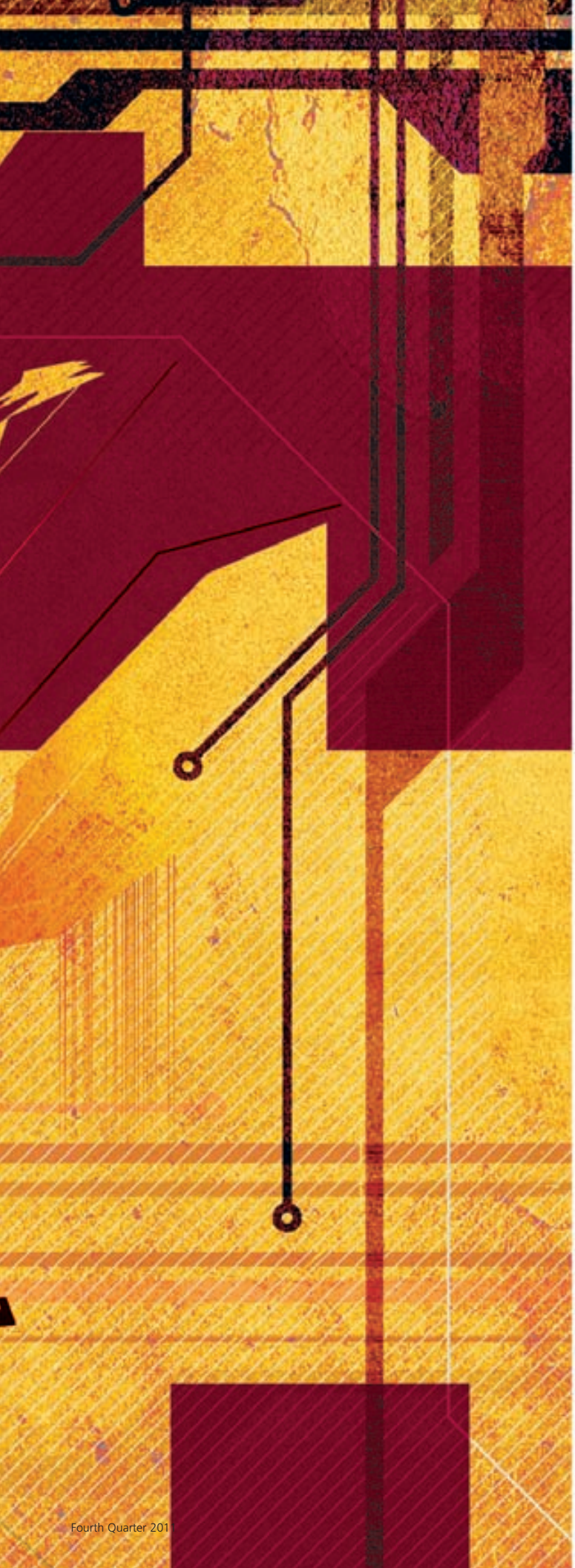
Using the Clock Period Constraint to Your Advantage

Understanding the TS_clk constraint in terms of Xilinx ISE and tool behavior will help you attain effective timing closure in FPGA designs.

by Sharad Sinha
PhD Candidate
Center for High-Performance Embedded Systems
Nanyang Technological University
sharad_sinha@pmail.ntu.edu.sg



Designers set timing constraints to meet the particular timing requirements of their chip designs. Then the physical-synthesis tools place and route the design so as to meet these timing constraints. One very common and important timing constraint is related to the maximum clock frequency and is commonly referred to as the period constraint. In the Xilinx® ISE® tool suite, this constraint goes by the name TS_clk in the user constraint file (UCF). The Xilinx Timing Constraints User Guide states that the period constraint is used to:

1) Define each clock in a design
2) Cover all synchronous paths within each clock domain
3) Cross-check paths between related clock domains
4) Define the duration of clocks
5) Define the duty cycle of clocks

The user guide offers a wealth of details and good, relevant explanations of the functions of the period constraint. But it's worth taking a closer look "under the hood" of the FPGA synthesis tools to explore a number of questions about the behavior of the period constraint, and to gain insight into the way placement-and-routing algorithms work. Specifically, let's consider ways to interpret a failing TS_clk constraint; examine whether you'll get progressively better results by constraining the tool progressively; and discuss why the tool shows a discrepancy in results. Finally, let's also ask whether such a timing constraint always helps vs. proceeding with an unconstrained design.

