This application note gives a brief introduction to the new Altera® HyperFlex™ architecture and the Hyper-Pipelining optimization process. It also provides a working example of Hyper-Pipelining in the Example Design Flow section.

HyperFlex Architecture Overview

The HyperFlex core architecture adds Hyper-Registers to every routing segment in the FPGA core and at all functional block inputs (Figure 1). Unlike conventional registers, Hyper-Registers can be bypassed. This allows design tools to maximize core timing performance by automatically selecting the optimal register location after placement and routing.

Figure 1: Layout of Hyper-Registers in HyperFlex Architecture

The image below is a mock-up/representation of a Logic Array Block (LAB).
Hyper-Registers

Figure 2 shows a Hyper-Register where the routing signal can bypass the register and go straight to the multiplexer (MUX). The multiplexer is controlled by one bit of the FPGA configuration memory (CRAM).

Figure 2: Bypassable Hyper-Register

Advantages of Hyper-Registers

- Performance optimizations by the Quartus® II Hyper-Retimer do not require the use of existing adaptive logic module (ALM) logic register resources. Therefore, you have more ALMs available for your design logic.
- Performance tuning does not require additional ALM register resources that add complexity to design placement and routing.
- Reduces routing congestion because unlike ALM registers, additional local routing is not required. Hyper-Registers are available along the current routing path and can be leveraged with almost no overhead as compared to ALM registers.
- FPGA routing delays contribute significantly to overall path delays. The placement of Hyper-Registers in the routing channels of the device is automatically optimized to maximize the design’s operating frequency. This allows the Quartus II Hyper-Retimer software to place the pipeline registers with minimal path delays.

Introduction to Hyper-Pipelining

The Hyper-Pipelining, Hyper-Retiming, and Hyper-Optimization design techniques for Stratix® 10 designs are based on the implementation of familiar design techniques (namely pipelining, retiming, and optimization) using Hyper-Registers. Hyper-Pipelining is the second step of a design flow that targets the HyperFlex architecture. For more information on Hyper-Retiming and Hyper-Optimization, refer to the following application notes:

- AN714: Hyper-Retiming for Stratix 10 Designs
- AN716: Hyper-Optimization for Stratix 10 Designs

Hyper-Pipelining eliminates long routing delays by adding pipeline stages between ALMs using Hyper-Registers. The Hyper-Registers allow fine-grained selection for register locations after placement and routing due to their placement throughout the routing channels of the device. This enables the design to run at a faster clock frequency without consuming additional FPGA logic and routing resources. In
many cases, a design with heavy register usage ends up requiring fewer ALMs to implement the design because previously orphaned registers can now be placed in Hyper-Registers.

Related Information
- AN714: Hyper-Retiming for Stratix 10 Designs
- AN716: Hyper-Optimization for Stratix 10 Designs

Prerequisites for Hyper-Pipelining Implementation

- To leverage the full potential of Hyper-Pipelining, it is important to implement the design practices described in AN714: Hyper-Retiming for Stratix 10 Designs.
- To maximize the benefits of Hyper-Registers and the Hyper-Pipelining feature that the Quartus II software can offer, you must understand the effects of synchronous and asynchronous resets on a design. Eliminate or minimize the use of asynchronous resets to provide the software with the most flexibility. When resets are required, the use of synchronous resets is preferred because they provide additional flexibility to the Quartus II Hyper-Retimer during design optimization. However, synchronous resets might still limit design performance. Refer to AN714: Hyper-Retiming for Stratix 10 Designs for more information.

Related Information
AN714: Hyper-Retiming for Stratix 10 Designs

Conventional Pipelining versus Hyper-Pipelining

In conventional pipelining, designers add register stages to break up long routing paths. By inserting and distributing additional pipeline registers throughout the circuit, you can improve the performance of designs that target traditional FPGA core logic architectures. Hyper-Pipelining leverages the Stratix 10 Hyper-Registers to add pipelining. This avoids the typical drawbacks of traditional pipelining.

Drawbacks of Conventional Pipelining

The lack of register granularity in conventional pipelining reduces the benefit of the optimization. In addition, the implementation process can be time consuming and difficult to verify. It often requires breaking up the HDL logic and inserting register stages at unnatural places in the design. It might also require modifications to legacy intellectual property (IP) blocks or modules, forcing design units to be requalified and reverified.

Figure 3 illustrates the process of conventional pipelining. Design modifications include adding two registers between logic clouds, and modifying the HDL to insert a third register or pipeline stage into the design’s logic cloud (that is, Logic Cloud 2). This effectively creates Logic Cloud 2a and Logic Cloud 2b in the HDL.

Figure 3: Conventional Pipelining User Modifications
Hyper-Pipelining simplifies this process. For the same design illustrated in Figure 3, you add the registers (Pipe 1, Pipe 2, Pipe 3) in aggregate at one location in the design, then use the Hyper-Retimer tool to retime the logic throughout the circuit (Figure 4). This allows you to pick the most convenient place to insert the registers. The Quartus II Hyper-Retimer provides guidance on the number of registers to insert and the path in which to insert them.

