Macro Constraint Usage in ProASIC^PLUS Design Flow

Table of Contents

Introduction . . . 1
Benefits of Macro Constraint Usage . . . 2
Syntax and Context of Macro Constraints . . . 2
A Sample Macro Constraint . . . 3
Macro Generation Procedure . . . 3
Step 1: Set the Macro as Root . . . 6
Step 2: Synthesize the Macro . . . 6
Step 3: Compile the Macro Netlist . . . 6
Step 4: Place the Primitives within the Macro . . . 6
Step 5: Construct the Macro Constraint . . . 7
Step 6: Invoke the Macro Call Instruction . . . 8
Step 7: Use the Macro Constraints in the Top-Level Design . . . 9
Comparison of Timing Performance . . . 10
Conclusion . . . 11
Related Documents . . . 12
Application Notes . . . 12
Datasheets . . . 12
User's Guides . . . 12
List of Changes . . . 13

Introduction

The use of macro constraints offers designers increased performance of sub-blocks and greater control over the configuration and placement of these individual blocks in their design. A macro constraint defines the placement of a macros primitives within a rectangular region of a user specified size. Instances of this macro can be placed at desired locations on a device by invoking macro call instructions. This technique together with the strategies presented in the *AC192: Floorplanning ProASIC/ProASICPLUS Devices for Increased Performance App Note* can be used to achieve timing closure.

This application note describes a method for implementing macro constraints in the ProASIC^PLUS® family of devices. The procedure utilizes Designer placement constraints from an existing layout pass as a basic template for the creation of a macro definition. The benefits of using this clear, block-based design methodology are outlined below.

© 2015 Microsemi Corporation

Benefits of Macro Constraint Usage

Macro constraint usage facilitates a bottom-up approach for achieving timing closure and timing consistency. It allows for the optimization of sub-blocks independently of the larger design to which they belong. In this way, designers can verify that hard to achieve macros meet timing requirements before integrating them into a higher level of a design. This integration can be done with confidence, as macro constraints ensure the highest probability that the timing attributes of a macro block will be preserved. Uniformity of the placement of logic resources provides nearly identical timing characteristics for every instance of the block. Therefore consistent skew, setup or hold timing is another benefit of macro constraint usage.

As with all constraints, placement constraints limit Designer softwares freedom when processing a design. The use of macro constraints transfers the control over placement from the software tool, Designer, to the human designer. Therefore, when a thorough understanding of a design exists, it is possible to achieve better timing performance through the implementation of macro constraints. An example of such a case is given later in this document. A further benefit of macro constraints is the relative ease with which they can be integrated into a design. The step-by-step procedure follows.

Syntax and Context of Macro Constraints

A macro must first be defined before it is placed. A macro definition has the form:

```
macro <macro name> (x1, y1  x2, y2) {
   set_location (<x relative to x1>, <y relative to y1>)
"<instance of/inside the macro>";
set_location (<x relative to x1>, <y relative to y1>)
"<instance of/inside the macro>";
...
}
```

The coordinates (x1, y1 x2, y2) define the size of the macro block. They are interpreted relative to one another. For instance, a macro definition with coordinates (10, 5, 50, 25) describes a block 40 tiles wide and 20 tiles tall. The set_location keyword is used to place the macros primitives within the macro block. The coordinates in these calls are relative to x1 and y1.

A macro calling constraint has the form:

```
set_location (x, y) <desired macro's instance name> <macro name>
   <transformation options>;
```

The coordinates (x, y) in the macro placement are with respect to the lower left hand corner of the device.

Transformations are optional. They can be any of the following, in any order:

1. flip lr – flip cell from left to right
2. flip ud – flip cell from up to down
3. rotate 90 cw – rotate 90° clockwise
4. rotate 270 cw – rotate 270° clockwise
5. rotate 90 ccw – rotate 90° counter-clockwise
6. rotate 180 ccw – rotate 180° counter-clockwise
7. rotate 270 ccw – rotate 270° counter-clockwise

Example:

The following macro call places an instance of the macro example, that has been flipped from left to right and rotated 90° clockwise, at coordinate (45, 30):

```
set_location (45,30) instance1 example flip lr rotate 90 cw;
```

Sample Macro Constraint

Macro Definition

```
macro my_macro_placement (1,1 20,20) {
set_location (1,1) "AND2_0";
set_location (1,2) "AND2_1";
set_location (1,3) "AND2_10";
set_location (1,4) "AND2_11";
set_location (19,8) "XOR2_Sum_7_inst";
set_location (19,9) "XOR2_Sum_8_inst";
set_location (19,10) "XOR2_Sum_9_inst";s
}
```

Macro Call

```
set_location (33,1) TOP_MACRO1 my_macro_placement;
set_location (49,1) TOP_MACRO2 my_macro_placement;
```

Macro Generation Procedure

A sample design (*http://soc.microsemi.com/download/rsc/?f=APA_MacroUsage_DF*) is reference in the remainder of this application note to illustrate the macro generation procedure. The top-level design hierarchy is shown in Figure 1. The design top-level schematic is shown in Figure 2 on page 4. It consists of two counter macros and a 32-bit wide FIFO.

