ISSUE 55, FOURTH QUARTER 2005 XCELL JOURNAL XILINX, INC.

Issue 55
Fourth Quarter 2005

Xcell journal

THE AUTHORITATIVE JOURNAL FOR PROGRAMMABLE LOGIC USERS

New ISE 8.1i Software – Faster Timing Closure

POWERED BY FMAX TECHNOLOGY

DESIGN PERFORMANCE

Physical Synthesis and Optimization with ISE Software

VERIFICATION

Early Defect Discovery with Assertion-Based Verification Accelerates Design Closure

PRODUCTIVITY

Using the ISE Foundation Architecture Wizards

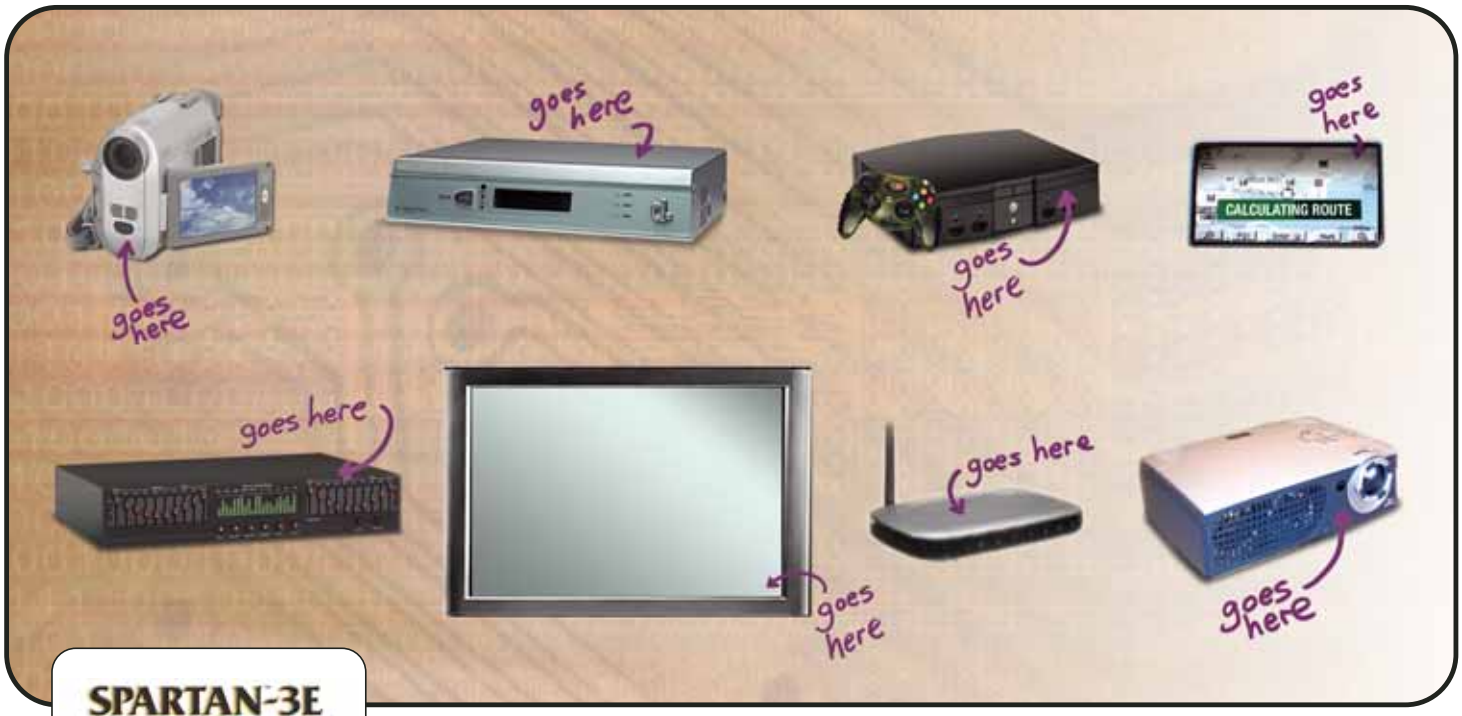
PARTIAL RECONFIGURATION

PlanAhead Software as a Platform for Partial Reconfiguration

XILINX®

Introducing the new Spartan-3E family — the world's lowest-cost FPGAs

Priced to go.



SPARTAN-3E FPGA

The industry's first 100,000-gate FPGA for only $2.00*

Spartan-3E Platform FPGAs offer an amazing feature set for just $2.00! You get 100K gates, embedded multipliers for high-performance/low-cost DSP, plenty of RAM, digital clock managers, and all the I/O support you need. All this in a production-proven 90nm FPGA with a density range up to 1.6 million gates.

Easiest to use Software

Perfect for digital consumer apps and much more!

With the Spartan-3E series, we've reduced the previous unit cost benchmark by over 30%. Optimized for gate-centric designs, and offering the lowest cost per logic cell in the industry, Spartan-3E FPGAs make it easy to replace your ASIC with a more flexible, faster-to-market solution. Compare the value for yourself . . . and get going on your latest design!

MAKE IT YOUR ASIC

XILINX®

The Programmable Logic Company℠

For more information visit
www.xilinx.com/spartan3e

Pb-free devices available now

FORTUNE 2005 100 BEST COMPANIES TO WORK FOR

* Pricing for 500K units, second half of 2006

A Roadmap to Productivity

There has never been a better time to be a design engineer. With today's products from Xilinx, you can accomplish more, in less time, with less risk, than ever before. With our Virtex™-4 FPGA family, you have the ideal silicon platform to tackle today's most complex system-design challenges, and with our ISE™ software you can unleash that power. The Xilinx ISE tools continue to be the design community's number-one choice; we hear this loud and clear from our customers.

We thank you for your support and we constantly strive to bring you more software innovations that maximize the benefits of our silicon solutions. For example, the 2005 EDA Survey conducted by *EE Times* indicates that the top three issues and challenges that you face are meeting timing budgets, getting your design to work on the PCB, and completing functional verification. The Xilinx software team has been hard at work solving these challenges for you, to help reduce your design time.

With the release of the ISE 8.1i software, we are introducing the new ISE Fmax technology, which as the name implies is designed to improve design performance and reduce design bottlenecks. With the addition of advanced options such as the PlanAhead™ hierarchical floorplanner, the ChipScope™ Pro analyzer, and wide industry support from leading EDA vendors, it is easy to see why our ISE software is the top choice.

We want to help you be more productive. With this in mind, this issue of the *Xcell Journal* includes a collection of articles highlighting tools and strategies to reduce the time you spend meeting your timing budgets, as well as articles on productivity enhancement tools and techniques in the areas of verification and board-level interfaces. We have also included case studies on how some of our customers are successfully using Xilinx software in their designs, as well as a poster on timing closure technologies, which we hope will be a handy reference to help you maximize performance in the shortest possible time.

Thank you for reading *Xcell*!

Forrest Couch
Executive Editor

Xcell PUBLICATIONS

Xilinx, Inc.
2100 Logic Drive
San Jose, CA 95124-3400
Phone: 408-559-7778
FAX: 408-879-4780
www.xilinx.com/xcell/

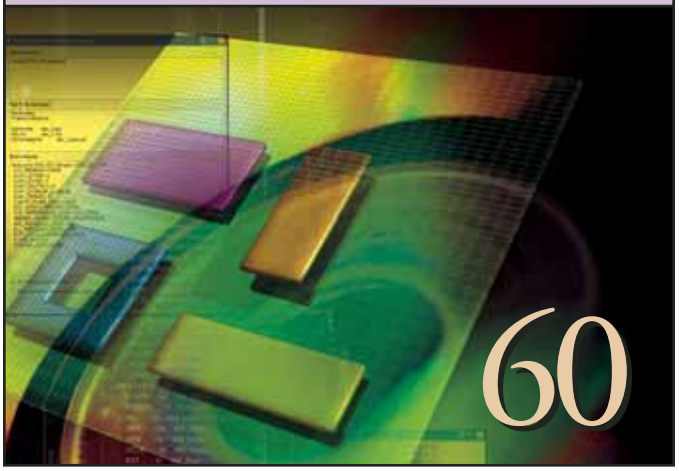
DESIGN PERFORMANCE

Physical Synthesis and Optimization
Meet your performance targets with these tips and strategies.

VERIFICATION

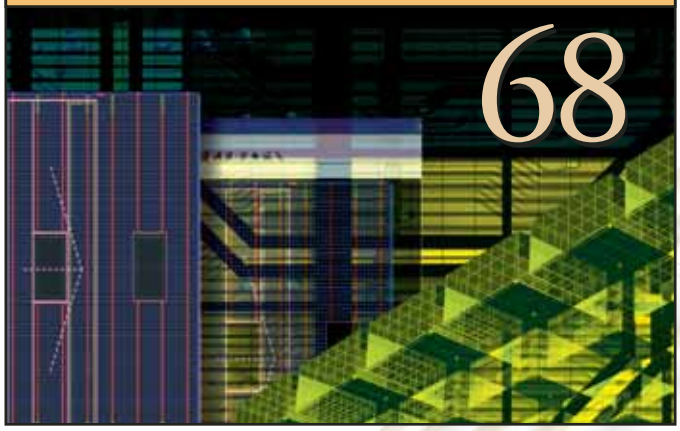
Early Defect Discovery with Assertion-Based Verification
The convergence of design, synthesis, and verification.

PRODUCTIVITY



Using the ISE Foundation Architecture Wizards
Streamline the process of configuring and instantiating the complex blocks found in Xilinx devices.

PARTIAL RECONFIGURATION

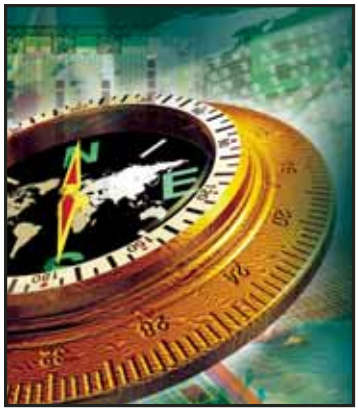


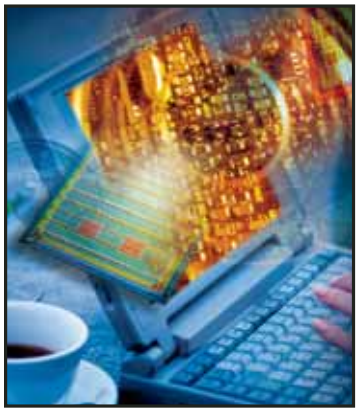
PlanAhead Software as a Platform for Partial Reconfiguration
PlanAhead software delivers a streamlined environment to reduce space, weight, power, and cost.

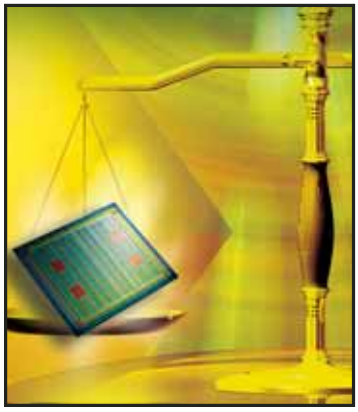
Viewpoint

Timing closure is perhaps the single most important design issue facing designers today. This article explores the methodology to achieve timing closure for Xilinx designs.

FOURTH QUARTER 2005, ISSUE 55







Xcell journal

Improving Time to Design Closure with ISE Software

Meet your design targets with these tips and strategies.

by Steve Lass
Director, Software Product Marketing
Xilinx, Inc.
steve.lass@xilinx.com

Timing closure is perhaps the single most important design issue facing designers today. With FPGAs and other deep submicron ICs, routing delays usually dominate logic delays. Although there are hundreds of ways to improve timing, such as using the embedded PowerPC™ processor or another high-speed core, the focus here is on improving performance on the logic and routing portion of the design. In this article, I will explore the methodology to achieve timing closure for Xilinx® designs.

Timing Closure Design Flow

Figure 1 shows a typical flow that starts with well-written HDL optimized for FPGA architectures. Xilinx ISE™ software provides Verilog and VHDL templates to begin crafting good HDL code. FPGAs have an abundance of registers, so adding pipeline stages can greatly improve timing and have very little impact on area. Some of the frequently used best practices include keeping critical paths in the same entity or module, utilizing clock enables instead of gated clocks, and avoiding the use of latches, nested for-loops, and if-then-else statements in the HDL code.

Xilinx also recommends using synchronous resets for modules such as DSP48, FIFO16, and block RAMs, and using adder chains instead of adder trees to help achieve 500 MHz DSP48 performance (illustrated in Figure 1). A comprehensive discussion on coding styles can be found in the article, "HDL Coding Practices to Accelerate Design Performance."

Synthesis

The flow continues with synthesis, which provides an early indication of whether or not your HDL has a chance of meeting timing and area requirements. Make sure you constrain timing in the synthesis tools to avoid minimizing area at the expense of timing – something that will likely occur if the tools are given no other direction. At a minimum, constrain your clocks and I/O paths.

You can also allow the synthesis tools to try harder by indicating an optimization effort. Another frequently used option to meet timing requirements is register balancing, which moves registers forwards or backwards through logic to increase clock frequency. If the synthesis tools indicate that timing cannot be met, or if timing is very tight, you may be able to further optimize your HDL code by using one or more of the coding techniques previously discussed.

Implementation

Having obtained an acceptable timing estimate from the synthesis tool, use the implementation tools (map, place, route, timing analysis) to determine the true timing of the design. Xilinx ISE tools, powered by Fmax Technology, proactively attempt to achieve the best performance possible, but do require that constraints are complete. A recommended set of timing constraints (as shown in Figure 2) should include clock period, I/O offset, multi-cycle path specification, and timing ignore (TIG) to ignore false paths. If you are missing timing by more than 20%, you may need to do further optimization of your HDL, especially within the module containing the worst-case path.

If you still haven't met your timing requirements in the implementation phase, there are many tool options that can provide dramatic improvements. A good starting place is the use of retiming in your synthesis tool. Another option is to turn on retiming and global optimization in ISE Mapper.

Xplorer Utility

Xilinx recently introduced a new utility called Xplorer that delivers optimal design results by employing smart constraining techniques and a variety of physical optimization strategies. You can

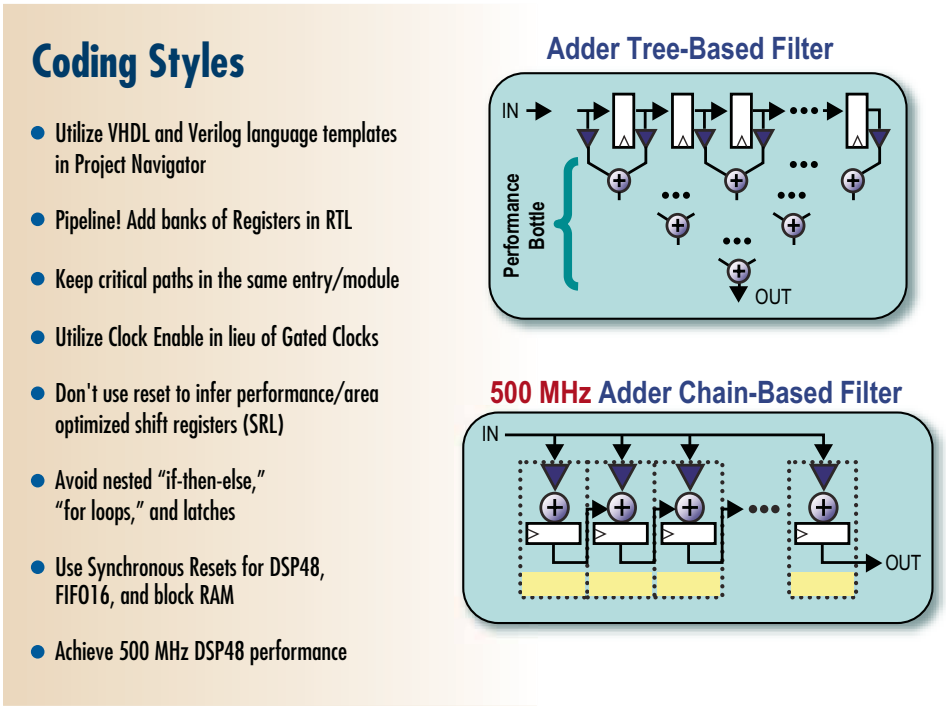


Figure 1 – Coding styles

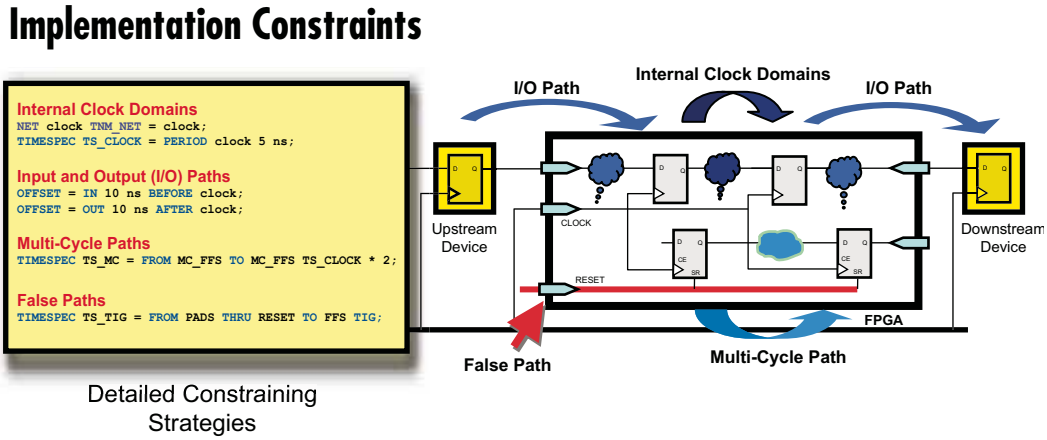


Figure 2 – Implementation constraints

use the Xplorer utility to automatically try these (implementation tool) options and even try different clock frequencies to find the maximum achievable speed of the design. Once Xplorer has found the best tool options, you should use those options the next time you run the implementation tools, avoiding the long runtimes of Xplorer. You can learn more on how to achieve the best performance with Xplorer in the article, "Accelerate Design Performance Using Xplorer."

PlanAhead Design Tools

If you've tried everything and still can't reach that timing closure pinnacle, there is still hope. The Xilinx PlanAhead™ tool can be used to analyze, and if necessary, floorplan the design to achieve higher performance (15% average, but can be as high as 2X). PlanAhead design tools provide better insight into the place and route process. You can quickly examine "what if" scenarios, enabling you to identify and fix potential problems early. You can also group critical paths and modules to increase routability through connectivity analysis and utilization control.

See the article, "Improve Design Performance Using PlanAhead Design Tools" to learn more about the capabilities of this incredible tool. There are many other advanced techniques like detailed floorplanning and creating RPMs (relationally placed macros), but we recommend you try the ideas in this article first.

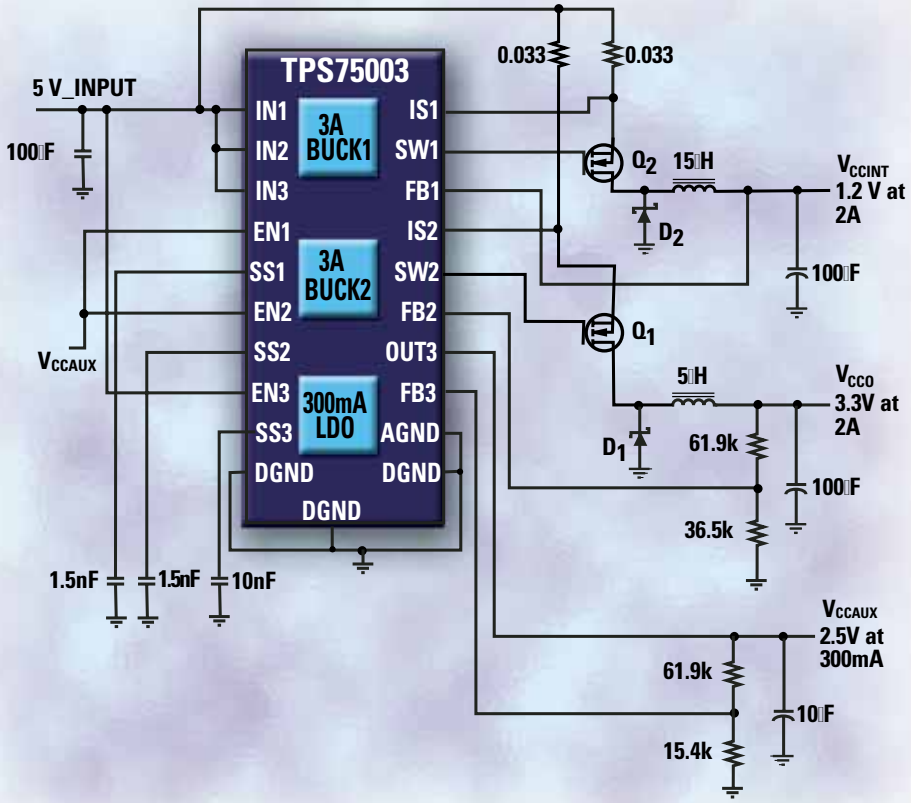
Conclusion

Xilinx provides a comprehensive suite of software tools, powered by ISE Fmax Technology, that you can use to improve design performance. ISE software, together with the tips and strategies in this article, can help you quickly achieve timing closure.

Additionally, we work with leading third-party synthesis vendors to optimize designs and improve design performance for Xilinx devices in their leading synthesis software. An entire section of the *Xcell Journal* has been devoted to achieving design performance using Xilinx software tools. Please read the related articles in this issue to learn more.

POWER MANAGEMENT

TI Powers Xilinx FPGAs



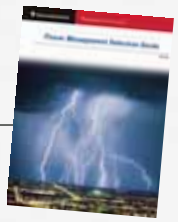
▸ Applications

- DSL modems
- Set-top boxes
- Plasma TV display panels
- DVD players

▸ Features

- Two 95%-efficient, 3-A buck controllers and one 300-mA LDO
- Adjustable output voltages
  - from 1.2 V for bucks
  - from 1.0 V for LDO
- Input voltage range of 2.2 V to 6.5 V
- Independent soft-start for all three power supplies
- LDO stable with small ceramic output capacitor
- Independent enable for each supply for flexible sequencing
- 4.5 mm x 3.5 mm x 0.9 mm 20-pin, QFN package
- $1.90: 1 K price

The **TPS75003** power management IC for Xilinx's Spartan™ and Virtex™ series of FPGAs integrates multiple functions to significantly reduce the number of external components required and simplify design. Combining increased design flexibility with cost-effective voltage conversion, the IC includes programmable soft-start for in-rush current control and independent enables for sequencing the three channels. The TPS75003 meets all Xilinx startup profile requirements, including monotonic ramp and minimum ramp times.



NEW!
Power Management Selection Guide
power.ti.com

For more information on TI's complete line of power management solutions for Xilinx FPGAs—including a library of reference designs, schematics and BOMs—visit **www.ti.com/xilinxfpga-u.**

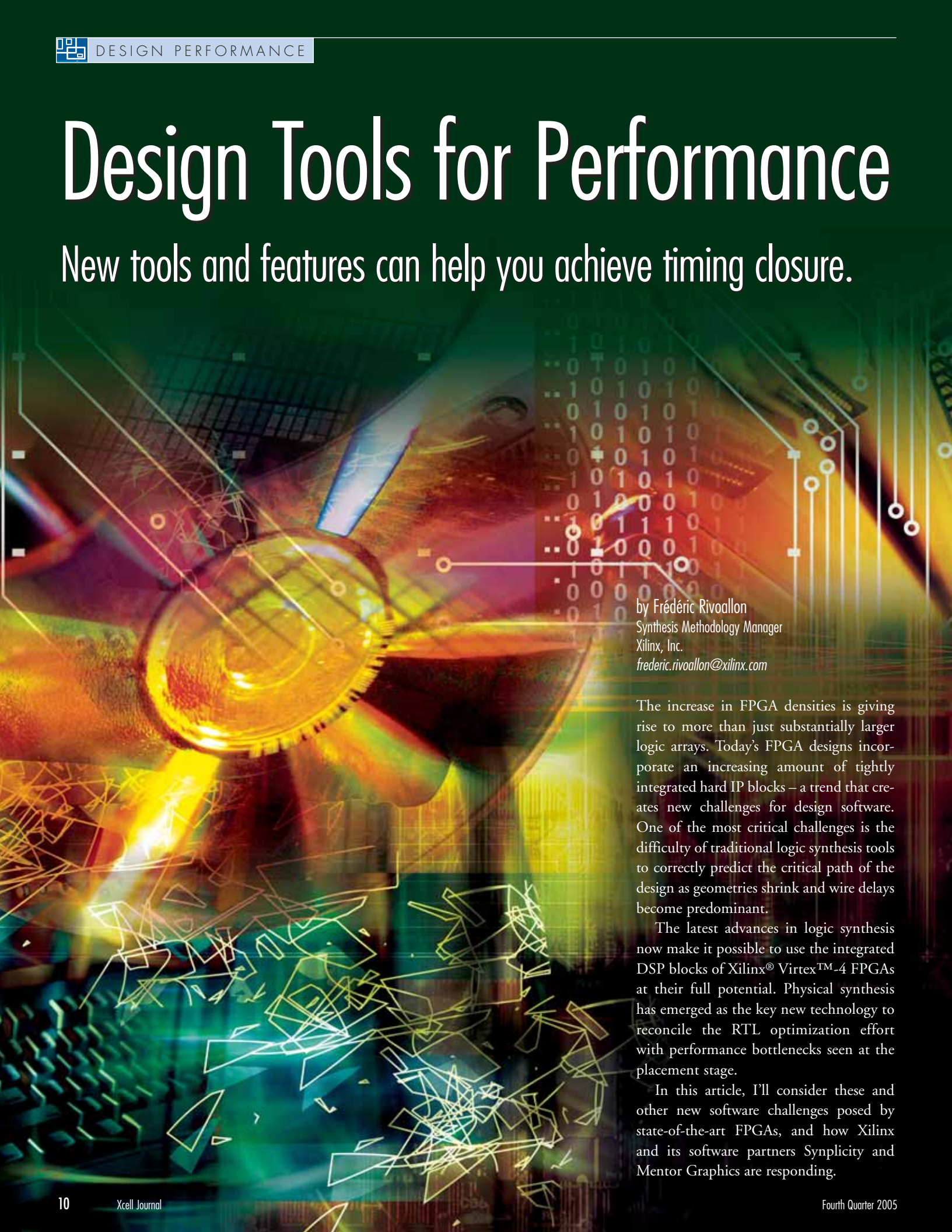
www.ti.com/xilinxfpga-u ○ 800.477.8924, ext. fpga

Technology for Innovators™

Texas Instruments

Design Tools for Performance

New tools and features can help you achieve timing closure.



by Frédéric Rivoallon
Synthesis Methodology Manager
Xilinx, Inc.
frederic.rivoallon@xilinx.com

The increase in FPGA densities is giving rise to more than just substantially larger logic arrays. Today's FPGA designs incorporate an increasing amount of tightly integrated hard IP blocks – a trend that creates new challenges for design software. One of the most critical challenges is the difficulty of traditional logic synthesis tools to correctly predict the critical path of the design as geometries shrink and wire delays become predominant.

The latest advances in logic synthesis now make it possible to use the integrated DSP blocks of Xilinx® Virtex™-4 FPGAs at their full potential. Physical synthesis has emerged as the key new technology to reconcile the RTL optimization effort with performance bottlenecks seen at the placement stage.

In this article, I'll consider these and other new software challenges posed by state-of-the-art FPGAs, and how Xilinx and its software partners Synplicity and Mentor Graphics are responding.

A Two-Fold Challenge

Most large designs are compiled from the top down. Given their size and complexity, it is not uncommon for such designs to engender several potentially critical paths. As a result, when operating predominantly from inaccurate estimations based on empirical wire-load models, a synthesis tool might optimize paths that are in fact not critical and not consider others that truly could be critical.

A second challenge for synthesis tools is inferring larger hard IP elements (memory or DSP blocks) found in Virtex-4 FPGAs. When you need to keep the RTL code for a given project portable (as is often the case), the burden is on the synthesis tool to generate excellent results based on that portable code. It must be able to accurately map the logic onto the hard IP blocks.

New Software Tools and Optimizations

Physical synthesis offers a solution to reconcile front-end optimizations with the actual results derived from performing place and route. Physical synthesis tightly couples synthesis and place and route by making synthesis aware of actual timing bottlenecks early in the design. It ensures that synthesis optimizations are effectively applied to the appropriate places and interacts with placement to deliver superior results. This technology does not require manual intervention and can be used in "push-button" flows.

Physical synthesis algorithms are now available in Xilinx ISE™ 8.1i software. Precision Physical from Mentor Graphics and Synplify Premier from Synplicity also provide physical synthesis capabilities. The latter uses an innovative approach called "graph-based synthesis" in which placement takes into account the available routing resources. Synplify Premier offers this technology to Xilinx FPGAs only.

Synthesis also addresses the use of more sophisticated hard IP blocks by providing inference algorithms that understand the detailed structure of the FPGA. This recent enhancement enables the tools to effortlessly produce pure RTL descriptions, yielding 500 MHz predictable performance without the need for instantiation.

To provide the best push-button results, Xilinx also provides a multi-compile script called Xplorer. Xplorer has two modes: the first one attempts to obtain the maximum performance for the design, while the second one works within the designer's constraints to meet timing. In this second timing closure mode, Xplorer applies different algorithms for logic packing and place and route, and reports the best settings for future design iterations.

Think Hardware

Successfully coding RTL for performance requires silicon considerations. Once you "think hardware," your RTL description directs synthesis to use specific silicon functions. Consider the integrated XtremeDSP™ blocks in Virtex-4 FPGAs. They enable ASIC performance, but that performance can be severely impacted if the RTL coding style implies an asynchronous reset. That's because the native reset of the block is synchronous. Using a synchronous reset enables registers to be merged into the block, thus improving performance (and area) to a large extend.

Regardless of software tools, coding styles are essential. Combined with synthesis tool constraints, options and synthesis directives can drastically affect performance.

Regardless of software tools, coding styles are essential. Combined with synthesis tool constraints, options and synthesis directives can drastically affect performance.

PlanAhead

FPGAs offer specialized resources that designers must manage to create an optimal solution. For example, you may want to align certain blocks to a given clock domain using specific resources, or group the design critical path logic to ensure a tight implementation of those resources. This requires more than a simple improvement to the "push-button" synthesis solution.

Xilinx offers a tool called PlanAhead™ software that provides just this type of resource management. PlanAhead software has an intuitive graphical interface that lets you browse the logic hierarchy of a design to create an optimal connection to the physical layout of the targeted device.

Additionally, PlanAhead design tools can help generate IP blocks and make it simple for you to export them to other designs. PlanAhead software also makes it easy to use advanced block-based flows such as incremental design, modular design, or even flows involving reconfigurability.

Conclusion

The evolution of larger, more complex FPGA designs with increasing amounts of hard IP poses a substantial challenge to tool suppliers. Advances in areas such as physical synthesis and new inference algorithms make the full performance potential of next-generation FPGAs accessible at virtually the push of a button, while PlanAhead software places the full suite of FPGA resources at your fingertips – even for the most complex, block-based design flows. The Xplorer script provides the most efficient path to discover the maximum performance or timing closure.

Other articles in this edition of the *Xcell Journal* will provide more details on these topics, including coding styles, the physical synthesis capabilities of ISE 8.1i software, Synplify Premier, the Xplorer tool, and PlanAhead design tools.

Achieve Faster Timing Closure with Graph-Based Physical Synthesis

Graph-based physical synthesis was invented to improve timing closure by means of a single-pass physical synthesis flow.

by Jeff Garrison
Director of Marketing, FPGA Products
Synplicity, Inc.
jeff@synplicity.com

Advances in FPGA technology have opened the door wide open for use in all types of applications, including wireless communications, computer, industrial, defense/aerospace, medical, automotive, and even consumer. Xilinx® Virtex™-4 devices have the capacity, performance, and cost structure to lead a migration from traditional cell-based ASICs to programmable devices in all but the highest volume and bleeding-edge applications. Along with this capability, however, are new challenges from a designer's perspective. In this article, I'll discuss a solution to one of these most important challenges – timing closure.

One of the biggest reasons to use FPGAs in the first place is their ability to deliver working silicon to an electronics system quickly and reliably. As the complexity of FPGAs and all integrated circuits has increased, the time-to-market advantage offered by FPGAs could be diminished if you do not make significant changes to your core design technology. The primary issue is timing closure – the ability to reach your design's timing goals in a fast, predictable way. Gone are the days when logic delay and wire-load models for interconnect delay are enough to estimate timing and give predictable results. In 90 nm FPGAs, you must incorporate actual routing delay into the synthesis process to achieve rapid timing closure for high-performance designs.

Timing Accuracy is Everything

The underlying problem that determines if you will be able to close on timing is estimation accuracy. Historically, synthesis and placement tools have been based on the assumption that the proximity of logic and wire-load estimation determines the routing delay. Although this used to work reasonably well for ASIC design, it does not work at all for FPGAs. Unlike an ASIC, FPGA routing is pre-determined. In an ASIC the routing is customized for the placement of the logic. In other words, once the placement of an ASIC is done, it is relatively easy to get a good estimation of routing delay by measuring Manhattan distances from one point to another (see Figure 1).

Because FPGAs have fixed routing resources, the design tool needs to understand the different types of routing and its implication on timing. In an FPGA, the fastest routing between two points may very well not be the shortest. Think of your commute to work – sometimes it's faster to go slightly out of your way and get on a freeway than to travel the shorter distance on side streets. The same concept applies to FPGAs: some direct routing resources (freeways) are faster than those that have to go through switch matrices (side streets) (Figure 2).

Figure 3 illustrates how placement is different for proximity- and graph-based tools.

Graph-Based Physical Synthesis

Synplicity invented graph-based physical synthesis to improve timing closure by means of a single-pass physical synthesis flow for 90 nm FPGAs. The essence of the graph-based approach is that the pre-existing wires, switches, and placement sites used for routing an FPGA can be represented as a detailed routing resource graph. The notion of distance then changes from proximity to a measure of delay and wire availability.

Synplicity's graph-based physical synthesis technology merges optimization, placement, and routing to generate a fully placed and physically optimized design, providing rapid timing closure and a 5% to 20% timing improvement.

Graph-based physical synthesis does not require you to create a floorplan or provide other information to the physical synthesis

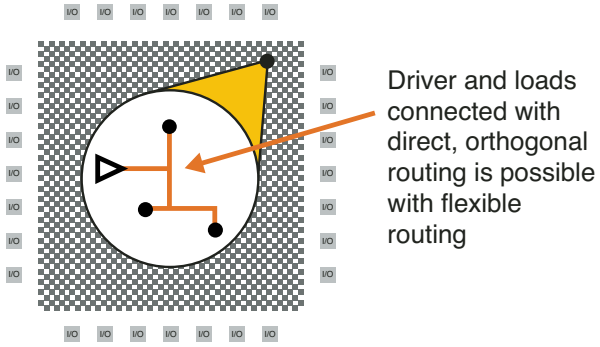


Figure 1 – Proximity-based placement is best when you use flexible routing.

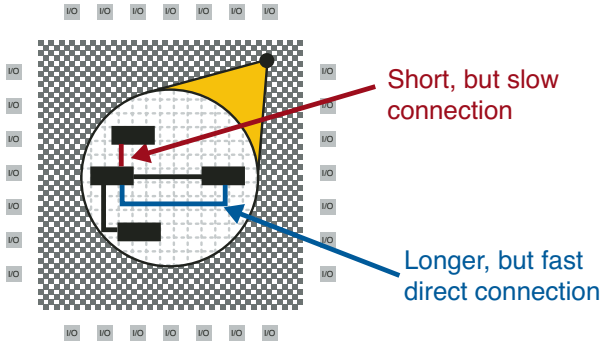


Figure 2 – A graph-based approach is best when routing is fixed and of differing performance levels.

process (often only known by expert users) in order to get good results. It is a fully automated methodology that can be used without special knowledge of the physical FPGA device. In addition to this fully automated mode, you do have the option to guide physical synthesis by providing design planning information (such as a floorplan) used during the physical synthesis process.

Synplify Premier

To directly address the challenge of keeping timing closure under control for advanced FPGA technologies, Synplicity has introduced Synplify Premier, its first FPGA design product based on graph-based physical synthesis technology. Synplify Premier includes all of the features in Synplify Pro and adds graph-based physical synthesis for the Virtex-4, Virtex-II Pro, and Spartan™-3 families. In addition to the new graph-based physical synthesis, Synplify Premier also offers a new capability for debugging and prototyping ASICs using FPGAs. One such technology is RTL instrumentation and debugging of live, running FPGAs.

This technology is based on Synplicity's Identify product, which allows you to navigate your design graphically and mark signals directly in your RTL code as probes or sample triggers. After synthesis, you can view the signal values of a live, running FPGA directly in the RTL source code or in waveform. An incremental place and route capability saves time by allowing you to quickly update instrumented nodes and debug. The debugging technology within Synplify Premier software is closely integrated with synthesis and Xilinx ISE™ software for a seamless development environment.

A second technology important for ASIC prototyping with FPGAs is the ability to convert gated clocks to FPGA clock-enable structures without modifying your RTL source. Synplify Premier performs this task automatically, along with handling generated clocks and instantiations of most common Synopsys DesignWare components.

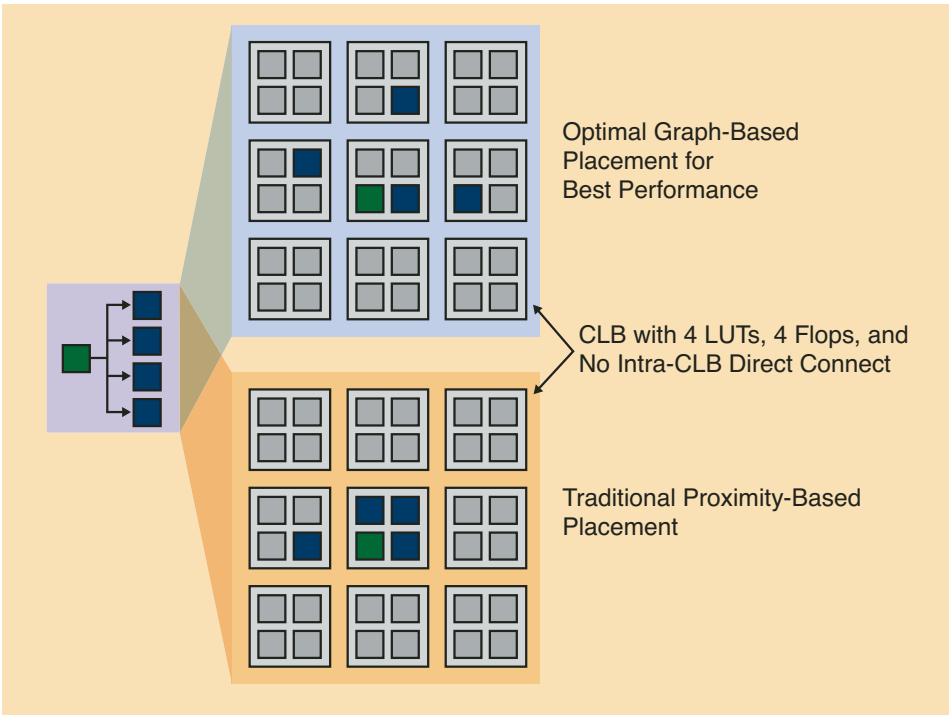


Figure 3 – Graph-based placement for LUT driving four flops

Conclusion

Because of the increased design complexity enabled by new devices such as Virtex-4 FPGAs, designers need EDA tools that can handle the physical properties of FPGA architecture to achieve acceptable timing closure. Several physical design tools based on ASIC technologies have been used to address FPGA design, but they have had little success. The ASIC approach does not work for FPGAs because the silicon fabric is completely different and, unlike ASICs, proximity does not imply better timing.

Synplicity's Synplify Premier product with graph-based physical synthesis directly addresses the challenges of FPGA physical design and results in faster designs done in less time. For more information on graph-based physical synthesis, ASIC prototyping, and Synplify Premier, visit *www.synplicity.com/products/index.html.*

Physical Synthesis and Optimization with ISE Software

These tips can help you get the most out of your implementation tools.

by Kevin Bixler
Manager, ISE Technical Marketing
Xilinx, Inc.
kevin.bixler@xilinx.com

David Dye
Senior Technical Marketing Engineer
Xilinx, Inc.
david.dye@xilinx.com

Advances in process technology have lead to dramatic increases in FPGA device densities. Several Xilinx® Virtex™ families have devices exceeding 1 million system gates. This increase in device density and the use of 300 mm wafers have made FPGAs affordable for volume production.

Designs that were once exclusively targeted at ASICs are now being implemented in programmable devices. The largest 90 nm Virtex-4 device provides more than 200,000 logic cells, 6 MB of block RAM, and nearly 100 DSP blocks. Creating a design to efficiently utilize the available resources in these devices and meet performance requirements can be challenging. Fortunately, today's EDA software tools have evolved to meet these challenges.

Logic optimization, logic placement, and minimized interconnection delays are all important to achieve maximum performance. Timing-driven synthesis technology has provided a significant improvement in design performance. The limiting factor to the effectiveness of timing-driven synthesis is the accuracy of estimating routing delays.

Physical synthesis – the use of physical placement and routing information during synthesis – has been at the forefront to effectively address these issues. Physical synthesis and optimization further expands on this technology by involving synthesis in implementation decisions after the netlist is generated. This allows for dynamic re-examination of synthesis mapping and packing decisions based on actual placement and routing information during implementation.

Benefits of Physical Synthesis and Optimization

Interconnection delays between logic levels are affected by the proximity of placement for the logic elements, routing congestion, and local competition between nets for the fastest routing resources. The answer to this problem is to revisit synthesis decisions during mapping, placement, and routing. During the mapping phase, the netlist can be reoptimized, packed, and placed based on the urgency of individual timing paths. This approach reduces the number of implementation cycles required for timing closure.

Physical Synthesis and Optimization Flows

Xilinx ISE software provides several software options to enable physical synthesis and optimization. You can use these options individually or together, depending on the specific needs of your design.

Define Timing Requirements

The most important step for effective physical synthesis is to set up accurate, comprehensive timing constraints. With these constraints in place, the implementation tools can make more informed decisions that will improve your overall results. Constrain the clocks and I/O pins that have firm requirements to allow the rest of the design to be relaxed.

During the mapping phase, the netlist can be reoptimized, packed, and placed based on the urgency of individual timing paths. This approach reduces the number of implementation cycles required for timing closure.

The easiest way to define these timing constraints is to use the Constraints Editor. This graphical tool allows you to enter clock frequencies, multi-cycle and false path constraints, I/O timing requirements, and a host of other clarifying requirements. Constraints are written to a user constraint file (UCF), which may also be edited in any text editor.

If user-defined timing constraints are not provided, a new feature in ISE™ 8.1i software will automatically generate timing constraints for each internal clock. In Performance Evaluation Mode (PEM), you can get the high-performance results of physical synthesis and optimization without having to provide timing targets.

Run Global Optimization

For designs containing IP cores or other netlists, the NGD file available after the translate (NGDBuild) phase of implementation represents the first time that the entire design has been completely assembled. Global optimization, a new feature added to the 7.1.01i version of Map, will take the fully assembled design and attempt to improve design performance by re-optimizing the combinatorial and register logic. Global optimization (map -global_opt on the command line) has been shown to increase design clock frequencies by an average of 7%.

Two other options let you further control the optimization completed during this phase: retiming (map -retiming) will move registers forward and back to balance combinatorial logic delays, and equivalent register removal (map -equivalent_register_removal) will remove registers with redundant functionality.

Enable Timing-Driven Packing and Placement

Timing-driven packing and placement is at the heart of the physical synthesis capabilities available within the implementation flow. When you enable this option (map -timing), the placement phase of place and route is done within Map, allowing packing decisions to be revisited when initial results are less than optimal. This iterative flow does away with unrelated logic packing.

Different levels of optimization exist in Xilinx physical synthesis and optimization. The first level was introduced in ISE 6.1i software and began with logic transformations, including fanout control, logic replication, congestion control, and improved delay estimation. These routines led to much more efficient packing and placement of designs, resulting in faster clock frequencies and denser logic utilization.

The next level added logic and register optimization; Map can now rearrange elements to improve critical path delays. These transformations give much greater flexibility to meet the timing requirements of the design. A number of different techniques (including pin swapping, basic element switching, and logic recombination) are used to massage the physical elements into a different yet logically identical structure that will meet the design requirements.

ISE 8.1i software introduces one more level of physical synthesis – combinatorial logic optimization. The -logic_opt switch enables a flow that examines all of the combinatorial logic in the design. Given placement and timing information, you can make more informed decisions about optimizing LUT structures to improve the overall design.

Examples of Physical Synthesis and Optimization

- Logic Duplication: If a LUT or flip-flop drives multiple loads, and the placement of one or more of those loads is too far away from the source to meet timing, the LUT or flip-flop can be replicated and placed close to that group of loads, thus reducing routing delays (Figure 1)
- Logic Recombination: If the critical path traverses through multiple LUTs through multiple slices, the logic can be reassembled utilizing fewer slices by using a more timing efficient combination of LUTs and muxes to reduce the routing resources needed for that path (Figure 2)
- Basic Element Switching: If a function is built with LUTs and muxes within a slice, physical synthesis and optimization can rearrange the function to give the fastest path (usually through the mux select pin) to the most critical signal (Figure 3)
- Pin Swapping: Each input pin of a LUT may have a different delay, so Map has the ability to swap pins (and the associated LUT equation) so that the most critical signal is placed on the fastest pin (Figure 4)

Conclusion

The physical synthesis and optimization capabilities within the Xilinx toolset will continue to mature and expand with each software release. Along with improved quality of results, you can expect to see greater control over the types of optimizations. Other planned enhancements include the consideration of more design elements in the reoptimization phase (such as registers allowing movement into and out of the I/O blocks or dedicated functions like block RAM and DSP blocks) and the inclusion of the routing phase into the reiterative physical synthesis and optimization system.

The physical synthesis and optimization tools in Xilinx ISE software have been created to re-examine the structure of your FPGA design during the packing and placement phases of implementation. With the knowledge of timing constraints and physical layout, optimizing synthesis decisions during map and place and route can significantly improve your results.

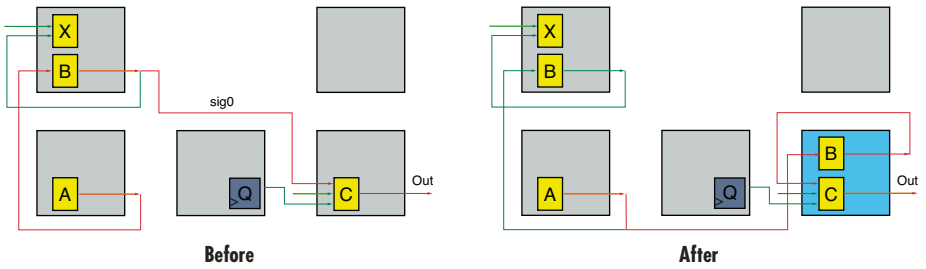


Figure 1 – Logic Duplication.
Path LUT A → LUT B →LUT C → Out is critical. LUT B is driving two critical loads and can be duplicated to reduce path delay.

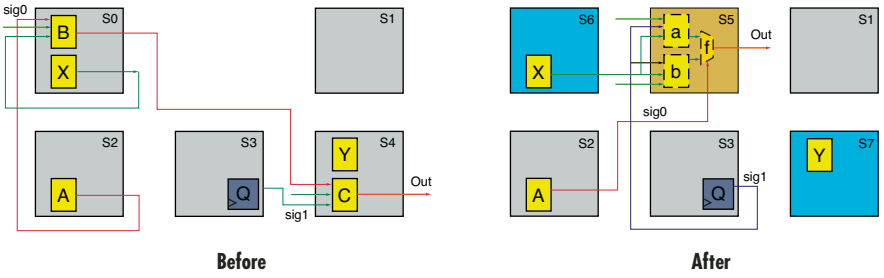


Figure 2 – Logic Recombination.
Path LUT A → LUT B → LUT C → Out is critical. LUTs B and C can be combined and replaced by LUT a, LUT b, and F5Mux f.

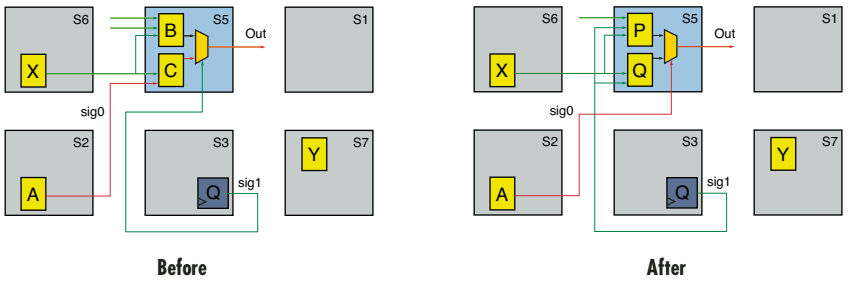


Figure 3 – Basic Element Switching.
Path LUT A → LUT C → MuxF5 → Out is critical. The path through the mux select pin is faster than through LUT.

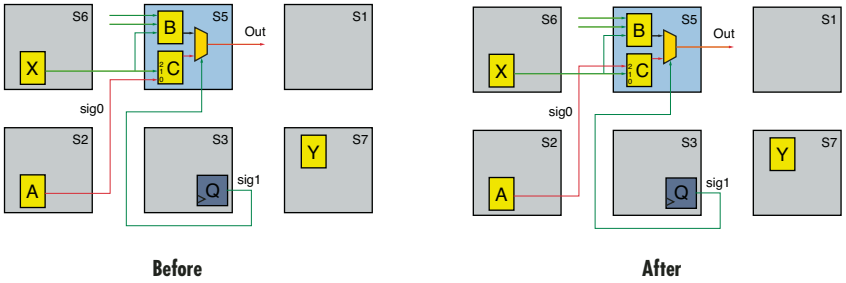


Figure 4 – Pin Swapping.
Path LUT A → LUT C → MuxF5 → Out is critical. Pin 2 is faster than Pin 0 for LUT C. Swap pins 0 and 2 for LUT C.

NU HORIZONS ELECTRONICS CORP.

XILINX®

intersil

LINEAR TECHNOLOGY

NU HORIZONS SIGNAL PROCESSING VIRTUALAB™

http://www.techonline.com/community/38647

Lab Summary

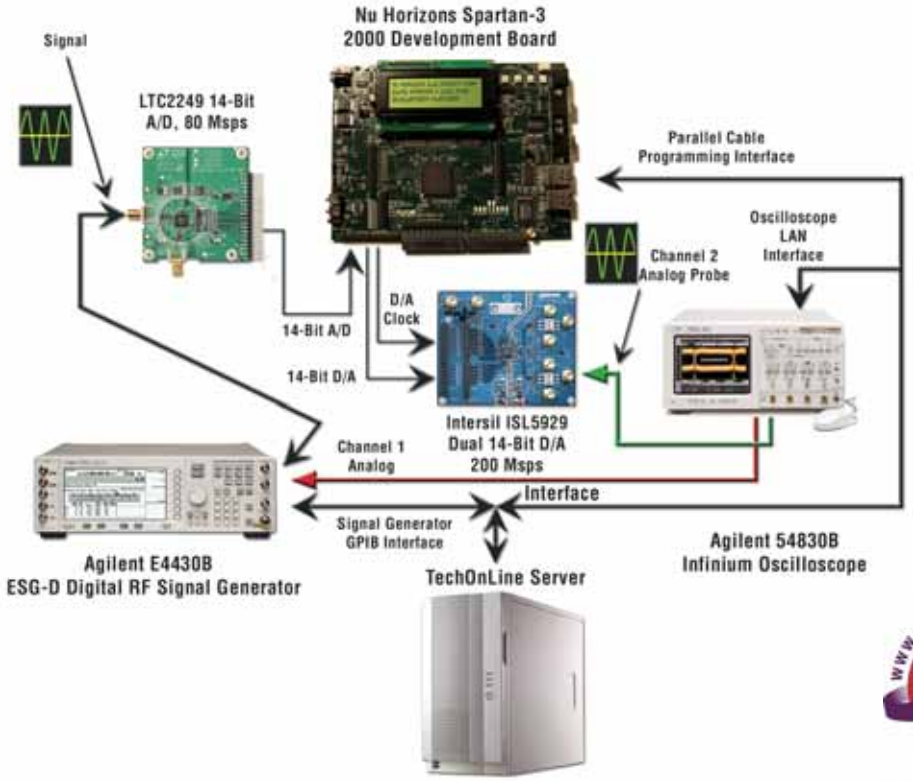
The lab is designed to allow the user to create and download a DSP application, provide signal insertion and measure output waveforms. Advanced algorithms can be developed and complex measurements can be preformed via a full complement of test equipment connected to the VirtuaLab environment. Both a Signal / Pattern Generator and High Speed Oscilloscope are connected to the A/D and D/A modules and each piece of test equipment is placed in the Host Mode so the user can remotely manage the equipments front panel controls. Settings and scripts can be easily saved for re-use on a future session. Real time access to silicon provides accurate measurements, real results and confidence in the findings. The signal insertion and output measurement capabilities enable designers to validate their algorithms.

Nu Horizons VirtuaLab Key Points:

- Enabled by collaboration from a number of industry leaders including: Nu Horizons, Xilinx, TechOnLine, Linear Technology, Intersil, and Mathworks
- Designed to support high performance DSP applications
- First VirtuaLab to incorporate entire signal chain including both A/Ds and D/As
- First VirtuaLab to incorporate Simulink from Mathworks
- Includes test equipment to provide signal insertion and measure output waveforms
- Xilinx ISE Foundation Development Environment
- Xilinx System Generator for DSP

Reference Labs:

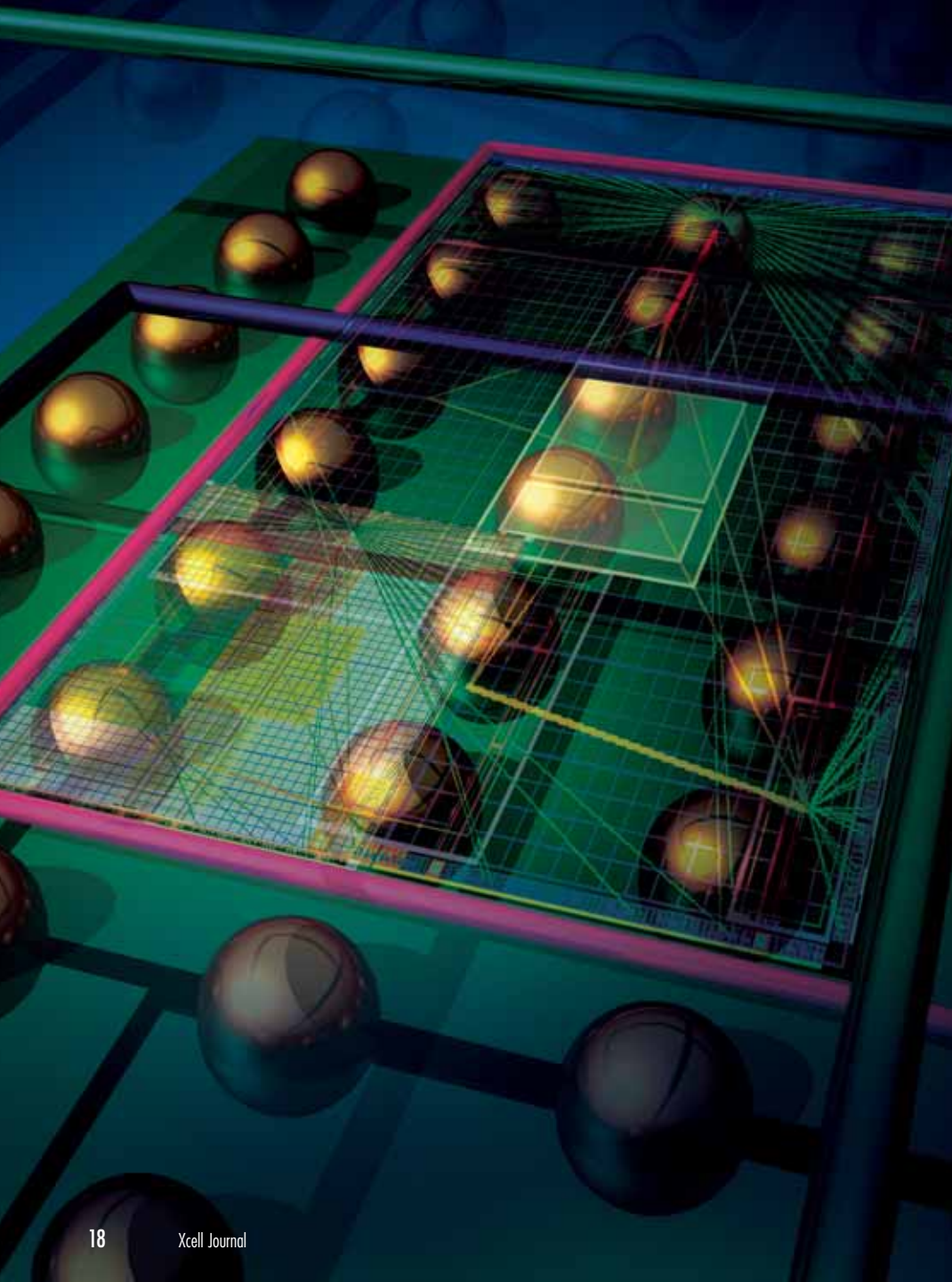
- Reference DSP designs are provided in the VirtuaLab environment, allowing the design engineer to evaluate the Spartan-3 2000 FPGA in a pre-verified environment with ease. Reference designs include:
  - Existing Sysgen Tutorial - which introduces the design engineer to other features such as: ChipScope, HDL Cosim, Hardware Cosim and PicoBlaze. Simple FFT with a 256 Tap FIR filter with interpolation by 3. Ease of use and reasonably high performance allow the design engineer to evaluate the tool interface as well as the hardware. The filter design will be provided to the design engineer using the FDA Tool to generate the coefficients, allowing the design engineer the ability to modify the coefficients and view the results. Simulation can be run in Sysgen, as well as run live in hardware.
  - Equalized 16-QAM demodulator including the Adaptive filter. The receiver architecture provides subsystems that demonstrate adaptive channel equalization and carrier tracking on a random QAM data source.



www.techonline.com
TechOnLine

Improve Design Performance Using PlanAhead Design Tools

PlanAhead software helps you tackle the need for speed.



by Mark Goosman
Marketing Product Manager
Xilinx, Inc.
mark.goosman@xilinx.com

Design problems – especially those characteristic of large, high-performance designs – are most effectively addressed by first investigating the problem and then breaking the larger design issues into smaller, more manageable hurdles. Looking at the evolution of programmable devices in recent years, it is apparent that FPGAs have undergone tremendous growth in size and complexity, but the PLD EDA tool flow has remained relatively unchanged.

