Designing a robust clock tree structure

Amol Agarwal and Priyanka Garg - Freescale Semiconductor - August 06, 2012

Clock tree synthesis (CTS) is at the heart of ASIC design and clock tree network robustness is one of the most important quality metrics of SoC design. With technology advancement happened over the past one and half decade, clock tree robustness has become an even more critical factor affecting SoC performance. Conventionally, engineers focus on designing a symmetrical clock tree with minimum latency and skew. However, with the current complex design needs, this is not enough.

Today, SoCs are designed to support multiple features. They have multiple clock sources and user modes which makes the clock tree architecture complex. Merging test clocking with functional clocking and lower technology nodes adds to this complexity. Due to the increase in derate numbers and additional timing signoff corners, timing margins are shrinking.

To meet the current requirements, designs that are timing friendly are needed and provide minimum power dissipation. This article describes the factors which a designer should consider while defining clock tree architecture. It presents some real design examples that illustrate how current EDA tools or conventional methodologies to design clock trees are not sufficient in all cases. A designer has to understanding the nitty -gritty of clock tree architecture to be able to guide an EDA tool to build a more efficient clock tree. First, the basics of CTS and requirements for good clock tree are presented.

Clock tree quality parameters

The primary requirements for ideal synchronous clocks are:

1. **Minimum Latency** – The latency of a clock is defined as the total time that a clock signal takes to propagate from the clock source to a specific register clock pin inside the design. The advantages of building a clock with minimum latency are obvious – fewer clock tree buffers, reduced clock power dissipation, less routing resources and relaxed timing closure.
2. **Minimum skew** – The difference in arrival time of a clock at flip-flops is defined as skew. Minimum skew helps with timing closure, especially hold timing closure. However there is a word of caution - targeting too aggressive minimum skews can be counterproductive because it may not help meeting hold timing but it can end up having other problems like increasing overall clock latency and increasing uncommon paths between registers in order to achieve minimum skew.
3. **Duty Cycle** – Maintaining a good duty cycle for the clock network is another important requirement. Many sequential devices, like flash, require minimum pulse width on the input clock to ensure error-free operation. Moreover many IO interfaces like DDR and QSPI can work on both edges of clock. A clock tree must be designed with these considerations and symmetrical cells having similar rise-fall delays should be used to build the clock tree.
4. **Minimum Uncommon path** - The logically connected registers must have minimum uncommon clock path. Timing derates are applied to the clock path to model process variations on the die. Using a standard timing derates methodology, derates are applied only on uncommon path of

launch and capture clock path because it is unlikely that common clock paths can have different process variations in launch and capture cycle. This concept is also called CRPR adjustment. The important concept is that a clock path should have minimum uncommon path between two connected registers.

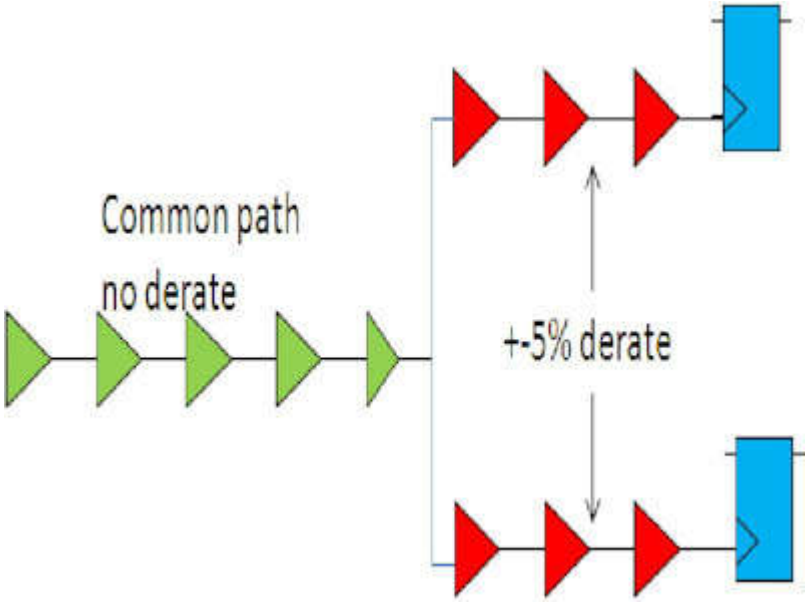

Figure 1: Common and ucommon clock paths between two registers

5. Signal Integrity – Clock signals are more prone to signal integrity problems because of high switching activity. To avoid the effect of noise and to avoid EM violations, clock trees should be constructed using a DWDS(Double width double spacing) rule. Increased spacing will help in minimizing noise effect. Similarly, increased width will help to avoid EM violations.
6. Minimum Power Dissipation – This is one of the most important quality parameter of a clock tree. At the architecture level, clock gating is done at multiple levels to save power and certain things are expected to done while building clock trees such as maintaining good clock transition, minimum latency etc.