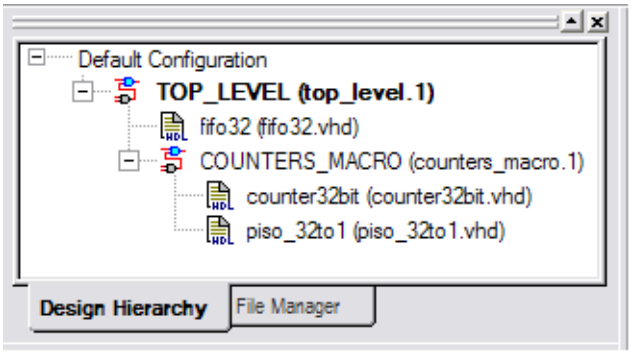


***Figure 1* • Libero IDE Design Hierarchy Window for Sample Design**

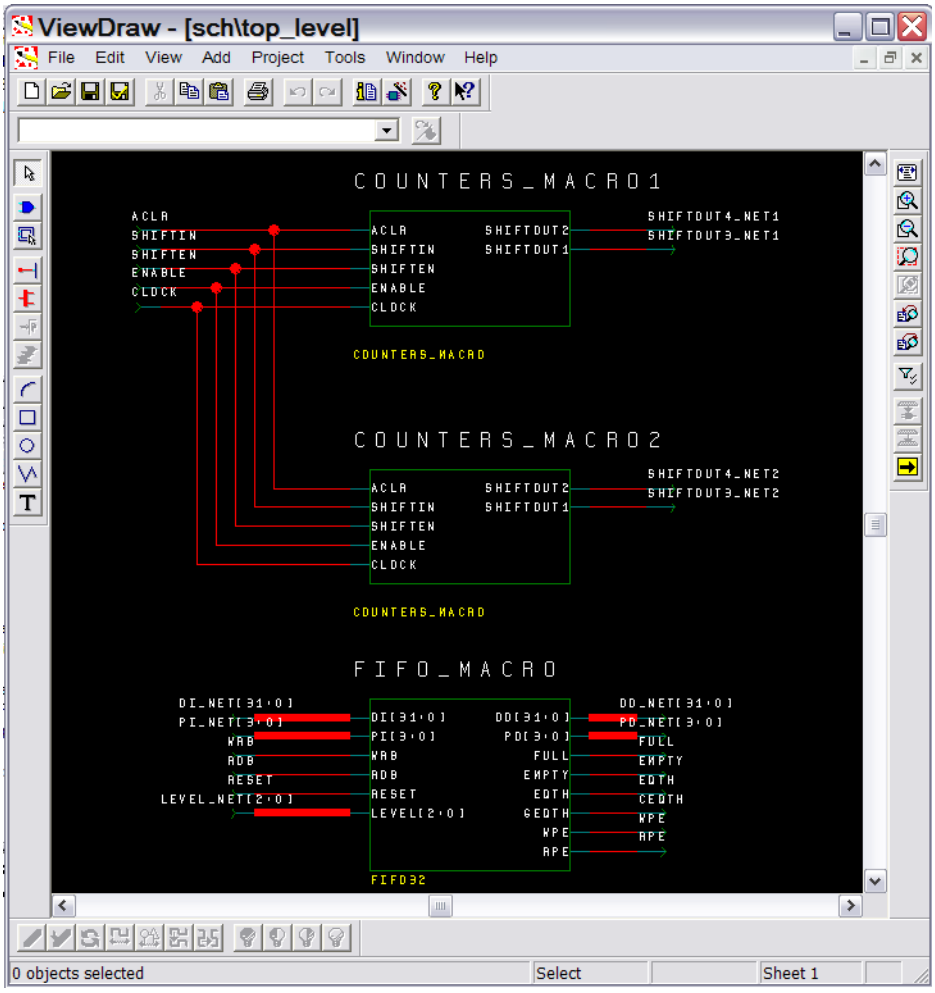


***Figure 2* • Top-Level Design (TOP_LEVEL)**

Each top-level counter is composed of an additional two parallel-in serial-out shift registers (PISO) and two 32-bit counters each, as shown in Figure 3. One top-level counter macro will be created in this design. Two instances of this macro, named **COUNTERS_MACRO1** and **COUNTERS_MACRO2** is placed on the device through macro call instructions. The results is as shown in Figure 4.