With a traditional flat design flow, each design change means re-synthesizing and re-implementing the entire design. With complex designs on multi-million-gate devices, even a minor change can lead to unacceptably long place and route (PAR) runtimes, which itself often leads to inconsistent results, not to mention the time lost from RTL to PAR iterations for a typical design.

Few design teams can tolerate unexpectedly low performance for a design that took longer than expected to complete, not to mention the associated frustration and stress. In addition, it may mean low utilization of the FPGA and even missed time-to-market opportunities.

PlanAhead Software Offers a Solution

A growing number of customers are finding a solution in the hierarchical design methodology offered by the Xilinx® PlanAhead™ design analysis tool. PlanAhead software adds visibility and control to the FPGA design flow. By addressing problems on the physical side (between logic synthesis and the implementation process), you can realize improved performance in your results.

Although advanced FPGA synthesis products provide a tremendous level of automated optimization for multi-million-gate designs, many designers require more heuristic techniques to achieve optimal performance goals. Through early analysis and floorplanning, PlanAhead design tools can apply physical constraints to help control the initial implementation of the design. After implementation, PlanAhead software can analyze placement and timing results to improve the floorplan used to complete the design. You can use the physical constraints derived from the imported results to lock placement during subsequent implementation attempts. These constraints can be used to create reusable IP, complete with locked placement, for other designs.

The PlanAhead design methodology provides performance, productivity, and repeatability of results. With its hierarchical design flow, PlanAhead software allows you to reduce the number of iterations spent running PAR and then returning to RTL and synthesis. Instead, you can analyze your design and address issues on the physical side before implementation.

Faster Results in Less Time

PlanAhead users are consistently seeing a 10-15% performance improvement, with some achieving much higher results. In addition, designers are also finding that they can squeeze an additional 10% logic into a tight device. This combination of faster performance and better utilization can translate into a smaller, less expensive device, or design goals achieved with a slower speed grade.

PlanAhead design tools help reduce overall design time while adding a level of consistency in the results. You can perform design iterations in much less time – with repeatable results – by leveraging previous floorplans or incremental design techniques. You can also leverage successful results by locking them down or reusing them in other designs.

Tackling really tough performance issues requires more than just the addition of new menu items or scripting capabilities. PlanAhead software provides a complete environment to make this hierarchical methodology interactive and easy to use by presenting design data through the use of various views (see Figure 1). These independent views are designed to work in conjunction with each other, allowing you to quickly identify and navigate critical design objects and information.

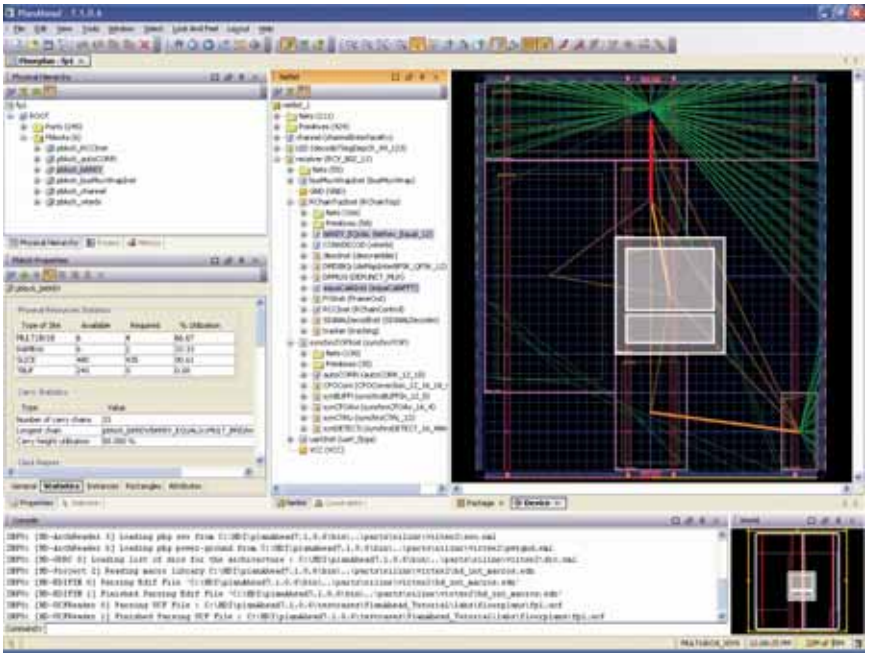


Figure 1 – PlanAhead software provides different views of the design to display physical hierarchy, properties, netlists and constraints, device package pins, schematics, and much more.

Visually Identify Performance Bottlenecks ...

The PlanAhead environment provides insight into the data flow of the design by displaying I/O interconnect as well as physical block (or "Pblock") net bundles. You can control color and line thickness of the bundles depending on the number of signals. This makes it easy to identify heavily connected Pblocks in the overall data flow through the design. You can then take corrective action to avoid routing congestion trouble spots and place heavily connected Pblocks close together, or merge them.

Clock regions are also displayed and can be used during floorplanning to optimize various clocks or minimize power usage within the device. By isolating clocks to specific clock regions, they can run faster and eliminate the need to power up other clock regions.

You can use the analysis and exploration environment of PlanAhead design tools at various stages in the design process. Initially, you can analyze the design before implementation. PlanAhead software provides a static timing engine, TimeAhead, that examines the design's feasibility relative to timing. You can also perform analysis using estimated routing delays by factoring in pure logic delays with no interconnect. This allows you to see how much timing tolerance is built into the design.

You can then edit and fine-tune timing constraints within the PlanAhead environment. These same analysis results can help determine what logic should be grouped together and floorplanned. Paths can be logically sorted, grouped, and selected for floorplanning. The same TimeAhead environment can also be

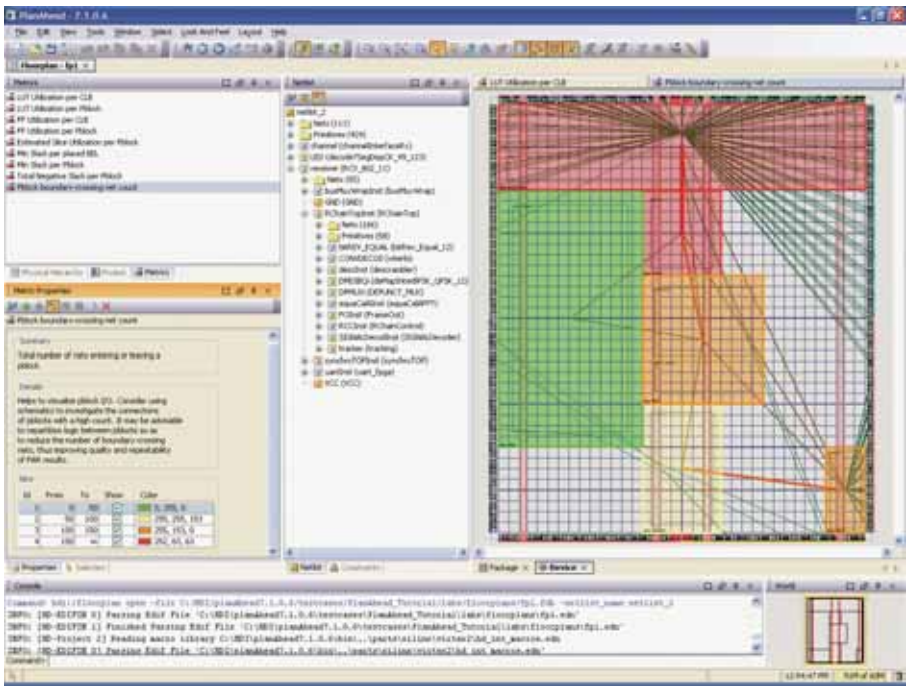


Figure 2 – Metric Maps provides a thermal metric display of the various potential problem areas in the design. The current metrics include utilization and timing checks at both the Pblock and placed design level.

leveraged with imported timing results from TRCE, the timing evaluation tool within Xilinx ISE™ software.

The timing constraints assigned to the design can be viewed and modified. You can define all ISE timing constraints as new constraints within the editor. This makes constraint assignment easier because you no longer have to remember specific constraint formats. You can use this with TimeAhead to validate and optimize the constraint set before running any ISE implementation tools.

PlanAhead design tools provide visual aides to help you comprehend the physical implementation results. Design rule checks (DRCs) are provided to catch errors early. It also flags designs that do not properly take advantage of certain device resources, such as the dedicated registers of the XtremeDSP™ slice or RAM within the Virtex™-4 FPGA.

By visualizing problem areas, you can address problems quickly, either in the RTL or on the physical implementation side, without having to continue RTL and synthesis iterations. The various logic modules can be selectively highlighted to better understand where they were placed, and Pblocks created where the logic is most concentrated. You can highlight failing timing paths to visualize and understand what is physically happening within your design.

PlanAhead software has includes metric maps to quickly identify problem areas of the design (Figure 2). These can be related to timing or utilization. This is helpful when trying to identify areas of the design to focus on for logic compression or timing connectivity.

PlanAhead design tools allow you to explore connectivity within the design. After selecting a particular net, Pblock, or instance within the design, you can highlight all of the nets connected to the selected elements with a single mouse click.

After an instance or Pblock is selected, all of the nets connecting to that element will be highlighted. This process can be continued, selecting and expanding the logic cone. Running "Show Connectivity" will highlight the next level of nets connected to the selected instances. This is an easy way to select a cone of logic starting at a particular instance or I/O port, taking real advantage of the design hierarchy.

... Then Address the Performance Problem

The whole idea is to provide a comprehensive environment to analyze timing issues and easily constrain that logic to avoid or correct it. You can use timing results from either TimeAhead or TRCE to drive a floorplan that will produce better performing designs by helping determine what logic should be grouped together and floorplanned.

Critical paths often traverse the logic hierarchy. PlanAhead software enables a physical hierarchy that is independent of the logic hierarchy, allowing logic from anywhere in the design to be grouped together and efficiently floorplanned.

PlanAhead software also provides resource utilization estimates to help size and shape Pblocks. These same statistics report clock information, carry chain, and RPM sizes for fit and a variety of other useful information.

PlanAhead design tools provide automatic floorplanning capabilities such as automatic partitioning based on the logic hierarchy and automatic Pblock sizing and placement. Because it is often difficult to encompass the required device resources within a single Pblock rectangle, non-recta-linear shapes can be created with multiple rectangles. PlanAhead software also allows you to create Pblocks within Pblocks, or "child" Pblocks, to help better maintain design hierarchy.

Device capacity can be improved by compressing the logic with Pblocks. This can be achieved in one of two ways. One is to use the Xilinx AREA_GROUP attribute called COMPRESSION. AREA_GROUP is a design implementation constraint that enables partitioning of the design into physical regions for mapping, packing, placement, and routing. Using the COMPRESSION attribute will cause the ISE Mapper to pack unrelated logic into unused CLB sites. Use this with care, as it can have an adverse affect on timing.

The best strategy for improving performance is to compress non-timing critical logic, thereby opening up more space in the device for timing critical logic. The second option is to use the PlanAhead capability to run PAR on Pblocks individ-

If necessary, you can even create nested Pblocks, creating a child/parent hierarchy to further constrain sub-modules for additional performance gains.

ually. You can continue to shrink the Pblock size until PAR fails. This will reduce and pack the logic as tightly as possible within the blocks and free up device space.

A Virtex-4 Floorplanning Example

PlanAhead design tools allow you to easily import placement and timing results. With this information, you can view and sort critical paths from the timing report and visualize paths using either the schematic or device views. Once you've identified failing paths, you can highlight all path instances on the floorplan to identify all path instances in the schematic view.

Figure 3 shows a floorplan of a design targeting a Virtex-4 FX140 device. In the display, we've highlighted the flip-flops along a particular path that was not able to meet timing. Because they are so widely distributed across the device, design implementation results in an unacceptably long delay. With the large number of clock domains available in Virtex-4 FPGAs, this is a common situation.

By selecting each of these flip-flops and restricting them to a single Pblock, you can then adjust and optimize the Pblock size and location to reduce delays on critical paths, as shown in Figure 4. If necessary, you can even create nested Pblocks, creating a child/parent hierarchy to further constrain sub-modules for additional performance gains. Depending on the resource requirements of the captured logic, you can lock down critical logic to locations for optimal access to necessary resources.

Conclusion

Visit *www.xilinx.com/planahead* to download a free evaluation of PlanAhead software today. This 30-day evaluation provides you with full access to all of the PlanAhead features and functionality. This site also allows you to view product demonstrations, download white papers, or just learn more.

Xilinx also offers PlanAhead QuickStart!, an exceptional level of support during the most critical phase of a project. With this service, you will receive a QuickStart! engineer at your site for one week who will train and empower your team to complete your project on time and make best use of the Xilinx device you have selected. This highly customizable service allows you to develop a training plan tailored specifically to the needs of your design team. This will help prevent schedule slips later in the project by ensuring that the team is skilled in the needed disciplines. It will also help you maintain a more effective and highly motivated team.

Figure 3 – Initial Virtex-4 FPGA floorplan with the path highlighted that did not initially meet timing

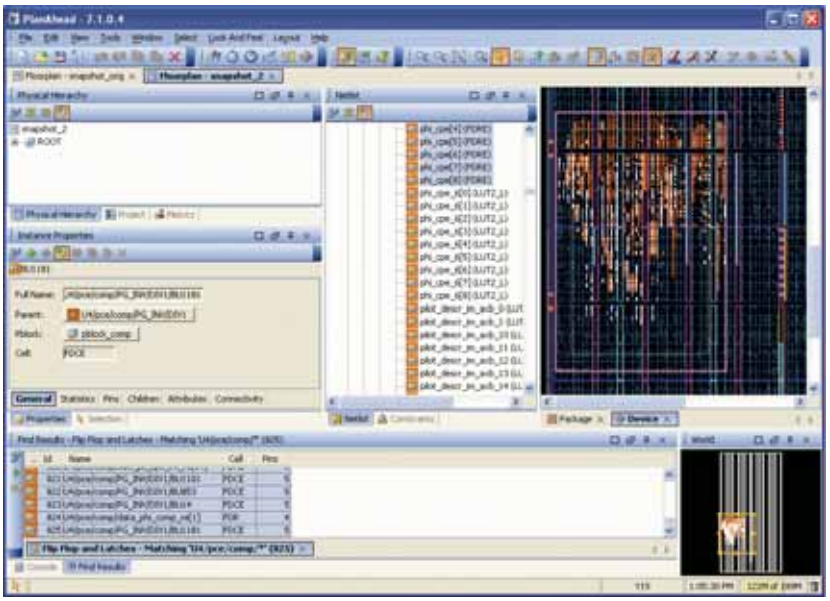


Figure 4 – After constraining all primitives associated with the path, you can optimize the Pblock to allow this path to achieve necessary timing.

Synthesis Tool Strategies

Set up your designs in Synplify for performance improvement and area savings.

by Steve Pereira
Technical Marketing
Synplicity, Inc
stevep@synplicity.com

You can benefit greatly from a proper synthesis strategy. Such strategies include knowing the final target architecture, knowing what coding problems could arise, and understanding what performance the periphery will require. You should also understand how to use IP. Are models available? Is cost an issue? Your initial setup can greatly affect productivity and help you achieve quicker peripheral timing closure.

In this article, I'll describe a known good strategy while using Synplify Pro tools.

Best Practices

Setting up your design correctly can result in huge performance increases or reductions in area. The following checklist describes the best practices to use when setting up your design.

1. Include any CoreGen EDIFs or timing models for black boxes. If you use black-box IP in the design, ensure that all EDIF, NGC netlists, or timing models are provided. It is essential that the Synplify Pro tool knows the timing requirements into and out of the box so that surrounding logic can be altered to reduce or remove criticality.

   If the design has an ngc file, use the ngc2edn (provided in the Xilinx® ISE™ /bin/ directory) converter utility to produce an edn file for synthesis.

2. Ensure that the device is correct and size the design. Ensuring that the wireload models are correct for synthesis can greatly affect the resulting logic. The selection of wireload models is simply a matter of selecting the correct device and speed grade. If synthesis is performed on a different device to the final implementation, sub-optimal results are quite likely.

   Selecting the correct device will also provide Synplify Pro with the accurate number of resources. One example of the importance of this is with block-select RAM mapping. Synplify orders all of the RAMs from biggest to smallest, and then starts to map the largest RAMs until there are no block RAMs left. The rest are placed into distributed RAM. If the wrong device is selected, sub-optimal mapping will occur.

   Sizing the design also has a significant impact. For example, if the design uses 80% of the device, the wireload models are correct for the design. If the design consumes only a small percentage of the total resources (for example, synthesizing just a part of the design for verification), the wireload models will be inaccurate. To resolve this problem, you can assign the logic to an area group. Please see the Synplify Pro documentation for instructions on how to do this.

3. Provide accurate clock constraints. Under- or over-constraining results in reduced performance. Do not over-constrain by more than 15%. For maximum performance, ensure that there is 10% negative slack on the critical clock. This ensures that critical paths are squeezed. The Fmax field on the front panel is fine for a quick run, but do not use it if you need maximum performance. Put unrelated clocks in separate clock groups in the Synplify Pro .sdc file. If your clocks are in the same group, the Synplify Pro tool works out the worst-case setup time for the clock-to-clock paths.

Figure 1 shows a timing diagram for two clocks that are in the same clock group. Synplify rolls the clocks forward until they match up again. The tool then calculates the minimum setup time between the clocks, in this case 10 ns.

If the clocks are unrelated, there may be several hundred clock periods before the clocks match up again. This may result in the worst-case setup time being very small (100 ps). You can check the setup time in the clock relationships table in the log file. If the setup time is too short, it is best to re-constrain the clocks so that they are more related.

4. Specify timing exceptions. Provide all timing exceptions, such as false and multicycle paths, to the Synplify Pro tool. With this information, the tool can ignore these paths and concentrate on the real critical paths.

5. Constrain I/Os. If the design has I/O timing constraints, it is likely that the critical path is through the I/O block (IOB). The Synplify Pro tool sees these paths as the most critical and tries to optimize them. Usually, I/O paths physically cannot be optimized any further; as they are the most critical, the Synplify Pro tool stops optimizing the rest of the design.

   A new switch has been added to the Synplify Pro 7.3 release called "use clock period for unconstrained I/O." When enabled, the tool does not include any unconstrained I/O paths in timing optimizations.

6. Keep code generic. Keeping code generic and not locked down to a particular architecture can aid design, reuse, and portability. For example, with the DSP48 block in the Virtex™-4 architecture, you can specify generic code to implement a DSP function. The tool will map to that component(s) when possible and reduce uncertainty regarding how the function was mapped. If DSP blocks are generated, timing is better known and can speed up debug – and time to market. You can specify the register configuration and code in the opcode to drive the DSP block configuration, all with generic code.

   If for some reason you wish to use a different device family, such as moving from Virtex-4 FPGAs to Virtex-II Pro FPGAs, the porting process itself should be seamless, but you may feel uncertain about implementation and logic levels.

Good Switch Settings

- Retiming and pipelining. Enabling retiming and pipelining options can improve your design performance by as much as 50%. Retiming attributes such as syn_allow_retiming let you refine your constraints by surgically applying retiming to a single register. Synplicity recommends that you enable both of these switches.
- Resource sharing. Always turn this switch on. The behavior of this switch was changed in Synplify 8.0. With the switch on, timing-driven resource sharing occurs. Non-critical logic will have resource sharing, but critical logic will not share resources (for performance reasons).
- FSM Compiler extracts and optimizes FSMs based on the number of states:
  - 2 to 4: sequential
  - 5 to 40: one-hot
  - More than 40: gray

  Synplicity also recommends enabling the FSM Compiler and setting default enumerated encoding to the "default" value for VHDL designs.
- FSM Explorer timing-driven state encoding. The Synplify Pro tool automatically selects the best encoding for the specified timing. This switch is design-dependent. If the critical path starts or ends at a state machine, turn the switch on.

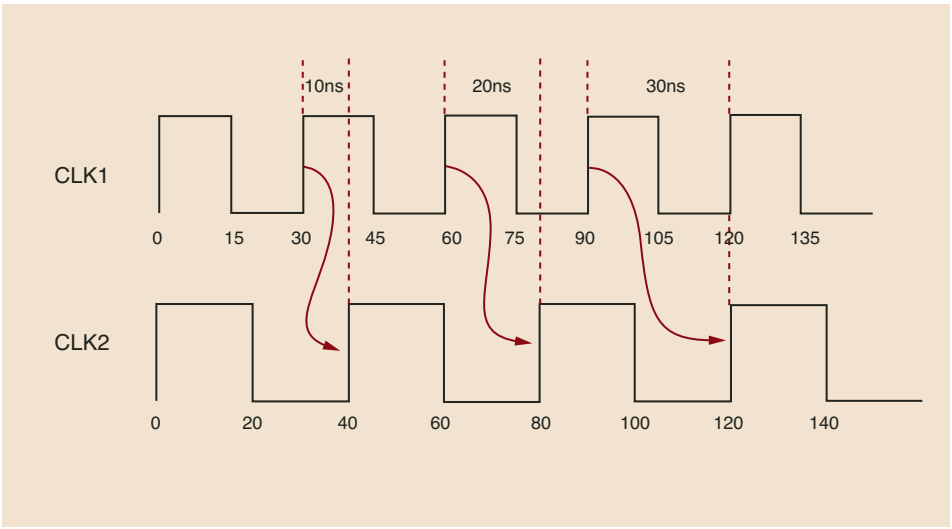


Figure 1 – Clock rolling for clocks in the same domain

Conclusion

Because synthesis tools are operating at higher levels of abstraction, synthesis optimizations can have a dramatic impact on design performance. After parsing through the HDL behavioral source code and extracting known functions (arithmetic functions, multiplexers, and memories), synthesis tools then map these functions on the target architecture features.

The tools trade off area and performance based on design constraints and tools settings; these influence the use of optimizations such as replication, merging, re-timing, and pipelining. As a result, the right tools settings in synthesis can greatly increase productivity and time to market.

PERFORMANCE

Tired of spinning in circles to meet timing on your FPGA designs?

Synplicity's **Synplify® Premier** solution solves FPGA timing closure through a unique graph-based physical synthesis technology providing highly accurate correlation to final timing and 5 to 20% better performance in a single-pass flow.

FPGA Solution

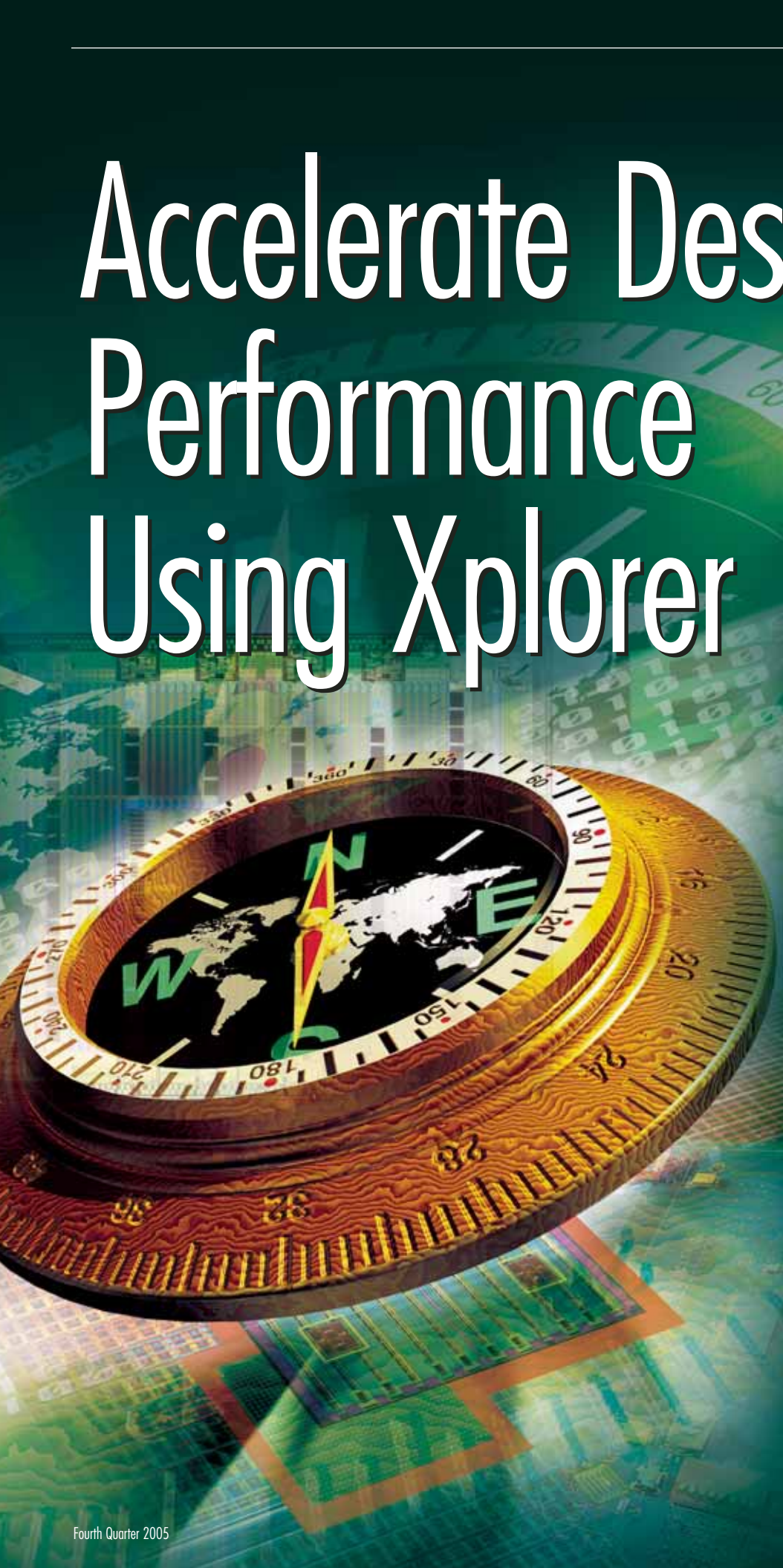
For more information on the Synplify Premier product and how it can help you quickly reach aggressive timing goals for your FPGAs, visit:
www.synplicity.com/products

Synplicity®

Simply Better Results

Accelerate Design Performance Using Xplorer

You can realize up to a 70% push-button performance improvement.



by Hitesh Patel
Senior Manager, Software Marketing
Xilinx, Inc.
hitesh.patel@xilinx.com

The 2005 EDA Branding Study shows that 71% of FPGA projects have difficulty meeting their timing budgets. Several strategies exist to help you meet your timing goals, such as HDL code changes and synthesis and implementation tools settings. In this article, we'll describe the Xplorer implementation tools strategy to maximize design performance, whether you are evaluating the best achievable performance for a specified clock domain or attempting to meet timing requirements for designs with user constraints.

Implementation with Xplorer

Xplorer is a perl script that seeks the best design performance using Xilinx® ISE™ software. After synthesis generates an EDIF (*.edf) file, the design is ready for implementation. During this phase, you could use Project Navigator in ISE software to apply design constraints and explore different tools settings for best performance.

An alternative approach may be to use Xplorer. Xplorer is designed to help achieve optimal results by employing smart constraining techniques and various physical optimization strategies. Because no unique set of ISE options or timing constraints works best on all designs, Xplorer finds the right set of tools options to either meet design constraints or find the best performance for the design. Hence, Xplorer has two modes of operation: best performance mode and timing closure mode.

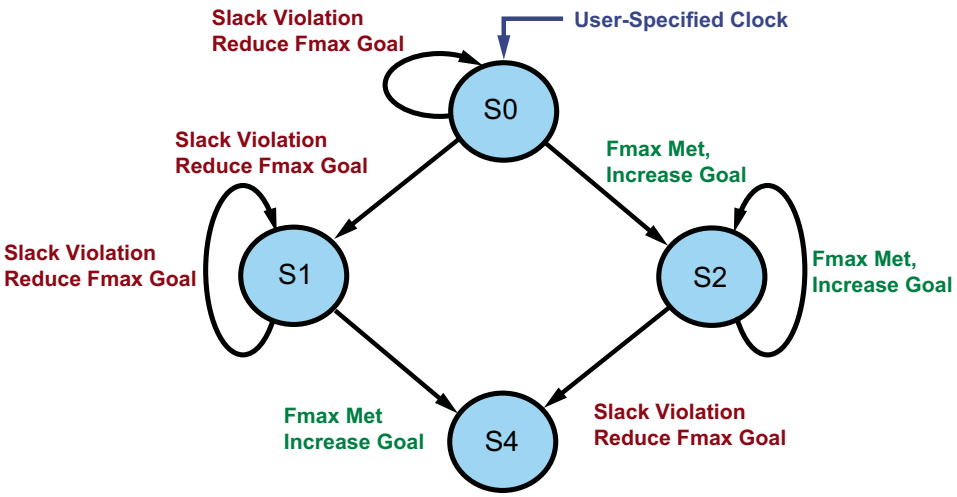


Figure 1 – Xplorer Best Performance Mode flow

Best Performance Mode

In this mode of operation, Xplorer optimizes design performance for a user-specified clock domain, allowing easy evaluation of the maximum achievable performance. You specify the design name and a single clock to optimize. Xplorer implements the design with different architecture-specific optimization strategies in conjunction with timing-driven place and route (PAR). It tightens or relaxes the timing constraints depending on whether or not the frequency goal is achieved, as shown in Figure 1. Xplorer estimates the starting frequency based on pre-PAR timing data. Adjusting timing constraints such that PAR is neither under- nor over-constrained enables Xplorer to deliver optimal design performance.

In addition to timing constraints, Xplorer also uses physical optimization strategies such as global optimization and timing-driven packing and placement. Global optimization performs pre-placement netlist optimizations on the critical region, while timing-driven packing and placement provides closed-loop packing and placement such that the placer can recommend logic packing techniques that deliver optimal placement. If the design has a user constraint file (UCF), Xplorer optimizes for the user constraints in addition to the specified clock domain.

Timing Closure Mode

If you have a design with timing constraints and your intent is for the tools to meet the specified constraints, use the timing closure mode. In this mode, you should not specify a clock using the -clk <clock name> switch. Xplorer looks at the UCF to examine the timing constraints goals. Using these constraints together with optimization strategies such as global optimization, timing-driven packing and placement, register duplication, and cost tables, Xplorer implements the design in multiple ways to deliver optimal design performance.

Because Xplorer runs approximately 10 iterations, you will experience longer PAR runtimes. However, Xplorer is something that users typically run once during their design cycle. After an Xplorer run, you can capture the set of options that will give the best result from the xplorer.rpt file and use that set of options for future design runs. Typically, designers will run the tools many times in a design cycle, so a longer initial runtime will likely reduce the number of PAR iterations later.

All Xilinx FPGA architectures are supported by Xplorer; optimizations are performed based on architecture features.

Using Xplorer

Xplorer is run from the command prompt by typing:

```
xplorer <design name> [-clk <clkname>] [-p <partname>]
```

<design name>: Name of the top level edif/ngc file.

-clk <clkname>: Name of the clock to be optimized. If the -clk option is omitted, the script uses the timespecs defined in the UCF file.

-p <partname>: The device name (for example, XC4VLX100-11FF1152). The default value is the part specified in the input design.

-uc <ucf file>: The UCF file name. The default value is <design name>.ucf.

Here is an example command for Best Performance Mode:

```
xplorer cordic -clk clk -p XC4VLX15-12FF668
```

Here is an example command for Timing Closure Mode:

```
xplorer <design name> -uc <ucf file> -p <partname>
```

Results, with the tools settings used, are summarized in xplorer.rpt. The best run is identified at the end of the report file.

These OpenCores designs are written in synthesizable RTL and synthesized to the target technology without any code modifications.

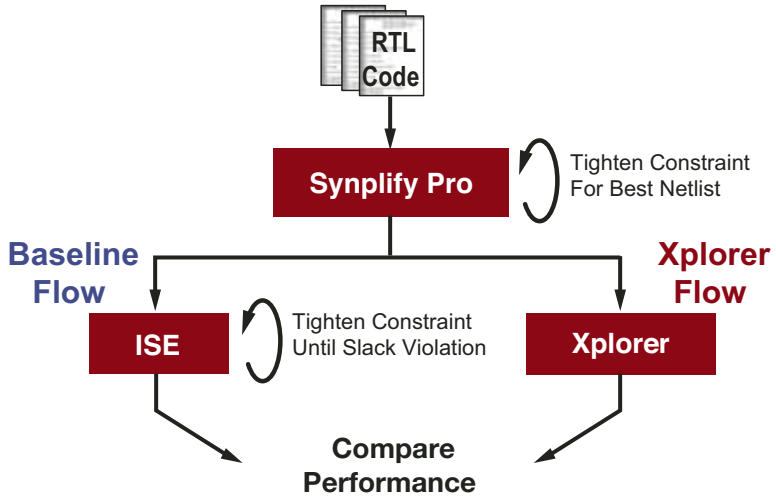


Figure 2 – Baseline and Xplorer flows

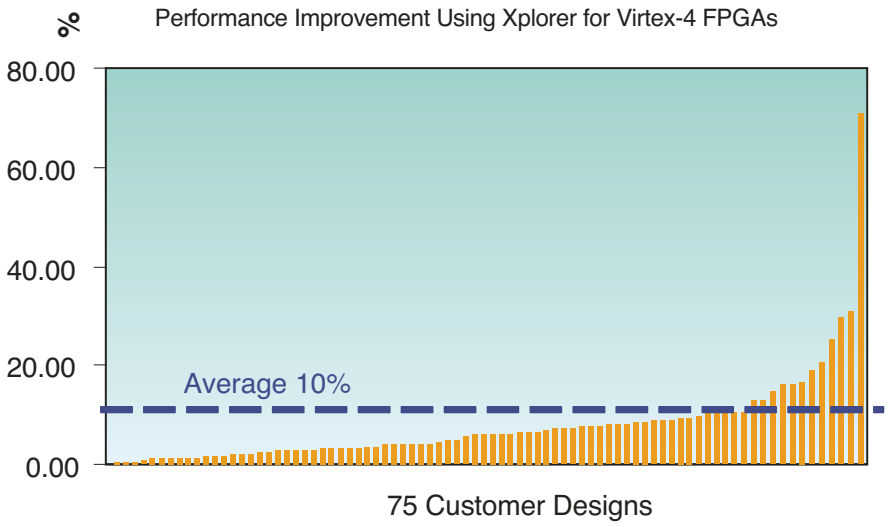


Figure 3 – Performance improvement for Virtex-4 FPGAs using Xplorer

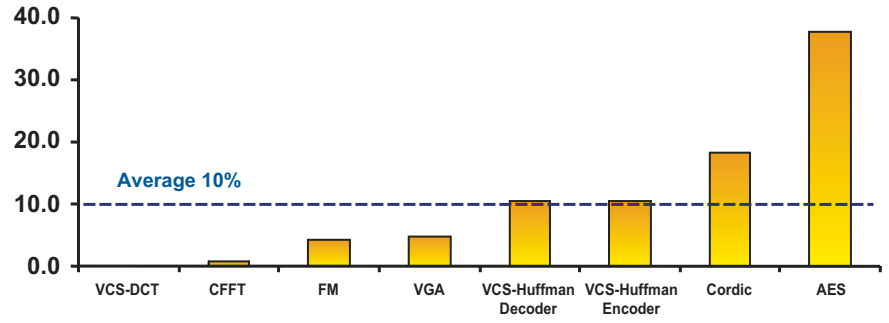


Figure 4 – Percentage performance improvement for Spartan-3 FPGAs using Xplorer on eight OpenCores designs

Performance Improvement Results

To highlight the performance impact of these optimization strategies, we compared baseline results (attainable using tightly constrained, high-effort timing-driven PAR) with Xplorer. Figure 2 shows the two flows.

For a high-density, high-performance Virtex™-4 customer design suite of more than 75 designs Xplorer provides up to 70% – and on average 10% – performance improvement, as shown in Figure 3. The designs range in density from LX15 to LX200, covering (but not limited to) market segments such as consumer, video, storage, telecom/datacom, DSP, and glue logic.

In addition, performance improvements for eight OpenCores designs for Spartan™-3 FPGAs are shown in Figure 4. These OpenCores designs are written in synthesizable RTL and synthesized to the target technology without any code modifications. The designs can be downloaded from *www.opencores.org*. For the eight OpenCores designs, the average performance improvement using Xplorer is 10%, with the AES design realizing a 38% performance improvement.

How to Achieve Additional Performance Gains

At times, the Xplorer implementation strategy still might not be enough to meet your target timing goals. In these cases, adopt synthesis and RTL coding strategies geared towards performance.

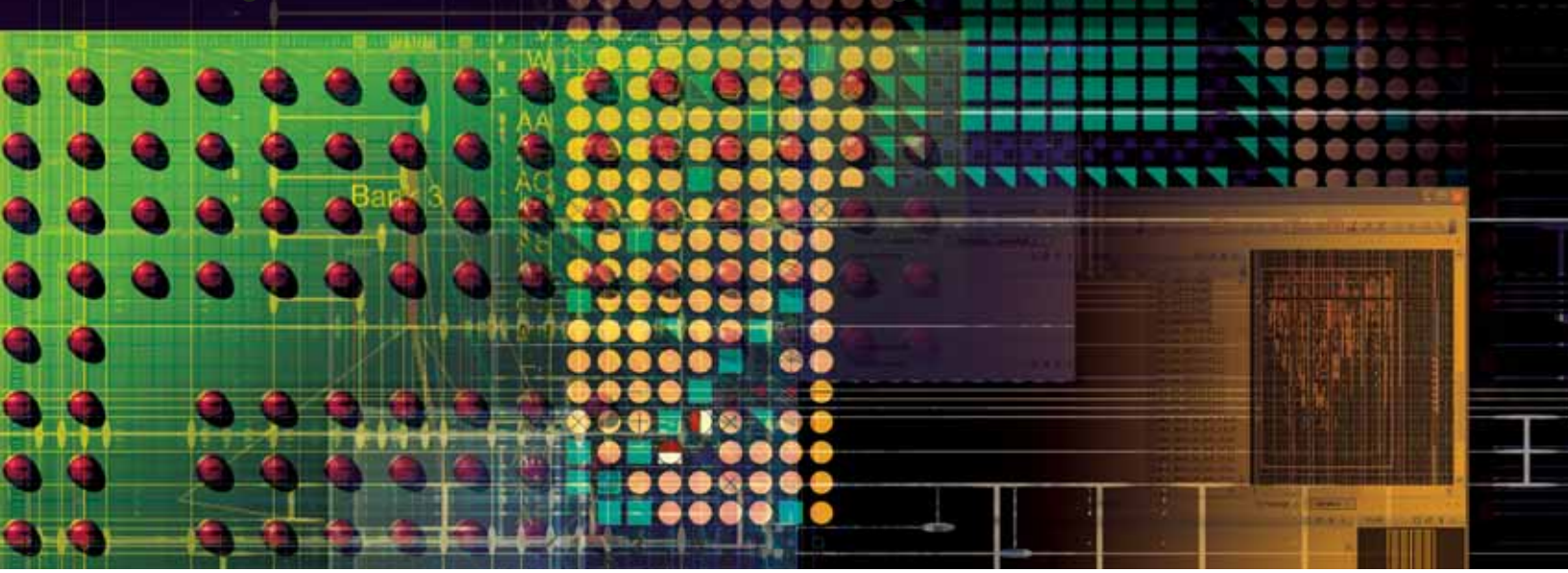
Conclusion

Xplorer helps you optimize logic performance and meet your timing goals by using smart constraining techniques and employing the right set of implementation tools strategies. Xplorer provides an average performance improvement of 10% for Xilinx FPGAs.

To view advanced options and to download Xplorer, visit *www.xilinx.com/xplorer*.

Achieve Your Performance Goals with PlanAhead Software

Obtaining the lowest cost FPGA solution through area and speed optimization.



by Bill Saperstein
Senior Director of Engineering
Anchor Bay Technologies, Inc.
ws@anchorbaytech.com

Sanjay Thatte
Product Marketing Manager
Xilinx, Inc.
sanjay.thatte@xilinx.com

The Xilinx® PlanAhead™ hierarchical design and analysis environment can be used in conjunction with Xilinx ISE™ tools to improve design performance and possibly enable incremental design and IP reuse. Several customers have benefited from the unique capabilities that PlanAhead software provides. In this article, we'll describe how one Xilinx customer, Anchor Bay Technologies of Campbell, California, was able to successfully utilize PlanAhead design tools.

A Custom Chip in Three Months

Anchor Bay Technologies specializes in designing and developing video processing system- and silicon-based solutions for scaling, de-interlacing, and noise reduction.

Recently, Denon Electronics Company required a very high-performance scaling chip for their high-end DVD players to take standard-definition 480P video from the MPEG decoder and scale it to 1080P resolution for large display applications. Anchor Bay had developed several scaling chips, but Denon required a custom design to fit their specific application. In particular, they wanted multiple video output streams, multiple video formats and resolutions, and a custom I^2C interface to the chip.

Faced with a very short development cycle and unable to turn an ASIC in this timeframe, Anchor Bay decided to develop a custom solution based on Xilinx Spartan™-3 FPGA technology. They had only three months to get the chip designed and tested for initial sampling.

Anchor Bay used ISE Foundation™ design tools to perform basic design and simulation. Because Denon required the lowest cost solution, they tried to fit the design into the smallest Spartan-3 device that had enough resources. But because of the aggressive performance requirements, achieving timing closure was close to impossible using conventional ISE floorplanning and place and route (PAR). The design involved four different clock domains – the highest frequency at 148 MHz. The design was using more than 80% of the Spartan-3 XC3S1000FT256-5 part, 100% of the multipliers and clock buffers, and 60% of the RAM blocks.

This heavy utilization made timing closure very difficult. There was no way to guide the tools adequately to close on the critical paths. In addition, the tools did not

clearly point out routing bottlenecks that were hindering timing closure.

After several attempts, Anchor Bay decided to explore the PlanAhead design tool from Xilinx to see if it could provide a solution. Their FAE support team was very responsive. They obtained an evaluation copy of PlanAhead software and quickly studied the tutorial before the FAEs came to their offices and walked them through the methodologies.

Problem Solved

PlanAhead design tools allowed Anchor Bay to quickly pinpoint the resource bottlenecks and the relationship between timing paths and placement. They were able to attempt several "what-if" scenarios to better open routing channels and group critical timing paths. Taking the results from the ISE timing analyzer and feeding them back into the floorplanning tool was invaluable. Also, the ability to view the schematic allowed them to change the logic where necessary to reduce the critical paths. They found that providing a simple floorplan was enough of a seed to allow the PAR tool to meet their timing needs.

They did not need to use PlanAhead software to heavily constrain the PAR tool, but only concentrated on the critical paths and congested areas. After more than two weeks of trying without success, they accomplished what they needed in two days using PlanAhead design tools.

Anchor Bay is continually pushing the limits of the FPGA devices in their systems. This allows them to not only sell ASIC solutions, but also to develop custom, cost-effective silicon solutions based on FPGA technologies.

Another Customer Example

Like Anchor Bay, a number of other customers have benefited from PlanAhead's advanced capabilities. For one such customer, the objective was to reduce PAR runtime and meet timing. To achieve this, they used PlanAhead software to quickly analyze the design, find the bottlenecks, and create the necessary physical constraints.

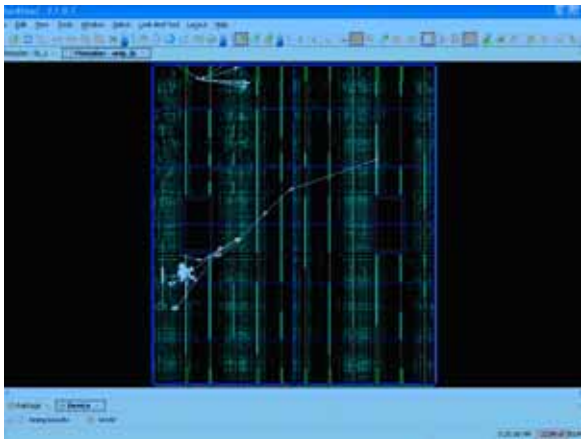


Figure 1 – Analyzing critical paths using PlanAhead software

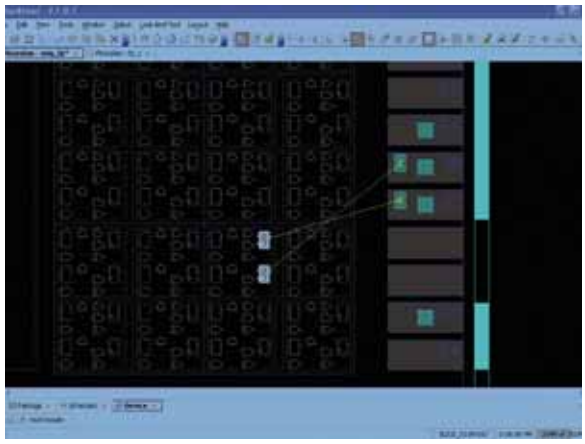


Figure 2 – Creating LOC constraints with PlanAhead design tools

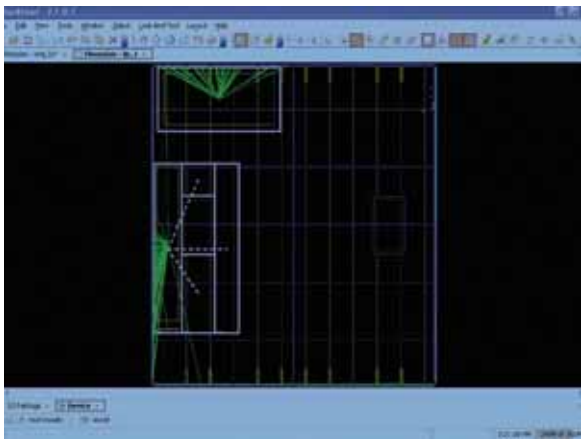


Figure 3 – Design floorplan created in PlanAhead software

	Without PlanAhead Software	With PlanAhead Software
Timing Errors	9	0
Timing Score	1746	0
Unmet Timing Constraints	3	0
PAR Runtime	234 min.	34 min.

Table 1 – Design results with and without PlanAhead design tools

PlanAhead's ability to place individual instances at specific locations and to constrain multiple instances to desired area groups was very useful in this process. Figure 1 shows a PlanAhead view showing the critical paths not meeting timing. Figure 2 shows how PlanAhead design tools were used to create specific LOC constraints. The final floorplan created using PlanAhead software is shown in Figure 3, while the results achieved through use of PlanAhead design tools are recorded in Table 1.

Conclusion

A growing number of customers are using PlanAhead software to help them tackle tough design problems. In doing so, they have increased their productivity while achieving and maintaining their design requirements. These benefits include:

- Reaching and maintaining performance goals
- Quicker incremental design changes
- Faster PAR time
- Fewer design iterations
- Tighter utilization control
- IP reuse

Getting started with PlanAhead software is easy. Visit *www.xilinx.com/planahead* to download a free, 30-day evaluation version, as well as additional information and an online demonstration. Also, customers interested in getting on-site design support can opt for the PlanAhead QuickStart! program.

We're With You
Right From The Start

www.xilinx.com/paq

PlanAhead™ QuickStart! provides a new level of personal, expert assistance to ensure your success. The solution includes configuration of the Xilinx ISE™ design environment, a comprehensive training plan, and a dedicated QuickStart! engineer on-site for one week to provide all the training you need to keep your project on time and on budget.

Your Complete Hierarchical Design Solution

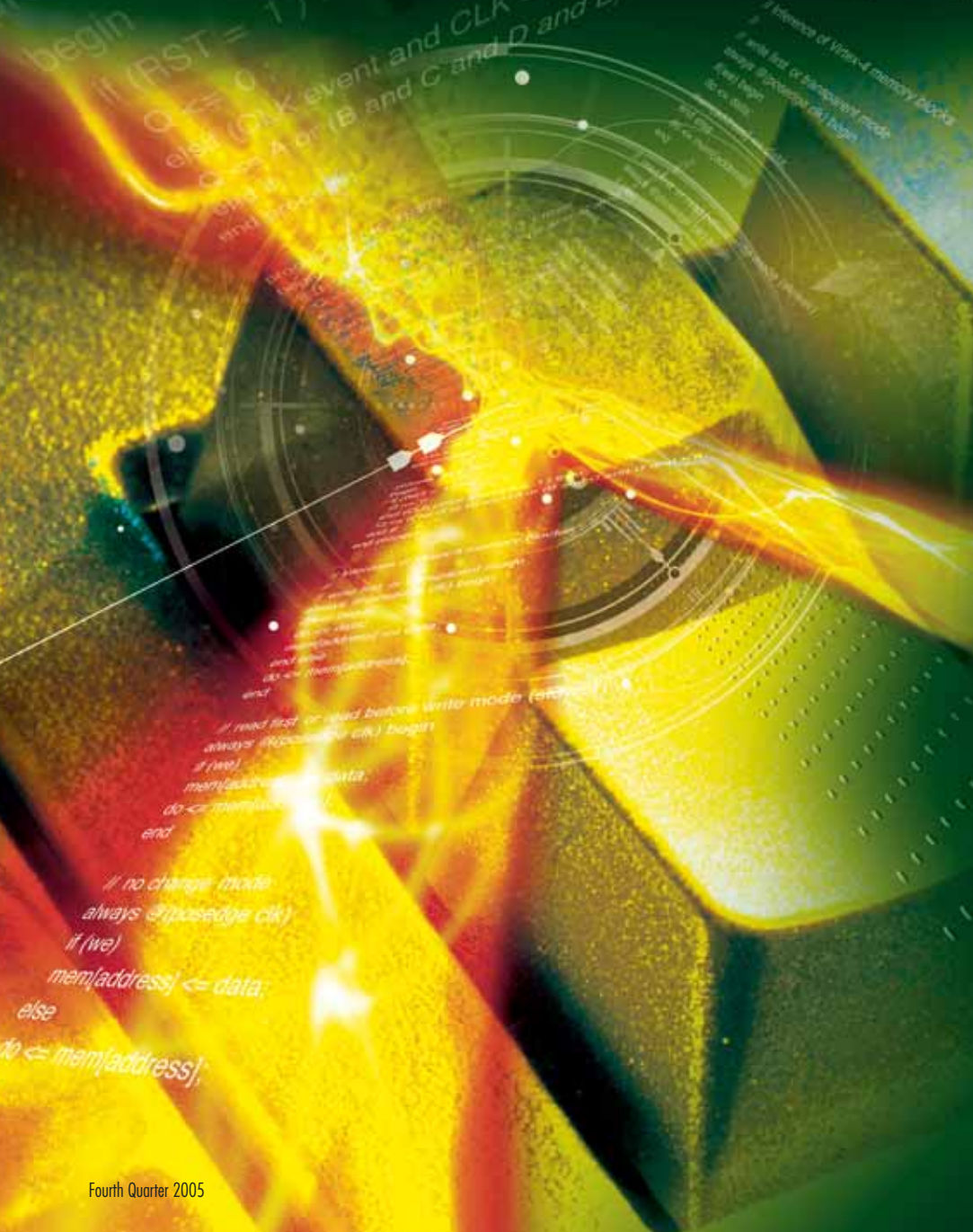
PlanAhead QuickStart! offers a complete system floorplanner solution to shorten development cycles and limit scheduling challenges. With PlanAhead QuickStart! your team has immediate design expertise to guarantee complete success…right from the start!

Contact your Xilinx representative or go to **www.xilinx.com/paq** for more information.

XILINX®

HDL Coding Practices to Accelerate Design Performance

Small code changes can make a big difference.



by Philippe Garrault
Technical Marketing Engineer
Xilinx, Inc.
philippe.garrault@xilinx.com

Brian Philofsky
Technical Marketing Engineer
Xilinx, Inc.
brian.philofsky@xilinx.com

You can achieve increases in design performance by selecting the right hardware platform and silicon features, being familiar with the device architecture, or having the proper settings and features in your implementation tools. But one of the most overlooked ways to increase design performance is to write HDL code that is very efficient for the targeted device. In this article, we'll present coding style tips to accelerate design performance.

Use of Resets and Performance

Few system-wide choices have as much of a profound effect on performance, area, and power as reset choice. Some system architects specify the use of a global asynchronous reset for the system. Whether it is truly needed or not, the ramifications of this choice are not always understood. With Xilinx® FPGA architecture, the use of and type of reset can have serious effects on the performance of your code.

SRLs

In all current Xilinx FPGA architectures, LUT (look-up table) elements are configurable as either logic, ROM/RAM, or a shift register (SRL, or shift register LUT). Synthesis tools can infer the use of any one of these structures from RTL code. However, in order to realize the use of the LUT as a shift register, a reset can not be described in the code, as the SRL does not have a reset. This means that shift registers coded with resets results in suboptimal implementation (requiring several flip-flops and the associated routing between them), while code without resets results in fast and compact implementation (using SRLs).

The effect on area and power is more obvious for these two cases, but the effect on performance is a little less clear. In general, a shift register built out of flip-flops is not going to be the critical path in a design because the timing path between registers is not normally long enough to be the longest path in the design. The added consumption of resources (flip-flops and routing) can have a negative influence on the placement and routing choices for other portions of the design, possibly resulting in longer routing paths.

Dedicated Multipliers and RAM Blocks

Multipliers are generally thought of for DSP designs. But because Xilinx FPGA architectures contain dedicated resources for multiplication, multipliers can be found in many types of designs, performing multiplication as well as other functions. Similarly, virtually every FPGA design uses RAMs of various sizes, regardless of the application.

Xilinx FPGAs contain several block RAM elements that can be used in a design as RAM, ROM, a large LUT, or even general logic. The use of both multipliers and RAM resources can result in more compact and higher performing designs, but reset choice can have either a positive or negative performance impact, depending on the type of reset used. Both RAM and multiplier blocks contain only synchronous resets; thus, if an asynchronous reset is coded for these functions, the registers within these blocks cannot be used. The effect this has on performance can be severe. For example, using a fully pipelined multiplier targeting Virtex™-4 devices with an asynchronous reset can result in a 200 MHz performance. Changing the code to a synchronous reset can more than double design performance to 500 MHz.

The issues with RAMs are twofold. Similar to the multipliers, Virtex-4 block RAMs have optional output registers which, when used, can reduce the clock-to-out times of the RAMs and increase overall design speed. These registers offer synchronous resets but not asynchronous resets, and thus cannot be used if the registers within the code describe an asynchronous reset.

A secondary issue comes to light when using the RAMs as a LUT or general logic. At times, it is advantageous for both area and performance reasons to condense several LUTs configured as ROM or general logic into a single block RAM. This can be done either by manually specifying these structures, or (in automated ways) mapping the portions of the logical design to unused block RAM resources. Because the block RAM has a synchronous reset, the mapping of general logic can occur without changing the specified functionality of the design – if a synchronous reset (or no reset) is used. If an asynchronous reset is described, this is not possible.

General Logic

Probably the least-known effect asynchronous resets have is on general logic structures. Because all Xilinx FPGA general-purpose registers contain the ability to program the set/reset as either asynchronous or synchronous, you might think that there is no penalty to use asynchronous resets. That assumption is often wrong. If an asynchronous reset is not used, the set/reset logic can be configured as synchronous logic; if so, this frees up added resources for logic optimization. To illustrate how asynchronous resets can inhibit optimization, let's look at the following suboptimal code examples:

VHDL Example #1

```
process (CLK, RST)
begin
  if (RST = '1') then
   Q <= '0';
  elsif (CLK'event and CLK = '1') then
   Q <= A or (B and C and D and E);
  end if;
end process;
```

Verilog Example #1

```
always @(posedge CLK, posedge RST)
  if (RESET)
   Q <= 1'b0;
  else
   Q <= A | (B & C & D & E);
```

To implement this code, the synthesis tool has no choice but to infer two LUTs for the data path, because there are five signals used to create this logic. A possible implementation of the above code would look like Figure 1.

