

# Unique Graph-Based Physical Synthesis Technology

## For Fast Timing Closure and Improved Performance of FPGA Designs



Synplicity®

Simply Better Results

### Introduction

Traditional synthesis technology is failing to address the needs of today's extremely large and complex FPGA designs implemented in devices at the 90 nm technology node and below. The problem is that conventional FPGA synthesis engines are based on ASIC-derived techniques such as floorplanning, in-place optimization (IPO), and — more recently — physically aware synthesis. However, these ASIC-derived synthesis algorithms are not appropriate for use with the regular architectures and pre-defined routing resources presented by FPGAs.

The end result is that all three of the traditional FPGA synthesis approaches require multiple time-consuming iterations between front-end synthesis and downstream place-and-route tools so as to achieve convergence and timing closure. The solution is a unique graph-based physical synthesis technology that provides a single-pass, push-button synthesis step requiring zero (or very few) iterations with the downstream place-and-route engines.

Furthermore, graph-based physical synthesis can provide 5 to 20% performance improvement in terms of the overall clock speed of the system. The first product to feature graph-based physical synthesis technology is Synplify® Premier advanced FPGA physical synthesis, which is tailored for high-end FPGA designers whose designs are of a complexity that mandates a true physical synthesis solution.

This paper first presents the main conventional synthesis approaches and explains the problems associated with these techniques. The paper then introduces the concepts underlying graph-based physical synthesis and shows how this technology uniquely addresses the requirements of today's state-of-the-art FPGA architectures.

### Problems with Conventional Synthesis Solutions

In the case of the early ASIC technology nodes of 2 microns and above circa the early 1980s, the delays associated with cells (logic gates) dominated over the delays associated with the interconnect (wires) by a ratio of around 80:20. That is, gate delays accounted for approximately 80% of each delay path. This allowed designers to estimate interconnect delays using simple wire-load models in which each logic gate input was assigned some "unit load" value and the delay associated with a particular route was calculated as a function of the strength of the driving gate and the total capacitive loading on the wire.

Similarly, when the first RTL synthesis tools were introduced for use with ASIC designs in the late 1980s — around the time the 1 micron technology node was introduced — cell delays continued to dominate wire delays by a ratio of around 66:34. Thus, early ASIC synthesis tools also based their delay estimations and optimizations on the use of simple wire-load models. Due to the fact that cell delays were so dominant, the wire-load-based timing estimations used by the original synthesis engines were sufficiently accurate such that downstream place-and-route engines could usually implement the design with relatively few iterations back to the RTL and synthesis stages.

With the introduction of each successive technology node, however, the delays associated with the interconnect became increasingly significant (in fact, the cell versus interconnect delay ratio is now around 20:80 with respect to standard cell ASICs created at the 90 nm technology node circa 2005). This causes delay estimations in the synthesis engine to become less and less correlated with the real-world delays following place-and-route.

This has some important implications, because synthesis engines select between different optimization techniques and alternative methods of realizing functions such as adders based on their timing predictions. For example, suppose that a particular timing path containing — among other things — an adder is predicted to have some amount of positive slack. In this case, the synthesis tool may select a slower version of the adder that consumes relatively little area on the chip. But if the timing estimations are not well correlated to the real-world, post-place-and-route delays, this path may end up being too slow. Thus, inaccurate delay estimations mean that the synthesis engine ends up optimizing the wrong things and it's only when you get through place-and-route that you discover that the problems are not where you [or the synthesis engine] thought they were going to be. The result is that the effort required to achieve timing closure is dramatically increased as are the number of time-consuming front-end to back-end iterations.

In order to address these issues, it is necessary to account for the physical characteristics associated with the design during the synthesis process. Thus, over time, ASIC synthesis technologies (followed by their FPGA synthesis cousins) have adopted — and, in some cases, discarded — a series of techniques such as floorplanning, in-place optimization (IPO), and physically-aware synthesis as discussed below.

### Floorplanning

In the case of RTL synthesis for ASICs, the floorplanning technique arrived in the early 1990s soon after the synthesis technology itself. Floorplanning tools allow designers to define physical regions (“rooms”) on the device, to place these regions by hand or to use auto-interactive techniques, and to assign different portions of the design to the regions.