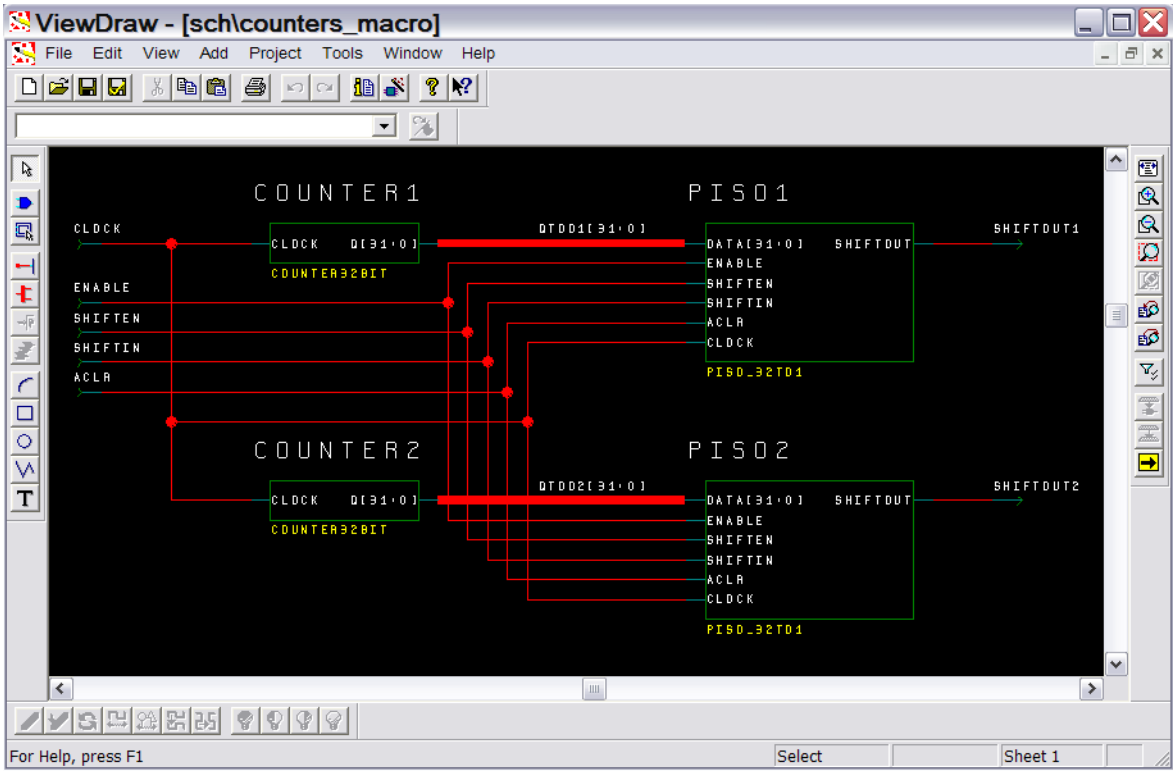


***Figure 3* • Macro Level (COUNTERS_MACRO)**

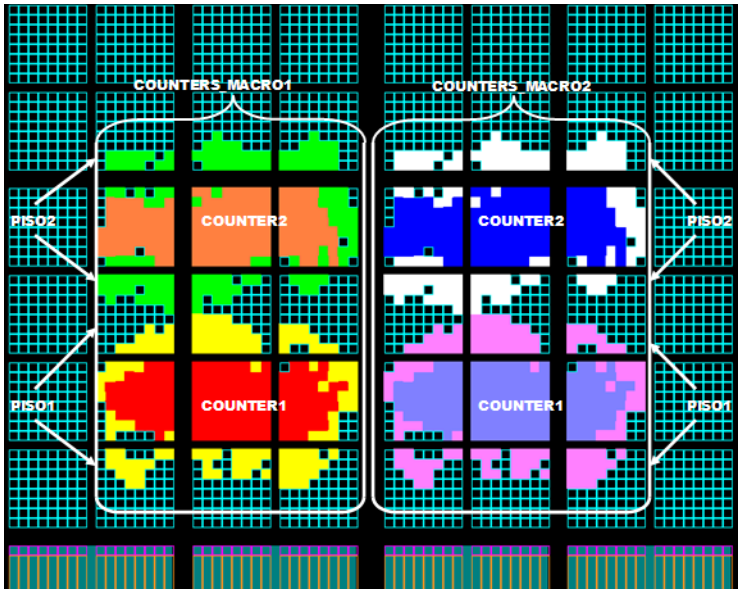


***Figure 4* • Design Block Diagram**

The use of macro constraints in this application ensures consistent timing across both counter macros. It will also be shown that performance gains were achieved in this design through the use of macro constraints.

Step 1: Set the Macro as Root

Within Libero® System-on-Chip integrated design environment (IDE), set the macro level to be the root of the design. User can do this in the design hierarchy window by right-clicking the macro and selecting set as root. The macro level is displayed in bold. The result of setting the **COUNTERS_MACRO** as root in the sample design can be seen in Figure 5.

