

AN 851: Incremental Block-Based Compilation Tutorial

for Intel® Arria® 10 FPGA Development Board

Updated for Intel® Quartus® Prime Design Suite: **18.0**



Subscribe



Send Feedback

AN-851 | 2018.06.27

Latest document on the web: [PDF](#) | [HTML](#)



Contents

1. AN 851: Incremental Block-Based Compilation Tutorial for Intel® Arria® 10 FPGA Development Board.....	3
1.1. Tutorial Design Overview.....	4
1.2. Downloading Tutorial Design Files.....	5
1.3. Incremental Block-Based Compilation Tutorial.....	6
1.3.1. Step 1: Compile the Flat Design.....	7
1.3.2. Step 2: Identify Timing-Critical Design Blocks.....	7
1.3.3. Step 3: Create Design Partitions.....	10
1.3.4. Step 4: Analyze Timing of the Partitioned Design.....	11
1.3.5. Step 5: Preserve Timing-Closed Partitions.....	12
1.3.6. Step 6: Optimize Timing-Critical Design Blocks.....	13
1.3.7. Step 7: Verify Preservation and Optimized Results.....	15
1.3.8. (Optional) Step 8: Device Programming.....	17
1.3.9. (Optional) Step 9: Verify Results in Hardware.....	19
1.4. Incremental Block-Based Compilation Tutorial Revision History.....	19

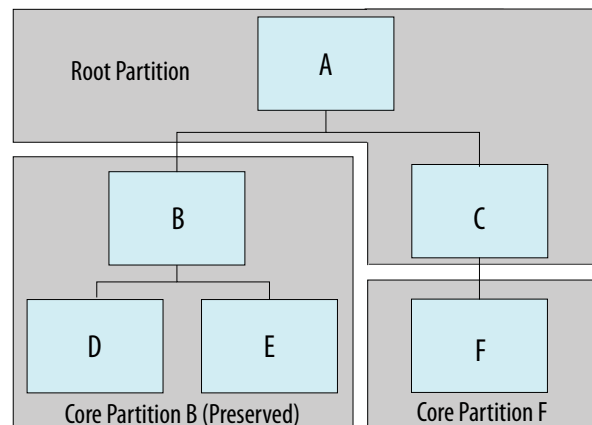


1. AN 851: Incremental Block-Based Compilation Tutorial for Intel® Arria® 10 FPGA Development Board

This tutorial demonstrates using incremental block-based compilation to improve the predictability of results and reduce design iterations in the Intel® Quartus® Prime Pro Edition software.

Incremental block-based compilation enables you to preserve satisfactory compilation results for specific FPGA core logic design blocks (or logic that comprises a hierarchical design instance), and then reuse those results in subsequent compilations. You assign the hierarchical instance as a design partition, which you can then preserve following successful compilation. The preserved design partition must only include core resources (such as LUTs, registers, memory blocks, and DSP blocks), and cannot include any periphery resources.

Figure 1. Partitioned Design with Preserved Core Partition



This tutorial uses an Intel Arria® 10 design example to show you how to improve the predictability of results and reduce design iterations by:

- Preserving a design partition after synthesis or final compilation, and reusing the preserved results in subsequent compilations.
- Targeting only specific design partitions for optimization, while leaving other design partitions unchanged.

Related Information

- [Block-Based Design User Guide](#)
- [AN 839: Design Block Reuse Tutorial for Intel Arria 10 FPGA Development Board](#)
- [AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board](#)

This tutorial includes a prepared design example to demonstrate incremental block-based compilation. You can download the design example to follow along with the tutorial steps in the Intel Quartus Prime Pro Edition software, as [Downloading Tutorial Design Files](#) on page 5 describes.

Figure 2. Incremental Block-Based Compilation Tutorial Design Example

AN 851: Incremental Block-Based Compilation Tutorial for Intel® Arria® 10 FPGA Development Board
4



The duplicate OpenCores design instances have the following characteristics:

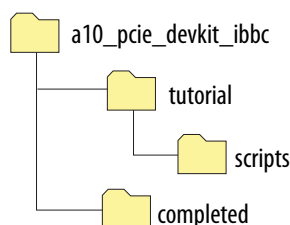
- The design implements each instance in parallel.
- I/O wrapper logic is present to reduce the number of I/O pins that the larger design requires.
- No timing-critical paths exist between the instances and the wrapper logic.

1.2. Downloading Tutorial Design Files

Follow these steps to use the design example files with this tutorial:

1. Download and extract the tutorial design files at:
https://www.altera.com/content/dam/altera-www/global/en_US/others/literature/an/a10_pcie_devkit_ibbc.zip
2. View the extracted tutorial design file directory structure. The `completed` directory contains the final versions of all the files for the tutorial. You can use the files in the `completed` directory for comparison to confirm successful completion of the tutorial steps. The `scripts` folder contains the original files.

