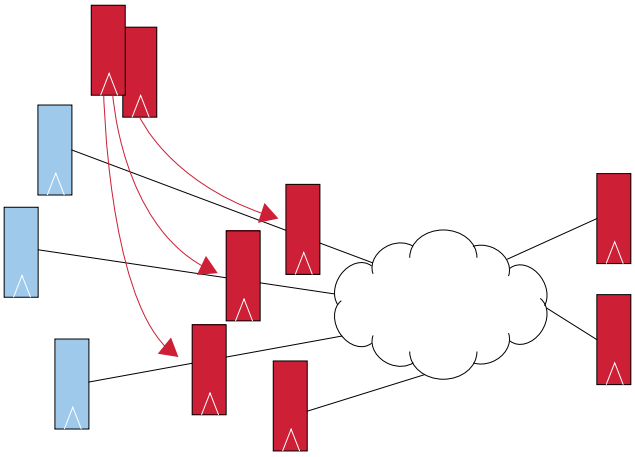
Figure 15. Adding Latency at Multiple Clock Domain Crossing Locations



For Intel Hyperflex architecture FPGAs, synchronize the clock crossing paths before entering combinational logic. Adding latency is then more simple when you compare with the previous example.

Clock Domain Synchronization Improvement shows blue domain registers synchronizing to the red domain before entering the combinational logic. This method allows safe addition of pipeline registers in front of synchronizing registers, without contacting a red to red path inadvertently. Implement this synchronization method for the highest performance in Intel Hyperflex architecture FPGAs.

Figure 16. Clock Domain Synchronization Improvement



2.2.5.1. Clock Domain Crossing Constraint Guidelines

You must apply appropriate timing constraints to any multi-bit clock domain crossing. The `set_false_path` constraint has a higher precedence than all other path-based constraints. Therefore, when a clock domain crossing has a `set_false_path` constraint, timing analysis can ignore other lower precedence constraints like skew.

Follow these guidelines to properly constrain a clock domain crossing:

- Review the SDC timing constraints to ensure that no `set_false_path` constraint exists between the two clock domains.
- To remove any false paths between two clock domains from setup and hold timing analysis, apply the `set_clock_groups` constraint, rather than `set_false_path` constraint. `set_clock_groups` has a lower precedence than `set_false_path`.
- Constrain the paths between the two clock domains with `set_net_delay` to make the nets as short as possible.
- Constrain the nets between the two clock domains with `set_max_skew`. You can view the results in comparison to your constraint in the Timing Analyzer reports.

The following shows example constraints for a clock domain crossing between `data_a` in `clk_a` clock domain, and `data_b` in `clk_b` clock domain:

```
create_clock -name clk_a -period 4.000 [get_ports {clk_a}]
create_clock -name clk_b -period 4.500 [get_ports {clk_b}]
set_clock_groups -asynchronous -group [get_clocks {clk_a}] -group \
     [get_clocks {clk_b}]
set_net_delay -from [get_registers {data_a[*]}] -to [get_registers \
     {data_b[*]}] -max -get_value_from_clock_period dst_clock_period \
     -value_multiplier 0.8
set_max_skew -from [get_keepers {data_a[*]}] -to [get_keepers {data_b[*]}] \
     -get_skew_value_from_clock_period src_clock_period \
     -skew_value_multiplier 0.8
```

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.

The default metastability synchronizer chain length for Intel Hyperflex architecture FPGAs is three. The Critical Chain report marks the registers that metastability requires with `REG (Metastability required)` text.

If your design includes two-register chains as synchronizers, you can specify the following setting to modify the default chain length from 3 to 2:

1. Click **Assignments ➤ Settings**.
2. Click **Compiler Settings** under **Category**.
3. Click the **Advanced Settings (Synthesis)** button.
4. For **Synchronization Register Chain Length**, type 2 in the **Setting** column.

Alternatively, you can specify this setting in the `.qsf` file:

```
set_instance_assignment -name SYNCHRONIZATION_REGISTER_CHAIN_LENGTH 2 \
     -to * -entity <top_module_name>
```

2.2.7. Initial Power-Up Conditions

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

Therefore, do not rely on initial conditions when designing for Intel Hyperflex architecture 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 Memory Conditions

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

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

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

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

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

2.2.7.2. Initial Conditions and Retiming

The initial power-up conditions can limit the Compiler's ability to perform logic optimization during synthesis, and to 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 17. 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')`.

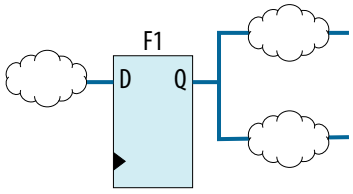
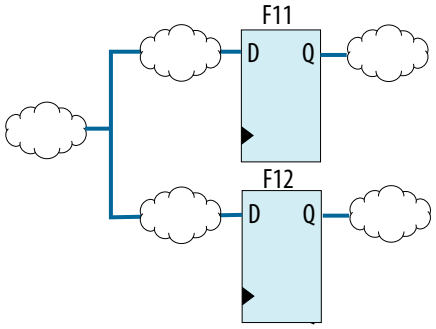


Figure 18. Circuit After Retiming Forward

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 7. 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. Initial Conditions and Hyper-Registers

The Intel Hyperflex architecture routing fabric includes Hyper-Registers throughout to achieve the highest performance. However, unless properly accounted for, initial power-up conditions can limit the Compiler's ability to retime registers into Hyper-Registers. Rather than relying on initial conditions, use a single reset signal to place the design in a known, functional state until all the interfaces have powered up, locked, and trained.

If you must rely on initial conditions, and your system requires that all registers start synchronously, the use of clock gating is recommended. Because Hyper-Registers lack a reset or enable signal, you cannot initialize them to a specific value using a reset control signal. Intel Stratix 10 Hyper-Registers can power up to 0 or 1. Intel Agilex 7 Hyper-Registers power up to 1 during configuration. When the system starts up, right after configuration, the initial values are present without the need for an explicit reset.

Clock Gating For ALM and Hyper-Registers

Independent signals drive the internal clock controls of ALM registers and Hyper-Registers in Intel Hyperflex architecture FPGAs. During the configuration process, the registers become active row by row (as opposed to device wide). In addition, ALM register clocks can potentially enable independently from Hyper-Register clocks. If the design clock is free running, this can cause potential race conditions between rows and between ALM registers and Hyper-Registers. These conditions can result in potential overwrite of initial conditions. To avoid these scenarios, gate the clock until after all clock controlling logic de-asserts, and all registers are active.

2.2.7.3.1. Implementing Clock Gating

To implement clock gating, you access the USER_CLKGATE signal by use of the following Intel FPGA IP available in the Intel Quartus Prime software:

- Reset Release Intel FPGA IP—holds your design in reset until configuration is complete by gating clocks, resets, or write enables. This IP outputs the `nINIT_DONE` signal. When `nINIT_DONE` is low, the device is no longer in configuration mode.
- Clock Control Intel FPGA IP—uses the inverted `nINIT_DONE` signal as a clock enable signal.

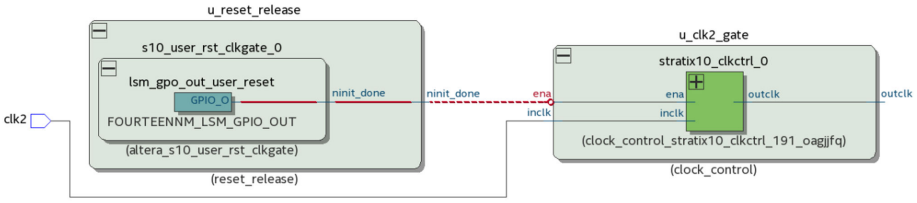
Follow these steps to implement clock gating:

1. Open a design in the Intel Quartus Prime software.
2. In IP Catalog, type `reset release` in the search field, and double-click the **Reset Release Intel FPGA IP**.
3. Specify appropriate parameters for your configuration in the parameter editor, and then click **Generate HDL**.
4. Repeat steps 2 and 3 to add the Clock Control Intel FPGA IP to your project. Prior to IP generation, specify the following options for the IP in the parameter editor:
   - Under **Clock Gating**, turn on the **Clock Enable** option.
   - For **Clock Enable Type**, select **Root Level**.
   - For **Enable Register Mode**, select **Negative Latch**.
5. Connect the Reset Release and Clock Control Intel FPGA IP together:

- To gate the clocks, use inverted `nINIT_DONE` as the enable input to the Clock Control Intel FPGA IP.
- If initial conditions are required, Intel recommends that the Clock Control Intel FPGA IP also use root clock gating.

The following figure shows proper connections between the Reset Release and Clock Control Intel FPGA IP to ensure accurate initial conditions after configuration:

Figure 19. Connections between the Reset Release (reset_release) and Clock Control (clock_control) Intel FPGA IP Cores



The Clock Control Intel FPGA IP uses the `ena` signal to perform the clock gating function. The clock signal on the output of the Clock Control Intel FPGA IP is then safe for use with the initialized registers (ALM and Hyper-Registers).

2.2.7.3.2. Intel Quartus Prime Settings for Initial Conditions

You can use the following Intel Quartus Prime settings that have an impact on initial conditions.

Power-Up Don't Care Logic Option

You can enable the retiming of more registers by making sure that the Power-Up Don't Care Logic option (`ALLOW_POWER_UP_DONT_CARE`) is On. This option specifies that registers without explicit initial conditions in the RTL power-up to don't care. This option is set to On by default.

```
set_global_assignment -name ALLOW_POWER_UP_DONT_CARE ON
```

Any initial conditions that you specify in the RTL, or any initial conditions that are implied (because of language specifications or FSM initial states) still apply when using `ALLOW_POWER_UP_DONT_CARE`.

Ignoring Initial Conditions

Because of factors such as language specifications and conservative synthesis on structures such as state machines, it is possible for initial conditions to appear on certain registers, even though you expect no initial conditions to appear. Such initial conditions can potentially cause performance limitations. If you verify that removing such initial conditions is functionally correct, you can specify the following `.qsf` assignment to remove those initial conditions:

```
set_instance_assignment -name IGNORE_REGISTER_POWER_UP_INITIALIZATION ON \
     -to <instance name>
```

The Synthesis report identifies registers affected by `IGNORE_REGISTER_POWER_UP_INITIALIZATION` in the "Registers with Power-Up Settings Ignored" report.

2.2.7.4. Retiming Reset Sequences

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

Modifying the Reset Sequence

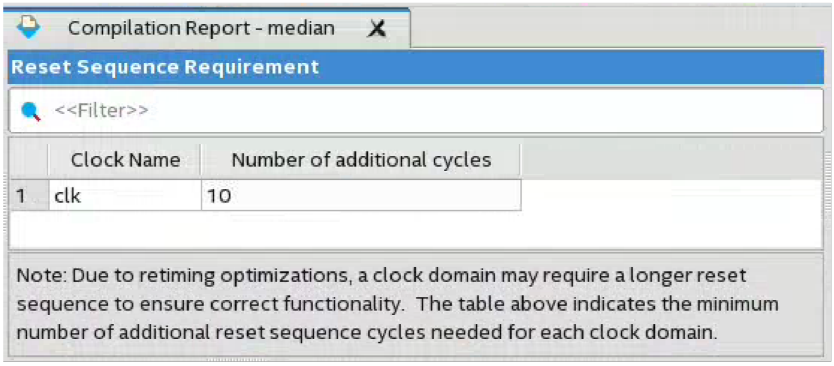
Follow these recommendations to maximize operating frequency of resets during retiming:

- Remove `sclr` signals from all registers that reset naturally. This removal allows the registers to move freely in the logic during retiming.
- Make sure that the `ALLOW_POWER_UP_DONT_CARE` global assignment is set to `ON`. This setting maximizes register movement.
- Compute and add to the reset synchronizer the relevant amount of extra clock cycles due to c-cycle equivalence.

Adding Clock Cycles to Reset

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

Figure 20. Reset Sequence Requirement Report

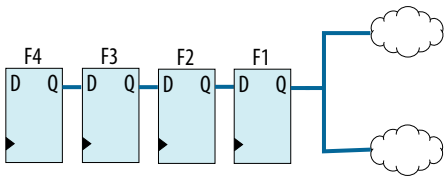


Register duplication into multiple branches has a c-cycle of 1. Regardless of the number of duplicate registers, the register is always one connection away from its original source. After one clock cycle, all the branches have the same value again.

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

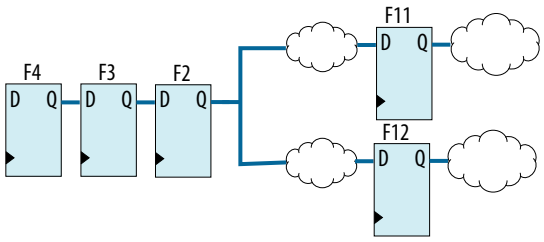
Pipelining and Register Duplication shows pipelining of registers with potential for forward retiming. The c-cycle value equals 0.

Figure 21. Pipelining and Register Duplication



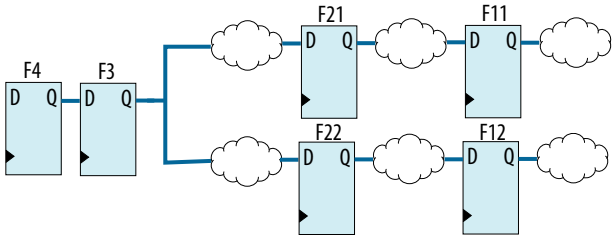
Impact of One Register Move shows a pipelining of registers after forward retiming of one register. Because the c-cycle value equals 1, the reset sequence for this circuit requires one additional clock cycle for functional equivalence after reset.

Figure 22. Impact of One Register Move



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

Figure 23. Impact of Two Register Moves



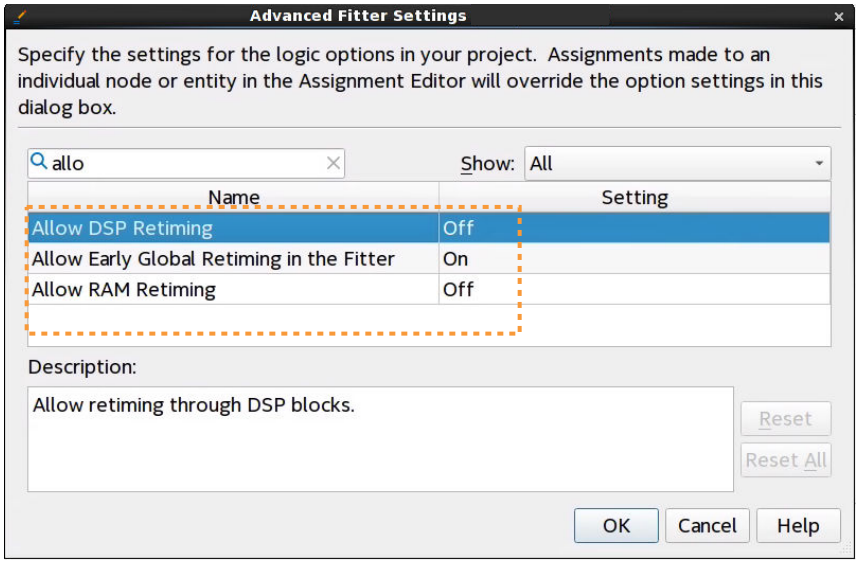
Each time a register from the pipeline moves into the logic, the register duplicates and the C-cycle value of the design increases by one.

2.2.8. Retiming through RAMs and DSPs

The Compiler can use Hyper-Registers on paths to and from RAM or DSPs, regardless of any RAM or DSP retiming setting. However, turning on the **Allow RAM Retiming** or **Allow DSP Retiming** options allows the Compiler to retime registers *over* RAM and DSPs. When the RAM or DSP retiming settings are disabled (the default), the Compiler does not retime registers over RAMs or DSPs.

To access these settings, click **Assignments ➤ Settings ➤ Compiler Settings ➤ Advanced Settings (Fitter)**.

Figure 24. Register Optimization Settings



The following diagrams illustrate the impact of these settings:

Figure 25. RAM or DSP Timing Path

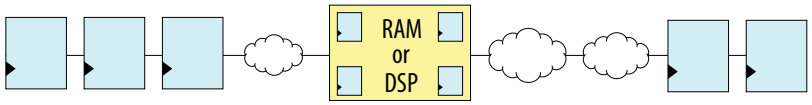


Figure 26. Default RAM or DSP Retiming Optimization

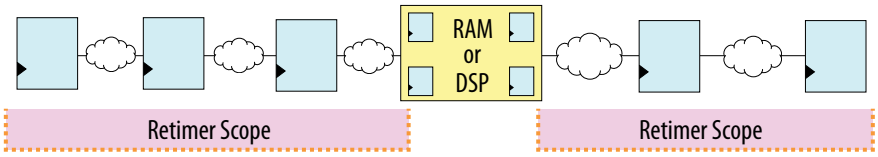
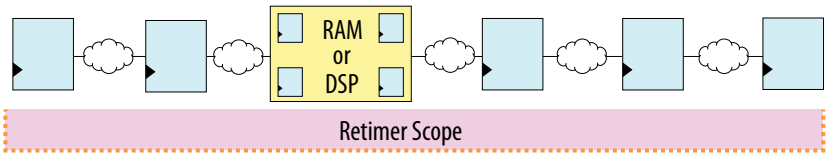


Figure 27. Allow RAM Retiming or Allow DSP Retiming



2.3. Hyper-Pipelining (Add Pipeline Registers)

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

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

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

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

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

After you modify the RTL and place the appropriate number of pipeline stages at the boundaries of each clock domain, the 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 in comparison with conventional pipelining.

Related Information

- Appendix A: Parameterizable Pipeline Modules on page 132
- Precomputation on page 51

2.3.1. Conventional Versus Hyper-Pipelining

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

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

Figure 28. Conventional Pipelining User Modifications

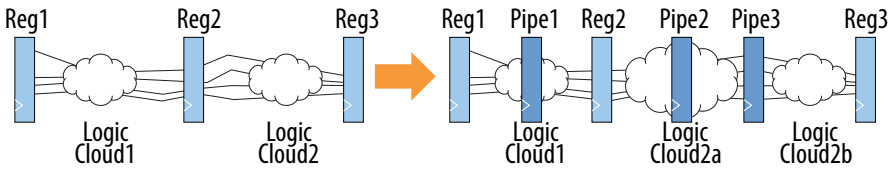


Figure 29. Hyper-Pipelining User Modifications

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.

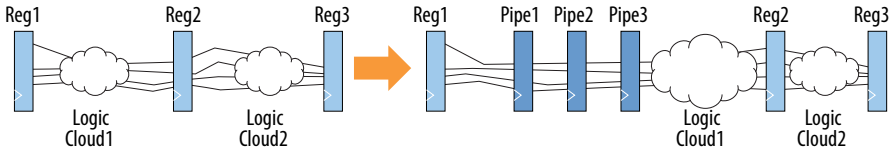
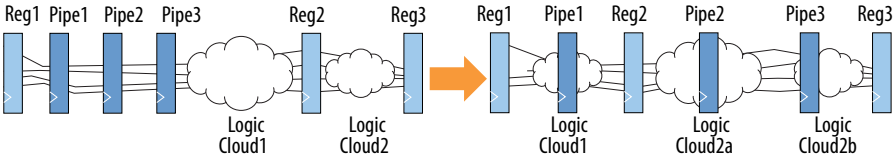


Figure 30. Hyper-Pipelining and Hyper-Retiming Implementation

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



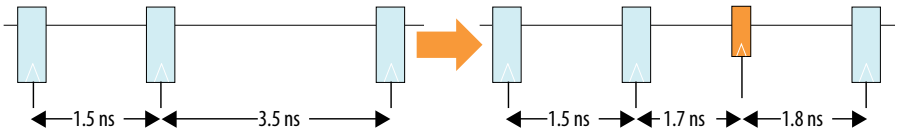
The resulting implementation in the Hyper-Pipelining flow differs from the conventional pipelining flow by the location of the Pipe 3 register. Because the Compiler is aware of the current circuit implementation, including routing, the Compiler can more effectively locate the aggregate registers to meet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

2.3.2. Pipelining and Latency

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

Figure 31. Hyper-Pipeline Reduced Latency

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



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

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

2.3.2.1. Pipelining at Variable Latency Locations

Commonly, FPGA designs include some locations that are insensitive to additional latency, such as at clock domain boundaries, connections between major functional blocks, and false paths. Best design practices recommend adding pipeline stages at clock domain boundaries or between major functional blocks to improve timing. However, adding excessive pipeline stages can also bloat area usage, and increase routing congestion.

The current version of the Intel Quartus Prime software includes new features to help improve timing performance for design paths that are insensitive to additional latency. The Hyper-Retimer can now automatically add pipeline stages on false paths that you tag as latency-insensitive, and also insert the appropriate number of pipeline stages at the registers you specify. The Hyper-Retimer retimes the added registers into timing-critical parts of the design. The number of pipeline stages that the Hyper-Retimer adds can change for each compilation, or any time you change the design.

Note:

- If you do not specify latency-insensitive false paths or use autopipelining, the Hyper-Retimer output netlist is cycle-equivalent to your RTL.
- If you specify latency-insensitive false paths or use autopipelining, the Hyper-Retimer output netlist is not cycle-equivalent to your RTL. Therefore, your simulation and verification environments must accommodate variations in the circuit latency to use these techniques.

2.3.2.1.1. Specifying a Latency-Insensitive False Path

You can specify a latency-insensitive false path to allow the Hyper-Retimer to automatically add pipeline stages to a path. Specify latency-insensitive false paths only on cross-clock domain paths, such as between a low-speed configuration clock domain, and a high-speed data path clock domain, as in a signal processing design.

Specify the `latency_insensitive` option for the `set_false_path` exception to designate a false path as latency-insensitive. Specify the clock names for the `from` and `to` options, as the following example shows:

```
set_false_path -latency_insensitive -from  [get_clocks {clock_a}]  \
     -to  [get_clocks {clock_b}]
```

Although not a syntax error to specify register, cell, net, pin, or keeper name for the `from` or `to` options, the Compiler interprets the false path as a retiming restriction, and prevents the Hyper-Retimer from retiming those endpoints. There is no benefit to using the `latency_insensitive` option on a register-to-register false path.

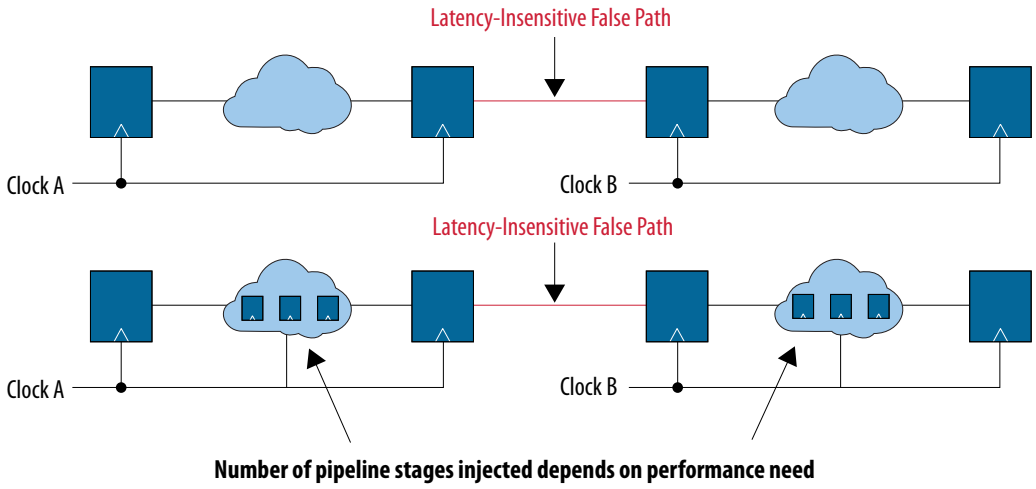
Note:

The `set_false_path` constraint has higher precedence than all other path-based SDC constraints. If your latency-insensitive false path is on a clock domain transfer that includes FIFOs, bus synchronizers, or other cross-domain circuits that have path-based constraints (such as `set_max_skew`, `set_max_delay`, or `set_min_delay`), a clock-to-clock `set_false_path` constraint overrides these constraints. If you constrain a clock domain crossing with any path-based constraint, the `latency_insensitive` option overrides these constraints. Only use a latency-insensitive false path on clock domain crossing paths that you have actually cut from timing analysis.

In the following figure, the top diagram represents the design RTL, indicating the false path tagged as latency-insensitive false path. The bottom diagram shows how the Hyper-Retimer adds pipeline stages on the other side of the registers at endpoints of the latency-insensitive false path.

The Hyper-Retimer can add registers to the input of the source of the latency-insensitive false path, and to the output of the destination of the latency-insensitive false path. The Hyper-Retimer then retimes the registers backward and forward through the two clock domains.

Figure 32. Effect of Latency-Insensitive False Path on Circuit



The Hyper-Retimer analyzes the performance of each cross-clock-domain path separately to determine the number of stages to automatically add. The Hyper-Retimer may insert different numbers of stages on each cross-clock-domain path.

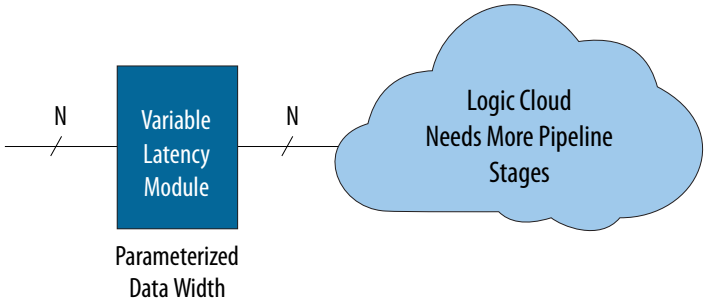
For example, a bus crossing a clock domain that is cut with the `latency_insensitive` option can have different latencies for each bit in the bus after the Hyper-Retimer runs. Therefore, ensure that the data crossing the clock domain remains constant for many clock cycles to ensure it becomes constant at the destination. For example, this can occur with a bus with different latencies on each bit.

The compilation report does not show the number of stages that the Hyper-Retimer inserts at a latency-insensitive false path. However, you can examine the connectivity in the timing netlist after the Hyper-Retimer finishes to determine the number of stages.

2.3.2.2. Automatic Pipeline Insertion

Automatic pipeline insertion allows the Hyper-Retimer to insert a number of pipeline stages at a location you specify in your design. You can specify the maximum number of pipeline stages to insert at each particular register.

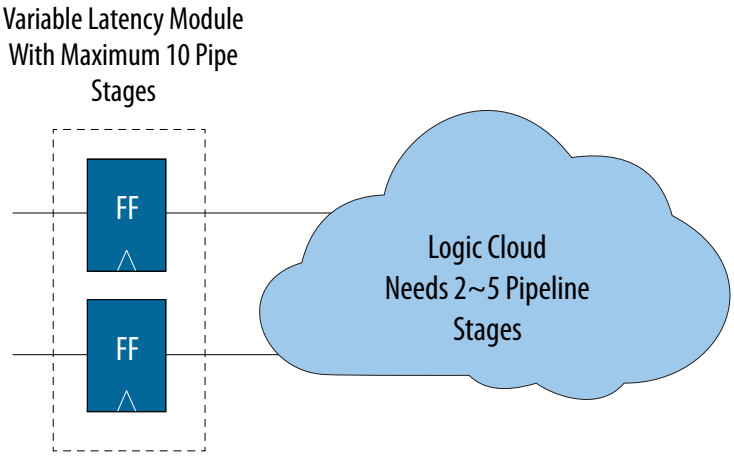
Figure 33. Typical Use of Variable Latency Module



The Intel Quartus Prime software includes the Variable Latency Module template (`hyperpipe_vlat`) that simplifies implementation. Alternatively, you can implement automatic pipeline insertion using a combination of `.qsf` assignments.

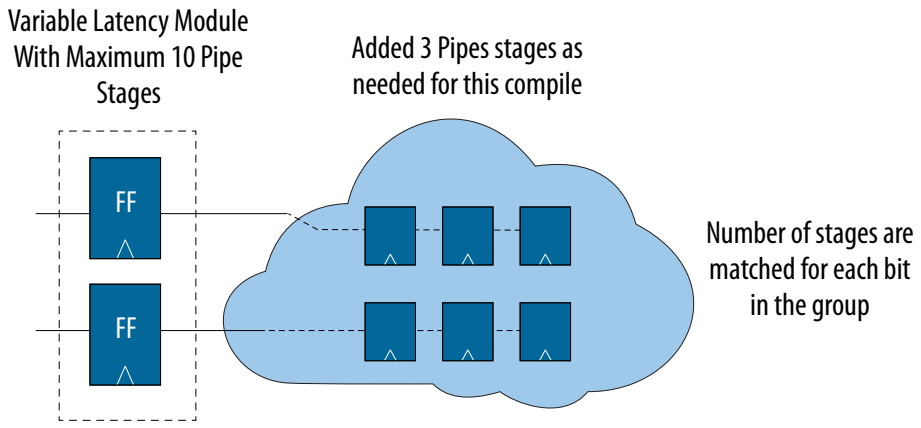
When you instantiate the `hyperpipe_vlat` module, and the **Enable Auto-Pipelining** (`HYPER_RETIMER_ENABLE_ADD_PIPELINING`) option remains enabled, the Hyper-Retimer adds the appropriate number of additional pipeline stages at the specified register during retiming, up to the maximum that you specify. This setting is enabled by default. Click **Assignments ➤ Settings ➤ Compiler Settings ➤ Advanced Settings (Fitter)** to access this setting.

