COMPUTER DESIGN

Embedded synthesis technology enhances high-density FPGA tools

Over the next few years, the FPGA industry will move toward a new breed of tools that works from within the synthesis process. Without embedded synthesis, it will be harder for designers to get the productivity and performance expected from the latest ultra-high-density programmable devices.

Ken McElvain, founder and CTO, Synplicity

When we read about the emerging deep-submicron crisis, we naturally think about the effect on complex ASICs. The deep-submicron crisis, however, has already arrived in programmable logic. Ever-improving IC manufacturing technology continues to drive programmable logic into new regions of complexity and performance. Programmable logic can now deliver more functionality and performance than the most complicated ASICs of only a few years ago.

The complexity of new programmable devices, coupled with the routing delays inherent in programmable logic, means that FPGA designers feel the crisis in the synthesis, placement, and routing flow. The size of the new devices leads to significantly longer run times, and less predictability of timing before the place-and-route stage. Run times are increasing faster than gate counts, and iteration after iteration may each show a new critical path, frustrating your designers' attempts to outsmart the tools. More iterations and longer run times lead to delays in design completion, and sabotage the fundamental value of programmable logic: faster design cycles.

These new design problems drive the creation of new tools and the reinvention of old ones. No longer can we assume that the steps in the design flow are independent. The high-level information in the HDL source can be useful in the placement and routing stages, while the detailed information from the placement can be used to improve logic optimization. One path to an improved flow is to use embedded logic synthesis to bring realistic guidance to tasks, such as floorplanning and partitioning.

The critical design tasks for high-complexity programmable devices demand a paradigm shift. New technology is required for design partitioning, synthesis, floorplanning, placement, and routing. Recently, dramatic advances have been made in synthesis algorithms for FPGAs, and it's through these new algorithms that a new class of tools has emerged. Over the next few years, the FPGA industry will move toward this new breed of tools that works from within the synthesis process. Without embedded synthesis, it will be harder for designers to get the productivity and performance expected from the latest ultra-high-density programmable devices.

Design flow review

The level of abstraction and automation in FPGA design tools has moved upward to keep pace with the new high-complexity and high-performance devices introduced by FPGA vendors. To achieve the level of productivity necessary to deliver products in fast-paced electronics markets, high-level design languages (HDLs) have become the preferred way to describe complex FPGAs, and logic synthesis is the preferred way to implement them. A typical HDL-based synthesis flow follows these steps:

- Write HDL source;
- Compile to the RTL structure;
- Logic optimization;
- Mapping to a specific technology;
- Timing optimization;
- Placement;
- Routing.

The HDL source describes the device's function. The compilation step interprets the HDL in terms of registers, logic and arithmetic operations, and control circuits. Logic optimization attempts to reduce the complexity of the implementation using sophisticated algorithms. To this point, the design has remained independent of the implementation device architecture (ASIC library or FPGA device family). Technology mapping represents the design in specific device architectures. Timing optimization attempts to improve the design to meet your timing constraints. Placement locates each technology's cell within the chip, and finally, routing allocates the wiring resources to provide the necessary interconnections.

The naïve assumption that each of these steps is independent leads to the non-convergent iteration loops described earlier. At each stage, the tools (or you in the case of HDL source) make implementation decisions based on estimates of what will happen in the rest of the flow. Each following stage is stuck with the decisions made at earlier stages. In the independent model, the placement stage can't change logic to make its job easier. The limitation to this flow is that changes made in early stages have the most impact, but the ability to estimate is weakest there. The later stages have the best information, but the ability to make significant changes is gone.

To overcome the limitations of the flow, we must improve estimation for the early stages, and preserve high-level information for the later stages. If improved timing estimation is necessary to break the endless iteration cycle, then we must first

Reprinted from the August 1998 edition of **COMPUTER DESIGN SUPPLEMENT**
Copyright 1998 by PennWell

understand the components of timing, and what optimization tools can do to change the timing. With FPGAs of 100 kgates or more, good estimation requires an understanding of the device's underlying architecture.