Figure 3. Tutorial Directory Structure



The `tutorial` directory includes the following files:

Table 1. Tutorial Directory Files

File Name	Description
<code>top.sv</code>	Top-level file that instantiates the <code>iopll</code> , <code>big_partition1_top</code> , <code>blinking_led_2s</code> , <code>blinking_led_4s</code> , <code>blinking_led_8s</code> , and <code>blinking_led_16s</code> instances. The file also includes logic to drive <code>LED[4:7]</code> as a single, shifting bit.
<code>top.qpf</code>	Intel Quartus Prime project file that stores project name and revisions.
<code>top.qsf</code>	Intel Quartus Prime settings file containing the project assignments and settings.
<code>big_partition1_top.v</code>	Design file that instantiates 20 instances of an OpenCores design.
<code>blinking_led_2s.sv</code>	Design file that includes logic to drive <code>LED[0]</code> every two seconds.
<code>blinking_led_4s.sv</code>	Design file that includes logic to drive <code>LED[1]</code> every four seconds.
<code>blinking_led_8s.sv</code>	Design file that includes logic to drive <code>LED[2]</code> every eight seconds.
<code>blinking_led_16s.sv</code>	Design file that includes logic to drive <code>LED[3]</code> every 16 seconds.
<code>blinking_led.sdc</code>	A Synopsys Design Constraints file that creates the 50 MHz <code>clock</code> .
<code>iopll.ip</code>	The IOPLL Intel FPGA IP instantiated in <code>top</code> . The IP uses 50 MHz as the reference clock frequency, and generates 100 MHz and 550 MHz clocks.
<i>continued...</i>	



File Name	Description
tx_dcfifo.ip	The FIFO Intel FPGA IP instantiated in <code>blinking_led_2s</code> , <code>blinking_led_4s</code> , <code>blinking_led_8s</code> , and <code>blinking_led_16s</code> instances. This is a dual clock FIFO with a write clock of 550 MHz and read clock of 100 MHz.
compile.tcl	A bash script that compiles the tutorial design at the command line.
partitions.tcl	A tcl script that includes the assignments to create the partitions that the tutorial describes. Running the script writes the assignments to the Intel Quartus Prime Settings File (<code>.qsf</code>).
preserve.tcl	A tcl script that includes the assignments to preserve the partitions that the tutorial describes. Running the script writes the assignments to the <code>.qsf</code> .
report_timing.tcl	A tcl script that includes Intel Quartus Prime Timing Analyzer commands that generate summary of paths reports with least positive or worst slack in each partition, along with commands to report timing for two specific nodes in the three partitions that meet timing requirements.

- To restore all of the tutorial files to their original state, run `scripts/restore.tcl` from the `a10_pcie_devkit_ibbc/tutorial` directory.
- To compile the tutorial design from command line, run `compile.tcl` from the `a10_pcie_devkit_ibbc/tutorial` directory.

1.3. Incremental Block-Based Compilation Tutorial

The steps in this tutorial demonstrate how to improve the predictability of results and reduce design iterations by preserving the successful compilation results of design partitions, and optimizing specific partitions.

Process Description

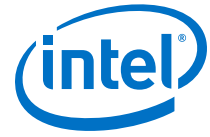
You determine which design blocks might be suitable for preservation and optimization by running a flat compilation and timing analysis to identify the most timing-critical blocks. You then preserve the partitions for blocks that meet timing, so that the Compiler can reuse the successful results for those partitions in subsequent compilations. When you preserve a partition at the final snapshot, the Compiler preserves the final device resource utilization, placement, routing, and hold time fix-up.

After optimizing the timing-critical design blocks, you can preserve those partitions and focus optimization on other parts of the design.

Tutorial Steps

This tutorial includes the following steps:

- [Step 1: Compile the Flat Design](#) on page 7
- [Step 2: Identify Timing-Critical Design Blocks](#) on page 7
- [Step 3: Create Design Partitions](#) on page 10
- [Step 4: Analyze Timing of the Partitioned Design](#) on page 11
- [Step 5: Preserve Timing-Closed Partitions](#) on page 12
- [Step 6: Optimize Timing-Critical Design Blocks](#) on page 13



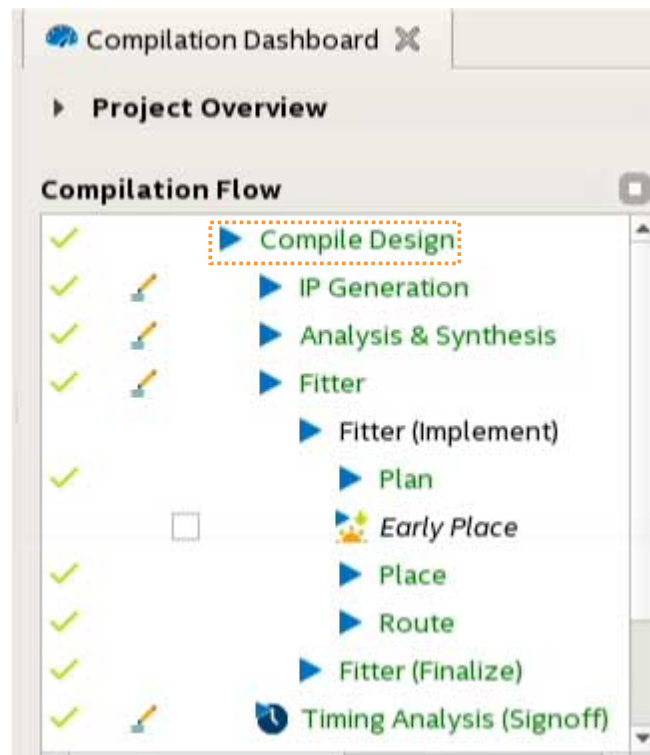
- [Step 7: Verify Preservation and Optimized Results](#) on page 15
- (Optional) [Step 8: Device Programming](#) on page 17
- (Optional) [Step 9: Verify Results in Hardware](#) on page 19