EDA tool role in clock tree synthesis

Today, a lot of R&D has been done on EDA tools to design an ideal clock tree. The CTS engines of these tools support most of the SOC requirements to design a robust clock tree. These tools even generate clock spec definitions from SDC(timing constraint files). A typical clock spec file includes:

- All clock sources information
- Synchronous/Asynchronous relationships between various clocks
- Through pins
- Exclude pins
- Clock pulling pushing information
- Leaf Pin

Going one level down in SoC to design an ideal clock tree

For most SoCs, the existing EDA tools are sufficient for CTS engine to generate an ideal clock tree. However, this is not always the case. This approach presented in this paper is suitable for SoCs or IPs, which have few clock sources and a simple clock architecture with minimum muxing of multiple clocks.

Today's microcontrollers generally don't have such a simple clock architecture. Microcontrollers designed for the automotive world have multiple IPs integrated into a single SoC. For example, a single SOC may have multiple cores, IO peripherals like SPI, DSPI, LIN, DDR interfaces for multiple automotive control applications. Considering human safety in automotive SoCs, testing requirements are also very stringent in terms of test coverage such as atspeed and stuckat. This leads to a very complex clocking architecture because it requires multiple clock sources (both on SoC clock sources such as PLLs, IRC oscillators and off SoC clock sources like EXTAL) and clock dividers in order to supply the required clock frequency to multiple IPs within a SoC.

In such cases, CTS engines cannot be relied upon to build a clock tree. Due to complicated muxing of various clock sources in multiple functional and test modes , EDA tools sometimes are not able to build the clock tree properly, often resulting in problems of increased latency, skew mismatch and huge uncommon clock path problems.

In the next section some real design case studies are used to illustrate how current EDA tools might fail to build the clock tree as expected by the designer and how a backend engineer can help design a robust clock tree either by providing proactive feedback to architecture designers or to improve the clock structure at the RT level itself or by using better implementing techniques.

Case study 1 - Clock logic cloning

Suppose a clock tree is required for the following logic.

In functional mode there is one master clock source *func* and one generated clock source gen_clk1. In test modes there is one test clocks tck1. In functional mode register set 2 is clocked by gen_clk1 but in test mode, test clock tck1 is used instead.

The conventional way to define the clock tree spec for this design fragment would be to define the master clock sources (*func* and tck1) and generated clock (gen_clk1) and to define ***through*** pin for generated clock source so as to balance the latency of the master clock and the total latency of the generated clock(source latency to register clock pin plus latency from flop output to register group3). Defining a ***through*** pin for the generated clock source ensures that a CTS engine does not consider the generated clock flop as a sink pin and instead traces the clock path through CK-> Q arc of flop.

Assume that the latency of *func* clock while in functional mode is constrained by register set2 (highlighted in red in figure 2). This will force a CTS engine to build the generated clock source flop 2 with minimum latency. This will only be possible if the minimum buffering is done from func clock source to mux1 input *D0* as well as from mux1 output to generated clock source gen_clk1. In order to balance the latency of register set 1 and register set 2, the tool will be forced to insert buffering between mux1 output and register set 1 clock pins. This implementation is correct in functional mode but will cause problems in test mode.