Figure 34. Variable Latency Module with Maximum of Ten Pipeline Stages



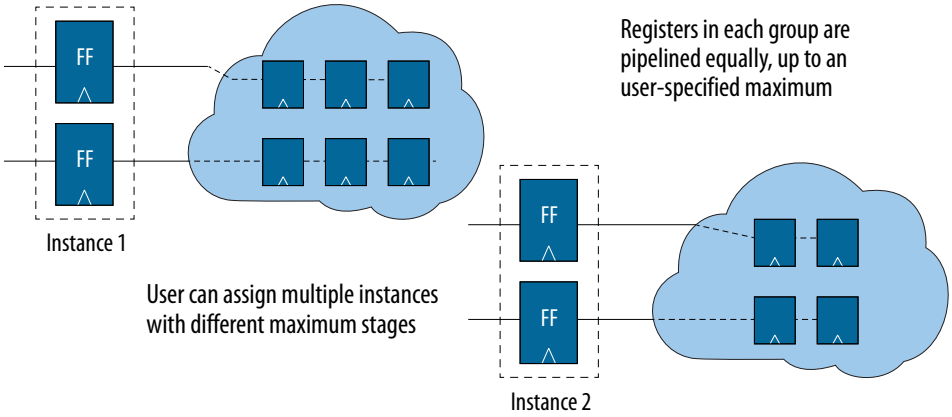
For example, if you specify a maximum of 10 pipeline stages, the Hyper-Retimer may determine that only three additional pipeline stages are necessary to maximize the timing performance. The Hyper-Retimer adds only the appropriate number of pipeline stages necessary.

Figure 35. Hyper-Retimer Adds Only Additional Stages Needed



You can specify different numbers of pipeline stages for separate instances of the `hyperpipe_vlat` module, as the following diagram illustrates:

Figure 36. Different Maximum Pipeline Stages Per Module



The following steps describe how to implement automatic pipeline insertion in detail:

- Step 1: Create the Variable Latency Module on page 35
- Step 2: Instantiate the Variable Latency Module on page 37
- Step 3: Verify Automatic Pipeline Insertion Option on page 40
- (Optional) Auto-Pipeline Insertion without a Variable Latency Module on page 41

Valid values for the maximum number of additional stages are 1 to 100, inclusive.

2.3.2.2.1. Step 1: Create the Variable Latency Module

You can use the Hyper-Pipelining Variable Latency Module template (`hyperpipe_vlat`), available in the Intel Quartus Prime software, to create the variable latency module for use in automatic pipeline insertion.

The hyperpipe_vlat module contains a single pipeline stage. The Hyper-Retimer adds the same number of pipeline stages to all the bits in one instance of the hyperpipe_vlat module. The module includes the following customizable parameters:

- WIDTH—specifies the width of the bus, with a default value of one.
- MAX_PIPE—specifies the maximum number of pipeline stages the Hyper-Retimer can add at that instance. The value must be between 1 and 100, inclusive. The default value is 100.

Figure 37. Hyper-Pipelining Variable Latency Module Templates

```
// Quartus Prime Verilog Template
//
// Hyper-Pipelining Variable Latency Module

module hyperpipe_vlat
#(parameter WIDTH = 1, parameter MAX_PIPE = 100
(
   input clk,
   input  [WIDTH-1:0] din,
   output [WIDTH-1:0] dout
);
```

```
-- Quartus Prime VHDL Template
--
-- Hyper-Pipelining Variable Latency Module

library ieee;
use ieee.std_logic_1164.all;

entity hyperpipe_vlat is

   generic
   (
      WIDTH     : integer := 1;
      MAX_PIPE  : integer := 100 -- Valid r
   );

   port
   (
end entity;
```

Follow these steps in the Intel Quartus Prime software to create a variable latency module:

1. Click **File ➤ New** and create a new Verilog HDL or VHDL design file.
2. Right-click in the new file, and then click **Insert Template**.
3. Select the **Verilog HDL (or VHDL) ➤ Full Designs ➤ Pipelining ➤ Hyper-Pipelining Variable Latency Module**, and then click **Enter** and **Close**. The module template inserts into the file.
4. Specify appropriate values for the `WIDTH` and `MAX_PIPE` parameters when you instantiate the `hyperpipe_vlat` module.
5. Save the file.

2.3.2.2.2. Step 2: Instantiate the Variable Latency Module

You can use the Fast Forward Compilation feature to help identify suitable locations for automatic pipeline insertion. The following locations are typically suitable for automatic pipeline insertion:

- Clock boundaries that are transferring constantly changing data allow the Fitter to spread the blocks far apart when advantageous.
- Locations adjacent to a complex combinational function are suitable for a function that has trouble meeting timing.
- Locations between two independent functional blocks on the same clock domain allow the Fitter to spread the blocks far apart when advantageous.

Instantiating Variable Latency at Clock Domain Boundaries

Fast Forward Compilation recommends adding pipeline stages at clock domain boundaries, where additional latency can be simple to accommodate. Instantiating variable latency at clock boundaries allows the Fitter to spread these blocks far apart when advantageous. In such cases, you can simply instantiate the `hyperpipe_vlat` module either before or after a synchronizer or FIFO. Instantiate `hyperpipe_vlat` with no other registers or logic between the `comb` function and `hyperpipe_vlat`, with respect to netlist connectivity.

This instantiation allows the Hyper-Retimer to automatically insert just enough pipeline registers to meet the timing requirement. A latency-insensitive false path is inappropriate in this case because the data is constantly changing.

Instantiating Variable Latency Adjacent to Complex Combinational Functions

You can instantiate the `hyperpipe_vlat` module adjacent to a complex combinational module to allow the Hyper-Retimer to insert just enough registers to meet the timing requirement. Instantiate the `hyperpipe_vlat` module after the complex combinational module because backwards retiming does not require additional reset cycles to accommodate any initial conditions. You cannot control whether a register in the `hyperpipe_vlat` module retimes forward, into the logic following it, or backward, into the combinational module.

Instantiating Variable Latency Between Independent Functional Blocks

You can instantiate the `hyperpipe_vlat` module between two independent functional blocks in the same clock domain. This instantiation allows the functional blocks to spread apart during placement, while only adding the number of pipeline stages between the blocks necessary to meet the timing requirements.

Note: Do not place the variable latency module directly adjacent to a partition boundary. Rather, place a register between the latency module and the partition port to separate them. The variable latency module does not autopipeline during retiming if you place the `hyperpipe_vlat` module immediately adjacent to a partition boundary, as Improper Variable Latency Module Placement Prevents Autopipelining shows.

Figure 38. Improper Variable Latency Module Placement Prevents Autopipelining

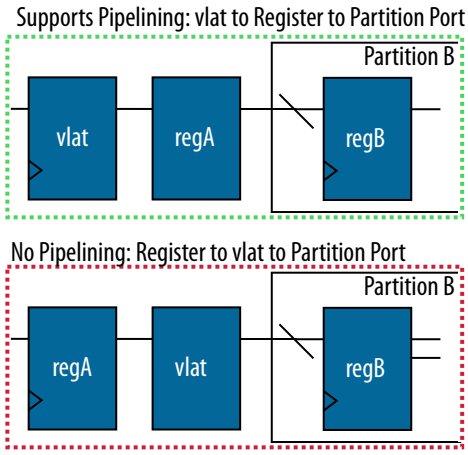
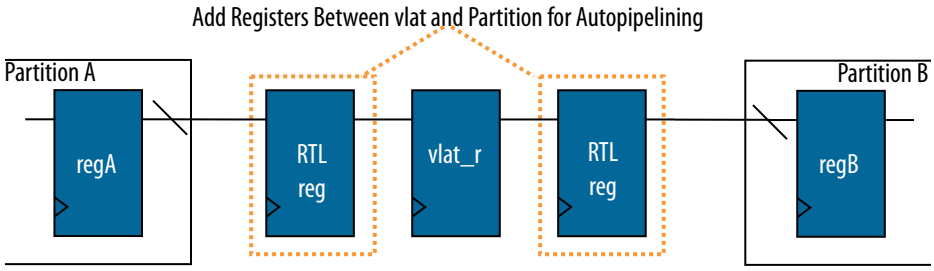


Figure 39. Place Registers Between Variable Latency Module and Partition to Allow Autopipelining



Applying False Path or Exception Constraints

If instantiating the `hyperpipe_vlat` module between independent functional blocks, you must add a false path (`set_false_path`) or other timing exception to allow the connected blocks to float apart during placement. You apply the false path or exception to the `vlat_r` register in the `hyperpipe_vlat` module. By placing the timing exception within a conditional if statement, the Timing Analyzer does not use the exception when the vlat adds registers, nor during final timing sign-off, ensuring each register meets the clock requirement.

Without the corresponding false path or exception constraint, `hyperpipe_vlat` provides little benefit. Without a false path or exception constraint, the Hyper-Retimer only recognizes a single pipeline stage in `hyperpipe_vlat` during placement and routing. The Hyper-Retimer only adds the additional pipeline stages after placement and routing completes. The Compiler tends to place two functional blocks connected by a single pipeline stage close together, unless the paths between them are cut.

The following lines show the appropriate `.sdc` syntax to apply a `set_false_path` exception for the `hyperpipe_vlat` instance at `my|top|design|hyperpipe_vlat_inst`. Add similar lines to your `.sdc` for any `hyperpipe_vlat` instances that connect to independent functional blocks:

```
if { ! [is_post_route] } {
     set_false_path -to my|top|design|hyperpipe_vlat_inst|vlat_r[*]}
```

Furthermore, if you limit the number of pipelines with the `MAX_PIPE` parameter, consider applying the `max_delay` or `multicycle` exception, rather than a `set_false_path` exception. If there is a `MAX_PIPE` constraint on the vlat instance, then a `set_false_path` exception may move the logic so far away, that the `MAX_PIPE` constraint is insufficient. For this reason, the `multicycle` exception is better. For example, If `NUM_PIPES=3`, you can add a multicycle exception equal to `NUM_PIPES(3)`.

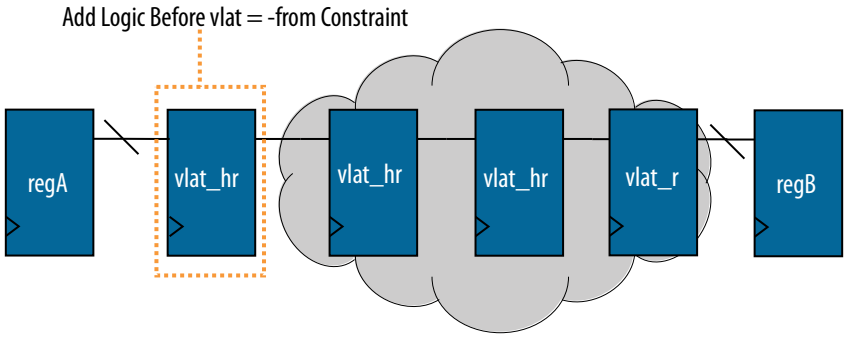
```
if {![is_post_route]} {
set_multicycle_path -setup –to my|top|design|hyperpipe_vlat_inst|vlat_r[*] 3
set_multicycle_path –hold –to my|top|design|hyperpipe_vlat_inst|vlat_r[*] 2}
```

Apply Variable Latency Constraints -from or -to

You can use the variable latency module to add Hyper-Registers forward or backward in the register chain. Use the following constraint methods when pushing registers through combinational logic:

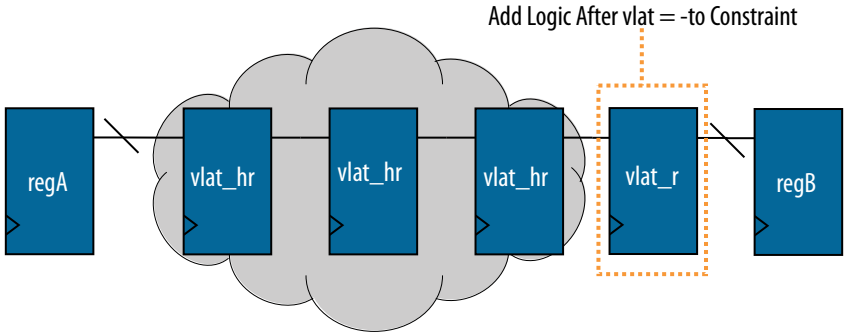
- **-from**—place `vlat_r` before combinational logic that you want to pipeline, then apply the multicycle or false path `–from` constraint.

Figure 40. Inserting vlat_hr Before Combinational Logic



- **-to**—place `vlat_r` after combinational logic that you want to pipeline, then apply the multicycle or false path `–to` constraint

Figure 41. Inserting vlat_r After Combinational Logic



2.3.2.2.3. Step 3: Verify Automatic Pipeline Insertion Option

The **Enable Auto-Pipelining** option (HYPER_RETIMER_ENABLE_ADD_PIPELINING) is required for automatic pipeline insertion, and is enabled by default in the Intel Quartus Prime software.

Follow these steps to verify or change the **Enable Auto-Pipelining** setting:

1. Click **Assignments ➤ Settings ➤ Compiler Settings ➤ Advanced Settings (Fitter)**.
2. To use automatic pipeline insertion, ensure that **Enable Auto-Pipelining** is **On**. You can turn this setting **Off** to prevent the addition of further pipeline stages in the instances of the `hyperpipe_vlat` module.
3. Click **OK**.

   Alternatively, you can enable or disable this option by specifying the following assignment the `.qsf` directly:

   ```
   set_global_assignment -name HYPER_RETIMER_ENABLE_ADD_PIPELINING <ON|OFF>
   ```

4. To compile the design, click **Processing ➤ Start Compilation**.

2.3.2.2.4. (Optional) Auto-Pipeline Insertion without a Variable Latency Module

You can optionally enable auto-pipeline insertion, without use of the variable latency module (`hyperpipe_vlat`) by following these steps for the target registers:

1. To specify the maximum number of stages to insert, click **Assignments ➤ Assignment Editor**, and then select **Maximum Additional Pipelining** for **Assignment Name**, enter the maximum number of pipelines for **Value**, and the hierarchical path to the register for **To**. Alternatively, you can add the following equivalent assignment to the `.qsf`.

   ```
   set_instance_assignment -name HYPER_RETIMER_ADD_PIPELINING \
        <maximum stages> -to <register path>
   ```

   Note: If you embed the assignment in RTL with the `altera_attribute` statement, rather than adding to the `.qsf`, you must specify the numeric value as a string in Verilog HDL and VHDL.

2. To prevent any optimization of the bus before auto-pipelining inserts additional stages, specify the `preserve` pragma, and set **Netlist Optimizations** to **Never Allow** for the target registers in the Assignment Editor or with the following `.qsf` assignment. Any optimization of the bus before autopipelining can impact the signal integrity of if autopipelining adds additional stages to some but not all bits of the bus.

   ```
   set_instance_assignment -name \
        ADV_NETLIST_OPT_ALLOWED NEVER_ALLOW -to <register path>
   ```

3. To ensure that related registers receive the same number of additional pipeline stages, create an assignment group to associate and assign all registers in the group. If you do not define an assignment group, the group names auto-generate with a prefix of `add_pipelining_group`, and each register that you specify for `HYPER_RETIMER_ADD_PIPELINING` becomes a group.

   The following line shows the syntax of the `.qsf` group assignment:

   ```
   set_instance_assignment -name \
       HYPER_RETIMER_ADD_PIPELINING_GROUP <group name string> \
       -to <register path>
   ```

2.3.3. Use Registers Instead of Multicycle Exceptions

Often designs contain modules with complex combinational logic (such as CRCs and other arithmetic functions) that require multiple clock cycles to process. You constrain these modules with multicycle exceptions that relax the timing requirements through the block. You can use these modules and constraints in designs targeting Intel Hyperflex architecture FPGAs. 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 Information

- Optimize Multicycle Paths on page 19

- Appendix A: Parameterizable Pipeline Modules on page 132

2.4. Hyper-Optimization (Optimize RTL)

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

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

2.4.1. General Optimization Techniques

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

2.4.1.1. Shannon's Decomposition

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

Figure 42. Shannon's Decomposition

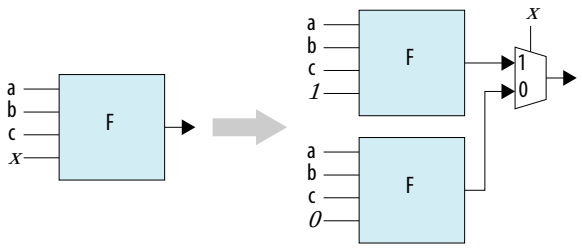


Figure 43. Shannon's Decomposition Logic Reduction

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

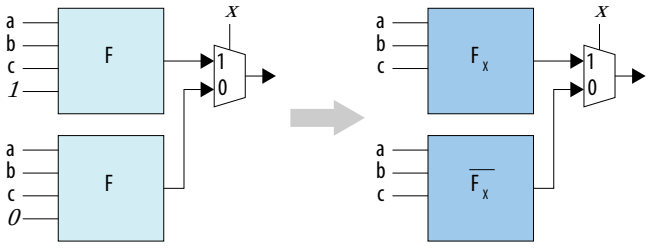
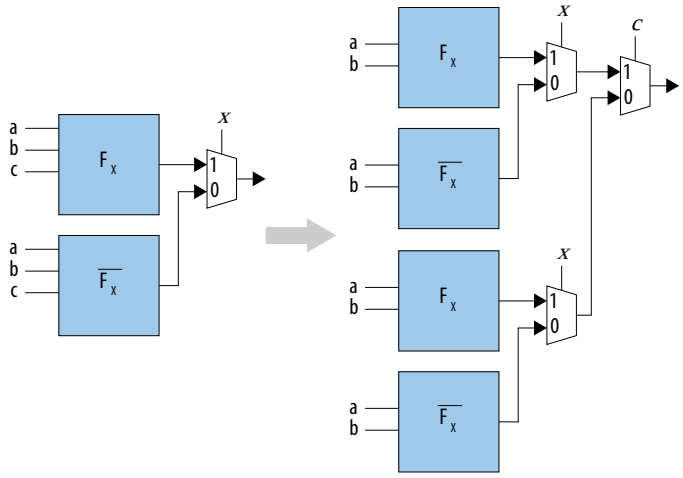


Figure 44. Repeated Shannon's Decomposition

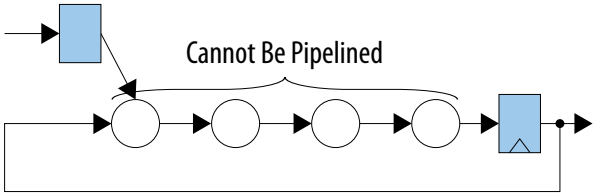
The following diagram shows how you can repeatedly use Shannon's decomposition to decompose functions with more than one critical input signal, thus increasing the area cost.



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 45. Loop Example before Shannon's Decomposition

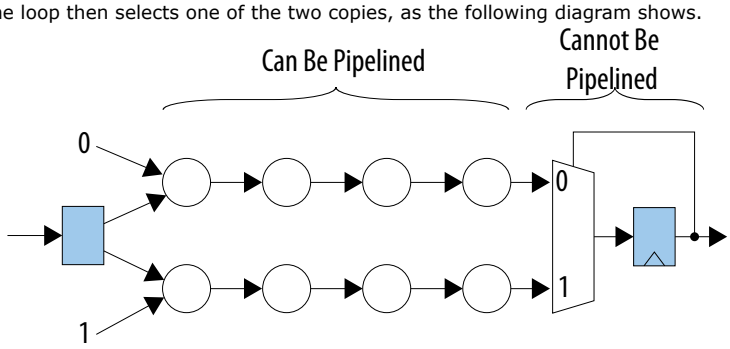
This diagram shows a loop that contains a single register, four levels of combinational logic, and an additional input. Adding registers in the loop changes the functionality, but you can move the combinational logic outside the loop by performing Shannon's decomposition.



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 46. Loop Example after Shannon's Decomposition

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



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

2.4.1.1.1. Shannon's Decomposition Example

The sample circuit adds or subtracts an input value from the `internal_total` value based on its relationship to a target value. The core of the circuit is the `target_loop` module, shown in Source Code before Shannon's Decomposition.

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.

Fast Forward Compile Report before Shannon's Decomposition shows the Fast Forward Compile report for the `target_loop` module instantiated in a register ring.

Figure 47. Fast Forward Compile Report before Shannon's Decomposition

Fast Forward Details for Clock Domain clk

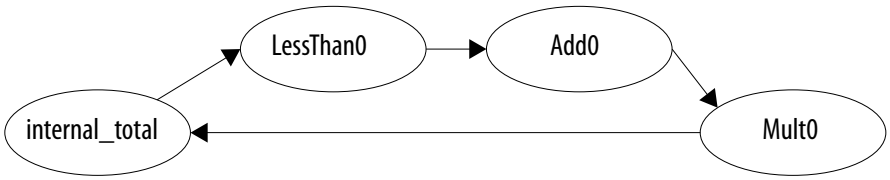
Fast Forward Summary for Clock Domain clk | Fast Forward Limit Critical Chain Schematic

	Step	Fast Forward Optimizations Analyzed	Estimated Fmax	Slack	Relationship
1	Base Performance	None	204 MHz	-3.897	1.000
2	Fast Forward Step #1 (Hyper-Pipelining)	Added up to 1 pipeline stage in 8 Paths	209 MHz	-3.786	1.000
3	Fast Forward Step #2 (Hyper-Optimization)	Added up to 1 pipeline stage in 8 Paths Fully registered 2 DSP blocks	302 MHz	-2.306	1.000
4	Fast Forward Limit	Performance Limited by: Retiming Dependency Loop	--	--	--

Hyper-Retiming reports about 302 MHz by adding a pipeline stage in the Fast Forward Compile. The last Fast Forward Limit row 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.

Elements of a Critical Chain Sub-Loop 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 48. 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 Source Code after Shannon's Decomposition.

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
```

Fast Forward Summary Report after Shannon's Decomposition shows the performance almost doubles after recompiling the design with the code change.

Figure 49. Fast Forward Summary Report after Shannon's Decomposition

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	419 MHz	-1.386	1.000
2	Fast Forward Step #1 (Hyper-Pipelining)	Added up to 1 pipeline stage in 10 Paths	431 MHz	-1.318	1.000
3	Fast Forward Step #2 (Hyper-Pipelining)	Added up to 1 pipeline stage in 32 Paths	438 MHz	-1.282	1.000
4	Fast Forward Limit	Performance Limited by: Retiming Dependency Loop	--	--	--

2.4.1.1.2. Identifying Circuits for Shannon's Decomposition

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

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

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

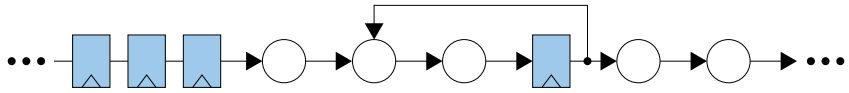
2.4.1.2. Time Domain Multiplexing

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

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

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

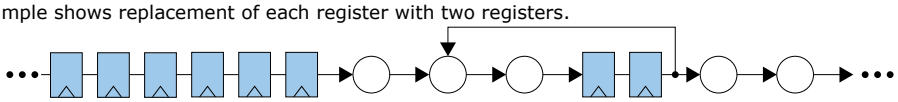
Figure 50. C-slow Retiming Starting Point



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.

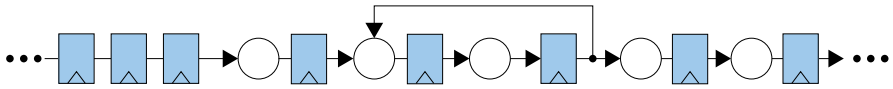
Figure 51. C-slow Retiming Intermediate Point

This example shows replacement of each register with two registers.



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

Figure 52. C-Slow Retiming Ending Point



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

2.4.1.3. Loop Unrolling

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

2.4.1.4. Loop Pipelining

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

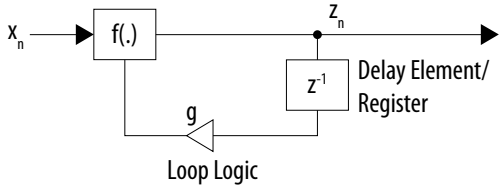
However, you can modify the loop structure to allow the Compiler to insert pipeline stages, without changing the functionality of the design, as the following topics demonstrate. Properly pipelining a loop involves the following steps:

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

2.4.1.4.1. Loop Pipelining Theory

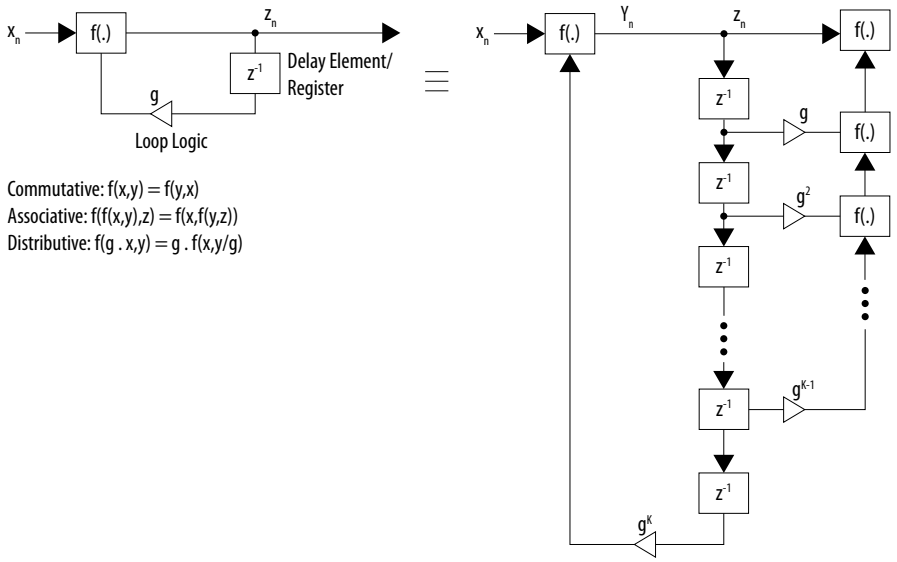
The following figure illustrates the definition of a logical loop. The result (Z_n) is a function of input X_n and a delayed version of that input.

Figure 53. Simple Loop Example



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

Figure 54. Equivalent Loop After Modification

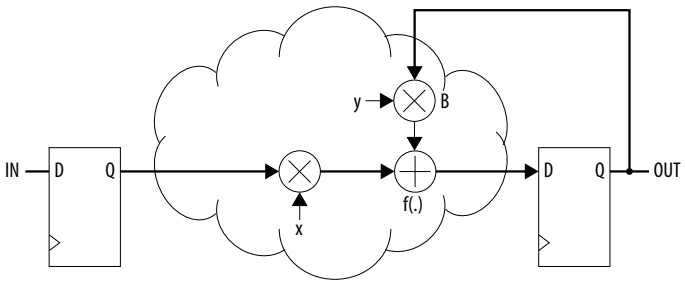


2.4.1.4.2. Loop Pipelining Demonstration

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

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

Figure 55. Original Loop Structure



Example 7. Original Loop Structure Example Verilog HDL Code

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

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

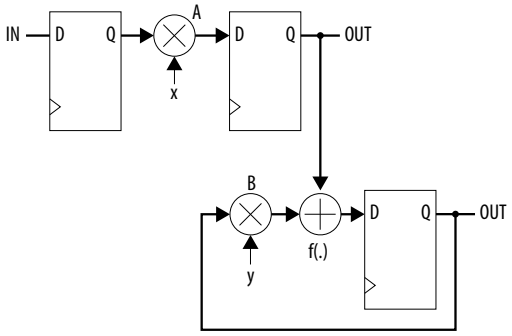
```
   end

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

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

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

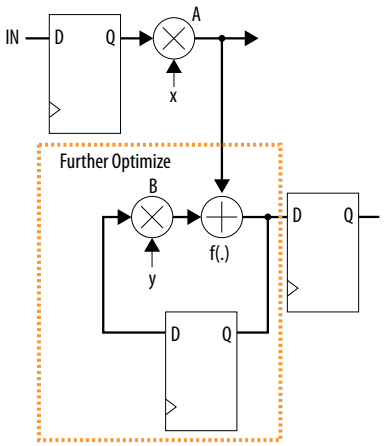
After rewriting the logic, the Compiler can now freely retime the logic that you move to the forward path.

Figure 56. Separation of Forward Logic from the Loop



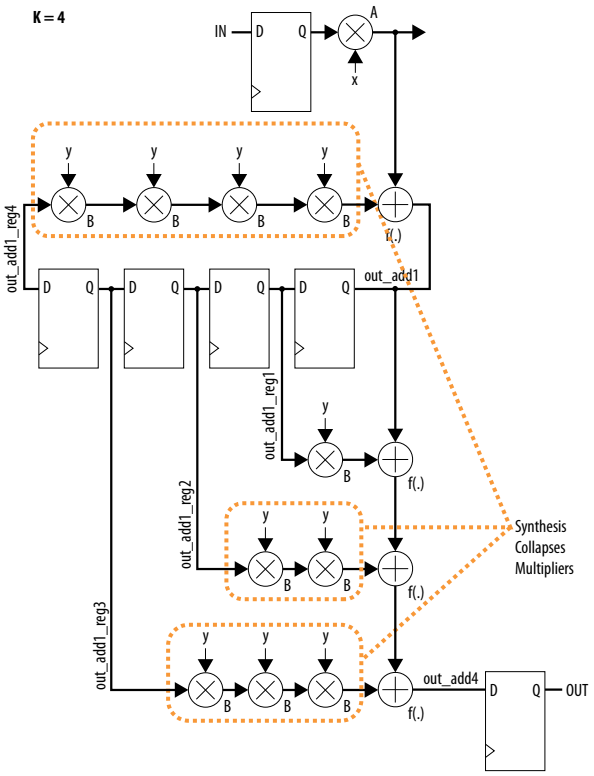
In the next optimization stage, retime the loop register to ensure that the design functions the same as the original loop circuitry.