**Figure 4: Hyper-Pipelining User Modifications**

After you add the necessary pipeline registers, the Hyper-Retimer tool can move the aggregate registers through the logic circuit until it finds an optimal location in the path. This reduces path delay and maximizes the design’s operating frequency. Figure 5 shows how the Hyper-Retimer tool implements the additional registers.

**Figure 5: Hyper-Pipelining and Hyper-Retimer Implementation**

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

**Advantages of Hyper-Pipelining**

The HyperFlex architecture allows you to concentrate on reducing overall latency. Adding Hyper-Registers reduces the overall latency through a hierarchical block in the design. This greatly reduces the amount of backend work you need to perform. You can focus your efforts on design requirements instead of optimization and placement.

During the design phase, you can simply add generic pipeline blocks at convenient locations within your design. These generic blocks can be configured to insert any number of pipeline stages. A Verilog and VHDL example of this block is available in the Appendix. During the backend compilation flow, you can easily change the number of pipeline stages to the value recommended by the Quartus II Hyper-Retimer to maximize your design’s operating frequency. For example, you can place a generic module before or
after a FIFO buffer used to cross between clock domains. The Quartus II Hyper-Retimer can then use this module to optimize both clock domains.

Because implementation of Hyper-Pipelining does not require additional FPGA resources, you can use this technique aggressively, particularly in data path and feed-forward logic. The Typical Timing Path in Figure 6 shows an example circuit design that can benefit from Hyper-Pipelining. The maximum bandwidth of the circuit, 286 MHz, is limited by the 3.5 ns path segment.

**Figure 6: Typical Timing Path**

Using the Typical Timing Path in Figure 6, Figure 7 demonstrates the advantage of Hyper-Pipelining. In this example, the Hyper-Pipelining implementation reduces the worst-case timing path delay from 3.5 ns to 1.8 ns, which increases the design’s operating frequency to 555 MHz.

**Figure 7: Typical Timing Path with Hyper-Pipelining**

This is almost a 2x improvement. Although an extreme example, it shows that in a design where you can insert a register stage in between existing registers, you can reduce the effective routing delay by approximately one-half, resulting in an approximately 2x increase in operating frequency for that path. It is also worth noting that because the setup and hold requirements on the Hyper-Register are not taken into consideration, the 2x improvement is an approximation. With the Hyper-Registers embedded in the routing channels of the FPGA, the existing routing remains unchanged and a decrease in latency can be achieved in any path where an additional register can be added into a path’s segment. As illustrated in Figure 2, because the Hyper-Registers are bypassable, it is easy for the Quartus II Hyper-Retimer to modify the routing path to include the Hyper-Registers. Notice in Figure 6 and Figure 7, the only change is the addition of the Hyper-Register; there is no change in the routing.

In areas of a design where the functionality of the design does not allow for the insertion of a register or the addition of the register is impractical, the Quartus II Hyper-Retimer has the ability to relocate an existing ALM register into a Hyper-Register in the routing channel, thus preserving the existing routing while taking advantage of the available Hyper-Registers. Using the design path shown in Figure 6, Figure 8 illustrates the relocation of an ALM register into a Hyper-Register.
In this example, the path restricting the 286-MHz operating frequency is the 3.5 ns path between the second and third ALMs. The Hyper-Retimer bypasses the second ALM’s register, and leverages a Hyper-Register in the routing channel to equally partition the path delay from the first to the third ALM. As a result, the paths now have an equal delay of 2.5 ns, which translates to an operating frequency of 400 MHz.

Another advantage of the Hyper-Pipelining infrastructure in Altera’s HyperFlex devices and the Quartus II Hyper-Retimer software is that it allows you to place a set of pipeline registers at the beginning or end of a path. After you insert the pipeline registers, the Quartus II software can then place them optimally throughout the design. The software effectively exploits the Stratix 10 HyperFlex architecture, and helps your design achieve the highest operating frequency possible. It is much easier to visualize an FPGA design from a high-level view, focusing only on the beginning and end points of timing paths. As such, this Hyper-Pipelining functionality simplifies the pipelining process.

Pipelining and Latency

Adding pipeline registers in a design increases the number of clock cycles necessary for a signal value to propagate through the design. Fortunately, the increase in operating frequency and the decrease in delay between registers in the Stratix 10 architecture compensate for the increase in clock cycles, which ultimately decreases the throughput latency of the data path.