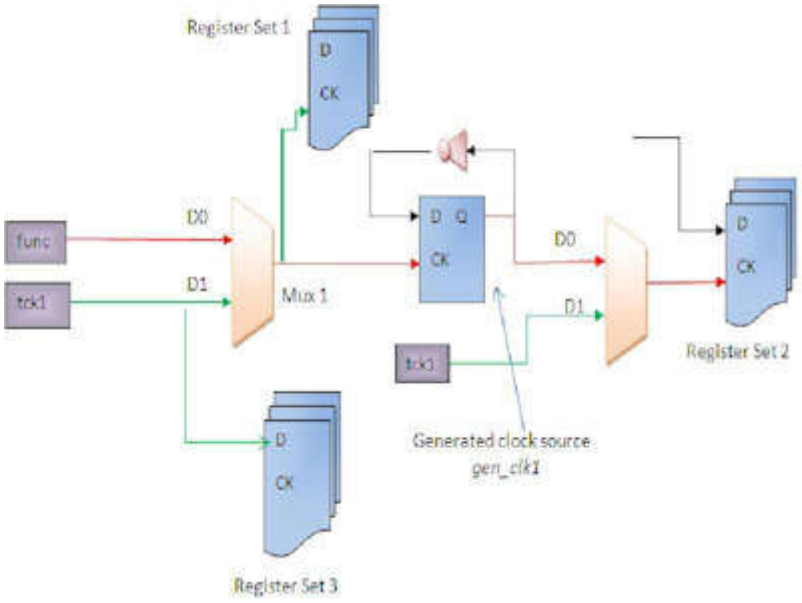

Figure 2: Original design

Test mode CTS: The architecture of the design is such that test clock tck1 to all register sets 1-3 can be built with very low latency. However due to functional mode clocking constraints, as explained above, clock latency for the test clock will be high. The latency for test mode will be constrained by register set 1 as shown in above diagram by green line. This cannot be avoided because of the need to balance register set1 and register set 2 latency in functional mode and buffering can only be done after mux1 output because it is not possible to increase the latency of the generated clock source. The consequence of this is that there is no option other than to increase the latency of register set 2 and register set 3 in test modes. This is a serious problem because the latency of the test clock is increased because of functional mode clocking constraints. As discussed above in robust clock tree guidelines, an increase in clock latency can lead to multiple problems. This problem cannot be solved using advanced features like MMMC (Multi Mode Multi Corner)CTS of current EDA tools.

Solution: The solution to problem lies in cloning the clock logic as shown in figure 3. EDA tools generally do not implement cloning of non-buffer logic in the clock path network. The problem can be solved if there is a separate dedicated clock mux for the generated clock source flop. The limitation of placing clock buffers after the mux output has been removed and for register set 1, the clock buffering to balance latency in functional mode can be done between func clock source and cloned mux input D0. Since there is now the bare minimum buffering done between cloned mux output D0 and register set 1, the latency numbers in test mode are not limited by register set 1 and it is possible to achieve minimum latency in test mode.