Floorplanning involves synthesizing and optimizing the design on a block-by-block basis and then stitching everything together at the end (the synthesis algorithms used with early floorplanning tools were based on the use of wire-load models). This means that the floorplanning tools cannot optimize the logic per se, but can only affect the placement of blocks of logic. Furthermore, the floorplanning tools do not, by definition, consider routing resources globally and it is not possible to accurately analyze all of the timing paths until after the design has been fully routed. This can result in large numbers of time-consuming iterations between the front-end and back-end tools. Although this technique can improve timing and reduce power consumption in the ASIC domain, it requires sophisticated analysis of the design and a high level of expertise.

In the early days, ASIC floorplanning was used for several reasons: as a means of achieving timing closure, to solve the problem of limited capacity, and to support incremental changes on a block-by-block basis. More recently, floorplanning is not perceived as a way of achieving timing closure on its own; floorplanning remains a useful technique, but only when combined with other approaches such as physical optimization, and floorplanning using post-synthesis gate-level netlists still requires a lot of designer expertise.

In the case of FPGAs, floorplanning technology didn't start to enjoy any level of mainstream usage until the late 1990s. On average, critical paths in an FPGA design typically end up passing through three regions (“rooms”) in the floorplan. Due to the way in which FPGAs are typically designed, if floorplanning is performed using post-synthesis (“gate-level”) netlists, subsequently making even relatively small changes to the RTL can cause large portions of earlier floorplanning effort to be lost. The solution to this problem is to perform floorplanning at the RTL level. Once again, however, in order to be useful this has to be combined with some form of physical optimization, and ASIC-derived physically-aware synthesis algorithms are not appropriate for use with the regular architectures and pre-defined routing resources presented by FPGAs.

## In-Place Optimization

As floorplanning began to run out of steam in the ASIC world, it was augmented and/or replaced in the mid-1990s by a technique known as in-place optimization (IPO). Once again, this involved synthesis whose timing analysis and estimations were based on the use of wire-load models.

In this case, the ensuing netlists are passed to the downstream place-and-route engines. Following place-and-route and parasitic extraction, real-world delays are back-annotated into the synthesis domain. These new values trigger incremental optimizations in the synthesis engine such as logic restructuring and replication. The result is a new netlist that has been partially modified. This netlist is then handed over to incremental place-and-route engines which generate a modified design topology.

The end results from an IPO-based flow are typically better than those achieved using a floorplanning approach. Once again, however, this technique can require numerous time-consuming iterations between the front-end and back-end tools. Furthermore, a significant problem with an IPO-based technique is that the modifications to the placement and routing can result in new critical paths that were not seen in the previous iteration. That is, fixing one problem can instigate other problems, which can result in convergence issues.

In the case of FPGA designs, IPO-based design flows only started to receive mainstream attention circa 2003. Having said this, even after such flows had become available, they were not adopted in any meaningful way, because the IPO technique of optimizing timing paths on an individual basis often leads to degradation of other paths and incomplete timing closure. Designers need reliable results that allow them to make changes to the design without losing what they have achieved in earlier versions. But the IPO-based technique does not produce stable results over multiple design iterations because optimizing critical paths in one iteration can create new critical paths in the next. Similarly, adding constraints to improve timing in one area can worsen it in other areas.

## Physically-aware Synthesis

The current state-of-the art in ASIC synthesis technology is physically-aware synthesis, which started to receive mainstream attention circa 2000. Irrespective of the actual technique (there are several different algorithms), the underlying concept behind physically-aware synthesis is to combine placement and synthesis in a single pass.

This works well in the ASIC domain, because the placement-aware synthesis engine can base its timing estimations on the proximity of the placed cells and on the ensuing Steiner and Manhattan routing estimations. The reason this works so well in the ASIC world is that wires are built-to-order. This means that the delays associated with the final placed-and-routed design tend to correlate reasonably well to those estimated by the synthesis engine.