In Figure 9, the design path on the left shows the timing path depicted in Figure 6, which is limited by the 3.5-ns delay segment. The time to propagate through the three registers at 286 MHz, or 3.5 ns, is 10.5 ns. After adding the pipeline register, the Quartus II Hyper-Retimer optimizes the design using the available Hyper-Registers. The path now includes one additional register stage. As shown in the design path on the right, it now takes four clock cycles to traverse the path, but the path’s limiting segment is only 1.8 ns, resulting in an operating frequency of 555 MHz. This results in an overall delay of 7.2 ns which translates to a net path latency reduction of 3.3 ns.

In summary, using Hyper-Registers in Hyper-Pipelining decreases the latency of a signal’s propagation through a design path by increasing the device’s overall operating frequency. In addition, Hyper-
Pipelining does not require additional register resources because the new Hyper-Registers are present throughout the routing channels of the Stratix 10 devices.

Hyper-Pipelining and Fast Forward Compile

The new Quartus II Fast Forward Compile flow advises you on the optimal number of registers necessary to maximize your design’s operating frequency. It allows you to explore design performance improvements available in the Stratix 10 HyperFlex architecture without modifying your design. After you review the Fast Forward analysis reports, you can modify your design to achieve the reported gains.

**Figure 10: Quartus II Fast Forward Compile Flow**

The Quartus II Fast Forward Compile Flow (Figure 10) shows the Quartus II compilation flow and where the new Hyper-Retimer and Fast Forward Compile processes occur in the Quartus II tool flow. It also introduces the Hyper-Aware CAD flow. Hyper-Aware CAD is an improvement to the Fitter (FIT) that makes the Fitter aware of the Hyper-Retimer and Retiming optimizations. The Fitter performs optimizations that are adjusted to allow the Hyper-Retiming process to best utilize the Hyper-Registers available in the Stratix 10 architecture. The Quartus II Fast Forward Compile flow explores the possible performance outcomes when a design uses the new Hyper-Registers available in the architecture.

The Fast Forward compile flow anticipates design changes that you are able to implement based on the enabled Quartus II settings. After the Fast Forward Compile flow completes, the Quartus II software generates a set of reports. You can use the information to make design modifications that maximize your design’s performance.

**Related Information**

- **Example Design Flow** on page 8
- More information about Quartus II Fast Forward Compile flow.

**The HYPER_AWARE_OPTIMIZE_REGISTER_CHAINS QSF Setting**

You can use the `HYPER_AWARE_OPTIMIZE_REGISTER_CHAINS` Quartus II Settings File (QSF) setting to further leverage the Hyper-Registers in the Stratix 10 HyperFlex architecture. This QSF setting enables the Quartus II Fitter to take advantage of Hyper-Registers when it encounters register chains. With this setting on, the Fitter marks registers for placement in Hyper-Registers, freeing up ALM register resources. Enabling this QSF setting makes it possible to place and route a design with more registers than are available in the ALMs.
Design Considerations for Multicycle Paths

In conventional designs, you may add multicycle paths exceptions to allow for relaxed timing in situations where the transfer of data is slower than the circuit path’s operating frequency. This allows data to propagate over multiple clock periods. Conventional software implementation tools make use of this exception to meet the required design path’s functionality while reducing the effort necessary to achieve the design paths timing requirements. In some cases, this is put in place to reduce resource usage and/or reduce compilation effort. In the Stratix 10 architecture, these multicycle path exceptions limit the Quartus II Hyper-Retimer from fully optimizing paths with these exceptions. To allow the software the most flexibility, you should whenever possible remove these exceptions and insert the number of registers defined in the setup relationship of the multicycle path exception, thus allowing the Quartus II Hyper-Retimer software the ability to optimize the design path. With the availability of the Stratix 10 Hyper-Registers, the additional registers can be placed in the Hyper-Registers and will therefore be transparent to the designer.

Example Design Flow

The following simple multiplex switch Verilog design (Figure 11) highlights the benefits of Hyper-Pipelining using the Stratix 10 Hyper-Registers. It also demonstrates the following new Quartus II software features:

- Hyper-Retimer
- Fast Forward Compile functionality
- Hyper-Aware CAD

The example design flow is as follows:

1. Compile the design targeting the Stratix V device to generate baseline results.
2. Modify the Quartus II project settings to target the early access Stratix 10 device.
3. Modify the Quartus II project settings to explore design improvements using the Fast Forward Compile functionality and the Hyper-Aware CAD flow. Use the results report to identify design modifications that help achieve the reported performance gains.
4. Implement the suggested design modifications from Step 3, and observe the resulting gains. This step is referred to as realization.
5. Enable the HYPER_AWARE_OPTIMIZE_REGISTER_CHAINS QSF setting and recompile the design.
Figure 11: Cross-Bar Switch

Downloading the Example Designs

Altera delivers five example projects with this document as a .zip archive (an715_hyper_pipelining_examples.zip). This .zip archive includes all you need to explore the advantages of the Stratix 10 Hyper-Registers in the Hyper-Pipelining flow.