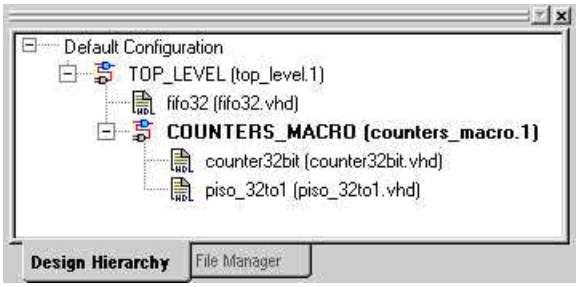


Figure 5 • **Macro Level Set as Root**

Step 2: Synthesize the Macro

Within Libero IDE, invoke Synplify® and run synthesis. As the macro level is now set to root, the resulting netlist will contain only one instance of the macro.

Step 3: Compile the Macro Netlist

Following synthesis, invoke Designer from within Libero IDE. In Designer, compile the netlist from step 2.

Step 4: Place the Primitives within the Macro

The relative placement of the macros primitives within the macro block can be done in one of two ways: by using Designers automatic placement or by using manual placement. The manual placement method is suitable for smaller macros, whereas for larger, more complex macros, it is recommended that user have Designer automatically optimize the layout.

Designer Placement

Within Designer, open ChipPlanner. Create an inclusive region at the lower left corner of the array tiles. For example, for APA300, the array coordinate for the lower left corner of the array is (1, 5). The dimensions of this inclusive region will define the dimensions of the macro block. Ensure to make the inclusive region large enough to accommodate all of the macros primitives. In other words, the area of the inclusive region must be at least as large as the number of core cells used in the design (which consists of only the macro). Place-and-route will generate errors if the logic assigned to user region exceeds its capacity. If user size the region too tight, place-and-route may not have enough room to route the logic within the region and may fail because of routing congestion. Therefore, it is a good practice to oversize your region by approximately 10-20% to provide place-and-route enough room to route the assigned logic.

When the inclusive region has been defined, user must assign the macro logic to it. User can do so by right-clicking the region and selecting **Assign/Unassign…**. In the **Assign Instances to Region** dialog, select **Assign All >>** and click **OK**. For more information about using region constraints, refer to the *MultiView Navigator v9.1 User's Guide*.

Manual Placement

Within Designer, open ChipPlanner. Start from the lower left corner of the array. Hand place each primitive by dragging it from the logical view of the hierarchy window and dropping it onto the desired location within the chip. For more information about assigning logic to specific locations, refer to the *MultiView Navigator v9.1 User's Guide*.

When choosing primitive placements, user must consider the macros interaction with other components in the design. For example, it may be beneficial to place the macros inputs and outputs closer to the perimeter of the macro region rather than in the very center of the region. This is especially true for larger macros. In general, it is essential to consider the macro relative to the overall design in order to achieve optimal performance.

When placement is complete, **Commit** your changes and close MultiView Navigator. Open timer and enter any timing constraints including the required clock frequency. **Commit** the timing constraints and close Timer. Within Designer, run **Layout**.

If users are using the Designer placement method, user may wish to take advantage of the **Multiple Pass Layout** option in Designer. By doing so, Designer will place-and-route the primitives within the inclusive region multiple times using a different placement seed for each pass. Multiple pass layout attempts to improve layout quality by selecting from a greater number of layout results. For more information about multiple pass layout, refer to the *Designer v9.1 User's Guide*.

After **Layout**, open timer and observe the timing performance of the macro. If greater performance is required, make adjustments in chipplanner and rerun **Layout**. This is when the effort must be spent to perfect the timing characteristics of the block.

When user is satisfied with the timing performance of the macro, move on to Step 5.

Step 5: Construct the Macro Constraint

User now construct the macro constraint definition using the layout created in the previous step. To do so, navigate to the */<project_name>/designer/impl<#>/<design_name>.dtf* directory and open the *last_placement.gcf* file in a text editor. This placement constraint file contains the coordinates of the macros primitives within the macro block.

The following is the *last_placement.gcf* file for the sample design:

```
// Design: TOP_LEVEL
// Technology: APA300
// Array: APA300-PQ208
…
set_io E 29 "ACLR_pad/IOTILE";
set_io E 32 "ACLR_pad/MUXTILE";
set_io W 36 "CLOCK_pad/IOTILE";
set_io W 33 "CLOCK_pad/MUXTILE";
...
set_initial_location (17,10) "COUNTER1/AND2_0";
set_initial_location (11,9) "COUNTER1/AND2_1";
set_initial_location (11,16) "COUNTER1/AND2_11";
set_initial_location (13,9) "COUNTER1/AND2_12";
set_initial_location (6,12) "COUNTER1/AND2_13";
set_initial_location (5,14) "COUNTER1/AND2_14";
set_initial_location (18,9) "COUNTER1/AND2_3";
set_initial_location (21,10) "COUNTER1/AND2_4";
set_initial_location (7,9) "COUNTER1/AND2_5";
set_initial_location (7,9) "COUNTER1/AND2_5";
set_initial_location (6,16) "COUNTER1/AND2_6";
set_initial_location (8,15) "COUNTER1/AND2_7";
set_initial_location (5,11) "COUNTER1/AND2_9";
...
set_io W 33 "SHIFTEN_pad/IOTILE";
set_io W 36 "SHIFTEN_pad/MUXTILE";
set_initial_io W 47 "SHIFTIN_pad";
set_initial_io N 65 "SHIFTOUT1_pad";
set_initial_io S 59 "SHIFTOUT2_pad";
```