1.3.1. Step 1: Compile the Flat Design

Follow these steps to compile the flat (non-partitioned) design:

1. In the Intel Quartus Prime Pro Edition software, click **File** ► **Open Project** and open the /tutorial/top.qpf project file.
2. To compile the flat design, click **Compile Design** on the Compilation Dashboard. A check mark appears as each stage completes. The compilation may require 30 minutes or more, depending on your system.

Figure 4. Compilation Dashboard

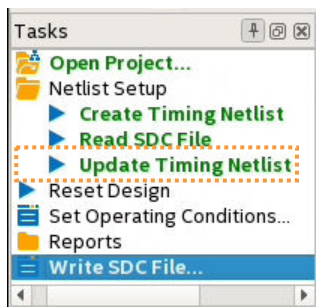


1.3.2. Step 2: Identify Timing-Critical Design Blocks

Follow these steps to identify the timing-critical design blocks in the Intel Quartus Prime Timing Analyzer:

1. To open the Timing Analyzer, click **Tools** ► **Timing Analyzer**.
2. In the Timing Analyzer, on the **Tasks** pane, double-click **Update Timing Netlist** to load the final timing netlist generated during the compilation.

Figure 5. Timing Analyzer Tasks Pane



- To run the `report_timing.tcl` script to identify any failing paths in the timing-critical design blocks, type the following command in the Console window. If not already visible, click **View** > **Console** in the Timing Analyzer to display the Console. The script runs commands to identify any failing paths.

```
source report_timing.tcl
```

The tcl script runs the `report_timing` command, capturing timing for the top 100 paths with the worst slack. The script is also preconfigured to capture timing between specific nodes for some of the design blocks. You analyze timing for these nodes later in this tutorial.

Figure 6. Timing Analyzer Report Folders

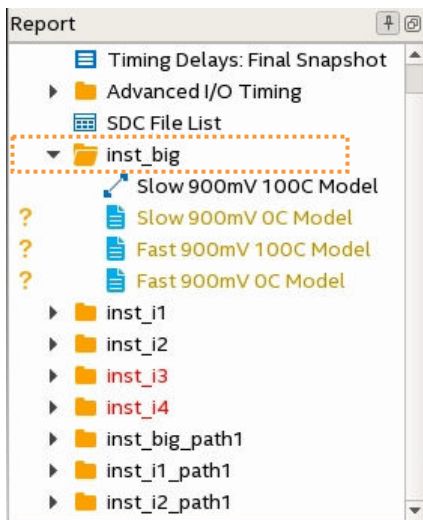


Table 2. Timing Analysis Reports that `report_timing.tcl` Generates

Timing Analysis Folder	Generated For	Timing Reports Show
inst_big	u_big_partition1_top	Analysis of top 100 paths with worst slack
inst_i1	u_blinking_led_i1	
inst_i2	u_blinking_led_i2	
inst_i3	u_blinking_led_i3	
continued...		



Timing Analysis Folder	Generated For	Timing Reports Show
inst_i4	u_blinking_led_i4	Analysis of timing between specific nodes for some partitions
inst_big_path1	u_big_partition1_top	
inst_i1_path1	u_blinking_led_i1	
inst_i2_path1	u_blinking_led_i2	

- In the **inst_big** folder, right-click the **Slow 900 mV 100C Model** report, and then click **Generate in All Corners**. Repeat this step for the **inst_i1**, **inst_i2**, **inst_i3**, and **inst_i4** folders.
- View the **Multi Corner Summary** report that generates under each folder in the **Report** pane. Reports in red text in the **inst_i3** and **inst_i4** folders indicate timing-critical design blocks with failing paths.
- Open the **Multi Corner Summary** report in the **inst_i3** folder. Check the values in the **From Node** and **To Node** fields. Analysis indicates that the failing paths in **u_blinking_led_i3** are in the 64-bit counter. This counter counts the number of cycles equivalent to 8s, where each cycle is of 1.818 ns.

