Prototyping an ASIC with FPGAs

By Rafey Mahmud, FAE at Synplicity.

With increased capacity of FPGAs and readily available off-the-shelf prototyping boards sporting multiple FPGAs, it has become feasible and affordable to prototype a multi-million gate ASIC by mapping the original RTL code to several FPGAs on a board. There are many off-the-shelf boards available with the right IO interfaces for verifying the functionality of an ASIC. Also, most common high speed IO standards are available on today's FPGAs. Using an FPGA prototype, you can exercise more product function in hours than what you can in days using simulation and a lot of stimulus does not even have to be developed as you can use real hardware to produce real time stimulus. One cannot afford to re-spin and ASIC due to a functional bug not caught early on due to lack of proper verification as the ASIC mask costs are exorbitantly high. Since actual software can be run on a prototype hardware system, overall system development cost can be reduced significantly by co-developing hardware and software.

Most ASICs are too large to fit on a signle FPGA therefore the challenge is partitioning the ASIC design into several FPGAs. Luckily there are EDA tools available to auto partition. In this article I am going to discuss the challenges faced when partitioning and mapping an ASIC RTL to multiple FPGAs and provide some solutions.

Gated clock conversion:

One of the major challenges in targeting an ASIC RTL to an FPGA is the prevalent use of gated clocks in most ASIC designs. FPGAs have pre-synthesized clock trees for providing synchronized clock to the finite number of flip-flops and memories across the chip. When additional logic is applied to the clock signal in the RTL, the logic translates into physical gates outside the pre-synthesized balanced clock tree, through which the clock signal passes, thus causing large clock skews between registers of the FPGA. There are many forms of clock gating. We are going to discuss many possible sources of clock gating and how RTL changes can be made to eliminate clock gating without changing the functionality of the original design.

The simplest and most common form of clock gating is when a logical "AND" function is used to selectively disable the clock by a control signal. This is commonly employed in ASIC designs for saving dynamic power consumption by selectively stopping the high capacitance clock tree from switching.

gated_clock = **CLK** && **Enable**

As shown in Figure 1, the **gated_clock** signal acts as the source clock, **CLK**, when enable is "1" but is forced to "0" when enable is "0". When clock of a flip-flop is "0" it

retains the last captured data. This function can also be implemented by feeding the **CLK** signal directly to the flip-flop's clock pin and using the enable control signal to selectively latch the input data in the flip-flop instead. So when the **Enable** is '0' output of the flip-flop is fed back into the input thus retaining its state. Most flip-flops have an enable pin, **En**, which is implemented by using a multiplexer selecting between input data and output of the flip-flop as shown in figure 1. Even if the flip-flop does not have an **En** pin, it can be implemented by inserting a multiplexer in the data input path of the flip-flop.

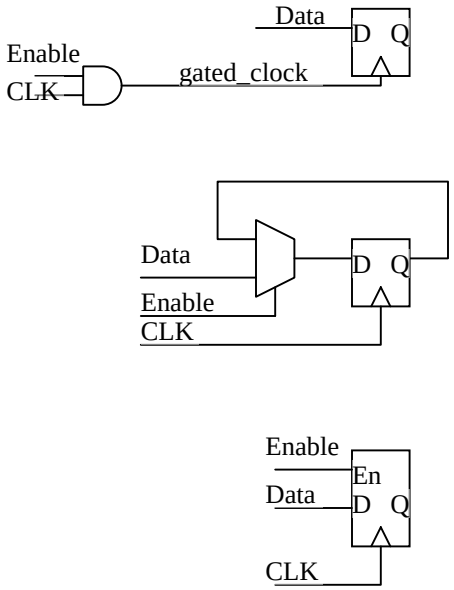


Figure 1. Conversion of combinatorial gated clock

Another common type of clock gating in ASICs is due to multiplexing of two clock signals of different frequencies as shown in Figure 2. The output of the multiplexed clocks, **gated_clock**, then clocks a range of flip-flops. If the two source clocks are harmonically related then the fast clock can be used to clock the flip-flops directly, while the slower clock can be used to selectively enable the flip-flops' data input to avoid clock gating of such type. If the flip-flops are enabled every two cycles, it is essentially same as their clock being half the frequency.