The file will contain different placement constraints. User must remove all placement constraints with the exception of the set_initial_location constraints. Next, replace all instances of set_initial_location with the keyword set_location. Save the file with a name that will be significant, such as *my_macro_placement.gcf*.

Above the first set_initial_location command, type the following:

```
macro <macro_name> (<x1,y1> <x2,y2>) {
```

where <macro_name> and the filename can be the same so as to eliminate any potential for confusion. <x1,y1> represents the lower left corner of the region within which the macro's primitives were placed in Step 4. <x2,y2> represents the upper right corner.

Finally, place a closing brace (}) below the last set_location constraint to encompass the placements. The macro definition is now complete. The code below shows an example of how the macro is defined with the coordinates.

```
macro my_macro_placement (1,1 24,36) {

set_location (17,10) "COUNTER1/AND2_0";
set_location (11,9) "COUNTER1/AND2_1";
set_location (11,16) "COUNTER1/AND2_11";
...
set_location (16,31) "COUNTER2/AND2_0";
set_location (14,25) "COUNTER2/AND2_1";
set_location (1,27) "COUNTER2/AND2_11";
...
set_location (13,8) "PISO1/DFFC_0";
set_location (9,20) "PISO1/DFFC_1";
set_location (18,7) "PISO1/DFFC_10";
...
set_location (22,31) "PISO2/DFFC_0";
set_location (12,35) "PISO2/DFFC_1";
set_location (17,34) "PISO2/DFFC_10";
...
}
```

Step 6: Invoke the Macro Call Instruction

User can now place instances of this macro in the device by invoking the macro call instruction. In the constraint file, set_location calls must never precede the definition they refer to. Again, the coordinates in these calls are with respect to the lower left hand corner of the device (Refer to the *ProASICPLUS Flash Family FPGAs Datasheet (v5.9)* for the array coordinates of each device).

In sample design, the top-level counter macros were placed as follows:

```
set_location (41,5) COUNTERS_MACRO1 my_macro_placement;
set_location (65,5) COUNTERS_MACRO2 my_macro_placement;
```

Notice that the macro call instructions call for the my_macro_placement macro, which contains the locations coordinates for individual primitives. Therefore, as mentioned earlier, the primitives for each macro must have the same primitive instance names. Save your file when completed.

Step 7: Use the Macro Constraints in the Top-Level Design

User will now run the entire design in Designer with the macro constraints. First, set the top-level of the design as root and synthesize. Import the resulting netlist and the GCF containing the macro constraints (*my_macro_placement.gcf*) into Designer. **Compile** and run **Layout** on the design. User may wish to take advantage of the multiple pass layout option in Designer at this point. Doing so will modify the routing and likely improve routing quality. Figure 6 shows the layout results using macro constraints created by manual placement.