If, however, this same code is re-written for a synchronous reset, as in the following

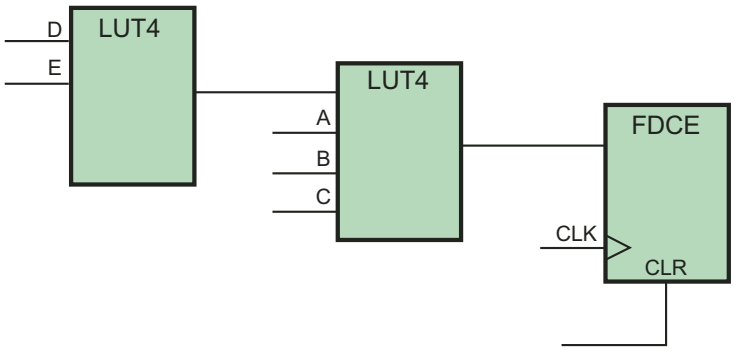


Figure 1 – Synthesis infers two LUTs

examples of corrected code with reduced area and improved performance:

VHDL Example #2

```
process (CLK)
begin
 if (CLK'event and CLK = '1') then
  if (RST = '1') then
   Q <= '0';
  else
   Q <= A or (B and C and D and E);
  end if;
 end if;
end process;
```

Verilog Example #2

```
always @(posedge CLK)
 if (RESET)
  Q <= 1'b0;
 else
  Q <= A | (B&C&D&E);
```

The synthesis tool now has more flexibility as to how this function can exist. A possible implementation of the preceding code would look like Figure 2.

In this implementation, the synthesis tool can identify that any time A is active high, Q is always a logic one (the OR function). With the register now configured with the set/reset as a synchronous operation, the set is now free to be used as part of the synchronous data path. This reduces the amount of logic necessary to implement the function, as well as reducing the data path delays for the D and E signals from the previous example. Logic could have also been shifted to the reset side as well, if the code was written in a way that was a more beneficial implementation.

Consider the following addition to these examples:

VHDL Example #3

```
process (CLK, RST)
begin
 if (RST = '1') then
  Q <= '0';
 elsif (CLK'event and CLK = '1') then
  Q <= (F or G or H) and (A or (B and C
  and D and E));
 end if;
end process;
```

Verilog Example #3

```
always @(posedge CLK, posedge RST)
 if (RESET)
  Q <= 1'b0;
 else
  Q <= (F|G|H) & (A | (B&C&D&E));
```

Now that there are eight signals that contribute to the logic function, a minimum of three LUTs would be needed to implement this function. A possible implementation of the above code would look like Figure 3.

If the same code is written with a synchronous reset:

VHDL Example #4

```
process (CLK)
begin
 if (CLK'event and CLK = '1') then
  if (RST = '1') then
   Q <= '0';
  else
   Q <= (F or G or H) and (A or (B and C
   and D and E));
  end if;
 end if;
end process;
```

Verilog Example #4

```
always @(posedge CLK)
 if (RESET)
  Q <= 1'b0;
 else
  Q <= (F|G|H) & (A | (B&C&D&E));
```

A possible implementation of the above code would look like Figure 4. Again, the resulting implementation not only uses fewer LUTs to implement the same logic function, but also could potentially result in a faster design because of the reduction of logic levels for practically every signal that creates this function.

These examples are simple, but they do illustrate our point of how asynchronous resets force all synchronous data signals on the data input to the register, thus resulting in possibly more logic levels and less optimal implementation. In general, the more signals that fan into a logic function, the more effective the use of synchronous sets/resets (or no resets at all) in minimizing logic resources or maximizing design performance.

Adder Chains Instead of Adder Trees

Many signal processing algorithms perform an arithmetic operation on an input stream of samples, followed by a summation of all outputs of this arithmetic operation. The

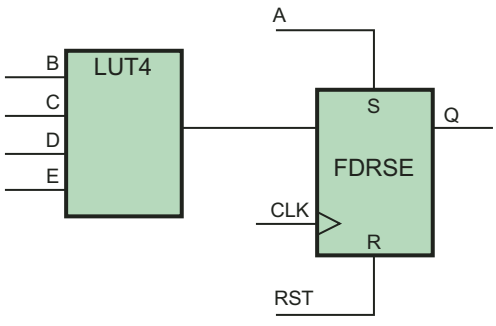


Figure 2 – More flexible LUT inference

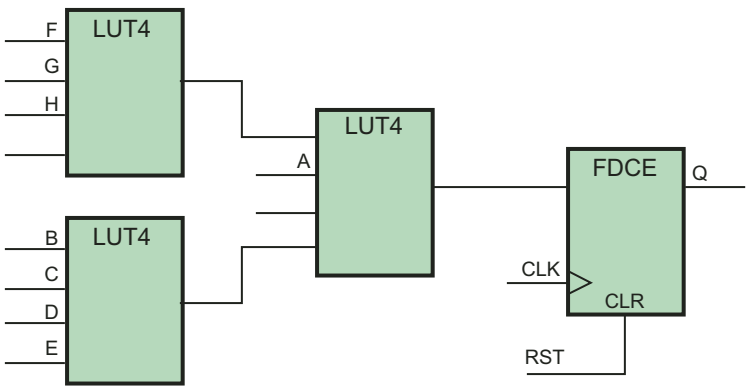


Figure 3 – Synthesis infers three LUTs

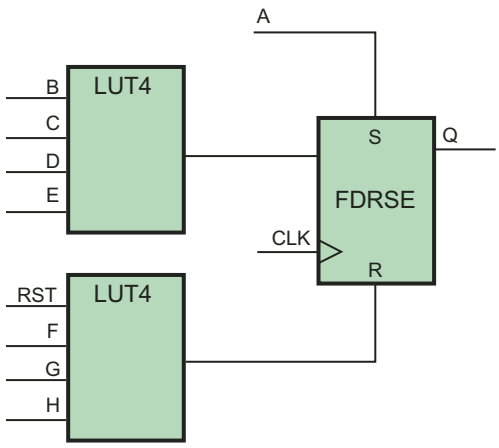


Figure 4 – Inferred with synchronous reset

adder tree structure is typically used to implement the summation in parallel architectures such as FPGAs.

One difficulty with the adder tree concept is the varying nature of its size. The number of adders is dependent on the number of inputs in the adder tree. The more inputs in the adder tree, the more adders you need, which increases both the number of logic resources and power consumption. Larger trees also mean larger adders in the last stages of the tree, which further reduces system performance.

To reduce power consumption and maintain high performance, adder trees should be implemented as dedicated silicon resources. But placing a number of fixed-size adder tree components in silicon is not efficient because you would have to use logic resources when the fixed number of additions is exceeded or even go to a larger FPGA, thereby increasing the cost of the device.

With its columns of DSP48 dedicated silicon, the Virtex-4 device family takes a different approach in implementing summations. It involves computing the summation incrementally using chained adders instead of adder trees. This approach is a departure from any existing FPGA and is key to maximizing performance and lowering power for DSP algorithms because both logic and interconnect are contained entirely within the dedicated silicon.

When pipelined, performance of the DSP48 block is 500 MHz – independent of the number of adders. As illustrated in Figure 5, cascading ports combined with the 48-bit resolution of the adder/accumulator allow computing of the current sample calculation, along with the summation of all computed samples so far.

To take advantage of the Virtex-4 adder chain structure in the RTL, simply replace the adder tree description with an adder chain description. This process of converting a direct form filter to a transposed or systolic form is detailed in the XtremeDSP Design Considerations User Guide.

Once the conversion is complete, you may find that the algorithm runs much faster than your application needs. In that case, you could further reduce device utilization and power consumption by using either

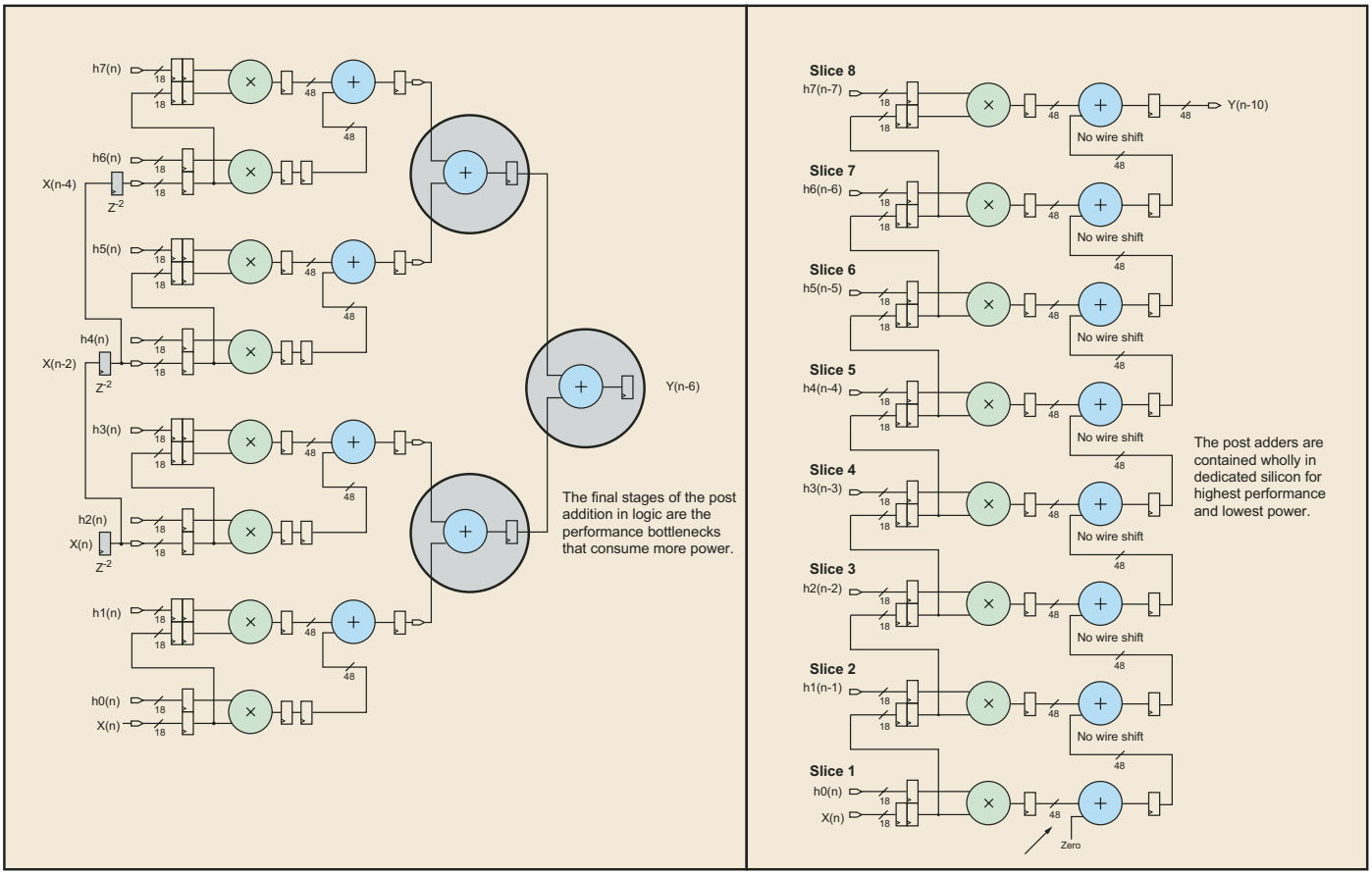


Figure 5 – Chaining adders provide predictable performance

folding or multi-channeling techniques. Both techniques help implement designs in smaller devices or allow you to add functionality to a design using the freed resources.

Multi-channeling is a process that leverages very fast math elements across multiple input streams (channels) with much lower sample rates. This technique increases silicon efficiency by a factor almost equal to the number of channels. Multi-channel filtering can be looked at as time-multiplexing single-channel filters. For example, in a typical multi-channel filtering scenario, multiple input channels are filtered using a separate digital filter for each channel. Taking advantage of the Virtex-4 DSP48 block, you could use a single digital filter to filter all eight input channels by clocking the single filter with an 8x clock. This reduces the number of FPGA resources needed by almost 8x.

Maximize Block RAM Performance

When inferring memory elements, factors affecting performance include:

- using dedicated blocks or distributed RAMs
- using the output pipeline register
- not using asynchronous resets

There are also a couple of lesser known areas – HDL coding style and synthesis tool settings – that can substantially impact memory performance.

HDL Coding Style

When inferring dual-port block memories, it is possible that both ports could try to access the same memory cell at the same time. If both ports are simultaneously writing different values at the same memory cell, this creates a collision and the memory cell content cannot be guaranteed. But what happens if one port reads while the other port is writing at the same address? Well, it depends on the target device. The latest Virtex and Spartan™ families have three programmable operating modes to govern memory output while a write operation is occurring. Additional information about these operating modes is provided in the device user guides.

Note that the different modes affect how the memory outputs behave and also affect the performance of the memory. As illustrated in the following example, your coding style determines in which mode the memory is operating:

```
// Inference of Virtex-4 memory blocks
//
// 'write first' or transparent mode
always @(posedge clk) begin
 if(we) begin
  do <= data;
  mem[address] <= data;
 end else
  do <= mem[address];
end

// 'read first' or read before write mode (slower)
always @(posedge clk) begin
 if (we)
  mem[address] <= data;
 do <= mem[address];
end

// 'no change' mode
always @(posedge clk)
 if (we)
  mem[address] <= data;
 else
  do <= mem[address];
end
```

Add Pipeline Levels

Another way to increase performance is to restructure long data paths made of several levels of logic, breaking them up over multiple clock cycles. This method allows for a faster clock cycle and increased data throughput, at the expense of latency and pipeline management overhead logic. Because FPGAs are register-rich, the additional registers and overhead logic are usually not an issue.

Because the data is now on a multicycle path, you must use special considerations for the rest of the design to account for the added latency. The following example presents a coding style to add five levels of registers on the output of a 32 x 32 multiplier. The synthesis tool will pipeline these registers to the registers available in the Virtex-4 DSP48 block so as to maximize data throughput.

```
// 32x32 multiplier with 4 DSP48 (PIPE=5)
always @(posedge clk) begin
  prod[0] <= a * b;
  for (i=1; i<=PIPE-1; i=i+1)
        prod[i] <= prod[i-1];
end
```

Nests in the Code

Try not to make too many nests in the code, such as nested if and case statements. If you have too many if statements inside of other if statements, it can make the line length too long, as well as inhibit synthesis optimizations. By following this guideline, your code is generally more readable and more portable.

When describing "for-loops" in HDL, it is preferable to place at least one register in the data path, especially when there are arithmetics or other logic-intensive operations. During compilation, the synthesis tool will unroll the loops. Without these synchronous elements, it will concatenate logic created at each iteration of the loop, resulting in a very long combinatorial path that may limit design performance.

Conclusion

Recent advances in synthesis and place and route algorithms have made achieving the best performance out of a particular device much more straightforward. Synthesis tools are able to infer and map complex arithmetics and memory descriptions onto the dedicated hardware blocks. They will also perform optimizations such as retiming and logic and register replications. Based on timing constraints, the place and route tool can now restructure the netlist and perform timing-driven packing and placement to minimize placement and routing congestions.

However, today (just as yesterday), there is only so much the tools can do to maximize performance. If you need more performance out of your design, then a very efficient way to proceed is by learning more about the target device, the synthesis tool, and by using the coding guidelines illustrated in this article.

Writing RTL Code for Virtex-4 DSP48 Blocks with XST 8.1i

Writing RTL code for your DSP applications is easy and efficient.

by Edgard Garcia
Xilinx Consultant/Designer
Multi Video Designs
edgard.garcia@mvd-fpga.com

The Xilinx® Virtex™-4 family introduced a new high-performance concept for fast and complex DSP algorithm implementation. The XtremeDSP™ Design Considerations User Guide, available on the Xilinx website (*www.xilinx.com/bvdocs/userguides/ug073.pdf*), describes how you can take advantage of the DSP48 architecture and includes several examples.

When you have to develop a real DSP application, you can of course instantiate each DSP48 block and assign their respective attribute values to obtain the correct behavior. But did you know you can also infer most of the useful DSP48 configurations by writing very simple RTL code?

Developing DSP algorithms in VHDL (or Verilog) is a nice way to maintain designs over a long period of time, but the synthesis results must meet your performance requirements. In this article, I will show you how to write RTL code to take full advantage of Virtex-4 DSP48 blocks.

DSP48 Architecture

The Virtex-4 DSP48 architecture is extensively described in the XtremeDSP User Guide. Let's start, however, with an overview of some very important aspects of DSP48 blocks:

- DSP48 blocks have two 18-bit inputs to feed the multiplier. If you want to work with unsigned data, 17 bits is the maximum width of the multiplier inputs. Don't forget to expand the unsigned data/coefficients by concatenating one or more '0' to the most significant bit (MSB). Similarly, if using the adder/subtracter, its inputs and output will have to be 48 bits or less for signed arithmetic and 47 bits or less for unsigned.

For the examples described in this article, we will use signed data. You will have to use the IEEE.STD_LOGIC_SIGNED package.

- Another important parameter for describing DSP behavior for Virtex-4 DSP48 blocks is that all DSP48 internal registers have a synchronous reset (using asynchronous reset will prevent the synthesis tool from using the DSP48 internal registers). The reset functionality has priority, regardless of OpCode or other control inputs.

- It is important to note that the last stage of the adder/subtracter can be driven dynamically to take a 48-bit input (from the output stage feedback or from the DSP48 C or Pcin input) and to add or subtract another 48- or 36-bit input (originating for most common cases from the multiplier output).

Basic Examples

1. Multiplier_accumulator. This commonly used function is our first example, useful for FIR filters and other DSP functions. Here is the source code:

```
library IEEE;              use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;                  — Signed arithmetic is used

entity MULT_ACC is
   Port ( CK : in std_logic;
          RST : in std_logic;                   — Synchronous reset
          Ain, Bin : in std_logic_vector(17 downto 0);   — A and B inputs of the multiplier
          S : out std_logic_vector(47 downto 0)      );  — Accumulator output
end MULT_ACC;

architecture Behavioral of MULT_ACC is

signal ACC : std_logic_vector(47 downto 0);           — Accumulator output

begin

process(CK)  begin
          if CK'event and CK = '1' then
                    if RST = '1' then       ACC <= (others => '0');
                    else                    ACC <= ACC + (AIN * BIN);
                    end if;
          end if;
end process;

S <= ACC;

end Behavioral;
```

This example will be synthesized into a single DSP48 block – no other logic resource is necessary. The performance is about 180-200 MHz, depending on placement and routing.

2. Fully pipelined Multiplier_accumulator. If you need more performance and less dependency on place and route tools, you can still improve the performance of the Multiplier_accumulator. The DSP48 blocks have internal input registers (zero, one, or two stages for A and B inputs), as well as one selectable multiplier output register. The following RTL code uses one level of registers at the A and B inputs, as well as the multiplier output register:

```
library IEEE;              use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;                  — Signed arithmetic is used

entity MULT_ACC is
   Port ( CK : in std_logic;
          RST : in std_logic;                   — Synchronous reset
          Ain, Bin : in std_logic_vector(17 downto 0);   — A and B inputs of the multiplier
          S : out std_logic_vector(47 downto 0)      ); — Accumulator output
end MULT_ACC;

architecture Behavioral of MULT_ACC is

signal AinR, BinR : std_logic_vector(17 downto 0);     — Registered Ain and Bin
signal MULTR : std_logic_vector(35 downto 0);          — Registered multiplier output
signal ACC : std_logic_vector(47 downto 0);            — Accumulator output

begin

process(CK)  begin
          if CK'event and CK = '1' then
                    if RST = '1' then       AinR <= (others => '0');
                                            BinR <= (others => '0');
                                            MULTR <= (others => '0');
                                            ACC <= (others => '0');
                    else                    AinR <= Ain;
                                            BinR <= Bin;
                                            MULTR <= AinR * BinR;
                                            ACC <= ACC + MULTR;
                    end if;
          end if;
end process;

S <= ACC;

end Behavioral;
```

This example will be synthesized by using just a single DSP block. You can take advantage of the internal registers to greatly improve performance to more than 400 MHz for the slowest Virtex-4 speed grade, independent of the implementation (place and route) tools.

3. Fully pipelined Loadable_Multiplier_accumulator. You can improve the design further by using a loadable multiplier accumulator. For more details, please refer to the class material of the Xilinx course, "DSP Implementation Techniques for Xilinx FPGAs" (*www.xilinx.com/support/training/abstracts/dsp-implementation.htm*). Let's modify the previous code for the load functionality:

```
library IEEE;              use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;                  — Signed arithmetic is used

entity MULT_ACC_LD is
   Port ( CK : in std_logic;
          RST : in std_logic;                   — Synchronous reset
          Ain, Bin : in std_logic_vector(17 downto 0);   — A and B inputs of the multiplier
          LOAD : in std_logic;                  — Active high LOAD command
          S : out std_logic_vector(47 downto 0)      ); — Accumulator output
end MULT_ACC_LD;

architecture Behavioral of MULT_ACC_LD is

signal AinR, BinR : std_logic_vector(17 downto 0);     — Registered Ain and Bin
signal MULTR : std_logic_vector(35 downto 0);          — Registered multiplier output
signal ACC : std_logic_vector(47 downto 0);            — Accumulator output

— 48 bit "ZERO" constant used for MULTR sign extension to 48 bits
constant ZERO : std_logic_vector(47 downto 0) := (others => '0');

begin

process(CK)  begin
          if CK'event and CK = '1' then
                    if RST = '1' then   AinR <= (others => '0');
                                        BinR <= (others => '0');
                                        MULTR <= (others => '0');
                                        ACC <= (others => '0');
                    else                AinR <= Ain;
                                        BinR <= Bin;
                                        MULTR <= AinR * BinR;
                                    if LOAD = '1' then
                                            ACC <= ZERO + MULTR; — OpCode = x05
                                    else
                                            ACC <= ACC + MULTR;  — OpCode = x25
                                    end if;
                    end if;
          end if;
end process;

S <= ACC;

end Behavioral;
```

4. Multiplier_accumulator_or_adder. This is another useful version of the multiplier accumulator. It is useful for multiplications of data buses of more than 18 bits (see Figure 1-18 in the XtremeDSP User Guide). Here is the RTL code:

```
library IEEE;                use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;                — Signed arithmetic is used

entity MULT_ACC_ADD is
   Port ( CK : in std_logic;
          RST : in std_logic;
          SEL : in std_logic;
          A_in, B_in : in std_logic_vector(17 downto 0);
          C_in : in std_logic_vector(47 downto 0);
          S : out std_logic_vector(47 downto 0));
end MULT_ACC_ADD;

architecture Behavioral of MULT_ACC_ADD is

constant ZERO : std_logic_vector(47 downto 0) := (others => '0');

signal AR, BR : std_logic_vector(17 downto 0);
signal MULT : std_logic_vector(35 downto 0);
signal Pout : std_logic_vector(47 downto 0);

begin

process(CK)  begin
    if CK'event and CK = '1' then
            if RST = '1'  then     AR <= (others => '0');
                                   BR <= (others => '0');
                                   MULT <= (others => '0');
                                   Pout <= (others => '0');
            else
                   AR <= A_in;
                   BR <= B_in;
                   MULT <= AR * BR;

                   if SEL = '0' then Pout <= C_in + MULT;  — Opcode = 0x35 for C input
                                                           —  0x15 for PCIN input
—                  if SEL = '0' then Pout <= ZERO + MULT; — Opcode =  0x05 for ZERO
                                                           — constant as input (Note 1)
                  else    Pout <= Pout + MULT;  — Opcode = 0x25 (Notes 2, 3)
                  end if;

            end if;
    end if;
end process;

S <= Pout;

end Behavioral;
```

Note that the synthesis results are not currently as optimized as we could expect with XST 8.1. Some combinatorial logic will be used to implement the multiplexer between C_in and Pout, while the same function was available inside the DSP48 block. The performance is still 220 MHz for the -10 speed grade, and 270+ MHz for -12. However, Synplify Pro 8.2 provides the ideal implementation with the same RTL code.

Note 1 : Adding ZERO to Pout is equivalent to the previously described load function.

Note 2 : You can also use the 17-bit right shift on Pout by changing this line as follows (at this time, this feature is supported only by Synplicity Synplify Pro 8.2):

```
else Pout <= ZERO + Pout(47 downto 17) + MULT;
```

Note 3 : If for any reason you do not want to use the output register of the multiplier, you can write:

```
Pout <= Pout + (AR * BR);
```

instead of declaring a combinatorial multiplier output. The resulting RTL code is also more compact.

5. Symmetric rounding. Another simple but useful example is a multiplier with symmetric rounding (see Table 1-9 in the XtremeDSP User Guide). Assuming that you want to round the result of the multiplication Ain x Bin to 20 bits, the following RTL code will be synthesized in just one DSP48 block and one slice:

```
library IEEE;                use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

entity ROUNDING is
   Port ( CK : in std_logic;
          RST : in std_logic;
          Ain, Bin : in std_logic_vector(17 downto 0);
          P : out std_logic_vector(19 downto 0));
end ROUNDING;

architecture Behavioral of ROUNDING is

constant ZERO : std_logic_vector(47 downto 0) := (others => '0');

signal AR, BR : std_logic_vector(17 downto 0);
signal MULTR : std_logic_vector(35 downto 0);
signal Pout : std_logic_vector(47 downto 0);

signal Carry_in, Carry_inR : std_logic;

begin

process(CK)  begin
            if CK'event and CK = '1'  then
                        Carry_in <= not(Ain(17) xor Bin(17));
                        Carry_inR <= Carry_in;
                        if RST = '1'  then          AR <= (others => '0');
                                                    BR <= (others => '0');
                                                    MULTR <= (others => '0');
                                                    Pout <= (others => '0');
                        else
                                    AR <= Ain;
                                    BR <= Bin;
                                    MULTR <= AR * BR;
— Note that the following 4 operands adder will be implemented as a 3 operand one :
— ZERO is a constant that allows easy sign extension for the VHDL syntax
                                    Pout <= ZERO + MULTR + x"7FFF" + Carry_inR;
                        end if;
            end if;
end process;

P <= Pout(35 downto 16);

end Behavioral;
```

This example will also work at 400 MHz for the Virtex-4 -10 speed grade device and 500 MHz for the -12 speed grade device. Only one LUT and its associated slice flip-flop is used, as the second flip-flop is pushed inside the DSP48 block for carry input.

All of these examples can be used in a wide range of applications. You can see that they are very efficiently synthesized, and all of the logic is mapped into the DSP48 blocks. The performance for each of these DSP functions is independent of the place and route tools.

To make it easier for synthesis tools to recognize the DSP48 struc-

ture, it is important to write the code in a simple way, giving your tools the best option to pack your desired functions into each DSP48 block. For this reason, each code has been written in a single process.

The more simple and compact your RTL code, the more efficient the synthesis result. Of course, depending on your synthesis tool, other alternatives can also give you excellent results, but they will be more dependent on the synthesis tools.

Higher Complexity Designs

What happens when you need more complex DSP functions? You can use a similar approach for many complex DSP algorithm implementations by describing each block separately to ensure optimal synthesis results.

You will find many other examples, most of them directly related to those explained in their algorithmic and schematic form, in the XtremeDSP User Guide.

Conclusion

This article is excerpted from the application note, "Virtex-4 DSP48 Inference," which is available at *www.mvd-fpga.com/en/publi_V4_DSP48.html*.

The application note includes additional examples, such as:

- Single DSP slice 35 x 18 multiplier (Figure 1-18 in the XtremeDSP User Guide)
- Single DSP slice 35 x 35 multiplier (Figure 1-19 in the XtremeDSP User Guide)
- Fully pipelined complex 18 x 18 multiplier (Figure 1-22 in the XtremeDSP User Guide)
- High-speed FIR filter (Figure 1-17 in the XtremeDSP User Guide)

The application note also describes many of the important features of DSP48 blocks supported by XST 8.1i and Synplify Pro 8.2. Map reports, Timing Analyzer reports, and a detailed view of the FPGA Editor show the efficiency of the synthesis and implementation tools. You can also see how the cascade chain between adjacent DSP48 slices is used to improve both performance and power consumption. Almost all of these widely used configurations provide the best implementation results – in terms of resources used as well as performance. However, some remaining limitations are also described. We expect these few points to be resolved in future releases.

For more information, see the XtremeDSP Design Considerations User Guide at *www.xilinx.com/bvdocs/userguides/ug073.pdf*. The methodology is clearly explained and implementation results analyzed in detail, with ISE™ software tools like Timing Analyzer and FPGA Editor.

Multi Video Designs (MVD) is a training and design center specializing in FPGA designs, PowerPC™ processors, RTOS for embedded/real-time applications, and high-speed buses like PCI Express and RapidIO. MVD as an Approved Training Partner and a member of the Xilinx XPERTS program, with offices in France, Spain, and South America.

XST Support for DSP48 Inference

XST, the synthesis engine included with the Xilinx® ISE™ toolset, contains extensive support for inference of DSP48 macros. A number of macro functions are recognized and mapped to these dedicated resources, including adders, subtracters, multipliers, and accumulators, as well as combinations like multiply-add and multiply-accumulate (MAC). Register stages can be absorbed into the DSP48 blocks, and direct connect resources are used to cascade large or multiple functions.

Macro implementation on DSP48 blocks is controlled by the USE_DSP48 constraint with a default value of auto. In auto mode, XST attempts to implement all aforementioned macros except adders or subtracters on DSP48 resources. To push adders or subtracters into a DSP48, set the USE_DSP48 constraint value to yes.

XST performs automatic resource control in auto mode for all macros except adders and subtracters. In this mode you can control the number of available DSP48 resources for synthesis using the DSP_UTILIZATION_RATIO constraint, specifying either a percentage or absolute number. By default, XST tries to utilize, as much as possible, all available DSP48 resources within a given device.

With the 8.1i release of ISE software, XST has introduced further enhancements to its DSP support. XST can now infer loadable accumulators and MACs, which are critical for filter applications. XST can recognize chains of complex filters or multipliers – even across hierarchical boundaries – and will use dedicated fast connections to build these DSP48 chains. The Register Balancing optimization feature will consider the registers with DSP48 blocks when optimizing clock frequencies. Consult the XST User Guide (*http://toolbox.xilinx.com/docsan/xilinx7/books/docs/xst/xst.pdf*) for details about coding styles, and watch the synthesis reports for specific implementation results for your Virtex™-4 designs.

– David Dye
Senior Technical Marketing Engineer
Xilinx, Inc.

Un-Tethered Debugging

Remote debugging enables designers to configure, debug, and verify Xilinx FPGA systems remotely.

by Brent Przybus
ChipScope Pro Product Marketing Manager
Xilinx, Inc.
brent.przybus@xilinx.com

For years I have looked at other career choices with envy. Pilots have an office in the sky, traveling routes like New York to London and Los Angeles to Tokyo. Archeologists go on digs to such exotic locales as Egypt, Africa, and South America. Even cable television installers get to visit exotic homes in the Bay Area hills.

Engineers, on the other hand, toil away in cold, sterile labs within the bowels of a corporate complex, trying to solve problems and bring their next engineering marvel to life. As FPGA designers, most of that time is spent running simulations, verifying, and debugging designs implemented with the latest Xilinx® FPGA technology.

I am happy to say that the days of pasty white skin and wearing sweaters in summer are over. Remote debugging is finally here.

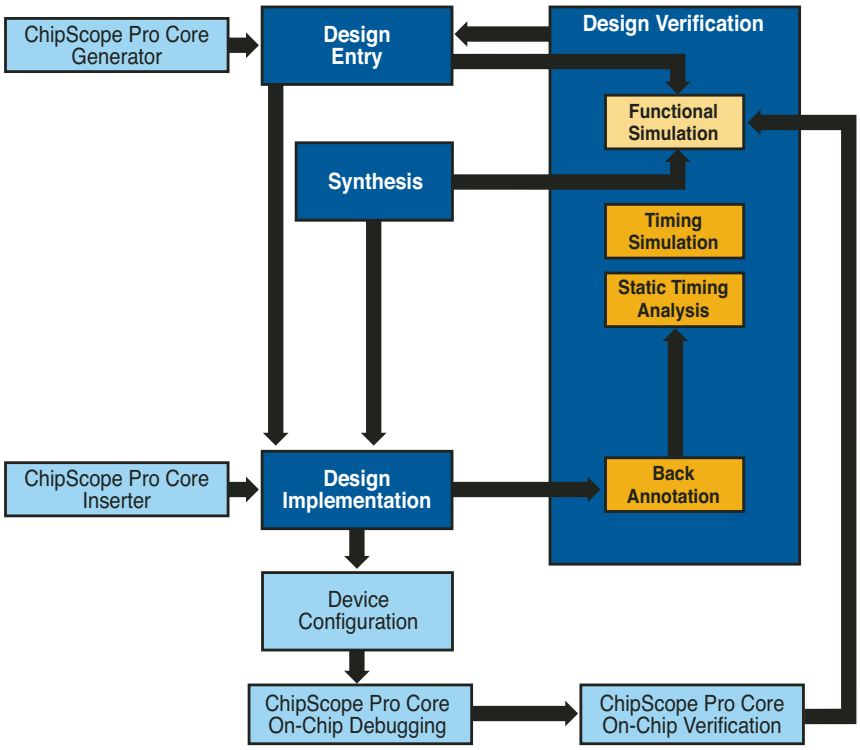


Figure 1 - ChipScope Pro tool integration into the Xilinx FPGA design flow

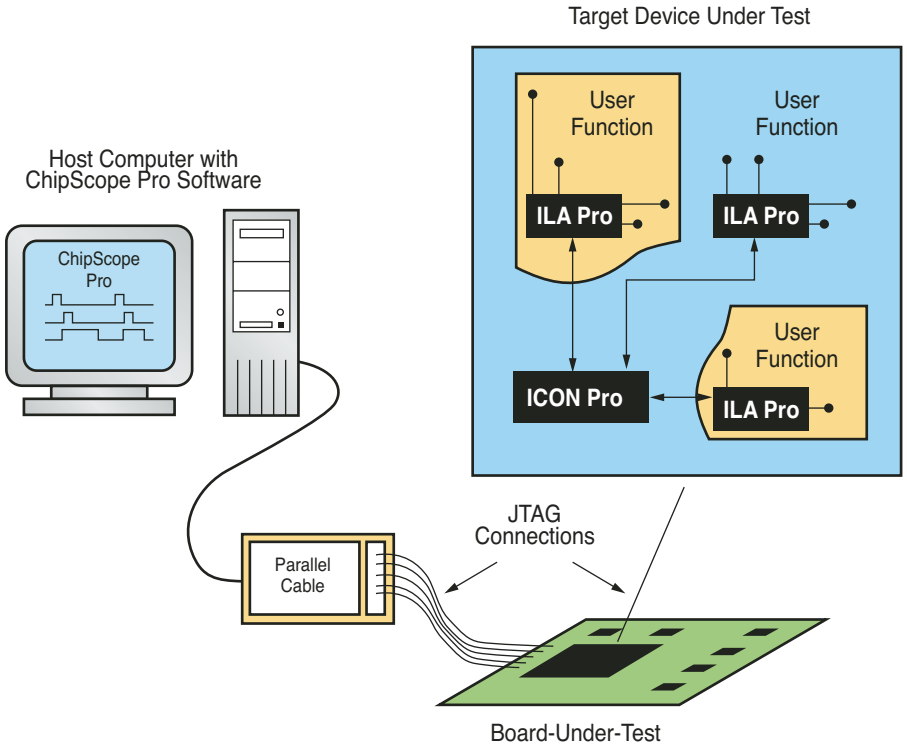


Figure 2 – Typical ChipScope Pro lab setup

Preparing for Remote Debugging

If you have used ChipScope™ Pro tools, then you understand the value of placing ILA, VIO, IBA, and ATC2 cores within your FPGA designs to gain access and insight into what is happening. If you have not used the ChipScope Pro analyzer before, do not despair. The ChipScope Pro debugging and verification design tool allows you to debug and verify your Xilinx FPGA designs on-chip by following these steps:

- Instrument your design by using either the ChipScope Pro core inserter or core generator. These tools allow you to add ChipScope Pro debugging cores to your new or existing designs. The debugging cores come in several different flavors: integrated logic analysis (ILA), integrated bus analysis (IBA), virtual input output (VIO), and the Agilent Trace Core 2 (ATC2).

- Configure, debug, and verify your design using the ChipScope Pro analyzer. The ChipScope Pro analyzer runs on a standard PC running either Windows or Linux and interfaces to the FPGA through the JTAG port. Through the analyzer, you can interact with the debugging cores you have placed in your design, setting trigger conditions, capturing data, and displaying that data in a convenient waveform or list view interface similar to a bench-top logic or bus analyzer.

Figure 1 shows how ChipScope Pro tools are integrated into the standard Xilinx FPGA design flow.

The latest version of the ChipScope Pro analyzer includes remote debugging capability. To use this new feature, you need to specify a ChipScope Pro server – a lab machine with network access – connected to the board and the FPGA through the JTAG port using a Parallel IV or USB configuration cable. In most cases, this is the standard setup that has kept designers in the lab all these years (Figure 2).

By running the Xilinx ISE tools on the client machine, you can create and modify your design and add ChipScope Pro cores from the comfort of your remote location.

Setting up and starting the ChipScope Pro analyzer server is easy using version 7.1i or later of the ChipScope Pro tools:

- Start the server on a Windows machine by executing:

  $CHIPSCOPE/cs_server.bat

  or start the server on a Linux machine by executing:

  $CHIPSCOPE/bin/lin/cs_server.sh

  ($CHIPSCOPE refers to the directory path where ChipScope Pro tools are installed on the server machine.)

- Use the following command line option, as required:
  - -port <portnumber>. Specify the TCP/IP port number to be used by the client and server to establish a connection. The default port number is 50001.
  - -password <password>. Protect the server from unauthorized access. No password is set by default.
  - -l <logfile>. Specify a location for the log file. The default location is $HOME/.chipscope/cs_analyzer_<portnumber>.log.

Moving Out of the Lab

With the ChipScope Pro server running back in the lab, the next step is to run the ChipScope Pro client software on your remote system with a network connection, such as a laptop with wireless access. Setting up the client is easy from within the ChipScope Pro analyzer software:

- Launch the ChipScope Pro analyzer from the start menu or from the ISE Project Navigator.
- Select the JTAG Chain > Server Host Settings menu option. This will launch a server settings dialog box.
- For remote operation, set the host setting to the IP address of the server or the appropriate network system name for the server. Set the port and password setting to the same values used on the server. If default settings were used on the server, these settings are already set correctly. Click OK.
- Open a connection to the JTAG download cable you are using and start device configuration and debugging. The remote connection to the server will not be established until you open a connection to a JTAG download cable.

Note that you must use the same version of ChipScope Pro tools on both the client and server machines.

By running the Xilinx ISE™ tools on the client machine, you can create and modify your design and add ChipScope Pro cores from the comfort of your remote location. When you have finished implementing your design and creating the configuration file, simply launch the ChipScope Pro analyzer, connect to the server in the lab, reconfigure the FPGA, and debug just as if you were in the lab.

Practical Applications

Beyond debugging from anywhere, there are some very real, practical applications for remote debug.

- Sharing resources. Xilinx FPGA designs utilize embedded processors: the PowerPC™ 405 hard-core or MicroBlaze™ soft-core processor. The designs may consume as many as 100,000 logic cells, more than 10 Mb of blockRAM, and leverage as many as 512 XtremeDSP™ slices. Often an entire design team works on a single FPGA design, and at some point each member of the design team requires access to the FPGA. Remote debugging allows the design team to share a single board in one lab location, with design team members accessing the lab from offices throughout the world.
- Support fielded systems. Your product has shipped, but there is a problem. The field application resource is on-site with a laptop but needs factory assistance. ChipScope Pro remote debugging allows the factory to access the fielding system through the field application engineer's laptop connected to the board through JTAG. The factory can quickly uncover the design problem using ChipScope Pro cores left in the original design; determine, implement, and test a fix; and upgrade the system remotely from the factory.
- Move to hardware faster. The availability of hardware sooner in the design process can greatly accelerate the overall FPGA design flow. Rather than spending hours simulating a complex sequence of events, run this portion of the design in hardware. By using an evaluation development board or a prototype system with a target FPGA, you can run simulations in hardware – with real system data, at the system clock rate – and complete analysis in a fraction of time software simulation takes. The remote debugging and verification solution enabled by ChipScope Pro tools allows you to do this from your design machine, in the comfort of your office.

Conclusion

Remote debugging can increase productivity and the quality of the work environment. For more information about how to use ChipScope Pro tools, visit *www.xilinx.com/chipscopepro*, where you will find ChipScope Pro documentation, links to demo-on-demand sessions, and answers to frequently asked questions.

Shorter Verification Cycles at Lucent Technologies

ChipScope Pro real-time debugging software has reduced project verification times for Lucent's high-speed Optical Networking Products Division.

by Arun Thakkar
Member of the Technical Staff-1
Lucent Technologies
thakkara@lucent.com

The Optical Networking Products Division at Lucent Technologies is involved in producing next-generation SONET optical networking equipment for Internet service providers. Our current project adheres to the OC48 standard, with serial I/O as high as 2.5 GHz. Internal logic clock speeds can run anywhere from 155 MHz on down, with heavy system interface and bus traffic on our system boards.

Our past projects have relied heavily on HDL simulation, using VHDL models for component modeling and bus functional models to simulate the interface and backplane traffic. To accurately model the entire system, we also added external memory interface device models and modeled the effects of interface trace delays. Together, this system simulation would let us check new design concepts in our existing and future projects. The downside was that we had to create HDL test benches and maintain them in parallel. System simulation times could be quite long and tedious, and there remained the unanswered question, "Have we modeled enough to accurately predict system behavior"?

ChipScope Pro Analyzer – Real-Time Debugging

We began using Xilinx® ChipScope™ Pro in our systems with the 6.1i software release. In our current project, we are using two to three ChipScope Pro ILA cores per clock domain inserted into our target Virtex™-II Pro XC2VP30 device. The soft debug cores are inserted after the synthesis stage using the ChipScope Pro netlist inserter, and we debug primarily by using the ChipScope Pro logic analyzer. We are capturing roughly 100 transitions in any one debugging cycle, depending on the particular problem we are researching, and we use trigger ports to save onboard block RAM memory.

ChipScope Pro tools let us capture data at any point in the FPGA, while the chip is interacting with the rest of the system and running at operating speed. We have been able to reduce the number of pins on the FPGA and on the board that were previously dedicated to verification, since we debug directly through the JTAG programming cable.

Uncovering Problems

One example, in which by using the ChipScope Pro analyzer we debugged a problem that we wouldn't have otherwise uncovered, was in a new revision of one of our in-production products. At the vendor's recommendation, our manufacturer had recently upgraded one of the system's external memories. Suddenly we were seeing degraded performance and memory parity errors. Nothing else had changed, yet the system stopped working. Simulation didn't reveal the error, but we started tracing through with the ChipScope Pro tools, in real time. By triggering on the parity byte, we were able to discover a handshake problem when writing data to and from the controller. We were issuing an auto-refresh before bank pre-charge before a write cycle was complete. This was fine in the older part, but the tolerances had changed slightly in the new part.

Conclusion

We still use HDL simulation in our projects here at Lucent, but ChipScope Pro tools have now become a vital part of our design and verification cycle, for all of our projects. By catching problems in real time in the lab and by taking advantage of the reprogrammability of FPGAs, we are able to turnaround design problems in a matter of hours and get more out of our project time.

Verifying Your Logic Design for First-Time Success

Xilinx and its Alliance Members have the latest tools and methodologies to support your verification requirements.

by Hamid Agah
Senior Technical Marketing Manager,
Design Software Division
Xilinx, Inc.
hamid.agah@xilinx.com

Howard Walker
Technical Marketing Engineer,
Design Software Division
Xilinx, Inc.
howard.walker@xilinx.com

Scott Campbell
Technical Marketing Engineer,
Design Software Division
Xilinx, Inc.
scott.campbell@xilinx.com

When Xilinx invented the FPGA in the mid 1980s, the preferred way to verify a design was to program the FPGA in the actual system and see if it operated properly from both a timing and functional standpoint.

Those days are long gone.

According to a 2005 EDA study by CMP Media, verification is one of the top three considerations for FPGA designers. A fast and successful verification experience is essential to get your product to market on time. But how do you know if your current flow is the best choice, especially for today's high-density FPGAs? At a minimum, a sound verification strategy should include static/dynamic timing and dynamic simulation. Optional advanced methodologies such as equivalency checking and assertion-based verification are now also available to Xilinx FPGA users (Figure 1).

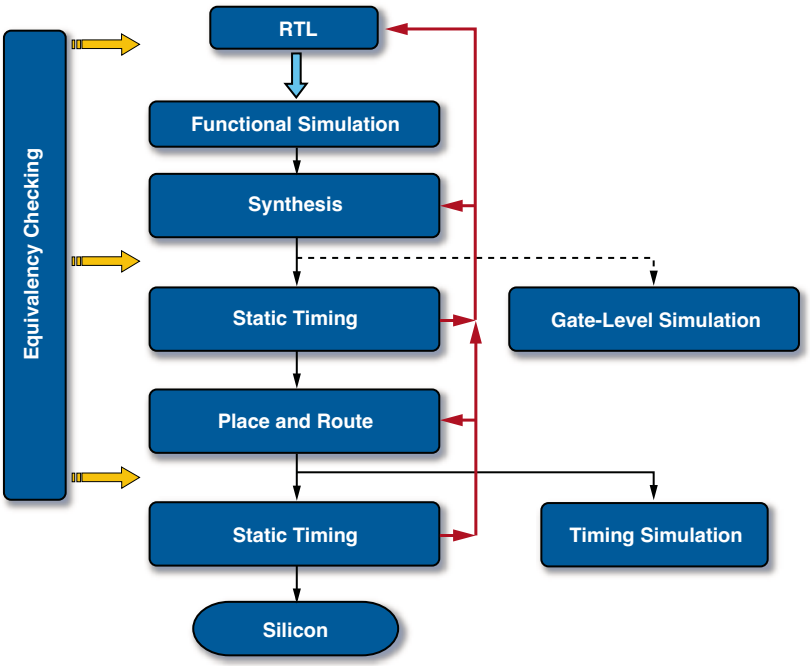


Figure 1 -Verification solution available to Xilinx users

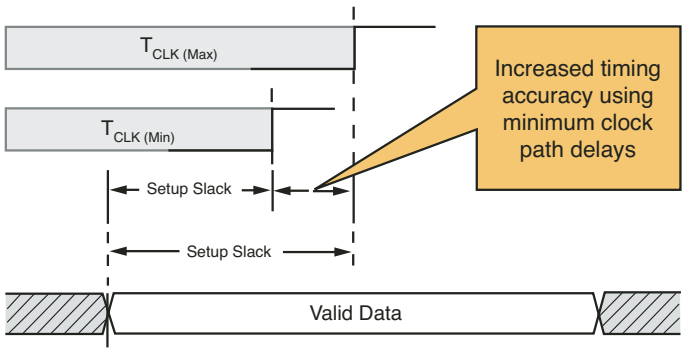


Figure 2 – Improved accuracy of static timing analysis (STA)

In this article, we'll discuss improvements and additions to the verification solutions available from Xilinx and its Alliance Program Members.

Improved Timing Analysis in ISE 8.1i Software

A complete timing verification must include checking the FPGA design under both the best- and worst-case operating conditions. The worst-case conditions occur when the voltage supply to the FPGA is at a minimum and the temperature of the FPGA is at a maximum. These conditions increase the internal delays of the device, and thus increase the potential for setup time violations. The best-case conditions occur when the voltage supply to the FPGA is at a maximum and the temperature of the FPGA is at a minimum. These conditions decrease the internal delays of the device and increase the potential for hold-time violations.

In addition to voltage and temperature variations, the clock system is subject to uncertainty because of various sources of jitter throughout the system. Jitter can cause the early or late arrival of a clock edge and thus increase the chance of a setup or hold-time error.

Traditionally, FPGA-based static timing analysis software has only been able to analyze device operation under worst-case temperature and voltage conditions without regard to clock uncertainty. Although this methodology was sufficient for slower system-level interface standards, the increasing speed of today's dual-data-rate (DDR) source-synchronous interface standards demands a more complete verification solution.

STA Analysis for Real-World Conditions

To provide the most accurate timing verification, Xilinx® static timing analysis software automatically analyzes the design under the best- and worst-case operating conditions. This analysis is performed simultaneously for the both the system I/O interface and the internal logic of the design. The methodology uses a combination of minimum and maximum delays for setup and hold-time analysis. In setup time analysis, the worst-case condition occurs when the data path delay is at a maximum and the clock path delay is at a minimum. Conversely, the worst-case hold-time analysis condition occurs when the data path delay is at a minimum and the clock path delay is at a maximum.

To account for clock uncertainty in the design, the Xilinx software system allows the specification of the input jitter for each clock. In addition to the incoming clock jitter, the clock uncertainty because of system jitter and the clocking system design automatically takes into account all timing analysis checks. Clock uncertainty increases the potential for a setup time violation by effectively decreasing the clock path delay. In the same manner, clock uncertainty increases the chance of a hold-time violation by increasing the clock path delay.

By providing the ability to simultaneously model both the minimum and maximum delays of the clock and data paths, Xilinx timing analysis software ensures the greatest reliability of your system design across all operating conditions (Figure 2).

ISE 8.1i Software Breaks New Ground

Although static timing analysis verifies that the physical delays of data and clock paths in the FPGA design will not cause setup or hold-time violations, you must also verify the design's functional operation. Because of the dynamic nature of the design, the functional timing operation must be tested

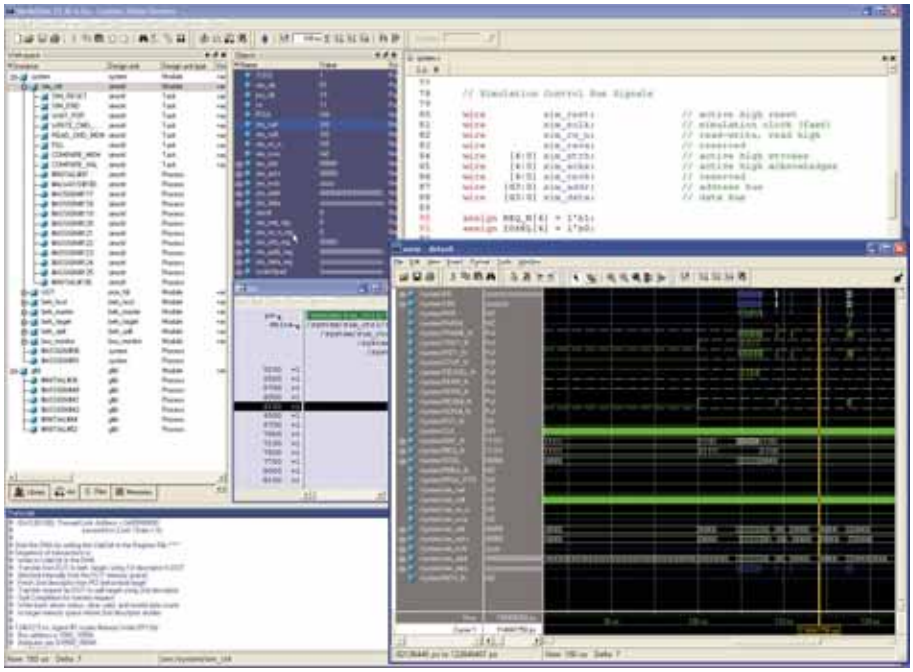


Figure 3 – Easy-to-use Xilinx ModelSim Edition (MXE III)

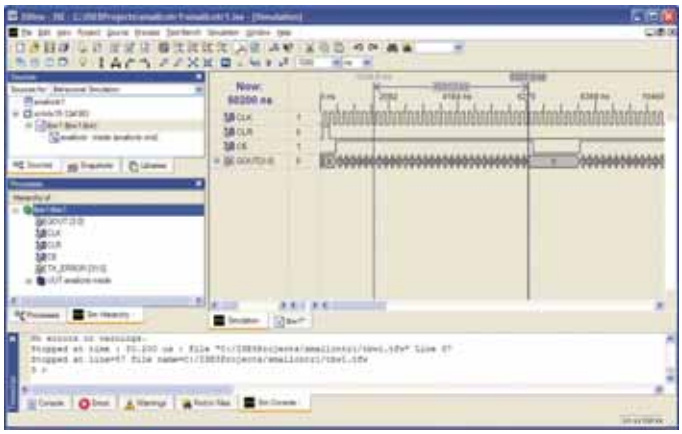


Figure 4 – Xilinx ISE simulator provides streamlined integration.

at system-level speeds. In a manner similar to static timing analysis, for an accurate timing simulation you must take into account the best- and worst-case conditions due to process, voltage, temperature, and clock uncertainty. Xilinx ISE™ 8.1i software breaks new ground in dynamic timing simulation accuracy by allowing both minimum and maximum clock and path delays to be simulated simultaneously. This unique ability ensures that setup and hold-time violations will be accurately accounted for, and works automatically with all simulators.

Easy-to-Use Simulators from Xilinx

ModelSim Xilinx Edition III

Working with the Model Technology division of Mentor Graphics, Xilinx has developed a customized, lower cost version of the popular ModelSim PE simulator called ModelSim Xilinx Edition III (MXE III) (Figure 3). MXE III is ideal for medium-density FPGAs with capacities as high as 2 million system gates, such as the Spartan™-3E FPGA family. It enables you to verify the functional and timing models of your design and your HDL source code (for more information, see *www.xilinx.com/ise*). A lower performance version of MXE III called ModelSim Starter is a no-charge feature of the ISE Foundation™ toolset.

MXE III's features and capabilities include:

- Seamless integration with ISE software, delivering better dynamic verification through automated graphical test bench generation and easy viewing in the Project Navigator processes window
- More capacity and faster performance than MXE-II
- Support for system Verilog and Verilog PLI/VPI
- Excellent debug environment
- Waveform management tools
- Customizable user interface
- Batch-mode simulation
- HDL editor
- Source code debugging
- Verilog-2001 or VHDL-93 support (single language product)
- The MXE-III Starter version offers 50% faster HDL simulation and 20 times more design capacity than MXE-II
- Upgrade path to more powerful simulators such as ModelSim PE and SE

ISE Software SIM 8.1i

Xilinx provides an integrated full-featured HDL simulator as an optional design product for ISE Foundation users (Figure 4). Also included at no charge in ISE Foundation is ISE Simulator Lite. This starter version of ISE Simulator is ideal for smaller devices.

ISE SIM 8.1i features include:

- Mixed-language Verilog 2001 and VHDL-93 design support
- Simple user interface
- Export to XPower for easier device power estimation
- Integrated wave editor for test bench creation
- Design hierarchy, waveform, and console views
- Source-level debugging
- Command-line console with TCL interface
- Does not require FlexLM licensing
- "Generate Expected Results" process generates expected design output behavior based on input stimulus

Easier Verification of Hierarchical Designs

A capability in the ISE Foundation toolset called KEEP_HIERARCHY makes it much easier and faster to debug hierarchical designs. This design flow not only decreases the time it takes to run timing simulation, but also addresses the bigger problem of finding and resolving problems during the debugging stage.

The idea is to maintain the hierarchy of selected sub-modules when the design goes through the synthesis and implementation flow and then verify these sub-modules in timing simulation before the entire design is assembled and verified. Each sub-module can be written out as a separate netlist and verified both in RTL simulation as well as in timing simulation with a separate associated SDF file. Because each sub-module for a timing netlist looks the same as the RTL version (with the same top-level port names), the same test benches can be used in timing simulation that were used in RTL simulation with little or no additional work.

Analysis of this methodology on typical Virtex™-4 designs has shown that both the simulation run times as well as the memory requirements required for simulation were considerably reduced. See the "Design Hierarchy and Simulation" section in the ISE 8.1i Synthesis and Verification Design Guide for more information.

Faster Simulator Performance for Xilinx Devices

Xilinx and its partners understand that shorter simulation runtime is a never-ending goal. Cadence NC-Sim v5.5, with its hard-coded Xilinx library primitives, and Mentor ModelSim SE 6.1, with 'vopt', have improved simulation runtime.

Advanced Verification Methodologies

Assertion-Based Verification

Assertion-based verification (ABV) is a blend of assertions, functional coverage, and formal model checking technologies applicable to both ASIC and Xilinx FPGA designs. Assertions are explicit expressions of design intent, capturing what a circuit structure should or should not do. By embedding assertions in a design and having them monitor design activities, assertions improve the observability of the design. For more information on ABV for Xilinx FPGAs, see "Early Defect Discovery with Assertion-Based Verification Accelerates Design Closure," also in this issue of the *Xcell Journal*.