The speed of an FPGA design depends on many factors:

- Delay in the simplest programmable logic cell.
- Delay introduced by the programmable routing.
- Interconnection of logic to form a critical path through the circuit.
- Timing requirements of the external circuitry.

Timing estimation problem

The delay in the simplest programmable cell is easy to estimate once the design has been mapped to a particular device architecture, but very difficult to estimate before mapping. The cell is intentionally very rich in its ability to represent logic functions, but its capabilities vary among FPGA vendors, and even between families for a given vendor. Good estimation requires mapping to the chosen device architecture.

In a typical FPGA, the sum of the routing delays along the critical path comprises well over half the total delay. In a mapped design, routing delay is the most important estimation. The delay due to a particular interconnection depends on the interconnection's capacitance and resistance, which are in turn dependent on the connections' physical location, the routing architecture for the programmable device, and the size of the particular device. Good estimation of routing delays depends on good estimates of physical placement, and on a detailed knowledge of the device architecture.

Static timing analysis algorithms for finding the most critical path and the techniques for considering the external timing requirements are complex, but well understood. Given a mapped design with good routing estimates, static timing analysis will reliably identify the most critical path.

Timing optimization tools have only a few options to improve the timing: Change the circuit topology of the critical path so that the new critical path has fewer delays, change the placement so that it's easier to route with less resistance and capacitance, or change the routing to reduce resistance and capacitance.

Without a link between synthesis and physical implementation technology, there's little hope for putting together a reliable and predictable methodology for designing high-density devices.

Floorplanning perks

Floorplanning puts location constraints on the logic of a design to improve routing estimation and to improve place-and-route run times.

A typical floorplanner divides the circuit area into rectangular regions (called blocks), and then assigns logic to reside in a block. Partitioning has two effects: the estimation error for the location of a cell is reduced from the size of the chip to the size of the block, and the placement and routing runs faster because it's been reduced from one very large problem into a series of much simpler problems.

If we put floorplanning after synthesis (after mapping), then we can improve the placement and routing stages, but logic optimization can't occur. Also, your understanding of the design has deteriorated because most of the contextual information from the HDL has been hidden within the design's programmable logic cells, and the level of detail has increased dramatically. You can't easily make decisions about which cells belong together.

> **Embedded synthesis algorithms must be fast, even when handling huge blocks of HDL, or they won't be practical for real design use. The goal is to provide real-time feedback.**

If we put it before synthesis (at the HDL source level), then you can understand the design tradeoffs, and the placement information can be used by the synthesis tool to make logic optimization decisions. Unfortunately, you can't easily know if you've overflowed the capacity of a block, or which logic has the most critical timing impact. In addition, the design's granularity matches your hierarchy that prevents manipulation of lower level functions, such as counters, adders, state machines, etc.

The best answer is to put floorplanning in the middle of the synthesis flow, after compilation, where you can still understand the design in terms of the HDL source, and then to embed high-performance logic synthesis algorithms within the floorplanner, providing feedback on block capacity and critical paths.

Embedded synthesis means sharing fundamental optimization and mapping technology across traditional design boundaries, such as logic capture, optimization, and place-and-route. Embedded synthesis allows logic to be restructured based upon physical constraints, such as those created during floorplanning or other design tasks that have an effect on the device's actual layout.

Embedded synthesis algorithms must be fast, even when handling huge blocks of HDL, or they won't be practical for real design use. The goal is to provide real-time feedback.

Performance and capacity of traditional synthesis technology are major hurdles to overcome before embedded synthesis can support a new class of tools. Waiting several hours for synthesis results won't allow an interactive design process. Synthesis must run in a few seconds, or at most, minutes, to let you see results, change the design, and then see the new results immediately.