This scenario is shown in figure 2 where there are two inputs, **CLK**, and **CLKdiv2** of the multiplexer. **CLKdiv2** is the half frequency clock derived from the **CLK** clock. The **SEL**

signal is used to switch the clock from one source to another. Since the multiplexer is in the clock path it causes extra skew in the FPGA implementation.

Figure 2 also shows how to avoid this type of clock gating by a different implementation when the flip-flops are substituted by flip-flops with enable pins and the enable control is driven by the divided clock while the **CLK** clock is clocking the flip-flops directly. There is a multiplexer in the enable control path which keeps the flip-flops enabled all the time when the **SEL** is '1' and every other clock cycle when **SEL** is '0', essentially switching the clock frequency of the flip-flops.

Figure 3 shows the timing diagram to verify that both implementations produce the same result, one with gated clock and the other with non-gated clock. The timing diagram shows that **CLKdiv2inv** signal (inverted **CLKdiv2** clock) becomes a '1' at every second positive edge of **CLK**, which is equivalent to latching data and every positive clock edge of **CLKdiv2**. The inverted divided clock has been chosen to enable the flip-flops instead of the divided clock itself to match the first capture edge with the original implementation.

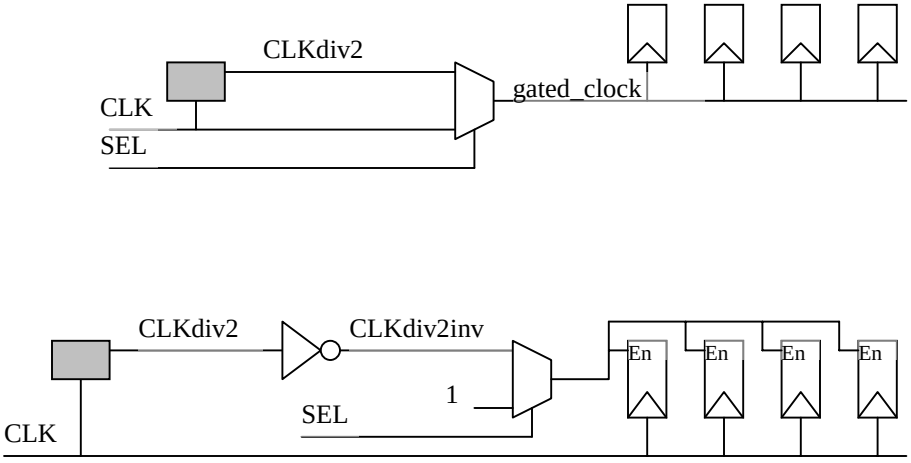


Figure 2. Conversion of sequential gated clock.

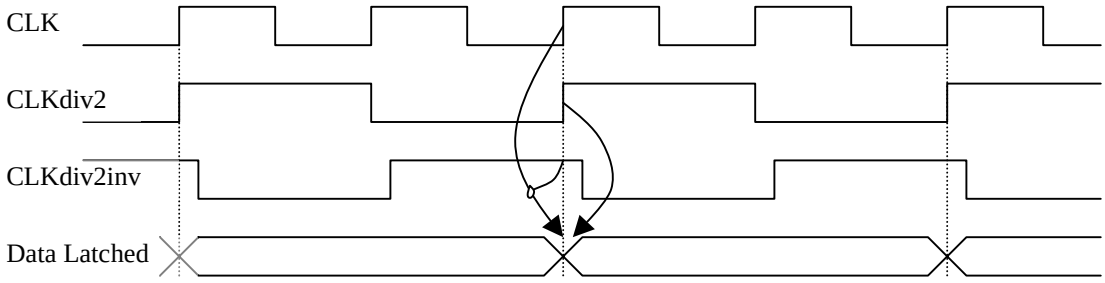


Figure 3. Timing diagram of converted sequential gated-clock

If the two clocks are totally unrelated then it is not possible to use the fast clock as the base clock and qualify the data by the slow clock as they are asynchronous to each other. In such a case, if the total number of flip-flops is not very high, flip-flops can be duplicated and clocked directly by each un-related clock separately. Outputs of these flip-flops can then be multiplexed instead of multiplexing the clock sources.