Figure 7. Multi Corner Summary for u_blinking_led_i3

Multi Corner Summary (4/4 corners)				
Corner Information		Summary of Paths		
	Corner	Slack	From Node	To Node
1	Slow 900mV 100C Model	-0.198	u_blinking_led_i3 count_in[61]	u_blinking_led_i3 count_in[61]
2	Slow 900mV 100C Model	-0.186	u_blinking_led_i3 count_in[59]	u_blinking_led_i3 count_in[61]
3	Slow 900mV 100C Model	-0.186	u_blinking_led_i3 count_in[61]	u_blinking_led_i3 count_in[59]
4	Slow 900mV 100C Model	-0.185	u_blinking_led_i3 count_in[61]	u_blinking_led_i3 count_in[60]

Note: Due to processor, memory, or OS variations, the slack values in this tutorial are only for reference and may vary from the actual values you observe.

- Open the **Multi Corner Summary** report in the **inst_i4** folder. Check the values in the **From Node** and **To Node** fields. Analysis indicates that the failing paths in **u_blinking_led_i4** are in the 64-bit counter. This counter counts the number of cycles equivalent to 16s, where each cycle is of 1.818 ns.

Figure 8. Multi Corner Summary for u_blinking_led_i4

Multi Corner Summary (4/4 corners)				
Corner Information		Summary of Paths		
	Corner	Slack	From Node	To Node
1	Slow 900mV 100C Model	-0.058	u_blinking_led_i4 count_in[17]	u_blinking_led_i4 count_in[17]
2	Slow 900mV 100C Model	-0.056	u_blinking_led_i4 count_in[17]	u_blinking_led_i4 count_in[27]
3	Slow 900mV 100C Model	-0.055	u_blinking_led_i4 count_in[27]	u_blinking_led_i4 count_in[17]
4	Slow 900mV 100C Model	-0.053	u_blinking_led_i4 count_in[27]	u_blinking_led_i4 count_in[27]

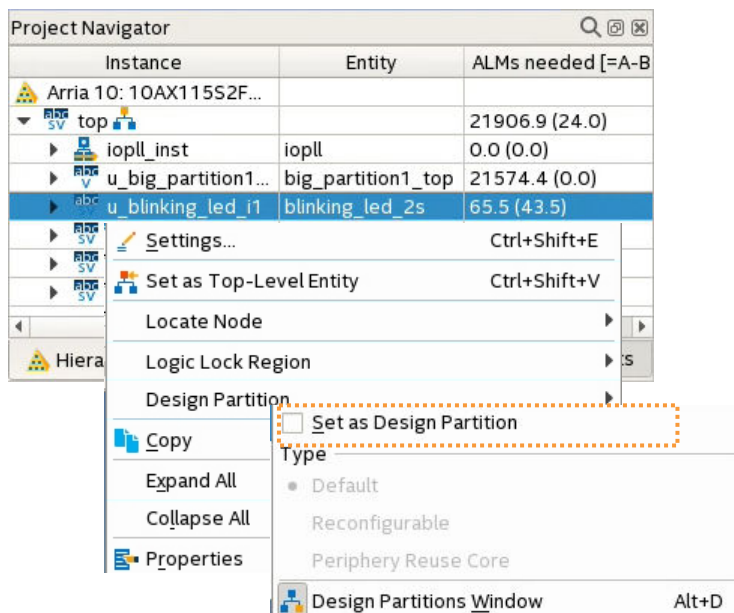
The timing analysis identifies **u_blinking_led_i3** and **u_blinking_led_i4** as timing-critical design blocks for optimization.

1.3.3. Step 3: Create Design Partitions

After identifying the timing-critical design blocks, you can partition and recompile the design to preserve the results for the partitions that meet timing. Follow these steps to partition the design:

1. In the Project Navigator, right-click the `u_blinking_led_i1` instance in the **Hierarchy** tab, and then click **Design Partition** ➤ **Set as Design Partition**. A design partition icon appears next to each instance you assign.

Figure 9. Create Design Partitions



2. Repeat step 1 to create partitions for the `u_big_partition1_top`, `u_blinking_led_i2`, `u_blinking_led_i3`, and `u_blinking_led_i4` instances.
3. If the Design Partitions Window is not already open, click **Assignments** ➤ **Design Partitions Window**. The Design Partitions Window lists the partitions you define, along with the root partition (|) the Compiler automatically creates for each project.

Figure 10. Design Partitions Window

Design Partitions Window							
Assignments View				Compilation View			
Partition Name	Hierarchy Path	Type	Preservation Level	Empty	Partition Database File	Entity Re-binding	Color
<<new>>							
root_partition							
blinking_led_2s	u_blinking_led_i1	Default	Not Set	No			
big_partition1_top	u_big_partition1_top	Default	Not Set	No			
blinking_led_4s	u_blinking_led_i2	Default	Not Set	No			
blinking_led_8s	u_blinking_led_i3	Default	Not Set	No			
blinking_led_16s	u_blinking_led_i4	Default	Not Set	No			