Equivalency Checking

EC is a static verification technology that uses formal techniques to determine if two versions of the same design, at different stages of development, are functionally equivalent. This offers 10 to 100 times faster verification time and 100% functional coverage, with no test vectors required. It is also able to check for any tool-induced bugs. EC can eliminate long simulation runtimes when doing functional checks, as code is modified to meet your timing goals.

To use the EC flow for Xilinx devices:

1. Download the EC libraries at *www.xilinx.com/ise/partner_libraries*.
2. Set up the synthesis and implementation tools to generate "formally verifiable" netlists by turning off optimizations such as constant registers removal, register duplication, register merging, and disable re-timing.
   a. In Synplify Pro, use the following variables to enable EC and write out the automated setup file, <deign>.vif (text):
      set_option -verification_mode 1
      set_option -write_vif 1
   b. In DC FPGA, include set_fpga_default -formality to enable EC and write out the automated setup file, .svf (encrypted).
   c. For ISE enable 'netgen -ecb' to ensure that the <design>.svf file includes a listing of ISE optimized constant registers.
3. Output an EC-friendly Verilog netlist from CoreGen. The netlist is used as a functional model for each instantiated CoreGen IP to successfully verify CoreGen blocks for the post synthesis to post-PAR check.

Here are some tips for having a successful EC experience:

- Use CoreGen to instantiate large block RAMs in the RTL (Figure 5)
- Turn off re-timing in the synthesis tool
- Isolate wide multipliers to a separate hierarchy and then black box it
- Replace old instantiated components when targeting the RTL to a new architecture

Synplicity (Synplify), Cadence Design Systems (Conformal), and Synopsys (DC-FPGA and Formality) have been working with Xilinx to enable this for you. Contact these companies to learn more about:

- Synplicity Synplify Pro V8.0+ synthesis with Conformal V5.0+
- Synplicity Synplify Pro V8.0+ synthesis with eCheck V4.3+
- Synopsys DC FPGA V2005.03+ with Formality V2005.03+

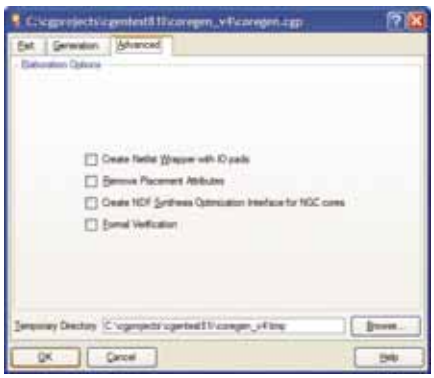


Figure 5 – Xilinx CoreGen netlist for equivalency checking

Conclusion

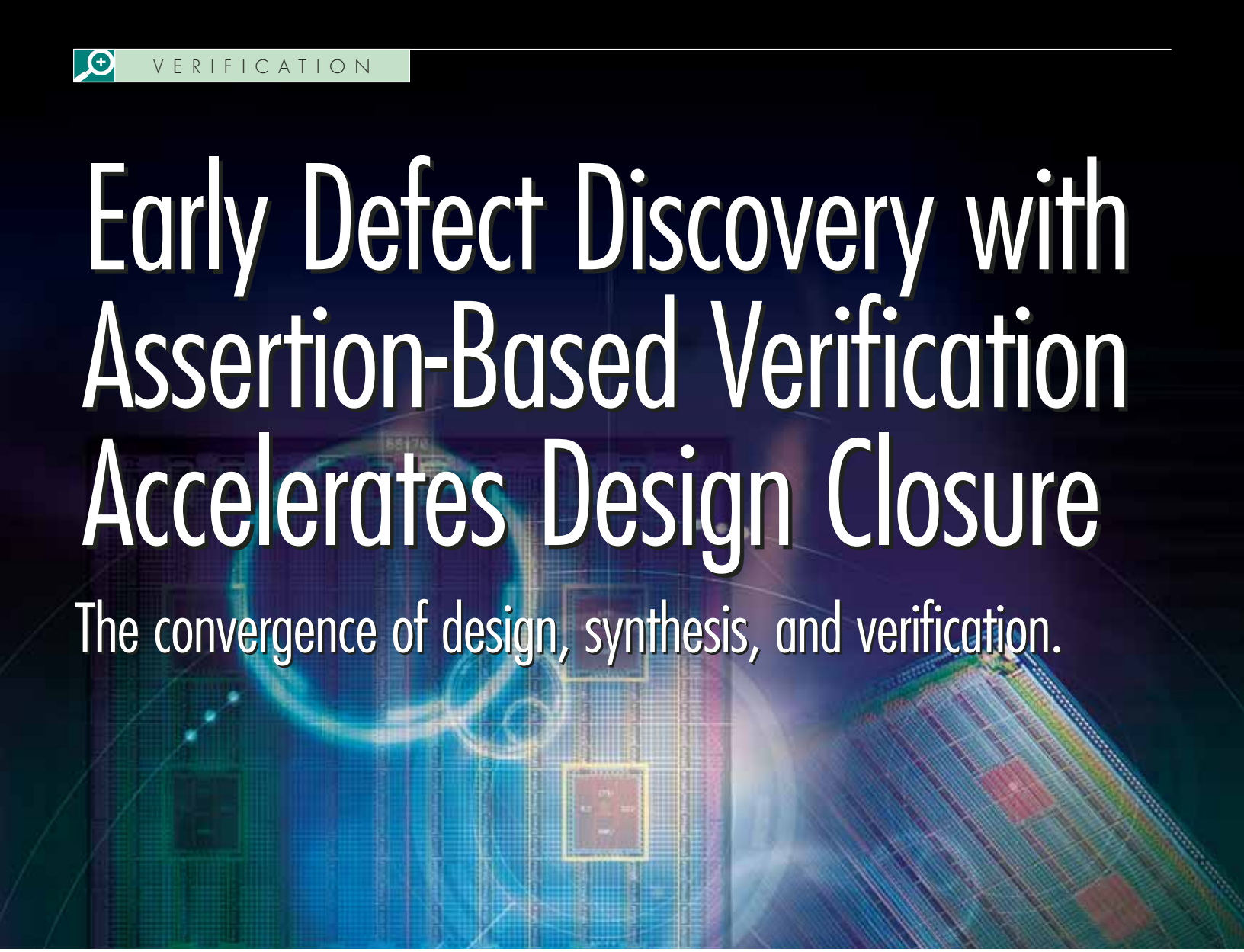
Verification is now one of the top three considerations for designers. Xilinx and our Alliance Program Partners are continually investing to improve your verification experience through:

- Improved timing accuracy
- Better integrated and easier-to-use tools
- Hierarchical design support
- Faster simulator performance
- New methodologies

For more information, visit *support.xilinx.com*.

Early Defect Discovery with Assertion-Based Verification Accelerates Design Closure

The convergence of design, synthesis, and verification.



by Ping Yeung
Principal Engineer
Mentor Graphics Corporation
ping_yeung@mentor.com

Darren Zacher
Technical Marketing Engineer
Mentor Graphics Corporation
darren_zacher@mentor.com

When designing a system-on-chip (SoC) using a platform FPGA for mobile, wireless, networking, or media applications, designers typically integrate many IP components within a short period of time. By offering advanced Xilinx® MicroBlaze™ and on-chip PowerPC™ processors, platform FPGAs such as the Virtex™-II Pro and Virtex-4 families give you a head start on integrating processor IP.

Regardless of your IP choice, market requirements are driving the need for more features, such that processor-based platform designs are becoming more complex – incorporating multiple processor cores and multi-layered bus architectures. This spiraling increase in complexity drives the need for new approaches to platform FPGA design and verification. The simple, push-button flow of synthesis logic mapping, with verification as an afterthought, just doesn't cut it.

Just as design flows for platform FPGAs have come to more closely resemble those adopted by ASIC SoC designers – where synthesis is closely linked to verification at every step – so too have FPGA developers begun to see the benefits of leveraging the convergent synthesis and verification strategies developed for complex ASIC SoCs.

Assertion-based verification (ABV) has emerged as a major player within this convergent design and verification flow. Assertions have been used successfully for more than a decade in microprocessor design. But only recently has their full potential been realized. The payoff is big for large, complex designs, including platform-based designs that contain a growing amount of third-party and in-house IP. As FPGAs approach the complexity and size of larger ASICs, assertions have become particularly useful to FPGA designers.

With the goal of examining the value of ABV within a convergent flow that encourages early discovery of design deficiencies (hereafter referred to as defects), we'll recommended a few strategies for successful design closure and explore ABV at greater length.

Essential Strategies for Success

Defect discovery must be consciously pursued earlier in the design cycle, where the overall pain and cost of fixing errors is much less. Here are some recommended best practices for accelerating design closure:

- Do more timing and performance analysis up-front. Are you willing to wait until after your design is functionally verified, synthesized, and placed and routed to find out whether your chosen arbitration scheme can keep up with incoming traffic? Early discovery of throughput issues requires more in-depth analysis of performance and timing issues throughout the synthesis process. Before burning cycles trying to meet timing constraints in place and route, you must ensure that constraints are complete. Early discovery of timing issues also requires constraint coverage analysis during synthesis.
- Use interactive synthesis techniques for greater predictability. An indispensable weapon in your FPGA design toolkit is a capable, interactive synthesis and analysis environment that goes from RTL to physical implementation. Interactive synthesis techniques allow you to perform "what-if" explorations earlier in the design cycle. A robust synthesis environment also provides several design representations such as high-level operators and architecture-specific technology cells. Interactive synthesis capabilities give you an earlier understanding of the nature of the design and indicate whether it will meet specifications.
- Check HDL code early and often. Today's design-management tools assist in checking code against an established set of agreed-upon (by the design team or silicon vendor) coding style rules. Before burning simulation cycles, you can use these coding style rule checkers to catch actual defects and flag potential defects, thus bringing defect discovery up-front in the design cycle.
- Implement a more effective functional verification strategy. Synthesis tools for platform FPGAs do more than simply generate a technology-mapped netlist. Best-of-breed synthesis tools contain important analysis capabilities that provide more insight into your design at every stage in the cycle. These capabilities can identify potential problem areas, such as clock domain crossing points, where you need to handle functional verification delicately. Moreover, when applying a traditional functional verification approach (using VHDL or Verilog test benches) to platform FPGAs, modeling random or pseudo-random stimuli and checking circuit response against designer intent becomes increasingly tedious. This is where ABV can play a significant role.

Assertion-Based Verification

To go beyond traditional simulation-based verification, you must improve the observability of the verification process and control of the design. This is best done using ABV: a blend of assertions, functional coverage, and formal model-checking technologies. Assertions are explicit expressions of design intent, capturing what a circuit structure should or should not do. By embedding assertions in a design and having them monitor design activities, assertions improve observability. Formal model checking (FMC) analyzes the RTL structure, characterizing the internal nature of a design to augment control.

Because ABV creates an unlimited number of observation points (assertions) throughout the design, it speeds debugging by identifying errors close to or at their source. Assertions also identify problems that do not propagate to a primary output; thus ABV improves design quality by exposing bugs that might otherwise go undetected.

Assertions provide a set of properties as targets for FMC. They also enable simulation test vectors to be used by dynamic formal verification (which we will explain later) to find potential violations of a module's assertions.

The basic ABV methodology compares the implementation of a design against its specified assertions. Therefore, the quality and comprehensiveness of a design's assertions are critical. These assertions originate from the following sources:

- Using proprietary or standard assertion languages, such as SystemVerilog Assertions (SVA) and the Property Specification Language (PSL), design engineers can write assertions as you implement various structures. They may sometimes make an inadvertent mistake in their implementations; assertions provide critical cross-checks between intended and actual behaviors. For example, suppose the intended maximum value of a pointer is 48. An assertion might detect that a coding mistake exists that allows the pointer to go beyond 48 in otherwise legal situations.
- Verification engineers can write assertions from a specification of the design, a design document, or from their personal understanding of the design. Normally these assertions address concerns at a high level and cover critical behaviors of the design. For example, suppose a particular DMA controller has a channel configured to do 10 memory transfers. Before the channel is de-allocated, the total number of transfers must be exactly 10, or a serious problem will occur. The verification engineer can add an assertion to check for such an event.
- Both design and verification engineers insert protocol assertion monitors to check for violations of the corresponding interface protocols. These monitors ensure that transactions between components are properly generated and handled correctly. EDA vendors have developed off-the-shelf protocol assertion monitors for common standard interfaces (such as Mentor Graphics CheckerWare protocol monitors). These monitors come with all of their protocols' rules encapsulated, and are kept up-to-date to account for protocol standards revisions. Used with simulation, protocol assertion monitors collect useful statistical information. For formal verification, they act as constraints.

- Some verification tools insert assertions into the design by analyzing the RTL source-code structure. Solutions for many common design problems provide the basis for this analysis, allowing automation of the instrumentation process. Many problems are discovered statically, either by the tool itself (using netlist analysis) or with formal analysis. Other problems are detected only with dynamic verification methods, such as functional simulation or dynamic formal verification. Examples of problems analyzed by automatic assertion insertion include synthesis-to-simulation mismatch problems, unreachable code or FSM states, clock domain crossing (CDC) errors, X-semantic problems, and CDC clock jitter.

ABV Essentials

ABV accelerates debug and improves design quality by finding and fixing bugs early, at or near their source.

Simulation-Based Verification with Assertions

During functional simulation, a design's assertions monitor activities both inside the design and on its interfaces (Figure 1). Assertion violations are reported immediately; the problem need not propagate to the design's output ports to be detected by the test bench. This high observability simplifies bug triage and increases the understanding of potential causes of discovered problems.

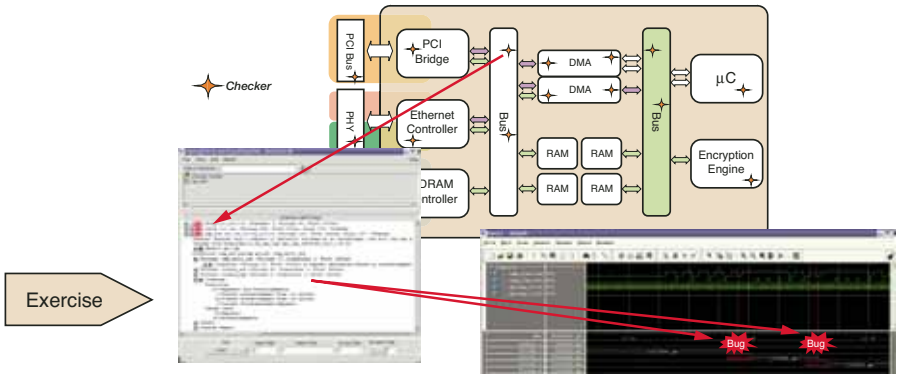


Figure 1 – Simulation-based verification with assertions

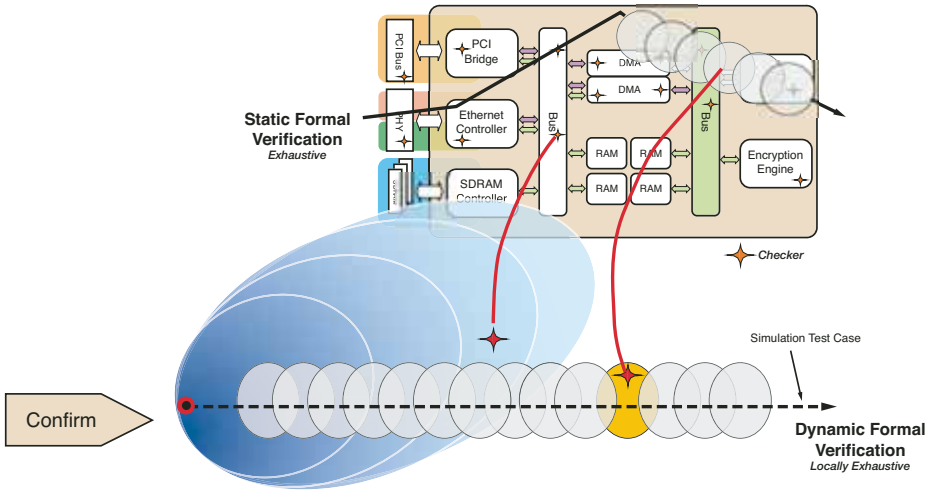


Figure 2 - Formal model checking with assertions

Formal Model Checking with Assertions

FMC is a complementary technology to simulation and an integral part of an ABV methodology. FMC uses mathematical techniques to prove some assertions true (given a set of assumptions provided by the assertion library) and other assertions false (by discovering counterexamples). A proof means that FMC has explored all possible behavior with respect to the assertion and has determined it cannot be violated. A counterexample shows the circumstances under which the assertion is not satisfied. FMC examines all possible states of a design to determine if any of them violate a specified set of properties.

Dynamic formal verification, a new FMC technique pioneered by the 0-In business unit of Mentor Graphics, overcomes the memory and computational limitations of static formal verification by amplifying a given simulation trace. This enhances ABV by enabling FMC at both the sub-block and chip levels, using static and dynamic formal techniques, respectively (Figure 2).

Given a set of target behaviors, expressed as assertions or coverage points, dynamic formal verification uses formal analysis of "interesting states" along the simulation trace to determine if it is possible to reach any of the coverage points or violate any assertion. For functional verification, an "interesting state" is one where a new or rare design behavior has occurred.

Consider a bug that occurs many cycles deep into a simulation. This particular bug might occur only if a particular FIFO in the design is full and a particular FSM is in state FOO. With simulation, it is possible that a test will manage to fill up the FIFO, while the same or another test can also get the FSM into state FOO. To get exactly the right combination of random stimuli for both events to occur may be a low-probability event.

With dynamic formal verification, from a simulation state where the FIFO is full, formal analysis examines all possible states leading up to and beyond that point in time,

identifying and reporting the particular sequence that leads to the state that manifests the bug. Thus, dynamic formal verification coupled with constrained random stimuli uncovers a bug that would have otherwise required additional test bench code to detect.

Without dynamic formal verification, you would have to know that a potential bug is lurking and modify the stimulus constraints in an attempt to guide the test to the right set of states. In actuality, dynamic formal verification can amplify any single scenario into 10,000 or more effective tests. With this in mind, the actual set of constraints for the test bench can be much simpler because the constraints do not need to accommodate the fine-grained tweaking discussed above, saving the substantial time it takes to conceive and code such elaborate constraint environments.

Interface Protocol Monitors

Interface protocol monitors allow devices in platform-based designs to be isolated and verified. In any good processor-based design, every component must communicate correctly with the available interfaces. Protocol monitors check for violations of the corresponding interfaces at the block or chip level.

A protocol monitor includes assertions to enforce the protocol rules. It ensures that transactions between components are properly generated and handled correctly. During simulation or formal verification, incorrect protocol transactions will be identified at the source.

For hardware-related issues, an ABV methodology using the CheckerWare assertion library can improve defect discovery in processor-based platform FPGA designs. Besides checking for violations during simulation, CheckerWare protocol monitors collect functional coverage and statistical information and act as constraints for formal functional verification. Monitors affirm the transactions performed, highlight "holes" in the regression suite, and audit the quality of pseudo-random stimulus generation environments.

Hard-to-Verify Structures

As processor-based designs become more complex, components become buried so deeply in designs that it is difficult to control and test them effectively. For example, in any multi-layered bus architecture, multiple masters can talk to multiple slaves concurrently. This capability increases overall system bandwidth, but it also increases verification complexity.

To tackle these hard-to-verify structures, you can capture design intent with assertions and checkers and embed them in the RTL code. The embedded assertions automatically check for incorrect behavior resulting from errors in the RTL code itself, catching design problems locally without manual intervention by the designer or verification engineer.

Embedded assertions also allow you to locate the source of problems faster. In a traditional verification flow, errors detected during simulation are time-consuming to trace to the originating problem because there are many possible causes for each failure symptom. For instance, if multiple DMA channels are set up to transfer data from several interfaces to memory, you may detect an error in the memory data at the end of the test, but it is difficult to track down the exact transaction that produced the data error. Using assertions, you can quickly identify the exact time of the data corruption and the hardware resources involved with the problem.

Conclusion

The higher capacity and complexity of modern platform FPGAs has opened up unprecedented opportunities for companies, but it has also created increasingly tough challenges for designers. Adopting – and adapting to – strategies that leverage a convergent design, synthesis, and verification flow for earlier defect discovery can prove beneficial for FPGA engineering teams in terms of reduced iterations and faster design closure.

ABV accelerates debugging by finding and fixing bugs early, at or near their source, and improves design quality while reducing development costs by discovering errors that would otherwise go undetected until after tape-out.

For more information on advanced verification methodologies from Mentor Graphics, visit *www.mentor.com/fv*.

IS YOUR CURRENT FPGA DESIGN SOLUTION HOLDING YOU BACK?

FPGA Design | Ever feel tied down because your tools didn't support the FPGAs you needed? Ever spend your weekend learning yet another design tool? Maybe it's time you switch to a truly vendor independent FPGA design flow. One that enables you to create the best designs in any FPGA. Mentor's full-featured solution combines design creation, verification, and synthesis into a vendor-neutral, front-to-back FPGA design environment. Only Mentor can offer a comprehensive flow that improves productivity, reduces cost and allows for complete flexibility, enabling you to always choose the right technology for your design. To learn more go to **mentor.com/techpapers** or call us at **800.547.3000.**

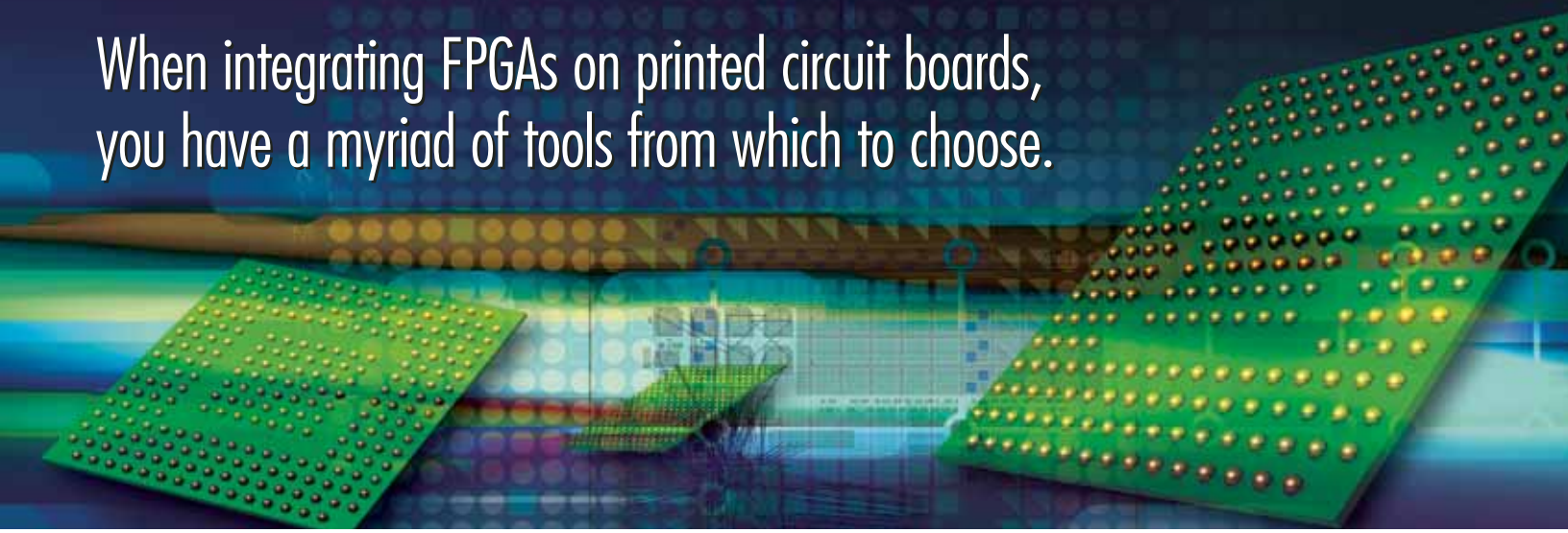
DESIGN FOR MANUFACTURING + INTEGRATED SYSTEM DESIGN
ELECTRONIC SYSTEM LEVEL DESIGN + FUNCTIONAL VERIFICATION

Mentor Graphics®

THE EDA TECHNOLOGY LEADER

Simplifying FPGA Pin Assignment Closure

When integrating FPGAs on printed circuit boards, you have a myriad of tools from which to choose.



by Philippe Garrault
Technical Marketing Engineer
Xilinx, Inc.
philippe.garrault@xilinx.com

Apart from the devices we proudly display in our cubicles here at the factory (because they represent the fruits of a lot of labor), FPGAs usually end up on printed circuit boards (PCBs).

Closing on a pin assignment that will meet requirements from both the PCB and FPGA environments is becoming more challenging. On one side of the interface, ever-increasing FPGA performance, density, and I/O count are placing tighter board constraints on the layout of the signal to and from the FPGA. On the other side, timing, congestion, and signal integrity of ever-faster signals on the PCB are placing constraints on FPGA pin assignment.

The latest EDA survey conducted by *EE Times* tends to reflect this challenge. Two-thirds of respondents said that their latest design uses two or more programmable devices. They also selected "getting the FPGA to work on the PCB" as the second most challenging part of FPGA design projects.

Until recently, very few tools or processes existed to assist with FPGA pin assignments, but this is changing. In this article, I'll look at the causes of pin assignment changes in today's design environments, describe the implications of such changes, and review the different tools available to simplify and automate FPGA pin assignment closure.

The Need for FPGA Pinout Changes

There are many good reasons to modify the FPGA pinout throughout the system design process. The flowchart in Figure 1 presents a typical design flow, with an emphasis on FPGA pinout closure and the sources for these changes. It clearly shows many areas where changes may occur; however, these are introduced by three specific sources:

1. Pinout changes because of design flow constraints. With today's highly competitive and constantly evolving electronic markets, it has become critical for companies to shorten design cycles so as to react faster to market demand changes. Therefore, product development is parallelized wherever possible. This goes for FPGA and PCB design processes too. The system architect defines an initial list of interface characteristics, which PCB and FPGA designers use to start their design. This assignment is later refined (that is, changed) as both the PCB and FPGA teams progress with product development. Additionally, market changes throughout the development cycle may require changes to the pinout, such as adding support for a new protocol or adding a feature at the last minute.

2. Pinout changes because of PCB constraints. Because of form-factor restrictions or board cost control, the available real estate on the board may be limited. In such cases, using the programmability and flexibility of the FPGA pinout can help solve PCB congestion or routing problems. Some of the FPGA features you can use include:

   - Swapping pins to untangle nets on the board. This diminishes the number of vias needed and may reduce the number of layers. Reducing the number of layer changes a signal encompasses will also improve its signal integrity and electromagnetic emissions.
   - Adjusting I/O properties to augment board signal integrity by lowering the signal drive strength or slew rate.
   - Using the programmable internal terminations to save on discrete component costs or to save board space in congested areas.

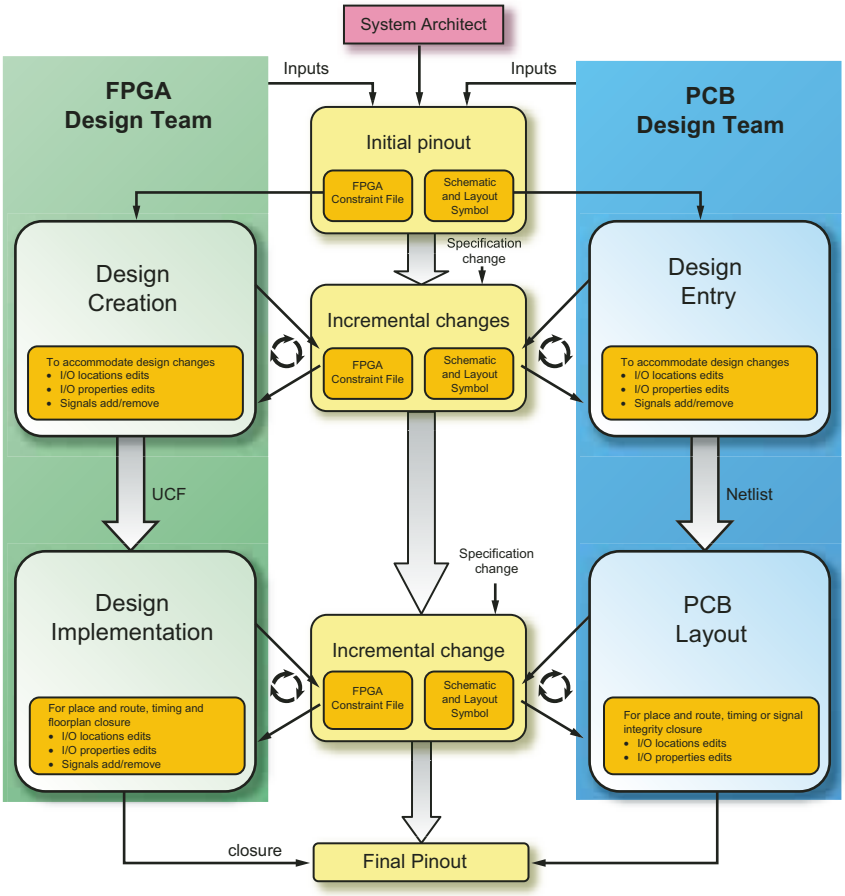


Figure 1 – Typical pin assignment closure process

3. Pinout changes because of FPGA constraints. In turn, FPGAs impose constraints on the design of the PCB, which may require pinout changes as the implementation progresses. These pinout changes are because of:

- Timing. Margins on some signals going into or out of the device may be tight enough that only a limited set of package pins will work.
- Dedicated/special-purpose pins. Because only a subset of package pins can be programmed to function as special-purpose pins (such as global or regional clocks or programming pins), this places constraints on the board to route signals to these capable I/Os.
- Voltage/termination compatibility. FPGA I/Os are grouped in banks, with all pins in a particular bank sharing power and reference voltages. This means that once a particular voltage is used in this bank, only I/Os with compatible voltage can be assigned in the same bank. This too may force you to select pins on the FPGA package that are not optimal from a PCB routing standpoint.
- Simultaneous switching outputs (SSO). To ensure that signals driven by the FPGA will meet I/O standard electrical characteristics, Xilinx publishes a recommended maximum number of SSOs per area of the device. You might have to scatter pins across different areas if you exceed these recommendations.
- Device decoupling circuits. As with any other IC on the board, FPGAs require certain characteristics for the power delivery system, which includes adding local decoupling networks (discrete components) to supply power to the circuit (the voltage regulator is typically unable to respond to rapid device demand changes).

Implications of Pin Assignment Change

Given the reasons I have discussed, it is almost certain that the FPGA pinout will change during the system design process. But what implications – in terms of time and effort – do these changes have on both the FPGA and PCB environments? What steps are necessary to verify that both environments are in sync and that constraints on either side are met?

From the FPGA side, after each pinout change you will need to update the user constraint file (.UCF) and verify that the internal timing constraints are met. You can also run PACE and SSO calculations to verify the assignment's I/O banking or SSO guidelines.

From the PCB side, you will need to update the schematic and layout symbols following a pinout change. You may also want to run the DRC to ensure that electrical and physical properties of the changed signal(s) pass the constraints. You can also run signal integrity analysis to verify that timing and amplitude margins are still adequate.

The important step is to communicate pin assignment changes to the other environment. Historically, PCB and FPGA design environments have been pretty isolated, with only limited communication channels between them. As I will explain in the next section, there are now tools and options available that make synchronization and data transfer far less cumbersome and time-consuming.

Available Tools and Processes

Now that you know that FPGA pin mapping will change, and you understand the frequency and implication of such changes in both FPGA and PCB designs, what are your options to minimize the burden of incorporating these changes? How can the pin mapping be propagated between environments in an automated fashion so that no data gets lost in translation? How can this be done quickly so that you can iterate pinout changes until a solution satisfactory to both the PCB and FPGA teams is found?

Within the FPGA Environment

The PACE (Pin and Area Constraint Editor) program allows you to graphically assign design signal names to package pins.

PACE also presents a die view with a graphical representation of the FPGA internal logic. Thus, you can assign pins close to the driving logic while also being mindful of the location recommendations from the PCB designer. You can enter pin properties for I/O standard, drive, and slew rate. The graphical view easily identifies special-purpose pins such as global or regional clocks. As illustrated in Figure 2, PACE also performs voltage compliance and SSO checks so that the pin assignment is correct by construction.

This tool proves valuable at different stages in the design process. You can create a pin assignment from scratch or read in the signal names from a synthesized netlist. After place and route, you can interactively adjust the pin assignment to satisfy the implementation or PCB tool constraints. PACE generates a Verilog module or VHDL entity, along with a pinout file that can be loaded in most PCB tools to generate and update schematic and layout symbols. PACE also reads pin files from the PCB tool and generates a UCF constraint file for ISE™ software.

Another tool worth mentioning here is DesignF/X from Product Acceleration. With this application, you define the I/O properties (standards, drive strength) and general area of the package on which your interfaces will be placed. The tool will then place every I/O so as to facilitate PCB routing, while making sure the FPGA banking, clocking, and SSO rules are obeyed.

Within the PCB Environment

Your favorite PCB design tool likely has a wizard or some documented process to assist with the creation and maintenance of your FPGA schematic and layout symbols. There is probably an assisted way to update these symbols after a pinout change. Finally, to assist with high-pin-count devices, there is also an option to fracture symbols onto multiple sheets to simplify documentation and the connectivity process.

FPGA/PCB Co-Design Tools

Tools designed to bridge the FPGA and PCB environments have recently debuted, with their feature lists continuously expanding. Shown in Figure 3, programs such as Mentor I/O Designer (although not the panacea, because they do not yet understand all of the FPGA and PCB constraints related to pin assignment) already go a long way toward simplifying and accelerating FPGA pinout closure. From a single environment, you can:

- Automate the synchronization of both environments after a pinout change. The program will update the Xilinx UCF file and schematic and layout symbols. This greatly reduces manual and error-prone data entry and the ensuing verification process to make sure that both the FPGA and PCB environments are in sync.
- Assign or swap pins from a graphical view of the FPGA package. Like PACE, it will perform DRCs to ensure that the pin assignment does not violate any device rules.
- Assign or swap pins from a graphical view of the PCB. Visualizing the physical location of the other components the FPGA is interfacing provides you with information you can use to assign pins in a manner that will reduce wire crossover on the board and simplify the FPGA breakout. This can in turn lead to saved design time (PCB router iterations) and money (better use of board real estate, reduced layer, or via count).

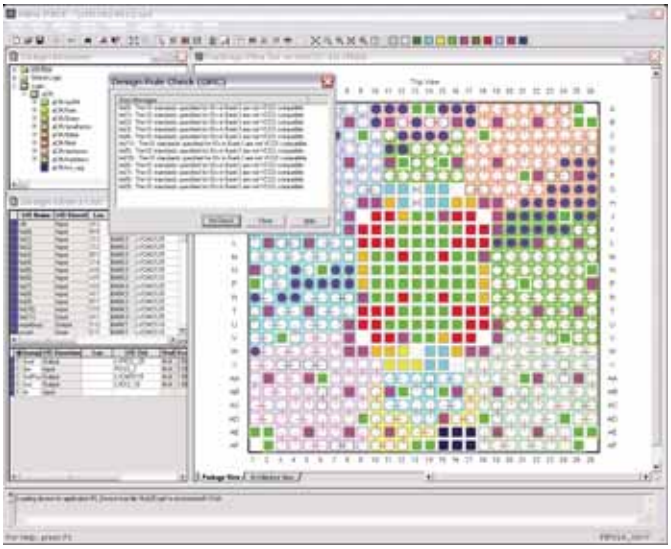


Figure 2 – PACE pinout export to PCB tools

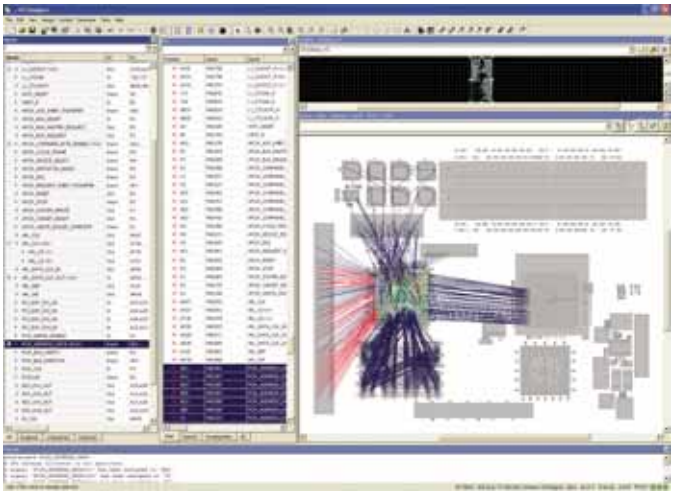


Figure 3 – PCB wire uncrossing within Mentor I/O Designer

Conclusion

Using traditional spreadsheet-based approaches, it is becoming more cumbersome to close on programmable device pinout, because both FPGA and PCB environments impose ever-tightening or sometimes conflicting constraints. Manual and time-consuming data entry and verification to ensure synchronization of both environments sometimes results in designers creating suboptimal PCBs, or not taking advantage of all of the power and flexibility of the FPGA. Thankfully, there is a growing set of tools and techniques that FPGA and EDA companies are putting in place to assist you with pinout data creation, management, and synchronization. Check out these tools, as they may help make better use of your time.

Power Considerations in 90 nm FPGA Designs

Learn how to design for reduced power with Virtex-4 FPGAs.

by Matt Klein
Sr. Staff Engineer, Applications Engineering,
Advanced Products Division
Xilinx, Inc.
matt.klein@xilinx.com

Managing your system power within budget is essential to maintain reliability in your system. Failure to do so may cause components to break down and reduce reliability.

The semiconductor industry's rapid move toward 90 nm silicon processes benefits the performance and cost aspects, but places enormous pressure on power budgets. As transistor sizes decrease, leakage current (and hence static power) increases exponentially. Dynamic power also increases, with increasing system speed and larger design density, but in a more linear fashion.

Today, many designs have 50/50 static and dynamic power dissipation. According to International Technology Roadmap for Semiconductors (ITRS) projections, static power is increasing exponentially at every process node; thus, innovative process technologies are imperative.

With the adoption of FPGAs in more markets and systems every year, driven by increasing performance/density and decreasing price, FPGA power consumption within the entire system is becoming critical. Leading FPGA vendors are already adopting new techniques to mitigate static and dynamic power consumption.

Power Consumption Considerations

The amount of power being consumed in the system is very important, and FPGAs (often used for system integration functions) make up the majority of power consumed in these systems. A given system or individual components usually have a power budget, which falls into two major areas. The first area is a simple practical one, which is to accommodate the power capacity of the supplies used in a design to meet system power needs. The second area is about thermal concerns, which you need to understand to keep the system working within its various components' temperature specifications. To this end, it is important to know where power consumption comes from in the FPGAs you choose and how you can optimize it.

Working Within a Power Budget

As mentioned, the power budget arises because of power consumption and thermal concerns. Here is a typical example: a board has a power budget of 20W with a normal operating environment of 10°C to 40°C. Under conditions of a failed fan(s), ambient air above certain components may rise above 70°C. Many components' manufacturers have operating conditions that range as high as 85°C in junction temperature (for commercial grade) and 100°C for industrial grade parts. Using Xilinx® ISE™ XPower or Web-based power estimation tools, you can see where power consumption will fall and if you will need to optimize your FPGA's power consumption. It is also important to learn what consumes power in the FPGA and what methods of design optimization may be available to help reduce consumed power.

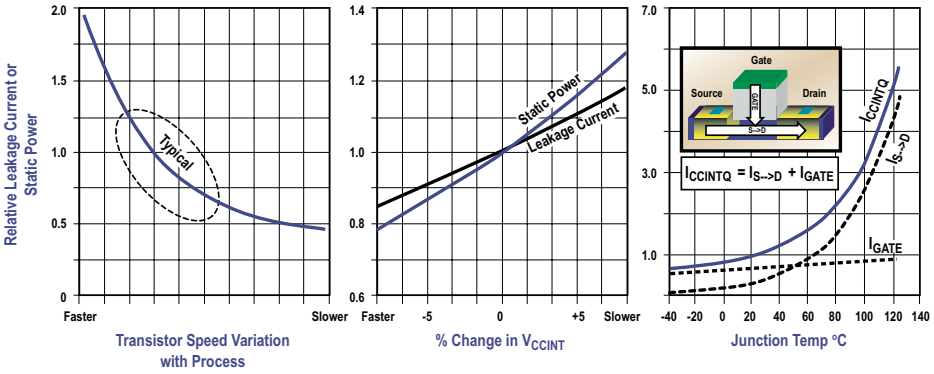


Figure 1 – Leakage and static power variations with process, voltage, and temperature

Where is Power Consumed in the FPGA?

There are two primary areas of power consumption in FPGAs. Static power comes from transistor leakage; dynamic power comes from voltage swing, toggle rate, and capacitance. Both are important factors in meeting a power budget and power optimization. It is therefore important to know what each factor is and how it varies with different operating conditions.

V_{CCINT} Variation from Nominal	Dynamic Power Change
-10.0%	-19.0%
-5.0%	-9.8%
0.0%	0.0%
5.0%	10.3%
10.0%	21.0%

Figure 2 – Variation in dynamic power due to core voltage variation

Static Power

Static power is power consumed by transistors due to leakage. This leakage is now significant for 90 nm devices. To get higher performance from the transistor, you need to lower the voltage threshold, (V_T) of the transistor, which also increases leakage.

Leakage of the 90 nm transistors varies strongly with process: the V_T of the transistors varies because of doping, and the gate length varies because of lithography. This can produce strong changes in transistor speed and leakage. Reduced V_T or gate length both increase leakage and speed, while the converse is also true. The variation in leakage and static power is about 2 to 1 between worst-case and typical process.

Leakage and static power are also influenced strongly by core voltage, V_{CCINT}, with variations that go approximately as the square and cube, respectively, of V_{CCINT}. Static power shows a ~15% increase, with only a 5% increase in V_{CCINT}. Leakage is very strongly influenced by junction (or die) temperature, T_J.

Because each of these factors – process, voltage, and temperature – have a strong effect on leakage and static power of the FPGA, it is important for you to understand them and how they might influence total power consumption of the FPGA or ASIC. Gate-to-substrate leakage is also part of total leakage, but is not highly temperature-dependent.

Figure 1 shows the variation in transistor leakage and static power in 90 nm FPGAs due to process, voltage, and temperature.

Seeing the increasing transistor leakage when moving toward a high-performance 90 nm FPGA, Xilinx IC designers chose to adopt the use of a third gate-oxide thickness in the transistors of the newest Xilinx Virtex™-4 FPGAs. In previous FPGAs and ASICs, only two oxide (dual-oxide) thicknesses exist: a thin oxide for core transistors and a thick oxide for I/O transistors. The use of a third middle thickness of oxide (triple-oxide) and higher V_T in a portion of the transistors of Virtex-4 FPGAs allows for a dramatic reduction in overall leakage and static power compared to other competitive FPGAs, which do not use triple oxide. So although variations with process, voltage, and temperature are still present, the absolute leakage in Virtex-4 FPGAs is about one-third that of competing high-performance 90 nm FPGAs.

Dynamic Power

Dynamic power is power consumed by transistors and traces that are toggling. The effect, simply put, is from changing an internal voltage from a logic "0" to a logic "1" (or vice versa) and charging a capacitance to that voltage. The more often this is done, the more power consumed. In the FPGA, transistors are used for logic and programmable interconnects between metal traces. The capacitance that we are talking about is transistor parasitic capacitance and metal interconnect capacitance.

The formula for dynamic power is:

$$P_{DYNAMIC} = nCV^2f$$

where n = number of toggling nodes, C = capacitance, V = voltage swing, and f = frequency.

All nodes in the FPGA consume power through a combination of charging transistor parasitic capacitance and metal interconnect capacitance. The latter depends on the length of routes in the FPGA, while net node capacitance is determined by the number of transistors that are switching. Tighter logic packing will reduce the number of switching transistors and minimize routing lengths, which will reduce dynamic power.

Figure 2 shows the variation of dynamic power with voltage swing and core voltage, V_{CCINT}. Process and temperature cause little variation in dynamic power. Taken together, their effect is less than 5-10%.

Lowering Design Power by Changing the FPGA Environment

To optimize the power consumption of a given design, there are certain things that you can do independent of the design contained in the FPGA. Knowing the environment is therefore important.

Temperature

Controlling temperature can help reduce static power. A reduction in junction temperature from 100 °C to 85 °C will reduce static power by ~ 20% (as shown earlier in

Figure 2). Even though the static power of the Virtex-4 FPGA is already low compared to other 90 nm FPGAs, reducing it by another 20% is valuable, since in some designs static power of the FPGA represents a sizeable portion (30-40%) of the total power budget. The reduction in junction temperature can be achieved by increased airflow and larger heat sinks, which will transfer heat away from the FPGA, reducing junction temperature. The reduction in junction temperature also has the added benefit of increasing reliability.

Voltage

Keeping core voltage at or below nominal will reduce static and dynamic power. Static and dynamic power consumed at the core voltage, V_{CCINT}, is often the largest power consumer in the FPGA. FPGAs are usually specified to be able to run and meet performance with power supply voltage within +/- 5% of nominal. Figure 2 shows that a +/- 5% variation in V_{CCINT} causes a ±15% and ±10% variation in static and dynamic power, respectively. To the extent that the V_{CCINT} power supply can be specified more tightly, you can also set it to be at or even slightly below nominal rather than being able to have a worst case that is 5% above nominal.

Xilinx has several design tools that can help you get an early or detailed estimate of design power consumption.

Lowering Design Power by Changing the Design

To make FPGA design-related tradeoffs in power consumption, it is important to know where you should start. Xilinx has several design tools that can help you get an early or detailed estimate of design power consumption.

Based on your estimates of design size (logic and flip-flops), operating frequency, toggle rates, embedded block utilization, and environment conditions such as temperature, the Xilinx Web Power tool shown in Figure 3 allows initial estimates to be made on the power consumption of a given design. It does not rely on detailed information about the design such as exact routing, placement, and utilization.

Figure 3 – Xilinx Web-based power estimation tools

The XPower tool included with all configurations of ISE software is a detailed power analysis program that allows you to input stimulus vectors for the design. Along with the information from the actual routed and placed design, the tool calculates power consumption much more accurately, as shown in Figure 4.

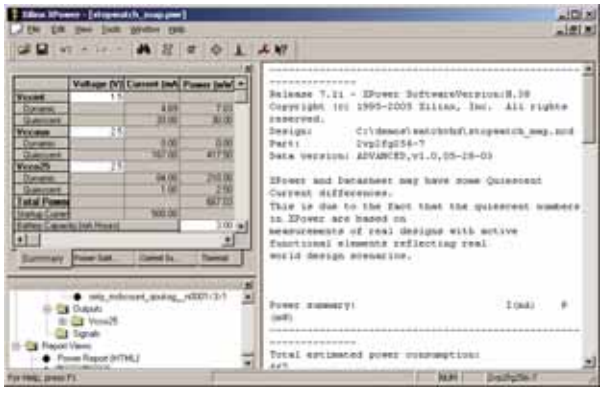


Figure 4 – ISE XPower power estimation tool

FPGA Design Techniques to Reduce Power

Several design-specific techniques can also reduce power consumption. These include constraining logic to a small area where possible, setting synthesis flags to minimize area, and minimizing layers of logic. Pipelining is also a good technique because it allows a higher timing constraint to be set, which allows reduction of capacitance and thus dynamic power reduction.

Setting Placement and Timing Constraints

Floorplanning the design properly reduces dynamic power. ISE Floorplanner allows you to create placement constraints. A more sophisticated floorplanning tool from Xilinx called the PlanAhead™ design tool allows you to observe hierarchy in a design and group sets of related logic into small areas. Using the placement and grouping constraints reduces the physical area, allowing you to achieve higher performance while minimizing routing capacitance and reducing dynamic power consumption.

Other techniques include constraining timing in a design. Synthesis tools as well as ISE routing and placement tools allow you to input timing constraints. If you raise the target timing constraint – especially the clock target – the router will try harder to meet it through more aggressive placement and routing efforts. The net effect is to minimize routes, which reduces routing power.

Other Partial Shutdown or Reprogramming Methods

Other commonly used design techniques can help reduce dynamic power consumption. One of these techniques is to use clock multiplexing. This entails turning off sections of the FPGA. A hardware feature in the Virtex-4 FPGA and its predecessors is a clock gating block, which allows a smooth way to turn off or on a global clock net. Better than clock enables on flip-flops, this method allows the entire large toggling clock net to be gated off, which saves power on the net and on the flip-flops.

Some types of designs, especially those used in battery-type applications, only need to consume power at certain times. In these applications, you can turn off clocks and lower the core voltage to the minimum

FPGA Family	Average Reduction Capacitance	Maximum % Capacitance Reduction
Spartan-3	11%	21 %
Virtex-4	6 %	9.5 %
Virtex-II Pro	13 %	18 %

Figure 5 – Interconnect capacitance reduction using power-based routing optimization in ISE 8.1i software

level that still allows FPGA data retention. In the Virtex-4 device, this is 0.9V. At this level, static power is reduced by greater than 60% when compared with the nominal 1.2V level.

A way to shrink the FPGA size required for a given set of tasks is to use the dynamic reconfigurability port, or DRP, which is available in Virtex-4 and other Xilinx FPGAs. If there are several functions that don't need to coexist, this port allows reloading only a portion of the FPGA. In doing so, you can choose a much smaller FPGA, which reduces static power.

Use of Embedded Blocks

Another method of reducing power consumption is through the use of embedded blocks. Although it is more work in some cases to instantiate special blocks, Virtex-4 FPGAs have a number of pieces of hard IP, which are essentially ASIC gates. Some of these functions are in fact automatically synthesized by some of the modern synthesis tools. These new blocks have between 5x and 20x lower power than programmable logic and programmable interconnect implementations. The embedded blocks reduce static power by not having extra transistors (as in programmable logic) and by not using programmable interconnect transistors. They reduce dynamic power by using only metal interconnects versus metal and programmable interconnects; reducing trace lengths; reducing extra node capacitance because of lack of pass transistors; and minimizing layers of logic.

Some of these blocks include the following:

- PowerPC™ – embedded high-performance processor
- DSP – XtremeDSP™ slice, a high-performance sophisticated multi-function arithmetic and logic block
- SSIO – New ChipSync™ block available in every I/O pin reduces logic cell counts for source-synchronous I/O designs
- Embedded Ethernet MAC(s) – no need to use logic and interconnect for MAC functions
- FIFO – SmartRAM memory includes built-in FIFO controllers
- SRL16 – Allows multiple cascade flip-flops to be used without programmable interconnects

Power-Based Routing Optimization

Another exciting tool available in the 8.1i release of ISE software allows the optimization of a design for power consumption. This initial release allows automatic capacitance minimization without the need for you to enter faster timing constraints.

Figure 5 shows testing results of this new interconnect capacitance reduction. It is estimated that interconnect capacitance is approximately two-thirds of the dynamic power, so this improvement can be valuable to reduce dynamic power.

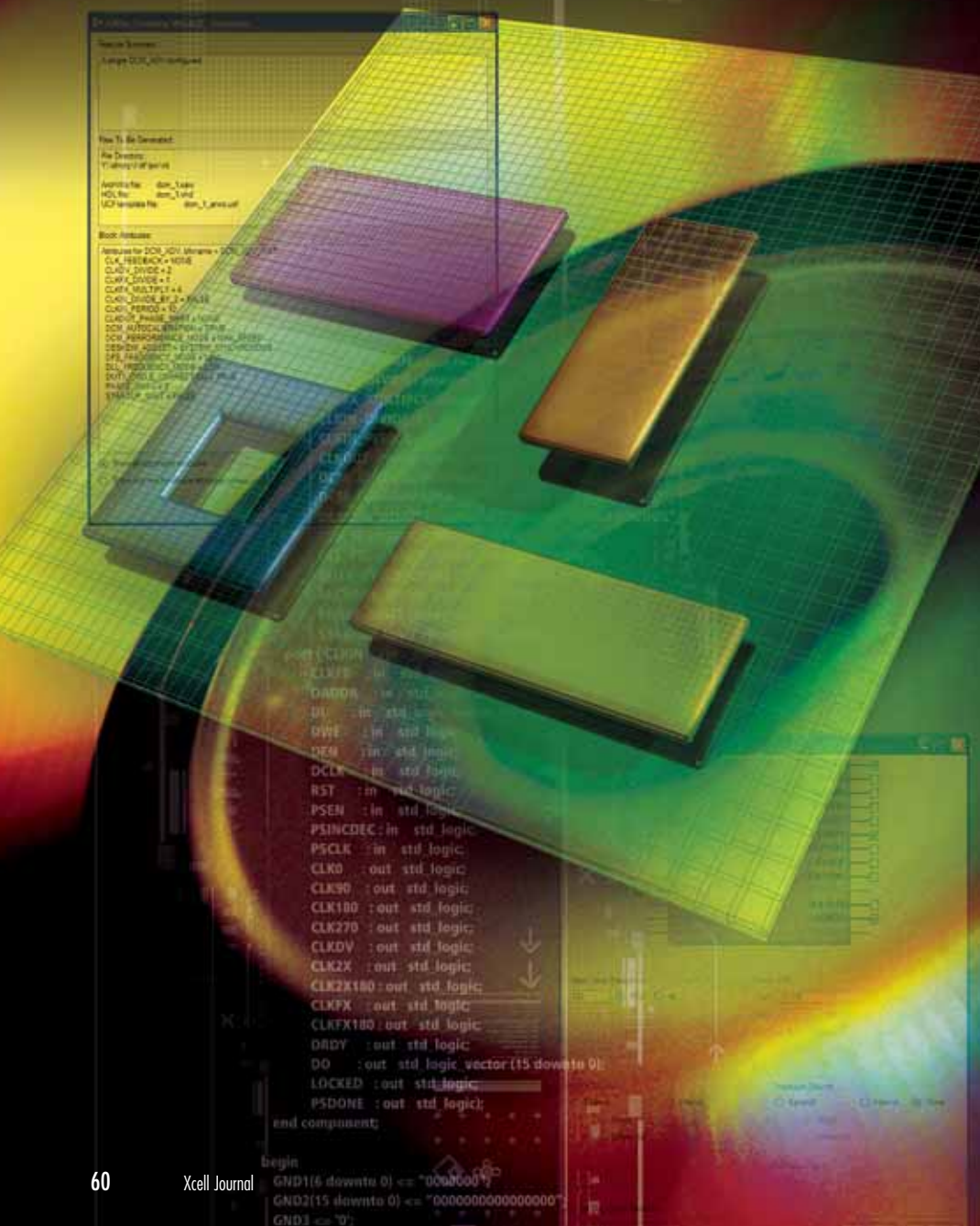
Future releases of ISE software will offer more power-optimization enhancements, including power-optimized synthesis and power-optimized placement.

Conclusion

It is very important to know the system power budget and operating environment. Understanding where various forms of power consumption come from allows you to adjust the FPGA environment and design characteristics to minimize power consumption and successfully meet a given power budget.

Using the ISE Foundation Architecture Wizards

Streamline the process of configuring and instantiating the complex blocks found in Xilinx devices.



by David W. Blevins
Staff Software Marketing Engineer
Xilinx, Inc.
david.blevins@xilinx.com

Xilinx® device architectures include several configurable functional blocks – including clocking, digital signal processing, and high-speed I/O blocks – that provide you with advanced functionality. Typically, these blocks are fairly complex in their operation and yet very flexible, so parameterizing them for the desired behavior can be a daunting and time-consuming task if done by hand.

The architecture wizards found in the ISE™ Foundation™ design environment can streamline the process of customizing such blocks. There are several wizards available; each one leads you through a sequence of screens that allows you to precisely define the behavior of the block at hand. Each screen has built-in design rule checking that ensures that the result will be "correct by construction" – that is, that the combination of selections made is a legal configuration of the target block.

After selecting "Finish" on the final screen, the wizard creates a customized block definition file (filename.xaw). ISE Foundation software then translates that definition into a HDL description of the block and creates a corresponding HDL instantiation template that you can then use in your design.

Examining the resulting HDL code and instantiation template generated by ISE Foundation software after the wizard finishes clearly shows the benefit of using an architecture wizard for this type of task.

The Clocking Wizard

As an example, let's use the clocking wizard to create a digital clocking manager module for a Virtex™-4 FPGA, which will be driven by a clock source external to the device. It will generate a main clock signal, as well as a frequency-doubled clock that drives an enabled buffer. A LOCKED signal will indicate when the DCM clock signal has stabilized after the FPGA is powered up or reset.

Figure 1 -Select IP dialog box

Examining the resulting HDL code and instantiation template generated by ISE Foundation software after the wizard finishes clearly shows the benefit of using an architecture wizard for this type of task.

Starting the Clocking Wizard

From within ISE's Project Navigator, we start the clocking wizard by selecting "Add New Source" and selecting the "IP (CoreGen and Architecture Wizard)" source type. The resulting dialog will let us select the "Single DCM ADV v8.1i" variant (see Figure 1).

Pressing "Next" and "Finish" on the subsequent dialog boxes invokes the clocking wizard.

General Setup Screen

On the first clocking wizard screen, we select the desired inputs and outputs of the block, the clock source, phase shift type, frequency, and feedback configuration (Figure 2).