Figure 57. Retime Loop Register



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

Figure 58. Results of Cascade Loop Logic, Hyper-Retimer, and Synthesis Optimizations (Four Level Optimization)



Example 8. Four Level Optimization Example Verilog HDL Code

```
module cll_hypr_rtm_synopt ( rstn, clk, x, y, in, out);
   input rstn, clk, x, y, in;

   output out;
   reg    out;

   reg in_reg;

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

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

always @ ( posedge clk )
   if ( !rstn ) begin
      in_reg <= 0;
   end else begin
      in_reg <= in;
   end

always @ ( posedge clk )
```

```
   if ( !rstn ) begin
      out_add1_reg1 <= 0;
      out_add1_reg2 <= 0;
      out_add1_reg3 <= 0;
      out_add1_reg4 <= 0;
   end else begin
      out_add1_reg1 <= out_add1;
      out_add1_reg2 <= out_add1_reg1;
      out_add1_reg3 <= out_add1_reg2;
      out_add1_reg4 <= out_add1_reg3;
   end

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

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

2.4.1.4.3. Loop Pipelining and Synthesis Optimization

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

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

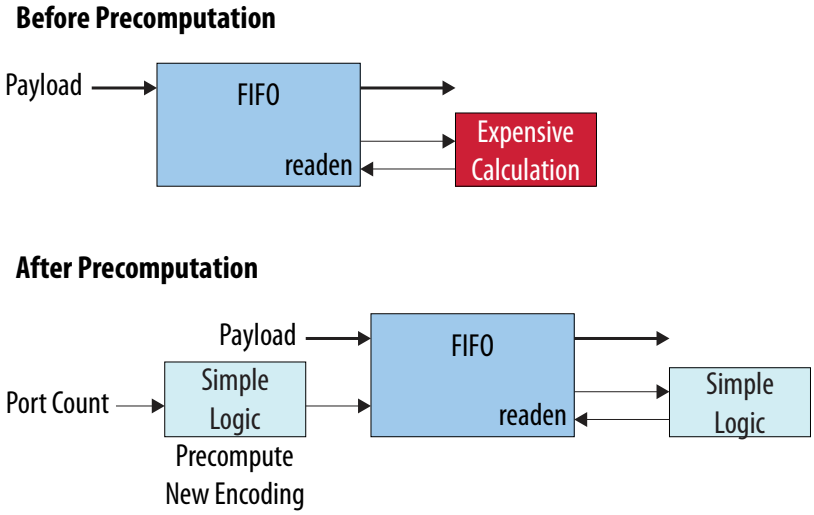
2.4.1.5. Precomputation

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

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

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

Figure 59. 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 the computation with simple boolean operations. The modified code is:

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

2.4.2. Optimizing Specific Design Structures

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

2.4.2.1. High-Speed Clock Domains

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

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

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

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

2.4.2.1.1. Visualizing Clock Networks

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

To visualize design clock networks in the Chip Planner:

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

Figure 60. Clock Network in Chip Planner

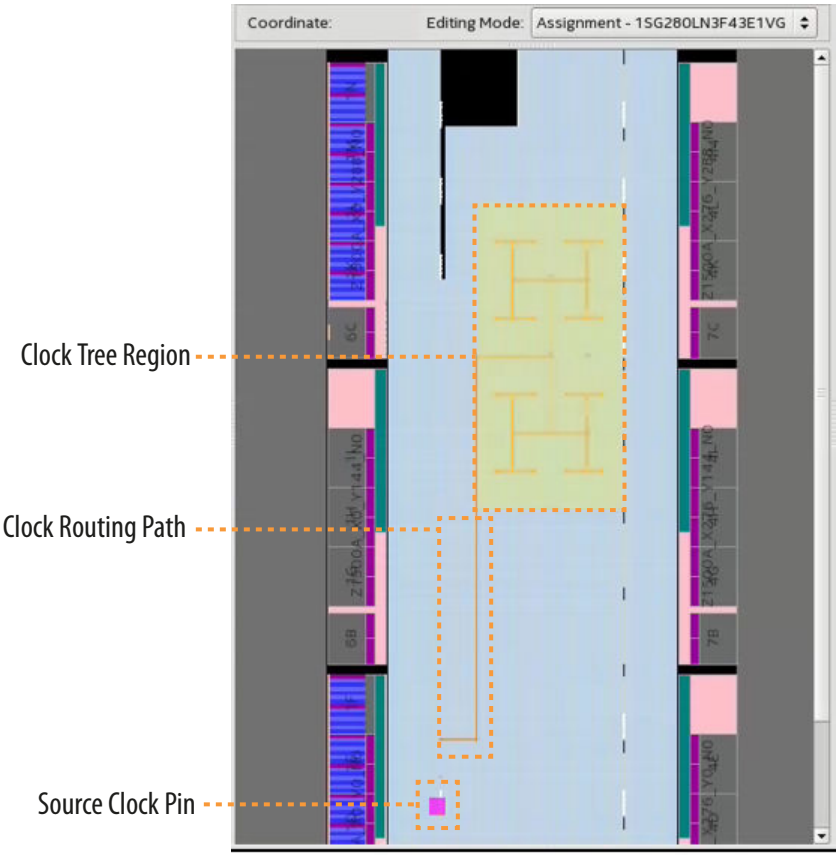


Figure 61. Clock Sector Boundary Layer in Chip Planner

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 62. Clock Network Details in Fitter Report

Global & Other Fast Signals Details

<<Filter>>

	Property	Value
1	Name	dut|dut|hip|altera_pcie_s10_hip_ast_..._chan
1	-- SDC Name	dut|dut|hip|altera_pcie_s10_hip_ast_pipen1b
2	-- Source Type	HSSI PLDADAPT TX
3	-- Source Location	HSSIPLDADAPTTX_1C0
4	-- Fan-Out	36674
5	-- Promotion Reason	Standard promotion candidate (required due t
6	-- Clock Region	Sectors (0, 0) to (1, 1)
7	-- Clock Region Size (in Sectors)	2 x 2 (4 total)
8	-- Spine Index used in each Sector	1
9	-- Path Length from...ource to Clock Tree	1 clock sector wire(s)
10	-- Clock Tree Depth	1.5 clock sector wire(s)
2		
3	Name	auto_fab_0|alt_sld_fab_0|alt_sld_fab_0|s10
4		
5	Name	dut|dut|hip|altera_pcie_s10_hip_ast_..._chan
6		

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

2.4.2.1.3. Viewing Clocks in the Timing Analyzer

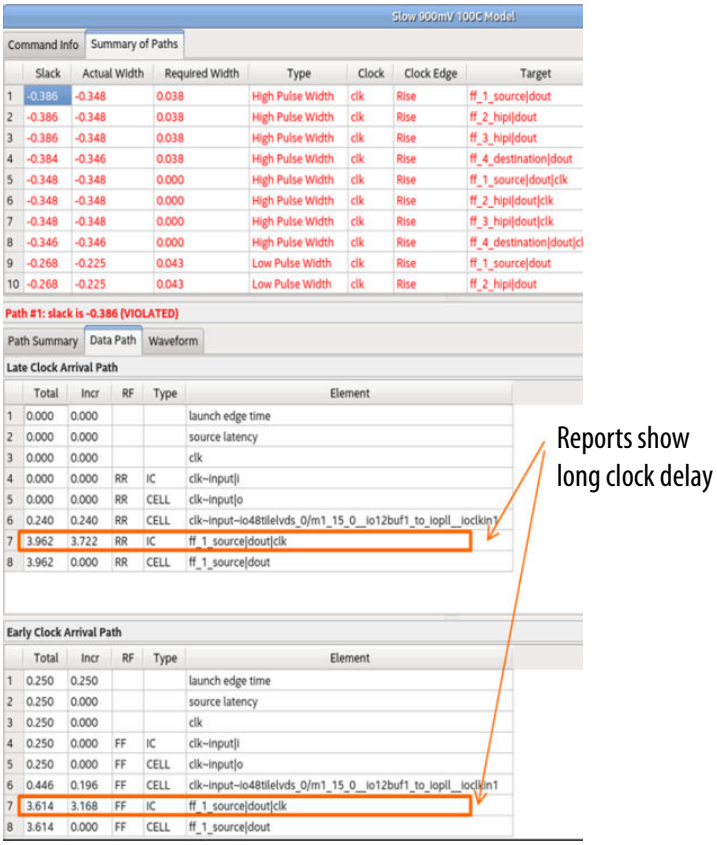
The Timing Analyzer reports high speed clocks that are limited by long clock paths. Open the Fmax Summary report to view any clock f_{MAX} that is restricted by high minimum pulse width violations (t_{CH}), or low minimum pulse width violation (t_{CL}).

To view clock network data in the Timing Analyzer:

1. Open a project.
2. On the Compilation Dashboard, click **Timing Analysis**. After timing analysis is complete, the **Timing Analyzer** folder appears in the Compilation Report.
3. Under the **Slow 900mV 100C Model** folder, click the **Fmax Summary** report.

4. To view path information details for minimum pulse width violations, in the Compilation Report, right-click the **Minimum Pulse Width Summary** report and click **Generate Report in Timing Analyzer**. The Timing Analyzer loads the timing netlist.
5. Click **Reports ➤ Custom Reports ➤ Report Minimum Pulse Width**.
6. In the **Report Minimum Pulse Width** dialog box, specify options to customize the report output and then click **OK**.
7. Review the data path details for report of long clock routes in the **Slow 900mV 100C Model** report.

Figure 63. Minimum Pulse Width Details Show Long Clock Route



2.4.2.2. Restructuring Loops

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

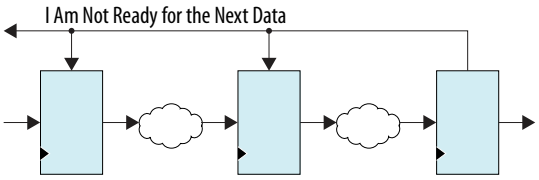
The Compiler never retimes registers into a loop, because adding a pipeline stage to a loop changes functionality. However, change your RTL manually to restructure loops to improve performance. Perform loop optimization after analyzing performance bottlenecks with Fast Forward compile. Also apply these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

2.4.2.3. Control Signal Backpressure

This section describes RTL design techniques to control signal backpressure. The Intel Hyperflex 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.

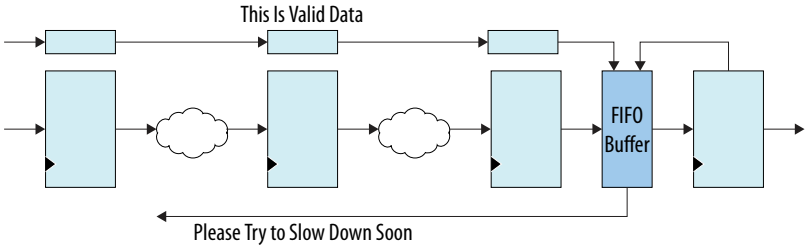
Single clock cycle control signals create loops that can prevent or reduce the effectiveness of pipelining and register retiming. This example depicts a ready signal that notifies the upstream register of readiness to consume data. The ready signals must freeze multiple data sources at the same time.

Figure 64. 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 65. 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.

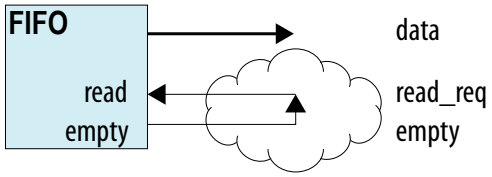
2.4.2.4. Flow Control with FIFO Status Signals

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 or empty signals. Instead of working with immediate control signals, use a delayed signal. You can build a buffer within the FIFO block. The control signals indicate to the upstream data path that the path is almost full, leaving a few clock cycles for the upstream data to receive their gating signal. This approach alleviates timing closure difficulties on the control signals.

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

Figure 66. FIFO Flow Control Loop

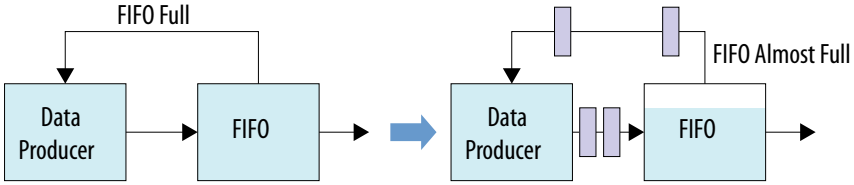
The loop forms while reading control signals from the FIFO.



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

Figure 67. Improved FIFO Flow Control Loop with Almost Full instead of Full FIFO

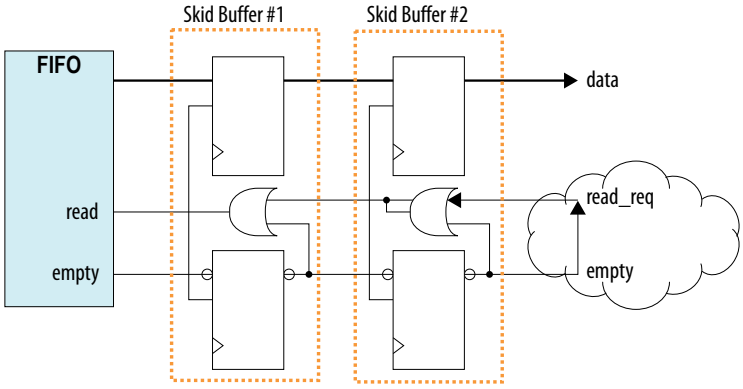
The following example shows two extra registers in the full control flow signal. After the FIFO generates the almost full flag, it takes two cycles for the signal to reach the data producer block. Two additional cycles are then required before the data sent stops arriving at the FIFO. This condition results in four cycles of latency. Anytime the almost full flag is asserted, the data producer can send 4 more words before the FIFO is actually full. 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.



2.4.2.5. Flow Control with Skid Buffers

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 when possible.

Figure 68. FIFO Flow Control Loop with Two Skid Buffers in a Read Control Loop



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

Example 9. Skid Buffer Example (Single Clock)

```
/ 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
```

Example 10. Skid Buffer Example (Dual Clock)

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

module  skid_dualclock_fifo
#(

    parameter DATA_WIDTH      = 8,
    parameter FIFO_DEPTH      = 16,
    parameter SHOWAHEAD       = "ON",
    parameter RAM_TYPE        = "AUTO", // "AUTO" or "MLAB" or "M20K".
    // Derived
    parameter ADDR_WIDTH      = $clog2(FIFO_DEPTH) + 1
)
(
    input  wire                   rd_clk,
    input wire                    wr_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 rd_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 rd_clk)
    begin
      if(e_pop_1)
        out_data_1q  <= scfifo_data_out;
      if(e_pop_2)
        out_data_2q   <= out_data_1q;
    end
dcfifo  dcfifo_component
    (
        .data      (in_data),
        .rdclk     (rd_clk),
        .rdreq     (e_pop_qual),
        .wrclk     (wr_clk),
        .wrreq     (push),
        .q         (scfifo_data_out),
        .rdempty   (scfifo_empty),
        .rdusedw   (scfifo_usedw),
        .wrfull    (scfifo_full),
        .wrusedw   (),
        .aclr      (1'b0),
        .eccstatus (),
        .rdfull    (),
        .wrempty   ()
    );
    defparam
        dcfifo_component.add_usedw_msb_bit       = "ON",
```

```
        dcfifo_component.enable_ecc              = "FALSE",
        dcfifo_component.intended_device_family  = "Stratix 10",
        dcfifo_component.lpm_hint                =  (RAM_TYPE == "MLAB") \
             ? "RAM_BLOCK_TYPE=MLAB" : ((RAM_TYPE == "M20K") \
             ? "RAM_BLOCK_TYPE=M20K" : ""),
        dcfifo_component.lpm_numwords            = FIFO_DEPTH,
        dcfifo_component.lpm_showahead           = SHOWAHEAD,
        dcfifo_component.lpm_type                = "dcfifo",
        dcfifo_component.lpm_width               = DATA_WIDTH,
        dcfifo_component.lpm_widthu              = ADDR_WIDTH+1,
        dcfifo_component.overflow_checking       = "ON",
        dcfifo_component.read_aclr_synch         = "ON",
        dcfifo_component.rdsync_delaypipe        = 5,
        dcfifo_component.underflow_checking      = "ON",
        dcfifo_component.write_aclr_synch        = "ON",
        dcfifo_component.use_eab                 = "ON",
        dcfifo_component.wrsync_delaypipe        = 5;


endmodule
```

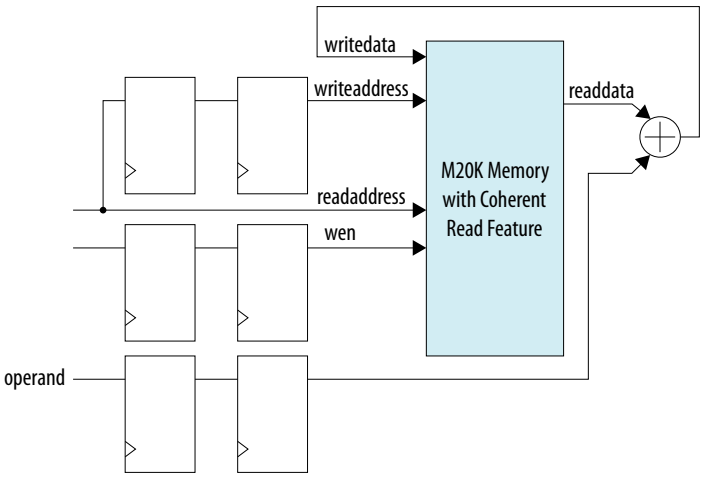
2.4.2.6. Read-Modify-Write Memory

Intel Hyperflex architecture FPGA 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.

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

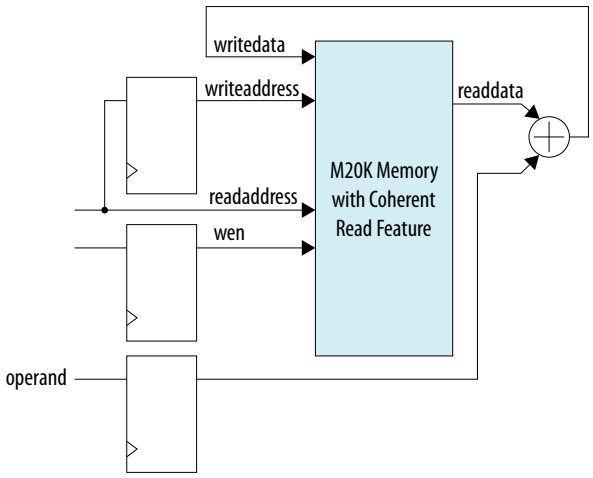
To use the coherent read feature, connect memory according to whether you register the output data port. If you register the output data port, add two register stages to the write enable and write address lines when you instantiate the memory.

Figure 69. 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 70. Unregistered Output Data Requires One Register Stage

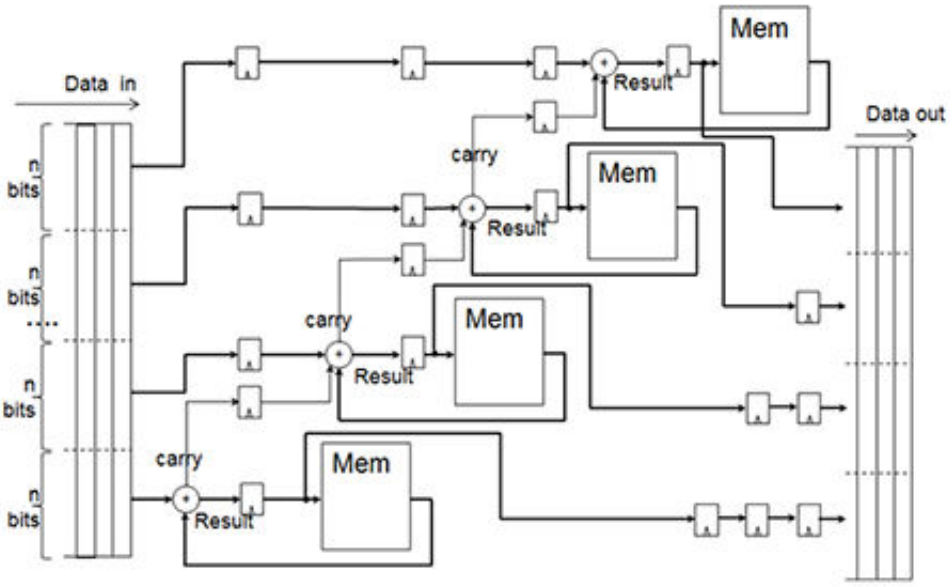


Use of coherent read has the following restrictions:

- 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.

Pipelining Read-Modify-Write Memory shows a pipelining method for a read-modify-write memory that improves performance, without maintaining a cache for tracking recent activity.

Figure 71. Pipelining Read-Modify-Write Memory



If you require M20K memory features that are incompatible with coherent read, or if you do not want to use coherent read, use the following alternative approaches to improve the f_{MAX} performance of memory:

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

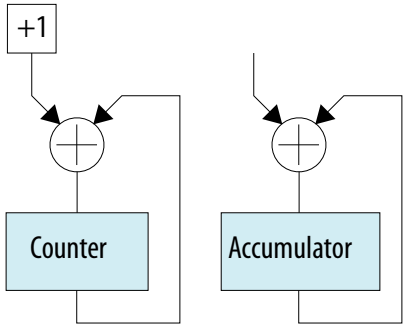
2.4.2.7. Counters and Accumulators

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

You can also use loop unrolling to improve counter performance.

Figure 72. 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.



2.4.2.8. State Machines

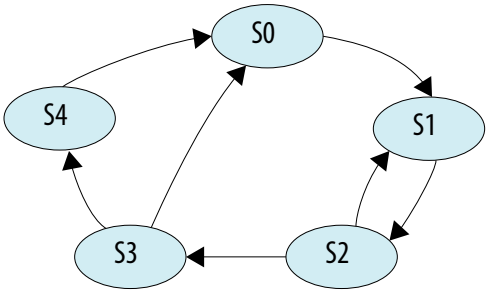
Loops related to state machines can be difficult to optimize. Carefully examine the state machine logic to determine whether you can precompute any signals that the next state logic uses.

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

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

Figure 73. State Machine Loop

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



Related Information

- Appendix A: Parameterizable Pipeline Modules on page 132
- Precomputation on page 51

2.4.2.9. Memory

The section covers various topics about optimization for hard memory blocks in Intel Hyperflex architecture FPGAs.

Note: Refer to the following documents for details about frequencies supported by various RAM modes and device speed grades.

Related Information

- Intel Stratix 10 Device Datasheet
- Intel Agilex 7 Device Datasheet

2.4.2.9.1. Intel Hyperflex Architecture True Dual-Port Memory

The Intel Hyperflex architecture supports true dual-port memory structures. True dual-port memories allow two write and two read operations at once.

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

The Intel Hyperflex architecture supports true dual-port memories in independent clock mode. When you use memory in this mode, the maximum f_{MAX} associated with this memory is 600 MHz

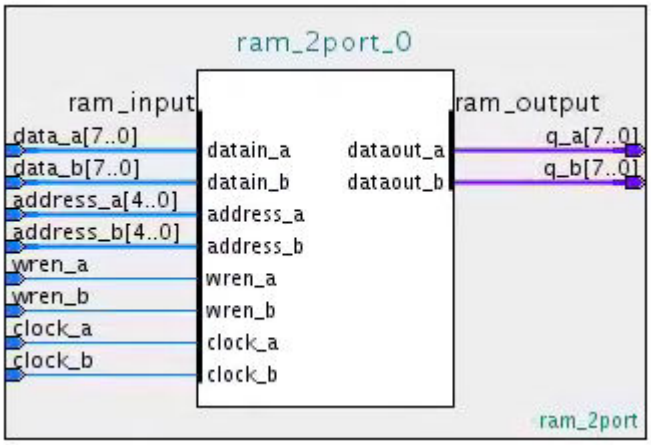
2.4.2.9.2. Use Simple Dual-Port Memories

When migrating a design to the Intel Hyperflex architecture, consider whether your original design contains a dual-port memory that uses different clocks on each port, and the maximum frequency you plan to operate the memory. 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 74. Intel Arria 10 True Dual-Port Memory Implementation

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



Example 11. 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
```

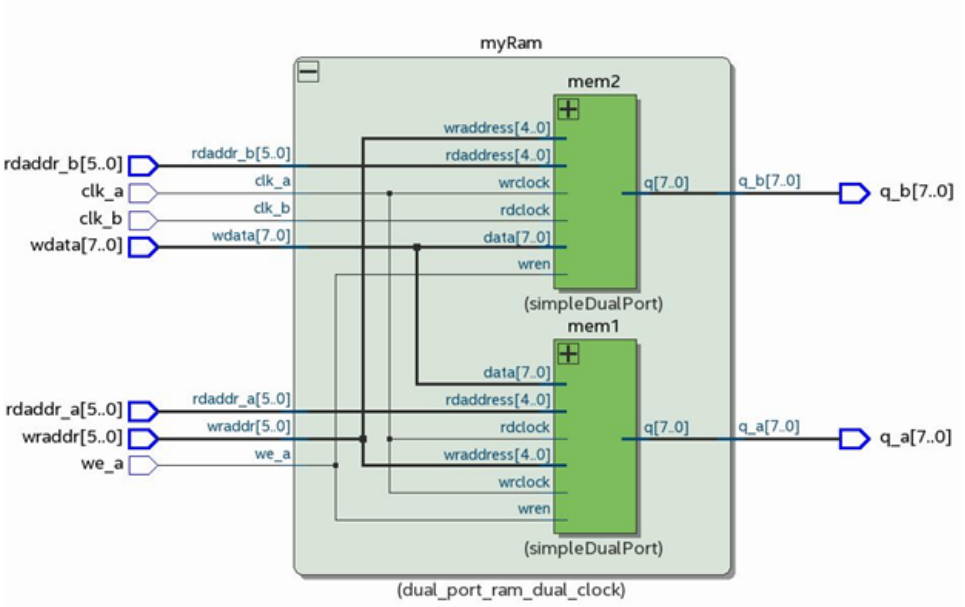
```
            q_b <= ram[addr_b];
        end
    end

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.

2.4.2.9.3. Intel Hyperflex Architecture Simple Dual-Port Memory Example

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

Figure 75. 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 Intel Quartus Prime IP parameter editor.

Example 12. 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 13. True Dual-Port RAM Behavior Emulation

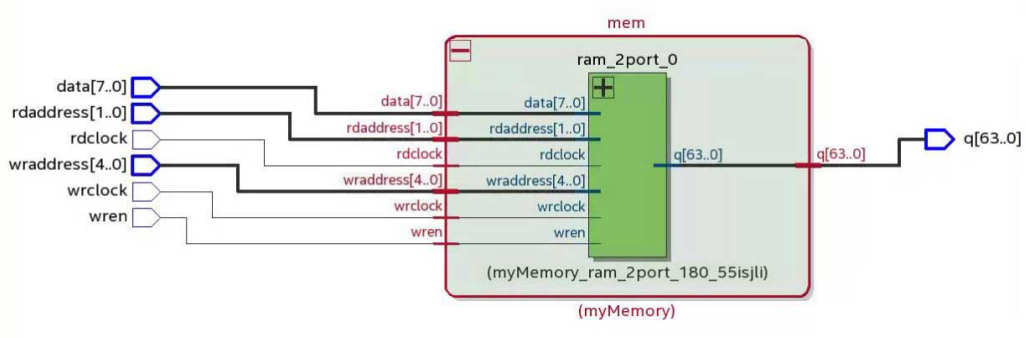
```
module test (wrdata, wraddr, rdaddr_a, rdaddr_b,
    clk_a, clk_b, we_a, q_a, q_b);

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

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

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

endmodule
```

2.4.2.9.4. Memory Mixed Port Width Ratio Limits

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

When migrating a design that uses invalid port width ratios for Intel Hyperflex architecture FPGAs, modify the RTL to create the desired ratio.