4. To compile the partitioned design, click **Compile Design** on the Compilation Dashboard.



1.3.4. Step 4: Analyze Timing of the Partitioned Design

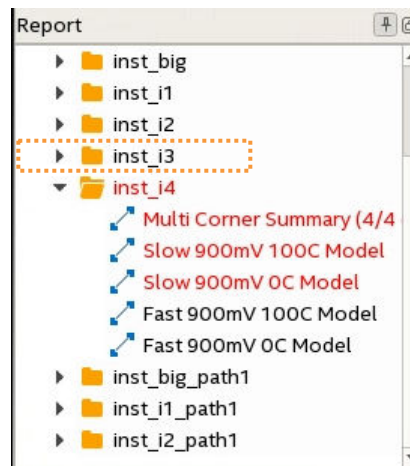
Follow these steps to analyze the timing of the partitioned design:

1. Click **Tools** ► **Timing Analyzer**, and then double-click **Update Timing Netlist**.
2. Run the `report_timing.tcl` script to regenerate the timing analysis reports for failing paths:

```
source report_timing.tcl
```

The timing analysis reports in the **inst_i3** folder are no longer red, indicating that `u_blinking_led_i3` now meets timing requirements in the partitioned design. Later in this tutorial you optimize this design block to ensure that it meets timing requirements in the flat design.

Figure 11. u_blinking_led_i3 Meets Timing In Partitioned Design



3. View the **Multi Corner Summary** reports in the **inst_big1_path1**, **inst_i1_path1**, and **inst_i2_path1** timing analysis folders. The `report_timing.tcl` script includes commands to generate these reports for pre-selected nodes. Note the slack and placement results for the paths in 3 partitions, as the following figure shows. Later in the tutorial you compare these results with those after compilation of the final snapshot.



Figure 12. Multi Corner Summary for u_big_partition1_top

Multi Corner Summary (4/4 corners)

Corner Information		Summary of Paths				
	Corner	Slack	From Node	To Node		
1	Slow 900mV 100C Model	7.776	u_big_p...[14][1]	u_big_part...it add1		
2	Slow 900mV 0C Model	7.878	u_big_p...[14][1]	u_big_part...it add1		
Path #1: Setup slack is 7.776						
Path Summary		Statistics	Data Path	Waveform	Extra Fitter Info	
Data Arrival Path						
	Total	Incr	RF	Type	Fanout	Location
1	0.000	0.000				
2	4.350	4.350				
3	6.634	2.284				
1	4.571	0.221	RR	uTco	1	FF_X139_Y112_N2
2	4.711	0.140	RR	CELL	2	FF_X139_Y112_N2
3	4.852	0.141	RR	IC	4	MLABCELL_X139_Y112_NO
4	5.256	0.404	RR	CELL	1	MLABCELL_X139_Y112_NO
5	5.262	0.006	RR	CELL	1	MLABCELL_X139_Y112_NO
6	5.423	0.161	RR	IC	3	LABCELL_X140_Y112_N3
7	5.847	0.424	RR	CELL	1	LABCELL_X140_Y112_N3
8	5.852	0.005	RR	CELL	1	LABCELL_X140_Y112_N3

1.3.5. Step 5: Preserve Timing-Closed Partitions

You can preserve the final snapshot of the partitions that meet timing requirements, to retain the same implementation in subsequent compilations. In this tutorial you do not preserve the final snapshot for the **blinking_led_8s** partition, even though this partition meets timing in the partitioned design. You do not preserve the final snapshot for the **blinking_led_8s** partition to ensure that whatever placement the Fitter chooses, the design block meets timing whether partitioned or not.

The Compiler preserves the final device utilization, placement, routing, and hold time fix-up for the partitions that you preserve. The preserved partition becomes the source for each subsequent compilation.

Follow these steps to preserve the timing-closed partitions:

1. Click **Assignments > Design Partitions Window**.
2. Select **final** as the **Preservation Level** for the **blinking_led_2s**, **big_partition1_top**, and **blinking_led_4s** partitions.



Figure 13. Setting Partition Preservation Levels

Design Partitions Window							
Assignments View		Compilation View					
Partition Name	Hierarchy Path	Type	Preservation Level	Empty	Partition Database File	Entity Re-binding	Color
<<new>>							
root_partition							
blinking_led_2s	u_blinking_led_i1	Default	final	No			
big_partition1_top	u_big_partition1_top	Default	final	No			
blinking_led_4s	u_blinking_led_i2	Default	final	No			
blinking_led_8s	u_blinking_led_i3	Default	Not Set	No			
blinking_led_16s	u_blinking_led_i4	Default	Not Set	No			

1.3.6. Step 6: Optimize Timing-Critical Design Blocks

Follow these steps to optimize the 64-bit counters in `blinking_led_8s.sv` and `blinking_led_16s.sv` to improve timing. These changes implement 32-bit addition, rather than 64-bit addition in the counters.