1. Unzip the an715_hyper_pipelining_examples.zip file into a location of your choice.
2. Extract the contents of the .zip archive. The an715_hyper_pipelining subdirectory contains the following subdirectories:

Table 1: Hyper-Pipelining Example Design Directories

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>syn_sv</td>
<td>Stratix V xbar_mux project (baseline)</td>
</tr>
<tr>
<td>syn_s10</td>
<td>Stratix 10 xbar_mux device project</td>
</tr>
<tr>
<td>syn_s10_ff</td>
<td>Stratix 10 xbar_mux Fast Forward Compile results</td>
</tr>
<tr>
<td>syn_s10_real</td>
<td>Stratix 10 xbar_mux realized results</td>
</tr>
<tr>
<td>syn_s10_real_regOpt</td>
<td>The syn_s10_real xbar_mux project with HYPER_AWARE_OPTIMIZE_REGISTER_CHAINS setting enabled</td>
</tr>
</tbody>
</table>
**syn_sv: Compiling the Baseline Design**

The following tasks describe the process for performing a standard Quartus II compilation targeting the Stratix V FPGA, using the Quartus II software version 14.0a10 or later.

**Opening the Project**

1. Start the Quartus II software.
2. Click File > Open Project.
3. Navigate to the location of the extracted project directories.
4. Open the syn_sv/xbar_mux.qpf project.

![Figure 12: Open Project](image)

Available with the design archive are the full report files for the base compilation, located in the `<project directory>/an715_hyper_pipelining/syn_sv/alteraReportFiles` directory. Review and compare these results with those you generate in subsequent compilations. The compilation files you generate are located in the `<project directory>/an715_hyper_pipelining/syn_sv/reportFiles` directory.

**Starting the Compilation**

1. In the main Quartus II software window, locate the Tasks pane.
2. Select the Early Timing Estimate with Synthesis flow option, as shown in Figure 13.
3. Hover your mouse over the TimeQuest Timing Analysis task.
4. Select Start from the right-click menu (Figure 14). This runs Analysis & Synthesis, Fitter (Place & Route) and TimeQuest Timing Analysis.

Figure 14: Start TimeQuest Timing Analysis

The compilation takes about 10 minutes. The duration of the process depends on the computation resources you have available.

Reviewing the Results
After the compilation completes, explore the design and compilation results. The TimeQuest Timing Analysis should complete with a maximum operating frequency (f_{MAX}) of 286 MHz.

After you complete the initial review of the baseline design targeting the Stratix V device, proceed to modify the design to target the early access Stratix 10 device.
syn_s10: Compiling Stratix 10 Design

Within the extracted .zip archive, the QSF file in the `<project_directory>/an715_hyper_pipelining/syn_s10` directory includes the necessary QSF modification to target the Stratix 10 device.

**Note:** You must install the Quartus II software Arria® 10 Edition version 14.0 or later before proceeding with the remaining three example projects. Contact your local Altera sales representative for a Stratix 10 Early Access license.

1. Modify the QSF settings for this project as follows:

   ```
   # Compile your design targeting Stratix 10 Early Access device
   set_global_assignment -name FAMILY Arria10
   set_global_assignment -name DEVICE S10_EARLY_ACCESS
   ```

2. Compile this initial design. Results from Analysis & Synthesis, Fitter (Place & Route), and TimeQuest Timing Analysis show the improvements achieved when migrating to the new Stratix 10 device, with the assumption that none of the new HyperFlex functionality is utilized. The TimeQuest Timing Analysis reports an $f_{\text{MAX}}$ of 329 MHz. This is an operating frequency increase of 15% over the Stratix V device project.
Enabling the HyperFlex Compilation Task Flow

Prior to implementing Hyper-Pipelining, there is a one-time setup necessary to enable the HyperFlex Compilation Flow.

1. In the Tasks pane, click Customize.

Figure 15: Customize Task Flow

2. In the Customize Flow dialog box, click New....
3. In the New Flow dialog box, enter HyperFlex as the Custom flow name.
4. In the Edit Flow dialog box (Figure 16), select HyperRetimer and the sub-task View Report. Deselect Assembler, EDA Netlist Writer, and Program Device. Click OK.

For the Based on existing flow option, select Compilation from the pull-down menu and then click OK.
5. Click OK to close the Customize Flow dialog box.
   You now have an active HyperFlex compilation flow available in the Tasks pane.

Starting the Compilation

1. Open the QSF file in the `<project_directory/an715_hyper_pipelining/syn_s10_ff` directory and ensure that the following settings are in the file.

   ```
   # Settings to Enable Fast Forward Compile
   set_global_assignment -name HYPER_RETIMER ON
   set_global_assignment -name HYPER_RETIMER_FAST_FORWARD ON
   # Settings to turn on Hyper Aware CAD
   set_global_assignment -name HYPER_AWARE_OPTIMIZE_TIMING ON
   ```

2. Start the compilation process by running the HyperFlex task flow. This runs all the necessary compilation steps. The process takes about 15 minutes.