Figure 76. Dual-Port Memory with Invalid 1/8 Mixed Port Ratio

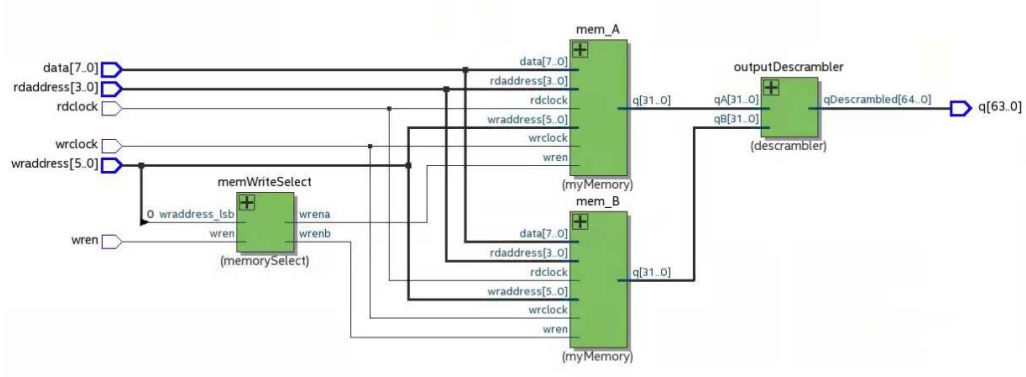


To create a functionally equivalent design, create and combine smaller memories with valid mixed port width ratios. For example, the following steps implement a mixed port width ratio:

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 77. 1/8 Width Ratio Example

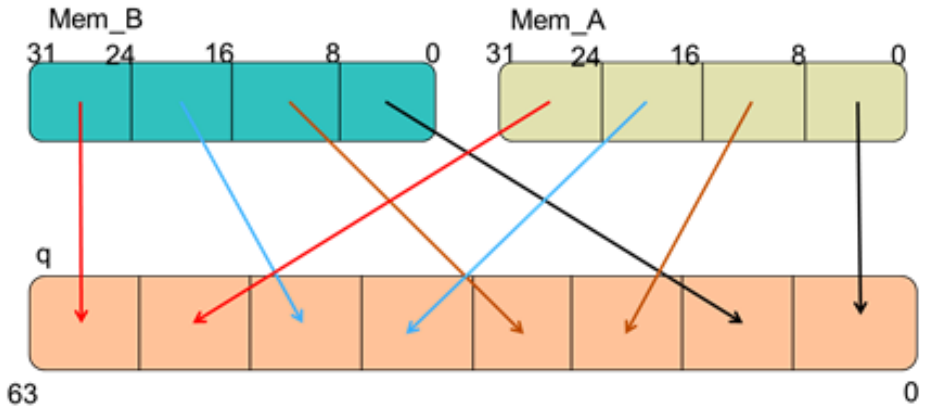
This example shows the interleaving of two memories and the write logic. The chosen write logic uses the least significant bit of the write address to decide which memory to write. Even addresses write in memory `mem_A`, odd addresses write in memory `mem_B`.



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 78. 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 14. 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 15. 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
```

2.4.2.9.5. Unregistered RAM Outputs

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

You can unknowingly use unregistered memory outputs, followed by combinational logic, if you implement a RAM using the read-during-write new data mode. The Compiler implements this mode with soft logic outside of the memory block that compares the read and write addresses. This mode muxes the write data straight to the output. If you want to achieve the highest performance, do not use the read-during-write new data mode.

2.4.2.10. 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.

Note: Refer to the following documents for details about frequencies supported by various DSP modes and device speed grades.

Related Information

- Intel Stratix 10 Device Datasheet
- Intel Agilex 7 Device Datasheet

2.4.2.11. 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.

2.4.2.12. 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.

2.4.2.13. Resets

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

2.4.2.14. Hardware Re-use

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

2.4.2.15. 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.

2.4.2.16. 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.

2.4.2.17. Ternary Adders

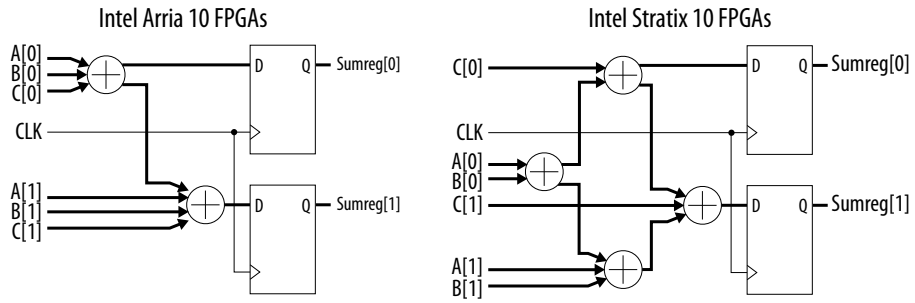
Implementing ternary adders can increase resource usage in Intel Hyperflex architecture FPGAs. However, unless your design heavily relies on ternary adder structure, additional resource usage may not be noticeable at the top design level. However, a review of the design level at which you add a ternary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

Ternary Adder RTL Code

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

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

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

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

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

Figure 79. RTL View of Intel Arria 10 versus Intel Hyperflex Architecture FPGAs to add 2 LSBs from a three 8-bit input adder



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

Figure 80. Intel Arria 10: ALMs used to add 2 LSBs from a three 8-bit input adder

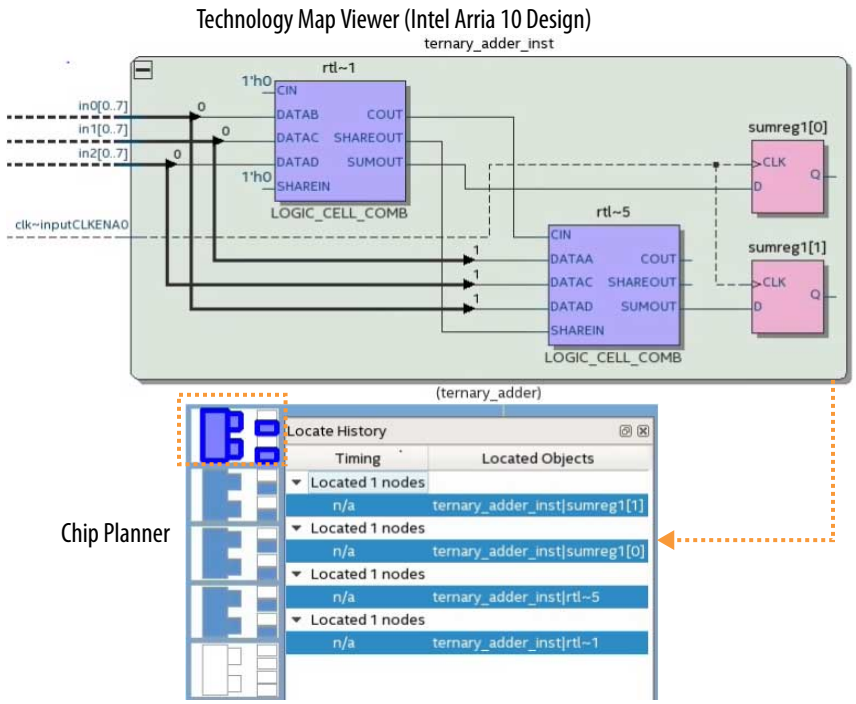
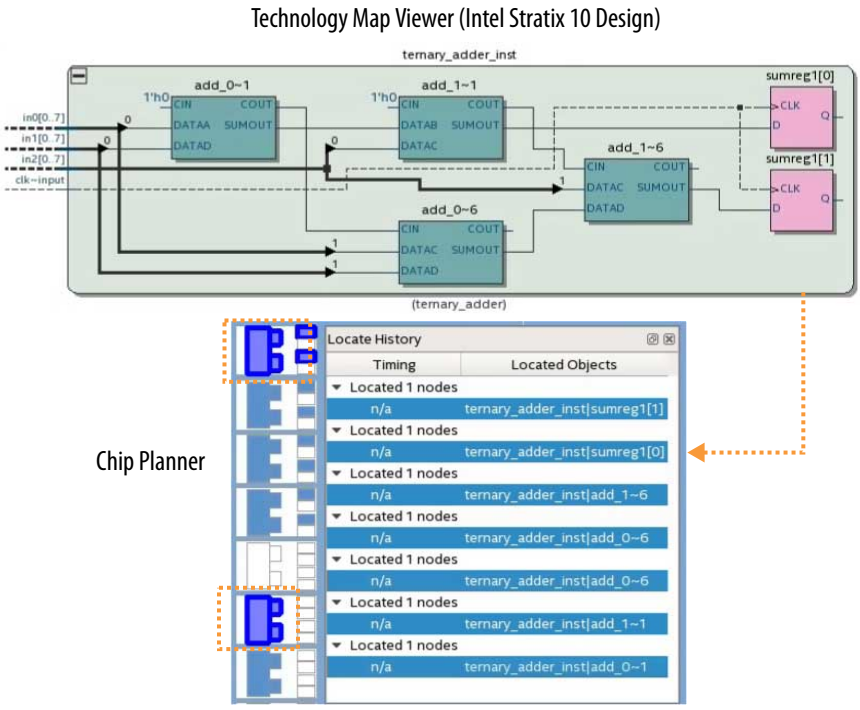


Figure 81. Intel Hyperflex Architecture FPGAs: ALMs used to add 2 LSBs from a three 8-bit input adder



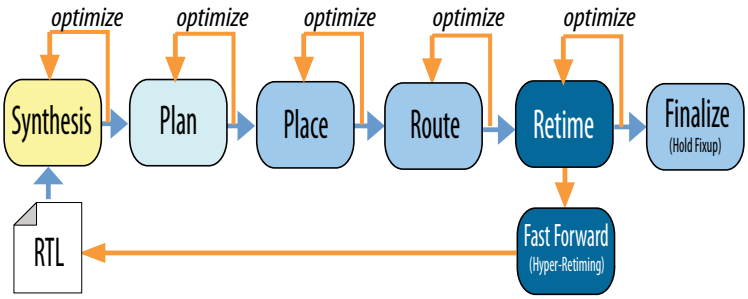
3. Compiling Intel Hyperflex Architecture Designs

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

Hyper-Aware Design Flow

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

Figure 82. Hyper-Aware Design Flow



Register Retiming

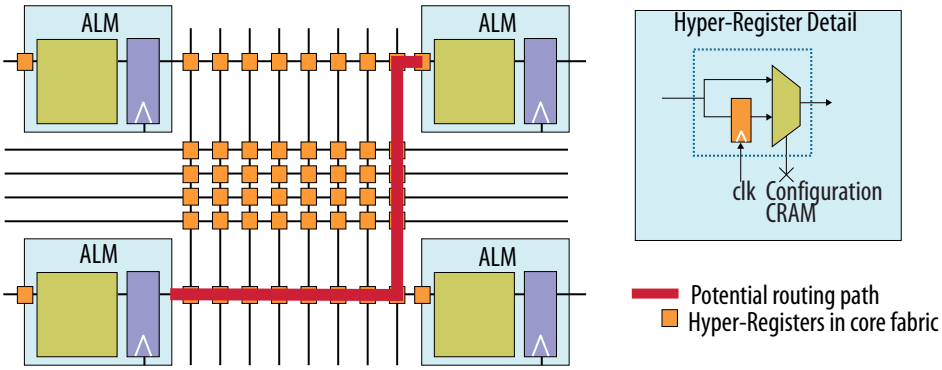
A key innovation of the Intel Hyperflex architecture is the addition of multiple Hyper-Registers in routing segments and block inputs. Maximizing the use of Hyper-Registers improves design performance. The prevalence of Hyper-Registers improves balance of time delays between registers and mitigates critical 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 83. Hyper-Register Architecture



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

Table 8. Optimization Steps

Optimization Step	Technique	Description
Step 1	Register Retiming	The **Retime** stage performs register retiming and moves existing registers into Hyper-Registers to increase performance by removing retiming restrictions and eliminating critical paths.
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.

Verifying Design RTL

The Intel Quartus Prime software includes the Design Assistant design rule checking tool to verify the suitability to your design RTL for the Intel Hyperflex architecture. These rules include Hyper-Retimer Readiness Rules (HRR) that specifically target Intel Hyperflex FPGA architecture designs, as describes.

3.1. Compiling Submodules Independently

You can independently compile submodules of a larger design. Compiling a larger submodule independently allows you to obtain submodule performance information, and to optimize the submodule for better performance. Depending on your goals in compiling the submodule, you may choose to handle the I/Os differently, as the following techniques describe:

- If you require additional Hyper-Optimization for the submodule during compilation, instantiate the submodule within a wrapper, adding two or more register stages at each input and output. This technique allows retiming of those additional registers into the circuit. If the Compiler retimes the additional registers into the submodule, then you can modify the full design to provide extra registers to the submodule. This method is useful when testing Hyper-Optimization on your submodule to determine the impact of retiming when the submodule is part of the larger design.
- If you do not require further submodule Hyper-Optimization, but want to isolate a module to save compile time and mimic the module timing budget in the full design, define virtual pin constraints to enable retiming of the registers connecting directly to the submodule. Specify appropriate `set_input_delay` or `set_ouput_delay` SDC constraints to account for the clock skew of the internal clock tree. Account for the clock skew of the internal clock tree by specifying either a `reference_pin` or a static maximum constraint, according to the following guidelines and examples:

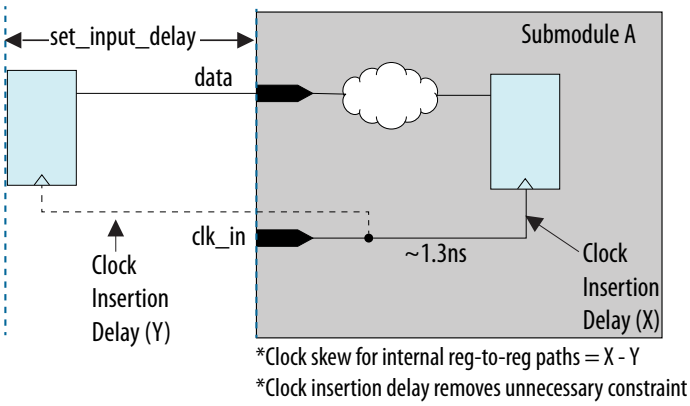
Table 9. Submodule Compilation Constraint Guidelines

SDC Argument	Guideline	Example

`clock`	Specify the internal clock domain name (no virtual clock). This argument causes the Compiler to treat external and internal clocks the same, allowing retiming of I/O registers no longer on the boundary.	`reference_pin` argument:
		`set_input_delay -clock [get_clocks <name>] \`
		`  0 [get_ports <name>] \`
		`  -reference_pin [get_pins [<names>]`
		static maximum constraint:
		`set insert_delay <value>`
		`set_input_delay -clock [get_clocks <name>] \`
		`  $insert_delay [get_ports <name>]`

delay	Specify one of the following to account for the clock skew of the internal clock tree: • Set `max` (setup) argument to `0` and set the `reference_pin` argument to the clock pin of an I/O register. The Compiler automatically subtracts the clock tree insertion delay to the I/O register. • Set the max argument ≥ the clock tree insertion delay.	

Figure 84. Submodule Compilation Input Constraint Example




```
set_input_delay -clock [get_clocks clk_in] \
     0 [get_ports data] \
     -reference_pin [get_pins [int_reg|clk]
```

Or:

```
set insert_delay 1.3
set_input_delay -clock [get_clocks clk_in] \
     $insert_delay [get_ports data]
```

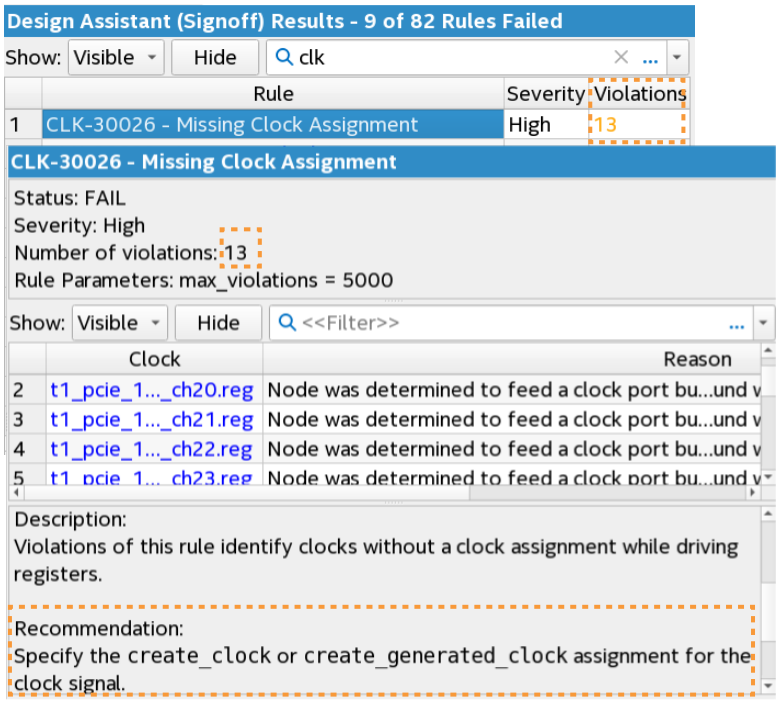
3.2. Design Assistant Design Rule Checking

The Intel Quartus Prime Design Assistant increases productivity by reducing the total number of design iterations for design closure, and by minimizing the time in each iteration with targeted rule checks and guidance at each stage of compilation.

The Design Assistant detects and helps you to resolve design rule violations by providing recommendations for correction and pathways to the violation source. Avoiding design rule violations improves the reliability, timing performance, and logic utilization of your design.

When enabled, Design Assistant automatically reports any violations against a standard set of Intel FPGA-recommended design guidelines [(1)]. You can run Design Assistant automatically during compilation, and report violations detected throughout the compilation process.

Figure 85. Design Assistant Recommends Corrections for Design Rule Violations



Alternatively, you can run Design Assistant in analysis mode, which allows you to launch Design Assistant checks from other Intel Quartus Prime tools, such as Chip Planner. For some rules, Design Assistant supports cross-probing to the Timing Analyzer and Intel Quartus Prime design visualization tools for root cause analysis and correction.

You can specify which rules Design Assistant checks, thus eliminating the rule checks that are unimportant for your design.

(1) A set of default rules ensures design health without significant runtime increase.

Related Information

AN 919: Improving Quality of Results with Design Assistant

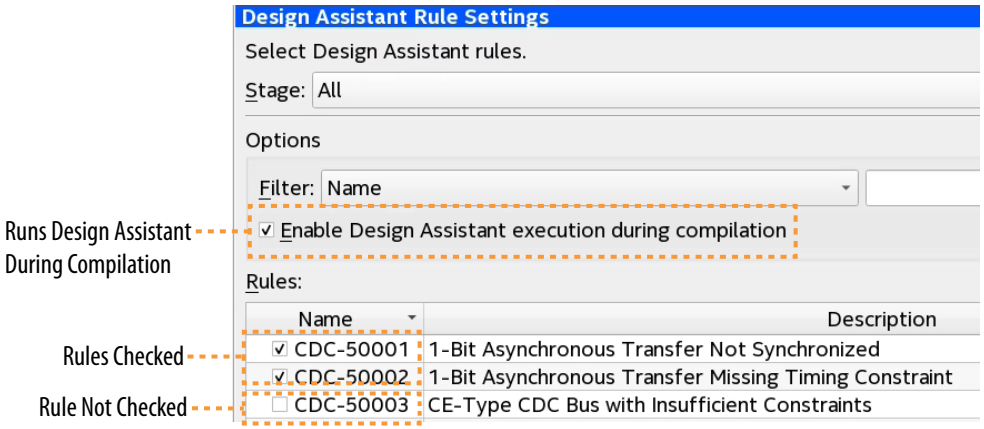
3.2.1. Running Design Assistant During Compilation

When enabled, Design Assistant runs automatically during compilation and reports design rule violations in the Compilation Report.

When you enable or specify parameters for a rule check in compilation mode, those specifications apply by default to running Design Assistant in compilation mode. If you change the rule settings for analysis mode, those settings are independent from the rule settings in compilation mode.

1. To run Design Assistant checking during compilation flows, ensure that **Enable Design Assistant execution during compilation** is on.
2. To enable or disable specific design rule checks, turn on or off the checkbox for that rule in the **Name** column. If the rule is unchecked, Design Assistant does not report violations for the rule.
3. In the **Parameters** field, consider changing default values for rules you enable.

Figure 86. Design Assistant Rule Settings

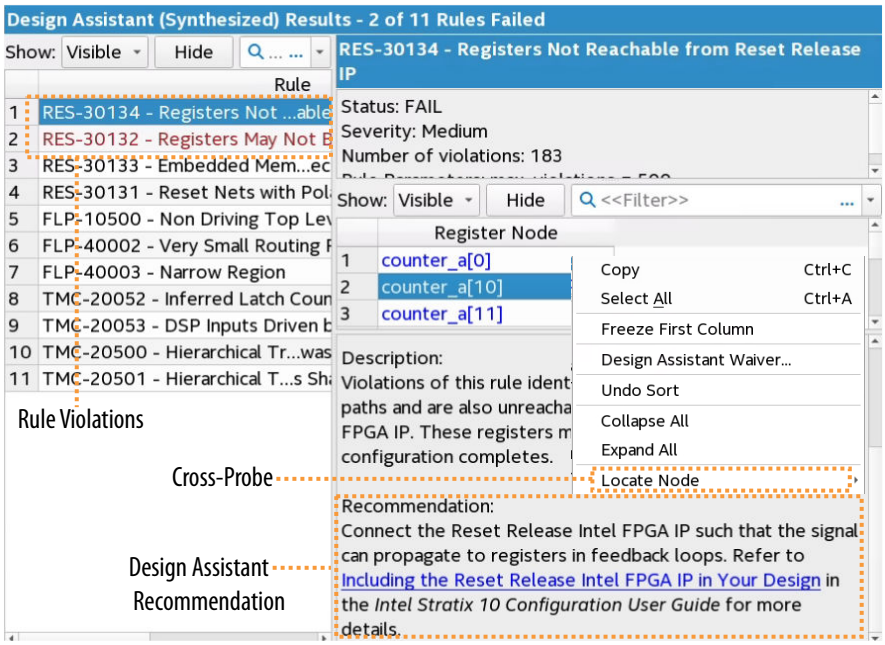


4. To run Design Assistant during compilation, run one or more stages of the Compiler from the Processing menu or Compilation Dashboard.

Figure 87. Example Design Assistant Results in Compilation Reports

5. To view the results for each rule, click the rule in the **Rules** list. A description of the rule and design recommendations for correction appear.

Figure 88. Design Assistant Rule Violation Recommendation



3.2.2. Running Design Assistant in Analysis Mode

You can launch Design Assistant in analysis mode directly from the Timing Analyzer or Chip Planner to rapidly run the specific rule checks that relate to those tools. For example, when you launch Design Assistant from the Chip Planner, Design Assistant is preset to check only a subset of the FLP (floorplanning) Design Assistant rules.

Similarly, when you launch Design Assistant from the Timing Analyzer, Design Assistant is preset to check only a subset of rules that are helpful during timing analysis. You can cross-probe to the Timing Analyzer and design visualization tools to determine the root cause of violations.

When you enable or specify parameters for a rule check in analysis mode, those specifications do not apply to running Design Assistant in compilation mode. The rule settings for analysis mode are independent from the rule settings in compilation mode.

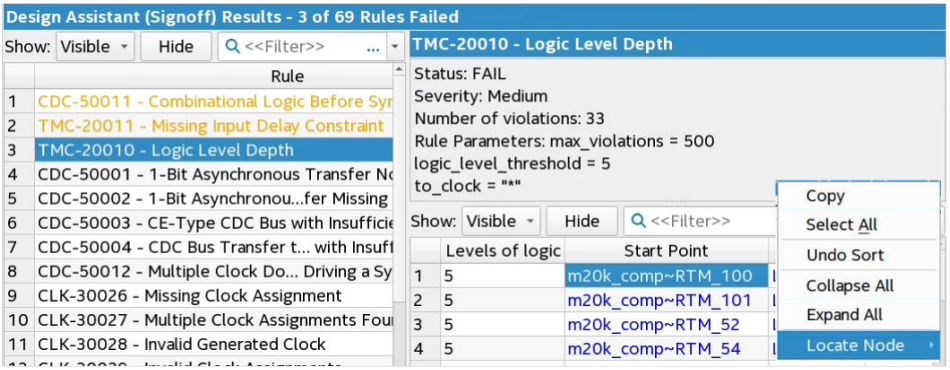
3.2.2.1. Cross-Probing from Design Assistant to Visualization Tools

Design Assistant can cross-probe from rule violations to the source in various Intel Quartus Prime design visualization tools. The following example demonstrates expanding from the cross-probing location for violation analysis.

The following example illustrates cross probing for the TMC-20010 Logic Level Depth rule violation to the RTL Viewer:

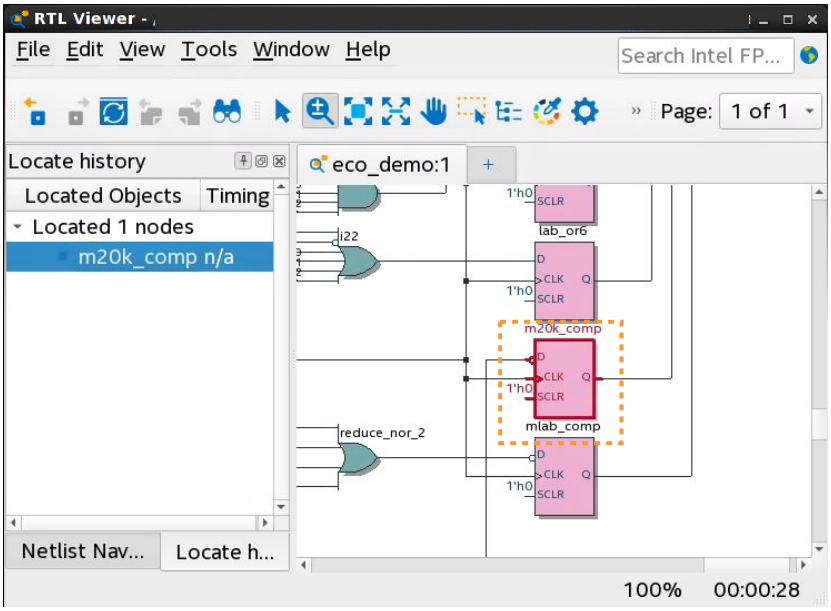
1. When Design Assistant reports FAIL status for rule TMC-20010, you can right-click any of the rule violations in the Design Assistant report, and then click **Locate Node ➤ Locate in RTL Viewer**.

Figure 89. Locate in RTL Viewer



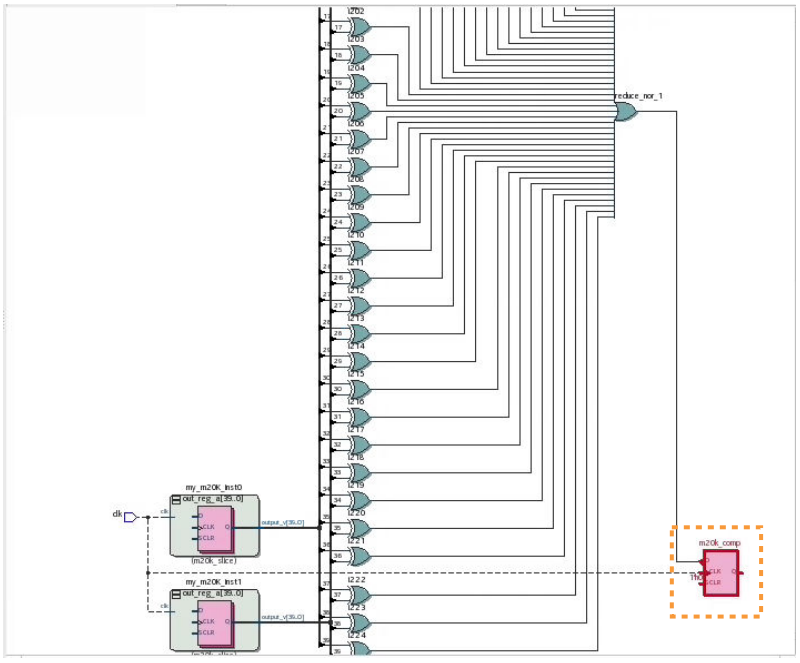
Cross-probing allows you to locate the driver register in the RTL Viewer.

Figure 90. Driver Register in RTL Viewer



2. To then fully visualize the logic level depth, right-click the register and click **Filter** to display **Sources and Destinations** of the register.

Figure 91. Expanded Connections

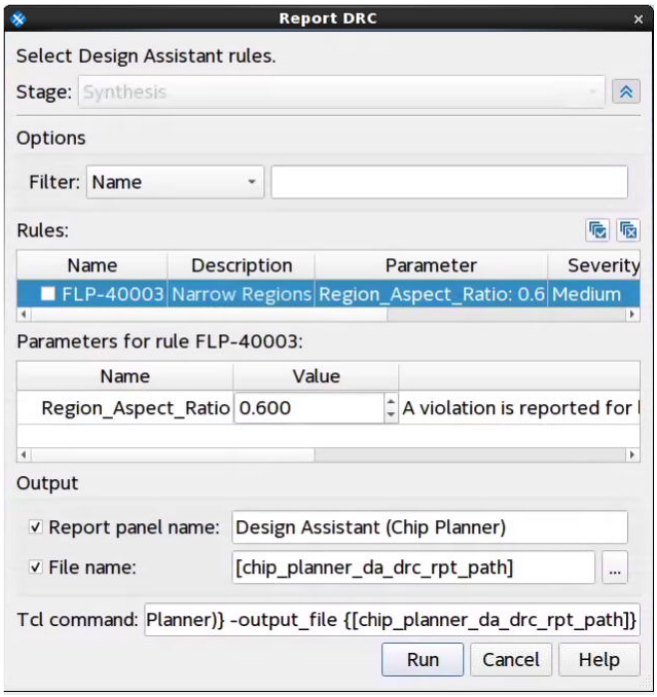


3.2.2.2. Launching Design Assistant from Chip Planner

You can run Design Assistant directly from Chip Planner to assist when optimizing the floorplan in the tool. When you launch Design Assistant from the Chip Planner, Design Assistant is preset to check only the FLP (floorplanning) Design Assistant rules. Follow these steps to run the Design Assistant from the Chip Planner:

1. Run any stage of the Compiler. You must run at least the Analysis & Elaboration stage before running Design Assistant from Chip Planner.
2. Click **Tools ➤ Chip Planner**.

Figure 92. Report DRC Dialog Box in Chip Planner



3. In Chip Planner **Tasks** pane, click **Report DRC** under **Design Assistant**. The **Report DRC** (design rule check) dialog box appears.
4. Under **Rules**, disable any rules that are not important to your analysis by removing the check mark.
5. Consider whether to adjust rule parameter values in the **Parameters** field.
6. Under **Output**, confirm the **Report panel name** and optionally specify an output **File name**.
7. Click **Run**. The Results reports generate and appear in the **Report** pane and in the Compilation Report.

Figure 93. Rule Violations in Chip Planner Reports Pane

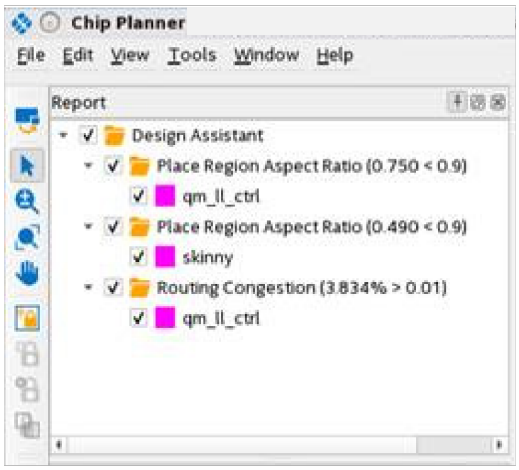
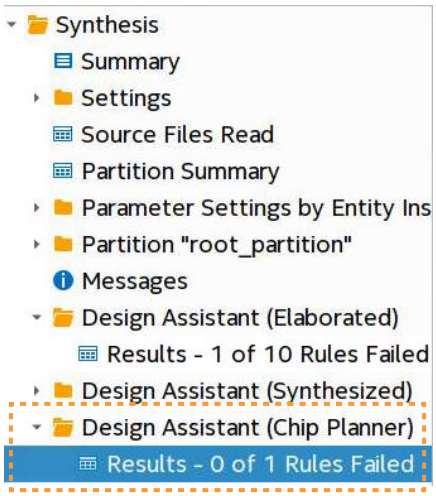


Figure 94. Chip Planner Rule Violations in Main Compilation Report



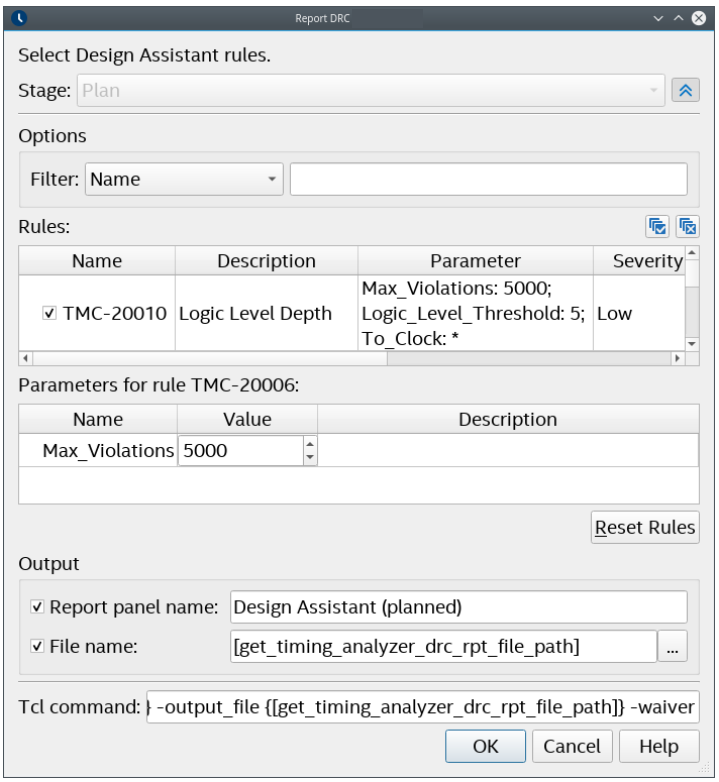
3.2.2.3. Launching Design Assistant from Timing Analyzer

You can run Design Assistant directly from the Timing Analyzer to assist when optimizing timing paths and other timing conditions. When you launch Design Assistant from the Timing Analyzer, Design Assistant is preset to check only rules that relate to timing analysis.

Follow these steps to run the Design Assistant from the Timing Analyzer:

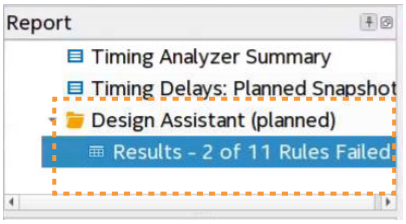
1. Compile the design through at least the Compiler's Plan stage.
2. Open the Timing Analyzer for the Compiler stage from the Compilation Dashboard.
3. In the Timing Analyzer, click **Reports ➤ Design Assistant ➤ Report DRC...**. The **Report DRC** (design rule check) dialog box opens.
4. Under **Rules**, disable any rules that are not important to your analysis by removing the check mark.
5. Consider whether to adjust rule parameter values in the **Parameters** field.

Figure 95. Report DRC (Design Rule Check) Dialog Box



6. Confirm the **Report panel name** and optionally specify an output **File name**.
7. Click **Run**. The Results reports generate and appear in the **Report** pane, as well as the main Compilation Report.

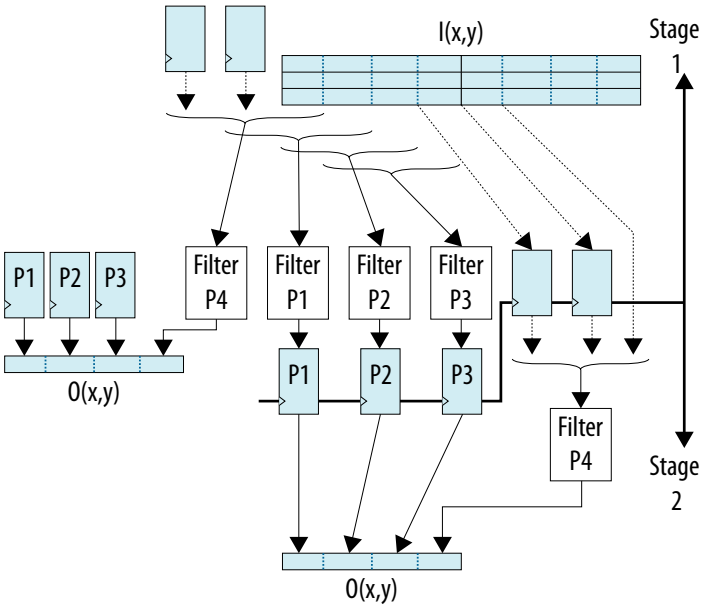
Figure 96. Design Assistant Reports in Timing Analyzer Report Pane



4. Design Example Walk-Through

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

Figure 97. Median Filter Operational Diagram

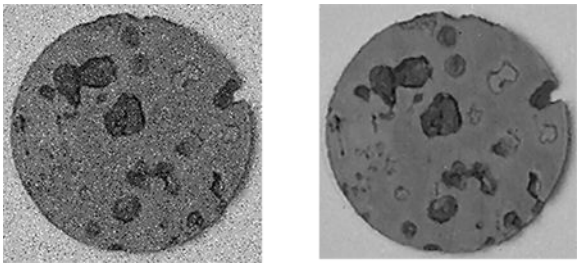


4.1. Median Filter Design Example

The Median filter is a non-linear filter that removes impulsive noise from an image. These filters require the highest performance. The design requirement is to perform real time image processing on a factory floor.(2)

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

Figure 98. Before and After Images Processed with Median Filtering



Median Filter Design Example Files

The Median filter design example `.zip` file contains the following directories under the `Median_filter_design_example_<version>` directory:

Table 10. Median Filter Design Example Files

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

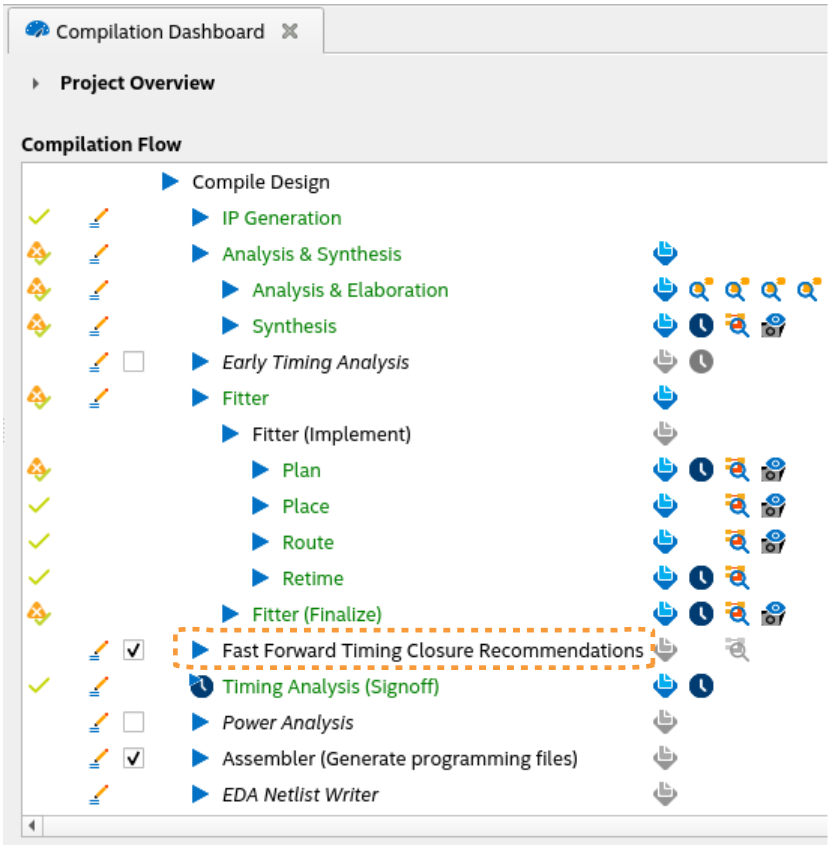
This walk-through covers the following steps:

1. Step 1: Compile the Base Design on page 88
2. Step 2: Add Pipeline Stages and Remove Asynchronous Resets on page 90
3. Step 3: Add More Pipeline Stages and Remove All Asynchronous Resets on page 93
4. Step 4: Optimize Short Path and Long Path Conditions on page 94

4.1.1. Step 1: Compile the Base Design

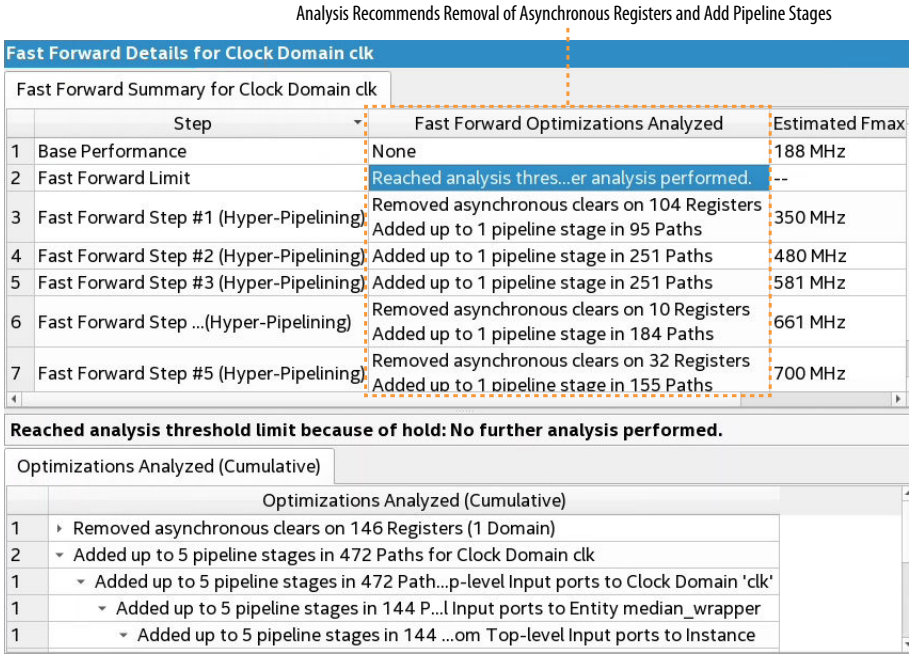
Follow these steps to compile the base design of the median project:

1. In the Intel Quartus Prime Pro Edition software, click **File ➤ Open Project** and select the `Median_filter_<version>/Base/median.qpf` project file. The base version of the design example opens.
2. To compile the base design, click **Compile Design** on the Compilation Dashboard. By default, the Fast Forward Timing Recommendations stage runs during the Fitter, and generates detailed recommendations in the **Fast Forward Details** report.



3. Click the report icon for **Fast Forward Timing Closure Recommendations**. In the **Fast Forward Details** report, view the compilation results for the `Clk` clock domain.

Figure 99. Fast Forward Details Report



The report indicates a **Base Performance** of 188 MHz, with the following design conditions limiting further optimization:

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

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

4.1.2. Step 2: Add Pipeline Stages and Remove Asynchronous Resets

This first optimization step adds five levels of pipeline registers in the design locations that Fast Forward suggests, and removes the asynchronous resets present in a design module. Adding additional pipeline stages at the interconnect between the ALMs eliminates some of the long routing delays. This optimization step increases f_{MAX} performance to the level that Fast Forward estimates.

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

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

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

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

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

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