Conquering complexities

Most of today's synthesis tools can't deliver results in minutes because they're based on 12-year-old technology that has been incrementally modified over time for better optimization and chip performance. The fundamental algorithms used in these synthesis tools have begun to break down as the industry pushes past 100 kgate complexities.

You currently create hierarchy and break up the designs into synthesizable chunks to manage complexity. A certain amount of hierarchy is recommended for large designs, since it makes the design much more manageable and easy to understand. But the amount of hierarchy introduced to accommodate the limitations of traditional synthesis tools has an adverse affect on the design cycle. Additionally, many of the traditional synthesis algorithms have run times that grow exponentially as the design size increases.

This characteristic makes embedded synthesis using traditional technology unrealistic, especially when combined with already long place-and-route run times.

One technique to conquer the complexity challenge is to abstract your RTL to a higher-level behavioral representation, and to conserve this abstracted information throughout the synthesis process (language compilation, logic optimization, and technology mapping) until the final mapping step. This method differs from traditional synthesis tools that fragment designs into fine, low-level (gate) representations immediately after doing language compilation.

By preserving a higher-level behavioral abstraction, a synthesis tool can perform optimization at a much more global level (because it can more easily optimize across hierarchical boundaries) and ultimately deliver better results. By operating on abstracted data, the tool can run much faster and handle larger designs.

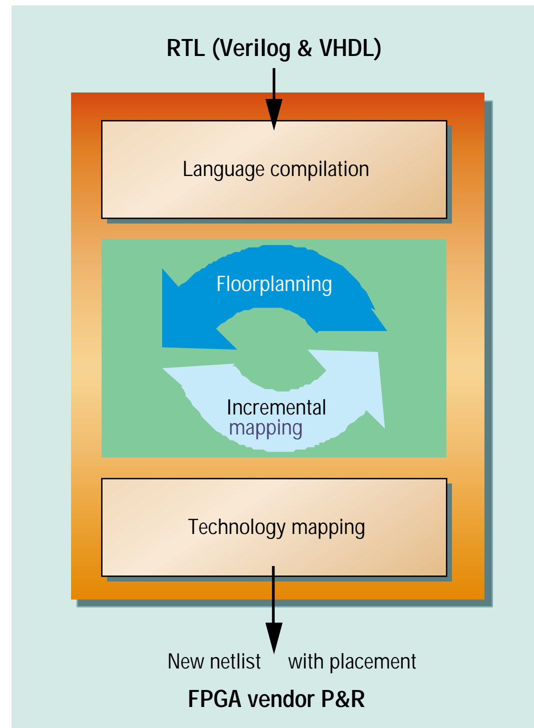
Synthesis tools should also be able to analyze the initial design hierarchy and further optimize it by creating a new hierarchy or dissolving existing hierarchy, where it's beneficial to do so from a logic synthesis point of view. The benefit of this technique is that a timing budget for each block along a critical path is automatically produced, making it easy for you to see where further improvements may be necessary.

Device partitioning

Before diving more deeply into floorplanning, let's look at device partitioning, a related application of embedded synthesis. Device partitioning is useful when a design needs to extend across multiple FPGAs or when prototyping an ASIC that is much larger than the available FPGAs. Today, experienced designers partition their designs into multiple devices manually.

Previous attempts at automatic partitioning tools haven't been accepted because they aren't effective in producing good partitions for real-world designs that meet the timing goals. Generally, previous attempts have operated on the post-mapped netlist, without access to the HDL's high-level content.

An embedded synthesis approach to partitioning that combines user interactivity with fast, accurate estimation and manipulation of HDL blocks allows you to interactively perform what-if analysis and quickly determine the best partition across devices. High-quality, flexible partitioning is increasingly important as the need for accelerated systems design time becomes the driving factor in many electronics market segments.



Partitioning is done at a point between HDL compilation and technology mapping. If done before compilation, the partitioner must work with complete modules, whose large size reduces the flexibility of the partitioning. If partitioning is done after technology mapping, too many irrevocable decisions have been made to permit effective partitioning and floorplanning, and too much contextual information is lost so that you have difficulty controlling the partitioning.