Starting around the 2002-2003 timeframe, a number of vendors started to consider ASIC-derived physically-aware synthesis technology to FPGA designs, but they never pursued it further and — with the exception of the Synopsys Synplicity Business Group's new graph-based synthesis approach — no vendor currently offers placement-aware RTL synthesis for FPGAs. The problem is that, unlike wires in the ASIC world that are "built-to-order," an FPGA has a fixed amount of pre-defined routing resources, and not all of these routes are created equal (some are short and fast, some are long and fast, some are short and slow, and some are long and slow).

One way to think of this is that working with an ASIC is like living in an undeveloped area (such as a desert) with a large budget. In this case you can build new roads as you need them and — generally speaking — each road can be as fast as you wish. Thus, if you wished to construct a path from your home to the office, this path could be implemented as a super-fast highway. By comparison, working with an FPGA is similar to living in the middle of a large, well-established city, whose “interconnects” are a mixture of small back streets, medium-sized roads, and really fast highways. In that case, when you are returning home from the office, you don’t necessarily aim at driving the shortest distance. Instead, you may commence your journey by heading the “wrong way” until you can get on a highway, at which point you may reverse your direction and pass the office again as you head toward your home.

What this means in real terms is that ASIC-based physically-aware synthesis can base its routing and timing estimations on the proximity of the placed cells forming the design. In the case of FPGAs, however, having two logical functions in close proximity to each other does not necessarily imply the availability of a fast connection between them. Although this is somewhat non-intuitive — and depending on the available routing resources — one may achieve better routing and timing results by placing the logic functions further apart. This is why ASIC-derived physically-aware synthesis technologies return less-than-optimal results when applied to FPGA architectures. In turn, design flows using these technologies continue to require large numbers of time-consuming iterations between the front-end (synthesis) and back-end (place-and-route) engines in order to achieve correlation and timing closure. If only there were some other way.

### **Some Considerations Associated with FPGA Architectures**

Before we introduce the concept of graph-based physical synthesis in detail, it is important to have some feel for the complexity of the task at hand. As was previously noted, an FPGA has fixed connectivity resources; all of the wires in the FPGA have already been created, but not all routes are created equal (there are short, medium, and long wires, each of which may have fast, medium, or slow characteristics).

Adding to the complexity is the fact that routes may have multiple “tap points” (like exit ramps from a highway). The problem here is that you may have a fast route that can quickly convey a signal from a source function (at the initial entry ramp to the highway) to a destination function (located at the last exit ramp on the highway). However, if we add a second destination function to an internal tap point, this can slow the signal down dramatically.

Furthermore, the majority of today’s FPGA architectures are based on the concept of a look-up table (LUT) with multiple inputs and a single output. Some FPGA architectures have different delays through the LUT associated with each input-to-output path. More importantly, however, is that fact that each input to the LUT may have access to only a subset of the different wire types. In our city analogy, this is like having a house (LUT) with several drives (input pins), where one drive leads to a small side street (a slow wire) while another is connected to a highway (a fast wire). What this means is that if the output from one LUT is driving another LUT, there may be both slow and fast paths between them depending on the specific input we use on the receiving LUT.

The whole situation is further confused by the hierarchical nature of FPGA architectures. For example, there may be several LUTs in a small logic block; several of these small logic blocks in a larger logic block; and a tremendous number of these larger logic blocks presented across the face of the FPGA. Within one of these larger logic blocks there may be only a few direct connection possibilities between the output of one LUT and the input of another LUT; in order to achieve additional connections we may have to go outside the logic block and then come back in again. Once again, this serves to illustrate the fact that we are dealing with a very complex problem:

although this may seem counterintuitive, if you know where to put them and which pins to use, placing two objects/instances in different logic blocks can return better delays than putting the objects/instances in the same logic block while using less optimal interconnect resources.

Last but not least, any proposed solution has to account for the extra delays associated with routing wires around fixed hard macro resources such as blocks of RAM, multipliers, and so forth. Similarly, the solution has to account for the increased routing congestion that will be found close to these hard macros. These hard macros are all device-specific, so any proposed solution must be able to handle every device in every FPGA family.