```
. . .

hyper_pipe #(
    .WIDTH (DATA_WIDTH),
    .NUM_PIPES(4)
) hp_d0 (
    .clk(clk),
    .din(d0),
    .dout(q0_int)
);
. . .
always @(posedge clk)
begin : register_bank_3u
    if(~rst_n) begin
        q0 <= {DATA_WIDTH{1'b0}};
        q1 <= {DATA_WIDTH{1'b0}};
        q2 <= {DATA_WIDTH{1'b0}};
    end else begin
          q0 <= q0_int;
          q1 <= q1_int;
          q2 <= q2_int;
    end
end
```

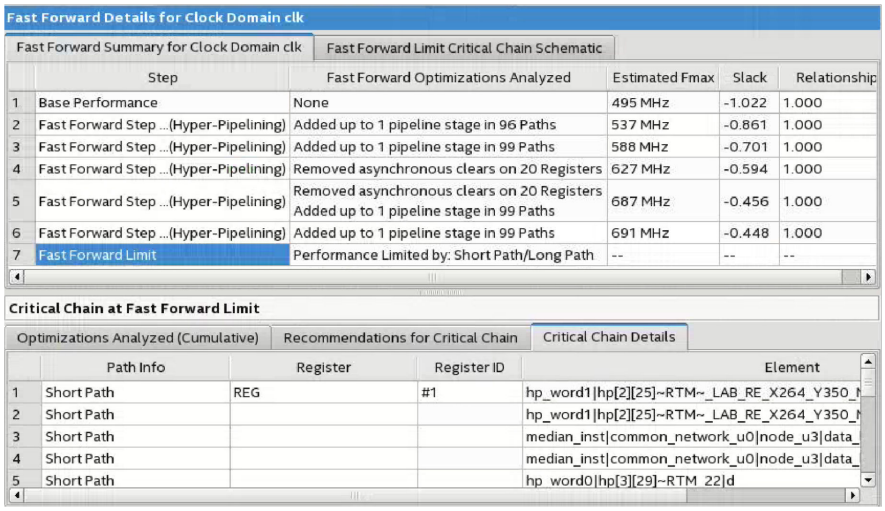
3. Remove the asynchronous resets inside the `dff_3_pipe` module by simply changing the registers to synchronous registers, as shown below. Refer to Reset Strategies on page 12 for general examples of efficient reset implementations.

```
always @(posedge clk or negedge rst_n) // Asynchronous reset
begin : register_bank_3u
    if(~rst_n) begin
        q0 <= {DATA_WIDTH{1'b0}};
        q1 <= {DATA_WIDTH{1'b0}};
        q2 <= {DATA_WIDTH{1'b0}};
    end else begin
        q0_reg <= d0;
        q1_reg <= d1;
        q2_reg <= d2;
          q0 <= q0_reg;
          q1 <= q1_reg;
          q2 <= q2_reg;
    end
end

always @(posedge clk)
```

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

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

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



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

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

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

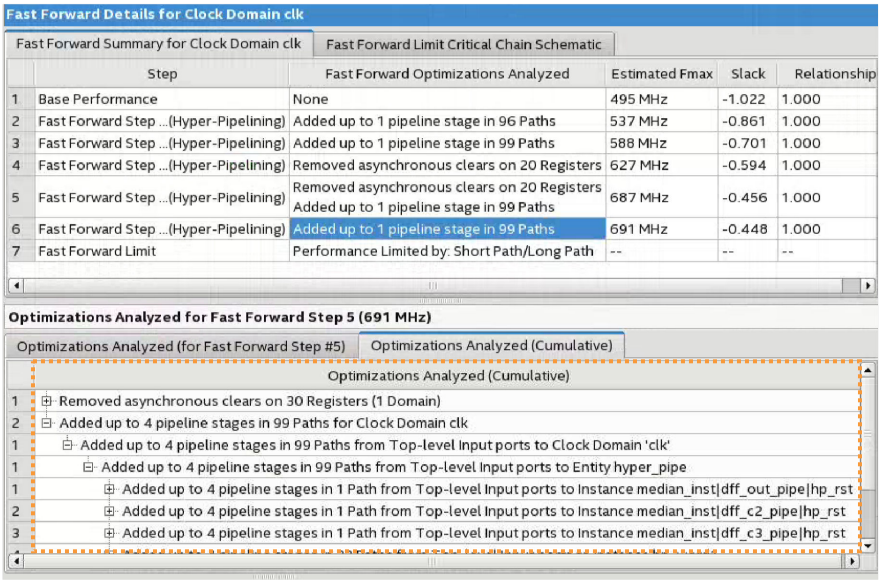
Related Information

- Removing Asynchronous Resets on page 13
- Hyper-Pipelining (Add Pipeline Registers) on page 29

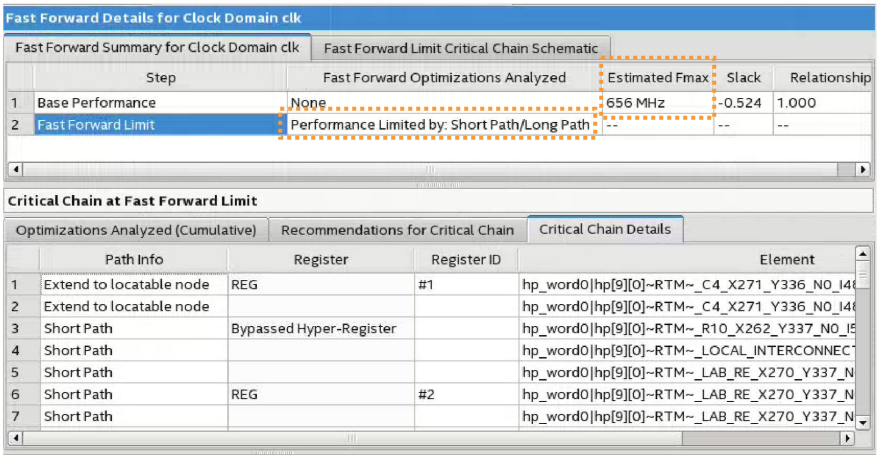
4.1.3. Step 3: Add More Pipeline Stages and Remove All Asynchronous Resets

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

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



3. Use the techniques and examples in Step 2: Add Pipeline Stages and Remove Asynchronous Resets on page 90 to add the number of pipeline stages at the locations in the **Optimizations Analyzed** tab.
4. Once again, compile the design and view the Fast Forward Details report. The performance increase is similar to the estimates, but short path and long path combinations still limit further performance. The next step addresses this performance limit.

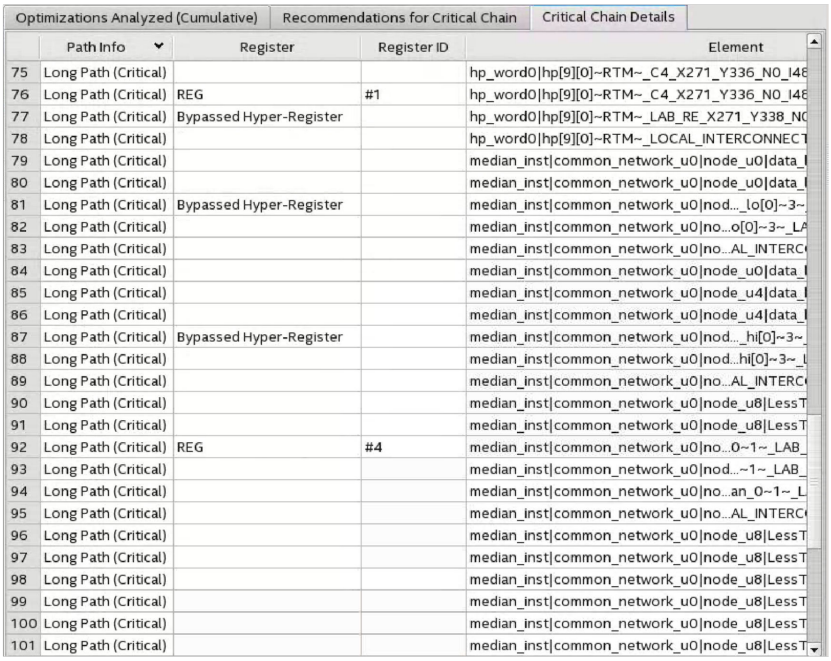


Note: As an alternative to completing the preceding steps, you can open and compile the `Median_filter_<version>/Step_2/median.qpf` project file that already includes these changes, and then observe the results.

4.1.4. Step 4: Optimize Short Path and Long Path Conditions

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

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



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

	Path Info	Register	Register ID	Element
75	Long Path (Critical)			hp_word0|hp[9][0]~RTM~_C4_X271_Y336_N0_I4
76	Long Path (Critical)	REG	#1	hp_word0|hp[9][0]~RTM~_C4_X271_Y336_N0_I4
77	Long Path (Critical)	Bypassed Hyper-Register		hp_word0|hp[9][0]~RTM~_LAB_RE_X271_Y338_N
78	Long Path (Critical)			hp_word0|hp[9][0]~RTM~_LOCAL_INTERCONNECT
79	Long Path (Critical)			median_inst|common_network_u0|node_u0|data_
80	Long Path (Critical)			median_inst|common_network_u0|node_u0|data_
81	Long Path (Critical)	Bypassed Hyper-Register		median_inst|common_network_u0|nod..._lo[0]~3~
82	Long Path (Critical)			median_inst|common_network_u0|no...o[0]~3~_LA
83	Long Path (Critical)			median_inst|common_network_u0|no...AL_INTERC
84	Long Path (Critical)			median_inst|common_network_u0|node_u0|data_
85	Long Path (Critical)			median_inst|common_network_u0|node_u4|data_
86	Long Path (Critical)			median_inst|common_network_u0|node_u4|data_
87	Long Path (Critical)	Bypassed Hyper-Register		median_inst|common_network_u0|nod..._hi[0]~3~_
88	Long Path (Critical)			median_inst|common_network_u0|nod...hi[0]~3~_
89	Long Path (Critical)			median_inst|common_network_u0|no...AL_INTERC
90	Long Path (Critical)			median_inst|common_network_u0|node_u8|LessT
91	Long Path (Critical)			median_inst|common_network_u0|node_u8|LessT
92	Long Path (Critical)	REG	#4	median_inst|common_network_u0|no...0~1~_LAB_
93	Long Path (Critical)			median_inst|common_network_u0|nod...~1~_LAB_
94	Long Path (Critical)			median_inst|common_network_u0|no...an_0~1~_L
95	Long Path (Critical)			median_inst|common_network_u0|no...AL_INTERC
96	Long Path (Critical)			median_inst|common_network_u0|node_u8|LessT
97	Long Path (Critical)			median_inst|common_network_u0|node_u8|LessT
98	Long Path (Critical)			median_inst|common_network_u0|node_u8|LessT
99	Long Path (Critical)			median_inst|common_network_u0|node_u8|LessT
100	Long Path (Critical)			median_inst|common_network_u0|node_u8|LessT
101	Long Path (Critical)			median_inst|common_network_u0|node_u8|LessT

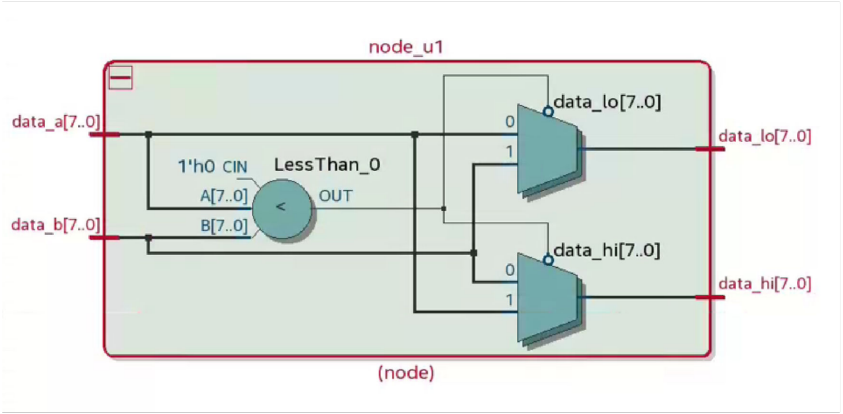
The following shows the original RTL for this path:

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

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

The Compiler infers the following logic from this RTL:

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

Figure 100. Node Component Connections

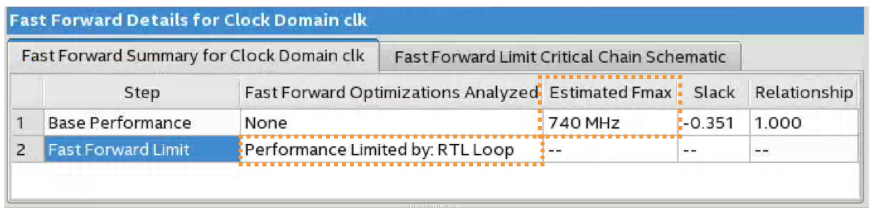


2. Review the `pixel_network.v` file that instantiates the `node` module. The `node` module's outputs are unconnected when you do not use them. These unconnected outputs result in no use of the `LOW_MUX` or `HI_MUX` code. Rather than inferring muxes, use bitwise logic operation to compute the values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

```
reg [DATA_WIDTH-1:0] sel0;