Embedded synthesis in partitioning lets you determine the best implementation of a design across multiple devices. By performing optimization and actually mapping the design to its target technology, you can make accurate estimations for performance and utilization. This real-time information is used to display how selected logic will fit in various combinations of devices.

Partitioning is done at a point between HDL compilation and technology mapping. If partitioning is done before compilation (at the register transfer level) then the partitioner must work with complete modules, whose large size reduces the flexibility of the partitioning. If partitioning is done after technology mapping, too many irrevocable decisions have been made to permit effective partitioning and floorplanning, and too much contextual information is lost. So, you have difficulty controlling the partitioning.

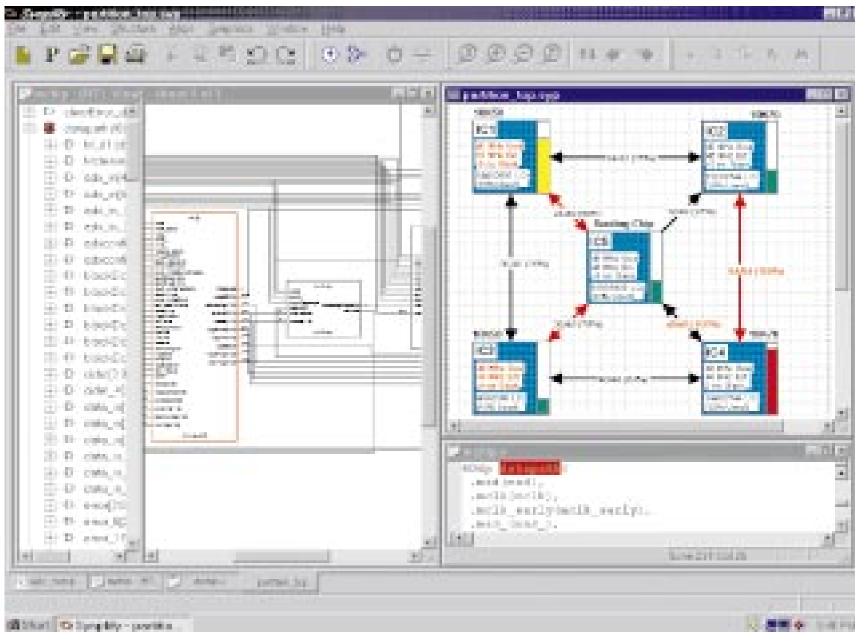
Before technology mapping, high-level information about the design is still available, so the partitioner can work with blocks instead of the mass of gates that represents the design after technology mapping. Performing partitioning between HDL compilation and technology mapping strikes a good balance.

Better performance

Partitioning the design before technology mapping also makes it possible to achieve better performance, using high-level optimizations such as object replication. Consider a design containing a register that drives two time-critical logic blocks, which you want to put in different chips. Putting the register in one of the chips will lead to performance problems in the other because of the long critical path from chip to chip. But the partitioner can provide the option to replicate the register in both chips to ensure good timing results.

Because the design is re-mapped (that is, a quick estimation synthesis is completed) when the partitioning is changed, the partition diagram can show exactly how each partition performs. The partition diagram can then compare the timing goal with the actual speed of the logic in each chip, and display the difference in terms of slack. Colored bar graphs on each chip give a quick visual indication of the amount of available slack. It's important to reiterate that this approach to partitioning only makes sense if synthesis (technology mapping, in this case) is fast enough to provide interactive performance. With mapping completed in less than a minute, however, you can change the partition and see the results in the on-screen diagram almost immediately.