1. Open `blinking_led_8s.sv` in a text editor and uncomment the following lines:

```
//reg [31:0] count_msb;
//reg [31:0] count_lsb;
//reg [1:0] state=2'b00;
//always_ff @(posedge fast_clock) begin
//    fifo_wreq <= 1'b0;
//    case (state)
//        2'b00: begin
//            count_lsb <= count_lsb + 1;
//            if (count_lsb[31:0]==32'hFFFFFFF) begin
//                state <= 2'b01;
//            end
//        end
//        2'b01: begin
//            count_lsb <= count_lsb + 1;
//            if (count_lsb[31:0]==32'h064962EC) begin
//                count_lsb <= 1;
//                value_in <= !value_in;
//                fifo_wreq <= 1'b1;
//                state <= 2'b00;
//            end
//        end
//        default: begin
//            count_msb <= 0;
//            count_lsb <= 0;
//            state <= 2'b00;
//        end
//    endcase
//end
```

2. In `blinking_led_8s.sv`, comment the following lines and save the changes:

```
reg [63:0] count_in;
always_ff @(posedge fast_clock) begin
    count_in <= count_in + 1;
    fifo_wreq <= 1'b0;
    if (count_in==64'd4400440044) begin
        count_in <= 0;
        value_in <= !value_in;
    end
end
```



```
        fifo_wreq <= 1'b1;
    end
end
```

3. Open `blinking_led_16s.sv` and uncomment the following lines:

```
//reg [31:0] count_msb;
//reg [31:0] count_lsb;
//reg [1:0] state=2'b00;
//
//always_ff @(posedge fast_clock) begin
//    fifo_wreq <= 1'b0;
//
//    case (state)
//    2'b00: begin
//        count_lsb <= count_lsb + 1;
//        if (count_lsb[31:0]==32'hFFFFFFF) begin
//            state <= 2'b01;
//        end
//    end
//
//    2'b01: begin
//        count_lsb <= count_lsb + 1;
//        if (count_lsb[31:0]==32'hFFFFFFF) begin
//            state <= 2'b10;
//        end
//    end
//
//    2'b10: begin
//        count_lsb <= count_lsb + 1;
//        if (count_lsb[31:0]==32'h0C92C5D8) begin
//            count_lsb <= 1;
//            value_in <= !value_in;
//            fifo_wreq <= 1'b1;
//            state <= 2'b00;
//        end
//    end
//
//    default: begin
//        count_msb <= 0;
//        count_lsb <= 0;
//        state <= 2'b00;
//    end
//
//    endcase
//end
```

4. In `blinking_led_16s.sv`, comment the following lines and save the changes:

```
reg [63:0] count_in;
always_ff @(posedge fast_clock) begin
    count_in <= count_in + 1;
    fifo_wreq <= 1'b0;
    if (count_in==64'd8800880088) begin
        count_in <= 0;
        value_in <= !value_in;
        fifo_wreq <= 1'b1;
    end
end
```

5. To compile the design with these changes, click **Compile Design** on the Compilation Dashboard.

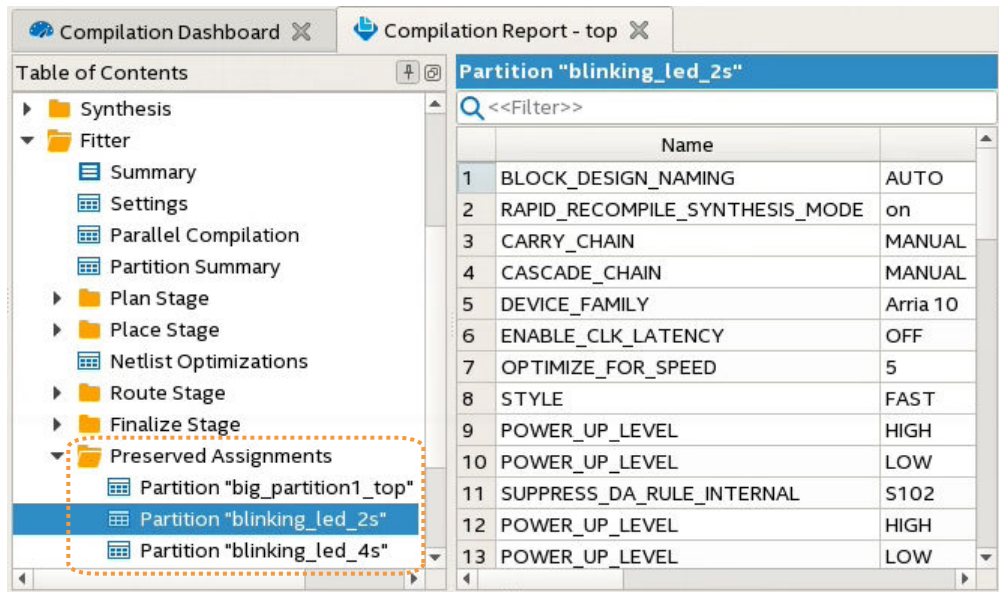


1.3.7. Step 7: Verify Preservation and Optimized Results

After compilation is complete, follow these steps to verify that the Compiler uses the preserved partitions, and that the optimized design block now meets timing requirements:

1. In the Compilation Report (**Processing > Compilation Report**), under the **Fitter** folder, expand the **Preserved Assignments** folder. The reports indicate use of the preserved partitions.