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

 data_lo = (data_b & sel0) | (data_a & sel0);
 data_hi = (data_a & sel0) | (data_b & sel0);
end
```

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

Figure 101. Short Path and Long Path Conditions Optimized



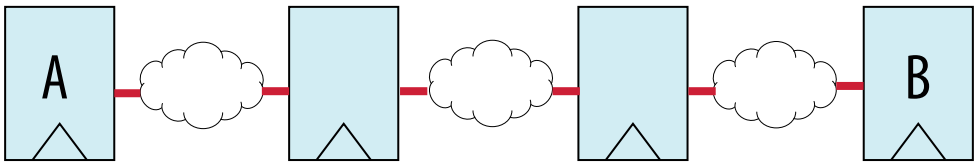
Note: As an alternative to completing the preceding steps, you can open and compile the `Median_filter_<version>/Final/median.qpf` project file that already includes these changes, and then observe the results.

5. Retiming Restrictions and Workarounds

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

In the diagram of a simple critical chain that follows, the red line represents the same critical chain. Timing restrictions prevent register A from retiming forward. Timing restrictions also prevent register B from retiming backwards. A loop occurs when register A and register B are the same register.

Figure 102. Sample Critical 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**—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 improves performance with more registers, and another path has no place for additional hyper-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. However, the Compiler cannot place additional registers in the loop.

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

Table 11. Hyper-Register Support for Various Design Conditions

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

continued...

Design Condition	Hyper-Register Support
	number for that particular synchronizer chain **Synchronization Register Chain Length** (default is 3). In some cases a synchronizer chain isn't necessary, and shouldn't be inferred.
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 Information

Timing Constraint Considerations on page 19

5.1. Setting the dont_merge Synthesis Attribute

You can set the `dont_merge` attribute in your HDL code, as shown in the following examples.

Table 12. Setting the attribute in HDL code

`dont_merge` prevents the `my_reg` register from merging.

HDL	Code
Verilog HDL	`reg my_reg /* synthesis dont_merge */;`
Verilog-2001 and SystemVerilog	`(* dont_merge *) reg my_reg;`

VHDL	`signal my_reg : stdlogic;`
	`attribute dont_merge : boolean;`
	`attribute dont_merge of my_reg : signal is true;`

Related Information

Avoid Broadcast Signals on page 9

5.2. 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.

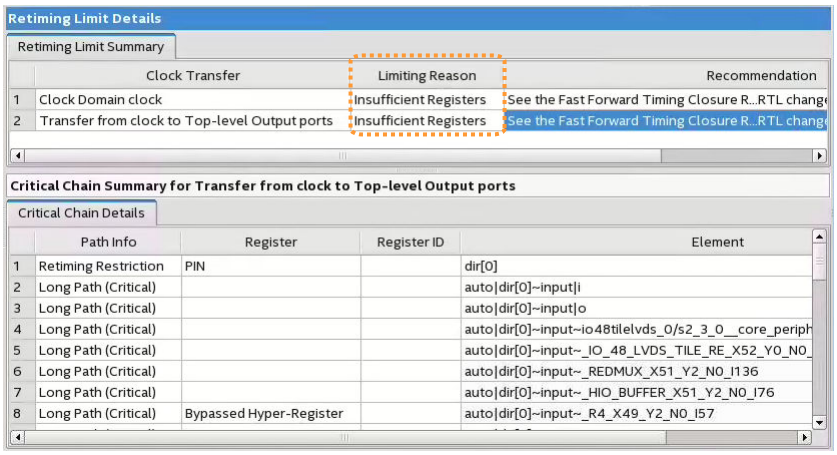
The Retiming Limit Details report the limiting reasons preventing further retiming, and the registers and combinational nodes that comprise the chain. The Fast Forward recommendations list the steps you can take to remove critical chains and enable additional register retiming.

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

5.2.1. Insufficient Registers

When the Compiler cannot retime registers at either end of the chain, but adding more registers would improve performance, the Retiming Limit Details report shows Insufficient Registers as the **Limiting Reason**.

Figure 103. Insufficient Registers Reported as Limiting Reason

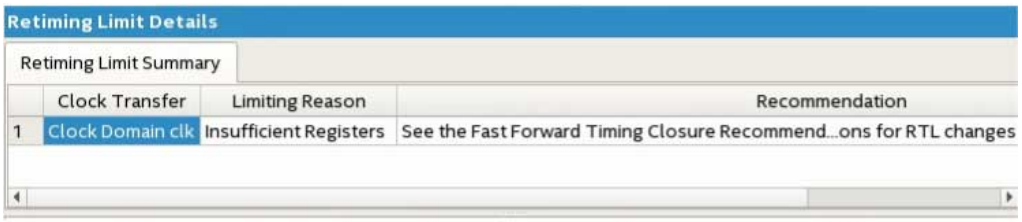


5.2.1.1. Insufficient Registers Example

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

The Retiming Limit Details report indicates that the performance of the `clk` domain fails to meet the timing requirement .

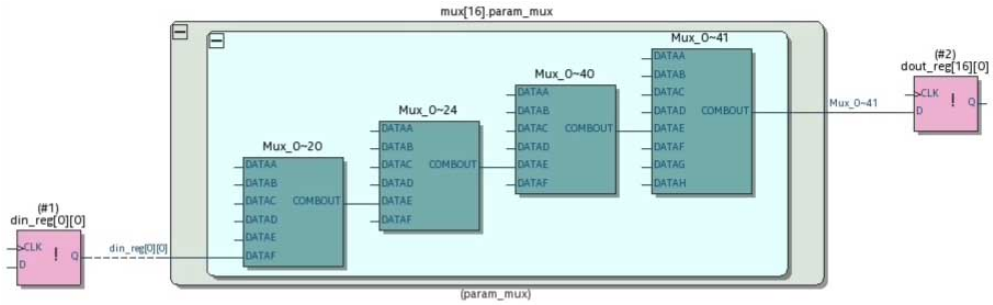
Figure 104. Retiming Limit Details



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

In 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.

Figure 105. Critical Chain in Post-Fit Technology Map Viewer



In Critical Chain with Insufficient Registers Reported During Hyper-Retiming, line 1 shows a timing restriction in the **Path Info** column. Line 33 also lists a retiming restriction. The asynchronous resets on the two registers cause the retiming restrictions.

Figure 106. Critical Chain with Insufficient Registers Reported During Hyper-Retiming

Retiming Limit Details

Retiming Limit Summary

	Clock Transfer	Limiting Reason	Recommendation
1	Clock Domain clk	Insufficient Registers	See the Fast Forward Tim...nd estimated performance

Critical Chain Summary for Clock Domain clk

Recommendations for Critical Chain | Critical Chain Details

	Path Info	Register	Register ID	Element
1	Retiming Restriction	REG (Metastability required)	#1	din_reg[0][0]
2	Long Path (Critical)			din_reg[0][0]|q
3	Long Path (Critical)			din_reg[0][0]~LAB_RE_X267_Y239_N0_I96
4	Long Path (Critical)	Bypassed Hyper-Register		din_reg[0][0]~R4_X267_Y239_N0_I55
5	Long Path (Critical)			din_reg[0][0]~R24_C16_INTERCONNECT_DRIVER_X270_Y239_N0_I0
6	Long Path (Critical)	Bypassed Hyper-Register		din_reg[0][0]~R24_X247_Y239_N0_I0
7	Long Path (Critical)	Bypassed Hyper-Register		din_reg[0][0]~R4_X256_Y239_N0_I63
8	Long Path (Critical)			din_reg[0][0]~LOCAL_INTERCONNECT_X258_Y239_N0_I11
9	Long Path (Critical)	Bypassed Hyper-Register		din_reg[0][0]~LAB_RE_X258_Y239_N0_I23
10	Long Path (Critical)			mux[16].param_mux|Mux_0~20|dataf
11	Long Path (Critical)			mux[16].param_mux|Mux_0~20|combout
12	Long Path (Critical)			mux[16].param_mux|Mux_0~20~_LAB_RE_X258_Y239_N0_I103
13	Long Path (Critical)	Bypassed Hyper-Register		mux[16].param_mux|Mux_0~20~_R2_X259_Y239_N0_I57
14	Long Path (Critical)			mux[16].param_mux|Mux_0~20~_LOCAL_INTERCONNECT_X259_Y239_N0_I34
15	Long Path (Critical)	Bypassed Hyper-Register		mux[16].param_mux|Mux_0~20~_LAB_RE_X259_Y239_N0_I32
16	Long Path (Critical)			mux[16].param_mux|Mux_0~24|datae
17	Long Path (Critical)			mux[16].param_mux|Mux_0~24|combout
18	Long Path (Critical)			mux[16].param_mux|Mux_0~24~_LAB_RE_X259_Y239_N0_I110
19	Long Path (Critical)			mux[16].param_mux|Mux_0~24~_LOCAL_INTERCONNECT_X258_Y239_N0_I42
20	Long Path (Critical)	Bypassed Hyper-Register		mux[16].param_mux|Mux_0~24~_LAB_RE_X258_Y239_N0_I56
21	Long Path (Critical)			mux[16].param_mux|Mux_0~40|datae
22	Long Path (Critical)			mux[16].param_mux|Mux_0~40|combout
23	Long Path (Critical)			mux[16].param_mux|Mux_0~40~_LAB_RE_X258_Y239_N0_I122
24	Long Path (Critical)	Bypassed Hyper-Register		mux[16].param_mux|Mux_0~40~_R4_X254_Y239_N0_I15
25	Long Path (Critical)			mux[16].param_mux|Mux_0~40~_R24...RCONNECT_DRIVER_X253_Y239_N0_I2
26	Long Path (Critical)	Bypassed Hyper-Register		mux[16].param_mux|Mux_0~40~_C16_X253_Y223_N0_I53
27	Long Path (Critical)	Bypassed Hyper-Register		mux[16].param_mux|Mux_0~40~_R10_X244_Y227_N0_I8
28	Long Path (Critical)			mux[16].param_mux|Mux_0~40~_LOCAL_INTERCONNECT_X253_Y227_N0_I6
29	Long Path (Critical)	Bypassed Hyper-Register		mux[16].param_mux|Mux_0~40~_LAB_RE_X253_Y227_N0_I16
30	Long Path (Critical)			mux[16].param_mux|Mux_0~41|datae
31	Long Path (Critical)			mux[16].param_mux|Mux_0~41|combout
32	Long Path (Critical)			dout_reg[16][0]|d
33	Retiming Restriction	REG (Metastability required)	#2	dout_reg[16][0]

Correlation Between Critical Chain Elements and Technology Map Viewer shows the correlation between critical chain elements and the Technology Map Viewer examples.

Table 13. Correlation Between Critical Chain Elements and Technology Map Viewer

Line Numbers in Critical Chain Report	Circuit Element in the Technology Map Viewer
1-2	`din_reg[0][0]` source register and its output
3-9	FPGA routing fabric between `din_reg[0][0]` and `Mux_0~20`, the first stage of mux in the crossbar
10-11	Combinational logic implementing `Mux_0~20`
12-15	Routing between `Mux_0~20` and `Mux_0~24`, the second stage of mux in the crossbar
16-17	Combinational logic implementing `Mux0~24`
18-20	Routing between `Mux0~24` and `Mux0~40`, the third stage of mux in the crossbar
21-22	Combinational logic implementing `Mux_0~40`
23-29	Routing between `Mux_0~40` and `Mux_0~41`, the fourth stage of mux in the crossbar
30-31	Combinational logic implementing `Mux_0~41`
32-33	`dout_reg[16][0]` destination register

In the critical chain report in Critical Chain with Insufficient Registers Reported During Hyper-Retiming, there are 11 lines that list bypass Hyper-Register in the **Register** column. Bypassed Hyper-Register indicates the location of a Hyper-Register for use if there are more registers in the chain, or if there are no restrictions on the endpoints. If there are no restrictions on the endpoints, the Compiler can retime the endpoint registers, or retime other registers from outside the critical chain into the critical chain. If the RTL design contains more registers through the crossbar switch, there are more registers that the Compiler can retime. Fast Forward compilation can also insert more registers to increase the performance.

In the critical chain report, lines 2 to 32 list "Long Path (Critical)" in the **Path Info** column. This indicates that the path is too long to run above the listed frequency. The "Long Path" designation is also related to the Short Path/Long Path type of critical chain. Refer to the *Short Path/Long Path* section for more details. The (Critical) designation exists on one register-to-register segment of a critical chain. The (Critical) designation indicates that the register-to-register path is the most critical timing path in the clock domain.

The **Register ID** column contains a "#1" on line 1, and a "#2" on line 33. The information in the **Register ID** column helps interpret more complex critical chains. For more details, refer to *Complex Critical Chains* section.

The **Element** column in Critical Chain with Insufficient Registers Reported During Hyper-Retiming shows the name of the circuit element or routing resource at each step in the critical chain. You can right-click the names to copy them, or cross probe to other parts of the Intel Quartus Prime software with the **Locate** option.

Related Information

- Short Path/Long Path on page 104
- Complex Critical Chains on page 113
- Hyper-Retiming (Facilitate Register Movement) on page 11

5.2.1.2. Optimizing Insufficient Registers

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

Related Information

- Hyper-Retiming (Facilitate Register Movement) on page 11
- Hyper-Pipelining (Add Pipeline Registers) on page 29

5.2.1.3. Critical Chains with Dual Clock Memories

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

If the limiting reason is Insufficient Registers, and the chain is between dual clock memories, you can add pipeline stages to the functional block. Alternatively, add a bank of registers in the RTL, and then allow the Compiler to balance the registers. Refer to the *Hyper-Pipelining (Add Pipeline Registers)*, *Add Pipeline Stages and Remove Asynchronous Resets*, and *Appendix A: Parameterizable Pipeline Modules* for a pipelining techniques and examples.

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 Information

- Appendix A: Parameterizable Pipeline Modules on page 132
- Hyper-Pipelining (Add Pipeline Registers) on page 29
- Step 2: Add Pipeline Stages and Remove Asynchronous Resets on page 90

5.2.2. Short Path/Long Path

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

A critical chain is categorized as short path/long path when there are conflicting optimization goals for Hyper-Retiming. Short paths and long paths are always connected in some way, with at least one common node. Retimed registers must maintain functional correctness and ensure identical relative latency through both critical chains. This requirement can result in conflicting optimization goals. Therefore, one segment (the long path) can accept the retiming move, but the other segment (the short path) cannot accept the retiming move. The retiming move is typically retiming an additional register into the short and long paths. Figure 108 on page 106 illustrates this concept.

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

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

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

5.2.2.1. Hyper-Register Locations Not Available

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

In Figure 107 on page 105, the short path includes two Hyper-Register locations that are in use. One or more names in the **Element** column end in `_dff`, indicating that the Hyper-Registers in those locations are in use. The `_dff` represents the D flop-flop. No other Hyper-Register locations are available for use in that chain segment. Available Hyper-Register locations indicate status with a bypassed Hyper-Register entry in the **Register** column.

Figure 107. Critical Chain Short Path Segment with no Available Hyper-Register Locations

Critical Chain at Fast Forward Limit

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

	Path Info	Register	Register ID	Element
1	Short Path	REG	#1	round_robin_last_r[1]~LOCAL_INTERCONNECT_X237_Y50_N0_I21_dff
2	Short Path			round_robin_last_r[1]~LOCAL_INTERCONNECT_X237_Y50_N0_I21
3	Short Path			round_robin_last_r[1]~LAB_RE_X237_Y50_N0_I5_dff.d
4	Short Path	REG		round_robin_last_r[1]~LAB_RE_X237_Y50_N0_I5_dff
5	Short Path			round_robin_last_r[1]~LAB_RE_X237_Y50_N0_I5
6	Short Path			rr|next[0]~11|dataf
7	Short Path			rr|next[0]~11|combout
8	Short Path			round_robin_next[1]~reg0|d
9	Short Path	REG (required)	#2	round_robin_next[1]~reg0

5.2.2.2. Example for Hold Optimization

For some designs, the **Register** column indicates unusable (hold). This data implies that you cannot use this register location because the location does not meet hold time requirements. The Compiler cannot retime (forward or backward) the registers that occur before or after the register that indicates unusable (hold).

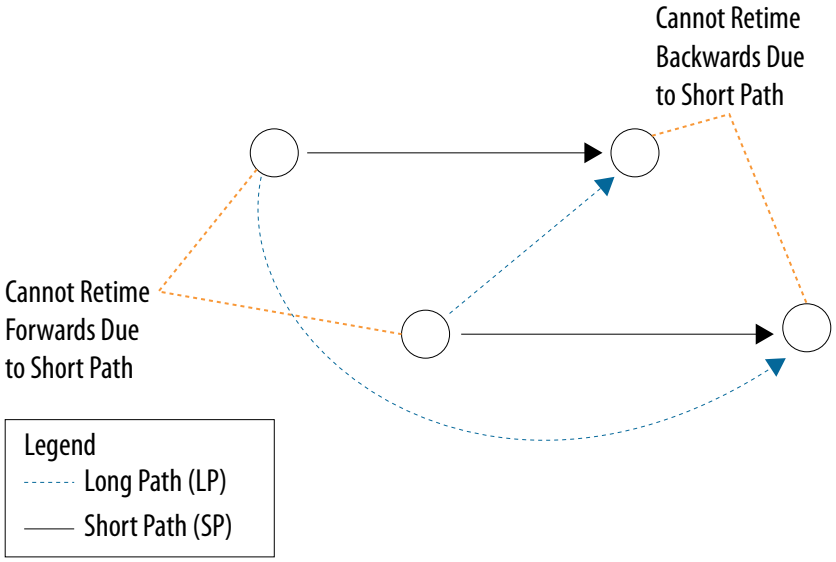
5.2.2.3. Optimizing Short Path/Long Path

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

5.2.2.4. Add Registers

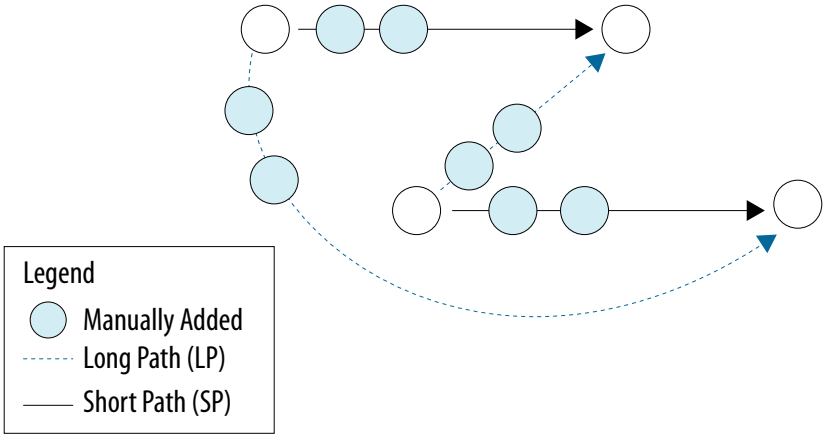
Manually adding registers on both the short and long paths can be helpful if you can accommodate the extra latency in the critical chain.

Figure 108. Critical Chain with Alternating Short Path/Long Path



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

Figure 109. Sample Short Path/Long Path with Additional Latency



5.2.2.5. Duplicate Common Nodes

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

Figure 110. Critical Chain with Alternating Short Path/Long Path

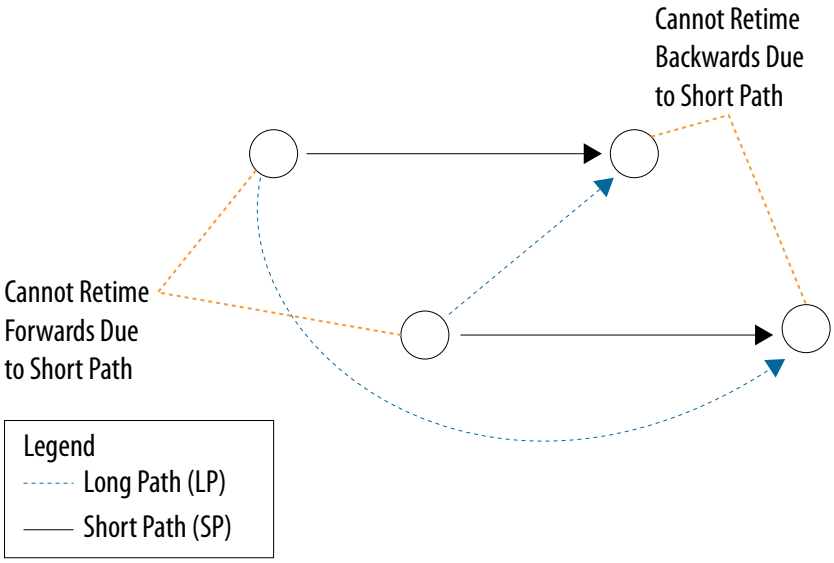
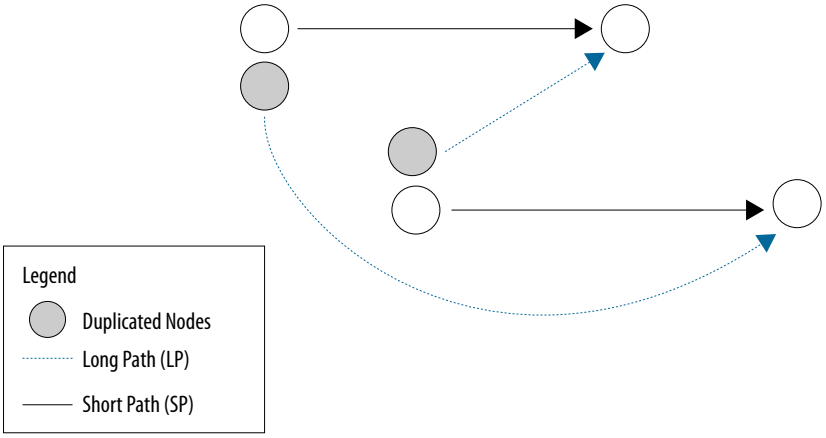


Figure 111. Short Path/Long Path with Two Duplicate Nodes



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

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

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

5.2.2.6. Data and Control Plane

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

5.2.3. Fast Forward Limit

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

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

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

5.2.3.1. Optimizing Path Limit

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

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

Turn on optional registers using the IP parameter editor to parameterize hard DSP or memory blocks. If you infer DSP or memory functions from your RTL, ensure that you follow the *Recommended HDL Coding Styles* to ensure that you use the optional input

and output registers of the hard blocks. The Compiler does not retime into or out of DSP and hard memory block registers. Instantiate the optional registers to achieve maximum performance.

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

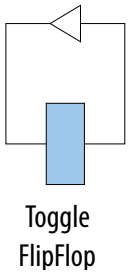
Related Information

- Recommended HDL Coding Styles
- Intel Hyperflex Architecture True Dual-Port Memory on page 65

5.2.4. 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 112. Simple Loop



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

Figure 113. FIFO Flow Control Loop

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

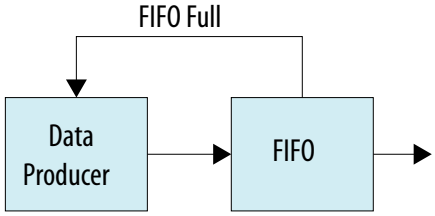


Figure 114. Counter and Accumulator Loop

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

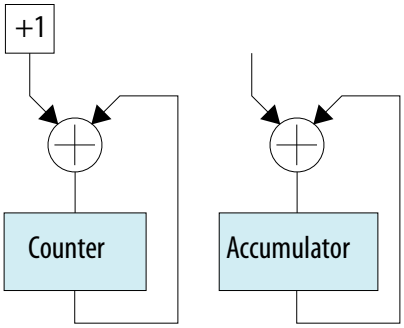


Figure 115. State Machine Loop

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

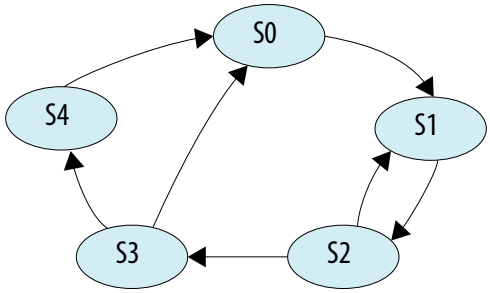
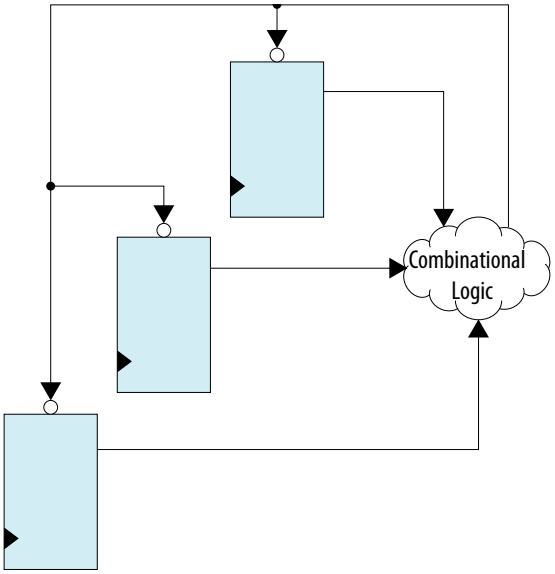


Figure 116. Reset Circuit Loop

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



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

5.2.4.1. Example of Loops Limiting the Critical Chain

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

Figure 117. Fast Forward Details Report showing Limiting Reason for Hyper-Optimization is a Loop

Fast Forward Details for Clock Domain clk_user

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

	Step	Fast Forward Optimizations Analyzed	Estimated Fmax	Slack	Relationship
1	Base Performance	None	438 MHz	-1.282	1.000
2	Fast Forward Limit	Performance Limited by: RTL Loop	--	--	--

In Figure 118 on page 111, the **Register ID** for the start and end points is the same, which is #1. This case indicates that the start and end points of the chain are the same, thereby creating a loop.

Figure 118. Critical Chain with Loop (lines 1-34)

Critical Chain at Fast Forward Limit

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

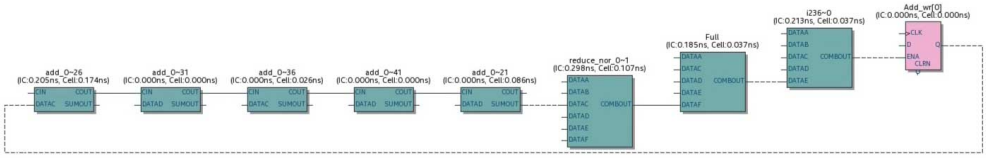
	Path Info	Register	Register ID	Element
1	Extend to locatable node	REG	#1	U_MAC_tx|U_MAC_tx_FF|Add_wr[0]
2	Extend to locatable node			U_MAC_tx|U_MAC_tx_FF|Add_wr[0]|q
3	Extend to locatable node			U_MAC_tx|U_MAC_tx_FF|Add_wr[0]~LAB_RE_X173_Y113_N0_I114
4	Extend to locatable node			U_MAC_tx|U_MAC_tx_FF|Add_wr[0]~L...AL_INTERCONNECT_X173_Y113_N0_I16
5	Long Path	Bypassed Hyper-Register		U_MAC_tx|U_MAC_tx_FF|Add_wr[0]~LAB_RE_X173_Y113_N0_I42
6	Long Path			U_MAC_tx|U_MAC_tx_FF|add_0~26|datac
7	Long Path			U_MAC_tx|U_MAC_tx_FF|add_0~31|cout
8	Long Path			U_MAC_tx|U_MAC_tx_FF|add_0~36|cin
9	Long Path			U_MAC_tx|U_MAC_tx_FF|add_0~41|cout
10	Long Path			U_MAC_tx|U_MAC_tx_FF|add_0~21|cin
11	Long Path			U_MAC_tx|U_MAC_tx_FF|add_0~21|sumout
12	Long Path			U_MAC_tx|U_MAC_tx_FF|add_0~21~la_lab/laboutb[8]
13	Long Path			U_MAC_tx|U_MAC_tx_FF|add_0~21~_LAB_RE_X173_Y113_N0_I122
14	Long Path			U_MAC_tx|U_MAC_tx_FF|add_0~21~_...L_INTERCONNECT_X173_Y113_N0_I21
15	Long Path	Bypassed Hyper-Register		U_MAC_tx|U_MAC_tx_FF|add_0~21~_LAB_RE_X173_Y113_N0_I18
16	Long Path			U_MAC_tx|U_MAC_tx_FF|reduce_nor_0~1|datac
17	Long Path			U_MAC_tx|U_MAC_tx_FF|reduce_nor_0~1|combout
18	Long Path			U_MAC_tx|U_MAC_tx_FF|reduce_nor_0~1~_LAB_RE_X173_Y113_N0_I103
19	Long Path			U_MAC_tx|U_MAC_tx_FF|reduce_nor_...CAL_INTERCONNECT_X174_Y113_N0_I8
20	Long Path	Bypassed Hyper-Register		U_MAC_tx|U_MAC_tx_FF|reduce_nor_0~1~_LAB_RE_X174_Y113_N0_I23
21	Long Path			U_MAC_tx|U_MAC_tx_FF|Full|dataf
22	Long Path			U_MAC_tx|U_MAC_tx_FF|Full|combout
23	Long Path			U_MAC_tx|U_MAC_tx_FF|Full~la_lab/laboutt[8]
24	Long Path			U_MAC_tx|U_MAC_tx_FF|Full~LAB_RE_X174_Y113_N0_I102
25	Long Path			U_MAC_tx|U_MAC_tx_FF|Full~LOCAL_INTERCONNECT_X174_Y113_N0_I7
26	Long Path	Bypassed Hyper-Register		U_MAC_tx|U_MAC_tx_FF|Full~LAB_RE_X174_Y113_N0_I39
27	Long Path			U_MAC_tx|U_MAC_tx_FF|i236~0|datae
28	Long Path			U_MAC_tx|U_MAC_tx_FF|i236~0|combout
29	Long Path			U_MAC_tx|U_MAC_tx_FF|i236~0~_LAB_RE_X174_Y113_N0_I112
30	Long Path			U_MAC_tx|U_MAC_tx_FF|i236~0~_LOCAL_INTERCONNECT_X173_Y113_N0_I43
31	Long Path			U_MAC_tx|U_MAC_tx_FF|i236~0~_LAB_RE_X173_Y113_N0_I86
32	Long Path	Bypassed Hyper-Register		U_MAC_tx|U_MAC_tx_FF|Add_wr[8]~xbot...ale1/xlut/xreghipi/xcetop0/hipi_out
33	Long Path			U_MAC_tx|U_MAC_tx_FF|Add_wr[2]|ena
34	Long Path (Critical)	REG	#2	U_MAC_tx|U_MAC_tx_FF|Add_wr[2]

Figure 119. Critical Chain with Loop (lines 35-65)

35	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|Add_wr[2]|q
36	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|Add_wr[2]~LAB_RE_X173_Y113_N0_I118
37	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|Add_wr[2]~L...AL_INTERCONNECT_X173_Y113_N0_I20
38	Long Path (Critical)	Bypassed Hyper-Register		U_MAC_tx|U_MAC_tx_FF|Add_wr[2]~LAB_RE_X173_Y113_N0_I50
39	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|add_0~36|datac
40	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|add_0~41|cout
41	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|add_0~21|cin
42	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|add_0~21|sumout
43	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|add_0~21~la_lab/laboutb[8]
44	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|add_0~21~_LAB_RE_X173_Y113_N0_I122
45	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|add_0~21~_...L_INTERCONNECT_X173_Y113_N0_I21
46	Long Path (Critical)	Bypassed Hyper-Register		U_MAC_tx|U_MAC_tx_FF|add_0~21~_LAB_RE_X173_Y113_N0_I18
47	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|reduce_nor_0~1|datac
48	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|reduce_nor_0~1|combout
49	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|reduce_nor_0~1~_LAB_RE_X173_Y113_N0_I103
50	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|reduce_nor_...CAL_INTERCONNECT_X174_Y113_N0_I8
51	Long Path (Critical)	Bypassed Hyper-Register		U_MAC_tx|U_MAC_tx_FF|reduce_nor_0~1~_LAB_RE_X174_Y113_N0_I23
52	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|Full|dataf
53	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|Full|combout
54	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|Full~la_lab/laboutt[8]
55	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|Full~LAB_RE_X174_Y113_N0_I102
56	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|Full~LOCAL_INTERCONNECT_X174_Y113_N0_I7
57	Long Path (Critical)	Bypassed Hyper-Register		U_MAC_tx|U_MAC_tx_FF|Full~LAB_RE_X174_Y113_N0_I39
58	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|i236~0|datae
59	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|i236~0|combout
60	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|i236~0~_LAB_RE_X174_Y113_N0_I112
61	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|i236~0~_LOCAL_INTERCONNECT_X173_Y113_N0_I43
62	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|i236~0~_LAB_RE_X173_Y113_N0_I86
63	Long Path (Critical)	Bypassed Hyper-Register		U_MAC_tx|U_MAC_tx_FF|Add_wr[8]~xbot...ale0/xlut/xreghipi/xcetop0/hipi_out
64	Long Path (Critical)			U_MAC_tx|U_MAC_tx_FF|Add_wr[0]|ena
65	Long Path (Critical)	REG	#1	U_MAC_tx|U_MAC_tx_FF|Add_wr[0]

Figure 120 on page 112 shows the output of the `Addr_wr[0]` register feeding back to its enable input through eight levels of combinational logic.

Figure 120. Critical Chain in Technology Map Viewer



The figure does not show the other inputs to the logic cone for the `Addr_wr[0]` register. Source Code for Critical Chain on page 112 shows portions of the source, and some inputs to the `Addr_wr` registers.

Example 16. Source Code for Critical Chain