Clock Buffers Screens

Selecting "Next" takes us to the clock buffers screen. By default, all clock outputs drive global buffers, but for our design, we need to place an enabled buffer after the frequency-doubled clock output. To do this, we select "Customize Buffers" and then click the "Global Buffer" button next to CLK2X to change its global buffer to an enabled buffer (Figure 3).

Figure 2 – General Setup screen

Figure 3 – Clock Buffers screen

Using the "Add Buffer" button on the next screen, we place an enabled buffer at the block's clock input (Figure 4).

Summary Screen

After creating the buffer by pressing the "OK" button, we click the "Next" button

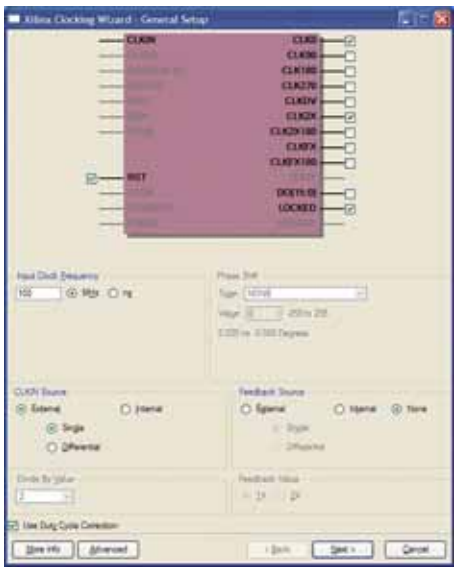


Figure 4 – View/Edit Buffer screen

Figure 5 – Summary screen

on the clock buffers screen to advance to the summary page, which provides a detailed report on the block that will be generated (Figure 5).

Selecting the "Finish" button creates a binary filename.xaw source file in the ISE Foundation project directory. The .xaw will automatically be converted by ISE Foundation software to the corresponding HDL description required for synthesis when you implement your design, but you can view that HDL at any time after the .xaw file has been created. Either VHDL or Verilog is available; VHDL is shown in our example.

Here is where you can really see the true value of the architecture wizard – imagine having to write this code by hand:

```
-- Module dcm1
-- Generated by Xilinx Architecture Wizard
-- Written for synthesis tool: XST

library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
-- synopsys translate_off
library UNISIM;
use UNISIM.Vcomponents.ALL;
-- synopsys translate_on

entity dcm1 is
  port ( CLKIN_ENABLE_IN : in    std_logic;
         CLKIN_IN        : in    std_logic;
         RST_IN          : in    std_logic;
         CLKIN_IBUFG_OUT : out   std_logic;
         CLKIN_OUT       : out   std_logic;
         CLK0_OUT        : out   std_logic;
         LOCKED_OUT      : out   std_logic);
end dcm1;

architecture BEHAVIORAL of dcm1 is
  signal CLKFB_IN        : std_logic;
  signal CLKIN_IBUFG     : std_logic;
  signal CLK0_BUF        : std_logic;
  signal GND1            : std_logic_vector (6 downto 0);
  signal GND2            : std_logic_vector (15 downto 0);
  signal GND3            : std_logic;
  component BUFGCE
    port ( I  : in    std_logic;
           CE : in    std_logic;
           O  : out   std_logic);
  end component;

  component IBUFG
    port ( I : in    std_logic;
           O : out   std_logic);
  end component;

  component BUFG
    port ( I : in    std_logic;
           O : out   std_logic);
  end component;

  component DCM_ADV
    generic( CLK_FEEDBACK : string :=  "1X";
             CLKDV_DIVIDE : real :=  2.000000;
             CLKFX_DIVIDE : integer :=  1;
             CLKFX_MULTIPLY : integer :=  4;
             CLKIN_DIVIDE_BY_2 : boolean :=  FALSE;
             CLKIN_PERIOD : real :=  10.000000;
             CLKOUT_PHASE_SHIFT : string :=  "NONE";
             DCM_AUTOCALIBRATION : boolean :=  TRUE;
                    DCM_PERFORMANCE_MODE  :  string  :=
"MAX_SPEED";
             DESKEW_ADJUST : string :=  "SYSTEM_SYNCHRO-
             NOUS";
             DFS_FREQUENCY_MODE : string :=  "LOW";
             DLL_FREQUENCY_MODE : string :=  "LOW";
             DUTY_CYCLE_CORRECTION : boolean :=  TRUE;
             FACTORY_JF : bit_vector :=  x"F0F0";
             PHASE_SHIFT : integer :=  0;
             STARTUP_WAIT : boolean :=  FALSE);
    port ( CLKIN    : in    std_logic;
           CLKFB    : in    std_logic;
           DADDR    : in    std_logic_vector (6 downto 0);
           DI       : in    std_logic_vector (15 downto 0);
           DWE      : in    std_logic;
           DEN      : in    std_logic;
           DCLK     : in    std_logic;
           RST      : in    std_logic;
           PSEN     : in    std_logic;
           PSINCDEC : in    std_logic;
           PSCLK    : in    std_logic;
           CLK0     : out   std_logic;
           CLK90    : out   std_logic;
           CLK180   : out   std_logic;
           CLK270   : out   std_logic;
           CLKDV    : out   std_logic;
           CLK2X    : out   std_logic;
           CLK2X180 : out   std_logic;
           CLKFX    : out   std_logic;
           CLKFX180 : out   std_logic;
           DRDY     : out   std_logic;
           DO       : out   std_logic_vector (15 downto 0);
           LOCKED   : out   std_logic;
           PSDONE   : out   std_logic);
  end component;

begin
  GND1(6 downto 0) <= "0000000";
  GND2(15 downto 0) <= "0000000000000000";
  GND3 <= '0';
  CLK0_OUT <= CLKFB_IN;
  CLKIN_BUFGCE_INST : BUFGCE
    port map (CE=>CLKIN_ENABLE_IN,
              I=>CLKIN_IBUFG,
              O=>CLKIN_OUT);

  CLKIN_IBUFG_INST : IBUFG
    port map (I=>CLKIN_IN,
              O=>CLKIN_IBUFG);

  CLK0_BUFG_INST : BUFG
    port map (I=>CLK0_BUF,
              O=>CLKFB_IN);

  DCM_ADV_INST : DCM_ADV
  generic map( CLK_FEEDBACK => "1X",
               CLKDV_DIVIDE => 2.000000,
               CLKFX_DIVIDE => 1,
               CLKFX_MULTIPLY => 4,
               CLKIN_DIVIDE_BY_2 => FALSE,
               CLKIN_PERIOD => 10.000000,
               CLKOUT_PHASE_SHIFT => "NONE",
               DCM_AUTOCALIBRATION => TRUE,
               DCM_PERFORMANCE_MODE => "MAX_SPEED",
               DESKEW_ADJUST => "SYSTEM_SYNCHRONOUS",
               DFS_FREQUENCY_MODE => "LOW",
               DLL_FREQUENCY_MODE => "LOW",
               DUTY_CYCLE_CORRECTION => TRUE,
               FACTORY_JF => x"F0F0",
               PHASE_SHIFT => 0,
               STARTUP_WAIT => FALSE)
    port map (CLKFB=>CLKFB_IN,
              CLKIN=>CLKIN_IBUFG,
              DADDR(6 downto 0)=>GND1(6 downto 0),
              DCLK=>GND3,
              DEN=>GND3,
              DI(15 downto 0)=>GND2(15 downto 0),
              DWE=>GND3,
              PSCLK=>GND3,
              PSEN=>GND3,
              PSINCDEC=>GND3,
              RST=>RST_IN,
              CLKDV=>open,
              CLKFX=>open,
              CLKFX180=>open,
              CLK0=>CLK0_BUF,
              CLK2X=>open,
              CLK2X180=>open,
              CLK90=>open,
              CLK180=>open,
              CLK270=>open,
              DO=>open,
              DRDY=>open,
              LOCKED=>LOCKED_OUT,
              PSDONE=>open);

end BEHAVIORAL;
```

The architecture wizards augment other Xilinx IP block creation tools such as Platform Studio, CORE Generator software, and System Generator for DSP – all of which help you to get your product to market as quickly as possible.

The preceding HDL source code describes the DCM module to the synthesis tool that we are using, but we will still need to create an instance of the module in our design. This is achieved with an "instantiation template," which is created by using the "View HDL Instantiation Template" process in Project Navigator. The resulting code snippet can be inserted into our design to create an instance of the DCM module:

```
COMPONENT dcm1
PORT(
        CLKIN_ENABLE_IN : IN std_logic;
        CLKIN_IN : IN std_logic;
        RST_IN : IN std_logic;
        CLKIN_IBUFG_OUT : OUT std_logic;
        CLKIN_OUT : OUT std_logic;
        CLK0_OUT : OUT std_logic;
        LOCKED_OUT : OUT std_logic
        );
END COMPONENT;

Inst_dcm1: dcm1 PORT MAP(
        CLKIN_ENABLE_IN => ,
        CLKIN_IN => ,
        RST_IN => ,
        CLKIN_IBUFG_OUT => ,
        CLKIN_OUT => ,
        CLK0_OUT => ,
        LOCKED_OUT =>
);
```

Supported Block Configurations

The above example illustrates only a single configuration of one type of block that the architecture wizards support. Other supported blocks and configurations include:

Clocking Wizard

The digital clock management module provides you with extensive control over the global clocking configuration(s) in your design and features the choice of either an external or internal clock source, clock deskew, phase shift control, and frequency synthesis.

The clocking wizard can create several DCM configurations using one or more DCM modules:

- Single DCM
- Single DCM_ADV
- Clock forwarding/board deskew
- Board deskew with an internal deskew
- Clock switching with two DCMs
- Cascading in series with two DCMs
- PMCD (Virtex-4 FPGAs)

Rocket IO Wizard

The Xilinx Rocket IO™ wizard configures the multi-gigabit transceiver block in Virtex-II Pro and Virtex-II Pro X devices (Virtex-4 Rocket IO support is found in CORE Generator™ software). For certain protocols, it also allows the configuration of multiple channels of transceivers.

For Virtex-II Pro devices, the following Rocket IO protocols are available through the wizard:

- Fibre Channel
- Gigabit Ethernet
- XAUI
- Infiniband
- Aurora
- PCI Express
- SONET OC-48 (Virtex-II Pro X devices)
- SONET OC-192 (Virtex-II Pro X devices)
- Custom user-defined configurations

Xtreme DSP Wizard (Virtex-4 Devices)

This wizard can create several different types of DSP blocks using the Virtex-4 Xtreme DSP™ slice, including:

- Multiplier
- Accumulator
- Multiplier-accumulator (MAC)
- Adder/subtractor

ChipSync Wizard (Virtex-4 Devices)

Two basic types of ChipSync configurations can be created:

- The memory applications mode configures a block of I/O for memory application usage. The wizard allows you to set up the data bus and clocks/strobes, including specifying delay information. You can also configure the address bus, reference clocks, and control signals.
- The non-memory applications mode configures a block of I/O for non-memory application usage, such as for networking cases. The wizard allows you to set up the data bus and clocks, including specifying delay information. You can also configure reference clocks and control signals.

Conclusion

As shown in this article, the architecture wizards act as intelligent assistants that facilitate easy creation of customized instances of the various built-in complex blocks found in Xilinx FPGAs.

The architecture wizards augment other Xilinx IP block creation tools such as Platform Studio, CORE Generator software, and System Generator for DSP – all of which help you to get your product to market as quickly as possible.

FREE on-line training *with* **Demos On Demand**

A series of compelling, highly technical product demonstrations, presented by Xilinx experts, is now available on-line. These comprehensive videos provide excellent, step-by-step tutorials and quick refreshers on a wide array of key topics. The videos are segmented into short chapters to respect your time and make for easy viewing.

Ready for viewing, anytime you are

Offering live demonstrations of powerful tools, the videos enable you to achieve complex design requirements *and* save time. A complete on-line archive is easily accessible at your fingertips. Also, a *free* DVD containing all the video demos is available at **www.xilinx.com/dod**. Order yours today!

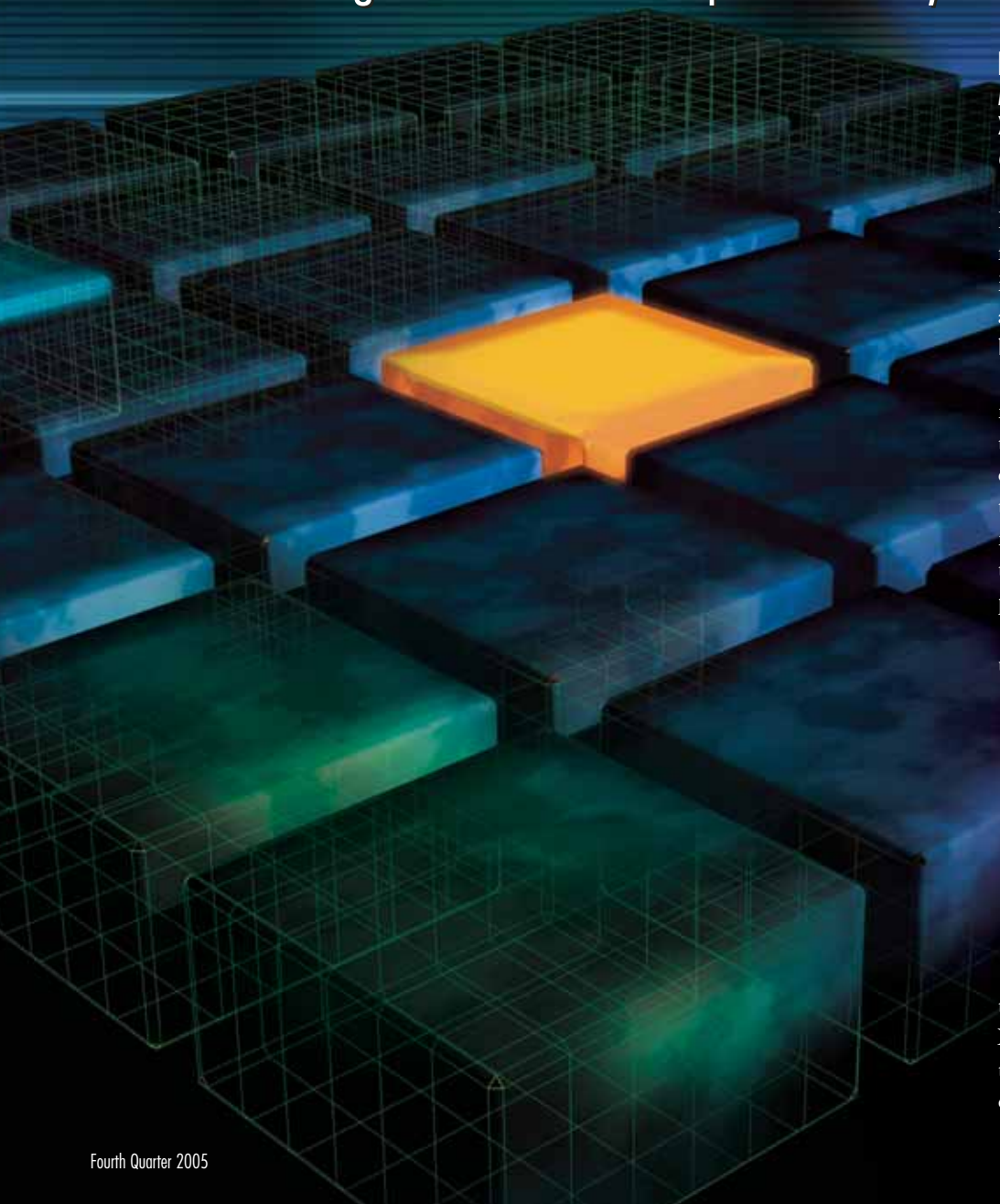
XILINX®

www.xilinx.com/dod

Pb-free devices available now

Benefits of Partial Reconfiguration

Take advantage of even more capabilities in your FPGA.



by Cindy Kao
Marketing Specialist
Xilinx, Inc.
cindy.kao@xilinx.com

Partial reconfiguration offers countless benefits across multiple industries. It can be an important component to any design or application – allowing designers more capabilities and resources than meets the eye.

Partial reconfiguration is the ability to reconfigure select areas of an FPGA anytime after its initial configuration. You can do this while the design is operational and the device is active (known as active partial reconfiguration) or when the device is inactive in shutdown mode (known as static partial reconfiguration).

By taking advantage of partial reconfiguration, you gain the ability to:

- Adapt hardware algorithms
- Share hardware between various applications
- Increase resource utilization
- Provide continuous hardware servicing
- Upgrade hardware remotely

Xilinx has supported partial reconfiguration for many generations of devices. All Xilinx® FPGAs support partial reconfiguration, from Virtex™-4 devices to our lowest cost FPGAs, the Spartan™-3/E family.

How Partial Reconfiguration Works

Xilinx supports two basic styles of partial reconfiguration: module-based and difference-based. Module-based partial reconfiguration uses modular design concepts to reconfigure large blocks of logic. The distinct portions of the design to be reconfigured are known as reconfigurable modules. Because specific properties and specific layout criteria must be met with respect to a reconfigurable module, any FPGA design intending to use partial reconfiguration must be planned and laid out with that in mind. These properties and layout criteria are outlined in the Xilinx Development System Reference Guide, which can be found at *http://toolbox.xilinx.com/docsan/xilinx7/books/docs/dev/dev.pdf*.

Difference-based partial reconfiguration is a method of making small changes in an FPGA design, such as changing I/O standards, LUT equations, and block RAM content. There are two supported ways to make such design changes: at the front end or the back end. Front-end changes can be HDL or schematic. You must re-synthesize and re-implement the design, creating a new placed and routed native circuit description (NCD) file. Back-end modifications are made in FPGA Editor, a GUI tool within ISE™ software used to view/edit device layout and routing.

You can also perform partial reconfiguration by implementing a basic controller to manage the reconfiguration of an FPGA. This could be in the form of an embedded or external processor.

Xilinx offers a suite of processor solutions. The PicoBlaze™ and MicroBlaze™ soft-core processors both support the Spartan and Virtex families. The Virtex-II Pro FPGA embodies the hard-processor solution with the integration of an IBM PowerPC™ 405 32-bit RISC processor into the FPGA. Now, with the introduction of the Virtex-4 FX platform FPGA, Xilinx has increased processing power by introducing two PowerPC 405 processor cores in a single device (see Table 1).

For more information, visit Processor Central at *www.xilinx.com/products/design_resources/proc_central/index.htm*.

Benefits

There are many benefits that come with partially reconfigurable devices:

- Applications. Partial reconfiguration is useful in a variety of applications across many industries. The aerospace and defense industries have certainly taken advantage of its capabilities. Partially reconfigurable devices have benefited the Joint Tactical Radio System (JTRS) Program, which I'll discuss more in the following section.
- Increased system performance. Although a portion of the design is being reconfigured, the rest of the system can continue to operate. There is no loss of performance or functionality with unaffected portions of a design – no down time. It also allows for multiple applications on a single FPGA.
- The ability to change hardware. Xilinx FPGAs can be updated at any time, locally or remotely. Partial reconfiguration allows you to easily support, service, and update hardware in the field.
- Hardware sharing. Because partial reconfiguration allows you to run multiple applications on a single FPGA, hardware sharing is realized. Benefits include reduced device count, reduced power consumption, smaller boards, and overall lower costs.
- Shorter reconfiguration times. Configuration time is directly proportional to the size of the configuration bitstream. Partial reconfiguration allows you to make small modifications without having to reconfigure the entire device. By changing only portions of the bitstream – as opposed to reconfiguring the entire device – the total reconfiguration time is shorter.

Processor	Type of Processor	Supported Devices
PicoBlaze	Soft-Processor Core	Virtex Family Spartan Family CoolRunner-II CPLDs
MicroBlaze	Soft-Processor Core	Virtex Family Spartan Family
PowerPC	Hard Processor	Virtex-II Pro Virtex-4 FX

Table 1 – Xilinx processor solutions and supported devices

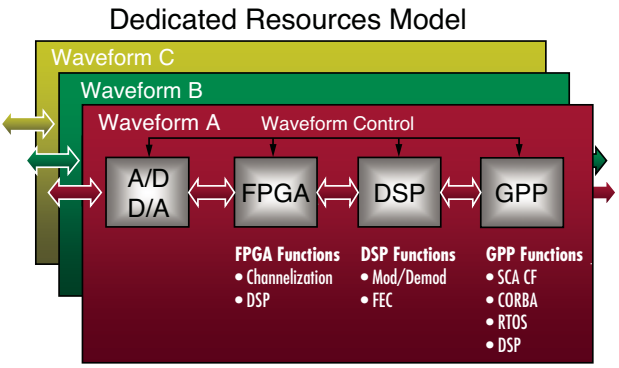


Figure 1 – Dedicated resources model: three-channel SDR modem

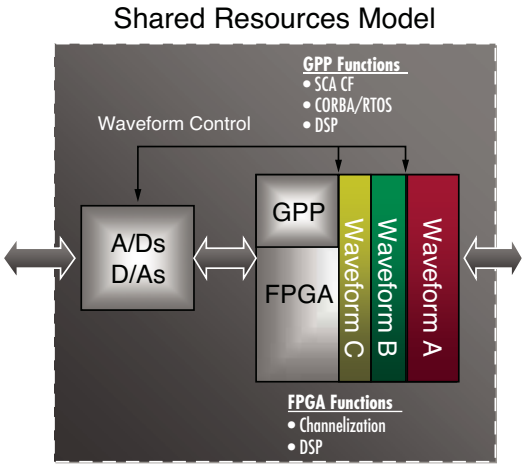


Figure 2 – Shared resources model: three-channel SDR modem

Applications

Partial reconfiguration is the cornerstone for power-efficient, cost-effective software-defined radios (SDRs). Through the JTRS Program, SDRs are becoming a reality for the defense industries as an effective and necessary tool for

communication. SDRs satisfy the JTRS standard by having both a software-reprogrammable operating environment and the ability to support multiple channels and networks simultaneously.

Figure 1 shows a three-channel SDR modem supporting a Software Communications Architecture Core Framework (SCA CF), as mandated for JTRS. Current implementations of SCA-enabled SDR modems with multiple channels require multiple sets of processing resources and a dedicated set of hardware for each channel. The more channels SDR must support, the more dedicated resources needed. This aversely affects space, weight, power consumption, and cost.

With partial reconfiguration, the ability to implement an SDR modem using shared resources is realized, as shown in Figure 2. A shared resources model enabled by partial reconfiguration of an FPGA to support multiple waveforms can be supported by the SCA as mandated by JTRS. FPGA implementations of SDR, with partial reconfiguration, results in effective use of resources, lower power consumption, and extensive cost savings.

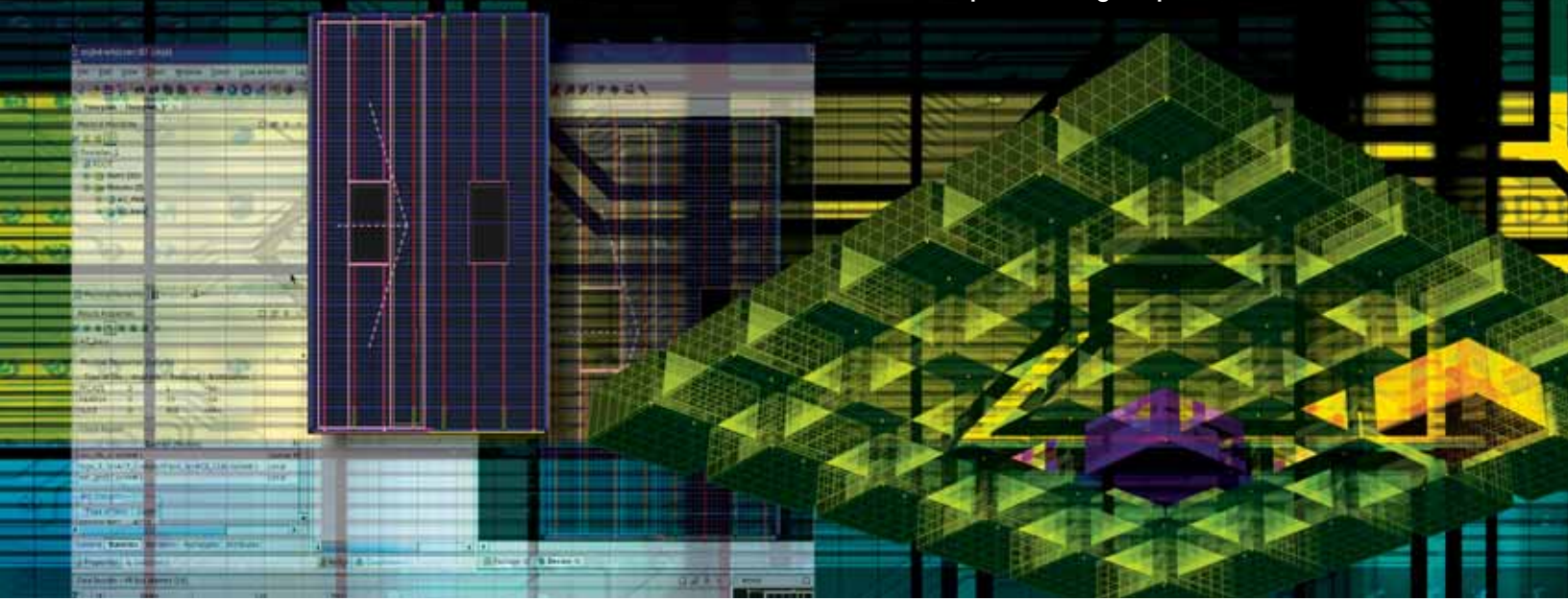
Partial reconfiguration can also be used in many other applications. Another example is in mitigation and recovery from single-event upsets (SEU). In-orbit, space-based, and extra-terrestrial applications have a high probability of experiencing SEUs. By performing partial reconfiguration, in conjunction with readback, a system can detect and repair SEUs in the configuration memory without disrupting its operations or completely reconfiguring the FPGA. (Readback is the process of reading the internal configuration memory data to verify that current configuration data is correct.)

Conclusion

The capabilities and benefits offered by partial reconfiguration reach across many industries and applications. Leverage partial reconfiguration by using Xilinx FPGAs in your next design, the only truly partially reconfigurable devices. You can take advantage of any of its benefits, from remote hardware upgrading to on-chip hardware sharing, and give your designs the reconfigurability advantage.

PlanAhead Software as a Platform for Partial Reconfiguration

PlanAhead software delivers a streamlined environment to reduce space, weight, power, and cost.



by Nij Dorairaj
Senior Staff Engineer
Xilinx, Inc.
nij.dorairaj@xilinx.com

Eric Shiflet
Technical Marketing Engineer
Xilinx, Inc.
eric.shiflet@xilinx.com

Mark Goosman
Product Marketing Manager
Xilinx, Inc.
mark.goosman@xilinx.com

The architecture of the Xilinx® Virtex™ platform family of FPGAs allows design modules to be swapped on-the-fly using a partial reconfiguration (PR) methodology. This powerful capability allows multiple design modules to time-share resources on a single device, even while the base design operates uninterrupted.

Partial reconfiguration is a process of device configuration that allows a limited, predefined portion of an FPGA to be reconfigured while the remainder of the device continues to operate. This is especially valuable where devices operate in a mission-critical environment that can not be disrupted while some subsystems are being redefined.

Using partial reconfiguration, you can dramatically increase the functionality of a single FPGA, allowing for fewer, smaller devices than would otherwise be needed. Important applications for this technology include reconfigurable communication and cryptographic systems.

Virtex Configuration Background

Partial reconfiguration is supported in both the Virtex and Spartan™ families. In this article, we will focus on implementing the partial reconfiguration methodology as it applies to Virtex-II and Virtex-II Pro FPGAs.

All user-programmable features inside a Virtex FPGA are controlled by memory cells that are volatile and must be configured on power-up. These memory cells are known as the configuration memory, and define the look-up table (LUT) equations, signal routing, input/output block (IOB) voltage standards, and all other aspects of the design.

To program configuration memory, instructions for the configuration control logic and data for the configuration memory are provided in the form of a bitstream, which is delivered to the device through the JTAG, SelectMAP, serial, or ICAP configuration interface.

A programmed Virtex FPGA can be partially reconfigured using a partial bitstream. You can use partial reconfiguration to change the structure of one part of an FPGA design as the rest of the device continues to operate.

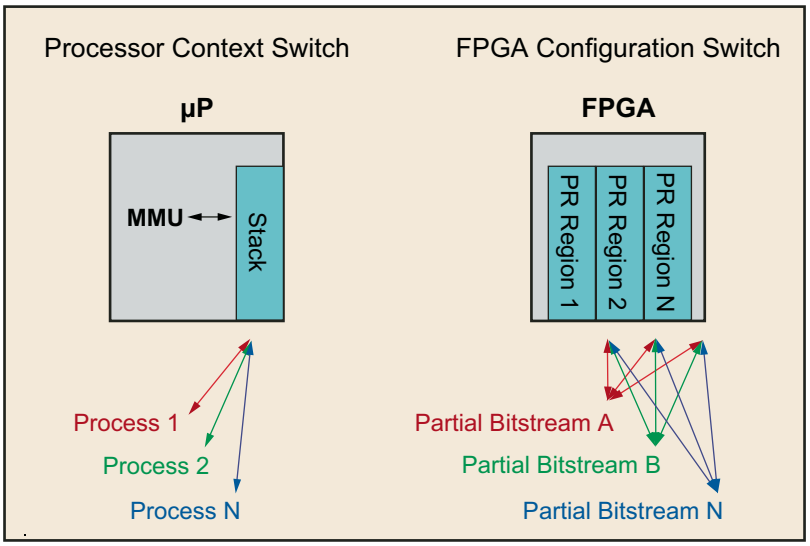


Figure 1 – The analogy between microprocessor context switching and FPGA partial reconfiguration regions

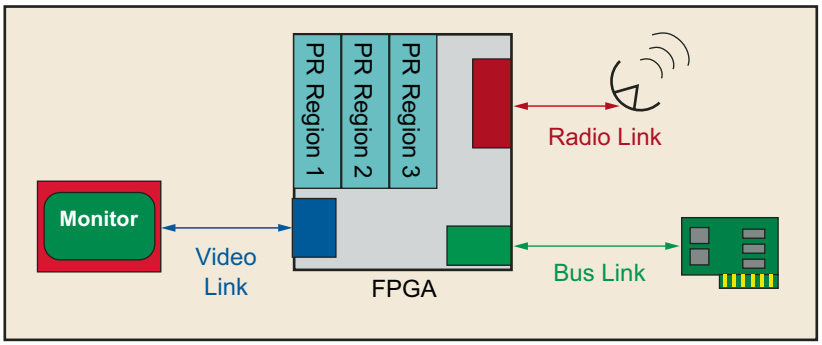


Figure 2 – Maintaining real-time links during partial reconfiguration

Use Cases

Partial reconfiguration is useful for systems with multiple functions that can time-share the same FPGA device resources. In such systems, one section of the FPGA continues to operate, while other sections of the FPGA are disabled and partially reconfigured to provide new functionality. This is analogous to the situation shown in Figure 1, where a microprocessor manages context switching between software processes. Except in the case of partial reconfiguration of an FPGA, it is the hardware – not the software – that is being switched.

Partial reconfiguration provides an advantage over multiple full bitstreams in applications that require continuous operation not otherwise accessible during full reconfiguration. One example, illustrated in Figure 2, is a graphics display that utilizes horizontal and vertical synchronization. Because of the environment in which this application operates, signals from radio and video links need to be preserved – but the format and data processing format require updates and changes during operation. With partial reconfiguration, the system can maintain these real-time links while other modules within the FPGA are changed on the fly.

Methodology

To implement a partial reconfiguration design successfully, you have to follow a strict design methodology. Here are some of the guidelines to follow:

- Insert bus macros between modules that need to be swapped out (called partial reconfiguration modules, or PRMs) and the rest of the design (static logic)
- Follow synthesis guidelines to generate a partially reconfigurable netlist
- Floorplan the PRMs and cluster all static modules
- Place bus macros
- Follow PR-specific design rules
- Run the partial reconfiguration implementation flow

PlanAhead™ software provides a single environment (or platform) to manage the preceding guidelines.

Here is a list of the steps in which you can use PlanAhead design tools to implement a partial reconfiguration design:

- Netlist import
- Floorplanning for partial reconfiguration
- Design rule checks
- Netlist export
- Implementation flow management
- Bitstream size estimation

Netlist Import

PlanAhead software works with any synthesized netlist, such as XST or Synplify. You can import any hierarchical netlist (single edf/ngc or multiple edf/ngc files). Follow the regular guidelines to import the design into PlanAhead design tools and create a floorplan for the design.

Floorplanning for Partial Reconfiguration

This is an important step in the partial reconfiguration flow. Floorplanning within PlanAhead software is based on design partitions referred to as physical blocks, or Pblocks. A Pblock can have an area (such as a rectangle) defined on the FPGA device to constrain the logic. You can define Pblocks without rectangles and ISE™ software will attempt to group the logic during placement. Netlist logic placed inside of Pblocks will receive AREA_GROUP constraints.

The key floorplanning tasks for partial reconfiguration are:

1. Setting up a PRM:

- Assign an area for the PRM by creating a Pblock with an area defined within the fabric
- Assign RANGES for the Pblock

- Define the MODE=RECONFIG attribute on the PRM

  (Pblock property->attribute section)

2. Setting up static logic:

- Every top-level module, other then PRMs, should be grouped together in a single Pblock. This is called a static logic block. This block should not have a RANGE defined; this will cluster the static logic together in a single Pblock. Select all top-level modules (except the PRM) and assign it to a Pblock. Figure 3 shows the static logic grouped in a Pblock named AG_base.

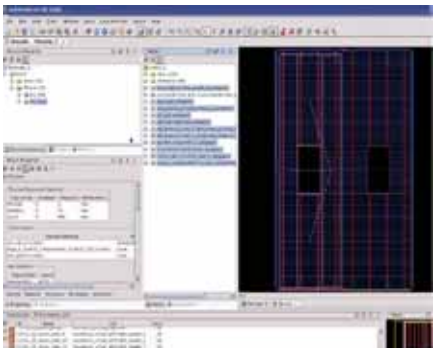


Figure 3 – All static logic is grouped in a single Pblock AG_base.

- When you have completed the floorplanning in PlanAhead design tools, the resulting physical hierarchy will be organized as shown in Figure 4.

3. Bus macro placement:

- Bus macros are physical ports that connect a PRM to static logic. Any connection from a PRM to static logic should always go through a bus macro. Bus macros are instantiated as black boxes in RTL and are filled with a predefined routing macro in the form of an .nmc file. Bus macros are placed on the PRM boundary. Static logic connected to PRMs will migrate towards the bus macro during placement.

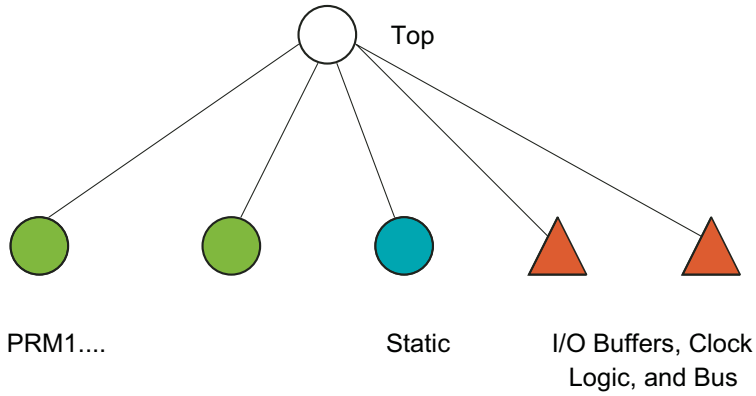


Figure 4 – The hierarchy for a partial reconfiguration design

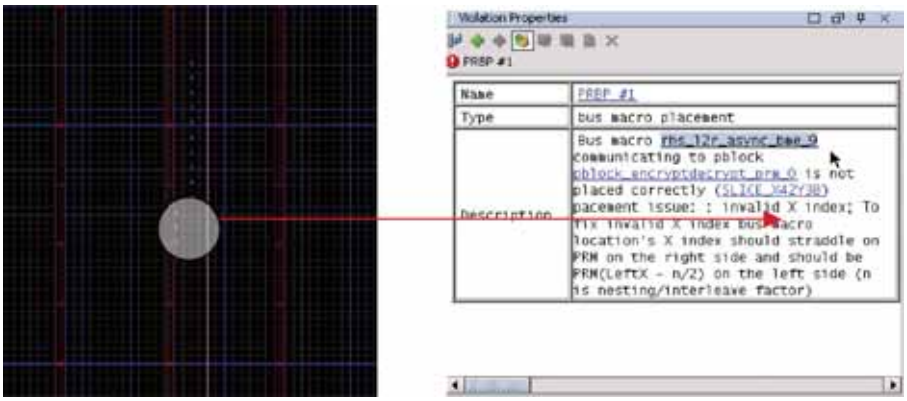


Figure 5 – The PRBP DRC verifies all rules that should be followed for bus macro placement.

Design Rule Checks

Given the complexity of the flow, it is common for mistakes to be introduced in the original RTL and during the floorplanning process. PlanAhead-PR design tools check for design violations. Also integrated into this feature is the PR-Advisor, which provides feedback on how to improve your design.

The PR design rule checks are:

1. Bus macro DRC. This provides verification for all design rules related to bus macro connectivity and placement. One example of a bus macro DRC is the PRBP check. This DRC checks for all rules that should be followed for bus macro placement. Figure 5 shows an example of a design that failed the PRBP DRC. In this case, the interleaved/nested macro should be placed at SLICE_X41Y*.

2. Floorplanning DRC. This covers floorplanning rules. Clock objects (global clock buffers, DCM) and I/Os should be placed and static logic clustered.

3. Glitching logic DRC. This verifies glitching logic elements (SRL and distributed RAM) above and below PRM regions.

4. Timing Advisor/DRC. This provides a check for timing-related issues. One example of timing DRC is the PRTP check. The static module is implemented before the PRM during the implementation phase. Regular timing constraints do not cover the paths that cross between the static and a PRM. This does not present a problem, provided that the bus macro is synchronous. However, if it is an asynchronous bus macro, the static module does not know about the propagating of asynchronous paths, as shown in Figure 6. This could be important if these paths are timing-critical. One way to pass this information to the static module is to specify a TPSYNC constraint on the bus macro output net. PlanAhead software will recommend a TPSYNC constraint that can be added to the .UCF file.

Netlist Export

Once the design is floorplanned and passes the DRC checker, it is ready to be exported. PlanAhead design tools take care of exporting the original hierarchical netlist into a PR-style netlist that has a specific format (static and PRM in separate directories). The export directory will appear as shown in Figure 7.

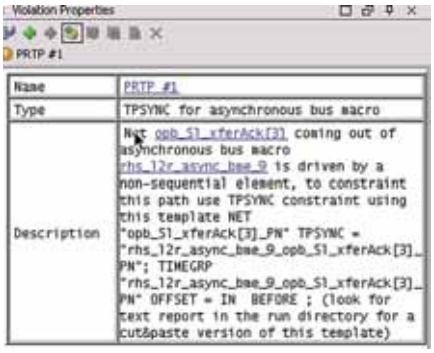


Figure 6 – Special consideration should be given to timing-critical paths, which include an asynchronous path.

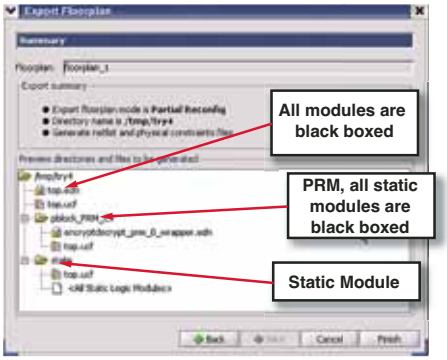


Figure 7 – Upon successful completion of the floorplanning DRC processes, all modules will be displayed in black text within the Export Floorplan dialog box.

Implementation Flow Management

The partial reconfiguration flow wizard, shown in Figure 8, runs the partial reconfiguration implementation on the exported design. It will produce a full bitstream for the complete design and a partial bitstream for each of the PRMs. The implementation steps are:

- Initial budgeting
- Static module implementation
- PRM implementation (one implementation for each version of every PRM)
- Assembly and bitstream generation (results are stored in the merge directory)

To run the tools, start the flow wizard by selecting "Tools > Run Partial Reconfig." This wizard will guide you through each of the implementation steps. You can either run the implementation or just generate the scripts, which can be used from a command line outside of the PlanAhead user interface.

Bitstream Size Estimation

The Pblock statistics report includes a section that reports PRM bitstream size (Figure 9). This information can be used for estimating the size of configuration memory storage such as external flash and DDR. This information can also be used to calculate how long it will take to swap the module based on your bitstream memory interface.

Conclusion

PlanAhead software is the first graphical environment for partial reconfiguration. Using PlanAhead design tools as a platform for partial reconfiguration applications can greatly simplify the complexities of juggling the dynamic operating environment of these cutting-edge applications, allowing a single device to operate in applications that previously required multiple FPGAs.

The methodology offered by PlanAhead software can dramatically increase productivity and decrease time-to-solution for designers using partial reconfiguration.

The capability of designs to leverage partial reconfiguration opens doors to a whole host of applications. By providing a platform that leverages the advantages of partial reconfiguration for designs targeting Virtex-II and Virtex-II Pro FPGAs, PlanAhead design tools allow you to dramatically extend the functionality of your design. With similar support for designs targeting the Virtex-4 multi-platform FPGA in the near future, the application space for partial reconfiguration is practically limitless.

Figure 8 – The PlanAhead partial reconfiguration flow wizard provides an easy-to-follow flow.

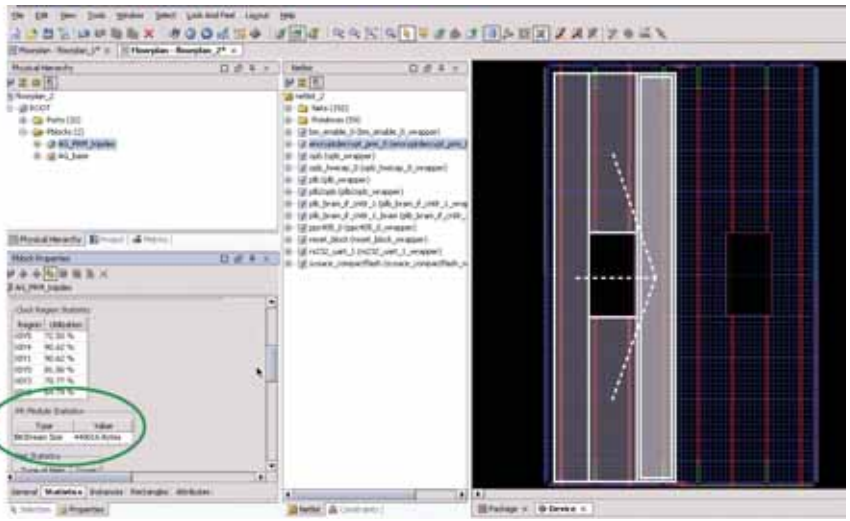


Figure 9 – The display of Pblock properties includes the estimated size of the bitstream.

Supporting Players

Titanium Dedicated Engineering provides assistance in overcoming technical obstacles.

by Jonathan Trotter
Titanium Business Development Manager
Xilinx, Inc.
jonathan.trotter@xilinx.com

Once a specification has been defined and authorized, several critical stages remain in every design cycle. The first stage is design entry. HDL coding is the most prominent form of entry when designing platform FPGAs. The code must be written properly for the design to pass through synthesis. Xilinx recommends that designers simulate their HDL before moving on.

The next stage is implementation, with the most important aspect being place and route. Implementation maps the HDL-entered design into FPGA building blocks, creating a bitstream. Analysis is required before downloading the bitstream into an FPGA. Downloading the FPGA without this analysis can damage the device, or even the system.

Designs must also meet system timing parameters. If timing is not met after an implementation, then the code, timing constraints, implementation options, or internal logic placement must be changed. Yet adjusting any one of these areas can yield your desired results or further remove you from them.

Determining which area to adjust first can be challenging to design teams, especially those feeling the heat of a deadline. The best design teams are those that can make the necessary changes or corrections the fastest. Once the changes have been made, a final verification or timing simulation is required. If the design does not pass, more modifications are needed.

To the Rescue

Xilinx has the expertise to solve your problem and meet your deadline regardless of your application or specification requirements. Under our Titanium Dedicated Engineering program, Xilinx engineers can successfully:

- Aid designers with embedded processing, timing closure, signal integrity, and DSP challenges
- Assist teams starting their first FPGA design with design flow and fundamentals
- Help with adding a feature to a design that is already at capacity
- Meet a deadline with timing closure assistance

Titanium Dedicated Engineering can improve your design productivity and accelerate your time to market by providing you with a dedicated application engineer on a contract basis. This expert can offer the technical assistance that your team and design require, either remotely or at your site. Not only will your design team have access to the engineer's expertise, they will also have access to the knowledge possessed by Xilinx product and development groups.

Design Challenges

To create a complete and powerful platform FPGA design requires knowledge, skill, time, resources, and patience. To fully utilize and take advantage of all of the new features of a Xilinx device requires ramp-up time for any design team, regardless of their experience level. A good example of this is Xilinx embedded processing technology. Designing with embedded features requires extra skills and knowledge above and beyond those needed for a successful FPGA design. With time, most capable teams will learn how take advantage of these features, but the competition may learn faster with assistance.

Each design team operates and functions differently, but they all seem to have similar practices and styles. The greater familiarity they have with FPGAs, the easier it is to utilize advanced features like the embedded processor or multi-gigabit transceiver, or to reach desired performance levels. Problems can occur if the team is relatively new or inexperienced. The quicker designers adopt our fundamental design style, the quicker they will complete successful designs. For example, timing verification must be performed before transferring the design in software to silicon. The step is not mandatory, but precautionary.

Success Story

Many ASIC design teams prototype their ideas in an FPGA. This allows them to make changes or modifications without the time and expense of re-spinning an ASIC. One Xilinx customer wanted to implement an ARM processor core and additional custom logic inside a Xilinx device. The design was so large that it had to be split up between two Virtex™-II devices. It can be challenging enough to successfully implement a function in one platform FPGA, let alone splitting it across two of them.

The customer's design was not meeting timing, nor could they configure their system utilizing our System ACE™ technology. Because two devices were required, board connectivity techniques were critical; the board timing and FPGA timing had to be in harmony for the two devices to function in unison.

There were other several crucial design challenges for this team:

- They were one month behind schedule.
- They had a customer demonstration in six weeks
- System frequency was 15 Mhz below the application's minimum requirement
- Configuring via JTAG was successful, but not utilizing System ACE technology

Xilinx sent a Titanium Dedicated Engineer to work onsite with the team to identify their timing, integration, and configuration issues. The engineer first helped them understand the device architecture to ensure that they had the knowledge to utilize the device features most applicable to their design. In this case, the application engineer quickly identified and recommended that the team take advantage of the DCMs as well as IOB registers.

After an architecture-specific redesign, the next issue was to apply proper timing and area constraints based on their timing report. The constraints force Xilinx place and route tools to concentrate on the most critical areas of the design. The application engineer also focused on educating the design team so that they could apply these fundamentals on their own during their next design.

The main culprit of the configuration issue turned out to be a faulty crystal oscillator. The frequency was a little lower than what its datasheet specified. Once it was replaced, the FPGAs could be configured via System ACE technology.

Conclusion

Xilinx has encountered and solved almost every presentable difficulty at every stage of the design cycle. Most of the issues and solutions have been documented in a database. In a critical design situation, this knowledge can be the difference between success and failure.

Xilinx will give you personalized access to one of our application engineers for the duration of your choosing. This Titanium Dedicated Engineer will be an expert in the area of your design needs and can provide an understanding of Xilinx design methodologies and techniques. The engineer will provide a solution to your problem, at any stage of the design cycle, and also educate your design team on the steps needed to prevent those issues from recurring.

A Titanium Dedicated Engineer can work at Xilinx, on-site with your design team, or a mix of both. This flexibility allows our engineers to fully understand the needs and requirements of our clients, as well as leverage Xilinx factory resources to resolve problems and accelerate production. For more information about Titanium Dedicated Engineering, call (800) 888-FPGA (3742) or visit *www.xilinx.com/titanium.*

X-ray vision for your designs

Agilent 16900 Series logic analysis system with FPGA dynamic probe

- *Increased visibility with FPGA dynamic probe*
- *Intuitive Windows® XP Pro user interface*
- *Accurate and reliable probing with soft touch connectorless probes*
- *16900 Series logic analysis system prices starting at $21,000*

Agilent Direct

Get a quick quote and/or FREE CD-ROM with video demos showing how you can reduce your development time.

U.S. 1-800-829-4444, Ad# 7909
Canada 1-877-894-4414, Ad# 7910
www.agilent.com/find/new16900
www.agilent.com/find/new16903quickquote

Now you can see inside your FPGA designs in a way that will save days of development time.

The FPGA dynamic probe, when combined with an Agilent 16900 Series logic analysis system, allows you to access different groups of signals to debug inside your FPGA—without requiring design changes. You'll increase visibility into internal FPGA activity by gaining access up to 64 internal signals with each debug pin.

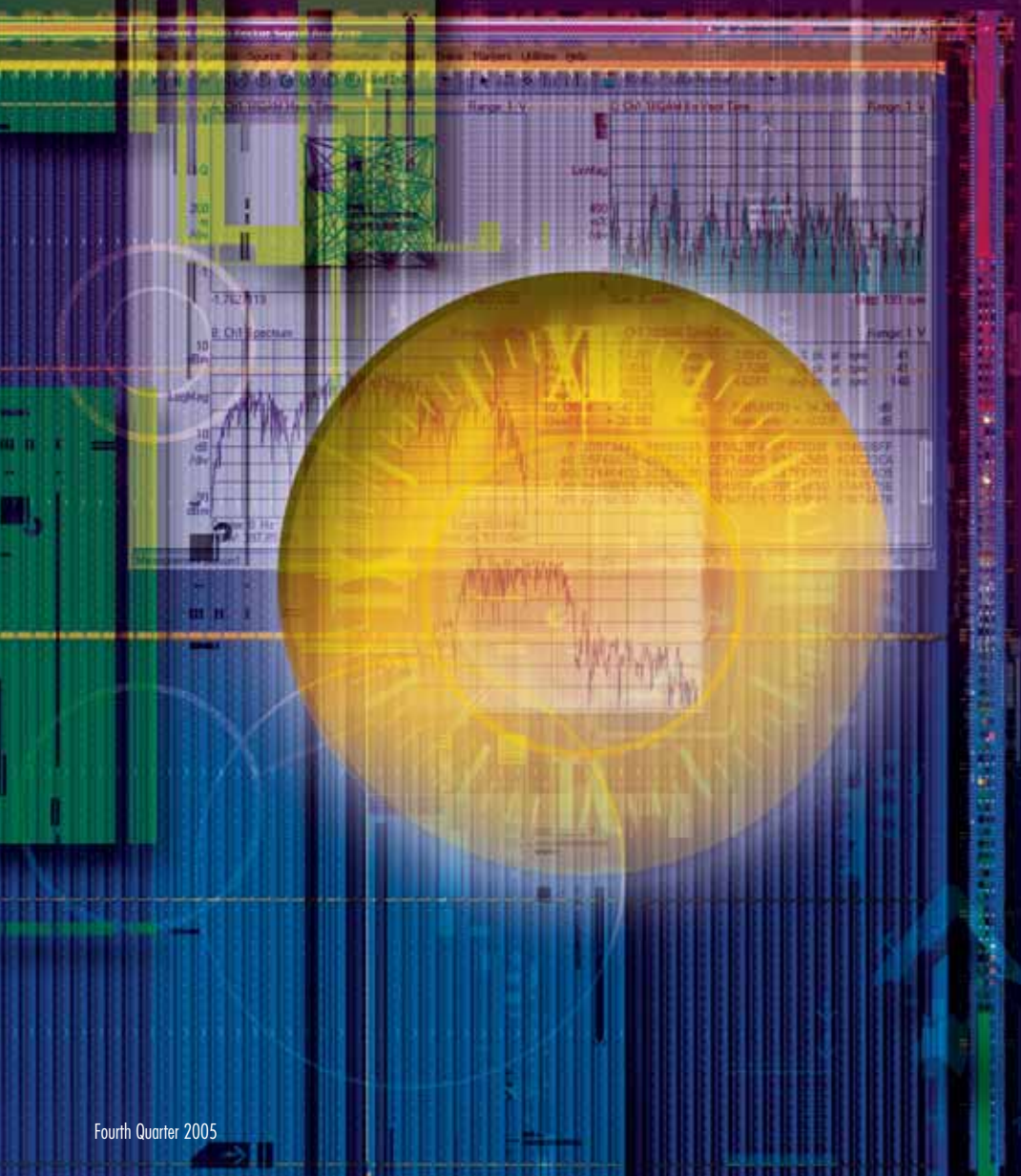
You'll also be able to speed up system analysis with the 16900's hosted power mode—which enables you and your team to remotely access and operate the 16900 over the network from your fastest PCs.

The intuitive user interface makes the 16900 easy to get up and running. The touch-screen or mouse makes it simple to use, with prices to fit your budget. Optional soft touch connectorless probing solutions provide unprecedented reliability, convenience and the smallest probing footprint available. Contact Agilent Direct today to learn more.

Agilent Technologies
dreams made real

Real-Time Analysis of DSP Designs

Agilent combines the FPGA Dynamic Probe and digital VSA.



by Scott Ferguson
Factory Application Engineer, Logic Analyzers
Agilent Technologies, Inc.
sferguson@agilent.com

As FPGAs become a viable option for high-performance signal processing in the digital communications design space (cellular base stations, satellite communications, and radar), analysis and debug tools must include new techniques to help you get the most optimal performance in your circuits in the least amount of time.

Although signal analysis tools that connect to simulation and RF analog signals are available, it's important to be able to measure signal quality (frequency spectrum, I-Q constellation, and error vector magnitude [EVM]) in the sub-circuits of your FPGA. Thus, Agilent has linked its 89601A Vector Signal Analysis (VSA) software with its line of logic analyzer products (1680, 1690, and 16900 families) to create a digital VSA tool. This tool, when combined with the Xilinx® ChipScope™ Pro Agilent Trace Core, allows you to perform signal analysis anywhere inside your FPGA design quickly and easily.

In this article, we'll show how this combination of tools works – and how it can help you get the most from your Xilinx-based DSP circuits.

Digital VSA

VSA uses Fast Fourier Transform (FFT)-based data processing to provide a combination of time- and frequency-domain displays and measurements. Figure 1 shows a typical VSA display. Although the display is extremely flexible and configurable, the main components include the I-Q constellation plot (upper left), magnitude spectrum (lower left), error vector (upper right), and measurements (lower right). The EVM is displayed in the measurements section. This single value is a key indicator of the quality of the modulated signal.

EVM is computed by extracting I-Q symbols from the captured data; the symbols are the grid points in the constellation defined by the QPSK, QAM, or other modulation scheme. Once extracted from the measured signal, the symbol sequence is used to create an ideal (theoretically perfect) signal known as the "reference" signal. Each measured signal is compared to the reference signal, and the difference is known as an error vector. (The error can contain both I and Q, or magnitude and phase components). The individual error vectors for a single capture are combined to make a single EVM measurement.

Although this analysis software was originally created to analyze analog RF signals, it was developed in a hardware-independent, PC-based software package. Because Agilent logic analyzers are also PC-based, it was easy to extend the VSA software to link to the logic analyzers.

Digital baseband and IF signals are representations of analog signals. Rather than using an instrument that digitizes a signal to enable FFT analysis (like an RF signal analyzer), the signal is digital from the start. These digital versions of analog signals can be displayed in a logic analyzer in a chart-style waveform, which resembles an oscilloscope display (as in Figure 2).

As you can see, when the bus is synchronously sampled and the sample rate meets the Nyquist requirements, the logic analyzer captures a sufficiently accurate version of the "once-was" or "will-soon-be" analog signal.

FPGA Dynamic Probe

The FPGA Dynamic Probe, working with the ChipScope Pro analyzer, can provide access to any part of a DSP design without recompiling. In Figure 3, a simplified digital radio transmitter design is connected to the Agilent Trace Core 2 (ATC2). This core is a switching MUX incorporated into the design using the ChipScope Pro Core Inserter, typically post-synthesis. During core insertion, you select the internal nets to connect to the trace core, and the physical pads to which you will connect the MUX output. These pads are then routed on the circuit board to a logic analyzer probe.

The logic analyzer controls the FPGA through JTAG (downloading the bit file and selecting banks). When you select a new bank, the logic analyzer automatically reconfigures itself to match the names of the nets now connected to the probe.