Reviewing the Results

1. After compilation completes, click Processing > Compilation report to open the Compilation Report pane. Locate the Hyper-Retimer folder.
2. Within the **Hyper_Retimer** folder, select the **Clock Fmax Summary** report to display the following information:

![Hyper-Retimer Clock Fmax Summary Report GUI](image)

<table>
<thead>
<tr>
<th>Clock Name</th>
<th>Fmax (MHz)</th>
<th>Achieved with Hyper-Retiming (MHz)</th>
<th>Achieved with Hyper-Pipelining (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>329.27</td>
<td>329.27</td>
<td>708.72</td>
</tr>
</tbody>
</table>

Line 1 in the summary report indicates an $f_{\text{MAX}}$ of 329 MHz. This is the same maximum operating frequency achieved in the previous syn_s10 design compilation targeting the Stratix 10 device.

The results in the **Achieved with Hyper-Retiming** and **Fmax** columns are the same for the following reasons:

- The cross-bar mux design does not contain any synchronous or asynchronous reset inputs. Therefore, no reset logic optimization is available.
- The design contains only an input and output register. As a result, there are no opportunities for the software to move an ALM register into a Hyper-Register to balance net delays during the Hyper-Retiming phase.

For detailed information on these Hyper-Retiming optimization strategies, refer to **AN714: Hyper-Retiming for Stratix 10 Designs**.

**Related Information**

**AN714: Hyper-Retiming for Stratix 10 Designs**

**The Hyper-Retimer Report File**

In the report file directory, there is an **xbar_mux.rtm.rpt** report file. It contains the following compilation reports:
Clock Fmax Summary

The `xbar_mux.rtm.rpt` file contains a Clock Fmax Summary report section that is the same as the one in the GUI (Figure 18).

Figure 19: Hyper-Retimer Clock Fmax Summary Section of *.rtm.rpt File

<table>
<thead>
<tr>
<th>; Clock Fmax Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>; Clock Name ; Fmax ; Achieved with Hyper-Retiming ; Achieved with Hyper-Pipelining ;</td>
</tr>
<tr>
<td>; clk ; 329.27 MHz ; 329.27 MHz ; 708.72 MHz ;</td>
</tr>
</tbody>
</table>

Under the Achieved with Hyper-Pipelining column, the operating frequency is 708 MHz, which is a 115% improvement, or an approximately 2.1x increase, in operating frequency!

Hyper-Retimer Summary

The Hyper-Retimer report file also includes a Hyper-Retimer Summary section (Figure 20).

Figure 20: Hyper-Retimer Summary Section of *.rtm.rpt File

<table>
<thead>
<tr>
<th>; Hyper-Retimer Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>; Hyper-Retimer Status ; Successful - Thu Jul 24 16:46:34 2014 ;</td>
</tr>
<tr>
<td>; Quartus II 64-bit Version ; 14.0a10.0 Build 368 07/23/2014 SJ Full Version ;</td>
</tr>
<tr>
<td>; Revision Name ; xbar_mux ;</td>
</tr>
<tr>
<td>; Top-level Entity Name ; xbar_mux ;</td>
</tr>
<tr>
<td>; Family ; Stratix 10 Early Access ;</td>
</tr>
<tr>
<td>; Device ; s10_early_access ;</td>
</tr>
<tr>
<td>; Timing Models ; Preliminary ;</td>
</tr>
<tr>
<td>; Registers staying in ALMs ; 1,408 / 1,408 (100 %) ;</td>
</tr>
<tr>
<td>; Hyper-Registers Used ; 44,037 ;</td>
</tr>
</tbody>
</table>

The Hyper-Retimer Summary indicates that the design uses 44,037 Hyper-Registers and achieves the 708-MHz timing goal reported for clk in the Clock Fmax Summary section. Ensuing information in this section demonstrates how you can easily achieve this type of performance in the Stratix 10 device by exploiting the HyperFlex architecture.

The results for this compilation run are exploratory because the Fast Forward Compile flow is enabled. In addition, the use of Hyper-Registers means that ALM register resources are available for use in the routing channels of the device to maximize the design’s operating frequency. There are 1,408 registers staying in ALMs, which is 100% of the registers as implemented in the current design.

It is important to note that the Fitter uses these 1,408 ALM registers, and all locations remain occupied by registers at the end of the Hyper-Retiming flow.
Fast Forward Clock Domain Summary

The cross-bar mux design contains only one clock domain \( \text{clk} \). The Fast Forward Details for Clock Domain \( \text{clk} \) section of the report file identifies the changes necessary to achieve an approximate 1.5x operating frequency improvement using 44,037 Hyper-Registers.

Figure 21: Fast Forward Clock Domain Summary Section of *.rtm.rpt File

The Fast Forward Step #3 entry in the summary report shows that if you add two pipeline stages to the logic registered by the clock domain \( \text{clk} \), you can achieve an operating frequency of 708 MHz.