In addition to obtaining better partitioning for designs whose final implementation will be in FPGAs, embedded synthe-



These windows show various design views, including the RTL hierarchy on the left and the code on the lower right, illustrating the way an embedded-synthesis partitioner might work. The characteristics of the partition appear in the diagram at the upper right. You can drag compiled blocks from one partition to another and observe the effects in the partition diagram. The diagram also shows the number of available logic cells in each chip. I/Os are profiled here, too, using colored arrows to indicate whether I/O congestion is becoming a problem.

sis simplifies the task of partitioning ASIC designs into FPGAs for prototyping purposes. Building FPGA partitions into the original HDL of an ASIC design is generally inconvenient. The ability to compile the HDL and then create the appropriate partitioning solves the problem, giving you an easy way to target both ASICs and FPGAs with the same HDL code.

Floorplanning can make even better use of embedded synthesis than partitioning. While both tasks deal with a design's physical hierarchy, partitioning involves a more rigid view because of the fixed resources (logic cells, I/Os, etc.) of the target FPGAs. Floorplanning is more fluid, especially for Xilinx (San Jose, CA) parts. The row structure of Altera's (San Jose, CA) 10-k devices makes floorplanning these FPGAs a little more like partitioning. But no matter what type of FPGA you're targeting, embedded-synthesis floorplanning makes excellent use of the design representation between HDL compilation and technology mapping.

The floorplanner can make manipulations based on individual registers, but isn't overwhelmed by the gate-level detail that follows technology mapping. If a design contains a 16-bit register, for example, technology mapping expands the register into 16 objects plus some other associated logic. The result is a complex map that's impractical for floorplanning.

Similar to the situation for partitioning, floorplanning before technology mapping provides an opportunity to replicate logic for performance purposes. This capability can help correct routing problems, because global routing can be reduced by replicating objects that have a small number of inputs and a large number of outputs.

Suppose a decoder with four inputs and 16 outputs drives three modules that have been floorplanned into different areas of the chip. Replicating the decoder into those three areas greatly reduces the number of global routes, letting you make timing and routability tradeoffs.

It's important to make such decisions after HDL compilation because at the HDL level, the decoder is part of a larger module. But only the decoder should be replicated.

By doing a better job in floorplanning, embedded synthesis will help reduce the number of iterations that are necessary to meet timing budgets. Another benefit will be a dramatic decrease in place-and-route run times—currently the bottleneck in the iterative loop.

Major FPGA vendors have mechanisms in their place-and-route tools that allow floorplanning information to be passed in for netlist-level placement. As mentioned earlier, at this level, placement is tedious and can't effectively handle large RTL blocks. The place-and-route tools need a way to accept floorplanning information at a higher level.

Incremental place-and-route would also help open up the bottleneck. Once the design has been partitioned into blocks, if you need to make a change that affects only one of the blocks, it should be possible to re-route only that one block and put all the blocks back together. That capability is part of a natural evolution of FPGA place-and-route toward more efficient operation.

A new class of tools developed specifically for deep submicron, high-complexity programmable devices is emerging, enabling 100,000 gate and greater FPGAs to become mainstream choices.

To maintain the needed productivity for complex programmable devices and still deliver leading-edge device performance and utilization, new methods and tools will have to be adopted. The driving technology behind this new class of tools will be embedded synthesis technology. Embedded synthesis will provide the needed link between logic optimization, technology mapping, and the back-end place-and-route tools that represent a bottleneck for high-end FPGA designers.

__Ken McElvain__ is founder, chief technical officer and director of Synplicity. Since starting Synplicity in February 1994, he has been the inventor and chief developer of the company's flagship product "Synplify" (a logic synthesis tool for FPGAs and CPLDs), a contributor on other products, and a primary strategist on company direction. Before Synplicity, Ken worked at Mentor Graphics for 10 years, as the principal designer of the AutoLogic synthesis tool. Mr. McElvain has a total of 16 years of CAE and hardware design experience. His other accomplishments while at Mentor and, previously, at Hewlett-Packard include: HDL synthesis, a static timing verifier, simulators, sequential ATPG, schematic editor, extension languages, and CPU design.