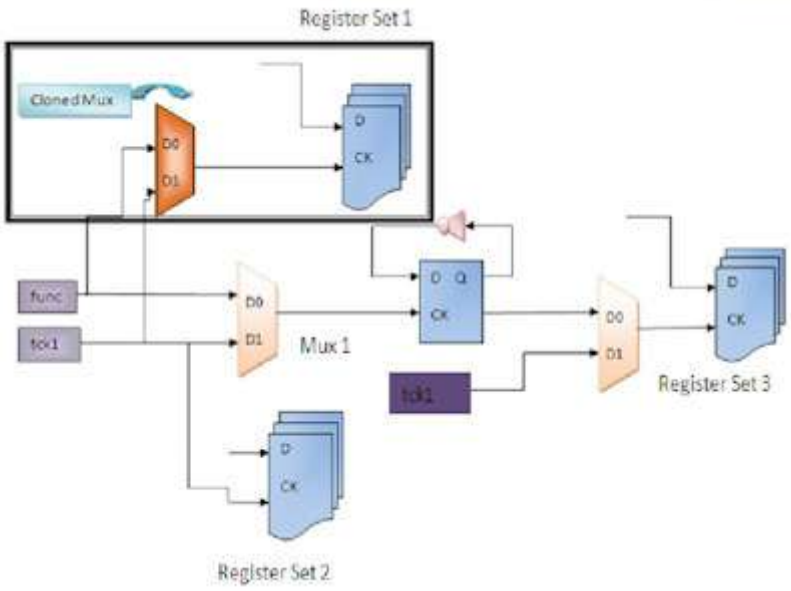

Figure 3: Modified design with cloned mux

Case study 2 - Clock muxing of two synchronous clocks

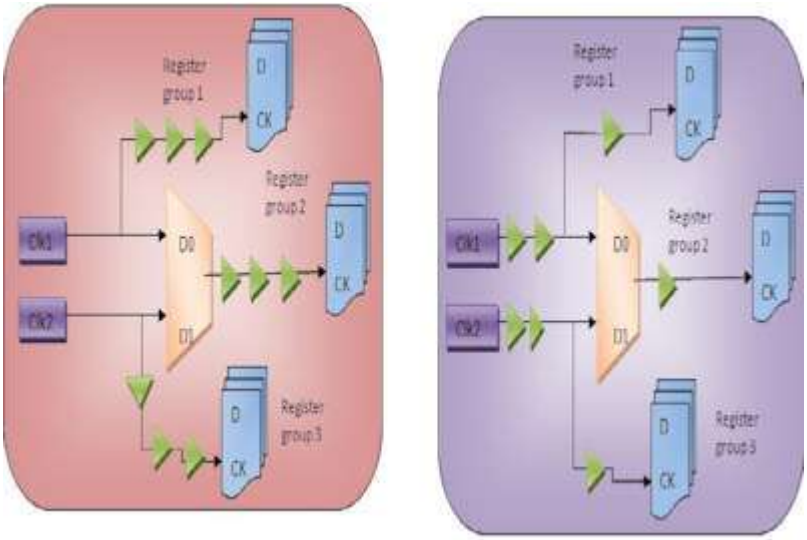

Figure 4(a) Figure 4(b)

Figure 4: Design with two synchronous clocks

In this example clock 1 and clock2 are synchronous to each other. The assumption is that the minimum latency of both clock1 and clock2 is not limited by register group 1-3, but by some other clock group (not shown in diagram).

A typical behavior of most of CTS engines would be to insert clock buffers after the mux output to register group2 in order to save overall clock buffers. However this will introduce a problem of larger uncommon path between register group 1 and register group2 as well as between register group 2 and register group3. A CTS engine is not intelligent enough to understand that the architecture ensures that mux select will not toggle on the fly and there cannot be a case of launch on clock 1 and capture on clock 2 for register group2. An alternate approach for CTS for these type of cases is shown in Figure 4(b). In figure 4(b), the clock buffers have been moved for both clock 1 and clock2 before the clock mux in order to have a greater common clock path between register set 1-2 and register set 2-3. Note that this is under the same assumption that latency of clocks Clk1 and Clk2 is limited by some other register group other than 1-3 and extra clock buffers were placed by

the CTS engine after clock mux to balance skew requirements.

Case study 3 - Centralized vs decentralized clocking scheme

There is debate among designers about how to manage clock muxing and clock divider logic for the SoC. Proponents of a centralized clocking scheme argue that doing all clock muxing at a single place helps managing things in a better way, while opponents question this approach citing timing issues that crop up due to centralized muxing. Both possibilities will be considered.

Assume there are three IPs and one clock of 200 MHz frequency. The design requirement says that both IP1 and IP2 require two synchronous 200 MHz and 100 MHz clocks. Moreover, IP1 and IP2 can handshake data synchronously both at 200 MHz and 100 MHz. Now, there are two options to implement a clocking scheme that meets this requirement. First is to divide the 200 MHz clock to generate the 100 MHz clock inside a centralized clocking module and then provide both 200 MHz and 100 MHz clocks to both IPs. The second option is to divide the 200 MHz clock separately in both IPs. In this scenario, option one is the better option because IP1 and IP2 both need divided clocks and they are exchanging data synchronously as well. If division is done independently in both IPs, there is duplication of the divider logic and there are chances of phase mismatch in the divided clocks and additional logic may be required to solve this problem. In this case, a centralized clocking scheme is better than decentralized one even though there may be some problem of uncommon path between the 200MHz and 100MHz clocks.

For IP3, a decentralized clocking scheme is the best approach. IP3 requires 200MHz, 100MHz and 50 MHz clocks and IP3 is exchanging data only with the external world and not with any other IP within the SoC. In this case there is no point placing the dividers in one centralized clocking block because it will introduce uncommon path between all divided clocks. The better option would be to divide the 200 MHz clock inside IP3 to generate 100MHz and 50MHz clocks.