Base Performance

You may use the GUI or the \text{xbar_mux.rtm.rpt} file to examine the critical paths for each of the entries listed in the Fast Forward Summary for Clock Domain report. The following data is reported for the Base Performance entry:

Figure 22: Fast Forward Base Performance Critical Chain Section of *.rtm.rpt File

The base performance limitation is a combinatorial Long Path from #1 (\( \text{din_reg[16]}[0] \) register) to #2 (\( \text{dout_reg[43]}[0] \) register). These two entries are categorized as \textit{required}. Recall from the design in

Hyper-Pipelining for Stratix 10 Designs

Altera Corporation

Send Feedback
Figure 11, the din_reg and dout_reg register groups are the source and destination, respectively, of an asynchronous data transfer. As such, they cannot be moved by the Hyper-Retimer, and therefore they are marked as required registers in the Critical Chain report.

Fast Forward Step #1

As mentioned in Hyper-Pipelining and Fast Forward Compile on page 7, the Fast Forward Compile feature provides design recommendations you can implement to maximize your design’s operating frequency. After you identify the limiting path, the next step is to review these design recommendations in the Critical Chain and Recommendations to Achieve Fast Forward Step 1 (599.88 MHz) section of the report file (Figure 23):

Figure 23: Fast Forward Step #1 Critical Chain Section of *.rtm.rpt File

The report shows that there are 1152 sources where if you add one pipeline register, the f_MAX increases from 329 MHz, the base f_MAX, to 599 MHz. Adding one pipeline register shifts the design performance from Fast Forward Step #1 to Fast Forward Step #2. The asterisks (*) next to some entries in the table indicate that the Quartus II Hyper-Retimer predicts that the design path might require one or two more registers than recommended.

The Recommendations for Critical Chain section of the report (Figure 24) provides you with the source and destination locations of the limiting path that the Fast Forward Compile feature identifies in the critical chain. You can use this information to easily find the source HDL that is limiting your design performance.
You can also navigate the design report files using the GUI. The GUI allows you to collapse the various sections for easier navigation. Figure 25 shows an example of the report GUI.

**Note:** You can expand or collapse the report entries by clicking the expand (+) or collapse (-) icon, respectively.

**Figure 25: Hyper-Retimer GUI Critical Chain Report**

Fast Forward Step #2

This report section shows that the same paths that can benefit from a single pipeline stage can also benefit from a second pipeline stage. Adding two pipeline stages increases the design performance to 635 MHz (Figure 26).
Figure 26: Fast Forward Step #2 Critical Chain Section of *.rtm.rpt File

```c
Critical Chain and Recommendations to Achieve Fast Forward Step 2 (635.73 MHz)

; Fast Forward Optimizations Applied in Clock Domain clk
;
; Fast Forward Optimizations
;
; Added up to 2^2 pipeline stage in 1152 Paths for Clock Domain clk
; Added up to 2^2 pipeline stage in 1152 Paths from Top-level Input ports to Clock Domain 'clk'
; Added up to 2^2 pipeline stage in 1152 Paths from Top-level Input ports to Entity xbar_mux
; Added up to 2^2 pipeline stage in 1152 Paths from Top-level Input ports to Instance |xbar_mux|
; Added up to 2^2 pipeline stage at destinations of paths
; from 1152 sources:
; bus din_virt (256 signals)
; din_virt[64][0]
; din_virt[80][0]
; din_virt[96][0]
/~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/
/~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/
; sel_reg[127][1]
+------------------------------------*

* These paths may require one or two additional pipeline stages to achieve the reported fmax.
```
Fast Forward Step #3

If you take the design from Step #2 (635 MHz) to Step #3 (708 MHz), the relevant report file section reveals the following information:

Figure 27: Fast Forward Step #3 Critical Chain Section of *rtm.rpt File

Notice the dout_reg bus at the bottom of the report section in Step #3. This indicates that adding a register to the dout_reg signal leads to an operating frequency of 708 MHz, as indicated in the Fast Forward Details for Clock Domain clk section of the report.
After reviewing the Hyper-Retimer report and determining the necessary design modifications to achieve the 708-MHz operating frequency, you may implement the modifications and recompile your design. You can then verify whether the modified design realizes the gains detailed in the Hyper-Retimer report file. The following code examples show the unoptimized Verilog code (Example 1) and two options (Example 2 and Example 3) for implementing the registers necessary to achieve the targeted 708-MHz operating frequency.