Another common scenario is when exclusive “OR” logic is used to selectively invert the source clock. This kind of clock gating can be removed by having a duplicate set of flip-flops, one set clocked by the true clock and other with the invert clock. Outputs of the two sets can be selected through a multiplexer.

There are other ways of dividing clock frequency in an FPGA by utilizing the built in PLLs without causing any clock skew.

Gate count estimation.

It is very difficult to estimate during partitioning how much RTL code is going to fit into an FPGA comfortably. Not only gate count estimate is required but routing area also needs to be figured in. An automatic synthesis tool is a must at this stage to get a quick estimate of the fitting as different partitions are experimented with. Off course IO pin count of each FPGA and number of available traces on the board have to be looked at in conjunction with the logic fitting so it is an iterative process.

Instantiated memories and other RTL changes

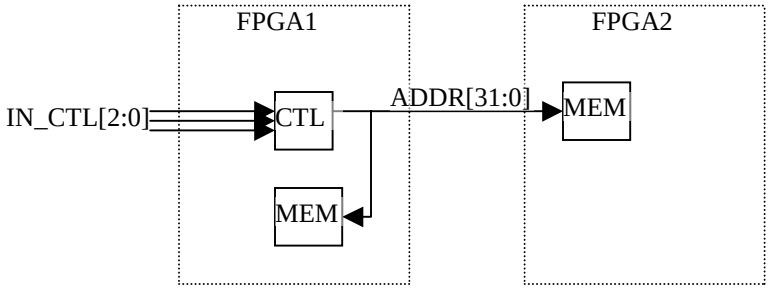
Internal memories in ASIC RTL are instantiated often times with addition BIST logic. There are other instantiated library components in most ASICs which will not be understood by FPGA P&R tools. Due to this either minor RTL changes have to be made to infer logic rather than instantiating or special library has to be included in the project which describes those instantiated blocks. At times blocks of RTL are commented out for the logic that is not relevant to the prototype, like a BIST controller. Commenting out chunks of logic from RTL leaves un-driven inputs hanging in the rest of the RTL which have to be tied to constants.

Limited number of FPGA IO pins and board traces.

IO pins of each FPGA are limited in number. The total number of board traces in-between FPGAs are also limited and predefined for an off-the-shelf board. So partitioning is a three step iterative process. Step one is to divide the logic into how ever many FPGAs you have on your prototype board such the logic fits in each FPGA. Second step is the adjustment of logic such that IO signals of each FPGA, due to inter-FPGA interconnects, do not exceed the total IO pin count for that FPGA. Third step is adjustment of logic so that total number of board traces between FPGAs should be enough to accommodate all the inter-FPGA signals. Since step one has direct impact on step two and three, it becomes an iterative process to meet all three requirements by trial and error, either manually or through a tool. The following two sections describe ways to reduce the total number of inter-FPGA signals for successfully completing step two and step three.

Logic replication

Since the goal is to minimize the use of IOs for each FPGA and inter-FPGA signals, sometimes it is very useful to simply replicate chunks of logic in each FPGA thus paying an area penalty but reducing the number of inter-FPGA signals. For example, in Figure 3, a control block, CTL, in FPGA1 is generating a 32 bit address, ADDR[31:0], for a memory in FPGA1 and also a memory in FPGA2. There are only 3 input signals going into the control block. By replicating the control block, CTL, in FPGA2, we can avoid the ADDR[31:0] bus connection between FPGA1 and FPGA2 and add 3 signals instead, IN_CTL[2:0], between FPGA1 and FPGA2. There is a net reduction of 29 signals between the two FPGAs. Off course, replication of logic utilizes more area but if the limitation is IO count or the number of traces available between two FPGAs then it helps in achieving a successful partition.