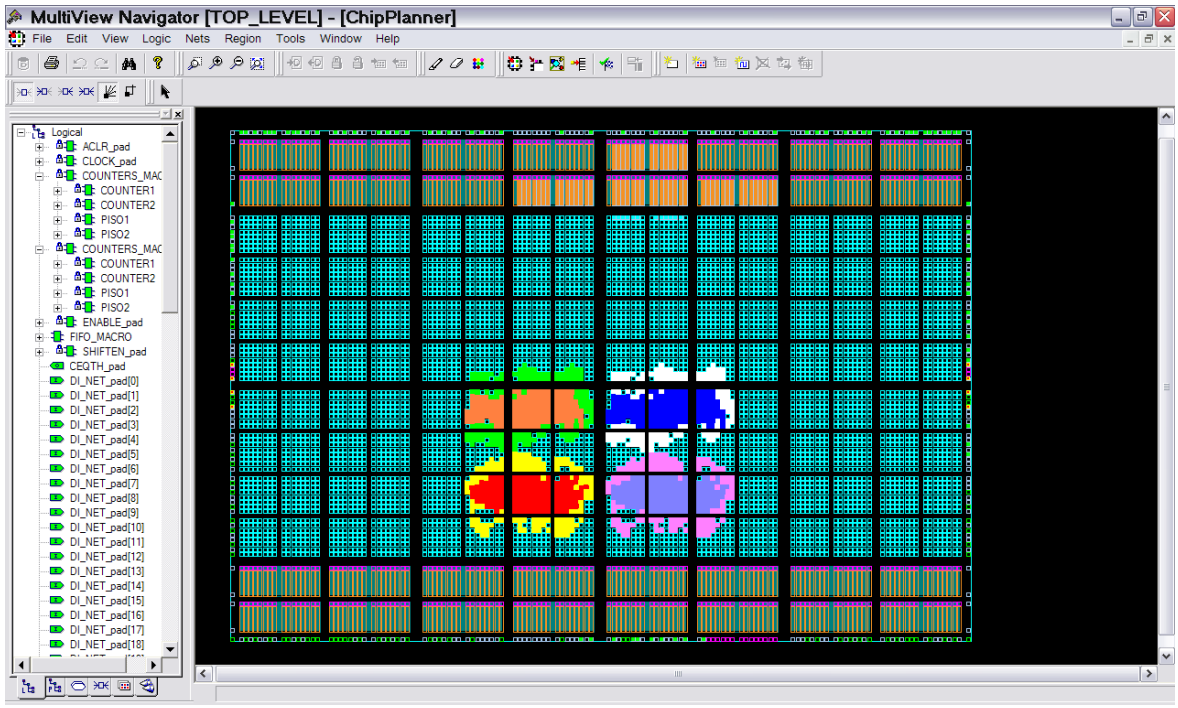


***Figure 6* • Layout Results with Macro Constraints**

Comparison of Timing Performance

In the sample design, a slight performance increase is realized with the addition of macro constraints (Figure 7 without macro constraints vs Figure 8 with macro constraints). Figure 9 on page 11 and Figure 10 on page 11 show comparison of individual timing paths between designs with and without macro constraints. Most importantly, consistent timing can be achieved for desired individual paths if the macro constraint was used in the design, as shown in Figure 10 on page 11.

In general, performance results will vary with design. Reaching timing closure using macro constraints is an iterative process. Using macro constraints without a clear understanding of the target architecture may result in a reduction of performance. Always run place-and-route without constraints to determine if this meets the designs targeted performance. Moreover, to achieve optimal timing results it is critical to place macros appropriately. This requires a strong understanding of the interaction between the various components in the design.