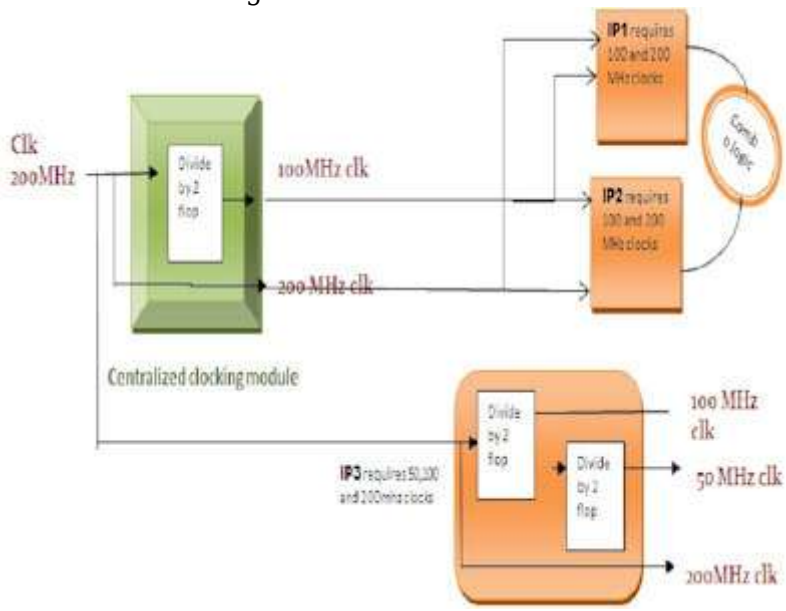

Figure 5: centralized and decentralized clocking

In summary, it might look tempting and convenient to keep all clock muxing and dividing logic in one place, but in some cases it might introduce timing closure problems. The better approach is to analyze the impact of centralized/decentralized clocking on a case to case basis and to take the appropriate decision after that analysis.

Case study 4 - Power vs. timing

A clock tree designer often has to choose between power and timing. One such example is shown in figure 6. Different CTS engines can behave differently. The first CTS tools prefers power saving over
uncommon path because when clock gating is done, the maximum number of clock buffers will stop toggling . The second solution favors timing over power as both register groups now have the minimum uncommon path. A CTS designer must choose their preference between power and timing on a case by case basis. Whatever the tool algorithm, the clock spec can be modified to force the CTS tool to build the required structure.

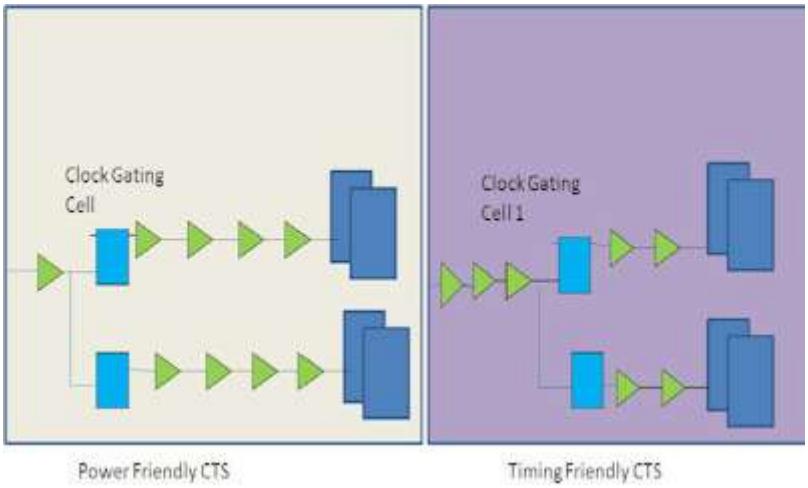

Figure 6: Alternative ways of building a clock tree

Case study 5 – Back-to-back clock gating cells

Many times due to third party IPs and logic synthesis clock gating insertion, back-to-back clock gating cells may have been created. Because of this, clock latency to that register group can increase since a clock gating cell typically has a higher delay than a clock buffer. This can be rectified either by merging these clock gating cells at the RT level or if that is not possible because of integration and third-party IP issues, it can also be done during logic synthesis. Most EDA tools doing logic synthesis have a feature to merge such back-to-back clock gating cells, but the default is not to merge these clock gating cells in order to preserve the RTL implementation . This feature can be used on a case to case basis.