HOW TO INTERPRET A FAILING TS_CLK CONSTRAINT

If a design fails to meet the clock period constraint, it means that it cannot run at that clock frequency. You can attempt thereafter to pipeline the design so as to relax the timing budget in the slow paths. Pipelining may be enough to make the design meet the constraint. Another way to improve timing is to reduce the number of logic levels between two registers—essentially, you need to simplify the logic design in the critical path. These two techniques are applicable at the design level, where a designer can do the necessary modifications to the RTL code. If your design still has not met timing after the RTL modifications, the next step is to enable the Xilinx ISE switches -register_balancing (which is meant for register retiming) and -register_duplication (which duplicates registers to ease high fanout at a particular register).

Another way to improve timing is to assign pins to I/O signals properly. A good design practice is to assign adja-

cent pins to adjacent signals. For instance, you should assign all the signals on an I/O bus to adjacent pins in one bank. Use adjacent banks while assigning large numbers of pins.

These points are significant because they act as constraints for the place-and-route tool. The tool would generally try to keep related logic together. This effort improves when relevant I/Os are assigned adjacent pins, because the technique will likely decrease routing delays. The tool would then not scatter the logic on the device. Scattering logic increases routing delays.

Generally, when an FPGA has to sit on a printed-circuit board, you need to take additional board-related considerations into account while assigning pins. Since the FPGA would interface with other chips on a board, adjacent pin assignments may not always be possible. Therefore, it is always best for an FPGA designer to consult with the board designer early in the design cycle to reduce pin assignment conflicts.

Still another way to improve timing is to use a higher-speed-grade device. However, this affects the price of the product and hence is not an easy option. Not only is there the higher cost of the device itself to take into account, but also a higher-speed part has an impact on board design and could easily increase the board design cost as well.

DOES THE TOOL ALWAYS GIVE YOU A BETTER RESULT AS YOU CONSTRAIN IT PROGRESSIVELY?

Sometimes we would like to know the maximum frequency at which a particular design can run. To investigate this, we constrain the design progressively. For instance, we start with a clock period constraint of 8 nanoseconds (corresponding to a clock frequency of 125 MHz). If the tool succeeds in placing and routing the design under the constraint, it might report a minimum clock period of something like 7.68 ns. We can then constrain the clock period to 7.68 ns and rerun ISE. This time the tool might report a minimum clock period of 7.56 ns. However, when we constrain the design again to 7.56 ns, the tool might report timing failure with the minimum possible clock period being, say, 7.74 ns. This means that we need to constrain the design to 7.68 ns in order to achieve the figure of 7.56 ns. So, there is a limit to progressive constraining of a design in order to improve timing. After a certain level of constraint, the results might deteriorate.

If the design is small and the period constraint is very tight (a very small period), the tool may report a clock value less than the period constraint in the post-place-and-route (PPR) static timing report. But it would still show a timing-error score (which is zero when there is no timing error) like the one on this page, from an actual static timing report of a design (targeted at XC4VFX140-11FF1517) in which the period constraint was set to 1.5 ns with 50 percent duty cycle:

```
1 constraint not met.
Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)
Clock to Setup on destination clock clk
-------------+-----------+----------+-----------+--------------+
             | Src: Rise | Src: Fall | Src: Rise |  Src: Fall   |
Source Clock | Dest: Rise | Dest: Rise | Dest: Fall | Dest: Fall|
-------------+-----------+----------+-----------+---------------+
clk          | 1.489                |           |              |
-------------+-----------+----------+-----------+---------------+
Timing summary:
---------------
Timing errors: 1  Score: 722  (Setup/Max: 0, Hold: 0, Component
Switching Limit: 722)
Constraints cover 40 paths, 0 nets and 62 connections

Design statistics:
---------------
Minimum period:   2.222 ns{1}   (Maximum frequency: 450.045 MHz)
```

The report clearly shows a clock period of 1.489 ns, which is less than 1.5 ns. However, this design was targeted at a speed-grade -11 Virtex®-4 device with a maximum frequency of 450.05 MHz. Hence, there is a timing error. The point is that it is important to read about the device's switching characteristics also while setting the constraint.

WHY DOES THE TOOL SHOW A DISCREPANCY IN RESULTS?

The tool begins to show a discrepancy in results because it works based on heuristic algorithms. Designers use these algorithms to solve problems for which exact algorithms are either unsuitable, for reasons of time and space complexities, or extremely difficult to develop. To choose a solution, heuristic algorithms generally use a so-called cost function, which takes into account some information about the device or some other empirically derived constants. These algorithms do not, however, guarantee that the solution will be the best or the optimal one.