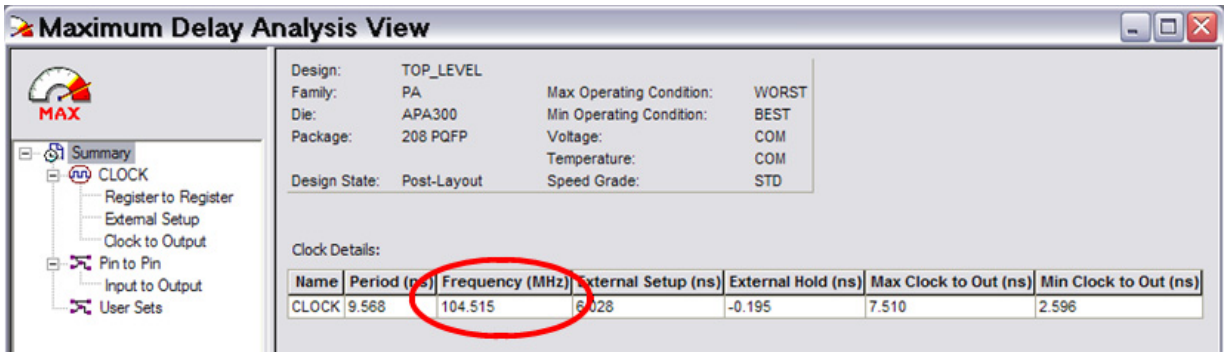


***Figure 7* • Timing Without Macro Constraints**

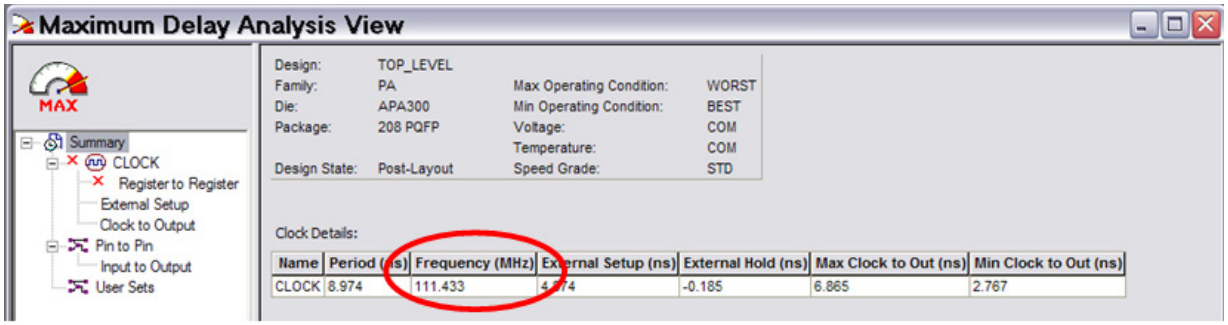
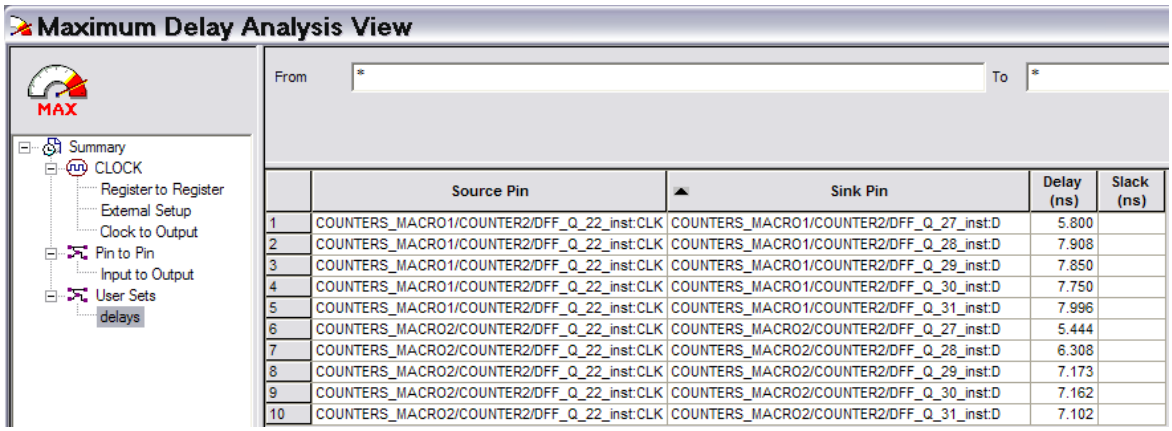
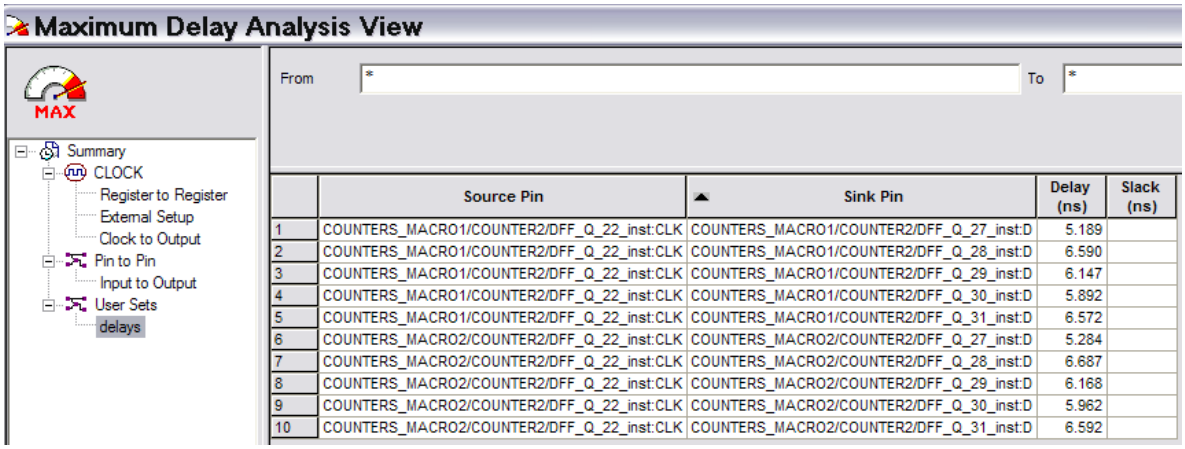


***Figure 8* • Timing With Macro Constraints**



	Source Pin	Sink Pin	Delay (ns)	Slack (ns)
1	COUNTERS_MACRO1/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO1/COUNTER2/DFF_Q_27_inst:D	5.800	
2	COUNTERS_MACRO1/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO1/COUNTER2/DFF_Q_28_inst:D	7.908	
3	COUNTERS_MACRO1/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO1/COUNTER2/DFF_Q_29_inst:D	7.850	
4	COUNTERS_MACRO1/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO1/COUNTER2/DFF_Q_30_inst:D	7.750	
5	COUNTERS_MACRO1/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO1/COUNTER2/DFF_Q_31_inst:D	7.996	
6	COUNTERS_MACRO2/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO2/COUNTER2/DFF_Q_27_inst:D	5.444	
7	COUNTERS_MACRO2/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO2/COUNTER2/DFF_Q_28_inst:D	6.308	
8	COUNTERS_MACRO2/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO2/COUNTER2/DFF_Q_29_inst:D	7.173	
9	COUNTERS_MACRO2/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO2/COUNTER2/DFF_Q_30_inst:D	7.162	
10	COUNTERS_MACRO2/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO2/COUNTER2/DFF_Q_31_inst:D	7.102	