### Unique Graph-Based Physical Synthesis Solution

The solution to creating a physically-aware synthesis solution that can truly handle the complexities associated with FPGA architectures is to approach the problem from a radically different viewpoint. The way this works is to characterize all of the wires in the FPGA — including entry points, end points, and internal exit points, and to then build a “map” of all of these wires. In the software world this type of map is referred to as a Graph; hence the reason why this technique is known as “Graph-Based Physical Synthesis.”

In addition to the wires themselves, this map also includes details as to which LUT pins have access to which types of wire, any differences in input-to-output delays through each LUT, and the size and locations of any hard macros in the device. Returning to our analogy for a moment, this is like having access to a map of a city showing the streets and highways and features such as parks (hard macros) that you will have to drive around. When wishing to travel between two locations in the city, you would use the map to select the fastest route, which will often not be the shortest point-to-point path.

Similarly, instead of looking for proximity, the graph-based physical synthesis engine focuses on speed using an interconnect-centric approach. Starting with the most critical paths and working its way down to the least critical paths (thereby ensuring that the fastest routes are available for the most critical paths), the graph-based physical synthesis engine will select wires and their associated entry points and exit points; from these wires it will derive placements; from the wires and placements it will derive accurate delays; and it will then optimize and iterate as required.

The point is that all of this optimization and iteration is performed in the front-end (synthesis) portion of the flow. The output from the graph-based physical synthesis is a fully placed netlist (including the specific LUT pins that are to be associated with each wire) that can be handed over to the FPGA’s back-end place-and-route engines.

The end result is a single-pass, push-button synthesis step requiring zero (or very few) iterations with the downstream place-and-route engines. Furthermore, based on the analysis of more than 200 real-world designs, it has been shown that graph-based physical synthesis can provide 5 to 20% performance improvement in terms of the overall clock speed of the system.

### Summary

There is an increasingly serious timing closure problem with regard to high-performance, high-complexity FPGAs implemented at the 90 nm technology node and lower. The solution is a unique graph-based physical synthesis approach. (Note that graph-based physical synthesis is also ideally suited to the regular structures found in Structured ASIC architectures; especially those featuring a small number of configurable routing layers.)

Unlike the ASIC-centric physically-aware synthesis world in which wires are derived from placement selections, it is the placements that are derived from the wire selections when using a graph-based physical synthesis approach in the FPGA domain.

With regard to the timing closure issue, flows based on existing (ASIC-derived) physical synthesis engines may require many time-consuming iterations between the front-end (synthesis) and back-end (place-and-route) portions of the flow. And, after all of this, they may still fail to converge on a solution. By comparison, an analysis of the graph-based physical synthesis results from more than 200 designs shows that 90% are within 10% of the final real-world timing and 80% of these designs are within 5% (by comparison, only 30% of designs using logic synthesis are within 5% of the real-world timing values and many designs can easily be in error by 30% or more). Furthermore, graph-based physical synthesis can provide 5 to 20% performance improvement in terms of the overall clock speed of these designs.

In addition to the fact that graph-based physical synthesis requires zero (or very few) iterations with the downstream place-and-route engines, the quality of the placed netlist is dramatically improved; this means that the timing-driven router has less work and optimizations to perform and thus it runs significantly faster.

Once one has been exposed to the concept, graph-based physical synthesis seems intuitively obvious. In reality, however, this is an extremely complex problem. Several EDA companies have considered the problem of combining placement and routing in a single step, but the general consensus has been that this simply was not possible to achieve an effective solution to this problem with the computational resources available today.

Synplicity's breakthrough is the concept of basing everything on a routing-centric approach, of representing everything as a graph, and then of solving that graph. After expending many engineer-years of research and development, Synplicity's synthesis experts have created a true graph-based physical synthesis solution. The first product to feature graph-based physical synthesis is the Synplify Premier, advanced FPGA physical synthesis, which is tailored for high-end FPGA designers whose designs are of a complexity that mandates a true physical synthesis solution (the Synplify Premier tool also includes sophisticated features such as RTL source-level debug and support for Synopsys® DesignWare® for ASIC designers who are prototyping an ASIC — or portion thereof — using a single FPGA).