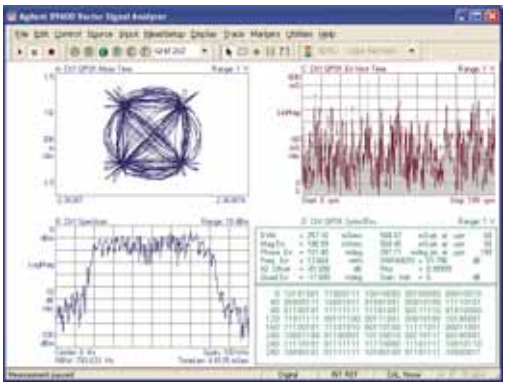


Figure 1 – VSA display

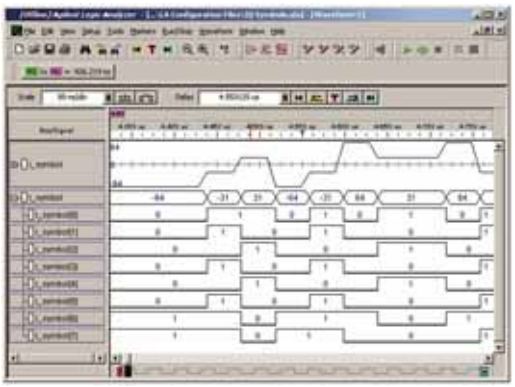


Figure 2 – Chart display of digital bus

Design Example – QAM16 Modulator

With help from our local Xilinx DSP specialist FAE, we created a demo that fits into a small Virtex™-II part (XC2V250-FG256) using Xilinx System Generator for DSP. This tool makes creating DSP designs quick and easy. The design (shown in the block diagram in Figure 3) contains a 25 MHz symbol encoder; a root-raised cosine filter with 24 taps and 4X interpolation (the output running at 100 MHz); and an IF modulation stage with a 25 MHz local oscillator.

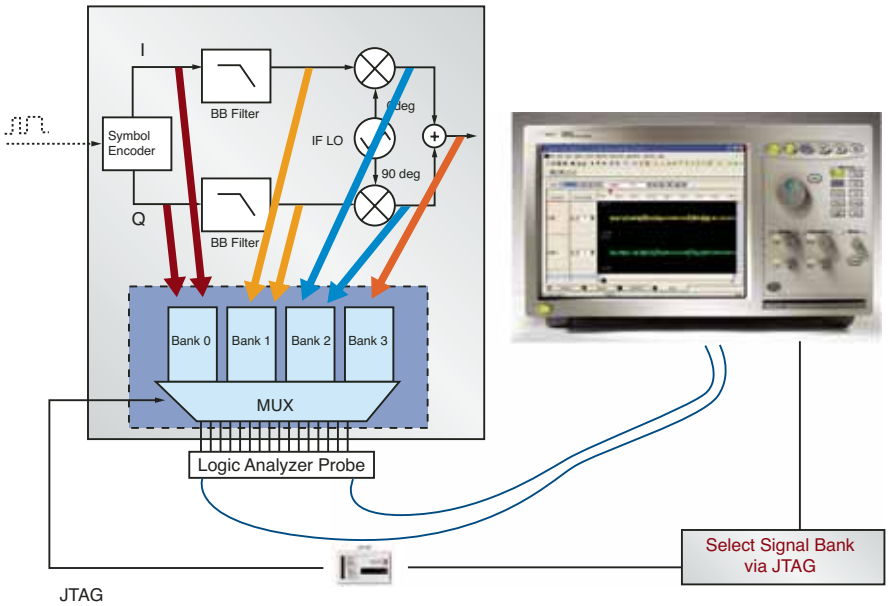


Figure 3 – FPGA Dynamic Probe

Integrating the ATC2 Core in a System Generator Design

After compiling this design into VHDL, we inserted the ATC2 core. To make the signal names more logical on the logic analyzer display, we did some hand-editing of the VHDL. (You could avoid this step by carefully choosing net names in the System Generator.) We then connected most of the interesting nets as output ports from the top-level object to make the net names short enough to fit on the logic analyzer screen.

When connecting nets to output ports solely for use with the FPGA Dynamic Probe, a good trick is to use the "keep" attribute in the VHDL. Because you don't add the ATC2 core to the design until after synthesis, many nets would otherwise be optimized out because they're not connected to anything. In VHDL, the syntax to use the "keep" attribute looks like this:

```
attribute keep : string;

attribute keep of i_symbol:
signal is "true";

attribute keep of q_symbol:
signal is "true";
```

We created an ATC2 core with four banks, each with 48 signals. Using the ATC2 core's 2X TDM option (time-slicing two signals at a time on each pad), this requires only 25 package pads on the FPGA (one for a clock and 24 for data). This gives us access to 192 signals. Actually, we only need to view about 92 signals:

- I-Q symbols, 8 bits each (16)
- I-Q filter output, 24 bits each (48)
- IF local oscillator sine and cosine, 2 bits each (4)
- Combined IF signal (24)

The output of the RRC filter with 24-bit I and Q signals was the largest requirement, defining the number of pins required. If 24 pins were not available, you could drop the least significant bits, losing some dynamic range but still being able to view the signals.

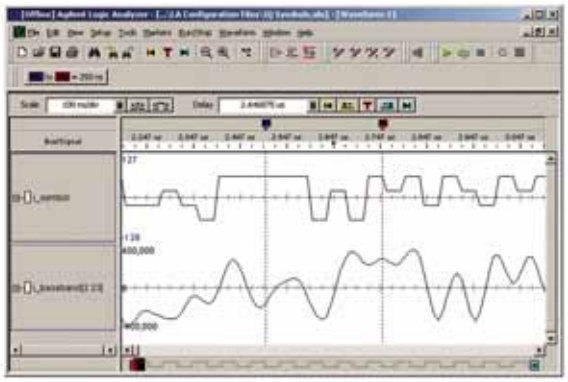


Figure 4 – Filter group delay measurement

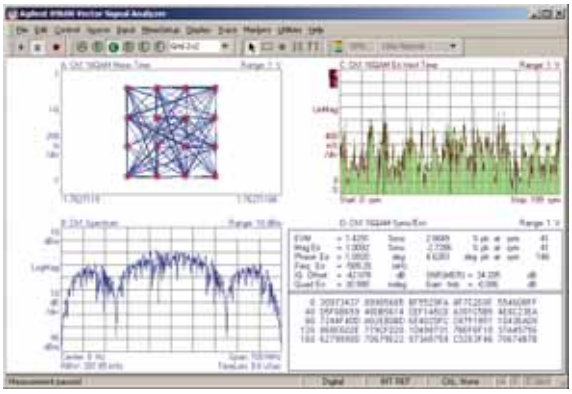


Figure 5 – Unfiltered QAM16 symbols

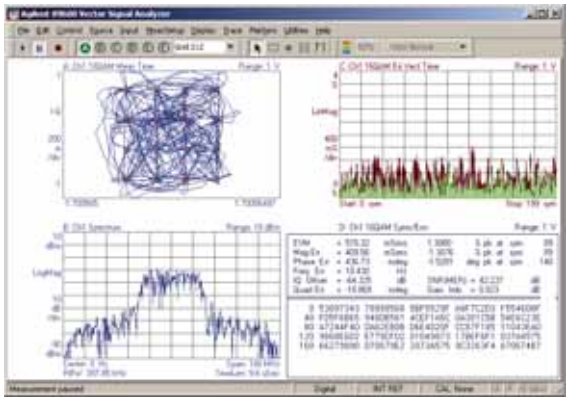


Figure 6 – Filtered IQ baseband data

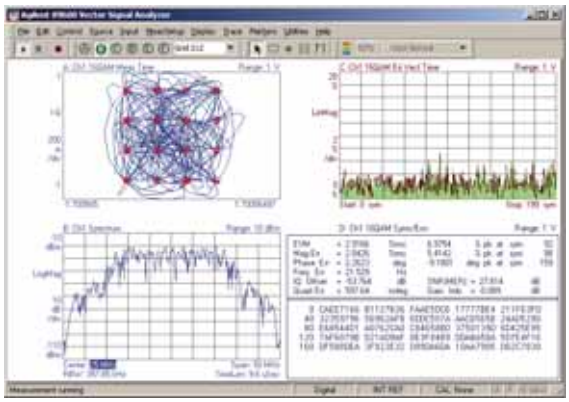


Figure 7 – Digital IF signal

Time-Domain, Logic, and VSA Measurements

The logic analyzer uses synchronous sampling (or "state mode") to capture the output of the ATC2 core. This means that data is sampled on each edge of the ATC2's output clock. Our design has two clock rates in the circuit – 25 MHz for the symbol data before the RRC filter and 100 MHz for all parts after the filter. Because the ATC2 core supports only one clock per core, two options exist for debug:

- Using two cores, one for each clock rate
- Using one core with the faster clock rate and over-sampling the 25 MHz bus

Because the two clocks are correlated – and one is an integer multiple of the other – you can just over-sample the slower bus. If over-sampling is not desirable, the logic analyzer can use a setup that stores every fourth sample, thereby capturing the 25 MHz bus accurately with one sample per 25 MHz clock.

With the extra signals available in the MUX, we were able to double-probe some of the interesting signals. For example, in bank 0 we have the I and Q symbols before the filter, and also the I component after the RRC filter. This means we can do some time-domain analysis in the logic analyzer to measure group delay in the filter, as in Figure 4. Two markers indicate a common signal feature: a wide, flat top and the marker measurement display showing an interval of 250 ns.

After probing the interesting parts of the circuit, we performed vector signal analysis on the signals and measured the quality of our RRC filter and IF modulation stages.

Looking at the QAM16 I-Q symbols before they were filtered (as shown in Figure 5), you can see the 16-point QAM constellation (upper left graph).

With one point per symbol, the lines between the constellation points are straight. The frequency spectrum (in the lower left graph) is centered at 0 Hz and has a 25 MHz pass-band with power in adjacent channels. Adjacent channel power is undesirable in the RF signal, of course, which is the reason for the baseband filter.

By selecting a different bank in the ATC2 core (controlled by the logic analyzer), you can perform analysis on the IQ signal after the baseband filter, as seen in Figure 6. Now the spectrum has sidebands removed, and the measurement display (in the lower right quadrant) shows an EVM of 0.5%. The next time your RF team complains of errors in the baseband design, you can point to this measurement (with which they are quite familiar), and prove that it's not your filter's fault.

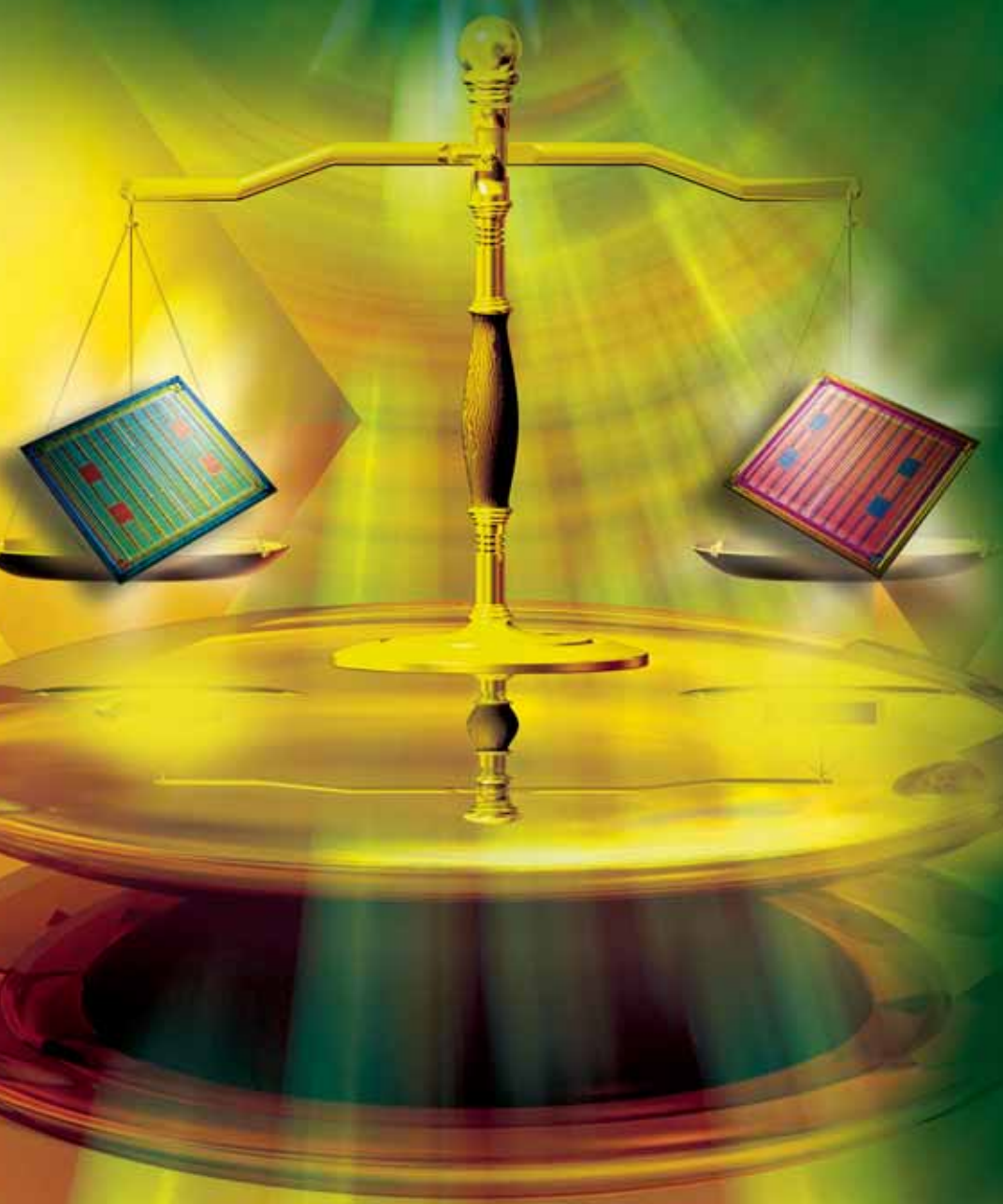
In many digital radio designs, this IQ signal would now be converted to analog. However, we performed the IF modulation digitally inside the same FPGA. Switching banks in the FPGA Dynamic Probe gives us access to the digital IF (again, without another synthesis and place and route step), as shown in Figure 7. Note that the spectrum and I-Q constellation are roughly the same, only now centered about 25 MHz. The EVM is a little bit higher, indicating that you may want to use a higher quality local oscillator or another filter stage.

Conclusion

Xilinx System Generator and ChipScope Pro analyzer, combined with the Agilent logic analyzer and Agilent VSA software, allow you to perform real-time in-depth analysis on digital baseband and IF signals inside your Xilinx FPGA. This will save you time and eliminate doubts about the difference between simulation and real hardware. It can also help you communicate with your colleagues on the RF design team, enabling you to speak their language and use the same analysis software regardless of signal format (analog, digital, baseband, or RF).

For more information about these applications, visit *www.agilent.com/find/logic-sw-apps*, or contact your Agilent representative.

Configuration Choices — Platform Flash or Commodity Flash



Xilinx provides flexibility for configuration memory so that you can make the best decision for your design.

by Anthony Le
Product Marketing Manager,
Configuration Memory Solutions
Xilinx, Inc.
anthony.le@xilinx.com

FPGA configuration is often a last-minute design decision, because engineers view FPGA configuration as an easy, no-brainer step in the design cycle. That is true when customers use the Xilinx "recipe" – a Xilinx® FPGA, Platform Flash PROM, ISE™ software, and platform cable USB. However, FPGA configuration becomes increasingly complex if you use a non-Xilinx solution. In this article, I'll discuss the differences between Platform Flash and commodity Flash (see Table 1 for a summary).

Added Flexibility in Configuration Solutions

Before the introduction of the Spartan™-3E FPGA family, customers who configured Xilinx FPGAs with commodity Flash would use a three-chip solution: an FPGA, a commodity Flash PROM, and a CPLD. Spartan-3E FPGAs eliminate the need for a CPLD (or other controller) by providing a direct interface for leading commodity Flash devices, thus reducing the chip count to two (FPGA and PROM).

Because Xilinx gives you complete flexibility to use multiple memory sources for FPGA configuration, you should consider the following factors at the start of the design process: total cost of ownership, board space, configuration speed, source of supply, value-added features, and ease of use. If these factors are not thoroughly thought out from the beginning, then the final design can incur additional costs and possible board redesign.

Total Cost of Ownership

On a per-unit basis, commodity Flash might appear to be attractively priced; however, you need to consider the total cost of ownership. Total cost of ownership is the summation of per-unit cost, design and prototyping cost, and manufacturing and test cost.

1. The cost difference between a commodity Flash PROM versus a Platform Flash PROM is negligible when compared to the overall board cost. In fact, Xilinx Platform Flash PROMs are competitively priced with all non-volatile memories in the market (commodity Flash PROM and competing PROMs).
2. The prototyping phase of a design significantly favors Platform Flash over commodity Flash because Xilinx offers one of the lowest cost in-system programming (ISP) solutions:
   - Platform cable USB: $150
   - Programming software – iMPACT: $0
   - Xilinx award-winning support: $0 (included)
3. Once in production, you can significantly reduce costs by utilizing the Boundary Scan (JTAG) capability of Xilinx FPGAs and Platform Flash PROMs (along with other JTAG devices on the board) for low-cost Boundary Scan testing and programming. Commodity Flash devices do not offer JTAG interfaces; therefore, customers cannot take advantage of the low cost of Boundary Scan testing. In most cases, expensive Automatic Test Equipment (ATE) is required to test and perform in-system programming of commodity Flash memories.

Board Space

If board space is critical to your design, then consider the following:

- Standard SPI PROMs are typically offered in the smallest form factor. The 1 Mb to 4 Mb SPI PROMs are usually offered in a SOIC-8L (5 x 6 mm) package and 8 Mb (and larger) devices are usually offered in a SOIC-16L (10 x 6 mm) package.
- Platform Flash PROMs are a close second with 1, 2, and 4 Mb PROMs offered in a TSOP-20L (6.5 x 6.4 mm) package and 8, 16, and 32 Mb PROMs in a TFBGA-48 (8 x 9 mm) package.
- Parallel commodity Flash devices have large packages to address the additional control, address, and I/O pins.

SPI PROM has the advantage here, but Platform Flash is a close second considering that there is only a 12 mm^2 difference in area. The difference in area is minute compared to the overall board space.

Configuration Speed

Parallel commodity Flash devices are typically the fastest memory on the market. They are offered in either x8 or x16 I/O configurations. The theoretical data transfer rate can be as fast as 50 MHz x 16 I/O, but there are limitations when configuring a Xilinx FPGA.

1. At this time, Xilinx FPGAs can only be configured in x8 mode.
2. Before Spartan-3E devices, configuring a Xilinx FPGA with a commodity Flash PROM would require using a CPLD device to translate memory into the FPGA bitstream (refer to XAPP058, "Xilinx In-System Programming Using an Embedded Microcontroller"). Resulting data transfer rates can degrade based on the translation logic.
3. If using a Spartan-3E device with a parallel commodity Flash, the configuration mode is limited to 6 MHz. Platform Flash features a maximum transfer rate of 40 MHz x 8 I/O (or 320 bps) with Spartan-3E devices.

In theory, Parallel commodity Flash is faster, but given the limitations listed above, the practical transfer rate is significantly less than that of Platform Flash.

Source Supply

Although there are many commodity Flash vendors, you should be aware of two potential pitfalls. First, every vendor provides similar commodity Flash PROMs, but there are nuances with each vendor that can limit their interoperability. For

	Platform Flash	SPI Flash	Parallel Flash
Minimum Config Pins	**3**	4	37
Configuration	**Serial & Parallel***	Serial Only	Parallel Only
Storage Density	1 – 32 Mb	512K – 128 Mb	512K – 256 Mb
Sourcing	Xilinx only	**Multiple**	**Multiple**
Supply Management	**Yes**	No	No
Storage Cost	Low	**Lowest**	Low
FPGA Read/Write	Yes**	Yes	Yes
JTAG Interface	**Yes**	No	No
Configuration Time	Fast	Slow	**Fastest**
Compression	Yes***	No	No
Design Revisioning	Yes***	No	MultiBoot
Power Reliability	**Excellent**	Fair	Fair
Xilinx Support	**Excellent**	Limited	Very Limited
ISE 7.1i Support	**Excellent**	None	None

* XCF00P supports both Serial & Parallel, XCF00S only suports Serial
** Read-Only using XAPP694
*** Only with XCF08P, XCF16P, and XCF32P

Table 1 – Memory summary chart

example, a STMicro SPI is not fully compatible with an Atmel SPI PROM. Second, during a period of tight source supply, customers might find themselves paying more for expedites or end up with very long lead times. Xilinx answers the source supply conundrum by holding a large inventory of Platform Flash at finished goods, which allows Xilinx to quickly react to increased demand.

Ease of Use

Platform Flash was designed to work seamlessly with all Xilinx FPGAs and is also supported by an award-winning support team. Xilinx provides a total configuration solution that includes software and hardware. No other configuration memory solution offers this type of support.

Value-Added Features

Finally, Platform Flash offers the following value-added features that are not found in commodity Flash:

1. Compression. The higher density Platform Flash PROM devices have built-in de-compressors, which, on average, can fit 50% more configuration data into the same memory space. Xilinx patented compression technology can help you reduce costs in two ways:

   a. Reduce component costs by fitting a large bitstream into a lower density Platform Flash PROM device. For example, a Virtex™-4 LX60 design requiring more than 17 Mb of configuration bits can fit into a XCF16P instead of a 32 Mb PROM.

   b. Reduce component count by fitting a design into one PROM as opposed to two or more. For example, a Virtex-4 LX160 device requires more than 40 Mb of configuration bits, which normally would require a 32 Mb and 8 Mb PROM, but compression enables the design to fit into a single XCF32P.

2. JTAG. Allows low-cost board-level Boundary Scan testing for opens and shorts, as well as programmability during prototyping and in the production environment.

3. Design Revisioning. Allows one board to have many functions. Platform Flash PROMs (XCF08P, XCF16P, and XCF32P) have blocks of memories that can be written and read independently of one another. The logic to switch between each block is already built into Platform Flash, thus reducing design time and cost. Although commodity Flash devices have a similar feature called "sectors," you would need glue logic and software to access the various sectors.

4. Access to unused memory. Most FPGA bitstreams will not use all of the memory of a PROM. Thus, any unused memory can be used for processor "scratch pad" or "boot code." You can access unused memory within a Platform Flash PROM through JTAG (refer to XAPP544, "Using Xilinx XCF02S/XCF04S JTAG PROMs for Data Storage Applications"). Note that you can still access unused memory within commodity Flash PROM, but you might need to design additional logic and software to access the unused memory.

Conclusion

When planning your next board design, use Platform Flash for your FPGA configuration and you can beat your competitors to market and lower development cost. Platform Flash is an innovative configuration memory with value-added features that enable greater flexibility and performance for Virtex and Spartan FPGAs.

Platform Flash PROMs provide you with a system-level drop-in solution that allows you to maximize the flexibility of Virtex and Spartan FPGA-based systems to significantly reduce your design effort and accelerate time to market. Platform Flash PROMs are competitively priced, reduce the amount of board space required for configuration, and offer a complete 1 to 32 Mb PROM density solution (Table 2).

For more information, visit *www.xilinx.com/products/silicon_solutions/proms/pfp/index.htm.*

	XCF01S	XCF02S	XCF04S	XCF08P	XCF16P	XCF32P
Density	1 Mb	2 Mb	4 Mb	8 Mb	16 Mb	32 Mb
JTAG Prog	✔	✔	✔	✔	✔	✔
Serial Config	✔	✔	✔	✔	✔	✔
SelectMap Config				✔	✔	✔
Compression				✔	✔	✔
Design Revisions				✔	✔	✔
VCC (V)	3.3	3.3	3.3	1.8	1.8	1.8
VCCO (V)	1.8 – 3.3	1.8 – 3.3	1.8 – 3.3	1.5 – 3.3	1.5 – 3.3	1.5 – 3.3
VCCJ (V)	2.5 – 3.3	2.5 – 3.3	2.5 – 3.3	2.5 – 3.3	2.5 – 3.3	2.5 – 3.3
Clock (MHz)	33	33	33	40	40	40
Packages	VO20	VO20	VO20	FS48 VO48	FS48 VO48	FS48 VO48
Pb-Free Pkg	VOG20	VOG20	VOG20	FSG48 VOG48	FSG48 VOG48	FSG48 VOG48
Availability	Now	Now	Now	Now	Now	Now

Table 2 – Platform Flash features

Accelerating PowerPC Software Applications

Using custom APU peripherals, C-to-hardware tools enable fast creation of Virtex-4 hardware accelerators.

David Pellerin
Chief Technology Officer
Impulse Accelerated Technologies, Inc.
david.pellerin@impulsec.com

Greg Edvenson
Senior Software Engineer
Pico Computing, Inc.
greg@picocomputing.com

Kunal Shenoy
Design Engineer
Xilinx, Inc.
kunal.shenoy@xilinx.com

Dan Isaacs
Director, Embedded PowerPC Marketing, APD
Xilinx, Inc.
dan.isaacs@xilinx.com

The Xilinx® Virtex™-4 FX family of FPGA devices provides embedded systems developers with new alternatives for creating high-performance, hardware-accelerated applications. With an integrated industry-standard PowerPC™ processor and innovative Auxiliary Processor Unit (APU) interface, the Virtex-4 FX device allows system designers to efficiently connect custom hardware accelerators to the integrated processor, yielding unprecedented performance.

In the past, software programmers who wanted to take advantage of FPGAs for algorithm acceleration have experienced significant technical barriers because of the complexity of writing low-level hardware descriptions to represent higher level software functions.

In this article, we'll show how the power of the Virtex-4 FX FPGA can be made readily available to embedded systems designers and software programmers through the use of software-to-hardware tools. The emergence of such tools bring the performance benefits of FPGAs to anyone who can program in C. Accelerated FPGA-based designs are now easier and more practical for a wide range of application domains, including image processing, DSP, and data encryption.

From Software to FPGA Hardware

By virtue of their massively parallel structures, FPGAs have the potential to dramatically accelerate embedded software applications. But because these devices require different (hardware-oriented) skills than traditional processors, the creation and programming of a system based on an FPGA has remained challenging for all but the most hardware-savvy software programmers. This is changing, however. With the introduction of simplified FPGA-based computing platforms and streamlined tools for platform building, most of the barriers to FPGA adoption have been removed. In addition, the introduction of software-to-hardware tools for FPGAs has dramatically improved the practicality of these devices as software-programmable computing platforms.

The tools that make this shift possible – enabling FPGA-based platforms to be considered viable alternatives to traditional processors for embedded systems – serve two basic needs. At the front end, software-to-hardware compiler tools accept high-level descriptions written in a language familiar to embedded software programmers. The de facto standard for embedded systems design is standard C, with C++ and Java beginning to make inroads as well.

At the back end, existing synthesis and place and route technologies are combined with system-level platform building tools, allowing designers to develop and target complete systems on programmable logic to specific development boards. Both of these needs are being met today, by tools currently available.

In the area of software-to-hardware compilation, compiler tools such as Impulse CoDeveloper (Figure 1) can simplify the generation of FPGA hardware from higher level C-language descriptions of software algorithms. These tools provide the necessary bridge between the domains of software programming and lower level hardware design.

Serving the need for platform building tools is Xilinx Platform Studio, which supports a wide variety of Xilinx FPGA-based boards and systems. Platform Studio makes it practical for an embedded systems designer – who may have little or no experience with low-level FPGA design – to assemble a complete system within a single FPGA, including one or more processors and related peripherals. When these tools are combined with a platform-aware software-to-hardware compiler, the complete system can include custom accelerators originally written in C.

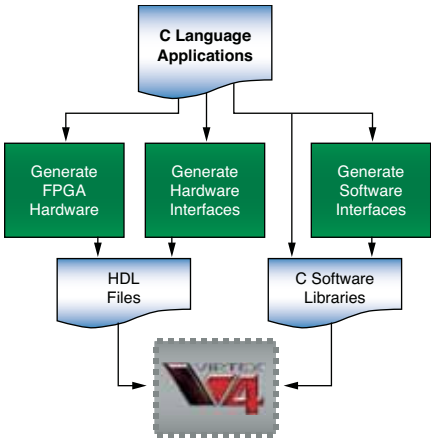


Figure 1 – Impulse CoDeveloper tools simplify the conversion of C subroutines to lower level FPGA logic and provide the necessary software-to-hardware communications on the Virtex-4 platform.

Accelerating Embedded Applications

The Virtex-4 FX family of devices provides an ideal platform for hardware acceleration of embedded applications. The Virtex-4 FX12 FPGA, for example, includes more than 12,000 logic cells; an integrated PowerPC 405 core, which can operate at speeds as fast as 450 MHz; and dual 10/100/1000 Ethernet MACs, configured by the processor through the device control register (DCR) interface or through the FPGA fabric.

Looking inside the embedded processor block (see Figure 2), the PowerPC 405 CPU is directly coupled to the unique and innovative APU controller, which provides direct access to hardware accelerators implemented in the FPGA logic. The APU controller supports three classes of instructions: PowerPC floating-point instructions, APU load and store instructions, and user-defined instructions (UDI). UDIs are pro-

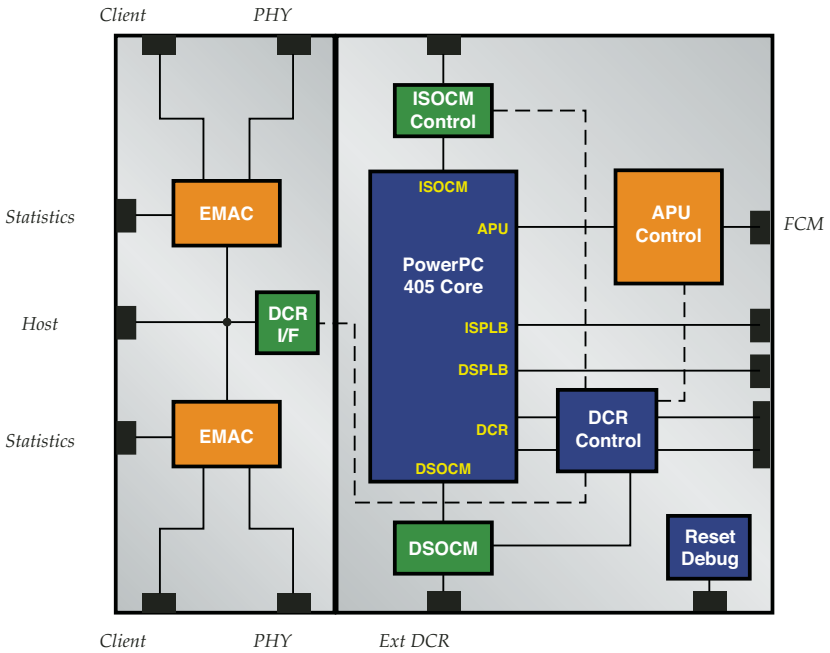


Figure 2 – The Virtex-4 FX12 embedded processing block includes the embedded PowerPC 405 processor and a high-performance APU interface, along with dual 10/100/1000 EMACs.

grammed into the APU controller either dynamically through the PowerPC 405 via the DCR bus or statically during FPGA configuration via the bitstream. The APU supported instructions are executed by hardware acceleration co-processing engines implemented in the FPGA logic.

When packaged in a highly integrated compact device such as the Pico Computing E-12 card (see sidebar, "A Wide Range of Development Platforms"), the FX12 device becomes a complete embedded development platform that requires little or no hardware design expertise. For embedded application developers requiring a wider range of hardware peripherals (such as direct access to video and audio signals), the Xilinx ML403 board, also based on the FX12 device, provides an excellent embedded systems development platform.

On-chip, the Virtex-4 FX APU controller provides a flexible high-bandwidth interface between the FPGA fabric and the

Taking Advantage of Parallelism in FPGAs

A key aspect of any software-to-hardware design flow is the use of parallelism to increase performance. When accelerating C applications using FPGAs, parallelism can be exploited at two distinct levels: at the application system level and at the level of statements (or blocks of statements) within a specific subroutine or loop.

Although there are ongoing attempts to create compiler technologies that can exploit both levels of parallelism with a high degree of automation, the best approach today is to focus automation efforts (represented by the software-to-hardware compiler) on the lower level aspects of the problem, while at the same time providing software programmers an appropriate and easy-to-use programming model that allows higher level, coarse-grained parallelism to be expressed. In this way programmers can make hardware/software partitioning decisions and experiment with alternative algorithmic approaches, leaving the task of low-level optimization to automated compiler tools. This approach is particularly useful for platforms such as the Virtex-4 device that include embedded processors.

A number of programming models can be applied to FPGA-based programmable platforms, but the most successful of these models share a common attribute: they support modularity and parallelism through a dataflow-like method of design partitioning and abstraction. Communicating sequential processes, or CSP, is one such programming model. CSP has proven to be highly effective in expressing application-level parallelism for FPGA targets. This programming model is directly supported in the Impulse C tools provided by Impulse Accelerated Technologies, Inc.

At the heart of the Impulse C programming model are processes and streams (Figure 4). Processes are independently synchronized, concurrently operating portions of an application that are written in a standard language (in this case C language). Processes perform the work of the application by accepting data, performing computations, and generating relevant outputs.

Unlike traditional C subroutines, processes are considered persistent; they are normally called once (whether in hardware or software) and continue as long as there is streaming data to be processed. The data processed by such an application flows from process to process by means of streams, or in some cases by means of messages or shared memories, which are also supported in the programming model. Streams represent one-way channels of communication between concurrent processes and are self-synchronizing with respect to the processes by virtue of buffering. The primary method of synchronization between processes is therefore the data being passed on the streams.

The key to allocating processing power within such a system – and using such a programming model – is to implement one or more processes in the FPGA to handle the heavy computation, and implement other processes on embedded or external microprocessors to handle file I/O, memory management, system setup, and other non-performance-critical tasks. Using tools such as those included with Impulse C, an application comprising multiple parallel C processes can be modeled entirely in software, verified using a standard desktop C debugging environment, and then, after the application is functionally complete, incrementally moved into the FPGA for further optimization and acceleration.

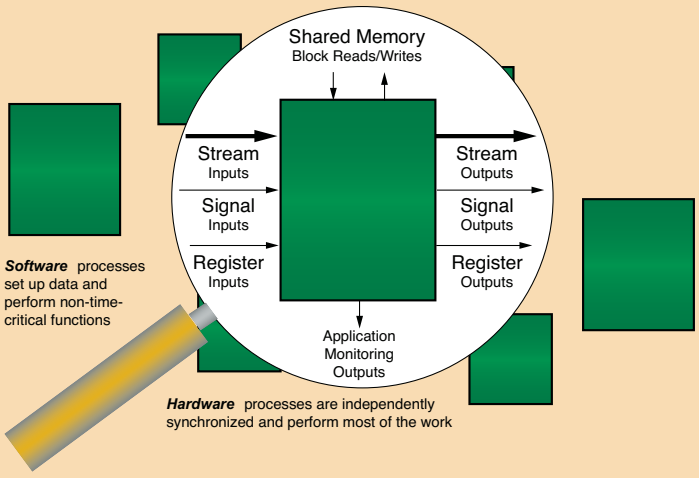


Figure 4 – The Impulse C programming model emphasizes the use of processes, streams, and shared memories for hardware/software partitioning.

pipeline of the on-chip PowerPC. Fabric co-processor modules (FCMs) implemented in the FPGA fabric are connected to the embedded PowerPC processor through the APU interface, allowing the creation of custom hardware accelerators. These hardware accelerators operate as extensions to the PowerPC, thereby offloading the CPU from demanding computational tasks.

Software engineers can access the FCM from within assembler or C code. Assembler mnemonics are available for user-defined instructions and pre-defined load/store instructions, enabling programmers to invoke hardware-accelerated functions into the regular program flow. Programmers can also define custom instructions designed specifically for the hardware functionality of the FCM. When combined with C-to-hardware compiler tools, the APU controller allows software programmers to create hardware-accelerated software applications with little or no FPGA design expertise.

C-to-Hardware Tools Increase Design Productivity

To make productive use of any computing platform, software programmers need appropriate compiler and debugging tools. Impulse C, from Impulse Accelerated Technologies, gives software programmers access to FPGAs by allowing hardware accelerators to be compiled directly from software descriptions.

These accelerators, which are typically represented by one or more software subroutines, are automatically compiled into efficient, high-performance hardware that can be mapped directly into FPGA gates. In the case of the Virtex-4 FPGA, Impulse C is also capable of automatically generating software/hardware interfaces using the APU. This is particularly useful for applications that combine both traditional and FPGA-based processing.

Because it is based on standard C, Impulse C allows FPGA algorithms to be developed and debugged using popular C and C++ development environments, including Microsoft Visual Studio and GCC-based tools. The CoDeveloper software-to-hardware compiler translates specific C-language subroutines to low-level FPGA-hardware (see Figure 3) while optimizing the generated logic and identifying opportunities for parallelism. The compiler is also capable of unrolling loops and generating loop pipelines to exploit the extreme levels of parallelism possible in an FPGA. Instrumentation and monitoring functions generate debugging visualizations for highly parallel multi-process applications, helping system designers identify dataflow bottlenecks and other areas for acceleration.

For applications involving the embedded PowerPC and MicroBlaze™ processors, the Impulse C compiler automates the creation of hardware/software interfaces and generates outputs compatible with Xilinx Platform Studio. This makes it possible to create high-performance, mixed hardware/software applications for FPGA-based platforms without the need to write low-level VHDL or Verilog.

For large applications comprising multiple hardware and software elements, Impulse C includes interface libraries (see sidebar, "Taking Advantage of Parallelism in FPGAs") and related compiler features, allowing parallelism to be expressed at the level of multiple and independently synchronized processes. These processes can be mapped either to software running on an embedded PowerPC or MicroBlaze processor, or to FPGA hardware.

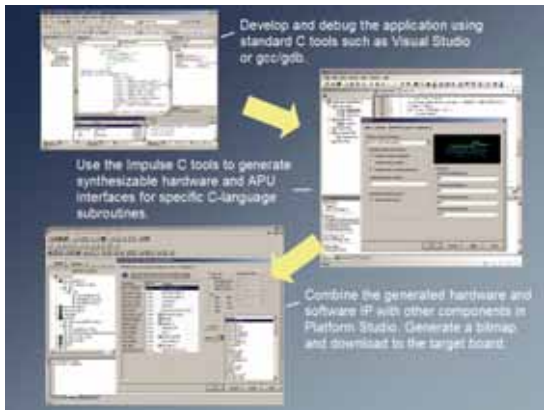


Figure 3 – The C-language-to-FPGA-accelerator design flow

For all such applications, integration of front-end compiler tools and back-end platform building tools is important. The design process is highly iterative, reflecting the fact that decisions made up-front (such as C coding styles and system-level partitioning decisions) may have a dramatic impact on the results obtained after C compilation, synthesis, place and route, and final bitmap generation. At each point in the process, the tools provide feedback, allowing you to evaluate and estimate performance before moving to subsequent (and perhaps more time-consuming) phases of the platform generation process.

Let's summarize the steps required for a typical PowerPC-based application using Impulse and Xilinx tools:

1. The application is initially written in standard C, using common C development tools. These tools include readily available tools such as Visual Studio, Eclipse, or GCC and GDB, and may also involve more comprehensive cross-development tools. During this phase, a baseline for validation (a software test bench, also written in C) is established, allowing you to quickly test later design iterations.
2. A C profiler such as gprof may be invoked, or other, less sophisticated methods are used to identify computational hotspots. Often these hotspots can be isolated to a few C subroutines or inner code loops requiring acceleration. Application monitoring (made possible by instrumenting the C code during software testing) can help characterize these hotspots and analyze data movement.
3. Using software-to-hardware interface functions provided in the Impulse C library, data streams or shared memories create abstract connections between the main algorithm running on the PowerPC and hardware-accelerated subroutines running in the FPGA. The modified software algorithm, which now includes one or more independently synchronized processes, is simulated again in a standard C environment to ensure its correct behavior.

4. C-language subroutines (or processes, as they may now be referred to) representing hardware accelerators are analyzed and optimized by the Impulse C compiler, resulting in hardware description files compatible with FPGA synthesis tools. Optimization reports generated in this phase help you understand the impact of various coding styles, and make appropriate revisions in the original C code for improved performance. During this compilation process, additional compiler outputs are generated that represent hardware-to-software interfaces, including (in the case of the Virtex-4 FPGA) the necessary APU interface logic. Software run-time libraries are also generated at this point, corresponding to the abstract stream and shared memory interfaces specified on the processor side of the application.

5. The generated hardware and software files are exported from the Impulse tools (as a PCORE peripheral) and imported directly into the Xilinx Platform Studio environment.

The stream and shared memory interfaces defined in the C application are mapped to APU, PLB, or other interfaces where appropriate, along with other components (such as standard processor peripherals or non-standard IP blocks) to create the complete system. From within the Platform Studio interface, the entire application (both hardware and software) is built, resulting in a downloadable bitmap.

Evaluating FPGA Acceleration

Using the Pico E-12 card and the Xilinx ML403 development kit, in conjunction with Impulse C and Platform Studio, we set out to compare the relative performance of the embedded PowerPC 405 processor both with and without APU hardware acceleration – and using only C programming techniques. To investigate a range of potential application domains, we selected the following three representative algorithms:

1. An image filter. This algorithm allowed us to evaluate two pipelined hardware routines for processing a stream of image data. The algorithm chosen for this experiment is a relatively simple 3 x 3 edge-detection function operating on a 512 x 512 image buffer. This algorithm allowed us to quickly evaluate the performance of data streaming through the Virtex-4 APU interface, as well as the potential speedups of using multiple, pipelined hardware processes.

2. A triple-DES encryption engine. This algorithm allowed us to evaluate the impact of various C-level optimization strategies, as well as the practicality of adapting and optimizing legacy C code for a streaming programming model. One million character blocks (of eight characters each) were processed to obtain performance numbers for this test.

3. A fractal image generator. This algorithm is computationally intensive and can be characterized in many ways to explore the size/performance space. For this experiment, we created a single hardware process in the FPGA as an APU peripheral. This hardware process communicates with a single controlling software process running on the embedded PowerPC. The design of this algorithm, which generates a 1024 x 768 pixel image with a selectable level of image accuracy, is scalable such that additional hardware accelerator processes can be easily added, up to the limit of the target FPGA.

For each of these algorithms, various combinations of compiler loop unrolling, pipelining, and maximum stage delays were selected in the Impulse C compiler. In this way the applications could be optimized (in most cases without modifying the original C code) to obtain a desirable balance of size, cycle delays, and maximum clock speed in the generated hardware. In most cases, we determined that using a relatively low clock rate (50 MHz) in the FPGA fabric – in combination with increased cycle-by-cycle throughput (through the use of automated pipelining) – produced the best overall results given the nominal overhead of software-to-hardware data communication.

Using these algorithms as a baseline, numerous tests were performed in which the same C code was compiled both to the FPGA (as an APU accelerator) and to the embedded PowerPC processor as a software-only application. The results of these tests are summarized in Table 1.

As the chart shows, the hardware-accelerated algorithms show an impressive increase in performance, even at reduced FPGA clock rates, compared to the PowerPC software-only version.

Conclusion

In this article, we have demonstrated how it is possible, using an FPGA-based platform and C-to-hardware tools, to create highly accelerated systems without low-level hardware design skills. The Virtex-4 FX device, when implemented in a card such as the Pico E-12 or on a prototyping board such as the Xilinx ML403, promises to revolutionize the way that FPGA devices are applied for high-performance embedded computing.

Software-to-hardware tools such as Impulse C, when combined with the platform building capabilities of Platform Studio, make programming for such devices practical and efficient.

Application	PowerPC Only (300 MHz)	PowerPC/APU (300/50 MHz)	Acceleration
Image Filter (512 x 512 Image)	0.1414 sec	0.0124 sec	11.4 X
Encryption (8M Characters)	2.257 sec	0.0667 sec	33.84 X
Fractal Image (10K Max Iterations)	660 sec	31 sec	21.29 X

Table 1 – Virtex-4 APU acceleration results

A Wide Range of Development Platforms

Providing embedded application developers – software programmers – with an easy-to-use hardware platform is a critical first step in making FPGAs viable as embedded development platforms. A growing number of vendors are offering FPGA-based prototyping and high-performance computing platforms ranging from low-cost, single-FPGA systems to larger FPGA grids intended for hardware-accelerated computing.

The Pico E-12 card mentioned in the main article (which is available from Pico Computing, *www.picocomputing.com*) is a CompactFlash form-factor package that draws well under 2W. It features 10/100/1000 Ethernet, 64 MB of Flash, 128 MB of RAM, and a wide host of peripheral adapters such as A/D, D/A, asynchronous serial, synchronous serial, CAN bus, relay control, and JTAG. The card is supported by platform development and programming tools appropriate for software developers. The Pico E-12 platform advances desktop and portable computing by providing massively parallel hardware computing resources in a low-power, self-contained package (Figure 5).

Figure 5 – The Pico Computing EP-12 card packages the FX12 or LX-25 device with a CompactFlash interface in an extremely compact form-factor.

There are two versions of the Pico E-12. The Logic Optimized (LO) version is based on the Virtex-4 LX-25 device, while the Embedded Processor (EP) version is based on the Virtex-4 FX12 device, with its integrated PowerPC processor.

In either case, the FPGA on the E-12 card is configured from the 64 MB of on-board flash memory using an on-board loader. The unique design of this loader allows new FPGA images to be swapped into the FPGA on demand. A large number of FPGA images can be stored in flash memory, and any image can be loaded at any time through on-board software or external software communicating with the E-12 card through its external interfaces. The contents of the 128 MB of external RAM remain intact through the swapping sequence, allowing subsequent FPGA images to operate on existing RAM data.

The Xilinx ML403 (shown in Figure 6), the first of several Virtex-4 FX embedded processing development boards, combines the Virtex-4 FX12 with a wide variety of software-configurable interfaces, including network interfaces; serial, parallel, and USB ports; LVDS and D/A and A/D interfaces; and a VGA driver. As such, the ML403 is ideal for embedded systems designers requiring direct FPGA access to external hardware devices.

Figure 7 shows a comparison between the APU with Impulse-generated hardware accelerators and a processor/software-only implementation. Both systems are utilizing the ML403 in this example. You can see that the APU-accelerated version is significantly faster than the processor-only version.

Figure 6 – Xilinx ML403 development system

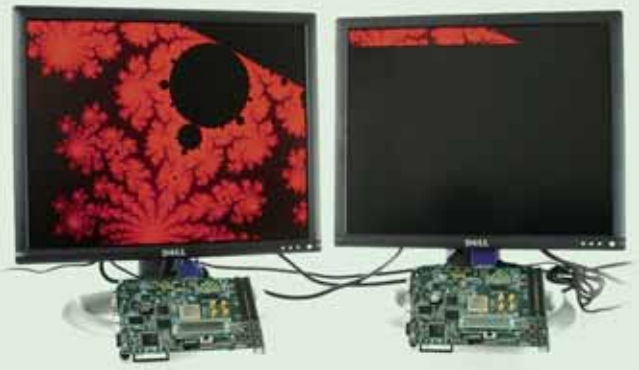
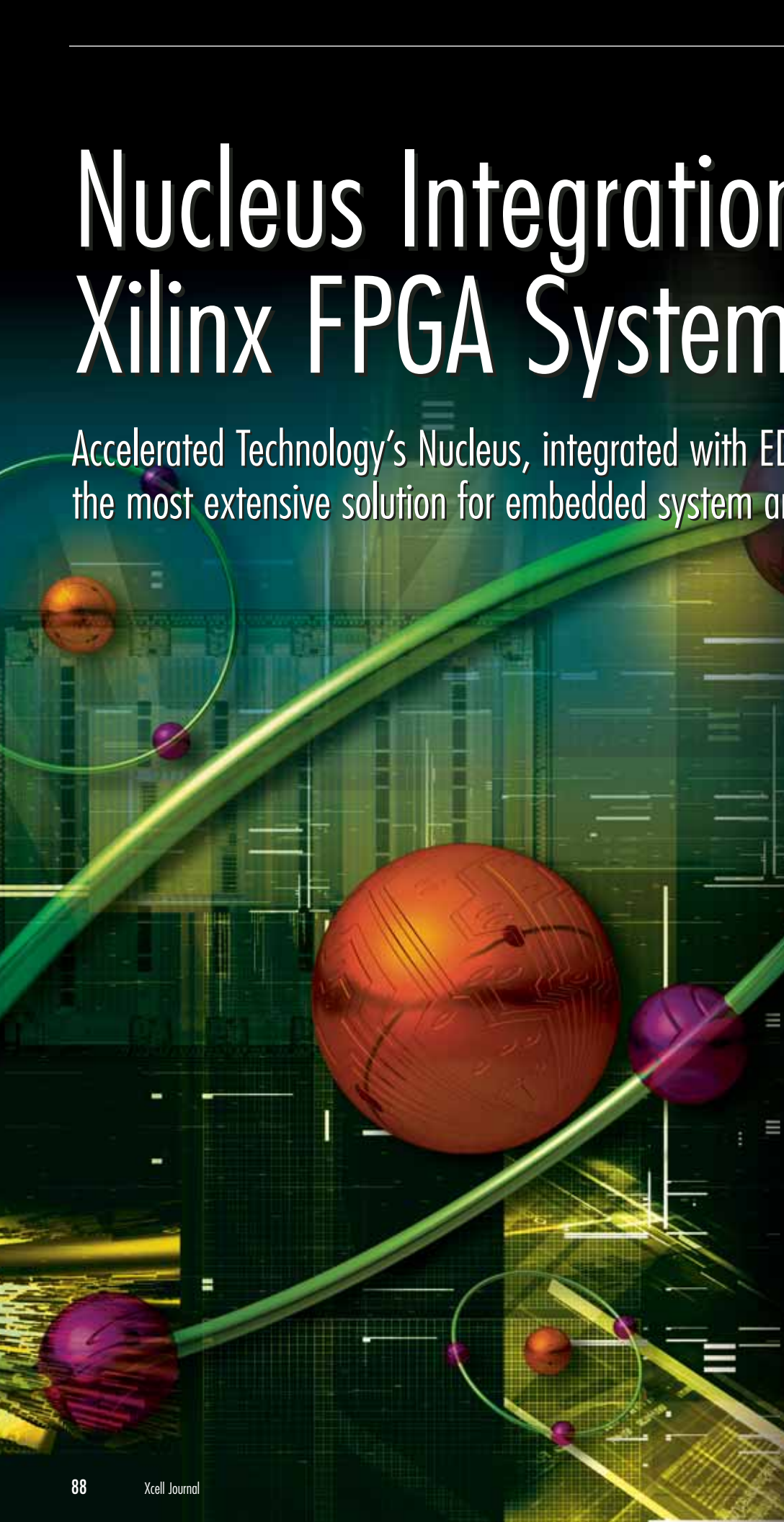


Figure 7 – Xilinx ML403 development system showing APU hardware accelerated implementation versus software only executing on the PowerPC

Nucleus Integration with Xilinx FPGA System Design

Accelerated Technology's Nucleus, integrated with EDK, provides the most extensive solution for embedded system architects.



by Aaron Spear
Lead Architect, Tools Solution
Accelerated Technology, A Mentor Graphics Division
aaron_spear@mentor.com

Phillip Walker
Technical Marketing Engineer
Accelerated Technology, A Mentor Graphics Division
phillip_walker@mentor.com

The FPGA-based embedded system design flow provides many benefits to system developers, as well as new challenges. Chief among the benefits are accelerated design flows that allow you to move quickly from the design and testing cycles to marketing and selling. With this accelerated flow, it is more important than ever that the hardware and software designs are in sync throughout the entire engineering design cycle.

Accelerated Technology, A Mentor Graphics Division, has developed a version of its Nucleus embedded software suite that integrates with the Xilinx® Embedded Development Kit (EDK). This provides a tight integration of software systems in the FPGA embedded systems design flow. EDK is based on a data-driven code base that makes it extensible and open. By leveraging this functionality, Nucleus software is able to achieve a level of integration into the FPGA-based embedded system design flow that was previously not possible.

The Nucleus embedded software suite for Xilinx FPGA system design includes a complete tool offering and target software platform, including high-level modeling with xtUML and advanced target software debugging with the Eclipse-based Nucleus EDGE environment.

Auto Configuration with MLD

The underlying technology of the Xilinx approach is microprocessor library definition (MLD). This technology allows for automatic kernel configurations and the generation of board support packages (BSPs). This unique and straightforward approach allows the Nucleus embedded software suite to configure to FPGA system designs created in EDK. This eliminates the need to re-port the software system to the new memory map and peripherals for every hardware design cycle.

The Data-Driven Approach

EDK uses two main data repositories to store information on hardware- and software-related settings. All hardware-related settings are stored in the MHS (microprocessor hardware specification) file, while software-related settings are stored in the MSS (microprocessor software specification) file. These files provide a database that other tools in the Xilinx system can access for any given project.

The data-generation component, as defined in a TCL file, takes hardware details from the MHS data file and custom information from the MLD file and decides which files to generate and what parameters to customize. The Nucleus-specific functions of the TCL file encapsulate the entire algorithm, generating consistent information used by the Nucleus kernel to support a wide range of hardware IP configurations (see Figure 1).

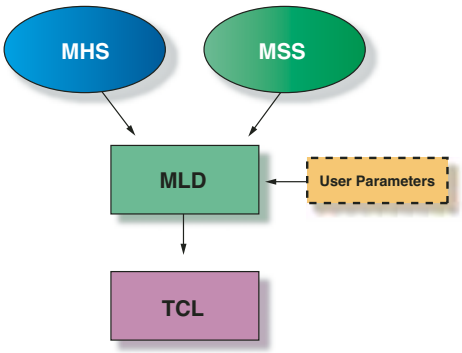


Figure 1 – Dataflow in Xilinx EDK

The elements of the Nucleus PLUS real-time kernel modified by the data generation file include:

- The number and type of peripherals used
- Memory map information
- Locations of memory-mapped device registers
- Timer configurations
- Interrupt controller configurations

Core Generation

Xilinx currently offers two processor choices for implementation in their FPGAs: the PowerPC™ 405 hard-core processor and the MicroBlaze™ soft-core processor. The Nucleus embedded software suite currently works with both options. Once you have selected your processor and configured your system inside EDK, enabling Nucleus is as simple as a drop-down menu selection.

Configuring Nucleus and BSP Generation

After you have generated your basic system design and core selection in EDK, you are ready to implement the Nucleus embedded software suite. As Figure 2 illustrates, we leveraged Xilinx MLD technology to add this functionality to EDK.

Once you have configured Nucleus to fit your system requirements, generating the corresponding BSP is straightforward. Simply choose the "Generate Libraries and BSPs" from the Tools menu option in EDK. This will build the correct libraries and any associated applications that the Nucleus embedded suite requires in order to run on your newly designed system.

System Debugging with Nucleus EDGE

You now have your hardware defined and the Nucleus PLUS kernel configured to run on your new platform. But what about software development? Where do we go from here?

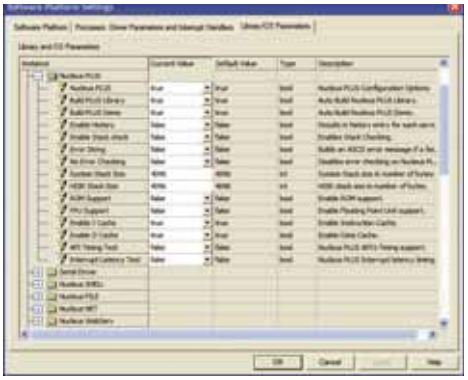


Figure 2 – Components of the Nucleus system are now configurable from within the EDK through the Library/OS Parameters dialog.

Introduction to Nucleus EDGE

Accelerated Technology offers the Eclipse-based Nucleus EDGE development environment, a complete development environment that supports JTAG debugging of both PowerPC and MicroBlaze processors. Nucleus EDGE extends the Eclipse platform for multi-core, multi-process, multi-thread debugging. It can be installed as a stand-alone or into Platform Studio SDK to provide advanced debugging and project management capabilities.

Nucleus EDGE is state-of-the-art debugging technology that includes features such as:

- Full multi-core/multi-process/multi-thread debugging, with support for synchronous operation on multiple cores simultaneously
- Advanced C-like scripting language (codelets)
- Data-driven target/core/peripheral descriptions (XML)
- Support for freeze-mode/run-mode debugging (OS- and hardware-dependent)
- Pluggable kernel awareness (data-driven)
- Pluggable connection devices
- Pluggable core support
- Pluggable real-time trace support
- Pluggable profiling engine

Creating and Building Applications

Nucleus EDGE provides a powerful build and project management environment. The Nucleus EDGE builder is a front end for any tool that transforms one or more files from one format into another format; examples include a compiler that transforms a C file into an object module or a linker that transforms N object modules into an executable. You can plug tools into the Nucleus EDGE builder by writing a simple XML description for that tool. We currently have built-in support for 32 different tool sets, including Xilinx GNU for both PowerPC and MicroBlaze processors.

BSPs

When targeting traditional processors with Nucleus EDGE, you are responsible for creating and maintaining BSPs that the debugger and project manager use. One advantage of Nucleus software integration with EDK is that BSPs are generated automatically, making maintenance painless. When you finish your hardware design and generate the BSP, the Nucleus EDGE BSP is also generated. This BSP is then used to determine appropriate tool defaults when creating applications, or knowing the layout of memory and peripherals when debugging.

Getting Started with Project Management

Nucleus EDGE provides a powerful user interface to change compiler settings for a given project, or optionally override them for a particular file. You are free to type in the command-line arguments if you know them, or you can peruse the options using a tree, which contains information about the command and allowable settings for it. It is nice not to have to wade through obscure compiler documentation to find the setting you need and its syntax (Figure 3).