**Example 1: Cross-Bar Mux Original Verilog Implementation**

```verilog
class always @ (posedge clk) begin
  din_reg <= din;
  sel_reg <= sel;
  dout_reg <= mux_out;
end
```

**Example 2: Cross-Bar Mux Manual Register Implementation**

```verilog
class always @ (posedge clk) begin
  din_reg <= din;
  sel_reg <= sel;
  din_reg_PIPE1  <= din_reg;
  din_reg_PIPE2  <= din_reg_PIPE1 ;
  din_reg_PIPE3  <= din_reg_PIPE2 ;
  sel_reg_PIPE1  <= sel_reg;
  sel_reg_PIPE2  <= sel_reg_PIPE1 ;
  sel_reg_PIPE3  <= sel_reg_PIPE2 ;
  dout_reg <= mux_out;
end
```

**Example 3: Cross-Bar Mux Parameterizable Hyper-Pipelining Implementation**

```verilog
hyperpipe #(.
  .CYCLES ( 3 ),
  .WIDTH  ( 256 )
) in_pipe (.
  .clk    ( clk    ),
  .din    ( din     ),
  .dout   ( din_reg )
);
```

**Note:** The design archive includes the `<project_directory>/an715_hyper_pipeling/syn_s10_real` project directory. Navigate to this directory and open the `xbar_mux.qsf` file. Locate the `# Settings to Enable Fast Forward Compile` section. In this section, you will find that the `HYPER RETIMER_FAST_FORWARD` setting is disabled, as shown below:

```plaintext
# Settings to Enable Fast Forward Compile
set_global_assignment -name HYPER RETIMER ON
set_global_assignment -name HYPER RETIMER_FAST FORWARD OFF
```
Starting the Compilation

With the aforementioned change in the project settings, you can start the compilation process by running the HyperFlex task flow. This compilation takes about 15 minutes. When compilation completes, you can review the Hyper-Retimer Compilation Report to determine if the design realizes the \( f_{\text{MAX}} \) indicated by the Fast Forward Compile flow.

Reviewing the Results

Figure 28: Review Results

The report indicates that the design realizes a 712-MHz operating frequency!
syn_s10_real_regOpt: Recompiling the Stratix 10 Design with Register Chain Optimization

The design archive includes the `<project_directory>/an715_hyper_pipelining/syn_s10_real_regOpt` final project directory. Navigate to this directory and open the `xbar_mux.qsf` file. Locate the `# Additional Hyper-Flex Settings` section. In this section, you will find that the `HYPER_AWARE_OPTIMIZE_REGISTER_CHAINS` setting is enabled, as shown below:

```
# Additional Hyper-Flex Settings
set_global_assignment -name HYPER_AWARE_OPTIMIZE_REGISTER_CHAINS ON
```

Starting the Compilation

With the aforementioned change in the project settings, you can start the compilation process as described in the previous section by running the HyperFlex task flow. This will run all the necessary steps so you can review the compilation results based on the new QSF optimization setting. This compilation also takes about 15 minutes. After compilation completes, the Hyper-Retimer Compilation Report becomes available, and you can review the compilation results based on the new QSF optimization setting.

Reviewing the Results

You may view the Hyper-Retimer summary report (Figure 29) in the `xbar_mux.rtm.rpt`, or in the compilation report GUI in the Compilation Report pane by clicking Hyper-Retimer > Summary.

Figure 29: Hyper-Retimer Summary

```
+-----------------------------------------------+
<table>
<thead>
<tr>
<th>Hyper-Retimer Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hyper-Retimer Status; Successful; Mon Aug 18 09:16:51 2014</td>
</tr>
<tr>
<td>Quartus II 64-Bit Version; 14.0a10.0 Build 368 07/23/2014 SJ Full Version</td>
</tr>
<tr>
<td>Revision Name; xbar_mux</td>
</tr>
<tr>
<td>Top-level Entity Name; xbar_mux</td>
</tr>
<tr>
<td>Family; Stratix 10 Early Access</td>
</tr>
<tr>
<td>Device; 810 EARLY ACCESS</td>
</tr>
<tr>
<td>Timing Models; Preliminary</td>
</tr>
<tr>
<td>Registers staying in ALMs; 1,601 / 4,584 (41%)</td>
</tr>
<tr>
<td>Hyper-Registers Used; 45,373</td>
</tr>
<tr>
<td>-- by Fitter; 1,152</td>
</tr>
<tr>
<td>-- by HyperRetimer; 44,221</td>
</tr>
</tbody>
</table>
+-----------------------------------------------+
```

Under **Registers Staying in ALMs**, it indicates that the Fitter uses 4,584 register locations. In the previous `syn_s10_real` compilation, the Fitter uses 5,788 registers. This translates to a reduction of 1,204 registers used.

The `HYPER_AWARE_OPTIMIZE_REGISTER_CHAINS` setting allows the Fitter and Hyper-Retimer to work together to reduce overall device utilization. In our example, the reduction in device resource utilization is the result of better Fitter placement and reservation of design registers for placement by the Hyper-Retimer. This leads to an overall reduction in ALM register utilization.
The Fitter report (Figure 30) shows that the `din_reg_stage2` registers are removed from Fitter placement. These registers are added to Hyper-Register locations within the FPGA fabric. The Fitter reserves these registers for placement by the Hyper-Retimer later in the flow. In Figure 31, notice how the last register removed by the Fitter is on line 1,152. This is the exact number of Hyper-Registers the Fitter uses in the Hyper-Retimer Summary (Figure 29).