CTL block in FPGA1 is generating 32 ADDR[31:0] signals going to FPGA2

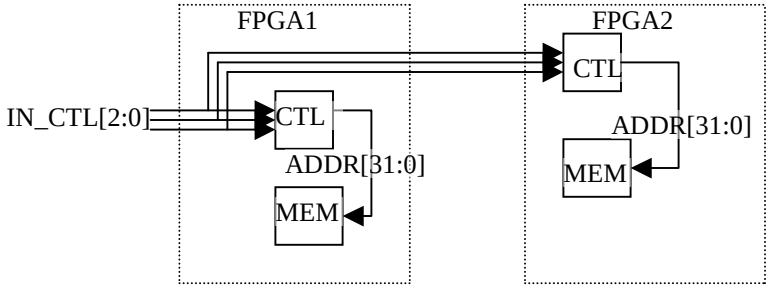


Figure 4. By duplicating the CTL block in FPGA2, instead of 32 ADDR[31:0] signals, only 3 IN_CTL[2:0] signals are going from FPGA1 to FPGA2

IO multiplexing.

Due to limited number of traces and IO pins, inter-FPGA signals can be multiplexed over time in one system clock cycle, through the same IO pin and trace. For example as shown in figure 5, four output signals A, B, C, and D are multiplexed and driven on the same output pin in FPGA1. A clock, X4CLK, which is four times faster than the system clock, is used to selectively drive the four signals in separate time slots of one system clock cycle, and then de-multiplexed on the FPGA2 side in the same fashion. As far as the system clock is concerned, all four signals passed through during one clock cycle. The

overhead of sharing the same pin for more than one signal is the multiplexing and de-multiplexing logic and the control logic to synchronize both sides. Also, a fast clock is needed to ensure that all the signals reach the destination in one system clock cycle. Multiplexing signals in this fashion also has an adverse affect on the maximum frequency a prototype system can run at as each signal's setup time requirement is tightened by a factor of however many signals are multiplexed.

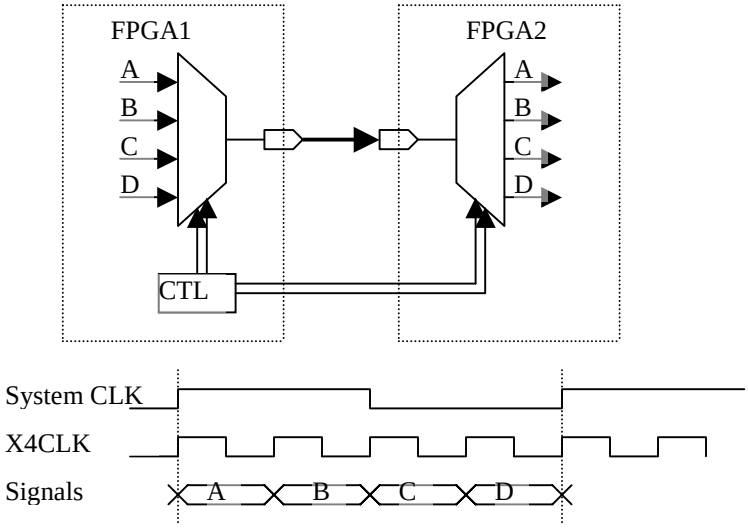


Figure 5. IO multiplexing to reduce inter-FPGA signals

System frequency and design constraints.

System frequency often hinges on the multiplexed inter-FPGA signals as they have the tightest setup time requirements due to the fast clock and high board trace delays. Inter-FPGA signals are generally the slowest timing paths in a prototype and determine the highest system clock frequency of the prototype. The original ASIC usually runs at a higher clock frequency than the FPGA implementation, therefore most synthesis and timing constraints for an ASIC RTL have to be re-specified for the FPGAs.

Prototyping and verifying an ASIC with FPGA implementation provides high level of confidence in the functionality of the design before masks are generated and reduces the total cost of ownership considerably. There are some inherent differences in the way RTL is developed for ASICs and FPGAs and it is often difficult to partition the RTL design with limited IO pins of each FPGA and limited number of board traces. With some minor modifications to the RTL code such as gated-clock conversion, possibly automatically using EDA tools, and partitioning the design keeping the inter-FPGA signals to a

minimum by multiplexing these signals and trying different partitions, most ASICs can be prototyped using FPGAs quickly and affordably.