```
assign          Add_wr_pluse        =Add_wr+1;
assign          Add_wr_pluse_pluse  =Add_wr+4;

always @ (Add_wr_pluse or Add_rd_ungray)
    if (Add_wr_pluse==Add_rd_ungray)
        Full    =1;
    else
        Full    =0;

always @ (posedge Clk_SYS or posedge Reset)
    if (Reset)
        Add_wr  <= 0;
    else if (Wr_en&&!Full)
        Add_wr  <= Add_wr +1;
```

5.2.5. One Critical Chain per Clock Domain

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

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

Related Information

Critical Chains in Related Clock Groups on page 113

5.2.6. Critical Chains in Related Clock Groups

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

5.2.7. Complex Critical Chains

Complex critical chains consist of several segments connected with multiple join points. A join point is indicated with a positive integer in the **Register ID** 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 14. Sample Critical Chain

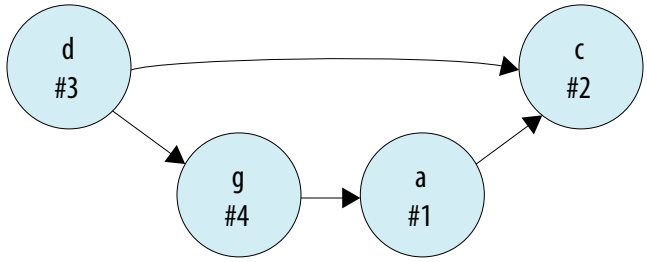
Path Info	Register	Register ID	Element
	REG	#1	a
			b
	REG	#2	c
----------	----------	----------	----------
	REG	#3	d
			e
	REG	#2	c
----------	----------	----------	----------
	REG	#3	d

continued...

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

Figure 121. 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.



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

5.2.8. Extend to locatable node

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

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

5.2.9. Domain Boundary Entry and Domain Boundary Exit

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

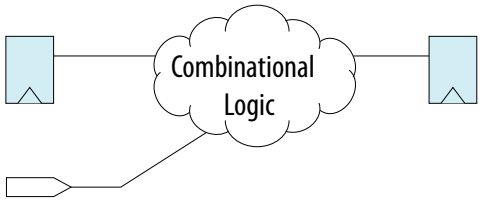
A domain boundary entry refers to a point in the design topology, at a clock domain boundary, where Hyper-Retiming can insert register stages (where latency can enter the clock domain) if Hyper-Pipelining is enabled. The concept of a domain boundary entry is independent of the dataflow direction. Hyper-Retiming can insert register

stages at the input of a module, and perform forward retiming pushes. Hyper-Retiming can also insert register stages at the output of a module, and perform backward retiming pushes. These insertions occur at domain boundary entry points.

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

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

Figure 122. Domain Boundary with Unregistered Input/Output



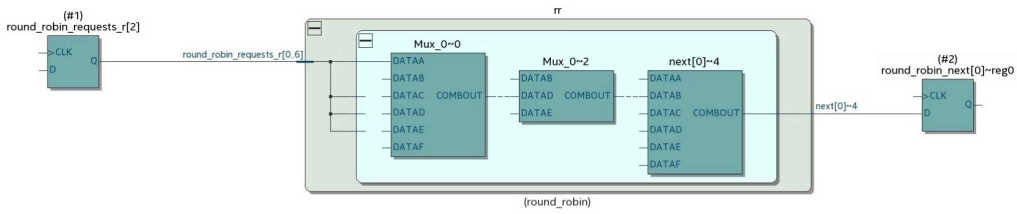
The register-to-register path might be shown as a critical chain segment with a domain boundary entry or a domain boundary exit, depending on how the path restricts Hyper-Retiming. The unregistered input prevents the Hyper-Retiming from inserting register stages at the domain boundary, because the input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

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

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

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

Figure 123. Critical Chain Schematic with Domain Boundary



The limiting reason for the base compile is Insufficient Registers.

Figure 124. Retiming Limit Summary with Insufficient Registers

Retiming Limit Details

Retiming Limit Summary

	Clock Transfer	Limiting Reason	Recommendation
1	Clock Domain clk_mod	Insufficient Registers	See the Fast Forward ...estimated performance
2	Clock Domain clk	Insufficient Registers	See the Fast Forward ...estimated performance

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

Figure 125. Critical Chain with Domain Boundary Entry

Recommendations for Critical Chain | Critical Chain Details

	Path Info	Register	Register ID	Element
1	Domain Boundary Entry	REG (required)	#1	round_robin_requests_r[2]
2	Long Path (Critical)			round_robin_requests_r[2]|q
3	Long Path (Critical)			round_robin_requests_r[2]~LAB_RE_X236_Y50_N0_I103
4	Long Path (Critical)	bypassed Hyper-Register		round_robin_requests_r[2]~LOCAL_INTERCONNECT_X237_Y50_N0_I8
5	Long Path (Critical)	bypassed Hyper-Register		round_robin_requests_r[2]~LAB_RE_X237_Y50_N0_I43
6	Long Path (Critical)			rr|Mux_0~0|dataa
7	Long Path (Critical)			rr|Mux_0~0|combout
8	Long Path (Critical)			rr|Mux_0~0~la_lab/laboutb[1]
9	Long Path (Critical)			rr|Mux_0~0~_LAB_RE_X237_Y50_N0_I115
10	Long Path (Critical)	bypassed Hyper-Register		rr|Mux_0~0~_LOCAL_INTERCONNECT_X237_Y50_N0_I45
11	Long Path (Critical)	bypassed Hyper-Register		rr|Mux_0~0~_LAB_RE_X237_Y50_N0_I57
12	Long Path (Critical)			rr|Mux_0~2|datad
13	Long Path (Critical)			rr|Mux_0~2|combout
14	Long Path (Critical)			rr|Mux_0~2~_LAB_RE_X237_Y50_N0_I123
15	Long Path (Critical)	bypassed Hyper-Register		rr|Mux_0~2~_LOCAL_INTERCONNECT_X237_Y50_N0_I52
16	Long Path (Critical)	bypassed Hyper-Register		rr|Mux_0~2~_LAB_RE_X237_Y50_N0_I12
17	Long Path (Critical)			rr|next[0]~4|datab
18	Long Path (Critical)			rr|next[0]~4|combout
19	Long Path (Critical)			round_robin_next[0]~reg0|d
20	Domain Boundary Entry	REG (required)	#2	round_robin_next[0]~reg0

Both the input and output registers are indicated as Domain Boundary Entry because the Fast Forward Compile often inserts register stages at these boundaries if Hyper-Pipelining were enabled.

5.2.10. Critical Chains with Dual Clock Memories

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

If the limiting reason is Insufficient Registers, and the chain is between dual clock memories, you can add pipeline stages to the functional block. Alternatively, add a bank of registers in the RTL, and then allow the Compiler to balance the registers. Refer to the *Hyper-Pipelining (Add Pipeline Registers)*, *Add Pipeline Stages and Remove Asynchronous Resets*, and *Appendix A: Parameterizable Pipeline Modules* for a pipelining techniques and examples.

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 Information

- Appendix A: Parameterizable Pipeline Modules on page 132
- Hyper-Pipelining (Add Pipeline Registers) on page 29
- Step 2: Add Pipeline Stages and Remove Asynchronous Resets on page 90

5.2.11. Critical Chain Bits and Buses

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

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

5.2.12. Delay Lines

If your design includes a module that delays a bus by some number of clock cycles, the Compiler may implement such structures using the altshift_taps Intel FPGA IP. When this implementation occurs, the critical chain includes the design hierarchy of `altshift_taps:r_rtl_0`, indicating that synthesis replaces the bank of registers with the altshift_taps IP core.

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

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

6. Optimization Example

This section contains a round robin scheduler optimization example.

6.1. Round Robin Scheduler

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

Example 17. Source Code for Round Robin Scheduler

```
module round_robin_modulo (last, requests, next);

parameter CLIENTS       = 7;
parameter LOG2_CLIENTS  = 3;

// 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 the for-loop
    end   // of 'find_next'
end

 endmodule
```

Figure 126. Fast Forward Compile Report for Round Robin Scheduler

Fast Forward Summary for Clock Domain clk_mod

	Step	Fast Forward Optimizations Analyzed	Estimated Fmax	Slack	Relationship
1	Base Performance	None	263 MHz	-2.809	1.000
2	Fast Forward Step #1 (Hyper-Pipelining)	Added up to 1 pipeline stage in 9 Paths	473 MHz	-1.112	1.000
3	Fast Forward Step #2 (Hyper-Pipelining)	Added up to 1 pipeline stage in 4 Paths	651 MHz	-0.536	1.000
4	Fast Forward Limit	Performance Limited by: Short Path/Long Path	--	--	--

The Retiming Summary report identifies insufficient registers limiting Hyper-Retiming on the critical chain. The chain starts from the register that connects to the last input, through the modulus operator implemented using a divider, and continuing to the register that connects to the next output.

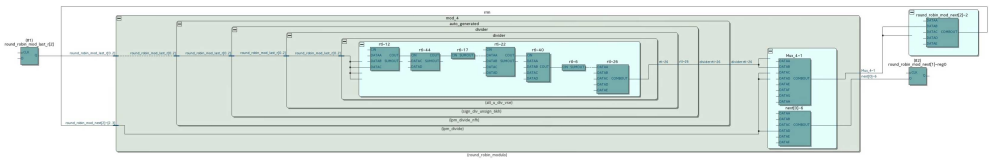
Figure 127. Critical Chain for Base Performance for Round Robin Scheduler

Recommendations for Critical Chain | Critical Chain Details

	Path Info	Register	Register ID	Element
1	Domain Boundary Entry	REG (required)	#1	round_robin_mod_last_r[2]
2	Long Path (Critical)			round_robin_mod_last_r[2]|q
3	Long Path (Critical)			round_robin_mod_last_r[2]~LAB_RE_X232_Y57_N(
4	Long Path (Critical)	bypassed Hyper-Register		round_robin_mod_last_r[2]~C4_X232_Y53_N0_I51
5	Long Path (Critical)	bypassed Hyper-Register		round_robin_mod_last_r[2]~R2_X231_Y56_N0_I22
6	Long Path (Critical)	bypassed Hyper-Register		round_robin_mod_last_r[2]~LOCAL_INTERCONNEC
7	Long Path (Critical)	bypassed Hyper-Register		round_robin_mod_last_r[2]~LAB_RE_X231_Y56_N(
8	Long Path (Critical)			rrm|mod_4|auto_generated|divider|divider|rtl~12|
9	Long Path (Critical)			rrm|mod_4|auto_generated|divider|divider|rtl~12|
10	Long Path (Critical)			rrm|mod_4|auto_generated|divider|divider|rtl~44|
11	Long Path (Critical)			rrm|mod_4|auto_generated|divider|divider|rtl~17|
12	Long Path (Critical)			rrm|mod_4|auto_generated|divider|divider|rtl~17-
13	Long Path (Critical)			rrm|mod_4|auto_generated|divider|d...er|rtl~17~_
14	Long Path (Critical)	bypassed Hyper-Register		rrm|mod_4|auto_generated|divider|...LOCAL_INTEI
15	Long Path (Critical)	bypassed Hyper-Register		rrm|mod_4|auto_generated|divider|divider|rtl~17-
16	Long Path (Critical)			rrm|mod_4|auto_generated|divider|divider|rtl~22|
17	Long Path (Critical)			rrm|mod_4|auto_generated|divider|divider|rtl~40|
18	Long Path (Critical)			rrm|mod_4|auto_generated|divider|divider|rtl~6|c
19	Long Path (Critical)			rrm|mod_4|auto_generated|divider|divider|rtl~6|s
30	Long Path (Critical)			rrm|Mux_4~1|combout
31	Long Path (Critical)			rrm|Mux_4~1~la_lab/laboutb[0]
32	Long Path (Critical)			rrm|Mux_4~1~_LAB_RE_X232_Y57_N0_I114
33	Long Path (Critical)	bypassed Hyper-Register		rrm|Mux_4~1~_LOCAL_INTERCONNECT_X232_Y5
34	Long Path (Critical)	bypassed Hyper-Register		rrm|Mux_4~1~_LAB_RE_X232_Y57_N0_I28
35	Long Path (Critical)			round_robin_mod_next[2]~2|datab
36	Long Path (Critical)			round_robin_mod_next[2]~2|combout
37	Long Path (Critical)			round_robin_mod_next[2]~2~la_lab/laboutt[15]
38	Long Path (Critical)			round_robin_mod_next[2]~2~_LAB_RE_X232_Y57_
39	Long Path (Critical)	bypassed Hyper-Register		round_robin_mod_next[2]~2~_LOCAL_INTERCONN
40	Long Path (Critical)	bypassed Hyper-Register		round_robin_mod_next[2]~2~_LAB_RE_X232_Y57_
41	Long Path (Critical)			rrm|next[0]~6|datad
42	Long Path (Critical)			rrm|next[0]~6|combout
43	Long Path (Critical)			round_robin_mod_next[1]~reg0|d
44	Domain Boundary Entry	REG (required)	#2	round_robin_mod_next[1]~reg0

The 44 elements in the critical chain above correspond to the circuit diagram below that has 10 levels of logic. The modulus operator contributes significantly to the low performance. Seven of the 10 levels of logic are part of the implementation for the modulus operator.

Figure 128. Schematic for Critical Chain



As Figure 126 on page 118 shows, Fast Forward compilation estimates a 140% performance improvement from adding two pipeline stages at the module inputs, for retiming through the logic cloud. At this point, the critical chain is a short path/long path and the chain involves the modulus operator.

Figure 129. Critical Chain Fast Forward Compile for Round Robin Scheduler

Recommendations for Critical Chain | Critical Chain Details

	Path Info	Register	Register ID	Element
1	Short Path	REG (required)	#1	round_robin_mod_last_r[2]
2	Short Path			round_robin_mod_last_r[2]|q
3	Short Path			round_robin_mod_last_r[2]~LAB_RE_X232_Y57_N0_I123
4	Short Path			round_robin_mod_last_r[2]~LOCAL_INTERCONNECT_X232_Y57_N0_I52_dff.d
5	Short Path	REG		round_robin_mod_last_r[2]~LOCAL_INTERCONNECT_X232_Y57_N0_I52_dff
6	Short Path			round_robin_mod_last_r[2]~LOCAL_INTERCONNECT_X232_Y57_N0_I52
7	Short Path			round_robin_mod_last_r[2]~LAB_RE_X232_Y57_N0_I12_dff.d
8	Short Path	REG		round_robin_mod_last_r[2]~LAB_RE_X232_Y57_N0_I12_dff
9	Short Path			round_robin_mod_last_r[2]~LAB_RE_X232_Y57_N0_I12
10	Short Path			rrm|next[0]~8|datab
11	Short Path			rrm|next[0]~8|combout
12	Short Path			round_robin_mod_next[2]~reg0|d
13	Short Path	REG (required)	#2	round_robin_mod_next[2]~reg0
14	----------	----------	----	----------
15	Long Path	REG (required)	#1	round_robin_mod_last_r[2]
16	Long Path			round_robin_mod_last_r[2]|q
17	Long Path			round_robin_mod_last_r[2]~LAB_RE_X232_Y57_N0_I123
18	Long Path	bypassed Hyper-Register		round_robin_mod_last_r[2]~C4_X232_Y53_N0_I51
19	Long Path	bypassed Hyper-Register		round_robin_mod_last_r[2]~R2_X231_Y56_N0_I22
20	Long Path	bypassed Hyper-Register		round_robin_mod_last_r[2]~LOCAL_INTERCONNECT_X231_Y56_N0_I57
21	Long Path	bypassed Hyper-Register		round_robin_mod_last_r[2]~LAB_RE_X231_Y56_N0_I5
22	Long Path			rrm|mod_4|auto_generated|divider|divider|rtl~12|datac
23	Long Path			rrm|mod_4|auto_generated|divider|divider|rtl~12|cout
24	Long Path			rrm|mod_4|auto_generated|divider|divider|rtl~44|cin
25	Long Path			rrm|mod_4|auto_generated|divider|divider|rtl~17|sumout
40	Extend to locatable node	bypassed Hyper-Register		rrm|mod_4|auto_generated|divider|divider|rtl~6~_LAB_RE_X232_Y56_N0_I31
41	Extend to locatable node			rrm|mod_4|auto_generated|divider|divider|rtl~37|datae
42	Extend to locatable node			rrm|mod_4|auto_generated|divider|divider|rtl~37|combout
43	Extend to locatable node			rrm|mod_4|auto_generated|divider|d...er|rtl~37~_LAB_RE_X232_Y56_N0_I108
44	Extend to locatable node	bypassed Hyper-Register		rrm|mod_4|auto_generated|divider|divider|rtl~37~_C4_X231_Y56_N0_I9
45	Extend to locatable node			rrm|mod_4|auto_generated|divider|divider|rtl~37~_R4_X232_Y57_N0_I32_dff.d
46	Extend to locatable node	REG	#3	rrm|mod_4|auto_generated|divider|divider|rtl~37_R4_X232_Y57_N0_I32_dff
47	----------	----------	----	----------
48	Retiming Dependency	REG	#4	round_robin_mod_last_r[1]~LAB_RE_X231_Y56_N0_I44_dff
49	Retiming Dependency			round_robin_mod_last_r[1]~LAB_RE_X231_Y56_N0_I44
50	Retiming Dependency			rrm|mod_4|auto_generated|divider|divider|rtl~33|datab
51	Retiming Dependency			rrm|mod_4|auto_generated|divider|divider|rtl~33|cout
52	Retiming Dependency			rrm|mod_4|auto_generated|divider|divider|rtl~22|cin
53	Retiming Dependency			rrm|mod_4|auto_generated|divider|divider|rtl~40|cout
54	Retiming Dependency			rrm|mod_4|auto_generated|divider|divider|rtl~6|cin
55	Retiming Dependency			rrm|mod_4|auto_generated|divider|divider|rtl~6|sumout
56	Retiming Dependency			rrm|mod_4|auto_generated|divider|divider|rtl~6~_LAB_RE_X231_Y56_N0_I123
158	Long Path	bypassed Hyper-Register		round_robin_mod_next[2]~1~_LAB_RE_X231_Y57_N0_I32
159	Long Path			round_robin_mod_next[2]~10|datae
160	Long Path			round_robin_mod_next[2]~10|combout
161	Long Path			round_robin_mod_next[2]~10~_LAB_RE_X231_Y57_N0_I111
162	Long Path	bypassed Hyper-Register		round_robin_mod_next[2]~10~_LOCAL_INTERCONNECT_X232_Y57_N0_I13
163	Long Path	bypassed Hyper-Register		round_robin_mod_next[2]~10~_LAB_RE_X232_Y57_N0_I30
164	Long Path			round_robin_mod_next[2]~2|datad
165	Long Path			round_robin_mod_next[2]~2|combout
166	Long Path			round_robin_mod_next[2]~2~la_lab/laboutt[14]
167	Long Path			round_robin_mod_next[2]~2~_LAB_RE_X232_Y57_N0_I108
168	Long Path	bypassed Hyper-Register		round_robin_mod_next[2]~2~_LOCAL_INTERCONNECT_X232_Y57_N0_I9
169	Long Path	bypassed Hyper-Register		round_robin_mod_next[2]~2~_LAB_RE_X232_Y57_N0_I8
170	Long Path			rrm|next[0]~8|datae
171	Long Path			rrm|next[0]~8|combout
172	Long Path			round_robin_mod_next[2]~reg0|d
173	Long Path	REG (required)	#2	round_robin_mod_next[2]~reg0

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

Example 18. Source Code for Round Robin Scheduler with Performance Improvement with 2^n Client Inputs

```
module round_robin_modulo (last, requests, next);

parameter LOG2_CLIENTS  = 3;
parameter CLIENTS       = 2**LOG2_CLIENTS;

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


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

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

 endmodule
```

Figure 130. Fast Forward Summary Report for Round Robin Scheduler with Performance Improvement with 2^n Client Inputs

Fast Forward Summary for Clock Domain clk

	Step	Fast Forward Optimizations Analyzed	Estimated Fmax	Slack	Relationship
1	Base Performance	None	589 MHz	-0.699	1.000
2	Fast Forward Step #1 (Hyper-Pipelining)	Added up to 1 pipeline stage in 10 Paths	801 MHz	-0.248	1.000
3	Fast Forward Step #2 (Hyper-Pipelining)	Added up to 1 pipeline stage in 8 Paths	983 MHz	-0.017	1.000
4	Fast Forward Step #3 (Hyper-Pipelining)	Added up to 1 pipeline stage in 10 Paths	1040 MHz	0.038	1.000
5	Fast Forward Limit	Performance Limited by: Short Path/Long Path	--	--	--

Even without any Fast Forward optimizations (the Base Performance step), this round robin implementation runs at double the frequency compared with the version without the performance improvement in Source Code for Round Robin Scheduler on page 118. Although critical chains in both versions contain only two registers, the critical chain for the 2^n version contains only 26 elements, compared to 44 elements in the modulus version.

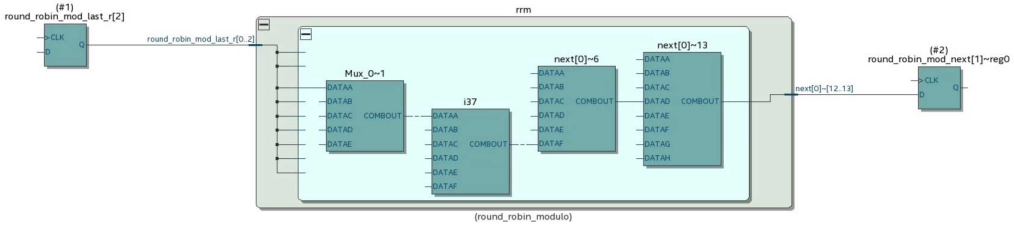
Figure 131. Critical Chain for Round Robin Scheduler with Performance Improvement

Recommendations for Critical Chain | Critical Chain Details

	Path Info	Register	Register ID	Element
1	Domain Boundary Entry	REG (required)	#1	round_robin_mod_last_r[2]
2	Long Path (Critical)			round_robin_mod_last_r[2]|q
3	Long Path (Critical)			round_robin_mod_last_r[2]~LAB_RE_X237
4	Long Path (Critical)	bypassed Hyper-Register		round_robin_mod_last_r[2]~LOCAL_INTER
5	Long Path (Critical)	bypassed Hyper-Register		round_robin_mod_last_r[2]~LAB_RE_X237
6	Long Path (Critical)			rrm|Mux_0~1|dataa
7	Long Path (Critical)			rrm|Mux_0~1|combout
8	Long Path (Critical)			rrm|Mux_0~1~la_lab/laboutb[18]
9	Long Path (Critical)			rrm|Mux_0~1~_LAB_RE_X237_Y105_N0_I'
10	Long Path (Critical)	bypassed Hyper-Register		rrm|Mux_0~1~_LOCAL_INTERCONNECT_X
11	Long Path (Critical)	bypassed Hyper-Register		rrm|Mux_0~1~_LAB_RE_X237_Y105_N0_I'
12	Long Path (Critical)			rrm|i37|dataa
13	Long Path (Critical)			rrm|i37|combout
14	Long Path (Critical)			rrm|i37~la_lab/laboutt[8]
15	Long Path (Critical)			rrm|i37~LAB_RE_X237_Y105_N0_I102
16	Long Path (Critical)	bypassed Hyper-Register		rrm|i37~LOCAL_INTERCONNECT_X237_Y'
17	Long Path (Critical)	bypassed Hyper-Register		rrm|i37~LAB_RE_X237_Y105_N0_I7
18	Long Path (Critical)			rrm|next[0]~6|dataf
19	Long Path (Critical)			rrm|next[0]~6|combout
20	Long Path (Critical)			rrm|next[0]~6~_LAB_RE_X237_Y105_N0_
21	Long Path (Critical)	bypassed Hyper-Register		rrm|next[0]~6~_LOCAL_INTERCONNECT_
22	Long Path (Critical)	bypassed Hyper-Register		rrm|next[0]~6~_LAB_RE_X237_Y105_N0_
23	Long Path (Critical)			rrm|next[0]~13|datad
24	Long Path (Critical)			rrm|next[0]~13|combout
25	Long Path (Critical)			round_robin_mod_next[1]~reg0|d
26	Domain Boundary Entry	REG (required)	#2	round_robin_mod_next[1]~reg0

The 26 elements in the critical chain correspond to the following circuit diagram with only four levels of logic.

Figure 132. Schematic for Critical Chain with Performance Improvement



By adding three register stages at the input, for retiming through the logic cloud, Fast Forward Compile takes the circuit performance to 1 GHz. Similar to the modulus version, the final critical chain after Fast Forward optimization has a limiting reason of short path/long path, as Figure 133 on page 123 shows, but the performance is 1.6 times the performance of the modulus version

Figure 133. Critical Chain for Round Robin Scheduler with Best Performance

Critical Chain at Fast Forward Limit

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

	Path Info	Register	Register ID	Element
1	Short Path	REG	#1	round_robin_mod_last_r[1]~LOCAL_INTERCONNECT_X227_Y66_N0_I46_dff
2	Short Path			round_robin_mod_last_r[1]~LOCAL_INTERCONNECT_X227_Y66_N0_I46
3	Short Path			round_robin_mod_last_r[1]~LAB_RE_X227_Y66_N0_I20_dff.d
4	Short Path	REG		round_robin_mod_last_r[1]~LAB_RE_X227_Y66_N0_I20_dff
5	Short Path			round_robin_mod_last_r[1]~LAB_RE_X227_Y66_N0_I20
6	Short Path			rrm|next[0]~13|datab
7	Short Path			rrm|next[0]~13|combout
8	Short Path			round_robin_mod_next_d[1]|d
9	Short Path	REG	#2	round_robin_mod_next_d[1]
10	----------	----------	----	----------
11	Long Path (Critical)	REG	#1	round_robin_mod_last_r[1]~LOCAL_INTERCONNECT_X227_Y66_N0_I46_dff
12	Long Path (Critical)			round_robin_mod_last_r[1]~LOCAL_INTERCONNECT_X227_Y66_N0_I46
13	Long Path (Critical)	bypassed Hyper-Register		round_robin_mod_last_r[1]~LAB_RE_X227_Y66_N0_I4
14	Long Path (Critical)			rrm|Mux_0~7|datab
15	Long Path (Critical)			rrm|Mux_0~7|combout
16	Long Path (Critical)			rrm|Mux_0~7~_LAB_RE_X227_Y66_N0_I95
17	Long Path (Critical)	bypassed Hyper-Register		rrm|Mux_0~7~_LOCAL_INTERCONNECT_X228_Y66_N0_I1
18	Long Path (Critical)	bypassed Hyper-Register		rrm|Mux_0~7~_LAB_RE_X228_Y66_N0_I34
19	Long Path (Critical)			rrm|i37|datac
20	Long Path (Critical)			rrm|i37|combout
21	Long Path (Critical)			rrm|i37~la_lab/laboutt[17]
22	Long Path (Critical)			rrm|i37~LAB_RE_X228_Y66_N0_I111
23	Long Path (Critical)			rrm|i37~LOCAL_INTERCONNECT_X228_Y66_N0_I42_dff.d
24	Long Path (Critical)	REG	#3	rrm|i37~LOCAL_INTERCONNECT_X228_Y66_N0_I42_dff
25	----------	----------	----	----------
39	Retiming Dependency	REG	#3	rrm|i37~LOCAL_INTERCONNECT_X228_Y66_N0_I42_dff
40	----------	----------	----	----------
41	Extend to locatable node	REG	#4	round_robin_mod_requests_r[0]~LAB_RE_X227_Y66_N0_I68_dff
42	Extend to locatable node			round_robin_mod_requests_r[0]~LAB_RE_X227_Y66_N0_I68
43	Extend to locatable node			rrm|Mux_0~5|datab
44	Extend to locatable node			rrm|Mux_0~5|combout
45	Extend to locatable node			rrm|Mux_0~5~la_lab/laboutb[13]
46	Extend to locatable node			rrm|Mux_0~5~_LAB_RE_X227_Y66_N0_I127
47	Extend to locatable node	bypassed Hyper-Register		rrm|Mux_0~5~_LOCAL_INTERCONNECT_X228_Y66_N0_I24
48	Long Path	bypassed Hyper-Register		rrm|Mux_0~5~_LAB_RE_X228_Y66_N0_I35
49	Long Path			rrm|i37|dataa
50	Long Path			rrm|i37|combout
51	Long Path			rrm|i37~la_lab/laboutt[16]
52	Long Path			rrm|i37~LAB_RE_X228_Y66_N0_I110
53	Long Path			rrm|i37~LOCAL_INTERCONNECT_X227_Y66_N0_I41_dff.d
54	Long Path	REG		rrm|i37~LOCAL_INTERCONNECT_X227_Y66_N0_I41_dff
55	Long Path			rrm|i37~LOCAL_INTERCONNECT_X227_Y66_N0_I41
56	Long Path	bypassed Hyper-Register		rrm|i37~LAB_RE_X227_Y66_N0_I78
57	Long Path			rrm|i61|datad
58	Long Path			rrm|i61|combout
55	Long Path			rrm|i37~LOCAL_INTERCONNECT_X227_Y66_N0_I41
56	Long Path	bypassed Hyper-Register		rrm|i37~LAB_RE_X227_Y66_N0_I78
57	Long Path			rrm|i61|datad
58	Long Path			rrm|i61|combout
59	Long Path			rrm|i61~LAB_RE_X227_Y66_N0_I133
60	Long Path	bypassed Hyper-Register		rrm|i61~LOCAL_INTERCONNECT_X227_Y66_N0_I59
61	Long Path	bypassed Hyper-Register		rrm|i61~LAB_RE_X227_Y66_N0_I23
62	Long Path			rrm|next[0]~13|datah
63	Long Path			rrm|next[0]~13|combout
64	Long Path			round_robin_mod_next_d[1]|d
65	Long Path	REG	#2	round_robin_mod_next_d[1]

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

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

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

7. Intel Hyperflex Architecture Porting Guidelines

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

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

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

7.1. Design Migration and Performance Exploration

You can migrate to an Intel Hyperflex architecture FPGA to evaluate performance improvement. Migrating a design for an Intel Hyperflex architecture FPGA *requires* only minor changes. However, you can apply additional *non-required* changes for further performance improvement. This performance improvement helps you to close timing and add functionality to your design.

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

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

The high level steps in the migration process are:

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

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

  Note: If you migrate a design from a previous version of the Intel Quartus Prime software, some Intel FPGA IP may require replacement if incompatible with the current software version. For example, you cannot upgrade IP based transceivers that a different between different device families.

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

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

7.1.1. Black-boxing Verilog HDL Modules

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

Before:

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

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

After:

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

7.1.2. Black-boxing VHDL Modules

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

Before:

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

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

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( V, W, Sel )
    BEGIN
        IF Sel = '0' THEN
            F <= V ;
        ELSE
            F <= W ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

After:

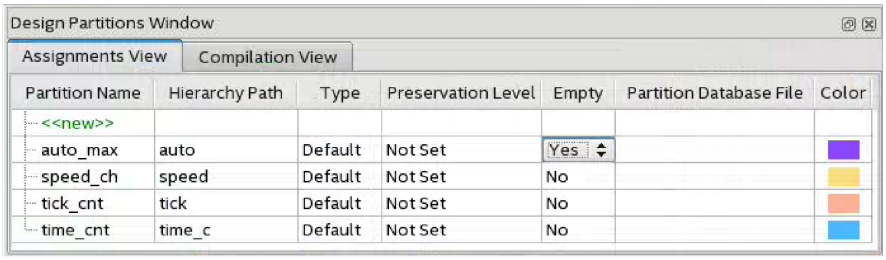
```
-- k-bit 2-to-1 multiplexer
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