Figure 31 shows the section of the Fitter report detailing the registers in the design that are duplicated. A closer look at the full list of duplicated registers reveals that without using the `HYPER_AWARE_OPTIMIZE_REGISTER_CHAINS` setting, additional register duplication is necessary for both the `din_reg` and `sel_reg` register sets to optimize performance.
Appendix

Example HDL Code

Example 4: Parameterizable Hyper-Pipelining Verilog Module

// Hyper-pipelining module HyperPipe Altera Version 2014/08/12
(* altera_attribute = "-name AUTO_SHIFT_REGISTER_RECOGNITION off" *)
module hyperpipe # ( 
    parameter CYCLES
    parameter WIDTH,
)
  ( 
    input clk,
    input [WIDTH-1:0] din,
    output [WIDTH-1:0] dout
  );
generate
  if (CYCLES==0) begin : GEN_COMB_INPUT
    assign dout = din;
  end
  else begin : GEN_REG_INPUT
    integer i;
    reg [WIDTH-1:0] R_data [CYCLES-1:0];
    always @(posedge clk) begin
      R_data[0] <= din;
      for(i=1;i<CYCLES;i=i+1) R_data[i] <= R_data[i-1];
    end
    assign dout = R_data[CYCLES-1];
  end
generate
endmodule

Example 5: Parameterizable Hyper-Pipelining Verilog Instance

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

Example 6: Parameterizable Hyper-Pipelining System Verilog Module

// Hyper-pipelining module HyperPipe Altera Version 2014/08/12
(* altera_attribute = "-name AUTO_SHIFT_REGISTER_RECOGNITION off" *)
module hyperpipe_2d # ( 
    parameter CYCLES
    parameter PACKED_WIDTH,
    parameter UNPACKED_WIDTH,
) ( 
    input clk,
input [PACKED_WIDTH-1:0] din [UNPACKED_WIDTH-1:0],
output [PACKED_WIDTH-1:0] dout [UNPACKED_WIDTH-1:0]
);
generate
if (CYCLES==0) begin : GEN_COMB_INPUT
  assign dout = din;
end
else begin : GEN_REG_INPUT
  integer i;
  reg [PACKED_WIDTH-1:0] R_data
  [CYCLES-1.0][UNPACKED_WIDTH-1:0];
  always @(posedge clk) begin
    R_data[0] <= din;
    for(i=1; i<CYCLES; i=i+1)
      R_data[i] <= R_data[i-1];
  end
  assign dout = R_data[CYCLES-1];
end
gendgenerate

Example 7: Parameterizable Hyper-Pipelining System Verilog Instance

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

Example 8: Parameterizable Hyper-Pipelining VHDL Entity

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

entity hyperpipe is
generic ( 
  CYCLES : integer
  WIDTH : integer;
);
port ( 
  clk : in std_logic;
  din : in std_logic_vector (WIDTH - 1 downto 0);
  dout : out std_logic_vector (WIDTH - 1 downto 0)
);
end hyperpipe;
architecture arch of hyperpipe is
  -- Prevent large hyperpipes from going into memory-based
  altshift_taps,
  -- since that won’t take advantage of Hyper-Registers
attribute altera_attribute of hyperpipe :
entity is "-name AUTO_SHIFT_REGISTER_RECOGNITION off";

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

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

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

Example 9: 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 => ,
                clk => , din => ,
                dout => );

Hyper-Register Naming Convention

Retimed and bypassed registers are named so that they can be identified in retiming reports. The retiming reports show where Hyper-Registers are enabled and where ALM registers are bypassed. If a Hyper-Register is used, the corresponding resource name ends with _dff. If an ALM register is bypassed, the corresponding resource name ends with _.comb.
Hyper-Registers used on routing wires have the following naming convention:

- `<Original Name>_Routing Wire Name>_dff`
- For example: `param_mux\_Mux3-2\_R3_X119\_Y142\_N0\_I25\_dff`

Hyper-Registers at inputs to look-up-tables (LUTs), ALM flipflops, digital signal processing (DSP) blocks have the following naming convention:

- `<input>_dff`
- Register at input example: `mux_tophalf[27].param_mux\_Mux3-2\_datab\_dff`
- Register at RAM input example: `s_rams[27].inst\_ram\_auto_generated\_ram_block1a28\_portaddress[0]\_dff`

Bypassed flipflops in ALMs are appended with `~.comb`:

- `<input>~.comb`
- For example: `param_mux\_data_reg[27][28]~.comb`

Figure 32: Hyper-Register Locations in the Quartus II Software