Figure 14. Preserved Partitions Report

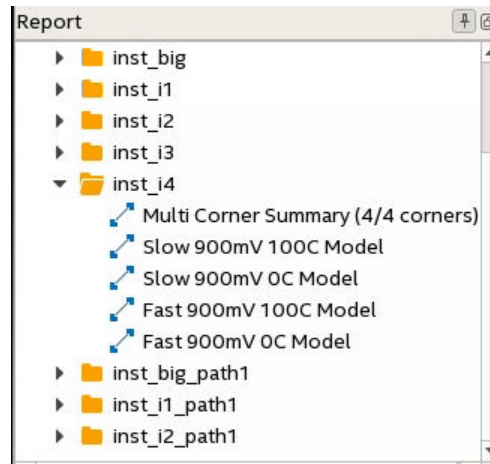


2. Click **Tools > Timing Analyzer**, and then double-click **Update Timing Netlist**.
3. Run the `report_timing.tcl` script to regenerate the timing analysis reports:

```
source report_timing.tcl
```

Timing analysis data in the **inst_i4** report folder now indicates that the **blinking_led_i4** partition meets timing requirements. Reports in the **inst_i3** folder also indicate slack improvement for the **blinking_led_i3** partition.

Figure 15. Optimized u_blinking_led_i4 Meets Timing



4. In the Timing Analyzer reports , right-click the **Slow 900 mV 100C Model** report in each folder, and then click **Generate in All Corners**.
5. Open the **Multi Corner Summary** report to check the slack and placement results for the **big_partition1_top** partition. The slack value is similar to performance at the time of preservation. The placement results are the same as at the time of preservation. You can compare these preserved results with Figure 12 on page 12.

Figure 16. Preserved Slack and Placement

Multi Corner Summary (4/4 corners)

Corner Information		Summary of Paths		
	Corner	Slack	From Node	To Node
1	Slow 900mV 100C Model	7.777	u_big_p...[14][1]	u_big_part...it add1
2	Slow 900mV OC Model	7.872	u_big_p...[14][1]	u_big_part...it add1

Path #1: Setup slack is 7.777

Path Summary		Statistics	Data Path	Waveform	Extra Fitter Info	
Data Arrival Path						
	Total	Incr	RF	Type	Fanout	Location
1	0.000	0.000				
2	4.339	4.339				
3	6.624	2.285				
1	4.561	0.222	RR	uTco	1	FF_X139_Y112_N2
2	4.701	0.140	RR	CELL	2	FF_X139_Y112_N2
3	4.842	0.141	RR	IC	4	MLABCELL_X139_Y112_N0
4	5.246	0.404	RR	CELL	1	MLABCELL_X139_Y112_N0
5	5.252	0.006	RR	CELL	1	MLABCELL_X139_Y112_N0
6	5.413	0.161	RR	IC	3	LABCELL_X140_Y112_N3
7	5.837	0.424	RR	CELL	1	LABCELL_X140_Y112_N3
8	5.842	0.005	RR	CELL	1	LABCELL_X140_Y112_N3

6. Repeat step 4 to verify the slack and placement results for the **blinking_led_i1** and **blinking_led_i2** partitions.



1.3.8. (Optional) Step 8: Device Programming

You can optionally configure the FPGA on the Intel Arria 10 GX Development Kit to verify the results in hardware. You can adapt the following steps if you are using a different device or development kit. Configuring the FPGA involves opening the Intel Quartus Prime Pro Edition Programmer, connecting to the Development Kit board, and loading the configuration SRAM Object File (.sof) into the SRAM of the FPGA.

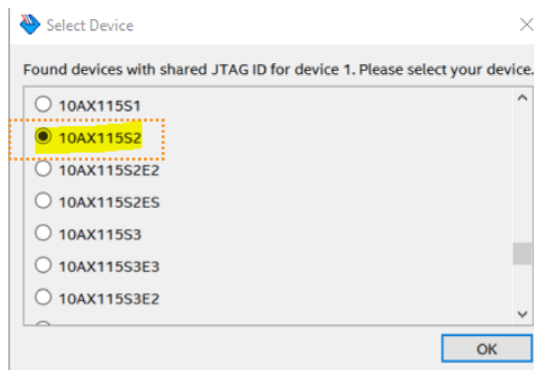
Note: A .sof file configures the SRAM of an Intel FPGA. A Programmer Object File (.pof) programs a flash memory device with an FPGA configuration image for subsequent loading to an FPGA.

Follow these steps to configure the FPGA on the Intel Arria 10 GX Development Kit:

1. To open the Intel Quartus Prime Programmer, click **Tools ► Programmer**.
2. Connect the board cables:
 - JTAG USB cable to board
 - Power cable attached to board and power source
3. Turn on power to the board.
4. In the Intel Quartus Prime Programmer, click **Hardware Setup**.