Editing and Building

Nucleus EDGE provides a full-featured context-sensitive editor for C/C++ as well as assembly. The editor provides the following features:

- Configurable syntax highlighting (you can change the colors)
- Outliner that aids in navigation for your active source file
- Right-click navigation for declaration/definition of function calls
- During debugging, hovering over variables displays their current value; additionally, you can define your own script functions to render tool tips for your application data types
- Code completion for both functions and macros

Any errors in your source during building are displayed in the build console. You can click on errors; the editor synchronizes to the location for you. The editor decorates all warning or error source locations with special icons (Figure 4). Figure 4 also shows the outliner at the right side of the file.

Debugging PowerPC and MicroBlaze Processors

For Nucleus PLUS kernel applications, Nucleus EDGE can support run-mode debugging – the debug of individual tasks while the rest of the system continues to run. To accomplish this, it can use a serial port, Ethernet connection, or even the Xilinx JTAG UART.

For MicroBlaze processors, Nucleus EDGE currently supports debugging through XMD (Xilinx Microprocessor Debugger). For PowerPC, connection options include XMD, or, for PowerPC designs in which you have instantiated a dedicated JTAG scan chain, third-party JTAG devices such as Abatron's BDI2000 and MacCraigor Systems On Chip Demon family of connections.

Platform Debugging (Hardware/Software Co-Debugging)

One useful feature gained from connecting through XMD is that you can leverage ChipScope™ Pro hardware debugging features simultaneously while you debug your software using Nucleus EDGE. In the ChipScope Pro GUI, you get a logic analyzer view of signals inside your core. You can then configure the ChipScope Pro analyzer to halt the processor when the state of a certain peripheral changes, for example.

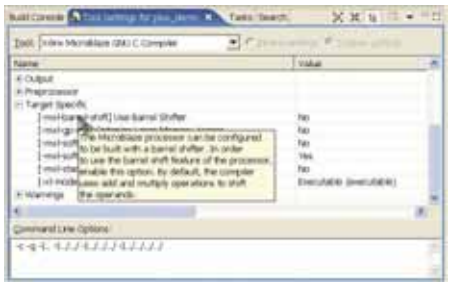


Figure 3 – Nucleus EDGE build settings for MicroBlaze GNU

Figure 4 - Building with errors

Figure 5 - MicroBlaze registers

When this occurs, Nucleus EDGE synchronizes, and you see the exact state of your software when the event occurred.

Debugging

Nucleus EDGE contains many special features for embedded debugging. The register view, for example, shows groups of native processor registers as well as memory-mapped peripherals. Those bits in the register that are set are highlighted in an optional graphical control. Bit-mapped registers show the bits set, and allow you to control them individually.

In Figure 5, you can see that "Exception Enable" is bit 6 in the MSR (the bold box around the bit), and that the bit is not currently set (the blue background). Gone are the days of getting out your calculator to do binary conversions and counting bits to

figure out if a bit is set. Figure 5 also shows how values that changed from the last step are color-coded (red).

Breakpoints

One other compelling feature that Nucleus EDGE offers above and beyond the capabilities of Platform Studio SDK is built-in integration with hardware breakpoints. As you may know, you can locate as many as eight program counter hardware breakpoints (used for stepping), as well as four read watch points and four write watch points in a given MicroBlaze design. Nucleus EDGE offers a completely integrated and graphical method to set both types of breakpoints. Also, the Nucleus EDGE debug engine is able to use the hardware breakpoints seamlessly to enable stepping in ROM.

Multi-Core Debugging

The number of MicroBlaze cores that can be placed in a design is only limited by the size of the FPGA. However, the MicroBlaze debug module can support debugging of as many as eight MicroBlaze cores simultaneously. The Nucleus EDGE user interface and debug engine have the ability to create "synchronization groups" of different cores. When one of these cores stops, all of the cores in the group are stopped.

Although Xilinx does not currently ship an IP block that supports configuration of synchronous control of multiple cores, it is a relatively trivial matter to implement it yourself – after all, you have an FPGA. Simply tie together a memory-mapped register with some MUX logic on the MB_HALT pin (that indicates that a core has gone into debugging state) as well as the DBG_STOP pin (that can force a core into debugging state). This way, when one core in a group either hits a breakpoint or has an exception, all of the cores stop. Then, in Nucleus EDGE, you can provide a codelet script that sets this register appropriately.

Codelets/Scripting

Nucleus EDGE contains support for a scripting language that we call "codelets." The syntax is standard ISO/ANSI C, with a few extensions. Simply put, codelets are scripts that run in the debugger but have full visibility and control over the target.

You can access target registers, memory, and variables, as well as call target functions from within a codelet. You can read and write host files as well as sockets. You can open "channel viewers" in the debugger GUI and execute them through any different expression evaluation. You can call them from the command line, or when hitting a breakpoint, or by typing an expression in the watch window. Codelets are meant to be an enabling technology. They allow you to get inside your hardware in a way that is not otherwise possible. Some things that customers have done with codelets include:

- Board initialization during debug
- Complex conditional breakpoints
- Custom hardware validation/ regression testing
- Virtual console I/O
- "Poor man's" kernel awareness
- SmartWatch – the ability to define a codelet that is used to "render" a given data type to a string, giving you nice tool tips for your data structures when debugging

Channels

Nucleus EDGE contains an abstraction for communications that we call channels. Any byte stream can be a channel. Files, sockets, and serial ports can all be channels. Codelets can also be used to create channels. On top of that, the GUI provides the ability to write "channel viewer plug-ins," a way to render the data that comes from these channels. Using this infrastructure offers all kinds of interesting capabilities. Nucleus EDGE currently ships with the following built-in channel viewers:

- Generic text console I/O (standard I/O with the app)
- VT-100 compatible console I/O (supports escape sequences)
- Strip chart recorder that allows you to plot any value over time in real time
- Windows Media Player streaming plug-in (plays MP3s, MPEG video) (on Windows hosts only)
- Binary data viewer (like the memory view, in effect a "protocol analyzer")

These viewers are just the beginning. Channels also allow us to abstract the mechanism used to connect to a profiling agent or run-mode debugging, for instance. When coupled with the Xilinx JTAG UART, this yields a powerful infrastructure for getting inside your application.

Kernel Awareness

Nucleus EDGE kernel awareness gives you the ability to see a snapshot of the state of your system, as well as providing the ability to set thread-dependent breakpoints. We currently provide out-of-the-box kernel awareness for the Nucleus PLUS kernel. However, Nucleus EDGE also gives you the ability to configure your own kernel awareness. This can be done for a third-party RTOS, an in-house kernel, or no RTOS at all.

Nucleus EDGE provides a data-driven mechanism to describe how it should iterate objects of a given type and display their attributes. They do not even have to be software objects – they could be anything that is memory-mapped (Figure 6).

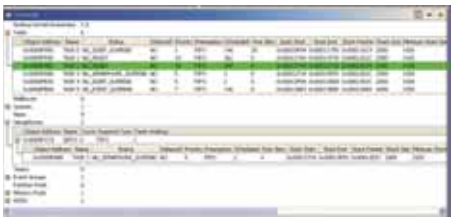


Figure 6 - Kernel awareness

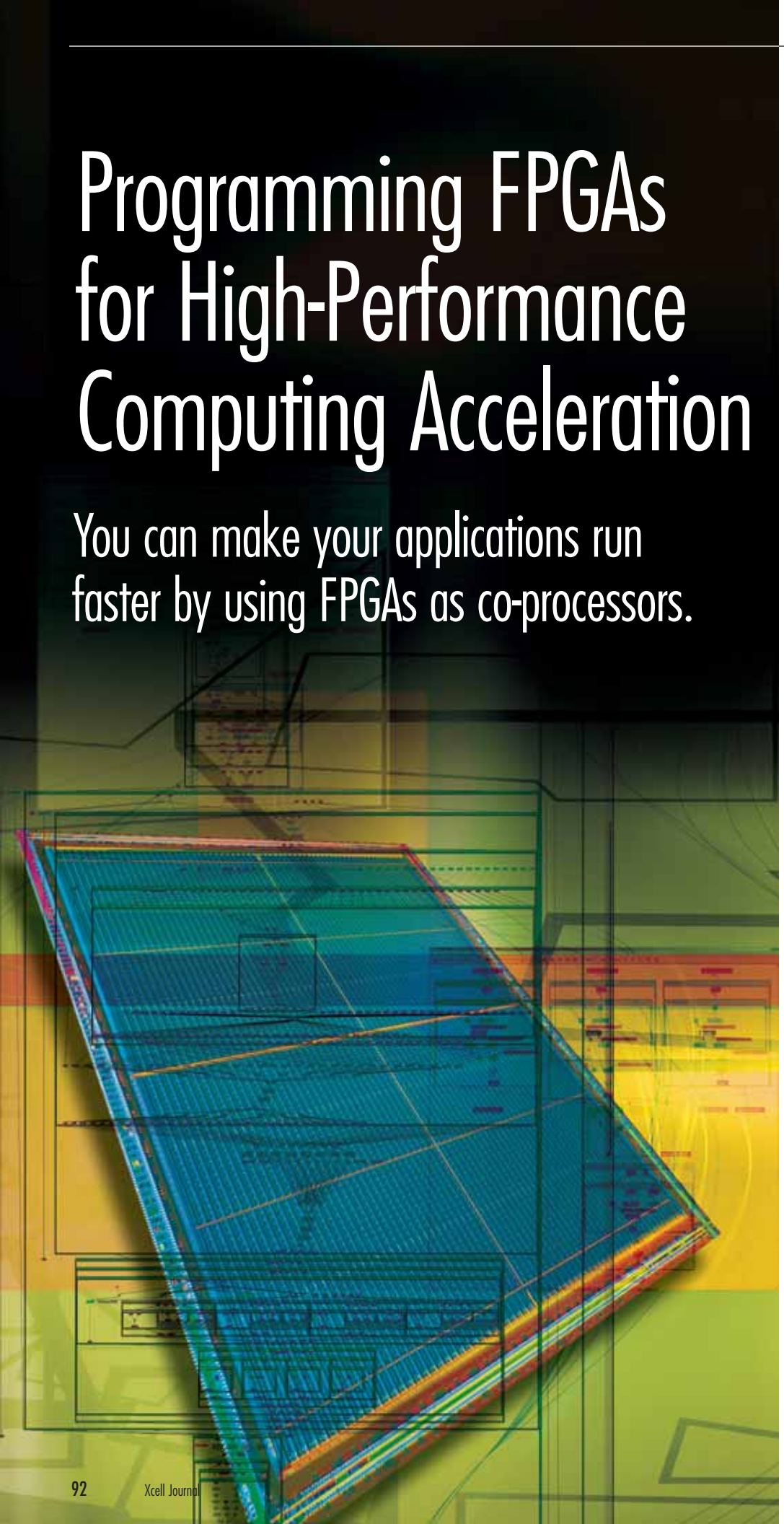
Conclusion

Configurable cores are the future of embedded development. With the combination of auto configuration of Nucleus target software and advanced debugging with Nucleus EDGE, Accelerated Technology has bridged long-standing gaps in integrated system design. By supporting both PowerPC- and MicroBlaze-based FPGA systems, Accelerated Technology distinguishes itself from the competition and provides unparalleled software tools and support for FPGA system designers.

For more information, evaluations, and updates to these exciting technologies, visit *www.acceleratedtechnology.com/xilinx*.

Programming FPGAs for High-Performance Computing Acceleration

You can make your applications run faster by using FPGAs as co-processors.



by Anders Dellson
President and CEO
Mitrionics Inc.
anders.dellson@mitrionics.com

An entirely new and exciting market and technology segment is emerging with the growth of FPGA-based high-performance computing (HPC). To date, technical obstacles have made practical success in this area more of a challenge than an opportunity. This year, the FPGA-based HPC market takes off, with system vendors (such as Cray and Silicon Graphics) and FPGA board suppliers (such as Nallatech) spearheading the hardware side of the equation. Now Mitrionics is providing its Mitrion Platform for fast and easy software programming of FPGAs.

FPGAs have the potential to be great application accelerators by working as co-processors, off-loading computationally intensive tasks from conventional CPUs. The promise of achieving orders of magnitude acceleration in processing has fueled a number of projects and enterprise ventures over the last decade. Some were proven very successful, while others were not. Until now, the critical component of converting applications to run on FPGAs has depended on the services of highly skilled hardware designers. This has put the benefits of FPGA performance boosts out of reach for the majority of software developers, who do not have the ability nor inclination to go into the extensive details of designing hardware.

At least that was the case until now. With the Mitrion Platform, software developers have the ability to convert applications to be accelerated on an FPGA without knowing the first thing about the complexities of hardware design. The Mitrion Platform allows FPGA-based HPC applications to be written in days or weeks with just a few hundred lines of code – versus taking months or years and tens of thousands of lines of code to write with other FPGA programming tools.

Putting FPGAs to Work

FPGAs are potentially small supercomputers. One of the main reasons FPGAs have not been prevalent in supercomputing is their lack of programmability. The users and application developers for supercomputers do not know hardware design, and have no wish to learn it. These users are often researchers with their own complex field of work.

The Mitrion Platform gives these users access to the performance of FPGAs, while still allowing them to write their algorithms in software. It is applicable to fields such as gene sequencing, weather prediction, image analyses, industrial automation, and geosciences. You will find FPGA-based application acceleration highly attractive in two main areas: from a raw processing performance perspective and because of the significant reduction of power consumption compared to clusters of regular microprocessors. Many problems within these areas have been difficult to address with traditional FPGA design methods, mainly because of their complexity.

The Mitrion Virtual Processor

Compiling software into a hardware design is not a trivial task. The program code that makes up the software and the transistors, wires, and gates that make up the hardware are very different things. We say that the best solution is simply not to do it.

Traditional processors solve this problem by using the von Neumann architecture, a machine (designed in hardware) that reads the program code in sequence and executes the instructions. The problem with the von Neumann architecture is that it really does not lend itself to the high level of parallelism that is required to extract the performance benefits that you can get from an FPGA.

Pontus Borg and Stefan Möhl, two "software guys" and the founders of Mitrionics, realized that the concept of an abstract machine that executes the software is the key to bridging the software-to-hardware gap. Software developed for the Mitrion Platform is compiled into instructions for the Mitrion Virtual Processor, which is then adapted accordingly and instantiated in programmable logic. It delivers massive parallelism and high silicon utilization, and is aimed at the acceleration of calculation-intensive programs.

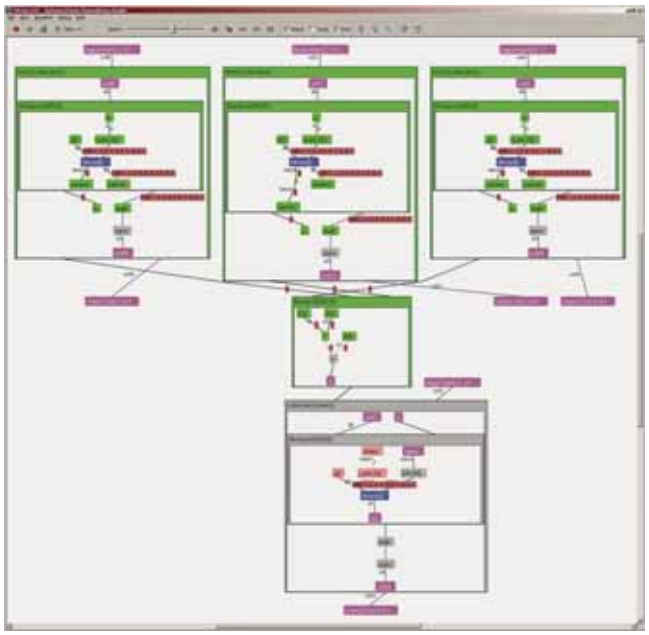


Figure 1 – The Mitrion debugger and simulator running the program shown in Figure 2.

```
Mitrion-C 1.0;

// Options: -cpp

mem2collection(a0, range)
{
  (values, a2) = foreach(e in range)
  {
    (value, a1) = _memread(a0, e);
  } (value, a1);
  a3 = _wait(a2);
} (values, a3);

collection2mem(data, a0)
{
  a2 = foreach(value in data by index)
  {
    a1 = _memwrite(a0, index, value);
  } a1;
  a3 = _wait(a2);
} a3;


#define RANGE <0 .. 0xffff> //Defines how many values to read from each memory

(mem int:64[0x100000], mem int:64[0x100000],
 mem int:64[0x100000], mem int:64[0x100000])
  main
(mem int:64[0x100000] extA, mem int:64[0x100000] extB,
 mem int:64[0x100000] extC, mem int:64[0x100000] extD)
{
  (a, extAr) = mem2collection(extA, RANGE); //Read values into vector a
  (b, extBr) = mem2collection(extB, RANGE); //Read values into vector b
  (c, extCr) = mem2collection(extC, RANGE); //Read values into vector c

  d = foreach(e0, e1, e2 in a, b,c) e0 * e1 + e2; //Calculate each element in vector d

  extDr = collection2mem(d, extD); //Write vector d back to memory
} (extAr, extBr, extCr, extDr);
```

Figure 2 – A Mitrion-C program that multiplies two vectors and adds a third vector element by element.

The Mitrion Virtual Processor performs thousands of operations simultaneously by allocating computational units for each instruction. The fine-grained nature of the processing elements permits every individual operation of the program to run in parallel.

To assure sufficient memory bandwidth, the Mitrion processor operates with simultaneous access to multiple shared external memories as well as all internal memory banks in the FPGA. Using heavy pipelining, data is communicated directly between the thousands of processing units, and we reduce the number of memory accesses and I/O performance bottlenecks.

With the Mitrion Virtual Processor, we are able to eliminate the direct translation of software to hardware.

The Mitrion-C Programming Language

To exploit the parallel processing capabilities of the Mitrion Virtual Processor, traditional, sequential programming languages are not sufficient. For that reason we have developed Mitrion-C, an implicitly parallel C-family programming language. Mitrion-C helps you (as a programmer) reveal and utilize the parallelism inherent in your algorithm. It gives you access to implicit parallelism. This means that parallelism is

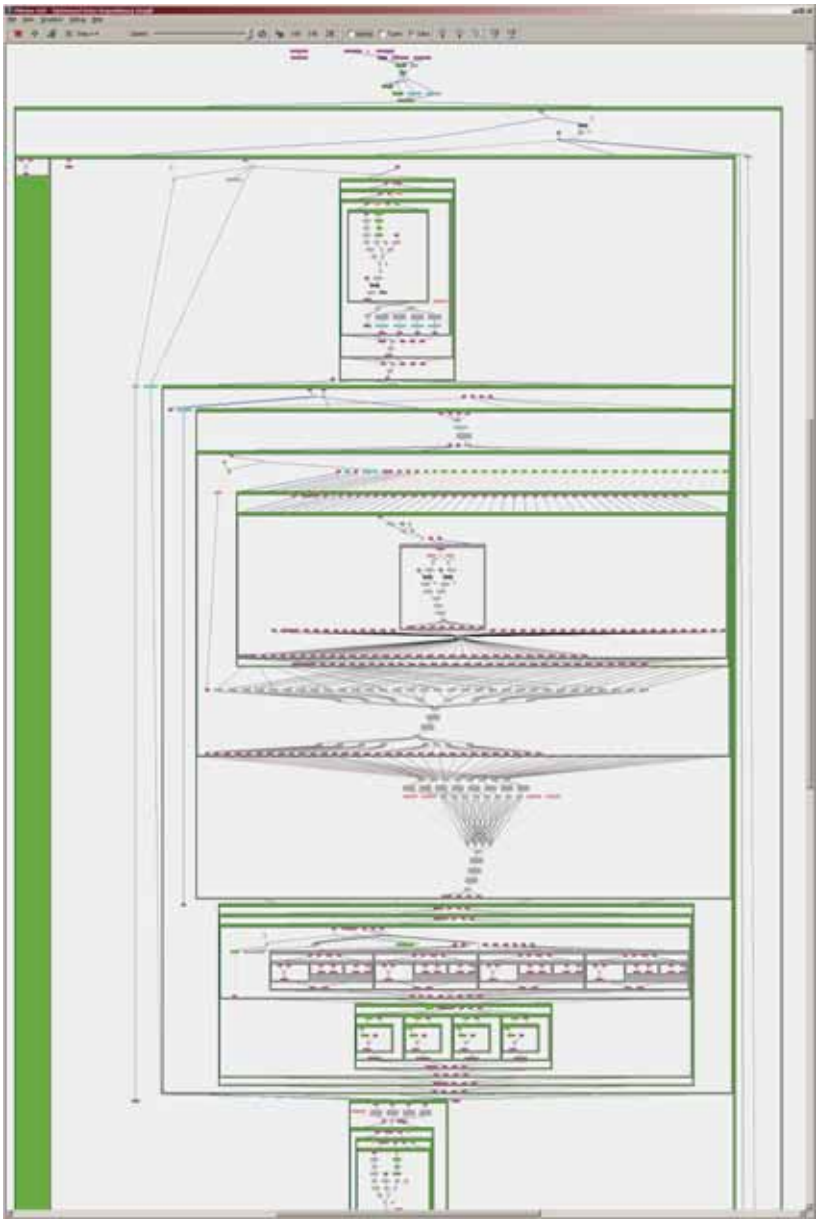


Figure 3 – A Bayesian Confidence Propagating Neural Network running in the Mitrion simulator and debugger.

found automatically, without you having to manually specify which parts of the algorithm are to be executed simultaneously. With Mitrion-C, many hundreds of dissimilar parallel interacting operations are created without the risk of deadlocks or race conditions. Mitrion-C is a C-family language, which is easy to learn and has a syntax familiar to C programmers.

Developing Software for the Mitrion Virtual Processor

For the great majority of applications, only a small amount of code limits the performance. Typically, when you develop an application to be run on the Mitrion Virtual Processor, you would identify the critical code and write that in Mitrion-C. You would then write the rest of the application in the language of your choice to execute on a host CPU, calling the critical code to be run on the Mitrion processor.

The Mitrion Software Development Kit comprises the Mitrion-C compiler, a graphical debugger, a code simulator, and a processor configurator. A C/C++ library is included that gives easy access to the Mitrion processor from the host application. Several integration modules interface the Mitrion Virtual Processor to a number of system platforms.

The graphical debugger and code simulator give you a hierarchical overview of all the parallel operations and data dependencies in the program. Figure 1 illustrates the results from running the debugger and simulator on the sample Mitrion-C code shown in Figure 2; Figure 3 shows a non-trivial example. Through the debugger, you will find it easy to locate programming errors and identify performance bottlenecks and inefficient code. This lets you efficiently refine your Mitrion-C program and then recompile.

The final output from the Mitrion Software Development Kit is a hardware design in VHDL for a Mitrion Virtual Processor, programmed and optimized to execute the software algorithm. You will then use third-party tools to complete synthesis and place and route. Because the algorithm you are developing can be tested and simulated inside the Mitrion Software Development Kit, there is no need for tedious and time-consuming iterations through the synthesis and place and route steps to confirm the functionality of your finished Mitrion Virtual Processor.

Conclusion

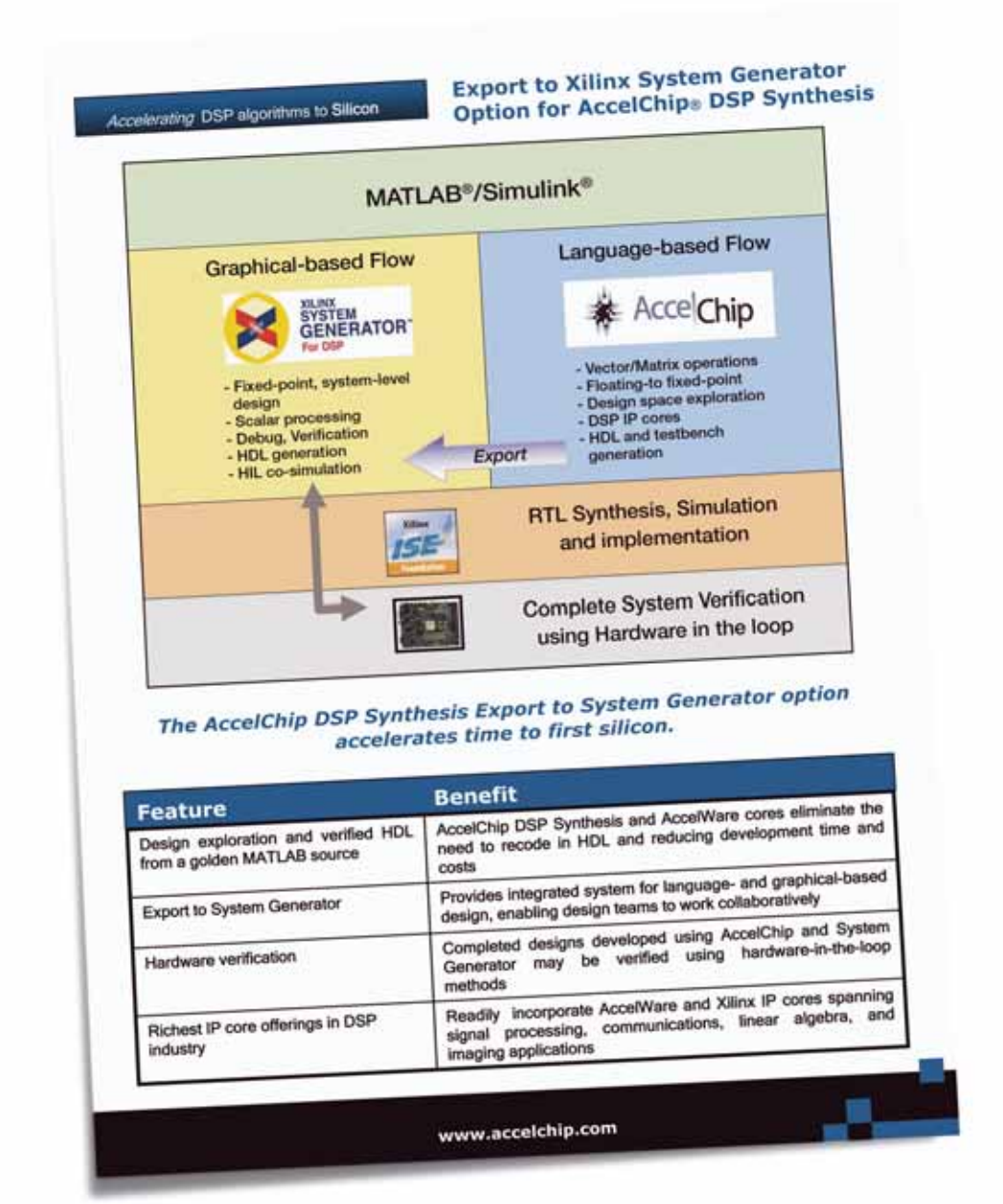
The Mitrion Platform allows rapid software development for FPGA-based systems to be completed by regular programmers who have no hardware design experience. The resulting applications are typically accelerated 10x to 30x compared to execution on a sequential CPU, with a fraction of the power consumption.

The development of FPGA technology is set to outpace the performance of traditional CPUs. Solutions created on the Mitrion Platform leverage this through the platform's portability. When a new, more powerful FPGA is available, all you have to do is reconfigure the Mitrion Virtual Processor to take advantage of the capabilities of the new device.

For more information, contact Mitrionics Inc. at (310) 558-9495, visit *www.mitrionics.com*, or e-mail *info@mitrionics.com*.

STOP OVER-THE-WALL ENGINEERING.

(FINALLY, ALGORITHM DEVELOPERS, DSP SYSTEM ARCHITECTS AND HARDWARE DESIGNERS ARE ALL ON THE SAME PAGE)



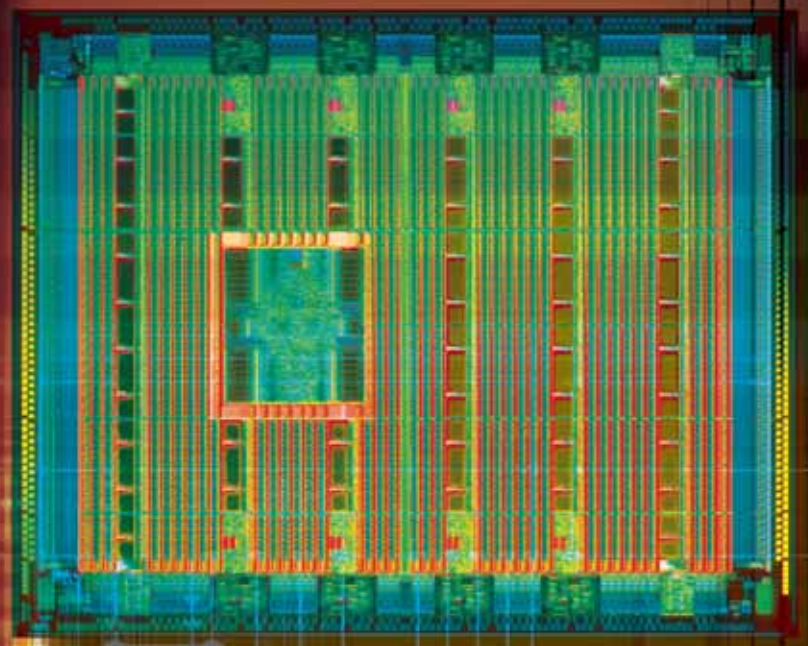
Company partnerships don't usually generate a lot of interest, but a partnership that yields team-building technology does. For example, the industry's first DSP design flow from a mixed MATLAB/Simulink environment to a verified FPGA. It's based on a tight integration between AccelChip DSP Synthesis and Xilinx System Generator for DSP that accelerates development of signal processing systems. Work more collaboratively and get designs to market faster by eliminating the need to recapture in HDL. Readily incorporate AccelWare and Xilinx IP cores and verify the entire system using hardware-in-the-loop simulation, further speeding development. How's that for teamwork? **Stop your Over-the-Wall engineering. Download free whitepapers at accelchip.com/papers or call (408) 943-0700.**

XILINX®

AccelChip

Flexibility with EasyPath FPGAs

You can now seamlessly convert to production and still retain some unique flexibility features.



by Gokul Krishnan
Sr. Marketing Manager, Market Specific Products
Xilinx, Inc.
gokul.krishnan@xilinx.com

Xilinx® EasyPath™ FPGAs are the industry's only customer-specific solution that gets you to volume production with very little risk and in as few as eight weeks. EasyPath FPGAs use the same silicon as standard FPGAs. The key difference between the two is that while the former are tested for specific customer designs, the latter are tested for all possible customer designs. By testing to a specific design, EasyPath FPGAs provide as much as an 80% reduction in unit price (due to improved yields) compared to the equivalent standard FPGAs.

One of the major advantages of EasyPath devices is that because they are identical (in all aspects except testing) to standard FPGAs, all features supported in standard FPGAs are in turn supported in the analogous EasyPath FPGA. From a customer perspective, this means that very little engineering resources are required to interface with Xilinx. The migration process itself is essentially risk-free. Once the design files are handed off, Xilinx creates custom test patterns based on the design to get to a guaranteed 99.9% stuck-at fault coverage. With EasyPath FPGAs, you get the same extensive Xilinx IP portfolio without any additional licensing fees for the EasyPath conversion.

Xilinx Virtex-4 and Spartan-3 EasyPath FPGAs allow you to retain some of the design flexibility of standard FPGAs even after the devices are in production.

Just as with structured ASICs (or standard cell ASICs), you would typically move from a standard FPGA to an EasyPath FPGA when your design has been frozen and the volumes justify a cost-reduction path. However, unlike structured ASIC solutions, EasyPath FPGAs provide you with some unique flexibility features, including:

* Dual bitstream and in-system engineering change order (ECO) capability
* Flexibility in production and lifecycle management
* Ability to use all standard FPGA features without any constraints

Dual Bitstream and In-System ECOs

Xilinx Virtex™-4 and Spartan™-3 EasyPath FPGAs allow you to retain some of the design flexibility of standard FPGAs even after the devices are in production. Specifically, the dual bitstream option allows you to target two different designs in a single EasyPath device – so long as their pinouts remain the same. This allows you to combine, for example, two different modes of operation in the same socket. One design could provide a diagnostic check that is active only at or soon after power up; another design could be active during the normal functional mode of the device.

A second way you can use the dual bitstream option is to try and address two different industry standards at the same time. Product specifications and market viability in many wired and wireless segments, for example, depend heavily on evolving standards. Given long product development cycles and the importance of being first to market, you can now go to market with potentially two different versions of a product to hedge your market position. When you exercise the dual bitstream option, Xilinx ensures that all of the resources corresponding to both designs are tested.

Another important flexibility feature in Virtex-4 and Spartan-3 families is the ability to make in-system ECOs. Specifically, the ECO feature allows you to make modifications to the combinatorial logic in an EasyPath FPGA (contained in look-up tables [LUTs]) and I/O block (IOB) parameters (such as drive strength and slew rate) even after volume shipments have begun and devices are deployed. Xilinx ensures that 100% of the LUTs used in the design and all combinations of drive strengths and slew rates for IOBs are completely tested to allow for any potential changes later.

This feature allows you to make simple bug fixes (within a LUT) such as adding or removing an AND gate or tweaking the drive strength of an I/O based on system requirements with minimal disruption to ongoing production. You can open the configuration equation of a LUT or the relevant IOB within the FPGA Editor tool and make modifications without having to rerun synthesis and implementation. To learn more about making these changes using FPGA Editor, see XAPP803, "Leveraging 'In-System ECO' Capability of Spartan-3 and Virtex-4 EasyPath FPGAs" at *www.xilinx.com/bvdocs/appnotes/xapp803.pdf*.

Figure 1 shows a screenshot of the FPGA Editor user interface that you would use to drill down into the functional blocks you want to change. The change process itself requires no intervention from Xilinx. You can make changes on your own, generate a new bitstream, download it into the EasyPath devices already in production, and implement a bug fix in the field in a very short time.

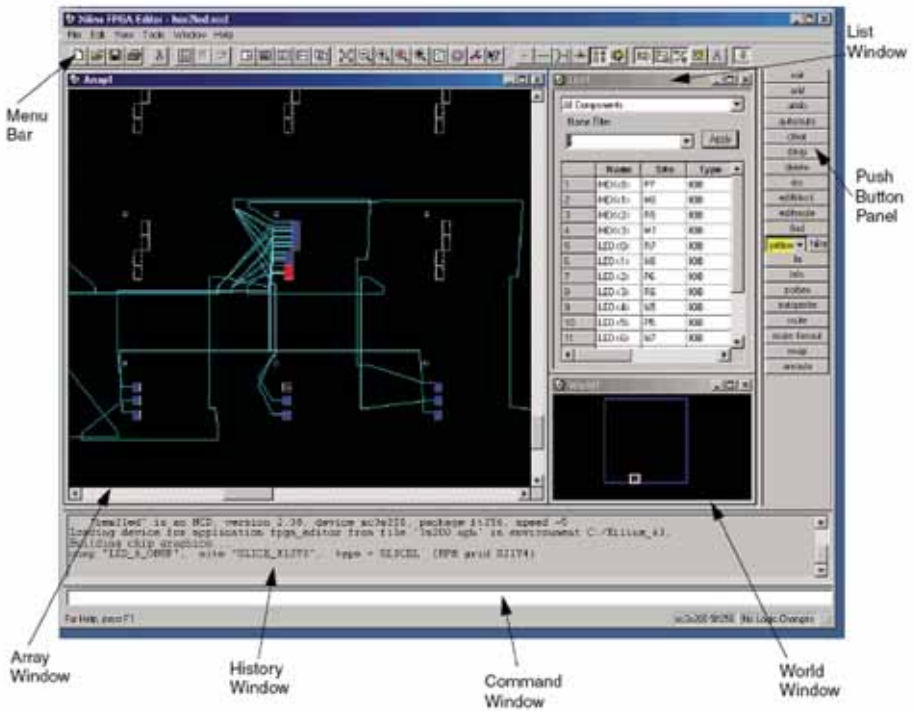


Figure 1 – Main window in FPGA Editor used to implement in-system ECOs in Virtex-4 and Spartan-3 EasyPath FPGAs
(Courtesy: Elizabeth Janney, Application Engineer, Xilinx, Inc.)

Production Flexibility

Another big advantage of EasyPath FPGAs is that because they are very similar to standard FPGAs, they can be used inter-

changeably with a standard FPGA on a given board. (All EasyPath devices are offered in the full range of packages as the corresponding standard FPGAs.) The advantage of this interchangeability is that should a design modification be required that cannot be accommodated by the ECO feature, you can quickly make modifications in a standard FPGA and continue shipping in production with standard FPGAs for the eight weeks or so that it takes to do the EasyPath migration of the modified design.

Because of the identical nature of EasyPath FPGAs and standard FPGAs, no additional prototyping phase or re-qualification is required. Instead, you go directly from design freeze to full production in as little as eight weeks. This quick turnaround allows you to postpone your design freeze milestone in the product development cycle and adapt to any last-minute changes in market conditions, demand, or design specifications. In today's age of just-in-time supply chain management, EasyPath devices enable you to get to market quickly without the constraints of large inventories.

No Constraints on FPGA Designs

With alternative cost-reduction solutions such as structured ASICs (or standard cell ASICs), you typically have to plan ahead to take advantage of the lowered cost. Some of the migration issues you may face when converting from FPGAs to structured ASICs are well documented (see the November 2004 article in the *FPGA Journal*, "Customer-specific FPGAs: Low-cost solution for volume production"). To get around some of these issues, some vendors impose significant constraints on the up-front FPGA design, thus reducing the flexibility that FPGAs are designed to provide.

EasyPath FPGAs, on the contrary, do not impose any design constraints. You can take full advantage of the flexibility and embedded features (such as multipliers, PowerPC™, Ethernet MACs, and high-speed transceivers) in standard FPGAs to cost-reduce if the design/market conditions are appropriate.

Conclusion

With the introduction of Spartan-3 and Virtex-4 EasyPath FPGAs, you can now prototype with standard FPGAs and then move to the corresponding lower cost EasyPath FPGA in a seamless fashion. The unique in-system ECO and dual bitstream capabilities in these EasyPath FPGAs allow you to make changes in your design and target different functional modes even after you have moved into high-volume production.

The quick time to market and interchangeability of standard FPGAs with EasyPath FPGAs allows you to implement bug fixes and efficiently manage inventory without loss of revenue or disruption of production. In addition, EasyPath FPGAs do not impose constraints on the FPGA design, thus leaving you with the flexibility to choose if and when you want to freeze your design and cost-reduce.

The EasyPath "Migration-Free Advantage"

Since their introduction in March 2002, EasyPath FPGAs, which offer an innovative approach to high-volume cost reduction, have received broad market acceptance. This has been driven by the higher level of flexibility and ease of migration offered by EasyPath FPGAs when compared to traditional ASIC solutions.

In the past year, EasyPath FPGA usage has grown by more than 600% by enabling customers in applications from communications equipment to storage solutions to achieve a total cost of ownership that is lower than any ASIC. With the introduction of Virtex-4 EasyPath and Spartan-3 EasyPath FPGAs, customers will continue to benefit from the EasyPath "migration-free" advantage.

Xilinx Embedded Ethernet MACs Negotiate the Data

You can use the Xilinx Embedded Ethernet MAC as an interoperable standard for data communication between the FPGA and external devices.

by Timothy Campbell
FPGA Programmer
University of Vermont, Burlington, VT
tcampbel@uvm.edu

Tian Xia
UVM Assistant Professor of EE
University of Vermont, Burlington, VT
xiat@cems.uvm.edu

DSP algorithmic realization in FPGA fabric can be easily controlled and debugged through high-speed Ethernet communication. This interoperable solution allows for portability and usage in other designs. Because of the widespread use of the standard, the FPGA is capable of transmitting data at high speeds to a vast array of external devices.

In the example we'll present in this article, a broadcast HTML page is used as a debugging interface to the FPGA. This broadcast page can be easily modified through software to tailor to your debugging needs. The design is built on a sample design packaged with the Xilinx® ML403 Virtex™-4 evaluation board, in which the HTML code is stored locally to the FPGA. Connecting to the IP address of the Ethernet MAC (media access controller) through a Web browser loads the HTML file.

Ethernet Basics

The first Ethernet standard was produced in 1985. Ethernet fits into the Open Standards Interface model of the International Standards Organization, as illustrated in Figure 1. Several LAN technologies are in use today, but Ethernet is by far the most popular technology for departmental networks. The vast majority of computer vendors provide equipment with Ethernet attachments, making it possible to link all manner of computers with an Ethernet LAN.

Transmitted data is encapsulated in a so-called Ethernet frame, which has defined fields for data and other information such that the data gets to its destination and the destination computer is able to discern whether the data it receives is valid. The frame format defines Ethernet and is illustrated in Figure 2. Frame sizes vary from 64 to 1518 bytes, and can be up to 1522 bytes when VLAN (virtual bridged local area network) is tagged.

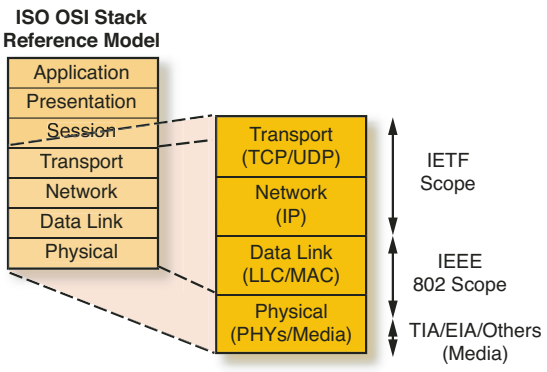


Figure 1 – Ethernet's fit in the Open Standards Interface model of the International Standards Organization

As noted in Figure 2, the source and destination address of the transmitted data is encapsulated within the frame. The other fields consist of:

- The preamble, which is a repeating pattern of 1010 needed for some PHYs
- Start-of-frame delimiter (SFD), which marks the byte boundary for the MAC
- Type/length of frame
- Data
- Pad, which is only necessary to extend the frame to 64 bytes
- Checksum, which implements a cyclic-redundancy check (CRC) to determine if the frame is sent in error
- Idle, which occurs between frames and must be at least 96 bit times

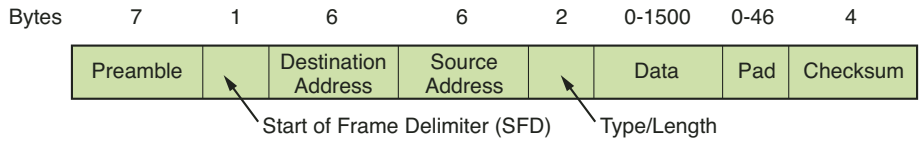


Figure 2 – Ethernet data frame

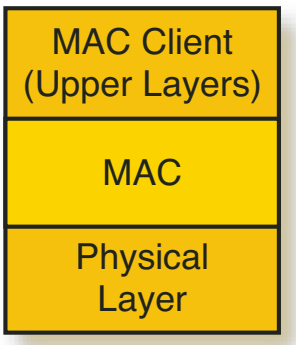


Figure 3 – Ethernet MAC layer

Ethernet frame transmission is controlled by the MAC layer. The MAC handles data encapsulation from the upper layers, frame transmission and reception, data decapsulation, and delivery to upper layers. The MAC operates independently of the physical layer employed and thus does not need to know the speed of the physical layer (as shown in Figure 3).

The Virtex-4 series offers two embedded Ethernet MACs. In this way, C-code software implementation through the PowerPC™ processor can be used to tailor the MAC to the particular application. You are thus encapsulated from the lower layers of the MAC.

The Xilinx Ethernet Register Interface

The Ethernet capability of a Xilinx FPGA allows for possibilities such as broadcasting captured data from the FPGA to an HTML-based website. In this manner, data can be sent to and retrieved from the FPGA through user I/O to the HTML GUI. The HTML page we used in building our project is shown in Figure 4.

A register interface is critical in the debugging of a design. It allows you to trigger events, obtain the status of intermediate results, and dump the values stored in block RAM. The latter is especially useful for processing and verifying a partial piece of an algorithm. As an example, an FPGA with an ADC feeding into it can serve as a sampler through a register dump of stored ADC sampled data in block RAM.

The contents of block RAM data can be accessed through a simple register read/write procedure. To read, we select an SRAM block by writing the block number in Register 1, and write the SRAM address to Registers 2 and 3. After the completion of register writing for Register 6, the SRAM contents corresponding to the SRAM address written in the SRAM block selected are available for read out through Registers 4, 5, and 6.

Figure 4 – Example HTML page connection through communication to the Xilinx embedded MAC IP address

To write a register, we select an SRAM block by writing the block number in Register 1, and write the SRAM address to Registers 2 and 3. Then we fill the SRAM contents we want to write into Registers 4, 5, and 6. After the completion of register writing for Register 6, SRAM contents will changed to new value.

The schematic shown in Figure 5 shows a single 8-bit register and the signals needed to control it. A select bus allows for multiple registers based on the size of the select bus. The "write_d0" signal allows the register to be written to. When reading a register, its contents appear on the REGOUT(7:0) output, and this output is fed back to the HTML page during a register read. The alternative register output is used to act as control signals, sample input to block RAMs, or whatever the situation may call for. We'll provide a detailed description of the transmission of data from the register to the HTML debugging page in the next section.

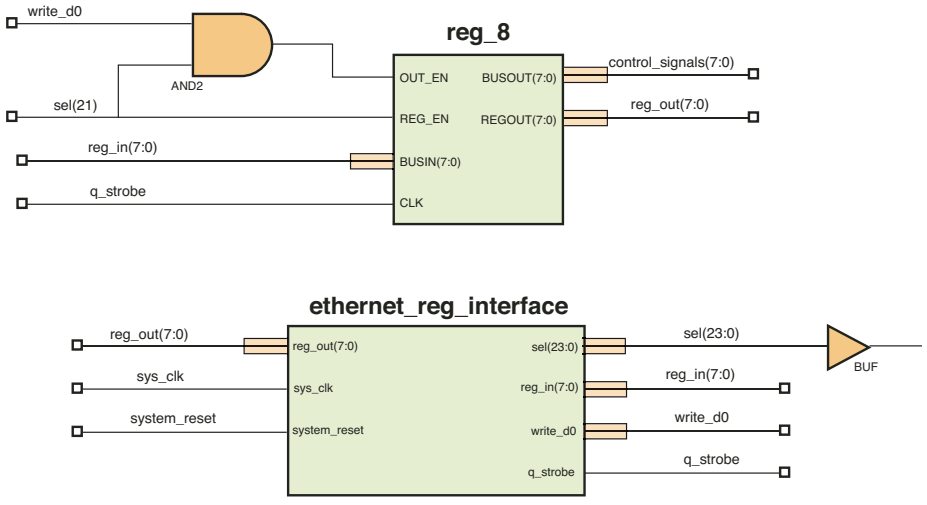


Figure 5 – Ethernet register interface schematic

Implementation of the Ethernet Register Interface

The Xilinx EMAC interface we used supports the IEEE Std. 802.3 media independent interface (MII) to industry-standard physical layer (PHY) devices and communicates to the PowerPC processor through an IBM on-chip peripheral bus (OPB) interface. The EMAC comprises two IP blocks as shown in Figure 6. The IP interface (IPIF) block is a subset of the OPB bus interface features chosen from the full set of IPIF features.

The proposed EDK design to implement the register interface through Ethernet uses a general-purpose input/output (GPIO) core for the processor local bus (PLB) bus. The GPIO is a 32-bit peripheral that attaches to the PLB. We used this GPIO capability to communicate between the PowerPC C-software implementation and the register interface linked to the DSP algorithmic implementation using dedicated logic. Figure 7 illustrates the block diagram for the Ethernet register interface.

The Xilinx EDK PowerPC design was easily integrated into ISE™ design tools to access DSP algorithmic implementation (the non-software portion of the design implemented in dedicated logic). This was achieved by exporting the design to ISE software and encapsulating the logic to communicate to the PowerPC inside a schematic block, as shown in Figure 6. The GPIO bus was used to feed input and output to the processor, as different bus lines served different purposes. At a higher level, this schematic block was packaged with a state-machine interface to achieve the register communication to the FPGA, as illustrated in Figure 5.

Xilinx ML403 Board: Test Platform for the Interface

At first, the task of employing Ethernet as a means of communication between the FPGA and the "outside world" seems pretty daunting. This is not the case, as Xilinx provides a reference design from which to build your own custom design. Xilinx also offers a tutorial for those not familiar with implementing processor designs using EDK.

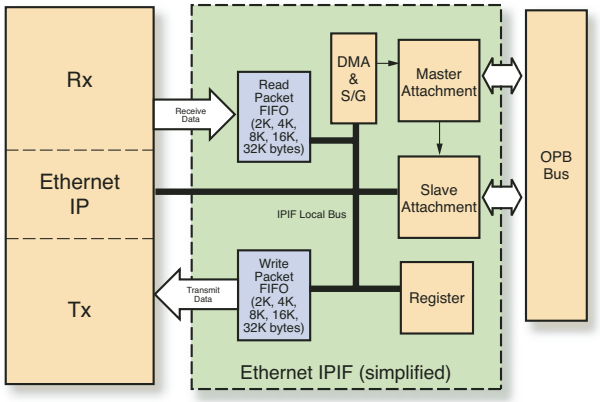


Figure 6 – IPIF and EMAC modules

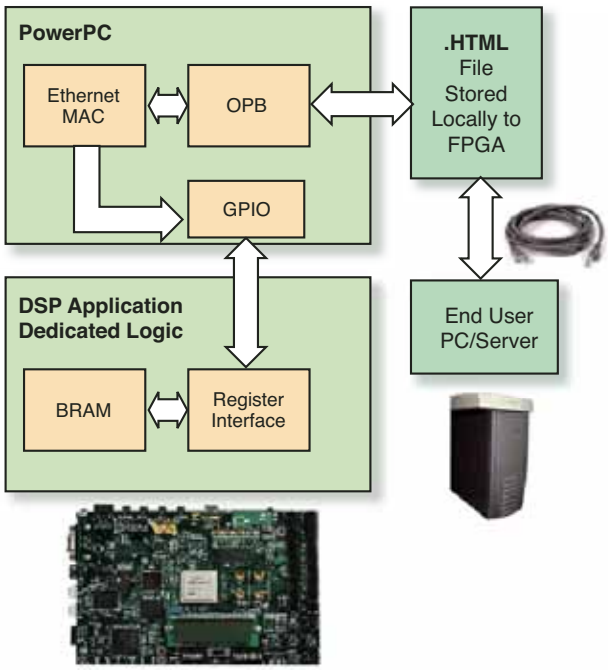


Figure 7 – Block diagram of proposed application

The ML403 Embedded Processor Reference System contains a combination of known working hardware and software elements. The reference system demonstrates a system utilizing the PLB, OPB, device control register (DCR) bus, and the PowerPC 405 or MicroBlaze™ processor core. The design operates under the EDK suite of tools, which yields a graphical tool framework for designing embedded hardware and software. Each of the pieces of the system can be separately activated as a stand-alone project.

The proposed Ethernet register interface was built from the Webserver project, which comes as part of the ML403 reference system. The Webserver project implements an Ethernet MAC through an IBM OPB, built-in for interfacing with the PowerPC 405. This core supports 10BASE-T- and 100BASE-TX/FX-compliant PHYs in full- or half-duplex mode.

The reference design serves as a good starting point to get you up and running with a design. The groundwork is laid out for you, allowing for additional modifications and enhancements.

Conclusion

What better way to transmit data from the FPGA to an auxiliary device, be it a web page or server, than Ethernet? Regarded as one of the most interoperable communication standards, Ethernet allows you to achieve high-speed data transmission with a widely employed standard.

Virtex-4 devices offer two embedded Ethernet MAC cores, and when combined with the design environment ease of EDK, an application for data transmission can be achieved with relative ease through C-code software implementation. The capabilities of such a system allow for FPGA algorithmic debugging and trigger, as well as SRAM fill and read at speeds as high as 1 Gbps, providing control and debugging of dedicated logic DSP algorithmic realizations.

Getting You Started With
On-Site Embedded Training

www.xilinx.com/epq

With Embedded Processing QuickStart!, Xilinx offers on-site training and support right from the start. Our solution includes a dedicated expert application engineer for one week, to train your hardware team on creating embedded systems, instruct software engineers how to best use supporting FPGA features, and help your team finish on time and on budget.

The Quicker Solution to Embedded Design

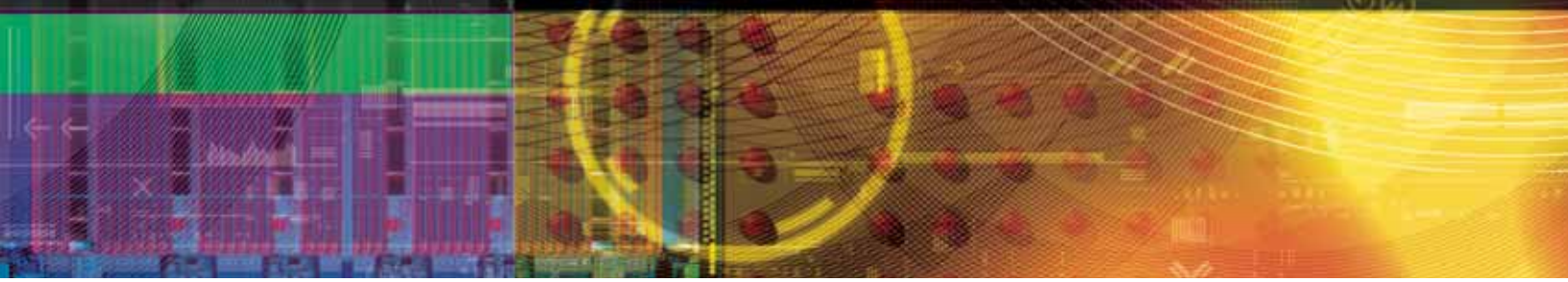
QuickStart! features include configuration of the Xilinx ISE™ design environment, an embedded processing and EDK (Embedded Development Kit) training class, and design architecture and implementation consultations with Xilinx experts. Get your team started today!

Contact your Xilinx representative or go to **www.xilinx.com/epq** for more information.

XILINX®

Program SPI Serial Flash from Xilinx FPGAs and CPLDs

It's easy to do it yourself by using tools available today.



by Rick Folea
CTO
Ricreations, Inc.
rfolea@UniversalScan.com

The good news: starting with the Xilinx® Spartan™-3E device, you can now use industry-standard SPI Serial Flash to configure your Xilinx FPGAs. Even better news: this will be the trend for all new Xilinx devices going forward, meaning that you will have multiple vendors from which to choose inexpensive and readily available memories.

The bad news: Xilinx won't be supporting these devices with programming tools – you are left on your own to program them. In this article, I'll take a look at SPI Flash, explain your options, and help you get up and running quickly.

SPI Flash – What's the Big Deal?

Until now, memories used to configure Xilinx devices had to be specific Xilinx-supported memories. This meant potentially higher prices per byte of storage and being subject to limited deliveries of devices through limited sources.

By making the switch to SPI Flash, you now have many vendors from which to choose, a wider variety of memory densities and types, and most importantly, lower cost and better availability. Table 1 shows a list of SPI Flash vendors and devices compatible with Xilinx FPGAs supporting SPI Flash.

SPI Flash Varieties

SPI Flash can be lumped into three general categories: SPI Serial Flash for code storage, SPI Serial Flash for data storage, and Atmel's DataFlash.

DataFlash devices are Atmel's older Serial Flash AT45DB parts. They are mature, plentiful, and have densities up to 128 MB. They are slightly more difficult to use because the programming and erase algorithms don't conform to the SPI Flash algorithms, and there aren't many vendors from which to choose. So unless you absolutely have to have 128 MB of storage, you will probably want to look at the more recent SPI Flash memories.

Figure 1 shows two types of SPI memories: code and data. These are really the same thing, except that data memories have a few extra instructions that allow you to do things like write to memory without erasing first; this comes in handy when you are trying to quickly read/write data from your application. Unless you have a real need for the data memories' special features, you will probably want to stick with using the programming features of the code memories for now – just because there are a lot more vendors and devices from which to choose. You can still use the data memories, as they are a superset of the code memories, but don't rely on the data memories' special command sets if you want to keep your options open.

SPI Flash Memory Caveats

SPI Flash memories from the various vendors are very similar, but there are a few things to watch out for. Most of the newer devices support a "ReadID" instruction so that you can identify the device by polling it. Make sure that you read the fine print in the datasheet. For example, although the ST datasheet for some of the M25P series devices have the ReadID instruction listed, you have to read the fine print, which says that only devices with an "X" at the end of the part number support this feature. I was unable to find these devices anywhere.

Although all vendors' devices support read/erase/program instructions, they all use different command sets. The commands do the same things, but use different bits for the commands (a page program for an ST device is a 0xD8, but a page program for an ATMEL device is 0x52, for example). See the individual instructions below for more caveats.

Vendor	Series	Type	1M	2M	4M	8M	16M	32M	64M	128M
ST	**M25P**	Code								
ST	M2/45PE	Data								
ATMEL	**AT45DB**	Data								
ATMEL	AT25F	Code								
NexFlash	**NX25P**	Code								
SST	**SST25VF**	Code								
SST	**SST25LF**	Code								
AMD/Fujitsu	S25FL	Code								
PMC	**Pm25LV**	Code								
SAIFUN	SA25F	Code								
SAIFUN	SA25C	Code								
Sanyo	LE25FW	Code								
MXIC	MX25L	Code								

Figure 1 – SPI Flash is available from many vendors in a wide variety of memory densities (green = available, yellow = planned, gray = not available, and boldface series have been tested in Xilinx Labs).