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 134. Create New Empty Partition



7.1.3. Clock Management

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

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

7.1.4. Pin Assignments

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

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

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

```
set_instance_assignment –name IO_STANDARD “2.5 V” –to hip_serial_rx_in1
set_instance_assignment –name IO_STANDARD “2.5 V” –to hip_serial_rx_in2
set_instance_assignment –name IO_STANDARD “2.5 V” –to hip_serial_rx_in3
set_location_assignment IOBANK_4A –to hip_serial_rx_in1
set_location_assignment IOBANK_4A –to hip_serial_rx_in2
set_location_assignment IOBANK_4A –to hip_serial_rx_in3
```

Dangling pins

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

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

GPIO pins

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

```
set_instance_assignment VIRTUAL_PIN –to *
```

7.1.5. Transceiver Control Logic

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

7.1.6. Upgrade Outdated IP Cores

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

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

   Note: You cannot upgrade some IP components. If those components are critical (for example, PLL), modify your design and replace them with the latest compatible IP components.

7.2. Top-Level Design Considerations

I/O constraints

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

- `set_input_delay`
- `set_output_delay`

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

Resets

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

Special Blocks

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

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

Register the Block

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

8. Appendices

8.1. Appendix A: Parameterizable Pipeline Modules

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

Example 19. Parameterizable Hyper-Pipelining Verilog HDL Module

```
(* altera_attribute = "-name AUTO_SHIFT_REGISTER_RECOGNITION off" *)
module hyperpipe
#(parameter CYCLES = 1, parameter WIDTH = 1)
(
    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 20. Parameterizable Hyper-Pipelining Verilog HDL Instance

```
hyperpipe # (
        .CYCLES  ( ),
        .WIDTH   ( )
    ) hp (
        .clk      ( ),
        .din      ( ),
        .dout     ( )
    );
```

Example 21. Parameterizable Hyper-Pipelining SystemVerilog Module

```
(* altera_attribute = "-name AUTO_SHIFT_REGISTER_RECOGNITION off" *)
module hyperpipe
    #(parameter int
        CYCLES = 1,
        PACKED_WIDTH = 1,
        UNPACKED_WIDTH = 1
)
(
    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_ff@(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 : hyperpipe
```

Example 22. Parameterizable Hyper-Pipelining SystemVerilog Instance

```
// Quartus Prime SystemVerilog Template
//
// Hyper-Pipelining Module Instantiation

hyperpipe # (
        .CYCLES         ( ),
        .PACKED_WIDTH   ( ),
        .UNPACKED_WIDTH ( )
    ) hp (
        .clk      ( ),
        .din      ( ),
        .dout     ( )
    );
```

Example 23. Parameterizable Hyper-Pipelining VHDL Entity

```
library IEEE;
use IEEE.std_logic_1164.all;
library altera;
use altera.altera_syn_attributes.all;

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

architecture arch of hyperpipe is

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

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

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 24. Parameterizable Hyper-Pipelining VHDL Instance

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

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

8.2. Appendix B: Clock Enables and Resets

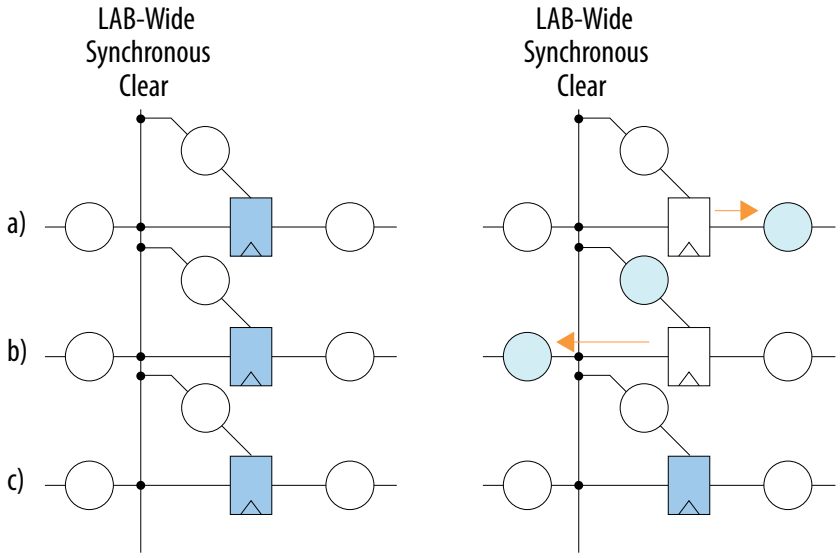
8.2.1. Synchronous Resets and Limitations

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

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

Figure 135. Retiming Example for Synchronous Resets

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

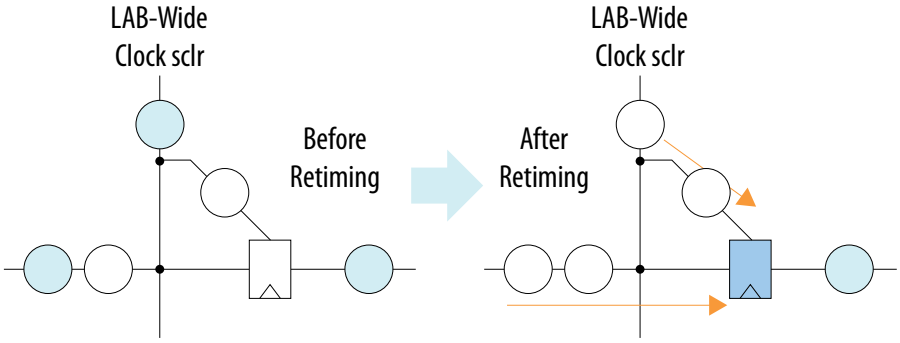


During retiming, the Compiler pushes top register in row (a) into a Hyper-Register. The Compiler implements this by bypassing the ALM register, but still using the SCLR logic that feeds that register. When you use the LAB-wide SCLR signal, an ALM register must exist on the data path, but you need not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

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

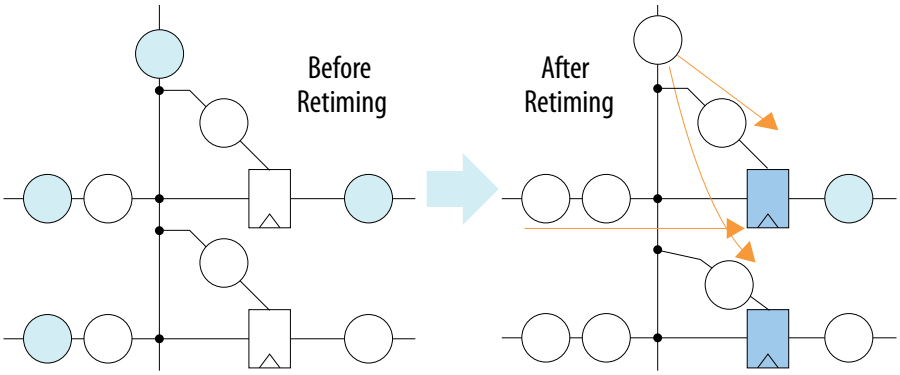
Retiming is more complex when another register pushes forward into the ALM. As shown in the following figure, a register from the synchronous clear port, and a register from the data path, merge together.

Figure 136. Retiming Example – Second Register Pushes out of ALM



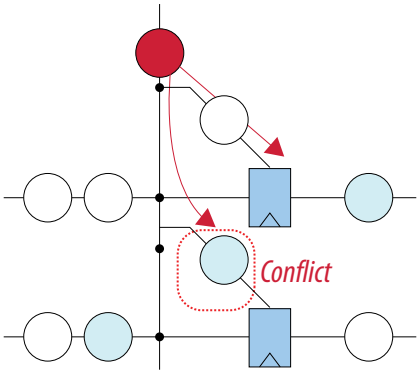
Because other registers share the synchronous clear path, the register splits on the path to other synchronous clear ports.

Figure 137. 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 138. Retiming Example – Conflict at Synchronous Clear



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

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

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

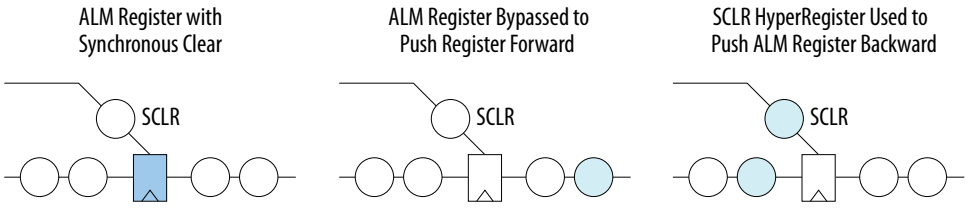
8.2.1.1. Synchronous Resets Summary

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

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

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

Figure 139. Aggressive Retiming



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

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

- Avoid synchronous resets in new code that must run at high speed. This limitation generally applies to data path logic that flushes out while the system is in reset, or logic with values that the system ignores when coming out of reset.
- Control logic often requires a synchronous reset, so there is no avoiding the reset in that situation.
- For existing logic that runs at high speeds, remove the resets wherever possible. If you do not understand the logic behavior at reset, retail the synchronous reset. Only remove the synchronous clear if a timing issue arises.

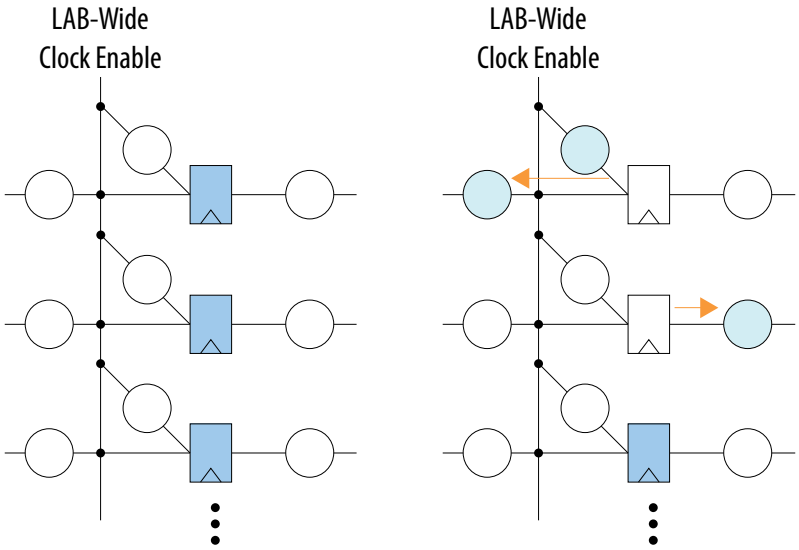
- Pipeline the synchronous clear. This technique does not help when you must pull registers back, but can help when you need to pull registers forward into the data path.
- Duplicate synchronous clear logic for different hierarchies. This technique limits the fan-out of the synchronous clear, so that the Compiler can retime the clear with the local logic. Apply this technique only after you determine that an existing synchronous clear with large fan-out limits retiming. This technique is not difficult on the back-end, because the technique does not change design functionality.
- Duplicate synchronous clear for different clock domain and inverted clocks. This technique can overcome some retiming restrictions due to boundary or multiple period requirement issues.

8.2.2. Retiming with Clock Enables

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

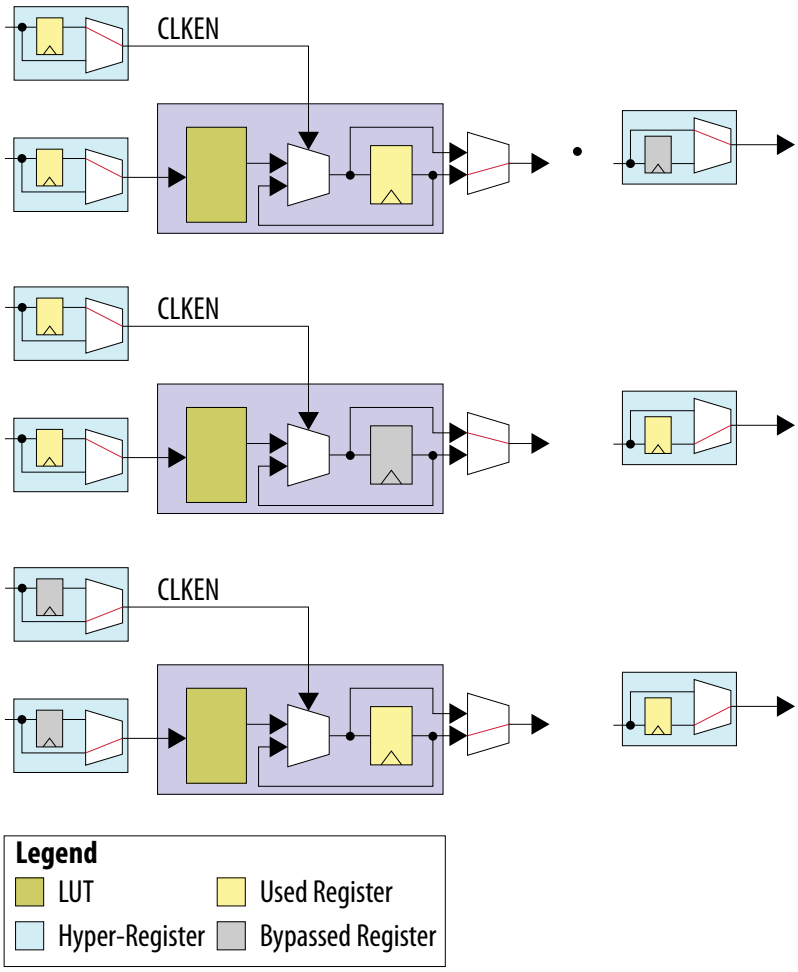
Figure 140. ALM Representing Clock Enables

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



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

Figure 141. Retiming Steps and Structure with an ALM register and Hyper-Registers



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

Related Information

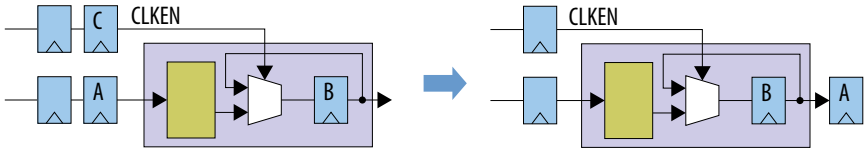
Synchronous Resets and Limitations on page 134

8.2.2.1. Example for Broadcast Control Signals

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

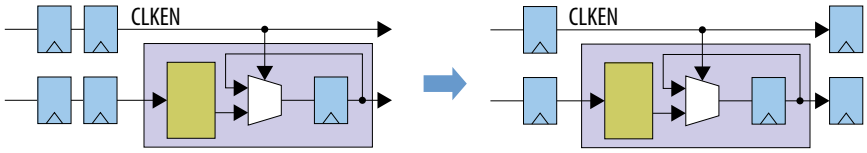
A register must be available on all of the node's inputs to forward retime a register over a node. To retime register A over register B in the following diagram, the Compiler must pull a register from all inputs, including register C on the clock enable input. Additionally, if the Compiler retimes a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

Figure 142. Retiming through a Clock Enable



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

Figure 143. Retiming through a Clock Enable with a Branch Point

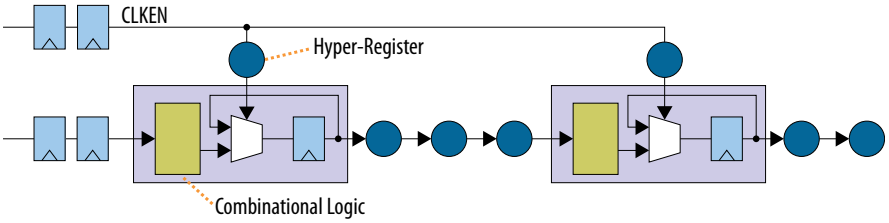


Retiming Example

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 144. Retiming Example Starting Point

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



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

One stage of the input registers retime into a Hyper-Register location between the two registers. Figure 145 on page 141 shows one part of the Hyper-Retiming forward push. One of the registers on the clock enable input retimes over the branch point, with a copy going to a Hyper-Register location at each clock enable input.

Figure 145. Retiming Example Intermediate Point

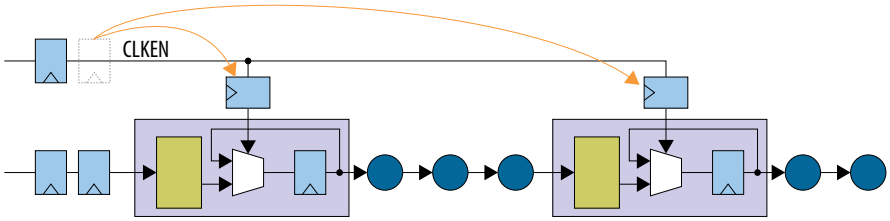


Figure 146 on page 141 shows the positions of the registers in the circuit after Hyper-Retiming completes the forward push. The two registers at the inputs of the left register retime to a Hyper-Register location. This diagram is functionally equivalent to the two previous diagrams. The one Hyper-Register location at the clock enable input of the second register remains occupied. There are no other Hyper-Register locations on the clock enable path to the second register, yet there is still one register at the inputs that the Compiler can retime.

Figure 146. Retiming Example Ending Point

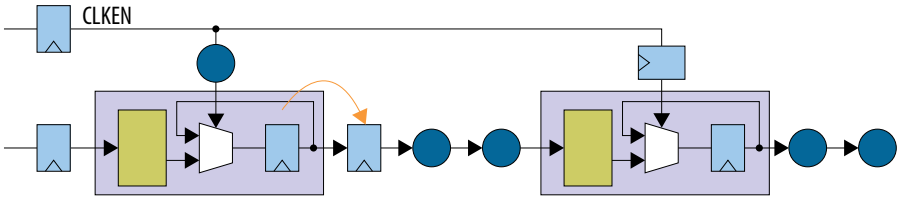
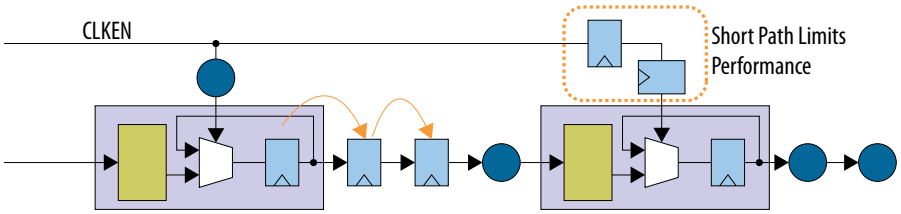


Figure 147 on page 141 shows the register positions Hyper-Retiming uses if a short path/long path critical chain do not limit the path. However, because no Hyper-Registers are available on the right-hand clock enable path, Hyper-Retiming cannot retime the circuit as shown in the diagram.

Figure 147. Retiming Example Limiting condition



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

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

Related Information

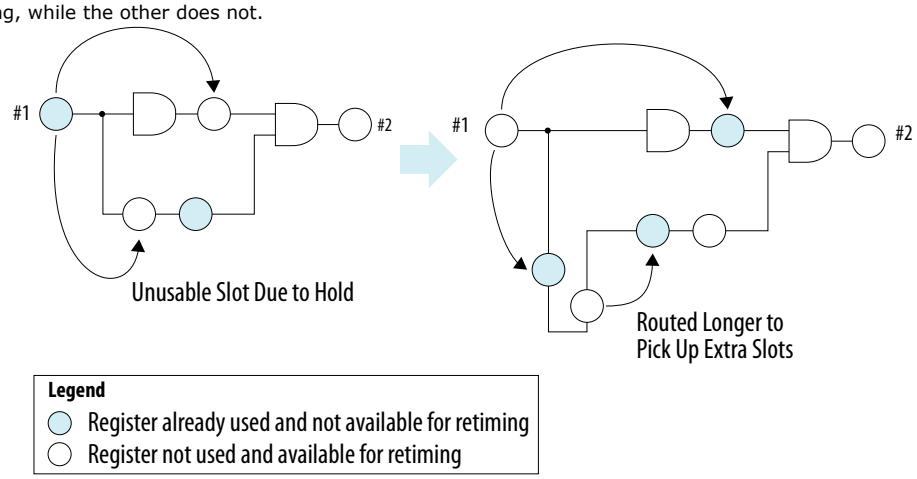
Appendix A: Parameterizable Pipeline Modules on page 132

8.2.3. Resolving Short Paths

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

Figure 148. Short Paths Limiting Retiming

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



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

The following two examples address short paths:

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

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

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

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

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

9. Intel Hyperflex Architecture High-Performance Design Handbook Archive

For the latest and previous versions of this handbook, refer to Intel Hyperflex Architecture High-Performance Design Handbook. If an IP or software version is not listed, the user guide for the previous IP or software version applies.

10. Intel Hyperflex Architecture High-Performance Design Handbook Revision History

Document Version	Intel Quartus Prime Version	Changes
2023.12.08	23.4	• Added Top FAQs navigation to document cover. • Revised *Preserving Registers During Synthesis* topic for clarity. • Updated the product family name to "Intel Agilex 7." • Revised *Median Filter Design Example* topic to remove broken link.
2021.10.11	21.3	• Removed obsolete *Hyper-Retimer Readiness Rules* topic. These rules are now in other rule categories. • Replaced missing figure in *Step 2: Instantiate the Variable Latency Module* topic.
2021.10.04	21.3	• Revised *Step 2: Instantiate the Variable Latency Module* with explanation on partition boundary limits, correct use of constraints, and illustrations.
2021.06.21	20.1	• Added links to *Intel Agilex Device Datasheet* and *Intel Stratix 10 Device Datasheet*.
2020.07.13	20.1	• Changed reference from asynchronous reset to synchronous reset in reference to "Retiming Example – Second Register Pushes out of ALM" figure. • Updated "Initial Power-Up Conditions" section to reflect latest IP and default state of QSF options. • Removed obsolete "Synchronous Start System Clock Gating Examples" topic.
2020.05.01	20.1	• Added "Clock Domain Crossing Constraint Guidelines" topic. • Added Synchronization Register Chain Length assignment details to "Metastability Synchronizers" topic.
2019.12.16	19.4.0	• Referenced programming file generation support for Intel Agilex devices. • Added details, example, and table to "Preserving Registers During Synthesis" topic. • Added note and links about reset release requirement to "Reset Strategies" topic. • Added "Compiling Submodules Independently" topic.
2019.11.15	19.3.0	• Added note about assignment precedence to "Specifying a Latency-Insensitive False Path." • Clarified insertion of vlat module in "Step 2: Instantiate the Variable Latency Module."
2019.11.04	19.3.0	• Retitled document from *Intel Stratix 10 High-Performance Design Handbook* and updated throughout to include Intel Agilex devices. • Added "Design Rule Checking with Design Assistant" topic. • Added "Running Design Assistant During Compilation" topic. • Added "Running Design Assistant in Analysis Mode" topic. • Added "Cross-Probing from Design Assistant" topic.

continued...

Document Version	Intel Quartus Prime Version	Changes
		• Added "Running Design Assistant from Chip Planner" topic. • Added "Running Design Assistant from Timing Analyzer" topic. • Added "Hyper-Retimer Readiness Rules" topic and link to specific rule descriptions.
2019.07.01	19.2.0	• Improved quality of various screenshots. • Updated "Step 1; Compile the Base Design" with results. • Added more detail about the purpose of adding the 5 pipeline stages to "Step 2: Add Pipeline Stages and Remove Asynchronous Resets." • Minor wording changes and updated figure and table references throughout. • Updated link to design example files. • Updated diagrams in "Synchronous Resets and Limitations" topic.
2018.12.30	18.1.0	• Added description of variable latency auto pipelining feature. • Updated new section on "Initial Conditions and Hyper-Registers." • Added new "Synchronous Start System Example" topic. • Added new "Implementing Clock Gating" topic.
2018.10.04	18.0.0	• Minor text change in "Fast Forward Limit." • Minor text change in "Delay Lines."
2018.10.01	18.0.0	• Corrected typo in "Retiming through RAMs and DSPs."
2018.07.12	18.0.0	• Updated all code templates in *Appendix A: Parameterizable Pipeline Modules*. • Added Dual Clock Skid Buffer Example to *Flow Control with Skid Buffers* topic. • Updated various screenshots for improved visibility and accuracy of results.
2018.06.22	18.0.0	Corrected error in *Original Loop Structure* diagram in *Loop Pipelining Demonstration*.
2018.05.22	18.0.0	• Retitled *Removing Asynchronous Clears* to *Removing Asynchronous Resets*. • Converted code images to code examples and corrected code syntax in *Removing Asynchronous Resets*. • Updated signal names in *Removing Asynchronous Resets* images to match code examples. • Corrected syntax error in *Shannon's Decomposition Example*. • Moved information about flow control with skid buffers into new *Flow Control with Skid Buffers* topic. • Enhanced description of *FIFO Flow Control Loop with Two Skid Buffers* diagram. • Clarified description of *Improved FIFO Flow Control Loop with Almost Full instead of Full FIFO* diagram.
2018.05.07	18.0.0	• Removed references to dont_touch synthesis attribute. • Added *Retiming through RAMs and DSPs* topic and diagrams. • Clarified use of `preserve_syn_only` synthesis attribute • Updated Intel Quartus Prime Pro Edition screenshots. • Corrected syntax errors in Round Robin Scheduler examples. • Updated description of Retime stage to include traditional register retiming.
2018.02.05	17.1.1	Updated link to Median Filter design examples files.

Date	Version	Changes
2017.11.06	17.1.0	• Revised *Design Example Walkthrough* steps and results. • Provided link to available design example files for each stage. • Moved step-by-step design compilation instructions to *Design Compilation* chapter, Intel Quartus Prime Pro Edition Handbook. • Added *Ternary Adders* topic and examples. • Added *Loop Pipelining* topic and examples. • Added description of Reset Sequence Requirement report. • Updated for latest Intel branding conventions.
2017.05.08	Quartus Prime Pro v17.1 Stratix 10 ES Editions	• 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.