Often heuristic algorithms start with an initial random seed value for placement of logic; then the placement process grows around the seed location based on the cost function analysis, and routing follows. Since the seed value may change with each invocation of the tool following each change in constraint, the results can get worse beyond a certain point. The tool has no reference of what it did and what results it reported in the last run so as to improve its working further. It is extremely difficult to design a place-and-route algorithm that takes a prior

placement-and-routing strategy into account, and compares the current and the previous results.

The SmartGuide™ technology in Xilinx ISE can guide a new implementation based on results from a previous one. But SmartGuide works only if there is some change in logic between two iterations of an implementation. It is not applicable when there is no change in logic and the same design is simply constrained progressively. Quite often, designers get confused with this distinction.

For its part, the SmartXplorer option in Xilinx ISE is simply one way of speeding up the process of investigating timing with different timing constraints. This strategy allows a designer to investigate the same design with different constraints as the tool executes them in parallel on different machines: on a Linux network or a Linux/Windows machine with multiple processors.

Thus, even with these options available in Xilinx ISE (and similar options in other FPGA design suites), the tool does not remember what it did in the previous run in order to compare and improve its results in the next run when timing constraints are progressively tighter. If that were the case, then beyond a certain point, the tool should simply report one minimum value instead of different values on constraining progressively. Since it's so difficult to design an algorithm that uses prior placement-and-routing information as a feedback to improve the timing of the same design, it pays to know and understand the limitations of the tools.

DOES A TIMING CONSTRAINT ALWAYS HELP OVER AN UNCONSTRAINED DESIGN?

The traditional thinking is that a constrained design will always have better timing than an unconstrained design. This is true in general. However, it is not always the case. Sometimes the tool places and routes the design in the best possible way when the design is unconstrained. The maximum achievable clock frequency is highest in the unconstrained implementation, as highlighted in Table 1. The reason for this discrepancy is again the way place-and-route algorithms work.

Our team at the Center for High-Performance Embedded Systems at Nanyang Technological University implemented the sum-of-absolute-differences (SAD) algorithm on a Virtex-4 XC4VFX140-11FF1517 FPGA using Xilinx ISE version 12.2 M.63C (see *Xcell Journal* Issue 75, page 38). We employed an 8 x 8 SAD that used eight image pixels (16 bits each) and eight reference pixels (16 bits each), using external select signals to choose which two pixels to subtract so that the design would have only one subtractor. We did not use any conditioning registers, and all internal registers were initialized to zero. We did no pin assignment for this experiment.

As you can see from Table 1, we achieved the best minimum clock period of 2.607 ns when no constraint was set. When we set 2.607 ns as the period constraint, the tool reported 2.863 ns as the best achievable clock period. Setting 2.863 ns as the period constraint resulted in a best achievable clock period of 2.795 ns. This is because the tool stops trying to meet the constraint once it achieves a value close to it. Setting 2.795 ns as the new constraint did not bring down the best achievable clock period to 2.607 ns but raised it to 2.966 ns. In this case, the tool failed to meet the constraint.

Period constraint (ns)	PPR reported value (ns)
No constraint	2.607
2.607	2.863
2.863	2.795
2.795	2.966
2.966	2.762

Table 1 – Effect of period constraint on actual timing

This randomness in results stems from the heuristic nature of place-and-route algorithms. This is also the reason why designers need to spend significant time in setting and resetting period constraints to meet timing closure.

PSEUDO-RANDOM SOLUTION

The period constraint is one of the most important constraints in FPGA design and is critical to timing closure. It is therefore important to understand how it behaves and how to interpret its results. Progressively constraining the clock period does not always improve the result. Vendors generally enhance the place-and-route algorithm implemented in their FPGA design tools upon every major release of the software. Hence, the timing results may vary from one version to another.

Unlike ASICs, where routing and placement are highly deterministic, FPGA placement-and-routing algorithms are heuristic in nature. This is easy to understand because of the very nature of FPGAs, where random logic has to be mapped onto a fixed hardware architecture with fixed components and routing resources.

FPGA placement-and-routing is an NP-complete problem —one for which there is no efficient way to locate a solution. For such problems, no known polynomial-time algorithms exist that can give an accurate or optimal solution. Hence, solving them involves using heuristics or some approximation or similar methods based on pseudo-random processes.

Also, the runtimes of these algorithms can vary rapidly with any increase in the size of the input, as many of us have experienced with large FPGA designs. This is a fundamental aspect of NP-complete problems. For this reason, the quality of results depends a lot on the type of heuristics used or the approximation method employed.