***Figure 9* • Without Macro Constraints**



	Source Pin	Sink Pin	Delay (ns)	Slack (ns)
1	COUNTERS_MACRO1/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO1/COUNTER2/DFF_Q_27_inst:D	5.189	
2	COUNTERS_MACRO1/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO1/COUNTER2/DFF_Q_28_inst:D	6.590	
3	COUNTERS_MACRO1/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO1/COUNTER2/DFF_Q_29_inst:D	6.147	
4	COUNTERS_MACRO1/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO1/COUNTER2/DFF_Q_30_inst:D	5.892	
5	COUNTERS_MACRO1/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO1/COUNTER2/DFF_Q_31_inst:D	6.572	
6	COUNTERS_MACRO2/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO2/COUNTER2/DFF_Q_27_inst:D	5.284	
7	COUNTERS_MACRO2/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO2/COUNTER2/DFF_Q_28_inst:D	6.687	
8	COUNTERS_MACRO2/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO2/COUNTER2/DFF_Q_29_inst:D	6.168	
9	COUNTERS_MACRO2/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO2/COUNTER2/DFF_Q_30_inst:D	5.962	
10	COUNTERS_MACRO2/COUNTER2/DFF_Q_22_inst:CLK	COUNTERS_MACRO2/COUNTER2/DFF_Q_31_inst:D	6.592	

***Figure 10* • With Macro Constraints**

Conclusion

Macro constraint usage is an effective strategy for achieving timing closure in a bottom-up fashion. Furthermore, it gives designers full control over the configuration and placement of the macro blocks in their design. This constraining of Designer softwares layout freedom does not necessarily have a negative effect on performance. It is shown in this application note that it is possible to maintain timing performance after the integration of macro constraints; or even improve it.

Related Documents

Application Notes

AC192: Floorplanning ProASIC/ProASICPLUS Devices for Increased Performance

Datasheets

ProASICPLUS Flash Family FPGAs Datasheet

User’s Guides

Designer v9.1 User's Guide

MultiView Navigator v9.1 User's Guide

List of Changes

The following table shows important changes made in this document for each revision.

Revision	Changes	Pages
Revision 1 (December 2015)	Non-technical updates.	NA
Revision 0 (January 2006)	Initial release.	NA

**The part number is located on the last page of the document.*

Microsemi Corporate Headquarters
One Enterprise, Aliso Viejo,
CA 92656 USA

Within the USA: +1 (800) 713-4113
Outside the USA: +1 (949) 380-6100
Sales: +1 (949) 380-6136
Fax: +1 (949) 215-4996

E-mail: sales.support@microsemi.com

© 2015 Microsemi Corporation. All rights reserved. Microsemi and the Microsemi logo are trademarks of Microsemi Corporation. All other trademarks and service marks are the property of their respective owners.

Microsemi Corporation (Nasdaq: MSCC) offers a comprehensive portfolio of semiconductor and system solutions for communications, defense & security, aerospace and industrial markets. Products include high-performance and radiation-hardened analog mixed-signal integrated circuits, FPGAs, SoCs and ASICs; power management products; timing and synchronization devices and precise time solutions, setting the world's standard for time; voice processing devices; RF solutions; discrete components; security technologies and scalable anti-tamper products; Ethernet Solutions; Power-over-Ethernet ICs and midspans; as well as custom design capabilities and services. Microsemi is headquartered in Aliso Viejo, Calif., and has approximately 3,600 employees globally. Learn more at **www.microsemi.com**.

Microsemi makes no warranty, representation, or guarantee regarding the information contained herein or the suitability of its products and services for any particular purpose, nor does Microsemi assume any liability whatsoever arising out of the application or use of any product or circuit. The products sold hereunder and any other products sold by Microsemi have been subject to limited testing and should not be used in conjunction with mission-critical equipment or applications. Any performance specifications are believed to be reliable but are not verified, and Buyer must conduct and complete all performance and other testing of the products, alone and together with, or installed in, any end-products. Buyer shall not rely on any data and performance specifications or parameters provided by Microsemi. It is the Buyer's responsibility to independently determine suitability of any products and to test and verify the same. The information provided by Microsemi hereunder is provided "as is, where is" and with all faults, and the entire risk associated with such information is entirely with the Buyer. Microsemi does not grant, explicitly or implicitly, to any party any patent rights, licenses, or any other IP rights, whether with regard to such information itself or anything described by such information. Information provided in this document is proprietary to Microsemi, and Microsemi reserves the right to make any changes to the information in this document or to any products and services at any time without notice.

51900125-1/12.15