5. In the **Hardware** list, select **USB-BlasterII**, and then click **Close**. The device chain appears.
Note: If the device chain does not appear, verify the board connections.
6. Click **Auto-Detect**. The device chain populates.
7. In the **Found Devices** list, select the device that matches your design and click **OK**. For this tutorial, select the **10AX115S2** device that matches the 10AX115S2F45I1SG FPGA on the Intel Arria 10 GX Development Kit.

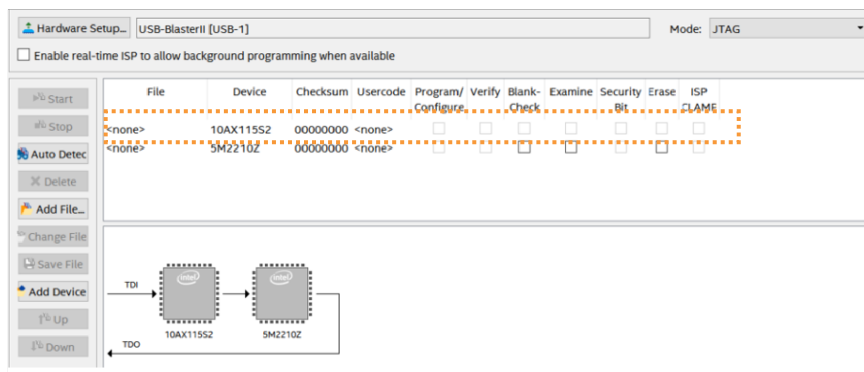


8. Right-click the 10AX115S2 row in the file list, and then click **Change File**.



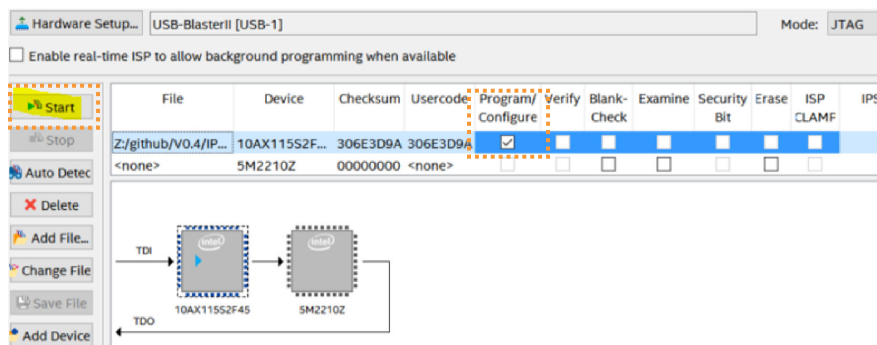
1. AN 851: Incremental Block-Based Compilation Tutorial for Intel® Arria® 10 FPGA Development Board

AN-851 | 2018.06.27

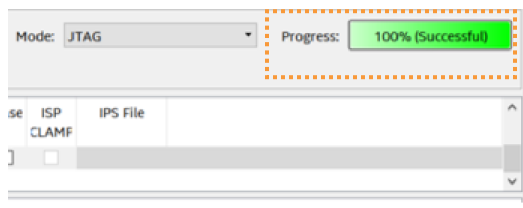


9. Browse to select the `top.sof` file from the appropriate `tutorial/output_files/` directory.

10. Enable the **Program/Configure** option for the 10AX115S2 row.



11. Click **Start**. The progress bar reaches 100% when device configuration is complete. The device is now fully configured and in operation.



Note: If device configuration fails, make sure the device you select for configuration matches the device you specify during `.sof` file generation.



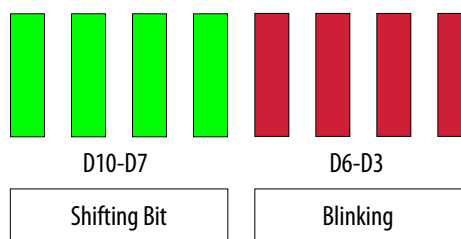
1.3.9. (Optional) Step 9: Verify Results in Hardware

After device programming you can verify the results of this tutorial in hardware. After completing this tutorial, LEDs D6-D3 map to the `blinking_led_top` instance, and LEDs D10-D7 map to the top-level design. After you configure the FPGA with the SRAM Object File (`.sof`), `blinking_led` flashes red LEDs in the following order:

1. D3 blinks every two seconds
2. D4 blinks every four seconds
3. D5 blinks every eight seconds
4. D6 blinks every 16 seconds

The top-level design illuminates LEDs D10-D7 as a shifting bit in green.

Figure 17. Illumination of LEDs during Hardware Verification



1.4. Incremental Block-Based Compilation Tutorial Revision History

Table 3. Document Revision History

Document Version	Software Version	Changes
2018.06.27	18.0	Replaced references to <code>compile.sh</code> with <code>compile.tcl</code> . Replaced references to <code>restore.sh</code> with <code>restore.tcl</code> .
2018.06.26	18.0	Corrected link to design example files.
2018.06.22	18.0	Initial release of the document.