Programming SPI Flash Memory

As long as you stick with the basic instructions and keep an eye out for the caveats I've discussed, it is really pretty simple to write a small, dedicated SPI flash programmer yourself. Note that all data and commands are shifted in msb first.

To erase the device, simply:

- Lower chip enable
- Serially shift in the 8-bit erase command
- Raise chip enable

You can then check the status register to see when the erase cycle is done or wait the maximum expected erase time (one to several hundred seconds, depending on manufacturer and device density).

To read data from the memory array:

- Lower chip enable
- Serially shift in the 8-bit ReadData command
- Serially shift in the address (24 bits on most devices, msb first)
- Continue issuing clocks, capturing data from the memory serial output
- Raise chip enable when you have shifted out as many bytes as you want

You can continue issuing clocks right up to the end of the device, at which point the address will wrap and start back at the beginning (on most devices).

To write data to the memory array:

- Lower the chip enable
- Serially shift in the WriteEnable command
- Raise chip enable
- Lower chip enable
- Serially shift in the 8-bit PageProgram command
- Serially shift in the address (24 bits on most devices, msb first)
- Shift 8-bit data in, msb first (you can clock up to 256 bytes at a time on most devices)
- Raise chip enable
- Monitor status register or just wait the maximum time (a few msec, typically) to see when write is complete
- Repeat for next page (typically 256 bytes)

Note that if you go past a page boundary, the data will wrap and overwrite the data at the bottom of the page – it will not continue on to the next page.

That's all you really need to get started. As you can see, it is really pretty simple to program an SPI Flash memory, especially if you are creating a small programmer for a dedicated application and you stick with the basic command set.

Tools Available Today

Creating your own programmer works well on a microprocessor where you have an SPI port and room in your memory for a few extra functions. But what if you want to program a memory connected to a small CPLD or FPGA? You may be surprised to find that there are not a lot of low-cost, simple, general-purpose programmers available. The options we have today for non-microprocessor applications fall into the categories I'll discuss next.

Dedicated FPGA Configuration

Because the command set is small and simple, some designers have resorted to creating a dedicated FPGA configuration that can be used to program the SPI Flash.

The designers at Memec (recently purchased by Avnet) have taken this one step further – their configuration uses the MicroBlaze™ soft-core processor and on-chip peripheral bus (OPB) SPI core. Given the processor in the FPGA, they simply wrote an application to read and write the SPI Memory. They ran a great demo of this at the XFEST seminars last Spring that showed how the SPI Flash could be used to:

- Configure the FPGA (Spartan-3E device)
- Boot the MicroBlaze processor
- Be used for non-volatile data storage

Call your local Memec/Avnet FAE for a demo or more information.

These methods provide very fast SPI Flash access, but require a device with enough space to accommodate the configuration. You could not use it on a CPLD or other non-configurable devices.

X-SPI Utility

There is a little-known utility on the Xilinx website called "X_SPI." If you put an extra header on your board and can 3-state all devices connected to the SPI bus, then you can use this command-line utility to directly program your SPI Flash with a Xilinx Parallel-III or Parallel-IV cable. This is documented in XAPP800, "Configuring Xilinx FPGAs with SPI Flash Memories Using CoolRunner-II CPLDs," along with

a method to use a CoolRunner™ CPLD to transform the SPI signals into the signals required by Xilinx FPGAs (pre-Spartan-3E devices). The downside to this approach is that you have to add an extra header to your board, be able to 3-state all devices on the SPI bus, and hope that the utility supports the SPI Flash device you want to use.

A Better Way – JTAG

XAPP800 also mentions JTAG, but at the time there were no simple, inexpensive general-purpose tools available to program SPI Flash through JTAG. Now, the latest release of Universal Scan supports programming of SPI Flash. We offer a free trial and the SPI programming feature has been added to this hardware debugging and memory programming tool without increasing the price.

Using the Boundary Scan chain in your Xilinx device to program SPI Flash is a snap – you just specify which pins are connected to the SPI device, choose a data file, and hit "program." You don't need any test vectors, test executives, CAD data, or anything else normally associated with Boundary Scan test and programming operations. The pins on the Xilinx device are placed into EXTEST; vectors are automatically formatted and shifted in to drive the SPI Flash memory pins, and the next thing you know the memory is programmed. It includes the usual erase, program, and verify functions along with a memory viewer, utilities, sector erase capability, and automatic device ID interrogator.

The beauty of this method is that you can use it on any FPGA, CPLD, microprocessor, Ethernet switch, or DSP – any device that has a JTAG port. The device does not need to be configured or programmed (or even necessarily working entirely correctly) for this to work. Plus, it requires no additional precious device resources in your FPGA or CPLD, no special code or configurations, and does not require any special headers or other support devices on your circuit card.

There is a downside to this – it can be much slower than the other methods because you have to shift a full-scan vector into the JTAG chain for each and every clock edge, data setup, and chip enable. As an example, suppose you have an FPGA with a scan chain of 2,000 bits. It takes one scan to setup the serial data into the SPI Flash, another to raise the clock, and yet another to lower the clock. We need eight sets to shift a single byte into the SPI Flash memory, and that's 48,000 clocks just to get 8 bits into the Flash device. Multiply that by the number of bytes you have to program and you can see why it takes a while, especially if you use a parallel port that is fundamentally limited to a few hundred kilohertz TCK rate.

If you need to speed up the programming time, take a look at the newer USB2.0 versions of these products; they increase the TCK rate by an order of magnitude. You can also make sure the SPI device is connected to a device with a short scan chain and put all the other devices into bypass mode to help speed things up.

There are also traditional JTAG tools available that will program SPI Flash, and if you are lucky enough to have access, definitely take advantage of them. These tools are very powerful and can program memories rapidly. The only drawback is that they can require more setup than the method outlined previously and also tend to be more expensive. But if you need speed, the traditional Boundary Scan tools are a great option. (See the sidebar for a list of JTAG tool vendors.)

Other Methods

Most SPI Flash vendors have some kind of utility for programming their devices, but they are usually limited to their devices, require direct connection to the device, or are a cost adder to another set of utilities.

Conclusion

Figure 2 summarizes all of these options I've described. If you need speed – design your own. If you can afford some board modifications and an extra header, consider using the X-SPI utility, which is free. Otherwise, take a look at the new JTAG tools available for your SPI Flash programming needs; they are easy to use and inexpensive.

The proliferation of SPI Flash devices, memory densities, and ease of use make SPI Flash an ideal configuration or data storage solution for your next design. And because Xilinx is leading the way for future devices, there is no time like the present to get started.

For more information about Universal Scan, visit *www.universalscan.com.*

Method	Use on FPGAS	Use on CPLDs	Use on Any JTAG Device	General Purpose	Extra HW Reqd.	Multiple Vendors	Prog Time
Special Configuration	Some	No	No	No	No	Yes	Fast
X-SPI	N.A.	N.A.	N.A.	Yes	Yes	Some	Medium
JTAG	Yes	Yes	Yes	Yes	No	Yes	Slow or Medium
Miscel	No	No	N.A.	Yes	Yes	No	Medium

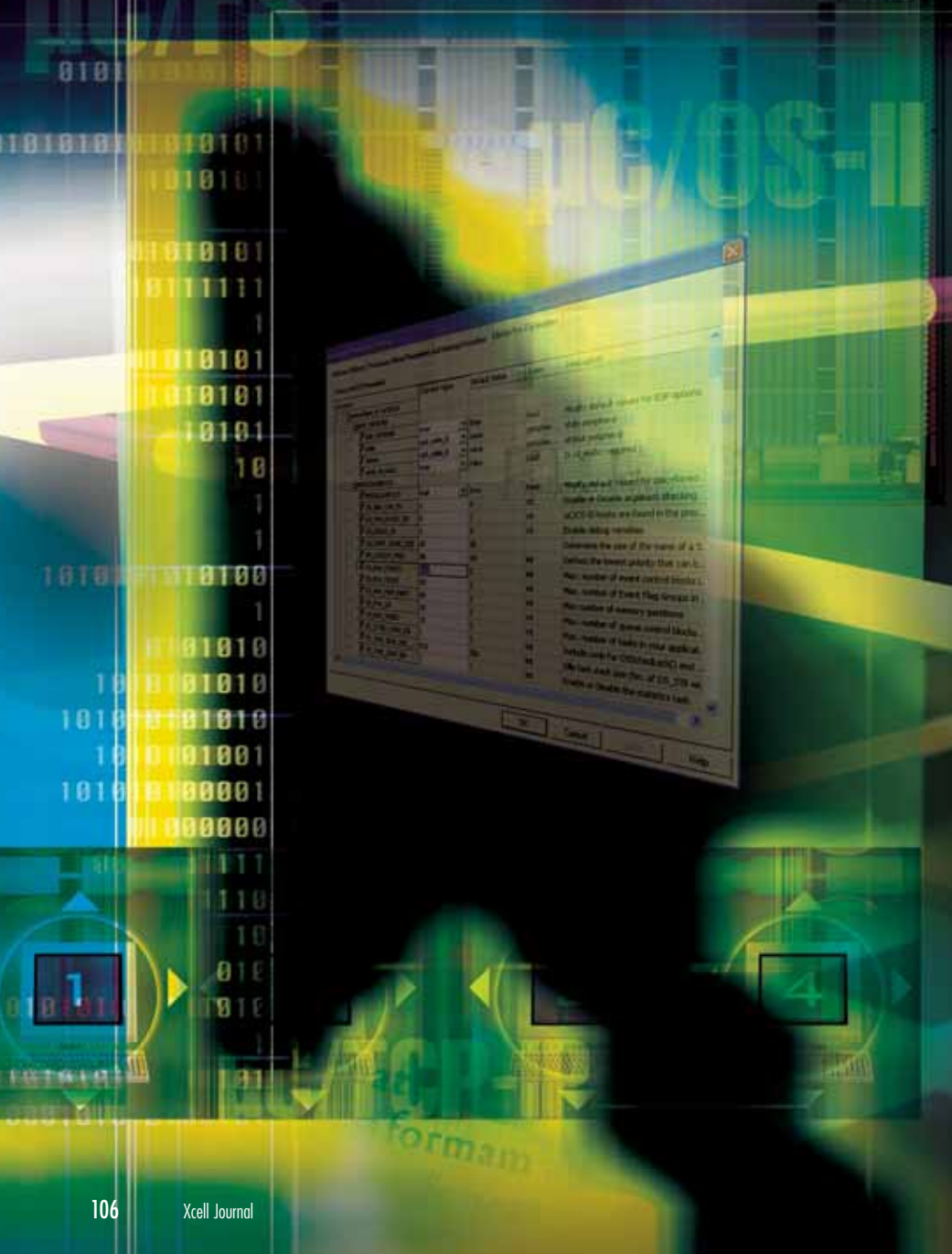
Figure 2 – Options for quick, easy, and inexpensive general-purpose in-circuit SPI Flash programming

JTAG Tool Vendors

JTAG Technologies	*www.JTAG.com*
Acculogic	*www.acculogic.com*
Corelis	*www.corelis.com*
Goepel	*www.goepel.com*
Assett-Intertech	*www.assett-intertech.com*
Intellitech	*www.intellitech.com*
Flynn	*www.flynn.com*
Universal Scan	*www.universalscan.com*

Add Valuable Software Modules with XPS

XPS, with its MLD interface, simplifies the process of configuring and compiling software modules.



by Matt Gordon
Software Engineer
Micrium
matt.gordon@micrium.com

The Xilinx® Embedded Development Kit (EDK) enables you to create powerful embedded processor systems, as well as the complex applications running on them. These applications may need to perform numerous tasks and access a wide range of peripherals, such as memory controllers, display interfaces, and network interface cards. It is difficult, expensive, and inefficient to design such large applications from scratch, especially when existing software modules are capable of meeting some or all of your system's goals.

Integrating these modules with your application, however, can sometimes prove to be more difficult than writing equivalent modules yourself. Luckily, Xilinx Platform Studio (XPS) provides a standard interface, known as the microprocessor library definition (MLD) interface, to facilitate adding such modules to your projects. Operating systems and other software modules adhering to this interface can conveniently be configured for your application from within XPS. Micrium has adapted many of its software modules to the MLD interface, allowing you to quickly and easily add high-quality, well-documented modules to your embedded applications.

The MLD Format

Each Micrium software module that can be configured using XPS has an MLD file defining that module's configurable parameters. Generally, configurable parameters correspond to features and services that will comprise the compiled module. These parameters can also be used to identify the hardware components that the module will utilize. For example, the MLD file for a graphical user interface (GUI) module might define parameters stipulating the colors and resolutions supported by the compiled module, as well as a parameter indicating the memory-mapped location of the display controller used by the module. You would be able to configure these parameters from the Software Platform Settings dialog box in XPS, tailoring modules as needed to create a suitable software platform for nearly any application.

Once you have configured a software platform, you can build it with Library Generator (LibGen), a utility that is automatically invoked when a project is compiled in XPS. LibGen is responsible for building software modules according to the parameter values set forth in XPS, and it uses Tool Command Language (Tcl) files to meet this objective. Tcl files, which are provided with each Micrium software module that complies with the MLD interface, contain functions that are able to access the parameter values you specify. These functions are called by LibGen, and they normally generate source and header files that allow the module to be built as per the requirements of the application.

The specific role that Tcl and MLD files play in the process of configuring and building software modules can be better understood by examining one of Micrium's software modules, µC/OS-II, The Real-Time Kernel. µC/OS-II is a popular real-time operating system (RTOS) that has been proven effective in hundreds of products. The module's clean and consistent source code, which is certifiable for use in systems deemed safety-critical by the FAA and FDA, is described in "MicroC/OS-II, The Real-Time Kernel," by Jean Labrosse. This book describes µC/OS-II's efficient implementation of semaphores, message queues, timers, and other standard operating system services.

As Labrosse's book explains, µC/OS-II's services are normally configured by manipulating constants in a header file, os_cfg.h. The MLD interface eliminates the need to manually edit this file, instead letting you configure the operating system from XPS's Software Platform Settings dialog box, as shown in Figure 1. The contents of this dialog box are determined by µC/OS-II's MLD file, which defines configurable parameters representing the constants normally found in os_cfg.h. Because µC/OS-II's Tcl file has access to the values specified for each of these configurable parameters, it can generate os_cfg.h whenever LibGen is run. This automatic generation of the configuration file gives you a custom version of µC/OS-II each time you build your project, making the operating system instantly amendable to the changing needs of an application.

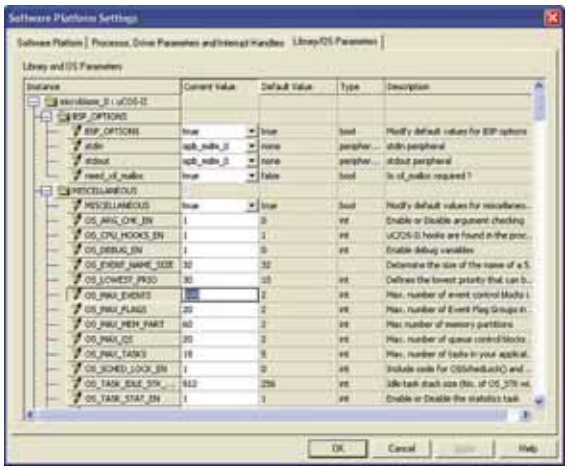


Figure 1 – You can configure µC/OS-II in XPS.

An Example Application

Although the benefits of the MLD interface are apparent even when considering a single module like µC/OS-II, they are more pronounced in a large, multi-faceted application such as a Web server. A Web server has many responsibilities, and from a design perspective, it is convenient to view each of these responsibilities as a separate component, as depicted in Figure 2. Once an application has been logically divided in this manner, you can choose software modules to implement each component.

A Web server might, for example, use modules to handle the various protocols, such as HTTP and TCP, involved in serving Web pages. It might also take advantage of a module capable of inter-

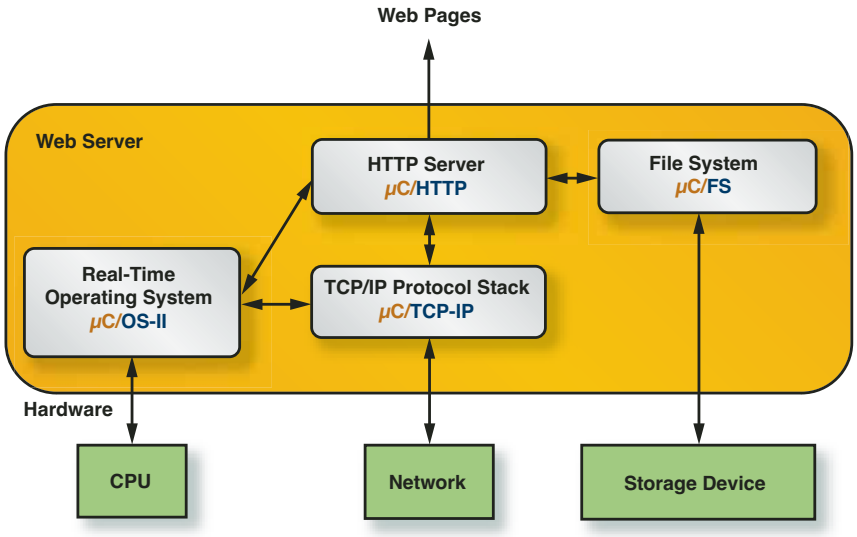


Figure 2 – You can efficiently implement each of the components of a Web server with software modules from Micrium.

By offering a convenient way of configuring useful software modules, such as μC/FS, μC/TCP-IP, and μC/OS-II, the MLD interface accelerates the process of constructing complex applications from these embedded building blocks.

facing with a storage device, allowing Web pages to be saved as files. The Web server could rely on an RTOS to coordinate the tasks performed by each of these modules.

The use of these types of software modules as "building blocks" for constructing large applications, such as Web servers, significantly reduces the time and effort needed to develop such applications, provided that you can easily incorporate the modules in your design. Many software modules, though, are unwieldy, and they introduce unnecessary complexities into projects that use them. Our example Web server could encompass hundreds of source files, requiring the inclusion of a comparable amount of header files. Adding these files to an application in XPS would result in a cumbersome project, teeming with files that shouldn't be of concern to application developers. To avoid such a confusing muddle of files, you could create an object code library, but this would force you to learn the compilation procedures for several software modules, and the resultant library would have to be rebuilt for different configurations and versions of the modules.

The MLD interface obscures these compilation details, resulting in organized and efficient applications running on highly configurable software platforms. A Web server designed on such a platform would be outwardly simple, with the multitude of files composing the server's modules invisible to designers. The Web server would appear to consist only of the files directly implicated in the application's primary task of serving Web pages, so it could be debugged quickly, even by developers unfamiliar with the project and its modules. Updating or revising the Web server would be similarly painless, because well-documented and dependable software modules, such as those provided by Micrium, could be added to the application without needing to understand the mechanics of configuring and compiling the modules.

The Micrium Software Modules

Micrium's software modules are readily adaptable to most applications. The process of adding such high-quality software to a project is expedited in XPS, as Micrium offers an assortment of modules that comply with the MLD interface. You can add any of these modules to a project by using the Software Platform Settings dialog box, as shown in Figure 3. This dialog box gives you the ability to rapidly deploy practical software platforms to support complex applications, including Web servers. In fact, you can construct a Web server entirely of software from Micrium simply by specifying the appropriate modules in XPS.

Figure 3 – The Software Platform Settings dialog box lets you quickly add Micrium's modules to your project.

The centerpiece of such a Web server would be μC/TCP-IP, Micrium's TCP/IP protocol stack. μC/TCP-IP was meticulously developed using stringent coding standards. The module's completely original design, which was not derived from any existing protocol stacks, supports an extensive array of network options that are presented to XPS users, along with a selection of useful add-on modules. You can use these modules in conjunction with μC/TCP-IP to accommodate the needs of a variety of applications. At least one such module, μC/HTTPs (which offers a consistent means of serving Web pages), would be an integral part of a Web server based on Micrium's modules.

Another key component of a Micrium Web server would be μC/FS, Micrium's embedded file system. μC/FS is a compact and versatile module, with an API similar to that available from the standard C library. The files accessed through this API can reside on storage devices implementing any of the numerous formats that μC/FS recognizes, including CompactFlash, Secure Digital (SD), and SmartMedia. The files themselves can also be of varying formats, because μC/FS is compatible with both the Microsoft FAT file system and a proprietary file system. Support for either of these formats is one of the many features reflected in the module's MLD file, allowing μC/FS, like Micrium's other modules, to be configured in XPS and seamlessly included in a Web server or any other application.

Conclusion

By offering a convenient way of configuring useful software modules, such as μC/FS, μC/TCP-IP, and μC/OS-II, the MLD interface accelerates the process of constructing complex applications from these embedded building blocks. The benefits afforded by the interface may be unintentionally relinquished, however, if poorly documented, unreliable modules are selected.

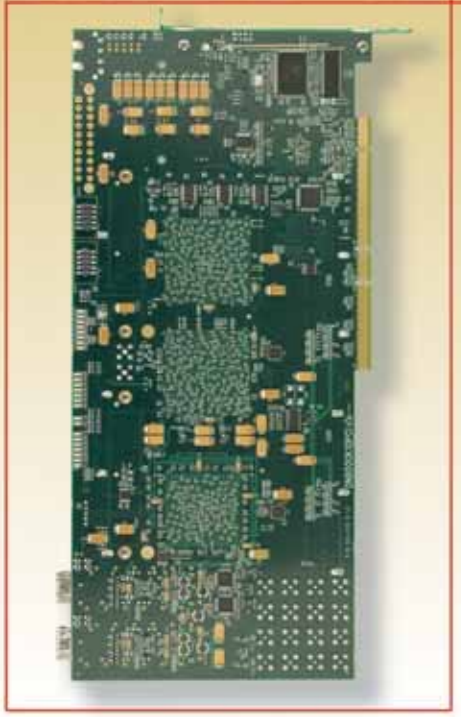
To avoid the headaches that can result from the use of such modules, you should choose dependable and easy-to-use modules. Micrium's software modules complement the valuable tools provided with the EDK, enabling the efficient development of powerful applications.

For more information about Micrium's modules, please visit *www.micrium.com/microblaze/*.

VIRTEX-4

INCREDIBLY

INTERCONNECTED



Partitioning your design onto the 3.7 million ASIC gates (LSI measure) of our new board is a lot easier. With three of the biggest, fastest, new Xilinx Virtex-4 FPGAs this PCI hosted logic prototyping system takes full advantage of the integrated ISERDES/OSERDES. 400MHz LVDS differential communication with 10X multiplexing means more than 1800 signals between FPGA A & B. Synplicity Certify™ models are provided for partitioning assistance.

A dedicated PCI Bridge (533mb/s data transfer) means that all FPGA resources are available for logic emulation. Other features are designed to ease your prototyping job:

- 5 programmable clock synthesizers
- 2 DDR2 SODIMMs (custom DIMMs for SSRAM, QDR, Flash ...)
- Two, 200 pin expansion connectors for daughter cards
- 10 GB/s serial I/O interfaces with SMA connectors and XFP or SFP Modules
- Configured by PCI, USB 2.0, or SmartMedia with partial reconfiguration support on all FPGAs

Various stuffing options and a wide selection of daughter cards let you meet your exact design requirements. Prices start at less than $10,000. Call The Dini Group today for the latest ASIC prototyping solution.

The DiNI Group

1010 Pearl Street, Suite 6 • La Jolla, CA 92037 • (858) 454-3419
Email: sales@dinigroup.com
www.dinigroup.com

XILINX®
The Programmable Logic Company℠

Xilinx Virtex-4™ FPGAs

www.xilinx.com/devices/

Product Selection Matrix

		Virtex-4 LX (Logic)								Virtex-4 SX (Signal Processing)			Virtex-4 FX (Embedded Processing & Serial Connectivity)					
		XC4VLX15	XC4VLX25	XC4VLX40	XC4VLX60	XC4VLX80	XC4VLX100	XC4VLX160	XC4VLX200	XC4VSX25	XC4VSX35	XC4VSX55	XC4VFX12	XC4VFX20	XC4VFX40	XC4VFX60	XC4VFX100	XC4VFX140
CLB Resources	CLB Array (Row x Column)	64 x 24	96 x 28	128 x 36	128 x 52	160 x 56	192 x 64	192 x 88	192 x 116	64 x 40	96 x 40	128 x 48	64 x 24	64 x 36	96 x 52	128 x 52	160 x 68	192 x 84
	Slices	6,144	10,752	18,432	26,624	35,840	49,152	67,584	89,088	10,240	15,360	24,576	5,472	8,544	18,624	25,280	42,176	63,168
	Logic Cells	13,824	24,192	41,472	59,904	80,640	110,592	152,064	200,448	23,040	34,560	55,296	12,312	19,224	41,904	56,880	94,896	142,128
	CLB Flip Flops	12,288	21,504	36,864	53,248	71,680	98,304	135,168	178,176	20,480	30,720	49,152	10,944	17,088	37,248	50,560	84,352	126,336
Memory Resources	Max. Distributed RAM Bits	98,304	172,032	294,912	425,984	573,440	786,432	1,081,344	1,425,408	163,840	245,760	393,216	87,552	136,704	297,984	404,480	674,816	1,010,688
	Block RAM/FIFO w/ECC (18 kbits each)	48	72	96	160	200	240	288	336	128	192	320	36	68	144	232	376	552
	Total Block RAM (kbits)	864	1,296	1,728	2,880	3,600	4,320	5,184	6,048	2,304	3,456	5,760	648	1,224	2,592	4,176	6,768	9,936
Clock Resources	Digital Clock Managers (DCM)	4	8	8	8	12	12	12	12	4	8	8	4	4	8	12	12	20
	Phase-matched Clock Dividers (PMCD)	0	4	4	4	8	8	8	8	0	4	4	0	0	4	8	8	8
I/O Resources	Max Select I/O™	320	448	640	640	768	960	960	960	320	448	640	320	320	448	576	768	896
	Total I/O Banks	9	11	13	13	15	17	17	17	9	11	13	9	9	11	13	15	17
	Digitally Controlled Impedence	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	Max Differential I/O Pairs	160	224	320	320	384	480	480	480	160	224	320	160	160	224	288	384	448
	I/O Standards	LDT-25, LVDS-25, LVDSEXT-25, BLVDS-25, ULVDS-25, LVPECL-25, LVCMOS25, LVCMOS18, LVCMOS15, PCI33, LVTTL, LVCMOS33, PCI-X, PCI66, GTL, GTL+, HSTL I (1.5V,1.8V), HSTL II (1.5V,1.8V), HSTL III (1.5V,1.8V), HSTL IV (1.5V,1.8V), SSTL2I, SSTL2II, SSTL18 I, SSTL18 II																
DSP Resources	XtremeDSP™ Slices	32	48	64	64	80	96	96	96	128	192	512	32	32	48	128	160	192
Embedded Hard IP Resources	PowerPC™ Processor Blocks	—	—	—	—	—	—	—	—	—	—	—	1	1	2	2	2	2
	10/100/1000 Ethernet MAC Blocks	—	—	—	—	—	—	—	—	—	—	—	2	2	4	4	4	4
	RocketIO™ Serial Transceivers	—	—	—	—	—	—	—	—	—	—	—	0	8	12	16	20	24
Speed Grades	Commercial (slowest to fastest)	-10, -11, -12	-10, -11, -12	-10, -11, -12	-10, -11, -12	-10, -11, -12	-10, -11, -12	-10, -11, -12	-10, -11	-10, -11, -12	-10, -11, -12	-10, -11, -12	-10, -11, -12	-10, -11, -12	-10, -11, -12	-10, -11, -12	-10, -11, -12	-10, -11
	Industrial (slowest to fastest)	-10, -11	-10, -11	-10, -11	-10, -11	-10, -11	-10, -11	-10, -11	-10	-10, -11	-10, -11	-10, -11	-10, -11	-10, -11	-10, -11	-10, -11	-10, -11	-10
	Configuration Memory Bits	4,765,568	7,819,904	12,259,712	17,717,632	23,291,008	30,711,680	40,347,008	51,367,808	9,147,648	13,700,288	22,745,216	4,765,568	7,242,624	13,550,720	21,002,880	33,065,408	47,856,896
	EasyPath™ Cost Reduction Solutions[1]	—	XCE4VLX25	XCE4VLX40	XCE4VLX60	XCE4VLX80	XCE4VLX100	XCE4VLX160	XCE4VLX200	XCE4VSX25	XCE4VSX35	XCE4VSX55	—	XCE4VFX20	XCE4VFX40	XCE4VFX60	XCE4VFX100	XCE4VFX140

Package[1]	Area	MGT[2]	Pins	4VLX15	4VLX25	4VLX40	4VLX60	4VLX80	4VLX100	4VLX160	4VLX200	4VSX25	4VSX35	4VSX55	4VFX12	4VFX20	4VFX40	4VFX60	4VFX100	4VFX140
SF363	17 x 17 mm	—	240	240	240										240					
FF668	27 x 27 mm	—	448	320	448	448	448					320	448		320					
FF1148	35 x 35 mm	—	768			640	640	768	768	768				640						
FF1513	40 x 40 mm	—	960						960	960	960									
FF672	27 x 27 mm	12	352													320 (8)[3]	352 (12)[3]	352 (12)[3]		
FF1152	35 x 35 mm	20	576														448 (12)[3]	576 (16)[3]	576 (20)[3]	
FF1517	40 x 40 mm	24	768																768 (20)[3]	768 (24)[3]
FF1760	42.5 x 42.5 mm	24	896																	896 (24)[3]

Notes: 1. SFA Packages (SF): flip-chip fine-pitch BGA (0.80 mm ball spacing).
FFA Packages (FF): flip-chip fine-pitch BGA (1.00 mm ball spacing).
All Virtex-4 LX and Virtex-4 SX devices available in the same package are footprint-compatible.
2. MGT: RocketIO Multi-Gigabit Transceivers.
3. Number of available RocketIO Muti-Gigabit Transceivers.
4. EasyPath solutions provide conversion-free path for volume production.

Pb-free solutions are available. For more information about Pb-free solutions, visit www.xilinx.com/pbfree.

Important: Verify all data in this document with the device data sheets found at http://www.xilinx.com/partinfo/databook.htm

Product Selection Matrix

	CLB Resources					Memory Resources			DSP	CLK Resources		I/O Features				Speed		PROM	
	System Gates (see note 1)	CLB Array (Row x Col)	Number of Slices	Equivalent Logic Cells	CLB Flip-Flops	Max. Distributed RAM Bits	# Block RAM	Block RAM (bits)	Dedicated Multipliers	DCM Frequency (min/max)	# DCMs	Digitally Controlled Impedance	Number of Differential I/O Pairs	Maximum I/O	I/O Standards	Commercial Speed Grades (slowest to fastest)	Industrial Speed Grades (slowest to fastest)	Configuration Memory (Bits)	EasyPath
Spartan-3E Family – 1.2 Volt																			
XC3S100E	100K	16 x 22	960	2,160	1920	15K	4	72K	4	5/326	2	NO	40	108	Single-ended LVTTL, LVCMOS3.3/2.5/1.8/1.5/1.2, PCI 3.3V – 32/64-bit 33/66MHz, PCI-X 100MHz, SSTL I 1.8/2.5, HSTL I 1.8, HSTL III 1.8	-4 -5	-4	0.6M	
XC3S250E	250K	26 x 34	2448	5,508	4896	38K	12	216K	12	5/326	4	NO	68	172		-4 -5	-4	1.4M	
XC3S500E	500K	34 x 46	4656	10,476	9312	73K	20	360K	20	5/326	4	NO	92	232		-4 -5	-4	2.3M	
XC3S1200E	1200K	46 x 60	8672	19,512	17344	136K	28	504K	28	5/326	8	NO	124	304		-4 -5	-4	3.8M	
XC3S1600E	1600K	58 x 76	14752	33,192	29504	231K	36	648K	36	5/326	8	NO	156	376	Differential LVDS2.5, Bus LVDS2.5, mini-LVDS, RSDS, LVPECL	-4 -5	-4	5.9M	
Spartan-3 and Spartan-3L Families – 1.2 Volt (see note 2)																			
XC3S50		50K	16 x 12	768	1,728	1,536	12K	4	72K	4	24/280	2	YES	56	124	Single-ended LVTTL, LVCMOS3.3/2.5/1.8/1.5/1.2, PCI 3.3V – 32/64-bit 33MHz, SSTL2 Class I & II, SSTL18 Class I, HSTL Class I, III, HSTL1.8 Class I, II & III, GTL, GTL+	-4 -5	-4	.4M	
XC3S200		200K	24 x 20	1,920	4,320	3,840	30K	12	216K	12	24/280	4	YES	76	173		-4 -5	-4	1.0M	
XC3S400		400K	32 x 28	3,584	8,064	7,168	56K	16	288K	16	24/280	4	YES	116	264		-4 -5	-4	1.7M	
XC3S1000	XC3S1000L	1000K	48 x 40	7,680	17,280	15,360	120K	24	432K	24	24/280	4	YES	175	391		-4 -5	-4	3.2M	
XC3S1500	XC3S1500L	1500K	64 x 52	13,312	29,952	26,624	208K	32	576K	32	24/280	4	YES	221	487		-4 -5	-4	5.2M	✔
XC3S2000		2000K	80 x 64	20,480	46,080	40,960	320K	40	720K	40	24/280	4	YES	270	565	Differential LVDS2.5, Bus LVDS2.5, Ultra LVDS2.5, LVDS_ext2.5, RSDS, LDT2.5, LVPECL	-4 -5	-4	7.7M	✔
XC3S4000	XC3S4000L	4000K	96 x 72	27,648	62,208	55,296	432K	96	1,728K	96	24/280	4	YES	312	712		-4 -5	-4	11.3M	✔
XC3S5000		5000K	104 x 80	33,280	74,880	66,560	520K	104	1,872K	104	24/280	4	YES	344	784		-4 -5	-4	13.3M	✔

Note: 1. System Gates include 20-30% of CLBs used as RAMs.
2. Spartan-3L devices offer reduced quiescent power consumption. Package offerings may vary slightly from those offered in the Spartan-3 family. See Package Selection Matrix for details.

Package Options and UserI/O[1]

		Spartan-3E (1.2V)						Spartan-3 (1.2V)							
			XC3S100E	XC3S250E	XC3S500E	XC3S1200E	XC3S1600E		XC3S50	XC3S200	XC3S400	XC3S1000	XC3S1500	XC3S2000	XC3S4000	XC3S5000
Pins	Area[2]	I/O's	108	172	232	304	376	I/O's	124	173	264	391	487	565	712	784
PQFP Packages (PQ) – wire-bond plastic QFP (0.5 mm lead spacing)																
208	30.6 x 30.6 mm			158	158				124	141	141					
240	34.6 x 34.6 mm															
VQFP Packages (VQ) – very thin QFP (0.5 mm lead spacing)																
100	16.0 x 16.0 mm		66	66					63	63						
TQFP Packages (TQ) – thin QFP (0.5 mm lead spacing)																
144	22.0 x 22.0 mm		108	108					97	97	97					
Chip Scale Packages (CP) – wire-bond chip-scale BGA (0.5 mm ball spacing)																
132	8 x 8 mm			92	92				89							
Chip Scale Packages (CS) – wire-bond chip-scale BGA (0.8 mm ball spacing)																
144	12 x 12 mm															
FGA Packages (FT) – wire-bond fine-pitch thin BGA (1.0 mm ball spacing)																
256	17 x 17 mm			172	190	190				173	173	173				
FGA Packages (FG) – wire-bond fine-pitch BGA (1.0 mm ball spacing)																
256	17 x 17 mm															
320	19 x 19 mm				232	250	250				221	221	221			
400	21 x 21 mm					304	304									
456	23 x 23 mm										264	333	333	333		
484	23 x 23 mm						376									
676	27 x 27 mm											391	487	489	489	
900	31 x 31 mm													565	633	633
1156	35 x 35 mm														712	784
BGA Packages (BG) – wire-bond standard BGA (1.27 mm ball spacing)																
256	27 x 27 mm															

Notes: 1. Numbers in table indicate maximum number of user I/Os.
2. Area dimensions for lead-frame products are inclusive of the leads.

Pb-free solutions are available. For more information about Pb-free solutions visit *www.xilinx.com/pbfree*.

Platform Flash Features

	XCF01S	XCF02S	XCF04S	XCF08P	XCF16P	XCF32P
Density	1 Mb	2 Mb	4 Mb	8 Mb	16 Mb	32 Mb
JTAG Prog	✔	✔	✔	✔	✔	✔
Serial Config	✔	✔	✔	✔	✔	✔
SelectMap Config				✔	✔	✔
Compression				✔	✔	✔
Design Revisions				✔	✔	✔
VCC (V)	3.3	3.3	3.3	1.8	1.8	1.8
VCCO (V)	1.8 – 3.3	1.8 – 3.3	1.8 – 3.3	1.5 – 3.3	1.5 – 3.3	1.5 – 3.3
VCCJ (V)	2.5 – 3.3	2.5 – 3.3	2.5 – 3.3	2.5 – 3.3	2.5 – 3.3	2.5 – 3.3
Clock(MHz)	33	33	33	40	40	40
Packages	VO20	VO20	VO20	FS48 VO48	FS48 VO48	FS48 VO48
Pb-Free Pkg	VOG20	VOG20	VOG20	FSG48 VOG48	FSG48 VOG48	FSG48 VOG48
Availability	Now	Now	Now	Now	Now	Now

Important: Verify all data in this document with the device data sheets found at http://www.xilinx.com/partinfo/databook.htm

For the latest information and product specifications on all Xilinx products, please visit the following links:

FPGA and CPLD Devices
www.xilinx.com/devices/

Packaging
www.xilinx.com/packaging/

Development Reference Boards
www.xilinx.com/board_search/

Platform Flash
www.xilinx.com/products/silicon_solutions/proms/pfp/

Configuration and Storage Systems
www.xilinx.com/configsolns/

Software
www.xilinx.com/ise/

IP Reference
www.xilinx.com/ipcenter/

Global Services
www.xilinx.com/support/gsd/

XILINX®
The Programmable Logic Company℠

Xilinx Spartan™-3 FPGAs and Platform Flash
www.xilinx.com/devices/

Product Selection Matrix – CoolRunner™ Series

		System Gates	Macrocells	Product Terms per Macrocell	Input Voltage Compatible	Output Voltage Compatible	I/O Features: Maximum I/O	I/O Features: I/O Banking	Speed: Min. Pin-to-pin Logic Delay (ns)	Speed: Commercial Speed Grades (fastest to slowest)	Speed: Industrial Speed Grades (fastest to slowest)	Speed: IQ Speed Grade	Clocking: Global Clocks	Clocking: Product Term Clocks per Function Block
	CoolRunner-II Family – 1.8 Volt													
CoolRunner-II	XC2C32A	750	32	40	1.5/1.8/2.5/3.3	1.5/1.8/2.5/3.3	33	2	3.8	-4 -6	-6	-6	3	17
	XC2C64A	1,500	64	40	1.5/1.8/2.5/3.3	1.5/1.8/2.5/3.3	64	2	4.6	-5 -7	-7	-7	3	17
	XC2C128	3,000	128	40	1.5/1.8/2.5/3.3	1.5/1.8/2.5/3.3	100	2	5.7	-6 -7	-7	-7	3	17
	XC2C256	6,000	256	40	1.5/1.8/2.5/3.3	1.5/1.8/2.5/3.3	184	2	5.7	-6 -7	-7	-7	3	17
	XC2C384	9,000	384	40	1.5/1.8/2.5/3.3	1.5/1.8/2.5/3.3	240	4	7.1	-7 -10	-10	-10	3	17
	XC2C512	12,000	512	40	1.5/1.8/2.5/3.3	1.5/1.8/2.5/3.3	270	4	7.1	-7 -10	-10	-10	3	17
	CoolRunner XPLA3 Family – 3.3 Volt													
CoolRunner XPLA3	XCR3032XL	750	32	48	3.3/5	3.3	36		5	-5 -7 -10	-7 -10	-10	4	16
	XCR3064XL	1,500	64	48	3.3/5	3.3	68		6	-6 -7 -10	-7 -10	-10	4	16
	XCR3128XL	3,000	128	48	3.3/5	3.3	108		6	-6 -7 -10	-7 -10	-10	4	16
	XCR3256XL	6,000	256	48	3.3/5	3.3	164		7.5	-7 -10 -12	-10 -12	-12	4	16
	XCR3384XL	9,000	384	48	3.3/5	3.3	220		7.5	-7 -10 -12	-10 -12	-12	4	16
	XCR3512XL	12,000	512	48	3.3/5	3.3	260		7.5	-7 -10 -12	-10 -12	-12	4	16

Package Options and User I/O

Pins	Area[1]	CoolRunner-II: XC2C32A	XC2C64A	XC2C128	XC2C256	XC2C384	XC2C512	CoolRunner XPLA3: XCR3032XL	XCR3064XL	XCR3128XL	XCR3256XL	XCR3384XL	XCR3512XL
QFN Packages (QFG) – quad flat no-lead (0.5 mm lead spacing)													
32	5 x 5 mm	21											
48	7 x 7 mm		37										
PLCC Packages (PC) – wire-bond plastic chip carrier (1.27 mm lead spacing)													
44	17.5 x 17.5 mm	33	33					36	36				
PQFP Packages (PQ) – wire-bond plastic QFP (0.5 mm lead spacing)													
208	30.6 x 30.6 mm				173	173	173				164	172	180
VQFP Packages (VQ) – very thin QFP (0.5 mm lead spacing)													
44	12.0 x 12.0 mm	33	33					36	36				
100	16.0 x 16.0 mm		64	80	80				68	84			
TQFP Packages (TQ) – thin QFP (0.5 mm lead spacing)													
144	22.0 x 22.0 mm			100	118	118				108	120	118*	
Chip Scale Packages (CP) – wire-bond chip-scale BGA (0.5 mm ball spacing)													
56	6 x 6 mm	33	45						48				
132	8 x 8 mm			100	106								
Chip Scale Packages (CS) – wire-bond chip-scale BGA (0.8 mm ball spacing)													
48	7 x 7 mm							36	40				
144	12 x 12 mm									108			
280	16 x 16 mm										164		
FGA Packages (FT) – wire-bond fine-pitch thin BGA (1.0 mm ball spacing)													
256	17 x 17 mm				184	212	212				164	212	212
FBGA Packages (FG) – wire-bond fine-line BGA (1.0 mm ball spacing)													
324	23 x 23 mm					240	270					220	260

* JTAG pins and port enable are not pin compatible in this package for this member of the family.

Note 1: Area dimensions for lead-frame products are inclusive of the leads.

Product Selection Matrix – 9500 Series

	System Gates	Macrocells	Product Terms per Macrocell	Input Voltage Compatible	Output Voltage Compatible	I/O Features: Maximum I/O	I/O Features: I/O Banking	Speed: Min. Pin-to-pin Logic Delay (ns)	Speed: Commercial Speed Grades (fastest to slowest)	Speed: Industrial Speed Grades (fastest to slowest)	Speed: IQ Speed Grade	Clocking: Global Clocks	Clocking: Product Term Clocks per Function Block
XC9500XV Family – 2.5 Volt													
XC9536XV	800	36	90	2.5/3.3	1.8/2.5/3.3	36	1	5	-5 -7	-7	NA	3	18
XC9572XV	1,600	72	90	2.5/3.3	1.8/2.5/3.3	72	1	5	-5 -7	-7	NA	3	18
XC95144XV	3,200	144	90	2.5/3.3	1.8/2.5/3.3	117	2	5	-5 -7	-7	NA	3	18
XC95288XV	6,400	288	90	2.5/3.3	1.8/2.5/3.3	192	4	6	-6 -7 -10	-7 -10	NA	3	18
XC9500XL Family – 3.3 Volt													
XC9536XL	800	36	90	2.5/3.3/5	2.5/3.3	36		5	-5 -7 -10	-7 -10	-10	3	18
XC9572XL	1,600	72	90	2.5/3.3/5	2.5/3.3	72		5	-5 -7 -10	-7 -10	-10	3	18
XC95144XL	3,200	144	90	2.5/3.3/5	2.5/3.3	117		5	-5 -7 -10	-7 -10	NA	3	18
XC95288XL	6,400	288	90	2.5/3.3/5	2.5/3.3	192		6	-6 -7 -10	-7 -10	NA	3	18
XC9500 Family – 5 Volt													
XC9536	800	36	90	5	5	36		10	-5 -6 -10 -15	-7 -10 -15	-15	3	18
XC9572	1,600	72	90	5	5	72		10	-7 -10 -15	-10 -15	-15	3	18
XC95108	2,400	108	90	5	5	108		10	-7 -10 -15 -20	-7 -10 -15 -20	NA	3	18
XC95144	3,200	144	90	5	5	133		10	-7 -10 -15	-10 -15	NA	3	18
XC95216	4,800	216	90	5	5	166		10	-10 -15 -20	-10 -15 -20	NA	3	18
XC95288	6,400	288	90	5	5	192		10	-10 -15 -20	-15 -20	NA	3	18

Pb-free solutions are available. For more information about Pb-free solutions visit *www.xilinx.com/pbfree*

Package Options and User I/O

		XC9500XV				XC9500XL				XC9500					
Pins	Area[1]	XC9536XV	XC9572XV	XC95144XV	XC95288XV	XC9536XL	XC9572XL	XC95144XL	XC95288XL	XC9536	XC9572	XC95108	XC95144	XC95216	XC95288
PLCC Packages (PC) – wire-bond plastic chip carrier (1.27 mm lead spacing)															
44	17.5 x 17.5 mm	34	34			34	34			34	34				
84	30.2 x 30.2 mm										69	69			
PQFP Packages (PQ) – wire-bond plastic QFP (0.5 mm lead spacing)															
100	23.3 x 17.2 mm										72	81	81		
160	31.2 x 31.2 mm											108	133	133	
208	30.6 x 30.6 mm				168				168					166	168
VQFP Packages (VQ) – very thin TQFP (0.5 mm lead spacing)															
44	12.0 x 12.0 mm	34	34			34	34			34					
64	12.0 x 12.0 mm					36	52								
TQFP Packages (TQ) – thin QFP (0.5 mm lead spacing)															
100	16.0 x 16.0 mm		72	81			72	81			72	81	81		
144	22.0 x 22.0 mm			117	117			117	117						
Chip Scale Packages (CS) – wire-bond chip-scale BGA (0.8 mm ball spacing)															
48	7 x 7 mm	36	38			36	38			34					
144	12 x 12 mm			117				117							
280	16 x 16 mm				192				192						
BGA Packages (BG) – wire-bond standard BGA (1.27 mm ball spacing)															
256	27 x 27 mm								192						
352	35.0 x 35.0 mm													166	192
FBGA Packages (FG) – wire-bond Fine-line BGA (1.0 mm ball spacing)															
256	17 x 17 mm				192				192						

Note 1: Area dimensions for lead-frame products are inclusive of the leads.

XILINX®
The Programmable Logic Company℠

Xilinx Software – ISE 8.1i

www.xilinx.com/ise

ISE™ 8.1i – Powered by ISE Fmax Technology

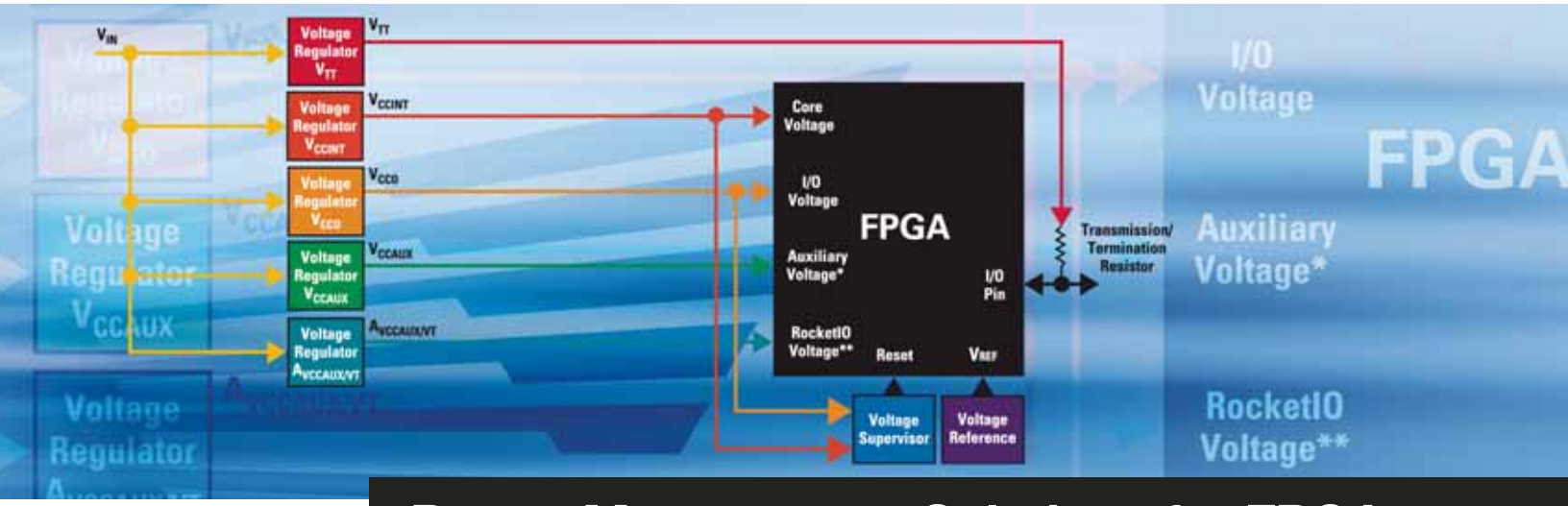
	Feature	*ISE WebPACK™*	*ISE Foundation™*
Platforms		Microsoft Windows 2000 / XP Red Hat Enterprise Linux 3 (32 bit)	Microsoft Windows 2000 / XP Sun Solaris 2.8 or 2.9 Red Hat Enterprise Linux 3 (32 & 64 bit)
Devices	Virtex™ Series	**Virtex:** XCV50 - XCV600 **Virtex-E:** XCV50E- XCV600E **Virtex-II:** XC2V40 - XC2V500 **Virtex-II Pro:** XC2VP2 - XC2VP7 **Virtex-4:** **LX:** XC4VLX15, XC4VLX25 **SX:** XC4VSX25 **FX:** XC4VFX12 **Virtex Q:** XQV100- XQV600 **Virtex QR:** XQVR300, XQVR600 **Virtex-E Q:** XQV600E	ALL
	Spartan™ Series	**Spartan-II/IIE:** All **Spartan-3:** XC3S50 - XC3S1500 **Spartan-3E:** All **Spartan-3L:** XC3S1000L, XC3S1500L **XA (Xilinx Automotive) Spartan-3:** All	**Spartan-II/IIE:** All **Spartan-3:** All **Spartan-3E:** All **Spartan-3L:** All **XA (Xilinx Automotive) Spartan-3:** All
	CoolRunner™ XPLA3 CoolRunner-II™ CoolRunner-IIA	All	
	XC9500™ Series	All	
Design Entry	Schematic Editor	Yes	
	HDL Editor	Yes	
	State Diagram Editor	Microsoft Windows only	
	Xilinx CORE Generator System™	Yes	
	RTL & Technology Viewers	Yes	
	PACE (Pinout & Area Constraint Editor)	Yes	
	Architecture Wizards	Yes	
	3rd Party RTL Checker Support	Yes	
	Xilinx System Generator for DSP	Sold as an Option	
Embedded System Design	Embedded Design Kit (EDK)	Sold as an Option	
Synthesis	XST - Xilinx Synthesis Technology	Yes	
	Mentor Graphics LeonardoSpectrum	Integrated Interface (EDIF Interface on Linux)	
	Mentor Graphics Precision RTL	Integrated Interface	
	Mentor Graphics Precision Physical	EDIF Interface	
	Synopsys DC-FPGA Compiler	EDIF Interface	
	Synplify/Pro/Premier	Integrated Interface	
	Synplicity Amplify Physical Synthesis	EDIF Interface	
	ABEL	CPLD (Microsoft Windows only)	

	Feature	*ISE WebPACK™*	*ISE Foundation™*
Implementation	FloorPlanner		Yes
	PlanAhead™		Sold as an Option
	Timing Driven Place & Route		Yes
	Incremental Design		Yes
	Timing Improvement Wizard		Yes
	Xplorer		Yes
Programming	IMPACT /System ACE™/ CableServer		Yes
Board Level Integration	IBIS / STAMP / HSPICE* Models		Yes
	ELDO Models* (MGT only)		Yes
Verification	ChipScope PRO™		Sold as an Option
	Graphical Testbench Editor		Microsoft Windows only
	ISE Simulator Lite		Yes
	ISE Simulator		ISE Simulator is available as an Optional Design Tool for ISE Foundation only
	ModelSim® XE III Starter		Yes
	ModelSim XE III		Sold as an option
	Static Timing Analyzer		Yes
	FPGA Editor with Probe		Yes
	ChipViewer		Yes
	XPower (Power Analysis)		Yes
	3rd Party Equivalency Checking Support		Yes
	SMARTModels for PowerPC™ and RocketIO™		Yes
	3rd Party Simulator Support		Yes

*HSPICE and ELDO Models are available at the Xilinx Design Tools Center at ***www.xilinx.com/ise***

For more information, visit ***www.xilinx.com/ise***

Support Across The Board.™



Power Management Solutions for FPGAs

National Devices supported:

- Voltage Regulators
- Voltage Supervisors
- Voltage References

Xilinx Devices supported:

- Virtex™
- Virtex-E
- Virtex-II
- Virtex-II Pro
- Virtex-4FX, 4LX, 4SX
- Spartan™-II
- Spartan™-IIE
- Spartan-3, 3E, 3L

National Semiconductor
The Sight & Sound of Information

XILINX®

Avnet Electronics Marketing has collaborated with National Semiconductor® and Xilinx® to create a design guide that matches National Semiconductor's broad portfolio of power solutions to the latest releases of FPGAs from Xilinx.

Featuring parametric tables, sample designs and step-by-step directions, this guide is your fast, accurate source for choosing the best National Semiconductor Power Supply Solution for your design. It also provides an overview of the available design tools, including application notes, development software and evaluation kits.

Go to em.avnet.com/powermgtguide to request your copy today.

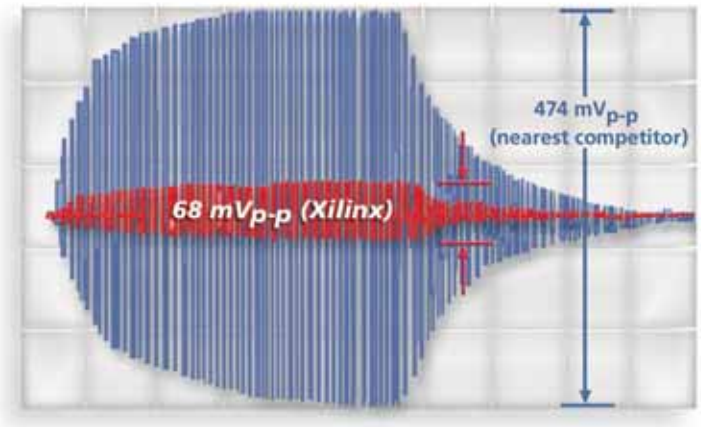
AVNET electronics marketing

Enabling success from the center of technology™

1 800 332 8638
www.em.avnet.com

Supplier Authorized Distributor

"High Signal Integrity Demands a Low-Noise Chip".



Design Example: 1.5 volt LVCMOS 4mA, I/O, 100 aggressors shown.

Best Signal Integrity: 7x Less SSO Noise

Virtex-4 FPGAs deliver the industry's best signal integrity, allowing you to pre-empt board issues at the chip level, for high-speed designs such as memory interfaces. Featuring a unique SparseChevron™ pin out pattern, the Virtex-4 family provides the highest ratio of VCCO/GND pin pairs to user I/O pins available in any FPGA. By strategically positioning one hard power pin and one hard ground pin adjacent to every user I/O on the device, we've reduced signal path inductance and SSO noise to levels far below what you can attain with a virtual ground or soft ground architecture.

The Industry's Highest Signal Integrity, Proven By Industry Experts

Incorporating continuous power and ground planes, plus integrated bypass capacitors, we're eliminating power-supply noise at its source. In addition, we provide on-chip termination resistors to control signal ringing. The lab tests speak for themselves. As measured by signal integrity expert Dr. Howard Johnson, no competing FPGA comes close to achieving the low-noise benchmarks of Virtex-4 devices.

Visit ***www.xilinx.com/virtex4/sipi*** today, and choose the right high-performance FPGA before things get noisy.

Dr. Howard Johnson, author of *High-Speed Digital Design,* frequently conducts technical workshops for digital engineers at Oxford University and other sites worldwide. Visit www.sigcon.com to register.

XILINX®
The Programmable Logic Company℠

www.xilinx.com/virtex4/sipi

Breakthrough Performance At The Lowest Cost

PN 0010885