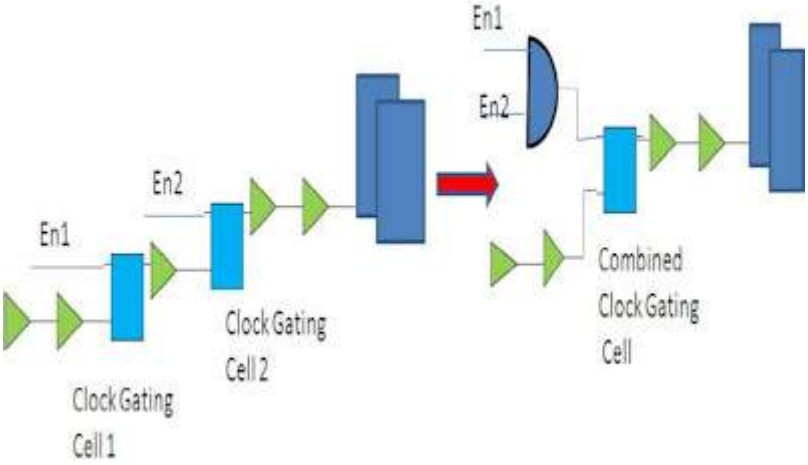

Figure 7: back-to-back clock cells

Recommendations and guidelines/experiments for designing clock trees
For a new design when the clock tree is being constructed for first time, it is important to know optimum latency and skew numbers. Some suggested experiments for this include:

1. Build a clock tree with no skew balancing requirements. This will force the CTS engine to build a clock tree to all registers at its lowest latency possible without caring about skew balancing. The clock path of the register group having the highest latency should be analyzed in detail because when that clock tree is going to be built with skew requirements specified, this register group will determine the latency of that whole clock group. Explore architectural improvement that can be made to reduce latency for this register group. This exercise should be repeated for subsequent highest latency register groups until no further latency improvements can be made.
2. After minimum latency has been established, skew numbers should be targeted. Two or three experiments, with different skew numbers, should be performed to see if overall latency is increasing in order to meet clock skew requirements. Inappropriate clock buffer selection could be an issue. Skew numbers should be double checked. Very low skew numbers might look tempting but too aggressive skew numbers can increase overall clock latency and can increase peak power dissipation due to all flops toggling at the same time.
3. Another suggested way to target uncommon path problem is to compare timing reports between pre CTS stage and post CTS stages. Ideally the timing status of a design should remain the same between pre CTS and post CTS stages because projected deterioration in the timing profile is already taken care of at pre CTS by extra clock skew and derate uncertainty. If timing violation are seen after post CTS stage and a clock tree with respectable skew numbers has been built, the culprit is probably a huge uncommon path between launch and capture registers. Root cause analysis of the uncommon path should be done to determine if architectural improvements can be made to reduce the uncommon path.

Conclusion

The case studies, guidelines and experiments are neither compulsory nor exhaustive enough to cover all aspects of an ideal clock tree. Moreover, there are a lot of other issues such as signal integrity and clock gate ratio which have not been considered. These could be important, particularly in smaller technology nodes. This article should serve as an eye opener to change the perception about how CTS activity is taken generally in our design cycle. With timing margins being reduced, it has become very important to scrutinize the clock tree architecture thoroughly and look for every single possibility of improving clock structure.

About the authors

Amol Agarwal is working with Freescale Semiconductor as Senior Design Engineer and has experience of more than 6 years. He is currently working in physical design team at Freescale with STA & Synthesis as area of specialization. He has been involved in several block-level and chip-level designs in technology ranging from 250nm to 40nm.

Priyanka Garg is working with Freescale semiconductor as a Design Engineer for more than a year. She is currently working in the Physical Design team with Placement and Routing as areas of specialization. She has also successfully formulated various Clock Tree Synthesis strategies at 55 and 40 nm.

If you found this article to be of interest, visit **EDA Designline** where you will find the latest and greatest design, technology, product, and news articles with regard to all aspects of Electronic Design Automation (EDA).

Also, you can obtain a highlights update delivered directly to your inbox by signing up for the EDA Designline weekly newsletter – just **Click Here** to request this newsletter using the Manage Newsletters tab (if you aren't already a member you'll be asked to register, but it's free and painless so don't let that stop you).